CPSC 418: Homework 3

Tristan Rice, q7w9a, 25886145

Question 1

(b) Measure the speedup…

Sequential:

```
9> time_it:t(fun() -> hw3:primes(1000000) end, 10).
[{mean,0.6915273465},{std,0.02324092802214097}]
```

Parallel:

```
4> hw3:bench(fun(W) -> hw3:primes(W, 1000000, primes) end).
[{4,[{mean,0.2155105605},{std,0.027648849586389867}]},
 {8,[{mean,0.1467121877},{std,0.020825639597937356}]},
 {16,[{mean,0.0979616279},{std,0.003727850728028083}]},
 {32,[{mean,0.07133679429999999},{std,0.009338938858238176}]},
 {64,[{mean,0.04987700180000001},{std,0.013670005283548841}]},
 {128,[{mean,0.030091090400000003},{std,0.003495737656517365}]},
 {256,[{mean,0.030351726300000003},{std,0.001120994325492375}]}]
```

It seems like there's about a 23 times speedup with 128 processes. Given that there's about 32 cores (64 virtual processors), that makes a fair amount of sense with some small amount of overhead.

(c)

```
5> hw3:bench_n(fun(W, N) -> hw3:primes(W, N, primes) end).
[{10,[{mean,0.0010798956999999998},{std,8.645389528482865e-4}]},
 {100,[{mean,7.235081e-4},{std,3.771179012649232e-5}]},
 {1000,[{mean,6.389056e-4},{std,5.747343394841069e-5}]},
 {10000,[{mean,0.0014029079000000002},{std,7.326087694965982e-5}]},
 {100000,[{mean,0.005242039700000001},{std,0.00208646083376196 45}]},
 {1000000,[{mean,0.029594235400000002},{std,0.006488786981967219}]}]
```

(e) (5 points) Measure the speed up of your implementation of sum_int_twin_primes

Note: these benchmarks also include the time taken to generate the prime numbers.

Sequential

```
4> time_it:t(fun() -> hw3:sum_inv_twin_primes(1000000) end, 10).
[{mean,0.6929405766},{std,0.026442211991545932}]
```

Parallel

```
5> hw3:bench(fun(W) -> hw3:primes(W, 1000000, primes), hw3:sum_inv_twin_primes(W, primes) end).
[{4,[{mean,0.25865003880000004},{std,0.022694313082794}]},
 {8,[{mean,0.16316187040000002},{std,0.010083812151270955}]},
 {16,[{mean,0.1028823829},{std,0.006246809549989811}]},
 {32,[{mean,0.06624921680000001},{std,0.012216561426371333}]},
 {64,[{mean,0.0425495496},{std,0.006864981151478264}]},
 {128,[{mean,0.030518914},{std,0.003925533867756061}]},
 {256,[{mean,0.0288563711},{std,0.001263944620042701}]}]
```

(f) (10 points) Briefly explain the trends you observed in questions 1b, 1c, and 1e.

We see a clear trend from 1b and 1e, that increasing the number of workers increases the performance greatly until you start going beyond the number of virtual cores in the machine. Computing primes and sum_inv_twin_primes are both embarrassingly parallel problems so it makes sense that they scale very well to all of the CPU cores on a machine. Beyond that, there's limited returns since they just start interfering with each other.

When increasing N, it also makes sense that it's more efficient per N the larger it is. When N is small, and the number of workers is high, there's a lot of time spent in communication between the cores relative to the amount of work to be done.

When N is very small, there's also likely some initial slowness caused by cold branch prediction and caches. That might account for the fact that when N=10 it is slower than when N=100.

Question 2

(b) (4 points) Report the elapsed time for

Note: Compiled with `gcc -O3` on thetis.

i. Sorting an array of 1,000,000 random elements.

```
$ ./hw3 random array 1000000 100
random array n=1000000 n_trial=100 t_avg=7.868e-02
```

ii. Sorting an array of 1,000,000 ascending elements.

```
./hw3 ascending array 1000000 100
ascending array n=1000000 n_trial=100 t_avg=2.112e-02
```

iii. Sorting a list of 1,000,000 random elements.

```
./hw3 random list 1000000 100
random list n=1000000 n_trial=100 t_avg=2.174e-01
```

iv. Sorting a list of 1,000,000 ascending elements.

```
./hw3 ascending list 1000000 100
ascending list n=1000000 n_trial=100 t_avg=8.000e-04
```

(c) (8 points) Which takes longer: sorting an array of ascending elements or sorting an array of random elements? By what factor?

Sorting an array of ascending elements is significantly faster, by almost a factor of 4.

Branch prediction plays a large role in this, since at every compare operation the left element will be smaller than the right one. This allows for it to predict the correct branch in every iteration of the merge operation, this saves wasted cycles due to prediction errors.

We also get a lot of cache performance since the merge operation only needs to access the first element of the right side since all the left elements will be smaller than all those on the right. This means that sorting ascending numbers needs to access half the number of elements as sorting random numbers in each merge step leading to vast performance improvements since there will be way fewer cache misses.

(d) (8 points) Which takes longer: sorting a list of ascending elements or sorting a list of random elements? By what factor? Is the ratio bigger or smaller than for arrays?

Sorting a list of ascending elements is much faster than sorting a list of random elements by a factor of ~270. The ratio is much larger than for arrays.

Sorting lists is much harder on the cache since you don't get the benefit of sequences of elements being sequential in memory. Thus, sorting the random list is much slower than a random array.

However, with lists we get a large performance increase with ascending numbers. The same factors that increase performance for ascending arrays also play in here. As well as the fact that with sorted lists there are no need to do extraneous memory copying with a temp array as with sorting arrays. Thus, there are no changes to the list in memory which allow for good caching and no slow writes.

Question 3

(a) Derive a formula for p(k).

$p(k) = 6^k$.

(b) (5 points) What is the bisection width of a machine with 10^k nodes?

For k=1 nodes, the bandwidth is 5.

For k=2 nodes, there are 6 top level switches, each with 10 connections. We have to cut half of those to bisect. Thus the width is 30.

For k=3 nodes, there are 36 top level switches, each with 10 connections. Thus the bandwidth is 180.

For k=n nodes, we see that there are $5(6^{k-1})$.

(c) How long does it take to send these messages?

Each node on the left is sending 1KB to the opposite side. The total amount of data that needs to be transfered is $1KB(\frac{10^k}{2})$. Since each connection can send 1gbps, and the bisection width is the number of connected, we know the total bandwidth is $5(6^{k-1})$ gbps.

$$\frac{1KB(\frac{10^k}{2})}{5(6^{k-1})gbps}$$

$$= \frac{100B}{1gpbs}\frac{(10^k)}{6^{k-1}}$$

$$= 800ns(\frac{10^k}{6^{k-1}})$$

When $k = 4$, it will take $37.04\mu s$.

(d) What is the bisection width of a toroidal machine with 10^k nodes?

Since it's a grid, we can bisect by cutting anywhere aligned with the grid. This would give us a bisection width of $2(10^{k/2})$, since the grid has height $10^{k/2}$, and the edges wrap around.

(e) If we only consider the time to transfer the data across the network bisection, how long does it take to send these messages?

Once again we have to send $1KB(\frac{10^k}{2})$. The total bandwidth is $2(10^{k/2})gbps$.

$$\frac{1KB(\frac{10^k}{2})}{2(10^{k/2})gbps}$$

$$= 2\mu s(10^{k/2})$$

$$2\mu s(10^{4/2}) = 200\mu s$$