Estimating Pi

Setup

We ran our tests on a single node of the Pleiades Cluster. The main goal was to estimate pi. For this task, we used the fact that for a circle of radius *r* and a square with side length 2*r*, the ratio of the circle's area divided by the square's area is pi/4. We used a square with side length 2 and a circle with radius 1 both centered at (0.5,0.5). We then randomly generated a 2-tuple with values between 0 and 1 and used the distance formula to determine whether the randomly generated point was a "hit" (i.e., if the point was inside the circle) or a "miss." The estimate for pi was then 4 • (# hits/total points generated). To generate these points we used multithreading using the OpenMP library. To test speedup, we ran our code on a fixed input size *n* for number of threads *p*=1,2,4, and 8 on input sizes $n = 2^{10}, 2^{11}, \dots, 2^{31}$. As a note, the code must be compiled with "-lm" as an additional argument in order to access the math.h library.

Runtime and Speedup

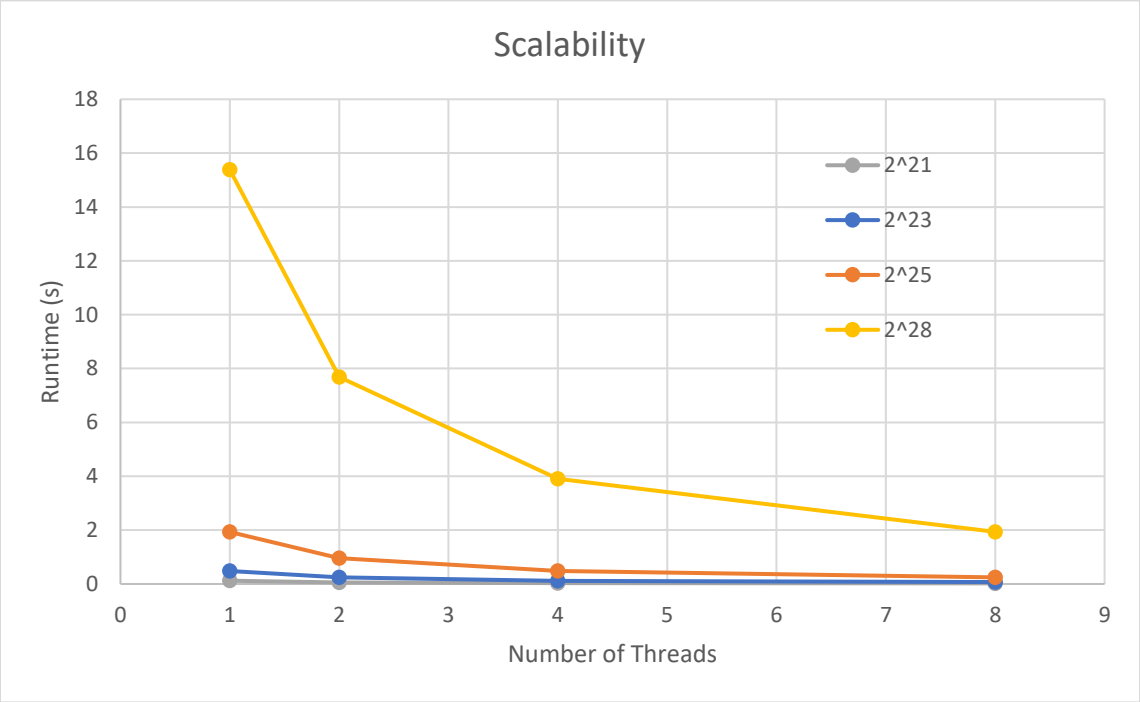
The table of runtimes for our tests is included below where runtime is measured in seconds.

Table 1: Pi Estimator Runtimes (s)

Input size\Threads	1	2	4	8
1024	0.00007	0.00005	0.000035	0.000031
2048	0.000128	0.000084	0.000052	0.003305
4096	0.000244	0.000151	0.000091	0.000051
8192	0.000481	0.00027	0.000161	0.000079
16384	0.000947	0.000504	0.000279	0.003456
32768	0.001896	0.000986	0.000504	0.000261
65536	0.003748	0.001911	0.000975	0.000493
131072	0.007542	0.003802	0.001912	0.005128
262144	0.015018	0.007563	0.003797	0.007024
524288	0.030176	0.015122	0.007671	0.010783
1048576	0.060153	0.03009	0.015105	0.014368
2097152	0.120656	0.060219	0.030209	0.022179
4194304	0.240454	0.120298	0.060175	0.032379
8388608	0.480917	0.241487	0.120463	0.071438
16777216	0.961726	0.481551	0.240483	0.127656
33554432	1.929601	0.962752	0.482472	0.24441
67108864	3.845254	1.924323	0.963347	0.490842
134217728	7.697403	3.847314	1.932007	0.97054
268435456	15.383957	7.681089	3.900172	1.935858
536870912	30.886346	15.437839	7.719335	3.861996
1073741824	61.402746	30.778885	15.407568	7.732065
2147483648	123.114754	61.630794	30.776749	15.533064

