SAS® GLOBAL FORUM 2020 #SASGF

Paper SAS4319-2020

Write Custom Parallel Programs by Using the iml Action

Rick Wicklin and Arash Dehghan Banadaki, SAS Institute Inc.

ABSTRACT

This paper introduces the **`iml`** action, which is available in SAS® Viya® 3.5. The **`iml`** action supports most of the same syntax and functionality as the SAS/IML® matrix language that SAS® software has supported for decades. With minimal changes, most programs that run in the IML procedure can also run in the **`iml`** action.

In addition, the **`iml`** action supports new programming features for parallel programming. The **`iml`** action is different from most actions, which perform a specific task. The **`iml`** action provides a set of general programming tools that you can use to implement a custom parallel algorithm. The programmer can control the computation itself and can control how the computation is distributed among nodes and threads on a cluster of machines (or threads on a single machine). The parallel programming features are demonstrated by using examples of simulation, power estimates, and scoring regression models.

THE IML ACTION

The SAS/IML language is a matrix-vector language that supports a rich library of functions in statistics, data analysis, matrix computations, numerical analysis, simulation, and optimization. The SAS/IML language enables programmers to extend the capabilities of SAS software by writing programs in a high-level language.

In SAS 9, statistical programmers can access the SAS/IML language by licensing the SAS/IML product and calling the IML procedure. In SAS Viya 3.5. you can wield the power of the SAS/IML language by licensing the SAS IML product and calling the **`iml`** action. The **`iml`** action enables you to run SAS/IML programs in SAS Viya and use SAS Cloud Analytic Services (CAS).

The **`iml`** action belongs to the **`iml`** action set, which is part of the SAS IML product. Notice that there is no “slash” in the product name. The SAS IML product gives you access to the **`iml`** action and to the IML procedure in the SAS programming run-time environment. The **`iml`** action runs programs on a CAS server; the IML procedure runs programs on a SAS client.

In addition to supporting most of the statements and functions in the SAS/IML language, the **`iml`** action supports new functionality that enables you to take advantage of the distributed computational resources in SAS Viya. In particular, you can use the **`iml`** action to implement custom parallel algorithms that use multiple nodes and threads on a cluster of machines.

If you do not have a cluster of machines, you can still run custom parallel programs on a multicore processor. Computers that run multiple threads are both cheap and ubiquitous. In 2020, a standard desktop computer that costs $1,000 comes with a quad-core processor that can run eight threads simultaneously. You can upgrade to a processor that has eight cores and 16 threads for another $500.

COMPARE THE IML ACTION AND PROC IML

Many SAS/IML programs that run in PROC IML also run in the **iml** action. The mathematical and statistical function library is essentially the same in the action and in the procedure. Both environments support arithmetic and linear algebraic operations on matrices, operations to subset and query matrices, and programming features such as writing loops and using IF-THEN/ELSE logic.

Of the 300 functions and statements in the SAS/IML run-time library, only a handful of statements are not supported in the **iml** action. Most differences are related to the difference between the SAS 9 and SAS Viya environments. PROC IML interacts with traditional SAS constructs (such as data sets and catalogs) and supports calling SAS procedures. The **iml** action interacts with analogous constructs in the Viya environment, can read and write CAS tables and write analytic stores (astores), and can call other Viya actions.

The **iml** action is documented in the *SAS IML: Programming Guide*, which describes new functions and statements that are supported by the action (but not by the IML procedure). The documentation also lists the SAS/IML functions and statements that are not supported by the **iml** action.

Why might you choose to use the **iml** action instead of PROC IML? There are two main reasons:

1. You want to use the SAS/IML language to analyze data that are in CAS tables. It is more efficient to read and write a CAS table by using the **iml** action than to pull the data from CAS, run the analysis in PROC IML, and then push the results back to CAS.
2. You want to take advantage of the capabilities of the CAS server to perform parallel processing. You can use the **iml** action to create custom parallel computations.

The second reason is the main focus of this paper.

To demonstrate the similarities between PROC IML and the **iml** action, the following sections show how to run the same program in each environment.

Use PROC IML to Run a Program

The following statements use PROC IML to run a program. The program defines two vectors and computes their inner product. It then computes the variance of the **x** data vector. Lastly, it centers and scales the data and computes the variance of the standardized data. The program prints these three results, which are shown in Figure 1.

```
PROC IML;
   c = {1, 2, 1, 3, 2, 0, 1};          /* weights                   */
   x = {0, 2, 3, 1, 0, 2, 2};          /* data                      */
   wtSum = c` * x;                     /* inner product (weighted sum)  */
   var1  = var(x);                     /* variance of original data     */
   stdX = (x-mean(x)) / std(x);        /* standardize data              */
   var2 = var(stdX);                   /* variance of standardized data */
   print wtSum var1 var2;
QUIT;
```

Figure 1 Results of Computations in PROC IML

wtSum	var1	var2
12	1.2857143	1

As expected, the variance of the standardized data is 1.

Use the iml Action to Run a Program

You can run the same program in the **iml** action. The following statements assume that you have a license for the SAS IML product and that you have already connected to a CAS server. You have to load the **iml** action once per session by using the **loadactionset** action. The following template shows how to call the **iml** action by using the CAS procedure:

```
PROC CAS;
loadactionset 'iml';                /* load the action set */
source ProgramName;
   < put program here >
endsource;
iml / code=ProgramName;             /* run the program in the action */
RUN;
```

Notice that PROC CAS supports a SOURCE-ENDSOURCE block, which you can use to define a program and preserve its formatting, such as line breaks, indented regions, and blank lines. The following program loads the action, defines the program, and calls the **iml** action to run the program. As shown in Figure 2, the results are identical to Figure 1.

```
/* Example of using PROC CAS in SAS to call the iml action */
PROC CAS;
loadactionset 'iml';                /* load the action set           */
source pgm;
   c = {1, 2, 1, 3, 2, 0, 1};       /* weights                       */
   x = {0, 2, 3, 1, 0, 2, 2};       /* data                          */
   wtSum = c` * x;                  /* inner product (weighted sum)  */
   var1  = var(x);                  /* variance of original data     */
   stdX = (x-mean(x)) / std(x);     /* standardize data              */
   var2 = var(stdX);                /* variance of standardized data */
   print wtSum var1 var2;
endsource;
iml / code=pgm;               /* run the 'pgm' program in the action */
RUN;
```

