SAS® Meets Big Iron: High Performance Computing in SAS Analytic Procedures

Robert A. Cohen
SAS Institute Inc.
Cary, North Carolina, USA

Abstract

Version 9 targets the heavy-duty analytic procedures in SAS® for high performance computing enhancements. These enhancements encompass both algorithmic improvements and modifications to exploit multiprocessor hardware. This paper provides a survey of this development and the performance gains obtained in several procedures in SAS/STAT® and Enterprise Miner™. Some general scalability issues are discussed and their practical implications in providing high performance scalable implementations in the following SAS facilities are addressed:

- linear regression
- analysis of variance
- local regression
- robust regression
- logistic regression

Introduction

Current-generation servers are equipped with multiple CPUs, vast amounts of RAM, and large-capacity, high-bandwidth file systems. Achieving scalable performance on these "big iron" boxes requires specially designed software. In Version 6, SAS introduced the SPDS (Scalable Performance Data Server) to exploit such hardware for data services. With Version 9, SAS is embarking on an evolutionary strategy to best exploit such hardware in a much broader class of SAS solutions. Underlying this work is the new SSA (SAS Scalable Architecture) that provides technologies to take advantage both of multiple CPUs and multiple I/O channels.

This paper describes how SSA services are being leveraged to provide scalable performance in some of the heavy-duty modeling procedures in SAS. Since this work requires substantial modifications to the legacy procedure code, several challenging problems arise. In almost all cases, it is infeasible to get a job to execute entirely in parallel, and this has some significant scalability implications. These issues are explored in the first part of this paper, providing a basis for understanding the results of scalability experiments presented in the second part.

Multithreading and Parallelization

Threads are parts of programs that execute sequentially, and a program is multithreaded when it can create more than one such sequential path simultaneously. A thread is active when the operating system is free to schedule it for processing. The number of threads that a program can create does not depend on the number of processors on the hardware on which it runs — multithreaded programs can execute on single-processor machines, and unthreaded programs execute on multiprocessor boxes. If a program creates more active threads than there are processors available to execute these threads, then the operating system allocates interleaved slices of time to each thread on the available CPUs. For programs to work correctly, the work done in an active thread needs to be independent of the work that can be done by any other simultaneously active thread. Ensuring that this is the case is the programmer's responsibility, and programs with this property are called thread-safe.

On hardware with more than one CPU, multithreading provides a mechanism for a program to exploit more than one CPU simultaneously. By creating multiple simultaneously active threads, the program enables the operating system to schedule these threads concurrently on more than one CPU. When this happens, the program is said to be processing in parallel.

In order to maximize CPU utilization without incurring unnecessary overhead, a multithreaded program will often create the same number of active threads to perform a task as there are processors available to it. However, this is not always the best policy, and sometimes creating more active threads than CPUs

available allows a program to better balance the work load across the CPUs or use other system resources (such as I/O channels) to best advantage.

Scalability Measures and Ahmdal's Law

There are many different notions of scalability. For example, in a real time transactional database, you may be interested in how the performance scales as the number of simultaneous requests grows. Another notion is that of problem scalability, which measures how an algorithm scales as the size of the problem grows. For example, you may be interested in knowing how a sorting application behaves as the number of observations to be sorted increases. In both of these cases, you are interested in scalability as the size of the problem itself varies. In contrast, this paper focuses on how the time required to solve a fixed problem scales as the number of CPUs and/or I/O channels increases.

On multiprocessor machines, the *total* CPU time used by all the CPUs is usually larger than the CPU time used by the same job using a single processor. However, for jobs that can execute mostly in parallel, the time spent on the respective CPUs overlaps, and so the real (wall clock) time to complete the job decreases even as the total CPU time increases. Since the goal of parallel processing is to reduce the real time a job requires, it makes sense to define measures of scalability in terms of real time.

There are two commonly used measures of scalability. The first of these is "speedup," which is simply how fast a given job completes using p processors as compared to using one processor. More precisely, if t_j denotes the real time required to execute a given job on j parallel processors, the speedup S_p for a job scheduled on p processors is defined as

$$S_p = \frac{t_1}{t_p}$$

A second measure of scalability is "efficiency," which indicates how well a job is able to exploit the p CPUs available. For example, if a job runs on a two processor machine but uses only one of the CPUs at any time, then the efficiency of using the processors is only 50%. More precisely, the efficiency percentage E_p of a job on a p processor system is defined as

