A callback-based approach for parallel processing

Yong Cao, PPD, part of Thermo Fisher Scientific

Abstract

With SAS/CONNECT module, SAS programmers are enabled to use multiple SAS sessions to run a single program faster than it would take in a single SAS session. There are articles on how to use SIGNON and RSUBMIT to asynchronously execute different parts of a program. Examples in these articles are often too trivial. There are no off-the-shell solutions that are easy to use and accessible to the wider user group of statistical programming.

This paper introduces a callback-based approach for parallel processing. This approach transforms the code that needs parallel processing to a %Do-Loop and hands over that loop to a macro (%AsyncLoop) which executes iterations in parallel. Use of %AsyncLoop requires users to create and pass another macro (%__Iteration__) that %AsyncLoop will invoke during each iteration. A few different examples are used to illustrate how this approach works and how to deal with code that is not originally a loop. %AsyncLoop supports both MP Connect and SAS Grid to fully use the hardware. Macros and macro variables available in client session are passed to server sessions so that user does not need to manually submit the definitions of macros again. At the end of the paper, factors that affect performance of asynchronous executions are discussed.

With the ever-increasing complexity of statistical models and magnificence of data to work with, statistical SAS programmers often find themselves dealing with programs with intolerable execution time. The solution introduced in this paper shields the technical details of parallel processing so that regular users can write asynchronous code easily. The callback-based approach is proven to be both universal and of great flexibility.

Keywords: Asynchronous; Parallel Processing; SAS Grid; SAS/CONNECT; RSUBMIT; SIGNON;

Terms

Client Session	The SAS session that creates and manages one or more server sessions.
Server Session	SAS session created by client session. Server session can run either on the same computer as the client (MP Connect) or on separate hardware (SAS Grid).
MP Connect	Multi-Process Connect enables server sessions and client sessions running in parallel. Nowadays our CPU usually has multicores, so a single machine can run both client session and server sessions.
Parallel Processing	Parallel processing is a method of simultaneously breaking up and running program tasks on multiple microprocessors, thereby reducing processing time. Parallel processing may be accomplished via a computer with two or more processors or via a computer network.
Asynchronous	In computer programming, asynchronous execution refers to the occurrence of events independent of the main program flow and ways to deal with such events.
Synchronous	Codes are executed in linear order.

Asynchronous execution in SAS

Leonid Batkhan in his blog “Running SAS programs in parallel using SAS/CONNECT” excellently explained how to use SIGNON and RSUBMIT to run different parts of a program in parallel by using the example below (slightly updated). The two DATA steps below are independent of each other. Each of them takes a few minutes. By running them parallelly, the final execution time is a little more than the maximum execution time of the two DATA steps (the first one). This is “the magic of 3+2=3" in the blog.

Listing 1 Example from Lenoid's blog

```
libname SASDL 'C:\temp';

/* Specifies the command that starts a server session on a multiprocessor computer */
options sascmd="sas";

/* Current datetime */
%let _start_dt = %sysfunc(datetime());

/* Prosess 1 */
signon task1 inheritlib=(SASDL);
/* wait=no specifies that the RSUBMIT blocks execute asynchronously */
rsubmit task1 wait=no;

   data SASDL.DATA_A (keep=str);
      length str $1000;
      do i=1 to 1150000;
         str = '';
         do j=1 to 1000;
            str = cats(str,'A');
         end;
         output;
      end;
   run;

endrsubmit;

/* Process 2 */
signon task2 inheritlib=(SASDL);
rsubmit task2 wait=no;

   data SASDL.DATA_B (keep=str);
      length str $1000;
      do i=1 to 750000;
         str = '';
         do j=1 to 1000;
            str = cats(str,'B');
         end;
         output;
      end;
   run;

endrsubmit;

waitfor _all_;
/* Merge SAS log and outputs from server sessions to client session */
rget task1 task2;
signoff _all_;

/* Print total duration */
data _null_;
   dur = datetime() - &_start_dt;
   put 30*'-' / ' TOTAL DURATION:' dur time13.2 / 30*'-';
run;
```

The key takeaways from this example are:

- Use multiple SIGNON statements with different server IDs (task1 and task2 in the example above) to start multiple server sessions

- INHERITLIB option can be used to pass SAS libraries from client session to server sessions
- Use WAIT=NO option in RSUBMIT statement to execute code asynchronously on a specified remote server
- Use WAITFOR to wait for all asynchronous executions to finish running other parts of the SAS code in the client session that is dependent on the execution of server session code
- Use RGET to merge SAS log and output from server session to client session so that the final client session log and output is complete

In the blog SYSLPUT and SYSRPUT are also introduced. The former passes macro variables from client session to server session and the latter works the other way around.

