Paper CT03

This Data Step Has Been Running for Ages – Can You 'Progress' It?

Jean-Michel Bodart, Business & Decision Life Sciences, Brussels, Belgium

ABSTRACT

The SAS® option `FULLSTIMER` and the notes it generates in the log are useful to identify Data or Proc Steps that take a long time to execute – after the fact. When running programs interactively, waiting for half a minute or more for a step (or for multiple iterations over a sequence of steps in a loop) to complete while anxiously watching the log window becomes soon frustrating and irritating when we have no clue how much longer it will take to complete. And so much more when a previous test execution on a subset of the data took mere seconds! Should we wait patiently for a few more dozens of seconds, go and get a cup of coffee, do something else while the process is running, or cancel the process and start reviewing the code, check for infinite loops or other optimizable sections? To answer that nagging question, we need to get real-time feed-back about that process. This paper will look at some techniques that can be useful to follow the progress of a long Data Step, Proc SQL Step, or loop with numerous iterations.

INTRODUCTION

As a SAS programmer, I like developing and running my programs interactively, because this allows me to submit one step at a time, to check the results by looking at the resulting dataset in a ViewTable window, to update my code if necessary and rerun that specific step until I'm confident that every issue has been fixed, before continuing with the subsequent steps.

However, when large sets of data need to be processed, this can take much time, in which case developing and testing on a small subset makes sense, as much faster, before re-running on the full data set.

Complications may often arise at this latter stage: special cases and exceptions are encountered, the limits of the memory allowed for sorting are reached and the system starts swapping memory to disk, the log window is full and an action from the user is required, sometimes even the WORK library can run out of disk space.

All of this concurs to try a programmer's patience, especially when run times increase much more than proportionally to the amount of data. At that time, we are faced with questions such as:

- How much time will this take? Shall we just wait patiently? Or do something else while the process continues?
- What if the program had encountered an unanticipated case, and run into an infinite loop?
- Should we interrupt the process, review our code and optimize it before re-submitting it?

EVALUATING PROGRESS

Useful progress information could entail:

- A counter of work units (iterations, observations, by-groups, datasets, files, folders) that have been processed, together with the total number to process and the number remaining.
- A measure of the time already spent processing those work units, the average processing speed, the current speed, an estimation of the time to completion at the current speed
- A trend estimator indicating changes in processing speed
- If the progress information is not continuously updated, a time stamp allowing us to know how long ago the last piece of information was generated

FEED-BACK TECHNIQUES

The following methods can be used to communicate or obtain information about the progress of a SAS process:

SYSTEM PERFORMANCE MONITORS

Tools like the Windows® Task Manager can show us the computer resources (Processor time, Memory) used by a SAS process, the speed at which it is reading/ writing data, and whether it has started swapping memory to disk (a process aimed at sparing physical memory by transferring information to the hard drive, which severely affects the processing speed).

OUTPUT FILES CREATION AND GROWTH

By browsing or querying the file system it is possible to check that files are being created and are growing regularly (output datasets, ODS outputs, log redirected to an external file); it may be possible to actually view the contents of external files while they are being created (ODS HTML output, log file) but it is often difficult to locate/extract/summarize and interpret the relevant information in real time. Note that SAS interactive Display Manager[1] sessions let us directly access the SAS Work folder and its intermediate temporary datasets, while other SAS clients like Enterprise Guide and SAS Studio do not.

MESSAGES TO THE SAS LOG WINDOW

Work units processed, elapsed time, processing speed, remaining time and trends can be measured, calculated, and combined into status messages in Data step, macros, and FCMP functions. Such messages can be written to the LOG using `PUT` and `PUTLOG` statements in Data Steps and `%PUT` statements in macros; in procedures e.g. PROC SQL, they can be generated with `PUT` statements in user-written functions created with PROC FCMP [1].

Watching the log window for such progress messages, and for the occurrence of any green warning and red error messages may not be possible if the log window scrolls to fast or fills up too quickly; as a result, we may need to reduce the amount of information written to the log by setting the `NOMPRINT` or `NOSOURCE` system options [2], or even to redirect the log to an external file using PROC PRINTTO [3]. In the latter case nothing more would appear in real time in the log window, unless the log redirection is interrupted shortly to write a progress message, and restarted immediately afterwards. But even if they cannot be read in real time, messages to the log can be useful to retrospectively review the execution process.

