PS5

Ben Francis

Question 1

At what problem size do the answers between the computed norms start to differ?

A: At problem size 24

How do the absolute and relative errors change as a function of problem size?

(The following results are experimental; I ran everything one time. Sometimes, this resulted in "None", which is a result in its own right.)

24 R Abs: 8.8 e-16
24 R Rel: 1.4 e-16
24 S Abs: 8.8 e-16
24 S Rel: 1.4 e-16

30 R Abs: 8.8 e-16
30 R Rel: 1.3 e-16
30 S Abs: None
30 S Rel: None

50 R Abs: None
50 R Rel: None
50 S Abs: None
50 S Rel: None

100 R Abs: 1.77 e-15
100 R Rel: 1.5 e-16
100 S Abs: None
100 S Rel: None

1000 R Abs: 7.1 e-15
1000 R Rel: 1.9 e-16
1000 S Abs: None
1000 S Rel: None

10000 R Abs: 1.42 e-14
10000 R Rel: 1.22 e-16

10000 S Abs: None
10000 S Rel: None

100000 R Abs: 5.6 e-14
100000 R Rel: 1.5 e-16
100000 S Abs: 2.2 e-13
100000 S Rel: 6.2 3-16

1000000 R Abs: None
1000000 R Rel: None
1000000 S Abs: 1.1 e-12
1000000 S Rel: 9.8 e-16

10000000 R Abs: 6.4 e-12
10000000 R Rel: 1.7 e-15
10000000 S Abs: 1.3 e-12
10000000 S Rel: 3.7 e-16

100000000 R Abs: 2.7 e-10
100000000 R Rel: 2.3 e-14
100000000 S Abs: 1.3 e-10
100000000 S Rel: 1.1 e-14

500000000 R Abs: 4.0 e-10
500000000 R Rel: 1.6 e-14
500000000 S Abs: 7.4 e-10
500000000 S Rel: 2.9 e-14

10000000000 R Abs: 2.7 e-10
10000000000 R Rel: 2.3 e-14
10000000000 S Abs: 1.2 e-10
10000000000 S Rel: 1.1 e-14

What can we gleen from this? The error grows sublinearly, as seen by the 1e6 growth when we increased problem size from
24 to 10 billion, which is on the order of 1e9 growth. It is also erratic, as at 1e6 problem size there was not enough error to trigger the alert.
The 10 fold increase from 1 billion to 10 billion also did not have noticable change.

Does the Vector class behave strictly like a member of an abstract vector class?
No; the only operator overloading is for indexing.

Do you have any concerns about this kind of behavior?
It forces all of its operations to act on individual members, and the user to self-define vector operations.
My concern is that people will expect the ability to add vectors, and this will not work.

Pnorm/Question 2

**** What was the data race?***

Each thread had a: "partial += x(i) * x(i);" line.
If we expand this shorthand out, we see it is:
partial = partial + x(i) * x(i)
partial is a double, so we cannot use atomic.
There is an "inbetween period", which we can see if we space it out:
partial = partial + x(i) * x(i)
In the RH, the following steps are made (I'm not sure what the exact order is, but it'll be something like the following)
(1) Get partial; this is some numeric value
(2) Get x(i)
(3) Get x(i) (probably cached, so will take less time than the first call to x(i))
(4) Multiply both x(i)
(5) Add the product to the numeric value previously grabbed for partial

(6) Assign partial the sum calculated above

If inbetween "get partial" and "assign partial", one of the other threads has changed partial, the reassigning of partial will nullify
the changes the other thread made. Then we will not get the correct value.
Then the critical section is between getting partial and assigning partial; any overlap here between threads will mess up the result.

**** What did you do to fix the data race? Explain why the race is actually eliminated (rather than, say, just made less likely).***

First, I'm working with async and futures, since we should be thinking of tasks, not threads.
This doesn't by itself solve the problem, but is a good idea.
Second, I'm storing the values received in an array, and then when each future is gotten I'm adding those together.
The partial norm each is calculating does not reference our variable for the "total" result, partial.
As such, there is no shared memory that could result in a race condition.
We calculate the final result using the results from each thread, by waiting to get the futures from each.
This guarantees we will get all of the answers, which are now mutually independent from each other.
In short, we have removed the shared data that caused the race condition, and have used async/futures

to make sure we wait for all of the threads before adding the independent results together.

**** How much parallel speedup do you see for 1, 2, 4, and 8 threads?***

There is a speedup from 1 to 2 threads (my mac has 2 cores, so this makes sense).
Then there is a little bit more speedup from 2 to 4, and then 8 threads there is loss.
I expect the slight benefit from 2 to 4 is due to hyperthreading, and 8 threads is actually worse than 4 because that is just extra threads
to deal with that cannot be parallelized.

