Paper 195-2007

Two heads are better than one – Getting the most out of multiprocessing

Faisal Dosani, RBC Royal Bank, Toronto, ON

ABSTRACT

This paper explores the multiprocessing / multithreading capabilities of SAS® 9. Questions such as "What is multi processing?", "How does it apply when using PROC SORT and PROC SQL?" and "what are the pros and cons" will be addressed. The intention is to help users implement these features and understand the advantages of using multiprocessing and multithreading.

INTRODUCTION

Before my days of using SAS I was well versed in the world of JAVA®, a very complete and powerful language in its own right. Out of all the bells and whistles of that language, threading functionality was available to everyone and a very powerful tool if used correctly. What is threading? It is not sewing or making the next fashion statement! It is the notion that two heads are better than one. Threading, or Multiprocessing, comprises of splitting up your work into chunks and running code in parallel. This is especially useful when you have more than one CPU. So rather than having one flow of execution, you can have multiple flows, which in turn should run more efficiently. With SAS, we now have in our hands the ability to take advantage of some of the powers of this functionality.

Some things to take note with SAS's multiprocessing capabilities is that it works best with large amounts of data as it is intended for resource intensive operations. The SAS system will ultimately decide if multiprocessing takes place or not based on different factors such as the number of the CPU's or options selected within a specific procedure.

If the dilemma arises that you do not have a multiple processors on your server or computer, not to worry. Some of the performance improvements can work on single CPU machines. Let's look at two of the most commonly used PROC's and see how this functionality works in them.

PROC SORT AND THREADING

PROC SORT is one of the staples of SAS which we have all used at one point in our SAS jobs and is now one of the procedures which supports multiprocessing. The idea is that you can have the sort essentially broken up in chunks and processed by the two or more CPU's simultaneously.

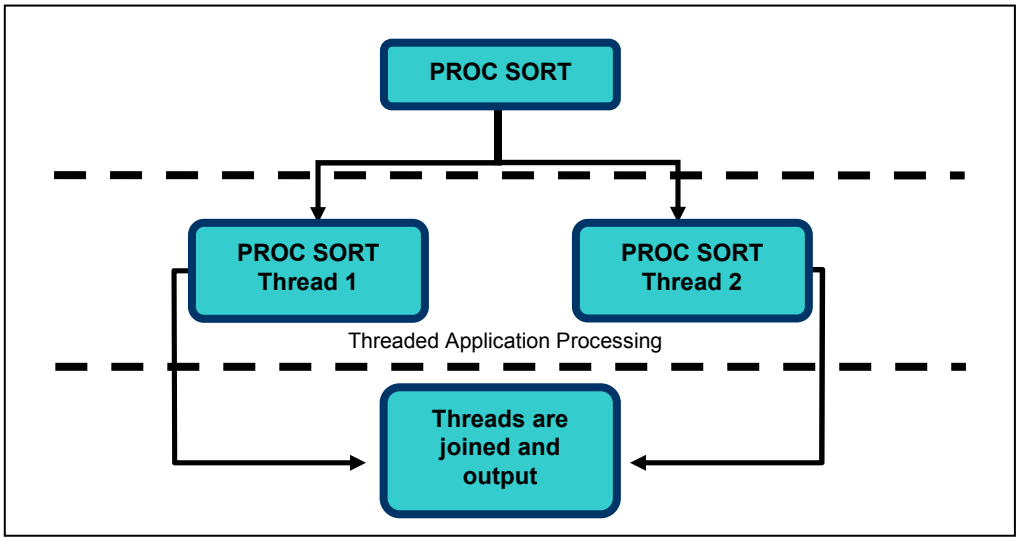

Figure 1

There are a couple of system options which relate to multiprocessing which can be set. These are the CPUCOUNT and THREADS/NOTHREADS options. They don't need to be specified but if you are planning on using THREADS for certain PROCS it would be wise to set at the minimum the CPUCOUNT so SAS knows how many processors it has at its disposal.

```
/* Set the threads option and number of CPU's that SAS */
/* should use                                          */
OPTIONS THREADS CPUCOUNT=2;

/* PROC OPTIONS will display the options and their     */
/* current settings                                    */
PROC OPTIONS GROUP=performance;
RUN;
```

With CPUCOUNT the value after the equals sign has to be a number ranging from 1 – 1024 or the word ACTUAL. ACTUAL is the real number of CPU's which are available. You may think that increasing this number will have a positive impact on performance regardless of your number of CPU's. However this is not the case since the algorithm in the background uses this number to optimally schedule the threads thereby reducing efficiencies. Best practices would state to just use the ACTUAL option which will automatically use the optimal number of CPU's you have at your disposal. This can be especially useful if you are not entirely sure of the machine specs or unsure if your user rights permit you to use 1 or more CPU's.

