Lecture 8
Announcements
Recapping from last time:
Minimal barrier synchronization in odd/even sort

Global bool AllDone;
for (s = 0; s < MaxIter; s++) {
    barr.sync();
    if (!TID)
        AllDone = true;
    barr.sync();
    int done = Sweep(Keys, OE, lo, hi);
    barr.sync();
    done &= Sweep(Keys, 1-OE, lo, hi);
    mtx.lock();
    AllDone &= done;
    mtx.unlock();
    barr.sync();
    if (allDone)
        break;
}
The barrier between odd and even sweeps

```c
int done = Sweep(Keys, OE, lo, hi);
barr.sync();
done &= Sweep(Keys, 1-OE, lo, hi);
```
Is it necessary to use a barrier between the odd and even sweeps?

A. Yes, there’s no way around it
B. No
C. We only need ensure that each processor’s neighbors are done
D. Not sure

```c
int done = Sweep(Keys, OE, lo, hi);
barr.sync();
done &= Sweep(Keys, 1-OE, lo, hi);
```
Today’s lecture

- OpenMP
OpenMP

- A higher level interface for threads programming [http://www.openmp.org](http://www.openmp.org)
- Parallelization via source code annotations
- All major compilers support it, including gnu
- Compare with explicit threads programing

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

```c
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i< i1; i++)
    work(i);
```
OpenMP’s Fork-Join Model

- A program begins life as a single thread
- Enter a parallel region, spawning a team of threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope…
  - The team of threads synchronize at a barrier and are disbanded; they enter a wait state
  - Only the initial thread continues
- Thread teams can be created and disbanded many times during program execution, but this can be costly
- A clever compiler can avoid many thread creations and joins
Fork join model with loops

cout << “Serial\n”;
N = 1000;
#pragma omp parallel{
#pragma omp for
for (i=0; i<N; i++)
  A[i] = B[i] + C[i];
#pragma omp single
M = A[N/2];
#pragma omp for
for (j=0; j<M; j++)
  p[j] = q[j] - r[j];
}
Cout << “Finish\n”;

Seung-Jai Min
Loop parallelization

• The translator automatically generates appropriate local loop bounds
• Also inserts any needed barriers
• We use private/shared clauses to distinguish thread private from global data
• Handles irregular problems
• Decomposition can be static or dynamic

```c
#pragma omp parallel for shared(Keys) private(i) reduction(&:done)
for  i = OE; i to N-2 by 2
    if (Keys[i] > Keys[i+1]) swap Keys[i] ↔ Keys[i+1]; done *= false;  
end do
return done;
```
Another way of annotating loops

• These are equivalent

```c
#pragma omp parallel
{
#pragma omp for
  for (int i=1; i< N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h
}
```

```c
#pragma omp parallel for
for (int i=1; i< N-1; i++)
  a[i] = (b[i+1] - b[i-1])/2h
```
Variable scoping

- Any variables declared outside a parallel region are shared by all threads
- Variables declared inside the region are private
- **Shared & private** declarations override defaults, also usefule as documentation

```c
int main (int argc, char *argv[]) {
 double a[N], b[N], c[N];
 int i;

#pragma omp parallel for shared(a,b,c,N) private(i)
 for (i=0; i < N; i++)
   a[i] = b[i] = (double) i;

#pragma omp parallel for shared(a,b,c,N) private(i)
 for (i=0; i<N; i++)
   c[i] = a[i] + sqrt(b[i]);
```
Dealing with loop carried dependences

• OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code

    int* fib = new int[N];
    fib[0] = fib[1] = 1;
    #pragma omp parallel for num_threads(2)
    for (i=2; i<N; i++)
        fib[i] = fib[i-1]+ fib[i-2];

• Sometimes we can restructure an algorithm, as we saw in odd/even sorting

• OpenMP may warn you when it is doing something unsafe, but not always
Why dependencies prevent parallelization

• Consider the following loops

```
#pragma omp parallel
{
#pragma omp for nowait
for (int i=1; i< N-1; i++)
    a[i] = (b[i+1] – b[i–1])/2h
#pragma omp for
for (int i=N–2; i>0; i––)
    b[i] = (a[i+1] – a[i–1])/2h
}
```

• Why aren’t the results incorrect?
Why dependencies prevent parallelization

• Consider the following loops
  
  ```
  #pragma omp parallel
  {#pragma omp for nowait
   for (int i=1; i< N-1; i++)
     a[i] = (b[i+1] – b[i-1])/2h
  #pragma omp for
  for (int N-2; i>0; i--)
     b[i] = (a[i+1] – a[i-1])/2h
  }
  ```

• Results will be incorrect because the array a[], in loop #2, \textit{depends} on the outcome of loop #1
  (a \textit{true dependence})
  
  ‣ We don’t know when the threads finish
  ‣ OpenMP doesn’t define the order that the loop iterations will be incorrect
Barrier Synchronization in OpenMP

• To deal with true- and anti-dependences, OpenMP inserts a barrier (by default) between loops:

```c
for (int i=0; i < N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h
BARRIER
for (int i=N-1; i >= 0; i--)
    b[i] = (a[i+1] - a[i-1])/2h
```

• No thread may pass the barrier until all have arrived hence loop 2 may not write into b until loop 1 has finished reading the old values

• Do we need the barrier in this case? Yes

```c
for (int i=0; i < N-1; i++)
    a[i] = (b[i+1] - b[i-1])/2h
BARRIER?
for (int i=N-1; i >= 0; i--)
    c[i] = a[i]/2;
```
Which loops can OpenMP parallelize, assuming there is a barrier before the start of the loop?

