PhUSE EU Connect 2018

Paper SM06

Speed up evaluation by parallelization

Michael Weiss, Bayer AG, Berlin, Germany

ABSTRACT

This paper describes a SAS ® software based macro system called POStER - **P**arallel **O**ptimized **St**atistical **E**xecution **R**untime that allows easy and automatic parallel execution of multiple SAS based programs. POStER combines technics for tracing related file system changes and a scheduled queue execution approach to optimize the overall runtime speed of programs. SAS programs are by design mostly single threaded. Most output generation programs are independent of each other. This allows parallelization and leverage from the increasing amount of CPU cores. This paper shows the requirements and technics of the POStER system as it could be used in the process from SDTM(+) to ADaM and TLF generation. POStER actually decreased the overall runtime of study evaluation programs in an actual study by around 90%.

INTRODUCTION

"Can you deliver the results a day earlier?" or "Why does this step need so long?" are common management questions when talking about timelines for important studies. One of the most common answers is "The execution of the programs takes very long."

In the past years Bayer started multiple initiatives to reduce the number of days between date of "last patient last visit" and submission. In the same time it was noticed, that the amount of data and the amount of outputs per study increases. This results in longer runtime of programs to generate ADaM and TLF.

In some cases, the long runtime is a result of the programming and could be reduced by providing trainings for performance optimized programming to the users. If and how far these guidance is applied is highly depending on the programmers and often dropped because of high workload.

Another reason for the long runtime is the sequential execution. It is common to execute one step after the other, even if these steps are unrelated and could be performed in parallel. For example the execution of multiple TLF programs. At Bayer a TLF program only creates a single output and only depends on the availability of required ADaM data sets. All these TLF programs are independent of each other. This means that the order of execution does not matter and all programs could be executed in parallel.

In case 20 programs will require 30 minutes each, the full sequential execution requires 10 hours. If all 20 programs are executed at the same time, the overall runtime can (theoretically) be reduced to 30 minutes.

This is the point where POStER comes into place. This SAS macro system provides an easy to use API for the parallel execution of multiple SAS programs.

THE BASICS BEHIND POSTER

POStER is built almost entirely in SAS. Only for tracing of file system access, external applications can be used to increase correctness. Therefore POStER is mostly independent from operating system and can be used for example on Windows, Linux and HP-UX.

MULTIPLE PROGRAMS IN SERIAL ORDER (WITHOUT POSTER)

The analysts at Bayer are used to a so called "run-all principle". This means that all SAS programs of an analysis are referenced by an `%INCLUDE` statement from a single (run-all) SAS program. This run-all.sas would look like in Example 1.

```
%INCLUDE "&prgdir./pgm1.sas";
%INCLUDE "&prgdir./pgm2.sas";
%INCLUDE "&prgdir./pgm3.sas";
%INCLUDE "&prgdir./pgm4.sas";
```

Example 1: Sequential Execution of four SAS programs

As this example shows, all programs are executed after each other. This is called sequential execution. The sequential execution is easy to understand but results in poor performance on multi CPU systems.

MULTIPLE PROGRAMS IN PARALLEL (WITH POSTER)

In a POStER based version of this run-all, all jobs are initially added to a queue by invoking a macro called `%add_job`. The execution of all jobs is triggered at the end by a macro called `%run_jobs`. Applied to Example 1, this would look like in Example 2.

```
%add_job(programFile = &prgdir./pgm1.sas)
%add_job(programFile = &prgdir./pgm2.sas)
%add_job(programFile = &prgdir./pgm3.sas)
%add_job(programFile = &prgdir./pgm4.sas)
%run_jobs(count = 4)
```

Example 2: Parallel execution using POStER

As it looks quite similar to the `%INCLUDE` version, it's easy for a user to apply the change and even migrate existing studies to the new way very easily.

There are two main differences between both ways. First the execution of a single `%add_job` call will not execute the program, but add an execution request to the internal queue and only the `%run_jobs` macro call at the end will start the execution. Second the execution of a program through POStER is always performed in a new SAS session. Therefore macro variables or temporary data sets defined before the `%add_job` call are not available within the SAS program.

