Paper 388-2008

Making sense of Multi-core Processor technology for SAS Environment

Carl E. Ralston, Intel Corp., Cary, NC

ABSTRACT

The computer industry has been rapidly transitioning from CPUs (chips) with a single processing core to CPUs with dual-, quad- and multi-core configurations. This has changed traditional thinking of SMP servers and has propagated to both desktop and laptop systems. Multiple concurrent jobs and multi-threaded applications are techniques to utilize this change. This paper/presentation will introduce and explain this direction and it's relevance to the SAS environment. Scalability will be discussed by using examples such as Enterprise Miner scoring, Forecast Server workload, random number parallelization and the inclusion of Intel's Math Kernel Library in SAS 9.2. Recommendations on how to configure the CPUs, memory and input/output subsystems of your system for optimal SAS software performance will be included.

INTRODUCTION

In a few short years, the computer industry has been rapidly transitioning from CPUs (chips) with a single processing core to CPUs with dual-, quad- and multi-core configurations. This is changing the way we should be designing applications and impacts the way we deploy systems where the server could have 4 times as many processors and approximately that much or greater increase in compute power.

Some terminology:
A **multi-core** CPU combines two or more independent cores into a single package comprised of a single piece of silicon integrated circuit, called **die**. A **dual-core** device contains two independent microprocessors and a **quad-core** device contains four microprocessors.

Socket is a physical and electrical location where the integrated circuit die plugs into a motherboard. We talk of single, dual and quad socket systems that could have processor chips with single, dual or quad core device per socket. Example: A four socket system with quad-core integrated circuits would be a 16 CPU system.

We are not talking about **hyper-threading** in the context of multi-core technology. Hyper-threading may or may not be implemented for a given microprocessor design.

We need to think of application design and deployment is the following three basic ways:

- Concurrent multiple jobs (Symmetric multiprocessing, SMP)
- Parallel decomposition (Changing a job/application or work flow to parts that can run independently. The subparts may need to synchronize and come together as part of the overall work flow.)
- Multithreading, Design the application to use multiple threads of execution. This for the most part the responsibility of SAS Institute R & D organization. Several other papers have and will discuss this. This will not be the focus of this paper.

The old single socket system would be a uni-processor. Today, that single socket system could have one, two or four CPU cores. This is true for laptops, desktops and server systems available today.

SCALING OF SAS FORECAST SERVER

The SAS Institute Enterprise Excellence Center (EEC) has created a "Forecast Server (FS) Performance Test Suite". These tests highlight the number of time series that can be processed per hour. You can read more about Forecasting and Forecast Server at references 2, 3 & 4. Note: there is a SAS Forecast Studio Java-based GUI client that is **not** used for this workload. It is what customer could use to generate the server based code like this workload represents.

The FS test suite is designed to provide customers with an understanding of the performance they can expect and the system resources required to support their forecasting activities. The FS test suite highlights the **number of time series that can be processed per hour**.

Three specific real-world data volumes and structures are defined for these tests:

SMALL:	50,000 series, 100 variables, 500 by-groups ~ 466 MB

MEDIUM:	500,000 series, 500 variables, 1000 by-groups ~ 4.5 GB
LARGE:	3,000,000 series, 2000 variables, 1500 by-groups ~ 27 GB

Each series contains 3 years of daily data.

