Paper 277-27

Pushing the Envelope: SAS System Considerations for Solaris/UNIX in Threaded, 64 bit Environments

Maureen Chew

Abstract: With the release of 64 bit support in the SAS system, we investigate the issues and implications of having a large application address space; what can take advantage of 64−bit support, what are the benefits, tradeoffs, and multi−user considerations from a system resource perspective.

The target audience is SAS users concerned with overall application performance or UNIX systems managers who primarily support SAS application users. The work was done on Solaris but is generally relevant to SAS applications running on any UNIX platform.

Contents

Performance Monitoring Tools

All the performance monitoring tools discussed in this paper are either freely available or bundled with the Solaris Operating Environment.

Command line tools bundled w/ Solaris:

- *prstat(1)* − new in Solaris 8 and is similar to the freeware "top" program, but can do microstate accounting and consumes less resources. This is a very useful utility and does not require root access. Common invocations include:

  prstat −a, prstat −u <user>, prstat −am (shows user lock time),

  prstat −L −m −p <SAS process>
- ***iostat(1) −xtc 5*** − display disk stats; watch for any disks which display consistent service time (svc_t) > 50ms or greater than 20% busy.
- ***vmstat(1)*** − virtual memory & system stats; A new option, −p in Solaris 7, gives you a breakdown as to reasons for paging. You have a serious RAM shortage if executable pages are being paged out. Solaris 8 has a new memory management subsystem which should result in near zero scan rates. Additionally, the free memory value in Solaris 8 vmstat is also now a true measure of how much memory really is free for new application usage and can be used for headroom estimation.[1]

SE Toolkit

This is a freeware package written by Rich Pettit and Adrian Cockcroft, two of the best known Sun performance experts. Based on an interpretive C like language, the distribution comes with an interpreter and numerous scripts. Although many scripts are included, the ones I've found most useful are ***zoom, perfmon, mpvmstat, xio***, and ***live_test. zoom*** gives you a visual overview of all key aspects of the system in a traffic light fashion. ***live_test*** is the non GUI version. I also really like the disk utilization traffic lighting feature in ***zoom***. Using this, I was able to discover that I erroneously configured one of my "high performance" SAS data volumes in a concatenated fashion instead of as a striped volume. ***xio*** is useful because it gives an indicator as to the cumulative I/O pattern : <100% randsvc implies sequential access while >100% is random access.

E**xactly what does a 64−bit SAS application buy you?**[2]

SAS Version 8.2 (supported on Solaris 7 or higher) is available as both a 32−bit and 64 bit release. SAS Version 9 (supported on Solaris 8 or higher) is a 64−bit release only. If Solaris is booted into 64−bit mode, all 32−bit applications can run. However, the converse is not true. From here on, SAS Versions will be referred to as V9, V8.2, V6.12.

Performance Improvement

A number of PROCs in V9 and V8.2/64 can now take advantage of systems with large memory configurations and can potentially run substantially faster (**SORT, GLM, REG, MDDB, IML, LOESS** among others).

Other than that, there is nothing inherent about the 64−bit port that would provide automatic "free lunch" SAS performance boosts. If you are moving to a 64−bit port, we recommend going

1 Capacity Planning for Internet Services

2 SAS TechNote TS660

directly to V9 for several reasons:

- V8.2/64 was the initial 64−bit port for Solaris and thus a number of performance optimizations were deferred until V9
- V9 can natively read any/all V6.12 and forward data sets
- Extended support for SAS/ACCESS engines

PROCs such as **SORT** can improve greatly if the data set can be sorted in memory as opposed to paging in through utility files since accessing data in memory is much faster than requesting data from disk.

SORT does (as of all of SAS) a good job with it own internal virtual memory management. If you turn down the *MEMSIZE/SORTSIZE* knob so that your own particular job doesn't consume too much of a system wide resource, it will do so in an efficient manner. The effect of doing this though, is that **SORT** will then no longer be able to sort "internally" (all in RAM) and will then have to depend on writing to utility files for various sorting and merging phases. In this case, **SORT** is considered to have been done "externally".

For external sorts, increasing *SORTSIZE* will create larger "bins" that **SORT** uses. This can increase efficiency. However, if the additional memory utilized, starts pushing the system towards RAM shortfalls, then all gains will be more than completely lost.

However, optimization can get tricky for cases when complex system interactions come into play. There are two areas where these interactions produce behavior which is counterintuitive to to the concept just discussed (the more you put in RAM, the faster it goes). The first area is that sometimes, applications which use a large address space (even with sufficient configurations) access memory in such a way that introduces address translation thrashing in the kernel (TLB miss), unoptimal caching, or other inefficiencies related to how free memory is coalesced.

Secondly, sometimes applications which write to utility files in lieu of RAM counterintuitively go "faster" because the Solaris file system cache kicks in and data that gets written to/read from "disk" utility files end up being accessed at memory speeds.
Decreasing *SORTSIZE*s can cause smaller and potentially more contiguous memory allocations and can lead to the unexpected result where **SORT** runs faster externally than internally.