The first difference might require some small changes in the way of working and program development. The second difference provides special features:

- Programs can have individual initialization / SAS options
- Programs do not have unwanted relationships between each other
- In case of an ERROR in a single program, subsequent programs do not need to be re-run.

START SAS FROM SAS

To be able to run multiple SAS programs in parallel, POStER is starting additional SAS processes and executes each SAS program in its own SAS session.

To start an external process from SAS, there are several ways. Most, like the "`FILENAME PIPE`" or `CALL SYSTEM` commands, are synchronized with the SAS process. This means that the initial SAS process waits for the new process to finish before it continues. This is easy to handle in a lot of cases, but prevents parallel execution.

A special case is the `SYSTASK EXECUTE` command with the `NOWAIT` option. As the name suggests, this option tells SAS to continue execution immediately without waiting for the process to finish.

To finally wait for these processes to finish, SAS provides the separate `WAITFOR` command.

The Example 3 shows how Example 1 can be adapted to execute all four jobs in parallel.

```
SYSTASK COMMAND "sas -sysin &prgdir./pgm1.sas"
    NOWAIT TASKNAME=_n1;
SYSTASK COMMAND "sas -sysin &prgdir./pgm2.sas"
    NOWAIT TASKNAME=_n2;
SYSTASK COMMAND "sas -sysin &prgdir./pgm3.sas"
    NOWAIT TASKNAME=_n3;
SYSTASK COMMAND "sas -sysin &prgdir./pgm4.sas"
    NOWAIT TASKNAME=_n4;
WAITFOR _ALL_ _n1 _n2 _n3 _n4;
```

Example 3: Parallel execution using pure SAS

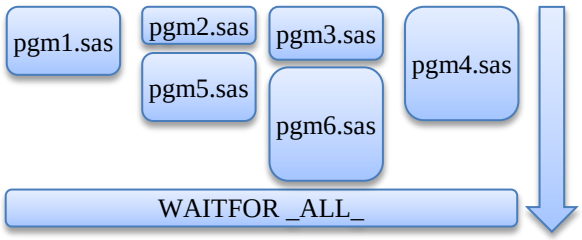
The first four commands start a new SAS process without waiting for the process to finish. The fifth line waits for all these processes to finish.

CAN NOT EXECUTE ALL AT A TIME

Up to now, the implementation would start all programs immediately in parallel and wait for them to finish. In case of four programs, this would be a valid option. In real life scenarios, a study might contain 100 or even more programs. Starting all of these at once would overload even big servers.

POStER contains a scheduling algorithm that ensures a configured maximum of simultaneous running processes. This algorithm will start as many processes as configured by the `count` parameter and only starts the next process after a previous process was finished.

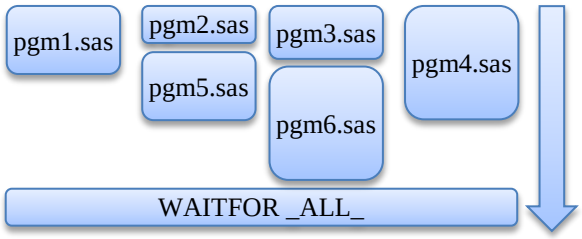
```
%add_job(programFile = &prgdir./pgm1.sas)
%add_job(programFile = &prgdir./pgm2.sas)
%add_job(programFile = &prgdir./pgm3.sas)
%add_job(programFile = &prgdir./pgm4.sas)
%add_job(programFile = &prgdir./pgm5.sas)
%add_job(programFile = &prgdir./pgm6.sas)
%run_jobs(count = 4)
```