MESSAGES TO OTHER SAS WINDOWS

When SAS executes in an interactive Display Manager session (this is the case when the data step expression `GETOPTION('DMS')` returns the value `'DMS'` or when the macro expression `%SYSFUNC(GETOPTION(DMS))` resolves to `DMS`), the `WINDOW` and `DISPLAY` statements [4] can be used in the Data Step, to display custom messages in separate windows. The `%WINDOW` and `%DISPLAY` statements can do the same in macros. The content of those windows is updated at each call of the `DISPLAY` or `%DISPLAY` statement. If set, the content of the data step variable `_MSG_` at the time of the call to the `DISPLAY` statement is displayed in the status line of the Display Manager window. The same applies to the content of macro-variable `&SYSMSG` at the time of the call to the `%DISPLAY` statement. It is possible to update the status line by those methods without displaying any visible window, by specifying a window position that would be out of the screen.

Since the contents of these windows do not accumulate but are replaced each time the window is refreshed, they can be updated almost continuously e.g. at each iteration of the data step.

Note: when these statements are executed in a SAS Batch process, a main window similar to an empty Display Manager session opens up, in which the 'DISPLAY WINDOW' is shown, although `GETOPTION('DMS')` still returns the value `'NODMS'`; it closes automatically at the end of the batch execution. When these statements are executed in a remote session (via `RSUBMIT`) or in a 'headless client' such as Enterprise Guide or SAS Studio, an error occurs:

```
ERROR: The MACRO windowing environment cannot be initialized due to a XU supervisor
failure.
```

In addition, the text of the last warning and the last error messages encountered, available through the automatic macro-variables `&syswarningtext` and `&syserrortext` [5] can also be included on separate lines in these windows. Recognizing that an error or serious warning has occurred will sometimes allow the user to interrupt early the execution of a program that is not going to complete successfully. These automatic macro-variables are read-only, and their value remains unchanged until the occurrence of the next warning or error respectively. But it is possible to reset them to 'empty' values at the start of a new program execution by submitting the code:

```
filename trash dummy;
proc printto log=trash; run;
%*- in the next 2 lines, 2 (or more) spaces after the colon (:) are necessary ! -*;
%put ER%bquote(ROR:  );
%put WAR%bquote(NING:  );
proc printto log=log; run;
filename trash;
```

Note: the temporary redirection of the log to a temporary, dummy file prevents the red text '`ERROR:`' and the green text '`WARNING:`' from appearing in the log window.

[1] Many users still refer to this windowing environment as "SAS Base"

MESSAGES TO 'SAS MESSAGE LOG' (TERMINAL) WINDOW

The FILE statement has an optional device type argument that can be set to `TERMINAL`, which according to SAS on-line Help "specifies the user's terminal". When writing text lines to that destination, they appear in an external window titled 'SAS Message Log' which pops to the front every time a new line is written to it, which can be annoying. It appears to display the last 20 lines of its contents, but by scrolling up the last 750 to 800 lines are accessible. Closing the window does not remove its previous content, which reappears when a new line is written to it. That window also appears when the SAS program is run as batch, and remains open when the main SAS process terminates. When this is done in a 'headless client' such as SAS Studio, no error message occurs but the 'SAS Message Log' does not show up either.

MESSAGES TO EXTERNAL FILES

In BATCH mode, or when the `ALTLOG` option is used, the full log will be written to an external file, while with PROC PRINTTO, a part of the log can be redirected.

The combined use of `FILE` and `PUT` statements [4] in the Data step allows writing messages to separate external files. Alternatively the functions `FOPEN()`, `FPUT()`, `FWRITE()` and `FCLOSE()` [6] can be used in the Data step, in PROC FCMP functions, and (through the `%SYSFUNC()` macro-function) in macros.

