Nate Stewart
Dr. Jeffrey Jones
03-11-16
Lab 2 – AMR Dissipation using PThreads

Environment:

All test results presented in this report were run on the Oakley cluster using 1 node, 12 ppn, 1GB ram, and a walltime of 59 minutes. The epsilon and affect rate parameters were both set to 0.1, identical to the setup of Lab 1. The program was written in pure C and was compiled with an optimization level of 3. Unfortunately, since the program was written in pure C, the only timing methods used were time, clock and unix time (no chrono).

Results:

Detailed runtime information is available in the table below, Figure 1. For brevity, the iterations, Max DSV, Min DSV, and data file values were removed from Figure 1. All runs were done with the testgrid_400_12206 data file. For epsilon and affect rate parameters of value 0.1, the number of iterations was 75197 with Max DSV of 0.086671 and Min DSV of 0.078004.

Program	Threads	Clock	Time	Real Time
Serial	1	2141000	22	21.465
Disposable	2	65650000	40	40.383
Disposable	4	69450000	26	25.886
Disposable	8	89990000	22	22.078
Disposable	16	113940000	26	26.019
Disposable	24	138540000	36	35.834
Disposable	32	181400000	61	60.923
Persistent	2	89700000	46	45.671
Persistent	4	53020000	17	17.793
Persistent	8	184850000	33	33.130
Persistent	16	248470000	37	37.608
Persistent	24	185610000	36	35.603
Persistent	32	149180000	41	40.801

Figure 1: Completion metrics for differing number of threads

From the results, it can be seen that running the program in parallel is almost always worse than running it sequentially. The one case where this is false happens using four persistent threads. There is also one edge case where variance could tilt the results either way and that happens with 8 disposable threads. This can be seen more clearly in Figure 2, below, where the real time execution values are plotted and compared to the serial program. This graph clearly shows that using a large number of threads has a detrimental effect on execution time, independent of if the threads are disposable or persistent. Also, the graph illustrates how rapidly disposable threads deteriorate in execution time compared to the

persistent threads as the number of threads increases.

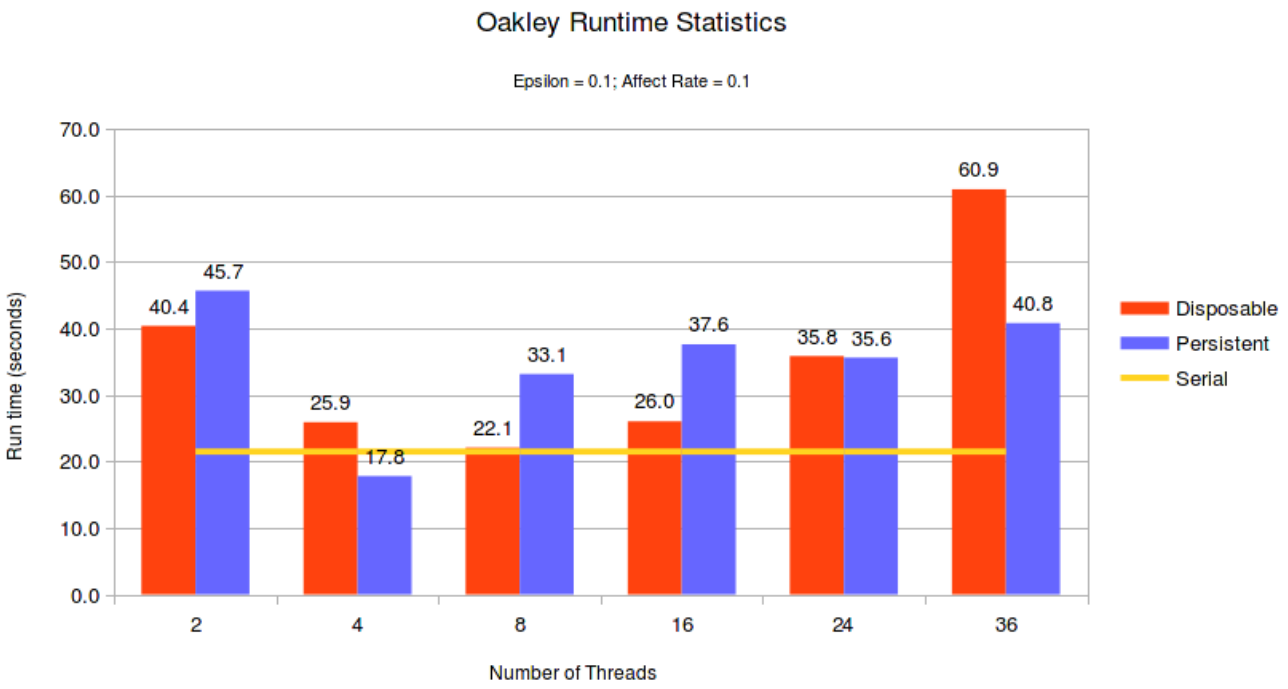

Figure 2: Graphical representation of real time execution values for serial and threaded computations

Conclusion:

The program performed better sequentially than it did in parallel in all but one case, as explained in the results. The most effective thread count for disposable threads was 8; the most effective for persistent threads was 4. Unfortunately, in my case, the question of which parallel program is more effective is difficult to answer. For low numbers of threads, it seems the key is to use disposable threads. But as the number of threads grows, it seems best to use persistent threads. This surprised me because I was under the impression that creating and destroying each thread for every iteration would have a much larger impact on the performance of the program. However, it seems that blocking on barriers inside the critical section was much more time consuming than the repetitive create/destroy cycle. I hypothesize that as the affect rate and epsilon values decrease (thus increasing the execution time of the program), the efficiency of the persistent threads will outweigh the impact of blocking on the barriers. This leads me to believe that the longer the program takes to complete, the better suited persistent threads are for overall execution time. This can be seen in Figure 2, above; as the number of threads increases (past 4 threads), the execution time of the disposable threads increases much more rapidly than the persistent threads.

The unix time(1) and time calls resulted in plausible data for every execution. The clock call, on the other hand, resulted in very weird values for persistent threads (8 and 16 threads had larger values than 24 and 36 threads) that were very inconsistent. This ran counter to my prediction that for parallel programs, clock would yield the best timing result. Now, I feel unix time(1) produces the most consistent timing results for both sequential and parallel programs.