Homework 3

Name: Chenxi Liu
CWL: chenxil

1. Twin Primes

4	8	16	32	64	128	256	512	1024
3.543	5.593	8.990	14.141	16.025	15.575	22.720	27.598	25.980

Table 1: Speedups of `primes` with respect to different numbers of workers

10	100	1000	10000	100000	1000000
0.005	0.023	0.407	2.273	14.819	18.224

Table 2: Speedups of `primes` with respect to different inputs. Number of workers is 256.

4	8	16	32	64	128	256
0.615	0.563	0.919	1.812	1.726	2.751	2.376

Table 3: Speedups of `sum_inv_twin_primes` with respect to different numbers of workers

The trend in 1b, 1c is that:

1. The speedup increases as the number of workers increases (though not linearly);
2. The speed up increases as the input increases.

The problem itself is embarrassingly parallel since there is no communication/synchronization between workers during the computation. However, the measured speed-up is not linear. One possible factor is that though each worker receives a range of equal length, workers with higher ranges are required to do more work (computing more SmallPrimes) to determine primes. This causes inbalanced work distribution which makes the speedup not linear.

The machine has 64 cores and hyperthreading which makes it capable to handle at least 128 threads without many scheduling conflicts. The real data shows that as the number of workers increases to 1024, the speedup vs the number of workers curve starts to drop.

When the input is small, the scheduling (communications between the scheduler and workers) overhead and idle workers dominate, which makes the speedup even < 1 for inputs, 10-1000. To make the speedup at least 90% of the peak, the input needs to be roughly above 100000.

For 1e, the problem is solved with a parallel reduce. Since there are now collaborations, communication and synchronization overheads make the speedup drop significantly compared to the primes. Furthermore, each worker makes a single pass of its list of primes to find the twin primes; which is much less work than the original sieve. Thus, there is less sequential work to do and more communication overhead. Both reduce the speed-up. Notice that the speedup of 128 workers is even slightly higher than that of 256.

2. Architecture

1. Array results: 2.04081633x speedup

   random array n=1000000 n_trial=10 t_avg=1.600e-01

   ascending array n=1000000 n_trial=10 t_avg=7.840e-02

2. List results: 6.00500x speedup

   random list n=1000000 n_trial=10 t_avg=4.804e-01

   ascending list n=1000000 n_trial=10 t_avg=8.000e-02

In the array case, the two arrays to be merged are accessed sequentially, regardless of the original order of the data – the cache behaviour should be roughly the same for the random and ascending cases. However, the branch in the merge to determine which array has the smaller element is random for random data and should be mis-predicted roughly half of the time. For the ascending data, all of the data will be taken from the first array and then the remaining data will be taken from the second array. There will be a few branch mis-predicts at the beginning of each segment, but the remaining branches should be predicted correctly. For large arrays, this means that most of the branches are predicted correctly. It appears that the difference in the execution time for the random and ascending data is due to branch mis-predictions.

For the list case, the timing measurements show that with ascending data, the time for the list version is about 2% slower than for the array version – this is probably smaller than the measurement randomness. List splitting and merging takes more machine instructions than the corresponding operations for arrays. I'm not sure if the near-match on speed is because of instruction-parallelism in the list version, or if the extra work for lists is balanced by the time to `malloc()` and `free()` the temporary array.

With random data, the slow-down for lists is much more dramatic than that for arrays. For lists, the memory-ordering of the list-cells becomes random as their data values become sorted. Thus, the list version will end up with lots of cache misses once the list is large enough that it doesn't fit in the L1 cache.

3. Message Passing Networks

Let $p(0) = 1$.

(a) For level k (with $k \geq 1$), each of the 10-node clusters in the previous level has $p(k-1)$ communication ports. Each new 16-port crossbar switch uses 10 ports to connect one port of all 10 clusters and has 6 ports remaining for next level. Thus, $p(k) = 6p(k-1)$. Telescoping this formula yields $p(k) = 6^k$.

(b) For level k (with $k \geq 1$), to cut the network into two, for each top-level crossbar switch, we cut 5 out of 10 ports connecting to the next lower level. There are $p(k-1)$ crossbar switches in the top level. Thus, we cut overall $5p(k-1) = 5 \cdot 6^{k-1}$ ports to bisect the network.

(c) The amount of data crossing the bisection is $5 \cdot 10^{k-1}$ KBytes. According to (b), the bisection bandwidth is $5 \cdot 6^{k-1}$ GBytes/second. Assuming 1 GBytes is 10^6 KBytes, the time is $(5/3)k - 1$ microseconds. When $k = 4$, the time is $(125/27) \approx 4.63$ microseconds.

(d) Cutting a torus vertically and horizontally generate the same bisection width. Consider cut a torus into two tubes (each contains $\frac{10^{\frac{k}{2}}}{2} \times 10^{\frac{k}{2}}$ nodes). The bisection width is $2 \cdot 10^{\frac{k}{2}}$.

(e) The amount of data crossing the bisection is $5 \cdot 10^{k-1}$ KBytes. According to (d), the bisection bandwidth is $2 \cdot 10^{\frac{k}{2}}$ GBytes/second. Assuming 1 GBytes is 10^6 KBytes, the time is about $2.5 \times 10^{\frac{k}{2}-1}$ microseconds. When $k = 4$, the time is 25 microseconds.