PharmaSUG 2019 – Paper AD-228

User-Defined Multithreading with the SAS® DS2 Procedure: Performance Testing DS2 Against Functionally Equivalent DATA Steps

Troy Martin Hughes

ABSTRACT

The Data Step 2 (DS2) procedure affords the first opportunity for developers to build custom, multithreaded processes in Base SAS®. Multithreaded processing debuted in SAS 9, when built-in procedures such as SORT, SQL, and MEANS were threaded to reduce runtime. Despite this advancement, and in contrast with languages such as Java and Python, SAS 9 still did not provide developers the ability to create custom, multithreaded processes. This limitation was overcome in SAS 9.4 with the introduction of the DS2 procedure—a threaded, object-oriented version of the DATA step. However, because DS2 relies on methods and packages (neither of which have been previously available in Base SAS), both DS2 instruction and literature has predominantly fixated on these object-oriented aspects rather than DS2 multithreading. Each paired example concludes with performance metrics that compare the DATA step with the DS2 language—even on a stand-alone laptop. All examples can be run in Base SAS and do not require in-database processing or the purchase of the DS2 Code Accelerator or other optional SAS components.

INTRODUCTION

This text focuses squarely on DS2 multithreading and compares performance of multithreaded DS2 processes with functionally equivalent DATA steps. In *Overview of Threaded Processing*, SAS documentation states that "DS2 threading works well both on a machine with multiple cores and within a massively parallel processing (MPP) database."[i] This text focuses on the former environment only—multi-core machines, which can include servers, desktops, and even laptop instances running Base SAS.

Examples compare three modalities: single-threaded processing, multithreaded processing (in the DS2 procedure), and multiprocessing (invoked with the SYSTASK statement). Single-threaded processing occurs when the DATA step reads and processes data—one observation is read at a time and acted upon. Multithreaded processing occurs through the DS2 procedure, where each observation is farmed out to one thread for processing. In so doing, multiple threads can simultaneously process different observations; however, as performance testing in this text demonstrates, bottlenecks still occur because the input/output (I/O) tasks of reading and writing are still performed as a serialized action.

Multiprocessing can be invoked in SAS by having the SYSTASK statement spawn separate instances of SAS. In so doing, these instances cannot communicate with each other (like threads) and thus are less efficient because each SAS instances has the additional overhead of an entire SAS session (not just a thread). However, the benefits of multiprocessing (over multithreading) are the lack of I/O bottlenecks, because I/O processing can also occur in parallel (unlike multithreading). Thus, the FIRSTOBS and OBS statements can be used to subset specific observations in parallel, which cannot be accomplished using the DS2 procedure. This means that in several instances, as demonstrated in this paper, despite multiprocessing being less efficient than DS2 multithreading, it nevertheless performs faster.

DS2 MULTITHREADING LITERATURE REVIEW

The most ubiquitous example is:

```
options fullstimer;
proc ds2;
   thread t / overwrite=yes;
      dcl int x;
      method init();
         dcl int i;
         do i=1 to 3;
```

```
            x=i;
            output;
            end;
         end;
      endthread;
run;
data;
   dcl thread t t_instance;
   method run();
      set from t_instance threads=2;
      put 'x= ' x;
      end;
   enddata;
run;
quit;
```

The output follows, always with the caveat that the order may vary, yet it never does:

```
165  options fullstimer;
166  proc ds2;
167     thread t / overwrite=yes;
168        dcl int x;
169        method init();
170           dcl int i;
171           do i=1 to 3;
172              x=i;
173              output;
174              end;
175           end;
176        endthread;
177  run;
NOTE: Created thread t in data set work.t.
NOTE: Execution succeeded. No rows affected.
178  data;
179     dcl thread t t_instance;
180     method run();
181        set from t_instance threads=2;
182        put 'x= ' x;
183        end;
184     enddata;
185  run;
x=  1
x=  2
x=  3
x=  1
x=  2
x=  3
186  quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           6.30 seconds
      user cpu time       0.07 seconds
      system cpu time     0.51 seconds
      memory              3937.00k
      OS Memory           24816.00k
      Timestamp           04/18/2019 12:49:56 PM
      Step Count                        12  Switch Count  6
```

The order never varies because the first thread completes so quickly that it is finished before the second thread can begin. In essence, the code is designed to be multithreaded, and in truth is multithreaded, but nevertheless is processed in series—not parallel—because the limitations of the data that are supplied.

