Paper SAS366-2017

Microservices and Many-Task Computing for High-Performance Analytics

Henry Bequet, SAS Institute Inc., Cary, NC

ABSTRACT

A microservice architecture prescribes the design of your software application as suites of independently deployable services. In this paper, we detail how you can design your SAS® 9.4 programs so that they adhere to a microservice architecture. We also describe how you can leverage Many-Task Computing (MTC) in your SAS programs to gain a high level of parallelism. Under these paradigms, your SAS code will gain encapsulation, robustness, reusability, and performance. The design principles discussed in this paper are implemented in the SAS® Infrastructure for Risk Management (IRM) solution.

Readers with an intermediate knowledge of Base SAS and the SAS Macro Language will understand how to design their SAS code so that it follows these principles and reaps the benefits of a microservice architecture.

INTRODUCTION

The purpose of SAS® Infrastructure for Risk Management (IRM) is to provide SAS programmers with an easy-to-use development platform for the fastest analytics. To support the decentralized nature of today's development teams, the ease of use must include premium support for a distributed development model. This is where microservices are essential: each piece of SAS code defines and documents a public interface that other developers can code against or even replace if necessary. In this paper, we will look at the details of the methodology that makes this possible. To support the fastest analytics, IRM leverages the Many-Task Computing (MTC) paradigm: each piece of *independent* SAS code (or SAS microservice) can run on as many cores as available. We will also look at the approach that supports the automatic identification of parallelism in SAS microservices.

But before we can get into these details, let's have a closer look at the problem that gave birth to IRM.

CHALLENGES

In order to understand the challenges that we encountered while deploying SAS® Firmwide Risk for Solvency II, one has to understand the nature of the deployment.

Hundreds of SAS MVA files make up the bulk of the body of code that needs to be deployed in order to take advantage of large servers. Previous attempts that relied heavily on the SAS Stored Process Server failed in the past for the following reasons:

1. **Complexity of the solution.**
   Writing and deploying hundreds of stored processes can be very complex because of the sheer volume and interactions of the SAS macros involved in such an endeavor. We needed a solution that promoted encapsulation and reuse of the code.
2. **Speed of the solution.**
   To manage the interactions of multiple users, the concept of a playpen was introduced. The main drawback of playpens was the fact that you had to manage them: operations like creation and promotion to a shared playpen would introduce a lot of data movement. We needed a solution that would eliminate the management of playpens and their associated data movements.
3. **Quality of the solution.**
   With complexity and lack of encapsulation typically comes a lack of quality.

To have an answer to those challenges, we developed IRM.

It is nearly impossible to write a paper on computer performance without mentioning Moore's law (Moore 1965) combined with the fact that CPUs are not getting any faster. The net-net for Computer Scientists is that to go faster, you need to go parallel. In other words, since your programs will not run on faster

computers, all other things being equal, to go faster you must split your programs into parallel sub-programs that will run on the many cores that Moore's law keeps giving us.

Now that you have a better understanding of the challenges that we are facing, you are probably eager to dive into details, but first we need to define some concepts.

MICROSERVICES

Martin Fowler defines microservices as follows (Fowler 2014):

> *a particular way of designing software applications as suites of independently deployable services*

So how do you write SAS code so that the result constitutes an *independently deployable service*? One possible answer, but not necessarily very useful, is to deploy SAS code that cannot interface with anything. That code will be independently deployable, but it cannot do anything useful since it cannot consume or produce any data. If we want the SAS code to do something useful, it must define an interface. In the case of microservices, deployed as components of a Web application, the interface is typically a REST API (Fielding 2000). In the case of microservices deployed as components of a SAS application, we propose to define the interface in terms of SAS data sets. More specifically, each SAS microservice must define its inputs and its outputs in terms of data objects, where a data object can be a SAS data set, an Excel document, or basically any file. The definition of a data object typically implies a data model (e.g. the variables of the data set), but not the actual values.

One of the implications of defining inputs and outputs is that SAS code defined (and deployed) as a microservice may not use any global data. By global data we mean *anything* that is shared outside the SAS code that constitutes the microservice. In that category, you would find any macro variables, any SAS data set, any file, … anything!

