Bar file Parser comparison in Node, Python, Go, FSharp and Java

Joseph Ellsworth Feb-2013

This code is fairly simple. It t reads the file. Splits the lines on \n and parses 5 numeric fields out into a output array. This is similar to the logic I use in my Stock trading system to load data. The sample data tested 96964 lines of data totaling 5.96 Meg.

Two approaches where used depending one is an array of structures or array of arrays. The other is a set of parallel arrays each containing all the values for a single column from the CSV. This makes statistical functions which normally work on only 1 column fast because they are moving through contiguous memory. The vector approach seems to be faster to load and faster calculate than the individual records for most languages that support allocating large blocks of contiguous memory such as C and GO.

Of these solutions I have to say that Microsoft F# solution seems like the best combination. It provides a 9.7X slowdown with a 55% reduction in code. The C seems like a viable choice as well but it expands rapidly as soon as re-factoring for re-use. With F# we receive a 9.7 times slowdown for a code reduction of 2.2 or a ratio of 4.4. In python we have a 61.3 times slowdown for a code savings of 2.32 for a ratio of 26.4. In Java we have a code 22.6 slowdown for a code savings of 1.36 for a ratio of 17.4 so it is technically better than the python but way worse than F#. In reality the Java saves very little code compared to C so it isn’t worth the tradeoff. Go may provide the best absolute performance but it’s code is so close to the C in size that it is not worth any degradation and it’s compiler produces such bad error messages that it dramatically slows things down compared to gcc which produces very nice errors.

Some interesting finding:

* The C version loaded 2.5 times faster than the next fastest Java.
* The C version computes averages 25 times faster than Java.
* The Node code was the slowest in calculating the averages. I think this is because it does not have a true array type with pre-allocated contiguous storage and must work harder to compute the location of an array element.
* There is a convenience method in FSharp, Python and Java to read all lines which was used. The vector load version in F# was changed to use .net 4 ieterator. I think I could take out another 100ms by converting this to bytes.
* # Lines of code shown with all comments and blank lines removed

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Total Load Time ms | Compute Average ms | Lines of Code | Code # lines as % of C code | Ratio slower in Avg() than C | Slowdown times / by code times |
| ANSI C – GCC vectors | 372 | 0.31 | 121 | 100% | 1.0 |  |
| Idiomatic C++ |  |  |  |  |  |  |
| Java 7 struct | 1,071 | 13.00 | 95 | 79% | 41.9 |  |
| Java 7 vectors | 1,090 | 7.00 | 89 | 74% | 22.6 | 17.4 |
| GO Struct | 1,835 | 1.39 | 117 | 97% | 4.5 |  |
| GO Vectors | 1,767 | 0.63 | 111 | 92% | 2.0 |  |
| GO modified to C design |  |  |  |  |  |  |
| Python Struct | 1,325 | 25.00 | 41 | 34% | 80.7 |  |
| Python Vector | 1,393 | 19.00 | 52 | 43% | 61.3 | 26.4 |
| F# Struct | 890 | 4.0 | 37 | 31% | 12.9 |  |
| F# Vectors | 905 | 3.0 | 55 | 45% | 9.7 | 4.4 |
| Node.js | 1,337 | 145.00 | 64 | 53% | 467.7 |  |
| D Vectors | 661 | 1.25 | 123 | 102% | 4.0 |  |
| Mozilla Rust |  |  |  |  |  |  |
|  | Smaller is better | Smaller is better | Smaller is better | Smaller is better | Smaller is better | Smaller is better |

## ROI Balancing of compute performance to programming overhead

Assuming that code volume is more expensive to maintain but only to a point where the cost of hardware and hacking on performance begin to outweigh the savings in code. So that if compute time can increase by a factor of 3 for each 100% increase in in code. EG: A program than is 1/3 the length could run 3 times slower and deliver equivalent 2 year production costs. Still need to figure out what an accepted metric is for this. Eventually code performance will offset code reduction because as compute performance degrades larger computers are required and that at some point slower compute time requires more work in distributed architectures which requires more code which offsets the original terseness and that as you approach the limit of available compute power before moving to the next level of distribution you will spend immense time debugging and optimizing which will eventually offset the original performance. The main question is what the real net ROI factor is over a 2 year period. Using a recent project for Search we spent about 375K on computers which in our facility works out to about 850K loaded cost for network, IS overhead etc. At the same time we consumed 5 people for 2 years coding and adding features which at a loaded cost of 150K per person works out to 1.5 million. Using this cost there is a factor of 57% programming labor to hardware ratio. If we assume that code has a linear cost so that 20 lines costs twice as much to maintain as 10 lines then we can afford a factor of where a 57% reduction in code volume should be able to fund a 100% increase in hardware purchased provided you do not lose the savings in late bound optimization trying to obtain adequate performance from the hardware. Of the languages which yielded 43% or greater 43% reduction in code volume the fastest python vectors was 61 times slower.

