Lecture 4

Programming with threads

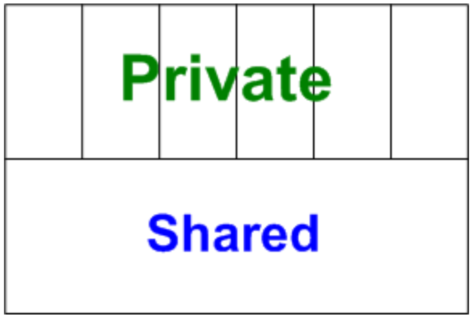
Announcements

SPMD execution model

- Most parallel programming is implemented under the **S**ame **P**rogram **M**ultiple **D**ata 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, threads 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

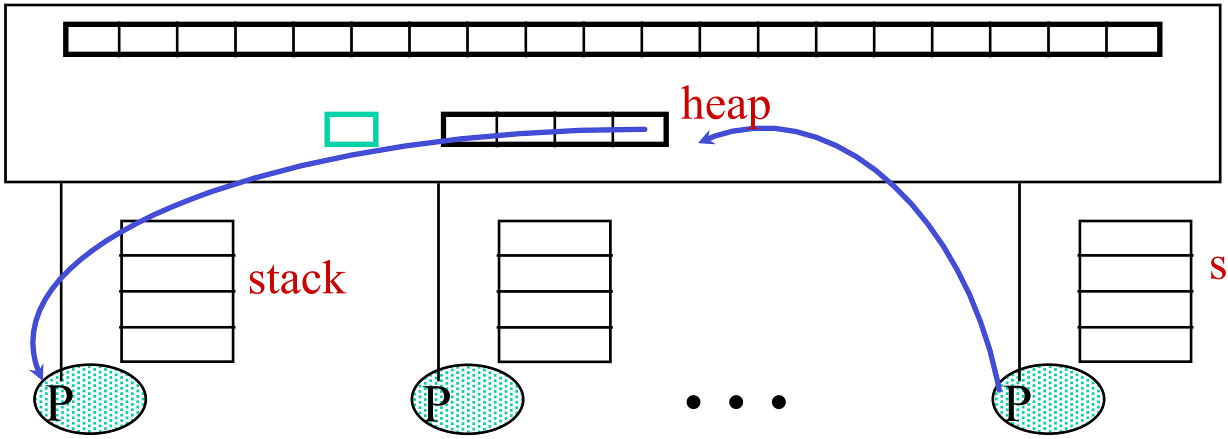


Private

Shared

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/pthreads

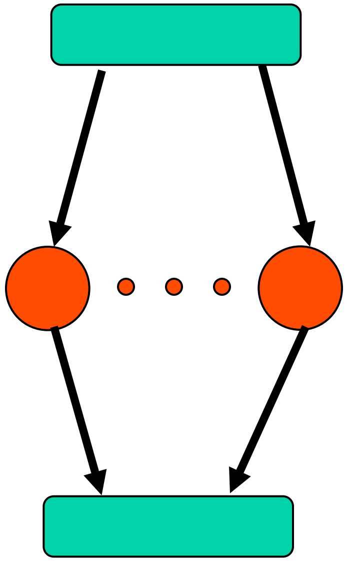
	CPU/ node	Fork (μs)	Create (μs)	MPI Shared Mem (GB/ sec)	Mem - CPU (GB/sec)
AMD 2.4 GHz Opteron	8	41.1	0.66	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	8	64.2	1.75	4.1	16
INTEL 2.4 GHz Xeon	2	55.0	1.64	0.3	4.3
Intel 1.4 GHz Itanium 2	4	54.5	2.03	1.8	6.4

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



Coding with pthreads

```
#include <pthread.h>
#include <assert.h>
#include <stdint.h>
void *Hello(void *tid) {
    sleep(3);
    int64_t  _tid = reinterpret_cast<int64_t>(arg);
    int TID = _tid;
    printf("Hello from thread %d\n", (int) tid);
    pthread_exit(NULL);  return 0;
}
  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++ th8.c -lpthread
% a.out
Hello from thread 0
Hello from thread 1
Hello from thread 2
% a.out
Hello from thread 1
Hello from thread 0
Hello from thread 2
```

