A Generic Method of Parallel Processing in Base SAS® 8 and 9
Sassoon Kosian, Inductis, New Providence, NJ

ABSTRACT
When you work with large data sets and large volumes of SAS® processes, leveraging multiple CPUs to run parts of a SAS job in parallel threads can significantly increase the speed of performance. There are several ways to multi-thread jobs in SAS, for example, you can use threaded procedures in SAS®9 or run parallel jobs by using MP Connect in SAS/CONNECT® 8 or SAS/CONNECT® 9. However, in many scenarios, you cannot take advantage of these methods.

This paper suggests a generic parallelization technique that can be used as a work-around solution when other methods cannot be used. This parallelization technique works in SAS®8 and SAS®9, and only Base SAS® software is needed. This technique is based on a mechanism that spawns multiple jobs from a single instance of code, and, depending on how your SAS code is structured, either the whole process or parts of the process can be run in parallel.

By using the technique that is explained in this paper, you can significantly increase performance time. However, this result also is affected by the number of CPUs that are available. This technique applies to more types of SAS code than the threaded procedures in SAS®9 and the use of MP Connect in SAS/CONNECT® 8 or SAS/CONNECT® 9. More importantly, the DATA step can be multi-threaded.

INTRODUCTION
Parallel processing or multi-threading a SAS job is a mode of execution in which two or more portions of a job run in parallel streams to accelerate processing and reduce the real time to completion. Performance gain is achieved by simultaneously using multiple processors which can benefit from added CPU power as well as multiple I/O channels. For this reason multi-threading is more meaningful when the hardware has multiple CPUs such as a Unix machine or a multi-processor Windows server.

The case scenario discussed in this paper is specifically for a Unix machine but the logic can be applied to Windows with some modifications for operating system differences. In addition, the method explained herein is most appropriate for CPU bound jobs and would have minimal effect on I/O or memory bound jobs.

OUT-OF-THE BOX MULTI-THREADING METHODS
SAS® offers two principal methods of multi-threading for CPU bound jobs: threaded procedures in SAS®9, and MP Connect facility in SAS/CONNECT® 8 and SAS/CONNECT® 9. Both are effective and useful methods. However, these methods have limitations in application. To take advantage of them certain conditions have to be in place.

THREADED PROCEDURES
Threaded procedures in SAS®9 were introduced to enable parallel processing. Multi-threading a qualifying procedure is only a matter of adding the option THREADS to the code. Unfortunately only a handful of procedures are threaded to date (e.g., PROC SORT, PROC REG, PROC SQL, etc.), and DATA step obviously cannot be multi-threaded because it is not a procedure.

MP CONNECT
MP Connect is a multi-processing facility in SAS/CONNECT® 8 and SAS/CONNECT® 9. One can use MP Connect to run independent steps of a SAS job in parallel streams. But in order to take advantage of this method at least two independent steps must exist in the code. Even so, having

---

1 A job is CPU bound when overall performance is limited because of a lack of processing power while I/O and memory resources are utilized below capacity.
many more multiple CPUs than independent steps leaves much of available CPU power underutilized. Another disadvantage of MP Connect has to do with the lack of flexibility in using macro variables between the main SAS session and MP Connect parallel tasks.

As we can see these methods are useful only in certain qualifying conditions. But what if your job does not qualify for MP Connect or threaded procedures? What if the Unix machine you are working on still uses SAS®8, or does not have SAS/CONNECT®?

There is an alternative solution that does not require MP Connect or threaded procedures to boost SAS performance.

VIRTUAL MULTI-THREADING

Suppose we have a very large input dataset, and a large chunk of code that sequentially executes and produces an output dataset. One path that some programmers take is to run multiple instances of the same job concurrently on smaller subsets of the input dataset. They monitor the concurrent jobs to completion and aggregate the resulting individual output datasets. Thus the same output dataset is created but only faster. The cost the programmer pays is writing additional code and executing and monitoring many secondary jobs.

This process is a simple yet effective parallel processing approach. Typically, in parallel processing the parallel processes run behind the scenes. That is not the case with this approach yet it works as effectively because it takes advantage of multiple processors and utilizes them concurrently much the same way. Though the user does not assign individual CPUs to each process the operating system does so behind the scenes.

This manual multi-threading approach has its own drawbacks

- The user has to maintain many codes instead of one. If there are changes to the program the user must change all instances individually
- The user has to keep an eye on parallel running jobs so that time does not lapse between the completion of parallel jobs and the creation of an aggregate output. Because the purpose of parallel processing is to accelerate the overall process rather than individual components it would not be very useful if the parallel jobs ran fast yet the final output dataset was created with a delay. This would be even more problematic if the concurrent jobs finished off hours at different server load times the user may need to use different numbers for parallel jobs. To adjust the number of splits/threads the user will have to manually update the splitting and aggregating codes

For these reasons manual multi-threading may be less desirable or even infeasible depending on business requirements and expected turnaround time.