What do we need to extend from the example above?
Notice that the two DATA steps are quite similar and only differ in number of iterations in the outer do-loop. We would therefore want to write a macro and expose the number of iterations as a parameter. Then replace the two DATA steps with two invocations of that macro. Below is the simplified code.

Listing 2 Modified version that does not work

```
%macro iotest(iterations=, postfix=);
data SASDL.DATA_&postfix (keep=str);
   length str $1000;
   do i=1 to &iterations.;
      str = '';
      do j=1 to 1000;
         str = cats(str,'A');
      end;
      output;
   end;
run;
%mend iotest;

signon task1 inheritlib=(SASDL);
rsubmit task1 wait=no;

   %iotest(iterations=1150000, postfix=A);

endrsubmit;

signon task2 inheritlib=(SASDL);
rsubmit task2 wait=no;

   %iotest(iterations=750000, postfix=B);

endrsubmit;

waitfor _all_;
signoff _all_;
```

However, if we submit this code, SAS will complain %IOTEST does not exist. Macros in the client session are not passed to server sessions. The obvious solution to this, as some suggested, is to include the macro definition in the two RSUBMIT code blocks. But it loses the point of creating such macro. And if we were to update an existing program, we may have to change the program structure by moving code blocks around and create several copies of some code. If a macro needs to be called in both client session and server sessions, we will have to duplicate that macro. The other solution is to move these macros to a separate program file and %include them in RSUBMIT code block. This again needs us to heavily change the existing code. In some cases, we may prefer a single-program-code solution.

The other issue of the example is that it uses the same amount of server sessions for pieces of code that need parallel processing. For simple cases like the example, where only two RSUBMITs are needed, it is not a problem. Imagine that we are developing a tool that compares a set of datasets with user specified key variables and creates customized Excel reports to show new/changed/deleted records and highlights the updates. Obviously, there is a loop on datasets. Each iteration is independent of each other and may take considerable time if the underlying dataset is large enough. It is a perfect scenario where parallel processing comes handy for reducing the overall execution time. For such use cases it is not proper to use one server session per dataset. Otherwise, we could need hundreds of server sessions which could crash the machine (or machines in SAS Grid environment) SAS is running on. In addition, we also want our program to be scalable. That is, to be able to increase and reduce the number of server sessions depending on the scale of the problem.

Lastly, it is a bit verbose to insert all these SIGNON and RSUBMIT into our code. The big overheads scare people who are not familiar with them but still want to take advantage of parallel processing capabilities. It also makes our code less portable. For example, our program needs to be ported to another environment where SAS/ACCESS module is not available.

Our model for parallel processing

Still use the example of data comparison tool mentioned earlier. It's straightforward to use a %DO-LOOP to iterate all datasets. Our goal is to run iterations in parallel. It is easy to use the code Listing 3 to achieve our goal.

Listing 3 Asynchronous loop

```
%do i = 1 %to &ndset;
  signon task&i;
  rsubmit task&i wait=no /* inheritlib = () */;
  /* SAS Code to compare a single dataset and create Excel report */
  endrsubmit;
%end;
```

However, this code is no better than Listing 1. It creates just as many server sessions as the number of datasets to process. To meet the scalability target, we need to isolate SIGNON and RSUBMIT. Instead of using a dedicated server session for all iterations, we can change our strategy by creating fewer server sessions and sharing tasks (iterations) among these server sessions. For example, we can start 5 server sessions (whose IDs are ss1-ss5) to process 40 datasets. The immediate problem is to decide which server session to use for a given iteration. A given server session is either idle or occupied with a prior RSUBMIT. At any given time, there could be 0, or 1 or more idle server sessions. So, on top of Listing 3, we need to monitor the status of each server session.

