Paper 10741-2016

Using SAS® Comments to Run SAS Code in Parallel

Jingyu She, Tomislav Kajinic, Danica Pension

ABSTRACT

Our daily work in SAS® involves manipulation of many independent data sets, and a lot of time can be saved if independent data sets can be manipulated simultaneously. This paper presents our interface RunParallel, which opens multiple SAS® sessions and controls which SAS® procedures and DATA steps to run on which sessions by parsing comments such as /*EP.SINGLE*/ and /*EP.END*/. The user can easily parallelize any code by simply wrapping procedure steps and DATA steps in such comments and executing in RunParallel. The original structure of the SAS® code is preserved so that it can be developed and run in serial regardless of the RunParallel comments.

When applied in SAS® programs that access data from many different data sources and heavy computations, RunParallel can give major performance boosts. Among our examples we include a simulation that demonstrates how to run DATA steps in parallel, where the performance gain greatly outweighs the single minute it takes to add RunParallel comments to the code. In a world full a big data, a lot of time can be saved by running in parallel in a comprehensive way.

INTRODUCTION

The practice of running code in parallel is not new. On the contrary, it is a well-established subtopic of advanced computing. Most computers run on processors with multiple cores, which beg for parallel code. On the other hand, parallel computation has a reputation so intimidating that even seasoned programmers have a tough time getting started. We have made the Base SAS® add-in RunParallel to make the process of creating parallel code as intuitive as possible.

This paper consists of two parts. Part one is mainly user-oriented, as we explore the uses of RunParallel, the syntax, how to work it and when it works best. In part two, we target the tech-savvy with an explanation of how you can attach your own plug-ins in Base SAS® by use of Win32 API and Microsoft® COM technology.

PART I: LET'S RUN IN PARALLEL!

THE MAIN GIST OF IT

RunParallel is built on the idea that time can be saved by executing SAS® code in parallel on multiple SAS sessions. We insert comments in the code to control which process runs which parts of the code. To use RunParallel, we start off with a SAS® program you have already written. RunParallel is installed as a plug-in in Base SAS® and is started directly through the menu. It helps to think of RunParallel as a "little helper":

1. RunParallel starts up multiple SAS® sessions, e.g. **1**, **2**, **3**.
2. RunParallel reads your code from top to bottom. Parts that are wrapped in /*EP.ALL*/ … /*EP.END*/ are executed on all SAS® processes. The parts wrapped in /*EP.SINGLE*/ … /*EP.END*/ are executed in parallel on whichever SAS® processes are available.
3. When RunParallel encounters a statement or SAS® step which is not wrapped inside any /*EP…*/ comments, it reads this as a natural synchronization stop sign. RunParallel waits for all parallel commands to finish executing before it goes on to the unwrapped code, which is by default always runs on SAS® process **1**.
4. If you want to run e.g. a group of STEPS in parallel, and wait for these to finish off before running the next group of STEPS in parallel, manual synchronization is achieved with the comment /*EP.SYNC*/.

INSTALL RUNPARALLEL AND GET STARTED

We have included the installation file for RunParallel on http://tatesoft.com/runparallel.html. Please follow the instructions on the page to get started and try out RunParallel. Installation requires a running version of Base SAS® on a Windows® operating system and administrator rights.

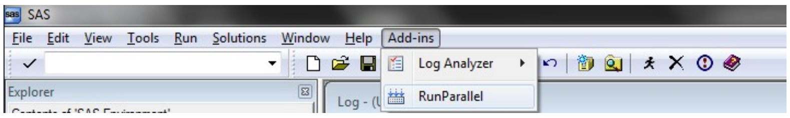
1, 2, 3 GO! SOME SAMPLE CODE TO GET STARTED

Let us start off with a simple example where we simulate 100 million random numbers in two different data steps. Copy the following into an empty Base SAS editor:

```
/*EP.SINGLE*/
data _NULL_;
   call streaminit(123);
   do i = 1 to 100000000;
      u = rand('Uniform');
   end;
run;
/*EP.END*/

/*EP.SINGLE*/
data _NULL_;
   call streaminit(456);
   do i = 1 to 100000000;
      u = rand('Uniform');
   end;
run;
/*EP.END*/
```

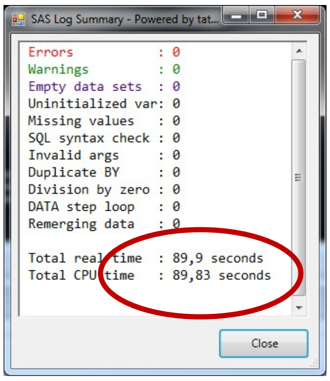
Since the RunParallel commands are wrapped as comments, you may press F3 and run the code in the old fashioned way. Alternatively, you can run the two data steps in two parallel SAS® sessions by opening the RunParallel interface from the menu:



RunParallel will then start up containing the code in the active editor window. Just click submit and watch RunParallel go.

