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 programming (OOP) aspects of the language rather than DS2 multithreading. To complicate the adoption of DS2 multithreading, one of the most ubiquitous examples of "multithreading" promulgated throughout SAS documentation and literature unfortunately fails to show any performance advantages in using DS2 over previous single-threaded methods—so many would-be DS2 developers may have cautiously dipped their toes in the multithreaded waters and, horrified with their own performance testing results from these published examples, quickly retreated back to the safer DATA step land. This text explores DS2 multithreading and demonstrates performance testing between DS2 procedures and functionally equivalent DATA steps and SAS procedures.

## INTRODUCTION

This text focuses on DS2 multithreading and compares performance of multithreaded DS2 processes with functionally equivalent DATA steps and SAS procedures. 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 laptops running Base SAS.

Single-threaded processing occurs when the DATA step ingests data, reading and interacting with observations in series. Multithreaded processing occurs through the DS2 procedure, where observations are farmed out to different threads 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 in series.

Prior to the development of the DS2 language, SAS had already implemented multithreading in many of its procedures (e.g., SORT, SQL, MEANS) with the release of SAS 9. Unfortunately, the FREQ procedure was left behind in single-threaded Sheol, so it lacks these performance improvements. For a clear demonstration of the advantages of multithreading over single-threaded processing, the author demonstrates a parallelized FREQFAST macro that uses divide-and-conquer methods to produce frequency results four times faster than the out-of-the-box SAS FREQ procedure, in: *From FREQing Slow to FREQing Fast: Facilitating a Four-Times-Faster FREQ with Divide-and-Conquer Parallel Processing*.[ii]

This text demonstrates the advantages of DS2 multithreading over single-threaded, functionally equivalent DATA steps—*in certain computationally intensive scenarios, and given sufficient system resources*. In many cases, the DS2 procedure performed slower than the DATA step while consuming substantially more system resources. Thus, before embarking on potentially fruitless, exhausting journeys that aim to overhaul legacy DATA steps with DS2 code, SAS practitioners should understand resource utilization as well as the types of problem sets for which DS2 multithreading is an ideal solution. Armed with this information, refactoring DATA steps to DS2 multithreaded procedures has the capacity to improve greatly the performance of legacy SAS software.

## SHOW ME WHAT YOU WORKIN' WITH

One of the first priorities in performance testing, similar to Mystikal's *Shake It Fast*, is to "show me what you workin' with." The following code evaluates the number of CPUs available to SAS, sets the CPUCOUNT to the maximum (i.e., "AVAILABLE"), and subsequently displays both the CPUCOUNT (which may have increased) and available memory (in GBs):

```
%let cpucount=%sysfunc(getoption(cpucount));
%put CPUs: &cpucount;
options fullstimer cpucount=actual;
%let cpucount=%sysfunc(getoption(cpucount));
%let memsize=%sysfunc(putn(%sysevalf(%sysfunc(getoption(xmrlmem))
   /1073741824),8.2));
%put CPUs: &cpucount;
%put Mem (GBs): &memsize;
%let loc=D:\sas\ds2\;       * CHANGE TO USER-SPECIFIED LOCATION!!! *;
libname ds2 "&loc";
```

On the desktop on which this performance testing was conducted for this text, the output demonstrates that four CPUs were available by default, although this was increased to eight with the OPTIONS CPUCOUNT statement. The output also demonstrates that 19.15 GBs of memory are available for the SAS application:

```
CPUs: 4
CPUs: 8
Mem (GBs): 19.15
```

This system, having 8 CPUs, should be sufficient to show performance advantages yielded by DS2 multithreading. All examples demonstrate this 8-CPU system, except where a separate 4-CPU system is explicitly denoted.

## 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, 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 of not only performance time but also system resource (e.g., memory, CPU cycles) utilization.

The following DATA step creates the 50-million observation Test data set derived from the SASHELP library (SASHELP.Cars):

```
%let loc=D:\sas\ds2\;       * CHANGE TO USER-SPECIFIED LOCATION!!! *;
libname ds2 "&loc";
data ds2.test (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 Test data set:

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

This produces the following log output:

```
NOTE: There were 50000000 observations read from the data set DS2.TEST.
NOTE: The data set DS2.TESTSERIAL has 50000000 observations and 15 variables.
NOTE: DATA statement used (Total process time):
      real time           1:34.79
```

```
      user cpu time       2.76 seconds
      system cpu time     3.18 seconds
      memory              621.71k
      OS Memory           31216.00k
```

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

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.test;
         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 log demonstrates that with four threads running, the DS2 procedure performs slower while using nine 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
```

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 additional time, however, the results should vary, 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 is 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 first builds a ten-million observation data set having 100 numeric variables that will be used in subsequent examples:

```
%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=10000000, numvar=100, nummax=100000);
```

The log follows::

```
NOTE: The data set DS2.SOMEDATA has 10000000 observations and 100 variables.
NOTE: DATA statement used (Total process time):
      real time           47.36 seconds
      user cpu time       23.68 seconds
      system cpu time     1.75 seconds
      memory              497.62k
      OS Memory           28148.00k
```

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

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

```
run;
```

The SUM function in the MEANS procedure is an ideal scenario for multithreading—because observations are not dependent on each other, and thus can be added in separate bundles (in parallel, on separate threads), then combined in one, final serialized aggregation step. Moreover, because the OUT statement produces only one observation, the tedious I/O work of writing ten million observations is avoided.

