Nate Stewart
Dr. Jeffrey Jones
03-28-16
Lab 3 – AMR Dissipation using OpenMP

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 parameter was set to 0.01 and the affect rate parameter was set to 0.05. 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 run time 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. The number of iterations was 415,949 with Max DSV of 0.084242 and Min DSV of 0.083399.

Program	Requested Threads	Actual Threads	Clock	Time	Real Time
Serial	1	1	116280000	116	116.343
Disposable Pthreads	2	2	361900000	223	223.115
Disposable Pthreads	4	4	382720000	143	142.488
Disposable Pthreads	8	8	492460000	121	121.005
Disposable Pthreads	16	16	640540000	144	143.842
Disposable Pthreads	24	24	760950000	199	199.404
Disposable Pthreads	32	32	987380000	338	338.004
Persistent Pthreads	2	2	478100000	245	245.205
Persistent Pthreads	4	4	326930000	100	99.250
Persistent Pthreads	8	8	1021780000	183	183.694
Persistent Pthreads	16	16	1367130000	206	205.989
Persistent Pthreads	24	24	1028290000	194	194.009
Persistent Pthreads	32	32	828410000	221	220.626
Disposable OpenMP	2	2	259210000	130	129.683
Disposable OpenMP	4	4	331440000	83	82.908
Disposable OpenMP	8	8	493940000	61	61.789
Disposable OpenMP	16	16	541590000	125	124.673
Disposable OpenMP	24	24	646160000	139	138.460
Disposable OpenMP	32	32	778980000	155	155.638
Persistent OpenMP	2	2	180420000	90	90.284
Persistent OpenMP	4	4	232770000	59	58.238
Persistent OpenMP	8	8	372410000	46	46.588
Persistent OpenMP	16	16	560460000	114	113.661
Persistent OpenMP	24	24	712080000	141	140.592
Persistent OpenMP	32	32	980390000	171	171.870

Figure 1: Completion metrics for differing number of threads

From the results, it can be seen that running the program using OpenMP is always better than using Pthreads, and almost always better than using the serial program. There is only one case where Pthreads perform better than the sequential program, and this occurs with 4 persistent Pthreads. Using 36 disposable Pthreads yielded the longest runtime of 5 minutes, 38 seconds. Using 8 persistent OpenMP threads yielded the shortest runtime of 46.6 seconds. This can be seen more clearly in Figure 2, below, where the real time execution values are plotted and compared to the serial program.

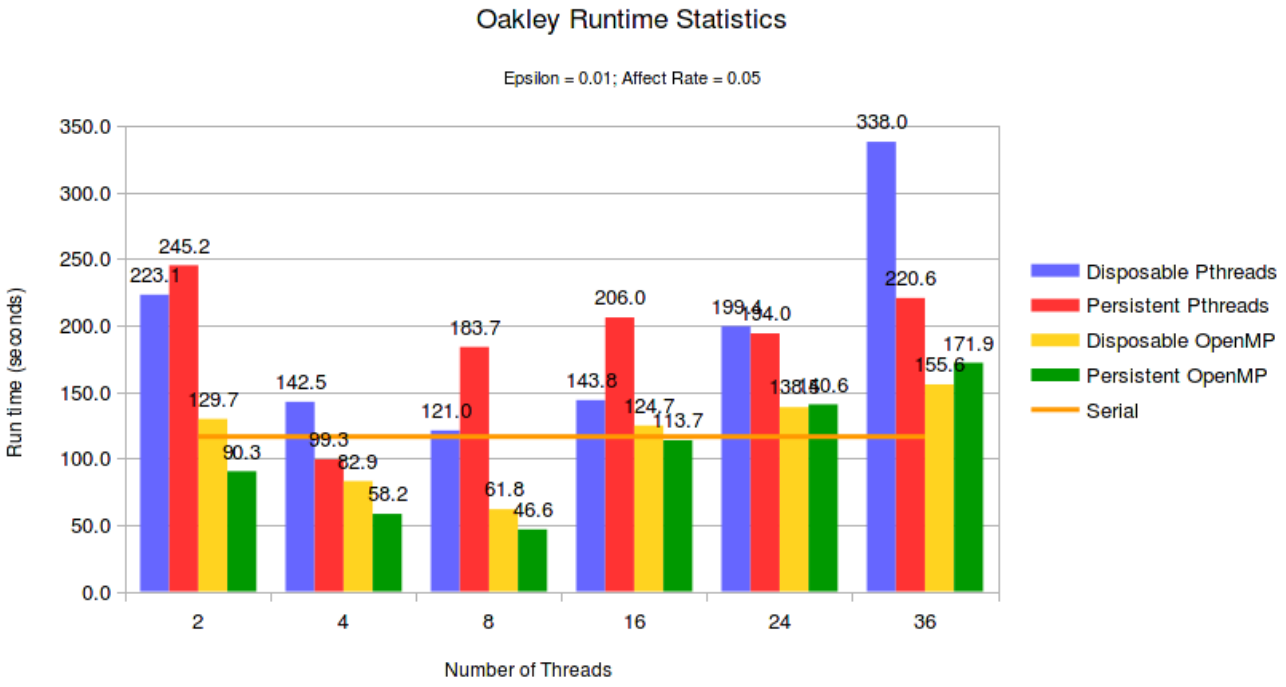

Figure 2: Graphical representation of real time execution values for serial and threaded computations

Conclusion:

Across the board, OpenMP performed better than Pthreads, with Persistent OpenMP threads performing better with fewer threads, and disposable OpenMP threads performing better with more threads. Comparing this to the Pthread implementation, OpenMP was much easier to implement, needing only about 3 additional lines of code added to the serial implementation for the disposable version and 5 for the persistent version. The Pthread implementation required several methods and variable scopes to be rearranged/added for it to work properly. Although OpenMP ran much faster and was much easier to implement, I prefer using Pthreads because of how much control you are given. For a similar application, I would undoubtedly use OpenMP because of how well it worked with this problem. But for a drastically different problem with a more complex critical section and less emphasis on run time, I would choose to use Pthreads for the higher level of control it provides. Also, I understand that OpenMP has more complex functionality such as mutex locks, but in my personal experience, if I were to implement a producer/consumer type application, I would

prefer to use something I am familiar with and confident in using.

In implementing Pthreads, it took nearly 10 hours to complete both the disposable and persistent versions and another 2 hours to draft and write the report. OpenMP took maybe 2 hours for both versions (and it only took so long because I got hung up on the syntax). This surpised me more than anything because I had no idea it was that easy to parallelize a program. Another thing that surprised me was how OpenMP granted my thread request every single time the program ran. After the discussion in class, I was expecting OpenMP to constantly switch how many threads were running based on how many I requested; that did not happen once, so looking at Figure 1, above, requested threads and actual threads are identical.