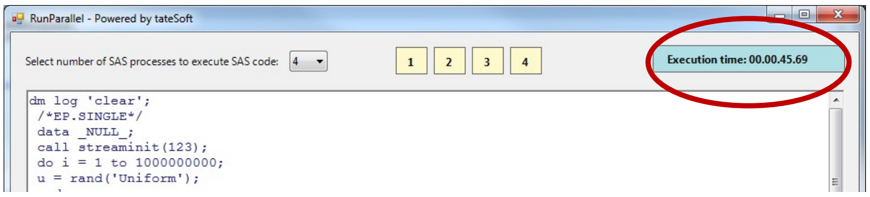
AN INTERLUDE ON PERFORMANCE

We have tried running four data steps as above, where each data step generates one billion random numbers. We then measured the total run time when running in serial (by pressing F3) using another plug-in we have made, named Log Analyzer. We bundle all installations of RunParallel with by Log Analyzer (read more on http://tatesoft.com/loganalyzer-documentation.html). Below images compare the performance of serial (left) against running the four data steps in four sessions in RunParallel (right).



```
Errors           : 0
Warnings         : 0
Empty data sets  : 0
Uninitialized var: 0
Missing values   : 0
SQL syntax check : 0
Invalid args     : 0
Duplicate BY     : 0
Division by zero : 0
DATA step loop   : 0
Remerging data   : 0

Total real time  : 89,9 seconds
Total CPU time   : 89,83 seconds
```



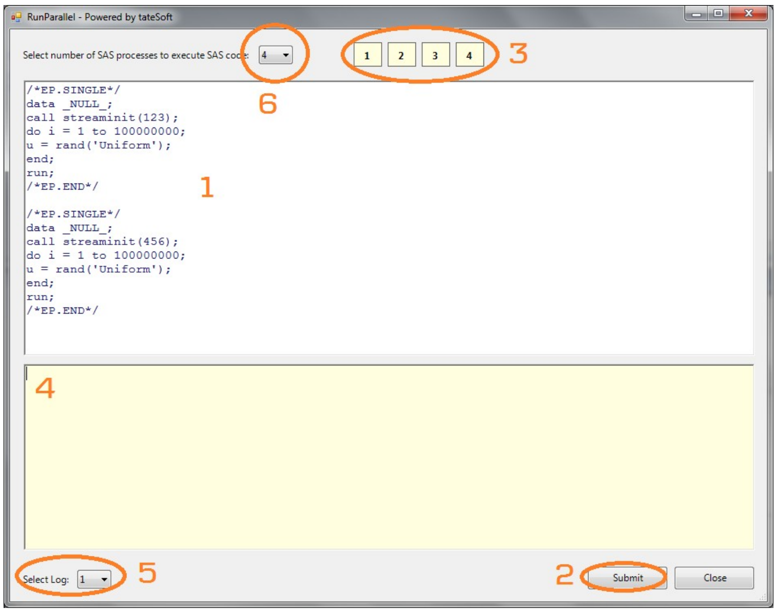
Select number of SAS processes to execute SAS code: 4

Execution time: 00.00.45.69

```
dm log 'clear';
 /*EP.SINGLE*/
 data _NULL_;
 call streaminit(123);
 do i = 1 to 1000000000;
 u = rand('Uniform');
```

THE RUNPARALLEL WINDOW

The first time you start RunParallel, it might take a few seconds to create the necessary SAS® sessions. The window should look like this:


RunParallel - Powered by tateSoft
Select number of SAS processes to execute SAS code: 4
1
2
3
4
Select Log: 1
Submit
Close
1
2
3
4
5
6

```
/*EP.SINGLE*/
data _NULL_;
call streaminit(123);
do i = 1 to 100000000;
u = rand('Uniform');
end;
run;
/*EP.END*/

/*EP.SINGLE*/
data _NULL_;
call streaminit(456);
do i = 1 to 100000000;
u = rand('Uniform');
end;
run;
/*EP.END*/
```

1. The code editor. You may edit your code inside the RunParallel interface. Note that your original code in the SAS editor is preserved.
2. Click "Submit" to run code in parallel
3. The four squares each represent a SAS session. When a SAS session is active, it will light up in green.
4. The Log window. Available for scrolling up and down even while code is running
5. Select which session's log you would like to read.
6. Choose how many SAS sessions you would like to use. Since running all four may hog all your resources, it is sometimes a good idea to run just three or two.

SYNTAX AND DOCUMENTATION

For details, please read http://tatesoft.com/runparallel-documentation.html.

For a brief overview of basic syntax, we have the following:

- **/*EP.SINGLE*/ SAS® code /*EP.END*/**
  The SAS® DATA and PROC steps encapsulated here will be run on an arbitrary single SAS® session, in parallel with code encapsulated in other EP.SINGLE/EP.END commands.
- **/*EP.ALL*/ SAS® code /*EP.END*/**
  The code encapsulated here will be run on all SAS® sessions. This can be useful for e.g. LIBNAME statements, if a library is needed on all SAS® sessions that are started by RunParallel.
- **/*EP.SYNC*/**
  This marks a manual synchronization. At this point in the code, RunParallel will not continue parsing anything before all code before this point has finished executing. This is especially useful if two datasets are first defined in two consequent DATA steps and then sorted in two consequent PROC steps, to prevent sorting of a data set that has not yet been created.
- **/*EP.CPU=2*/**
  This specifies the number of SAS® sessions we want RunParallel to run on (in this example, we choose two). Note that this feature also exists in the RunParallel user interface as a dropdown menu in the upper left corner.

LEARNING BY EXAMPLE:

SORT AND MERGE, THE SETUP:

We consider a folder containing two data sets, A and B, which we sort and then merge. The RunParallel setup would go as follows:

- Execute libname statement on all sessions
- Sort the two data sets A and B in parallel

- Wait for both data sets to be sorted
- Merge A and B into C.