If you don't specify THREADS or specify NOTHREADS then you can include it as an option in the PROC you are calling. Keep in mind that you will need to reference THREADS in each PROC if it isn't set globally with an OPTIONS statement. If the NOTHREADS option is selected, the CPUCOUNT is automatically ignored.

The actual syntax is fairly simple when you use the THREADS option in your PROC SORT code.

```
PROC SORT DATA = myLib.myDataset THREADS;
    BY ID Name;
RUN;
```

Conversely, if you want to ensure that multiprocessing is not used, your code would look like the following:

```
PROC SORT DATA = myLib.myDataset NOTHREADS;
    BY ID Name;
RUN;
```

The THREADS / NOTHREADS option overrides the system options.

So what exactly happens when you submit your program? Let's assume that you are just submitting a piece of code that is a simple PROC SORT step. The code is submitted to the SAS engine. The engine realizes via the THREADS option that you are using a multithreaded process. The dataset which you have selected to be sorted is then broken down into subsets. Each of these subsets are sorted individual and simultaneously. Once all of the subsets of the data are sorted they are then interleaved back into one dataset. The interleaving is all done behind the scenes.

PROC SQL (SAS/ACCESS) AND THREADING

With SAS/ACCESS we can connect to a variety of DBMS's with different methods. It is important to note that many people use the pass-through facility when querying their databases. This method does not support threaded reads as the query is passed along to the DBMS to process and return the results. It is essentially a black box. It takes in SQL and returns a set of information based on the input. But if the database itself supports some sort of multithreading within the database engine, then this can certainly be exploited to return queries much faster. An example of this is Teradata's Fast Export utility - a module that can help read large amounts of data with enhanced performance.

When it comes to SQL the multi processing should really only be used when accessing very large amounts of information in the millions of records. There are so many variables which might come into play that you always want

to test a multi threaded vs. non-multi threaded piece of code to ensure you see sufficient gains before implementing permanent changes.

The other way to access DBMS is through the LIBNAME reference. The basic syntax is as follows:

```
OPTIONS THREADS CPUCOUNT=2; /*Set the THREADS option */
LIBNAME myLib <database type> <connection options> <LIB name options>;
```

You can now access the database like you would any normal dataset like a SAS library. MyDS will now contain everything from the DBMS table Person.

```
DATA myDS;
    SET myLIB.Person;
WHERE gender = 'F' and age > 21;
RUN;
```

Behind the scenes SQL is generated to retrieve the information. With the new threaded functionality the query is essentially split into chunks, similar to what we saw with the PROC SORT functionality.