AUTOMATED VIRTUAL MULTI-THREADING

We propose a technique to automate the manual multi-threading method. The technique allows the user to maintain and execute a single copy of the code even though in the process many codes will be generated and executed. Once the code is submitted, splitting the input dataset, generation of multiple instances of the code, their execution and aggregation of output datasets all follow automatically. Because of automation there is no need to monitor spawned jobs.

There are five steps to add automated multi-threading functionality to an existing job. For simplicity of illustration let us assume that the code we will multi-thread consists of one CPU intensive data step. Typically CPU intensive data steps contain hundreds or thousands of lines of code. But for illustration we will only display a few lines.
ORIGINAL CODE
We would like to accelerate the following program by adding parallel processing functionality.

```
libname in_lib "/home/skosia/indata/"; ** Input library **;
libname out_lib "/home/skosia/outdata/"; ** Output library **;

data out_lib.output;
set in_lib.raw_data;
  costgr2=sum(carm_c,carecost);
  rev_nocontra=sum(usf,usage_rev_reprice,tot_local_fee,tot_local_MRC,
                 tot_ld_fees,tot_ld_MRC,non_toll,non_recur);
  rev_contra=sum(usf,usage_rev_reprice,tot_local_fee,tot_local_MRC,
                 tot_ld_fees,tot_ld_MRC,non_toll,non_recur,marketinc);
  profit3=sum(rev_nocontra,-access_tot,-costgr2,-net_bad_debt_tot,
              -interconnect_tot,-local_con_cost);
<   ……………………….>
many more lines of code
           ..................>
run;
```

MULTI-THREADED CODE
The code below is the conversion of the original code into an automated multi-threaded program. Statements that are specific to the multi-threading logic appear in UPPER CASE for ease of distinguishing from the original code.

```
%let dsn = raw_data;                         ** Input dataset **;
libname in_lib "/home/skosia/indata/";       ** Input library **;
libname out_lib "/home/skosia/outdata/";     ** Output library **;

%LET JOBNAME = MYJOB ;
*** Job name must be a unique name and should NOT have extensions **;
%LET NUM_PARTITIONS = 10; * Number of threads;

********** Randomly split the input dataset into equal subsets      ****;
%MACRO RANDOM_SPLIT(INLIB=,OUTLIB=,DSN=,NUM_PIECES=);
  DATA
    %DO K=1 %TO &NUM_PIECES;
    &OUTLIB..&DSN._&K
    %END;;
  SET &INLIB..&DSN;
  RANDOM_NUMBER=RANUNI(0);
  %DO K=1 %TO &NUM_PIECES;
  %IF &K NE 1 %THEN ELSE; IF (&K-1)/&NUM_PIECES <= RANDOM_NUMBER <
  &K/&NUM_PIECES THEN OUTPUT &OUTLIB..&DSN._&K;
  %END;
  DROP RANDOM_NUMBER;
RUN;
%MEND RANDOM_SPLIT;
%RANDOM_SPLIT(INLIB=IN_LIB,OUTLIB=LIB,DSN=&DSN,NUM_PIECES=&NUM_PARTITIONS);
```
OPTIONS OBS=0 NOSYNTAXCHECK FULLSTIMER SOURCE2;
*** Direct unnecessary portions of the log out of the main log file ***;
PROC PRINTTO LOG="./GARBAGE.LOG" NEW; RUN;
%MACRO PARTITION;

%DO PART_NUM=1 %TO &NUM_PARTITIONS;

FILENAME MPRINT "/&JOBNAME._PART&PART_NUM..SAS" LRECL=170;
OPTIONS MPRINT MFILE;

%MACRO CODE;

OPTIONS FULLSTIMER SOURCE2;
libname out_lib "/home/skosia/outdata/";

DATA LIB.OUTPUT_&JOBNAME._PART&PART_NUM;
SET LIB.&DSN._&PART_NUM;
   costgr2=sum(carm_c,carecost);
   rev_nocontra=sum(usf,usage_rev_reprice,tot_local_fee,tot_local_MRC,
      tot_ld_fees,tot_ld_MRC,non_toll,non_recur);
   rev_contra=sum(usf,usage_rev_reprice,tot_local_fee,tot_local_MRC,
      tot_ld_fees,tot_ld_MRC,non_toll,non_recur,marketinc);
   profit3=sum(rev_nocontra,-access_tot,-costgr2,-net_bad_debt_tot,
      -interconnect_tot,-local_con_cost);
   <  ......................
   many more lines of code
   ........................>
run;