SORT AND MERGE, WHEN IT WORKS:

RunParallel is bottlenecked by the fact that only so much can be written to and from disk. Hence performance can in particular be gained if you would like to sort on many variables, since this is computationally intensive, but still writes the same amount of observations. In particular, performance can be gained if you have multiple (not merely two) data sets to sort.

SORT AND MERGE, THE CODE:

```
/*EP.ALL*/
   LIBNAME MYLIB '[insert your folder name here]';

/*EP.SINGLE*/
   proc sort data = mylib.A; by v1 v2 v3; run;
/*EP.END*/

/*EP.SINGLE*/
   proc sort data = mylib.B; by v1 v2 v3; run;
/*EP.END*/

data mylib.C;
   merge mylib.A mylib.B;
   by v1 v2 v3;
run;
```

READING FROM DIFFERENT EXTERNAL DATABASES USING ODBC, THE SETUP

Let us consider a situation where you are reading data from different departments in the company, i.e. the case where you have multiple different external data sources, into a local SAS® library "mylib". We want to collect tableA from Department A, tableB from Department B and tableC from Department C.

READING FROM DIFFERENT EXTERNAL DATABASES USING ODBC, THE HOW AND WHY

Here, RunParallel is especially strong. In this case, your local task just consists of sending off queries to the separate machines that host the external data. Thus most of the work is done on those machines.

READING FROM DIFFERENT EXTERNAL DATABASES USING ODBC, THE CODE

```
/*EP.ALL*/
   LIBNAME mylib '[insert your folder name here]';

/*EP.END*/

/*EP.SINGLE*/
   LIBNAME depA ODBC DSN = '<DB_NAME_A>' SCHEMA = '<SCHEMA_NAME_A>';
   DATA mylib.tableA;
      SET depA.tableA;
   RUN;
/*EP.END*/

/*EP.SINGLE*/
  LIBNAME depB ODBC DSN = '<DB_NAME_B>' SCHEMA = '<SCHEMA_NAME_B>';
  DATA mylib.tableB;
      SET depB.tableB;
   RUN;
/*EP.END*/
```

```
/*EP.SINGLE*/
  LIBNAME depC ODBC DSN = '<DB_NAME_C>' SCHEMA = '<SCHEMA_NAME_C>';
  DATA mylib.tableC;
      SET extlibC.tableC;
   RUN;
/*EP.END*/
```

RUNNING MACROS, THE SETUP

We would like to achieve the same as we did in the previous example with external data, but copy-pasting the above code is depressingly tedious. Thus we choose to wrap it up in a macro GET_DAT that takes the arguments EXT_DBNAME (name of external database), EXT_SCHEMA (name of schema), EXT_DSN (external data set name) and LOCAL_DSN (local data set name). We execute the macro definition on all SAS® sessions, and call it thrice in parallel, once for each department.

RUNNING MACROS, THE HOW AND WHY

The combination of macros and RunParallel can be a little tricky to grasp. Since RunParallel parses the your SAS® program "as-is", the following will **not** run insides of the macro %WRONG twice in parallel:

```
%MACRO WRONG();

            /*EP.SINGLE*/

                  [DATA STEP]

            /*EP.END*/

      %MEND WRONG;

%WRONG;

%WRONG;
```

Instead, all RunParallel commands must be directly visible in the code. Below is a correct way of combining macros and RunParallel.

RUNNING MACROS, THE CODE

```
/*EP.ALL*/
   LIBNAME mylib '[insert your folder name here]';

   %MACRO GET_DAT(EXT_DBNAME, EXT_SCHEMA, EXT_DSN, LOCAL_DSN);
      LIBNAME dep ODBC DSN = "&EXT_DBNAME" SCHEMA = "&EXT_SCHEMA";

       DATA mylib.&LOCAL_DSN;
          SET dep.&EXT_DSN;
       RUN;
   %MEND GET_DAT;

/*EP.END*/

/*EP.SINGLE*/
   %GET_DAT(<DBNAME_A>, <SCHEMA_NAME_A>, tableA, tableA);
/*EP.END*/

/*EP.SINGLE*/
   %GET_DAT(<DBNAME_B>, <SCHEMA_NAME_B>, tableB, tableB);
/*EP.END*/

/*EP.SINGLE*/
   %GET_DAT(<DBNAME_C>, <SCHEMA_NAME_C>, tableC, tableC);
/*EP.END*/
```

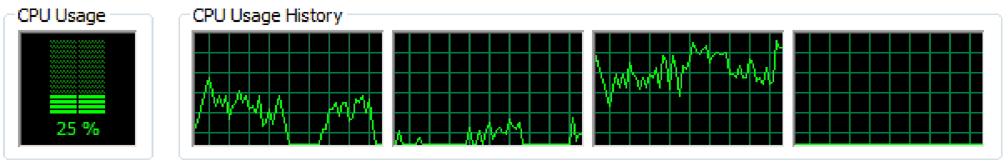
GEEK-OUT: MASS PARALLEL EXECUTION [PLEASE SKIP IF NOT PRONE TO TECHNICALITIES]

What if you would like to import external data from fifty different departments? Again, it would be depressingly tedious to copy/paste fifty instances of /*EP.SINGLE*/ %MY_IMPORT(…) /*EP.END*/. One way to go about this is to create a SAS® data set with the fifty values of the variables EXT_DBNAME, EXT_SCHEMA, EXT_DSN and LOCAL_DSN. then run through this data set using a data _NULL_ step, to create a SAS® program dynamically using the "file" and "put" statements. After getting used to this way of working, this can easily become one of the preferred ways to use RunParallel.