$$E_p = 100\frac{S_p}{p}$$

If a job could be carried out completely in parallel then you would expect $t_p = t_1/p$, in which case the speedup S_p would be p and E_p would be 100%. This ideal case is known as linear scalability. It is usually impossible to subdivide a real job into work that can execute completely in parallel. This fact can be expressed as

$$t_1 = t^U + t^S$$

where t^S is the fraction of the work that can be carried out in parallel and t^U is the rest of the work that always keeps only a single processor busy. If you ignore the overhead of scheduling and synchronizing the work on the multiple CPUs and assume that the parallelizable section scales perfectly, then

$$t_p = t^U + \frac{t^S}{p}$$

This relationship shows that there is an inherent limit to scalability — no matter how many CPUs are available the job will always take at least t^U. Let $r = \frac{t^S}{t_1}$ denote the parallelizable fraction of the job. Then some algebra shows that in terms of r the ideal speedup is given by

$$S_p = \frac{p}{(1-r)p + r}$$

This last expression is known as Amdahl's law. If you know the parallelizable fraction for a given job, you can use this law to give an upper bound for the speedup you will obtain by running this job on a multiprocessor machine. Figure 1 illustrates this speedup bound for a few values of the parallelizable fraction r.

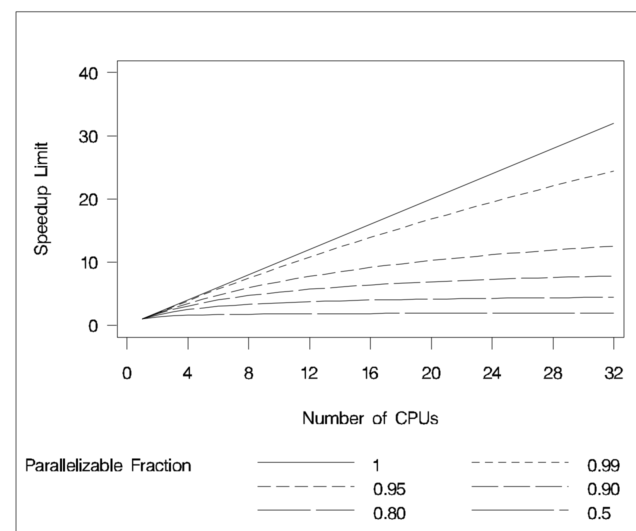

Figure 1. Amdahl's Law

Scalability and Problem Size

Amdahl's law illustrates the difficulty in achieving good scalability for a particular job on powerful servers with many CPUs. You can see in Figure 1 that even for highly parallel jobs ($r \geq 0.90$), the speedup soon falls well behind the number of processors and asymptotes at $\frac{1}{1-r}$. So for example, if $r = 0.9$ then the speedup will always be less than 10 no matter how many CPUs are available. At first sight this appears to limit the scalability you will observe on the larger servers available on the market today, which can have upwards of 64 processors. However, this viewpoint ignores a pertinent effect pointed out independently by Moler (1987) and Gustafson (1988). They noted that as the problem size grows, so often does the parallelizable fraction r, and in fact for many jobs r goes to 1 as the problem size increases. This potentially saves the day for achieving scalability for the very large problems that can be tackled on these "Big Iron" boxes — as you exploit the hardware's large memory and disk resources to tackle ever growing problem sizes, you simultaneously are able to exploit the large number of CPUs available.

At the other end of the problem size spectrum you will often not see much speedup as you move beyond one processor for "small" problems. This occurs not only because the parallelizable code fraction is frequently small in such cases, but also because the overhead of creating and scheduling multiple threads starts to overwhelm the benefits of the small amount of parallel computation that is being done. In fact, to keep performance from suffering for very small problems, many multithreaded codes are designed to switch to using unthreaded code paths in such cases, even when multiple CPUs are available.

Baseline Speedups

The scalability measures discussed in the previous two sections compare performance of programs executing in one thread with the corresponding programs executing in multiple threads. Multithreading of SAS procedures requires significant modifications to the existing legacy code. Usually these modifications in and of themselves do not enhance performance. However, in some cases, the multithreaded procedures run significantly faster than the original legacy code, even when a single thread is used on a one processor machine. There are several reasons why this happens.

Firstly, particularly in the area of I/O, new SAS services have been developed that can be exploited to improve performance even when used in a single execution thread. For example, procedures can now access data in blocks of multiple observations as opposed to one observation at a time, as in earlier releases. For PROC REG jobs that have many observations and few variables, this change in data access can yield about a two times speedup on some hardware.

