

# Multiprocessing with Version 8 of the SAS System

Cheryl Doninger, SAS Institute Inc.

## INTRODUCTION

With the ever increasing need to get more done in the same amount of time, naturally we want our computer applications to take advantage of the multi-processors available in many of today's desktop and server platforms as well as to be able to multi-process across platforms available in a network. By dividing time-consuming tasks into multiple independent units of work and executing these independent units of work in parallel, a job can be performed in substantially less time than if each task is performed sequentially. A new feature has been added to Version 8 of the SAS System that allows your SAS jobs to take advantage of SMP hardware and to also allow parallel processing with the remote hosts in your network. This feature is part of the SAS/CONNECT product and is called MP CONNECT. MP CONNECT allows you to perform disjoint units of work in parallel and coordinate all of the results into your original SAS session for the purpose of reducing the total elapsed time necessary to execute a particular application.

This paper will introduce the concept of multiprocessing and illustrate its benefits. Test results will be presented to show tangible proof of the time savings that are possible by modifying your existing jobs as well as designing new jobs to use MP CONNECT for multiprocessing. The syntax and options that enable MP CONNECT will be covered. And finally, the test case that was used to collect the results presented in this paper will be included in an appendix.

## WHY MULTIPROCESSING

In this paper the term *multiprocessing* refers to dividing an application into independent sub-units of work that can be executed simultaneously. The primary purpose of multi- or parallel processing is to complete a job in less total elapsed time than it would take to execute the same job serially. With this capability IT staffs are challenged to have their applications take advantage of the multiple processors available in today's server platforms. In addition to exploiting standalone machines, IT staffs also benefit by multiprocessing across their networks. However, generally, the SAS System is a single-threaded application that executes on a single processor or single machine at a time.

*Independent parallelism* is possible when the execution of Task A and Task B do not have any interdependencies. An example of this in SAS procedure terms would be if an application needed to run PROC SORT against two different SAS data sets and then merge the sorted data sets into one final data set. Because there is no dependency between the two data sets that initially need to be sorted, the two PROC SORTs can be performed in parallel and when they both finish, the merge can take place. It is this type of parallelism that is addressed by MP CONNECT. MP CONNECT

provides a convenient interface for spawning  $n$  SAS sessions to simultaneously execute  $n$  tasks as independent processes and coordinate the execution and results of all  $n$  tasks into the original SAS session. The  $n$  SAS sessions or processes can either all execute on the same machine with each session or process running on a separate processor or they can be directed to any number of remote machines. The remote machines can have either single or multiprocessor capabilities.

The following scenarios are meant to present real life applications that are ideally suited for independent parallelism and therefore MP CONNECT. Hopefully, these scenarios will help you to think of similar or additional SAS tasks which you could modify or develop to benefit from MP CONNECT.

Figure 1 illustrates a data warehouse scenario which requires extracting data from three unique and possibly remote data sources, combining that data, and then processing it. By performing the three extractions in parallel and then merging the three resulting data sets, you effectively decrease the time it takes to complete the entire task to the time of the longest extraction plus the time to perform the merge.

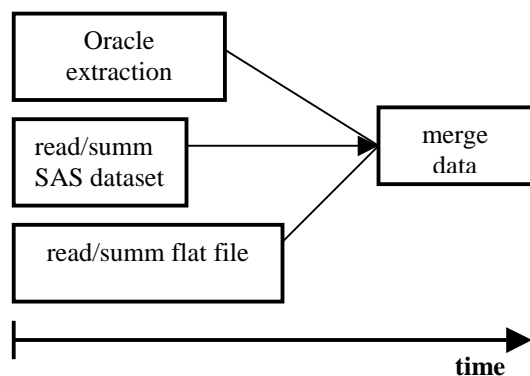


Figure 1. Data Warehouse Scenario

Figure 2 illustrates a SAS analysis example which requires running several independent SAS procedures against a single or possibly multiple data sources. Running these procedures in parallel can drastically reduce the total amount of time required to execute the whole job. Essentially, the time needed to execute the job now becomes the execution time of the longest running procedure.