%MEND CODE;

%CODE
%END; ** num_partitions;

*** Redirect the log stream back to the main log file ***;
PROC PRINTTO LOG=LOG;RUN;
OPTIONS NOMFILE;

*** Create a shell script ***;
DATA _NULL_;  
   FILE "RUN_&JOBNAME..SH";
%DO PART_NUM=1 %TO &NUM_PARTITIONS;
   PUT "SAS &JOBNAME._PART&PART_NUM..SAS &"
      "WAIT";
%END;
RUN;
%MEND PARTITION;

*** Execute the shell script ***;
X "CHMOD 755 RUN_&JOBNAME..SH";
X "RUN_&JOBNAME..SH";

*** Aggregate partitioned output datasets into one ***;
OPTIONS OBS=MAX PS=MAX NOCENTER;

%MACRO AGGREGATE;
  DATA OUT_LIB.OUTPUT;
  SET %DO PART_NUM = 1 %TO &NUM_PARTITIONS;
  LIB.OUTPUT_&JOBNAME._PART&PART_NUM
  %END;;
  RUN;
%MEND AGGREGATE;

%AGGREGATE

STEP 1
The code begins with randomly partitioning the input dataset into multiple smaller pieces with equal size. The number of partitions is a configurable macro variable. It corresponds to the number of jobs we plan to run in parallel. This parameter allows the code to be flexible as to how many parallel threads are to be triggered.

It is important to split the input randomly so that the subsets have the same distribution of data. Having the same distribution assures that processing time is approximately the same for all partitions which in turn assures that the overall time to complete execution on all partitions is minimized. With unequal processing time it would not matter if execution on some partitions completed early. The overall execution time would be equal to the execution time of the longest process. Hence splitting with even size and distribution is the best way.

STEP 2
Macro %code contains a copy of the original data step statements. Macro %partition iterates for each thread and generates SAS codes by copying the original code with the use of MPRINT and MFILE macro options. These options allow to output statements within a macro to an external file with the file reference of MPRINT. In this case, the external files are cloned SAS codes, i.e. the original data step code with different input and output data set names.

Important note: In SAS®8 the MFILE option introduces hexadecimal characters when there are more than a dozen lines of code contained in the macro. This causes errors when the generated code is submitted. A workaround to this issue uses %INCLUDE in the cloned code that points to a second SAS code containing a large number of lines. Thus the number of lines in the macro %code itself is minimized to a few lines yet the secondary code calls a large number of code lines through %INCLUDE. %NRSTR masking function assures that %INCLUDE does not execute in the main code.

The following example demonstrates what the macro %code would look like in SAS®8. In SAS®9 this workaround is not needed though it would work similarly.

%MACRO CODE;

  OPTIONS FULLSTIMER SOURCE2;
  libname out_lib "~/home/skosia/outdata/";

  DATA LIB.OUTPUT_&JOBNAME._PART&PART_NUM;
  SET LIB.&DSN._&PART_NUM;
  %NRSTR(%INCLUDE "MANY_LINES_OF_DATA_STEP_STATMENTS.SAS");
  RUN;
%MEND CODE;
STEP 3
As macro `%partition` iterates and generates SAS codes, in the process it also executes the statements contained in macro `%code`. If we let it run as it is that would defeat the purpose of multi-threading. By creating datasets sequentially rather than concurrently the program would spend the same amount of time or more as it would with the original code. To avoid this outcome we set the option OBS=0 which makes the program momentarily iterate through all partitions and create dummy datasets with zero records. Thus we practically do not spend any processing time.

In addition, since we are not interested to see the log notes for this part of the code, we direct them to a different file using PROC PRINTTO. This helps us keep the log clean of unneeded information. When the codes are generated we redirect the log to the main stream and set OBS=MAX.

STEP 4
After `%partition` and `%code` macros have generated SAS codes for parallel threads, a shell script is generated and executed to run the SAS codes simultaneously in independent sessions. The shell script is invoked by using the x command (for more information please check SAS documentation on the x command). A key component of the shell script is the `wait` Unix statement which suspends processing in the main job until all threads are complete, then only execution returns to the main session.

STEP 5
When all threads have executed output datasets are created for each thread and replace the dummy datasets from the previous step. Macro `%aggregate` collects the output datasets and aggregates them into a single file. This is the final output dataset that we would get in the original job.