SINGLE-THREADING VERSUS DS2 MULTITHREADING

Multithreading always consumes more system resources than functionally equivalent single-threaded processing because effort is required to coordinate the concurrent processes and, in many cases, because a terminal step is required to aggregate the various threads into a composite data set or other data product. Thus, multithreaded processes may perform faster than their single-threaded brethren, but they will never perform more efficiently—as efficiency requires the analysis not only of performance time but also system resources utilized.

The following DATA step creates the Testdata set from the SASHELP library:

```
libname ds2 'D:\sas\ds2';
data ds2.testdata (drop=i);
   set sashelp.cars (obs=100);
   do i=1 to 500000;
      output;
      end;
run;
```

A simple, single-threaded DATA step performs the single action of reading the Testdata data set:

```
data ds2.testserial;
   set ds2.testdata;
run;
```

This produces the following log output:

```
NOTE: There were 50000000 observations read from the data set DS2.TESTDATA.
NOTE: The data set DS2.TESTSERIAL has 50000000 observations and 15 variables.
NOTE: DATA statement used (Total process time):
      real time           27.16 seconds
      user cpu time       2.53 seconds
      system cpu time     2.86 seconds
      memory              628.96k
      OS Memory           34296.00k
```

The DATA step does not appear to be CPU-bound, so there should not be an expectation of multithreading providing greater performance—but let's try anyway. And, this is expected, because the only actions are input/output (I/O) related—reading the data and writing the data.

The following DS2 procedure, comprising a THREAD step and a DATA step, is functionally equivalent but runs on four threads rather than one:

```
proc ds2;
   thread t1 / overwrite=yes;
      method run();
         set ds2.testdata;
         end;
      endthread;
run;
data ds2.testmulti / overwrite=yes;
   dcl thread t1 t_instance;
   method run();
      set from t_instance threads=4;
      end;
   enddata;
run;
quit;
```

The output demonstrates that with four threads running, the DS2 alternative performs noticeably slower (and uses ten times more memory) than the respective single-threaded DATA step:

```
NOTE: PROCEDURE DS2 used (Total process time):
      real time           1:53.05
```

```
        user cpu time       13.85 seconds
        system cpu time     5.07 seconds
        memory              5950.56k
        OS Memory           37624.00k
```

The DS2 procedure took four times longer to complete! It also used significantly more system resources, including both CPU cycles and memory!

To understand which threads are doing what work, the _THREADID_ automatic variable can be added to the Testmulti output data set as a new variable Threadno:

```
proc ds2;
   thread t1 / overwrite=yes;
      dcl int threadno;
      method run();
         set ds2.testdata;
         threadno=_threadid_;
         end;
      endthread;
run;
data ds2.testmulti / overwrite=yes;
   dcl thread t1 t_instance;
   method run();
      set from t_instance threads=4;
      end;
   enddata;
run;
quit;
```

When a frequency analysis is subsequently run on Threadno, Table 1 reveals that the work was fairly evenly distributed across the four threads.

threadno	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	12703725	25.41	12703725	25.41
2	12774961	25.55	25478686	50.96
3	12144287	24.29	37622973	75.25
4	12377027	24.75	50000000	100.00

Table 1. First Frequency Analysis

If run an addition time, however, the results might vary slightly, as demonstrated in Table 2.

threadno	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	12691657	25.38	12691657	25.38
2	12331772	24.66	25023429	50.05
3	12476278	24.95	37499707	75.00
4	12500293	25.00	50000000	100.00