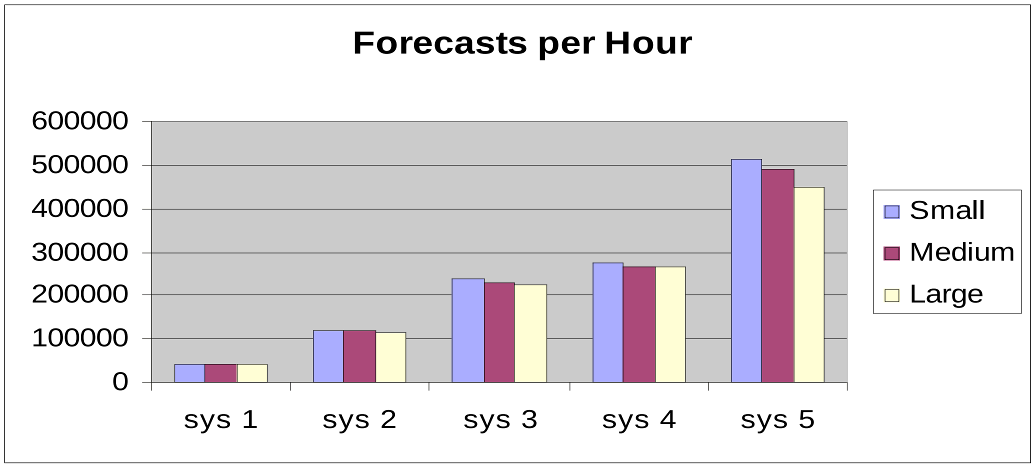
There are three distinct steps to each forecasting job:

1) Model selection: The tests choose the time-series model that best fits each series, form a list of 13 candidate models. The candidate models include 6 ARIMAX models and 7 Exponentials Smoothing models.
2) Model Fitting: After FS chooses the best model for the series, each parameter in the chosen model is estimated.
3) Forecasting: This step generates the actual forecasted values for each series.

The tests can be run from one SAS FS job running to 16 SAS FS jobs running concurrently. (It may scale beyond 16 but I currently haven't tested that. Given the needed hardware resources, I see no reason for it not to scale beyond 16). Initial performance analysis shows that it is about 98% CPU bound (one job consumes one CPU(core), ie 8 jobs keep 8 cores busy).

	sys 1	sys 2	sys 3	sys 5	sys 4
Small	40751	119803	237476	273069	513472
Medium	40049	118171	229673	266926	490289
Large	39383	115759	222614	264544	446857

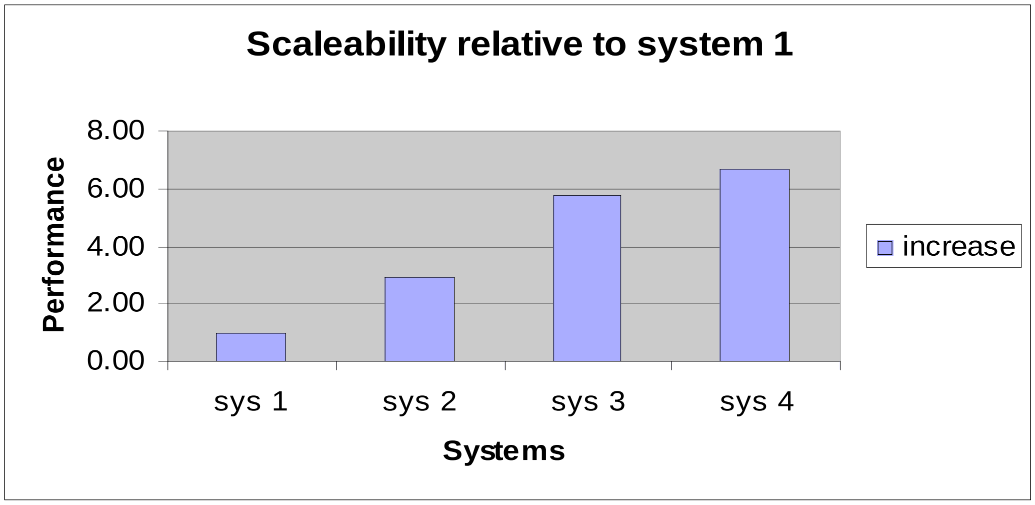
Forecasts per Hour as raw data



Forecasts per Hour as charted data

	sys 1	sys 2	sys 3	sys 4	sys 5
increase	1.00	2.94	5.74	6.69	12.06

Performance increase relative to system 1 as raw data



Performance increase relative to system 1 charted.

