Paper 1093-2017

Run It in Parallel: Improving the Flow of Windows Services

David Kratz, D-Wise Technologies Inc.

ABSTRACT

SAS® Job flows created by Windows operating system services have a problem: At present they can only execute jobs in series (one at a time). This can slow job processing down, and limits the utility of these flows. This paper shows how one can alter Windows operating system services flows after they have been generated to enable jobs to run in parallel (side by side). A high level overview of a SAS PROC GROOVY script which automates these changes is provided, as well as a summary of the positives and negatives of running jobs in parallel.

INTRODUCTION

A SAS® job flow is collection of SAS jobs and a set of dependencies which control when the jobs run. In the simplest case, when no dependencies are set, all the jobs run simultaneously. More commonly, one or more jobs are dependent on the successful completion of another job or jobs. Depending on the type of flow being created, other events can also be incorporated, such as date/time events, the existence of files, the failure of a job, etc. SAS job flows are particularly useful when a process consisting of many SAS jobs needs to be repeated at regular intervals.

Some years ago, while helping to construct a data warehouse for a health insurance client, a problem was encountered. Users wanted the ability to create their own small warehouses containing data from limited periods of time for testing purposes. For this project, we were using LSF's Platform Process Manager to run the job flows that constructed and maintained the data warehouse, which functioned very well. However for compliance reasons, access to the account which owned those flows was heavily restricted. It was technically possible to give each user their own copies of the job flows, but that would have resulted in a multiplication of metadata objects and made it difficult to keep each user's copies of the flows up to date. Users could still create their own warehouses by running the relevant jobs manually[1], but this was tedious, time consuming, and error prone.

Looking for an alternative solution, we considered making duplicate operating system service flows available to the users. These would have resolved the compliance issue and lessened the metadata burden, but they had a downside: they only ran jobs in series. This meant that creating a warehouse containing even a small amount of data could take a very long time. In the end, the series nature of operating system service flows proved too slow, and we took to generating the user's warehouses at their request using Process Manager. Since then, it has become clear that it is possible to work around this limitation as non-Windows operating system services scripts already support running jobs in parallel. This paper describes a method by which Windows operating system service flows can be modified to support running jobs in parallel, and why you might wish to do so.

RUNNING JOBS IN PARALLEL

WHAT IS PARALLEL PROCESSING?

Parallel processing is running multiple processes at the same time. The original definition of the term involves having multiple processors in a single computer, each running their own process or portion of a process. With regards to SAS there are several different ways of processing in parallel.

Threading is a programming technique by which a single processor runs one or more streams of executable code simultaneously, typically through timesharing. This allows the CPU to remain active on

[1] Users were able to do this because we had implemented user specific external options files, as well as user playpens. Those topics are outside the scope of this paper, but are covered in 039-2010 SAS® Data Integration Studio Tips and Tricks and 419-2013 Making do with less: Emulating Dev/Test/Prod and Creating User Playpens in SAS® Data Integration Studio and SAS® Enterprise Guide® respectively.

one thread, while another thread is waiting for input. When multiple processors are available, a single process can be broken down into several pieces, and each executed on a separate processor. In SAS, some PROCs are multithreaded, which means that they were written in such a way as to take advantage of this technique. A few examples are PROC Sort, PROC SQL (order by, and group by), and PROC MEANS.

Job based parallel processing occurs when multiple SAS jobs are run concurrently on a single machine. When this happens, the operating system will assign the jobs to individual processors or processor cores, allowing work to be done on all assigned jobs. In the case where there are more SAS jobs being run than there are processing units available, the jobs can either share processor time (essentially taking turns) or one job can be allowed to run to completion while the others wait.

There is another kind of job based parallel processing that occurs when a user's jobs are run on more than one machine. A SAS grid is an increasingly common example of this sort of setup. In this case, the jobs are sent to waiting machines (or nodes) where they are processed independently of each other. The results are then either returned to the submitting computer or written out to commonly available file space.

