Paper SAS1935-2015

Divide and Conquer— Writing Parallel SAS® Code to Speed Up Your SAS Program

Doug Haigh, SAS Institute Inc., Cary, NC

ABSTRACT

Being able to split SAS® processing over multiple SAS processers on a single machine or over multiple machines running SAS, as in the case of SAS® Grid Manager, enables you to get more done in less time. This paper looks at the methods of using SAS/CONNECT® to process SAS code in parallel, including the SAS statements, macros, and PROCs available to make this processing easier for the SAS programmer. SAS products that automatically generate parallel code are also highlighted.

INTRODUCTION

SAS/CONNECT software is a SAS client/server toolset that provides scalability through parallel SAS processing. By providing the ability to manage, access, and process data in a distributed and parallel environment, SAS/CONNECT enables users and applications to get more done in less time. This paper gives a high-level overview on how to start with simple remote execution of code on one machine and work its way to sophisticated parallel execution on the grid.

The programming techniques described in this paper are used in a number of SAS products. The 'Loop' transform in SAS® Data Integration Studio breaks up ETL processing into parallel SAS/CONNECT processes. SAS® Enterprise Miner™ will process parallel nodes in a diagram using parallel SAS/CONNECT processes.

Also, you can use the SAS® Code Analyzer (via 'Analyze for Grid Computing' action in SAS® Enterprise Guide® or via PROC SCAPROC directly) as a good first step to convert existing SAS programs to process in parallel. Generally, the SAS Code Analyzer completes 80% of the parallelization process with 20% left to be done manually.

BACKGROUND

SAS/CONNECT has the following two key statement combinations that are used to enable parallel processing:

- SIGNON/SIGNOFF to connect to and disconnect from a remote SAS process
- RSUBMIT/ENDRSUBMIT to specify the SAS code to be sent for execution in the remote SAS process

SAS/CONNECT links a SAS client session to a SAS server session. The client session is the initial SAS session that creates and manages one or more server sessions. The server sessions can run either on the same computer as the client (for example, an SMP computer) or on a remote computer across a network. Both of these statement groups have the ability to run asynchronously, enabling the SAS/CONNECT client to do other work while the SAS/CONNECT server is busy initializing or processing SAS code.

EXECUTION LOGIC

SIMPLE SIGNON

First, we will perform a `SIGNON` to spawn a SAS/CONNECT server on the same machine, submit a simple DATA step, wait for it to complete, and terminate the SAS/CONNECT server.

```
SIGNON mySess sascmd="!sascmd";
RSUBMIT mySess;
  LIBNAME myLib <path>;
  data myLib.myData; x=1; run;
ENDRSUBMIT;
SIGNOFF mySess;
```

This code starts a SAS/CONNECT server using the same command that was used to start the SAS/CONNECT client via the 'sascmd="!sascmd"' option to SIGNON. The SIGNON statement does not complete until the SAS/CONNECT server has started and connected back to the client. Once it has completed, all SAS code between the RSUBMIT and ENDRSUBMIT statements is sent to the server for processing. The RSUBMIT/ENDRSUBMIT block does not complete until the processing has completed on the server. Lastly, the SIGNOFF statement terminates the SAS/CONNECT server.

It should be noted that there are the following SAS options that can be used to combine SAS/CONNECT statements, although for the purposes of this paper they are not used:

- AUTOSIGNON automatically performs a SIGNON when a RSUBMIT is executed and the remote session is not currently connected.
- NOCONNECTPERSIST automatically performs a SIGNOFF when a RSUBMIT has completed.

MULTIPLE SIGNONS

Synchronous Execution

The first example was not very exciting and did not help speed up processing, but it was a good first step to illustrate the concepts. Next, a second SIGNON and RSUBMIT/ENDRSUBMIT block is added as the next step toward performing work in parallel.

```
SIGNON mySess1 sascmd="!sascmd";
SIGNON mySess2 sascmd="!sascmd";

RSUBMIT mySess1;
  LIBNAME myLib <path>;
  data myLib.myData1; x=1; run;
ENDRSUBMIT;

RSUBMIT mySess2;
  LIBNAME myLib <path>;
  data myLib.myData2; x=1; run;
ENDRSUBMIT;

SIGNOFF mySess1;
SIGNOFF mySess2;
```

As you can see, just adding a second set of statements does not create parallel or asynchronous execution without some additional help. As the code is now written, the second SIGNON waits for the first SIGNON to finish. Likewise, the first RSUBMIT/ENDRSUBMIT block waits for both SIGNON statements to finish before it runs, and the second RSUBMIT/ENDRSUBMIT waits for everything to finish before it runs. To make execution happen in parallel, additional options need to be added to both the SIGNON and RSUBMIT statements to tell them to return control to the client while the statements process in the background.