It is worth noting that system 2 should have a 2 times performance increase because of the doubling of the number of cores. However we get an extra .94 because of the switch in basic core architecture from Pentium 4 to the Core 2 Duo architecture. This is approximately 50% performance increase from system 1 single core at 3.6 GHz to the system 2 Core 2 Duo at 2.66 GHz. Note that GHz is not always the telling factor in overall system performance. The remaining systems are based on the Core 2 Duo architecture.

Performance runs have been made on the following hardware:

- System 1 Intel® Xeon® processor @ 3.6 GHz, 800MHz FSB, 16KB L1, 2MB L2 cache, 8GB, 2 socket, last and highest performance single core
- System 2 Core 2 Duo Intel® Xeon® processor 5100 @ 2.66 GHz, 1333 FSB, 32KB L1, 4MB L2 cache, 8GB, 2 socket
- System 3 Quad-Core Intel® Xeon® processor 5300 Series @ 2.67 GHz, 1333 FSB, 16KB L1, 8MB L2 cache, 16GB, 2 socket
- System 4 Quad-Core Intel® Xeon® processor 5x00 Series @ 2.83 GHz, 1333 FSB, 16KB L1, 8MB L2 cache, 16GB, 2 socket (45 nm of system 3)
- System 5 Quad-Core Intel® Xeon® processor 7300 Series @ 2.93 GHz, 16KB L1, 8MB L2 cache, 32GB, 4 socket

All the above systems will use the same storage, that is SmartArray 6402 with one shelf of 14 Ultra320 72GB 15,000 RPM disks. Hyper-threading was turned off if the system supported it. All systems will be running Windows Server 2003 x64 with current updates. SAS 9.1.3 SP4 (SE19) was used. More information on Intel platforms can be found at http://www.intel.com/products/server/platforms/index.htm?iid=serv_body+sys

References:
1) EEC web site: http://sww.sas.com/wwm/eec/ (maybe internal only)
2) Forecasting: http://www.sas.com/technologies/analytics/forecasting/index.html
3) Forecast Server: http://www.sas.com/technologies/analytics/forecasting/forecastserver/index.html
4) High-Performance Forecasting: http://www.sas.com/technologies/analytics/forecasting/hpf/index.html

SCALING OF SAS ENTERPRISE MINER SCORING
SAS HIGH PERFORMANCE SCORING STRATEGY(SHiPs)

SAS Institute's Enterprise Miner product provides predictive analytics to enterprises by automatically creating high-quality models from baseline data and then analyzing new data and scoring it using the same models. Enterprise Miner offers a variety of methods for creating and viewing the results of models: sampling, scoring, descriptive statistics, and graphs and visualizations. More about the product is available at http://www.sas.com/technologies/analytics/datamining/miner/index.html.

SAS Enterprise Miner's architecture is highly scalable to multiple cores. Intel developers were able to achieve over 4X speedup on an SAS Data Miner test case. The case consisted of ten million observations with 100 different models. They used different optimization techniques and scaled from single-core to quad-core processors in highly reliable Intel® Core™ Architecture-based multi-processor (MP) servers.

SAS Enterprise Miner uses the SAS High Performance Scoring (SHiPs) strategy to score observations. SHiPs offers a scalable, parallel scoring methodology for a symmetric multiprocessor (SMP) architecture. The basic strategy is as follows:

1. Start a battery of scoring engines, one engine per CPU.
2. Initialize the engines with meta data about the scoring task.
3. Divide the scoring task into an optimal number of models per engine (CPU).
4. Start a subset of the models in each scoring engine.
5. Wait for an engine to finish a subset and start new models. Repeat until all models are started.
6. Wait for the last task to finish.
7. Terminate the scoring engines.
8. Generate a report.

SAS Enterprise Miner includes the software tools to partition the scoring task to optimize performance in an SMP environment. More information on the SHiPs strategy can be found at [1]. An example of SHiPs scoring optimization can be found at [2].