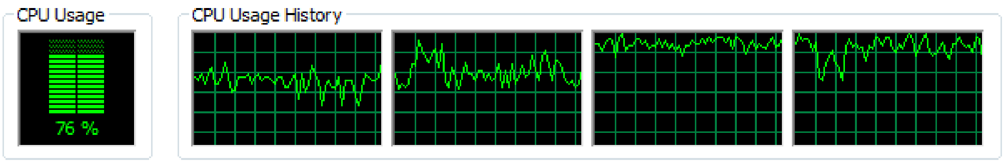
DETAILS THAT MATTER

THE SAS® LANGUAGE IS IDEAL FOR PARALLEL PROGRAMMING

Most basic SAS® programs consist of two components: PROC/DATA steps and global statements (e.g. "LIBNAME"). In each step, we read from data sources and write data to disk. In this respect SAS® is different from other languages, in the sense the beginning or end of a step creates a very natural breakpoint for parallelization. When we run code directly the old fashioned way in a single Base SAS® session, not all computational resources are used:



However, using RunParallel to start up three parallel sessions, performance improves as more CPU is used:



As seen above, starting multiple sessions allows us to take better advantage of the computational resources.

IT WORKS BOTH IN SERIAL AND IN PARALLEL

One of the most annoying aspects of working with parallel programming has traditionally been the fact that the whole development process has to be planned around a parallel coding structure. We wish to change this paradigm by making the parallel coding part the least intrusive possible, which is why RunParallel commands are wrapped inside SAS comments. This way, code can be developed as serial code, and even after RunParallel commands have been entered, the code can still be run inside a single Base SAS® session in the typical F3 fashion.

OUR DEFINITION OF PERFORMANCE BOOST

What do we mean by boosting performance? Traditionally, people like to talk about code that "runs a whopping 20% quicker on average" – for us, performance is boosted if we stand to win thirty minutes by making a one-minute change in the code, even if that only means the code runs 5% faster. We measure performance gain by measuring the gain in runtime against the loss in programming time.

AUTOMATIC SYNCING AND "SESSION 1"

Another hassle about parallel programming we seek to eliminate is the constant need for synchronization. A lot of synchronization in RunParallel is done automatically, in the sense that RunParallel synchronizes whenever it runs into code that is not wrapped in RunParallel commands. In the "Sort and Merge" example, the MERGE DATA step is an unwrapped standalone STEP, hence RunParallel waits for everything before this step to finish executing before executing this STEP. By default, all standalone code is executed on session **1**.

LOGS WHILE YOU RUN

In normal Base SAS® sessions, things have a tendency to “freeze” during code execution. In the RunParallel interface, you may switch smoothly between session logs and scroll up and down while the code is executing. This is important for two reasons. Firstly, it allows for quick scrutiny of whether something has gone wrong. Secondly, it is a fine way of keeping track of how far the code has progressed.

KEEP IT STABLE BY CHOOSING HOW MANY SAS SESSIONS YOU START UP

It is possible to choose the number of SAS® sessions that are used both in the user interface and through code. This allows for stable use of RunParallel, in particular if you would like to use the computer for something else while the code is running. Coding the option into your program makes it possible to vary the number of sessions you use throughout the execution of a single program.

THE USE OF MANUAL SYNCHRONIZATION

We have introduced the use of manual synchronization to keep the process of parallelizing the least intrusive possible. Let us consider an example where we import .csv data from two different sources. If we also want to sort the data after importing, the code could be written in two ways:

```
LIBNAME mylib '[insert your
folder name here]';

proc import
datafile="\\PATH_TO_DEP_A\tableA.
csv" out=mylib.tableA dbms=csv;
run;

proc sort data = mylib.tableA;
   by var1 var2;
run

proc import
datafile="\\PATH_TO_DEP_B\tableB.
csv" out=mylib.tableB dbms=csv;
run;

proc sort data = mylib.tableB;
   by var1 var2;
run
```

```
LIBNAME mylib '[insert your folder
name here]';

proc import
datafile="\\PATH_TO_DEP_A\tableA.c
sv" out=mylib.tableA dbms=csv;
run;

proc import
datafile="\\PATH_TO_DEP_B\tableB.c
sv" out=mylib.tableB dbms=csv;
run;

proc sort data = mylib.tableA;
   by var1 var2;
run

proc sort data = mylib.tableB;
   by var1 var2;
run
```

In the first case, it would be quite simple to wrap the bold block in /*EP.SINGLE*/ .. /*EP.END*/ and the following block in /*EP.SINGLE*/…/*EP.END*/. In the second case however, the first SORT STEP must wait for the first IMPORT to execute, and the second SORT STEP must wait for the second IMPORT to execute, so naively wrapping each step in /*EP.SINGLE*/…/*EP.END*/ would result in a mess. However, moving the code around is not always desirable (e.g. if you are only maintaining the code). This is where manual synchronization comes in. By typing /*EP.SYNC*/ after the second IMPORT STEP, we insert a manual stop light that let’s RunParallel wait for the first two IMPORT steps to finish executing, before it continues on.

PART II: BUT HOW DO IT KNOW?

The title of this section is a nerdy homage to J. Clark Scott's standard textbook on computer principles, detailing the inner workings of the miraculous black box. From this point forward, we move into technical territory. In the following sections, we present a walkthrough of how you can construct a plug-in such as RunParallel for Base SAS®, and make it latch on properly as a menu item in the menu bar.