The output follows, with multithreading clearly evinced by the substantially greater CPU time than real time:

```
NOTE: There were 10000000 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           6.68 seconds
      user cpu time       50.93 seconds
      system cpu time     1.98 seconds
      memory              6034.46k
      OS Memory           33268.00k
```

However, by specifying the NOTHREADS option, the MEANS procedure can produce identical output albeit while utilizing only one thread:

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

When executed, the output demonstrates substantially greater real time than the equivalent multithreaded MEANS procedure:

```
NOTE: There were 10000000 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           47.99 seconds
      user cpu time       46.90 seconds
      system cpu time     1.11 seconds
      memory              1731.62k
      OS Memory           29172.00k
```

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

```
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 slightly longer than the functionally equivalent multithreaded MEANS procedure. However, note that the DATA step required substantially less CPU time and substantially less memory than the MEANS procedure:

```
NOTE: There were 10000000 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           9.14 seconds
      user cpu time       7.90 seconds
      system cpu time     1.26 seconds
      memory              849.31k
      OS Memory           28148.00k
```

A functionally equivalent DS2 procedure can also be constructed to sum these data. To demonstrate its relative efficiency and performance to the previous single-threaded DATA step, this DS2 procedure is initially run on a single thread:

```
proc ds2;
   thread s / overwrite=yes;
      vararray double nums[100];
      vararray double sums[100];
      method run();
         dcl integer i;
         set ds2.somedata;
         do i=1 to 100;
            sums[i]+nums[i];
            end;
         end;
      method term();
         output;
         end;
      endthread;
run;
data sumdata (drop=(nums1-nums100 sums1-sums100))/ overwrite=yes;
   dcl thread s t_instance;
   vararray double numsums[100];
   vararray double sums[100];
   method run();
      dcl integer i;
      set from t_instance threads=1;
      do i=1 to 100;
         numsums[i]+sums[i];
         end;
      end;
   method term();
      output;
      end;
   enddata;
run;
quit;
```

Running on a single thread, the DS2 procedure performs slower than either the multithreaded MEANS procedure or single-threaded DATA step:

```
NOTE: Execution succeeded. One row affected.
2344   quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           10.40 seconds
      user cpu time       16.82 seconds
      system cpu time     1.37 seconds
      memory              7964.70k
      OS Memory           33288.00k
```


Increasing the number of concurrent threads to four and to eight also fails to yield faster performance, so unfortunately in this specific instance, and when tested on this specific data set, the DS2 procedure is not the overall performance winner.

## THE UBIQUITOUS (AND AWFUL) MULTITHREADING EXAMPLE

The most ubiquitous example of DS2 multithreading (which originates in the SAS documentation *Overview of Threaded Processing*) follows:

```
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;
```

This example is repeated verbatim in several subsequent publications, each of which contains no other multithreaded processing examples.[iii, iv, v, vi] The output follows, with documentation always decrying 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
```

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 data supplied are insufficient to demonstrate multithreading.

## EXPANDING THE UBIQUITOUS EXAMPLE

This invariance can be better demonstrated by adding the SAS automatic variable _THREADID_, which identifies the thread number that produced output. The following code now additionally demonstrates _THREADID_, which is passed to the Threadno variable and printed to the log:

```
proc ds2;
   thread t / overwrite=y;
      dcl int x;
      dcl int threadno;
      method init();
         dcl int i;
         do i = 1 to 3;
            x = i;
            threadno=_threadid_;
            output;
            end;
         end;
      endthread;
data;
   dcl thread t t_instance;
   method run();
      set from t_instance threads=2;
      put 'x= ' x threadno=;
      end;
   enddata;
run;
quit;
```

The output follows, demonstrating that thread 1 completed before thread 2 had started:

```
x=  1 threadno=1
x=  2 threadno=1
x=  3 threadno=1
x=  1 threadno=2
x=  2 threadno=2
x=  3 threadno=2
NOTE: Created thread t in data set work.t.
94   quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.06 seconds
      user cpu time       0.03 seconds
      system cpu time     0.04 seconds
      memory              3720.28k
      OS Memory           26080.00k
```

However, run the same code again, and you will likely see an example in which Thread 2 wholly completes before Thread 1 starts:

```
x=  1 threadno=2
x=  2 threadno=2
x=  3 threadno=2
x=  1 threadno=1
x=  2 threadno=1
x=  3 threadno=1
NOTE: Created thread t in data set work.t.
116  quit;
```

```
NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.07 seconds
      user cpu time       0.01 seconds
      system cpu time     0.03 seconds
      memory              3702.46k
      OS Memory           26080.00k
```

Notwithstanding the order in which the threads execute, in both cases, once a thread executes, it runs completely before the next thread can begin. Even after modifying the DO loop to count from one to 10,000 (rather than to 3), the log indicates that the threads still effectively execute in series rather than parallel, with one thread completing before the other begins. Note that because three threads are specified, and each thread iterates 10,000 times, the entire procedure contains 30,000 iterations total.

Increasing the number of threads (e.g., from 2 to 8) might yield the result that the example purports to deliver—that is, varied order of the output:

```
proc ds2;
   thread t / overwrite=y;
      dcl int x;
      dcl int threadno;
      method init();
         dcl int i;
         do i = 1 to 10000;
            x = i;
            threadno=_threadid_;
            output;
            end;
         end;
      endthread;
data;
   dcl thread t t_instance;
   method run();
      set from t_instance threads=8;
      put 'x= ' x threadno=;
      end;
   enddata;
run;
quit;
```