The tail of the growing contents of those external files can be displayed in real time in an external window. In the Unix world, the command: `tail -f external_file` [7] is well known for this purpose. On Windows, an equivalent result can be achieved using the powershell cmdlet (commandlet): `Get-Content 'external_file' -Wait` [8]. When SAS is executed locally (either as Batch or as an interactive Display Manager session), it can itself start those commands for parallel, asynchronous execution in an external window with the `SYSTASK` statement.

The messages will accumulate in the external window, the most recent ones being visible at the bottom.

Using more complex powershell commands, it is possible to display the last error message in red and the last warning in green. Those are retrieved respectively from the automatic macro-variables `&syswarningtext` and `&syserrortext` (see above) and written to the external file every time a message different from the previous one is identified. The status messages are written in white and overwrite each other, so that the previous error and warning remain visible without having to scroll up.

MESSAGES THROUGH NAMED PIPES

In theory, both SAS and powershell can read from and write to named pipes [9] [10], and this should avoid the latency due to buffering of output to an external file. However, although I have been able to make two SAS processes or two powershell processes communicate together this way, I have never managed to make SAS and powershell talk to each other through named pipes on a Windows platform. If you find out how to do it, please let me know. Things might be easier to implement on Unix.

The chosen technique(s) can generally be implemented as standard macros to be called either inside or outside a Data Step, and/or as PROC FCMP functions that can also be called from specific procedures such as PROC SQL and PROC REPORT.

SPEEDING UP PROGRAM EXECUTION

Other measures can help reduce the execution time [11] [12]:

- Clean up the WORK library at regular intervals to remove datasets that are no longer needed
- Shorten (character) variables that are unnecessarily long, keep needed variables and drop unneeded ones early … at the cost of spending time evaluating which are still needed, and having to rework your program and start again if it turns out some important data has been left out while doing so
- Compressing datasets may dramatically decrease the storage requirements, but increase the processor load
- Adjusting the memory size SAS is allowed to use (globally) and for sorting (they may already be at their maximum; changes may not be allowed by the organization, and may impact other processes and users sharing the same machine…)
- Running the more resource-demanding programs at times of decreased server load (night, weekend) is sometimes possible
- Splitting the data to process into smaller subsets, running the same process on each subset in a loop, and appending the outputs together (not always possible, choosing subset size is somewhat arbitrary)
- Reviewing and optimizing the SAS code, especially if the program needs to be run frequently.

BUILDING REAL-TIME FEEDBACK IN THE DATA STEP

MOTIVATION

I have recently been involved in importing into SAS datasets bundled Tables, Listings and Figures (TFLs), especially Summary Tables, delivered by a partner Clinical Research Organization (CRO) and exported as a single ascii text file by some non-SAS process. As the project advanced, we were more and more frequently receiving larger and larger files, with increasing numbers of tables, some unchanged, some new, and some updated since the previous delivery. The SAS processing of those files aimed at identifying and reporting the new and the updated tables in order to focus our review on those tables.

After reading each line of text as a single SAS observation, and identifying page breaks, one of the first tasks was to identify which lines were titles, headers, body text, blank space, and footnotes. For this, identifying the number and maximum size of sequences of multiple spaces appeared useful. For this, two FCMP functions were created, taking as input 2 character variables, a line of text and a Perl regular expression [6] that would specify the pattern of interest (in this case, sequences of 2 or more spaces):

```
proc fcmp outlib=work.functions.prx;
   *- Function to return the number of times a PRX pattern has matches in a string -*;
   function PRXNMATCH(pattern $, text $) ;
      prx=prxparse(pattern);
      start = 1;    stop = lengthn(text);
      pos = 0;      len = 0;
      matchnum = 0;
      if (stop > 0) then do until(pos = 0);
         ini=start;
         call prxnext(prx, start, stop, text, pos, len);
         if (len>0) then matchnum+1; *- count the number of matches -*;
      end;
      return(matchnum); *- return the number of matches -*;
   endsub;
   *-Function to return the max length of all matches a PRX pattern has in a string-*;
   function PRXMAXMATCHLEN(pattern $, text $) ;
      prx = prxparse(pattern);
      start = 1;    stop = lengthn(text);
      pos = 0;      len = 0;
      maxlen = 0;
      if (stop > 0) then do until(pos = 0);
         ini = start;
         call prxnext(prx, start, stop, text, pos, len);
         if (len>maxlen) then maxlen = len;  *- store the max length of matches -*;
      end;
      return(maxlen); *- return the maximum length of all matches -*;
   endsub;
run; quit;
option cmplib = work.functions;
```