CMACVAR= option in RSUBMIT statement indicates the status of a RSUBMIT. The value of this option is a macro variable name whose value is set by SAS (server session). 0 indicates RSUBMIT is completed, 1 indicates RSUBMIT was failed and 2 indicates RSUBMIT is ongoing. The trick here is to use a fixed name for CMACVAR for a given server session. For example, when RSUBMIT to server session ss2, we always specify CMACVAR=rstat2 in RSUBMIT regardless of what iteration it's for. Then corresponding to 5 server sessions (ss1-ss5), at any given time we only need to monitor 5 macro variables. Apparently, our status monitoring macro runs in client session within the loop before RSUBMIT statement. It loops through all status macro variables and returns the server ID whose status variable is 0. In case all server sessions are busy, it puts client session to sleep, i.e., to interrupt the %Do-Loop until a free server session is available.

We also mentioned that we want to be able to pass macros to the server session. For any server session this needs to be done only once. And it can be done through an asynchronous RSUBMIT as well. We do this immediately after SIGNON as initialization. We will talk about the details of passing macros from client session to server sessions later. Assuming we have the code to do this, our optimized version of Listing 3 will look like Listing 4. Note that the introduction of initialization simplifies the status monitoring macro because at first iteration those status macro variables already exist.

Listing 4 Optimized asynchronous version

```
/* Start server sessions and initialize them */
%do i = 1 %to &nserver;
  signon ss&i;
  rsubmit ss&i wait=no cmacvar=statss&i;
    /* SAS code to initializing this server, e.g. copying macros from client session */
  endrsubmit;
%end;

/* Dispatch tasks to server sessions */
%do i = 1 %to &ndset.;
  %let freeServerId = %GetFreeServer;
   /* Pass all user-created macro variables (global and local) from client session
    * to server sessions so that server sessions have access to them.
    */
  %syslput _user_ / remote=&freeServerId.;
  rsubmit &freeServerId wait=no cmacvar=stat&freeServerId;
    /* SAS Code to compare a single dataset and create Excel report */
  endrsubmit;
%end;
```

Here is a brief implementation of %GetFreeServer. We expose the number of servers and CMACVAR prefix as the parameters in this implementation. It iterates over all status variables and stops at the first 0. If a full scan does not find any 0, client session will sleep one second before another scan. The first %IF references a macro variable that is indicator of consecutive/synchronous mode. That is the portability consideration that we discussed. Several other macros do the same which we will talk about later.

Listing 5 Implementation of %GreeFreeServer

```
%macro getFreeServer(nServerSessions=, rsubmitStatusVarPrefix=);
%local __freeServerNo__;
%local serverno;

%if &__async_mode__. = 0 %then %do;
  -1
  %return;
%end;

%do %while(%length(&__freeServerNo__)=0);
  %do serverno = 1 %to &nServerSessions.;
    %if &&&rsubmitStatusVarPrefix.&serverno. = 0 %then %do;
      %let __freeServerNo__ = &serverno;
      &__freeServerNo__
      %return;
    %end;
  %end;
  %let rc = %sysfunc(sleep(1));
%end;
%mend getFreeServer;
```

In summary, Listing 4 represents an optimized model for parallel processing. Client session starts several server sessions and initialize them. These server sessions accept tasks (iterations) submitted from client session and execute them asynchronously. The macro %getFreeServer is the dispatcher running in the client session. It decides which iteration goes to which server sessions and interrupts the loop when the queue is full.

Callback-based approach

We made a promise to make our model universal. So, Listing 4 is better to be wrapped in a macro so that users can use it without knowing the implementation details. But Listing 4 also includes user codes (code that client session RSUBMIT to server sessions). For our dataset comparison tool, it's to compare SAS datasets and generate Excel reports. For other usage cases, it's obviously different. To be able to wrap Listing 4 in a macro while not wrapping any user code, it's intuitive to use a macro for actual user code. Given this macro is the code that is executed in a single iteration, it makes perfect

sense to call that macro %__iteration__ (the two underscores says that it's for special purpose). Server sessions call this macro in the second loop between RSUBMIT code block in Listing 4. If we analog SAS macros to functions in some programming languages that support functional programming (where functions can be passed as argument of another function), we are passing macros to another macro. This is usually called "callback". Listing 6 is updated from Listing 4 by introducing %__iteration__.

Note that %__iteration__ represents a single iteration and it has a positional parameter to allow server sessions to pass iteration number. Also note that this macro doesn't carry any information about number of iterations. The number of iterations is known by users and is passed to client session instead of server sessions.

