Paper 256-2013

Optimize SAS/IML Codes for Big Data Simulation

Chao Huang. Oklahoma State University

Yu Fu. Oklahoma State University

Goutam Chakraborty. Oklahoma State University

ABSTRACT

Data volume keeps growing dramatically in the past decade. At a number of occasions, simulation also creates large vectors and matrices. To speed up the processing of large datasets, vectorization is a popular optimization skill for many matrix languages, such as R, Matlab® and SAS/IML®. In this paper, a few examples in SAS/IML will be illustrated to introduce the vector-wise operation and other optimization strategies. Concerns regarding the computer memory are addressed to accommodate big vectors or matrices in SAS/IML. The result shows that optimized SAS/IML codes will significantly improve the computing efficiency and greatly simply the coding effort.

INTRODUCTION

SAS/IML is the matrix language module in SAS, and also a common toolbox for scientific simulation. Together with DATA Step and other SAS procedures, the IML procedure provides SAS users with a dynamic and interactive environment for matrix operation. Through our practice, we find that using some optimization techniques including rewriting codes in SAS/IML.to avoid loops will make the codes faster and simpler, and even achieve dramatic speedup. In addition, recently SAS/IML introduces a few new vector-wise subscripts, operators and functions for code optimization [1]. In this paper, we introduce our exploration toward the memory management in SAS/IML, and three examples about vectorization versus looping in the pursuit of computation efficiency increase. The examples are run on a desktop computer installed with Windows XP and SAS 9.3. And this PC has an Inter® Core 2 Duo CPU and 3GB memory.

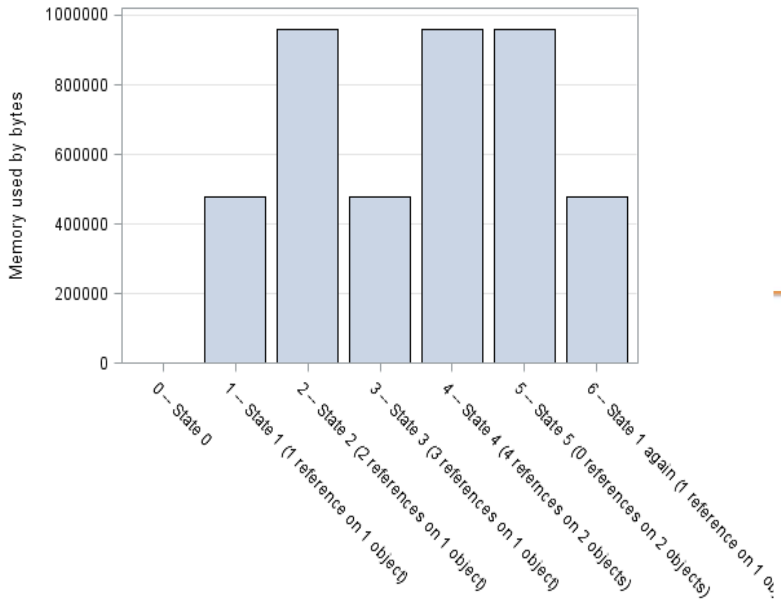
USE MEMORY CAUTIOUSLY IN SAS/IML

We design an experiment to check the behavior of SAS/IML's memory manager. In SAS/IML, usually a matrix of 200 rows and 300 columns occupies about 400kB memory. However, at the beginning, we only request 1MB memory from the system by specifying the WORKSIZE option. Thus, given 1MB memory, three matrices of such size would cause a stack overflow and therefore the failure of PROC IML. Firstly we assign two (State 2) or three references (State 3) toward a single random matrix, and secondly change the value of only one element in this matrix (State 4). Finally we use the CLEAR statement to clear all of the objects (State 5), and re-generate another matrix with the same size (State 6).

