Lecture 5

Under the hood of OpenMP
Vectorization (SIMD and SSE)
Announcements

• Turnin has been enabled for Programming Lab #1
• Don’t wait until the last minute to turnin: be sure you have all required files: teameval.txt, etc.
• Using the ‘at’ command
• A2 coming on Friday: GPUs
Stencil Methods in 3D
Motivating application

• Solve Laplace’s equation in 3-D with Dirichlet Boundary conditions
  \[ \Delta u = 0, \quad u = f \text{ on } \partial \Omega \]

• Building block: iterative solver using Jacobi’s method
  (7-point stencil)

\[
\text{for } (i,j,k) \text{ in } 1:N \times 1:N \times 1:N \\
u'[i,j,k] = \frac{u[i-1,j,k] + u[i+1,j,k] + u[i,j-1,k] + u[i,j+1,k] + u[i,j,k+1] + u[i,j,k-1]}{6.0}
\]
Memory representation and access

3D grid

U'

 Linear array space

10/7/10

©2010 Scott B. Baden /CSE 260/ Fall '10
Processor Geometry

![Graph showing speedup vs processor geometry with different N values.]

- N=256
- N=320
- N=400
- N=504
- N=640
- N=808

**Processor Geometry**

**Speedup**

**1x1x1, 2x1x1, 2x2x1, 1x2x2, 1x2x4, 2x1x4, 2x2x4, 4x2x1, 1x1x8, 1x8x1, 8x1x1, 2x2x2**
3D Jacobi in OpenMP

#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of openMP threads: %d\n", nthreads);
    }
}
#endif
Computational loop

FLOAT c = 1 / 6.0, h = 1.0, c2 = h * h;

for (it = 0; it < nIters; it++) {
#pragma omp parallel shared(U, Un, b, nx, ny, nz, c2, c) private(i, j, k)
#pragma omp for schedule(static, bi)
    for (int i = 1; i <= nx; i++)
        for (int j = 1; j <= ny; j++)
            for (int k = 1; k <= nz + 1; k++)
                Un[i][j][k] = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - c2 * b[i-1][j-1][k-1]);

    Grid3D tmp = U;
    U = Un;
    Un = tmp
}
Computing the residual

```c
FLOAT resid7(Grid3D U, Grid3D B, const int nx, const int ny, const int nz){
double c = 1 / 6.0, err=0;
#pragma omp parallel shared(U,B,c)
#pragma omp for reduction(+:err)
for (int i=1; i<=nx; i++)
    for (int j=1; j<=ny; j++)
        for (int k=1; k<=nz; k++){
            FLOAT du = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - 6.0*B[i-1][j-1][k-1]);
            FLOAT r = B[i-1][j-1][k-1] - du;
            err = err + r*r;
        }
return sqrt(err)/(float)((nx+2)*(ny+2)*(nz+2));
}
```
Under the hood of openmp: pthreads
Threads 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
POSIX Threads (Pthreads)

• A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
• Lower level than OpenMP
• Beware of non-standard features
Storage

- 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
Overheads

- 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

<table>
<thead>
<tr>
<th>Processor</th>
<th>CPU/node</th>
<th>Fork (μs)</th>
<th>Create (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD 2.4 GHz Opteron</td>
<td>8</td>
<td>41.1</td>
<td>0.66</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575</td>
<td>8</td>
<td>64.2</td>
<td>1.75</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon</td>
<td>2</td>
<td>55.0</td>
<td>1.64</td>
</tr>
<tr>
<td>Intel 1.4 GHz Itanium 2</td>
<td>4</td>
<td>54.5</td>
<td>2.03</td>
</tr>
</tbody>
</table>
Coding with pthreads

#include <pthread.h>
#include <assert.h>

void *Hello(void *tid) {
    sleep(3);
    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(long t=0; t<NT; t++)
        assert(!pthread_create(&th[t], NULL, Hello, (void *)t));
    for(long 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:
long long *x, global_sum, N, NT;

Main:
for (int i=0; i < N; i++) x[i] = i+1;
global_sum = 0;

for(long 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;
void *summ(void *arg) {
    long long _tid = reinterpret_cast<long long>(arg);
    int TID = _tid;
    int i0 = TID*(N/NT), i1 = i0 + (N/NT);
    for ( i=i0;  i<i1;  i++)
        global_sum += x[i];
    pthread_exit(NULL); return 0;
}
Race conditions

- A *Race* condition arises when the timing of accesses to shared memory can affect the outcome.

- Consider this statement, assume $x == 0$
  
  ```
  x = x + 1;
  ```

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

- Possible interleaving with two threads

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th></th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>r1</td>
<td>r1 + #1</td>
<td></td>
<td>r1 + #1</td>
</tr>
<tr>
<td>x</td>
<td>r1</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>x</td>
<td>r1</td>
<td></td>
<td>r1</td>
</tr>
</tbody>
</table>

  - $r1(P1)$ 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
Consequences of race conditions

• 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)
• We need to take steps to avoid race conditions through appropriate program synchronization
  ♦ Critical sections
  ♦ Barriers
  ♦ Atomic functions
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

```c
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

```c
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

• The program won’t always fail!
• 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

```c
void *summ(void *arg){
    long long _tid = reinterpret_cast<long long>(arg);
    int TID = _tid;
    int i0 = TID*(N/NT), i1 = i0 + (N/NT);
    for ( 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

```c
long long 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 29
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)

```c
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
Multithreaded Solver

Local $\text{mymin} = 1 + (TID \times n/NT)$,
    $\text{mymax} = \text{mymin} + n/NT - 1$;

Global resid, $U[:, :], U^\text{new}[:, :]$

Local done = FALSE;

while (!done) do
    Local myResid = 0;
    resid = 0;
    update $U^\text{new}$ and myResid
    resid += myResid; // Atomic
    if (resid < Tolerance) done = TRUE;
U[mymin:mymax, :] = $U^\text{new}$[mymin:mymax, :];
end while

for $i = \text{mymin}$ to $\text{mymax}$ do
    for $j = 1$ to $n$ do
        $U^\text{new}[i,j] = \ldots$
        myresid += $\ldots$
    end for
end for

Is this code correct?
Multithreaded Solve()

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

Global resid, U[:,;:], U^{new}[:,;]

Local done = FALSE;

while (!done) do
   Local myResid = 0;
   BARRIER
   resid = 0;
   BARRIER
   update U^{new} and myResid
   resid += myResid // Atomic
   BARRIER
   if (resid < Tolerance) done = TRUE;
   U[mymin:mymax,;] = U^{new}[mymin:mymax,;];
end while

for i = mymin to mymax do
   for j = 1 to n do
      U^{new}[i,j] = …
      myresid += …
   end for
end for

Does this code use minimal synchronization?