Computing a sum in parallel

- Also see: dotprod_mutex.c in the LLNL tutorial

Globals:

```
int64_t *x, global_sum, N, NT;
   Main:
for (int64_t i=0; i < N; i++) x[i] = i+1;
global_sum = 0;
pthread_t thrd[NT];
for(int t=0;t<NT;t++)
  pthread_create(&thrd[t], NULL, summ,reinterpret_cast<void *>(t));
   //Join threads…
cout << "The sum of 1 to " << N << " is: " << sum << endl;
```

The computation

```
void *summm(void *arg){
     comput TID
    int64_t i0 = TID*(N/NT),  i1   = i0 + (N/NT);
    for (int64_t  i=i0;  i<i1;  i++)
        global_sum +=  x[i] ;
    pthread_exit(NULL); return 0;
}
```

```
g++ summ.C -lpthread
% summ 2 1000
# threads: 2, N: 1000
The sum of 1 to 1000 is: 500500
```

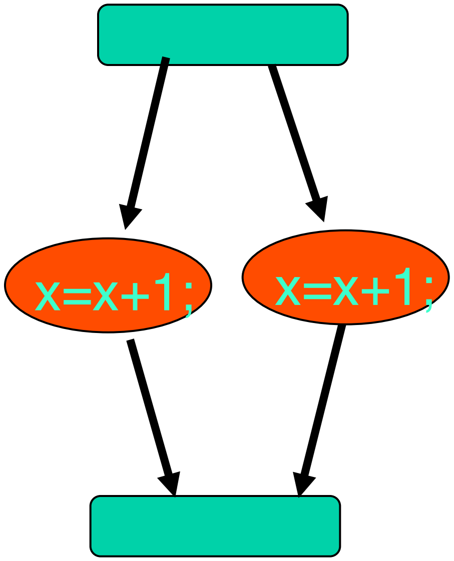
```
g++ summ.C -lpthread
% summ 2 10000
# threads: 2, N: 10000
The sum of 1 to 10000 is: 41957380
Result verified to be INCORRECT, should be 50005000
```

Race conditions

- Consider the statement, assuming x == 0

  x=x+1;

- Generated code
  - r1 ← `(x)`
  - r1 ← `r1 + #1`
  - r1 → (x)



- Possible interleaving with two threads

P1	P2	
`r1 ← x`		*r1(P1) gets 0*
	`r1 ← x`	*r2(P2) also gets 0*
`r1 ← r1+ #1`		*r1(P1) set to 1*
	`r1 ← r1+#1`	*r1(P1) set to 1*
`x ← r1`		*P1 writes its R1*
	`x ← 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
- Usually we want to avoid non-determinism
- If we compute with the same inputs we want to obtain the same results
- Not necessarily true for operations that have side effects (global variables, 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

Critical Sections

- Each thread sums into the shared variable *x,* to which it has momentary, exclusive access
- Threads take turns executing a ***critical section***
- A critical section is non-parallelizing computation

```
Begin Critical Section
    global_sum += x[i] ;
End Critical Section
```

Mutual exclusion

- Pthreads provides mutex variables (locks)
- May be CLEAR or SET
- Lock() waits if the lock is set, else sets the lock
- Unlock clears the lock if set

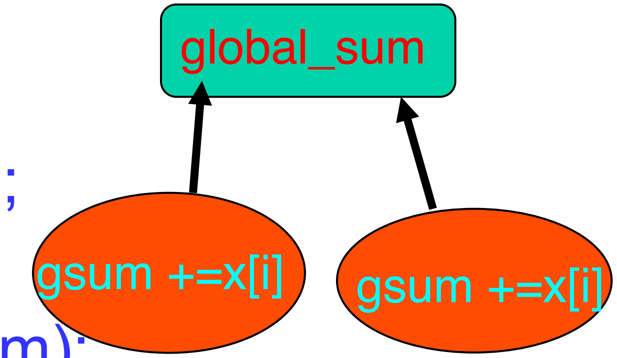
```
pthread_mutex_t mutex_sum;
pthread_mutex_init(&mutex_sum, NULL));
pthread_mutex_lock (&mutex_sum);
   global_sum += x[i] ; // Critical Section
pthread_mutex_unlock (&mutex_sum);
```