Over all of the six problem sizes, I will take an average, max, and min of the ratio from 1 thread to 2,4,8 threads:

Sizes: 2:1 4:1 8:1
1048576: 1.55 1.65 1.47
2097152: 1.69 1.82 1.64
4194304: 1.68 1.84 1.64
8388608: 1.77 2.14 2.09
16777216: 1.47 1.49 1.56
33554432: 1.47 1.62 1.59
AVG: 1.61 1.76 1.67
MAX: 1.77 2.14 2.09
MIN 1.47 1.49 1.47

Interestingly, the speedup has a "peak" around the problem size 8388608.

Fnorm/Question 3

**** How much parallel speedup do you see for 1, 2, 4, and 8 threads for ``partitioned_two_norm_a``?***
This is exactly the same code as what I did for pnorm, so I will quote the results:

Sizes: 1:2 1:4 1:8
1048576: 1.55 1.65 1.47
2097152: 1.69 1.82 1.64
4194304: 1.68 1.84 1.64
8388608: 1.77 2.14 2.09
16777216: 1.47 1.49 1.56
33554432: 1.47 1.62 1.59
AVG: 1.61 1.76 1.67
MAX: 1.77 2.14 2.09
MIN 1.47 1.49 1.47

I am hoping that it is ok; the first problem just said to implement something, so I did what was recommended, which is tasks, not threads.
I take it as a sign of success in program design that this was implemented here as well.

**** How much parallel speedup do you see for 1, 2, 4, and 8 threads for ``partitioned_two_norm_b``?***

There is no reliable speedup; for some sizes there is a difference in speed, but this is both above and below the 1 thread case for various
different problem sizes and thread counts.

**** Explain the differences you see between ``partitioned_two_norm_a`` and ``partitioned_two_norm_b``.***

For the deferred case, each thread doesn't run until we ask for them, with the get() call. These get calls are called sequentially, one per for loop, and stall the loop iteration until the result is "gotten".
Then the next thread is not called until the previous has finished, and we no longer have any parallelism at all, explaining the lack of speedup.

Cnorm/Question 4

**** How much parallel speedup do you see for 1, 2, 4, and 8 threads?***

Sizes: 2:1 4:1 8:1
1048576: 1.80 2.11 0.95
2097152: 1.82 2.06 1.10
4194304: 1.91 2.39 1.03
8388608: 1.89 2.23 1
16777216: 1.85 2.10 0.76
33554432: 1.85 1.99 0.66
AVG: 1.85 2.15 0.92
MAX: 1.91 2.39 1.10
MIN 1.80 1.99 0.66

**** How does the performance of cyclic partitioning compare to blocked? Explain any significant differences, referring to, say, performance models or CPU architectural models.***

The performance is better. With 2 threads, it reaches 1.85 on avg instead of 1.61, with 4 threads it got 2.15 instead of 1.76, but with
8 threads only got 0.92.

Why is it better with 2 and 4 threads? I expect because each of the threads was acting on one contiguous chunk of memory, instead of having to shuffle in different chunks that each was working on independently into caches.
For example, if there was 20 values and 2 threads, then all 20 values could be brought into a close cache, and then both threads work on it in parallel, without needing to move the cache line.
In the old way, both threads would be competing for space, causing unnecessary and harmful cache misses and movements.
The 4 threads is due to the same effect, but with hyperthreading.
The above argument doesn't work for any cache's that are on-core, since they can't share the cache's, but shared caches (like presumably L3) this does work with.
Why is 8 even worse? I'm not sure exactly, but something along the following lines:
8 is twice as many threads as can be handled in parallel, even with hyperthreading. This then just becomes overheard that the OS must deal with, with no parallelization benefit.
With the cyclic partitioning, perhaps the overhead handling of the extra threads interferes with these effects.

General/Question 6

**** For the different approaches to parallelization, were there any major differences in how much parallel speedup that you saw?***

Some differences, but the trend is the same: Between 1.5x to 2x better for 2 threads, even better for 4 threads (because hyperthreading),
and a decay in performance for 8 threads.

**** You may have seen the speedup slowing down as the problem sizes got larger -- if you didn't keep trying larger problem sizes. What is limiting parallel speedup for two_norm (regardless of approach)? What would determine the problem sizes where you should see ideal speedup? (Hint: Roofline model.)***

For smaller problem sizes, each core can use its own L1 and L2 caches, which should give twice the improvement in bandwidth.
This lets the machine improve its performance, since it is less memory bound and more compute bound.
For larger problem sizes, if L3 or higher memory storage is used, memory is still shared, and so the only benefit is computational, not memory.