#### Java 7

**Array of Structures :**

C:\joe\stock\jstrade>timer c:\jdk7\bin\java csv\_bar\_parse\_struct

TEST WITH STRUCT ARR

Size : 6098020

elap read File lines elap= 301

elap Create Empty Bar Arr elap= 1

elap Finished ieterator elap= 767

elap Finished complete load elap= 1071

elap Calc Average elap= 13

average close = 138.09462

0.00user 0.03system 0:01.46elapsed 2%CPU (0avgtext+0avgdata 207360maxresident)k

0inputs+0outputs (841major+0minor)pagefaults 0swaps

**Java 7 Vector arrays**

timer c:\jdk7\bin\java csv\_bar\_parse\_vector

TestWithVectors()

Size : 6098020

elap read File lines elap= 286

elap Create Empty Vector Arrays elap= 3

elap Finished ieterator elap= 798

elap Finished complete load elap= 1090

elap Calc Average elap= 7

average close = 138.09462

0.00user 0.03system 0:01.47elapsed 2%CPU (0avgtext+0avgdata 207616maxresident)k

0inputs+0outputs (842major+0minor)pagefaults 0swaps

* In Java I implemented a set of arrays each representing 1 column of the vector. Similar to how it would have been represented in R.

#### FSharp

* Net Ieterator is quite slow and requires 145 ms to read through and count the lines. But reading the same file in bytes and then processing in byte form to count to count the \n takes only 49ms with sum runs as low as 36ms which much closer to the 31ms required for C to do the same thing. I think the difference is the time .net spends in the Unicode decoder and there is no way to bypass it that I have found so far.
* The original 49 used to calculate the average assumed copying the value out to a vector and then using that value with build in method for average. As it turns out most of this was time was spent in building the vector. Only 3ms seems to be used in the actual average calculation which is much better than the 49ms.
* Modified the average calculation to also try iterating across the tuples and found that it also ran in 3ms so in F# there is not a clear benefit by using the single vector strategy like there is in most other languages.

C:\joe\stock\jstrade>timer ./ex-read\_and\_split.exe

Number of Lines= 96963

read file and split =191

parse and convert=862

Calv Avg=49

total Load time=1099

read file as bytes =11

0.00user 0.03system 0:01.30elapsed 2%CPU (0avgtext+0avgdata 208128maxresident)k

0inputs+0outputs (842major+0minor)pagefaults 0swaps

Turns out that the main cost in F# to calculate he array was to convert the tuples into an array of float against which the Average can be applied. The actual average statement was only 3 milliseconds.

\*\*\* Modified to allow decimals in time calc

C:\joe\stock\tests\ex-parse-bar>timer .\ex-read\_and\_split\_struct.exe

Parse and Convert elap=890.048

Number of Lines= 96963

Calc average from struct elap=4.0064

0.00user 0.03system 0:01.04elapsed 2%CPU (0avgtext+0avgdata 208384maxresident)k

0inputs+0outputs (836major+0minor)pagefaults 0swaps

\*\*\* Modified to support pre-created vectors \*\*\*

C:\joe\stock\tests\ex-parse-bar>timer ./ex-read\_and\_split.exe

parse bytes to count 96964 lines elap=29.0048

read file as bytes = elap=8.9984

Count Lines Bytes All = elap=33.9968

Parse and Convert elap=880.0512

Number of Lines= 96964

Calc Average = 138.093186605336 from Array vector elap=2.9952

Average of 1,000 avg runs=3.0731776

done

0.00user 0.01system 0:04.12elapsed 0%CPU (0avgtext+0avgdata 208128maxresident)k

0inputs+0outputs (837major+0minor)pagefaults 0swaps

#### GO

**Load as an array of structures**

timer ./csv\_bar\_parse\_struct.exe

fiName=c:/JOE/stock/JTDATA/symbols/SPY/2012.M1.csv

ELAP File Read = 11.001 ms

ELAP Convert string to bytes = 8.001 ms

ELAP Split String to rows = 124.007 ms

ELAP Allocate Bars = 0.000 ms

ELAP Make Records = 1688.097 ms

ELAP total File Load = 1835.105 ms

ELAP Calculate Avg() = 2.000 ms

Avg = 138.09

Run 100 avg() =139.01 each=1.390

0.01user 0.03system 0:02.12elapsed 2%CPU (0avgtext+0avgdata 207872maxresident)k 0inputs+0outputs (844major+0minor)pagefaults 0swaps

