Paper 002-30

Efficiency Considerations Using the SAS® System

Rick Langston, SAS Institute Inc., Cary, NC

ABSTRACT

There are many products being released by SAS Institute that either use an abundance of SAS code, or generate SAS code, or both. We have begun an internal process to assist product developers in reviewing this SAS code for issues of efficiency, host compatibility, performance, and so on. We also realize that these issues are of interest and benefit to all SAS programmers. This paper is a collection of some of our initial performance analyses in writing efficient SAS code.

INTRODUCTION

This paper examines a few of the most common questions regarding efficient SAS code. Its main focus is to examine CPU usage differences in the areas listed below. Except where noted, all examples were tested on production versions of SAS on UNIX platforms.

- IF statements versus WHERE statements in DATA steps
- IF statements versus SELECT statements
- PROC SORT versus SQL ORDER BY
- Issues involving DATA step views
- Functions versus arrays versus direct SAS coding
- DATA step stored programs
- Threaded versus non-threaded performance for PROC SUMMARY
- Table lookup comparing PROC FORMAT, merging, and the hash table object

It is important to understand that the real times and CPU times shown in the logs in this paper are not necessarily indicative of the times you may see if you try the same tests. These times vary greatly between operating systems and between machine configurations. The times are used here primarily for purposes of comparison to help you to make decisions on SAS coding practices.

IF OR WHERE

A common question regarding efficiency is "Which do I use: IF or WHERE?" These examples compare the performance of three different methods of subsetting a third of the observations in a 500,000-observation data set.

SUBSETTING IF STATEMENT

```
             data temp; set mydata.x500000;
                  if mod(obs,3)=0;
                  run;

   NOTE: There were 500000 observations read from
         the data set MYDATA.X500000.
   NOTE: The data set WORK.TEMP has 166666 observations
         and 11 variables.
   NOTE: DATA statement used:
         real time           8.44 seconds
         cpu time            1.73 seconds
```

WHERE CLAUSE

```
             data temp; set mydata.x500000;
                  where mod(obs,3)=0;
                  run;

   NOTE: There were 166666 observations read
         from the data set MYDATA.X500000.
         WHERE MOD(obs, 3)=0;
   NOTE: The data set WORK.TEMP has 166666
         observations and 11 variables.
```

```
NOTE: DATA statement used:
      real time           8.66 seconds
      cpu time            2.18 seconds
```

WHERE SPECIFICATION IN PARENTHETICAL OPTIONS

```
          data temp; set mydata.x500000(where=(mod(obs,3)=0));
               run;

NOTE: There were 166666 observations read
      from the data set MYDATA.X500000.
      WHERE MOD(obs, 3)=0;
NOTE: The data set WORK.TEMP has 166666 observations and 11 variables.
NOTE: DATA statement used:
      real time           8.89 seconds
      cpu time            2.18 seconds
```

In the three previous examples, the subsetting IF statement is slightly faster, but not significantly so. Interestingly, the WHERE statement is a little faster than the parenthetical WHERE statement.

Performance in these examples is close enough that the choice of an IF statement versus a WHERE statement versus a WHERE option is arbitrary. However, if the subsetting is to be performed using a LIBNAME engine against a database that is optimized for WHERE processing, then the WHERE choice is preferred. A subsetting IF statement requires all observations to be retrieved from the database, with the DATA step code determining whether the observation is to be kept. Using a WHERE statement in these circumstances retrieves only those observations that match the criteria for subsetting.

IF-THEN/ELSE OR SELECT

To generate a significant number of IF-THEN/ELSE statements and SELECT/WHEN statements, we use the following SAS code to create programs with 100 clauses of each type. The code assigns a random value to y based on increasing values of x. The file SASCODE1 will contain IF-THEN statements, and the file SASCODE2 will contain SELECT/WHEN statements. SASCODE1 and SASCODE2 are both TEMP files that are referenced by a %INCLUDE statement when each DATA step program is run.

```
filename sascode1 temp;
filename sascode2 temp;
data _null_;
     file sascode2; put 'select(x1);';
     do i=1 to 100;
        j=ranuni(43431);
        file sascode1;
        if i>1 then put 'else ' @;
        put 'if x1=' i ' then ' j= ';';
        file sascode2;
        put 'when(' i ')' j= ';';
        end;
     file sascode2; put 'otherwise; end;';
     run;
```