INTRODUCTION

Base SAS® is the primary integrated development environment (IDE) offered by SAS® Institute for creating statistical applications. In general, a development environment is an integrated collection of programming tools. A programming tool is a piece of software or hardware used to support a well-defined task during the program development process.

The idea of the integrated development environment (IDE) is central in the history of the software industry. Common for almost all modern integrated development environments is tight integration with programming language, existence of advanced editors, facilities for managing configuration, build and deployment issues, profiling tools, support for debugging, and much more. The list of all modern IDEs is exhaustive, and their quality and supported features vary to a great extent.

Base SAS® is a closed, non-modular system, whose design does not allow extension by third party components. It does not provide built-in mechanisms for discovering, integrating, and running external software components, also known as **plug-ins**. This is in contrast to the modern development environments built around modular, plug-in principles that make it possible to extend their Graphical User Interface with new functionality. However, even if Base SAS® does not expose well-defined interfaces that enable expansion of its Graphical User Interface, it is still possible to do so using advanced Win32 Application Programming Interface (Win32 API) and Component Object Model technology. In order to do that, we need to address several issues:

1. How to load dynamic link libraries (DLL) that implement plug-in functionality into the address space of Base SAS®?
2. How to get Base SAS® to call methods exported by plug-in?
3. How to connect the "Click" event of menu items added by plug-in with their respective event handlers?
4. How to collect the content of Base SAS® Windows and Views (Program editor, Log, etc.) and pass them to the plug-in for further processing?
5. How to implement plug-in functionality?
6. How to present plug-in output to a SAS® programmer?

In the following, we will try to give answers to these questions by taking a closer look at the implementation details of the RunParallel plug-in.

THE OVERALL DESIGN OF RUNPARALLEL

The RunParallel plug-in comprises two dynamic link libraries (DLL) – SasEx.dll and SasExGui.dll.

SasEx.dll is a native Windows DLL implemented in the C programming language. It loads into the address space of Base SAS® and executes in the same way as other native DLLs required for proper functioning of Base SAS®. It contains code that inserts new menu items ("Add-ins", "RunParallel", etc.) into the menu bar of Base SAS®. In addition, SasEx.dll intercepts "Click" events generated by newly added menu items, reads the content of Base SAS® editors and views, and makes it ready for further processing that takes place in SasExGui.dll.SasEx.dll exports a single method - InitSas. Base SAS® calls this method in the course of execution of autoexec.sas.

```
filename sascbtbl 'C:\Program Files\SAS 9.4\SASFoundation\9.4\sasex.dat';

data InitSas;
   call module('*e','InitSas');
run;
```

The InitSas method is invoked using the CALL MODULE routine. In short, the CALL MODULE routine executes a routine that resides in an external library with the specified arguments. The routine details and argument attributes of InitSas are described in the file sasex.dat.

```
routine InitSas minarg=0 maxarg=0 module=SasEx;
```

CALL MODULE is a well-documented SAS® routine described in detail in the SAS® Language Reference.

SasExGui.dll is a .NET dynamic link library implemented in the C# programming language. It implements the BaseSasEx interface that contains ExecuteParallel function:

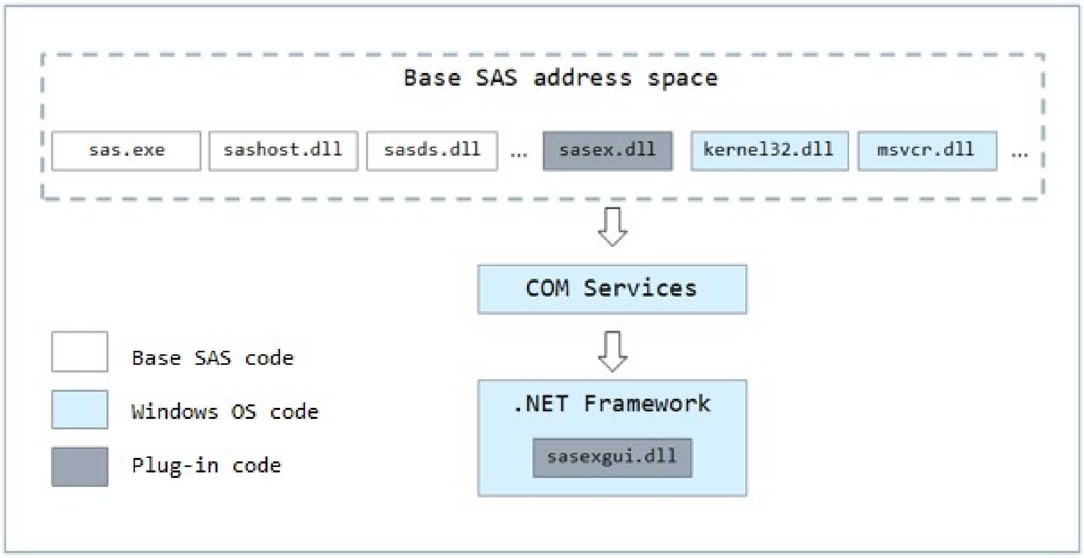
```
public interface IBaseSasEx
{
   ...
   bool ExecuteParallel();
}
```

The ExecuteParallel method is invoked from native SasEx.dll through Microsoft COM infrastructure. The method reads text from the SAS® Program Editor, creates a RunParallel window, and hands over control to it.

In order to be visible to Base SAS®, SasExGui.dll has to be registered for COM interoperability using an auto-generated type library SasExGui.tlb.

