Paper 109-2010

Faster results by multi-threading data steps

Ian J. Ghent, Technical Consultant – SAS Canada, Ottawa, ON

ABSTRACT

Many SAS users run programs on systems with multiple processors. Starting with SAS9, certain intensive SAS procedures take can advantage of these multiple processors to reduce run-time, and make more effective use of system resources. Many users have long running data steps, which could also benefit from parallel execution. This paper discusses how, and when to multi-thread an intensive data step, along with all the considerations need to do so. Topics covered include partitioning, I/O parallelism and SPDE/SPDS, run-time tracking, inter-row dependencies, thread execution, and re-joining. An example parallel data step macro package will also be included.

INTRODUCTION

Have you ever come across complicated data steps which take large amounts of run-time? Do you run on SAS on a high powered server or workstation with multiple processors/cores? Although many SAS procedures take advantage of multiple processors, data step iterations only run on one processor, creating a bottleneck for complex SAS applications. Depending on the nature of your data step, you might be able to improve the run-time of those data steps by a significant amount.

You can achieve this speed by dividing your dataset into multiple separate pieces, enabling the data step to be process the rows in parallel batch SAS sessions, and rejoin the data after all the sessions have completed. There are however some limitations to this approach, so careful considerations must be made before multi-threading a data step.

TO MULTI-THREAD OR NOT MULTI-THREAD

CPU V.S. IO - BATTLE OF THE ELECTRONS

There are many facets to understand when, and if a mutli-threaded data step should be used. The first question is determining the if the bottleneck is CPU or I/O. Looking at the log, there will be a note that displays after every data step which tells you the run-time and the CPU time.

If the CPU time and real-time are not that far apart (CPU > ~ %25 of run-time), a large portion of the time spent on the data step is spent doing CPU bound computation. Likewize, if the percentage of CPU time to real-time is low(< ~ %25), your I/O system is the constraining factor. CPU bottlenecks are much simpler to alleviate using a multi-threaded approach then I/O bottlenecks.

If the constraint is on the I/O side, things become a lot more complicated. Unless you have a system already setup with multiple I/O channels/paths, or have SPDS/SPDE libraries already setup and running, there's little point in multi-threading you data step.

INTER-ROW DEPENDENCIES

Another consideration when looking to multi-thread a data step is inter-row dependencies. That is, any function or statement, which uses information from another row in your dataset. The three prime examples are the LAGx() function set, the retain statement, and the “first.” and “last.” operators. Because these functions/statements need information from previous rows, they pose challenges in partitioning the dataset.

LAG functions are relatively easy to deal with, as long as it is not using another lagged variable in their input.

There are also ways of dealing with retain statements, as long as they have a defined periodic reset criteria . If the retain statement does not have a defined reset criteria, which is can be used to partition the dataset, there's no way to multi-thread that data step. Also, if the and “first.”/”last.” operators are used, the dataset will need to be partitioned using those variables, so that distinct values remain together in the same partition.

DIVIDE AND CONQUER

There are a variety of techniques which you can use to divide your data into equal parts. The examples below assume a 10,000 row dataset with 4 threads.

HORIZONTAL PARTITIONING

Horizontal partitioning divides your dataset by row number. Each dataset contains exactly the same number of rows (+- very few rows). For example, thread 1 would process rows 1-2,500, thread 2 would process rows 2501-5,000, and so on.

The main advantage is that rejoining the data after processing is a snap, and the order of the data is automatically preserved. Also, by including a few rows from the adjoining threads turf, you can make use of LAGx() functions. The main drawback to this approach is, the order of the data can cause certain threads to run longer then others, because of increased operations (i.e. if clusters of rows contain missing data).

INTERLEAVED PARTITIONING

Interleaving divides your row numbers by the number of threads used and by using the mod function to determine which thread will processes the row. For example, thread 1 would process rows 1,5,9,13,..., thread 2 would process rows 2,6,10,14..., and so on.

The main advantage of this approach is that the order of the data does not matter, effectively guaranteeing even loads. The disadvantage is that the rejoin process is more complicated and time consuming, if retaining the original order of the data is necessary.