Figure 2 Results of Computations in the **iml** Action

Results from iml.iml

wtSum	var1	var2
12	1.2857143	1

Call the iml Action from Python

You can call the **iml** action (in fact, all CAS actions) in several different languages. Although this paper uses PROC CAS, other choices include the Python, Lua, and R languages. For example, in the Python language, you can install the SAS Scripting Wrapper for Analytics Transfer (SWAT) package and then use the following statements to connect to a CAS server and load the action:

```
# Example of using Python to call the iml action
import swat                     # load the swat package
s = swat.CAS('myhost', 12345)   # server='myhost'; port=12345
s.loadactionset('iml')          # load the action set
```

You can then call the **`iml`** action to run a SAS/IML program, as follows:

```
m = s.iml(code=
"""
   c = {1, 2, 1, 3, 2, 0, 1};          /* weights                    */
   x = {0, 2, 3, 1, 0, 2, 2};          /* data                       */
   wtSum = c` * x;                     /* inner product (weighted sum)  */
   var1  = var(x);                     /* variance of original data     */
   stdX = (x-mean(x)) / std(x);        /* standardize data              */
   var2 = var(stdX);                   /* variance of standardized data */
   print wtSum var1 var2;
""")
```

Similar to the SOURCE/ENDSOURCE block in PROC CAS, the triple quotes (`"""`) in Python enable you to preserve indention and comments.

PARALLEL COMPUTATIONS IN THE IML ACTION

A primary reason for using the **`iml`** action is to implement parallel programs. In statistics, tasks that are easy to parallelize include simulations, bootstrap analyses, permutation tests, and multiple imputation. Another easy task to parallelize is a computation that is repeated many times, using a different parameter value each time. Lastly, you can score a model on many different inputs in parallel.

Terms Used in Parallel Computations

Before showing examples of using the **`iml`** action to run parallel computations, this section defines a few technical terms that are used in this paper and in the *SAS IML: Programming Guide*.

Node	A node is a self-contained computing unit that is part of a cluster of computers. A typical CAS session uses a controller node and zero or more worker nodes.
Controller	A controller node runs a SAS/IML program on a single thread until it encounters a statement that requests distributed processing. It then distributes work to multiple threads (and nodes, if available) and collects the results.
Worker	A worker is a node that can be used to perform distributed computations.
Multiple-machine mode	If your CAS session includes one or more worker nodes, the **`iml`** action runs in multiple-machine mode. The main program runs on the controller, but distributed computations run in multiple threads on the worker nodes. This mode is also known as massively parallel processing (MPP).
Single-machine mode	If your CAS session does not include any worker nodes, the **`iml`** action runs in single-machine mode. The program runs entirely on the controller, although it can still use multiple threads on the controller for parallel processing. This mode is also known as symmetric multiprocessing (SMP).
Threads	A (physical) thread is the smallest processing unit on a CPU. A node can contain many CPUs, and each CPU can run multiple threads. Multithreading enables a program to perform several computations concurrently. Threads on a single node can share a common block of memory. Threads on different nodes cannot share memory.

Summary of Parallel Computations in the iml Action

When you use the **`iml`** action to read and write CAS data tables, the data are automatically read and written in parallel. In addition, the action supports the following predefined functions that perform parallel computations. For more information about these functions, see the *SAS IML: Programming Guide*.

MAPREDUCE The MAPREDUCE function runs a SAS/IML module called the *mapping function* (also called the *mapper*) on every available node and thread in your CAS session. Each mapping function returns results. The results are aggregated by using the *reducing function* (also called a *reducer*). The final aggregated result is returned by the MAPREDUCE function.

The MAPREDUCE function is ideal for "embarrassingly parallel" computations, which are composed of many independent and essentially identical computations. Examples include Monte Carlo simulation and resampling methods such as the bootstrap.

PARTASKS The PARTASKS function enables you to run several independent modules in parallel. The PARTASKS function supports the following cases:

- Each thread runs the same module. The arguments to the module are different in each thread.
- Each thread runs a different module. The arguments to each module are the same.
- Each thread runs a different module. The arguments to each module are different.

You can control how the PARTASKS function distributes tasks to the available nodes and threads.

SCORE The SCORE function enables you to send each row of a CAS data table to a SAS/IML function, called the *scoring function*. If the scoring function is vectorized, you can also send a block of rows. The output from the scoring function is written to an output CAS table, which has the same number of rows as the input table. As its name indicates, the SCORE function is intended for scoring (evaluating a predictive model). However, you can also use the SCORE function to evaluate any function of the input variables.

MAPREDUCETABLE The MAPREDUCETABLE function combines features of the MAPREDUCE and SCORE functions. The MAPREDUCETABLE function processes rows of a CAS table and sends each row of the table to the mapping function, which runs on every thread. The results of the mapping function are sent to a user-defined pairwise-reducing function. The results are returned in a SAS/IML symbol.

ASTORE The ASTORE function creates an *analytic store* (usually abbreviated as "astore"), which is a binary file that stores information about a predictive model. The astore is used by the ASTORE procedure or the **`aStore.score`** action to evaluate (score) the model on new observations. The ASTORE function in the **`iml`** action enables you to use user-defined SAS/IML modules to score new data.

THE MAP-REDUCE PARADIGM

The map-reduce paradigm enables you to implement parallel algorithms by using two functions: a mapper and a reducer. The mapper runs on every thread and returns results. The reducer aggregates those results.

A simple example is adding large set of numbers. Suppose you have N numbers to add and you have access to k threads. You can tell each thread to add N/k numbers. Each thread runs a mapper function that adds the numbers it is given and returns the sum. Thus, the mapping step results in k partial sums. The next step is the reducing step. The k partial sums are passed to the reducer, which adds them and returns the total sum. In this way, the map-reduce paradigm computes the sum of the N numbers in parallel.

In the **`iml`** action, the map-reduce paradigm is implemented by using the MAPREDUCE function. The syntax of the MAPREDUCE function is

```
   result = MAPREDUCE( mapArg, 'MapFunc', 'RedFunc' );
```

In this syntax, 'MapFunc' is the name of the mapper function and **`mapArg`** is a parameter that is passed to the mapper function in every thread. The 'RedFunc' argument is the name of the reducer function. The **`iml`** action supports a number of predefined (built-in) reducers, and you can also define your own reducer functions. The examples in this paper use only predefined reducers.

