Paper 870-2017

Parallel Processing Your Way to Faster Software and a Big Fat Bonus: Demonstrations in Base SAS®

Troy Martin Hughes

ABSTRACT

SAS® software and especially extract-transform-load (ETL) systems commonly include components that must be serialized due to real process dependencies. For example, a transform module often cannot begin until the data extraction completes, and a corresponding load module cannot begin until the data transformation completes. While process dependencies such as these cannot be avoided in many cases and necessitate serialized software design, in other cases, programs or data can be distributed across two or more SAS sessions to be processed in parallel, facilitating significantly faster software. This text introduces the concept of *false dependencies*, in which software is serialized by design rather than necessity, thus needlessly increasing execution time and deprecating performance. Three types of false dependencies are demonstrated as well as distributed software solutions that eliminate false dependencies through parallel processing, arming SAS practitioners to accelerate both their software and salaries.

INTRODUCTION

Dependencies are an unavoidable reality of software and, in one definition, a *dependency* exists whenever a software module or component requires prerequisite user input, data input, or the initiation or completion of some module, task, or other activity. Dependencies aren't all bad as they often prescribe business rules that improve data quality or software quality. Especially in SAS data analytic development, dependencies often ensure that necessary data quality controls are enforced, helping drive and demonstrate data integrity. For example, the requirement that data be transformed before they are loaded within an ETL infrastructure yields confidence that requisite data cleaning, grooming, and standardization have occurred before data are passed to stakeholders or dependent processes.

In some cases, (false) dependencies result purely from the sequencing of code rather than necessity. For example, if one data set is ingested in a DATA step and, upon completion, a second DATA step ingests an unrelated data set, these two DATA steps could in theory be run at the same time. Similarly, if the MEANS procedure is run followed by the FREQ procedure, a false dependency exists because the procedures could be executed concurrently but are not. Both *false dependencies of process* and *false dependencies of data sets* involve sequencing processes in series that could be executed in parallel, and are described and distinguished in the following two sections.

A *false dependency of throughput* is a third type of false dependency and describes an unnecessarily serialized process that occurs *within* a procedure or DATA step rather than *between* software modules. For example, when data are ingested, typically the first observation is read, followed by the second, and so on until the entire data set is ingested sequentially. In some cases, this serialization is required, but in other cases, non-sequential input/output (I/O) processing can effectively distribute the data set for parallel processing. Some SAS procedures such as SORT or MEANS take advantage of this divide-and-conquer logic through multithreading, introduced in SAS 9, but creative I/O engineering can implement similar logic within the DATA step or other developer-built processes.

The identification and removal of false dependencies can dramatically improve software performance, yielding vigorous execution. One distinction should be made, however, that software *efficiency* is not being improved—only software *speed*. Parallel processing and distributed computing models often require more system resources (e.g., CPU cycles, memory, disk space) due to the increased overhead of running concurrent SAS sessions and, in some cases, the coordination and communication required among active sessions. Notwithstanding the increased resource consumption, the enhanced software performance often substantially outweighs the slight decrease in efficiency.

FALSE DEPENDENCIES OF PROCESS

A *false dependency of process* occurs when two or more processes could be executed simultaneously but instead are sequenced in serialized software design. A classic example of this false dependency occurs when a data set is created and thereafter is processed through a series of analyses that are independent of each other. For example, the following code creates the LIB.test data set, then runs the MEANS procedure on LIB.test, then runs the CORR procedure on LIB.test:

```
data lib.test;
    set lib.dsn;
run;

proc means data=lib.test;
    var numvar1 numvar2 numvar3 numvar4;
run;

proc corr data=lib.test;
    var numvar1 numvar2;
run;
```

However, as both MEANS and CORR require only a shared lock on the LIB.test data set, the two procedures could be run simultaneously to produce faster results. If the previous code is functionally decomposed from one program into three distinct modules, the MEANS and CORR procedures can be run in parallel. This transition from serialized to parallel software design is depicted in Figure 1, including the substantial time savings during execution.