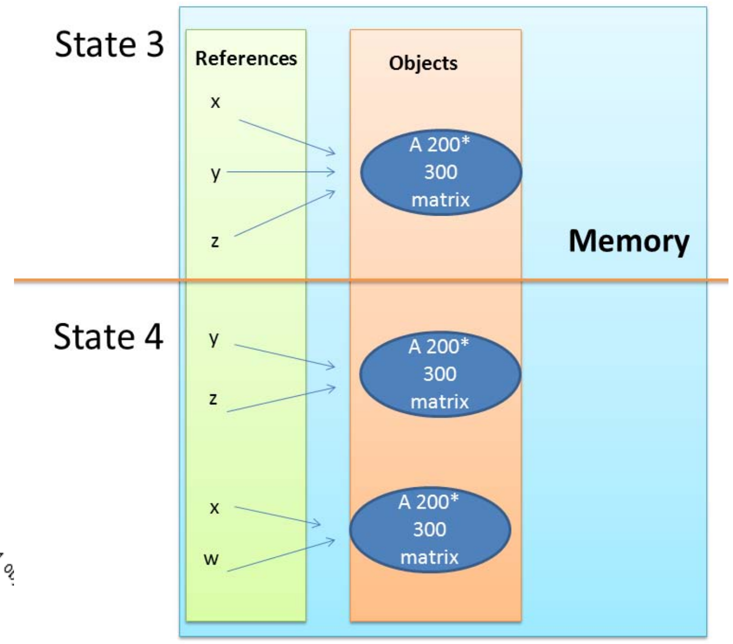
```
proc iml worksize=1048576; /* Only request 1MB memory for this test*/
    /* 0 -- State 0*/
     show space;
    /* 1 -- State 1 (1 reference on 1 object)*/
        x = j(200, 300, 0);
        call randgen(x, "Normal");
     show space;
    /* 2 -- State 2 (2 references on 1 object)*/
       y = x;
    show space;
    /* 3 -- State 3 (3 references on 1 object)*/
```

```
         z = x;
      show space;
      /* 4 -- State 4 (2 references on 1 object and 2 on another object)*/
         x[1, 1] = 5;
         w = x;
      show space;
      /* 5 -- State 5 (0 references on 2 objects) */
         free x y z w;
      show space;
      /* 6 -- State 1 again (1 reference on 1 object)*/
         x = j(200, 300, 0);
         call randgen(x, "Normal");
      show space;
quit;
```

The memory change at the seven states from State 0 to State 6 is displayed in Figure 1. If there is no object such as vector or matrix, the memory usage is zero at State 0. A 200 by 300 random matrix takes a little more than 400kB memory at State 1. Interestingly, an object with two references costs two times memory at State 2 than the same object with three references at State 3. Apparently with the two references (x and y), two identical copies of the matrix exist at State 2. If we increase the number of reference to 3, then all the three references (x, y and z) point to a single matrix at State 3. Although there is only one element different, a new matrix is created along with the initial matrix in the memory at State 4. The original matrix is referred by y and z, and the modified one is referred by x and w (Figure 1B). Running the FREE Statement doesn't immediately clear the memory at State 5. The unreferenced objects are eventually discarded during next operation at State 6 (Figure 1A).



A



B

Figure 1. Memory usage by PROC IML at different stages. (A) The memory costs by byte at each of the seven stages. (B) The comparison between 3 references on 1 object and 4 references on 2 objects.

Thus, we conclude that the memory manager of IML first checks the size available of the workspace or the memory allocated to IML. If the memory usage doesn't pass the alarming line, it will generously make copies. Otherwise, it will become very aggressive and compress the memory frenetically. SAS/IML follows the rule of copy-on-change. Once the value of any element in a matrix is changed, the memory manager will make a copy for the original object. At last, SAS/IML's FREE statement kills the references instead of the objects themselves.

STRATEGIES THAT OPTIMIZE SAS/IML

VECTORIZATION VERSUS DO LOOP TO SQUARE NUMBERS

To compare the efficiencies between the looping codes and the vectorized IML codes, we apply PROC IML to square every element in a vector from 1 to 50,000, and calculate the time spent by this operation. To obtain replications, two user-defined modules are created in PROC IML and repeatedly used to square those numbers for 100 times. System time spent is recorded by PROC IML's TIME function.