The following program uses the map-reduce paradigm to implement the sum-of-N-numbers algorithm. For this simple example, the program runs on four threads and sums the numbers 1, 2, . . . , 1,000. How you get the data to the mapper is a matter of choice. This program packs the data into a matrix that has four rows. Each thread analyzes one row. The program defines a helper function (getThreadID) and a mapper function (AddRow), which will run on multiple threads. On each thread, the AddRow function does the following:

1. Uses the getThreadID function to determine the thread in which the AddRow function is running. The getThreadID function is a thin wrapper around the NODEINFO function, which is a built-in function in the **`iml`** action. The thread ID is stored in the variable **`j`**.
2. Extracts the jth row of numbers. These are the numbers that the jth thread will sum.
3. Uses the SUM function to compute the partial sum, prints the sum, and returns it.

In the following program, the built-in '_SUM' reducer adds the partial sums and returns the total sum. The results are shown in Figure 3. When computations are performed in parallel, the PRINT statement displays the node and thread information after printing the result from each node and thread.

```
proc cas;
session sess0;                         /* SMP session: controller node only       */
loadactionset 'iml';                   /* load the action set                     */
source MapReduceAdd;
   start getThreadID(j);
      L = nodeInfo();                  /* get information about nodes and threads  */
      j = L$'threadId';                /* this function runs on the j_th thread    */
   finish;
   start AddRow(X);
      call getThreadId(j);             /* running in the j_th thread               */
      sum = sum(X[j, ]);               /* compute the partial sum for the j_th row */
      print sum;                       /* print partial sum for this thread        */
      return sum;                      /* return the partial sum                   */
   finish;

   /* ----- Main Program ----- */
   x = shape(1:1000, 4);               /* create a 4 x 250 matrix                  */
   Total = MapReduce(x, 'AddRow', '_SUM'); /* use built-in _SUM reducer            */
   print Total;
endsource;
iml / code=MapReduceAdd nthreads=4;
run;
```

Figure 3 Results of Map-Reduce Algorithm

```
          sum
        218875

Result from Node 0 Thread 4.

          sum
         31375

Result from Node 0 Thread 1.

          sum
        156375

Result from Node 0 Thread 3.
```

Figure 3 *continued*

sum
93875

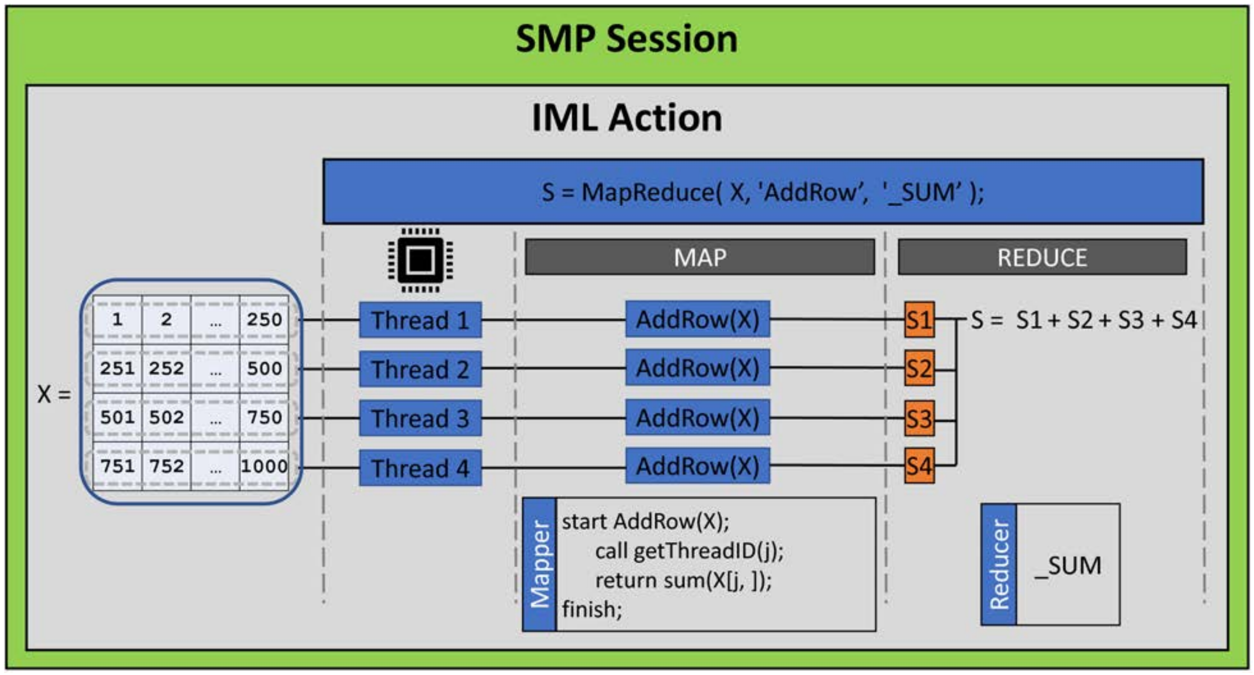
Result from Node 0 Thread 2.

Total
500500

The output displays the PRINT statement results from each of the four threads, which can appear in any order. The first thread computes the sum of the numbers 1, 2, . . . , 250 and returns the value 31,375. The second thread computes the sum of the numbers 251, 252, . . . , 500 and returns the value 93,875. The other threads perform similar computations. The partial sums are sent to the built-in '_SUM' reducer, which adds them together and returns the total sum to the main program. The total sum is 500,500 and appears in the output.

The following diagram illustrates how the MAPREDUCE function distributes the tasks and data for this example in SMP mode in four threads.

Figure 4 Distributed Computations for the MAPREDUCE Function



USE THE MAPREDUCE FUNCTION FOR MONTE CARLO SIMULATION

The map-reduce paradigm is ideal for Monte Carlo simulation. In a Monte Carlo simulation, you simulate B samples of size N from a probability distribution. For each sample, you compute a statistic. When B is large, the distribution of the statistics is a good approximation to the sampling distribution for the statistic. See Wicklin (2013) for many examples of Monte Carlo simulation in SAS.

This section uses a simulation study to understand the sampling distribution of the sample mean. You can use the simulation to estimate the standard error of the sample mean and to estimate a confidence interval for the mean. This section presents a parallel implementation of an algorithm in Wicklin (2013, p. 55–57).

