A Distributed Approach to IQM in MRI Ruby and JRuby

Background:

At my work, we have a challenge that we give all technical applicants:

Given an arbitrarily large data set, sort it and find the average. There's a little more to it than that, but the idea is that it tests your knowledge of memory use and algorithms.

I'm not very good at algorithms, but I am good at brute forcing the shit out of things, so I wondered: if I dispatch chunks of it to be calculated in parallel, could I get a *pretty close* approximation, and could I make it go faster?

To get the idea going quickly, I used Ruby and the thread library for "parallel processes". I say that in quotes, though, because threads in MRI Ruby don't actually get executed in parallel. So I'm sort of interested in what results that will net, and then compare it with a similar approach using true threads JRuby and distributed processes in Elixir.

Plus, it gets me started playing with Excel really fast.

Estimated IQM refers to the IQM calculated using parallelism/distribution, while *real IQM* refers to calculation on a single thread.

Comparing *estimated IQM* to *real IQM*, what is:

1. The effect of a growing set size on accuracy?
2. The effect of a growing set size on speed?
3. The effect of growing the number of parallel agents/processes on accuracy?
4. The effect of growing the number of parallel processing agents on speed?

Materials:

The data set is a set of N randomly generated numbers from 0 to N, with an expected mean of N/2.

To calculate an IQM, one must sort the set, strip off upper and lower bounds, apply a modifier depending on the size of the set, and averaging the inner range along with that modifier.

TL;DR – the IQM will be close to N/2 with a random distribution, but to get a comparison of speed I had to calculate the IQM for both scenarios.

Also, in real life we won't have a uniform distribution, but let's play with it for now.

Methods:

To answer questions [1] and [2], the data set size was increased while the sampling factor was held constant at 0.1. If we had a 10MB data set, we'd get:

10MB = 10 agents at 1MB each

Thus*, the number of parallel agents is the inverse of the sampling factor*. E.g.,

100MB set sampled at 0.01 results in 1MB chunks distributed across 100 agents, and with a constant sampling factor, the size of the subsets scales linearly with the data set size.

A sample size of N=20 was collected at each value for data set sizes ranging in powers of 10 (logarithmically) from 10^2 to 10^7.

Each run, I recorded:

- Set size
- Sampling factor
- Estimated IQM value
- Estimated IQM speed
- Real IQM value
- Real IQM speed

To get the *estimated IQM* standard deviation, the data set was kept constant within populations to give me the same *real IQM* each time. The *estimated IQM* calculation shuffled the data each run, as it would in the wild to promote uniform distribution across parallel agents. This sacrifices some speed for accuracy.

To answer questions [3] and [4], the data set size was held constant at 10^6 while the sampling factor was increased logarithmically from 10^-5 to 10^-1, resulting in 10 to 10,000 parallel agents.

At 10,000 agents, MRI Ruby would not run. Likely JRuby cannot perform under this condition as well. I cannot make an educated guess for Elixir.

Results/Discussion:

MRI Ruby:

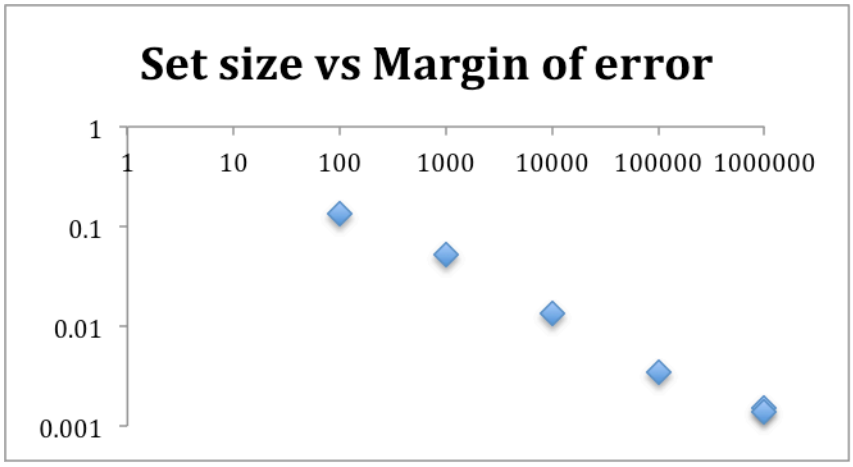


Figure 1. Processing in parallel with 10 agents; MRI Ruby

Accuracy scales with data set size. This is purely mathematical - we end up getting subsets that more approximate the larger set.