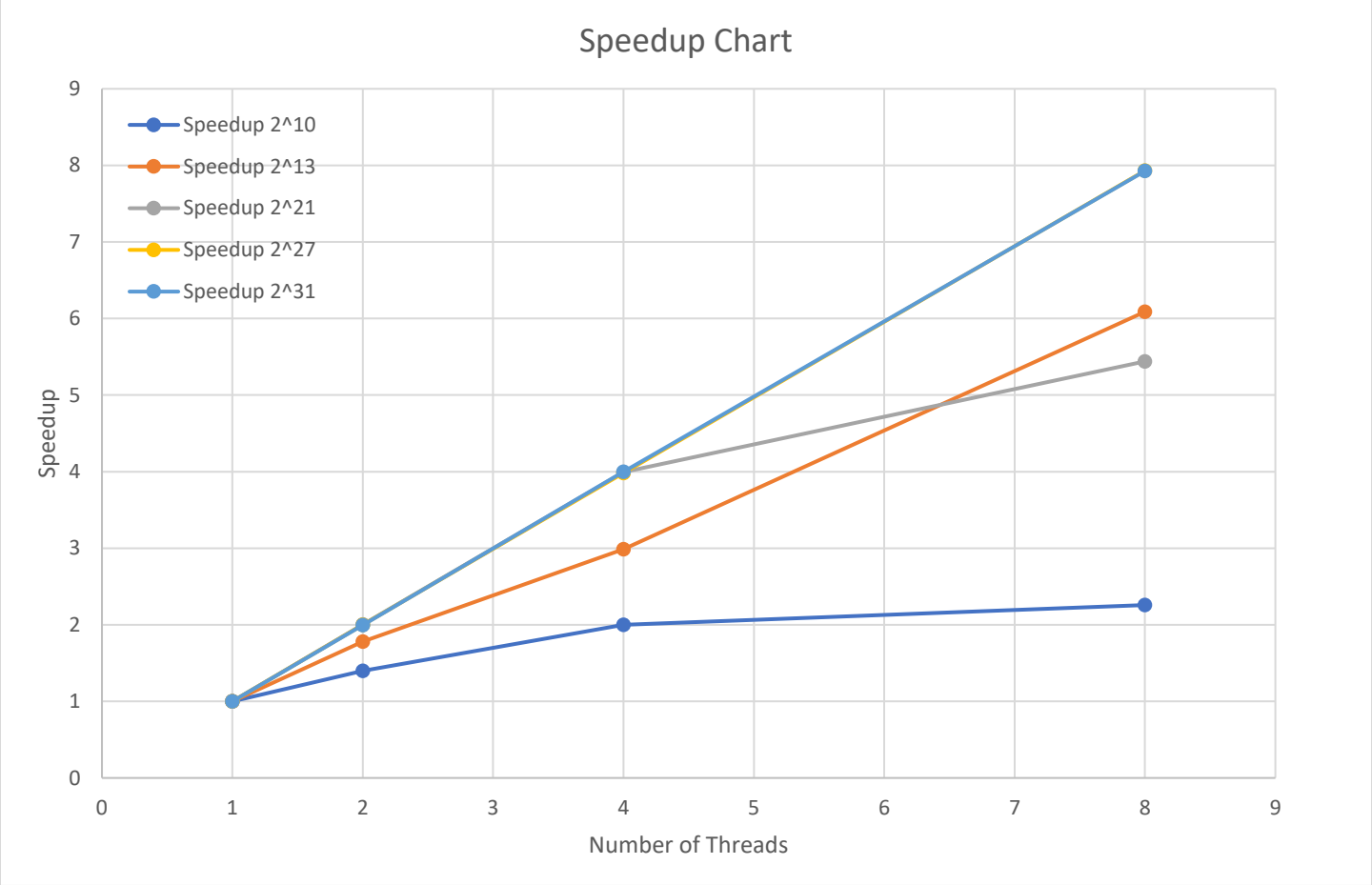
We plotted the runtimes for select input sizes, though this plot is limited in input size range compared to our entire data set since the runtime for a fixed number of threads *p* approximately doubles as the input size doubles (i.e., we limited our range so the differences between runtimes would be somewhat observable in the graph). This plot is included below.

