Paper 217-29

Threads Unraveled: A Parallel Processing Primer

David Shamlin, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® System 9 introduces a threaded kernel into the core part of the SAS System. These threaded kernel services enable SAS R&D to develop software that runs faster, increases throughput, and enhances scalability. This paper looks at the threaded kernel architecture from a fundamental perspective, answering the following questions:

- What are threads and how do they relate to symmetric multiprocessor (SMP) computers?
- How does threaded software improve job performance?
- What does parallel processing buy me and what does it cost?
- Exactly what kind of improvements should I expect with SAS 9?

Answers to these questions, as well as additional information about specific SAS 9 cases that highlight PROCs, engines, and other components, is provided in this paper.

INTRODUCTION

Threading is a popular trend in software design and implementation. The term “multi-threaded” is used to describe software systems that create and manage multiple streams of executable code simultaneously. Multi-threading is used often in performance-critical applications. Promotional material for these systems and applications is quick to emphasize threaded aspects of the software’s capabilities; this implies there is real value to such offerings. Beginning with SAS 9, threading technology was introduced into the SAS product line.

The threaded components of SAS 9 are positioned to improve performance and scalability. SAS 9 documentation lists an extensive set of components, including procedures, engines, service routines, and servers that have been enhanced to exploit threads. Because of threading, there is a basic expectation that SAS 9 runs faster and can simultaneously take advantage of multiple processors on both SMP and non-uniform memory access (NUMA) multiprocessor computers. This is generally true, however, actual performance gains depend on how the software is deployed, configured, and used to solve specific problems.

Over the past two years, I have met with user groups across the country and presented brief, technical introductions to some of the thread-enabled pieces of SAS 9. When sharing this material, questions have been posed about how specific SAS 9 components behave on a particular platform in a given situation. In addition, basic questions have been asked about what threads are and how they work. Given these frequently asked questions, I believe it is important to take a step back and review some of the fundamentals about threading.

Reviewing the underpinnings of threading will help you understand what parallel processing is and give you insights into the performance limits of threaded software, which can help you set your own expectations about the speed of SAS 9. Therefore, the goal of this paper is to explain threads as a programming tool, demonstrate how threads have been applied in SAS 9, and explain the basis behind the reasonable performance limits that exist. After reading this paper, you should have enough technical information to set a reasonable expectation of what kind of performance characteristics you will see when using SAS 9, along with practical ideas of how to optimize your applications.

WHAT IS A THREAD?

Simply stated, a thread is a single, sequential flow of control inside a process. Within a SAS process, there are tasks and threads. DATA steps and procedures (PROCs) are examples of tasks. While more than one task can exist at any given time within a SAS process, only one task executes at a given time. Tasks are cooperatively scheduled for execution, which means that after a task is created, it is placed in a queue with other tasks that will be run. When a task reaches the front of the queue, it cannot run until the currently executing task gives up control of the processor.

In computer science terms, a task is a single-threaded process. Within such a process, there is a single point of execution. The thread of execution has exclusive access to the memory used by the program. However, in a multi-threaded process, there can be more than one point of execution at any given time and multiple threads share the same memory. Figure 1 depicts the differences between single-threaded and multi-threaded processes.

