Shared memory programming

- Recapping from last time…
- Shared memory introduces a new set of programming issues
  - Synchronization
  - Cache management (locality and coherence)
- Certain applications are also highly amenable to automated (compiler) parallelization
- We’ll start with the most primitive programming model: threads
- We’ll then describe a high level model which is becoming popular: OpenMP
Shared memory programming model

- A collection of concurrent instruction streams, called **threads**, that share memory
- Each thread has a unique thread ID: ~ MPI rank
- We get a new kind of storage class: shared data
- Synchronization is needed when updating shared state
  - mutual exclusion
  - barriers

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
More on Threads

- A thread is similar to a procedure call with notable differences
  - A procedure call is “synchronous”: a return indicates completion
  - With a thread we get an independently schedulable unit, since a spawned thread executes asynchronously until it completes
- With a procedure call and a thread we get a local stack, and we share global storage with the caller
- A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  - But there are non-standard features some of which are outside the scope of the standard!

Programming model

- We start with a single root thread
- We employ fork-join parallelism to create new concurrently executing threads
- The root thread may execute concurrently with the spawned threads
- But at the end it waits for the spawned threads to complete
- Whether or not threads execute on different processors, and at the same time isn’t said
- The scheduler might interleave many threads on a single CPU or some other subset of the hardware
- Scheduling behavior is usually specified separately
A first program: hello world

```c
#include <pthread.h>

void *print_msg_fun( void *arg ) {
    cout << " Hello World called with " << * ( (int *) arg ) << endl; pthread_exit(0); }

int main(int argc, char** argv){
    pthread_t thrd1, thrd2; // Thread handles
    // Spawn 2 threads, each executing the same function but on different data
    int i0 = 0, i1=1;
    // The argument must be cast to (void *)
    // Do not to modify the argument -- why?
    assert(!pthread_create(&thrd1, NULL, print_msg_fun, (void *) &i0));
    assert(!pthread_create(&thrd2, NULL, print_msg_fun, (void *) &i1));
    assert(!pthread_join(thrd1, NULL));
    assert(!pthread_join(thrd2, NULL));
}
```

Synchronization

- When different threads modify shared locations, we must synchronize the writes to ensure that results are **deterministic**
- 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)
- $1 + 1 = 2$ ALWAYS
- Let’s consider a variant of Hello World that issues each thread with a unique thread ID
A second variant with synchronization

```c
// Global state
pthread_mutex_t mutex;
int _nt;
const int NT = 4;

void *print_msg( void *arg ) {  
  // Generate unique thread ids
  _nt = int(); _nt++;
  assert( !pthread_mutex_lock(&mutex)); // Begin critical section
  cout << "Hello World from thread " << tid << endl;
  assert( !pthread_mutex_unlock(&mutex)); // End critical section
}

int main(int argc, char** argv){  
  assert(!pthread_mutex_init(&mutex, NULL));
  for (int t=0; t<NT; t++)
    assert(!pthread_create(thrd+t, NULL, print_msg, (void *) &unk)); // Spawn NT threads
  for (int t=0; t<NT; t++)
    assert(!pthread_join(thrd+t, NULL)); // Join NT threads
}
```

Mutual exclusion

- In our latest program, each process will sample and increment the shared counter _nt
- The code that performs this operation is a **critical section**
- Only one thread at a time may access this code
- We use *mutual exclusion* to implement the critical section
Race conditions

- Consider the statement, assuming \_nt == 0
  \_nt++;  
- Generated code
  \- r1 ← (\_nt)  
  \- r1 ← r1 + #1  
  \- r1 → (\_nt)  
- Possible interleaving with two threads
  \begin{align*}
  &P1 &P2 \\
  r1 &← \_nt & r1 &← \_nt \quad r1(P1) \text{ gets } 0 \\
  r1 &← r1+ #1 & r1 &← r1+#1 \quad r1(P2) \text{ also gets } 0 \\
  \_nt &← r1 & \_nt &← r1 \quad r1(P1) \text{ set to } 1 \\
  \_nt &← r1 & \_nt &← r1 \quad r1(P1) \text{ set to } 1 \\
  \end{align*}

Mutual exclusion

- The outcome is non-deterministic without appropriate synchronization: there is a race condition
- To avoid the race condition, we serialize the updates to \_nt with a critical section
  \begin{verbatim}
  diff_lock.lock();  
  \_nt++;  
  diff_lock.unlock();
  \end{verbatim}
- Statement inside the lock pair executes atomically
- Initially the lock is cleared
  \- Lock() sets the lock, but waits if lock is set
  \- Unlock clears the lock, releasing one process to execute lock() and enter the critical section
Guidelines for mutual exclusion

• The lock and any other shared state are prone to contention – use a local variable to hold mydiff

    diff_lock.lock();
    _nt++;
    diff_lock.unlock();

• Since the critical section is non-parallelizing computation, keep it as short as possible

Motivating application

• Jacobi’s method for solving Poisson’s equation in two dimensions

    for  j=1 : N
    for  i=1 : M
        unew[i,j] = (u[i-1,j] + u[i+1,j] +u[i,j-1] +u[i,j+1])/4;
The code