The following program simulates $B = 10^6$ samples of size $N = 36$ from the uniform distribution, $U(0,1)$, and estimates the standard error of the sample mean. The program defines a mapper function named SimMeanUniform, which will run on multiple threads. On each thread, the function does the following:

1. Uses the RANDSEED subroutine to set a random number seed. The subroutine uses the node and thread information to augment the specified value of the seed. Consequently, each thread generates an independent stream of random values.
2. Uses the NPERTHREAD function (a new function in the **`iml`** action) to determine how many samples should be simulated on the current thread. If you are using k threads, each thread needs to compute $M \approx B/k$ samples.
3. Uses the RANDGEN subroutine to generate the samples from the $U(0,1)$ distribution. In the $N \times M$ matrix, X, each column is a sample and there are M samples.
4. Uses the MEAN function to compute the mean of each sample (column) and returns the row vector of the sample means.

The argument to the SimMeanUniform function is a SAS/IML list (Wicklin 2017) that contains three elements: the sample size, a random number seed, and the total number of Monte Carlo samples to generate. Alternatively, you could use a three-element numeric vector instead of a list.

The MAPREDUCE function (in the main program) runs the SimMeanUniform function (the mapper) on all threads. The results from the mapper are sent to the built-in '_HCONCAT' reducer, which concatenates the results into one very long row vector that has B values. (The doumentation describes all built-in reducers.) The distribution of the values approximates the sampling distribution of the sample mean. The remainder of the program summarizes the sampling distribution. The program is run on a cluster that has four worker nodes, each of which runs eight threads. The results are shown in Figure 5.

```
/* Simulate B independent samples from a uniform(0,1) distribution.
   Mapper: Generate M samples, where M ~ B/numThreads. Return M statistics.
   Reducer: Concatenate the statistics.
   Main Program: Estimate standard error and CI for mean.
 */
proc cas;
session sess4;                         /* use session with four workers     */
loadactionset 'iml';                   /* load the action set               */
source SimMean;

start SimMeanUniform(L);               /* define the mapper                 */
   call randseed(L$'seed');            /* each thread uses different stream */
   M = nPerThread(L$'B');              /* number of samples for thread      */
   x = j(L$'N', M);                    /* allocate NxM matrix for samples   */
   call randgen(x, 'Uniform');         /* simulate from U(0,1)              */
   return mean(x);                     /* row vector = mean of each column  */
finish;

/* ----- Main Program ----- */
/* Put the arguments for the mapper into a list */
L = [#'N'    = 36,                     /* sample size                       */
     #'seed' = 123,                    /* random number seed                */
     #'B'    = 1E6 ];                  /* total number of samples           */
```

```
/* Simulate on all threads; return Monte Carlo distribution */
stat = MapReduce(L, 'SimMeanUniform', '_HCONCAT');

/* Form Monte Carlo estimates */
alpha = 0.05;
stat = stat`;
numSamples = nrow(stat);
MCEst = mean(stat);                   /* estimate of mean,            */
SE = std(stat);                       /* standard deviation,          */
call qntl(CI, stat, alpha/2 || 1-alpha/2);    /* and 95% CI           */
R = numSamples || MCEst || SE || CI`;         /* combine for printing */
print R[format=8.4 L='95% Monte Carlo CI'
        c={'NumSamples' 'MCEst' 'StdErr' 'LowerCL' 'UpperCL'}];

endsource;
iml / code=SimMean nthreads=8;
run;
```

Figure 5 Results of Monte Carlo Simulation

95% Monte Carlo CI				
NumSamples	**MCEst**	**StdErr**	**LowerCL**	**UpperCL**
1000000	0.5000	0.0481	0.4058	0.5942

The Monte Carlo estimate of the mean is 0.5. The estimate of the standard error of the sample mean is 0.0481. A 95% confidence interval for the mean is $[0.4058, 0.5942]$.

For large sample sizes, the central limit theorem ensures that the distribution of the sample mean is approximately normal. The sample size in this simulation is 36. The asymptotic estimate of the standard error is $1/(12\sqrt{3}) \approx 0.0481$. Because the mean of the $U(0,1)$ distribution is $1/2$, a 95% confidence interval is $1/2 \pm 1.96(0.0481) = [0.4057, 0.5943]$. These asymptotic estimates are close to the Monte Carlo estimates.

When the action runs on a cluster that contains four worker nodes, each running eight threads, each thread simulates and computes $1/32$ of the simulation study. For complicated simulation studies that require a long time to run, you can substantially reduce the run time of the study by distributing the computations to many threads.

USE THE PARTASKS FUNCTION TO DISTRIBUTE TASKS IN PARALLEL

In the previous example, the mapping function ran in multiple threads. Each thread received exactly the same set of parameters. An alternative scenario is that you want to run a function several times but use a different value of the parameters each time. If each run of the algorithm is independent, then you can run multiple instances of the algorithm in parallel.

For example, suppose you have a large matrix and you need to compute three things: the determinant, the matrix inverse, and the eigenvalues. You can use the PARTASKS function to run the three tasks concurrently. Each task must be a SAS/IML function that has one input argument and returns one output argument. If a task requires more than one input argument, you can pack the arguments into a list and pass the list as a single argument.

The syntax of the PARTASKS function is

```
result = PARTASKS( tasks, taskArgs, options );
```

In this syntax, **`tasks`** is a character vector that names the SAS/IML functions, **`taskArgs`** is a list of arguments, and **`options`** is an (optional) vector that specifies how the tasks are distributed.

The following program shows how to distribute three tasks to three threads by using the PARTASKS function:

```
proc cas;
session sess0;                        /* SMP session: controller node only    */
loadactionset 'iml';
source partasks;
   start detTask(A);
      return det(A);
   finish;
   start invTask(A);
      return inv(A);
   finish;
   start eigvalTask(A);
      return eigval(A);
   finish;

   /* ----- Main Program ----- */
   A = toeplitz(1000:1);              /* 1000 x 1000 symmetric matrix         */
   Tasks = {'detTask' 'invTask' 'eigvalTask'};
   Args = [A, A, A];                  /* each task gets same arg in this case */
   opt = {1,                          /* distribute to threads on controller  */
          1};                         /* display information about the tasks  */
   Results = ParTasks(Tasks, Args, opt); /* results are returned in a list    */

   /* the i_th list item is the result of the i_th task */
   det    = Results$1;                /* get det(A)                           */
   inv    = Results$2;                /* get inv(A)                           */
   eigval = Results$3;                /* get eigval(A)                        */
endsource;
iml / code=partasks nthreads=4;
run;
```

This example runs the three tasks in three separate threads on the controller node. Each task gets the same 1000×1000 symmetric matrix. The second element of the **opt** vector requests that the PARTASKS function output a summary of the tasks, including which node and thread each task runs on. The output table is shown in Figure 6. The PARTASKS function returns a list. The first item in the list is the result of the first task, the second item is the result of the second task, and so on.