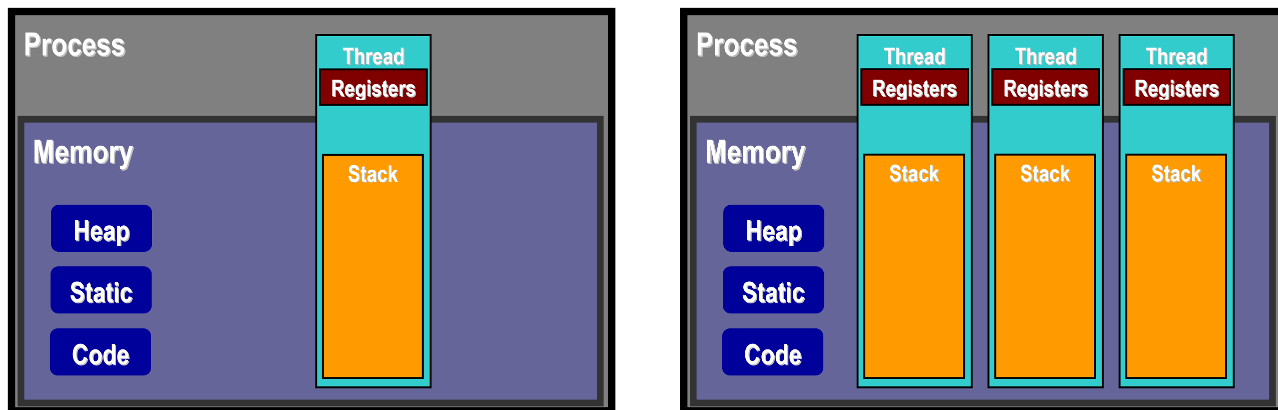

Figure 1: Single-Threaded versus Multi-Threaded Processes

A SAS task is called a "heavyweight process." In addition to the execution stream (for example, the program's instructions), a heavyweight process has other resources associated with it such as memory, file handles, and interrupt handlers. Because of all these resources, switching from one SAS task to another is time consuming. All state information associated with the process must be saved before the task can completely yield and relinquish control of the CPU to another task.

A thread is called a "lightweight process." The dynamic state of a thread is defined by a set of CPU registers and a stack. Compared to task switching, thread switching is more efficient because memory and resources are shared between threads. Multi-threaded programming is a form of parallel programming whereby several threads of control are scheduled concurrently in the same process, sharing the same memory space and sharing system resources. Furthermore, if the program is run on a machine with multiple processors, threads can execute concurrently.

Threads are useful for several reasons. Because threads can run in parallel on several processors, they enable a program to exploit multiprocessor machines. Work can be divided into units that can be performed simultaneously by a set of threads. As an example, consider a task that searches a list of items. In a single-threaded implementation, the task searches the entire list in a single context. Regardless of the number of processors available, the program will use only one processor at a time. The same program could be implemented to use multiple threads. This is accomplished by dividing the list into a set of sub-lists, one for each available processor. A separate thread can search each sub-list simultaneously by using a different processor. Such a multi-threaded implementation would find an element *n* times faster (where *n* is the number of processors) than a single-threaded implementation. In general, such multi-threaded programs are expected to run faster than a single-threaded program. (Even on single processor machines, threads allow overlapping I/O and computation in a simpler way and can speed up time to completion.)

For multi-threaded programming to improve software performance, threads that execute in parallel must have a high degree of concurrency. In other words, the instructions in each thread must execute independently of the instructions in each of the other threads. While the instructions in a single thread execute sequentially, their execution might be interleaved or simultaneous with the execution of instructions in other threads. Each thread should be as independent of other threads as possible for optimization.

This degree of independence is most likely to be problematic when shared resources are a factor. Updatable data shared by multiple threads is a typical case. For example, if one thread is updating shared data, other threads should not be accessing the shared data at the same time. The multi-threaded program must be implemented so that such concurrency violations do not occur. Yet, protections come with a cost. If one thread is frequently locking other threads from accessing shared data, the other threads can spend a lot of time waiting. If too much waiting occurs, the program will not achieve performance gains and parallel execution is thwarted. In the worst case scenarios, performance in a multi-threaded program can be slower than performance in a single-threaded program because more time is spent waiting for resources to become available than in actually solving the problem.

Intelligently written software will protect against these problems. However, there are boundary cases where smartly written, multi-threaded code degrades. Understanding that this can occur is the first step in safeguarding against it. You should identify lackluster job performance and optimize your code using relevant techniques to protect yourself from real-world problems. There are no tuning strategies that apply to all software—multi-threaded or otherwise. Instead, you must review how to appropriately use the software modules that your program relies on. In other words, read the documentation. If a particular component includes special tuning parameters for multi-threaded execution, its documentation will discuss them.