Tests
Performance testing on Intel® Architecture-based platforms comprised SAS software and data optimizations to improve performance on a single system (system 3), plus scaling from single-core to quad-core platforms to evaluate scalability performance.

Input
The test cases consisted of the following: a) 100 models b) 10 million observations (records) were used for optimization testing c) 50 million observations were used for scalability testing d) The number of engines running at one time equaled the number of processors in the test system.

Platforms
We evaluated performance on Intel® processor-based single-core(system 1), dual-core(system 2), and quad-core, four-socket (MP) system(system 5), providing four, eight, and 16 cores respectively. Intel® MP platforms are four-socket systems designed for enhanced scalability and reliability required in intense computing environments

Optimizations
The SHiPs code was designed for scalability. SAS' macro language for SHiPs allows scoring designers to adjust many attributes to optimize the scoring process for multiple cores. An example of how these attributes are used can be found in [1].

Data Set Partitioning
When scoring massive data sets across multiple engines and models, chunks of data are read into cache to start the scoring tasks. As scoring progresses, some scoring tasks finish faster than others, requiring more data to be read into cache. As this out-of-sync state accelerates, it eventually leads to frequent reads from storage and disk thrashing to feed the faster tasks, degrading overall system scoring performance.

By partitioning data reads, the code eliminates frequent calls to storage for new data at different disk locations, while continuously providing enough data to keep all the engines busy. This strategy optimizes scoring performance and I/O throughput. Experimentation and testing using different partition sizes resulted in finding an optimal partition size of 400 KB, delivering a 60 percent performance increase in scoring (Figure 1).

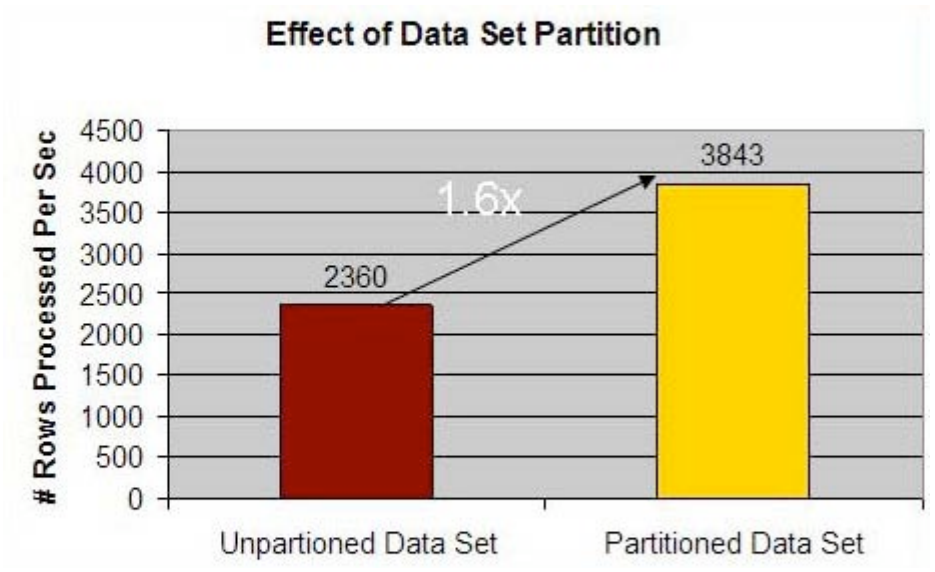


Figure 1. Effect of Data Set Partitioning on SHiPs Scoring

SGIO
In addition to classic buffered I/O operations, scatter/gather I/O (SGIO) is a feature option of the SAS system. SGIO (also called vectored I/O or direct I/O) accelerates reads and writes directly to application memory. SGIO delivered a 23 percent performance increase (Figure 2). The combination of data partitioning and SGIO doubled the performance.

