OpenMP 2

Using some slides from Peter Pacheco: Introduction to Parallel Programming, Chapter 5

Even easier

One thing that we are likely interested in parallelizing is a loop.

Can we just "thread" the loop, the rest being nicely serial.

Of course!

omp for

`#pragma omp parallel for …`

parallelize the for block that follows.

Unlike the `parallel` directive, it **automatically** divides the loop up into multiple threads.

trap-for

Note we do a lot less work and let OMP do a lot for us.

How it divides is system-dependent, but roughly speaking evenly, blockwise, by the:

number of iterations / number of threads.

Limits

There are various limits on the kind of loop that you can automate:

- for loops only (no while, no do-while)
- number of iterations can be calculated (no inifined loops, no non-local exits such as a break)
- in "canonical form"

Legal forms for parallelizable for statements

```
                                                   index++
                                                   ++index
                             index < end           index--
                             index <= end          --index
for ( index = start ;        index >= end    ;     index += incr                )
                             index > end           index -= incr
                                                   index = index + incr
                                                   index = incr + index
                                                   index = index - incr
```

more caveats

For the "*for* loop" that follows the *for* directive:

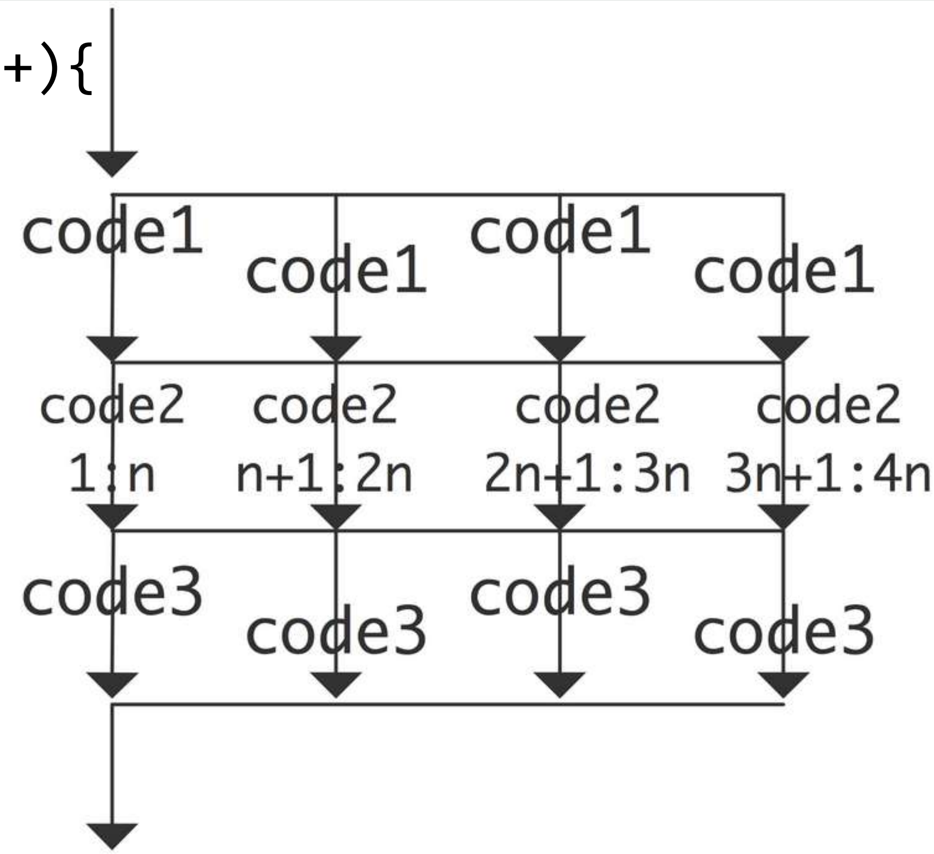
- It must not have a break statement
- The loop control variable must be an integer
- The initialization expression of the "*for* loop" must be an integer assignment.
- The logical expression must be one of $<, \leq, >, \geq$
- The increment expression must have integer increments or decrements only.

Perspective

- parallel section creates a SPMD construct, each thread running the same code
- a work-sharing construct (such as the for) is a way to divide work among the team. It does not spawn threads but it works with the team to assign work.