PARALLEL PROGRAMMING PATTERNS

As noted in the search example in the previous section, using threads to implement a program can reduce the time to solution by dividing the work into parts and performing it in parallel. There are three basic patterns for threaded programs that have evolved over time. Each pattern has a particular application, and for complex threaded programs, these patterns tend to be used in combination to address various pieces of the program.

The first pattern is the *Boss/Worker Model*. Figure 2 shows the main “Boss” thread and a set of subordinate “Worker” threads. Based on the input the Boss thread receives, it generates tasks that the Worker threads can complete. This model is used often in user interface software. For example, imagine an explorer window that has a hierarchical tree view of the space being explored in one pane (on the left side of the window) and a view of the contents of the selected tree element in a second pane (on the right side of the window). Figure 3 shows an example of this type of window, with a highlighted navigational button that moves the aggregate explorer view one level up in the hierarchical tree view. The Boss thread receives a button event when the user clicks this navigational button. As a result, both panes (on the left and right sides) in the window need to be updated. The Boss thread can delegate work to the Worker threads by sending two requests: one request for the Worker threads to update the hierarchical tree view, and one request for the Worker threads to update the detailed explorer view. The Worker threads can perform these work requests in parallel.

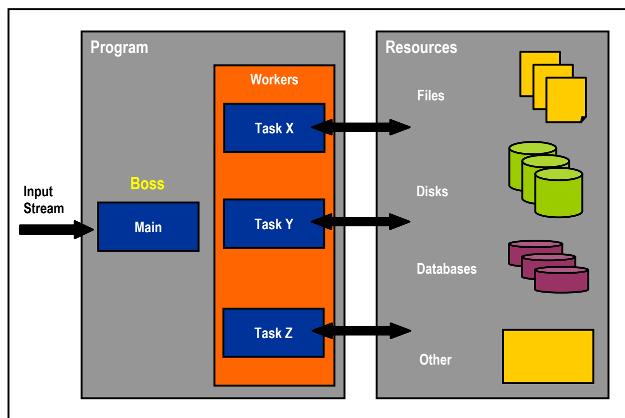

Figure 2: Boss/Worker Model

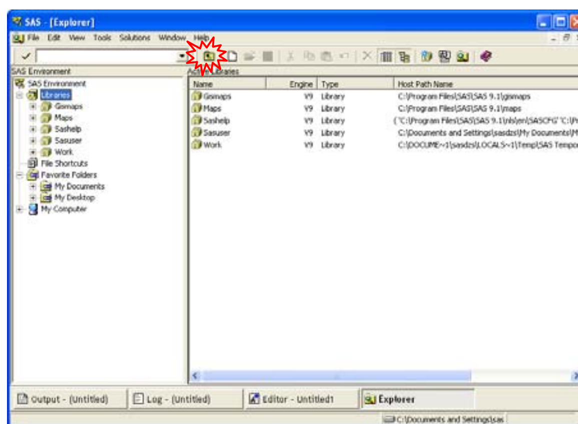

Figure 3: SAS Explorer Window

The second pattern is the *Peer Model*. As shown in Figure 4, several Worker threads are created to process static input simultaneously. Each Worker thread can operate on all rows of the data (for example, scoring or summarizing columns). Or, each Worker thread can do the same thing to different sets of rows of the data (for example, performing the search example in the previous section). In either situation, each Worker thread is responsible for its own input request. This is unlike the Boss/Worker Model, where a stream of input requests is directed through the Boss thread. The SORT procedure in SAS 9 employs this parallel-processing pattern. PROC SORT separates the data set into a series of data blocks. Each data block is made up of a unique set of rows from the input data set. The rows in each data block are sorted independently of the rows in other data blocks. Next, all data blocks are sorted into a single output data set during the PROC SORT merge phase.