A second reason for seeing baseline speedups with some of the new code has to do with space-time tradeoffs. The legacy code that the new code replaces was developed at a time when the memory and disk space constraints were much tighter than on current generation machines. The relatively vast amounts of RAM and disk space on more modern computers enable the new code to employ algorithms that exploit the greater space available to improve performance. Such performance gains are particularly evident in PROC DMREG where some logistic regressions run about three to four times faster than in earlier releases, on single CPU boxes. These larger memories have also been exploited in PROC GLM and PROC LOESS, which now use data structures that facilitate efficient memory access during some time-critical computations.

Note that these baseline speedups are independent of the scalable speedups obtained from using multiple CPU boxes. Hence, if you run a job that uses a thread-enabled SAS procedure on a multiple CPU machine you may obtain a speedup over the Version 8 time that is *better* than linear. For comparison purposes it is useful to define V9/V8 speedup for a job scheduled on p processors as the number of times faster the job runs in Version 9 as compared to the same job in Version 8.

Performance Results

The following sections detail typical speedups that you can obtain with some of the thread-enabled analytic procedures running on multiple CPU platforms. All these results were obtained on a Sun Enterprise Midframe Server configured with 16 gigabytes of RAM. For most of the tests, this server was configured with eight 750 MHZ processors, but where explicitly noted, a configuration with twelve processors was used.

Note that the results presented are for specific jobs. You need to exercise caution if you try to draw generalized conclusions from these performance results. Here are some factors that can dramatically affect the scalable speedup of a particular job:

- **Problem size.** Not all invocations of a SAS procedure will have the same parallelizable code fraction. As noted earlier, this fraction depends not only on the details of the analysis, but also on the problem size.

- **Problem shape.** The code sections that consume the time for a PROC invocation can vary widely depending on the shape of the data being analyzed. Often for data with many observations and few variables, much of the time will be spent doing I/O, so that achieving scalability in these cases will depend on having scalable I/O. At the other extreme, the same analysis for data with fewer observations but many more variables might require an insignificant time for I/O but a great deal of time for computation.
- **Options specified.** Most SAS analytic procedures can perform a variety of related analyses. In some cases, these analyses will use code paths that are not yet multithreaded.
- **System load.** Unless SAS is running on a dedicated server, there will usually be other tasks on a machine competing for system resources.
- **Hardware.** The scalability you will see for a given job will vary with the hardware used. Commonly, midrange servers are SMP (Symmetric Multiprocessor) machines where all the CPUs have equal access to a large shared memory. As processing speeds outstrip memory access times, fast memory caches are used to provide the CPUs with timely access to memory. More recently, to accommodate an ever growing number of CPUs, NUMA (Non-Uniform Memory Access) architectures have been introduced. In NUMA architectures, the CPUs are arranged in clusters with each cluster having its own local memory. While each CPU can "see all the memory," memory access within a cluster is significantly faster than across clusters. There are complex interactions between the topology of the CPU arrangements, cache sizes, and user code. In some cases a job will scale well on one vendor's server but less well on some other hardware architecture. SAS is working closely with several hardware vendors in an ongoing effort to learn how to best exploit each vendor's specific hardware.

Scalability in PROC LOESS

The LOESS procedure fits local regression models by piecing together low-degree polynomial regressions through local neighborhoods of the data, where the number of points in each local neighborhood is controlled by a smoothing parameter. Computing confidence limits in a LOESS model requires significantly more time than fitting the model itself, as modeling the error distribution depends on examining the data set as a whole. To help alleviate this bottleneck in many LOESS jobs, these confidence limit computations were the first section of the PROC LOESS code to be multithreaded. Furthermore the new code is itself substantially faster than the equivalent Version 8 code, even when no threading is used.

Figure 2 shows scalable speedups for fitting a LOESS curve with confidence bands through data with 4,000 points. The smoothing parameter value in this test was selected algorithmically requiring 18 different LOESS models to be evaluated before the multithreaded computations for degrees of freedom were done. The parallelizable fraction of this test is 0.95. Notice how the observed speedups reflect Amdahl's law by falling increasingly short of linear scalability as more CPUs are employed.