BENEFITS OF RUNNING JOBS IN PARALLEL

Running jobs in parallel can maximize the use of system resources. While some SAS processes can make use of multiple cores, most cannot. Similarly, most systems have RAM in excess of that needed for a single SAS job. Thus, running multiple jobs simultaneously can allow the utilization of system resources that would otherwise be idle.

This can result in lower total run times than running the same jobs in series. The effect varies based on how amenable the system is to running jobs in parallel and the kind of dependencies present in the flow. Essentially, during long running jobs, shorter jobs with the same dependencies can complete, rather than waiting for the longer job to finish before starting. Figure 1 illustrates an idealized case of how running jobs in parallel can reduce run times.

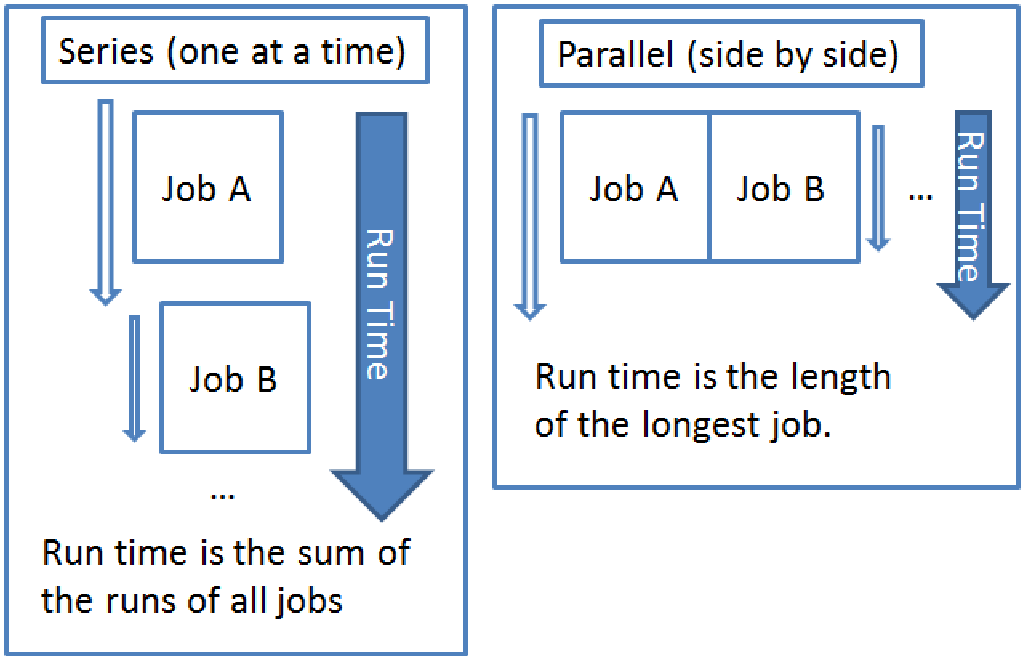
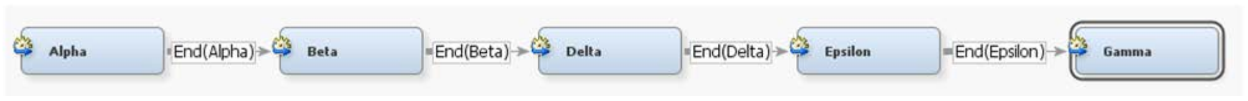


Figure 1. For a set of jobs with same dependencies, if hardware limitations are not a factor, running them in parallel will be faster than running them in series.

DANGERS OF RUNNING JOBS IN PARALLEL

The flipside of using more of a machine's resources, of course, is that running too many jobs in parallel can demand more from the system than it has to give. This can lead to contention between jobs as they demand limited memory, processor, and I/O resources, which may result in run times longer than the jobs would have had when run independently. The I/O issue is especially important, as SAS processes tend to be bottlenecked by I/O speed and availability.