The third pattern is the *Pipeline Model* and is shown in Figure 5. This pattern is used when the best way to process input is by dividing the work into independent stages. The work is divided into a series of steps. Each step is implemented by a separate thread. The output of one step becomes the input of the step that follows it. The individual steps tend to be small and simple. In SAS 9, PROC SUMMARY implements the Pipeline Model. The first step ranks the class variable, the second step searches the summary space for that class variable’s information, and the third step updates the class variable’s statistics.

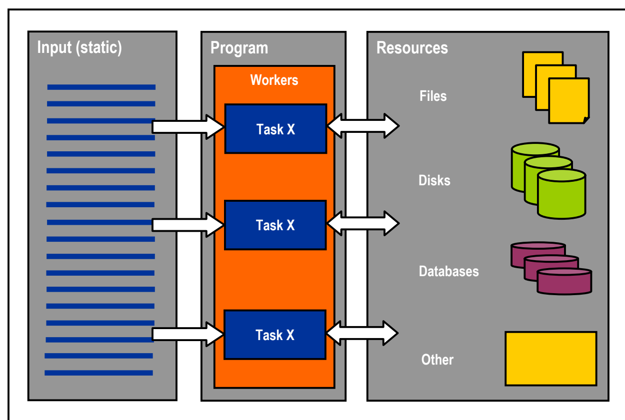

Figure 4: Peer Model

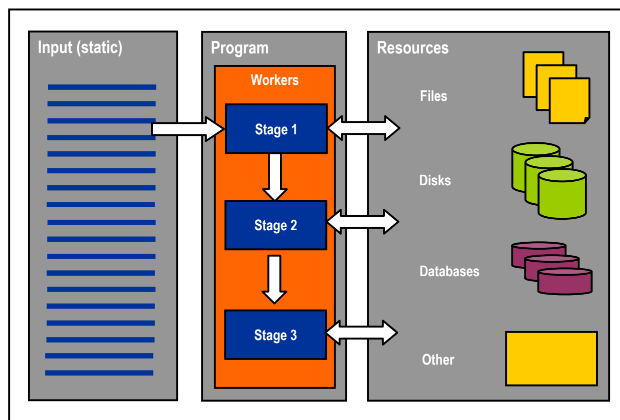

Figure 5: Pipeline Model

All three patterns occur repeatedly in most parallel-processing implementations. They are even used in combination by complex programs. The primary purpose of applying these patterns is to improve performance. In the case of parallel processing, doing subsets of the work in parallel reduces the time it takes to derive an answer. Multi-threaded applications are generally faster. However, executing software on multiple threads does not guarantee it will be faster than executing it on a single thread because there are other factors.

PERFORMANCE AND SCALABILITY

When discussing performance characteristics of a computer system, it is necessary to have clear terminology. *Software scaling* is the ability of software to leverage additional resources (such as CPUs or I/O channels) to reduce elapsed execution time by a corresponding amount. For example, if you increase the number of CPUs on your SMP machine, you can reasonably expect your job to complete in less time, assuming that another resource limitation such as I/O capacity is not encountered. If this is not the case, then the software scales. Scalability is one particular facet of software and hardware performance capabilities. In today's business landscape where the volume of data increases rapidly, solutions should scale.

There are two things to consider when looking at how solutions scale: hardware and software. Scaling up your hardware will not improve performance if the software does not scale. In other words, simply adding CPUs to a piece of hardware does not guarantee that your application will run faster. Multi-threaded software should partner with the multi-processor computer to achieve scalability.

So far, I've discussed what threading is and how it relates to parallel processing. During this discussion, there has been the underlying promise of improved performance when these types of techniques are applied, and certainly this can be said to be true. SAS 9 delivers on the promise to improve performance and scalability using multi-threaded technology. In many cases, gains are seen in "out-of-the-box" configurations, but there are exceptions and limitations. Understanding when and why these can occur is important to optimizing any SAS 9 deployment. The first step is to understand the limitations of scalability.