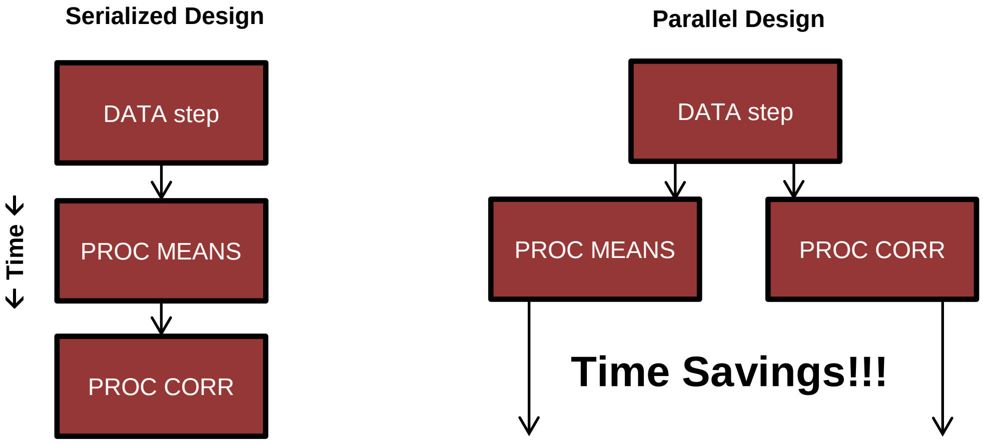

Figure 1. Transition from Serialized to Parallel Software Design

The following software engine can be saved as Engine.sas and spawns two SAS sessions that execute MEANS and CORR procedures in parallel:

```
* this code saved as ENGINE.SAS;
data lib.test;
    set lib.dsn;
run;

systask command """%sysget(SASROOT)\sas.exe"" -noterminal -nosplash -sysin
    ""C:\means.sas"" -log ""C:\means.txt"" -print
    ""C:\means.txt""" taskname=task_means status=rc_means;

systask command """%sysget(SASROOT)\sas.exe"" -noterminal -nosplash -sysin
    ""C:\corr.sas"" -log ""C:\corr.txt"" -print
    ""C:\corr.txt""" taskname=task_corr status=rc_corr;

waitfor _all_ task_means task_corr;
```

When executed in the SAS Display Manager (aka the windowing environment), the SYSTASK command spawns two new SAS sessions asynchronously (i.e., the second session does not wait for the first to complete, but executes immediately after the first session initiates) that run Means.sas and Corr.sas, respectively. The WAITFOR statement waits for both asynchronous sessions to complete, after which additional code (not shown) could start executing.

Note that that Means.sas and Corr.sas code must be saved as two separate programs, respectively, and that each program must have access to the LIB library and LIB.Test data set:

```
* this code saved as C:\MEANS.SAS;
proc means data=lib.test;
    var numvar1 numvar2 numvar3 numvar4;
run;


* this code saved as C:\CORR.SAS;
proc corr data=lib.test;
    var numvar1 numvar2;
run;
```

Another common false dependency of process occurs when separate but prerequisite processes must complete before a subsequent process can execute. For example, if two data sets must be joined to create a third data set, each data set might require separate sorting or transformation prior to the final join. In traditional data analytic design, these processes might be coded in series; however, parallel design could be implemented to remove false dependencies to facilitate faster software. Figure 2 represents the shift from serialized to parallel design required to remove the false dependency (in which the DSN2 sort must unnecessarily wait for the DSN1 sort to complete).

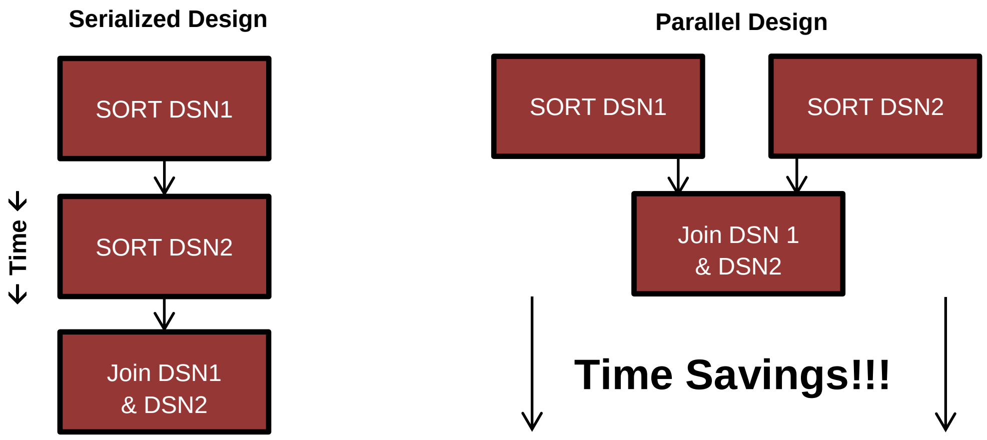

Figure 2. Transition from Serialized to Parallel Software Design

One drawback of the use of SYSTASK is its lack of availability within the SAS University Edition. Notwithstanding, the SYSTASK command is not the only method to implement asynchronous or parallel design, although it remains one of the most viable methods within the SAS Display Manager. Use of SYSTASK is detailed extensively in the “Automation” chapter of the author’s text *SAS Data Analytic Development: Dimensions of Software Quality*.[i]