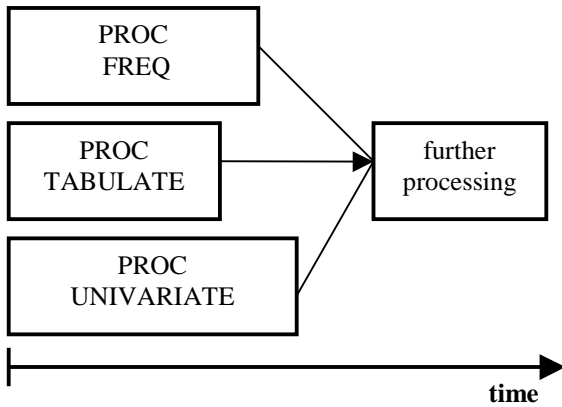


Figure 2. Multiple Analyses of Single SAS Data Set

Figure 3 is a HOLAP scenario which requires the creation of multiple MDDBs. Since the creation of an MDDB is an independent task and one which can require a significant amount of processing time, running the creation of multiple MDDBs in parallel is an excellent application for MP CONNECT. Depending on the number and size of the MDDBs that you create, the benefit of running these tasks in parallel with MP CONNECT could be very substantial.

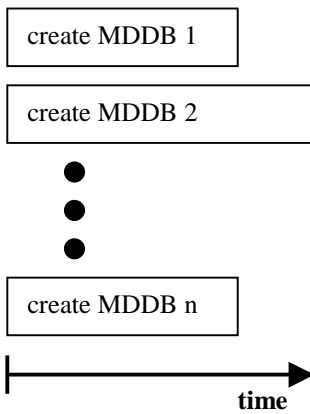


Figure 3. Parallel creation of Multiple MDDBs

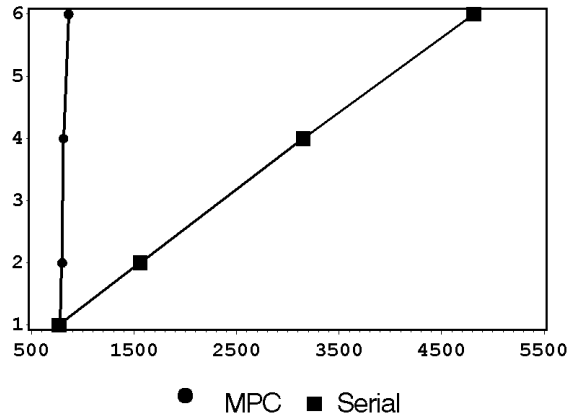
**TEST RESULTS SHOW THE BENEFIT**

The test that was used to create the results presented in this section consists of a variety of Base SAS procedures run against data sets that vary in size. This test was first run using MP CONNECT to spawn  $n$  SAS processes and each process executed one instance of the test simultaneously. Then the test was run by executing  $n$  serial iterations of the same test. It was implemented using the macro facility so that the size of the data set and the number of processes created by the test (or in the case of serial execution, the number of times the test was re-iterated) could be easily varied. These tests were all run on a multi-processor

machine, rather than remote machines on a network, in order to have a more consistent environment for the purpose of collecting test results. Version 8.0 of the SAS System was used to collect these results.

In the following graphs the x-axis represents time. The y-axis represents the number of parallel SAS processes or the number of iterations of the serial test.

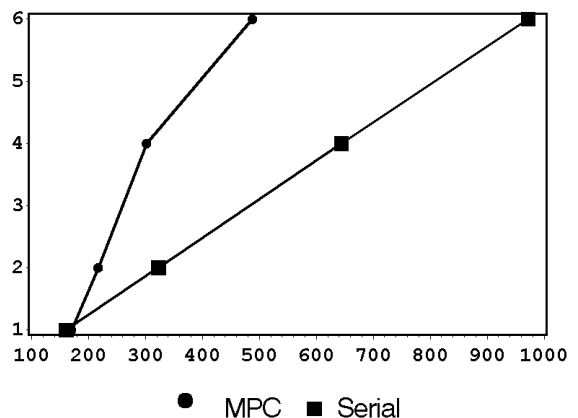
The first set of tests was run on a Sun Enterprise 10000 with twelve 400 MHz Ultrasparc processors and the results are summarized in Graph 1.



Graph 1. Sun Enterprise 10000 Test Results

These test results show a remarkable time savings using MP CONNECT instead of serial execution. These results are also extremely positive with respect to scalability. For the serial case, increasing the number of iterations of the test produced a linear increase in the time necessary to complete the job. With MP CONNECT, however, there was a negligible increase in elapsed time as additional SAS processes were added.