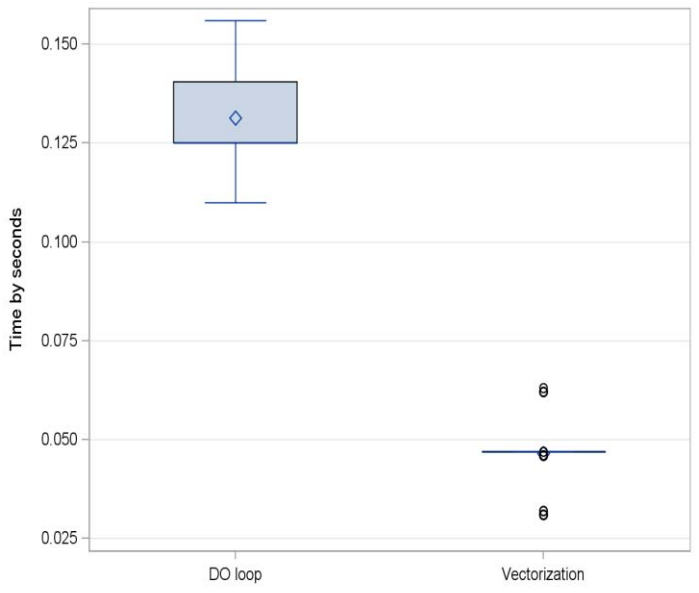
```
proc iml;
   t = j(100, 2, 1);
   /* 1 -- Build the modules*/
   start module1;
      result1 = j(5e5, 1, 1);
      do i = 1 to 5e5;
         result1[i] = i**2;
      end;
   finish;
   start module2;
      result2 = t(1:5e5)##2;
      store result2;
   finish;
   /* 2 -- Run the tests*/
   do n = 1 to 100;
      t0 = time();
         call module1;
      t[n, 1] = time() - t0;
      t0 = time();
         call module2;
      t[n, 2] = time() - t0;
   end;
   /* 3 -- Output the timing dataset*/
   create time_data from t;
      append from t;
   close time_data;
quit;
```

Then in Figure 2A, the timing dataset from PROC IML is visualized by box plots with the SGPLOT procedure. Overall, the vectorizatized codes perform much faster than a DO loop in PROC IML.

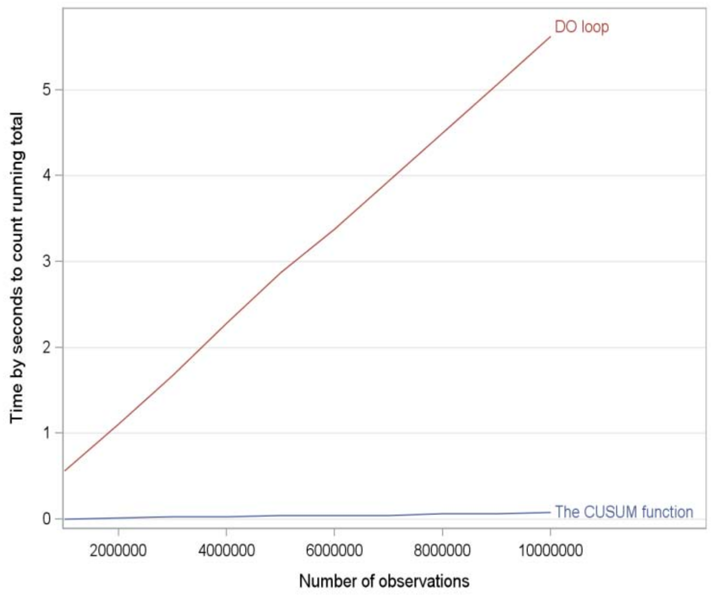
```
data report_data(keep= test time);
   set time_data;
   length test $25.;
   test = "DO loop"; time = col1; output;
   test = "Vectorization"; time = col2; output;
run;

proc sgplot data = report_data;
   vbox time / category = test;
   yaxis grid label = 'Time by seconds';
   xaxis label = ' ';
run;
```

