Look into tasking if necessary

5.10 Tasking

While many problems are straightforward to parallelize with OpenMP, they generally have a fixed or predetermined number of parallel blocks and loop iterations to schedule across participating threads. When this is not the case, the constructs we've seen

thus far make it difficult (or even impossible) to effectively parallelize the problem at hand. Consider, for instance, parallelizing a web server; HTTP requests may arrive at irregular times, and the server itself should ideally be able to respond to a potentially infinite number of requests. This is easy to conceptualize using a **while** loop, but recall our discussion in Section 5.5.1: **while** and **do–while** loops cannot be parallelized with OpenMP, nor can **for** loops that have an unbounded number of iterations. This poses potential issues for dynamic problems, including recursive algorithms, such as graph traversals, or producer-consumer style programs like web servers. To address these issues, OpenMP 3.0 introduced *Tasking* functionality [47]. Tasking has been successfully applied to a number of problems that were previously difficult to parallelize with OpenMP [1].

Tasking allows developers to specify independent units of computation with the `task` directive:

```
#pragma omp task
```

When a thread reaches a block of code with this directive, a new task is generated by the OpenMP run-time that will be scheduled for execution. It is important to note that the task will not necessarily be executed immediately, since there may be other tasks already pending execution. Task blocks behave similarly to a standard `parallel` region, but can launch an arbitrary number of tasks instead of only `num_threads`. In fact, tasks must be launched from within a `parallel` region but generally by only one of the threads in the team. Therefore a majority of programs that use Tasking functionality will contain an outer region that looks somewhat like:

```
#   pragma omp parallel
#   pragma omp single
    {
        ...
#       pragma omp task
        ...
    }
```

where the `parallel` directive creates a team of threads and the `single` directive instructs the runtime to only launch tasks from a single thread. If the `single` directive is omitted, subsequent `task` instances will be launched multiple times, one for each thread in the team.

To demonstrate OpenMP tasking functionality, recall our discussion on parallelizing the calculation of the first *n* Fibonacci numbers in Section 5.5.2. Due to the loop-carried dependence, results were unpredictable and, more importantly, often incorrect. However, we *can* parallelize this algorithm with the `task` directive. First, let's take a look at a recursive serial implementation that stores the sequence in a global array called `fibs`:

```
int fib(int n) {
    int i = 0;
    int j = 0;
```

```
    if (n <= 1) {
        // fibs is a global variable
        // It needs storage for n+1 ints
        fibs[n] = n;
        return n;
    }

    i = fib(n - 1);
    j = fib(n - 2);
    fibs[n] = i + j;
    return fibs[n];
}
```

This chain of recursive calls will be time-consuming, so let's execute each as a separate task that can run in parallel. We can do this by adding a `parallel` and a `single` directive before the initial (nonrecursive) call that starts `fib`, and adding **#pragma** omp task before each of the two recursive calls in `fib`. However, after we make this change, the results are incorrect—more specifically, except for `fib`[1], the sequence is all zeroes. This is because the default data scope for variables in tasks is private. So after completing each of the tasks

```
#  pragma omp task
i = fib(n − 1);
#  pragma omp task
j = fib(n − 2);
```

the results in `i` and `j` are lost: `i` and `j` retain their values from the initializations

```
int i = 0;
int j = 0;
```

at the beginning of the function. In other words, the memory locations that are assigned the results of `fib`(`n`−1) and `fib`(`n`−2) are not the same as the memory locations declared at the beginning of the function. So the values that are used to update `fibs`[`n`] are the zeroes assigned at the beginning of the function.

We can adjust the scope of `i` and `j` by declaring the variables to be `shared` in the tasks that execute the recursive call. However executing the program now will produce unpredictable results similar to our original attempt at parallelization. The problem here is that the order in which the various tasks execute isn't specified. In other words, our recursive function calls, `fib`(`n` − 1) and `fib`(`n` − 2) will be run eventually, but the thread executing the task that makes the recursive calls can continue to run and simply **return** the current value of `fibs`[`n`] early. We need to force this task to wait for its subtasks to complete with the `taskwait` directive, which operates as a `barrier` for tasks. We've put this all together in Program 5.6.

```c
int fib(int n) {
    int i = 0;
    int j = 0;

    if (n <= 1) {
        fibs[n] = n;
        return n;
    }

#   pragma omp task shared(i)
    i = fib(n - 1);

#   pragma omp task shared(j)
    j = fib(n - 2);

#   pragma omp taskwait
    fibs[n] = i + j;
    return fibs[n];
}
```

Program 5.6: Computing the Fibonacci numbers using OpenMP tasks.

Our parallel Fibonacci program will now produce the correct results, but you may notice significant slowdowns with larger values of *n*; in fact, there is a good chance that the serial version of the program executes much faster! To gain an intuition as to why this occurs, recall our discussion of the overhead associated with forking and joining threads. Similarly, each task requires its own data environment to be generated upon creation, which takes time. There are a few options we can use to help reduce task creation overhead. The first option is to only create tasks in situations where *n* is large enough. We can do this with the **if** directive:

```c
#pragma omp task shared(i) if(n > 20)
```

which in this case will restrict task creation to only occur when *n* is larger than 20 (chosen arbitrarily in this case based on some experimentation). Reviewing `fib` again, we can see that there will be a task executing `fib` itself, another executing `fib(n` $-$ 1), and a third executing `fib(n` $-$ 2) for each recursive call. This is inefficient, because the parent task executing `fib` only launches two subtasks and then simply waits for their results. We can eliminate a task by having the parent thread perform one of the recursive calls to `fib` instead before doing the final calculation after the `taskwait` directive. On our 64-core testbed, these two changes halved the execution time of the program with $n = 35$.

While using the Tasking API requires a bit more planning and care to use—especially with data scoping and limiting runaway task creation—it allows a much broader set of problems to be parallelized by OpenMP.