Asynchronous Execution

To make processing occur in the background (that is, asynchronously), an option is added to each set of statements: `SIGNONWAIT` for the `SIGNON` statement and `CONNECTWAIT` (or just `WAIT`) for the `RSUBMIT` statement. The highlighted options change the code as follows:

```
SIGNON mySess1 sascmd="!sascmd" SIGNONWAIT=NO;
SIGNON mySess2 sascmd="!sascmd" SIGNONWAIT=NO;

RSUBMIT mySess1 WAIT=NO;
  LIBNAME myLib <path>;
  data myLib.myData1; x=1; run;
ENDRSUBMIT;

RSUBMIT mySess2 WAIT=NO;
  LIBNAME myLib <path>;
  data myLib.myData2; x=1; run;
ENDRSUBMIT;

SIGNOFF _ALL_;
```

This simple change allows the `SIGNON` for `mySess1` to be processing at the same time as the `SIGNON` for `mySess2`. It also allows the `RSUBMIT` for `mySess1` to be processing at the same time as the `RSUBMIT` for `mySess2,` so now `mySess1` and `mySess2` are processing in parallel. The change in the `SIGNOFF` statement allows it to terminate all existing remote sessions in an asynchronous manner when the `RSUBMIT` statements have completed.

REUSING SESSIONS

The above example is fine, but what if you want to do some processing in parallel, and then do some more processing based on the output of the parallel processing? To do that, you would need to wait for the parallel processing to complete using the `WAITFOR` statement as highlighted in the following example:

```
SIGNON mySess1 sascmd="!sascmd" SIGNONWAIT=NO;
SIGNON mySess2 sascmd="!sascmd" SIGNONWAIT=NO;

RSUBMIT mySess1 WAIT=NO;
  LIBNAME myLib <path>;
  data myLib.myData1; x=1; run;
ENDRSUBMIT;

RSUBMIT mySess2 WAIT=NO;
  LIBNAME myLib <path>;
  data myLib.myData2; x=2; run;
ENDRSUBMIT;

WAITFOR _ALL_ mySess1 mySess2;

RSUBMIT mySess1 WAIT=NO;
  data myLib.myData3;
    set myLib.myData1, myLib.myData2;
  run;
ENDRSUBMIT;
SIGNOFF _ALL_;
```

In this example, the WAITFOR statement uses the _ALL_ option to wait for both remote sessions to complete before continuing to the next step. The next step is an RSUBMIT statement that allows the code to reuse mySess1 for additional processing.

Reusing an Available Session

Simple asynchronous processing has been illustrated up to this point where the next session used is specified in the code. A more advanced technique would be to optimize the code so that the next available session gets used so that the code will start executing as quickly as possible. To do that you have to know the status of each session that can be done using the CMACVAR option on the RSUBMIT statements. The macro variable specified by CMACVAR will contain a value that indicates the status of the RSUBMIT statement.

The code that follows shows the RSUBMIT statements being used with the highlighted CMACVAR option to determine when an RSUBMIT completes, and enable the next code submission to immediately use the available session.

```
SIGNON mySess1 sascmd="!sascmd" SIGNONWAIT=NO;
…
SIGNON mySessN sascmd="!sascmd" SIGNONWAIT=NO;

RSUBMIT mySess1 WAIT=NO CMACVAR=myVar1;
    <code to submit for 1st session>
ENDRSUBMIT;

…

RSUBMIT mySessN WAIT=NO CMACVAR=myVarN;
    <code to submit for Nth session>
ENDRSUBMIT;

WAITFOR _ANY_ mySess1 mySess2 … mySessN;

%determineAvailableSession(N);

RSUBMIT mySess&openSessID WAIT=NO CMACVAR=myVar&openSessID;
…
ENDRSUBMIT;

SIGNOFF _ALL_;
```

In the previous example, the WAITFOR statement uses the _ANY_ option to wait for just **one** session to finish. When a session has finished, the code calls the macro determineAvailableSession that returns the next available session ID in the macro variable openSessID. The openSessID macro variable can then be used on the next RSUBMIT call in the session name and RSUBMIT macro variable name so that the code can execute immediately.