Additionally, some flows will not, by their design, benefit from the ability to run in parallel. For example, if most of a flow's runtime consists of a chain of jobs, each dependent on the last then there will be little to no efficiency gain. Display 1 provides an example of what that kind of flow might look like. The dependencies of its jobs will force what is, in effect, a series run.



Display 1. A flow which will not benefit from being run in parallel, as viewed in SAS Management Console's Schedule Manager

WINDOWS OPERATING SYSTEM SERVICES FLOWS AS THEY ARE NOW

CAPABILITIES

Presently, Windows operating system services flows run all jobs in series (Scripts generated for UNIX and Linux systems do not share this behavior). They offer support for two different logical gates:

1. AND - Run the dependent job(s) when all of the AND gate dependencies are met.
2. OR - Run the dependent job(s) when any of the OR gate dependencies are met.

The dependencies themselves consist of four different events:

1. Job starts – The job begins processing.
2. Job ends with any exit code – The job completes, successfully or otherwise.
3. Job ends with certain exit codes – The job completes with an exit code matching a specified one.
4. Job completes successfully – The job ends without an error.

The fact that Windows operating system service flows only support series processing has two main consequences. First, the flows suffer the potential speed loss that has been previously outlined. Second, the Job starts event is functionally equivalent to the Job ends with any exit code event, as Figure 2 illustrates. Because the flow script runs in series, execution of the script is paused while jobs run. Control is not returned to the flow script until after the kicked off job (job a) finishes, which means that there is never a chance for a second job (job b), that was dependent on the first job starting, to be kicked off until after the first job ends.

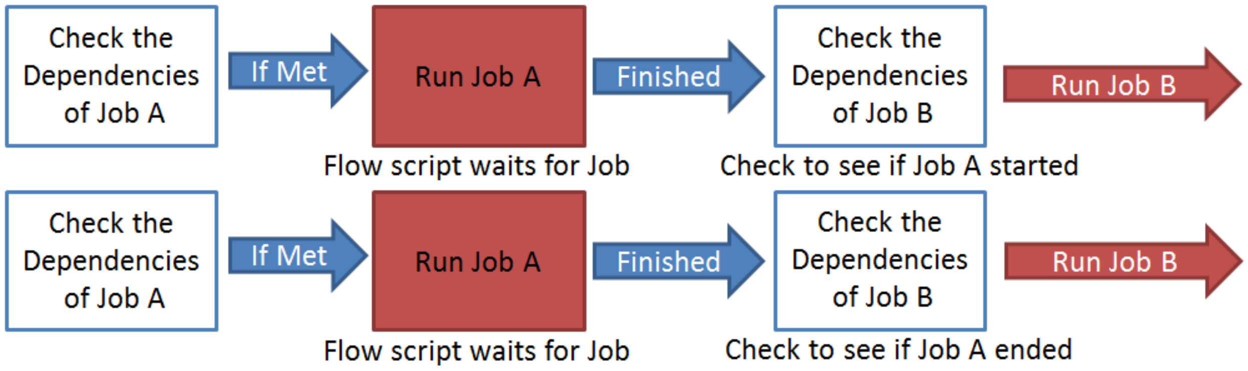

Check the Dependencies of Job A
If Met
Run Job A
Finished
Check the Dependencies of Job B
Run Job B
Flow script waits for Job
Check to see if Job A started
Check the Dependencies of Job A
If Met
Run Job A
Finished
Check the Dependencies of Job B
Run Job B
Flow script waits for Job
Check to see if Job A ended

Figure 2. A demonstration of the functional equivalence of the Job starts and the Job exits with any error code events. Note that regardless of which was used as a dependency, Job B would be

run at the same time.