INDEXED PARTITIONING (CLUSTERED PARTITIONING)

Indexed partitioning is useful when you have a I/O bottlenecks on an SPDE/SPDS dataset. To divide up the dataset, a list of all the index values is used. Each thread is then assigned a list of the index values which it uses in where clauses.

If the index is uneven or too coarse, you may not be dividing up the load evenly, reducing the potential benefits of multi-threading. If you have a “retained” variable in your data step, which resets based on another variables values, you must partition your dataset by that variable.

ONE THREAD TO RULE THEM ALL

There a several key steps which need to be setup by a “master” thread before it sends out it's minions to work.

The first thing a master thread needs to know is the path to the SAS executable. If you are using the SAS BI platform, the allowxcmd option must be enabled on the Object spawner, or you cannot initiate the slave sessions.

The master thread will need to also know how many threads to execute and what partition scheme they around going to use. If your system is used by multiple users, it may be best not to use all available processors. Generally, leaving 1 to 2 processors available for other users will help avoid getting angry emails from them, or your system admin. If the data is going to be physically divided into multiple paths for I/O purposes, the master thread will also need to do this.

Each thread should have it's own information dataset, which will change in name, and be written to, throughout the threads life-cycle. The starting name should be something similar to “THREAD_NOT_STARTEDx”. The dataset should contain:

- Thread number
- Total number of threads
- Partitioning scheme
- First row (if applicable)
- Last row (if applicable)
- Where clause (if applicable)
- Thread start time
- Thread end time

NOTE: To ensure a recursive loop of threads doesn't happen, the minion thread code should be in a separate SAS file from the master threads SAS file. If your data step is part of a larger program, then your master thread code should be in part of that code.

Once you have all the information, there's only one thing left to do, start the threads. Using the path to the sas executable, and the path to your minion thread SAS file, initiate a batch sas session. Add on the thread number as a command line input option using the sysparm statement. By using the “systask command/waitfor _all_” commands, you can ask that the master thread to sleep until all threads have completed. You could also use “x” command,

however it is not nearly as robust and offers no return error code. Using the "x" command would also necessitate that a sleep/wake-up routine be coded in the master thread code, to wake-up when all threads have finished.

MINION THREADS NEED INSRUCTIONS

When setting up the code for your minion threads, they need a bit of information in order to complete their tasks. The information they need will come from the information dataset (THREAD_NOT_STARTEDx) along with the sysparm variable, which helps give the minion thread its identity, or thread number. Using the thread number, the thread can pick up the information dataset that belongs only to it, and make the information contained in it available as macro variables via "call symput".

Depending on the partitioning type used, the where clause, or the first/last row numbers will get used by the minion thread to pick out only the rows that it should be processing. The output dataset should have the thread number on the end of its name, so it is easy to recombine them all when finished.

WHAT ARE MY THREADS DOING

Once the minion thread has read the information dataset, it needs to update the dataset change the name of it to reflect the current state of the thread (THREAD_NOT_STARTEDx ► THREAD_RUNNINGx). This change in name will allow you and the other processes to understand which threads are in what stage.

Another step at the end of the thread code should rename the information dataset once again (THREAD_RUNNING*x* ► THREAD_COMPLETE1), along with adding time-stamp information so thread execution time can be determined.

FINALLY IT'S DONE, TIME TO REJOIN

Using the systask "waitfor _all_ "statement, you can automatically wake your master thread when all the minion threads have complete. Using proc append is the quickest way to recombine your dataset, as it doesn't read all the data, it simply combines datasets together. However there is a catch; If the order of the final dataset matters, and you haven't selected horizontal partitioning, you will need to sort your data by the original record number, or whatever was the original sort order. This can be very expensive in terms of run-time, and may negate any benefits of multi-threading. When using interleaved partitioning, there is an alternative to sorting; By recombining your data in a data step, using multiple "set" and "output" statements for each partition, you can avoid the sorting process.