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

A performance bug

```
void *summ(void *arg){
   TID = …
  int  i0 = TID*(N/NT),  i1   = i0 + (N/NT);
  for (int64_t  i=i0;  i<i1;  i++){
      pthread_mutex_lock (&mutex_sum);
      global_sum +=  x[i] ;
      pthread_mutex_unlock (&mutex_sum);
  }
  pthread_exit(NULL); return 0;
}
```



More on Correctness

```
int64_t sum = 0;       // Global
void *sumIt(void *arg){
   int TID =  unique thread ID (arg);
   pthread_mutex_lock (&mutex_sum);
   sum += (TID+1);
   pthread_mutex_unlock (&mutex_sum);
   if (TID == 0)
        cout << "Sum of 1 : " << NT << " = " << sum << endl;
   pthread_exit(NULL); return NULL; }
```

```
% g++ sumIt.C -lpthread
% a.out 8
# threads: 8
The sum of 1 to 8 is 1
After join returns, the sum of 1 to 8 is: 36
```

Barrier synchronization

- Why was the sum reported incorrectly ?
- Don’t read a location updated by other threads that had not had the chance to produce its contribution (true dependence)
- Don’t overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)

```
pthread_mutex_lock (&mutex_sum);
sum += 2*(TID+1);
pthread_mutex_unlock (&mutex_sum);
Barrier();
if (TID == 0)
    cout << "Total sum is " << sum << endl;
```

Building a linear time barrier with locks

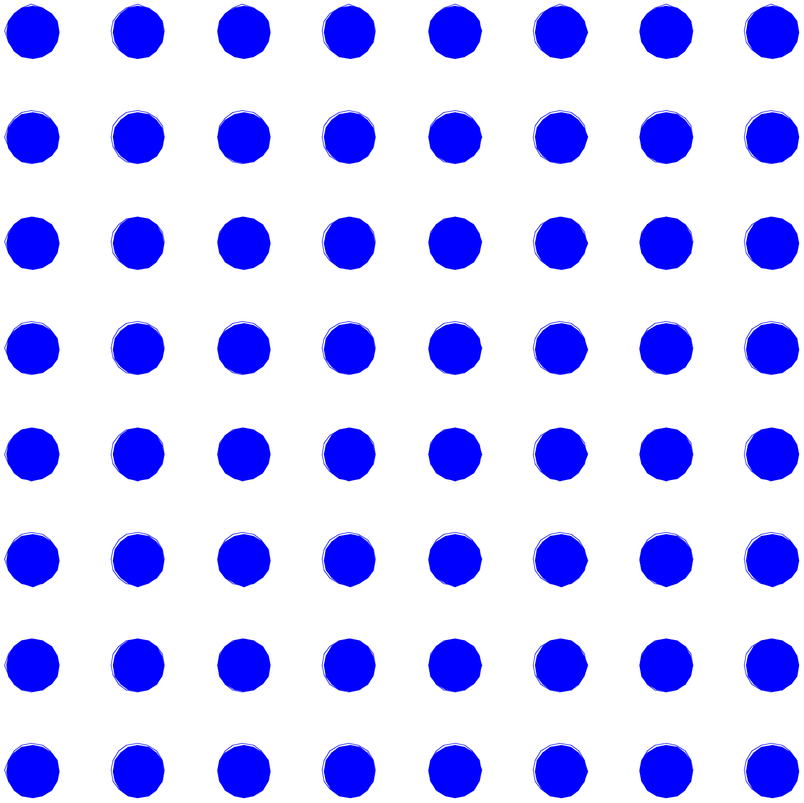
```
Mutex arrival=UNLOCKED, departure=LOCKED;
 int count=0;

 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
```

Iterative mesh methods

Mesh based methods

- Many physical problems are simulated on a uniform ***mesh*** in 1, 2 or 3 dimensions
- *Field variables* defined on a discrete set of points
- A ***mapping*** from ordered pairs to ***physical observables*** like temperature and pressure
- One application: differential equations



Differential equations

- A **differential equation** is a set of equations involving derivatives of a function (or functions), and specifies a solution to be determined under certain constraints
- Constraints often specify **boundary conditions** or **initial values** that the solution must satisfy
- When the functions have multiple variables we have a Partial Differential Equation (PDE)