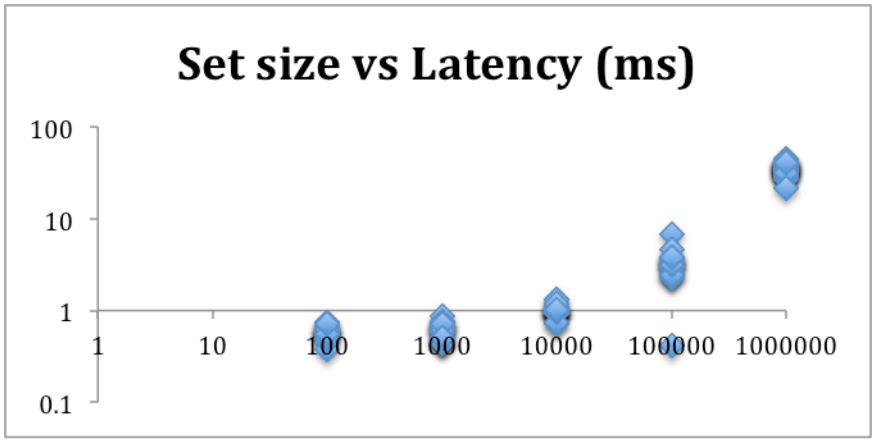


Figure 2. Processing in parallel with 10 agents; MRI Ruby

For a little while, our performance is better but it **quickly falls off** and becomes far worse than calculating IQM on a single thread.

Why is this? Perhaps it's the shuffle before we dispatch a data subset to an agent. Perhaps it's the overhead of threads with more and more memory, without the benefit of true parallel computation.

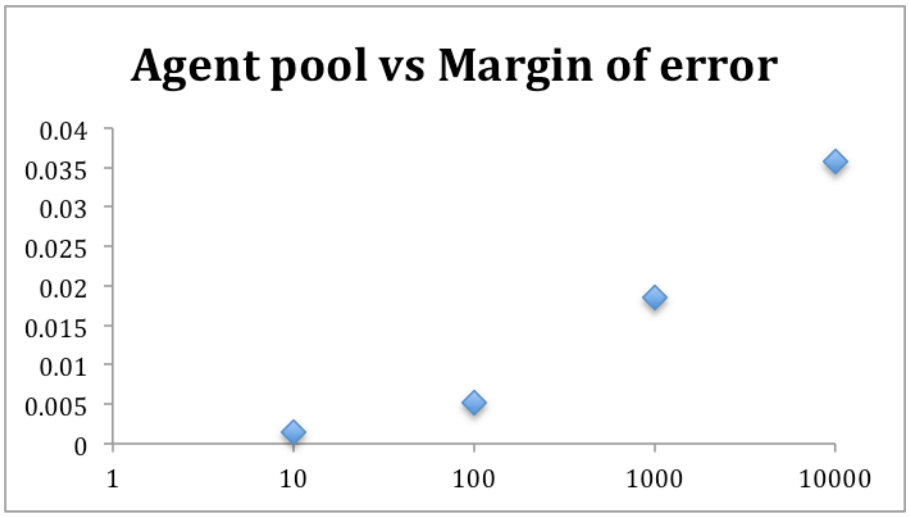


Figure 3. Processing in parallel with 1e6 data points ; MRI Ruby

Like Figure 1, this is purely mathematical. For a fixed size data set, the more agents we have, the smaller their subset will be and the less their calculations will approximate the larger data set.