CALCULATE RUNNING TOTAL BY EITHER THE DO LOOP OR THE CUSUM FUNCTION

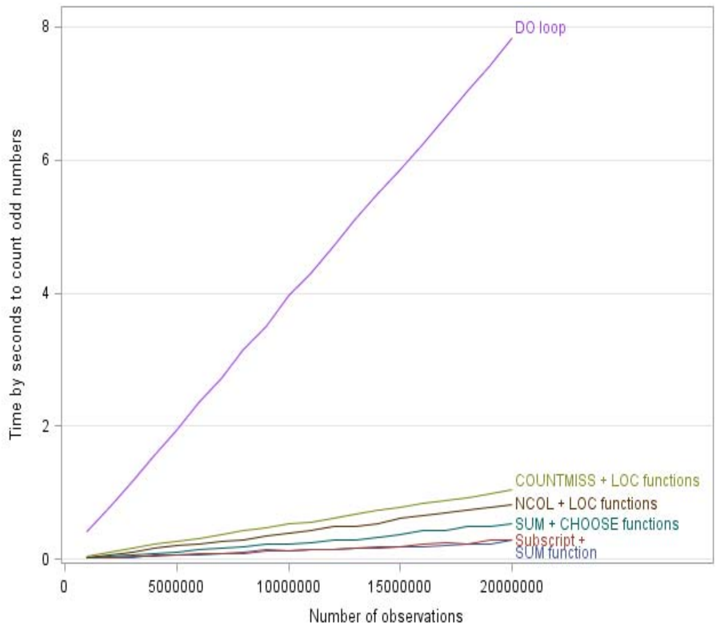
PROC IML can generate a new variable for the running total by either a DO loop or the CUSUM function. We set up an experiment to attain the running totals for a serial of random vectors from 1 million to 10 million elements with the step size of 1 million.



A



B



C

Figure 2. Performance comparisons between the looping method and the vectorization methods. (A) The example that squares number for random vectors. (B) The example that calculates running total for random vectors. (C) The example that counts odd numbers for random vectors.

Figure 2B shows that the CUSUM function is at least 100 times faster than the looping structure. With the increase of the vector size from 1 million to 10 million, the gap between the time costs of the two methods turns even wider.

```
proc iml;
   a = t(do(1e6, 1e7, 1e6));
   timer =  j(nrow(a), 2);
   do p = 1 to nrow(a);
      n = a[p];
      z = t(ranuni(1:n));
      t0 = time();
          x = cusum(z);
      timer[p, 1] = time() - t0;
      y = j(n, 1, .);
      y = z;
      t0 = time();
         do i = 2 to n;
            y[i] = y[i-1] + z[i];
         end;
      timer[p, 2] = time() - t0;
   end;

   t =  a||timer;
   create time_data from t;
      append from t;
   close time_data;
quit;

data report_data;
   set time_data;
   length test $100.;
   label col1 = "Number of observations"
   time = "Time by seconds to count running total";
   test = "The CUSUM function"; time = col2; output;
   test = "DO loop"; time = col3; output;
   keep test time col1;
run;

proc sgplot data = report_data;
   series x = col1 y = time / curvelabel group = test;
   yaxis grid;

run;
```

COUNT ODD NUMBERS FOR A RANDOM VECTOR BY SIX METHODS

Currently PROC IML provides a few vector-wise functions and operators [2]. To test their potentials in IML code optimization, we design an experiment to count how many odd numbers a random vector contains by 6 different methods. These methods include the SUM function, the "+" subscript, the combination between the SUM function and the CHOOSE function, the combination between the NCOL function and the LOC function, the combination between the COUNTMISS function and the LOC function, the combination and a DO looping structure with a counter variable as benchmark.

