Nathan Cochran
Operating Systems – Project #7
8/26/2013

Partitioning:

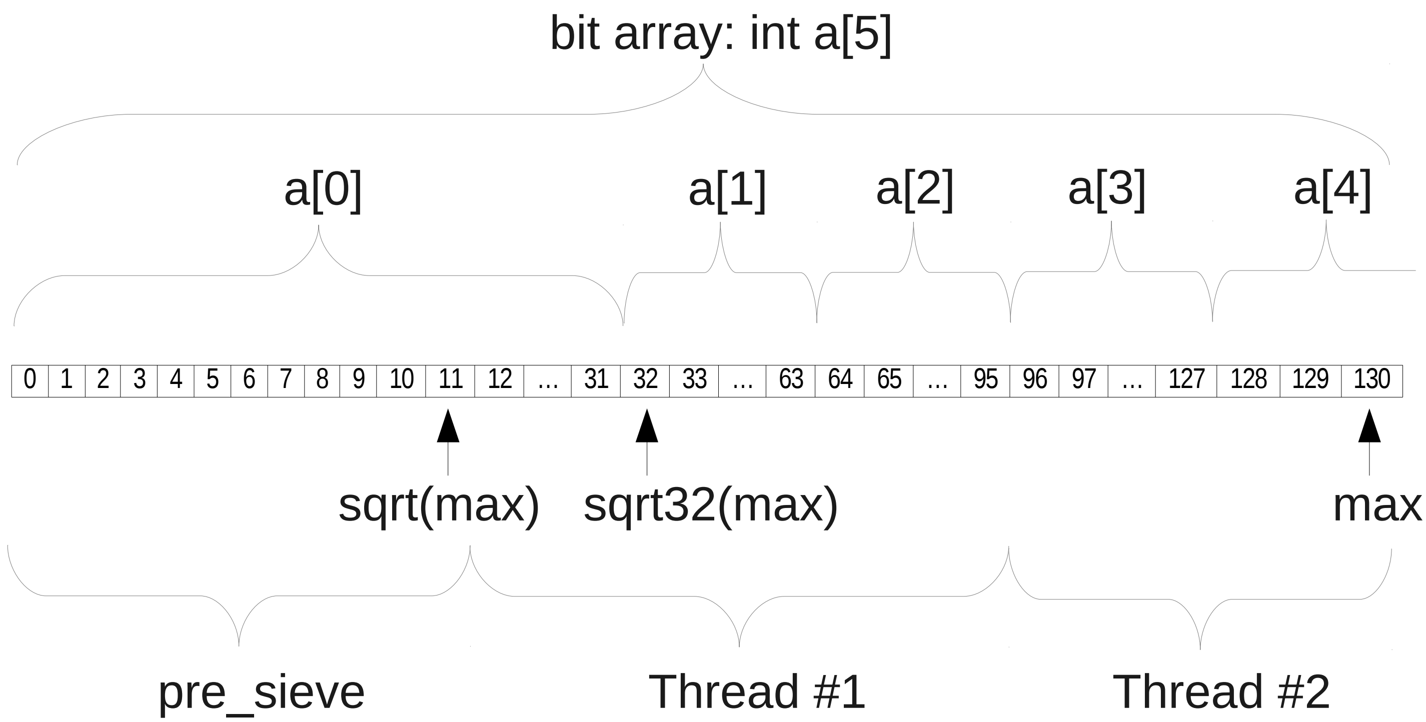
The bit array used in this assignment is formally an array of integers. Each segment of 32 bits, instead of representing a single integer value, represents 32 sequential numbers. In this implementation – a version of the Sieve of Eratosthenes – a bit is set (to 1) if the corresponding number is not prime. Numbers left unmarked are prime. It is noteworthy that the bitwise operation which sets a single bit must access and alter the entire integer in which the bit resides. This operation is therefore not thread safe. Specifically, a race condition arises when two threads attempt to set bits which reside in the same integer at the same time.

