Embedded computing for scientific and industrial imaging applications

Lecture 15 - loop dependencies, threadsafe, directives

Outline

- loop dependencies
- Thread-safe function
- other directives, beyond "parallel for"

Dependencies in loop

```
for (i=0; i<n; i++)
{
    z[i] = x[i] + y[i];
    w[i] = cos(z[i])
}
```

There is a data dependence between the two statements in this loop.

The value `w[i]` cannot be computed before `z[i]`.

However, this can be parallelized with a parallel for since the same thread will always execute both statements in the right order for each `i`.

Matrix-matrix multiplication

```
#pragma omp parallel for private(i,k)
for (j = 0; j < n; j++)
{
    for (i = 0; i < n; i++)
    {
        c[j * n + i] = 0.0;
        for (k = 0; k < n; k++)
        {
            c[j * n + i] = c[j * n + i] + a[j * n + k] * b[k * n + i];
        }
    }
}
```

This works since `c[j * n + i]` is only modified by thread handling row `j`.

Loop-Carried Dependencies

```
for(i=0; i < n; i++)
    x[i] = 1.0;   // initialize all elements to 1
x[0] = 5.0;
for(i=1; i < n; i++)
    x[i] = x[i-1];
```

There is a loop-carried data dependence in this loop.

The assignment for `i=2` must not be done before `i=1` or it may get the wrong value.

Loop-Carried Dependencies

Example: Solve ODE initial value problem

$$y'(t) = 2y(t),$$
$$y(0) = 1$$

with Euler's method

$$y(t + \Delta t) \approx y(t) + \Delta t y'(t) = y(t) + \Delta t(2y(t))$$

to approximate $y(t) = e^{2t}$ for $0 \leq t \leq 5$:

```
n = 5000; //number of steps to reach t = 5000
y = (double*)malloc(n * sizeof(double));
y[0] = 1.0;
dt = 0.001; //time step
for (i = 1; i < n; i++)
{
      y[i] = y[i - 1] + dt * 2.0 * y[i - 1];
}
free(y);
```

Cannot easily parallelize.

Loop-Carried Dependencies

```
y = 0.0;
for (i = 0; i < n; i++)
{
    if (i == 3)
        y = 1.0;
    x[i] = y;
}
```

There is a loop-carried data dependence in this loop.

In serial execution: only first two elements of `x` are `0.0`.

With `#pragma omp parallel for`:

later index (e.g. `i=6`) might be executed before `i=3`.

Thread-safe functions

Consider this code:

```
#pragma omp parallel do
for(i=0; i < n; i++)
    y[i] = myfcn(x[i]);
```

Does this give the same results as the serial version?

Thread-safe functions

Consider this code:

```
#pragma omp parallel do
for(i=0; i < n; i++)
    y[i] = myfcn(x[i]);
```

Does this give the same results as the serial version?

Maybe not… it depends on what the function does!

If this gives the same results regards of the order threads call for different values of i, then the function is thread safe.

Thread-safe functions

A thread-safe function:

```
void double myfcn(double x)
{
    double z;
    z = exp(x);
    return z * cos(x);
}
```

Executing this function for one value of `x` is completely independent of execution for other values of `x`.

Note that each call creates a new local value `z` on the call stack, so `z` is private to the thread executing the function.

Non-Thread-safe functions

Suppose `z`, `count` are global variables.

Then this function is not thread-safe:

```
double myfcn(double x)
{
    count = count+1; //counts times called
    z = exp(x);
    return z * cos(x) + count;
}
```

The value of `count` seen when calling y[i] = myfcn(x[i]) will depend on the order of execution of different values of `i`.

Moreover, `z` might be modified by another thread between when it is computed and when it is used.

Thread safe functions

A function can be declared thread safe if it:

- Does not alter global variables,
- Does not do I/O,
- Does not alter any input arguments.

Example:

```
void f(double x, double y)
{
    return x * x + y;
}
```

Good idea even for sequential codes: Allows some compiler optimizations.

OpenMP — beyond parallel loops

The directive `#pragma omp parallel` is used to create a number of threads that will each execute the same code...

```
#pragma omp parallel
{
    ! some code
}
```

The code will be executed `nthreads` times, once by each thread.

SPMD: Single program, multiple data

Terminology note:

SIMD: Single instruction, multiple data

refers to hardware (vector machines) that apply same arithmetic operation to a vector of values in lock-step.

OpenMP parallel with for loops

Note: This code...

```
#pragma omp parallel
for(i=0; i < 10; i++)
    printf("i = %d\n", i);
```

... is not the same as:

```
#pragma omp parallel for
for(i=0; i< 10; i++)
    printf("i = %d\n", i);
```

OpenMP parallel with for loops

Note: This code...

```
#pragma omp parallel
for(i=0; i < 10; i++)
    printf("i = %d\n", i);
```

The entire do loop (i=0,1,...,9) will be executed by each thread! With 2 threads, less than 20 lines will be printed.

... is not the same as:

```
#pragma omp parallel for
for(i=0; i< 10; i++)
    printf("i = %d\n", i);
```

which will only print 10 lines!

OpenMP parallel with for loops

```
#pragma omp parallel
for(i=0; i < 10; i++)
    printf("i = %d\n", i);
```

could also be written as:

```
#pragma omp parallel
#pragma omp for
for(i=0; i< 10; i++)
    printf("i = %d\n", i);
```

More generally, if `#pragma omp for` is inside a parallel block, then the loop is split between threads rather than done in total by each

OpenMP parallel with for loops

The #pragma omp for directive is useful for...

```
#pragma omp parallel
{
    //some code executed by every thread
     #pragma omp for
    for(i=0; i < n; i++)
     {
         //loop to be split between threads
     } //omp end for

    //more code executed by every thread
} //omp end parallel
```

Some other useful directives...

Execution of part of code by a single thread:

```
#pragma omp parallel
{
    //some code executed by every thread
    #pragma omp single
    {
        //code executed by only one thread
    }
}
```

Can also use `#pragma omp master` to force execution by master thread.

Example: Initializing or printing out a shared variable.

Some other useful directives...

barriers:

```
#pramga omp parallel
{
    //some code executed by every thread
    #pragma omp barrier
    //some code executed by every thread
}
```

Every thread will stop at barrier until all threads have reached this point.

Make sure all threads reach barrier or code will hang!

Implied barriers after some blocks, e.g. `#pragma omp for` or `#pragma omp single`.

Some other useful directives...

Sections:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp sections
    {
        #pragma omp section
            // code executed by only one thread
        #pragma omp section
            //code executed by a different thread
    } //with implied barrier
}
```

Example: Read in two large data files simultaneously.

From `$CSE6000/codes/OpenMP/Examples/demo2/demo2.c`

```
#pragma omp parallel num_threads(2) // spawn two threads
{
        #pragma omp sections //split up work between them
        {
                #pragma omp section //one thread initializes x array
                for (i = 0; i < n; i++)
                        x[i] = 1.0;

                #pragma omp section //another thread initializes x array
                for (i = 0; i < n; i++)
                        y[i] = 1.0;
        }
        #pragma omp barrier // not needed, implied at end of sections

        #pragma omp single
        printf("Done initializing x and y\n");

        #pragma omp for
        for (i = 0; i < n; i++)
        {
                z[i] = x[i] + y[i];
        }
}
```