  $\partial^2 u/ \partial x^2 + \partial^2 u/\partial y^2 = 0$ within a square box, $x,y \in [0,1]$

  $u(x,y) = \sin(x)*\sin(y)$ on $\partial\Omega$, perimeter of the box
- When the functions have a single variable we have an *Ordinary Differential Equation* (ODE)

  $-u''(x) = f(x),\ x\in[0,1],\ u(0) = a,\ u(1) = b$

Solving an ODE with a discrete approximation

- Solve the ODE

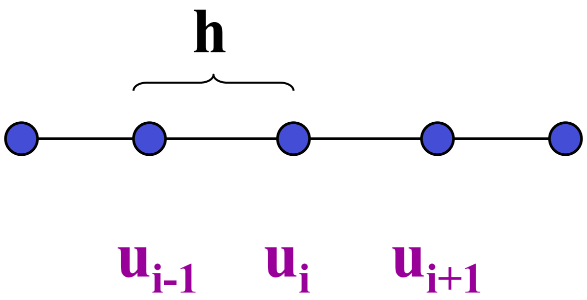
  $-u''(x) = f(x),\ x \in [0,1]$

- Define $u_i = u(i \times h)$ at points

  $x = i \times h, \qquad h = 1/(N-1)$

- Approximate the derivatives

  $u'' \approx (u(x+h) - 2u(x) + u(x-h))/h^2$



- Obtain the system of equations

  $(u_{i-1} - 2u_i + u_{i+1})/h^2 = f_i \quad i \in 1..n-2$

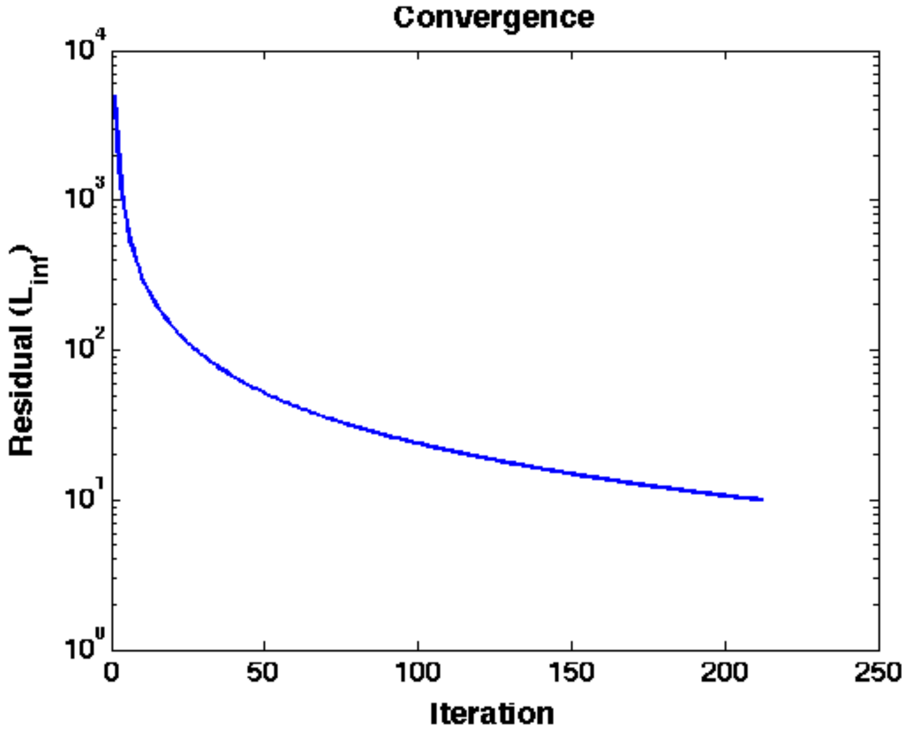
Iterative solution

- Rewrite the system of equations
  $(-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i$, $i \in 1..n-1$
- It can be shown that the following ***Gauss-Seidel*** algorithm will arrive at the solution …
- …. assuming an initial guess for the u_i

```
Repeat until the result is satisfactory
    for i = 1 : N-1
        $u_i = (u_{i+1} + u_{i-1} + h^2 f_i)/2$
    end for
end Repeat
```