IF-THEN/ELSE STATEMENTS

```
19         data temp; set mydata.x500000;
20              %include sascode1/source2;
21        +if x1=1  then j=0.1093699425 ;
22        +else if x1=2  then j=0.9230094435 ;
23        +else if x1=3  then j=0.7694377665 ;
          ...
119       +else if x1=99  then j=0.3876921946 ;
120       +else if x1=100  then j=0.9119509677 ;
121             run;
...
NOTE: DATA statement used:
      real time           9.17 seconds
      cpu time            2.61 seconds
```

SELECT/WHEN STATEMENTS

```
122        data temp; set mydata.x500000;
123             %include sascode2/source2;
124       +select(x1);
125       +when(1 )j=0.1093699425 ;
126       +when(2 )j=0.9230094435 ;
...
223       +when(99 )j=0.3876921946 ;
224       +when(100 )j=0.9119509677 ;
225       +otherwise; end;
226             run;
...
NOTE: DATA statement used:
      real time           9.08 seconds
      cpu time            2.51 seconds
```

The SAS log shows that 100 IF-THEN/ELSE statements took 2.61 CPU seconds to run, while the corresponding SELECT statement with 100 WHEN clauses took a little less time, CPU 2.51 seconds.

The SELECT statement is only slightly better in this example. WHEN clauses can have scalar values only, whereas IF-THEN/ELSE statements don't have that limitation.

Remember that the ELSE clause makes a big difference in IF-THEN/ELSE performance. If the ELSE clause had been omitted here, the example would have run much slower. Consider another example:

```
if x=1 and y=2 then z=3;
else if x=3 and y=4 then z=7;
```

In the first case, if x=1 and y=2, then it is not possible for x=3 and y=4, so it is appropriate to provide an ELSE clause. If the ELSE clause were omitted, and the condition of x=1 and y=2 were true, then the execution of the second IF statement would be unnecessary and cause additional execution time.

But consider this example:

```
if x=1 and y=2 then z=3;
if z=3 and q=7 then z=7;
```

The conditions in the two IF statements are not mutually exclusive. Therefore, adding an ELSE clause might cause z to not be properly set. You must examine your specifications carefully when using IF-THEN/ELSE statements.

Remember also that you can't always substitute a SELECT/WHEN statement for an IF-THEN/ELSE statement. Here is an example:

```
if x=1 and y=2 then z=3;
else if x=3 and y=4 then z=7;
```

This expression cannot be represented using a simple SELECT statement. You could use the following scenario, which has an implied ELSE clause, but it has no advantage over the IF-THEN/ELSE statement.

```
 select(1);
   when(x=1 and y=2) do; z=3; end;
   when(x=3 and y=4) do; z=7; end;
   end;
```

SORT OR SQL

This example demonstrates the performance difference between using SQL to sort a data set and using the traditional PROC SORT approach. Examples show three ways to create a data set TEMP1 with the variables obs and x3, to sort by those variables, and then to subset.

DATA STEP AND PROC SORT

```
6          data temp1; set mydata.x500000(keep=obs x3);

NOTE: DATA statement used:
      real time           8.31 seconds
      cpu time            1.54 seconds

7          proc sort; by x3 obs;

NOTE: PROCEDURE SORT used:
      real time           4.62 seconds
      cpu time            4.53 seconds
```

PROC SORT WITH KEEP OPTION

```
8          proc sort data=mydata.x500000(keep=obs x3)
9               out=temp2; by x3 obs;

NOTE: PROCEDURE SORT used:
      real time           14.41 seconds
      cpu time            5.67 seconds
```

The execution time is slightly longer for the multi-step process. You might find it faster to combine the subsetting and sorting into a single step.

PROC SQL WITH ORDER BY CLAUSE

```
11         proc sql;
12              create table temp3 as
13                 select obs,x3 from mydata.x500000
14                 order by x3,obs;
15              run;
16

NOTE: PROCEDURE SQL used:
      real time           16.33 seconds
      cpu time            7.96 seconds
```

Here we perform the same subsetting in PROC SQL using SELECT ... FROM and perform the sorting via the ORDER BY clause. The execution time is greater, just under eight seconds.

Deciding whether to use PROC SQL depends partly on what engine is involved. SAS/ACCESS engines might be able to pass SQL code directly to the database for faster execution. PROC SORT can be very competitive in its sorting techniques. Often you will either be intimately familiar with SAS code and choose PROC SORT, or you will choose SQL based on that familiarity.