SCRIPT STRUCTURE

Windows operating system services flows are deployed as Visual Basic script files (.vbs). The scripts themselves can be subdivided into three main sections:

1. A header portion – A timestamp is generated using the current system time, variables are defined, necessary scripting objects are instantiated, and a log file is begun.
2. Job invocations – There are as many of these as there are jobs in the flow. If dependencies were defined in the flow, all job invocations take place inside of a single loop, and the dependency checks happen before any job is kicked off.
   A timestamp is generated and the log is updated to reflect that a job has been launched. Afterwards, the job is kicked off and the script does not continue executing until the job has finished. Once it has finished the log is updated with its status, and various state variables regarding the success and failure of the job are set.
3. A footer portion – A timestamp is generated and the log is updated with the status of the flow when it finished. The log file is then closed, and the flow ends.

A note on the dependency loop referred to in step 2: The generated code is structured in such a way that the loop will recur if any job successfully has its dependencies met and is invoked during the current iteration of the loop. Any iteration of the loop could run an arbitrary number of jobs, but the loop is only guaranteed to iterate one additional time. If any iteration of the loop fails to launch a job, the loop will end.

An example of a commented, but otherwise unmodified Windows operating system services flow script (ExampleFlow_annotate.vbs) accompanies this paper.

CHANGES REQUIRED TO ENABLE JOBS TO RUN IN PARALLEL

Note - The steps detailed here are just one way of approaching this problem. This method was chosen for convenience and because it created output that was largely similar to the original structure of the generated scripts.

For an in depth example of how these steps are carried out, please refer to the attached SAS code (Parallellize_Flow.sas). Making use of PROC GROOVY, an example which automates the procedure has been written. It is important to note that the use of the GROOVY procedure was merely for convenience, any programming language capable of manipulating text could have been used. For further discussion of the code sample, please refer to the Appendix: Parallellize_Flow.sas - A SAS program for automatically enabling a flow to run in parallel.

SPLIT JOB INVOCATIONS INTO THEIR OWN SCRIPTS

There are two main techniques used to allow Windows operating system services scripts to run jobs in parallel. The first is separating the generated script's invocations of SAS jobs into separate subscripts. This allows the subscripts to run their respective jobs while the main script continues processing and for the necessary signaling logic to be added without modifying the text of the job.

To perform the separation, the main script is parsed and split into the three parts mentioned in the previous section. Once that's done, new subscripts can be generated for each job invoked. These subscripts combine a modified copy of the main script's header, the original job invocation, and the code responsible for the production of a signal file (the necessity of which will be explained in the next section). The master script's text is altered to call these subscripts, instead of the SAS jobs directly.

ENABLE MAIN SCRIPT TO DETECT THE PRESENCE OF SIGNAL FILES

A consequence of using subscripts is that there must be some method of communicating when each subscript's SAS job has finished as well as the job's return code back to the main script. The simplest way of doing this is to use the file system as a form of communication. If each subscript produces a signal file, the master script can be modified to use the presence or absence of a signal file as a test for job completion. The text of the signal file can be used to pass along the job's return code.

To do this, code is injected into the end of the master script's job invocation loop such that it searches for the signal files created by the subscripts. The return code pulled from those files is used to determine whether or not the dependencies have been met for other subscripts. The master script's job invocation loop will continue to run until no subscript is running and no other jobs can be invoked. To avoid a tight loop, a sleep command causes the system to wait between iterations.

Figure 3 illustrates how the modified script differs from the original script. Note that both scripts are not repeatedly invoking the same SAS job or subscript. Rather, the figure refers to an arbitrary job or scripts for example purposes.