In those functions, the SUM function aggregates the indicator of an odd number by the MOD function all at once, which is equal to how many odd numbers a random vector has. SAS/IML's "+" subscript serves the same purpose. The CHOOSE function plays a role as the IF-ELSE-THEN statement for binary selection. The LOC function can subset a vector like the WHERE statement, while the NCOL function finds the number of columns. At last, the COUNTMISS records the number of missing elements.

```
proc iml;
```

```
   a = t(do(1e6, 2e7, 1e6));
   timer = j(nrow(a), 6);
   do p = 1 to nrow(a);
      n = a[p];
        /* Simulate a numeric sequence */
      x = ceil(ranuni(1:n)*100000);
      /* 1 -- SUM function*/
      t0 = time();
         r1 = sum(mod(x, 2));
      timer[p, 1] = time() - t0;
      /* 2 -- Subscript + */
      t0 = time();
         r2 = mod(x, 2)[ , +];
      timer[p, 2] = time() - t0;
      /* 3 -- SUM + CHOOSE functions*/
      t0 = time();
         r3 = sum(choose(mod(x, 2), 1, 0));
      timer[p, 3] = time() - t0;
      /* 4 -- NCOL + LOC functions */
      t0 = time();
         r4 = ncol(loc(mod(x, 2) = 1));
      timer[p, 4] = time() - t0;
      /* 5 -- DO loop */
      t0 = time();
         r5 = 0;
         do i = 1 to ncol(x);
            if mod(x[i], 2) = 1 then r5 = r5 + 1;
         end;
      timer[p, 5] = time() - t0;
      /* 6 -- COUNTMISS + LOC functions */
      t0 = time();
         x[loc(mod(x, 2) = 1)] = .;
         r6 = countmiss(x);
      timer[p, 6] = time() - t0;
   end;
   /* 7 -- Validate the results */
   print r1 r2 r3 r4 r5 r6;
   t = a||timer;
   create time_data from t;
      append from t;
   close time_data;
quit;
```

The results are illustrated in Figure 2C. The CHOOSE function can replace the IF-ELSE-THEN statements for simple conditions and is far more efficient. In addition, the LOC function proves to be a convenient vector-wise filter. And the SUM function performs slightly faster than the "+" subscript in this example. In general, the more straightforward the method is, the faster the computation is. Most importantly, all vector-wise methods beat the DO loop, especially for the vector that has many elements. In general, de-looping the IML codes would significantly improve the efficiency.

```
data report_data;
   set time_data;
   length test $100.;
   label col1 = "Number of observations"
      time = "Time by seconds to count odd numbers";
   test = "SUM function"; time = col2; output;
   test = "Subscript + "; time = col3; output;
   test = "SUM + CHOOSE functions"; time = col4; output;
   test = "NCOL + LOC functions"; time = col5; output;
   test = "DO loop"; time = col6; output;
   test = "COUNTMISS + LOC functions"; time = col7; output;
   keep test time col1;
run;

proc sgplot data = report_data;
   series x = col1 y = time / curvelabel group = test;
   yaxis grid;
run;
```

CONCLUSION

We could make better use of SAS/IML by avoiding unnecessary looping and applying the vector-wise functions and operators, checking memory usage by the SHOW SPACE statement, and testing the codes' efficiency by its built-in TIME function.

REFERENCES

1. Rick Wicklin, "Statistical Programming with SAS/IML Software". SAS Publishing. 2011
2. Rick Wicklin. "SAS/IML tip sheets". The DO Loop. October 10, 2011
   http://blogs.sas.com/content/iml/files/2011/10/IMLTipSheet.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Chao Huang
Office of Institution Research and Information Management
Oklahoma State University
221 PIO Building
Stillwater, OK. 74075
Email: hchao8@gmail.com
Web: www.sasanalysis.com

Yu Fu
Department of Management Science and Information Systems
Oklahoma State University
Stillwater, OK. 74075
Email: yu.fu@okstate.edu

Goutam Chakraborty
Department of Marketing
Oklahoma State University
Stillwater, OK. 74075
Email: goutam.chakraborty@okstate.edu