

SESUG Paper DM-85-2017**Parallel Processing in a SAS® Grid**

Merry Rabb, Keith Brown, RTI International

ABSTRACT

In 2016 RTI International migrated over 400 users from SAS software® on PCs and several stand-alone Linux servers to a SAS Linux grid. In addition to other improvements the new grid provided an opportunity to improve the performance of some of our long-running jobs. SAS jobs often contain independent tasks that can be split up and distributed across multiple nodes of a SAS grid. Running independent multiple tasks in parallel will usually result in a reduction in total elapsed time for the job. This paper will discuss practical considerations for introducing parallel processing into a SAS grid job, including identifying types of concurrency and how best to divide the program into subtasks. We will cover the basics of how to add the required code, including how to pass information from one task to another. This paper will provide a sample program to illustrate code modifications to take advantage of parallel processing on the grid, and cover some real-life examples of programs that have been move to the grid using these techniques.

INTRODUCTION

As of earlier this year, RTI has completed the transition process of migrating over 400 SAS users from PC SAS® and a few standalone SAS servers to using a SAS grid platform (Wilson, Green, and Terminello 2017.) This decision was made partly to reduce long term costs by centralizing SAS administration, allowing for easier expansion of capacity and simplifying training and the onboarding of new users. Another goal was improved processing over all by better sharing of resources. A byproduct of this new environment is the opportunity to introduce parallel processing to some of our more resource-intensive SAS programs.

Long-running SAS jobs often include sets of independent subtasks that can be split up and distributed across a SAS grid. When these subtasks are then run in parallel the total elapsed time can often be made to decrease even though the total CPU time may increase. For programmers, it's generally the elapsed time that matters. We often need to cut down on how long something runs in order to meet aggressive project deadlines.

Not every program can be made faster with parallel processing. If each step in the program must execute in sequence, then it can't be divided into parallel subtasks. For some applications where the speed could be improved, perhaps it isn't worth the effort if the amount of time the job runs isn't a problem. To determine if parallelizing your program will significantly improve elapsed time you need to understand what factors will have an impact on elapsed time, and decide how you can break your program down into subtasks that can be run in parallel. What types of concurrency can you take advantage of? If you have tasks that operate against different data, then they may be candidates to run independently. If you have tasks that perform different operations on the same data and are not otherwise dependent on one another, those tasks also might be candidates for running in parallel.

The SAS Code Analyzer, PROC SCAPROC, can be used to create a new version of your program to run on the grid, and that can be very helpful in cases where you have a lot of SAS code that others have written and need to assess it quickly, but there is no substitute for a good SAS programmer taking the time to review a complex program and rework it to run some tasks in parallel. That is the programmer we are addressing here: one who can take the time to analyze their own code and their own environment to come up with the fastest way to get the job done.

CONSIDERATIONS FOR PARALLEL PROCESSING

Once you have identified the possible steps within your program that could be run in parallel, how do you decide how to divide it into independent subtasks? How many subtasks should you try to have -- as many as possible? This depends on a number of factors related to your program and your environment.

Is there a limit to how many jobs or processes that a user can have running at once? How much work should be done within each subtask? For example, suppose your program has 100 steps all of which could be run independently. Does it make sense to set up a job to submit all 100 at once in parallel? Keep in mind that the overhead of starting up SAS is added to each subtask, so CPU time may increase. If the subtasks themselves don't run for very long, elapsed time might increase too. The extra work of the spawning the additional SAS sessions might not be outweighed by the savings of running in parallel.

You can benchmark multiple approaches, for example test running the 100 steps in groups of 5, 10 or 20 to help determine what the optimal number seems to be. If your environment has a limit on the number of jobs a programmer can run at once that will need to be a factor in your decision-making. You'll want to revisit the approach against the benchmark any time the program changes in such a way that the amount of parallelization can change.

Another thing to consider is whether the subtasks share anything: do they read the same data, pass data from one task to another, or pass macro variables? When you have a single job, you don't need to worry about these things as much because temporary data and formats are kept in the WORK library, but these temporary datasets are not available to another subtask submitted separately to the grid. If your parallel subtasks all need a particular library, you could have a libname statement in each subtask. Another approach would be to use the INHERITLIB option.