With eight threads and 10,000 iterations per thread, the output finally demonstrates some actual multithreading. For example, in a test run, thread 1 first prints the numbers from one to 10,000. Thereafter, thread 8 begins and prints the numbers from one to 4,096; however, at this point, thread 2 breaks in and starts iterating from 1 to 4,096. This trade-off continues, demonstrating threads swapping in and out, and by saving this output to a text file (d:\sas\ds2\ds2log.txt), it can be analyzed:

```
%let loc=D:\sas\ds2\;       * CHANGE TO USER-SPECIFIED LOCATION!!! *;

data switch;
   infile "&loc.ds2log.txt" truncover end=eof dlm=' ';
   length x $3 num 4 y $12 threadno 3 oldnum 4 oldthread 3 start 4 line $40;
   input x $ num y $ threadno;
   if _n_=1 then start=1;
   else do;
      if oldthread^=threadno then do;
         line='Thread: ' || strip(put(oldthread,8.)) || '  iterations '
            || strip(put(start,8.)) || ' - ' || strip(put(oldnum,8.));
         start=num;
         put line;
         end;
      end;
   oldnum=num;
   oldthread=threadno;
```

```
   if eof then do;
      line='Thread: ' || strip(put(threadno,8.)) || '  iterations '
         || strip(put(start,8.)) || ' - ' || strip(put(num,8.));
      put line;
      end;
      retain oldnum oldthread start;
run;
```

The output clearly demonstrates the transition points among threads:

```
Thread: 8  iterations 1 - 10000
Thread: 3  iterations 1 - 4096
Thread: 1  iterations 1 - 4096
Thread: 4  iterations 1 - 4096
Thread: 3  iterations 4097 - 8192
Thread: 1  iterations 4097 - 8192
Thread: 6  iterations 1 - 4096
Thread: 3  iterations 8193 - 10000
Thread: 1  iterations 8193 - 10000
Thread: 4  iterations 4097 - 10000
Thread: 7  iterations 1 - 4096
Thread: 6  iterations 4097 - 8192
Thread: 5  iterations 1 - 4096
Thread: 6  iterations 8193 - 10000
Thread: 7  iterations 4097 - 8192
Thread: 5  iterations 4097 - 8192
Thread: 7  iterations 8193 - 10000
Thread: 5  iterations 8193 - 10000
Thread: 2  iterations 1 - 10000
```

When run multiple times, the same patterns (e.g., 4,096 and 4,097; 8,192 and 8,193) are repeated, even when the number of threads and number of iterations are increased. No explanation is apparent from SAS DS2 or multithreading documentation, so this eccentricity should be explored further.

## COMPARING THE UBIQUITOUS EXAMPLE TO SINGLE-THREADED PROCESSING

The previous DS2 procedure is modified to instead create the Multi data set, which comprises 80,000 observations produced by eight threads iterating 10,000 times each:

```
proc ds2;
   thread t / overwrite=y;
      dcl double x;
      method init();
         dcl int i;
         do i = 1 to 10000;
            x = i;
            output;
            end;
         end;
      endthread;
data multi / overwrite=y;
   dcl thread t t_instance;
   method run();
      set from t_instance threads=8;
      end;
   enddata;
run;
quit;
```

Note that the Threadno variable has also been removed to streamline the procedure to only the elements required for basic functionality. The log demonstrates FULLSTIMER performance metrics:

```
NOTE: Created thread t in data set work.t.
NOTE: Execution succeeded. 80000 rows affected.
```

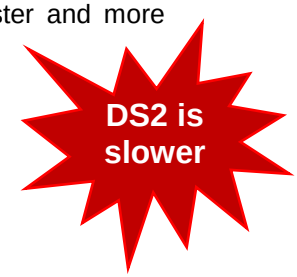
```
352  quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.06 seconds
      user cpu time       0.03 seconds
      system cpu time     0.03 seconds
      memory              4195.48k
      OS Memory           23788.00k
```

The functionally equivalent, single-threaded DATA step creates the Single data set:

```
data single (keep=i);
   length i 8;
   do thread=1 to 8;
      do i=1 to 10000;
         output;
         end;
      end;
run;
```

The FULLSTIMER log metrics demonstrate that the single-threaded solution is both faster and more efficient than DS2:




```
NOTE: The data set WORK.SINGLE has 80000 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      user cpu time       0.00 seconds
      system cpu time     0.00 seconds
      memory              384.71k
      OS Memory           22508.00k
```

Single-threading won this time—but will it win *every* time? And what happens to the performance (and efficiency) of each method as the number of iterations and threads are independently varied? Performance testing should always incorporate repeated measures analysis that demonstrates consistent performance across consistent testing conditions (or parameterized data injects), and scalability testing that demonstrates how parameters affect performance. In this case, it's important to understand how increased load (i.e., number of observations) and thread count influences the DS2 procedure, respective of functionally equivalent DATA step performance. This is the focus of the next section.

## LOAD PERFORMANCE TESTING FOR SCALABILITY

Load testing and optimization are critical to any performance analysis, so both the number of observations and number of threads should be incrementally increased and evaluated to determinate how these characteristics affect DS2 performance as well as DS2 performance respective of the performance of the single-threaded DATA step alternative.

For subsequent demonstrations, the PINCHLOG macro (included in Appendix A) is used to collect FULLSTIMER performance metrics and to aggregate these (over iterated processes) into a metrics data set. The PINCHLOG macro is featured in the author's text: *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*.[vii]

The PINCHLOG macro should be saved to the folder in which all other sample code is being executed; this location is initialized to the &LOC global macro variable.