OBEYING THE SPEED LIMIT

In a perfect world, any problem would scale perfectly. You would see a degree of speedup equal to the amount of resources added to the problem. For example, bubble sorts execute with an efficiency of $O(n^2)$. Assume it takes one minute to sort data on a single processor computer. If you add a second processor, and it takes 30 seconds to sort the data, you say that your sorting solution scales perfectly. This implementation requires the sorting algorithm to divide the data into two parts, with each part sorted in a separate thread. This is called *linear scalability* because the number of times faster your job runs (speedup) is equal to the number of CPUs added to the job.

In the real world, few jobs scale perfectly. This is because only part of the work can be done in parallel. Based on the preceding example of data sorting, try adding a step to print the data after it is sorted. The work involved in printing the data cannot be divided into tasks that can execute in parallel because of the physical constraints of writing information to an output file. Ordered lines of output must be written one at a time; it is not feasible to divide such work into multiple tasks that execute in parallel. As a result, part of the job is scalable and part of it is not (see Figure 6). In most situations, some part of the work must be done in isolation, for example, managing multiple threads and exclusively accessing shared data. To fully understand this concept, consider the concept of concurrency. The portion of work that can be divided into subtasks and executed concurrently is the *scalable* part of the job. The greater the portion that can be executed concurrently, the more the performance will benefit from multi-threaded execution.

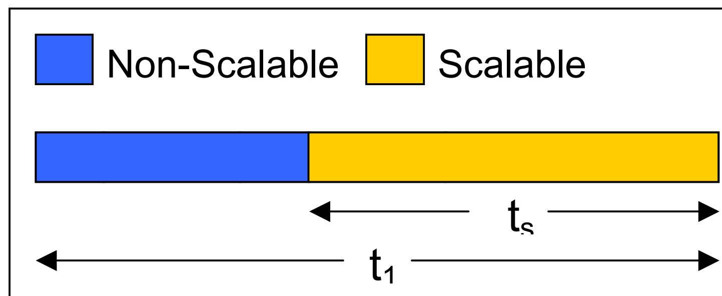

Figure 6: Scalable and Non-Scalable Job Portions

Using some simple math, the ratio of time it takes to complete the scalable portion of a job on a single processor, to the time it takes to complete the entire job, is called the *parallelizable fraction* of the job. Parallelizable fraction is $PF=t_s/t_1$, where t_s is the amount of time it takes to complete the scalable portion of the job, and t_1 is the amount of time it takes to complete the entire job using one processor. In the previous example, it took one minute to sort the data; let's assume it takes 40 seconds more to print the data; therefore, the total job time is 100 seconds. The parallelizable fraction of the job is $PF=60/100= 60\%$.

By knowing the parallelizable fraction of a job, you can predict the amount of speedup that should be achieved as you add processors. When you add a second processor to the job, the time it takes to sort the data is cut in half (for example, $t_s=30$), while the time it takes to print the data remains constant. The total job time becomes 70 seconds. Thus, you get speed up of 100/70=1.43. The speedup factor is expressed as $S_p=t_1/t_p$, where t_1 is the amount of time the job runs on one processor, and t_p is the amount of time the job runs on p processors. Regardless, no matter how many processors are added to the example problem, the job will never complete in less than 40 seconds because that is the amount of time it takes to complete the non-scalable (printing) portion of the job.

Most programming problems have a scalability limit: some part of the work must be done in isolation. The example here makes this obvious because the non-scalable portion is discrete from the scalable portion. In many job cases, the separation is not so obvious. An individual thread can block other threads for a period of time when it needs exclusive access to a shared resource. For example, a common data structure might need to be exclusively accessed. The code sections in a thread that allow exclusivity make that portion of a parallel program non-scalable. If you could observe a series of threads executing simultaneously to collaboratively complete a computation, you would probably see an individual thread monopolize a shared resource for a period of time. All other threads sit idle until the shared resource is released by the accessing thread. This common behavior limits the program's scalability. Therefore, the goal of a multi-threaded program is to minimize isolated execution and maximize concurrent execution.

