Lab 5

Parallel sorting with OpenMP

In this lab, you’ll practice using OpenMP to speed up a sorting algorithm. As mentioned in class, recursive sorting algorithms have a natural parallel implementation—hand off the recursive work to a new group of threads. In this lab, we will focus on quicksort. (There are other (and better) parallel sorting algorithms, but this is a natural algorithm to begin with.)

Begin by copying the file serial_sort.c from the course directory:

    cp /home/courses/cs226/serial_sort.c .

As suggested by the name, this file has a serial implementation of quicksort. Recall that this algorithm selects a pivot element for the array. Then it partitions the array around this pivot so that all elements smaller than the pivot occur before it in the array and all larger elements occur after it. Finally, it recursively sorts each half. Look through the code to verify that it follows this basic idea.

Since we’re interested in determining a speedup, we need a baseline running time. Begin by compiling and running the given code without modifications, using the command line argument 10000000 (10 million). Run it with time several times and record the results:

    gcc -Wall -std=gnu99 -o serial_sort serial_sort.c
    time ./serial_sort 10000000

Next parallelize the verification loop at the end by adding

    #pragma omp parallel for

immediately before the last for loop. Recompile with the -fopenmp flag and observe the difference in running time, which is measurable but relatively modest and it’s possible that the program is even slower after this step. (Normally, you want to try parallelizing the most computationally-intensive part of the code. We started here just because it was easy.)

Now we look at the main part of the work, which is the quicksort routine. The work in this function is performed by partition and the two recursive calls. The partition function would be tricky to parallelize since most of its work is a for loop where each iteration depends on the previous one (via the value of storeIndex). Instead, let’s make each of the recursive calls use its own thread. To do this, we will use the sections work-sharing construct. Put the two recursive calls into a block (statements surrounded by braces) preceeded by the line

    #pragma omp parallel sections

and preceed each recursive call with the line

    #pragma omp section

The first of these creates a team of threads and the section lines specify that exactly one thread should execute each of the recursive calls.
When you run this, the results will be disappointing. The code is actually significantly slower than our original serial implementation. There are two factors behind this. The first is that the overhead is massive because we’ve added work to each recursive call, even those too small to usefully exploit parallelism. The second is that we’re not actually getting much parallelism out of this; by default, only the first call will make more than one thread. Recursive calls below that each create a “team” consisting of a single thread, giving us all the overhead and none of the benefit. (Oops!)

The overhead is the more important of these factors, but the lower number of threads created is easier to fix so we’ll start there. Add the include file omp.h to the top of your file and then add the line

```
omp_set_nested(1);
```

to main before it calls quicksort. This has the effect of enabling “nested parallelism”, where one parallel region can be inside another and both will get multiple threads. (The “1” in this case denotes “true” rather than the value one per se.)

Once again, this change actually makes the program worse. It ends sooner now, but does so with errors about not being able to create threads. Basically the operating system choked on the massive number of threads we tried to create; remember that there are MANY recursive calls.

This brings us back to the problem of reducing the number of times we enter parallel regions. Our original motivation was overhead reduction and now we’ve added the need to reduce the number of threads created. To do this, we will select between different ways of making the recursive calls. By using an if statement, we can select between recursive calls inside a parallel region and “naked” recursive calls as in the original code. Try the following two ways of deciding between these versions:

- Keep track of the recursion depth (just add an argument to the function and increment it at each recursive call) and only create new threads the first couple of times the function recurses. This bounds the number of threads created and also uses them near the top of the recursion tree, where each call is responsible for a large amount of work.

- Have the function look at the size of the array that it is sorted and only create new threads when the array is big enough. This ensures that threads are created only when there is enough work that the benefit outweighs the overhead.

It is not obvious what depth or array size to use as thresholds for these schemes. Implement each of them and try different thresholds to determine what seems to work best.

One additional refinement is to add num_threads(2) to the end of the sections pragma. This ensures that the system will only create 2 threads in response to this pragma. The default is likely higher, but there are only two sections to run so additional threads are wasted. This should give another small improvement. (Actually, it will also change the optimal threshold you determined above.)

If you have additional time, either try to make further improvements or work to parallelize a different algorithm. Some ideas for further improving quicksort are to identify additional pivot elements or to restructure the partition method to allow it to be parallelized as well. Mergesort is a natural alternative algorithm to try. It also has a natural generalization where the array is broken into more than 2 pieces (though merging gets more complicated...).