To determine which session to use, the determineAvailableSession macro simply goes through all the RSUBMIT macro variables looking for one that indicates the RSUBMIT has completed as shown below:

```
%macro determineAvailableSession(numSessions);
  %global openSessID;
  %do sess=1 %to &numSessions;
```

```
      %if (&&myVar&sess eq 0) %then
      %do;
        %let openSessID=&sess;
        %let sess=&numSessions;
      %end;
    %end;
  %mend;
```

Waiting for the Best Available Session

The previous code example is more asynchronous, but the initial `RSUBMIT` statements must wait for the specific session to complete its `SIGNON`. In grid environments, some `SIGNON` statements might complete while others remain pending due to lack of resources. The first `RSUBMIT` statement for a session that has not completed its `SIGNON` waits until the `SIGNON` completes. Since the `RSUBMIT` does not complete, all SAS processing that follows is blocked until the `SIGNON` completes. Also, if a `SIGNON` fails, then all `RSUBMIT` statements that use that specific session will also fail. You can eliminate these problems by using the `CMACVAR` option on the `SIGNON` statement to always submit to the first available remote session. You can now change the code to look like the following:

```
%initVars(N);

SIGNON mySess1 sascmd="!sascmd" SIGNONWAIT=NO CMACVAR=mySignonVar1;
…
SIGNON mySessN sascmd="!sascmd" SIGNONWAIT=NO CMACVAR=mySignonVarN;

%waitForAvailableSession(N);

RSUBMIT mySess&openSessID WAIT=NO CMACVAR=myRsubmitVar&openSessID;
    <code to submit for session>
ENDRSUBMIT;

%waitForAvailableSession(N);

RSUBMIT mySess&openSessID WAIT=NO CMACVAR=myRsubmitVar&openSessID;
    <code to submit for session>
ENDRSUBMIT;

…

%waitForAvailableSession(N);

RSUBMIT mySess&openSessID WAIT=NO CMACVAR=myRsubmitVar&openSessID;
    <code to submit for session>
ENDRSUBMIT;

SIGNOFF _ALL_;
```

In the previous example, the `waitForAvailableSession` macro acts like a combination of the `WAITFOR` statement and the `determineAvailableSession` macro, but instead of waiting just for `RSUBMIT` statements to complete, it also waits for `SIGNON` statements to complete. Once a session has completed a `SIGNON` and is not processing a `RSUBMIT`, the macro returns the session ID in the `openSessID` macro variable.

To do this, the `waitForAvailableSession` macro goes through all `SIGNON` and `RSUBMIT` macro variables looking for a session where both the `SIGNON` has completed and the `RSUBMIT` has completed.

If it does not find one, it sleeps for a bit before trying again. When it finds one, it returns the session ID in the macro variable `openSessID` as shown below:

```
%macro waitForAvailableSession(numSessions);
  %global openSessID;
  %let sessFound=0;
  %do %while (&sessFound eq 0);
    %do sess=1 %to &numSessions;
      %if (&&mySignonVar&sess eq 0) %then
        %if (&&myRsubmitVar&sess eq 0) %then
          %do;
            %let openSessID=&sess;
            %let sessFound=1;
            %let sess=&numSessions;
          %end;
    %end;
    %if (&sessFound eq 0) %then %let rc=%sysfunc(sleep(60));
  %end;
%mend;
```

To make the `waitForAvailableSession` macro work properly, the macro variables used by the `SIGNON` and `RSUBMIT` are initialized to a known value at the start. The `initVars` macro does this in the previous examples as shown below:

```
%macro initVars(numSessions);
  %do sess=1 %to &numSessions;
    %global mySignonVar&sess;
    %global myRsubmitVar&sess;
    %let mySignonVar&sess=3;
    %let myRsubmitVar&sess=0;
  %end;
%mend;
```

ASYNCHRONOUS PROCESSING AT ITS BEST

While the code that reuses the best available session is much better than the initial code, this code still is not optimal. There is nothing in the code to handle problems when a `SIGNON` fails or a `RSUBMIT` fails. There is nothing in the code to terminate sessions that are no longer needed. What would be ideal is having code that does the following:

1. Start as many asynchronous `SIGNON` statements as needed for the number of sessions.
2. Wait for `SIGNON` or `RSUBMIT` to complete by checking macro variable values and sleeping if none is available.
    a. If a session's `SIGNON` fails, retry the `SIGNON`.
    b. If a session's `RSUBMIT` fails, retry the `RSUBMIT`.
    c. If a session's `SIGNON` and `RSUBMIT` are complete, `RSUBMIT` more code to the available session.
3. Repeat until all `RSUBMIT` code blocks are done.
4. If a session is available, but no more `RSUBMIT`s are needed, then `SIGNOFF` that session.