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\RegAsm.exe "$(TargetPath)" /tlb:
"C:\Program Files\SAS\SASFoundation\9.2\$(TargetName).tlb" /codebase
```

As it can be seen from the figure below, RunParallel executes partially as native code mapped in the address space of Base SAS®, and partially as managed code in the context of the .NET Framework.



IMPLEMENTATION OF SASEX.DLL

As explained in the previous section, **SasEx.dll** is a native Windows DLL that exports a single function named InitSas. This function contains a dozen program lines, of which the most important are:

```
extern "C" __declspec(dllexport) void InitSas();


void InitSas()
{
   ...
   GetModuleHandleEx(GET_MODULE_HANDLE_EX_FLAG_PIN, L"SasEx.dll", &hLib);

   DWORD updateMenuThreadId;
   CreateThread(0, 0, UpdateMenuThread, NULL, 0, &updateMenuThreadId);

   DWORD hookHandlerThreadId;
   CreateThread(0, 0, HookThread, NULL, 0, &hookHandlerThreadId);
}
```

The first line is call a to the Win32 API function GetModuleHandleEx. It's only argument is GET_MODULE_HANDLE_EX_FLAG_PIN. By calling this function, we ensure that SasEx.dll stays loaded until Base SAS terminates, no matter how many times FreeLibrary is called. This is important because Base SAS will try to unload **SasEx.dll** immediately after the call to InitSas returns. GET_MODULE_HANDLE_EX_FLAG_PIN effectively prevents **SasEx.dll** from being unloaded even if the reference counter becomes zero.

The rest of the InitSas method is rather straightforward - it creates two threads. The first thread is UpdateMenuThread. The purpose of this thread is to ensure that the RunParallel menu item always stays inserted in the menu bar of Base SAS®. The second thread sets a Windows hook. This hook makes it possible to intercept and process messages that RunParallel menu item generate in response to user actions.

UPDATEMENUTHREAD

UpdateMenuThread tests periodically whether plug-in menu items are included in the Base SAS® menu bar. If it discovers that this is no longer the case, it will insert them again. The problem is that Base SAS® deletes and repaints its menu bar every time one of its child windows gain focus. It inserts only those menus that make sense for the window that is currently in focus. For example, the Run menu item is shown only if the Program Editor is in focus. Since Base SAS® is not aware of the existence of plug-in menu items, it will simply delete them every time the menu bar is repainted. When it happens, UpdateMenuThread will insert them again ensuring that they are always presented in the menu bar of Base SAS®.

In order to insert plug-in menu items to the Base SAS® menu bar, we need to get a handle of the top-level window of Base SAS®. This window owns the menu bar, and can be easily obtained by using a couple of Win32 API functions:

1. First enumerate top-level windows of Base SAS® using the GetWindowThreadProcessId function:

```
vector<HWND> GetToplevelWindows()
{
   EnumWindowsCallbackArgs args(GetCurrentProcessId());

   if (EnumWindows(&EnumWindowsCallback, (LPARAM)&args) == FALSE)
   {
      return vector<HWND>();
   }

   return args.handles;
}
```

```
static BOOL CALLBACK EnumWindowsCallback(HWND hnd, LPARAM lParam)
{
   EnumWindowsCallbackArgs *args = (EnumWindowsCallbackArgs *)lParam;

   DWORD windowPID;

   GetWindowThreadProcessId(hnd, &windowPID);

   if (windowPID == args->pid)
   {
      args->handles.push_back(hnd);
   }

   return TRUE;
}

struct EnumWindowsCallbackArgs
{
   const DWORD pid;
   vector<HWND> handles;

   EnumWindowsCallbackArgs(DWORD p) : pid(p)
   {

   }
}
```

2. Then read the text in the title bar of each top-level window. If the text starts with “SAS”, try to get its menu handle. If this succeeds, use that handle to insert plug-in menu items:

```
vector<HWND> handles = GetToplevelWindows();

for (vector<HWND>::size_type i = 0; i != handles.size(); i++)
{
   WCHAR windowText[1024];

   int length = GetWindowText(handles[i], windowText, 1024);

   if (length != 0)
   {
      if (wcsncmp(windowText, L"SAS", 3) == 0)
      {
         HMENU hMenu = GetMenu(handles[i]);

         if (hMenu != NULL)
         {
            // Add plug-in menu items using hMenu handle
         }
      }
   }
}
```

Once we obtain the handle of Base SAS® menu bar, we can now insert RunParallel menu item.

1. Insert RunParallel submenu:

```
HMENU hAddInsSubMenu = CreatePopupMenu();

memset(&mii, 0, sizeof(MENUITEMINFO));

mii.cbSize = (UINT)   sizeof(MENUITEMINFO);
mii.fMask = (UINT)MIIM_STATE | MIIM_STRING | MIIM_ID | MIIM_BITMAP;
mii.fType = (UINT)MIIM_STRING;
mii.fState = (UINT)MFS_ENABLED;
mii.wID = UM_RUN_PARALLEL;
mii.hbmpItem = hBitmapRunParallel;
mii.dwTypeData = (LPTSTR)runParallel;
mii.cch = (UINT)lstrlenW(runParallel);

InsertMenuItem(hAddInsSubMenu, 0, TRUE, (LPCMENUITEMINFO)&mii);
```

2. Insert Add-ins menu and draw menu bar:

```
memset(&mii, 0, sizeof(MENUITEMINFO));
mii.cbSize = (UINT)   sizeof(MENUITEMINFO);
mii.fMask = MIIM_SUBMENU | MIIM_STATE | MIIM_STRING;
mii.fState = (UINT)MFS_ENABLED;
mii.hSubMenu = hAddInsSubMenu;
mii.dwTypeData = (LPTSTR)addIns;
mii.cch = (UINT)lstrlenW(addIns);

InsertMenuItem(hMenu, GetMenuItemCount(hMenu), TRUE, &mii);

DrawMenuBar(handles[i]);
```

HOOKTHREAD

`HookThread` installs an application-defined hook procedure into a hook chain. A hook is a mechanism by which a function can intercept Windows® messages and act on them before they reach an application. Functions that receive messages are called filter functions.

`HookThread` makes it possible to handle messages generated by the `RunParallel` menu item and internal plug-in messages:

```
#define UM_RUN_PARALLEL 1303
#define UM_SHOW_EXTENSION_WINDOW 1304
#define UM_SHOW_EMPTY_EDITOR_ERROR 1305
#define UM_SHOW_SELECT_EDITOR_ERROR 1306
```

The `UM_RUN_PARALLEL` message is generated when the user selects the `RunParallel` menu item. The other messages are used internally by the plug-in.

Setting a hook is rather straightforward – it only requires a call to the Win32 API function `SetWindowsHookEx`. In order to be able to intercept messages generated by the `RunParallel` menu item, the thread that sets the Windows hook has to be transformed into the GUI thread. This is done by creating a message loop. In short, a message loop is the programming structure that continually tests for external events and calls the appropriate routines to handle them. A simple message loop consists of one function call to each of these three Win32 API functions: `GetMessage`, `TranslateMessage`, and `DispatchMessage`.

```
DWORD WINAPI HookThreadGetMessage(LPVOID lpParameter)
{
   hHookGetMessage = SetWindowsHookEx(WH_GETMESSAGE, HookHandlerGetMessage,
                                      hLib, guiThreadId);

   if (hHookGetMessage == NULL)
   {
      return 0;
   }

   hHookCallWndProc = SetWindowsHookEx(WH_CALLWNDPROC, HookHandlerCallWndProc,
                                       hLib, guiThreadId);

   if (hHookCallWndProc == NULL)
   {
      UnhookWindowsHookEx(hHookGetMessage);
      return 0;
   }

   MSG message;

   while (GetMessage(&message, NULL, 0, 0) != 0)
   {
      TranslateMessage(&message);
      DispatchMessage(&message);
   }

   return 0;
}
```

The first `SetWindowsHookEx` function takes `HookHandlerGetMessage` as its second argument. This is the name of the filter function that will handle messages passed through `HookThread`. Since we are only interested in messages generated by our own plug-in, all other messages will be passed to the `windows proc of Base SAS`® without any further processing.

```
LRESULT CALLBACK HookHandlerGetMessage(int nCode, WPARAM wParam, LPARAM lParam)
{
   if (nCode >= 0)
   {
      ...
      const MSG* pMsg = (MSG*)lParam;

      if (pMsg->message == WM_COMMAND)
      {
         hSourceWindow = NULL;
         ...
         if (LOWORD(pMsg->wParam) == UM_RUN_PARALLEL)
         {
            currentMessage = UM_RUN_PARALLEL;

            EmptyClipboardEx();

            HWND hWindowInFocus = GetFocus();

            if (hWindowInFocus != NULL)
            {
               WCHAR classNameText[1024];

               GetClassName(hWindowInFocus, classNameText, 1024);

               if (wcslen(classNameText) != 0 && wcsncmp(classNameText,
                   L"AfxFrameOrView80u", 17) == 0)
               {
                   hSourceWindow = hWindowInFocus;
               }
               else
               {
                  PostThreadMessage(GetCurrentThreadId(),
                                    UM_SHOW_SELECT_EDITOR_ERROR, 0, 0);
               }
            }
         }

         if (hSourceWindow != NULL)
         {
             SendMessage(hSourceWindow, WM_LBUTTONDOWN, 0, MAKELPARAM(2, 2));
            SendMessage(hSourceWindow, WM_LBUTTONUP, 0, MAKELPARAM(2, 2));

            terminateSelectAndCopyLogTextThread = false;

            selectAndCopyLogTextSyncObject = CreateEvent(0, TRUE, TRUE, 0);

            guiThreadId = GetCurrentThreadId();

            CreateThread(0, 0, SelectAndCopyLogTextThread, NULL, 0,
                         &selectAndCopyLogTextThreadId);
         }
         ...
      }
   }
}
```

When the `plug-in` intercepts the message generated by the `RunParallel` menu item, the next step is to get the content of the Program Editor Window. This was one of the biggest challenges we had to deal with while working on this project. As described in the introductory part of this section, Base SAS® does not provide any interface that gives access to its internal components. Base SAS® does not expose interfaces that external components can use in order to get content of Program Editor Window.

But even if it is not possible to read the Program Editor Window directly, it is possible to do it in an indirect fashion. The key is to take a copy of the Program Editor Window using the copy functionality provided by Base SAS®, and save it to Windows Clipboard. Once in Windows Clipboard, the content of the Program Editor Window can easily be read and processed by any external component.

`HookHandler` saves the content of the Program Editor Window to Windows Clipboard, and then invokes the `ExecuteParallel` method exported by `BaseSasGuiEx` dll. The invoked method then reads Windows Clipboard, pastes the SAS code into its own editor, and makes it ready to run in parallel.

1. Set Program Editor Window in focus by sending WM_LBUTTONDOWN followed by WM_LBUTTONUP Windows messages.