Because most programs have some part that cannot be executed in parallel, there is a speed limit for parallel programs. This observation is known as Amdahl's law. Figure 7 shows the theoretical bounds of this law for a set of parallelizable fractions. Amdahl's law assumes that the goal of parallel programs is to decrease the time to solve a problem of a fixed size. For the data sorting example, the goal is to sort and print the data faster with each processor (or thread run in parallel), and Amdahl points out that there is a maximum number of processors beyond which the speedup factor remains constant. Cohen (2002) outlines the mathematics behind this law in more detail.

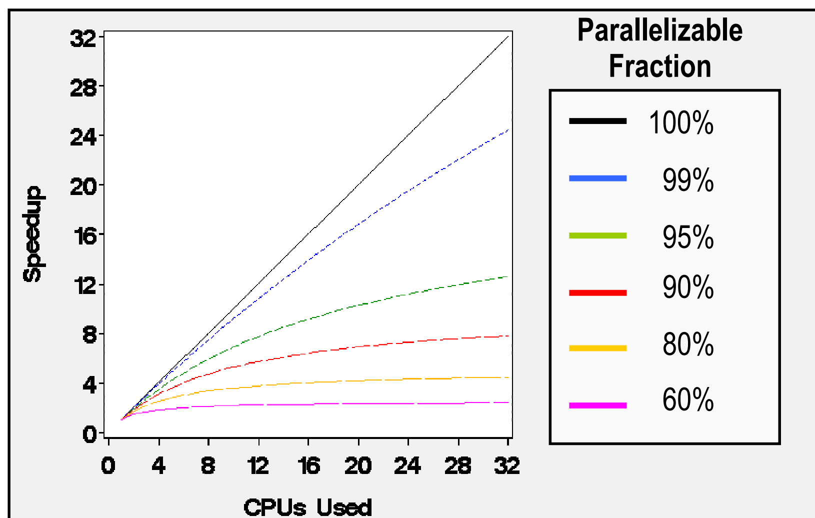

Figure 7: Examples of Amdahl's Law

Amdahl argues that for most real-world applications, the maximum amount of acceleration lies between 5 and 7 processors. This could lead to the belief that there are practical limits to the value of larger servers with many processors. If you are willing to change your goals for scalability, you can realize additional gains. Increased acceleration is possible when the problem size increases. As the problem size increases, it is common for the parallelizable fraction to increase as well. In the sorting and printing job example, the scalable work (sorting) increases proportionally to the square of the number of rows in the data; the non-scalable work (printing) increases at a linearly proportional rate. For example, if you double the number of rows in the data, the time that is required to perform the sort quadruples (240 seconds), while the time required to print only doubles (80 seconds). This raises the parallelizable fraction from *PF=60/100=60%* to *PF=240/320=75%*. The speedup factor for a two-processor computer becomes $S_2=320/120=2.67$. This is an 87% increase in throughput.

Although you might see performance gains when your problem size increases, you might see performance degradation when the problem size becomes very small. This can be due to the small parallelizable fraction of the job, or it can be due to the overhead involved in managing multiple threads. It takes time for the computer to create and schedule each thread. If many threads are created to do a small amount of work, the overhead involved in managing and maintaining the threads can overwhelm any acceleration that could have been gained through parallelization. For this reason, it is sometimes better to run small jobs in a single thread. In fact, smartly written, multi-threaded programs will switch to single-threaded code paths in such cases.

The scalability of a multi-threaded program is impacted not only by the number of processors available to solve a problem, but also by the problem itself. Other factors can impact multi-threaded program performance. Many computer jobs are I/O-intensive. For these jobs to perform optimally, the disks and controllers need to be tuned for the job at hand. In addition, how the data is buffered and moved around in very large memory arrays can impact performance. In many cases, SAS 9 performs better than previous releases when given the same job; however, the multi-threaded aspects of SAS 9 add new tuning considerations to many deployments. While the ideal is to achieve performance gains immediately with SAS 9, you must first tune your jobs, your SAS installation, and the hardware that runs SAS to get the maximum benefit.