USING DATA STEP VIEWS

DATA step views can be handy, but you must proceed with caution when using them. Using the DATA step to create a new data set might be less costly in terms of performance because the user of the view might not be aware of the code that is used to define the view. In the following examples, one DATA step reads a view that is created via the INFILE and INPUT statements. The other DATA step creates a permanent SAS data set.

DATA STEP VIEWS

```
1          data fromfile/view=fromfile;
2               infile 'mydata.dat';
3               input x y z;
4               run;
...
5
6          data new; set fromfile;
7               run;

NOTE: View WORK.FROMFILE.VIEW used:
      real time           5.69 seconds
      cpu time            3.94 seconds

NOTE: DATA statement used:
      real time           5.72 seconds
      cpu time            1.62 seconds
```

SAS DATA SET

```
9          data new;
10              infile 'mydata.dat';
11              input x y z;
12              run;

NOTE: DATA statement used:
      real time           4.24 seconds
      cpu time            4.02 seconds
13
14         data new2; set new;
15              run;

NOTE: DATA statement used:
      real time           1.81 seconds
      cpu time            1.70 seconds
```

Notice that the overall CPU execution time for the DATA step view example is 3.94 seconds (for the view execution) + 1.62 seconds (for the DATA step execution). By contrast, if we create the SAS data set and use a SET statement on the data set instead of the view, the CPU time is considerably less, only 1.7 seconds. You need to be careful when using views, especially when dealing with external files. The dynamic nature of external files makes them powerful but CPU intensive.

FUNCTION OR HARDCODING OR ARRAY

Here we consider several ways to accomplish the same task—determining the sum of a list of variables in a 500,000-observation data set. The first example uses the SUM function. The second uses a DIY approach—Do It Yourself—with SAS operators to see if the code runs faster. The third example tests the use of an ARRAY statement to accumulate sums. The execution time should be greater due to the loop iteration.

SUM FUNCTION

```
5          data new; set mydata.x500000;
6               sum = sum(of x1-x10);
7               run;

NOTE: DATA statement used:
      real time           9.84 seconds
      cpu time            2.04 seconds
```

SAS OPERATORS

```
9          data new; set mydata.x500000;
10              sum = x1+x2+x3+x4+x5+x6+x7+x8+x9+x10;
11              run;

NOTE: DATA statement used:
      real time           9.43 seconds
      cpu time            1.93 seconds
```

ARRAY STATEMENT

```
13         data new; set mydata.x500000;
14              array x x1-x10;
15              sum = 0;
16              do over x;
17                 sum + x;
18                 end;
19              run;

NOTE: DATA statement used:
      real time           10.37 seconds
      cpu time            2.88 seconds
```

Using the SUM function to sum a list of variables takes CPU 2.04 seconds, while the DIY method with SAS operators takes CPU 1.93 seconds. The ARRAY statement uses CPU 2.88 seconds to complete the task. There's not a great deal of performance improvement gained in the DIY method. If the function is performing more complex operations, it's certainly better to rely on the function because the code is simpler and easier to maintain.

STORED PROGRAM CONSIDERATIONS

This section considers DATA step stored programs as a means to improve efficiency. Stored programs are pre-compiled DATA steps that are subsequently executed with only a resolution phase and no subsequent compilation phase.

The following SAS macro program helps demonstrate stored program usage. The macro produces SAS code with a specified number of IF statements and with a different specified number of passes. This macro generates SAS code that in turn generates other SAS code. The program is run as a single DATA step and as a stored program for comparison.

```
filename sascode temp;
%macro runtest(dsname,npasses,nlines);
data _null_; file sascode;
     put "do npasses=1 to &npasses;";
     put "   x=ceil(ranuni(13131)*&nlines);";
     do i=1 to &nlines;
        put @5 'if x=' i ' then y=' i ';';
        end;
     put @5 'end;';
     put 'run;';
data &dsname.; %include sascode;
data &dsname./pgm=&dsname.; %include sascode;
data pgm=&dsname.; execute; run;
%mend;
```

