Programming Parallel Computers

Chapter 3: Multithreading with OpenMP

Examples of useful OpenMP constructions

Parallel for loops often follow this pattern:

```
global_initialization();
#pragma omp parallel
{
    local_initialization();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        do_some_work(i);
    }
    #pragma omp critical
    {
        update_global_data();
    }
}
report_result();
```

Here is a more concrete example. This will calculate the sum 0 + 1 + … + 9 = 45 in parallel:

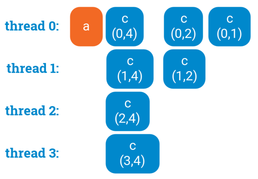
```
// shared variable
int sum_shared = 0;
#pragma omp parallel
{
    // private variables (one for each thread)
    int sum_local = 0;
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        sum_local += i;
    }
    #pragma omp critical
    {
        sum_shared += sum_local;
    }
}
print(sum_shared);
```

Note that there is no synchronization inside the loop. We use a critical section only once, outside the loop. In this application we could also use an atomic operation, but this is not a performance-critical part here.

Divide and conquer — bottom up

Here is one approach that we can use to implement "bottom up" algorithms in which we first process small items with a large number of threads and then combine the solutions iteratively.

```
a();
int p = omp_get_max_threads();
while (p > 1) {
    #pragma omp parallel num_threads(p)
    {
        int i = omp_get_thread_num();
        c(i, p);
    }
    p = (p + 1) / 2;
}
c(0, 1);
```



Divide and conquer — top down

Here is one approach that we can use to implement "top down" algorithms in which we first split large items in two parts and then handle each half recursively.

```
static void r(int x) {
    c(x);
    if (x > 0) {
        #pragma omp task
        r(x - 1);
        #pragma omp task
        r(x - 1);
    }
}

static void recurse() {
    a();
    #pragma omp parallel
    #pragma omp single
    {
        r(3);
    }
    z();
}
```