It is probably worth wondering for a paragraph or two if and why banishing global data is the way to go. Global variables have been recognized as a source of programmer's angst for a long time (Wulf 1973); most programmers have learned either through schooling or painful experience to stay clear of them. What we are saying in this case is that global data must be viewed in the same light. In short, global data makes the state of the overall system unpredictable and you don't want that type of chaos injected into a distributed development and deployment model. At least if you want to keep your sanity.

Defining inputs and outputs is easier said than done; one can quickly slip up by defining a LIBNAME or a fileref that is not part of the interface. A platform like IRM makes things a lot easier by enforcing those rules.

In addition to a well-defined interface through inputs and outputs, you don't have a microservice without the ability to specify the behavior of all the tiers: that is GUI, middle tier, data, and computational tier. We will discuss how IRM solves that problem in a few paragraphs.

MANY-TASK COMPUTING (MTC)

MTC is a fairly new parallel-computing methodology (Raicu 2008) that is used in super computing and high-performance computing. The idea is simple: organize your work in independent activities (called tasks) that are coupled via file system operations. The tasks can be small or large, but smaller is usually better because the main goal of MTC is to achieve a high degree of parallelism. Beyond the file system operations, there are no explicit synchronization primitives in MTC. This makes the methodology particularly suitable for legacy computing languages that don't support parallelism (like SAS macros).

As we discussed in the previous section, SAS microservices explicitly declare their inputs and outputs, which happen to be files. So, by postulating that SAS microservices are tasks, we can easily combine them with MTC under one umbrella to add parallelism. However, there is key ingredient missing in this recipe: immutability. Let's see why.

Software developers have known for quite some time that immutability of data in general and objects in particular contributes to safe parallelism (Goetz 2003). What we propose here is to extend that concept to data objects in general and SAS data sets in particular: once a SAS data set is produced it can no longer

be changed; that is, no values can be changed, the data set cannot be sliced, trimmed, augmented, or even sorted.

Armed with immutability, we can limit the file operations that are handled by MTC to read and write permission. Specifically, once a data object is produced, all the tasks that use that data object as input can now be scheduled for execution. In effect, the synchronization is on the data; more precisely, the synchronization is on the availability of the data.

The immutability of data objects has far-reaching consequences. For instance, there is never a need to synchronize beyond the availability of data because only one task can produce a given data object; if it were not the case, then the data object wouldn't be immutable. This synchronization based on the data is a lot easier to deal with than the typical thread-based model that is laden with nondeterminism (Lee 2006). Note that we must realize that synchronizing on the inputs and outputs rather than inside the code doesn't apply universally. Granularity of the shared resource (the data object) plays a great role in the decision. For instance, implementing a parallel matrix multiplication algorithm using MTC will yield disastrous performances. In this case, the unit of shared resources (a single cell in a matrix) is too small. Conversely, implementing an algorithm that calls for the transformation and replication of petabytes of data is unlikely to work well with immutable data.

The rule of thumb is simple: MTC works well with many tasks and many data objects. Tasks should be short in order to maximize parallelism. Data objects should be small in order to minimize the impact of synchronization and immutability.

DEFINING A MICROSERVICE IN SAS CODE

In IRM, we have elected to bundle the SAS code and the required metadata in one SAS file. Other systems like the SAS Cloud Analytic Services (CAS) support the definition of the metadata in a separate parameter file written in Lua. Either choice works but for IRM since the metadata tends to be small, we prefer to keep everything in one file so that changes to the code and the metadata can easily be kept in synch.