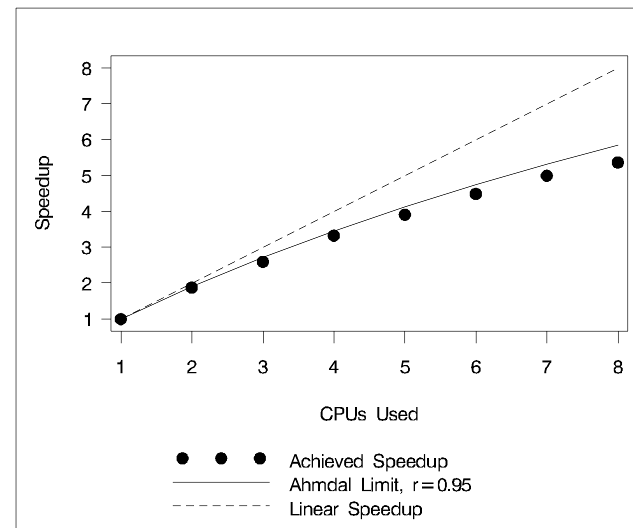

Figure 2. Scalable Speedup in PROC LOESS with Automatic Smoothing Parameter Selection

Figure 3 shows the scalable speedup for the same test except that in this case a single smoothing parameter value was specified. Hence, the unthreaded section of the code involves only a single LOESS fit before the multithreaded computations for degrees of freedom were done. This gives a parallelizable fraction of 0.99. Notice the improved scalability resulting from the higher parallelizable fraction for the test results shown in Figure 3 as compared to Figure 2.

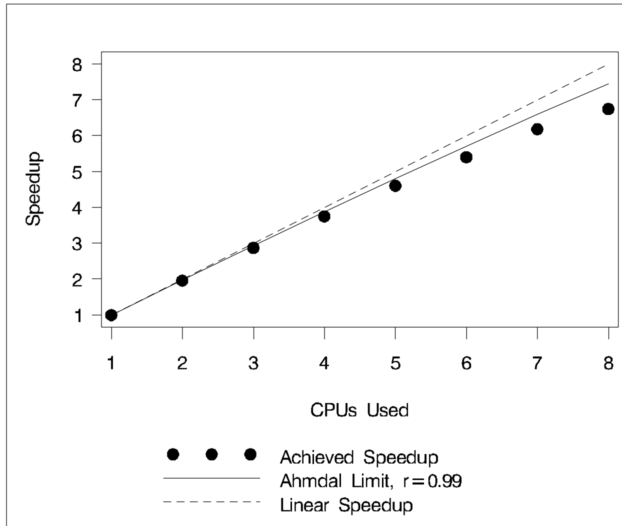

Figure 3. Scalable Speedup in PROC LOESS with a Specified Smoothing Parameter

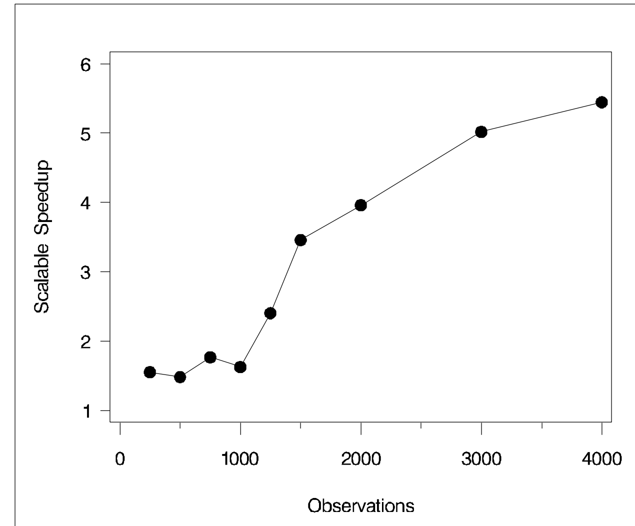

Figure 5. Scalable Speedup of a LOESS test with Problem Size

As noted in the "Scalability and Problem Size" section on page 3, the parallelizable fraction of a test usually grows with the size of the test. This is illustrated in Figure 4, which shows how the parallelizable fraction of the same LOESS test varies as the number of observations increases. The corresponding scalable speedups using eight CPUs are shown in Figure 5. Notice how scalability improves as the problem size grows.