CONCLUSION

By careful consideration and planning, some very intensive data steps can be engineered to run in considerably less time by using the concepts presented in this paper. Although the example shown is for a simple data/set data step, the techniques are applicable to more complicated data steps such as merges and Cartesian product joins, by picking one dataset to partition. These techniques, along with SAS9's multi-thread enabled procedures, SPDE/SPDS, and SAS/CONNECT's "MPConnect", are valuable tools in dealing with large scale data processing needs, in multiprocessor environments.

REFERENCES

- Sassoon Kosian, *Generic Method of Parallel Processing in Base SAS® 8 and 9*, SAS Global Forum 2007 Paper 036-2007, Orlando FL, 2007, www2.sas.com/proceedings/forum2007/036-2007.pdf

CONTACT INFORMATION

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

Name: Ian J. Ghent
Enterprise: SAS Institute (Canada) Inc.
Address: 1600 – 360 Albert St.
City, Province, Postal Code: Ottawa, ON, Canada K1R 7X7
Work Phone: (613) 656-1920
E-mail: ian.ghent@sas.com
Web: www.sas.com

APPENDIX - EXAMPLE MACROS

Master Thread File

```
/**
 * Multi-threaded data steps - Master Thread Code
 * <P>
 * Example code for how to create a multi-threaded data step. This example is for a simple, single
 * set statement input. More complex data steps may be possible, depending on the nature of the
 * data step. Inputs from flat files are not supported, as there is no random access into specific
 * observations.
 * </P>
 * <P>
 * <B>WARNING:</B> You must be very careful to ensure that the data processing is accurate when
 * multi-threading. Any inter-row dependencies such as use of LAG functions and retain statements
 * will affect the accuracy. It is suggested that you sample and compare the output of your
 * simple data step with the multi-threaded version, to ensure correct processing.
 * </P>
 * @author   Ian J. Ghent
 * @created  2009-08-23
 * @modified 2010-02-09 by Ian J. Ghent
 * @version  1.0
 * @see      multiThreadDataStep
 */

/**
 * multiThreadDataStep macro
 * @author   Ian J. Ghent
 * @param    numberOfThreads            Number of threads to divide the data step into
 * @param    inputDatasetName           The input dataset which will be used for the data step
 * @param    partitionMethod            Select HORIZONTAL, INTERLEAVED, OR INDEXED (custom needs)
 * @param    minionThreadCodeLocation   Location of the minion thread code
 * @param    minionThreadLogsLocation   Location where the minion thread logs are to go
 * @param    horizontalLeadObs          In horizontal partitioning, number of obs to lead, so that
 *                                      some inter-row dependent functions such as LAG() can be
 *                                      used
 * @return   outputDatasetName          The processes dataset with the desired name as specified
 */
%macro multiThreadDataStep(
    numberOfThreads,
    inputDatasetName,
    partitionMethod,
    minionThreadCodeLocation,
    minionThreadLogsLocation,
    horizontalLeadObs,
    outputDatasetName
);

  * assign any needed libraries ;
  libname example "D:\Example";

  * some stings used in this macro might be longer then 255 chars, so suppress the warnings ;
  %let quoteOption = %sysfunc(getoption(QUOTELENMAX));
  options NoQuoteLenMax;

  * pickup the sasroot path from the dictionary tables ;
  proc sql noprint;
    select path into :sasRoot from dictionary.members where libname='SASROOT';
  quit;
  * set values for SASExecPath and SASConfigPath to reference in the systask command ;
  data _null_;
    length path_name $ 256;
    path_name = "'" || trim("&sasRoot") || "\sas.exe" || "'";
    call symputx("SASExecPath",path_name);
    path_name = "'" || trim("&sasRoot") || "\sasv" || scan("&sysver",1,'.') || ".cfg" || "'";
    call symputx("SASConfigPath",path_name);
  run;

  %* if no number of threads is specified use the CPUCOUNT SAS system option ;
  %if &numberOfThreads. = %str() %then %do;
    %put NOTE: **** THREADS NOT SET AS OPTION, RETREIVING FROM SYSTEM CPUCOUNT OPTION ****;
    %let numberOfThreads = %sysfunc(getoption(CPUCOUNT));
  %end;

  %* output selected options to the log ;
  %put NOTE: **** MULTI-THREADED DATA STEP INFORMATION ****;
  %put NOTE- **** NUMBER OF THREADS: &numberOfThreads. ****;