RUNNING THE SAS CODE WITH 1 PASS AND 10 IF STATEMENTS

```
22         %runtest(new1,1,10);
MPRINT(RUNTEST):   data _null_;
MPRINT(RUNTEST):   file sascode;
MPRINT(RUNTEST):   put "do npasses=1 to 1;";
MPRINT(RUNTEST):   put "   x=ceil(ranuni(13131)*10);";
MPRINT(RUNTEST):   do i=1 to 10;
MPRINT(RUNTEST):   put @5 'if x=' i ' then y=' i ';';
MPRINT(RUNTEST):   end;
MPRINT(RUNTEST):   put @5 'end;';
MPRINT(RUNTEST):   put 'run;';
```

RUNNING THE SAS CODE WITH 1000 PASSES

```
MPRINT(RUNTEST):   data new3;
MPRINT(RUNTEST):   do npasses=1 to 1000;
MPRINT(RUNTEST):   x=ceil(ranuni(13131)*10);
MPRINT(RUNTEST):   if x=1 then y=1 ;
MPRINT(RUNTEST):   if x=2 then y=2 ;
...
MPRINT(RUNTEST):   if x=10 then y=10 ;
MPRINT(RUNTEST):   end;
MPRINT(RUNTEST):   run;

NOTE: DATA statement used:
      real time           0.03 seconds
      cpu time            0.01 seconds
```

COMPILING THE SAS CODE AS A STORED PROGRAM

```
MPRINT(RUNTEST):   data new3/pgm=new3;
...
```

STORED PROGRAM WITH 1000 PASSES AND 10 STATEMENTS

```
MPRINT(RUNTEST):   data pgm=new3;
MPRINT(RUNTEST):   execute;
MPRINT(RUNTEST):   run;

NOTE: DATA statement used:
      real time           0.03 seconds
      cpu time            0.02 seconds
```

With 1000 passes, but only 10 statements, the compile-and-run DATA step is a little faster than the stored program. The compilation time isn't counted toward the final execution time. This example shows that the compile phase for a small program is a tiny fraction of the overall execution, so creating a stored program is not very beneficial to overall performance.

STORED PROGRAM WITH 1000 PASSES AND MANY STATEMENTS

```
MPRINT(RUNTEST):   data new3;
MPRINT(RUNTEST):   do npasses=1 to 1000;
MPRINT(RUNTEST):   x=ceil(ranuni(13131)*1000);
MPRINT(RUNTEST):   if x=1 then y=1 ;
MPRINT(RUNTEST):   if x=2 then y=2 ;
...
MPRINT(RUNTEST):   if x=999 then y=999 ;
MPRINT(RUNTEST):   if x=1000 then y=1000 ;
MPRINT(RUNTEST):   end;
MPRINT(RUNTEST):   run;

NOTE: DATA statement used:
      real time           0.46 seconds
      cpu time            0.23 seconds

MPRINT(RUNTEST):   data new3/pgm=new3;

MPRINT(RUNTEST):   data pgm=new3;
MPRINT(RUNTEST):   execute;
MPRINT(RUNTEST):   run;

NOTE: DATA statement used:
      real time           0.18 seconds
      cpu time            0.09 seconds
```

Here the compile-and-execute time is greater than when running only the stored program. In those rare cases where the program size is indirectly proportional to execution passes, the stored program might be useful.

Be aware that if a stored program contains macros, they are resolved at compilation time and have no further bearing. In this example, the &MYVALUE macro variable is reset to second, but it has no bearing when the stored program is run a second time.

```
1          %let myvalue=first;
2          data xyz/pgm=xyz;
3               x="&myvalue.";
4               put x=;
5               run;

7          data pgm=xyz; run;

x=first

8
9          %let myvalue=second;
10         data pgm=xyz; run;

x=first
```

We can conclude that the larger the compiled code, with fewer execution passes, the more beneficial the stored program can be.

COMPARISON OF VARIOUS KEY LOOKUP METHODS

This section examines different key lookup methods:

- formats invoked via the PUT function
- sorting and merging
- the KEY= option
- the hash object

The first step is to create a demonstration key data set where the keys go from 1 to 5000, with the corresponding label being only the Z4 representation of the key, such as 15 having the label of 0015. The result is stored as a CNTLIN= data set for subsequent use by PROC FORMAT. The second step is to create the format, a one-time operation. The final step is to create some random test data. The test data set has 100,000 observations and three key variables, each getting random values between 1 and 5000.

```
data cntlin;
     fmtname='testfmt';
     do start=1 to 5000;
        label=put(start,z4.);
        output;
        end;
     run;
```

```
8          proc format cntlin=cntlin;
NOTE: Format TESTFMT has been output.
8        !                            run;

NOTE: PROCEDURE FORMAT used (Total process time):
      real time           0.26 seconds
      cpu time            0.21 seconds
11         data keytest;
12              array keys key1-key3;
13              do i=1 to 100000;
14                 do over keys;
15                    keys=ceil(ranuni(13131)*5000);
16                    end;
17                 output;
18                 end;
19              run;
```