FALSE DEPENDENCIES OF DATA SETS

A *false dependency of data sets* also involves data processes or modules that are unnecessarily serialized. The principle difference from a false dependency of process is that this false dependency occurs because a rote process is waiting for a data set to be processed before a subsequent iteration can process a subsequent data set. For example, the following code sorts three data sets in series and, at first glance, may appear to be well written because the macro code essentially kills three birds (data sets) with one stone (macro invocation):

```
%macro serial_sort(dsnlist=);
%let i=1;
%do %while(%length(%scan(&dsnlist,&i,,S))>1);
    %let dsn=%scan(&dsnlist,&i,,S);
    proc sort data=&dsn;
       by charvar1;
    run;
    %let i=%eval(&i+1);
    %end;
%mend;

%serial_sort(dsnlist=perm.dsn1 perm.dsn2 perm.dsn3);
```

The use of serialized input passed through the DSNLIST macro parameter may be an effective and efficient way to *write* this software, but its implementation is an exceptionally slow way to *run* this software because it introduces two false dependencies. Thus, the program cannot start sorting DSN2 until DSN1 has completed, and cannot start sorting DSN3 until DSN2 has completed. In theory, all data sets could be sorted in parallel with sufficient resources.

To remove these two false dependencies, all three data sets can be sorted simultaneously, again using a parent process (aka engine) to asynchronously spawn batch child processes that sort the respective data sets in one-third the time, assuming data sets of similar size and complexity. The following code should be saved as Engine.sas:

```
* this code saved as ENGINE.SAS;
%macro parallel_sort(dsnlist=);
%local sortvar i;
%let i=1;
%do %while(%length(%scan(&dsnlist,&i,,S))>1);
    systask command """%sysget(SASROOT)\sas.exe"" -noterminal -nosplash -sysin
    ""C:\sort.sas"" -log ""C:\sort&i..txt"" -sysparm ""dsn=dsn&i, dsnout=dsn&i,
    vars=charvar""" taskname=task_sort&i status=rc_task&i;
    %let i=%eval(&i+1);
    %end;
%let i=%eval(&i-1);
waitfor _all_
    %do j=1 %to &i;
       task_sort&j
       %end;;
%mend;

%parallel_sort(dsnlist=lib.dsn1 lib.dsn2 lib.dsn3);
```

The Engine.sas program spawns three instances of the Sort.sas program which run simultaneously in three separate SAS sessions. The log files created by the three sessions must be dynamically named to ensure that the SAS sessions do not attempt to write to the same log file, which would cause a file access collision. Thus, the first instance of Sort.sas creates the log file Sort1.txt, the second instance Sort2.txt, and the third instance Sort3.txt. The task name and status, whose functionality is not described in this text, also must be dynamically named with the incremental macro variable &I to facilitate success of SYSTASK.

The name of the data set to be sorted is dynamically passed through the SYSPARM parameter, which is evaluated in the respective instances of the Sort.sas program as the global macro variable &SYSPARM. The GETPARM macro, included in the Sort.sas program for reference, parses the SYSPARM parameter, tokenizes comma-delimited values, and dynamically assigns the parameterized values to global macro variables &DSN, &DSNOUT, and &VARS. Significantly more customization could be implemented by supplementing the SYSPARM parameter with additional fields and values to be parsed by the child processes via GETPARM. Thus, in the following code, when the first instance of Sort.sas executes, it sorts LIB.DSN1 by the CHARVAR variable:

```
* this code saved as C:\SORT.SAS;
%macro getparm();
%local i;
%let i=1;
%do %while(%length(%scan(%quote(&sysparm),&i,','))>1);
    %let var=%scan(%scan(%quote(&sysparm),&i,','),1,=);
    %let val=%scan(%scan(%quote(&sysparm),&i,','),2,=);
    %global &var;
    %let &var=&val;
    %let i=%eval(&i+1);
    %end;
%mend;

%getparm;

proc sort data=&dsn out=&dsnout;
    by &vars;
run;
```

The WAITFOR statement in the Engine.sas program again waits for each respective child process to complete (as indicated by the values assigned to the respective TASK_SORT&J macro variables). The three sorts are asynchronously executed in parallel and, after the last sort has completed, subsequent code (not demonstrated) could be executed. Given sufficient system resources and data sets of similar size and complexity, this design will sort these three data sets in the same time that would have been necessary to sort one data set. This time savings is demonstrated in Figure 3.

