7 May 2018
Project #3
OSU CS475 Sp2018
Joshua L. Rogers
rogerjos@oregonstate.edu

The Impact of False Sharing on Parallel Performance

I ran the code on flip3 because it had the lowest load and rabbit doesn't support compilation with the `-std=c11` flag. I incorporated the options into the code as macros, then set the macros within the loops of a bash script that compiled and ran the resulting executable with output redirected to a file. The C code produced output formatted as comma-separated values, and this was imported into an Excel workbook with preformatted graph.

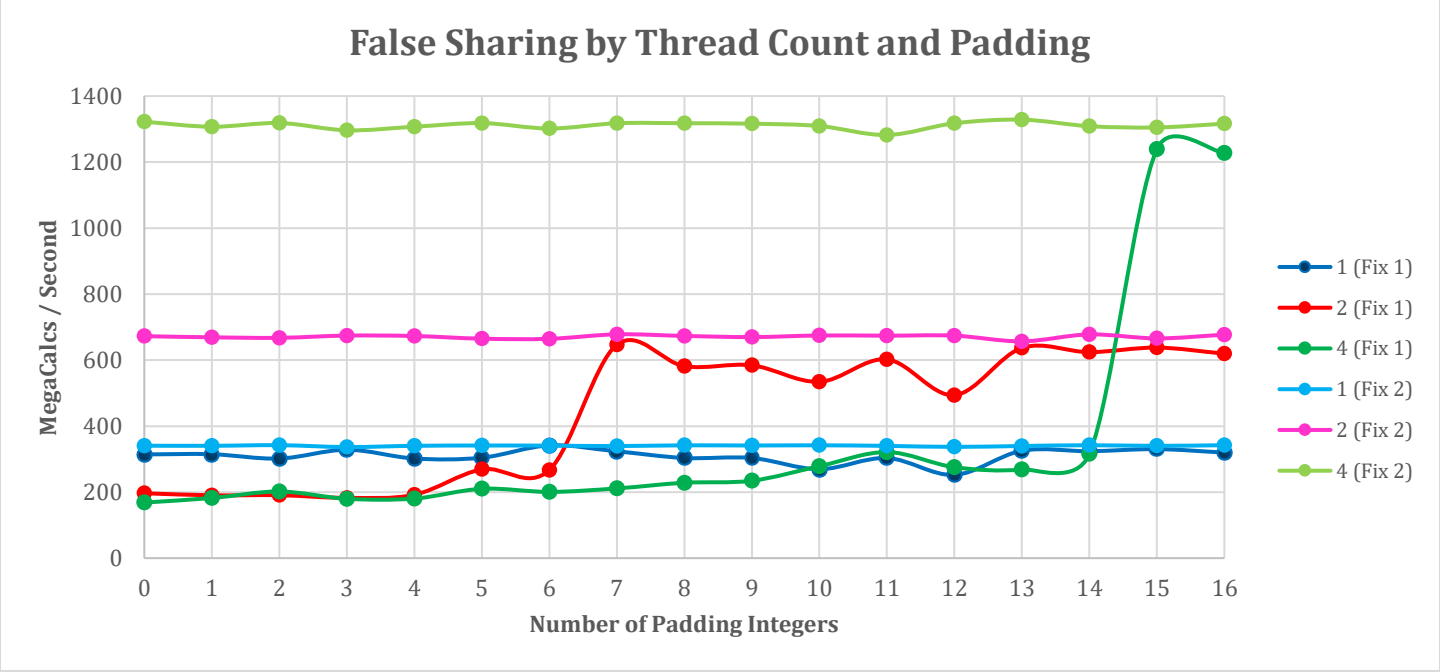
Initially my results were not consistent, with performance changes at unpredictable padding counts. Upon reflection it was clear that this was due to memory allocation. The bash script recompiles and runs the C code at each padding count, so there was no guarantee that the memory allocated for the array of structs would begin at the same cache line offset between any two iterations. In order to fix this I first implemented the aligned allocation solution in the notes, but then then discovered that C11 contains the function `aligned_alloc()` which accomplishes the same task more succinctly. Using this function to allocate memory resulted in more consistent performance, indicating that the cause of the inconsistency was non-aligned allocation. Some variance did remain, as I ran my code on flip3 and was unable to find a time when I was the only student using the server.