The second set of similar tests was run on a Compaq ProLiant 8000 8 way server with 550 Mhz, 1 MB cache cpus running Windows NT. The results are summarized in Graph 2.



Graph 2. Compaq ProLiant Test Results

These results also show a remarkable time savings using MP CONNECT instead of serial execution. And, these results are very positive with respect to scalability. For the serial case, adding additional iterations to the test resulted in a linear increase in the time necessary to complete the job. With MP CONNECT, however, there was only a small increase in elapsed time as additional SAS processes were added.

## GUIDELINES

It is important to note that not all applications are good candidates for the independent parallelism that MP CONNECT provides. Here are a few guidelines to help you determine if a particular application, or portions of it, can benefit by using MP CONNECT:

1. Determine if your data source(s) can be processed separately and independently. It may be worth while to duplicate some data in order to achieve this independence.
2. Look for ways to segment your job, or some portion of it, into sub-tasks that do not access joint non-sharable resources.
3. Ensure that your I/O subsystem can handle the additional data requirements introduced by parallel execution. You may need to spread your I/O across physical disks and/or I/O channels.

However, for those jobs that can be separated into independent units of work, the time it takes to complete the entire job can be drastically reduced by using MP CONNECT.

## MP CONNECT – THE DETAILS

Prior to Version 8, SAS/CONNECT was a synchronous client/server tool with the emphasis on the ability to connect a SAS session running on a local machine to a SAS session running on a remote machine. With Version 8, MP CONNECT allows you to perform multi-processing with the SAS System by establishing a connection between multiple SAS sessions and enabling each of the sessions to asynchronously execute tasks in parallel. You also have the ability to merge the async tasks back into your local execution stream at the appropriate time. In addition, establishing connections to processes on the same local machine has been greatly simplified. This gives you the ability to exploit MP/SMP hardware as well as network resources to perform parallel processing of self-contained tasks and easily coordinate all the results into the original SAS session.

You can use MP CONNECT to spawn  $n$  SAS processes where  $n$  is the number of independent units of work that you wish to perform in parallel. SAS processes that are spawned on a single multi-processor machine are independent unique processes just as they are if they are initiated on a remote host. On Windows and Unix, for example, each SAS session is a separate process having its own unique SASWORK library. Each process also assumes the user context of the parent, or the user that invoked the original SAS session, and possesses all of the rights and privileges associated with that parent. On MVS each SAS

session is an MVS BPX address space that inherits the same STEPLIB and USERID as the client address space. The client's SASHELP, SASMSG, SASAUTOS, and CONFIG allocations are passed to the new session as SAS option values.

Normally, with SAS/CONNECT, the SIGNON command or statement is used to establish a connection between two SAS sessions. The SIGNON interface has been simplified to facilitate establishing connections to SAS sessions on the same machine. This is possible because less information is required for communication between processes on a single machine versus inter-process communication across a network. This new SIGNON interface eliminates the need for a script file on the local host, the need to re-specify userid/password information, the need to have a spawner running, and the need to perform any access method file configuration such as transaction programs, etc. The only parameters that you are required to specify with the SIGNON statement is the command to be used to invoke the SAS session and a name to associate with this new SAS process. A new option, SASCMD, is used to specify the SAS command that is used to invoke the "remote" SAS session. The SASCMD option can be specified in a global options statement as well. In addition, the NOTIFY option can be used on the SIGNON command or statement to request the display of a notification message window to report the completion of any subsequent asynchronous RSUBMITs.

Once the SIGNON has been executed and a connection established to one or more SAS sessions, either on the same machine or a remote machine, then the RSUBMIT command or statement can be used to asynchronously execute multiple independent tasks and reduce the overall execution time of your SAS job. In fact, the autosignon feature of RSUBMIT can be exploited to reduce the required syntax for spawning a new SAS session, sending it a unit of work, and terminating this session on completion to just a single RSUBMIT/ENDRSUBMIT block. This is possible because the RSUBMIT statement now accepts all the same parameters that the SIGNON statement does. The exact syntax and an example of this will be given in the next section.