**GO Vectors**

Go modified for parallel vector array with parsing following the same model used by C program.

C:\joe\stock\gotrade>timer ./csv\_bar\_parse\_vector.exe

fiName=c:/JOE/stock/JTDATA/symbols/SPY/2012.M1.csv

ELAP File Read = 10.001 ms

ELAP Convert string to bytes = 6.000 ms

ELAP Split String to rows = 94.006 ms

ELAP Allocate Vectors = 0.000 ms

ELAP Make Records = 1653.094 ms

ELAP total File Load = 1767.101 ms

ELAP Calculate Avg() = 0.000 ms

Avg = 138.09

Run 100 avg() =63.00 each=0.630

0.00user 0.00system 0:01.95elapsed 0%CPU (0avgtext+0avgdata 207872maxresident)k

0inputs+0outputs (844major+0minor)pagefaults 0swaps

#### Python

**Array of Arrays instead of strcuct.**

time to read pre-split elap= 95.0

Parse Lines elap= 1219.0

Average close elap elap= 25.0

Total Run time elap= 1325.0

* Python and Node allow different types of data in arrays so I used that where each row is an array instead of a struct or object. The true struct construction was much slower. In GO, Java and C this is not allowed.

**Python Modified for Vectors**

timer python testParseBarFileVector.py

time to read pre-split elap= 105.0

Slice Marr to just data elap= 5.0

out contains 96963 records

ieterate array and split elap= 1276.0

Total Run time elap= 1393.0

Average close elap elap= 19.0

close avg= 138.09

0.01user 0.03system 0:01.64elapsed 2%CPU (0avgtext+0avgdata 207360maxresident)k 0inputs+0outputs (840major+0minor)pagefaults 0swaps

#### Node JS

**Parse to array of array similar to struct**

C:\joe\stock\jstrade>timer "c:\Program Files (x86)\nodejs\node" csv\_bar\_parse\_st

ruct.js

number of lines=96964

Elapsed Read and Split=102

BuildParsedArray=1090

calcAvgOpen=145

total Run= 1337

calculated average open=138.09454669306757

0.01user 0.01system 0:01.68elapsed 1%CPU (0avgtext+0avgdata 215552maxresident)k

0inputs+0outputs (881major+0minor)pagefaults 0swaps

**Use a Set of vectors pre created**

**instead of sub array object not**

Not shown in table because performance was so bad.

Elapsed Read = 39

Elapsed Split= 58

pre-alloc array = 0

BuildParsedArray = 1424

total Run = 1521

* I was surprised Node.JS the time to calculate the average was so much higher than the cost to do the same thing in GO than FSharp and Python. . I expected node to blow python away in a simple ieterate and add loop.
* Pre-allocation in Node was no faster than incremental push which does save time in GO.
* All implementations used the most simple method available to read the full contents of the file into a string because it generally tests faster than readln()
* I could probably make the GO a bit faster by directly parsing the lines from the input byte array.

### ANSI C – GCC

$ timer ./ex\_parse\_bar

ex\_parse\_bar.cRead in 50K blocks elap = 15.00

Read 122 of 50k blocks

Read 96963 lines

allocate vector array elap = 16.00

build Recs elap = 343.00

Total Load elap = 374.00

calc average 100 times 31.00 each= 0.31

avg of CLOSE = 138.09

cleanup elap = 0.00

0.40user 0.06system 0:00.49 elapsed 93%CPU (0avgtext+0avgdata 601088 maxresident)k0inputs+0outputs (3169major+0minor)pagefaults 0swaps

This one is the more common run. Note the 31 ms in reading to count the lines.