LIMITATIONS AND CAUTION
Not all types of SAS code or its components may be multi-threaded with the suggested method. Like MP Connect and Threaded procedures virtual multi-threading too requires qualification. Care must be exercised when planning to multi-thread an existing SAS job. As a general guidance, in order to correctly multi-thread any step in a SAS process it must be additive. DATA step has the advantage of being additive while many SAS procedures normally do not.

Technically, any SAS code placed inside macro `%code` in the illustrated example will be cloned and parallel processed. But multi-threading a non-additive process or a step may produce unexpected results and errors.

For example, if we try to multi-thread a PROC SORT step with the suggested method the resulting dataset will not be sorted because the aggregation of sorted datasets is not necessarily sorted. However, a small modification helps to produce a valid multi-threaded sort. Instead of aggregating sorted datasets sequentially we can interleave using the BY statement (use “BY sort_var;” after the SET statement). In SAS®9 we can simply use PROC SORT with the THREADS= option.

Another example where caution is required is a merge between two or more datasets. To correctly merge only one of the merged datasets should be partitioned, otherwise it will very likely result in unmatches.

We do not intend to suggest clear cut boundaries of when the generic method of virtual multi-threading is valid. The user is encouraged to take these limitations into account on a case by case basis and correctly assess whether or not each step in a process qualifies for virtual multi-threading. In the event a step cannot be multi-threaded exactly as in the illustrated code, it maybe possible to modify the multi-threading code like in the case of PROC SORT and DATA step merge examples and achieve a correct result. The key point is that the concept of virtual multi-threading may be used in many different ways.
CODE STRUCTURE

Depending on how the SAS code is structured, automated multi-threading can be applied to the entire job, to individual steps, or to groups of several steps. The illustrated example in this paper is a very simple one. In reality we often have programs with many steps of which some steps may qualify for multi-threading while others may not.

The illustration below shows more realistic code scenarios.

The recommended performance improvement strategy for multi-step SAS processes is the following:

- Multi-thread as many steps as possible
- Club adjacent steps together to minimize code overhead

In Scenario 1 only one step qualifies for virtual multi-threading, while in Scenario 2 there are different groups of steps that qualify. In Scenario 3 the entire job is multi-threaded. Scenario 3 takes the most advantage of multi-threading.

It is also recommended that whenever possible MP Connect and threaded procedures should be combined with virtual multi-threading. That would produce the best performance improvement. For example, use threaded PROC SORT to sort input variables, then use virtual multi-threading to merge them.

RESULTS

In our practice virtual multi-threading has produced up to 5-7 times acceleration for very large jobs. The method would probably have little or no effect on small jobs because of the coding overhead and the implicit splitting/aggregating overhead. The degree of performance improvement is also a function of
SAS code specifics, the number of threads/CPUs used and hardware specifics. As a general guidance the more steps we can multi-thread, the more performance improvement we can achieve.

Lastly, to help the user decide between alternative methods of parallel processing the comparison chart below highlights the differences between out of the box SAS methods and the proposed generic method.

<table>
<thead>
<tr>
<th>Available in Base SAS®8</th>
<th>Available in Base SAS®9</th>
<th>Can be used with an individual Data Step</th>
<th>Can be used with multiple Data Steps</th>
<th>Flexibility to use macro variables</th>
<th>Configurable number of threads</th>
<th>Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Very Simple</td>
</tr>
<tr>
<td>Available in Base SAS®8</td>
<td>No</td>
<td>(SAS Connect must be installed)</td>
<td>(SAS Connect must be installed)</td>
<td>No</td>
<td>Yes</td>
<td>Simple</td>
</tr>
<tr>
<td>Available in Base SAS®9</td>
<td>Yes</td>
<td>No</td>
<td>Only if data steps are independent</td>
<td>Yes</td>
<td>Yes</td>
<td>Not Simple</td>
</tr>
</tbody>
</table>

CONCLUSION

The generic method of virtual multi-threading is an effective alternative of achieving significant performance gains when other multi-threading methods cannot be used. Performance gain comes at the cost of writing and maintaining many lines of additional code. Rather than a packaged solution for all cases the method requires the user to carefully consider each step on a case by case basis. An important advantage of the generic method is that it can multi-thread the DATA step while out of the box methods do not have the ability to do so. At the same time many of commonly used SAS procedures are probably not good candidates for generic parallel processing though ultimately it depends on program context.

Our experience shows that for large datasets and large, CPU intensive codes the benefit of faster execution definitely outweighs the cost of coding overhead. This is particularly true for mission critical jobs when there is no other alternative than fast turnaround.

CONTACT INFORMATION

Sassoon Kosian
Inductis
571 Central Ave, Suite 105
New Providence, New Jersey 07974-1547
Work Phone: 732-470-5834
Fax: 501-634-3733
E-mail: skosian@inductis.com, skosian@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.