4 threads

```
#pragma omp parallel {
    code1();
#pragma omp for
   for (i=1; i<=4*N; i++){
      code2();
   }
   code3();
}
```



Work Sharing (more)

- Within the scope of a parallel directive, work-sharing directives allow concurrency between iterations or tasks
- Work-sharing constructs do not create new threads
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.

Will this work?

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

What happened?

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #1

loop dependency

```
double factor = 1.0;
double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

Classification of Data Dependences

- A data dependence is called loop-carried if the two statements involved in the dependence occur in different iterations of the loop.
- Let the statement executed earlier in the sequential execution be loop S1 and let the later statement be S2.
  - Flow dependence (Read after Write: RAW): the memory location is written in S1 and read in S2. S1 executes before S2 to produce the value that is consumed in S2.
  - Anti-dependence (Write after Read: WAR): The memory location is read in S1 and written in S2.
  - Output dependence (Write after Write: WAW): The memory location is written in both statements S1 and S2.

Find the dependencies

Flow: RAW
Anti: WAR
Output: WAW

```
S1: for (int i=1; i<10; ++i){
S2:     B[i] = temp;
S3:     A[i+1] = B[i+1];
S4:     temp = A[i]
```

Find the dependencies

Flow: RAW
Anti: WAR
Output: WAW

```
S1: for (int i=1; i<10; ++i){
S2:     B[i] = temp;
S3:     A[i+1] = B[i+1];
S4:     temp = A[i]
```

1: S3 →S2 anti (B)

Find the dependencies

Flow: RAW
Anti: WAR
Output: WAW

```
S1: for (int i=1; i<10; ++i){
S2:     B[i] = temp;
S3:     A[i+1] = B[i+1];
S4:     temp = A[i]
```

1: S3 → S2 anti (B)
2: S3 → S4 flow(A)

Find the dependencies

Flow: RAW
Anti: WAR
Output: WAW

```
S1: for (int i=1; i<10; ++i){
S2:     B[i] = temp;
S3:     A[i+1] = B[i+1];
S4:     temp = A[i]
```

1: S3 → S2 anti (B)
2: S3 → S4 flow(A)
3: S4 → S2 flow(temp)

Find the dependencies

Flow: RAW
Anti: WAR
Output: WAW

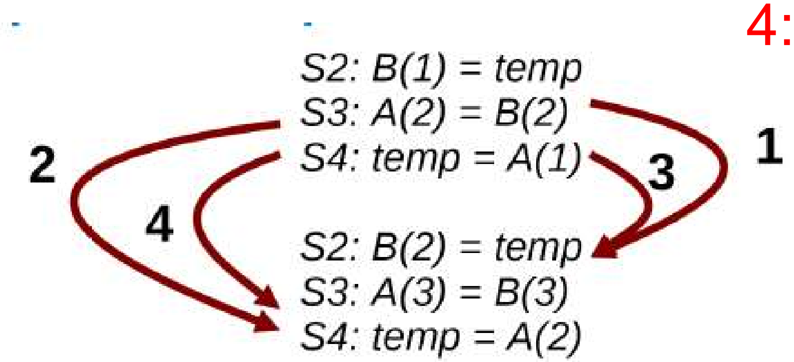
```
S1: for (int i=1; i<10; ++i){
S2:     B[i] = temp;
S3:     A[i+1] = B[i+1];
S4:     temp = A[i]
```

1: S3 → S2 anti (B)
2: S3 → S4 flow(A)
3: S4 → S2 flow(temp)
4: S4 → S4 output(temp)

Unroll the loop

Flow: RAW
Anti: WAR
Output: WAW

1: S3 → S2 anti (B)
2: S3 → S4 flow(A)
3: S4 → S2 flow(temp)
4: S4 → S4 output(temp)



Flow: RAW
Anti: WAR
Output: WAW

- Anti-dependence

```
for(i=0;i< N-1; i++)
{
   x = b[i] + c[i];
   a[i] = a[i+1] + x;
}
```

- Parallel version with dependence removed

```
#pragma omp parallel for shared (a, a2)
for(i=0; i < N-1; i++)
    a2[i] = a[i+1];
#pragma omp parallel for shared (a, a2) lastprivate(x)
for(i=0;i< N-1; i++)
{
   x = b[i] + c[i];
   a[i] = a2[i] + x;
}
```

```
for(i=1;i< m; i++)
    for(j=0;j<n;j++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

```
for(i=1;i< m; i++)
 #pragma omp parallel for
    for(j=0;j<n;j++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

Poor performance, it requires m-1 fork/join steps.