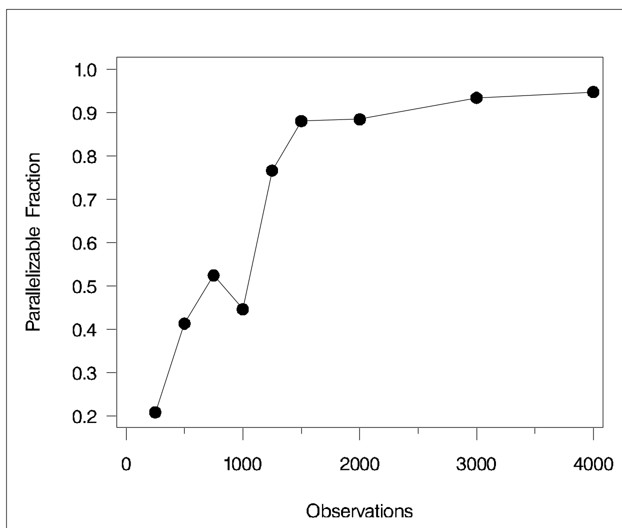

Figure 4. Parallelizable Fractions of a LOESS test with Problem Size

Scalability in PROC GLM

The GLM procedure fits linear models involving classification effects. Even without interaction, each level of a classification variable corresponds to one parameter in the model, so that many GLM analyses often involve models with a large number of parameters. The multithreading enhancements to PROC GLM have focused on alleviating the bottlenecks in fitting such models. These bottlenecks include inverting the SSCP matrix, deriving estimable linear functions, and computing effect tests. Figure 6 shows the scalability results obtained for one such test with 6,000 observations and four classification variables. The model, which included main effects and some interactions, had 2,000 parameters.

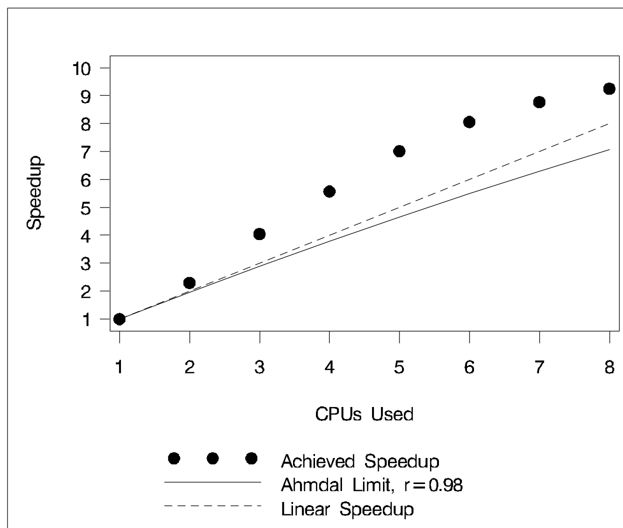

Figure 6. Scalable Speedup in PROC GLM

Figure 6 reveals a surprising fact—the scalable speedups obtained in this test are *better* than the Amdahl upper bound and in fact are *better* than linear! The reason for this is probably due to improved cache utilization in multiple threads. Due to the division of work between multiple threads, each processor may need to reference smaller contiguous blocks of memory than would be the case if the problem was not subdivided. When these smaller memory blocks are able to fit within each processor's fast memory cache, the number of expensive cache misses during the computation can be dramatically reduced and this produces the additional speed improvement. These beneficial cache-related effects depend on the memory hierarchies of the hardware, the algorithm, and the specific problem, and will not occur uniformly across the spectrum of these parameters.

Scalability in PROC REG

The REG procedure fits linear regressions and produces many diagnostics about these fits. The time spent in fitting such models is concentrated in three tasks, namely I/O, forming the SSCP (sum of squares and cross-products) matrix, and inverting this matrix. All three of these tasks have been multithreaded, making a broad spectrum of PROC REG jobs scalable. Even so, there are time-consuming sections of some PROC REG jobs that have not yet been multithreaded. One example is the Furnival and Wilson algorithm that PROC REG uses in all-subset based selection methods.

In order to obtain parallel reads of SAS data sets, you need to use one of the multithreaded partitioned data access engines available in Version 9. An inline implementation of SPDS has been incorporated as a new engine into BASE SAS software. Using this SPDE (Scalable Performance Data Engine), SAS data sets are stored in multiple partitions and each partition can be accessed in a separate thread. True parallel access occurs when these threads access partitions that are serviced by independent I/O controllers. However, even when multiple partitions are accessed using a single I/O channel, read-ahead buffering while other computational work is being done can sometimes be used to effectively serve up the data asynchronously in multiple threads.

Parallel data access is necessary for achieving scalability for REG jobs with many observations and few variables. As the number of variables grows, the fraction of time PROC REG spends doing I/O relative to the rest of the work diminishes. For data sets with many variables, some scalability can be obtained even when data is accessed serially with the base engine.

Figure 7 shows the scalability results of a PROC REG job that performs stepwise variable selection among 500 variables for data with 50,000 observations. In this test the data are accessed serially with the base engine, and the parallelizable fraction of the job is 0.93.

