Deqing Qu
qud@oregonstate.edu

1. Tell what machine you ran this on

   I ran the code on my own Macbook pro (2.3GHz dual-core Intel Core i5).

2. Create a table with your results

NUMT \ NUM	0	1	2	3	4	5
1	344.71	338.59	325.93	344.31	325.27	318.01
2	337.51	365.60	409.06	682.43	629.65	661.28
4	333.73	372.41	413.05	593.78	545.88	534.73
NUMT \ NUM	6	7	8	9	10	11
1	331.39	339.19	330.74	328.77	324.89	331.76
2	632.72	702.82	645.59	664.45	625.25	664.55
4	536.05	622.66	533.27	571.23	539.80	617.42
NUMT \ NUM	12	13	14	15	16	
1	330.33	321.31	325.50	342.43	328.49	
2	641.72	651.25	629.84	690.40	643.46	
4	586.57	907.19	949.64	1065.53	982.19	

Table 1 Fix#1 Results

NUMT	
1	348.98
2	690.90
4	1055.57

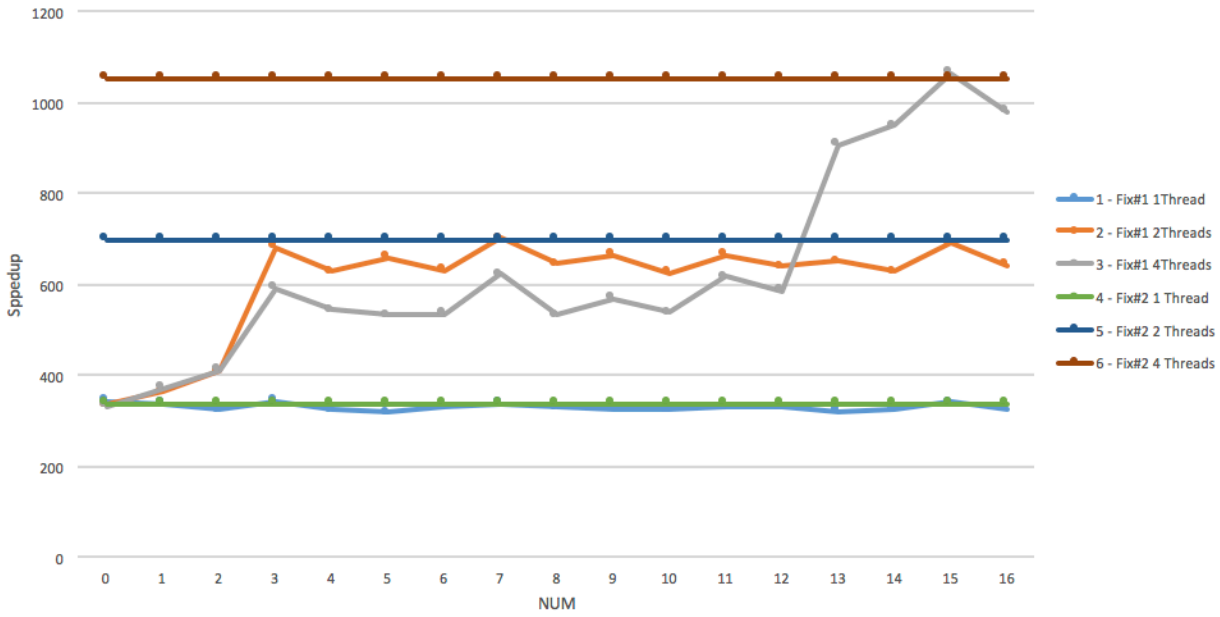
Table 2 Fix#2 Results

Table 1 shows the results using the Fix #1 solution (padding) with 1, 2 and 4 threads. Table 2 shows the results using the Fix #2 solution (private variable) with 1, 2 and 4 threads.

3. Draw a graph.

   The X axis is the number of pads and the Y axis is the performance in units of "MegaTimes Compared Per Second". Curve 1-3 uses the fix #1 solution (padding) with 1, 2, and 4 threads. Curve 4-6 uses the fix #2 solution (private value) with 1, 2, and 4 threads.

Deqing Qu
qud@oregonstate.edu



4. What patterns are you seeing in the performance?
   (1) The performance using the Fix#1 solution with 0 padding is much worse than using the Fix#2 solution, no matter how many threads are used.
   (2) The performance using the Fix#1 solution with 2 threads starts to increase when the number of paddings is 3 and is very close to the performance using the Fix#2 solution with 2 threads.
   (3) The performance using the Fix#1 solution with 4 threads starts to increase when the number of paddings is 3 and fluctuates in a certain range. The performance starts to increase again when using 13 paddings and reaches the maximum with 15 paddings. The top performance using the Fix#1 solution with 4 threads is very close to the performance using the Fix#2 with 4 threads.
   (4) The performance using the Fix#1 solution with 2 and 4 threads drops when using 16 paddings.
5. Why do you think it is behaving this way?
   (1) It is caused by the false sharing. In the program, each thread will read and write the Array data. Though more threads are used, the Array data share the same cache line. The caching protocol may force the cache line to be reloaded despite it is not necessary.
   (2) According to the course material, the performance using the Fix#1 solution with 2 threads should increase when using 7 paddings. The

reason is the data will be allocated in two different cache lines and starts from the cache boundary. But in my results, it starts to increase when using 3 paddings. I guess it may be related to the different hardware architectures or the default optimization by the gcc7 compiler (I have added –O0 parameter when compiling, but I am not sure whether there is any default optimization by gcc7).

(3) The performance using the Fix#1 solution with 4 threads should increase when using 7 paddings and increase again when using 15 paddings. When using 7 paddings, the reason is as same as with 2 threads. When using 15 paddings, the data will be allocated in four different cache lines and start on the cache line boundary. The false sharing will not happen in the ’15 paddings’ situation so the performance will reach the maximum.

(4) When the number of paddings is 16, the data will not be allocated on the cache line boundary and it will cost overheads for offset.