My initial plan for dividing the work among the threads/subprocesses involved a static integer which represented the next number to check and possibly sieve the multiples of. Each thread took turns accessing and incrementing it. However, this static number, as well as the bit array (for the above mentioned reasons), had to be locked with a mutex. This resulted in a very slow program. In fact, it got slower with more threads (because the threads had to wait and take turns setting the bits in the shared bit array).

My second approach, therefore, entirely avoids the use of mutex locks. In this implementation, the work is divided among the threads differently. Each thread is apportioned an interval in the bit array, and only that thread accesses or modifies the bits in its interval. Furthermore, to ensure that two threads never have to access bits that reside in the single integer, the upper and lower bounds of each interval fall on offsets of 32. In other words, for any given integer in the bit array, there is one thread alone that is responsible for its bits (though any one thread may be responsible for many integers' worth of bits).

However, this design requires that each thread has available to it, from the start, the list of primes to sieve from it's designated range. Hence, the function pre_sieve() sieves the primes up to the square root of the max prime (-m) before the threads are created. This portion of the program is therefore sequential. Once pre_sieve has completed, however, the threads can all access this initial portion of the bit array (from zero to the square root of the max) in parallel in order to determine which multiples to sieve from their respective ranges. Since no threads modify this portion of the bit array, no race conditions arise. Finally, it should be noted that the intervals begin to be apportioned at the first offset of 32 after the square root of the max . The first thread picks up the slack between the square root and the first offset of 32 after it.

The diagram on the following page represents this in a more schematic fashion.


bit array: int a[5]
a[0]
a[1]
a[2]
a[3]
a[4]
0 1 2 3 4 5 6 7 8 9 10 11 12 … 31 32 33 … 63 64 65 … 95 96 97 … 127 128 129 130
sqrt(max)
sqrt32(max)
max
pre_sieve
Thread #1
Thread #2

Timing Results:

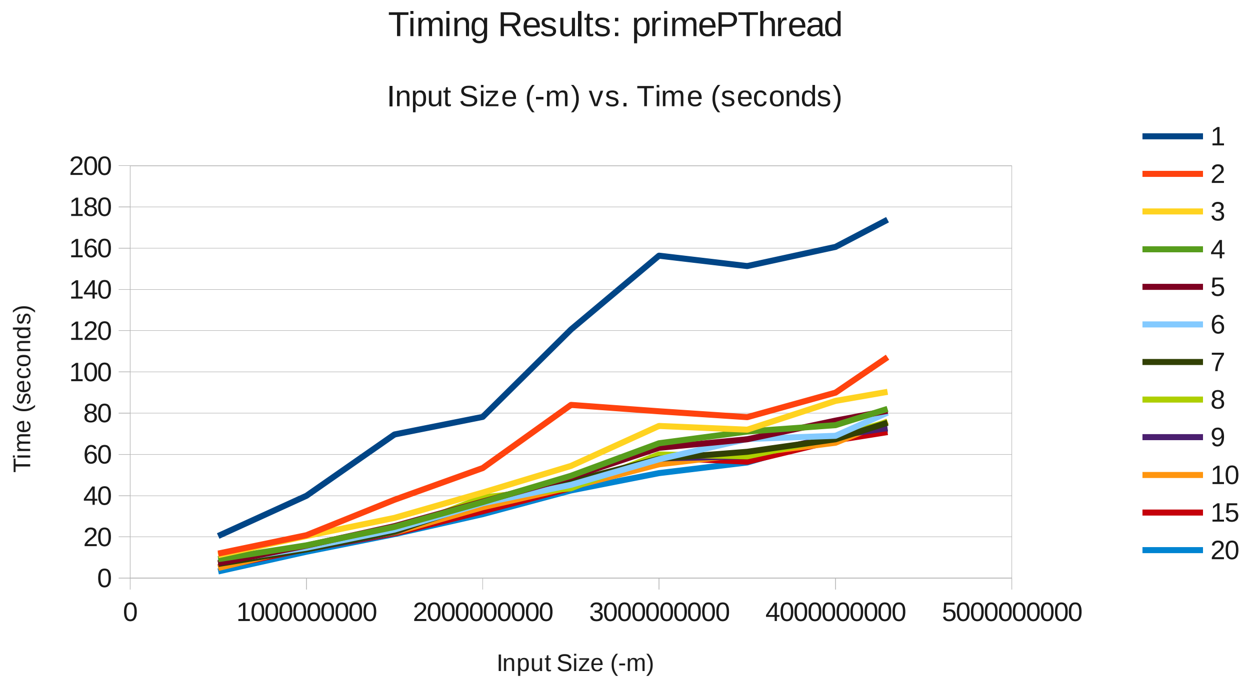
Tests were performed on the os-class server. The concurrency [-c] was varied in the following fashion: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20. The max prime [-m] was increased in increments of 5 million, ending with UINT_MAX. Tests were run on both programs: primePThread and primeMProc. All results were averaged over two trial runs. The raw data is as follows:

primePThread:	**Max Prime (-m):**								
Concurrency (-c):	**500000000**	**1000000000**	**1500000000**	**2000000000**	**2500000000**	**3000000000**	**3500000000**	**4000000000**	**4294967295**
1	20.360741415	39.912089886	69.573692794	78.11298737	120.49120013	156.39924105	151.3012248	160.59213276	173.82407408
2	11.891110956	20.73342068	37.984168014	53.354689829	83.974430109	80.827702874	78.009121329	89.958195261	107.17560226
3	11.043938168	20.363099581	29.198261183	41.474559548	54.355508028	73.831597567	71.966938429	85.911586422	90.288615471
4	9.1377502925	15.828999069	24.858848311	36.819847548	49.645711787	65.368899726	71.033633543	74.200818597	82.015350202
5	7.27016396	15.73112338	25.163995915	37.081141197	48.934952499	63.12095187	67.370928204	76.358216943	81.117329567
6	8.0368150045	14.735599732	23.487868726	36.284999973	45.216269713	57.604441272	67.455385893	69.024377966	80.227971372
7	6.8403936465	14.027880982	22.594439613	36.907973999	46.800967045	57.92167536	61.191144428	67.368252248	75.327978647
8	7.3389155185	14.295362218	22.479705141	39.188256471	43.549549211	59.931690302	59.116222129	68.241679734	75.783221211
9	7.144223283	13.961298849	22.25081488	38.478317606	45.321430615	57.996205162	59.118999357	68.807050583	72.750595321
10	4.890726485	15.265751331	22.103691955	34.393905616	44.140823128	55.163447734	60.225764327	65.707873239	75.930497237
15	5.3358663155	14.056425169	21.661191545	32.345976751	43.62519165	58.443595928	56.43243063	66.770654165	70.736850522
20	3.154691511	12.856978914	21.312655463	30.967206051	42.477866681	50.885555199	56.128920802	68.431036705	72.062098468