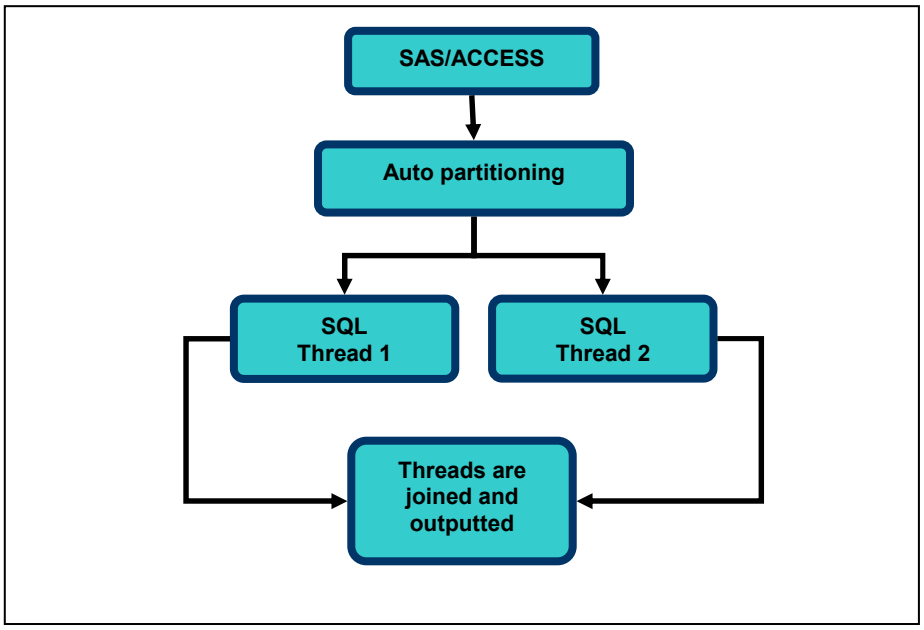

Figure 2

Figure 2 is similar to the Figure 1 except for the Auto partitioning. What this does is break the query into separate segments so separate threads can process those portions of the SQL and run in parallel. Ideally with auto partitioning we want the system to split up the data proportionately so we have an even split between the threads.

Depending on the DBMS you use, the Auto partitioning can be done in a variety of ways. Some DBMS's have specific auto partioning utilities while SAS's base Auto partitioning follows the MOD mathematical function. SAS takes the query which is generated and appends a WHERE clause with the MOD function with a column variable which it sees from the tables being queried. This variable must be a numeric field. MOD returns the remainder from a division operation, so this helps SAS decide which thread it goes into. Let's take a look at an example using two threads.

MOD EXAMPLE

We have our database Person with two fields. Field one is an employee number and field two is the person's name. For our SQL we just want to do a simple select of both fields.

Employee Number	Name
34245525	Employee 1
86787769	Employee 2
56246266	Employee 3

In this case Employee Number would be a great field for the auto partitioning feature since it is a numeric and will work well with the MOD function. Below we have a basic outline of the calculation done.

MOD (<column variable>, <number of threads>) = Remainder

So in our case since we are dealing with 2 threads our data will be partitioned as follows:

MOD Calculation	Partitioned in to
MOD (34245525, 2) = 1	Thread 2
MOD (86787769, 2) = 1	Thread 2
MOD (56246266, 2) = 0	Thread 1

The actual SQL generated would look like the following:

```
Thread 1 - SELECT emp_no, name FROM Person WHERE (MOD(emp_no,2)=0)
Thread 2 - SELECT emp_no, name FROM Person WHERE (MOD(emp_no,2)=1)
```

This is a very simplistic view of auto partitioning but shows how the data is split up. With a random grouping of employee numbers you can hope to achieve some sort of balance so one thread does not have more rows than another. If we had a case where all employee numbers were odd numbers then all the rows would end up being partitioned into one thread and might actually degrade performance. Being cognizant of these sorts of factors will help making the correct choices.

If there aren't any column variables which SAS can use for the MOD function, the auto partitioning will not work and neither will the threading. There are specific situations in which the multiprocessing will not work and these are noted in the SAS documentation. You can specify your own partitioning clause with the DBSLICE and DBSLICEPARM option, but you should only use this if you can guarantee that your partitioning is better than the auto partitioning which SAS provides.

We can also use the threading like we did with PROC SORT for the sorting involved with PROC SQL.

```
PROC SQL THREADS;
    .. SQL code ..
RUN;
```

Again this is pretty simple. The THREADS or NOTHREADS option essentially overrides the global option tag depending on how it is set. This allows parallel processing in the sorting features of PROC SQL and does not affect the querying itself but rather the sorting.

I/O MULTI PROCESSING

The previous two sections dealt with application multi processing. Let's now take a look briefly at "input-output multi processing". This deals with how data is read in and written out from disk. The key element is that your data set has to be a partitioned data set. To have these types of datasets you should be using the Scalable Performance Data Engine (SPD). This Engine is used for high performance input / output and really is used to take advantage of the multi processing capabilities. Creating an SPD dataset is quite easy as well and simply requires adding the spde keyword when creating the libname reference.

```
LIBNAME myLib spde "C:\myDS";
DATA myLib.myDataset;
```

```
    .. Your code ..
RUN;
```

The SPD libname reference has multiple options, so it is worth looking through the SAS documentation to ensure you are maximizing performance by specifying multiple index and data paths.

If you want to convert a permanent dataset from the base SAS engine format to the SPD type, you can use the following procedures to do so:

- PROC APPEND
- PROC COPY

Check the SAS documentation for specific details on how to convert to a SPD dataset.