The RSUBMIT statement is used along with the WAIT=NO option to identify that a unit of work should be asynchronously executed by the newly spawned SAS session. By executing the RSUBMIT asynchronously, you can start a long running task in one new SAS session and immediately be able to begin another task in another SAS session rather than wait until the first remote task is complete before regaining control of your original SAS session. And if NOTIFY=YES was specified during the signon process, a notification message window will be displayed in the local session when the asynchronous task completes execution.

For each asynchronous process, the default action is for SAS/CONNECT to spool the accumulated log and output lines from the remote process until you request the data by using the RGET command, executing a synchronous RSUBMIT, or issuing a SIGNOFF to terminate the remote SAS process. The RDISPLAY command can be used to view the current contents of the spooled log and output windows without merging the contents into the local log and output windows. Once the RGET command is issued, the accumulated log and output lines are retrieved and merged with the local log and output lines. Alternatively, you can use

the LOG= and OUTPUT= options on the RSUBMIT to either direct the log and output lines to a file or to purge them.

Another very important piece to this asynchronous multi-processing is the ability to synchronize any or all of your asynchronously executing tasks with subsequent local execution. This capability is provided with the WAITFOR statement which lets you suspend your local SAS processing pending the completion of any or all of your asynchronous tasks. For example, if you initiated two SAS sessions to simultaneously sort two data sources and then need to merge the sorted output, you could issue the WAITFOR statement in your original SAS session subsequent to the two async sorts and prior to the final data step used to merge the data.

The LISTTASK statement can be used to view the state of any or all active connections. It is most useful for viewing the state of asynchronously executing tasks. And finally, the KILLTASK statement lets you maintain complete control by enabling you to terminate any or all asynchronous tasks with a single statement.

The following sections detail the statements and options that enable multi-processing with the SAS System.

### SIGNON Command/Statement

The SIGNON command and the SIGNON statement are used to invoke another SAS session and establish a connection between the two SAS sessions. The newly created SAS session can be created either on the same machine or on a machine that is remote with respect to the parent or client SAS session. If the SAS session is being invoked on a remote machine, either a spawner and/or a script file can be used to provide the information necessary to invoke this new SAS session. If the SAS session is being spawned on the same machine as the originating or parent SAS session, then the new SASCMD option can be specified to provide the command necessary to invoke SAS. This option can be specified either with a global options statement, with SIGNON or, if the autosignon feature is being used, with RSUBMIT.

```
SASCMD=" SAScmd " ;
```

where SAScmd specifies the command to be used to spawn a new SAS process. Additional SAS options can be specified within the quotes. If you need to execute additional host commands prior to the SAS invocation, it is recommended that you write a host specific script that contains your host commands and the SAS invocation, and specify this script as the SASCMD value.

On OS/390 hosts the SASCMD option is specified as follows:

```
SASCMD=" :SAS-system-options " ;
```

where specifying any non-blank value will cause the UNIX fork command to spawn an MVS BPX address space which inherits the same STEPLIB and USERID as the client address space. The client's SASHELP, SASMSG, SASAUTOS, and CONFIG allocations are passed to the spawned SAS session as SAS option values. Any additional SAS invocation options can be specified following the colon. The XMS access method is used by the two sessions for communication.

The NOTIFY option can be used to request the display of a notification message window upon completion of an asynchronous RSUBMIT.

```
NOTIFY=YES | NO
```

where a value of YES causes a notification window to be displayed with the following message:

```
Asynchronous task JOB has completed.
```

where JOB is the remote session identifier. To acknowledge the message and to close the window, click the OK button.

### RSUBMIT Command/Statement

The RSUBMIT command and the RSUBMIT statement cause SAS programming statements that are entered in the local environment to execute in a remote SAS session. Even though the statements execute in the remote environment, all responses and output are displayed in your local SAS log and output windows as they would be if you executed the program in the local SAS session.

RSUBMITs are processed in either *synchronous* or *asynchronous* mode.

*Synchronous mode* means that the remote processing must complete before the local session can continue processing. Therefore, the RSUBMIT must run to completion before you regain control. Synchronous processing is the default processing mode.

*Asynchronous mode* allows you to start an RSUBMIT in the background to a remote host and to regain local control immediately to continue with local processing or remote processing to another host.

