Paper 16-25

Multiprocessing with Version 8 of the SAS® System

Cheryl Garner, SAS Institute, Inc.

INTRODUCTION

With the ever increasing need to get more done in the same amount of time, naturally we would want our computer applications to take advantage of the multi-processors available in many of today's desktop and server platforms. 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 MP and SMP hardware. 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 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. Benchmarks 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 benchmarks presented in this paper will be included in an appendix.

WHY MULTIPROCESSING

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. However, generally, the SAS System is a single-threaded application that executes on a single processor.

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 all execute on the same machine with each session or process running on a separate processor.

However, not all applications can benefit from parallel processing. Here are several questions to help you determine if a particular application can benefit by using MP CONNECT:

1. Can your data source(s) be split such that they can be processed separately and independently?
2. Can you segment your job into parallel running SAS sessions that do not access joint non-sharable resources?
3. Do you get a good return on investment – do you save more time by running a job in parallel than it takes to convert/design it using MP CONNECT?

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.

BENCHMARKS SHOW THE BENEFIT

The test that was used to create the following benchmarks 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. 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 iterated) could be easily varied. Version 8.0 of the SAS System was used to collect these benchmarks.

In the following tables the first column represents the number of times that the test was repeated for the serial execution or the number of processes or SAS sessions created to simultaneously execute the test using MP CONNECT. *Total bytes* represents the total number of bytes in the data set. The *MP CONNECT* and *Serial Execution* columns show the total elapsed time for the test to execute in hh:mm:ss format.

The first set of tests was run on a Solaris 7 with twelve 400 MHZ Ultrasparc processors with the following results:

Loops / Pro-cesses	Total Bytes	MP CONNECT	Serial Execution
2	48MG	00:01:14	00:02:11
4	48MG	00:01:17	00:04:24
2	240MG	00:05:21	00:09:32

4	240MG	00:05:31	00:19:22
2	480MG	00:13:21	00:26:09
4	480MG	00:13:32	00:52:29
6	480MG	00:14:23	01:20:16
4	1 G	00:27:27	01:48:28
6	1 G	00:28:58	02:40:51

These benchmarks show a remarkable time savings using MP CONNECT instead of serial execution. These results are also very positive with respect to scalability. For the serial case, adding two more iterations to the test usually nearly doubled 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.

MP CONNECT – THE DETAILS

Prior to Version 8, SAS/CONNECT has always been a 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. MP CONNECT allows you to perform multi-processing with the SAS System by establishing a connection between multiple SAS sessions, all of which run on the same local machine. This gives you the ability to exploit MP/SMP hardware 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. On Windows and Unix, for example, each SAS session is a separate process having it's 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 connect between two SAS sessions. The MP CONNECT capability is available through a simplified SIGNON interface. This new SIGNON interface eliminates the need for a script file on the local host, the need to have a spawner running, and the need to perform any access method file configuration such as transaction programs, etc. The only thing 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. Once the SIGNON has been executed and the connection established, the RSUBMIT command or statement can be used asynchronously to 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. The exact syntax and an example of this will be given in the next section.

To establish an MP connection, both SIGNON and RSUBMIT accept a new option called SASCMD. This option specifies the SAS command that is used to invoke the "remote" SAS session. The RSUBMIT statement is used along with the WAIT=NO option to identify the unit of work that you wish to have 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. The SASCMD option can be specified in a global options statement as well.

For each asynchronous process, SAS/CONNECT spools the accumulated log and output lines from the remote process until you request the data by using the RGET command or issue a SIGNOFF to terminate the "remote" SAS process. Once the RGET command is issued, the accumulated log and output lines are retrieved and merged with the local log and output lines. 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.

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 let's 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. And finally, the LISTTASK statement can be used to provide information about the asynchronous tasks that are currently executing.

The following sections explain the statements and options that enable asynchronous remote processing.

RSUBMIT Command/Statement

The RSUBMIT command and the RSUBMIT statement cause SAS programming statements that are entered in the local environment to execute on 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 user does not regain local control until the RSUBMIT has completed. The RSUBMIT must run to completion before the user regains control. Synchronous processing is the default processing mode.

Asynchronous mode allows the user 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. Synchronous processing indicates that you will wait for the remote processing to complete before regaining control in the local SAS session. This is the default processing technique for RSUBMIT.

In asynchronous processing, when the RSUBMIT begins to execute in the background to the remote host, you regain control of your local SAS session to continue local processing or to use RSUBMIT to other remote sessions.

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 will also be useful to specify the MACVAR= option. This will allow you to 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 at the completion of the RSUBMIT command 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, the user does 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.

WAITFOR Statement

The WAITFOR statement is used to make the local SAS session wait for the completion of one 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. 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 all 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, but instead they are spooled for retrieval at a later time.

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 were 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.

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 are 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; instead, they are spooled to disk until they are retrieved with the RGET statement. 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 must be used to actually merge the spooled lines into the local log and output windows.

LISTTASK Statement

The LISTTASK statement lists information about any asynchronous tasks that are either active or completed and have not been specified in a WAITFOR statement. Once a task has been specified in a WAITFOR statement, it is removed from the list of tasks that LISTTASK maintains.

The LISTTASK statement displays information such as the task name and it's current status of execution.

Example
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 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';
PROC SORT DATA=FOO.SALES OUT=FOO.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.

```
DATA LOC.MARCH;
DO I = 1 TO NREGIONS;
/* create local data set */
END;
RUN;

WAITFOR SORTASK;
DATA TOTAL;
SET FOO.SALES LOC.MARCH;
RUN;
```

CONCLUSION

This paper presents the new Version 8 MP CONNECT feature that enables you to perform parallel or multiprocessing 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.

Author

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

APPENDIX – THE TEST CASE

This appendix contains the SAS job that was run to collect the benchmarks 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 benchmarks 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 and dramatically reduce the total execution time.

```
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 postime(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 stname8 ;
  run ;

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

  proc freq data=foo.tstdata;
   tables stname20 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 stname20='ALABAMA' or
stname20='CALIFORNIA' or stname20='TEXAS' ;
  run ;

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

  data temp;
   set foo.tstdata;
   if stname20='ALABAMA' or
stname20='CALIFORNIA' or stname20='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 stname8 ;
   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 stname20='ALABAMA' or
stname20='CALIFORNIA' or stname20='TEXAS' ;
    output out=stats2 ;
   run ;

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

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

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

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