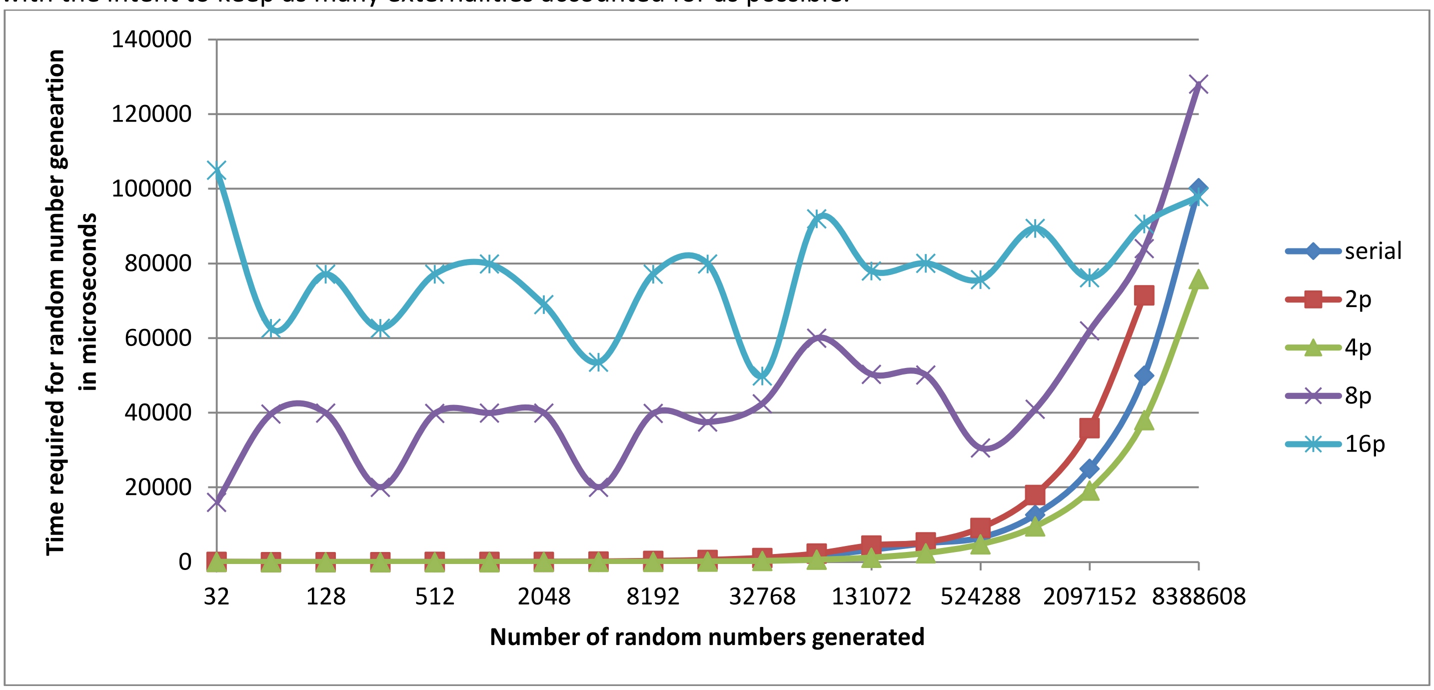
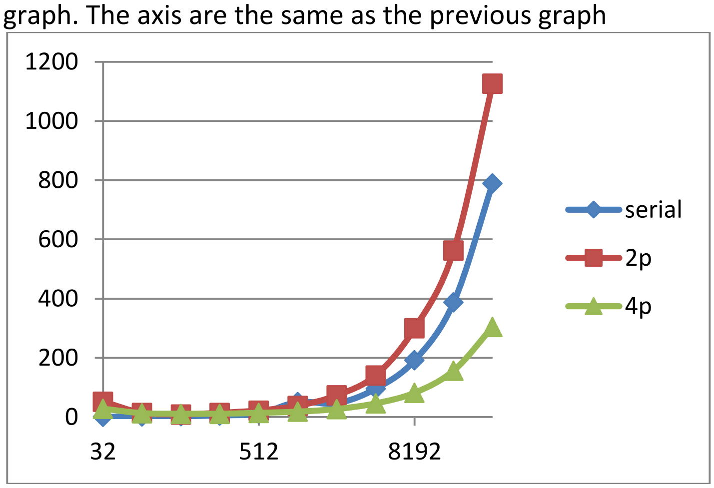
James Jessen and Joseph Villarreal
Parallel Processing: CPTS 483
Programming Project #3