Of course, this requires that the code that gets sent in each `RSUBMIT` statement is able to run on any machine and in any order. Programming this processing logic, along with the logic required to have SAS code that can execute in any order can be difficult, but there is help in the form of the %Distribute macro.

The %Distribute Macro

Initially developed by Cheryl Doninger and Randy Tobias of SAS Institute Inc. and detailed in their paper titled “The %Distribute System for Large-Scale Parallel Computation in the SAS System,” the %Distribute macro has been enhanced to be able to use either SAS/CONNECT servers spawned on the same machine or spawned on a grid using SAS Grid Manager.

The %Distribute macro’s job is to handle the initialization of SAS/CONNECT servers, distribution of SAS code blocks to those servers, and termination of the servers once all code blocks have been processed. The %Distribute macro also handles all error and retry logic for the servers and code blocks. To do this, the %Distribute macro breaks parallel tasks into the following two parts:

- Code that runs on a host when a `SIGNON` statement completes (`FirstRSub`). This is usually a one-time initialization to do things such as setting LIBNAME statements for libraries used by the code blocks.
- Code that runs for each iteration or subtask (`TaskRSub`). This code will be run on each host for every iteration. The %Distribute macro provides the subtask code with information about the iteration number, the total number of iterations, the number of iterations in a task, and the number of iterations to be processed in this subtask. Also, the subtask will be told the total number of subtasks that will be processed along with the subtask number of this current subtask.

The parallel SAS program then consists of the following:

1. creating the `RInit` macro to instantiate the `FirstRSub` and `TaskRSub` macros on the servers
2. setting up `%Distribute` parameters
3. calling `%Signon` to initialize the connection to the grid
4. calling `%Distribute` to perform the parallel processing
5. processing the results
6. calling `%Signoff` to terminate all servers

An example program follows along with its output.

SAS Code

```
/** Load in the %Distribute macro **/
%inc "GridDistribute.sas";

/** Define the code to execute on the grid nodes **/
%macro RInit;
    rsubmit remote=Host&iHost wait=yes macvar=Status&iHost;

        /*-------------------------------------------------------------
        ** This macro initializes the host, preparing it to run
        ** the fundamental task multiple times. In this case,
        ** all you need to do is initialize the data set in
        ** which all the results for this host are to be collected.
        */
        %macro FirstRSub;
            options nonotes;
            data Sample;        /* Create an empty dataset */
                if (0);
            run;
        %mend;

        /*------------------------------------------------------------
        ** This macro defines the fundamental task. SAS/IML® software
        ** is used to perform the basic simulation the number of
        ** times (&rem_NIter) required. Each chunk of results
        ** is created in a work data set called Sample, and all
        ** the results for this host are collected in
        ** Results.Sample&rem_iHost.
        */
        %macro TaskRSub;
            proc iml;

            x = rannor(&rem_Seed);

            /** For each job in this chunk, create random data,
                compute min eigenvalue, and save it. **/
            free data;
            do i = 1 to &rem_Niter;
                x = rannor( (1:&rem_Dimen)`*(1:&rem_Dimen));
                e = eigval(x`*x)[><];
                data = data // e;
                end;

            /** Save whole sample for this chunk. **/
            create iSample var {E1};
            append from data;
            quit;

            /** Append this sample to the accumulated set of all
                samples created on this host. */
            data Sample; set Sample iSample;
            run;
        %mend;
    endrsubmit;
%mend;

/** If you have macro variables to pass to server **/
%macro Macros;
    %syslput rem_Dimen=&Dimen;
%mend;

/** Additional %Distribute parameters **/
%let DistLog = <local_path>/distribute.log;
%let DistLst = <local_path>/distribute.lst;

/** Metadata server options to use for logical grid server
    If not specified, %Distribute will run locally.
options metaserver = "myhost.mycompany.com";
options metaport   = 8561;
options metauser   = myuserid;
options metapass   = "mypassword";

%let GridAppServer=SASApp;
**/
```

```
/** Initialize %Distribute's connection to the grid **/
%put Signing on...;
%Signon;

/** Perform the distributed computation for a
    10-dimensional problem of 100,000 samples
    where each RSUBMIT creates 200 samples. **/
%put Processing...;
%let NIter    = 2000;
%let NIterAll = 100000;
%let Dimen = 10;

%Distribute;

/** Special macro to gather data from all servers **/
%RCollect(Sample,Sample / view=Sample);

/** Print out mean of samples generated **/
proc means data=Sample n mean stderr lclm uclm;
    var e1;
run;

/** (This code bypassed to save time & output)
%macro MinEval;
    options nonotes;
    data MinEval; if (0); run;
    %do Dimen = 1 %to 30;
        %Distribute;
        %RCollect(Sample,Sample / view=Sample);
        proc summary data=Sample;
            var e1;
            output out=_temp mean=Mean StdErr=StdErr LCLM=LCLM UCLM=UCLM;
        data _temp; set _temp; Dimen = &Dimen;
        data MinEval; set MinEval _temp;
        run;
    %end;
    options notes;
%mend;

%MinEval;
**/

/** Terminate servers **/
%put Signing off...;
%Signoff;
```