Table 2. Second Frequency Analysis

The goal of multithreading system resource utilization is to build a process in which the Real Time (experienced by the user) is less than the CPU time, indicating that concurrent work was being done behind the scenes. In fact, the hallmark indicator of multithreading is not necessarily faster processing (than functionally equivalent single-threaded processing), but rather CPU time that exceeds real time. For example, consider the MEANS procedure, which can be used to sum numeric data across observations. The following macro builds a one million-observation data set having 100 numeric variables:

```
%macro makenums(dsn=, obs=, numvar=, nummax=);
data &dsn (drop=i j);
   array nums 8 nums1-nums&numvar;
   do i=1 to &obs;
      do j=1 to dim(nums);
         nums{j}=int(rand('uniform')*&nummax);
         end;
      output;
      end;
run;
%mend;

%makenums(dsn=ds2.somedata, obs=1000000, numvar=100, nummax=100000);
```

This DATA step is single-threaded and, as the output demonstrates, the CPU time is near the real time:

```
NOTE: The data set DS2.SOMEDATA has 1000000 observations and 100 variables.
NOTE: DATA statement used (Total process time):
      real time           2.73 seconds
      user cpu time       2.43 seconds
      system cpu time     0.32 seconds
      memory              497.90k
      OS Memory           37880.00k
```

The multithreaded MEANS procedure now sums each of the 100 variables across 1 million observations to produce an output data set:

```
proc means data=ds2.somedata noprint;
   var nums1-nums100;
   output out=meanout (drop=_type_ _freq_) sum=sum1-sum100;
run;
```

The output does demonstrate evidence of multithreading, in that the CPU time dramatically exceeds the real time:

```
NOTE: There were 1000000 observations read from the data set DS2.SOMEDATA.
NOTE: The data set WORK.MEANOUT has 1 observations and 100 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.70 seconds
      user cpu time       5.07 seconds
      system cpu time     0.20 seconds
      memory              6017.62k
      OS Memory           43000.00k
```

A single-threaded, functionally equivalent DATA step produces the identical summation:

```
data sumdata (keep=sum1-sum100);
   set ds2.somedata end=eof;
   array nums 8 num1-num100;
   array sums 8 sum1-sum100;
   retain sum1-sum100 0;
   do over sums;
      sums+nums;
      end;
   if eof then output;
run;
```

The single-threaded DATA step takes only a split-second longer, so the benefits of multithreading are shown; however, so too are the disadvantages. The following output demonstrates that the single-threaded CPU time was five times less than the multithreading equivalent:

```
NOTE: There were 1000000 observations read from the data set DS2.SOMEDATA.
NOTE: The data set WORK.SUMDATA has 1 observations and 100 variables.
NOTE: DATA statement used (Total process time):
      real time           0.98 seconds
      user cpu time       0.84 seconds
      system cpu time     0.18 seconds
      memory              932.12k
      OS Memory           37880.00k
```

It's clear from these examples that the first step in productive multithreading is to identify tasks that will actually benefit from being multithreaded—and CPU-bound processes is a great place to start this search. However, it also demonstrates that optimization must follow both to demonstrate that multithreading is faster, and also to demonstrate the number of threads that should be run. These topics are further explored.

CPU-BOUND PROCESSES

SAS Institute defines *CPU-bound* as applications that "receive data faster than they can perform the necessary processing on that data."[ii] Further SAS documentation expounds on how real time and CPU time can be utilized to evaluate whether a process is CPU-bound:

> If the Real time and total CPU time are within 15 percent of each other, this usually indicates that the system is moving data well (at least during the run time of that job/step processing). This means that the ratio of CPU process time is close to that of the total job. This indicates that the system memory, disk system, and file system are getting data to the CPU quickly enough to not be a problem. If you are experiencing bad task performance, and the real and CPU time are within 15 percent of each other, it most likely means that your task is CPU bound. The only way to improve the performance will be to get a faster CPU, split the process over more CPUs (multi-threading or parallel processing), or reengineer the code to be more efficient.[iii]

The operative phrase in interpreting the foregoing quote is "bad task performance," an enigma that's thrown at the reader, yet left undefined. On the one hand, SAS states that if real time and CPU time are close, your system is "moving well." Think of all those "get regular and stay regular" Metamucil "fiber therapy" commercials, some of which even touted getting a raise due to better bowels.[iv]

But the difference between "moving well" and "bad performance," unless defined and quantified, really comes down to perception. What's more important than this perception, is that single-threaded CPU-bound processes, regardless of whether they are perceived to be slow or not, can often be made faster through multithreading.

