Programming Assignment 3 – Results and Summary

1) Gathering run-times: pthreads vs. OpenMP

In order to make run-time comparisons between pthreads and OpenMP, four versions of the Adaptive Mesh Refinement problem were created: disposable pthreads, persistent pthreads (both from Programming Assignment 2), disposable OpenMP, and persistent OpenMP. The terms "disposable" and "persistent" are used more loosely with the OpenMP programs because there isn't the same level of control over thread creation and joining as with pthreads. Still, these general methods of disposable and persistent thread creation are achieved by putting the parallelizing pragmas inside or outside of the convergence loop, respectively.

In all run-time tests, the values `AFFECT_RATE=0.05` and `EPSILON=0.05` were used because these made the disposable pthreads program, the slowest of the 4 versions, take 3-5 minutes to reach convergence with 2 threads on `testgrid_400_12206`. Once calibrated with by these values, sufficient timing results were gathered by running all four programs four times each, specifying 2, 8, 16, and 32 threads as a command line argument. Additionally, a summary of the timing results and the requested versus actual thread counts for OpenMP programs are in the table at the end of this report. While many metrics were used to calculate the timing of each program the table highlights the "wall-clock" real time given by the Linux time command and the clock() function. These metrics were chosen because they show both sides of the story; the wall-clock gives a good sense of how much actual time passes while the loop is converging, while the clock() function sums the total processor time for each thread and is noticeably affected by more multithreading.

2) Discussion Questions

The computational work-load of this sample program can be characterized by looking at its two main phases. In the first phase, the next values of all of the temperature DSVs are calculated. This phase has a very high computational work-load because the computations require a lot of floating point math and many accesses to dynamically allocated memory, but is highly parallelizable

because the new temperatures can be calculated in any order. Each box's next value is a function of only the old temperatures of itself and its neighbors, and has no dependence on any updated temperatures or intermediate state. So, with more multithreading and more active computing cores, the high work-load of this phase can be achieved in much less time than if done serially. The second phase of this program is where the new grid of temperatures is copied into the old grid, the min and max temperatures are found, and the convergence condition is checked. These steps each aren't too computationally expensive, but they are only done by one thread so all other threads are waiting at a barrier while this phase is occurring. So, in this phase, the computational workload is lower, but the lack of easy multithreading reduces the amount the run-time can be lowered.

As a result of these run-time experiments, it seems that parallelization with OpenMP is slightly better than with pthreads. This is especially true for the disposable versions of the programs, where the OpenMP version's average wall-clock and clock() times were 44 and 147.7 seconds faster than the pthread version's, respectively. Since the disposable version creates many more threads than the persistent version, this indicates that the overhead for creating batches of threads with OpenMP is less than with pthreads. Meanwhile, the average times of the persistent versions of the programs were too close to determine a decisive winner. OpenMP was 14 seconds faster for wall-clock time, while pthreads were 20.1 seconds faster with the clock() function. This further leads me to believe that the main performance difference between pthreads and OpenMP has to do with the overhead of thread creation, because these persistent programs ran more similarly.

Furthermore, it was far easier to modify my serial program for OpenMP than it was for pthreads. The disposable version of OpenMP only required a couple lines of added code to lay out the parallel pragma and retrieve the actual number of threads. Meanwhile, making a disposable pthreads application took refactoring code into a thread-safe function and thinking harder about which variables needed to be thread-local and global. The same was true when implementing the persistent versions of these programs. The OpenMP version was simple and mostly only required a few pragmas to set up barriers and the portion of the code that would only be run by a single thread.

There was none of the refactoring and object creating/destroying that the pthreads versions of the code required.

Consequently, I would be much more likely to choose OpenMP if I needed to parallelize a similar application in the future. In the worst case, OpenMP is only marginally slower than pthreads. Yet, in many cases it is faster and a lot easier to implement. However, under some circumstances, it might be better to use pthreads than OpenMP. The AMR algorithm used in this experiment is fairly simple and required few pragmas. There could be more complex problems that would require declaring more pragmas in just the right way and in just the right place that there is no longer any programming benefit to OpenMP. Similarly, there may be kinds of problems where threads need to interact in very specific way and it would be more reliable, or even faster to utilize the full control over thread coordination given by pthreads.

The main surprise encountered this exercise had to do with how effective OpenMP is for parallelizing code! All of the available pragmas are very easy to use and seem catered to exactly what programmers need to do to parallelize a serial application. On the other hand, the only other surprise was that many omp pragmas, such as atomic, restrict the code that can go inside them to one or two expressions and a limited number of operators. This makes sense because OpenMP has to be able to implement these structures efficiently, but I was not fully aware of the consequences of this going into the experiment. As a result, I spent a lot of time trying to make a code block atomic that couldn't and didn't need to be synchronized like that.

Table 1: Run-time and thread count comparisons for the four programs at `AFFECT_RATE=0.05` and `EPSILON=0.05`.

Req. # threads	**Disposable pthreads**		**Persistent pthreads**		**Disposable OpenMP**			**Persistent OpenMP**		
	wall-clock	`clock()`	wall-clock	`clock()`	Actual # threads	wall-clock	`clock()`	Actual # threads	wall-clock	`clock()`
2	3m43s	300.2s	2m38s	261.0s	2	2m59s	335.4s	2	3m01s	303.9s
8	2m54s	353.1s	1m53s	289.6s	8	3m02s	283.2s	8	2m29s	283.0s
16	3m45s	471.0s	3m16s	302.9s	16	4m01s	274.5s	16	2m23s	312.6s
32	6m38s	669.5s	4m09s	323.0s	32	4m02s	308.2s	32	2m46	357.2s
Avg.	4m15s	448.0s	2m59s	294.1s	-	3m31s	300.3s	-	2m40s	314.2s