Figure 1: Runtimes for Select Input Sizes



We also plotted relative speedup for select input sizes, though in this case the range of input size is broader since speedup should theoretically not differ between input sizes (though we of course observed actual differences in our practical implementation). This plot is included on the next page.

Figure 2: Speedup for Select Input Sizes



The runtimes we observed seem reasonable. The overall trends show more threads on the same input size completed the task faster than fewer threads, and runtimes generally increased with input size. From the runtime table, it is evident that for smaller input sizes, the benefits from adding additional threads were lower than what the theoretical model would predict. In some cases, the runtimes were actually higher on more threads for a fixed input size (observe in Table 1 the differences in runtimes between 4 and 8 threads for input sizes 2048 and 16384; in fact, the runtimes on 8 threads were actually slower than with the subsequent input sizes of 4096 and 32768). This makes sense because the benefits of parallelizing code are less for smaller input sizes, which means outliers are more likely to occur for a variety of reasons (such as a brief hiccup with the cluster). The speedup results demonstrate this trend in the runtimes with lower overall speedup for smaller input sizes and higher speedup due to parallelizing as the input sizes grow larger. For large input sizes, speedup was very near the theoretical maximum (the above chart demonstrates this with the near overlap for input sizes of 2^{27} and 2^{31}).

Precision

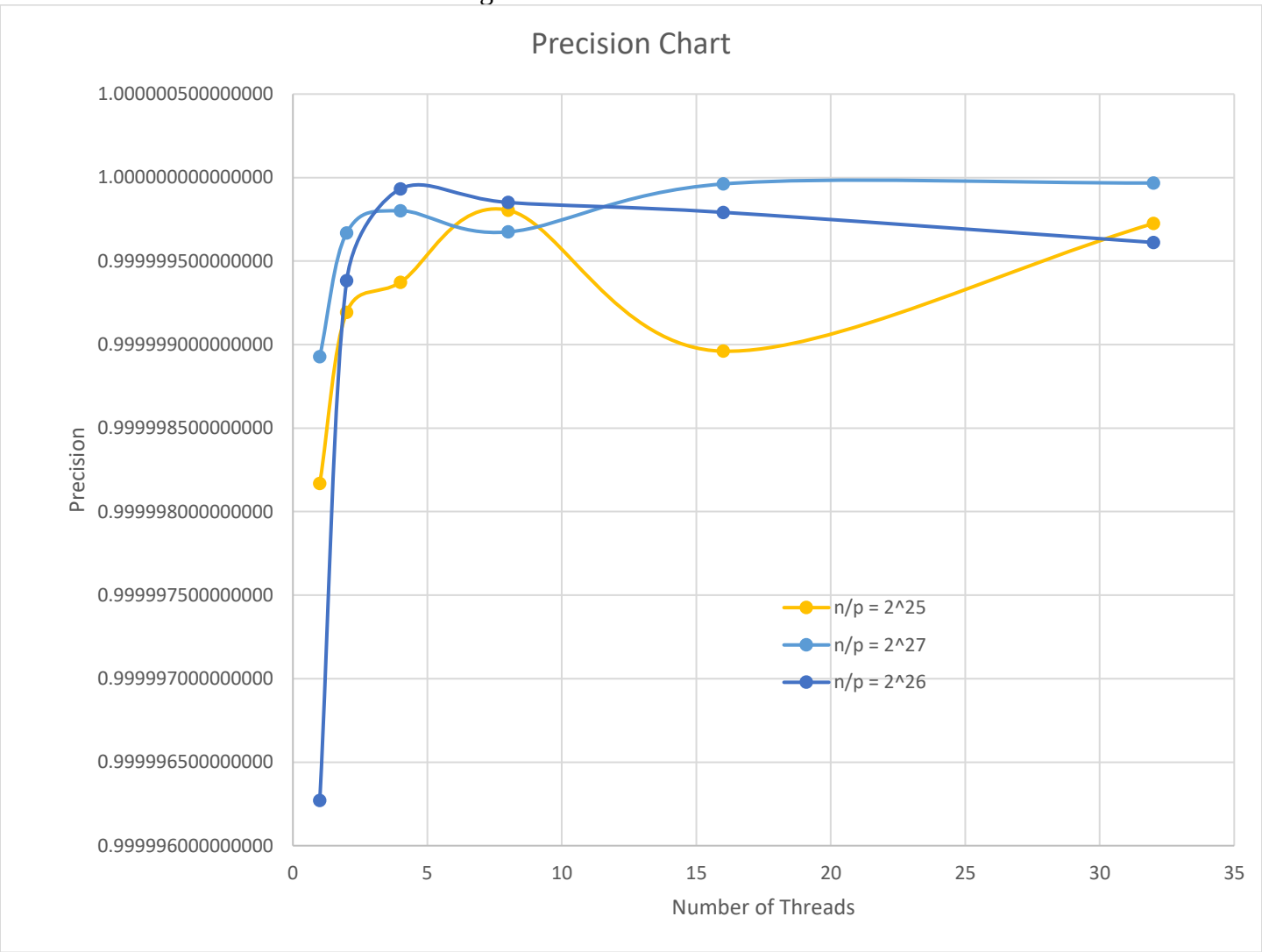
To test our method's precision, we kept n/p fixed and increased the number of threads. Because we ran our code on up to 32 threads and because each of these threads individually ran n/p trials, the max n/p value in this table is smaller than the max n value from Table 1. The results from these tests are tabulated below.