But what if your subtasks share more than that – what if they all use common macro variables, or create data sets in WORK that are needed in subsequent subtasks? For macro variables, you can define them in each subtask, or you can pass them to other remote sessions via the %SYSLPUT function. For data to pass from one step to another, you could use a libname other than WORK to pass data from one step to another. You can also use the PRESENV option and procedure to save global statements (such as %let or libname) for reuse in another SAS session. These techniques are discussed further below

There are also factors outside of your control that can have an impact on any given solution, such as physical location of grid servers as compared to the location of your client machine and the location of the input data; volume and destination of the output and log; or network bandwidth. If you are running jobs interactively with the log and output streamed back to SAS Enterprise Guide software® or SAS Studio, then the volume of the log and output and the time it takes to present it back to the SAS client can have an impact on the amount of clock time your job takes. Our examples were all run using batch submissions from a Linux SSH client, and we don't address the impact of large SAS logs and output.

GETTING STARTED

There are a few basic statements and functions that you need to use when converting a program to use distributed parallel processing: a call to the GRDSVC_ENABLE function, a RSUBMIT block around each subtask, and possibly other statements and options to get the behavior you want. For example, if we have two tasks that can run in parallel, two different analyses for example, the code is structured like this:

```
%let rc=%sysfunc(grdsvc_enable(_all_, resource=SASApp));
options autosignon;

rsubmit task1 wait=no ;
  <code for the first task>
endrsubmit;

rsubmit task2 wait=no ;
  <code for the first task>
endrsubmit;

waitfor _all_ task1 task2;

signoff _all_ ;
```

The GRDSVC_ENABLE function is used to enable grid execution. The AUTOSIGNON option automatically signs on to a remote session which eliminates the need for explicit SIGNON statements. The RSUBMIT/ENDRSUBMIT block contains the code for the subtask being submitted to the grid as a remote task, in other words it is submitted as a separate SAS session from your main program. The WAITFOR statement causes the main program to wait for one or more of your subtasks to complete before continuing. The SIGNOFF statement is used to end all parallel connections.

What else you need depends on what the code in your RSUBMIT block is doing. For example, there may be LIBNAME, OPTIONS and %LET statements, or other global statements that you would normally put at the start of a program. If these libraries or options are needed by the code in each RSUBMIT block, then you need to either include the required statements in each RSUBMIT block or find another way to pass the information. The final program might look like this:

```
%let rc=%sysfunc(grdsvc_enable(_all_, resource=SASApp));
options autosignon;

rsubmit task1 wait=no ;
  options nocenter nodate nonumber ps=max mprint fullstimer ;
  libname in "/vol5/project/CHDA_Library/publications/SESUG2017_mrabb/data";
  proc freq data=in.gridtestflags;
    tables proc_code_1 proc_code_50/missing;
    title 'Counts on selected Procedure code flags';
  run;
endrsubmit;

rsubmit task2 wait=no ;
  options nocenter nodate nonumber ps=max mprint fullstimer ;
  libname in "/vol5/project/CHDA_Library/publications/SESUG2017_mrabb/data";
  proc means data=in.gridtestbenes;
    var birth_dt;
    format birth_dt date9.;
    title 'Statistics on Birth date';
  run;
endrsubmit;

waitfor _all_ task1 task2;

signoff _all_ ;
```

PASSING INFORMATION TO SUBTASKS

The library defined by the IN libref is needed by both subtasks so the libname statement is repeated in each RSUBMIT block since these represent separate SAS sessions. There are several ways to avoid repeating such information in each RSUBMIT block. If all you need to do is make a library accessible to the subtask use the INHERITLIB option. Assuming the libname statement for the IN libref appears earlier in your main program, the RSUBMIT statements would take this form, and the libname statement inside the RSUBMIT blocks are no longer needed:

```
rsubmit task1 wait=no connectpersist=no inheritlib=(in);
```

Macro variables defined in the main program may also be needed in subtasks. The %SYSLPUT function is executed in your main program and makes a macro variable or group of macro variables available in the remote session, for example:

```
%SYSLPUT indata=gridbenes/REMOTE=task2;
```

Or:

```
%let data1=gridbenes;  
%let data2=gridflags;  
%syslput _all_;
```

Another approach for more complicated situations is to use the PRESENV Procedure and system option. This technique can be used to preserve all global statements and macro variables from one SAS session to the next, thus making them available in each of your subtasks. For more information see the Base SAS documentation, or see the referenced SAS Global Forum paper (Horwitz 2017) for some good examples of using PRESENV in a SAS Enterprise Guide environment.

PROCESS DATA SOURCES SEPARATELY

Suppose you have several large datasets that need to be sorted in preparation for merging. Perhaps you also want to run some simple statistics on each set of input data. Since each task operates against a separate data source, they can be sorted and analyzed in parallel subtasks. In our example, the four input datasets had 25 million observations each. The sequential job consisting of four SORT and MEANS procedures run sequentially ran for just under 25 minutes. When run using four independent parallel tasks the job took about 11 minutes.