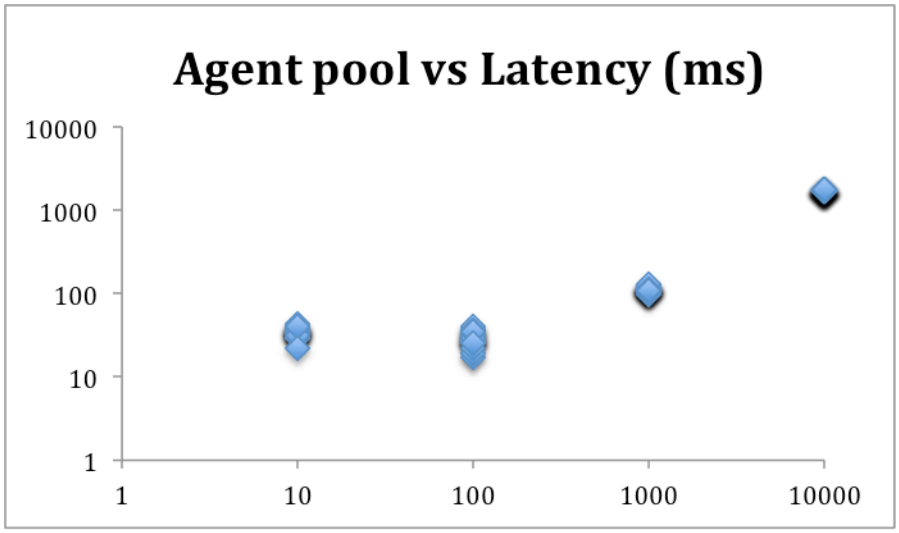


Figure 4. Processing in parallel with 1e6 data points; MRI Ruby

Again, with MRI Ruby we see an increase in latency with distributed processing as we increase the number of agents. At the high end, we're seeing an extra 75 seconds for distributed calculation that takes its local counterpart only 2 seconds to perform.

Results/Discussion (cont.):
JRuby:

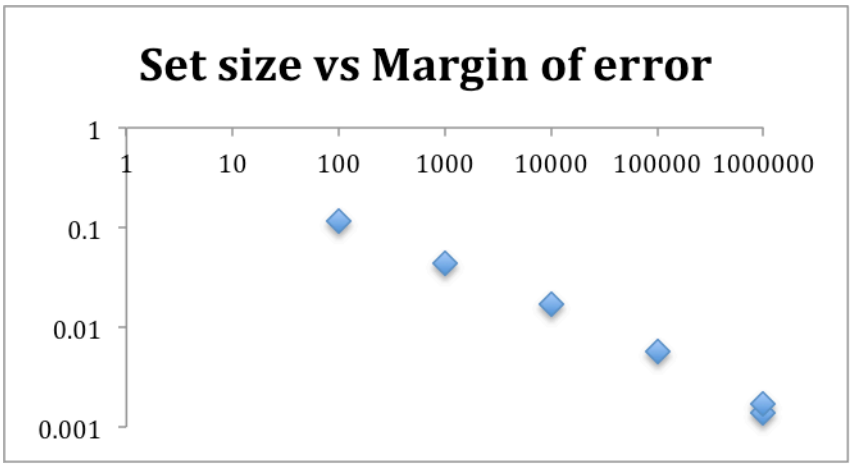


Figure 5. Processing in parallel with 10 agents; JRuby

No surprise here. Math transcends programming languages.