A. 1 & 2
B. 1 & 3
C. 3 & 4
D. 2 & 4
E. All the loops

All arrays have at least N elements
How would you parallelize loop 2 by hand?

1. for i = 1 to N-1
   A[i] = A[i] + B[i-1];

2. for i = 0 to N-2
How would you parallelize loop 2 by hand?

for i = 0 to N-2

for i = 0 to N-2
A[i+1] = A[0] + i;
To ensure correctness, where must we remove the nowait clause?

A. Between loops 1 and 2
B. Between loops 2 and 3
C. Between both loops
D. None

```c
#pragma omp parallel for shared(a,b,c) private(i)
  for (i=0; i<N; i++)
    c[i] = (double) i
#pragma omp parallel for shared(c) private(i) nowait
  for (i=1; i<N; i+=2)
    c[i] = c[i] + c[i-1]
#pragma omp parallel for shared(c) private(i) nowait
  for (i=2; i<N; i+=2)
    c[i] = c[i] + c[i-1]
```
Exercise: removing data dependencies

• How can we split into this loop into 2 loops so that each loop parallelizes, the result it correct?

  ‣ B initially: 0 1 2 3 4 5 6 7
  ‣ B on 1 thread: 7 7 7 7 11 12 13 14

#pragma omp parallel for shared (N,B)
for i = 0 to N-1
  B[i] += B[N-1-i];
Splitting a loop

• For iterations $i = N/2+1$ to $N$, $B[N-i]$ reference newly computed data

• All others reference “old” data

• $B$ initially: $0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7$

• Correct result: $7 \ 7 \ 7 \ 7 \ 11 \ 12 \ 13 \ 14$

```c
#include <omp.h>

for i = 0 to N-1
    B[i] += B[N-i];

#pragma omp parallel for ... nowait
for i = 0 to N/2-1
    B[i] += B[N-1-i];

for i = N/2+1 to N-1
    B[i] += B[N-1-i];
```
Reductions in OpenMP

• In some applications, we reduce a collection of values down to a single global value
  ‣ Taking the sum of a list of numbers
  ‣ Decoding when Odd/Even sort has finished

• OpenMP avoids the need for an explicit serial section

```c
int Sweep(int *Keys, int N, int OE, ){
bool done = true;
#pragma omp parallel for reduction(&:done)
  for (int i = OE; i < N-1; i+=2) {
    if (Keys[i] > Keys[i+1]){
      Keys[i] ↔ Keys[i+1];
      done &= false;
    }
  }
  //All threads ‘and’ their done flag into a local variable
  //and store the accumulated value into the global
return done;
}
```
Reductions in OpenMP

- In some applications, we reduce a collection of values down to a single value
  - Taking the sum of a list of numbers
  - Decoding when Odd/Even sort has finished
- OpenMP avoids the need for an explicit serial section

```c
int Sweep(int *Keys, int N, int OE, ){
    bool done = true;
    #pragma omp parallel for reduction(&:done)
    for (int i = OE; i < N-1; i+=2) {
        if (Keys[i] > Keys[i+1]){
            Keys[i] ← Keys[i+1];
            done &= false;
        }
    }
    //All threads ‘and’ their done flag into the local variable
    return done;
}
```
Which functions may we use in a reduction?

A. Add
   \( a_0 + a_1 + \ldots + a_{n-1} \)

B. Subtract
   \( a_0 - a_1 - \ldots - a_{n-1} \)

C. Logical And
   \( a_0 \land a_1 \land \ldots \land a_{n-1} \)

D. A and B

E. A and C
Odd-Even sort in OpenMP

for s = 1 to MaxIter do
    done = Sweep(Keys, N, 0);
    done &= Sweep(Keys, N, 1);
    if (done) break;
end do

int Sweep(int *Keys, int N, int OE){
    bool done=true;
    #pragma omp parallel for shared(Keys) private(i) reduction(&:done)
    for (i = OE; i < N-1; i+=2) {
        if (Keys[i] > Keys[i+1]){
            int tmp = Keys[i];
            Keys[i] = Keys[i+1];
            Keys[i+1] = tmp;
            done *= false;
        }
    }
    return done;
}

Scott B. Baden / CSE 160 / Wi '16
Why isn’t a barrier needed between the calls to sweep()?

A. The calls to sweep occur outside parallel sections
B. OpenMP inserts barriers after the calls to Sweep
C. OpenMP places a barrier after the for i loop inside Sweep
D. A & C
E. B & C

for s = 1 to MaxIter do
    done = Sweep(Keys, N, 0);
    done &= Sweep(Keys, N, 1);
    if (done) break;
end do

int Sweep(int *Keys, int N, int OE){
    bool done=true;
    #pragma omp parallel for shared(Keys) private(i) reduction(&:done)
    for i = OE; i to N-2 by 2
        if (Keys[i] > Keys[i+1]) {swap Keys[i] ↔ Keys[i+1]; done &= false; }
    end do
return done;