Lecture 12

Shared memory programming
Announcements

• UPC lecture on Thursday
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
- 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
OpenMP programming

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

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

```c
#pragma omp parallel // Begin a parallel construct
{
  // form a team
  ...
  // Each team member executes the same code
  #pragma omp sections // Begin worksharing
  {
    //
    #pragma omp section // One unit of work
    {x = x + 1;}
    #pragma omp section // Another unit of work
    {x = x + 1;}
  } // Wait until both units of work complete
} // End of Parallel Construct; disband team
// and 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
  \[ \begin{align*}
  & \text{P1} & \text{P2} \\
  r1 & \leftarrow x & r1 & \leftarrow x \quad \text{r1(P1) gets 0}
  \\
  r1 & \leftarrow r1 + #1 & r1 & \leftarrow r1 + #1 \quad \text{r1(P1) set to 1}
  \\
  x & \leftarrow r1 & x & \leftarrow r1 \quad \text{r2(P2) also gets 0}
  \\
  
  & \text{P1 writes its R1} & \text{P1 writes its R1} & \text{P1 writes its R1} & \text{P2 writes its R1}
  \end{align*} \]
Race conditions

- A *Race* condition arises within an application 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 usually we want to avoid it.
- 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

```c
#pragma omp parallel // Begin a parallel construct
{ // form a team
    // Each team member executes the same code
    #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
    } // End of Parallel Construct; disband team
    // and continue serial execution
```
How does mutual exclusion work?

• A simple solution is to use a mutex variable
• E.g. provided by pthreads

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

- Thread creation costs 10s of μsecs
- Moving data in shared memory is cheaper than passing a message through shared memory

www.llnl.gov/computing/tutorials/pthreads

<table>
<thead>
<tr>
<th></th>
<th>create</th>
<th>MPI Shared Mem BW</th>
<th>Mem to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 2.4 GHz Xeon</td>
<td>34 μs</td>
<td>0.3 GB/s</td>
<td>4.3 GB/s</td>
</tr>
<tr>
<td>Intel 1.4 GHz Itanium 2</td>
<td>42 μs</td>
<td>1.8</td>
<td>6.4</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4</td>
<td>30 μs</td>
<td>2.1</td>
<td>11</td>
</tr>
</tbody>
</table>
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
Building a linear time barrier with locks

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

void Barrier( )
    arrival.lock( ); // atomically count the
    count++;
    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
Motivating application

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

\[
\text{for } j=1 : N \\
\text{for } i=1 : M \\
\text{unew}[i,j] = (u[i-1,j] + u[i+1,j] + u[l,j-1] + u[l,j+1])/4;
\]
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) do // 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.                diff += abs(A[i, j] - A'[i, j]);
11.             end for
12.         end for
13.     A[:, :] = A'[ :, :] // Set old solution = new solution
14.     if (diff/(n^2) < TOL) done = TRUE;
15. **end while**
16. **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

10/31/06 Scott B. Baden/CSE 260/Fall 2006
procedure Solve (SharedArray2D<float>A)  // A is an (n + 2)-by-(n + 2) array

begin

  int done = FALSE;

  while (!done) do

    float diff = 0;

    forall i ← 1 to n do
      forall j ← 1 to n do
        diff += abs(A[i,j] – A'[i,j]);
      end for
    end for

    diff = REDUCE_SUM(diff)

    A[:,:] = A'[:,:]

    if (diff/(n^2) < TOL) done = TRUE;

  end while

end procedure

Exposing the parallelism

Automatic variables local to a processor

// outermost sweep loop

// sweep over interior points of grid

P4
P5
P6
P7
P8
P0
P1
P2
P3
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
OMP code

mymin = 1 + ($PID + n/nprocs), mymax = mymin + n/nprocs -1;

while (!done) do
    mydiff = diff = 0;
    $omp barrier
    for i = mymin to mymax do
        for j = 1 to n do
            A'[i,j] = …
            mydiff+= …
        end for
    end for
$omp critical
    { diff += mydiff }
$omp barrier
    if (diff / (n*n) < Tolerance) done = TRUE;
$omp barrier
end while

• 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)
Reductions in OpenMP

• OpenMP uses a local accumulator, which it then accumulates globally when the loop is over
• What does this remind us of?