Convergence

- Convergence is slow
- We reach the desired precision in $O(N^2)$ iterations



Estimating the error

- How do we know when the answer is “good enough?”
  - The computed solution has reached a reasonable approximation to the exact solution
  - We validate the computed solution in the field, i.e. wet lab experimentation
- But we often don’t know the exact solution, and must estimate the error

Using the residual to estimate the error

- Recall the equations

  $(-u_{i-1} + 2u_i - u_{i+1})/h^2 = f_i\ , i \in 1..n-1 \quad [Au = f]$

- Define the ***residual*** r_i: $\quad [r = Au-f]$

  $r_i = (-u_{i-1} + 2u_i - u_{i+1})/h^2 - f_i\ , i \in 1..n-1$

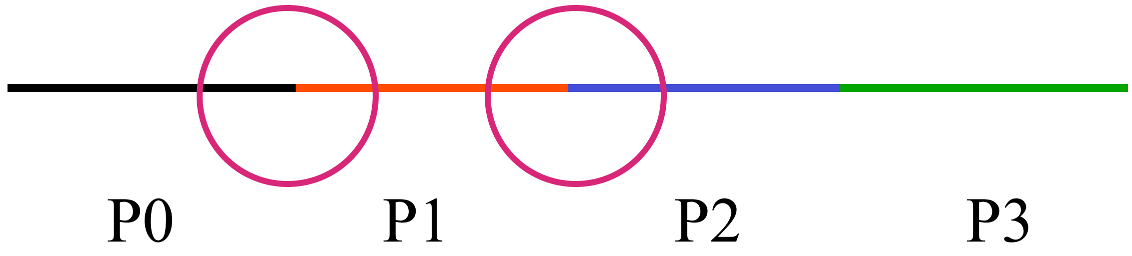
- Thus, our computed solution is correct when

  $r_i = 0$

- We can obtain a good estimate of the error by finding the maximum $r_i \ \forall i$
- Another possibility is to take the root mean square (L2 norm) $\sqrt{\sum_i r_i^2}$

Parallel implementation

- We partition the data into intervals, assigning each to a unique thread
- Each interval depends on two endpoint values that get updated by another thread



Dependences

- Our attempt to parallelize the algorithm fails: there are **loop carried dependences**
- The value of u[i] computed in iteration i depends on u[i] computed in iteration i−1

```
for i = 1 : N-1
     u[i] = (u[i-1]+u[i+1] +h*h*f[i])/2
end for
```

Parallel implementation

- Renaming the LHS of the assignment eliminates the dependences
- Two arrays u and unew
- This is Jacobi's method

```
for i = 1 : N-1
  unew[i] = (u[i-1]+u[i+1] +h*h*f[i])/2
end for
Swap u and unew
```

Tradeoffs

- We can now parallelize the algorithm, since we have eliminated the loop carried dependencies
- But we have reduced the convergence rate by about a factor of two
- Doubles the amount of work needed to solve the problem
- This kind of tradeoff is common
- Which algorithm should be used in the "fastest serial" implementation?

Convergence check

- Each thread computes the error for its assigned part of the problem
- We need a global error so that we compute a result that is consistent with the single processor implementation
- We form a global sum of the local contributions

Stencil operations in higher dimensions

- We call the numerical operator that sweeps over the solution array a **stencil operator**
- In 1D we have functions of one variable
- In n dimensions we have n variables
- In 2D:

  $\partial^2 u / \partial x^2 + \partial^2 u / \partial y^2 = \Delta u = f(x,y)$ within a square box, $x,y \in [0,1]$

  $u(x,y) = \sin(x)*\sin(y)$ on $\partial\Omega$, perimeter of the box

  Define $u_{i,j} = u(x_i, y_j)$ at points $x_i = i \times h$, $y_j = j \times h$, $h = 1/(N-1)$

- Approximate the derivatives

  $u_{xx} \approx (u(x_{i+1},y_j) + u(x_{i-1},y_j) + u(x_i,y_{j+1}) + u(x_i,y_{j-1}) - 4u(x_i, y_j)/h^2$

Jacobi's Method in 2D

- The update formula

```
for (i,j) in 0:N-1 x 0:N-1
    $u'[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1] - h^2 f[i,j])/4$
u =u'
```

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$