1. procedure Solve( sharedArray2D<float> A ) // A is an (n + 2)-by-(n + 2) array
2. begin
3.    int done = FALSE;
4.    float diff;
5.    while (!done) // outermost sweep loop
6.         diff = 0;
7.         for i ← 1 to n do // sweep over interior points of grid
8.             for j ← 1 to n do
10.                     A[i,j+1] + A[i+1,j] );
11.                 diff += abs(A[i,j] – A'[i,j]);
12.         end for
13.     end for
15.     if (diff/(n^2) < TOL) done = TRUE;
16. end while
17. end procedure

- Interior n x n points updated in each sweep
- Compute error. taking differences against old solution
- Update old solution from new solution
- Continue sweeping until solution has converged

Workload decomposition

- Static assignment
  - BLOCK decomposition, we use 1D here
- Dynamic assignment : self scheduling
  - get a row index, work on the row, get a new row, repeat
- Static assignment into rows reduces concurrency from n to p, but reduces synchronization overhead
- Unlike message passing, workload assignment is specified by the subsets of the global loop bounds
- Let’s look at the tradeoffs
Shared memory parallel code

1a. Lock diff_lock; // declaration of lock to enforce mutual exclusion
1b. Barrier bar1; // barrier declaration for global synchronization between sweeps
2. Array2D<float> A; // Shared array
2b. float diff;
3. main()
4. begin
5.   read(n); read(nprocs); bar1.init(nprocs);
6.   A ← new Array2D<float>(n+2,n+2)
7.   FORK (nprocs–1, Solve, A);
8.   Solve(A); /*main process becomes a worker too*/
9.   JOIN (nprocs–1); /*wait for all child processes created to terminate*/
10. end main

• Variables declared out of the scope of any function are global to all processes, all others are local to the process

Computation showing synchronization

11. procedure Solve(SharedArray2D<float> A)
11a. int mymin = 1 + ($PID * n/nprocs); // assume that n is exactly divisible by nprocs
11b. int mymax = mymin + n/nprocs - 1
12.   while (!done) do
13.     mydiff = 0; // set global diff to 0 – why is it okay for all to do it?
13a.   bar1.sync(); // all must reach this point before anyone may modify diff
14.   for i ← mymin to mymax do
15.     for j ← 1 to n do // for each of my rows
17.     mydiff += abs(A[i,j] - A'[i,j]);
18.   endfor
19.   endfor
20.   if (diff/(n*n) < TOL) done = TRUE; // check convergence; all get same answer
21.   bar1.sync(); // all must reach this point before anyone may modify diff
22.   diff_lock.lock(); // update global diff if necessary
23.   diff += mydiff;
24.   diff_lock.unlock();
25. end while
26. end procedure
Barrier synchronization

- Ensures that no process can move on until all have arrived
- Don’t overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- Don’t read locations updated by other processes in the previous iteration until they have been produced (true dependence)
- A barrier can be built with locks

Performing reductions with pthreads

- Reductions can be expensive to implement if we aren’t careful

```c
for (int i=i0; i< i1; i++)
    sum += x[i];
```

- We will experience contention on sum
- A better way is to use a local accumulator, and then use a critical section to accumulate a global sum

```c
for (int i=i0; i< i1; i++)
    local_sum += x[i];
```
OpenMP programming

- Simpler interface than pthreads
- Parallelization handled via annotations
- See the web page www.openmp.org
- Parallel loop:

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

OMP code

```c
for i = 1 to n do
    #pragma omp parallel for shared(A,A',n) private(j)
        for j = 1 to n do
            A'[i,j] = ...
        end for
end for
```
OMP code

for \(i = 1\) to \(n\) do
  #pragma omp parallel for shared(A,A',n) private(j) reduction(+:diff)
  for \(j = 1\) to \(n\) do
    \(A'[i,j] = \ldots\)
    \(\text{diff} += \ldots\)
  end for
end for

How do does OpenMP distribute the loops?

- With static decomposition, the process is straightforward: each process gets a unique range of indices based on its thread ID
- But with irregular problems, or when processor performance is not predictable, we can’t use static decomposition
- OpenMP provides a dynamic option
- Relies on processor self-scheduling
How does processor self-scheduling work?

- Processors “schedule” themselves
- Sample a shared counter to obtain work
- Adjust work granularity (“chunk size”) to trade off the overhead of sampling the queue against increased load imbalance
- Also used with work queues

![Diagram showing running time, load imbalance, and increasing granularity](image)

How self-scheduling works

- Self-scheduling is a statistical technique
- We need enough pieces of work so that the statistical distribution of costs balances
- But this requirement conflicts with desire to maintain low overheads and low surface-to-volume effects

![Diagram showing running time, load imbalance, and increasing granularity](image)
Details of self scheduling

```c
$omp parallel
while (!done) do
    mydiff = diff = 0;
$omp barrier
    while (getNextChunk(&mymin,&mymax )) do
        for i = mymin to mymax do
            for j = 1 to n do
                A'[i,j] = .... ;
        end for
    end do
$omp barrier
    ReduceSum(diff,mydiff);
$omp barrier
    if (diff/(n*n) < TOL) done = TRUE;
$omp barrier
end do
```

More details of self scheduling

```c
Selfscheduler S;

Boolean getNextChunk(int * mymin, int * mymax ){
    $omp critical
    {
        k = S.counter += S.chunksize;
    }
    mymin = k;
    mymax = k + S.chunksize;
}
```