jwork@Owner-THINK /cygdrive/c/joe/stock/ctrade

$ ./csvbar

Read in 50K blocks elap = 31.00

Read 96964 lines

allocate vector array elap = 0.00

build Recs elap = 312.00

Read 96962 numRec

Total Load elap = 343.00

tavg= 138.09

calc average 1000 times each= 0.3740

cleanup elap = 0.00

* For some reason the memory allocation for the C version is showing in timer as higher than it really is. I modified it with a while(1) to force it to remain resident before freeing up any memory and taskmgr showed it using 5.4 meg which is not too bad for a 5.9 meg input file. When using doubles instead of float it used 6.9 meg.
* For sake of convenience the fgets method was used in the main parsing loop. This is known to add about 90ms to this loop compared to the block read which would allow a build recs runtime closer to 343 – 90 = 253 which would be 1.36 times faster.
* My style preference for Java and C is to have { on next line. Go and Node specify { on same line as block open statement. The C was changed to Go standard for consistency to obtain 121 lines. The C code includes a custom function copyNext which is slightly faster than strtok. If not used it would have reduced C code by 11 lines which would have brought it’s size down to 110 lines which is shorter than either of the GO implementations.
* Compiled with g++ -O3 -march=native -flto -fwhole-program ex\_parse\_bar.c -o ex\_parse\_bar but similar results when using –O2 option. When running without the –O2 or –O3 it seems to run The average is 1.4 which is 4.5 times slower and is slower than GO vectors.
* In C we could cycle through byte buffer is so fast that I chose loop through the file once counting \n s I can determine how many rows of data are available. This is because for a data set this size the speed from cache was faster than memory allocation overhead required to read one rec at a time. This allows pre-allocation of the total vector array without the memory overhead of reading the full string into memory. It also removes the time to split the string into an array of lines which saves additional memory management overhead. Once in the main build loop the C code does not use any additional dynamic memory where all other implementations use string.split() for the line to create the 7 item vector prior to doing the numeric conversion. This saves roughly 676,000 temporary allocations. With the speed of the GO loop this may be viable in GO to.
* In C rather than use a string.split I use a single buffer into which we copy each sub field and then use that field to feed into atof and atoll. As result I avoid all the memory overhead of making new arrays.
* The compute average in C was so fast that it was below resolution of the clock So I ran it 100 time to get a large enough sampling to average.

### D Vectors

C:\joe\stock\ex-parse-bar>timer .\exparsebar

Read in 50K blocks elap = 127.86

Read 121 of 50k blocks

Read 96963 lines

allocate vector array elap = 16.71

build Recs elap = 516.45

Read 96962 numRec

Total Load elap = 661.15

calc average elap = 1.28

calc average 100 times 124.87 each= 1.25

cleanup elap = 0.09

0.03user 0.01system 0:00.90elapsed 5%CPU (0avgtext+0avgdata 208896maxresident)k

0inputs+0outputs (848major+0minor)pagefaults 0swaps

### Input format

Input File has 96,983 lines of CSV of roughly the following format.

dateTime,day, open, close, high, low, volume

2013-01-02 09:30:00.00,Wed,145.11,145.26,145.39,145.09,13889360

2013-01-02 09:40:00.00,Wed,145.25,145.215,145.3,144.935,9720625

2013-01-02 09:50:00.00,Wed,145.21,145.45,145.58,145.21,8063553

2013-01-02 10:00:00.00,Wed,145.46,145.32,145.59,145.26,6709568

# Blog Summary

I love the new auto set / get properties.    Now add short circuit support with break and continue and I will be happy.   It would also  be nice if  the Mono runtime was fixed so it isn't 3 to 5 times slower than the Msoft stack.

I recently ran extensive tests parsing massive amounts of stock data for a stock analytic system where I use the data in back traces.    F# was slower than C but it was  faster than Java.    GO beat F# for tight statistics calculations but was slower for loads and parsing.     
  
To obtain adequate performance I had to violate the idiomatic approach of passing slices to the statistics functions.   The original F# implementation was 70 times slower than C. I had almost decided to port the entire system to C but decided to test a direct port of the optimized C design to F#.  I used C optimization techniques such as large alloc of contiguous vectors, Using parallel vector view instead of arrays of records . The F# only grew by 8% in code volume but moved 70 times slower than C to one of the fastest at only 9 times slower than C.   I think it is OK to violate some of the purity of F# to make it more competitive as a general purpose compute tool.  In this vein some of the optimization techniques I used to make F# outperform Java would have been easier, cleaner and easier to maintain if break, continue and early return where directly supported.   
  