Let's look at a simplified example to see how this works with the content of the `price_option_binomial.sas` file (the meaning of the arguments is unimportant for this discussion):

```
/**
   \file
   \brief The node calculates the current price of option instrument using binomial model.
   <b> Identified Inputs </b>
   \param[in] MK_OPT.OPTION_INSTRUMENT.SAS7BDAT      input option instrument data
   \param[in] MK_TEMP.FUNCS.SAS7BDAT                 location of the functions registered before

   <b> Identified Outputs </b>
   \param[out] MK_OPT.PRICE_OPTION.SAS7BDAT          priced instrument data

   \author SAS Institute INC.
   \date 2017
*/
%irm_mkt_price_option_bi(IN_DS MK_OPT.OPTION_INSTRUMENT ,
                         FUNC_LOCATION=MK_TEMP.FUNCS ,
                         OUT_DS MK_OPT.PRICE_OPTION);
```

As you can see, the file contains metadata between `/**` and `*/` followed by the actual SAS code, a call to the `%irm_mkt_price_option_bi` macro. Via the `\param[in]` and the `\param[out]` tags, the SAS programmer specifies the inputs and outputs. The input/output metadata is the most important

information. The remainder of the tags allows for the automatic generation of help and documentation files.

This pattern is reproduced for all of the tasks that make up the deployment. There are no exceptions: all tasks must define their inputs and outputs.

Armed with the metadata, IRM can now build job flows on your behalf.

JOB FLOWS

A job flow is an ordered list of the tasks and data objects where the order is given by the data dependencies. These dependencies allow IRM to compute a Directed Acyclic Graph (DAG) that is essential to the execution of the SAS code. Let's look at an example to understand these concepts. Figure 1 below shows an example of a DAG. If you direct your attention to the tasks labeled 'Price Option Using Binomial Model' and the task labeled 'Option Fx Conversion Binomial Model', you'll notice that they are joined by an arrow, whichindicates that one task comes before the other. That order is implied because the data object highlighted in blue and labeled 'Price option by Binomial Model' is output of the first task and input to the second task. IRM computes (and draws) the DAG automatically.