The TESTTHREADS macro incrementally increases the number of observations and number of available threads while comparing this DS2 procedure to a functionally equivalent DATA step:

```
%let loc=D:\sas\ds2\;       * CHANGE TO USER-SPECIFIED LOCATION!!! *;
%include "&loc.pinchlog.sas";
%let log=&loc.log.txt;

%macro testthreads(threadno= /* max (even) number of threads */,
```

```
   obslow= /* base number of observations */,
   obshigh= /* high number of observations */,
   obsiter= /* number by which to increase obs */);
%local i j cpucount;
%let cpucount=%sysfunc(getoption(cpucount));
%do i=&obslow %to &obshigh %by &obsiter;
   %do j=2 %to &threadno %by 2;
      * single-threaded DATA step;
      %let syscc=0;
      proc printto log="&log" new;
      run;
      data single (keep=i);
         length i 8;
         do h=1 to &j;
            do i=1 to (&i/&j);
               output;
               end;
            end;
      run;
      proc printto;
      run;
      %pinchlog(logfile=&log, dsnmetrics=metrics,
      othervars=(var=method, val=DATA Step, len=$10, form=$10., lab=Method /
         var=obs, val=&i, len=8, form=15., lab=Observations /
         var=threads, val=1, len=8, form=8., lab=Threads /
         var=cpucount, val=&cpucount, len=8, form=8., lab=CPUCOUNT /
         var=err, val=&syscc, len=8, form=8., lab=Error Code));
      * multithreaded DS2 method;
       %let syscc=0;
      proc printto log="&log" new;
      run;
      proc ds2;
         thread t / overwrite=y;
            dcl int x;
            method init();
               dcl int i;
               do i = 1 to (&i/&j);
                  x = i;
                  output;
                  end;
               end;
            endthread;
      data multi / overwrite=y;
         dcl thread t t_instance;
         method run();
            set from t_instance threads=&j;
            output;
            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=obs, val=&i, len=8, form=15., lab=Observations /
         var=threads, val=&j, len=8, form=8., lab=Threads /
         var=cpucount, val=&cpucount, len=8, form=8., lab=CPUCOUNT /
         var=err, val=&syscc, len=8, form=8., lab=Error Code));
      %end;
   %end;
%mend;
```

To test the influence of thread count on DS2 performance, TESTTHREADS can be run on a 100 million-observation data set in which the number of threads is varied from two to 16:

```
%testthreads(threadno=16, obslow=100000000, obshigh=100000000, obsiter=100000000);

proc sgplot data=metrics;
   series y=realtime x=threads / dataskin=pressed group=method markers;
   xaxis display=(noticks) label='Number of Threads' values=(0 to 16 by 2);
   yaxis label='Realtime in Seconds';
run;
```

The Metrics data set is represented by the SGPLOT procedure, and the resultant output (Figure 1) demonstrates that the DS2 procedure takes significantly longer to run than the functionally equivalent DATA step.

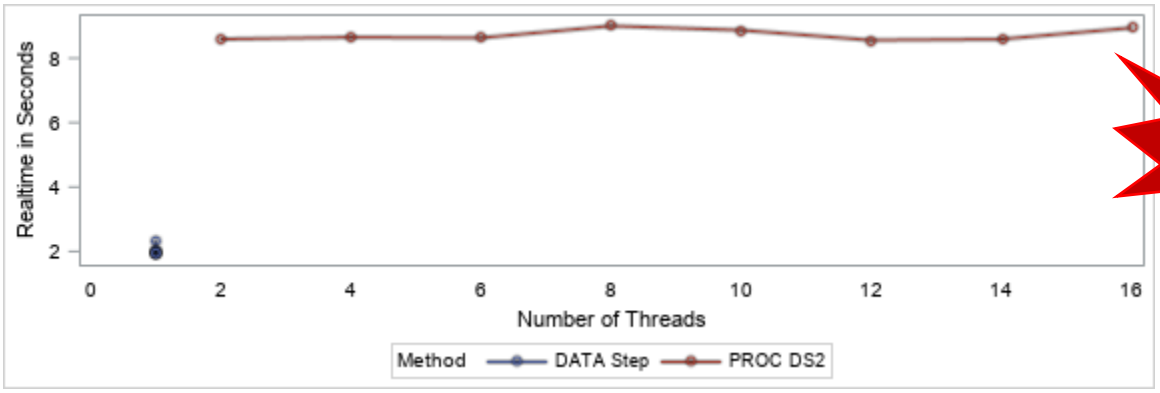



**Figure 1. Slower Performance of Ubiquitous DS2 Multithreading Example**

In this first example, only the number of threads was varied, although at this observation count, this factor seems not to have contributed noticeably to runtime. Thus, in the second example, the number of threads is iterated from two to four while the number of observations is iterated from 10 million to 100 million:

```
%testthreads(threadno=4, obslow=10000000, obshigh=100000000, obsiter=10000000);
```

The following DATA step transforms the Metrics data set to produce the Newmethod variable, which will differentiate the number of threads:

```
data metrics2;
   set metrics;
   length newmethod $40;
   newmethod=catx(' - ',method,put(threads,15.));
run;

ods graphics on / width=900 height=300;
proc sgplot data=metrics2;
   series y=realtime x=obs / dataskin=pressed group=newmethod markers;
   xaxis display=(noticks) label='Number of Observations';
   yaxis label='Realtime in Seconds';
run;
```

The output is demonstrated in Figure 2.

