Parallel Processing Vector Multiplication Using OMP
David Goldstein
12.11.2015

Overview

For this homework we were asked to write a report on the scalability and speedup of vector multiplication using multiple threads. This report details my methods and results and answers the following overall questions: what are the effects of using parallel processing on a program's elapsed time? At what point does the use of parallel threads fail to increase efficiency? How does the scale of program data change a program's run time?

The location of relevant code is in the `~/homework/hw6/`directory. All relevant data is saved in the file `output` and listed in data tables in this report.

Methods

Running parallel threads was implemented using the omp library, which supports easy concurrent processing with 'for' loops. All tests were run using a simple bash script (`mxv_run)` which compiles and runs `mxv-omp` using a specified number of rows (m), columns (n) and threads, respectively and averages together a set number of trials. All tests were output to the file `output` which was then extracted into Microsoft Excel to create the below data tables and graphs.

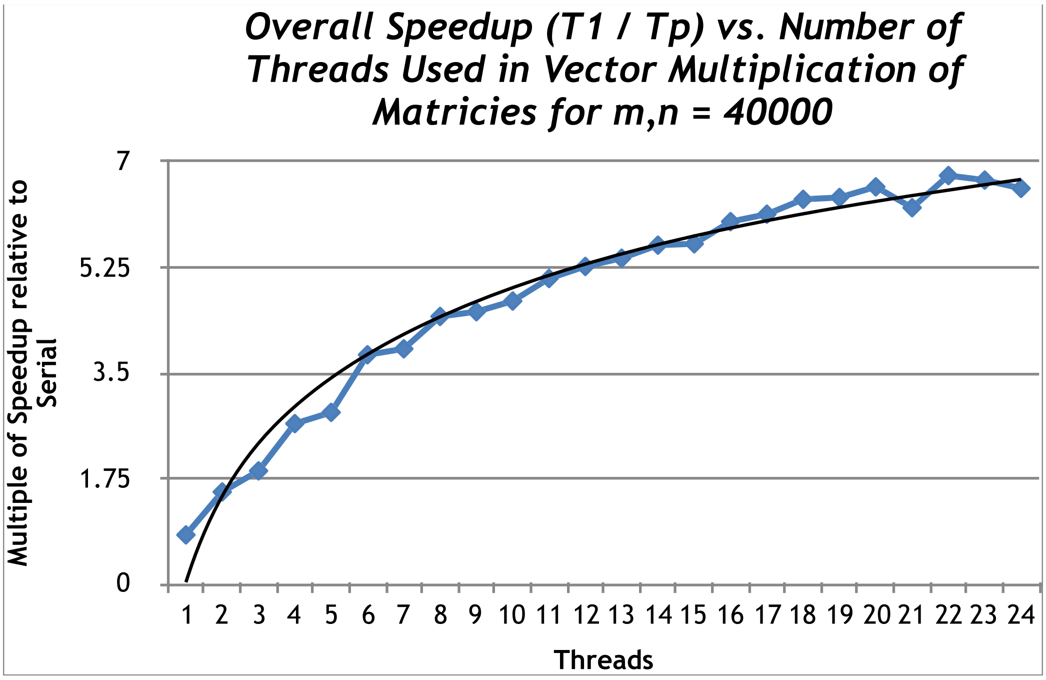
Two main tests were performed. The first aims to determine the overall effects of multi-threading on the vector multiplication problem. In this test, matrix vector multiplication was run on a static matrix with a preset size but a varying number of threads. After initial experimentation, a matrix size of 40,000 columns (m) and 40,000 rows (n). Smaller matrices, such as those in range 2,000 to 15,000 could not accurately indicate the effects of multi-threading, whereas arrays larger than m,n = 50,000 or 100,0000 resulted in overflow on the Nuggle server. A thread count range of 1 to 24 was chosen as this represents the range of cores on Nuggle, from 1 representing one core to 24 representing 12 cores with hyperthreading.

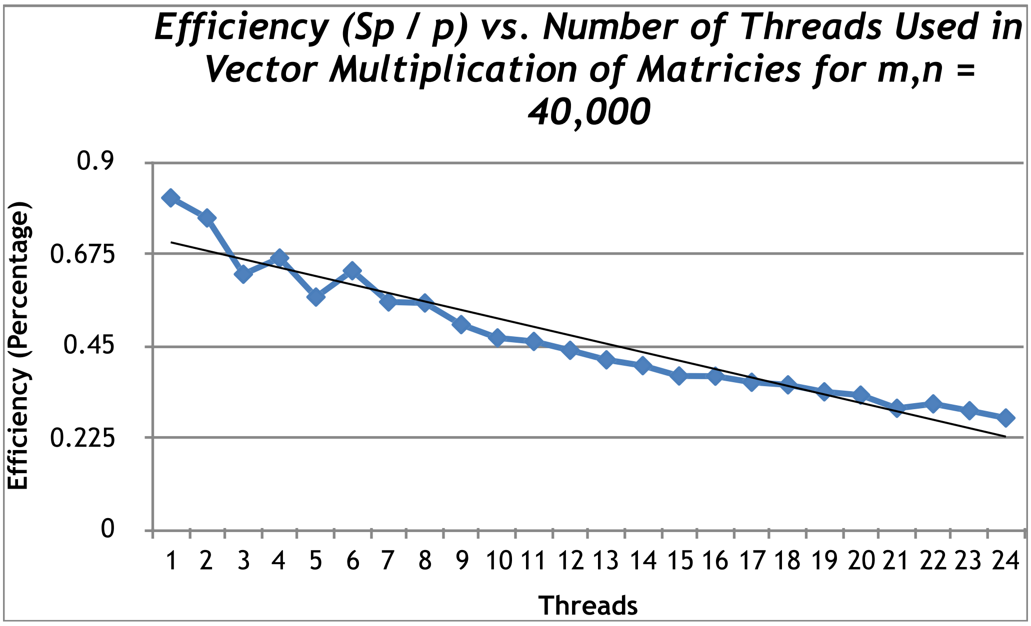
The second overall test performed tackles the question of scalability of a multi-threaded vector multiplication program. To do this, runtimes of vector multiplication method were compared with array sizes and number of threads used. This test was set up using two for loops in the scrip file (`mxv-run`) which incremented the size of array and number of threads used respectively. Similar ranges to those in the first test were chosen.