Our calculations of SMA,EMA and Stochastic require thousands of computations over rolling windows of the of the original vectors.   The F# slice currying techniques delivered poor performance but when you pass a pointer to the original array along with begin and end indexes it actually was only 9 times slower than C which made if faster than our best Java port that 22 times slower.      Only C,GO and D where faster than F# in the tight loop when passing in the entire vector pointer along with begin and end indexes.    GO supplies a unique feature for slices which creates a virtual slice think of it as a begin and end pointer which but does not require any data copying.  It seems that F# could borrow that same approach to deliver some interesting performance benefits.    
  
I found F# code easy to switch back and forth between the easy idiomatic approach and the more efficient imperative approach where it is quite fast.   This allows rapid prototyping with low cost optimization.   Either way the code remained 20 to 25% the length of the code volume compared to well factored C code.    
  
I it found interesting that F# while being concise still responded very well to the optimization techniques used in C to improve performance.     In contrast when I tried to bring the same techniques into Node.js (javascript) and Python I either saw no benefit or actually encountered a slowdown.

I love the new auto set / get properties.    Now add short circuit support with break and continue and I will be happy.   It would also  be nice if  the Mono runtime was fixed so it isn't 3 to 5 times slower than the Msoft stack.

I recently ran extensive tests parsing massive amounts of stock data for a stock analytic system where I use the data in back traces.    F# was slower than C but it was  faster than Java.    GO beat F# for tight statistics calculations but was slower for loads and parsing.     
  
To obtain adequate performance I had to violate the idiomatic approach of passing slices to the statistics functions.   The original F# implementation was 70 times slower than C. I had almost decided to port the entire system to C but decided to test a direct port of the optimized C design to F#.  I used C optimization techniques such as large alloc of contiguous vectors, Using parallel vector view instead of arrays of records . The F# only grew by 8% in code volume but moved 70 times slower than C to one of the fastest at only 9 times slower than C.   I think it is OK to violate some of the purity of F# to make it more competitive as a general purpose compute tool.  In this vein some of the optimization techniques I used to make F# outperform Java would have been easier, cleaner and easier to maintain if break, continue and early return where directly supported.   
  
Our calculations of SMA,EMA and Stochastic require thousands of computations over rolling windows of the of the original vectors.   The F# slice currying techniques delivered poor performance but when you pass a pointer to the original array along with begin and end indexes it actually was only 9 times slower than C which made if faster than our best Java port that 22 times slower.      Only C,GO and D where faster than F# in the tight loop when passing in the entire vector pointer along with begin and end indexes.    GO supplies a unique feature for slices which creates a virtual slice think of it as a begin and end pointer which but does not require any data copying.  It seems that F# could borrow that same approach to deliver some interesting performance benefits.    
  
I found F# code easy to switch back and forth between the easy idiomatic approach and the more efficient imperative approach where it is quite fast.   This allows rapid prototyping with low cost optimization.   Either way the code remained 20 to 25% the length of the code volume compared to well factored C code.    
  
I it found interesting that F# while being concise still responded very well to the optimization techniques used in C to improve performance.     In contrast when I tried to bring the same techniques into Node.js (javascript) and Python I either saw no benefit or actually encountered a slowdown.

The short answer is think about how you would optimize the algorithm to maximize speed in C and then try mimic that as close as possible in F#. Just like Java and C#, Object allocation, destruction and currying into different shapes in order to pass as a parameter is time consuming.

I recently ran extensive tests parsing massive amounts of stock data for analytic system F# started out close the slowest of the tested languages when using pure idiomatic F# as taught in the books. When the F# was optimized to follow a design closer to carefully optimized C it moved from very slow to the 2nd fastest. The F# code only grew about 8% in the change but performance improved by 770%. I think F# still delivered a benefit because it still saved 75% of the code volume.

To obtain adequate performance I had to violate the idiomatic immutable approach of passing slices to the statistics functions. The original F# implementation was 70 times slower than C. After optimizing to closely mimic the C design it was only 9 times slower which is pretty good compared to our best Java implementation which was 22 times slower. I used C optimization techniques such as large alloc of contiguous vectors and using parallel vector view of the data instead of arrays of records. Ultimately the key was passing pointers to arrays with begin and end indexes rather than currying sliced views.

Ultimately only C,GO and D where faster in tight statistics computation methods while F# was faster than Java, Python and Node.js (javascript).