In the above code, `CALL PRXNEXT` [6] searches a string 'text' for a pattern match (PERL regular Expression parsed as '`prx`') multiple times in succession, between the '`start`' and the '`stop`' positions (initially set to first and last positions in the string). When a match is found, the starting position '`pos`' and length '`len`' of the matched substring are updated, and the '`start`' for the next search is positioned at the first character after the end of the matched substring. When no match is found, the position '`pos`' and length '`len`' are set to 0, and the value of '`start`' is left unchanged. Both functions are similar except that one counts the number of matches, and the other retains their maximum length.

The use of FCMP functions is advocated [1] as independent, reusable and making a program easier to read, write, and modify.

The FCMP functions were called in a data step which initially was able to process about 150 pages in less than a second. For 1000 pages it took a bit less than a minute. But in the end the program had to swallow about 8,000 pages of text, and after 1 hour the data step was still running. What was happening? I started investigating by adding some code to the data step to track the time needed to process a given number of pages and calculate the average speed.

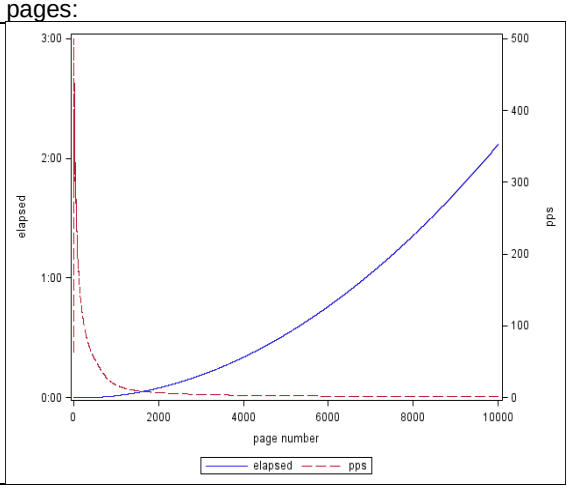
ILLUSTRATION AND MINIMUM REPRODUCIBLE EXAMPLE

I will demonstrate the results using a minimal example and a made-up dataset page10000, representing 10,000 landscape pages of 37 lines with a maximum length of 125 characters. The dataset creation and processing code and the macros mentioned hereafter are available on the PhUSE wiki **[13]**. To call the data step with different numbers of pages to process, we call it within a macro:

```
%macro process(pages = 1, report_when = last);
   data pages&pages(drop = starttime elapsed pps status)
        status&pages(keep = p elapsed pps status);
      retain starttime;
      length status $100;
      if _n_ = 1 then starttime = time();
      set page10000 (where = ( p <= &pages )) end = last;
      by p l;
      *- number of 'blank' fields of at least 2 consecutive spaces in current LINE -*;
      blankFieldsNum  =      PRXNMATCH('/(?<!\s)\s{2,}(?!\s)/', cats(line));
      *- max width of all 'blank' fields of at least 2 consecutive spaces -*;
      maxBlFieldWidth = PRXMAXMATCHLEN('/(?<!\s)\s{2,}(?!\s)/', cats(line));
      output pages&pages;
      if last.p then do; *- calculate status and save it at the end of every page -*;
         elapsed = time() - starttime;
         if elapsed > 0 then pps = p / elapsed;
         status = "Processed "||put(p, comma6.)||" pages in "||put(elapsed, time12.3)
                  || ", i.e. " || put(pps, 8.2) || " pps";
         output status&pages;
         if &report_when
            then put status;
      end;
      format starttime elapsed time12.3 pps 8.2;
   run;
%mend process;
option fullstimer;
%process(pages =   100);
%process(pages = 10000
        ,report_when = (p<=1000 and mod(p, 100)=0) or (p>1000 and mod(p, 500)=0) );
proc sgplot data = status10000;
   series x = p y = elapsed;
   series x = p y = pps / y2axis;
run;
```