SURVEY OF THREADED SAS 9 COMPONENTS

Some SAS 9 procedures and engines have been modified with thread technology. (Other SAS 9 components leverage the new Threaded Kernel capabilities as well, but discussion of those is outside the scope of this paper.) By reviewing these components and learning about their enhanced capabilities, you can make appropriate choices for optimizing your SAS applications.

The latest release of SAS introduces SAS Scalable Architecture (SSA). SSA includes a set of threaded kernel services known internally as the SAS 9 Threaded Kernel (TK). The functions that make up TK are not directly exposed to programmers writing applications with SAS 9. Instead, the SAS R&D staff members use these functions to make SAS components multi-threaded. These functions can exploit multiple processors simultaneously by dividing work across multiple threads in ways similar to those described earlier in this paper.

When considering SAS procedures, only some procedures were targeted for TK enhancement because of the complexity of rewriting legacy code to function using parallel programming techniques. The following procedures that are used to process large amounts of data are enhanced with TK technology: SORT, SUMMARY/MEANS, SQL, TABULATE, REG, DMREG, DMINE, GLM, and LOESS.

Ray (2003) discusses the details of Base SAS multi-threaded procedures in “An Inside Look at Version 9 and Release 9.1: Threaded Base SAS procedures.” Ray’s paper explains how threading can reduce the real time to completion and reviews the impact that several new options have on these procedures. The strategies of these procedures in SAS 9 align with the discussion of multi-threading and parallel programming in this paper. Furthermore, during the revision of these Base SAS procedures, developers changed how utility I/O is handled and improved PROC SORT, PROC SUMMARY, and PROC SQL to reduce conflicts that previously existed between reading and writing utility files. Ray notes that “a new predictive read-ahead model allows sort utility files to be read back at nearly the same rate as a straight sequential read.” All together, these enhancements make it less possible for large jobs to become I/O-bound.

Cohen (2002) gives similar treatment to SAS/STAT and Enterprise Miner procedures that were rewritten for SAS 9. For these procedures, Cohen says “enhancements encompass both algorithmic improvements and modifications to exploit multiprocessor hardware.” The enhancements focus on linear regression, analysis of variance, local regression, robust regression, and logistic regression. Cohen states that the new code employs “algorithms that exploit the greater [amounts of RAM and disk] space available” on current generation machines. These computational strategies lead to “efficient memory access during some time-critical computations” that result in “baseline speedups independent of the threading speedups obtained from using multiple CPU boxes.” Cohen has observed a few cases of super-linear speedup when comparing SAS, Version 8 to SAS 9. However, he points out that the problem size and shape, the options specified, the system load, and the hardware used affect specific results.

Both authors share the results of performance experiments. The experiments show significantly improved performance. In some cases, the experiments show a near-perfect speedup. But, both authors point out that their results should not be considered general benchmarks on which to set universal expectations for SAS 9 performance. All cases tend to be unique. The input given to a procedure and the environment in which the procedure runs clearly impact how well a job performs. Each author’s paper acknowledges the importance of appropriately tuned I/O and memory storage on procedure performance. Furthermore, Ray details new system and procedure options that can be used to tune SAS jobs that use these procedures.

Cohen and Ray conclude that threading services in SAS 9 represent revolutionary growth in the SAS System. As a result of the performance gains observed through experimentation, anyone using these procedures can expect to see performance gains with selective application. It is important to note that scalability results are affected by the type of analysis, the size of the problem, and the configuration of hardware. “SAS 9 provides the potential for significant improvements to your throughput. In order for scalable software to succeed, you must pay close attention to … details such as I/O layout, data partitioning, and utility file placement.”

Not all parts are multi-threaded in these procedures. PROC SQL and some of the statistical procedures have been updated only for specific cases. In fact, some procedures detect cases when running multiple threads is not optimal; when detected, the procedure runs in a single-threaded context. Although procedure performance varies depending on input attributes, platform, and hardware, you can expect the results of your job to be consistent.