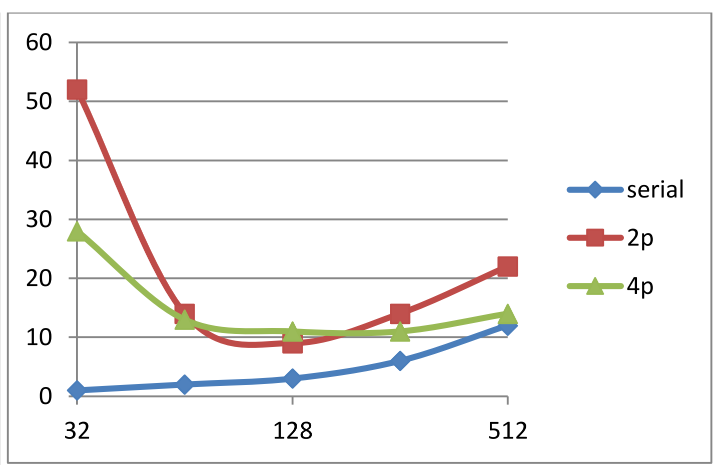
In this project, we wrote two programs, one was a series based random number generator, the other was a parallel prefix based parallel processing random number generator. We ran the series random number generator multiple times and recorded how long it took to generate varying quantities of random numbers. Next we ran the parallel prefix random number generator multiple times, with multiple numbers of processors to do the parallel computations. We had each different number of processors generate the same varying quantity of random numbers, and each time recorded how long it took to finish.

Below you can see the different amounts of time required (in microseconds) to generate a specific number of random numbers. Each different line represents a different number of processors used with the parallel prefix algorithm, with the exception that the serial data points are taken from a different program that generates random numbers in series. Although the serial program was still run on the same hardware that was then used to run the parallel prefix program, with the intent to keep as many externalities accounted for as possible.



It is difficult to see how the serial compares to 2 processors, and 4 processors, here is a closer look at that region of the graph. The axis are the same as the previous graph