FORMATS AND THE PUT FUNCTION

This example illustrates how to use the format for key lookup. The PUT function uses the format to create a character value. The variable key1x contains the label for the corresponding key1 value.

```
22         data new1a; set keytest;
23              key1x=put(key1,testfmt.);
24              run;

NOTE: DATA statement used (Total process time):
      real time           0.56 seconds
      cpu time            0.54 seconds
```

MERGING

The more traditional way to perform key lookup is a merge. To perform a merge, the input data set must first be sorted by the key. The sorting step is included in the overall CPU time.

```
26         proc sort data=keytest out=keytestx; by key1;

NOTE: PROCEDURE SORT used (Total process time):
      real time           2.96 seconds
      cpu time            0.65 seconds

27         data new1b; merge keytestx(in=want)
28                     cntlin(keep=start label rename=(start=key1));
29                     by key1;
```

```
     if want;
     run;
```

```
NOTE: There were 100000 obs read from WORK.KEYTESTX.
NOTE: There were 5000 obs read from WORK.CNTLIN.
NOTE: The data set WORK.NEW1B has 100000 obs and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.39 seconds
      cpu time            0.29 seconds
```

Note that the CPU time using the PUT function was .54 seconds. The sort, along with the merge, takes .65 + .29 seconds, or .94 CPU seconds, as compared to .54 CPU seconds for the format usage. But you might need an additional sort. Remember that if the necessary sort disrupted the observations so that they needed to be sorted again, more CPU time is consumed as shown here:

```
proc sort data=new1b; by i;
```

```
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.48 seconds
      cpu time            0.48 seconds
```

The format and PUT function method does not disrupt the observation order. Multiple keys can be handled within the same pass using the PUT function.

```
data new1a; set keytest;
     key1x=put(key1,testfmt.);
     key2x=put(key2,testfmt.);
     key3x=put(key3,testfmt.);
     run;
```

But for SORT/MERGE, you must sort and merge by each key separately, adding greatly to CPU time usage.

```
proc sort data=keytest out=new1b; by key1;
data new1b; merge new1b(in=want)
            cntlin(keep=start label
                 rename=(start=key1 label=key1x));
            by key1;
     if want;
     run;
proc sort data=new1b; by key2;
```

INDEXING

To illustrate using an indexed data set, the next step is to create the CNTLINX data set from the CNTLIN data set, adding an index so that a key lookup can be performed to locate the corresponding label value.

```
data cntlinx(index=(start)); set cntlin;

data new1b; set keytest;
     array keys key1-key3;
     length key1x key2x key3x $4;
     array keyx key1x key2x key3x;
     do over keys;
        start=9999; link getkey; _iorc_=0; _error_=0;
        start=keys; link getkey; keyx=label;
        end;
     return;
getkey: set cntlinx key=start;
     return;
     run;
```

```
NOTE: DATA statement used (Total process time):
      real time           26.85 seconds
      cpu time            26.53 seconds
```

Looking for the same key two times in a row is a problem that requires the introduction of a dummy key. Adding the dummy key adds to the execution time. The real time increases dramatically: 26.85 seconds as compared to less than 1 second using the other approaches.

HASH OBJECT

The hash object is a new feature in SAS®9 and a foray into object-oriented programming.

```
           data new1c; set keytest;
                array keys key1-key3;
                length key1x key2x key3x $4;
                array keyx key1x key2x key3x;
                length start 8 label $4;
                if _n_=1 then do;
                   declare hash h(hashexp: 4,
                    dataset:"work.cntlin", ordered: 'a');
                   h.defineKey('start');
                   h.defineData('label');
                   h.defineDone();
                   end;

                do over keys;
                   start=keys; rc = h.find();
                   if rc=0 then keyx=label;
                   end;
                run;
NOTE: DATA statement used (Total process time):
      real time           0.82 seconds
      cpu time            0.82 seconds
```

Populating the hash table along with all the searching took only .82 seconds. Although a little longer than the other methods, the hash object method is faster when you consider that the setup time is included. Notice that the hash table does not persist across DATA steps and must be re-created each time it is used.