The SPD engine only runs on the current platforms (UNIX, Windows, z/OS on zFS file system only), and OpenVMS Alpha on ODS-5 file systems so this might be an additional constraint for some.

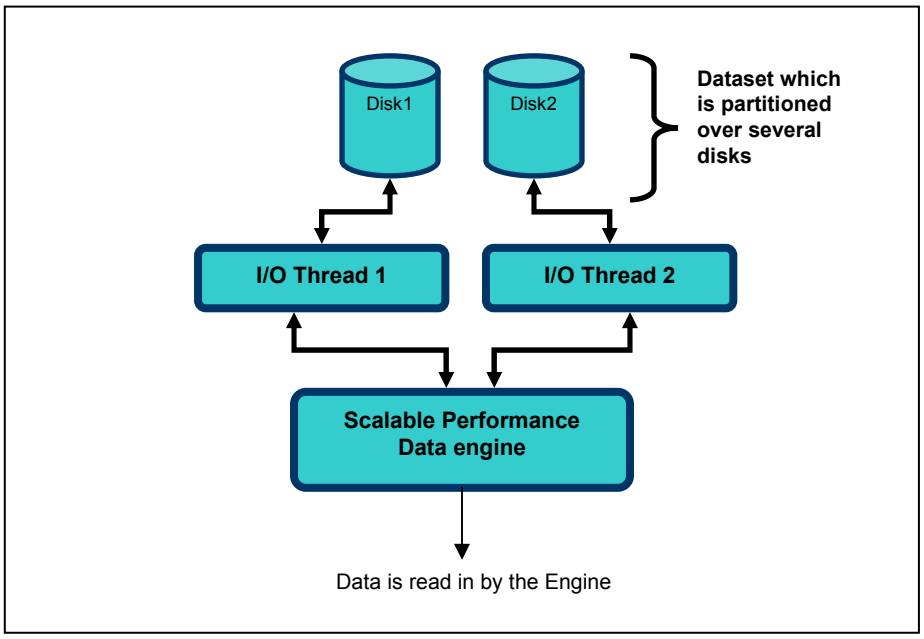

Figure 3

With the SPD I/O processing the key is also having the proper hardware (multiple disk drives) which can help boost performance. Figure 3 helps to illustrate the concept of reading data in parallel from a partitioned dataset.

The SPD datasets are different then regular SAS engine datasets. In a regular SAS dataset everything from the file descriptor to the data is within the one file. With a SPD dataset it is partitioned over several files and the data can be read by different processes and then combined back together into one final dataset for processing very similarly to the PROC SORT functionality which we saw earlier.

THE NUMBERS

Now that we have covered the theory lets take a look at some of the results from test runs which were done on a 6-CPU UNIX server. First and foremost, this is a shared server so when these tests were run other SAS processes and jobs were running which were utilizing CPU, memory, and storage resources. There is certainly a margin of error involved in the results which should be considered.

Using the RANUNI function we randomly fill a data set with approximately 52 million rows and 4 variables. It is a good idea to get as much data as possible since multiprocessing is best used with large amounts of data. Once the

data set is created we can simply sort a few of the variables with the THREADS option on and off. We run the code with both options twice just to make sure the results which we obtain are not an anomaly.

NO THREADS SORT

```
options COMPRESS=NO NOTHREADS;
PROC SORT DATA=temp; by x y z;
RUN;
```

THREADS SORT

```
options COMPRESS=NO;
PROC SORT DATA=temp THREADS; by x y z;
RUN;
```

The results from the code above are listed below.

NO THREADS RUN 1

```
NOTE: There were 52755920 observations read from the data set WORK.TEMP.
NOTE: The data set WORK.TEMP has 52755920 observations and 4 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           5:00.36
      cpu time            2:03.28
```

NO THREADS RUN 2

```
NOTE: There were 52755920 observations read from the data set WORK.TEMP.
NOTE: The data set WORK.TEMP has 52755920 observations and 4 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           7:46.85
      cpu time            2:07.81
```

THREADS RUN 1

```
NOTE: There were 52755920 observations read from the data set WORK.TEMP.
NOTE: The data set WORK.TEMP has 52755920 observations and 4 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           1:57.53
      cpu time            1:30.67
```

THREADS RUN 2

```
NOTE: There were 52755920 observations read from the data set WORK.TEMP.
NOTE: The data set WORK.TEMP has 52755920 observations and 4 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           1:21.95
      cpu time            1:23.50
```

As you can see from the real time results there is actually a huge discrepancy between the threaded version, and the non threaded version.