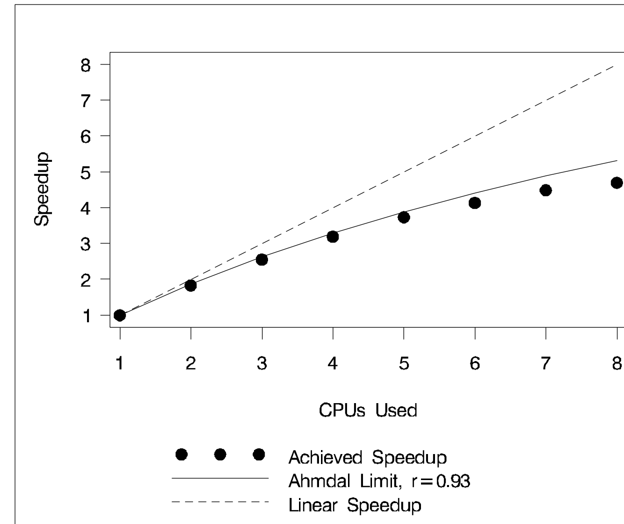

Figure 7. Scalable Speedup in PROC REG with Serial I/O and Many Variables

Figure 8 contrasts the V9/V8 speedup of the multithreaded code over the unthreaded code with the scalable speedup. There is about a 2.5 baseline speedup in this test, a result of improvements in one of the linear algebra routines on this host. Note that you will not obtain this baseline speedup on all plat-

forms.

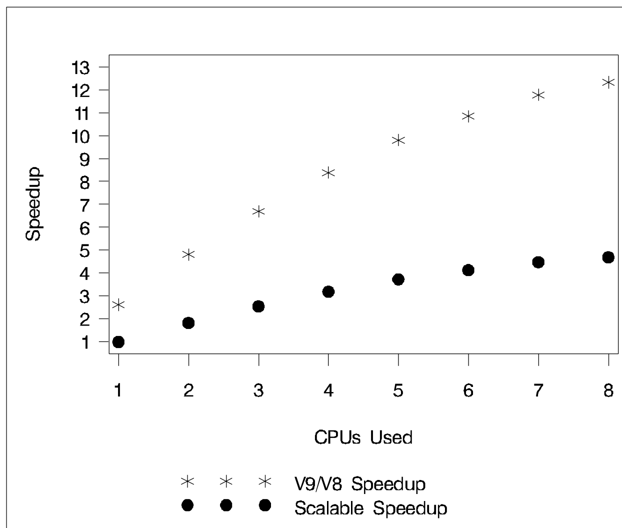

Figure 8. V9/V8 and Scalable Speedups in PROC REG with Serial I/O and Many Variables

Figure 9 shows the scalability results of a PROC REG job that fits a model with 20 regressors for data with four million observations. In this test, almost all the time is spent doing I/O and forming the SSCP matrix. With parallel data input being obtained using the SPDE, this highly scalable test has a parallelizable fraction of about 0.999. In this test, the SUN Enterprise Midframe Server was configured with 12 processors.

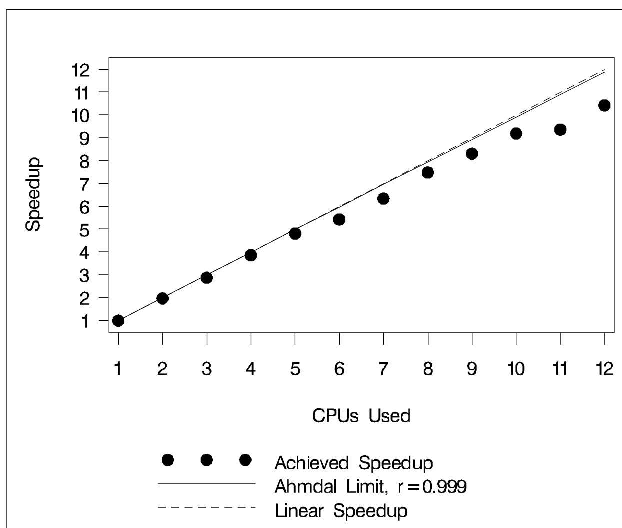

Figure 9. Scalable Speedups in PROC REG with Parallel I/O and Few Variables

Scalability in PROC DMREG

The regression node in Enterprise Miner software uses the DMREG procedure to fit a variety of linear and nonlinear regression models. Each iteration of the optimization methods used in fitting the nonlinear models requires at least one pass through the data to form the required gradient vectors and/or hessian matrices. This work consumes the majority of time in many DMREG jobs, and the multithreading enhancements have focused on alleviating this bottleneck.