SAS Log Output

When run, the output shows how many servers are created, their activity, the progress of the job, and statistics on the run.

```
Signing on...
GridDistribute: Grid is NOT Enabled - spawning locally.
GridDistribute: Maximum number of nodes is 4
Processing...
GridDistribute: Signing on to Host #1
GridDistribute: Signing on to Host #2
GridDistribute: Signing on to Host #3
GridDistribute: Signing on to Host #4
Stat: [0:00:00] ???? (0/0) 100000
GridDistribute: Host #1 is local machine     (or hostname of execution host in a grid)
GridDistribute: Host #2 is local machine
GridDistribute: Host #3 is local machine
GridDistribute: Host #4 is local machine
Stat: [0:00:02] !!!. (0/0) 100000
Stat: [0:00:02] .... (8000/0) 100000
Stat: [0:00:05] !!!! (8000/8000) 100000
Stat: [0:00:05] !... (16000/8000) 100000
     <similar lines deleted>
Stat: [0:00:14] !... (94000/86000) 100000
Stat: [0:00:14] !!!! (94000/94000) 100000
Stat: [0:00:14] ...! (100000/94000) 100000

               Start time:                       15:09:24
               End Time:                         15:09:41
               Total elapsed time:                0:00:16
               Sum total time running:            0:00:51
               Sum total time in job:             0:00:03
               Per job time running:              0:00:01
               Per job time in job:               0:00:00
               Per host time running:             0:00:51
               Per host time in job:              0:00:03
               Percent speed increase:              221%
               Percent speed increase in job: (      80%)

NOTE: DATA STEP view saved on file WORK.SAMPLE.
NOTE: A stored DATA STEP view cannot run under a different operating
      system.
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.03 seconds

NOTE: View WORK.SAMPLE.VIEW used (Total process time):
      real time           0.08 seconds
      cpu time            0.17 seconds

NOTE: There were 26000 observations read from the data set RWORK1.SAMPLE.
NOTE: There were 26000 observations read from the data set RWORK2.SAMPLE.
NOTE: There were 26000 observations read from the data set RWORK3.SAMPLE.
NOTE: There were 22000 observations read from the data set RWORK4.SAMPLE.
NOTE: There were 100000 observations read from the data set WORK.SAMPLE.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.10 seconds
      cpu time            0.17 seconds


Signing off...
```

CONVERTING PARALLEL PROGRAMS TO USE SAS GRID MANAGER

While it is recommended that users initially get their parallel SAS program working using a few locally spawned SAS/CONNECT servers, it only takes one new statement to convert the program to use SAS Grid Manager (assuming you already have a SAS® Metadata Server connection). If you are not using the %Distribute macro, then the easiest way to do this is to have two sets of options: one for running local and one for running with SAS Grid Manager. The macro would look something like the following:

```
%macro setOpts(useGrid);
  %if (&useGrid eq 1) %then
    %do;
      /* Metadata connection options if needed
        options metaserver="<metadata_server>";
        options metaport=<metadata_port>;
        options metauser="<metadata_user>";
        options metapass="<metadata_password>";
      */
      %put gridRC=
         %sysfunc(grdsvc_enable(_ALL_,
                                server="<SAS_application_server>"));
    %end;
  %else
    %do;
      options SASCMD="!sascmd";
    %end;
%mend;

%setOpts(1);  * Will run everything on the grid, 0 will run local;

SIGNON mySessN SIGNONWAIT=NO;
…
SIGNOFF _ALL_;
```