Conundrum #1/Question 7

---

1. What is causing this behavior?

When the problem is ran, run_partitioned is called.
This runs two_norm in a loop for "ntrials" times.
How big is ntrials, then? It is calculated using num_trials(size), where
size is iterated from the lower size provided to ./pnorm.exe to the higher size, in factors of 2.
This is calculated by ceiling dividing 5E8, which is 500 Million, divided by the size.
For "./pnorm 128 256", this will be 5E8/128=~4e6 and 5E8/256=~2e6.
Then this will run two_norm millions of times, which causes the slowdown.

Why does it do this? It's trying to get a good number of trials it should run for testing the time it takes,
but is assuming a linearity for how long the program will take.
However, there is a fixed cost associated with each program, such as the forking of threads, that this does not account for.

2. How could this behavior be fixed?

It could be fixed by limiting the size of the trials, to ,say, 10,000.

3. Is there a simple implementation for this fix?

This can be done by using fmin():
fmin(std::ceil(5E8 / static_cast<double>(nnz)),1e4);

This solves the problem.

Parallel matvec / Question 10

**** Which methods did you implement?***

I implemented CSR and CSC^T, since CSR^T and CSC don't partition y, which can lead to race conditions.

Algorithm	Partitioned vector
CSR	y
CSR^T	x
CSC	x
CSC^T	y

**** How much parallel speedup do you see for the methods that you implemented for 1, 2, 4, and 8 threads?***

Let us examine just one problem size, 512, which should approximately characterize the pattern, though there is fluctuation among the other sizes.
I will look at the ratios between 1:2, 1:4, and 1:8, for each type.

CSR 2:1 --> 2.95:1.52 --> 1.94
CSR^T 2:1 --> 1.38:1.36 --> 1.01
CSC 2:1 --> 1.37:1.36 --> 1.01
CSC^T 2:1 --> 2.95:1.52 --> 1.94

CSR 4:1 --> 3.32:1.52 --> 2.18
CSR^T 4:1 --> 1.37:1.36 --> 1.01
CSC 4:1 --> 1.37:1.36 --> 1.01
CSC^T 4:1 --> 3.38:1.52 --> 2.22

CSR 8:1 --> 3.17:1.52 --> 2.08
CSR^T 8:1 --> 1.38:1.36 --> 1.01
CSC 8:1 --> 1.38:1.36 --> 1.01
CSC^T 8:1 --> 2.91:1.52 --> 1.91

These methods do achieve the approximate 2X speedup expected.

Conundrum #2/Question 11

---

1. What are the two "matrix vector" operations that we could use?

We can consider CSR, CSR^T, CSC, and CSC^T.
They are distinguished by if they partition x or y;
CSR^T and CSC can only partition x, while CSR and CSC^T can only partition y. (due to the mechanical constraints of matrix multiplication)
We can only use CSR and CSC^T, since if y is not partitioned it will lead to race conditions (without extra work needed to prevent this).
To see this, consider that if y is partitioned, there will be no race conditions. One contiguous chunk of y is handed to each thread, so that the threads
don't act on the same element in memory, but rather independent parts of memory. It is the getting and writing the same piece of memory that results in race conditions.
So the two operations we choose from are CSR, which would require first taking the transpose, and CSC^T, which would not.
For this reason, we shall use CSC^T.

Furthermore, we need to use the transpose matrix multiplication, since to find the stationary distribution we need:
x = x*P, or x = P^T * x.
We do this by taking P^T*P^T*...*(P^T*x) many times.
If we are given P, each of these will be a tranpose matrix vector multiply.
For the above reasons, we cannot use CSR^T as currently formulated, since it will result in race condtions, whereas CSC^T will not.

So I suppose there are two ways of answering which matvec ops we could use:

(1) We need them to be transpose ops --> CSR^T and CSC^T
(2) We need them to not have race conditions --> CSR and CSC^T

The common denominator is CSC^T, so we use it.
Or rather, we change our functions so that they read the data as CSC, and then when it is multiplied
the C++ code looks to see if it is given the order of operations [matrix, vector] or [vector, matrix],
and chooses to use the transpose operation, which is CSC^T.

2. How would we use the first in pagerank? I.e., what would we have to do differently in the rest of pagerank.cpp to use that first operation?

If we did want to use CSR, we would have to transpose the data first, which would incur an unwanted computational cost.

3. How would we use the second?

We would have to read in our data as CSC; which is what we do!
This does indeed result in better results than for CSR.