The following RSUBMIT options enable asynchronous RSUBMITs:

```
CONNECTWAIT | CWAIT | WAIT=value
```

where value specifies whether this particular RSUBMIT is to be executed synchronously or asynchronously. This can also be specified with the CWAIT global option by submitting the following options statement:

```
OPTIONS CWAIT=NO;
```

The valid values for the WAIT= option are:

YES   Y	indicates a synchronous RSUBMIT.
NO   N	indicates an asynchronous RSUBMIT.

If WAIT=NO is specified, it may also be useful to specify the MACVAR= option. This will allow you to programmatically test the status of the current asynchronous RSUBMIT by determining whether it has completed or is still in progress.

```
CMACVAR | MACVAR=value
```

where value specifies the name of the macro variable to associate with this remote session. If specified on the

RSUBMIT command/statement, the MACVAR= option overrides any previous MACVAR= specifications for this remote session. The macro variable is NOT set if the RSUBMIT command fails due to incorrect syntax. Other than this one exception, the macro variable (value) is set to one of the following values:

0	Indicates that the RSUBMIT is complete.
1	Indicates that the RSUBMIT failed to execute.
2	Indicates that the RSUBMIT is still in progress.

Note: If a synchronous RSUBMIT (WAIT=YES) is issued while an asynchronous RSUBMIT (WAIT=NO) is still in progress, all spooled log and output statements are merged into the local log and output windows and the RSUBMIT continues as if it were synchronous. That is, you do not regain local control until the RSUBMIT has completed. If you don't want this to happen, use the MACVAR= option in the SIGNON or the RSUBMIT statements so that you can check the progress of RSUBMIT without causing it to execute synchronously.

```
LOG=KEEP | PURGE | filename | fileref
OUTPUT=KEEP | PURGE | filename | fileref
```

directs the SAS log or the SAS output that is generated by an asynchronous remote submit to the backing store, to be purged, or to a specified file.

KEEP spools log or output lines to the backing store for later retrieval with the RGET, RDISPLAY, or SIGNOFF statements. This is the default.

PURGE deletes all of the log or output lines that are generated by the current remote session. PURGE is used for economizing disk resources. If you can anticipate a large volume of log data or output data to the backing store that you do not want to receive, and you want to preserve disk space, setting PURGE can be useful.

**filename | fileref** specifies a file that is the destination for the log or output lines. The file is opened for output at the beginning of the asynchronous RSUBMIT and is closed at the conclusion of the asynchronous RSUBMIT. A subsequent RSUBMIT to the same file that you specify by using the LOG= or the OUTPUT= option overwrites the previous contents of that file with the contents of the current RSUBMIT.

### WAITFOR Statement

The WAITFOR statement is used to make the local SAS session wait for the completion of one or more asynchronously executing tasks. If more than one task is specified, then the WAITFOR statement can include either the \_\_ANY\_\_ or the \_\_ALL\_\_ option. The \_\_ANY\_\_ option suspends the SAS session until the completion of any of the specified tasks (a logical OR). The \_\_ALL\_\_ option suspends the SAS session until the completion of all of the specified tasks (a logical AND). You can also specify a timeout value with the TIMEOUT option. If the specified tasks have not finished processing by the timeout value specified, the local session regains control and the tasks continue to execute asynchronously. In addition, the

&SYSRC macro variable is set to indicate that a timeout occurred. If the specified tasks finish processing before the timeout value specified, the WAITFOR statement returns control to the local SAS session.

### RGET Command/Statement

The RGET command and the RGET statement cause the spooled log and output from the execution of an asynchronous remote submit to be merged into the local log and output windows. When an asynchronous remote submit executes, the log and output statements are not merged into the local log and output windows. By default, they are spooled for retrieval at a later time. However, if the LOG or OUTPUT options have been used on the RSUBMIT to redirect the lines to an external file or to purge them, they will not be available for the RGET statement to merge into the local log and output windows.

If the RGET command or RGET statement is executed while the asynchronous remote submit is still in progress, all currently spooled log and output lines are retrieved and merged into the local log and output windows, and the remote submit continues processing as if it had been submitted synchronously. That is, you will NOT regain control in your local SAS session until the remote submit has completed. If you don't want the remote submit to become synchronous, but you want to check its progress, use the MACVAR option in the SIGNON or the RSUBMIT statement. This allows you to check the progress of an asynchronous remote submit without causing it to execute synchronously. Or, if you are running interactively, you could use the LISTTASK statement to check the status of the asynchronous remote submit before issuing the RGET statement.