The MAKEDATA macro is used to create test data for performance testing or load testing and is featured in two of the author's white papers.[v, vi] A simplified version of MAKEDATA is presented here, which dynamically creates a test data set having a varying number of character variables of varying length:

```
%macro makedata(dsn= /* data set being created */,
   obs= /* number of observations */,
   charvar= /* number of character variables */,
   charlen= /* length of character variables */);
data &dsn (drop=i j k);
   array chars $&charlen char1-char&charvar;
   do i=1 to &obs;
      do j=1 to &charvar;
         chars{j}='';
         do k=1 to &charlen;
            chars{j}=cats(chars{j},byte(int(rand('uniform')*10)+65)); *A to J;
            end;
         end;
      output;
      end;
run;
%mend;
```

The MAKEDATA macro can be invoked to create a 100,000-observation data set:

```
%makedata(dsn=somedata, obs=100000, charvar=100, charlen=20);
```

This produces the following output, showing a real time and CPU time that are remarkably close:

```
NOTE: The data set WORK.SOMEDATA has 100000 observations and 100 variables.
NOTE: DATA statement used (Total process time):
      real time           15.08 seconds
      user cpu time       15.07 seconds
      system cpu time     0.04 seconds
      memory              723.75k
      OS Memory           36924.00k
```

Given the nested loops required to create multiple observations and variables, and given the loop required to randomize the characters that are produced, it’s not at all unexpected that the CPU would be fully engaged during this DATA step. And, whether you perceive the 16-second performance to be adequate or poor, the DATA step is an excellent candidate for multithreading with DS2.

This can be converted to DS2 with the following code:

```
proc ds2;
   thread thr / overwrite=yes;
      vararray char(20) charvar[1:100] charvar1-charvar100;
      method run();
         dcl int i j k;
         do i=1 to 12500;
            do j=1 to 100;
               charvar[j]='';
               do k=1 to 20;
                 charvar[j]=cats(charvar[j],byte(int(rand('uniform')*10)+65));*A-J;
                  end;
               end;
            output;
            end;
         end;
      endthread;
run;
data ds2.testmulti / overwrite=yes;
   dcl thread thr t_instance;
   method run();
      set from t_instance threads=8;
      end;
   enddata;
run;
quit;
```

When this executes, the multithreading is clearly evident because the CPU time dramatically exceeds the real time; however, the multithreaded real time nevertheless exceeds the single-threaded real time. The output follows:

```
NOTE: PROCEDURE DS2 used (Total process time):
      real time           18.00 seconds
      user cpu time       2:22.09
      system cpu time     0.12 seconds
      memory              9233.32k
      OS Memory           45388.00k
```

OPTIMIZING THREAD COUNT

The example in the previous section created a new data set Testdata. a 1.5GB data set having 10 million observations. The multithreaded DS2 procedure was significantly slower than the respective single-threaded DATA step, but the DS2 procedure was run with only four threads. Thus, it’s possible that optimizing the total number of available threads (via the THREADS statement) could improve performance. It’s also important that more than one test run be utilized for performance testing, so repeated measures are utilized.

The following TESTTHREADS macro tests the performance of the DS2 procedure by varying the number of threads from 1 to 24. The single-threaded DATA step is also included in this process:

```
%include 'd:\sas\pinchlog\pinchlog.sas';
%macro testthreads(loop= /* max number of threads */,
   log= /* log file */,
   metrics= /* metrics data set */);
%local i j;
proc ds2;
   thread t1 / overwrite=yes;
      method run();
         set testdata;
         end;
      endthread;
run;
quit;
%do j=1 %to 20;
   * single-threaded;
   %let syscc=0;
   proc printto log="&log" new;
   run;
   data testserial;
      set testdata;
   run;
   proc printto;
   run;
   %pinchlog(logfile=&log, dsnmetrics=&metrics,
      othervars=(var=method, val=DATA Step, len=$10, form=$10., lab=Method /
         var=threads, val=1, len=8, form=8., lab=Threads /
         var=err, val=&syscc, len=8, form=8., lab=Error Code));
   * multithreaded;
   %do i=1 %to &loop;
      %let syscc=0;
      proc printto log="&log" new;
      run;
      proc ds2;
         data testmulti / overwrite=yes;
         dcl thread t1 t_instance;
         method run();
            set from t_instance threads=&i;
            end;
         enddata;
      run;
      quit;
      proc printto;
      run;
      %pinchlog(logfile=&log, dsnmetrics=&metrics,
         othervars=(var=method, val=PROC DS2, len=$10, form=$10., lab=Method /
            var=threads, val=&i, len=8, form=8., lab=Threads /
            var=err, val=&syscc, len=8, form=8., lab=Error Code));
      %end;
   %end;
%mend;

%testthreads(loop=24, log=D:\sas\log.txt, metrics=threadmetrics);
```