Listing 6 __iteration__ macro

```
/* User needs to provide this macro */
%macro __iteration__(iternum);
/* SAS code to run a single iteration, which iteration to run is controlled by parameter iternum */
%mend __iteration__;

/* Our parallel processing model */

/* Start server sessions */
%do i = 1 %to &nserver;
  signon ss&i;
  rsubmit ss&i wait=no cmacvar=statss&i;
    /* SAS code to initializing this server, e.g. copying macros from client session */
  endrsubmit;
%end;

/* Dispatch tasks to server sessions */
%do __iternum__ = 1 %to &nIterations.;
  %let freeServerId = %GetFreeServer;
  %syslput _user_ / remote=&freeServerId.;
  rsubmit &freeServerId wait=no cmacvar=stat&freeServerId;
  /* Wrapper macro for __iteration__*/
  %macro __run__iteration__(iternum);
    /* Mask this variable from the outer scope so that %__iteration__
      cannot accidentally change it */
    %local __iternum__;
    %__iteration__(&iternum.)
  %mend __run__iteration__();
  %__run__iteration__(&__iternum__);
  endrsubmit;
%end;
```

One thing to note in Listing 6 is that %__iteration__ is wrapped in another macro (%__run__iteration__). This is for security consideration. It allows us to mask the iteration number macro variable (&__iternum__) so that %__iteration__ does not have written access to the original &__iternum__ (user may choose to pass macro variables back to client sessions using SYSRPUT). If the %__iteration__ had write access and accidentally changes &__iternum__, the job dispatch %DO-Loop could end up missing some of the iterations, or even worse – running indefinitely.

The last piece of our model is to deal with the SAS code that is not originally a loop. Listing 7 is transformed from Listing 1. It looks odd but it is a neat trick to extend the scope of our model to virtually any code that needs parallel processing.

Listing 7 An example of __iteration__ from non-loops

```
%macro __iteration__(iternum);
%if &iternum. = 1 %then %do;
  %iotest(iterations=1150000, postfix=A);
%end;
%else %if &iternum. = 2 %then %do;
  %iotest(iterations=750000, postfix=B);
%end;
%mend __iteration__;
```

Here is another example. The PROC MIXED in in Listing 8 takes more than 3 hours. BY statement produces 300 groups. The execution time of each of them ranges from a few seconds to a few minutes.

Listing 8 PROC MIXED with BY statement

```
proc mixed data=bcvatp;
  by tpshift mimpuseq;
  class trt01pn avisitn subjid;
  model chg2=base trt01pn avisitn trt01pn*avisitn/DDFM=KR CL;
  repeated avisitn/subject=subjid type=&cov.;
  lsmeans trt01pn*avisitn/diff cl;
  ods output convergencestatus=cnvg_status;
  ods output diffs=diffT;
run;
```

Listing 9 shows how to design %__iteration__ to run different by-groups of PROC MIXED in Listing 8 in parallel. It is a common design pattern of %__iteration__. First, a task dataset is created. This code is executed in the client session. Task dataset stores task information of all iterations. In this case, it's the value of 300 by-groups. Second, macro %__iteration__ references the task dataset by using libname REMOTE. That is the alias name for WORK library in the client session. %__iteration__ is executed in the server session. When we need to pass the WORK library of client session to server sessions, it needs to be "renamed" in the server session because all SAS sessions have their own WORK library. In the INHERITLIB option, it is specified as INHERITLIB=(WORK=REMOTE). The last thing to note is the two ODS OUTPUT statements. We appended iteration numbers in the output dataset names so that they are different on each iteration. This avoids the I/O conflict. It is an important thing to remember when writing code for asynchronous execution. It is best to avoid multiple sessions competing for writing to same files. After all iterations are executed, we can gather results of each iteration by appending these datasets together.