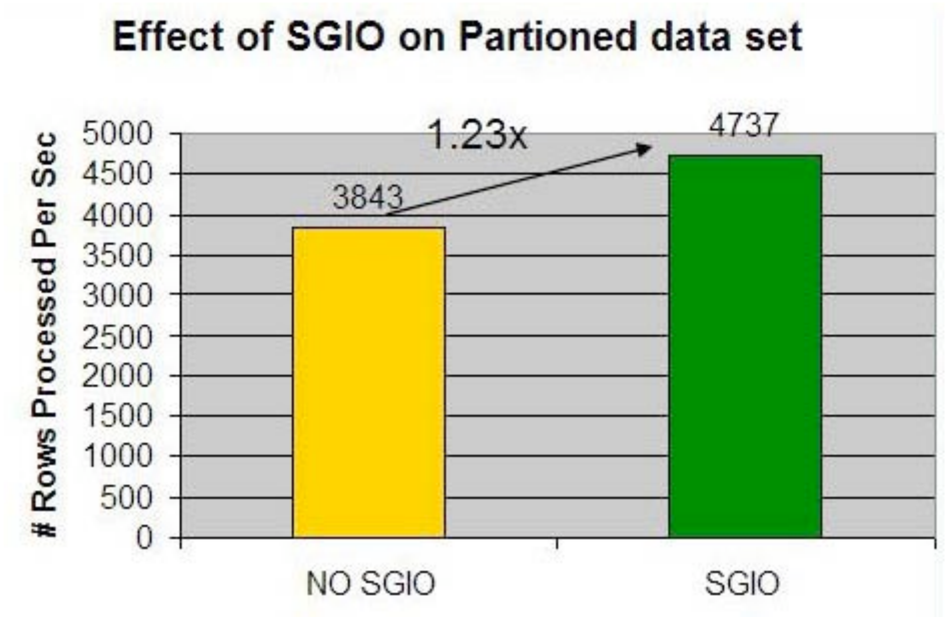


Figure 2. Effect of SGIO on Partitioned Data Set in SHiPs Scoring

Model Distribution
One of Enterprise Miner's attributes governs how models are distributed over available scoring engines. Model distribution can have a significant impact on performance, because it better utilizes more compute resources when distributed optimally. The change in model distribution increased performance by 27%. See the paper for details on this topic.

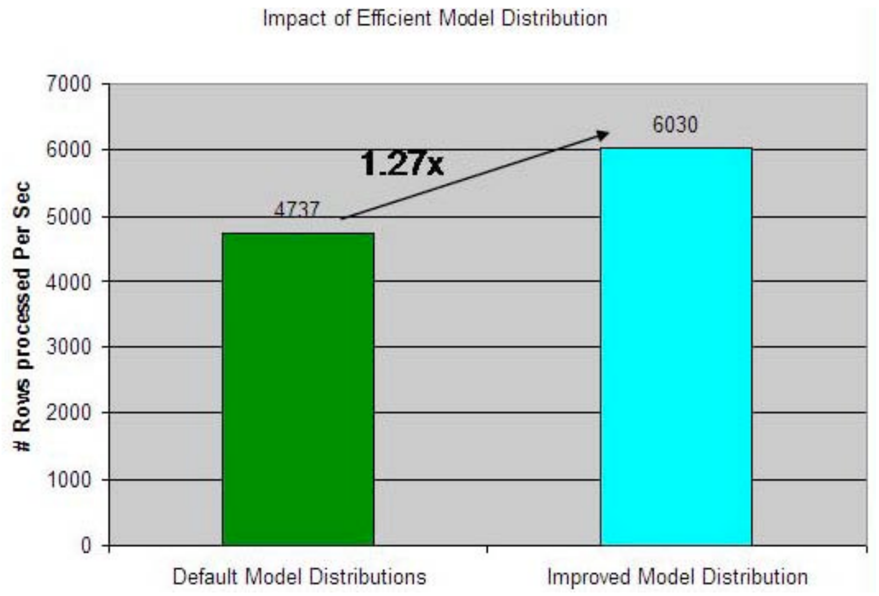


Figure 3. Effect of Modified Model Distribution on SHiPs Scoring

Scalability

Scaling from 64-bit Intel® Xeon® processor platforms (single-core) to Quad-Core Intel® Xeon® processor-based platforms illustrates the scalability of Enterprise Miner and the capabilities of Intel Xeon processor-based systems to accommodate large workloads for predictive analytics and other types of demanding data mining applications. All scalability tests were done on 50 million observations and 100 models. Figure 6 shows the impact on scoring performance from the aggregate effect of data partitioning, SGIO, model distribution optimizations, and platform scaling from single-core to quad-core Intel processors.