If you use one of the engines that can access partitioned-data, the initial pass through the data is multithreaded. To avoid having to scale the data and handle missing values on every subsequent data access, the design matrix, represented sparsely when appropriate, is formed and partitioned into a number of utility files that can be accessed in separate threads. This avoids unnecessary preprocessing duplication and facilitates multithreaded access to the data during the optimization iterations, even when the SAS data set itself is not partitioned.

Figure 10 shows the scalability results of a PROC DMREG job that fits a logistic model to data with one million observations. The target is binary and there are 38 interval and two classification variables. In this test, the SAS data set is stored and accessed with the base engine, and hence the initial data pass and SSCP formation are not multithreaded. The parallelizable fraction for the test is 0.86.

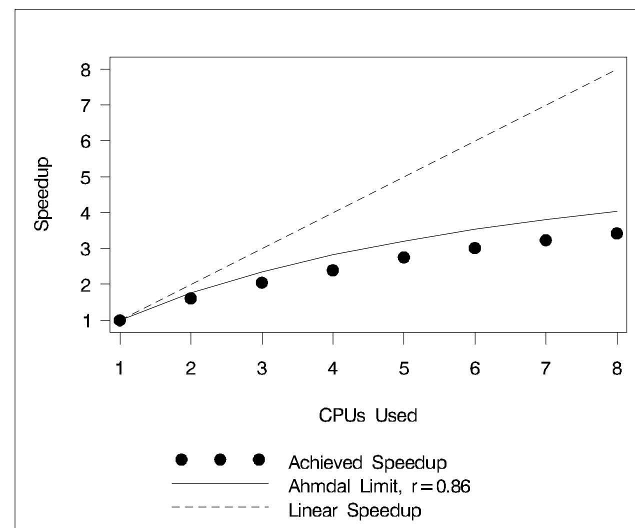

Figure 10. Scalable Speedup in PROC DMREG with Serial Data Access

Figure 11 contrasts the V9/V8 speedup with the scalable speedup. Note that you get a baseline speedup of about 2.7 times in this test.

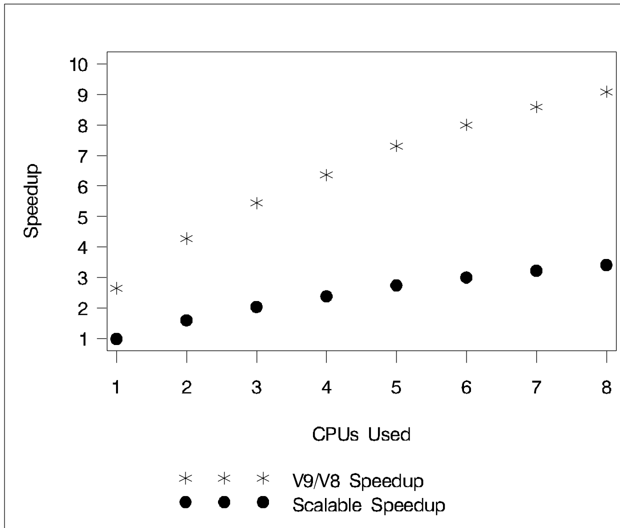

Figure 11. V9/V8 and Scalable Speedups in PROC DMREG with Serial Data Access

Figure 12 shows the scalability results of a PROC DMREG job that fits a logistic model to data with 500,000 observations. The target is binary and there are 50 continuous predictors and 15 classification variables, each having 20 levels. The data are accessed in parallel by using the SPDE. In this test, the SUN Enterprise Midframe Server was configured with 12 processors.

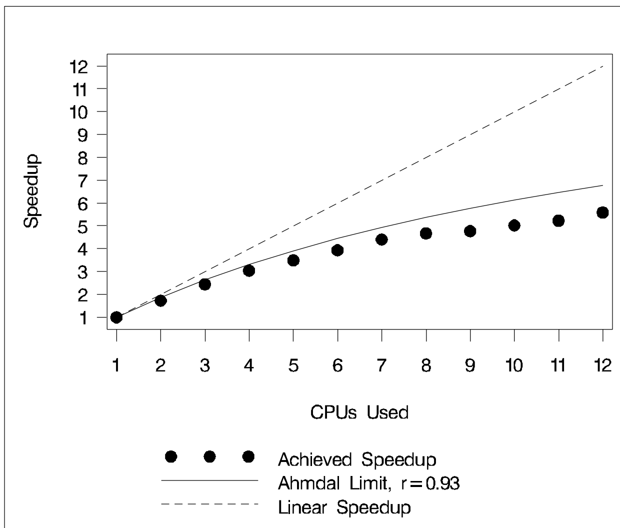

Figure 12. Scalable Speedup in PROC DMREG with Parallel Data Access

Figure 13 contrasts the V9/V8 speedup with the scalable speedup. The V9 code exploits the sparsity of the design matrix, contributing to the 3 times baseline speedup.