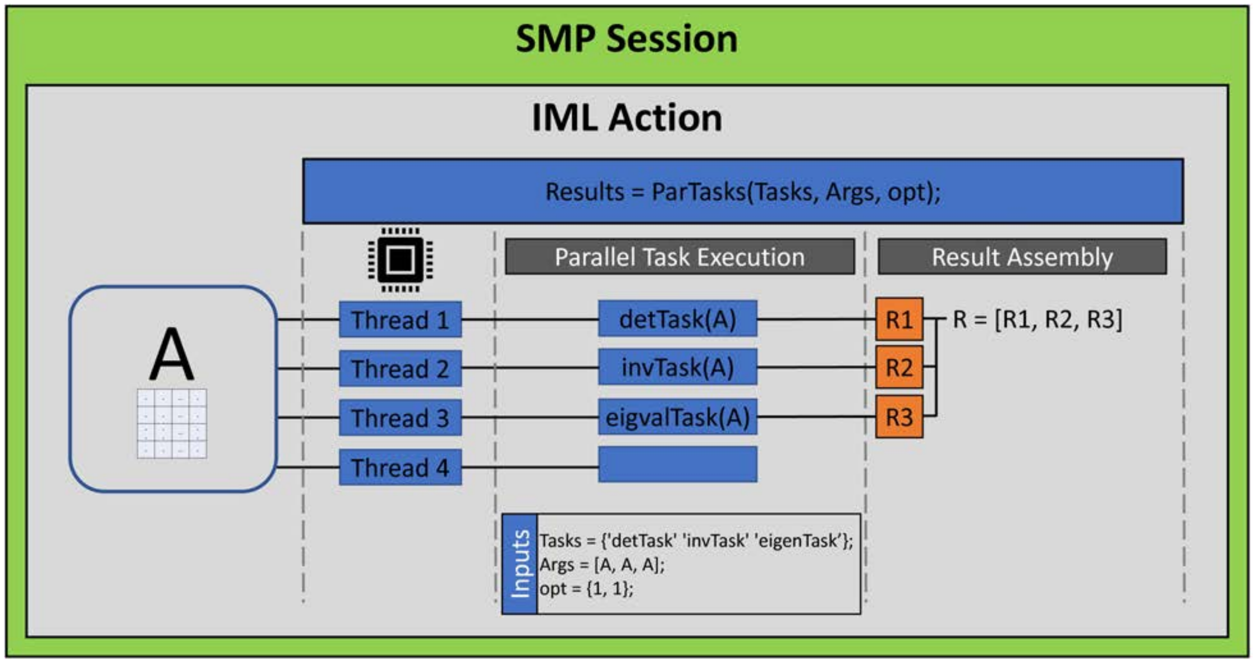
Figure 6 Results from Running Tasks in Parallel

Parallel Tasks Information

TaskNumber	Task	Node	Thread	Seconds
1	detTask	0	1	0.13
2	invTask	0	2	0.06
3	eigvalTask	0	3	0.39

The following diagram illustrates how the PARTASKS function distributes tasks for this example in SMP mode in four threads.

Figure 7 Distributed Computations for the PARTASKS Function



USE THE PARTASKS FUNCTION TO ESTIMATE POWER

This section shows how to use the PARTASKS function in a statistical application. The program in this section involves a hypothesis test. Each thread generates many random samples, computes a statistic for each hypothesis test, and returns the number of random samples for which the null hypothesis is rejected. Thus, each thread returns only one number (a count). In general, if you need to pass information between the nodes of a cluster, passing a small amount of data is more efficient than passing larger amounts.

The simulation study in this section is a power analysis of the two-sample t test for the difference between the means of two groups. Given two independent samples (assumed to be random samples from normal distributions with a common variance), the t test attempts to reject the null hypothesis that the two distributions have the same mean. You reject the null hypothesis if the test statistic is more extreme than a critical value.

Recall that the *power* of a statistical test is the probability that the test rejects the null hypothesis when the alternative hypothesis is true. Simulation is a good way to estimate the power of a test.

This study defines two functions. The TTestH0 function is a helper function that implements the t test and returns the proportion of samples that reject the null hypothesis. The SimTTest function simulates the first random sample from the $N(0,1)$ distribution and the second random sample from the $N(\delta,1)$ distribution. These samples are sent to the TTestH0 function, which estimates the power of the t test.

In this simulation, the parameter δ is the difference between the population means. If you estimate the power for a range of parameter values, you obtain a curve that shows the power as a function of δ. An easy way to compute points in the power curve in parallel is to have each thread simulate B pairs of samples and return the number of samples that reject the null hypothesis. You can assign each thread a different value of δ, as shown in the following program:

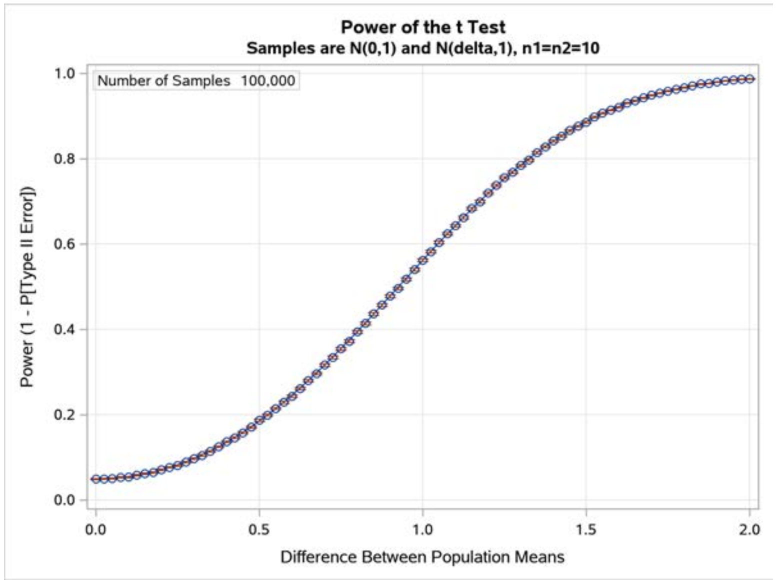
```
/* Power curve computation: delta = 0 to 2 by 0.025 */
proc cas;
session sess4;                         /* use session with four workers     */
source TTestPower;

/* Helper function: Compute t test for each column of X and Y.
   X is (n1 x m) matrix and Y is (n2 x m) matrix.
   Return the number of columns for which t test rejects H0 */
```

```
start TTestH0(x, y);
   n1 = nrow(X);     n2 = nrow(Y);     /* sample sizes                    */
   meanX = mean(x);  varX = var(x);    /* mean & var of each sample       */
   meanY = mean(y);  varY = var(y);
   poolStd = sqrt( ((n1-1)*varX + (n2-1)*varY) / (n1+n2-2) );

   /* Compute t statistic and indicator var for tests that reject H0 */
   t = (meanX - meanY) / (poolStd*sqrt(1/n1 + 1/n2));
   t_crit =  quantile('t', 1-0.05/2, n1+n2-2);       /* alpha = 0.05        */
   RejectH0 = (abs(t) > t_crit);                     /* 0 or 1              */
   return  sum(RejectH0);              /* count how many reject H0          */
finish;

/* Simulate two groups; Count how many reject H0: delta=0 */
start SimTTest(L);                     /* define the mapper                 */
   call randseed(L$'seed');            /* each thread uses different stream */
   x = j(L$'n1', L$'B');               /* allocate space for Group 1        */
   y = j(L$'n2', L$'B');               /* allocate space for Group 2        */
   call randgen(x, 'Normal', 0,         1);   /* X ~ N(0,1)                 */
   call randgen(y, 'Normal', L$'delta', 1);   /* Y ~ N(delta,1)             */
   return TTestH0(x, y);
finish;

/* ----- Main Program ----- */
numSamples = 1e5;
L = [#'delta' = .,   #'n1' = 10,  #'n2' = 10,
     #'seed'  = 321, #'B'  = numSamples];

/* Create list of arguments. Each arg gets different value of delta */
delta = T( do(0, 2, 0.025) );
ArgList = ListCreate(nrow(delta));
do i = 1 to nrow(delta);
   L$'delta' = delta[i];
   call ListSetItem(ArgList, i, L);
end;

RL = ParTasks('SimTTest', ArgList, {2, 0});  /* assign nodes before threads */

/* Summarize results and write to CAS table for graphing */
varNames = {'Delta' 'ProbEst' 'LCL' 'UCL'};  /* names of result vars        */
Result = j(nrow(delta), 4, .);
zCrit = quantile('Normal', 1-0.05/2);  /* zCrit = 1.96                      */
do i = 1 to nrow(delta);               /* for each task                     */
   p = RL$i / numSamples;              /* get proportion that reject H0     */
   SE = sqrt(p*(1-p) / L$'B');         /* std err for binomial proportion   */
   LCL = p - zCrit*SE;                 /* 95% CI                            */
   UCL = p + zCrit*SE;
   Result[i,] = delta[i] || p || LCL || UCL;
end;

call MatrixWriteToCas(Result, '', 'PowerCurve', varNames);
endsource;
iml / code=TTestPower nthreads=8;
quit;
```

By using four nodes, each running eight threads, it takes only 0.4 seconds to compute the entire power curve. There were 81 tasks and 32 threads, so each thread computes two or three points on the curve. The PARTASKS function supports several ways to distribute tasks to threads. See the *SAS IML: Programming Guide* for details.

Figure 8 shows a graphical summary of the results. The graph shows the 81 power estimates along with 95% binomial confidence intervals for the power. The confidence intervals are so small that they are hard to see.

Figure 8 Graphical Visualization of the Results of Parallel Tasks



USE THE SCORE FUNCTION FOR PARALLEL SCORING

Another application of parallel computing is scoring a model on new observations. You can interpret the phrase "score a model" very broadly to mean "evaluate a function on each observation." Examples of scoring include the following:

- Evaluate a regression model to predict a response for a new observation.
- Find the nearest point to an observation from among a list of candidate points. You can use this technique to classify an observation into a cluster according to its distance from a center.
- Smooth, filter, or denoise an audio signal.
- Filter or classify an image.

The SCORE function in the **iml** action enables you to process a CAS table's observations in parallel without loading the entire table into a SAS/IML matrix. You write a scoring function that will run in each thread. When you call the SCORE function, one or more observations from the input table are sent to the function in each thread. The results of the scoring function are written to an output CAS data table, which has the same number of observations as the input table. CAS tables do not necessarily preserve order of observations, so it is usually necessary to include an ID variable in the output table.

The syntax of the SCORE function is complicated, but essentially the SCORE function requires three things: an input data table, a function that evaluates observations from the input data table, and an output data table.

The following DATA step creates a CAS table that has one million observations and 100 variables that are named **X1**, **X2**, . . . , **X100**. This is the input data table to the SCORE function.

```
libname myLib  cas sessref=sess4;
%let nObs  = 1e6;
%let nVars = 500;
/* Simulate in parallel. nObsPerThread is the number of obs for each thread */
data myLib.ScoreData(drop=i j nExtras nObsPerThread) / single=no;
   array x[&nVars];
   call streamInit(73);
   nObsPerThread  = int(&nObs/_nthreads_);
   nExtras        = mod(&nObs,_nthreads_);
   if _threadid_ <= nExtras  then nObsPerThread = nObsPerThread + 1;
```

```
      do i = 1 to nObsPerThread;
         do j= 1 to &nVars;   x{j} = rand('Uniform');   end;
         ID = i + (_threadid_-1) * nObsPerThread;
         output;
      end;
   run;
```

Because this DATA step uses random numbers in parallel, the actual values in the data depend on the number of threads and nodes that you use to generate the data.

The second thing the SCORE function needs is a scoring function. Suppose you have previously fit a linear regression model on training data. You decide that the best fitting model is

$$Y = 2 + (1/100)X_1 + (2/100)X_2 + \ldots + (100/100)X_{100}$$

You can use matrix multiplication to evaluate a linear regression model. The predicted values are $b_0 + Xb$, where the b_0 is the intercept, X is the no-intercept design matrix for one or more rows of data, and b is the vector of parameter estimates for the explanatory effects. The design matrix for this linear regression is merely the matrix whose columns are the variables **X1**, **X2**, . . . , **X100**.