The 2.71X speedup from system 1, with Intel Xeon processor (single-core), to system 2, with Dual-Core Intel Xeon processor 5100 Series, shows the impact Intel® Core™ architecture brings to performance along with two cores instead of one. System 3, with Quad-Core Intel Xeon processor 5300 Series and 1.68X more performance, further illustrates the value of added processing resources from additional cores. Note that the Quad Core Intel Xeon processor 5300 Series is based on the Intel® NetBurst™ microarchitecture rather than the Intel Core microarchitecture.

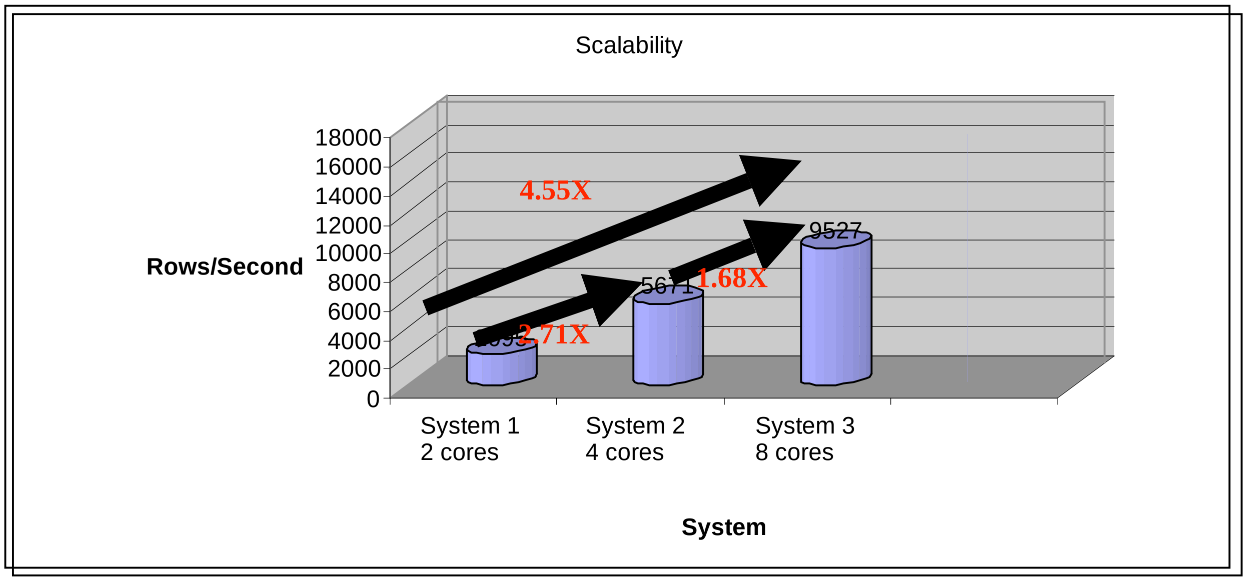


Figure 6. Results of Scalability Testing on Optimized Code

Conclusion
Optimized SAS Enterprise Miner scoring code running on Quad-Core Intel Xeon processor 7300 Series-based MP platforms can turn massive amounts of data into critical business information for enterprise decision makers overnight. With optimized data partitioning and instituting SGIO alone, throughput doubled from SAS default code configurations. Parallelizing all models through more effective distribution added another 27 percent performance boost. As a result of these optimizations, Dual-Core and Quad-Core Intel Xeon processor-based platforms enable highly scalable performance over 64-bit Intel Xeon processor-based servers with over 4X performance on Quad-Core Intel Xeon processor 5300 Series.
References:

1. SAS High Performance Scoring, SHiPs, Multi-Model Parallel Scoring. SAS, 2007
2. SAS Parallel Scoring Optimization. Unisys, 2005

For the complete detail see: **http://www.sas.com/partners/directory/intel/papers.html**

GAINING SPEEDUP IN A PARALLELIZABLE WORKLOAD WITH DEPENDENCE ON RANDOM NUMBER GENERATION

Parallelize Random Numbers Carefully Many statistical algorithms are highly parallel, providing real opportunities for developers to express concurrency and accelerate their code. These algorithms, however, often are used in conjunction with a set of pseudorandom numbers to randomly select sample points from a large set of data.

When it comes to parallelizing random number generation, programmers must take considerable care in assuring the random number set is of the same quality when generated in parallel. Simply by launching multiple threads, each running a random number generator function that is initialized with a randomly selected seed, does not guarantee the numbers, when the subsets are aggregated, will be free of overlap or uniformly distributed. Unless the generator is thread-safe, it cannot maintain continuity of numbers between threads.

Programmers at Intel parallelized simulation code used by the Food and Drug Administration (FDA) that contained dependency on a set of random numbers. The code runs on SAS* software and was written using the SAS macro language. Parallelization enabled over 14X speedup of the simulation by running it in multiple instances on 16 cores in a test system based on four Quad-Core Intel® Xeon® processors.[1] We also benchmarked performance on an 8-core test system.

The code used the language's random number generator function to generate the number set. Without special modifications to the function or the need for additional oversight code to check the quality of the set, programmers were able to parallelize the simulation code and random number generation such that, whether created serially or in multiple instances, the code created the identical set of numbers for the simulation.

A Couple Methods In the FDA's simulation code, a set of calculations is repeated thousands of times, with each iteration dependent on a different set of random numbers. The code is highly parallelizable. In fact, each iteration could realistically be run on its own processor core, significantly reducing the simulation run times.

Parallelizing the code meant also parallelizing the generation of random numbers. However, the native random number generator function was not designed to be thread safe; it had no continuity between different threads or instances. Generating a set of random numbers from subsets created in parallel could possibly result in subsets overlapping each other with identical numbers. Such a set of numbers would risk "silent errors" and misinterpreted results.

There are several ways to successfully generate a high quality set of random numbers in parallel tasks. Programmers can use existing thread-safe, parallelized random number generator functions, such as the Scalable Parallel Random Number Generators (SPRNG) library discussed in the article "Thread-safe Random Number Generation." They can also run multiple instances of the native random number function as discussed in this article. The method discussed here ensures a high quality set of numbers. Plus, it generates the same random number set, irrespective of how many instances are used—whether the numbers are generated sequentially (one instance) or in parallel (any number of instances). That is,

$$\text{random\_sequence}_{\text{serial}} = \text{random\_sequence}_{\text{parallel1}} + \text{random\_sequence}_{\text{parallel2}} + \text{random\_sequence}_{\text{parallel3}} + \text{random\_sequence}_{\text{paralleln}}$$

Parallelizing the FDA's Code Random number generation occurred early in the FDA's simulation code. The simulation consumed the vast majority of the run time; random number generation was an insignificant part.

The FDA provided the initial seed for the random number generator. From this seed, the random number function generated the first set of numbers. The simulation ran to completion, and then ran again, generating a new set of numbers based on the last number from the previous run. Thus, there was continuity through all the random numbers generated.

To parallelize the simulation code (and random number generation), the team evenly distributed the code's several thousand simulations over the number of cores in the test systems. They determined from the total number of simulations and random numbers for each run, the simulations required a total of 32 million random numbers for their test cases. This gave them a size of the set of random numbers in which overlap was prohibited.