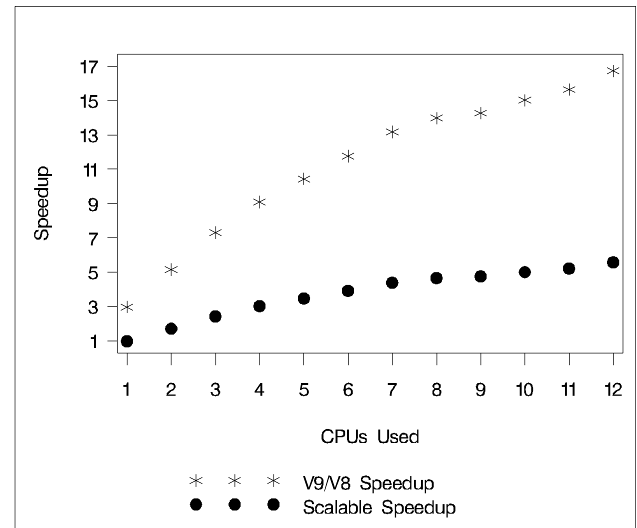

Figure 13. V9/V8 and Scalable Speedups in PROC DMREG with Parallel Data Access

Scalability in PROC ROBUSTREG

The ROBUSTREG procedure is a new SAS/STAT procedure in Version 9. It contains a variety of methods for computing robust regression fits for data corrupted with outliers. One of the more time-consuming methods available is the high-breakdown least trimmed squares fit, and the initial multithreading work has focused on the main bottlenecks in this section of the code. Future work will extend the multithreading to the other methods available in PROC ROBUSTREG.

Figure 14 shows the scalable speedups in fitting a least trimmed squares model using PROC ROBUSTREG for data with 20 variables and 500,000 observations. In this case the parallelizable fraction of the job is 0.87.

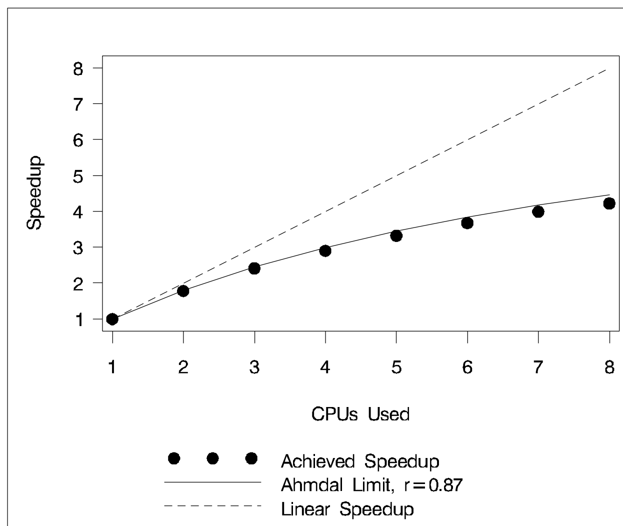

Figure 14. Scalable Speedups in PROC ROBUSTREG

Conclusions

Ongoing development at SAS aims to make parallel processing, parallel I/O, and algorithm enhancements available to speed your time-to-solution. With these evolutionary changes, expect to see performance gains from selected applications using multiprocessor hardware. The scalability you will see will vary with the type of analysis and problem size. As your problem sizes grow, you will be able to increasingly take advantage of additional CPUs. Scalability results are also sensitive to specific hardware configurations, making generalizations difficult. However, experimentation indicates that you will get very good scalability on hardware with up to four processors. Less consistent scalability results have been obtained in some initial tests on hardware with more than eight processors.

References

Gustafson, J. (1988), “Reevaluating Amdahl's Law,” *Communications of the ACM*, 31, 532-533–293.

Moler, C. (1987), “A Closer Look at Amdahl's Law,” *Intel Technical Report*, TN-02-687.

Contact Information

Robert A. Cohen, SAS Institute Inc.,
SAS Campus Drive,
Cary, NC 27513.
Email: robert.cohen@sas.com

SAS, SAS/STAT, and Enterprise Miner are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.