The following example uses the `iml` action to score the **ScoreData** table. The program defines a SAS/IML function named RegScore. The function runs in every thread. The function receives observations (up to 1,000 at a time) from the **ScoreData** table via the matrix `X`. The function also receives a vector of parameters (`b`) from the main program. It uses matrix multiplication to evaluate the model at each observation. The predicted values are written to a CAS table named **PredVals**.

```
   proc cas;
   session sess4;                       /* use session with four workers  */
   loadactionset 'iml';
   source ScorePgm;

   /* Score regression model. b[1] is intercept. X contains rows of data. */
   start RegScore(b, X);
      return ( b[1] + X*b[2:nrow(b)] );
   finish;

   b = T( 2 || (1:100)/100 );           /* column of parameters for model */
   inVars   = 'X1':'X100';              /* name of input variables        */
   outVars  = 'Pred';                   /* name of output variable        */
   copyVars = 'ID';                     /* names of variables to copy     */

   rc = Score('RegScore', b,            /* scoring function and arg       */
              inVars, outVars,          /* input & output variables       */
              'ScoreData', 'PredVals',  /* input & output CAS table names */
              1000, copyVars);          /* block size, variables to copy  */

   /* OPTIONAL: Save the function in an astore for scoring later */
   rc = astore('MyStore',               /* name of astore                 */
               'RegScore', b,           /* scoring function and arg       */
               inVars, outVars);        /* input & output variables       */

   endsource;
   iml / code = ScorePgm nthreads=4;
   run;
```

The program does not display any output, but it creates a new CAS table that is named **PredVals** and contains the **Pred** and **ID** variables. The **Pred** variable is the output from the RegScore function. You can use the `columninfo` action to display the variables. The results are shown in Figure 9.

```
   columnInfo / table='PredVals';       /* table of predicted values */
   run;
```

Figure 9 Variables in Output Data Table

Results from table.columnInfo

Column Information for PREDVALS in Caslib CASUSERHDFS(frwick)						
Column	Id	Type	Length	Formatted Length	Format Width	Format Decimal
Pred	1	double	8	12	0	0
ID	2	double	8	12	0	0

This example uses the SCORE function in the `iml` action to score a model on new data. However, if you prefer, you can save the scoring function in an analytic store, which you can use in the ASTORE procedure or the `aStore.score` action to score the model at a later time.

The last statement in the preceding program calls the ASTORE function, which saves the scoring function (RegScore) and the parameter vector (**b**) to an astore named **MyStore**. The following statements call PROC ASTORE to score the **ScoreData** table. Internally, PROC ASTORE calls the RegScore function and passes it two arguments (the **b** vector and rows of the **ScoreData** table), as follows:

```
/* PROC ASTORE can load iml action and evaluate model on new data */
proc astore;
score rstore = myLib.MyStore           /* astore written by iml action */
      data = myLib.ScoreData           /* new data table to score      */
      out  = myLib.PredVals2  copyvars=(ID);
run;

proc print data=myLib.PredVals2(where=(ID<=5)) noobs;
   var ID Pred;
run;
```

The **PredVals2** table contains the **Pred** and **ID** variables for the observations in the **ScoreData** table. A few observations are shown in Figure 10.

Figure 10 Predicted Values from PROC ASTORE

The ASTORE Procedure

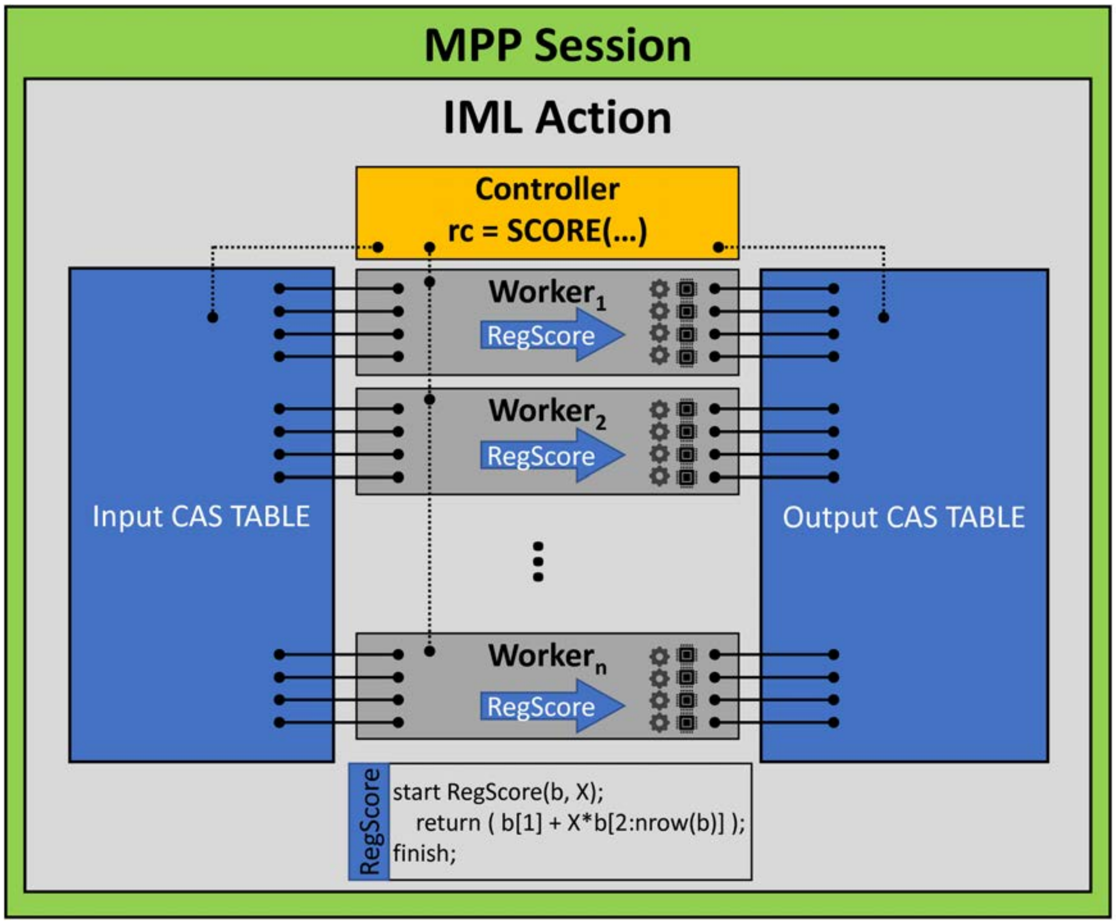
Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS(frwick)	PREDVALS2	1000000	2

Task Timing		
Task	Seconds	Percent
Loading the Store	0.01	0.49%
Creating the State	0.08	5.82%
Scoring	1.32	93.67%
Total	1.41	100.00%

ID	Pred
1	28.9287
2	28.6149
3	31.4543
4	24.9605
5	26.5414

Figure 11 illustrates how the SCORE function distributes data from an input CAS table in MPP mode. Every thread in every worker node receives observations, scores them in parallel by running the RegScore function, and writes the results to an output CAS table.