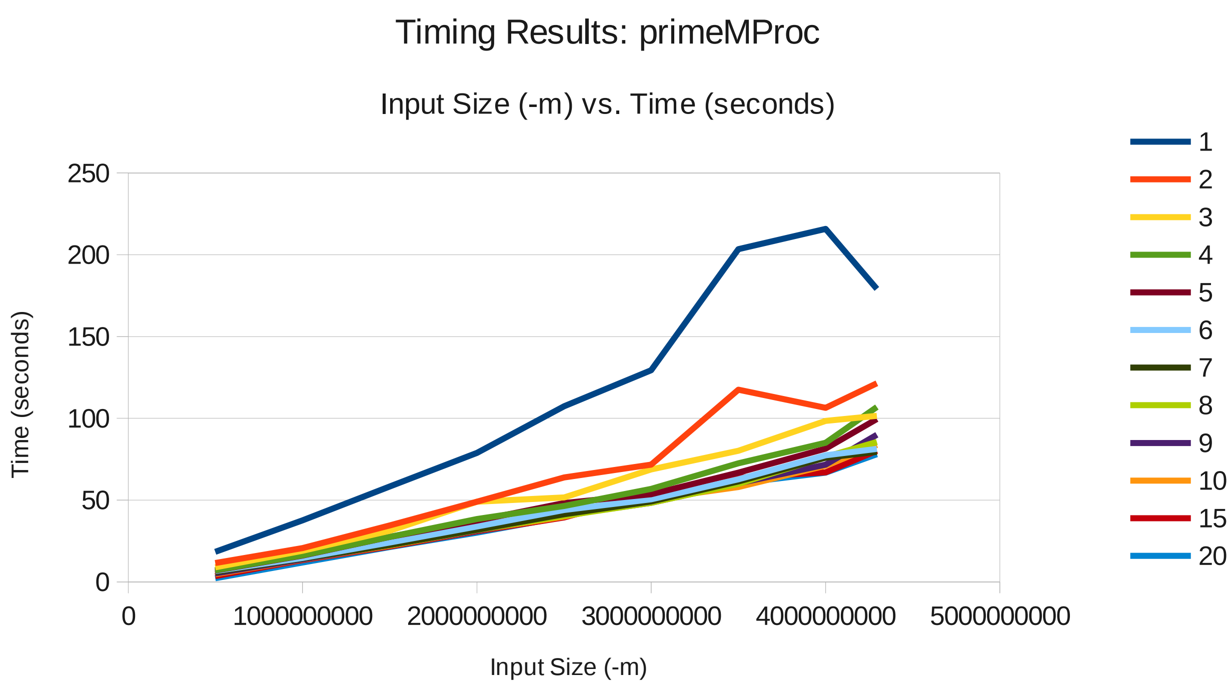
primeMProc:	**Max Prime (-m):**								
Concurrency (-c):	**500000000**	**1000000000**	**1500000000**	**2000000000**	**2500000000**	**3000000000**	**3500000000**	**4000000000**	**4294967295**
1	18.427454432	37.663936985	58.237215119	78.801272632	107.31726249	129.31826662	203.42056042	215.76077054	179.1707407
2	11.567715786	20.691583769	34.525937028	49.072035136	63.896120992	71.680908885	117.48938068	106.40960133	121.45533701
3	8.985714723	19.12083781	31.24519035	48.968498079	51.607232851	68.757354216	80.100363215	98.444873706	101.46245115
4	7.1249912565	16.292578597	27.486719365	38.455774221	46.350627681	56.918571965	72.545740587	85.113067873	106.94607182
5	7.0044294885	16.089491208	27.486313981	37.461326318	48.287115359	53.598006275	66.815381772	81.419776966	99.515584106
6	6.5673776955	14.568844653	24.162161424	33.785356365	43.997807505	50.506572517	62.734314064	77.450010642	81.310632086
7	6.012826067	14.294513061	22.584771433	31.973819327	41.383378464	49.176060435	61.348907168	75.963200351	79.499738121
8	6.093634096	14.498480662	22.497868326	31.747108598	40.431579355	48.455296074	59.992893295	76.34263364	85.478545168
9	5.2063540065	13.766238988	22.432345151	31.364756815	40.790872603	48.309484704	60.272508703	71.689104369	89.921680858
10	4.716305363	13.623420818	22.221059566	31.511929165	39.946613597	50.541858991	57.918163116	70.98030744	83.793153364
15	3.542044578	13.315011394	21.63952083	30.971501178	39.438615108	53.713971326	63.73944802	67.010775926	80.220834884
20	2.269887979	11.861809554	21.294526339	30.055205889	39.925183914	52.371139687	60.343767149	66.796119828	77.967859078

Graphs:

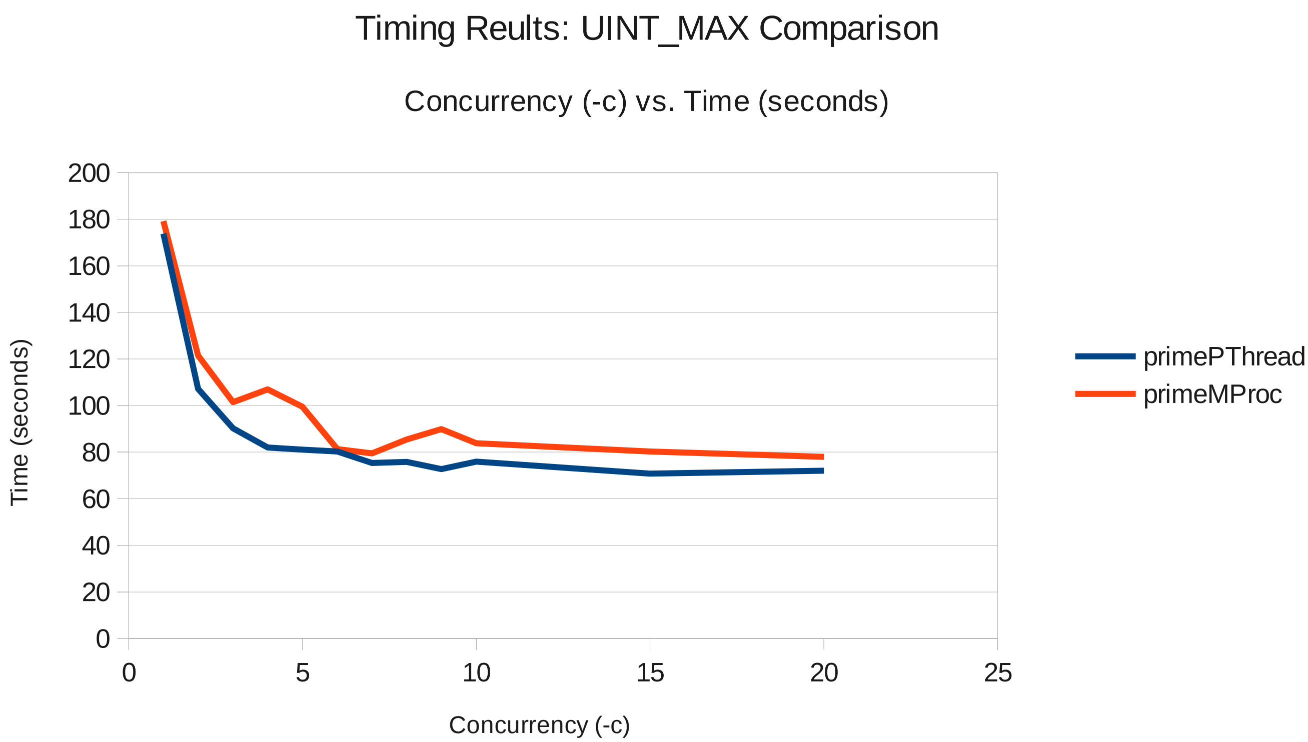
The following graphs depict the raw data in visual form. The first two graphs show time on the y-axis and input size (-m) on the x-axis for the primePThread and primeMProc programs, respectively. The third graph shows concurrency on the x-axis and time on the y-axis, and compares the results of the programs when run with UINT_MAX as the max prime. Both programs show similar results, and the results align with expectations to the extent that higher concurrency meant a faster program (particularly with the jump from c=1 to c=2), but this benefit leveled out with higher and higher concurrency levels.



Timing Results: primePThread
Input Size (-m) vs. Time (seconds)
Time (seconds)
200
180
160
140
120
100
80
60
40
20
0
0
1000000000
2000000000
3000000000
4000000000
5000000000
Input Size (-m)
1
2
3
4
5
6
7
8
9
10
15
20



Timing Results: primeMProc
Input Size (-m) vs. Time (seconds)
Time (seconds)
250
200
150
100
50
0
0
1000000000
2000000000
3000000000
4000000000
5000000000
Input Size (-m)
1
2
3
4
5
6
7
8
9
10
15
20


Timing Reults: UINT_MAX Comparison
Concurrency (-c) vs. Time (seconds)
Time (seconds)
200
180
160
140
120
100
80
60
40
20
0
0
5
10
15
20
25
Concurrency (-c)
primePThread
primeMProc