SAS Version 9, Through the Looking Glass of Multi−Threaded Applications

SAS/SPDS (Scalable Performance Data Server) has been supported since V6.12 and is a scalable, fully threaded SAS data store. Originally, developed on the Sun platform, excellent results have been seen from its usage.

Version 9 introduces the concept of a threaded kernel (TK) into the SAS system. Internal SAS developers can utilize TK as a foundation for building multi−threaded SAS PROCs. The SPDS engine has been integrated into the SAS System and can be invoked from a libname directive:

```
libname mylib sasspds "SPDS data directory";
```

This SPDS "lite" version will be included in base SAS but the fully featured product will continue to be available as a separate package.

With the initial release, the TK "hot" PROCs which have been re−written to take advantage of TK include **DMREG, GLM, IML Matrix Multiplication, LOESS, REG, ROBUSTREG, SORT, SQL, SUMMARY** (check the V9 SAS "What's New" documentation for a more comprehensive list) . Additionally, V9 introduces a Java Virtual Machine (JVM) based on JDK 1.4 will live in−process with the SAS kernel. In V9, the interfaces will be exposed only to Institute developers but there is ongoing discussion with the SAS user community in the area of exposing access to the JVM through the PROC and Data steps. While this is all interesting, the relevancy is that the JVM is a fully threaded virtual machine and potentially presents other avenues for surfacing scalable applications or components.

V9 has 2 global options, *THREADS and CPUCOUNT*, which control the threading behavior. If the underlying OS supports threading, then the option *THREADS* will be set and *CPUCOUNT* will default to the number of CPUs in the system. If you would like to suppress the use of threads, set *NOTHREADS*. Users can set these variables in any of the standard ways:

- SAS command line invocation (sas −NOTHREADS, sas −CPUCOUNT 4)
- Within the SAS program `options NOTHREADS; options CPUCOUNT=4;`
- Within the SAS config file, sasv9.cfg

The SAS system will create threads or light weight processes (LWPs) in direct proportion to the value of *CPUCOUNT*.

Attention Systems Managers: On large enterprise systems with many CPUs and many simultaneous V9 SAS users, *CPUCOUNT* should be set in $SASROOT/ sasv9.cfg to some reasonably small number (ie: 4 or 6). Users can override on an as needed basis. If the default is not changed, 25 simultaneously V9 users on a 48 CPU system could unknowingly spawn 1200+ (25 x 48) LWPs and could potentially consume the system.

While we have seen some solid scalability for certain V9 PROCS, keep in mind that your scaling mileage could greatly vary. First and foremost, this will vary based on a

PROC by PROC implementation. Then, many variables can affect scalability measurements: cumulative load, RAM configuration, I/O configuration, resource management partitioning, network load, etc. On SAS applications that scale well, you might see a SAS log file similar to this entry:

```
real time            4:14.61
user cpu time        23:36.91
system cpu time      0.34 seconds
```

This means that while the wall clock time was 4 minutes, 14 seconds, this process actually accumulated a total of 23+ minutes of user CPU time. When the real time value reported is much smaller than user cpu time, that is a good indicator of the scalability of the PROC. If the values exhibit a large differential in the other direction (real time is much larger than user cpu time), you can't tell if the PROC is particularly *not* scalable or if there is some other resource constraint.

In an ideal environment, SAS users would want to run every SAS job against every relevant data set in and iterate from 1 to the number of CPUs in the system in order to determine where the scalability curve starts flattening out. In other words, at what point does increasing *CPUCOUNT* be one of diminishing returns? At that point, the user is consuming extra system resources with no return. Additionally, to effectively run baseline tests such as this, you have to be the exclusive user on the system. However, this would like be tedious and impractical.

In large multi−user environments, SAS users have an obligation to have some basic understanding of the resources they utilize. The burden can't be left on the systems managers to "figure it out". I've had many conversations with systems admins on performance issues whose SAS user community has thrown the problem over the wall and are unwilling to make themselves available for problem diagnosis. The solutions to many issues often require iterative, trial and error efforts between both groups. Typically, systems admins have little working SAS knowledge and when asked to try different SAS options, they are walking in uncharted territory. On the other hand, the same is true for many SAS users who are not familiar with UNIX/Solaris commands or perhaps may be connecting remotely via **SAS/CONNECT**. So, the message to both SAS Users and Systems Managers: WORK WITH EACH OTHER; ITS GOING TO BE VERY IMPORTANT. There, now I feel better.

With minimal effort, SAS users should know these basics:

- how many CPUs are in the system
- how many LWPs does their application consume (*CPUCOUNT* could be set in any number of places and can vary throughout a SAS program)
- how much RAM is being used

Determining #CPUs:

```
$ /usr/sbin/psrinfo
0       on-line   since 01/10/02 08:53:41
1       on-line   since 01/10/02 08:54:06
2       on-line   since 01/10/02 08:54:06
3       on-line   since 01/10/02 08:54:06
```

Determining #LWPs

Determine SAS PID (Process ID)

```
$ ps
  PID TTY      TIME CMD
 2022 pts/4    0:00 ksh
 2038 pts/4    1:32 sas
```