Example 4: Parallel execution with scheduler in POStER

This is possible, as SAS provides an alternative parameter to WAITFOR. If `_ANY_` is specified instead of `_ALL_`, `WAITFOR` only waits for at least one job to be completed and continues afterwards. Applied to Example 3, when executing 6 SAS programs with a max of 4 at a time, this would look like Example 5.

```
SYSTASK COMMAND "sas -sysin &prgdir./pgm1.sas"
    NOWAIT TASKNAME=_n1;
SYSTASK COMMAND "sas -sysin &prgdir./pgm2.sas"
    NOWAIT TASKNAME=_n2;
SYSTASK COMMAND "sas -sysin &prgdir./pgm3.sas"
    NOWAIT TASKNAME=_n3;
SYSTASK COMMAND "sas -sysin &prgdir./pgm4.sas"
    NOWAIT TASKNAME=_n4;
WAITFOR _ANY_ &running_tasks;
SYSTASK COMMAND "sas -sysin &prgdir./pgm5.sas"
    NOWAIT TASKNAME=_n5;
WAITFOR _ANY_ &running_tasks;
SYSTASK COMMAND "sas -sysin &prgdir./pgm6.sas"
    NOWAIT TASKNAME=_n6;
WAITFOR _ALL_ &running_tasks;
```



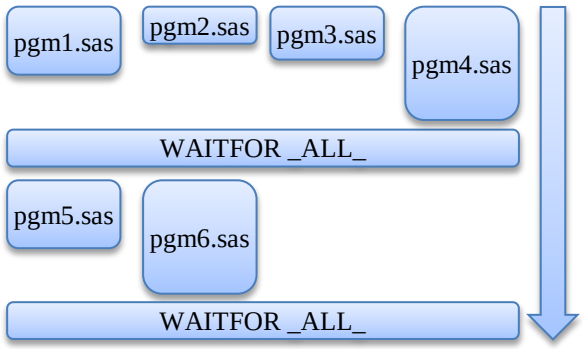
Example 5: Parallel execution with scheduler in pure SAS (pseudo SAS code)

PROGRAMS ARE NOT ALWAYS INDEPENDENT

Not all programs on the way from SDTM to TLF are completely independent. For example the TLF programs should not run before ADaM is generated and if ADSL would require values from other ADaM data sets, these should be available before ADSL is generated.

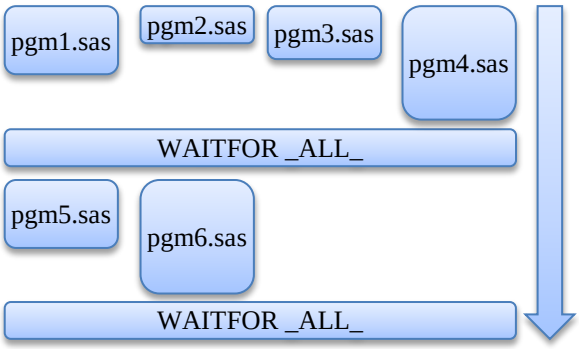
POStER provides a simple to use synchronization statement to handle this. If the macro `%sync_jobs` is called before an additional `%add_job` call, POStER ensures that all previous programs are finished, before starting the next one. This is implemented using the SAS command `WAITFOR _ALL_`. In POStER this will look like in Example 6.

```
%add_job(programFile = &prgdir./pgm1.sas)
%add_job(programFile = &prgdir./pgm2.sas)
%add_job(programFile = &prgdir./pgm3.sas)
%add_job(programFile = &prgdir./pgm4.sas)
%sync_jobs()
%add_job(programFile = &prgdir./pgm5.sas)
%add_job(programFile = &prgdir./pgm6.sas)
%run_jobs(count = 4)
```



Example 6: Parallel execution with synchronization in POStER

```
SYSTASK COMMAND "sas -sysin &prgdir./pgm1.sas"
    NOWAIT TASKNAME=_n1;
SYSTASK COMMAND "sas -sysin &prgdir./pgm2.sas"
    NOWAIT TASKNAME=_n2;
SYSTASK COMMAND "sas -sysin &prgdir./pgm3.sas"
    NOWAIT TASKNAME=_n3;
SYSTASK COMMAND "sas -sysin &prgdir./pgm4.sas"
    NOWAIT TASKNAME=_n4;
WAITFOR _ALL_ _n1 _n2 _n3 _n4;
SYSTASK COMMAND "sas -sysin &prgdir./pgm5.sas"
    NOWAIT TASKNAME=_n5;
SYSTASK COMMAND "sas -sysin &prgdir./pgm6.sas"
    NOWAIT TASKNAME=_n6;
WAITFOR _ALL_ _n5 _n6;
```



Example 7: Parallel execution with synchronization in pure SAS

This provides an easy way to implement relationships and between programs. An extended implementation could consider the exact dependencies per program. For example if pgm6 only depends on pgm2, it could be started as soon as pgm2 is finished, without waiting for pgm1, pgm3 and pgm4. This could increase the performance again, but would require the user to define all this in detail and therefore increase the complexity.