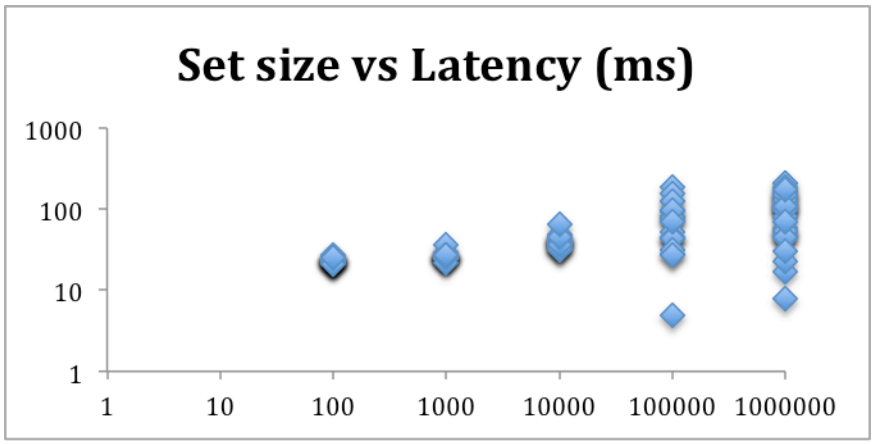


Figure 6. Processing in parallel with 10 agents; JRuby

This interests me because we don't see the same trend in MRI Ruby. Might have to run this for 1e8, 1e9, and 1e10 data sets to see if we start to dip below the 0 line, which would indicate we've started to achieve a performance gain.

Also, there seems to be more variation within populations.

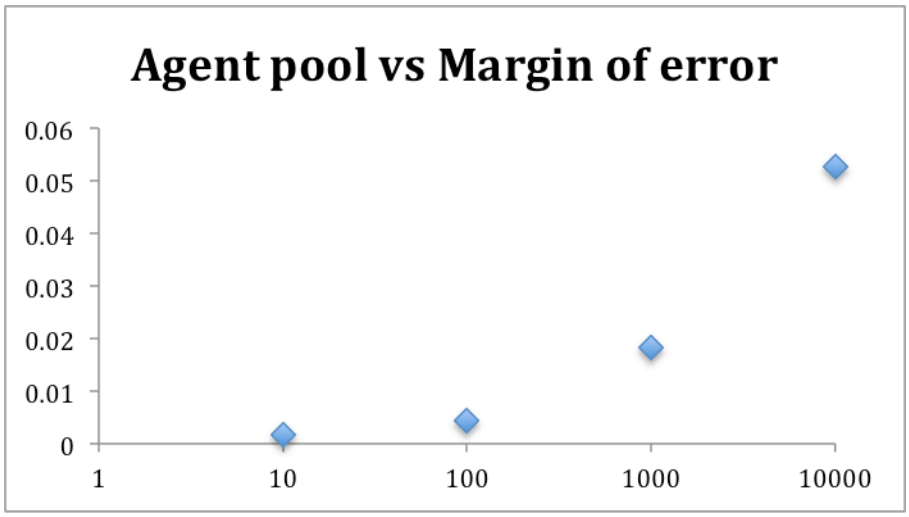


Figure 7. Parallel processing with 1e6 data points; JRuby

No surprise here.

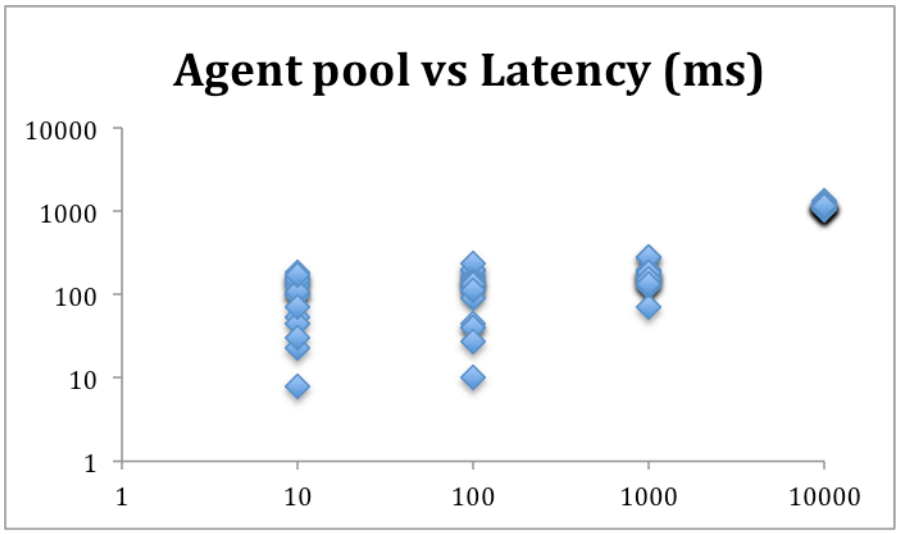


Figure 8. Parallel processing with 1e6 data points; JRuby

Here's what really cripples my thesis that distributed processing in JRuby would be better than single-threaded computation of an IQM. If it were true, I'd have expected to see that as we increase the # of threads, we dip below the 0 line to achieve a performance boost.