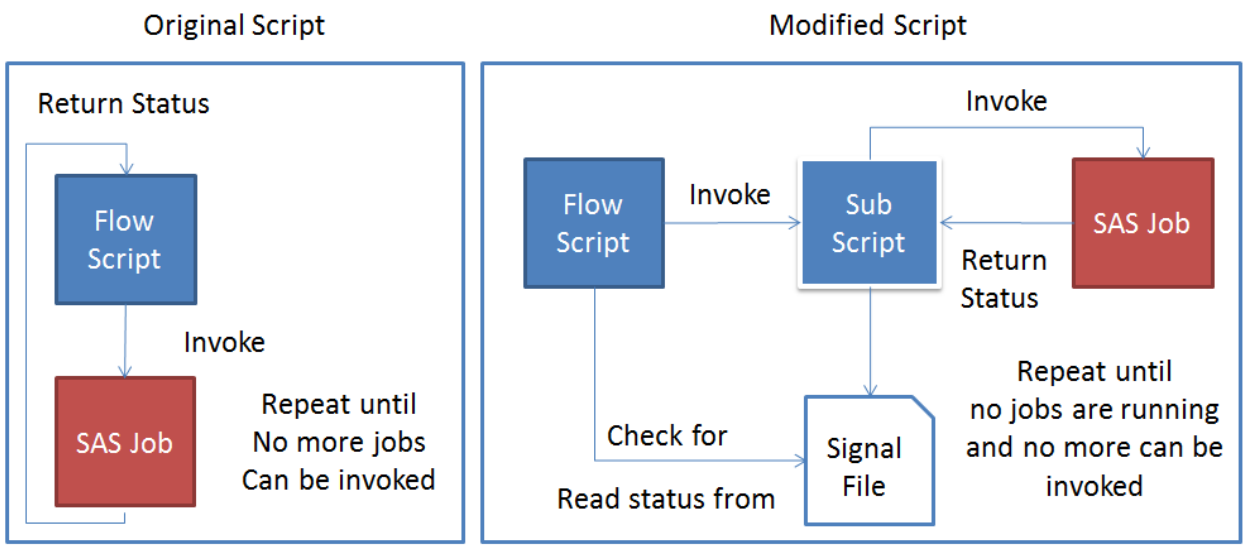


Figure 3. An illustration of how the modified flow script differs from that of the originally generated flow script.

An example of a commented and parallel modified Windows operating system services flow script (ExampleFlow_Modified_annotate.vbs) accompanies this paper, along with a commented example of a subscript (A571IL1G_AH000002_annotate.vbs).

CONCLUSION

Windows operating system services scheduling provides a lightweight alternative to using LSF's Platform Process Manager to create job flows. However, the flows it generates can only run jobs in series, which can limit their capabilities and slow them down considerably.

It is possible to alter the scripts generated by Windows operating system services, after they have been created, to support running jobs in parallel. The process to make these alterations can be automated, and the code required to do so can be run entirely within SAS.

While the ability to run jobs in parallel is useful, it is not appropriate in all cases. Care must be taken while designing jobs flows to take advantage of the capability and to avoid overtaxing the system on which the flows will be run. Running too many jobs at once can provide performance that is worse than simply running them all independently.

APPENDIX: PARALLELLIZE_FLOW.SAS - A SAS PROGRAM FOR AUTOMATICALLY ENABLING A FLOW TO RUN IN PARALLEL

COMMENTS ON PARALLELIZE_FLOW.SAS

Why PROC GROOVY

Parallellize_Flow.sas, the attached SAS program, was written to provide an example of a program which would automate the changes required by Windows operating system service generated flow scripts to run jobs in parallel. Parallellize_Flow.sas is a single PROC GROOVY step; however the GROOVY procedure is not required for the operations that it carries out. Any programming language capable of manipulating text, including Base SAS, could have been used. PROC GROOVY was chosen because it provided a set of data structures and functions which made the text manipulation required easy. Additionally, since the code can be run within Base SAS (versions 9.3 and later) it should be universally available to anyone hoping to modify Windows operating system services generated flow scripts in this manner.