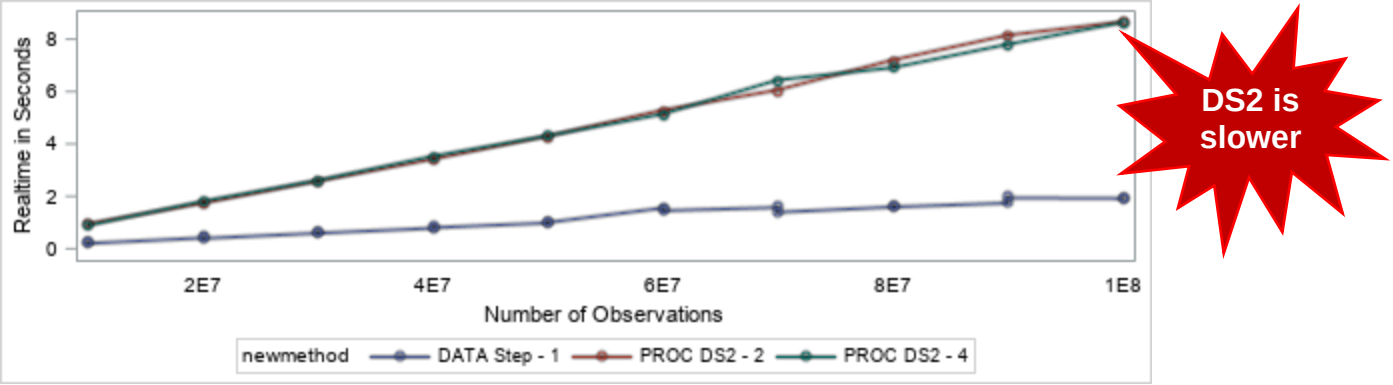



**Figure 2. Slower Performance of Ubiquitous DS2 Multithreading Example (10 to 100 Million Obs)**

Note that as the number of observations increases from 10 million to 100 million, the relative performance of the DS2 procedure (as compared with a functionally equivalent DATA step) continues to worsen. For example, at 10 million observations, the DATA step takes 0.25 seconds (Real Time) whereas the DS2 procedure takes 0.95 seconds, and; at 100 million observations, the DATA step takes 1.91 seconds whereas the DS2 procedure takes a whopping 8.64 seconds to complete!

SAS Institute does provide other examples of DS2 multithreading; however, many examples suffer from the similar lack of complexity, in which some output is generated to the log and the DS2 overhead by far exceeds that of a functionally equivalent single-threaded DATA step. These patterns are observed in the *Example A Simple Thread Program* and *SETPARMS Method* DS2 product documentation.[viii, ix]

Until realistic examples depict DS2 multithreading that performs better than functionally equivalent DATA steps, and until performance testing can demonstrate that clear advantage, SAS practitioners may be reluctant to explore DS2 multithreading due to ubiquitous DS2 examples that run *slower* than their single-threaded DATA step counterparts.

## CPU-BOUND PROCESSES

Much of the problem with the previous ubiquitous example is that the single-threaded process that is being multithreaded is not *CPU-bound*. SAS Institute defines *CPU-bound* as applications that "receive data faster than they can perform the necessary processing on that data."[x] 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.[xi]

The operative phrase in interpreting the foregoing quote is "bad task performance," an enigma that is 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.[xii]

But the difference between "moving well" and "bad performance," unless defined and quantified, really comes down to perception. What is more important than this perception, however, 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 and load testing and is featured in two of the author's white papers.[xiii, xiv] 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 is possible that this DATA step could be CPU-bound. And, whether you perceive the 16-second performance to be adequate or poor, the close proximity of the real time and CPU time values makes this DATA step is an excellent candidate for multithreading with DS2.

The DATA step can be converted to DS2:

```
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));
                  end;
               end;
            output;
            end;
         end;
      endthread;
run;
data ds2.somedatads2 / overwrite=yes;
   dcl thread thr t_instance;
   method run();
      set from t_instance threads=8;
```

```
      end;
   enddata;
run;
quit;
```

When the DS2 procedure 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 log 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
```

The output also demonstrates that the DS2 procedure utilized considerably more resources as compared to the functionally equivalent DATA step. For example, the DS2 procedure required nine times more CPU cycles to complete than the DATA step. In a world of limitless processing power, this is not an issue, and a solution that is faster yet less efficient is often preferred; however, the DS2 example was both slower and less efficient than the DATA step—no bueno.

It is possible that additional observations might push the DS2 procedure into the lead, so the number of observations is increased from 100,000 to one million. Note that because the DS2 procedure is running eight threads, each thread produces only 125,000 observations, which has been hardcoded:

```
%makedata(dsn=ds2.somedata, obs=1000000, charvar=100, charlen=20);

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 125000;
            do j=1 to 100;
               charvar[j]='';
               do k=1 to 20;
                  charvar[j]=cats(charvar[j],byte(int(rand('uniform')*10)+65));
                  end;
               end;
            output;
            end;
         end;
      endthread;
run;
data ds2.somedatads2 / overwrite=yes;
   dcl thread thr t_instance;
   method run();
      set from t_instance threads=8;
      end;
   enddata;
run;
quit;
```

The respective logs indicate that the DATA step completed in 2:27 minutes whereas the DS2 procedure completed in 3:03 minutes. Unfortunately, at 100,000 observations, the DS2 procedure was only 19.4 percent slower, whereas at one million observations, the DS2 procedure is 25.5 percent slower—so things are not looking good if the number of observations continues to scale upward. The logs follow:

```
NOTE: The data set DS2.SOMEDATA has 1000000 observations and 100 variables.
NOTE: DATA statement used (Total process time):
      real time           2:27.06
      user cpu time       2:26.28
```

```
      system cpu time      0.50 seconds
      memory               720.68k
      OS Memory            31284.00k

NOTE: Execution succeeded. 1000000 rows affected.
1611   quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time            3:03.66
      user cpu time        23:35.04
      system cpu time      0.93 seconds
      memory               8851.90k
      OS Memory            37552.00k
```


Thus, a process that looked promising (for being CPU-bound) appears not to have been, or at least was not made faster by implementing it with DS2. However, one technique used to simulate computational complexity has been nested loops, which can be run iteratively inside each thread that is concurrently running. One such published example is demonstrated in the following section.

## A SECOND EXAMPLE FROM LITERATURE

Kirk Paul Lafler offers this example, which he credits to Mark Jordan:[xv]