Further Investigation:

1e6 data points seems to be below the threshold for a potential performance boost in JRuby. Furthermore, anything over 10,000 threads seems to hurt us. I'll need to investigate what this looks like at 1e9 data points and above, as well as more than 10 threads but less than 10,000.

Other interesting notes:

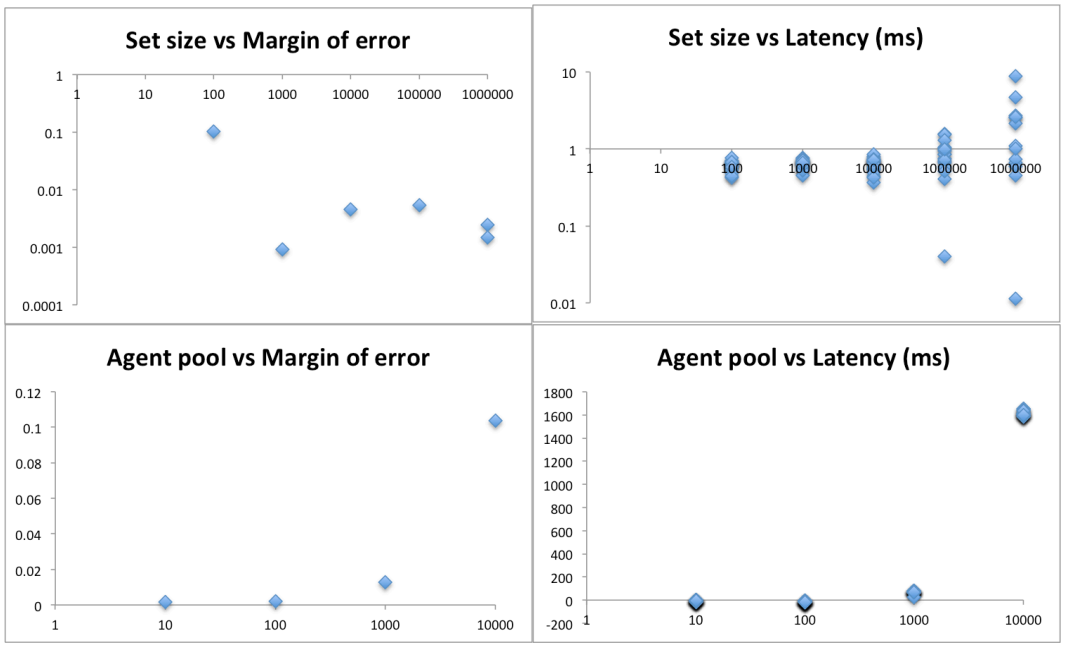
- JRuby took significantly longer to calculate the same values:
    - N=1e6: MRI (209ms, $\sigma = 7.7$), JRuby (415ms, $\sigma = 21.8$)
- In MRI Ruby, single-threaded computation of a fixed-size data set did not vary with increased agent pool size conditions, but in JRuby it did.
- JRuby can't allocate enough memory to have a subset of 1e5 on a single thread.

~~It's disappointing that I didn't see what I'd hoped for, but that's science for ya.~~