The PINCHLOG macro captures the FULLSTIMER system resource utilization metrics. Four user-defined variables are created, including:

- Method – differentiates the DATA step from the DS2 procedure
- Threads – shows the maximum number of threads that may be used
- Err – demonstrates the &SYSCC automatic macro variable after completion
- CPUCOUNT – shows the CPUCOUNT SAS system option

The CPUCOUNT can be evaluated with the following code:

```
%put %sysfunc(getoptions(cpucount));
```

In some cases, the CPUCOUNT may be initially set to a lower-than-necessary value. For example, despite the CPUCOUNT showing 4 in the previous log output, the following code (on certain systems that have available resources) will reset this to 8 CPUs:

```
options cpucount=actual;
```

This macro creates a data set Threadmetrics that includes the real time and CPU time for processes as the number of threads is varied. Figure 1 demonstrates that in this case, despite the attempt to optimize thread count, the DATA step nevertheless performs faster than the multithreaded DS2.

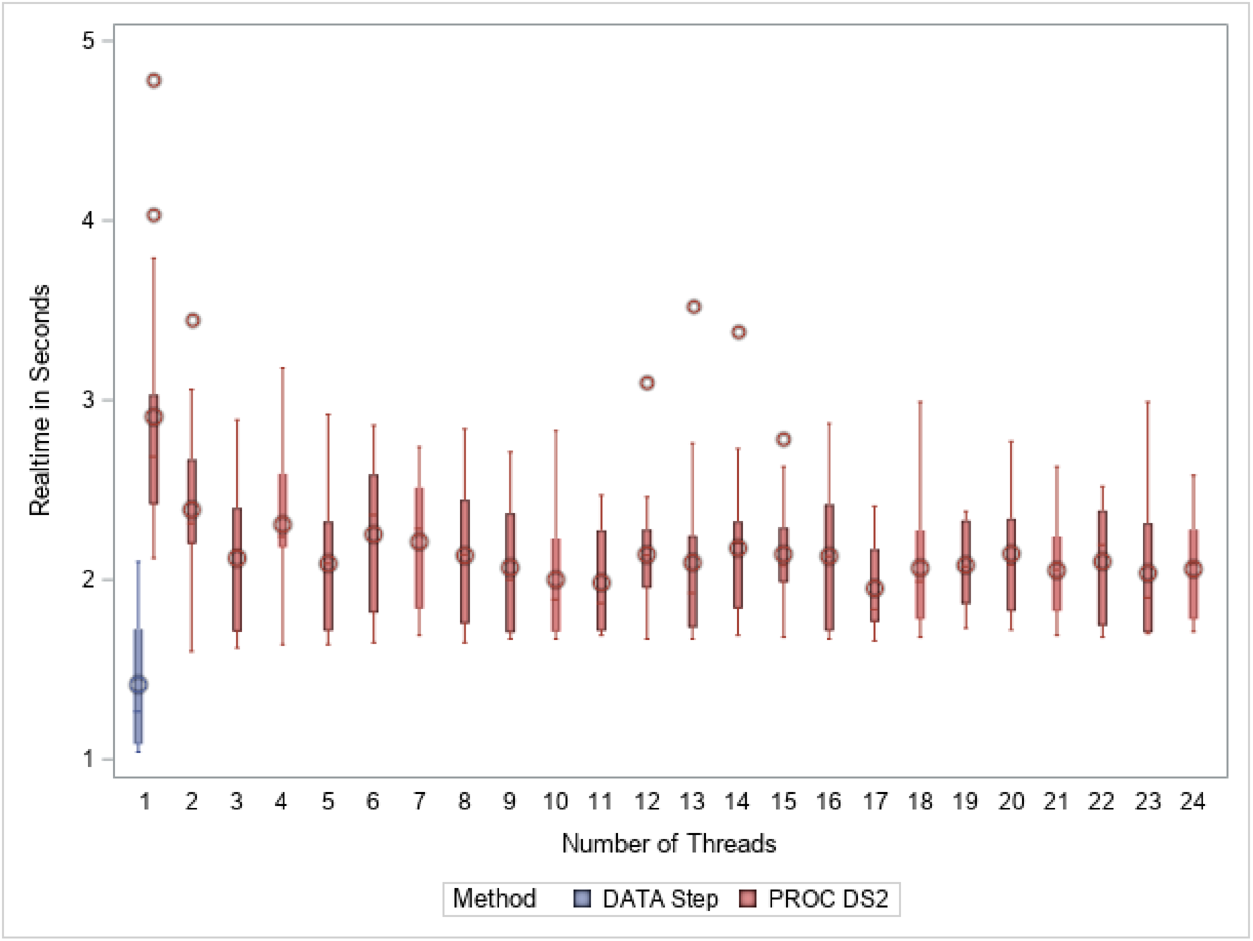

Figure 1. Demonstration of Higher Performance of DATA Step over DS2 Procedure

Note that the slowest performance results when the DS2 procedure uses THREADS=1, in which case the procedure has the extra weight of DS2 yet no benefits of multithreading.

CONCLUSION

DS2 multithreading performs faster than functionally equivalent single-threaded DATA steps in certain, very specific computationally complex tasks, especially those in which the ratio of computations (and the complexity thereof) to observation count is high. In these examples, the effects of I/O processing on performance are minimized and the DS2 procedure can shine. However, in other instances in which the ratio of computations to I/O actions is lower, despite not being I/O-bound, the DS2 procedure