EXTENDED FEATURES

Beside the features, discussed in the first part, POStER comes with some more extended features. These are not directly related to parallel execution, but help in improving the performance additionally or provide additional information about the executed programs.

CARE ABOUT THE ORDER

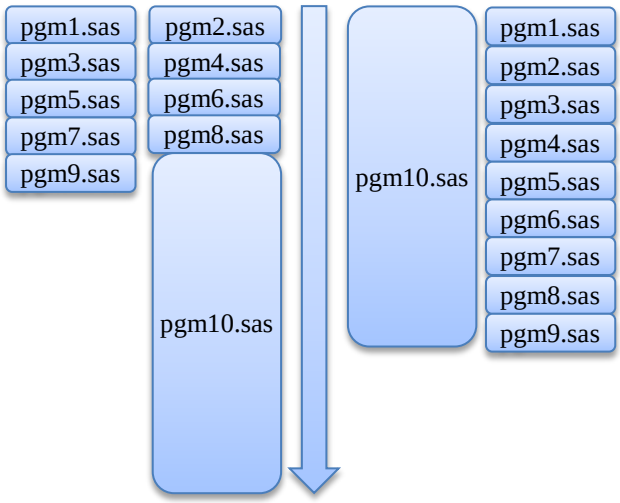
Assuming enough CPU and I/O power at the hardware level, this simple parallel execution can decrease the overall runtime by the number of parallel processes. For example with 2 parallel processes the runtime is only ½ the runtime of sequential execution, with 4 processes its only ¼ and so on.

In real life this might not be reached. One reason is the individual runtime of each program and the order of execution.

Example 8 shows a worst case scenario. There is one program with a much longer runtime than all other programs and this program is started last. In this example with 10 programs, 9 have 5 minutes runtime each and one has 45 minutes runtime. The sequential runtime would be 90 minutes (9 * 5 + 45).

The runtime with two programs parallel could differ between 45 and 65 minutes, based on the order of execution.

As seen in this example, the overall runtime can highly depend on the order of execution. POStER allows manual ordering as well as automatic re-ordering.



Example 8: Different runtime based on execution order

Initially the user can either put the `%add_job` calls in an optimized order or use the `weight` parameter to specify the expected runtime. POStER will than start the programs with the highest weight first.

Additionally, POStER generates metadata about an executed program. This metadata also contains the runtime of the previous execution and is used in the subsequent executions to automatically re-order the programs.

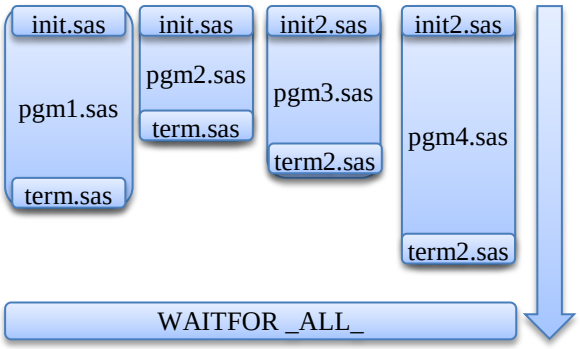
STANDARD AND INDIVIDUAL INITIALIZATION

As stated at the beginning, each program is executed in a separate SAS session (process). Beside the features, stated above, this results in the fact that each program has to handle the initialization on its own. For example if all programs require the same libraries (ADAM and SDTM) or global macro variables like &OUTDIR, these can't be defined once at the beginning of the run-all, but have to be defined in each program. Same is true for finalization routines like clean up.