```c
#pragma omp parallel reduction(+:sum)
for (int i=i0; i< i1; i++)
    sum += x[i];
```
#include <pthread.h>

void *PrintHello(void *threadid){
    cout << "Hello World from " << (int)threadid << endl; pthread_exit(NULL); }

int main(int argc, char *argv[]){
    int NT; cin >> NT; pthread_t th[NT];
    for(int t=0;t<NT;t++)    
        assert(!pthread_create(&th[t], NULL, PrintHello, (void *)t));

    pthread_exit(NULL);}
Computing the dot product

- Dot product: dotprod_mutex.c in www.llnl.gov/computing/tutorials/pthreads/samples

```c
pthread_t thrd[NT];
double *x = (double*) malloc (NT*N*sizeof(double));
for(i=0;i<NT;i++)
    pthread_create( &thrd[i], &attr, dotprod, (void *)i);
for(i=0;i<NT;i++) {
    pthread_join( thrd[i], (void **)&status);
```
The computation

```c
void *dotprod(void *arg){
    int TID = (int)arg;
    int  i0 = TID*N,  i1   = i0 + N;
    double mysum = 0;
    for ( i=i0;  i<i1;  i++)
        mysum +=  x[i] * y[i];
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);
    pthread_exit((void*) 0);
}
```
Measuring shared memory system performance

- Dot product: 2 mems / 2 flops
- DOT_PROD = SUM(x[:]*y[:])
- On valkyrie, 1M array per processor
  - NT=1: 44.6 MFlops
  - NT=2: 68.0 MFlops
- On a dual 1.8GHz Opteron: 250 and 500 MFlops
- SDSC Datastar, 1.5GHz Power4: 289&586 MFlops
Reducing memory impact

• Insert operations that use registers only

```c
for (int i=0; i < N; i++){
    result += x[i]*y[i];
    for (int j=0; j < K; j++) result *= c;
}
```

• On valkyrie, 1M array per processor

<table>
<thead>
<tr>
<th>NT\K</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.6</td>
<td>121</td>
<td>71</td>
<td>92</td>
<td>113</td>
<td>150</td>
<td>172</td>
<td>165</td>
</tr>
<tr>
<td>2</td>
<td>68.8</td>
<td>157</td>
<td>114</td>
<td>151</td>
<td>198</td>
<td>270</td>
<td>319</td>
<td>322</td>
</tr>
</tbody>
</table>
Workload decomposition

• Static assignment: BLOCK decomposition
• 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
  
  ```c
  #pragma omp parallel private(i) shared(n)
  {
  #pragma omp for
  for(i=0; i < n; i++)
    work(i);
  }
  ```
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 or work queue 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 the relationship between running time, load imbalance, high overheads, and increasing granularity.]
Details of self scheduling

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

SelfScheduler S(n,P,CSize);

Boolean getNextChunk(int * mymin, int * mymax ){

    //omp critical
    {
        k = S.counter += S.ChunkSize;
    }
    mymin = k;
    mymax = k + S.chunkSize;
}
Guided self-scheduling

- Observation: we need many more pieces of work than there are processors (say 10 times) at the time we are sampling the counter
- Adjust or “guide” task granularity in proportion to the amount of remaining work
- For example, $g = \frac{N_{\text{Remain}}}{10P}$
- When there’s lots of work to do, we hand out large chunks. Toward the end, we hand out smaller chunks
Hybrid MPI + thread programming

- Take advantage of faster shared memory on-node
- Higher complexity than “flat MPI”
- KeLP2: www.cse.ucsd.edu/groups/hpcl/scg/Research/MT.html
Multi-tier model

• Programs have 3 levels of control:
  – **Collective** level operations performed on all nodes
  – **Node** level operations performed on one node
  – Processor level operations performed on a single CPU
Managing locality

- Well performed programs use tricks from message passing
- Global address space is helpful
- Single sided communication: the best of both worlds: put and get
Next time

• Parallel programming with Uniform Parallel C