PARALLEL PROCESSING CASE STUDY

We have a large data set of 1 million observations and need to perform 100 analytic steps (FREQUENCY procedures) using this data. These steps are not dependent on one another so they could all be run in parallel, and we combine the results at the end to create a single table of output.

Figure 1 Sample Output

variable	value	COUNT	PERCENT
proc_code_1	0	1000802	50.0401
proc_code_1	1	999198	49.9599
proc_code_2	0	999713	49.9857
proc_code_2	1	1000287	50.0144
proc_code_3	0	1000398	50.0199
proc_code_3	1	999602	49.9801
proc_code_4	0	1000132	50.0066
proc_code_4	1	999868	49.9934

Before the grid, this program consisted of 100 PROC FREQS run sequentially in one job. We ran that job to get a baseline of how long this would run without parallelization. Next, we tried several different approaches to parallelization. In the first approach, we submitted each PROC FREQ in a separate subtask, meaning we were submitting 100 remote tasks from within the main job. In our environment, an individual user can only run 16 jobs at once, so the best-case scenario would be to have up to 16 subtasks running at once, and as each subtask finishes a new one can start. Don't forget each subtask is starting a new SAS session and there is overhead start-up time associated with that. We found that submitting this work as 100 subtasks did speed it up in terms of elapsed time, but given how quickly each PROC FREQ ran once it started we were sure we could do better with fewer jobs, meaning less SAS start-up time. We tried submitting 10 jobs, each one containing 10 PROC FREQ steps, and we saw a further improvement there.

All of these frequencies run against the same SAS dataset, meaning there could be an I/O bottleneck with multiple jobs reading the same data set at once. As an experiment, we tried splitting the data into ten parts, one section for each of the ten subtasks. Each data subset had only the ten variables needed for the ten frequencies being run in the specific task. Of course, the data step to split the data into parts added to the elapsed time over all, but the ten subtasks ran significantly faster than before, meaning this was the fastest of the tests we ran. The code for the final program can be found in the Appendix.

Table 1 Elapsed Time Comparisons

Program	Description	Real time
Baseline	100 FREQ procedures run sequentially	40:17.00
Ptables_all	100 FREQ procedures in 100 subtasks	14:46.85
Ptables_10	10 subtasks, each with 10 FREQ procedures	4:27.56
Ptables_data10	Split data, then 10 subtasks as above	2:12.56

ANOTHER EXAMPLE

Several of our projects use a bootstrapping technique that involves running 1000 GLIMMIX Procedure programs. Depending on the amount of data being processed and the exact model being run, this bootstrapping program can run for days, sometimes for over a week. Since we migrated to the SAS grid, a couple of different approaches have been used to run these PROC GLIMMIX steps in parallel. In one case the programmer used SAS techniques similar to those illustrated in this paper to submit the 1000 procedures in multiple subtasks. His program is generalized and scalable to use from 2 to 15 subtasks. Since our limit on the number of jobs per person is 16, he needs to make sure he always has the ability to run another job on the grid if all of his subtasks are running at once. This application runs six or seven times faster when running in parallel than when running sequentially.

ADDITIONAL CONSIDERATIONS

Another approach to the Bootstrapping problem used Linux scripts to submit multiple SAS jobs to the grid, in other words they are submitted as separate batch jobs, rather than as a single batch job that used RSUBMIT to submit the subtasks. We don't illustrate the use of Linux scripts here, but the topic is covered in Brinsfield 2017 referenced below.

Many of our programs make extensive use of Macro programming, for example our job that submitted 100 subtasks (one for each PROC FREQ) did so using a Macro %DO loop. You have to be especially careful when using Macro programming in conjunction with RSUBMIT blocks. The code inside a RSUBMIT block is executed in the remote session, but macro code can be compiled in the main SAS program task, which may not be what you intended. In order to simplify our code, we chose to use some hard coding of steps in our final version of the PROC FREQ program.

For additional programming techniques and more advanced use of grid software to control execution of subtasks, see Doug Haigh's excellent paper referenced below (Haigh, 2016)

CONCLUSION

Migration to a SAS grid presents new opportunities to improve the performance of long-running SAS jobs by introducing parallel processing. With proper analysis of the existing programs and consideration of the factors affecting performance on the grid, substantial improvements in elapsed time can often be achieved.