For the last run with 10,000 pages, the macro parameter `report_when` is set to an expression that will be evaluated as true (and then result in writing a progress status line) every 100 pages until page 1000, then every 500 pages. The elapsed time and number of pages processed per second are also plotted. The results show that the processing speed decreased exponentially, and it took >2 hours to process 10,000 pages:

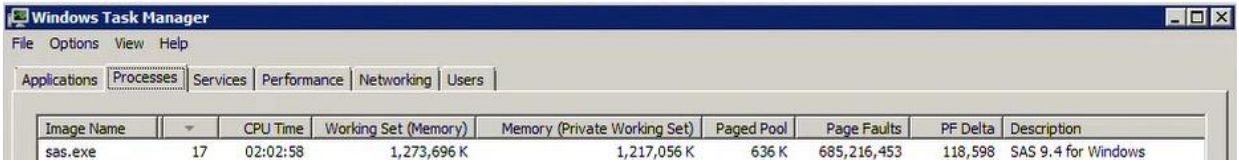
```
Processed    100 pages in  0:00:00.531, i.e.   188.32 pps
Processed    200 pages in  0:00:01.701, i.e.   117.58 pps
Processed    300 pages in  0:00:03.542, i.e.    84.70 pps
Processed    400 pages in  0:00:06.084, i.e.    65.75 pps
Processed    500 pages in  0:00:09.392, i.e.    53.24 pps
Processed    600 pages in  0:00:13.947, i.e.    43.02 pps
Processed    700 pages in  0:00:20.951, i.e.    33.41 pps
Processed    800 pages in  0:00:30.452, i.e.    26.27 pps
Processed    900 pages in  0:00:42.557, i.e.    21.15 pps
Processed  1,000 pages in  0:00:57.018, i.e.    17.54 pps
Processed  1,500 pages in  0:02:34.815, i.e.     9.69 pps
Processed  2,000 pages in  0:04:50.894, i.e.     6.88 pps
Processed  2,500 pages in  0:07:45.567, i.e.     5.37 pps
Processed  3,000 pages in  0:11:17.992, i.e.     4.42 pps
Processed  3,500 pages in  0:15:27.827, i.e.     3.77 pps
Processed  4,000 pages in  0:20:16.193, i.e.     3.29 pps
Processed  4,500 pages in  0:25:43.122, i.e.     2.92 pps
Processed  5,000 pages in  0:31:47.148, i.e.     2.62 pps
Processed  5,500 pages in  0:38:27.709, i.e.     2.38 pps
```



```
Processed  6,000 pages in  0:45:46.896, i.e.     2.18 pps
Processed  6,500 pages in  0:53:45.067, i.e.     2.02 pps
Processed  7,000 pages in  1:02:18.666, i.e.     1.87 pps
Processed  7,500 pages in  1:11:30.672, i.e.     1.75 pps
Processed  8,000 pages in  1:21:20.555, i.e.     1.64 pps
Processed  8,500 pages in  1:31:48.455, i.e.     1.54 pps
Processed  9,000 pages in  1:42:53.280, i.e.     1.46 pps
Processed  9,500 pages in  1:54:36.029, i.e.     1.38 pps
Processed 10,000 pages in  2:06:56.358, i.e.     1.31 pps
NOTE: There were 370000 observations read from the data set WORK.PAGE10000.
      WHERE p<=10000;
NOTE: The data set WORK.PAGES10000 has 370000 observations and 7 variables.
NOTE: The data set WORK.STATUS10000 has 10000 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           2:06:56.38
      user cpu time       1:48:29.03
      system cpu time     13:13.31
      memory              1101910.25k
      OS Memory           1214496.00k
      Timestamp           09/15/2019 02:17:07 AM
      Step Count                        110  Switch Count  0
```

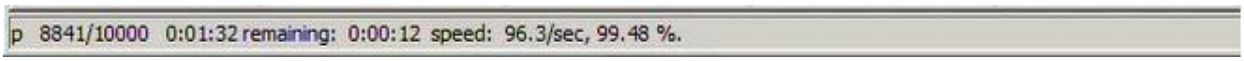
From the notes generated by the `FULLSTIMER` option we see that the system processor (CPU) time was only 13 minutes out of 2 hours total process time, and the memory used was about 1GB.