The results in tabular form:

Thread Count	Padding Integer Count: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1 (Fix 1)	314.25354	314.912079	301.748444	328.369141	301.884277	304.305664	341.590759	323.643982	303.500305	303.361969	269.529297	303.400665	252.264832	325.363556	323.951569	330.289978	319.847687
2 (Fix 1)	196.696381	190.322769	190.68721	182.68924	192.020889	269.141144	267.014557	646.437622	581.53009	584.267456	534.173584	602.249939	493.689362	636.879028	624.149658	637.904724	619.636047
4 (Fix 1)	168.394424	182.200195	201.799149	179.828735	180.411102	210.062698	200.658997	211.241867	228.394073	234.524796	278.535858	321.286957	275.574921	268.779114	315.847656	1238.890137	1227.600952
1 (Fix 2)	340.643372	340.388977	342.235229	336.550781	340.344543	341.280212	340.867157	339.57959	341.860443	341.268402	341.897644	340.125153	337.042938	339.841644	342.088593	340.192383	342.210114
2 (Fix 2)	672.543701	669.030334	667.481262	674.289185	672.981689	664.916748	664.49646	678.095093	673.210632	669.732971	674.632263	673.994873	674.108582	656.916077	677.858887	665.78186	676.969604
4 (Fix 2)	1322.139771	1307.085815	1318.258057	1296.34082	1307.050903	1318.016602	1301.729492	1317.652466	1317.667847	1316.227905	1309.272217	1282.074585	1317.540771	1328.462891	1308.650879	1304.914185	1316.266968

Note: All Results in MegaOperations per Second

The results in graph form:



Although the project video said we didn't have to use padding for fix 2 I chose to do so to demonstrate that the performance of fix 2 really is not impacted by the number of padding integers. Adding more threads to fix 2 improves performance, but no other interesting behavior was noted.

The padding integer count had a significant impact on fix 1 performance. Note that for one thread the performance was similar to that of fix 2 with one thread. Obviously no false sharing should occur with a single-threaded execution, so this is not surprising. When two or four threads were used with fix 1 at low padding integer counts the performance was significantly lower than single-threaded performance with fix 1. This is because the array values fall on the same cache line, so when any thread attempts to write to a value all other lines in cache are marked Invalid and must be reloaded after the writing thread flushes the updated line to memory.

However, when the number of padding integers increased to four there was an increase in performance for the two-threaded execution. This is because the cache line is only large enough for 16 4B values, so due to sequential memory allocation at four padding integers there is enough space for all four values `Array[n].value` and all twelve padding integers that fall between values, the final four padding integers flowing to the next cache line. However, with five padding integers the first cache line is contains `Array[0].value`, five padding integers, `Array[1].value`, five padding integers, `Array[1].value`, and three padding integers. Then with the first cache line filled `Array[3].value` is necessarily on the following cache line offset by the final two padding integers of `Array[2]`.

At padding integer count of seven the two-thread execution of fix 1 has a substantial increase in performance, and this is because with seven integers between each value in memory the array is split into two separate cache lines as follows:

Line 1: `Array[0].value (4B) | Array[0].pad (28B) | Array[1].value (4B) | Array[1].pad (28B)`

Line 2: `Array[2].value (4B) | Array[2].pad (28B) | Array[3].value (4B) | Array[3].pad (28B)`

This evenly distributes the values across two cache lines, further reducing the amount of false sharing and resulting in a performance increase that persists across further executions with >7 padding integers.

Execution using fix 1 with four threads did not see as significant a performance benefit from the padding integers offsetting the array values into two cache lines, and this is attributable to the increased thread count. Even when the values are on two cache lines, four threads will continue to load shared cache lines, then cause a flush + reload when one thread writes to a value thereon. There was a slight increase in performance since two cache lines mean that a flush + reload is not necessary for all writes as is the case with values contained within a single cache line.

Fix 1 performance did increase significantly with fifteen padding integers. This is because fifteen integers represent a 60B offset, which leaves room for only a single value per cache line:

Line 1: `Array[0].value (4B) | Array[0].pad (60B)`

Line 2: `Array[1].value (4B) | Array[1].pad (60B)`

Line 3: `Array[2].value (4B) | Array[2].pad (60B)`

Line 4: `Array[3].value (4B) | Array[3].pad (60B)`

This eliminates false sharing because each thread will load a separate line to cache, and so a write operation by a thread will not invalidate the contents of any other thread's cache. Performance in this case approaches that of fix 2's use of private variables, which follows because each scenario eliminates false sharing.