### RDISPLAY Command /Statement

The RDISPLAY command and the RDISPLAY statement create two windows for each asynchronous process that is executing to display the spooled log and output lines that have been generated. One window displays the log lines and the other window displays the output lines.

When an asynchronous remote submit executes, the log and output lines are not merged into the local log and output windows. By default, they are spooled to disk for retrieval at a later time. RDISPLAY allows you to view the spooled log and output lines created by the asynchronous remote submit without merging them into the local log and output windows. The log and output lines continue to scroll into the windows created by the RDISPLAY command as they are produced by the remote processing. The RGET command or statement, a synchronous RSUBMIT, or the SIGNOFF command or statement must be executed to actually merge the spooled lines into the local log and output windows. However, if the LOG or OUTPUT options have been used on the RSUBMIT to redirect the lines to an external file or to purge them, they will not be available for the RDISPLAY statement to display.

### LISTTASK Statement

The LISTTASK statement lists information about any or all active connections. The LISTTASK statement displays

information such as the task name and its current state of execution. A task can be in one of the following states of execution:

- READY
- RUNNING SYNCHRONOUSLY
- RUNNING ASYNCHRONOUSLY
- COMPLETE

When you SIGNOFF from a remote session or task it is removed from the list of tasks that LISTTASK maintains.

### KILLTASK Statement

The KILLTASK statement is used to immediately terminate one or more asynchronously executing tasks. You can either specify a list of task names separated by spaces or the keyword `__ALL__` if you want to terminate all of the asynchronously executing tasks. This statement is valid for tasks that are executing asynchronously on the same host and for tasks and SAS sessions that are executing asynchronously on remote hosts. KILLTASK causes any log or output lines that have accumulated in the backing store to be flushed to the parent or local Log and Output windows.

### CONNECT Monitor Window

The SAS Explorer now provides a menu selection that enables you to monitor SAS/CONNECT tasks. Use the following path to access the SAS/CONNECT Monitor window from the SAS Explorer:

View -> SAS/Connect Monitor

The SAS/CONNECT Monitor window displays information about the tasks in two columns: Name and Status. An example follows:

Name	Status
Task1	Complete
Task2	Running Asynchronously
Task3	Running Synchronously

The list of tasks is dynamically updated as new tasks start, and the Status field changes from "Running" to "Complete," as appropriate. When you use the SIGNOFF statement to terminate a connection, the task is automatically removed from the window.

You may also terminate a task that is running asynchronously by clicking the task in the monitor window and selecting the Kill option from the menu that displays when you right-click the mouse button. Similarly, you can select the Rdisplay option from the menu to display LOG and OUTPUT windows for a task that is running asynchronously.

### Example 1

The following example is a simplistic illustration of 2-way multiprocessing; the original SAS session executes a data step in parallel with an additional SAS session which executes a PROC SORT.

First, an asynchronous rsubmit is executed using the autosignon feature in order to spawn an additional SAS session and instruct it to execute the PROC SORT.

```

OPTIONS AUTOSIGNON=YES;
RSUBMIT SORTASK WAIT=NO SASCMD='SAS';
LIBNAME ANNUAL 'path to annual sales lib';
PROC SORT DATA=ANNUAL.SALES
          OUT=ANNUAL.SORT1;
          BY REGION;
RUN;
ENDRSUBMIT;

```

The local SAS session can then be used to simultaneously perform additional processing because the above PROC SORT is being processed asynchronously by a separate SAS session. In the following section, a data set is created and then the WAITFOR statement is used to synchronize the completion of the above PROC SORT with a subsequent data step to merge the two data sets.

```

LIBNAME LOC 'path to monthly sales lib';
DATA LOC.MARCH;
DO I = 1 TO NREGIONS;
/* create local data set */
END;
RUN;

```

```
WAITFOR SORTASK;
```

```
LIBNAME ANNUAL 'path to annual sales lib';
```

```

DATA TOTAL;
SET ANNUAL.SORT1 LOC.MARCH;
RUN;