typically performs slower than both the single-threaded DATA step and multiprocessing methods that implement the SYSTASK statement. Although the DS2 procedure can be beneficial in maximizing performance through multithreading, SAS practitioners must invest considerable effort into performing testing and tuning the DS2 procedure to evaluated whether its performance exceeds functionally equivalent single-threading or multiprocessing.

REFERENCES

[i] SAS® Viya 3.2: DS2 Programmer's Guide. SAS Institute. *Overview of Threaded Processing*. Retrieved from https://documentation.sas.com/?docsetId=ds2pg&docsetTarget=p0qykqw1fdra8dn1449vxg9ydfkk.htm&docsetVersion=3.2&locale=en.
[ii] SAS® 9.3 Language Reference: Concepts, Second Edition. SAS Institute. *Threaded Application Processing.* Retrieved from http://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#p0wxn869zvk4itn15k1hhro4qw5h.htm.
[iii] Resources / Focus Areas: Scalability and Performance. SAS Institute. *FULLSTIMER SAS Option*. Retrieved from https://support.sas.com/rnd/scalability/tools/fullstim/index.html.
[iv] Metamucil Get Regular and Stay Regular. 1988. Retrieved from https://www.youtube.com/watch?v=zZnZeCbCUDg.
[v] Troy Martin Hughes. From FREQing Slow to FREQing Fast: Facilitating a Four-Times-Faster FREQ with Divide-and-Conquer Parallel Processing. PharmaSUG 2018. Retrieved from https://www.lexjansen.com/pharmasug/2018/AA/PharmaSUG-2018-AA07.pdf.
[vi] Troy Martin Hughes. Pinching Off Your SAS® Log: Adapting from Loquacious to Laconic Logs To Facilitate Near-Real Time Log Parsing, Performance Analysis, and Dynamic, Data-Driven Design and Optimization. PharmaSUG 2018. Retrieved from https://www.lexjansen.com/pharmasug/2018/AD/PharmaSUG-2018-AD07.pdf.

CONTACT INFORMATION

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

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