To address this, POStER provides the two additional parameters `iniPROG` and `termPROG` in the `%add_job` macro. As the names suggest, these parameters can be used to define a program to be executed for initialization (`iniPROG`)

and a different one for termination (`termPROG`). Implementation wise, the values are mapped to the SAS options `INITSTMT` and `TERMSTMT` and passed to the SAS program start. Within POStER this will look like in Example 9.

```
%add_job(programFile=&prgdir./pgm1.sas
        ,iniPROG=&prgdir./init.sas
        ,termPROG=&prgdir./term.sas)
%add_job(programFile=&prgdir./pgm2.sas
        ,iniPROG=&prgdir./init.sas
        ,termPROG=&prgdir./term.sas)
%add_job(programFile=&prgdir./pgm3.sas
        ,iniPROG=&prgdir./init2.sas
        ,termPROG=&prgdir./term2.sas)
%add_job(programFile=&prgdir./pgm4.sas
        ,iniPROG=&prgdir./init2.sas
        ,termPROG=&prgdir./term2.sas)
%run_jobs(count = 4)
```

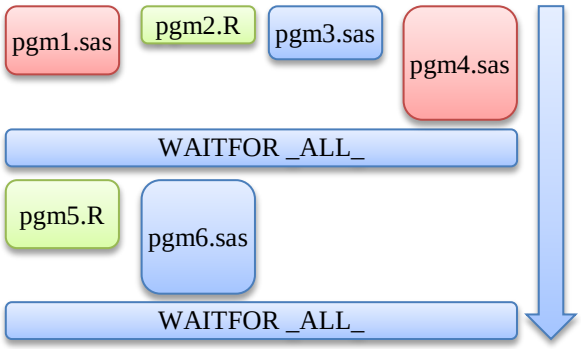


Example 9: iniPROG and termPROG in POStER

MIX IT ALL (NOT ONLY SAS)

As POStER executes each program in a separate SAS session, mixing SAS versions is as easy as executing a macro call. POStER provides a parameter called `interpreter` used to specify the SAS version to be executed.
If this parameter is not provided, the current SAS version is used for execution. If this parameter is filled, the specified SAS version will be used.
Additionally POStER is not limited to SAS programs. It is easily possible to start for example R programs the same way. If for example special statistical methods are to be performed in R or special graphics should be rendered this way, POStER can be called like in Example 10

```
%add_job(programFile = &prgdir./pgm1.sas
        ,interpreter = SAS9.2)
%add_job(programFile = &prgdir./pgm2.R
        ,interpreter = R3.1)
%add_job(programFile = &prgdir./pgm3.sas
        ,interpreter = SAS9.2)
%add_job(programFile = &prgdir./pgm4.sas
        ,interpreter = SAS9.4)
%sync_jobs()
%add_job(programFile = &prgdir./pgm5.R
        ,interpreter = R3.1)
%add_job(programFile = &prgdir./pgm6.sas
        ,interpreter = SAS9.4)
%run_jobs(count = 4)
```



Example 10: Interpreter parameter in POStER

As file level tracing is handled by an external OS based implementation, this works for R the same way as for SAS.

KNOW WHAT WAS DONE

POStER contains a pluggable file level tracing implementation. This allows automatically collecting file level metadata about the program that was executed. This metadata contains files that have been read, written or updated. Additionally files, that would have been read, if they would have been available, are traced.
Depending on the operating system, the implementation can either leverage from a SAS feature called `RTRACE` or from an OS dependent tracing implementation like `strace` or `ptrace`.
POStER groups all related files into four groups:

1. Files that already existed and have only been read (INPUT)
2. Files that have not existed before and have only been written (OUTPUT)
3. Files that have existed before and have been read and written (UPDATE)
4. Files that have not existed but would have been read if existing (EXPECTED)

In case the programs are designed, to have only one output (a single Data Set, table, listing or figure) this feature automatically collects dependencies about what input files are required for what output file. As POStER also handles