```
...
SendMessage(hSourceWindow, WM_LBUTTONDOWN, 0, MAKELPARAM(2, 2));
SendMessage(hSourceWindow, WM_LBUTTONUP, 0, MAKELPARAM(2, 2));

terminateSelectAndCopyTextThread = false;

selectAndCopyTextSyncObject = CreateEvent(0, TRUE, TRUE, 0);

guiThreadId = GetCurrentThreadId();

CreateThread(0, 0, SelectAndCopyTextThread, NULL, 0,
             selectAndCopyTextThreadId);
...
```

2. Select the content of the Program Editor Window and copy it into Windows Clipboard by generating `CTRL+A` and `CTRL+C` keystrokes. Finally, call the Win32 API function `PostThreadMessage` using `UM_SHOW_EXTENSION_WINDOW` as argument .

```
...
SendCtrlPlusKeyInput(0x41);
SendCtrlPlusKeyInput(0x43);

PostThreadMessage(guiThreadId, UM_SHOW_EXTENSION_WINDOW, 0, 0);

WaitForSingleObject(selectAndCopyTextSyncObject, INFINITE);
...
```

```
BOOL SendCtrlPlusKeyInput(INT virtualKey)
{
   int key_count = 4;

   INPUT* input = new INPUT[key_count];
   for (int i = 0; i < key_count; i++)
   {
      input[i].ki.dwFlags = 0;
      input[i].type = INPUT_KEYBOARD;
   }

   input[0].ki.wVk = VK_CONTROL;
   input[0].ki.wScan = MapVirtualKey(VK_CONTROL, MAPVK_VK_TO_VSC);
   input[1].ki.wVk = virtualKey;
   input[1].ki.wScan = MapVirtualKey(virtualKey, MAPVK_VK_TO_VSC);
   input[2].ki.dwFlags = KEYEVENTF_KEYUP;
   input[2].ki.wVk = input[1].ki.wVk;
   input[2].ki.wScan = input[1].ki.wScan;
   input[3].ki.dwFlags = KEYEVENTF_KEYUP;
   input[3].ki.wVk = input[0].ki.wVk;
   input[3].ki.wScan = input[0].ki.wScan;

   return SendInput(key_count, (LPINPUT)input, sizeof(INPUT));
}
```

3. Check whether the content of the Program Editor Window is ready to be read from Windows Clipboard. If this is not the case, SelectAndCopyTextThread will again generate CTRL+A and CTRL+C keystrokes. Otherwise, the control is passed to SasExGui.dll.

```
if (pMsg->message == UM_SHOW_EXTENSION_WINDOW)
{
   if (GetClipboardSize() != 0)
   {
     ...
      CoInitialize(NULL);

      IBaseSasExPtr pBaseSasEx(__uuidof(BaseSasExImpl));

      VARIANT_BOOL res;
      InterlockedExchangeAcquire(&extensionWindowShown, 1);

      if (currentMessage == UM_RUN_PARALLEL)
      {
         pBaseSasEx->ExecuteParallel(&res);
         SendKeyInput(0x27);
      }
     ...
      pBaseSasEx->Release();
      CoUninitialize();

      InterlockedExchangeAcquire(&extensionWindowShown, 0);
     ...
   }
}
```

CONCLUSION

Extending the Base SAS® Graphical User Interface is not a trivial task and requires expert level knowledge of Win32 API and Microsoft COM technology. Base SAS can be improved in many ways, and RunParallel is just one example of how to do it.

The core extension functionality implemented in `SasEx.dll` can be reused to a great extent if new menus need to be added to the Base SAS® menu bar. It is not difficult to imagine new extensions that add improvements to different parts of Base SAS® including the Program Editor, Log, Output and Explorer windows.

In an ideal world, a whole new marketplace could be created offering new extensions and improvements to Base SAS®.

In the current state of hardware development, RunParallel extends the capabilities of SAS® to squeeze as much computational juice as possible out of the multicore architecture of modern workstations. There are certainly a number of cases where RunParallel excels, such as reading external data from multiple separate data sources, or CPU-intensive DATA/PROC steps with little writing to disk. These might include sorting broad datasets on many variables, and computationally intensive simulations. However, even in the case where RunParallel only yields a small performance gain, the intuitiveness of RunParallel allows for quick parallelization of code, while keeping the original functionality of the code. Thus in most cases, the relative gain in performance greatly outweighs the few minutes or seconds it takes to add RunParallel commands.

REFERENCES

Kernighan, B.W. and Ritchie, D.M. 1998. *The C programming language*. Second edition. Prentice Hall.

Richter, Jeffrey. 1999. *Programming application for Microsoft Windows platform*. Fourth edition, Microsoft Press.

Russinovich, M.E. and Solomon, D.A. 2005. *Microsoft Windows Internals*. Fourth edition, Microsoft Press.

Box, Don. 1998. *Essential COM*. Addison-Wesley.

Schildt, Herbert. 2002. *C#: The complete reference.* McGraw-Hill.

CONTACT INFORMATION

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

Jingyu She
Danica Pension
+45 45131261
jish@danicapension.dk
http://www.danicapension.dk/en-dk/

Tomislav Kajinic
Danica Pension
+45 45131293
kaji@danicapension.dk
http://www.danicapension.dk/en-dk/