In both tests, the runtime (as listed in the below section) is a weighted average of ten consecutive tests, for consistent results. This is implemented in the `mxv-omp` file directly, which computes the average of a given number of trials.

Data

1. Effects of Multi-threading





Running Times for Multiple Threads m,n = 40,000

threads	t (micro s)	Speed up (T1 / Tp)	Efficiency (Sp / p)

1	9847641.068	0.81154736	0.81154736
2	5240672.852	1.524962029	0.762481015
3	4263497.194	1.874476925	0.624825642
4	3006548.07	2.658140473	0.664535118
5	2808625.815	2.845458112	0.569091622
6	2102681.413	3.800778881	0.633463147
7	2049258.269	3.899863296	0.557123328
8	1801193.868	4.43696109	0.554620136
9	1769702.28	4.515916151	0.501768461
10	1703075.52	4.692585275	0.469258528
11	1577265.18	5.066888694	0.460626245
12	1517340.561	5.266996292	0.438916358
13	1479035.518	5.403404456	0.415646497
14	1422073.917	5.619839456	0.401417104
15	1416283.261	5.642816891	0.376187793
16	1329669.338	6.010386853	0.375649178
17	1302914.349	6.133808499	0.360812265
18	1252554.84	6.380420923	0.354467829
19	1246513.205	6.411345725	0.337439249
20	1213177.118	6.587518829	0.329375941
21	1280443.585	6.241451948	0.297211998
22	1179594.498	6.775063059	0.307957412
23	1192901.65	6.699485333	0.291281971
24	1217803.2	6.562494751	0.273437281
serial	7991827.108	xx	xx

2. Scalability of Vector Multiplication

Run Times in Micro Seconds for Trials of Varying Column Size (m) and Number of Threads Used

Threads	10000	15000	20000	25000	30000	35,000	40,000	45000
1	78147.32	95889.81	124086.8	160649.7	191923.8	226672.3	318869.7	281396.5
4	17593.3	25874.35	33873.98	42233.05	52390.66	77039.7	68062.53	74600.14
8	17036.57	29707.23	37898.02	49482.76	51026.46	45890.11	59261.8	65026.42
12	11864.59	30070.23	22423.9	28286.17	38196.54	39700.22	46135.21	50053.93
16	12780.68	18347.71	22720.31	22668.47	25930.33	37606.11	40680.51	43593.63
20	16900.91	21982.73	27732.49	26687.8	32602.95	38282.69	43322.78	47188.41
24	17560.2	37986.13	32835.91	23712.75	37809.73	46520.09	50213.31	53044.53

Column Size

Threads

Discussion

The effects of multi-threading are apparent from both tests. In the first test, the overall speedup of vector multiplication increases significantly as more threads are used until a large number of threads is used (23 or 24) and the overall speedup decreases. This is also seen in the second test for the test of m = 45,000. As the number of threads increases, the overall time decreases, however when 23 and 24 threads are used, the overall time increases, as signified by the change of color from blue to red. This phenomenon is likely caused by the overhead thread management being greater than the relative benefit of further parallelization.

This conclusion is also supported by the relative efficiency, graphed above. As the number of threads used in each increases, the efficiency of vector multiplication decreases. This is likely attributed to a greater overhead cost of multi-threading compared to the increased benefits of multi-threading. This graph shows this phenomenon clearer as it takes into account the number of threads used (p) in each trial (as Efficiency = Sp / p).

The second test performed represents a cross section between the increases in performance using multi-threading vs. varying the problem size. Together, these variables speak to the scalability of vector multiplication, or being able increase the number of threads in proportion to the size of the problem and have similar running times. The strips of color within the chart represent this scalability, as these sections vary the number of threads and array size but share a similar run times. While the math to determine an exact scalability goes beyond the extent of this report, it is clear from this graph that multi-threading decreases run time logarithmically whereas problem size decreases run time more linearly. This is seen in the sudden drop in speed from 1 to 4 threads and the slow decline in speed throughout all array sizes.

This test also shows the relationship between cache usage, program size, and run times. As the size of the working set increases, the speed increases slowly until the working set exceeds the capacity of cache. This is because there are less cache hits when the working set becomes much larger (much greater than what the cache can hold). When the working set is too large and lager storage is needed, the program must use a new cache (possibly L2) and speed increases significantly. This is likely the cause for the large increase in speed around m = 35,000. However, after the working set is reallocated, there is a decrease in speed near the 'ridges of temporality' when the working size can comfortably fit in cache. This decrease in speed can be seen for m = 45,000

Conclusion

The results of the first test reveal the effects of multi-threading on vector multiplication. Overall, as the number of threads used increases, speedup increases exponentially until the overhead of thread management slows and even decreases speedup.

The second test performed reveals the rich relationship between problem size and multithreading, and also between cache and program runtime. Overall, multi-threading decreases program run time logarithmically and predictably. Dissimilarly, problem size decreases run time linearly and depends significantly on the size and type of caches available.