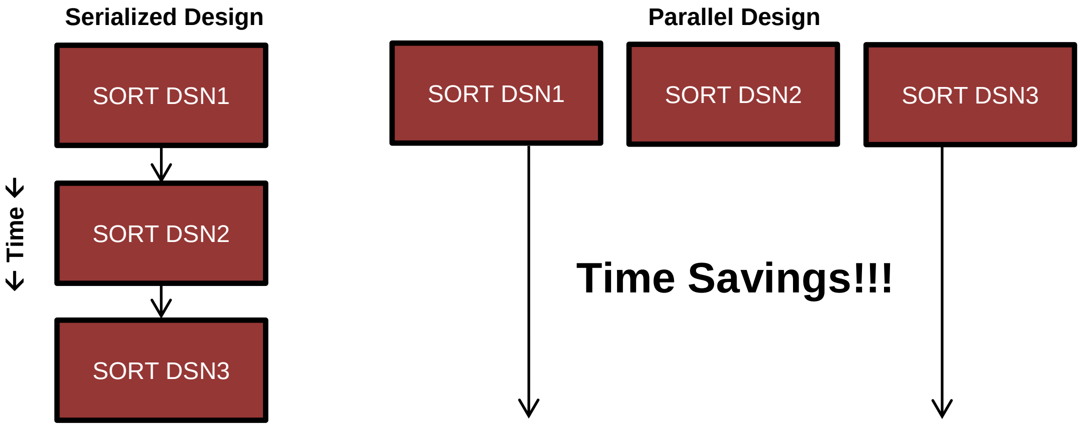

Figure 3. Transition from Serialized to Parallel Software Design

FALSE DEPENDENCIES OF THROUGHPUT

A *false dependency of throughput* exists not between processes but rather within a process due to serialized mechanics often endemic to the software language itself. For example, if a data process or procedure requires that observations be input or output in sequence, this can cause substantial delays when big data are encountered. With sufficient system resources, however, some serialized input and analysis processes can be distributed through a divide-and-conquer design that distributes disparate data across multiple SAS sessions for simultaneous processing.

For example, given a data set of 1 billion observations, the FIRSTOBS and OBS options in the SET statement of the DATA step could be implemented to read the data set in parallel as four distinct chunks. The first chunk would be created by setting FIRSTOBS to 1 and OBS to 250 million, the second chunk by setting FIRSTOBS to 250,000,001 and OBS to 500 million, the third chunk by setting FIRSTOBS to 500,000,001 and OBS to 750 million, and the fourth chunk by setting FIRSTOBS to 750,000,001 and OBS to 1 billion. The MAKE_GROUPS macro dynamically creates a space-delimited list of these FIRSTOBS and OBS pairings when the total number of observations and desired number of data chunks are provided as parameterized input:

```
%macro make_groups(obs= /* number of observations in data set */,
    groups= /* number of desired chunks into which to break data set */);
%global grouplist;
%let grouplist=;
%local i obs1 obs2;
%do i=1 %to &groups;
    %if &i=1 %then %let obs1=1;
    %else %let obs1=%sysevalf(&obs2+1);
    %if &i=&groups %then %let obs2=&obs;
    %else %let obs2=%sysevalf(&i*&obs/&groups,ceil);
    %let grouplist=&grouplist &obs1 &obs2;
    %end;
%mend;
```

After execution of MAKE_GROUPS, the &GROUPLIST macro variable can be used to assign values dynamically for FIRSTOBS and OBS pairs. These pairs can be passed through successive, dynamic SYSTASK commands that dictate parameterized values of LOW and HIGH that can be substituted into the FIRSTOBS and OBS options, respectively, within each child process. A parent process spawning four child processes of unspecified functionality (not demonstrated) could be represented in the following code fragment:

```
%macro distributed_process(dsn= /* data set name being distributed */,
    somevar= /* a variable that might be required */,
    groups= /* number of groups into which to divide data set for parallel
       processing */);
%local obs i low high;
proc sql noprint;
    select count(*) format=15. into :obs from &dsn;
quit;
%make_groups(obs=&obs, groups=&groups);
%do i=1 %to &groups;
    %let low=%scan(&grouplist,%sysevalf(((&i-1)*2)+1),,S);
    %let high=%scan(&grouplist,%sysevalf(&i*2));
    systask command """%sysget(SASROOT)\sas.exe"" -noterminal -nosplash -sysin
       ""C:\run_some_child_process.sas"" -log ""C:\run_some_child_process&i..txt""
       -print "" C:\run_some_child_process&i..out"" -sysparm ""dsn=&dsn,
       somevar=&somevar, low=&low, high=&high""" taskname=task&i status=rc&i;
    %end;
waitfor _all_
    %do i=1 %to &groups;
       task&i
       %end;;
*do some process after completion of the four parallel child processes;
%mend;

%distributed_process(dsn=lib.dsn1, somevar=charvar1, groups=4);
```