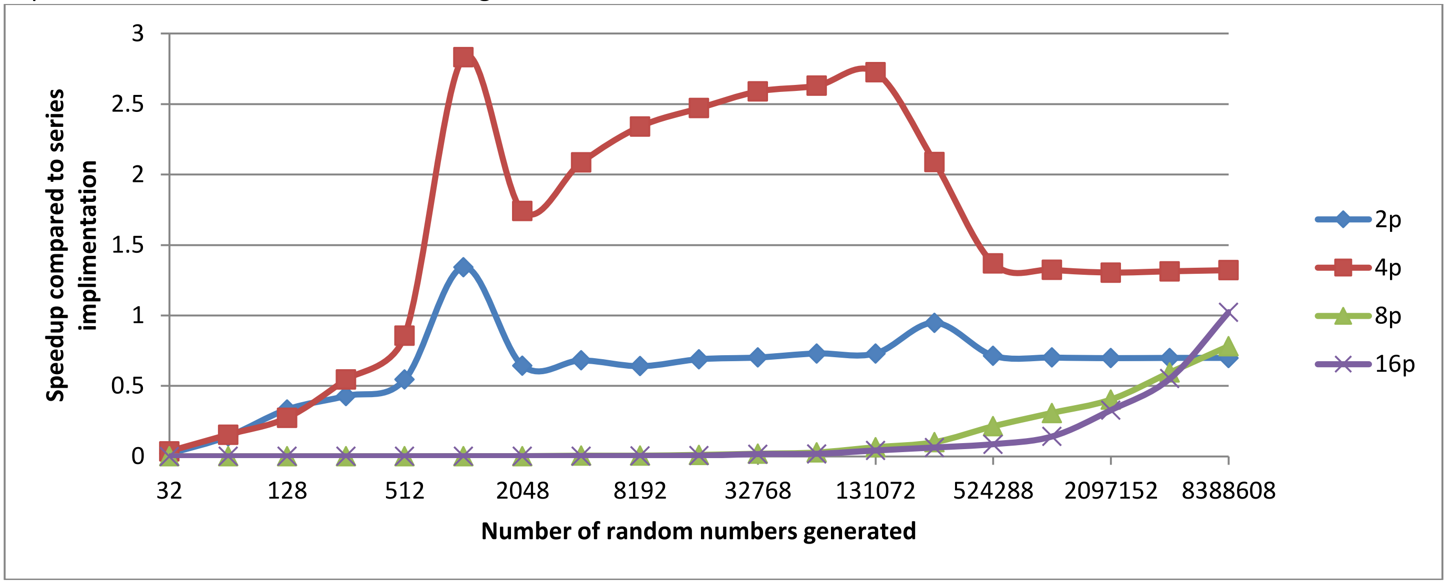




From these graphs, we can see that until the very last data point, the serial implementation consistently out performs the parallel prefix implementation. However, it appears that that condition would not continue to hold true as the number of randomly generated numbers continues to grow. Notably, at the last data point, parallel prefix using 16 processors is holding reasonably steady in the time it takes to complete, yet is now faster than the serial implementation.

Oddly, the parallel prefix implementation using 4 processors out performs the serial implementation for groups of randomly generated numbers greater than 512, and continues to do so all the way until the end of the data set. This could be because each node has 4 cores, and so communication time is almost non-existent, but that is just speculation.

Below you see a comparison of the speedup when using different numbers of processors on the parallel prefix implementation of the random number generator:



Other than the “bump” at 1024 random numbers generated, experienced by both 2 cores and 4 cores, the parallel prefix implementation is consistently slower than the series implementation, having a speedup value of less than one. This changes at the end of our data set, and 16 cores begins a steady improvement, and finally climbs above 1.

You can see though, that using 4 cores consistently yields a speedup value of great that 1. As speculated before, this could be because all 4 cores are location on the same physical processor.

Ideally, we believe that we should have seen all of the various number of processors used in the parallel prefix implementations having some level of overhead that made them less efficient that series for small numbers of random numbers generated, then as the quantities of random numbers being generate grew, we would see their trends indicating their efficiency gains from working in parallel, versus the series generator. This behavior is apparent in both the 4 processor test, and the 16 processor test, however the 2 processor and 8 processor tests do not appear to reflect this. To add some ambiguity to the results, we were unable to collect data for samples of quantities of random numbers generated greater than 10 million. Each attempt resulting in a core dump from the cluster.

APPENDIX

Data:

duration

# rands	serial	2p	4p	8p	16p
32	1	52	28	15976	105028
64	2	14	13	39697	62646
128	3	9	11	39946	77125
256	6	14	11	20079	62658
512	12	22	14	39829	77167
1024	51	38	18	39954	79876
2048	47	73	27	39950	68986
4096	96	141	46	20082	53586
8192	192	300	82	39835	77148
16384	388	563	157	37506	79904
32768	790	1126	305	42448	49905
65536	1647	2255	626	60014	91952
131072	3258	4459	1195	50328	77978
262144	5029	5306	2407	50098	80047
524288	6524	9137	4767	30566	75687
1048576	12627	18007	9540	40961	89434
2097152	24999	35866	19162	61980	76190
4194304	49973	71523	38048	84012	90615
8388608	100173		75814	128037	97918

speedup

# rands	2p	4p	8p	16p
32	0.01923	0.03571429	6.2594E-05	9.52127E-06
64	0.14286	0.15384615	5.0382E-05	3.19254E-05
128	0.33333	0.27272727	7.5101E-05	3.88979E-05
256	0.42857	0.54545455	0.00029882	9.57579E-05
512	0.54545	0.85714286	0.00030129	0.000155507
1024	1.34211	2.83333333	0.00127647	0.00063849
2048	0.64384	1.74074074	0.00117647	0.000681298
4096	0.68085	2.08695652	0.0047804	0.001791513
8192	0.64	2.34146341	0.00481988	0.002488723
16384	0.68917	2.47133758	0.01034501	0.004855827
32768	0.7016	2.59016393	0.01861101	0.015830077
65536	0.73038	2.63099042	0.0274436	0.017911519
131072	0.73066	2.72635983	0.06473534	0.041781015
262144	0.94779	2.08932281	0.10038325	0.06282559
524288	0.71402	1.36857562	0.21343977	0.086197101
1048576	0.70123	1.32358491	0.30826884	0.141187915
2097152	0.69701	1.3046133	0.40333979	0.328113926
4194304	0.6987	1.31341989	0.59483169	0.551487061
8388608		1.3212995	0.78237541	1.023029474