A screen capture of the Windows Task Manager taken a short time before the end of the process indicates a high number of 'Page Faults' and a high 'PF Delta', which show that the process was actively swapping memory to disk at that time, and must have been doing so for quite some time.



SOLUTION

Eventually we created a macro called `%progress` [13] that can be called in any data step. It would count the number of 'units' processed (observations or by-groups; here, pages), calculate the processing speed over consecutive time intervals (10 seconds by default), estimate the time to completion at current speed (when the total number of 'units' to process is specified), calculate the relative speed in % of the maximum speed previously observed. It writes a status message to the log after each interval, reports the number of work units completed as a macro-variable and optionally stops the data step when the relative speed drops below a specified percentage. In addition, if executed in an interactive Display Manager session, it shows the status in a data step display window and/or in the status line and updates it continuously.



In order to restart processing after stopping, the Data Step was inserted in a macro loop that would iterate until the known number of pages has been processed, each time restarting with the next page number. The output data would be appended to a separate dataset, which would be deleted at the initiation of the loop, and would contain all pages processed at the completion of the loop.

With this system in place, and without having identified the route cause, 10,000 pages could be processed in 5 minutes and 11.45 seconds through 15 iterations (stopping when the relative speed went under 30%), at an average speed of 32.1 pages per second. The log of a typical iteration would look like:

```
Starting iteration 14..
First observation read has p=8735 l=1
Elapsed time:  0:00:01 remaining:  0:00:10 p=8856  speed=119.3/sec, i.e. 100.0% of max.
Elapsed time:  0:00:11 remaining:  0:00:19 p=9250  speed=39.34/sec, i.e.  33.0% of max.
Elapsed time:  0:00:21 remaining:  0:00:36 p=9415  speed=16.45/sec, i.e.  13.8% of max.
Notice: Stopping at p=9415  remaining:  0:00:36 speed=16.45/sec, i.e.  13.8% of max.
NOTE: There were 25198 observations read from the data set WORK.PAGE10000.
      WHERE (p>8734 and p<=10000);
NOTE: The data set WORK._PAGES10000 has 25197 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           21.13 seconds
      user cpu time       16.13 seconds
      system cpu time     4.91 seconds
      memory              86047.46k
      OS Memory           128188.00k
      Timestamp           09/15/2019 03:54:09 PM
      Step Count                        308  Switch Count  2
```

ROOT CAUSE

The root cause was eventually identified. According to the on-line help for the `PRXPARSE()` function:

"If perl-regular-expression is a constant or if it uses the /o option, the Perl regular expression is compiled only once. Successive calls to `PRXPARSE()` do not cause a recompile, but returns the regular-expression-id for the regular expression that was already compiled. This behavior simplifies the code because you do not need to use an initialization block (`IF _N_ =1`) to initialize Perl regular expressions.

Note: If you have a Perl regular expression that is a constant, or if the regular expression uses the /o option, then calling `PRXFREE()` to free the memory allocation results in the need to recompile the regular expression the next time it is called by `PRXPARSE`.The compile-once behavior occurs when you use `PRXPARSE` in a DATA step. For all other uses, the perl-regular-expression is recompiled for each call to `PRXPARSE`."

As the `PRXPARSE` call is done in the FCMP function and not in the data step itself, it means the regular expression has to be (re-)compiled at each call to the FCMP function i.e. 2 x 37 x 10,000 = 740,000 times in total to process 10,000 pages.

Worse, we didn't have a `CALL PRXFREE()` call that "frees unneeded resources that were allocated for a Perl regular expression." In the Data Step, adding such a call after compiling a constant regular expression hardly makes a difference since the memory is allocated only once per regular expression, while here it was allocated 740,000 times!