Program Flow

At a high level, the program operates by performing the following steps:

1. Validate the target script – Parallellize_Flow.sas reads in the target flow script and performs some very rudimentary validation. The program determines if the target script has already been parallelized and if the target script has only one job invocation. In either case, the script aborts. While it's validating, the program also determines if the script has any defined dependencies. This is important because having no defined dependencies is a special case that alters the entire structure of a flow script, and it requires extra logic to deal with.
2. Parse the target script – Assuming the file is valid, the program breaks it into pieces. It does this by searching the text of the script for certain lines which delineate sections of the program. These were identified by examining the structure of several generated scripts. The various chunks are stored in array list structures.
3. Modify chunks – Once broken into pieces, the script can be adjusted. Modifications are made so that the main script will call the subscripts, and logic is injected such that the main script will be able to monitor for signal files. Chunks which will be part of the subscripts are modified as well, allowing them to write out their job specific signal files. This is also the time when special cases are dealt with, like when a job has no dependencies, or modifying the dependency check generated for the Job Starts event.
4. Output the modified file – The original script file is backed up first. Then, the main script and the subscript are reconstructed from the modified chunks. Rather than write the files out line by line, a single long string is created for each file, and the output takes place all at once.

Potential improvements

Parallellize_Flow.sas serves as a proof of concept that it's possible to programmatically generate Windows script flows which run jobs in parallel. There are several potential improvements, however, which include the following:

- Optimize the files used for communication - The use of both a subscript simple status file and a subscript status file is convenient, but unnecessary. With further refinement, it would be possible to parse the return code from the status file.
- Sort the log file entries – Since running SAS jobs in parallel is essentially a threaded process, it is possible get entries in the log out of order. If job b finishes first, but its signal file is checked after job a, its log entry will appear after job a's log entry as well, regardless of how much earlier job b finished. Incorporating a simple sorting algorithm into the main script would allow entries to be appended to the log in time order.
- Incorporating the use of functions – Windows operating system services flow scripts contain a lot of redundancy. At present, the code injected into the flow by Parallellize_Flow.sas does as well. By creating functions in the .vbs script and simply calling them, it would be possible to create more compact and easier to read code.

Availability Online

A copy of parallelize_flow.sas should accompany this paper. However, at time of writing, it is also available online (with the accompanying example scripts) at https://github.com/dakratz/parallelize_flow

INSTRUCTIONS FOR USE

In order to use Parallellize_Flow.sas, perform the following steps:

1. Modify the job to point at a targeted flow script – The copy of Parallellize_Flow.sas that accompanies this paper contains the line:
   ```
   targetScript = "C:/SAS/Config/Lev1/SchedulingServer/d-
   wise_dkratz/test3/test3.vbs";
   ```
   Modify the quoted path to point to the script you would like to enable parallel operation on. Note that the slashes are reversed from the usual for Windows paths ('/' instead of '\').

   The annotated example of an unmodified flow which accompanies this paper should not be used with Parallelize_flow.sas. The additional lines introduced by the annotations may cause unpredictable behavior.
2. Submit the script in SAS.
3. Run the generated .vbs file to launch your parallelized flow.

Interaction with Schedule Manager

When scheduling a Windows operating system services flow through SAS Management Console's Schedule Manager plug-in, a number of options are provided. While it had been hoped that maintaining the .vbs format would allow Schedule Manager to interact seamlessly with parallelized flows, the results were mixed. The three options provided are:

1. Manually in Scheduling Server – Every time a flow is scheduled using this option, the existing .vbs script in the appropriate folder is overwritten. This happens regardless of the .vbs scripts contents. Thus, if you use Parallellize_Flow.sas on a flow, and then reschedule it afterwards, you will need to use Parallellize_Flow.sas again before the flow will once again run jobs in parallel.
2. Run Now - Parallellize_Flow.sas cannot interact with flows submitted in this way. Even if no change has been made to the flow in metadata, choosing to Run Now will overwrite the existing .vbs script with a new copy of the unmodified script. It will then launch the unmodified script.
3. Manually in Scheduling Server with a date/time event – As in the first option, scheduling the flow will overwrite the existing version and necessitate rerunning Parallellize_Flow.sas. However, as long as this is done before the date time event triggers, the modified version of the flow will be run.

CONTACT INFORMATION

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

David Kratz
D-Wise Technologies, Inc.
David.Kratz@d-wise.com