Listing 9 Use %__iteration__ for PROC MIXED

```
** Create task dataset (values of unique by variables);
proc sql;
  create table proc_mix_tasks as
  select distinct tpshift, mimpuseq
  from bcvatp;
quit;

** Record number of iterations;
%let nTask = &sqlobs;

%macro __iteration__(iternum);
data _null_;
  /* remote is WORK library of client session */
  set remote.proc_mix_tasks (obs=&iternum. firstobs=&iternum.);
  call symputx("tpshift", tpshift, 'L');
  call symputx("mimpuseq", mimpuseq, 'L');
run;

proc mixed data=remote.bcvatp;
  where tpshift=&tpshift.
    and mimpuseq=&mimpuseq.
  ;
  /* still use by statement to make sure tpshift and
   * mimpuseq are in the output dataset
   */
  by tpshift mimpuseq;
  class trt01pn avisitn subjid;
  model chg2=base trt01pn avisitn trt01pn*avisitn/DDFM=KR CL;
  repeated avisitn/subject=subjid type=&cov.;
  lsmeans trt01pn*avisitn/diff cl;
  ods output convergencestatus=remote._cnvg_status&iternum.;
  ods output diffs=remote._diffT&iternum.;
```

```
run;
%mend __iteration__;
```

So far, we have used three examples, an explicit loop (Listing 3), and two implicit loops (Listing 7 and Listing 9), to prove that our model of parallel processing works for both loops and non-loops.

Pass macros from client session to server sessions

Now that we have proven that our model is universal, we can continue to complete the model in Listing 4. The last topic we haven't covered is to pass macros from client session to server sessions.

For a SAS session to recognize macros that are not defined in that SAS session, source codes of these macros need to be stored in a location that is part of SASAUTOS option. Alternatively, they may be compiled as catalogs in which case they need to be copied to default macro catalog of the SAS session. So, our task is easy, to replicate SASAUTOS of client session on server sessions, and to copy macro catalog of client session to server sessions.

Listing 10 Pass macros from client session to server sessions

```
/* This macro runs on client session to parse SASAUTOS
 * option and create a dataset to store each fileref
 * or directory that SASAUTOS points. For filerefs,
 * actual physical location is resolved and stored.
 */
%macro parseSasAutos(outds);

%mend parseSasAutos;

/* This macro runs on server session. Input dataset
 * is created by %parseSasAutos. It uses this input
 * dataset to restore SASAUTOS in the server sessions.
 */
%macro setSasAutos(inds);

%mend setSasAutos;

/* This code runs on client session
 * Create a copy of WORK.SASMACR to avoid I/O conflict
 */
proc catalog cat=work.sasmacr force;
    copy out=work.sasmacr_session&serverId.;
run; quit;

/* This code runs on server sessions */
proc catalog cat=remote.sasmacr_session&serverId. force;
    copy out=work.sasmacr;
run; quit;
```

%AsyncLoop – implementation of the model

Listing 12 is the macro that implements the model that we have discussed. The parameters exposed to users are number of server sessions to use (nParallel), number of iterations (nIterations), libnames to pass to server sessions (INHERITLIB, WORK is always shared with alias name REMOTE), and whether to use SAS Grid servers (instead of MP Connect).

If we go back to Listing 3, what %AsyncLoop does is that it takes over the %DO-Loop. User specifies the actual code to run in %__iteration__ macro and the number of iterations to run. Within this implementation, a macro variable __async_mode__ is used to signal other macros that parallel processing is disabled. Macros like %StartServers and %GetFreeServer respect this macro variable and behave accordingly. The job dispatching %DO-Loop fallbacks to regular linear execution when __async_mode__ equals to 1.