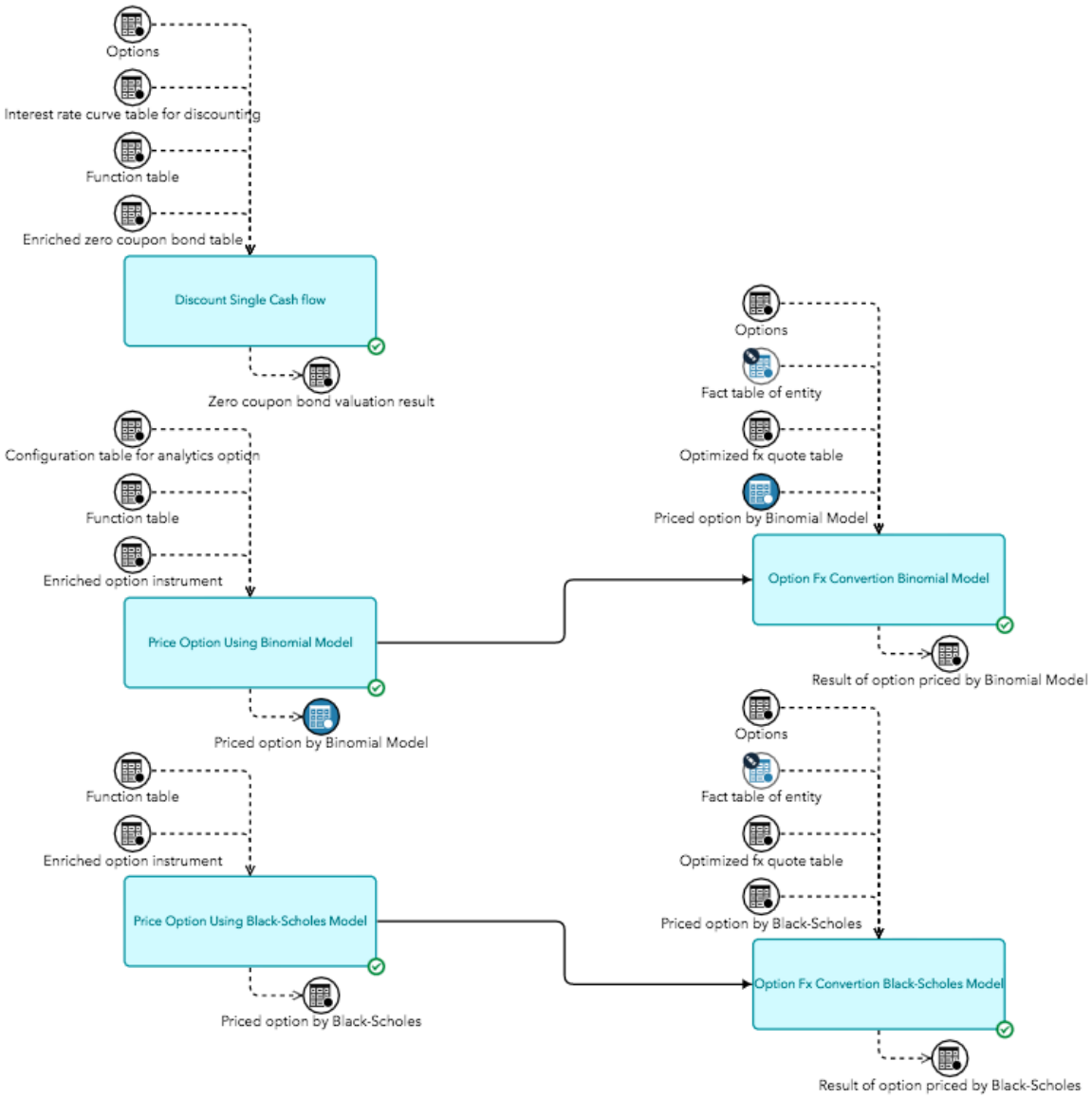

Figure 1. A Simple Job Flow

Regarding scheduling, it is worth pointing out that in the DAG of Figure 1 there are two (vertical) levels: the first one with 3 tasks and the second one with 2 tasks. Those levels constitute the tasks that are independent of each other and, consequently, that can run in multiple threads. That implicit parallelism is key to the ease of use that IRM provides: SAS programmers can define very complicated parallel algorithms that can run on hundreds of machines and thousands of cores by simply defining the inputs and outputs of their tasks.

PERFORMANCE OF GRID SCHEDULING

At this point, it is probably worth mentioning a key advantage of IRM when it comes to scheduling tasks in a multiserver environment.

RAM is typically much faster than disk I/O, unless you invest in a very expensive storage solution that is likely to rely on RAM (e.g. RAM SSD) to reduce latency and improve throughput. Modern operating systems like LINUX rely on a disk cache (or page cache) to speed things up (Love 2010), and in IRM job flows where I/O operations are significant, using the disk cache can mean orders of magnitude of performance improvements. When all tasks in a job flow run on one machine, the output of a task is typically in RAM by the time that the next task is ready to consume it. Essentially, there is nothing to do to gain the performance advantage of the disk cache in an IRM job flow: simply let the OS do its work. This is not true when the tasks of a job flow don't run on a single machine. If the first task in our example above runs on one machine and the second task runs on another machine, the data will not be in memory when the second task is ready to run. Even worse, the output of the first task must be transferred to the machine of the second task over the network. All this data movement significantly slows down the execution of tasks to the point that adding more machines slows down the execution of the job flows. To avoid this frustrating situation, IRM schedules the work of tasks on machines so as to minimize the data movements and to optimize the usage of the disk cache. Presenting the details of the scheduling is outside the scope of this introductory paper, but it is worth noting here that without the knowledge of the inputs and outputs, IRM would not be able to minimize the data movements and optimize the usage of the disk cache.

We can see that relying on microservices and MTC provides an easy solution to a complicated problem (Masahiro 2014).

DATA-OBJECT POOLING

We have seen that tasks perform all operations in IRM. In addition, we've postulated that all tasks must define their inputs and outputs to be valid microservices that can be scheduled for execution using the MTC paradigm. Another way to look at these statements is to say that all outputs in IRM are computed by a succession of inputs, tasks, and outputs.

For each output in the system, IRM remembers the subset of the DAG that was used for the computation. That information is a simple ordered set of data objects and task identifiers. So, whenever IRM needs to schedule a task for execution, it knows if that output already exists after a lookup for the ordered set.