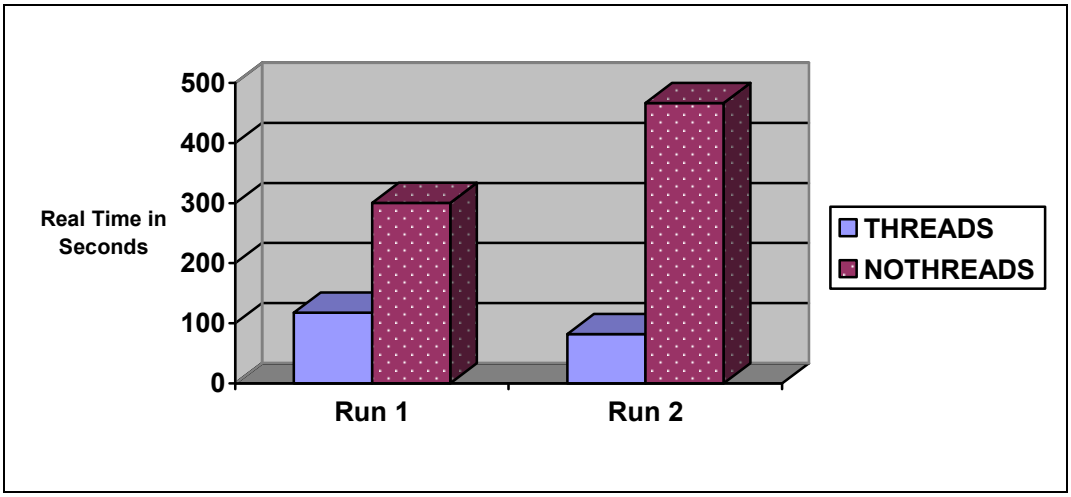

Figure 4

Depending on your specific setup the results will certainly vary but from these test runs we saw a 61% and 82% increase across our two runs. The best part is that the threading is automatically turned on by default with SAS 9, so it is not cumbersome to configure.

CONCLUSION

Overall, there seem to be many benefits of implementing the new multi-processing features of SAS into applications, especially those which are resource intensive and deal with large amounts of data. The true benefits can be seen when the hardware and database infrastructure are there to support the background workings of SAS. In addition a little know how of when, and when not to use these features goes a long way. Therefore below is a quick summary of some of the pros and cons of using multiprocessing.

PROS

- Increased performance
- Better productivity
- Easy to implement. SAS is able to do most of the work for you
- Cost savings in terms of resource utilization.

CONS

- Initial cost of Infrastructure can be costly
- Using multi-processing features can actually degrade performance in certain situations
- Not all Procedures or aspects of SAS support multi-processing.

Bottom line multiprocessing can improve performance if used correctly in the right environment. As a rule of thumb it makes sense to test with and without the multi process options to get an idea of the enhancement being made and adapt accordingly.

REFERENCES

SAS Institute Inc. 2006. **SAS OnlineDoc® 9.1.3.** Cary, NC: SAS Institute Inc.

Ray, Robert. 2003. “An Inside Look at Version 9 and Release 9.1: Threaded Base SAS® procedures.” *The Twenty-Eighth Annual SAS® Users Group International Conference Proceedings*, Seattle, WA. http:/www2.sas.com/proceedings/sugi28/282-28.pdf (November 22, 2006).

Langston, Rick. 2005. “Efficiency Considerations Using the SAS® System.” *Proceedings of the Thirtieth Annual SAS® Users Group International Conference*, Philadelphia, PA.

http://www2.sas.com/proceedings/sugi30/002-30.pdf (November 22, 2006).

AKNOWLEDGEMENTS
I would like to thank the following people for their help and input into this paper:

- Marje Fecht
- Adam Francis
- Robert Ray
- Olisa Hestick

CONTACT
Your suggestions and questions are welcomed. Please contact the author at:

Faisal Dosani
Senior Information Analyst
RBC Royal Bank
330 Front St W - 7th Floor
Toronto, Ontario
M5V 3B7
Canada
Faisal.Dosani@rbc.com
416-955-7654