```
data cars(drop=i) ;
   set sashelp.cars(keep=make model type) ;
   do i = 1 to 1000 ;
      output;
      end ;
run ;

proc ds2 ;
   thread work.myThread(double flag)/overwrite=y ;
      dcl int ThreadNo ;
      dcl int Count ;
      method run() ;
         set work.cars ;
         ThreadNo=_threadid_ ;
         count+1 ;
         end ;
      method term() ;
         ThreadNo=_threadid_ ;
         put ThreadNo= Count= ;
         end ;
      endthread ;
run ;
quit ;
proc ds2 ;
data cars3 / overwrite=yes ;
   dcl double i j k ;
   dcl thread work.myThread t ;
   method init() ;
      t.setparms(1) ;
      end ;
   method run() ;
      set from t threads=4 ;
      do i = 1 to 100 ;
         do j = 1 to 50 ;
            k = (j*i)**3 ;
            end ;
         end ;
      end ;
   enddata ;
run ;
```

```
quit ;
```

Note the nested DO loops, which point to possible heavy use of computational resources. In other words, the amount of effort required to process each observation is greater, or the CPU-cycle-to-observation ratio is higher than the examples demonstrated thus far.

The following functionally equivalent, single-threaded DATA step is not demonstrated in Kirk or Mark's papers, but adds to this collaborative effort:

```
data carssingle;
   set cars;
   length i j k threadno count 8;
   threadno=1;
   count=_n_;
   do i = 1 to 100 ;
      do j = 1 to 50 ;
         k = (j*i)**3 ;
         end;
      end;
run;
```

The following combined log indicates that when run on a 4-CPU desktop, DS2 again is slower than its functionally equivalent DATA step counterpart, at 27 seconds and 17 seconds (real time), respectively:

```
NOTE: Created thread mythread in data set work.mythread.
NOTE: Execution succeeded. No rows affected.
26   quit ;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           9.83 seconds
      user cpu time       0.18 seconds
      system cpu time     0.84 seconds
      memory              9071.68k
      OS Memory           19692.00k

ThreadNo=3 Count=108544
ThreadNo=2 Count=106464
ThreadNo=1 Count=106496
ThreadNo=4 Count=106496
NOTE: BASE driver, creation of a INTEGER column has been requested, but is not
supported by the BASE
      driver. A DOUBLE PRECISION column has been created instead.
NOTE: Execution succeeded. 428000 rows affected.
45   quit ;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           17.17 seconds
      user cpu time       10.71 seconds
      system cpu time     0.60 seconds
      memory              5973.50k
      OS Memory           22252.00k

NOTE: There were 428000 observations read from the data set WORK.CARS.
NOTE: The data set WORK.CARSSINGLE has 428000 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           17.25 seconds
      user cpu time       16.87 seconds
      system cpu time     0.25 seconds
      memory              632.00k
      OS Memory           18668.00k
```


But finally, at long last, when run on an 8-CPU machine, the DS2 procedure beats the DATA step by three seconds, as demonstrated in the following log:

```
NOTE: Created thread mythread in data set work.mythread.
NOTE: Execution succeeded. No rows affected.
1634   quit ;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.04 seconds
      user cpu time       0.03 seconds
      system cpu time     0.01 seconds
      memory              3736.81k
      OS Memory           31728.00k

ThreadNo=3 Count=106496
ThreadNo=2 Count=106496
ThreadNo=4 Count=108512
ThreadNo=1 Count=106496
NOTE: BASE driver, creation of a INTEGER column has been requested, but is not
supported by the
      BASE driver. A DOUBLE PRECISION column has been created instead.
NOTE: Execution succeeded. 428000 rows affected.
1652   quit ;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           8.14 seconds
      user cpu time       8.10 seconds
      system cpu time     0.11 seconds
      memory              5788.51k
      OS Memory           33776.00k

NOTE: There were 428000 observations read from the data set WORK.CARS.
NOTE: The data set WORK.CARSSINGLE has 428000 observations and 8 variables.
NOTE: DATA statement used (Total process time):
      real time           11.54 seconds
      user cpu time       11.46 seconds
      system cpu time     0.06 seconds
      memory              619.71k
      OS Memory           31216.00k
```


Thus, the nested looping inside the RUN method is computationally complex enough to demonstrate a DS2 advantage:

```
do i = 1 to 100 ;
   do j = 1 to 50 ;
      k = (j*i)**3 ;
      end ;
   end ;
```

Additional analysis would be required to determine the optimal number of threads to run concurrently, as well as to demonstrate similar computationally complex scenarios for which DS2 multithreading yields better performance than functionally equivalent DATA steps. These results would also need to be replicated on SAS systems having greater processing power (i.e., CPUCOUNT) to determine what effects this has.

## CONCLUSION

DS2 multithreading performs faster than a functionally equivalent, single-threaded DATA step in certain, very specific computationally intensive tasks, especially those in which the computation-to-observation ratio or computational complexity 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, the DS2 procedure performs slower than an equivalent DATA step while consuming more system resources. In the end, SAS practitioners will need to decide whether to invest the resources into learning and implementing the DS2 procedure to overcome specific, CPU-bound scenarios that occur within legacy DATA steps; with that caveat, the multithreaded future looks promising.

## 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] Troy Martin Hughes. From FREQing Slow to FREQing Fast: Facilitating a Four-Times-Faster FREQ with Divide-and-Conquer Parallel Processing. *Western Users of SAS Software (WUSS).* 2018. Retrieved from http://wuss18.org/wp-content/uploads/accepted2018/138_Final_Paper_PDF.pdf.

[iii] Viraj R. Kumbhakarna. PROC DS2: What's in it for you? *Western User's of SAS Software (WUSS)*. 2017. Retrieved from https://support.sas.com/resources/papers/proceedings17/0916-2017.pdf.

[iv] Andra Northup. A Brief Introduction to Some Object-Oriented Programming (OOP) Concepts for SAS Programmers. *Western Users of SAS Software (WUSS).* 2015. Retrieved from https://www.lexjansen.com/wuss/2015/97_Final_Paper_PDF.pdf.

[v] Peter Eberhardt. I Object: SAS® Does Objects with DS2. *PharmaSUG*. 2016. Retrieved from https://www.lexjansen.com/pharmasug/2016/BB/PharmaSUG-2016-BB09.pdf.

[vi] Peter Eberhardt. DS2 with Both Hands on the Wheel. *SAS Global Forum (SGF).* 2015. Retrieved from https://support.sas.com/resources/papers/proceedings15/2523-2015.pdf.

[vii] 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.

[viii] SAS® 9.4 DS2 Language Reference, Sixth Edition. SAS Institute. *Example: A Simple Threaded Program.* Retrieved from https://documentation.sas.com/?docsetId=ds2ref&docsetTarget=n01kc8vxfcgilxn16pd81wegr2ld.htm&docsetVersion=9.4&locale=en.

[ix] SAS® 9.4 DS2 Language Reference, Sixth Edition. SAS Institute. SETPARMS Method. Retrieved from https://documentation.sas.com/?docsetId=ds2ref&docsetTarget=n0xgqmk8naldnfn1m1tpbv6k2884.htm&docsetVersion=9.4&locale=en.

[x] 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.

[xi] Resources / Focus Areas: Scalability and Performance. SAS Institute. *FULLSTIMER SAS Option*. Retrieved from https://support.sas.com/rnd/scalability/tools/fullstim/index.html.

[xii] Metamucil Get Regular and Stay Regular. 1988. Retrieved from https://www.youtube.com/watch?v=zZnZeCbCUDg.

[xiii] 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.

[xiv] 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.

[xv] Kirk Paul Lafler. Modernizing Legacy SAS® Applications and Program Code. 2017. Retrieved from https://www.lexjansen.com/wuss/2017/122_Final_Paper_PDF.pdf.

## CONTACT INFORMATION

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

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

## APPENDIX A. THE PINCHLOG MACRO