  %put NOTE- **** DATASET TO PROCESS: &inputDatasetName. ****;
  %put NOTE- **** DATASET TO OUTPUT: &outputDatasetName. ****;
  %put NOTE- **** PARTITION METHOD: &partitionMethod. ****;
  %put NOTE- **** MINION THREAD CODE: &minionThreadCodeLocation. ****;
  %if &partitionMethod. = HORIZONTAL and &horizontalLeadObs. ^= %str() %then
      %put NOTE- **** HORIZONTAL LEAD OBSERATIONS: &horizontalLeadObs. ****;;
```

```
%put ;

* clean out any existing thread datasets so we dont get confused ;
proc datasets library = example nolist nodetails nowarn;
  delete
      thread_part:
      thread_not_started:
      thread_running:
      thread_completed:
  ;
run;

%* variable initialization ;
%let threadList = ;
%let firstRecordPointer = 1;
%let firstObs = .;
%let lastObs = .;
%* loop through each thread, writing out information datasets and executing the thread ;
%do i = 1 %to &numberOfThreads.;
   %* if horizontal partitioning is selected, calculate the first/last obs ;
   %* for each thread to process ;
   %if &partitionMethod. = HORIZONTAL %then %do;
     * There are lots of quick ways to pick up the number of observations ;
     * from the data descriptor. However, only a real count is accurate in ;
     * all situations. Proc sql is used to provide multi-threaded capabilities;
     proc sql noprint threads;
       select count(*)
       into: recordCount
       from &inputDatasetName.;
     quit;
     %let recordsPerThread = %eval(&recordCount./&numberOfThreads.);
     %let recordsPerThread = %sysfunc(floor(&recordsPerThread));
     %let firstObs = &firstRecordPointer.;
     %if &i. > 1 and &horizontalLeadObs. ^= %str() %then
         %let firstObs = %eval(&firstObs -  &horizontalLeadObs.);
     %let lastObs = %eval(&firstRecordPointer. + &recordsPerThread.  - 1);
     %* the last thread picks up any additional records that didnt divide evenly ;
     %if &i. = &numberOfThreads. %then %let lastObs = &recordCount.;
     %let firstRecordPointer = %eval(&firstRecordPointer. + &recordsPerThread);
   %end;
   data example.thread_not_started&i.;
     thread_id = &i.;
     number_of_threads = &numberOfThreads.;
     dataset_name = "&inputDatasetName.";
     partition_method = "&partitionMethod.";
     first_obs = &firstObs.;
     last_obs = &lastObs.;
     where_clause = '';
     start_datetime = .;
     finish_datetime = .;
     run_time_minutes = .;
     format
         start_datetime
         finish_datetime datetime.
     ;
  run;

  * build up the sas command (easier using a data step) ;
  data _null_;
      sascmd =
          "&SASExecPath." ||
          " -sysin '" || "&minionThreadCodeLocation." || "'" ||
          " -config " || "&SASConfigPath." ||
          " -log '&minionThreadLogsLocation.\thread&i..log" || "'" ||
          " -noterminal -nostatuswin -sysparm &i."
      ;
      call symput('sascmd', sascmd);
  run;
  * execute the command ;
  systask command "&sascmd." taskname=thread&i. status=threadRC&i.;

  %* build up a list of all the threads by taskname ;
  %let threadList = &threadList. thread&i.;
%end;

%put NOTE: **** WAITING FOR THREADS TO FINISH ****;
* tell SAS to wait for all threads to finish before continuing ;
waitfor _all_ &threadList.;
%put NOTE: **** THREADS ARE FINISHED ****;
%* error check and recombine all datasets together ;
%do i = 1 %to &numberOfThreads.;
  %* check to make sure every thread completed successfully ;
  %if &&threadRC&i. ^= 0 %then %do;
```

```
      %put ERROR: **** THREAD PROCESS &I. FAILED, PLEASE CHECK THE THREAD LOG ****;
      %ABORT;
    %end;
    %if &i. = 1 %then %do;
      data &outputDatasetName.;
        set example.thread_part1;
      run;
    %end;
    %else %do;
      proc append base = &outputDatasetName. data = example.thread_part&i.
      %if &horizontalLeadObs. ^= %str() %then (firstobs = %eval(&horizontalLeadObs. + 1));;
      ;
      run;
    %end;
  %end;
  %put NOTE: **** REJOIN COMPLETE ****;

  * clean up the partitioned datasets (but not the informational datasets) ;
  proc datasets library = example nolist nodetails nowarn;
    delete thread_part:;
  run;

  * reset changed options ;
  options &quoteOption.;
%mend multiThreadDataStep;

%* using sashelp.sales as an example dataset ;
%multiThreadDataStep(
    8,
    sashelp.prdsale,
    HORIZONTAL,
    D:\SGF2010\MinionThread.sas,
    D:\SGF2010,
    ,
    example.completedataset
);
```