the SAS program file as INPUT file, this can be used to produce relationship graphs or as an extra step in validation (e.g. ensure no dummy files have been used in a final analysis).

```
### STARTED:2018-08-13T09:32:04
### FINISHED:2018-08-13T09:33:27
### JOB-STATUS:0
### PGM:/var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/t_adce.sas
### LOG:/var/swan/root/bhc/948862/16244/stat/csr/dev/logs/t_adce_fas.log
### RES:/var/swan/root/bhc/948862/16244/stat/csr/dev/results/t_adce.lst
### INIPROG:/var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/ini_14_shell_fas.sas
### TERMPROG:/var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/term_14.sas
### INTERPRETER:sas9.4
### JOB-OPTIONS:
### LOG-STATUS:2
R   /var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/t_adce.sas
 W  /var/swan/root/bhc/948862/16244/stat/csr/dev/logs/t_adce_fas.log
r   /var/swan/root/bhc/general/tools/eva/eva1/prod/macros/prepare_job.sas
R   /var/swan/root/bhc/general/tools/poster/poster1/dev/macros/prepare_job.sas
R   /var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/ini_14_shell_fas.sas
...
R   /var/swan/root/bhc/948862/16244/stat/csr/dev/pgms/term_14.sas
W  /var/swan/root/bhc/948862/16244/stat/csr/dev/results/t_adce_fas.rtf
```

Example 11: POStER tracing log file

Example 11 shows how such a tracing file looks. Basic metadata about the job are stored at the beginning. These include:

- Start and end of last execution. This allows to determine the runtime
- Job Status (return code)
- The program that was executed
- The log file and results file that that was generated
- The iniPROG and termPROG values
- The interpreter that was used
- Additional options used to execute the program
- A log status (if ERRORs, WARNINGs or other critical messages are found in log file)

Afterwards the file level tracing information follows. At the beginning of a line, the access method is stored. The following access is possible:

- R – INPUT (read) access
- W – OUTPUT (write) access
- r – EXPECTED Tried to read, but not available
- RW – UPDATE access, first read previously existing file and afterwards update

DON'T DO IT AGAIN

POStER contains an algorithm that can predict if a program has to be re-executed or not, based on the metadata collected in previous execution.

A program is not re-executed as long as none of the following points match:

- Did at least one of the execution parameters (`iniPROG`, `termPROG`, `options`, `interpreter`) change?
- Was at least one INPUT file changed after previous execution START?
- Was at least one OUTPUT file deleted after previous execution START?
- Does at least one EXPECTED file exist now?

As long as UPDATED files are not considered, this algorithm can be used to determine if a program has to be executed right before the program will be executed. Especially within ADaM generation, it is common practice, to generate data sets (e.g. ADSL) partially and UPDATE them with results from other ADaM data sets later on.

As soon as UPDATED files are considered, it gets a way more complicated.

Assuming that program 10 has UPDATED a SAS Data Set that was OUTPUT of program 5. To ensure valid results, this program 5 has to be re-executed as well.

This consideration results in an extended prediction algorithm that predicts up front what programs will need to be re-executed and that checks within the execution again if prediction was correct.

This feature now avoids useless re-execution of already executed programs. This is helpful in development, because the user can simply run the complete run-all program and asking POStER to execute all programs, POStER will chose the programs to be executed and ignores the programs that would produce the same results.
In final execution, this can even help reducing the run time after a data base re-opening. If only one data set was changed, only related programs need to be run again.

PITFALLS

Within the development of POStER, some pitfalls have been found and needed to be overcome.

ENVIRONMENT VARIABLES

The most common pitfall is how SAS handles environment variables. Especially when executing SAS 9.4 from a SAS 9.2 session, or vice versa, this can result in unexpected errors. The solution is removing the related environment variables.

EXECUTION PREDICTION

When doing execution prediction, it is possible that users require eternal input. For example if input data is taken from an external Data Base. In this case, the execution prediction might not be possible, as database changes are not noticed as external file system changes. For these cases, POStER contains a force execution parameter. This forces the execution of a program independently from the prediction algorithm.

CONCLUSION

Using parallel execution of multiple SAS programs can result in big performance increase and can be archived relatively easy. A lot of additional features are possible bringing extra benefit but also create additional complexity.

ACKNOWLEDGMENTS

This paper and the complete POStER system would never have been possible without my colleagues at Bayer, who always come up with these fancy ideas.

RECOMMENDED READING

SAS-HELP related to SYSTASK and WAITFOR
Man page of strace

CONTACT INFORMATION

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

Michael Weiss
Bayer AG
Address
13342 Berlin, Germany
Email: michael.weiss[at]bayer.com