To safely generate all the random numbers over all instances of the simulation, they needed 16 safe seeds (for the 16-core system) to initialize the instances. Each instance would generate 2,000,000 numbers. For the first instance, they used the seed supplied by the FDA to generate numbers 1 through 2,000,000. They needed 15 more seeds: the second seed would initialize generation of numbers 2,000,001 to 4,000,000; the third seed started numbers 4,000,001 to 6,000,000, and so on.

To randomly select 15 other seeds would not guarantee a high-quality set of numbers. Instead, they would use 15 seeds from the serially generated set, which would then generate all the same numbers in the serial version.

Discovering the Seeds To find and save the additional seeds, the team created a multi-loop routine that generated 32 million random numbers. The routine used a function based on the random number generator executed in the simulations, but the function generated both a random number *and* the seed value for that number.

The FDA seed value initialized the routine. It then generated 32 million numbers and saved the seed values at 2,000,001, 4,000,001, 6,000,001, and so on, until 15 additional seeds were saved. These seeds initialized the random number generators for the parallel version of the simulation.

Random number algorithms consistently generate the same random number sequence from a given seed. Since the seeds in the parallelized simulations used seeds from the serial set of numbers, the parallelized instances consistently generated the same set of numbers as the serial version.

Results Based on tests on system 3 and system 5, parallelizing the FDA code enabled us to achieve significant speedup (S) of the simulation job. Generating 32 million random numbers took about 6 seconds. The seed generation time was insignificant compared to running the simulation code, inducing a very minor overhead on the job. On system 3, we were able to run the job up to about seven times faster on eight cores, while ensuring we did not compromise the quality of the random number sequence. For system 5, we achieved a 14.29X maximum speedup on 16 cores. Furthermore, the method the team used consistently generated a high-quality set of identical numbers, which can be very valuable in debugging code.

An important benefit to this method is that parallelizing and achieving improved performance did not require major modification of the code. With highly regulated industries, such as healthcare, major modifications to a product can require re-testing to prove the change did not significantly affect its function. The method described here did not alter any algorithms used in the simulation, and it generated the same set of random numbers used by the serial version.

Automation The team hard coded their work on the FDA simulation as a proof of the technique. But, all the steps they used can easily be automated with simple routines to discover how many CPUs are available, discover the needed seeds, and launch the required instances with the appropriate seed values.

Conclusion Many statistical algorithms can be easily parallelized. When they depend on a set of random numbers, programmers must take special care to generate the same high quality set of random numbers in parallel as done in the serial code.

The method described above parallelizes random number generation by distributing the generation of total random numbers over multiple instances and initializing each instance with a seed from the serial set of numbers. This affords an opportunity to produce significant speedup in a program without sacrificing the integrity of the random number sequence or changing the core of the code itself.

MATH KERNEL LIBRARY(MKL)

The SAS System version 9.2 has included the use of Intel's Math Kernel Library (MKL). There are about 100 SAS procedures that call (make use of) MKL. Examples of SAS procedures are: AUTOREG, CORR, FREQ, GENMOD, GLM, GLIMMIX, IML, LOGISTICS, MIXED, NLMIXED, PHREG, RISK, TIMESERIES and UNIVARIATE. The MKL provides routines and functions that perform a wide variety of operations on vectors and matrices including sparse matrices and interval matrices. The performance increase varies greatly for different SAS procedures and size of data. Smaller (~ few 10s) data sizes did not show as much performance improvements and larger data sizes showed greater performance improvements. The performance range was from ~10% to 40 times for different procedures and usage. This was based on comparing SAS 9.1.3 SP4 without MKL to SAS 9.2 with MKL. The MKL is supported on both Windows and Linux operating systems plus Itanium, Xeon and Pentium 4 processors.

A reference for Intel's MKL is:

http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm

CONTACT INFORMATION:

Carl E. Ralston

Software Systems Engineer of Intel Corporation onsite at SAS Institute Inc.

RA424

SAS Campus Drive

Cary, NC 27513

919-531-5905

Carl.Ralston@SAS.com or Carl.E.Ralston@Intel.com