```

### Example 2

The following example assumes that you have a multi-processor machine on which you would like to run MP CONNECT tasks in parallel. You do not have direct access to this machine but are able to signon to it from your local workstation. The following code is a template that could be used to signon to a multi-processor server and then remote submit several tasks to run in parallel on that server.

```

FILENAME RLINK <script file for server>;
%LET HOST1=<my.smp.server.box>;
/* signon from workstation to server */
SIGNON HOST1;

```

```

/* rsub multiple tasks to server */
RSUBMIT;

```

```

OPTIONS AUTOSIGNON=YES SASCMD="SAS";
RSUBMIT TASK1 WAIT=NO;
/* stmts processed by TASK1 */
ENDRSUBMIT;

```

```

RSUBMIT TASK2 WAIT=NO;
/* stmts processed by TASK2 */
ENDRSUBMIT;

```

• • •

```

RSUBMIT TASKn WAIT=NO;
/* stmts processed by TASKn */
ENDRSUBMIT;

```

```
WAITFOR __ALL__ TASK1 TASK2 TASKn;
```

```
ENDRSUBMIT;
```

## FUTURE RESEARCH & DEVELOPMENT

We are currently working on several enhancements to facilitate the use of MP CONNECT including but not limited to:

- specifying a remote session id on %syslput statements to direct execution to a specific remote session,
- detecting active asynchronous tasks upon normal SAS termination and prompting for the appropriate action to take, and
- allowing spawned or remote SAS sessions to have easy access to libraries defined to the local or parent SAS session.

In addition, we have prototyped an implementation of *pipeline parallelism*. Pipeline parallelism is possible when Task B requires output from task A, but it does not require all of the output before it can begin execution. In SAS terms, this means that two SAS procedures could run in parallel such that one proc feeds its output to the next proc as input. For example, a SAS data step could feed observations, as they are created, to PROC SORT allowing the SORT procedure to run in parallel with the data step minimizing the total time required to complete both steps. In other words, a procedure does not write its output to disk, but rather pipes it as input to the next procedure. We are currently researching the benefits and considerations for this type of parallelism.

## CONCLUSION

This paper presents the new Version 8 MP CONNECT feature that enables you to perform parallel- or multi-processing with the SAS System. This feature provides you with a straight forward syntax to allow you to make minimal modifications/additions to your existing or new SAS jobs in order to substantially decrease the total elapsed time necessary to execute a job.

It is strongly recommended that each SAS application be evaluated for potential benefits before implementing the MP CONNECT feature. For those jobs that perform time consuming tasks that can be separated into independent units of work, MP CONNECT can be used to decrease the time of execution to a fraction of what is required to execute the same job serially. MP CONNECT is also extremely scalable which means that you will continue to recognize tremendous time savings as the number of SAS processes running in parallel approaches the number of processors on your system or in your network.

SAS and SAS/CONNECT are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.

### Author

Cheryl Doninger  
SAS Institute Inc.  
SAS Campus Drive  
Cary, NC 27513  
(919) 677-8000  
Cheryl.Doninger@sas.com

## APPENDIX – THE TEST CASE

This appendix contains the SAS job that was run to collect the test results that are presented in this paper. This test was originally used for another purpose and was run serially. The test was modified to use MP CONNECT for the purpose of collecting results for this paper. The statements that were required by MP CONNECT have been highlighted with bold font in order to illustrate just how minimal the changes were to this particular test. Notice that there are only six such statements. Similar minor additions could be made to your existing SAS jobs to take advantage of your MP/SMP hardware as well as remote hosts on your network to dramatically reduce the total execution time of your jobs.

```
options fullstimer ;
options autosignon=yes;
options sascmd='sasv8';

data _null_ ;
host=sysget('HOST') ;

call
symput('numloop',compress(trim(scan("&sysparm",1,
".")))));
call symput('datsz',trim(scan("&sysparm",2,".")));
call symput('host',trim(host)) ;

run ;

%macro benchds(numrecs);
libname foo v8 '/tmp';
data foo.tstdata;
/* create a data set with 110 variables and obs equal
value of numrecs passed into the macro */
%mend ;

%benchds(&datsz) ;

%global time1;
%macro pretime;
data _null_;
time=put(time(),6.);
call symput('time1',time);
run ;
%mend;

%macro posttime(sec);
data _null_;
time=time()- %str(&time1);
put '*';
put "*****time elapsed(&sec) = " time 6.;
put '*';
run ;
%mend;

%macro shutdown() ;
```

```

%local runid;
%let runid=1 ;
%let remsessions=;
%do % while(&runid le &numloop);
  %let remsessions=&remsessions rem&runid;
  %let runid=%eval(&runid+1) ;
%end;

waitfor _all_ &remsessions;

%postime(0);

%let runid=1 ;
%do % while(&runid le &numloop);
  proc printto log="mploop&runid..log"
    print="mploop&runid..lst" new ; run;
  rget rem&runid;
  signoff rem&runid;
  %let runid=%eval(&runid+1) ;
  proc printto; run;
%end;

%mend ;

%macro startup(subsys) ;
%pretime;

proc datasets library=work ;
  delete stats1 stats2;
quit;

  %put APR HEADER os=&sysscp;
  %put APR HEADER host=&host;
  %put APR HEADER ver=&sysvlong;
  %put APR HEADER subsys=&subsys;
  %put APR HEADER numobs=&datsz;
  %put APR HEADER duration=&numloop;

%mend ;

%macro runtest(runid) ;
/*****
*   create a new MP CONNECT session to
*   handle this iteration
*****/
rsubmit rem&runid wait=no;
libname foo v8 '/tmp';

/*****
*   Step CONTENTS_2
*****/

proc contents data=foo.tstdata;