REFERENCES

Brinsfield, Eric C. 2017. "SAS Programmer's Guide to Life on the SAS Grid." *Proceedings of the 2017 PharmaSUG Conference*, Baltimore, Maryland. Available at <https://www.pharmasug.org/proceedings/2017/BB/PharmaSUG-2017-BB11.pdf>

Haigh, Doug 2016, "Divide and Conquer—Writing Parallel SAS® Code to Speed Up Your SAS Program" *Proceedings of the SESUG 2016 Conference*, Bethesda, Maryland. Available at: http://analytics.ncsu.edu/sesug/2016/PA-265_Final_PDF.pdf

Horwitz, Lisa 2017 "A Long-Time SAS® Programmer Learns New Tricks" *Proceedings of the 2017 SAS Global Forum*, Orlando, Florida. Available at: <http://support.sas.com/resources/papers/proceedings17/SAS0637-2017.pdf>

Wilson, David, Annette M. Green, and Karen Terminello. 2017. "Getting on SAS Grid, Our Journey." *Proceedings of the 2017 South East SAS Users Group (SESUG) Conference*, Cary, NC.

CONTACT INFORMATION

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

Merry Rabb
RTI International
919-990-8343
mrabb@rti.org

Keith Brown
RTI International
919-316-3177
kebrown@rti.org

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX: CASE STUDY MACROS

This appendix contains the final SAS code for the Case Study example. There is a main program called `ptables_data10` followed by the code for two Macros.

```

/*****
/* Programmer: Merry Rabb */
/* Program: ptables_data10 */
/* Date: 09SEP2017 */
/* Description: Frequencies for testing parallel processing */
/* - Splits input data into 10 pieces */
/* - Runs proc freq with out= for N flag variables. */
/* - hard-coded for 10 submissions. */
/* - Combines counts into a single table at the end. */
/* - Uses pfreqvar_data10, which uses separate libnames */
/* for input data in each submission via modification */
/* to macro freq. */
*****/

/* data library (with subfolder templib) */
%let mylib=/CHDA_Library/publications/SESUG2017_mrabb/data;
/* location for the output spreadsheet */
%let xlsout=/CHDA_Library/publications/SESUG2017_mrabb/data/freqout;
/* include file for freq macro definition */
%let incl=/CHDA_Library/publications/SESUG2017_mrabb/programs/freq.sas;
/* include file for freq macro definition */

libname in "&mylib";

```

```

libname templib "&mylib./templib";

%LET PROJECT=GRID TESTING;
%LET PROGRAM=ptables_data10;
options nocenter nodate nonumber ps=max fullstimer ;
ods listing close;

title 'Frequencies run on Grid - example with 10';

/* contains the macro to do run the proc freqs */
%include
"/CHDA_Library/publications/SESUG2017_mrabb/programs/pfreqvar_data10.sas";

/* sets macro variable flagvars - long list of variable names */
/* ct_f is set to the number of var names in the list */
%include
"/CHDA_Library/publications/SESUG2017_mrabb/programs/includeVars.sas";

%let ct_f=%sysfunc(countw(&flagvars));
%put number of vars is &ct_f;

/* split input data into 10 separate sets in templib */
%macro split;
  data
    %do i=1 %to 10;
      %let start=%eval(((&i-1) *10) + 1);
      %let stop=%eval(((&i-1) *10) + 10);

      templib.data&i(keep=

        %do j=&start %to &stop;
          proc_code_&j
        %end;
        /* end keep option */
      )
    %end;
  ; /* end data statement */
  set in.gridtestflags;
run;
%mend split;

%split
;

/* run flagfreq macro from freqvar_parallel10_fromkb */
%let rc=%sysfunc(grdsvc_enable(_all_, resource=SASApp));
options autosignon;

%flagfreq(data=data, folder=&mylib, list=&flagvars, n=&ct_f);

/* produce final output table */
ods html file="&xlsout/&program._&sysdate..xls" style=minimal;

proc freq data=in.gridtestbenes;
  tables sex/missing nocum ;
run;