Listing 11 %AsyncLoop

```
%macro AsyncLoop
(nParallel   = 3
,nIterations =
,inheritLib  =
,enableGrid  = N
);

%local __thisMacroName__;
%local __iternum__;
%local __freeServerId__;
%local __MAX_CONCURRENT__;
%local __async_mode__;

%let __thisMacroName__ = &sysmacroname;
%let __MAX_CONCURRENT__ = 10;
%let __async_mode__ = 1;

%__WriteToLog__(Start of &__thisMacroName__., decoration=%str(=));


*****************************************************************;
* Check parameters                                              *;
*****************************************************************;
%if &nParallel > &__MAX_CONCURRENT__. %then %do;
  %put AL%str(ERT_R(&__thisMacroName__)): Maximum number of server sessions allowed is
&__MAX_CONCURRENT__..;
  %let nParallel = &__MAX_CONCURRENT__.;
  %put AL%str(ERT_I(&__thisMacroName__)): Reset nParallel to &nParallel;
%end;

%if &nIterations. < &nParallel. %then %do;
  %put AL%str(ERT_R(&__thisMacroName__)): Number of iterations(&nIterations) > number of server
sessions(&nParallel). ;
  %let nParallel = &nIterations.;
  %put AL%str(ERT_I(&__thisMacroName__)): Reset nParallel to &nParallel;
%end;

%if &nParallel. <= 1 %then %do;
  %put AL%str(ERT_I(&__thisMacroName__)): Number of server sessions <= 1.;
  %put AL%str(ERT_I(&__thisMacroName__)): Fall back to synchronous loop.;
  %let __async_mode__ = 0;
%end;

%if %length(&enableGrid) = 0 %then %let enableGrid = N;
%else %let enableGrid = %qsubstr(&enableGrid, 1, 1);
%if &enableGrid. ^= Y %then %let enableGrid = N;

/* SAS Code to check validity of parameter inheritLib */


*****************************************************************;
* Start server sessions and initialize them                     *;
* When nServerSessions <= 2, fall back to synchronously mode    *;
*****************************************************************;
%StartServers
(serverSessionIdPrefix  = ss
,rsubmitStatusVarPrefix = rstat
,nServerSessions        = &nParallel.
,inheritLib             = &inheritLib.
,enableGrid             = &enableGrid.
);


*****************************************************************;
* Async Do-Loop                                                 *;
*****************************************************************;
%do __iternum__ = 1 %to &nIterations.;
  %let __freeServerId__ = %GetFreeServer(nServerSessions=&nParallel, rsubmitStatusVarPrefix=rstat);
  /* This macro calls %__iteration__ and pass value of &__iternum__.
   * Some utility macros for debugging and tracking purpose are embedded.
   */
```

```
  %RunSingleIteration
  (serverSessionId  = ss&__freeServerId__.
  ,rsubmitStatusVar = rstat&__freeServerId__.
  );
%end;


*************************************************************;
* Finalization                                              *;
* Wait for all server sessions to finish executions, merge SAS   *;
* logs and outputs to client session and then sign off server    *;
* sessions                                                  *;
*************************************************************;
%SignOffServers
(serverSessionIdPrefix = ss
,nServerSessions       = &nParallel
);

%__WriteToLog__(End of &__thisMacroName__., decoration=%str(=));
%__Print_Timer__;
%mend AsyncLoop;
```

Factors that affect performance

In our dataset comparison tool example, if we had 40 datasets (iterations) and 5 server sessions, the expected execution time is 1/5 of linear execution. However, this is hardly true. Despite the overheads due to starting servers and collecting logs and outputs (which are usually negligible), execution time of parallel executions is affected by a few factors.

The most obvious one is the order of iterations. Say the longest iteration takes one hour and the rest of them all takes 10 minutes. If we run the longest iterations first, the first iteration on server session 1 takes one hour, during the time the rest 4 server sessions can execute 4*6 = 24 iterations. The rest 15 iterations can be completed in 3 rounds, namely 30 minutes. Hence the final execution time is 60+30=90 minutes (about 1 and a half hours). If we run the longest iteration in the last, the first 39 iterations take 8 rounds. The 8th round uses 4 servers, and the remaining one server is used to execute the longest iteration. The final execution is 70+60=130 minutes (about 2 hours). Compared to 90 minutes (about 1 and a half hours), it is a considerable increase. In general, when the number of server sessions are less than the number of iterations (which is usually what we have), the iterations that take longer time are better to execute before the ones that take less time. This ensures no server sessions are sitting in idle when some others are running intensively. In the real world, we cannot know in advance the execution time of each iteration. However, for particular use cases, like our dataset comparison tool, we can roughly use size of datasets to project the execution time. Note that since %__iteration__ is a single iteration, in general there is no way to arrange the order of iterations in this macro. The better approach is to sort task dataset (Listing 9) based on projected execution time in descending order.

Discussions

Further improvements can be made to %AsycLoop:

- Add a timeout parameter. The client session should track executions on server sessions. If the execution time exceeds a threshold value, client session should kill the asynchronous task on a server session.
- Add support to dynamically adjust the order of iterations of subsequent runs. Iteration numbers are adjusted based on execution time in the previous run.

Contact Information

Yong Cao

Team Leader, Programming, Clinical Research

PPD, part of Thermo Fisher Scientific

Yong.Cao@ppd.com

Your comments and questions are welcomed.

Source Code

Readers can get a copy of full code of %AsyncLoop by contacting the author in email.

Reference

Leonid Batkhan (2021). Running SAS programs in parallel using SAS/CONNECT® . SAS Users Blogs.