OPTIMIZATION

The FCMP functions were ultimately optimized: In each function, a character array of 10 uncompiled Regular Expressions and a numeric array of 10 numeric identifiers of compiled Regular Expressions were created and made static (so that their values would be retained from one call to the next). When a function is called with an uncompiled character expression, it is compared to those previously used and stored in the array. If a match is found, the corresponding compiled Regular Expression identifier is used without recompilation. If no match is found, the new Regular expression is compiled and added to both arrays for further use. When the arrays are full, the last Regular Expression of the array is removed after freeing the corresponding memory with `PRXFREE`, and the new Regular Expression is compiled and stored in the freed array position. Thus, the amount of memory used up by compiled Regular Expressions is limited, and their storage allows their reuse without recompilation. The memory used however remains allocated until the SAS session is terminated.

With the optimized FCMP functions and the `%progress` macro called inside the data step, within a macro loop to restart after stopping when the speed dropped below 30%, 10,000 pages were processed in 1 minute and 45.107 seconds in a single iteration, at an average speed of 95.084 pages/sec. The relative speed always remained > 96%.

When the initial Data Step was run outside of any stop-restart loop, with the optimized FCMP functions and without calling the `%progress` macro, the process was even much faster (without the overhead of the status monitoring): 10.000 pages were processed in 2.200 seconds, at an average speed of 4,545 pages/sec. The speed remained about constant for the whole process.

CONCLUSIONS

Monitoring the progress of SAS execution can be done within a Data Step, Macro loop, and a few Procedures. Various methods can be used to provide feed-back in real time to the user, especially when using an interactive Display Manager session.

Monitoring the execution speed in a data step within a stop and restart loop can even lead to a large gain in performance without having to identify the root cause and to optimize the code, when the processing speed decreases with time. However, identifying the root cause and optimizing the code are still worth doing if the process has to be repeated time and again.
PROC FCMP functions, though a powerful tool, are not optimized for performance and ease of use as much as the data step equivalent code; they should therefore be used with caution, besides, they have numerous and little documented limitations, and some improvements by the SAS Institute would be much appreciated.

REFERENCES

[1] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / Base SAS Procedures Guide / Concepts: FCMP Procedure", https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=proc&docsetTarget=n0gievrvtf1f45n12in37e8dshsz.htm&locale=en.

[2] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / System Options", https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=lesysoptsref&docsetTarget=titlepage.htm&locale=en.

[3] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / Base SAS Procedures Guide / PRINTTO Procedure" https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=proc&docsetTarget=p08blwp6gwk25mn1hypjjrncx3bn.htm&locale=en.

[4] SAS Institude, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / DATA Step Statements", https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=lestmtsref&docsetTarget=titlepage.htm&locale=en.

[5] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / Macro Language Reference", https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=mcrolref&docsetTarget=titlepage.htm&locale=en.

[6] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / Functions and CALL Routines", https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=lefunctionsref&docsetTarget=titlepage.htm&locale=en.

[7] Wikipedia, "tail (Unix)", https://en.wikipedia.org/wiki/Tail_(Unix).

[8] Microsoft, "Docs / PowerShell / Scripting / Get-Content", https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-content?view=powershell-6.

[9] SAS Institute, "SAS® 9.4 and SAS® Viya® 3.4 Programming Documentation / / SAS Companion for Windows / Using Unnamed and Named Pipes under Windows" https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=hostwin&docsetTarget=n13sffgdom57irn13k3yx3e8rcha.htm&locale=en.

[10] G. '. Begerow, "Interprocess Communication in Powershell," 9 April 2012. https://gbegerow.wordpress.com/tag/powershell-named-pipes/.

[11] M. Foxwell, "Coding Efficiency", https://www.phusewiki.org/wiki/index.php?title=Coding_Efficiency.

[12] K. P. Lafler, "SAS Performance Tuning Techniques", https://www.phuse.eu/blog/sas%c2%ae-performance-tuning-techniques.

[13] J.-M. Bodart, "SAS macro %progress," 20 September 2019. https://www.phusewiki.org/wiki/index.php?title=SAS_macro_%25progress.

CONTACT INFORMATION

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

Jean-Michel Bodart
Business & Decision Life Sciences
Rue Saint-Lambert 141
1200 Brussels (Belgium)
Work Phone: +32 (0)2 774 11 00
Fax: +32 (0)2 774 11 99
Email: jean-michel.bodart@businessdecision.be
Web: www.businessdecision-lifesciences.com/