Minion Thread File

```
/**
 * Multi-threaded data steps - Minion Thread Code
 * <P>
 * This code is not meant to be stand alone. It is meant to be run in batch, via
 * a call from a master thread. Insert your normal data step code below the
 * "INSERT YOUR NORMAL DATA STEP CODE HERE" comment.
 * </P>
 * @author   Ian J. Ghent
 * @created  2009-08-23
 * @modified 2010-02-09 by Ian J. Ghent
 * @version  1.0
 */

/**
 * Thread macro
 * @author   Ian J. Ghent
 */
%macro thread();

  %let threadId = &sysparm;
  * If the host OS supports processor affinity commands (ie. Solaris pbind, Linux taskset) and you ;
  * want to make use of that, this would be where to put those calls via a systask or x command. ;
  * You can reference this process id using the SYSJOBID system macro variable. This should only be ;
  * done, if you know in advance, what the processor load for other tasks on the system will be. ;

  * assign any needed libraries ;
  libname example "E:\Example";

  * change the name of the dataset to reflect the status ;
  proc datasets library = example nolist nodetails;
    change thread_not_started&threadId. = thread_running&threadId.;
  run;

  * dump the information dataset to macro variables and update the start timestamp ;
  data example.thread_running&threadId.;
    set example.thread_running&threadId.;
    call symputx('numberOfThreads', number_of_threads);
    call symputx('datasetName', dataset_name);
    call symputx('partitionMethod', partition_method);
    call symputx('firstObs', first_obs);
    call symputx('lastObs', last_obs);
    call symputx('whereClause', where_clause);
    start_datetime = datetime();
  run;
```

```
  * start the main data step ;
  data example.thread_part&threadId.;
    set &datasetName.
    %* depending upon the partition method, only read selected observations ;
    %if &partitionMethod. = HORIZONTAL %then %do;
      (firstobs = &firstObs. obs = &lastObs.);
    %end;
    %else %if &partitionMethod. = INTERLEAVED %then %do;
      ;
      %if &threadId. = &numberOfThreads. %then %let modValue = 0;
      %else %let modValue = &threadId.;
      if mod(_n_, &numberOfThreads.) = &modValue.;
      * provided in case the dataset needs to be sorted in the original order ;
      record_number = _n_;
    %end;
    %else %if &partitionMethod. = INDEXED %then %do;
      (where = (&whereClause.));
    %end;

    **** INSERT NORMAL DATA STEP CODE HERE ****;
    difference = actual - predict;
    format difference dollar10.2;
  run;

  * record the timestamp when the data step finished execution, and calculate the run time ;
  data example.thread_running&threadId.;
    set example.thread_running&threadId.;
    finish_datetime = datetime();
    run_time_minutes = intck('MINUTE', finish_datetime, start_datetime);
  run;

  * change the information dataset name to reflect the status ;
  proc datasets library = example nolist nodetails;
    change thread_running&threadId. = thread_completed&threadId.;
  run;
%mend thread;
%thread();
```