Figure 11 Distributed Computations for the SCORE Function



```
start RegScore(b, X);
   return ( b[1] + X*b[2:nrow(b)] );
finish;
```

This paper does not discuss the MAPREDUCETABLE function, but it is very similar to the SCORE function. Like the SCORE function, the MAPREDUCETABLE function sends rows of a CAS table to a mapping function without ever loading the full table into a matrix. The results of the mapper functions are sent to a user-defined pairwise-reducing function. The reduced results are the return value for MAPREDUCETABLE. No output CAS table is created.

SUMMARY

This paper describes how you can use the **`iml`** action (first available in SAS Viya 3.5) to write parallel programs. In addition to supporting almost all of the SAS/IML functions and statements in the SAS/IML run-time library, the **`iml`** action supports subroutines that enable you to write custom parallel programs. This paper shows several examples of parallel programs, including simulating data, distributing tasks, and scoring a data table. For more information and examples about the **`iml`** action, see the *SAS IML: Programming Guide*.

APPENDIX: FREQUENTLY ASKED QUESTIONS ABOUT THE IML ACTION

Q: Does a parallel program always run faster than a serial program?

A: Not necessarily, but here are three tips that can help you write efficient parallel programs:

1. There are overhead costs associated with running a parallel function. You should ensure that the amount of work performed in each thread is large relative to those overhead costs.
2. In MPP mode, the overhead costs depend on the amount of data sent back and forth between the controller and worker nodes. Sending large amounts of data to and from worker nodes can severely degrade the performance.
3. When overhead costs are negligible, the time required to run k tasks in parallel is approximately equal to the time for the longest-running task. Therefore, the greatest possible speedup occurs in distributing homogeneous tasks that each take about T seconds. In an ideal world, running the tasks in parallel in k threads requires about T seconds of real time. In contrast, running the tasks sequentially in a single thread requires about kT seconds. For more information about performance and speedup in parallel computations, see (Cohen 2002).

Q: I've heard that CAS data tables do not have an intrinsic order. Can you order a table when you read it into a SAS/IML matrix in the **iml** action?

A: Yes, if the table contains a variable that specifies the sort order. The MatrixCreateFromCAS function creates a SAS/IML matrix from a CAS data table. If the table contains a variable named **_ROWID_** or **_ROWORDER_**, data are sorted by that variable when the matrix is assigned. If order is important, define one of these ordering variables when you upload data to CAS. Also, some actions support the **addRowOrder** parameter. If the parameter of that parameter is True, then the action adds a **_ROWORDER_** variable to the data table. For more information, see the documentation for the MatrixCreateFromCAS function in the *SAS IML: Programming Guide*.

Q: The "I" in "IML" stands for "interactive." Is the **iml** action interactive?

A: No, unfortunately not. If the program contains an error, the action terminates. For complicated programs, you might want to use PROC IML to debug and test your program.

Q: Can you control the order of output from threads? The output in Figure 3 of this paper is in a strange order!

A: No. The output is written when a thread completes its task. In a distributed environment, you cannot guarantee the order in which the nodes and threads will finish a task. However, the PRINT statement appends a message that specifies the node and thread number that created the output. You can also use the NODEINFO function to append the output with the node and thread IDs.

Q: It looks like lists play a big part in the functions that support parallel programming. I've never used lists before. How can I learn more?

A: Lists are relatively new objects that were introduced in SAS/IML 14.2. See (Wicklin 2017) for an overview of lists. For details, see the "Lists and Data Structures" chapter in the *SAS/IML User's Guide*.

Q: Are random numbers in parallel programs reproducible?

A: Yes and no. In a session that has N worker nodes, each running k threads, a program always generates the same set of random numbers. However, you will get a different set of random numbers if you run the program in a session that has a different number of nodes or threads. For more information, see the section "Generate Random Numbers in Parallel" in the *SAS IML: Programming Guide*.

Q: Are there any restrictions on calling the parallel functions?

A: Yes, you cannot call a parallel function from within another parallel function. For example, if you are using the MAPREDUCE function, the mapper function cannot call the PARTASKS function. The **iml** action detects this situation and issues an error.

Q: Does the **`iml`** action support the SAS macro language?

A: Technically, no action supports the SAS macro language, which is implemented by a preprocessor step that runs only on a SAS client. Because PROC CAS runs on a SAS client, you can use the SAS macro language when you submit a SAS/IML program that is defined in a SOURCE/ENDSOURCE block. The macro preprocessor will perform text substitution on the SOURCE/ENDSOURCE block before the program is sent to the CAS server. However, if you call an action from Python, Lua, or R, then you cannot use SAS macros because those languages do not support the SAS macro preprocessor.

REFERENCES

Cohen, R. (2002). "SAS Meets Big Iron: High Performance Computing in SAS Analytical Procedures." In *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*. Cary, NC: SAS Institute Inc. `http://www2.sas.com/proceedings/sugi27/p246-27.pdf`.

SAS Institute Inc. (2019). *SAS IML: Programming Guide*. Cary: SAS Institute Inc. `https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=casactiml&docsetTarget=titlepage.htm&locale=en`.

Wicklin, R. (2013). *Simulating Data with SAS*. Cary, NC: SAS Institute Inc.

Wicklin, R. (2017). "More Than Matrices: SAS/IML Software Supports New Data Structures." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. `https://support.sas.com/resources/papers/proceedings17/SAS0420-2017.pdf`.

ACKNOWLEDGMENTS

Some of the ideas in this paper also appear in the *SAS IML: Programming Guide*, which was written by the authors.

CONTACT INFORMATION

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

Rick Wicklin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513