```
%macro pinchlog(logfile= /* path, file name, and extension */,
    dsnmetrics= /* optional metrics data set in LIB.DSN or DSN format */,
    othervars= /* optional tokenized list of user-defined variables */);
* all times converted from HH:MM:SS.ss to SSSS.xx format;
%let syscc=0;
%global pinchlogRC pinchlogchildRC realtime usercputime systemcputime
    memory osmemory stepcount switchcount pagefaults pagereclaims
    volcontextswitches involcontextswitches blockinops blockoutops;
%let pinchlogRC=99;
%let pinchlogchildRC=0;
%let realtime=.;
%let usercputime=.;
%let systemcputime=.;
%let memory=.;
%let osmemory=.;
%let stepcount=.;
%let switchcount=.;
%let pagefaults=.;
%let pagereclaims=.;
%let volcontextswitches=.;
%let involcontextswitches=.;
%let blockinops=.;
%let blockoutops=.;
data _null_;
    length tab $500;
    infile "&logfile" truncover;
    input tab $500.;
    if strip(tab)=:'WARNING' or strip(tab)=:'ERROR' then do;
           call symput('pinchlogchildRC','4');
           return;
           end;
    if _n_>=7 then do; * skips the metrics produced by PRINTTO itself;
           if lowcase(substr(tab,1,9))='real time' then do;
                  if count(scan(substr(tab,10),1,' '),':')=0 then
                         call symput('realtime',scan(substr(tab,10),1,' '));
                  else if count(scan(substr(tab,10),1,' '),':')=1 then
                         call symput('realtime',
                         put(((input(strip(scan(substr(tab,10),
                         1,':')),8.0) * 60) +
                         input(strip(scan(substr(tab,10),2,':')),8.2)),8.2));
                  else if count(scan(substr(tab,10),1,' '),':')=2 then
                         call symput('realtime',
                         put(((input(strip(scan(substr(tab,10),
                         1,':')),8.0) * 3600) +
                         (input(strip(scan(substr(tab,10),2,':')),8.0) * 60) +
                         input(strip(scan(substr(tab,10),3,':')),8.2)),8.2));
                  end;
           else if lowcase(substr(tab,1,13))='user cpu time' then do;
                  if count(scan(substr(tab,14),1,' '),':')=0 then
                         call symput('usercputime',scan(substr(tab,14),1,' '));
                  else if count(scan(substr(tab,14),1,' '),':')=1 then
                         call symput('usercputime',
                         put(((input(strip(scan(substr(tab,14),
                         1,':')),8.0) * 60) +
                         input(strip(scan(substr(tab,14),2,':')),8.2)),8.2));
                  else if count(scan(substr(tab,14),1,' '),':')=2 then
                         call symput('usercputime',
                         put(((input(strip(scan(substr(tab,14),
                         1,':')),8.0) * 3600) +
                         (input(strip(scan(substr(tab,14),2,':')),8.0) * 60) +
                         input(strip(scan(substr(tab,14),3,':')),8.2)),8.2));
```

