15-418/618 Recitation: Open MP

• February 23, 2018
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
• Gcc 4.8 supports OpenMP version 3.1 [https://gcc.gnu.org/wiki/openmp](https://gcc.gnu.org/wiki/openmp)
• 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 = $TID\ast n/$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

Credit: Scott Baden, CSE@UCSD
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”;`
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 = 0E; 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

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

```
#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 useful 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]);

    return 0;
}
```

Credit: Scott Baden, CSE@UCSD
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

```c
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, e.g. odd-even sorting.
- OpenMP may warn you when it is doing something unsafe, but not always
Why dependencies prevent parallelization

• Consider the following loops

```c
#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

```c
#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, depends on the outcome of loop #1 (a 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:
  
  ```
  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
  
  ```
  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

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

All arrays have at least N elements

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

2. for i = 0 to N-2

3. for i = 0 to N-1 step 2

4. for i = 0 to N-2 {
   A[i] = B[i];
   C[i] = A[i] + B[i];
   E[i] = C[i+1];
}

Credit: Scott Baden, CSE@UCSD
Which loops can OpenMP parallelize, assuming there is a barrier before the start of the loop?

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

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

2. for i = 0 to N-2  

3. for i = 0 to N-1 step 2  

4. for i = 0 to N-2 {  
   A[i] = B[i];  
   C[i] = A[i] + B[i];  
   E[i] = C[i+1];  
}

All arrays have at least N elements

Credit: Scott Baden, CSE@UCSD
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

```
#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]
```
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 this loop into 2 loops so that each loop parallelizes, and the result is 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
#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];
for i = 0 to N-1
    B[i] += B[N-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;
            }
        }  
    return done;
} //All threads ‘and’ their done flag into a local variable
    // and store the accumulated value into the global

Credit: Scott Baden, CSE@UCSD
```
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

```cpp
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. LogicalAnd \[ a_0 \land a_1 \land \ldots \land a_{n-1} \]
D. A and B
E. A, B and C
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. LogicalAnd  \( a_0 \land a_1 \land \ldots \land a_{n-1} \)

D. A and B

E. A, B 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;
}
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

```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;
}
```

Credit: Scott Baden, CSE@UCSD
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

```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;
}
```

Credit: Scott Baden, CSE@UCSD
Another way of annotating loops

- These are equivalent
- Why don’t we need to declare private(i)?

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

```c
#pragma omp parallel for shared(a,b) schedule(static)
for (int i=1; i< N-1; i++)
  a[i] = (b[i+1] - b[i-1])/2h
```
The No Wait clause

• Removes the barrier after an omp for loop
• Why are the results incorrect?
  ➔ We don’t know when the threads finish
  ➔ OpenMP doesn’t define the order that the loop iterations will be incorrect

```c
#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 isn’t a barrier needed between the calls to sweep()?

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

Credit: Scott Baden, CSE@UCSD
Parallelizing a nested loop with OpenMP

• Not all implementations can parallelize inner loops

• We parallelize the outer loop index

```c
#pragma omp parallel private(i) shared(n)
#pragma omp for
for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
        V[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] - h2f[i,j])/4
    }
```

• Generated code

```c
mymin = 1 + ($TID * n/NT),
mymax = mymin + n/NT-1
for(i=mymin; i < mymax; i++)
    for(j=0; j < n; j++)
        V[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] - h2f[i,j])/4
Barrier();
```

Credit: Scott Baden, CSE@UCSD
An application: Matrix Vector Multiplication
Application: Matrix Vector Multiplication

double **A, *x, *y;
#pragma omp parallel shared(A,x,N)
#pragma omp for
for (i=0; i<N; i++)
    y[i] = 0.0;
for (j=0; j<N; j++)
    y[i] += A[i][j] * x[j];

\[
\begin{array}{cccc}
    a_{00} & a_{01} & \cdots & a_{0,n-1} \\
    a_{10} & a_{11} & \cdots & a_{1,n-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{array}
\begin{array}{c}
x_0 \\
x_1 \\
\vdots \\
x_{n-1}
\end{array}
= \begin{array}{c}
y_0 \\
y_1 \\
\vdots \\
y_{m-1}
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Support for load balancing in OpenMP

- OpenMP supports Block Cyclic decompositions with chunk size

```c
#pragma omp parallel for schedule(static, 2)
for ( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++ ){
        do
            z = z² + c
        while (|z| < 2 )
    }
}
```
OpenMP supports self scheduling

• Adjust task granularity with a chunksize

```c
#pragma omp parallel for schedule(dynamic, 2)
for( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++ ){
        do
            z = z^2 + c
        while (|z| < 2 )
    }
}
```

Credit: Scott Baden, CSE@UCSD
Iteration to thread mapping in OpenMP

#pragma omp parallel shared(N,iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
    iters[i] = omp_get_thread_num();

N = 9, # of openMP threads = 3 (no schedule)
0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N=9: 0 0 1 1 2 2 0 0 1
Initializing Data in OpenMP

- We allocate heap storage outside a parallel region
- But we should initialize it inside a parallel region
- Important on NUMA systems, which account for most servers  
  
  [http://goo.gl/ao02C0](http://goo.gl/ao02C0)

```c
double **A;
A = (double**) malloc(sizeof(double*)*N + sizeof(double)*N*N);
assert(A);

#pragma omp parallel private(j) shared(A,N)
for(j=0;j<N;j++)
  A[j] = (double *)(A+N) + j*N;

#pragma omp parallel private(i,j) shared(A,N)
for ( j=0; j<N; j++ )
  for ( i=0; i<N; i++ )
    A[i][j] = 1.0 / (double) (i+j-1);
```

Credit: Scott Baden, CSE@UCSD
OpenMP is also an API

- But we don’t use this lower level interface unless necessary
- Parallel for is much easier to use

```c
#ifndef _OPENMP
#include <omp.h>
#endif

int tid=0, nthrds,1;
#pragma omp parallel
{
    #ifdef _OPENMP
    tid = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    #endif
    int i0=(n/nthrds)*tid, i1=i0+n/nthrds;
    for(i=i0; i < i1; i++)
        work(i);
}
```

Credit: Scott Baden, CSE@UCSD
Summary: what does OpenMP accomplish for us?

- Higher level interface simplifies the programmer’s model
- Spawn and join threads, “Outlining” code into a thread function
- Handles synchronization and partitioning
- If it does all this, why do you think we need to have a lower level threading interface?

Credit: Scott Baden, CSE@UCSD