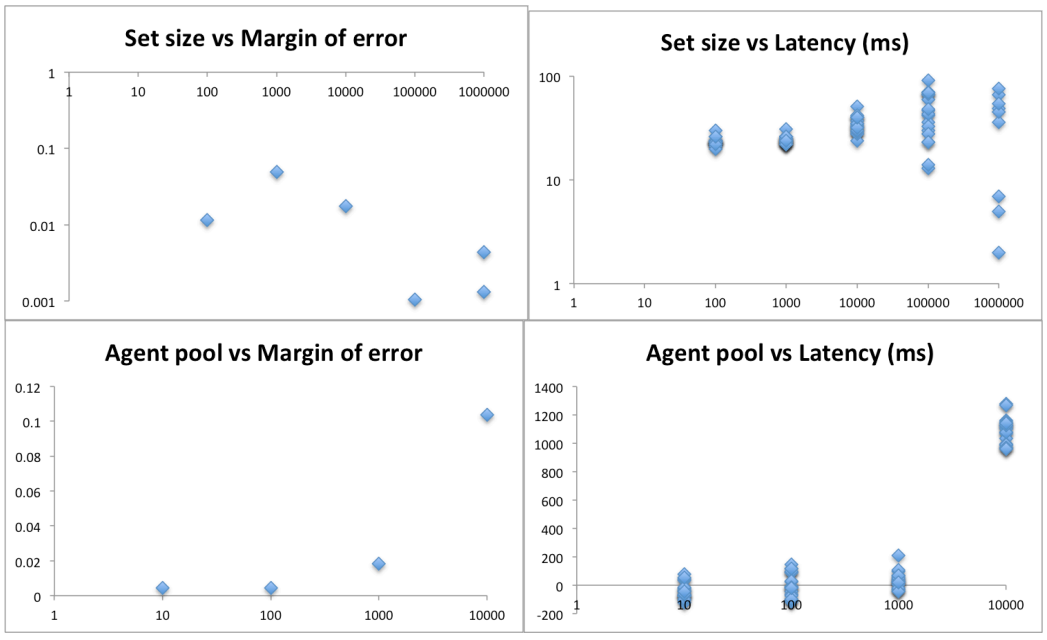
Check out Appendix B for evidence that after a certain point, JRuby threading does boost performance.

Appendix A: Cheating by not shuffling before parallel distribution

MRI Ruby:

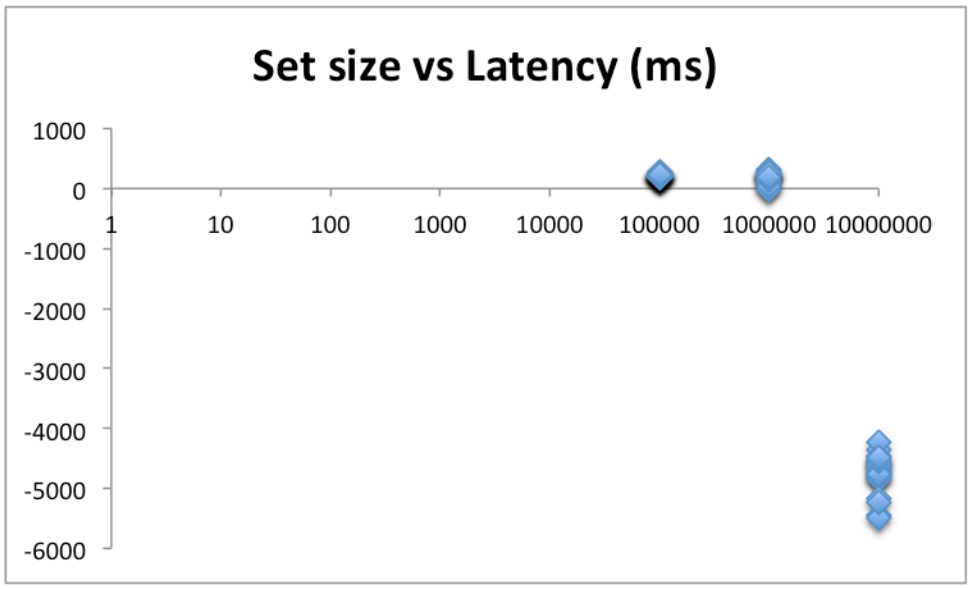


JRuby:



Result:
Parallel information is still useless in MRI; somehow faster at small data sets?
If we don't shuffle, we get a speed increase at the cost of accuracy.

Appendix B: 1e5, 1e6, 1e7 data set size with 1000 agents, JRuby



Result: Holy shit it worked! That is way faster!