Several engines have been enhanced in SAS 9 to take advantage of threads. These include the BASE engine, the Scalable Performance Data Engine (SPDE), and the SAS/ACCESS to Oracle engine. Doninger (2002) discusses the performance advantages of SPDE in “Up and Out: Where We’re Going with Scalability in SAS Version 9”. Plemmons (2003) reviews the SAS/ACCESS Interface to Oracle scalability enhancements in “Scaling SAS Data Access to Oracle RDBMS”. In general, these engines have been improved to read data on multiple threads; and, when used with some of the procedures mentioned above, additional speedups can be achieved. In these cases, the engine uses multiple threads to partition and simultaneously read the data, feeding it to multiple Worker threads in the partnering procedure.

If your job is performance-critical, you should review the behavior of your job running in SAS 9 to determine if tuning is needed. As Ray points out, “more than with any previous version of SAS, the phrase ‘your mileage will vary’ applies.” STIMER statistics provide a well-known means of measuring the performance of a SAS job. In addition, SAS 9 provides an Application Response Measurement (ARM) interface and macros documented in *SAS Language Reference: Concepts*. These features, along with any performance-monitoring tools that you generally use to optimize your system, provide a sufficient toolset for optimizing SAS jobs.

CONCLUSION

Multi-threaded systems and parallel programming are common tools in modern software applications, and are used to enhance the scalability and performance of large jobs. SAS 9 includes a core set of components enhanced to run in multiple threads and fully exploit SMP hardware architecture. In many cases, the improvements made to these components impact how I/O and memory systems are used, regardless of how and when threads are exploited. The result is increased performance in a variety of applications.

The parts of SAS 9 that include parallel-processing algorithms execute multi-threaded code paths by default, so SAS 9 will exploit multi-processor "out-of-the-box" architectures. However, generalizations of performance gains are hard to make. Savvy users will keep in mind that variability in results can be expected. Fortunately, ample documentation and tools exist to effectively tune any SAS 9 installation for optimal performance. With a basic understanding of how multi-threading techniques are applied in SAS 9 and through intelligent application of the functions and features available, you can expect to have greater success with this version of SAS than with any previous release.

REFERENCES

Cohen, Robert. 2002. "SAS® Meets Big Iron: High Performance Computing in SAS Analytic Procedures." *The Twenty-Seventh Annual SAS Users Group International Conference Proceedings,* Orlando , FL http://www2.sas.com/proceedings/sugi27/p246-27.pdf (January 1, 2004).

Doninger, Cheryl. 2002. "Up and Out: Where We're Going with Scalability in SAS® Version 9." *The Twenty-Seventh Annual SAS Users Group International Conference Proceedings*, Orlando, FL, http://www2.sas.com/proceedings/sugi27/p279-27.pdf (January 1, 2004).

Plemmons, Howard. 2003. "Scaling SAS® Data Access to Oracle® RDBMS." *The Twenty-Eighth Annual SAS Users Group International Conference Proceedings*, Seattle, WA http://www2.sas.com/proceedings/sugi28/151-28.pdf (January 1, 2004).

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 (January 1, 2004).

SAS Institute Inc. 2002. **SAS OnlineDoc 9**. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

I would like to recognize expert contributions to this material by the following SAS staff members:

- Robert Ray and Scott Mebust— Developers for the SORT and SUMMARY procedures
- Robert Cohen— Expert on SAS/STAT® and Threading
- Paul Kent and Jim Metcalf—Content Reviewers
- Allen Langlois—Source Material on Threading and SAS Threaded Kernel Technology

CONTACT INFORMATION

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

David Shamlin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Work Phone: 919-531-7755
Fax: 919-677-4444
Email: david.shamlin@sas.com
Web: http://support.sas.com/rnd/base/index.html

TRADEMARK CITATIONS

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 registered trademarks or trademarks of their respective companies.