```

```

/*****
*   STEP SORT_3
*****/

proc sort data=foo.tstdata out=out1 tagsort ;
  by descending x1_10 sname8 ;
run ;

/*****
*   STEP FREQ_4
*****/

proc freq data=foo.tstdata;
  tables sname20 onein10 onein100 onein1k ;
  title 'proc freq ' ;
run ;

/*****
*   STEP SUMMARY_5
*****/

proc summary data=foo.tstdata print ;
  title 'proc summary full data set ' ;
  var _numeric_ ;
run ;

/*****
*   STEP SUMMARY_6
*****/

proc summary data=foo.tstdata print ;
  title 'proc summary three state names subsetted by
where clause ' ;
  var _numeric_ ;
  where sname20='ALABAMA' or
sname20='CALIFORNIA' or sname20='TEXAS' ;
run ;

/*****
*   STEP DATA_7
*****/

data temp;
  set foo.tstdata;
  if sname20='ALABAMA' or
sname20='CALIFORNIA' or sname20='TEXAS';
run ;

/*****
*   STEP SUMMARY_8
*****/

proc summary data=temp print ;
  title 'proc summary three state names subsetted by
data set ' ;
  var _numeric_ ;

```

```

run ;

/*****
* STEP DATA_9
*****/

data _null_ ;
set foo.tstdata;
where onein100='y';
run ;

/*****
* STEP DATA_10
*****/

data _null_ ;
set foo.tstdata;
if onein100='y' ;
run ;

/*****
* STEP DATA_11
*****/

data _null_ ;
set foo.tstdata;
where x1_1000=9 or x1_1000=99 or x1_1000=999
;
run ;

/*****
* STEP DATA_12
*****/

data _null_ ;
set foo.tstdata;
run ;

/*****
* STEP TRANSPOSE_13
*****/

proc transpose data=foo.tstdata (keep=f1 x1_10)
out=trans ;
by x1_10 notsorted ;
var f1 ;
run ;

/*****
* STEP SORT_15
*****/

proc sort data=foo.tstdata out=out1 ;
by x1_100 sname8 ;
run ;

*****
* STEP SUMMARY_16
*****/

proc summary data=foo.tstdata;
title 'proc summary full data set ' ;
var _numeric_ ;
output out=stats1 ;
run ;

/*****
* STEP SUMMARY_17
*****/

proc summary data=foo.tstdata;
title 'proc summary three state names' ;
var _numeric_ ;
where sname20='ALABAMA' or
sname20='CALIFORNIA' or sname20='TEXAS' ;
output out=stats2 ;
run ;

/*****
* STEP SUMMARY_18
*****/

proc summary data=temp ;
title 'proc summary three state names' ;
var _numeric_ ;
output out=stats2 ;
run ;
endsubmit;
%mend ;

%macro duration();
%local runid ;
%let runid=1 ;
%do %while(&runid le &numloop);
%rntest(&runid) ;
%let runid=%eval(&runid+1) ;
%end;
%mend;

%startup(Perf);
%duration ;
%shutdown;

```