ADDITIONAL CONSIDERATIONS

Knowing how to write parallel code is a significant first step to speeding up SAS processing, but it is only part of the process. Careful thought must be given to how information is moved between the client and the servers, where the servers will get their data, and where output data, logs, and list files should be persisted. Also, care should be taken when using `RSUBMIT` statements inside a macro.

Macro Variable Movement

Often it is required to move macro variable values between the client and server sessions. This can be done by using the `%SYSLPUT` statement to pass the values from the client session to the server session or using the `%SYSRPUT` statement to move the values from the server session to the client session.

Data Movement

Because most of the time spent running SAS programs comes from data I/O, it is best to make the data always available to all places the server can run rather than moving data from client to server. For SAS/CONNECT servers spawned on the same machine, you would need to make sure you do not try to consume more I/O than the machine can handle. When running with SAS Grid Manager, I/O connections that will support simultaneous throughput of 75-100MB/s per core are required so that a shared file system should be used.

For small data sets, SAS/CONNECT provides a couple of options for moving data between the client and server, namely, Data Transfer Services or Remote Library Services. Data Transfer Services moves entire data sets, libraries, or files between client and server through the `PROC UPLOAD` or `PROC DOWNLOAD` statements. Remote Library Services allows a DATA step to access either a client's data from the server or a server's data from the client. A client can access data on a server using the `server=` option on a

`LIBNAME` statement. A server can access data on a client using the `INHERITLIB` option on a `SIGNON` or `RSUBMIT` statement.

LOGS and LISTING Output

By default, SAS/CONNECT routes all logs and listing output back to the client to be displayed in the client's session. Since it might be difficult to parse through a large log of parallel process output, it might be better to route each server's log to a shared location so that it is easier to debug if problems occur.

RSUBMIT and the SAS Macro Facility

Special care must be taken when using an `RSUBMIT` statement inside a macro since the code inside a `RSUBMIT` statement is executed on the server, but macro code can be compiled on the client. Often, programs will try to `RSUBMIT` macro statements outside a macro definition and find out they get executed on the client instead of the server. This tends to happen on `%do`, `%if`, `%let`, `%put`, and `%sysrput` statements that exist outside a macro definition. To fix this problem, change the '`%`' to '`%nrstr(%%)`' so that the statement is not seen as a macro statement on the client.

For example, the following code

```
RSUBMIT mySess;
   %SYSRPUT clientVar=&serverVar;
ENDRSUBMIT;
```

would need to be changed to either

```
RSUBMIT mySess;
   %nrstr(%%)SYSRPUT clientVar=&serverVar;
ENDRSUBMIT;
```

or

```
RSUBMIT mySess;
   %macro updateVar;
      %SYSRPUT clientVar=&serverVar;
   %mend;
   %updateVar;
ENDRSUBMIT;
```

CONCLUSION

Executing SAS code in parallel can provide significant performance improvements both locally on an SMP machine or distributed across a grid using SAS Grid Manager. While there are a number of SAS products that generate parallel code, with a little practice, a SAS programmer can easily convert code to run in parallel.

REFERENCES

Doninger, C. and R. Tobias. 2002. "The %Distribute System for Large-Scale Parallel Computation in the SAS® System." Available http://support.sas.com/rnd/scalability/papers/distConnect0401.pdf.

"The %Distribute Macro (GridDistribute.sas); SAS Grid Computing Toolbox." Available http://support.sas.com/rnd/scalability/grid/download.html.

RECOMMENDED READING

- *SAS/CONNECT® 9.4 User's Guide.*

  Available http://support.sas.com/documentation/onlinedoc/connect/index.html.

- *Grid Computing in SAS® 9.4.*

  Available http://support.sas.com/documentation/onlinedoc/gridmgr/index.html.

CONTACT INFORMATION

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

Doug Haigh
100 SAS Campus Drive
Cary, NC 27513
SAS Institute, Inc.
Doug.Haigh@sas.com
http://www.sas.com