Thus, the FIRSTOBS and OBS pairs, after dynamic assignment by MAKE_GROUPS, can be used with a DATA step, FREQ procedure, or other procedure or process to input or analyze data in parallel. Figure 4 depicts the performance advantage of this divide-and-conquer method, subsetting a data set into four equally sized data chunks.

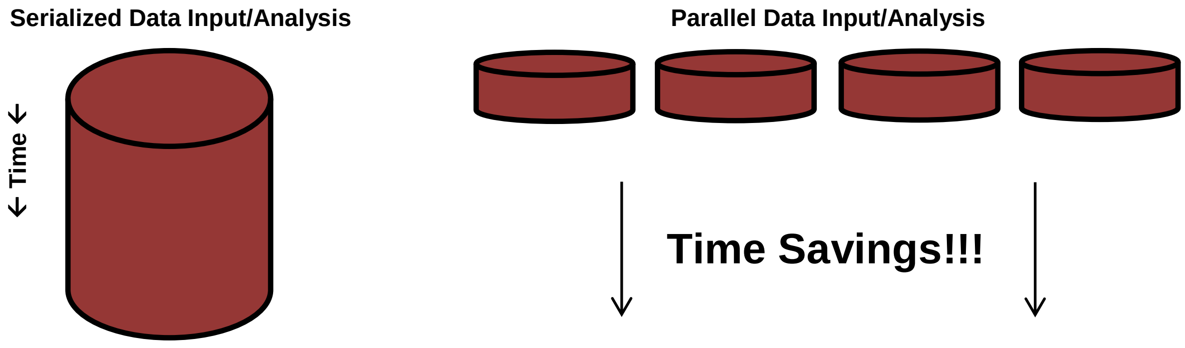

Figure 4. Divide-and-Conquer Design to Remove False Dependencies of Throughput

Divide-and-conquer solutions can vary in complexity and success depending on the type of process that is involved. For example, if Figure 4 represents the comparison between an out-of-the-box serialized SORT procedure and a divide-and-conquer SORT procedure run on four 250 million-observation chunks, an additional process (not depicted) would be required to aggregate the four sorted chunks into a final sorted data set. The author demonstrates a divide-and-conquer sort in a separate text: *Sorting a Bajillion Records: Conquering Scalability in a Big Data World*.[ii]

Finally, while divide-and-conquer solutions sometimes improve performance, they can also degrade performance due to system resource overhead or the relative efficiency of out-of-the-box SAS functionality, especially for multithreaded procedures. Load testing or stress testing should be employed both to demonstrate the range of file sizes for which parallel processing provides performance improvement, as well as the range of groups (i.e., number of subset data chunks) that optimizes performance. For example, SAS practitioners might discover that a specific divide-and-conquer solution improves performance only for data sets larger than 1 GB and only when the number of data chunks ranges from between four to six. Testing takes time but is essential in justifying the switch to a distributed solution.

CONCLUSION

In shifting from serialized to parallel design that removes false dependencies, the performance of SAS software can be dramatically improved as simultaneous SAS sessions execute software in parallel and as system resources are consumed concurrently rather than in series. With sufficient system resource availability, tremendous time savings can be achieved. A shift from monolithic to modular software design is often required, allowing parent processes to spawn batch child processes that asynchronously execute in parallel. While this shift can require substantial redesign and refactoring of software, when technical requirements or business needs specify faster software, the removal of false dependencies through parallel design can be an effective programmatic solution to achieve these objectives.

REFERENCES

[i] Troy Martin Hughes. 2016. *SAS Data Analytic Development: Dimensions of Software Quality*. John Wiley and Sons, Inc.

[ii] Troy Martin Hughes. 2016. *Sorting a Bajillion Records: Conquering Scalability in a Big Data World*. SAS Global Forum.

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com