THREADS OR NOTHREADS

To demonstrate threading in PROC SUMMARY, the first step is to create a large data set with over 7 million observations. There are about 500 different values of the CLASS variable and up to 30,000 observations per class. The values of x are random, with y and z being 1 and 2 larger, respectively.

```
data temp;
     do cv=1 to 500;
        classvar=ceil(ranuni(13131)*500);
        nobs = ceil(ranuni(13131)*30000);
        do i=1 to nobs;
           x = ceil(ranuni(13131)*1000);
           y = x + 1;
           z = x + 2;
           output;
           end;
        end;
     keep classvar x y z;
     run;
```

NOTHREADS OPTION

```
           proc summary data=temp nothreads;
                class classvar;
                format classvar z5.;
                var x y z;
                output out=new
                    mean=mx my mz min=minx miny minz
                    var=varx vary varz;
                run;

NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           16.32 seconds
      cpu time            16.31 seconds
```

THREADS OPTION

```
31         proc summary data=temp threads;
32              class classvar;
33              format classvar z5.;
34              var x y z;
35              output out=new
36                  mean=mx my mz min=minx miny minz
37                  var=varx vary varz;
38              run;

NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           13.93 seconds
      cpu time            31.16 seconds
```

The first example ran explicitly with the NOTHREADS option. CPU time and real time match at 16.31 seconds. Note that we use a FORMAT statement, multiple variables, and multiple statistics to compute this. Run with the THREADS option, the real time for the same example is less than the CPU time because multiple threads run simultaneously, and the CPU time is the sum of the thread execution time. The real time, 13.93 seconds, is a reduction over the NOTHREADS run, which is the desired result.

The following examples show how different hardware platforms can influence performance and how each platform can have significantly different performance indicators. To set up the example, the number of CLASS variables is increased and the examples are run with the NOTHREADS and THREADS options on two different UNIX machines.

```
26         proc summary data=temp [threads] [nothreads];
27              class classvar01-classvar10;
28              format classvar01-classvar10 z5.;
...
```

UNIX 1, USING NOTHREADS

```
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           1:10.76
      cpu time            1:00.62
```

UNIX 1, USING THREADS

```
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           32.36 seconds
      cpu time            1:38.40
```

UNIX 2, USING NOTHREADS

```
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           1:19.89
      cpu time            1:19.68
```

UNIX 2, USING THREADS

```
NOTE: PROCEDURE SUMMARY used (Total process time):
      real time           37.17 seconds
      cpu time            2:58.69
```

Using the THREADS option on UNIX 1 is a considerable improvement in real time, going from 1:10.76 down to 32.36, using only 38.40 more seconds of CPU. While UNIX 2 has somewhat similar improvements in real time, the THREADS option used more than twice the CPU time as the NOTHREADS option.

To benefit from threading, you need large amounts of data and you need to run on a multiprocessing machine. Not all SAS procedures are threaded, although more procedures are being converted to support threading capability. The THREADS/NOTHREADS global option can control whether any procedure will run with threading enabled.

CONCLUSION

This paper provides examples of several choices that the SAS programmer can make regarding efficiency. In each case, there are advantages and disadvantages to the choices. The SAS programmer should consider all aspects of the application to make the appropriate choice.

RESOURCES

Ray, Robert. 2000. "Version 8 Base SAS® Performance: How Does It Stack Up?" *Proceedings of the Twenty-fifth Annual SAS Users Group International Conference*. Available http://www2.sas.com/proceedings/sugi25/25/aa/25p009.pdf.

Ray, Robert. 2003. "An Inside Look at Version 9 and 9.1 Threaded Base SAS® Procedures." *Proceedings of the Twenty-eighth Annual SAS Users Group International Conference.* Available http://www2.sas.com/proceedings/sugi28/282-28.pdf.

Shamlin, David. 2004. "Threads Unraveled: A Parallel Processing Primer." *Proceedings of the Twenty-ninth Annual SAS Users Group International Conference*. Available http://www2.sas.com/proceedings/sugi29/217-29.pdf.

CONTACT INFORMATION

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

Rick Langston
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: (919) 677-8000
E-mail: Rick.Langston@sas.com