Table 2: Pi Estimates

n/p \ Pi Estimate	1	2	4	8	16	32
1024	3.14843750000000	3.13867187500000	3.13183593750000	3.14355468750000	3.14379882812500	3.14038085937500
2048	3.15039062500000	3.13671875000000	3.14111328125000	3.14111328125000	3.14196777343750	3.14135742187500
4096	3.13867187500000	3.14160156250000	3.14111328125000	3.13940429687500	3.14239501953125	3.14114379882812
8192	3.14453125000000	3.14135742187500	3.14440917968750	3.14135742187500	3.14230346679687	3.14175415039062
16384	3.14160156250000	3.14367675781250	3.14257812500000	3.14138793945312	3.14132690429687	3.14175415039062
32768	3.14172363281250	3.14215087890625	3.14208984375000	3.14144897460937	3.14163208007812	3.14155197143554
65536	3.14080810546875	3.14117431640625	3.14176940917968	3.14131164550781	3.14166259765625	3.14149475097656
131072	3.14071655273437	3.14193725585937	3.14183807373046	3.14160537719726	3.14152717590332	3.14156818389892
262144	3.14190673828125	3.14173889160156	3.14179229736328	3.14153861999511	3.14160728454589	3.14160823822021
524288	3.14170837402343	3.14154052734375	3.14163970947265	3.14163303375244	3.14159917831420	3.14159369468688
1048576	3.14149475097656	3.14157485961914	3.14158630371093	3.14159488677978	3.14159989356994	3.14159083366394
2097152	3.14150428771972	3.14151287078857	3.14159631729125	3.14159870147705	3.14159274101257	3.14159536361694
4194304	3.14158535003662	3.14161872863769	3.14160346984863	3.14159095287322	3.14160060882568	3.14160099625587
8388608	3.14156627655029	3.14158439636230	3.14159917831420	3.14159625768661	3.14159527420997	3.14158856868743
16777216	3.14160704612731	3.14158570766448	3.14158594608306	3.14159029722213	3.14159396290779	3.14159352332353
33554432	3.14158689975738	3.14159011840820	3.14159068465232	3.14159204065799	3.14158938825130	3.14159179106354
67108864	3.14158093929290	3.14159071445465	3.14159244298934	3.14159312099218	3.14159330725669	3.14159387350082
134217728	3.14159601926803	3.14159369468688	3.14159327745437	3.14159367606043	3.14159277267754	3.14159255195409

Even at our largest input size on 32 threads, our pi estimate is only correct to six digits after the decimal (we used 3.14159265358979323846 as our "true" value of pi). Even so, there is a general trend evident in the table where the further down and to the right a pi estimate is, the more accurate is its value. For select n/p values (our three largest), we also plotted precision as a function of the number of threads. This chart is included on the next page.

Figure 3: Pi Estimate Precision



While the pi estimate on one thread was clearly the worst for each of the plotted input sizes, there is not a particularly clear trend for these input sizes that shows doubling the input size leads to a more precise estimate of pi. That by itself is not an unreasonable observation. At these input sizes, our pi estimate is already pretty precise, so there is not a lot of room for improvement. Also, for a pi estimate to be more precise than another estimate, it would basically need to have at least one more accurate digit of pi. Because the digits of pi are in base 10, we hypothesize we would see a more meaningful trend if we multiplied our input size by 10 in between data points. Our current implementation only doubles input size between estimates, so it is possible that we are not increasing our input size fast enough to alleviate the random perturbations due to our algorithm.