Beyond Loops: The parallel Construct

SPMD Parallelism

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
parallel for: Restrictions

- The number of times that the loop body is executed (trip-count) must be available at runtime before the loop is executed.
- The loop body must be such that all iterations of the loop are completed.
- Also, some computations may be parallel, but not expressible easily via parallel loops.
The **parallel** Directive

• This allows the programmer to explicitly say what each thread does

```
#pragma omp parallel [clause [clause ...]]
structured block
```

• This directive tells all the threads to execute the structure block

• What is the point of having all threads execute the same code?

• Each thread has access to each id and so it can do different work depending on id
  • Ids are serial numbers from 0 to total number of thread
The **parallel** Directive:

- When a `parallel` directive is encountered, threads are spawned, which execute the code of the enclosed structured block (the parallel region).
- The number of threads can be specified just like for the `parallel for` directive.
- Each thread executes the code in the enclosed code-block.
- You can specify private variables as before (but no “`lastprivate`”).
**parallel vs. parallel for**

- Arbitrary structured blocks v/s loops
- Coarse grained v/s fine grained
- Replication v/s work division

```c
#pragma omp parallel for
for(i=0; i<10; i++) {
    printf("Hello world\n");
}
```

Output: 10 Hello world messages

```c
#pragma omp parallel private(i)
{
    for(i=0; i<10; i++) {
        printf("Hello world\n");
    }
}
```

Output: 10*T Hello world messages where T = number of threads
A Simple Example with `parallel`

```c
for( i=0; i<N; i++ ) {
    A[i] = x*B[i];
}

#pragma omp parallel
{
    int id = omp_get_thread_num();
    int p = omp_get_num_threads();
    int mystart = (N*id)/p;
    int myend = (N*(id+1))/p;
    if(myend>N) myend = N;
    for( i=mystart; i<myend; i++ ) {
        A[i] = x*B[i];
    }
}
```

Each thread calculates the range of iterations it’s going to work on using variables mystart and myend after finding out
- The total number of threads, p, in its team and
- Its own rank, id, within the team
parallel vs. parallel for

• The code with parallel is much longer and complicated
• But it gives you, the programmer, much greater control
  • And the cases where you will use it are typically much longer pieces of code
• To use the full power of this construct you will also need to use other constructs, particularly synchronization constructs
double x, y;
int i, j, m, n, maxiter, depth[300][200],
    dith_depth[300][200], mandel_val();
n = 300; m = 200;
maxiter = 200;

#pragma omp parallel for private(j, x, y)
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
        x = i / (double) m;
        y = j / (double) n;
        depth[j][i] = mandel_val(x, y, maxiter);
    }

#pragma omp parallel for private(j)
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        dith_depth[j][i] = 0.5*depth[j][i]
                         + 0.25*(depth[j-1][i] + depth[j+1][i])
parallel Construct: Example 2 Rewritten

• Rewrite with parallel

```c
double x, y;
int i, j, m, n, maxiter, depth[300][200],
    dith_depth[300][200], mandel_val();
n = 300; m = 200;
maxiter = 200;
int id, myfirst, mylast;
omp_set_num_threads(16);
#pragma omp parallel private(id, myfirst, mylast, i, j, x, y)
{
    id = omp_get_thread_num();
    myfirst = first_idx(id, m, 16);
    mylast = last_idx(id, m, 16);
    for (i = myfirst; i <= mylast; i++)
    {
        for (j = 1; j <= n; j++)
        {  
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    }
    for (i = myfirst; i <= mylast; i++)
    {
        for (j = 1; j <= n; j++)
        {  
            dith_depth[j][i] = 0.5*depth[j][i]
                + 0.25*(depth[j-1][i] + depth[j+1][i]);
        }
    }
}
```

Threads are forked once and then work independently

L.V.Kale