```
                end;
            else if lowcase(substr(tab,1,15))='system cpu time' then do;
                if count(scan(substr(tab,16),1,' '),':')=0 then
                    call symput('systemcputime',scan(substr(tab,16),1,' '));
                else if count(scan(substr(tab,16),1,' '),':')=1 then
                    call symput('systemcputime',
                    put(((input(strip(scan(substr(tab,16),
                    1,':')),8.0) * 60) +
                    input(strip(scan(substr(tab,16),2,':')),8.2)),8.2));
                else if count(scan(substr(tab,16),1,' '),':')=2 then
                    call symput('systemcputime',
                    put(((input(strip(scan(substr(tab,16),
                    1,':')),8.0) * 3600) +
                    (input(strip(scan(substr(tab,16),2,':')),8.0) * 60) +
                    input(strip(scan(substr(tab,16),3,':')),8.2)),8.2));
                end;
            * convert KB to MB;
            else if lowcase(substr(tab,1,6))='memory' then
                call symput('memory',
               put(input(scan(substr(tab,7),1,' k'),8.3)/1024,8.3));
            else if lowcase(substr(tab,1,9))='os memory' then
                call symput('osmemory',
               put(input(scan(substr(tab,10),1,' k'),8.3)/1024,8.3));
            else if lowcase(substr(tab,1,10))='step count' then do;
                call symput('stepcount',scan(substr(tab,11),1,' '));
                call symput('switchcount',scan(substr(tab,11),4,' '));
                end;
            else if lowcase(substr(tab,1,11))='page faults' then
                call symput('pagefaults',scan(substr(tab,12),1,' '));
            else if lowcase(substr(tab,1,13))='page reclaims' then
                call symput('pagereclaims',scan(substr(tab,14),1,' '));
            else if lowcase(substr(tab,1,26))='voluntary context switches' then
                call symput('volcontextswitches',scan(substr(tab,27), 1,' '));
            else if lowcase(substr(tab,1,28))='involuntary context switches' then
                call symput('involcontextswitches',scan(substr(tab,29),1,' '));
            else if lowcase(substr(tab,1,24))='block input operations' then
                call symput('blockinops',scan(substr(tab,25),1,' '));
            else if lowcase(substr(tab,1,25))='block output operations' then
                call symput('blockoutops',scan(substr(tab,26),1,' '));
            end;
    run;
    * optionally initialize user-defined variables;
    * at least VAR, VAL, and FORM sub-parameters are required for each variable;
    %if %length(othervars)>0 %then %do;
        %local otherval otherlen otherform otherlab var val len form lab i j;
        %let otherval=;
        %let otherlen=;
        %let otherform=;
        %let otherlab=;
        %let othervars=%sysfunc(compress(%bquote(&othervars),%nrstr(%))%nrstr(%(),));
        %let i=1;
        %do %while(%length(%scan(%bquote(&othervars),&i,/))>1);
            %let var=;
            %let val=;
            %let len=;
            %let form=;
            %let lab=;
            %let varstring=%scan(%bquote(&othervars),&i,/);
            %let j=1;
            %do %while(%length(%scan(%bquote(&varstring),&j,%str(,)))>1);
                %let valstring=%scan(%bquote(&varstring),&j,%str(,));
                %if %lowcase(%scan(&valstring,1,=))=var
                    %then %let var=%scan(&valstring,2,=);
```

```
                %if %lowcase(%scan(&valstring,1,=))=val
                       %then %let val=%scan(&valstring,2,=);
                %if %lowcase(%scan(&valstring,1,=))=len
                       %then %let len=%scan(&valstring,2,=);
                %if %lowcase(%scan(&valstring,1,=))=form
                       %then %let form=%scan(&valstring,2,=);
                %if %lowcase(%scan(&valstring,1,=))=lab
                       %then %let lab=%scan(&valstring,2,=);
                %let j=%eval(&j+1);
                %end;
         %if %length(&var)>0 and %length(&len)>0 and %length(&val)>0 %then %do;
                %let otherlen=&otherlen &var &len;
                %if %substr(&len,1,1)=$ %then
                       %let otherval=&otherval &var="&val"%str(;);
                %else %let otherval=&otherval &var=&val%str(;);
                %if %length(&form)>0 %then %let otherform=&otherform &var &form;
                %if %length(&lab)>0 %then %let otherlab=&otherlab &var="&lab";
                %end;
         %let i=%eval(&i+1);
         %end;
    %end;
* optionally create/modify metrics data set;
%if %length(&dsnmetrics)>0 %then %do;
    %if %sysfunc(exist(&dsnmetrics))=0 %then %do;
         data &dsnmetrics;
                length realtime 8 usercputime 8 systemcputime 8 memory 8
                       osmemory 8
                       stepcount 8 switchcount 8 pagefaults 8 pagereclaims 8
                       volcontextswitches 8 involcontextswitches 8 blockinops 8;
                format realtime 8.2 usercputime 8.2 systemcputime 8.2 memory 8.3
                       osmemory 8.3 stepcount 8. switchcount 8. pagefaults 8.
                       pagereclaims 8. volcontextswitches 8.
                       involcontextswitches 8.
                       blockinops 8.;
                label realtime='Real Time' usercputime='User CPU Time'
                       systemcputime='System CPU Time' memory='Memory in MB'
                       osmemory='OS Memory in MB' stepcount='Step Count'
                       switchcount='Switch Count' pagefaults='Page Faults'
                       pagereclaims='Page Reclaims' volcontextswitches=
                       'Voluntary Context Switches' involcontextswitches=
                       'Involuntary Context Switches'
                       blockinops='Block Input Operations'
                       blockoutops='Block Output Operations';
                %if %length(&otherval)>0 %then %do;
                       length &otherlen;;
                       %if %length(&otherform)>0 %then %do;
                              format &otherform;;
                              %end;
                       %if %length(&otherlab)>0 %then %do;
                              label &otherlab;;
                              %end;
                       %end;
                if not missing(realtime);
         run;
         %end;
    data pinchtemp;
         if 0 then set &dsnmetrics;
         realtime=&realtime;
         usercputime=&usercputime;
         systemcputime=&systemcputime;
         memory=&memory;
         osmemory=&osmemory;
         stepcount=&stepcount;
```

```
            switchcount=&switchcount;
            pagefaults=&pagefaults;
            pagereclaims=&pagereclaims;
            volcontextswitches=&volcontextswitches;
            involcontextswitches=&involcontextswitches;
            blockinops=&blockinops;
            %if %length(&otherval)>0 %then %do;
                  &otherval;
                  %end;
            output;
      run;
      proc append base=&dsnmetrics data=pinchtemp;
      run;
      %end;
%let pinchlogRC=&syscc;
%mend;
```