```

```

/* data set report is output from flagfreq */
option missing=0;
proc print noobs data=report;
run;
options missing=' ';

ods html close;

/*****
/* Programmer: Keith Brown, Merry Rabb */
/* Program: pfreqvar_data10.sas (%flagfreq) */
/* Date: 10SEP2017 */
/* Description: Proc freq tables - grid testing - hard code 10 jobs */
/* Data split into 10 portions in calling program */
*****/

/* parallel processing - use templib in place for WORK for intermediate
data */
*libname templib "&folder/templib";
%let rc=%sysfunc(grdsvc_enable(_all_,resource=SASApp));
options autosignon;

%macro flagfreq(data=, list=, folder=, n=);

/* macro variables to be passed to subtasks */
/* datasets are numbered due to split */
%macro pass1(n1=);
%SYSLPUT indata=&data&n1/REMOTE=task&n1;
%SYSLPUT invars=&list/REMOTE=task&n1;
%SYSLPUT inlib=&folder/REMOTE=task&n1;
%SYSLPUT tcnt=&n1/REMOTE=task&n1;
%SYSLPUT inclfile=&incl/REMOTE=task&n1;
%mend pass1;

/* task 1 */
%pass1(n1=1);

rsubmit task1 wait=no connectpersist=no;
libname in "&inlib";
libname templib "&inlib/templib";

/* include freq macro */
%include "&inclfile";

/* call freq macro for this subtask */
data _null_;
do i=10*(&tcnt-1)+1 to &tcnt*10;
call execute('%freq(i=' ||
compress(put(i,2.)) || ',name=temp)');
end;
run;

endrsubmit;

```

```

/* task 2 */
%pass1(n1=2);

rsubmit task2 wait=no connectpersist=no;
  libname in "&inlib";
  libname templib "&inlib/templib";

  %include "&inclfile";

  data _null_;
  do i=10*(&tcnt-1)+1 to &tcnt*10;
    call execute('%freq(i=' ||
      compress(put(i,2.)) || ',name=temp)');
  end;
run;

endrsubmit;

/* task 3 */
%pass1(n1=3);

rsubmit task3 wait=no connectpersist=no;
  libname in "&inlib";
  libname templib "&inlib/templib";

  %include "&inclfile";

  data _null_;
  do i=10*(&tcnt-1)+1 to &tcnt*10;
    call execute('%freq(i=' ||
      compress(put(i,2.)) || ',name=temp)');
  end;
run;

endrsubmit;

/* task 4 */
%pass1(n1=4);

rsubmit task4 wait=no connectpersist=no;
  libname in "&inlib";
  libname templib "&inlib/templib";

  %include "&inclfile";

  data _null_;
  do i=10*(&tcnt-1)+1 to &tcnt*10;
    call execute('%freq(i=' ||
      compress(put(i,2.)) || ',name=temp)');
  end;
run;

endrsubmit;

/* Similar TASKS 5 through 9 deleted due */
/* to space limitations for this paper */

```

```

/* task 10 */
%pass1(n1=10);

rsubmit task10 wait=no connectpersist=no;
  libname in "&inlib";
  libname templib "&inlib/templib";

  %include "&inclfile";

      data _null_;
      do i=10*(&tcnt-1)+1 to &tcnt*10;
          /* format to accommodate 3 digits */
          call execute('%freq(i=' ||
                      compress(put(i,3.)) || ',name=temp)');
      end;
      run;

  endrsubmit;

/* once all proc freqs are done, make a single table */
/* from the intermediate files */

waitfor _all_
  %do i=1 %to 10; /* 10 tasks */
    task&i
  %end;
; /* end waitfor stmt */
signoff _all_;

data report(keep=variable value count percent);
  length variable $20 value 8;
  set
  %do i=1 %to &n;
    %let var=%scan(&list,&i);
    templib.freq&var(in=in&var)
  %end;
  ;

  %do i=1 %to &n;
    %let var=%scan(&list,&i);
    %if &i=1 %then %do;
      if in&var then
        do;
          variable="&var";
          value=&var;
        end;
    %end;
    %else %do;
      else if in&var then
        do;
          variable="&var";
          value=&var;
        end;
    %end;
  %end;

```

```

        label count='Frequency'
              percent='Percentage' ;
run;

/* delete "temporary" data */

proc datasets nolist library=templib kill;
run;
quit;
%mend flagfreq;

/*****
/* Programmer: Merry Rabb, Keith Brown
/* Program: freq.inc
/* Date: 10SEP2017
/* Description: Definition of freq macro for including
/* Included in subtasks of %flagfreq, from
/* pfreqvar_data10.sas
*****/

/* paramter i is which iteration (task) is running */
/* name is either in for default input or temp for */
/* when data sets are split in temporary library */

%macro freq(i=,name=);
    %put i=&i name=&name;

    %if &name=in %then libname &name&i "&inlib";
    %else libname &name&i "&inlib/templib";
    ;

    proc freq data=&name&i..&indata noprint;
        tables proc_code_&i /missing out=templib.freqproc_code_&i ;
    run;

%mend freq;

```