The savings that are provided by data-object pooling are twofold: speed and space. Obviously, not executing the task at all can provide tremendous performance improvements, if the lookup is much faster than executing the task. In the case of IRM, the lookup is in fact an in-memory hash table lookup that will beat the combination of disk access and computation every time. The space savings are also significant because if IRM doesn't need to run the task to create the output, it can simply create a file system link to the existing output and be done. The savings come from the fact that storing the link is much smaller than storing the information. In practice, the outputs are managed using a reference counter and can be discarded when the reference counter reaches 0. There is also an ancillary savings that comes from the fact that by reusing outputs, the disk cache is better utilized: fewer page faults lead to more I/O operations performed in RAM.

With the job flows defined in SAS® Firmwide Risk for Solvency II, we have measured both a performance and a disk space improvement close to one order of magnitude with typical user workflows. The more users and the more often the flows are executed and/or modified, the better the savings.

One last word before we conclude this paragraph: there are no security risks introduced by data-object pooling because one user has no idea that their outputs were in reality computed by someone else. The only thing that the end user notices in IRM with data-object pooling is that the system becomes faster and faster as it is used.

Here again, relying on microservices and MTC provides an easy solution to a complicated problem.

CHALLENGES REVISITED

Let's revisit the challenges that we discussed at the beginning of this paper and examine how IRM addressed them:

1. **Complexity of the solution.**
   With microservices, SAS programs are independent of each other. Programs can only interact via a well-defined protocol (the inputs and outputs) and don't have side effects. This level of encapsulation greatly reduced the complexity of the deployments.
2. **Speed of the solution.**
   IRM reliance on MTC along with the scheduling that takes advantage of the file cache goes a long way to improving the speed of the solution. The concept of job flow coupled with the data-object pooling created an environment where the creation of a playpen is automatic. In addition, different users can share results without compromising security. These features kept data movements to a minimum, leading to much better performance.
3. **Quality of the solution.**
   The encapsulation provided by microservices goes a long way to improving the modularity and the testability of the solution.

CONCLUSION

We have seen that the combination of microservices and Many-Task Computing (MTC) is a powerful one. The union of these two paradigms enables SAS programs to reach unprecedented levels of performance with the simplicity of data synchronization. The SAS® Infrastructure for Risk Management (IRM) automates the infrastructure required to support microservices and MTC.

In future developments, IRM will concentrate on ease of use and integration with other SAS products, like SAS Studio and SAS Cloud Analytic Services (CAS).

REFERENCES

Moore, Gordon. 1965. "Cramming more components onto integrated circuits." *Electronics*. Volume 38, Number 8, April 19, 1965.

Fowler, Martin. 2014. "Microservices." Available at http://www.martinfowler.com/articles/microservices.html.

Fielding, Roy. 2000. "Architectural Styles and the Design of Network-based Software Architectures." Doctoral thesis available at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Wulf, William and Shaw, Mary. 1973. "Global Variable Considered Harmful." ACM SIGPLAN Notices, Volume 8, Issue 2, 1973 February, pp. 28–34.

Raicu, Ioan and Foster, Ian and Yong, Zhao. 2008. "Many-task computing for grids and supercomputers." 2008 Workshop on Many-Task Computing on Grids and Supercomputers. Available at http://ieeexplore.ieee.org/document/4777912/.

Goetz, Brian. 2003. "Java theory and practice: To mutate or not to mutate?" Available at http://www.ibm.com/developerworks/library/j-jtp02183/.

Lee, Edward. 2006. “The Problem with Threads.” Technical Report No. UCB/EECS-2006-1. Available at https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf.

Love, Robert. 2010. *Linux Kernel Development (3rd Edition).* Addison-Wesley. ISBN-13: 978-0672329463. Chapter 16, pp. 323-336.

Tanaka, Masahiro and Tatebe, Osamu. 2014. “Disk cache-aware task scheduling for data-intensive and many-task workflow.” 2014 IEEE International Conference on Cluster Computing (CLUSTER). Available at http://ieeexplore.ieee.org/document/6968774/?reload=true&arnumber=6968774.

CONTACT INFORMATION

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

Henry Bequet
SAS Institute
100 SAS Campus Drive
Cary, NC 27513
919-645-7923
henry.bequet@sas.com
http://www.sas.com