```
#pragma omp parallel for private (i)
for(j=0;j< n; j++)
  for(i=1;i<m;i++)
{
    a[i][j] = 2.0*a[i-1][j];
}
```

- Invert loop to yield better performance(?).
- With this inverting, only a single fork/join step is needed. The data dependences have not changed.
- However, this change affects the cache hit rate.

- Elimination of stored variables (do directly)

```
idx = N/2+1; isum = 0; pow2 = 1;
for(i=0;i< N/2; i++)
{
   a[i] = a[i] + a[idx];
   b[i] = isum;
   c[i] = pow2;
   idx++; isum += i; pow2 *=2;
}
```

- Parallel version

```
#pragma omp parallel for shared (a,b)
for(i=0;i< N/2; i++)
{
   a[i] = a[i] + a[i+N/2];
   b[i] = i*(i-1)/2;
   c[i] = pow(2,i);
}
```

- Remove flow dependence using loop skewing

```
for(i=1;i< N; i++)
{
   b[i] = b[i] + a[i-1];
   a[i] = a[i]+c[i];
}
```

- Parallel version

```
b[1]=b[1]+a[0];
#pragma omp parallel for shared (a,b,c)
for(i=1;i< N-1; i++)
{
   a[i] = a[i] + c[i];
   b[i+1] = b[i+1]+a[i];
}
a[N-1] = a[N-1]+c[N-1];
```

Flow: RAW
Anti: WAR
Output: WAW

- A flow dependence that can in general not be remedied is a recurrence:

```
for(i=1;i< N; i++)
{
   z[i] = z[i] + l[i]*z[i-1];
}
```

Flow: RAW
Anti: WAR
Output: WAW

C/C++ for Directive Syntax

```
#pragma omp for [clause list]
        schedule (type [,chunk])
        ordered
        private (variable list)
        firstprivate (variable list)
        lastprivate (variable list)
        shared (variable list)
        reduction (operator: variable list)
        collapse (n)
        nowait
```

Private Clause

- Direct the compiler to make one or more variables private.

```
#pragma omp parallel for private (j)
   for(i = 0; i < M; i++)
      for(j=0; j < N; j++)
          a[i][j] = min(a[i][j], a[i][k]+tmp[j]);
```

- We need every thread to work through N values of “j” for each iteration of the “i” loop.
- If we do not make “j” private, all of threads try to initialize and increment the same shared variable “j” – meaning the data race.
- The private copies of variable “j” will be accessible only inside the for loop. The values are undefined on loop entry and exit.

firstprivate Clause

```
x[0] = 1.0;
for(i=0; i < n; i++){
    for(j=1; j<4; j++)
      x[j]=g(i, x[j-1]);
     answer[i]=x[1]-x[3];
}
```

- We want each thread’s private copy of array element x[0] to start with the value that the shared variable was assigned in the master thread.

```
x[0] = 1.0;
#pragma omp parallel for private (j) firstprivate (x)
for(i=0; i < n; i++){
    for(j=1; j<4; j++)
      x[j]=g(i, x[j-1]);
     answer[i]=x[1]-x[3];
}
```

lastprivate Clause

- **Sequentially last iteration**: the iteration that occurs last when the loop is executed sequentially.
- The lastprivate clause directs the compiler to generate code at the end of the parallel for loop that copies back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration of the loop variable

```
for(i=0; i < n; i++){
    x[0] = 1.0;
    for(j=1; j<4; j++)
      x[j]= x[j-1]*(i+1);
     answer[i]=x[0]+x[1]+x[2]+x[3];
}
n_cubed = x[3];
```

- In the sequentially last iteration of the loop, x[3] gets assigned the value n^3.
- To have this value accessible outside the parallel for loop, we declare x to be a lastprivate variable

```
#pragma omp parallel for private(j) lastprivate(x)
for(i=0; i < n; i++){
    x[0] = 1.0;
    for(j=1; j<4; j++)
      x[j]= x[j-1]*(i+1);
     answer[i]=x[0]+x[1]+x[2]+x[3];
}
n_cubed = x[3];
```