Use ***prstat(1)***

```
$ prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE
TIME  CPU PROCESS/NLWP
  2036 allen     140M  133M run     52    0
0:01.28  26% sas/5
  2038 maureen   140M  133M cpu2    52    0
0:01.18  24% sas/6
  2042 maureen   280M  273M cpu1    52    0
0:01.25  23% sas/6
  2048 mike      140M  133M cpu3    51    0
0:01.24  22% sas/5
  2067 margo     140M  132M run     41    4
0:00.02 3.5% sas/9
  2103 sasmau   1528K 1272K cpu0    58    0
0:00.00 0.0% prstat/1
  2049 william   140M  133M cpu3    51    0
0:01.24  22% sas/5
```

Now, check CPU utilization on a per LWP basis

```
$ prstat -L -m -p 2038
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX
ICX SCL SIG PROCESS/LWPID
  2038 maureen   33 0.0  -   -   -   -  0.0  -    0
71   0   0 sas/4
  2038 maureen   24 0.0  -   -   -   -  0.0  -    0
24   0   0 sas/5
  2038 maureen   22 0.0  -   -   -   -  0.0  -    0
21   0   0 sas/6
  2038 maureen   21 0.0  -   -   -   -  0.0  -    0
25   0   0 sas/3
  2038 maureen  0.0 0.0  -   -   -   -  0.0  -    0
0    0   0 sas/2
  2038 maureen  0.0 0.0  -   -   -   -  100  -    0
0    0   0 sas/1
```

So, from the above output, a user can tell:

- there are 4 CPUs in the system
- there are a total of 6 SAS jobs running at this snapshot in time
- maureen's SAS job consumes 140 MB RAM, spawns 6 LWPs but only 4 are active (which is not a surprise since CPUCOUNT=4); the number of LWPs in existance at any given time can change throughout the course of the PROC.

Performance Cookbook

Now that we've told the SAS users how to run minimal performance characterization, we're now ready to look at more sophisticated usage of the tools.

Memory Allocations:

On a CPU bound application, memory utilization should be characterized.

Using ***prstat(1)***, sample the application run until you see peak memory allocation. Typically, multi−hour running procs such as **PROC REG** should peak memory utilization relatively early in the program execution.

```
ctcsun8. prstat 4
   PID USERNAME  SIZE   RSS STATE  PRI NICE
TIME  CPU PROCESS/NLWP
  2184 maureen   591M  585M cpu1     0    0
0:09.01  25% sas/2
   634 root     3512K 2288K sleep   58    0
0:00.04 0.1% automountd/5
   949 root       13M 9104K sleep   58    0
0:00.11 0.0% jre/8
  1010 root     3288K 2848K sleep   58    0
0:00.00 0.0% mibiisa/12
   652 root     2056K 1472K sleep   50    0
0:00.00 0.0% cron/1
   662 root     3144K 2648K sleep   55    0
0:00.00 0.0% nscd/9
   586 root     2960K 1256K sleep   50    0
0:00.00 0.0% keyserv/4
Total: 66 processes, 185 lwps, load averages: 0.92,
0.43, 2.16
```

We notice that RSS is close to SIZE so we're not experiencing RAM contention.

Let's take a closer look at this 591 MB in terms of number and size of memory allocations. The SAS kernel typically allocates mostly anonymous mmap(3)'ed memory. Determine the process id (2184 from above ***prstat(1)*** output) and use this as a parameter to ***pmap(1)*** :
NOTE: A number of pmap entries were trimmed out and the larger ones retained.

```
bash-2.03# pmap -x 2184 >/tmp/pmap.out
bash-2.03# more /tmp/pmap.out

2184:   /d0/v9/sas -memsize 2G test.sas -fullstimer
         Address   Kbytes Resident Shared Private
Permissions       Mapped File
0000000000002000       8       8       -       8
read                [ anon ]
0000000100000000    4896    3816       -    3816
read/exec         sas
00000001005C6000    1024     544       -     544
read/write/exec   sas
00000001006C6000     120     104       -     104
read/write/exec     [ heap ]
.....
FFFFFFFF61C00000   89856   89848       -   89848
read/write          [ anon ]
FFFFFFFF67400000   89856   89848       -   89848
read/write          [ anon ]
.....
```

In some instances, there will be so many mappings that it is necessary to redirect the ***pmap(1)*** output to a file. After doing so, view the first few pages of the file and determine if there is a pattern. With **PROC SORT**, we discovered some interesting considerations. By looking at the first few pages of the ***pmap(1)*** output, we noticed that the allocations typically consisted of 2 different sizes of mappings which we subsequently discovered was directly related to **SORT** options.

With the default **SORT** settings, ***pmap(1)*** output showed mappings for 2 similar sized entries(the majority of allocations were 952 KB and 512 KB).

```
2220:   /d0/v9/sas -fullstimer -memsize=16G
sortbig.sas
         Address   Kbytes Resident Shared Private
Permissions       Mapped File
....
FFFFFFFEADF00000     512     264       -     264
read/write          [ anon ]
FFFFFFFEAE000000     952     952       -     952
read/write          [ anon ]
FFFFFFFEAE100000     512     264       -     264
read/write          [ anon ]
FFFFFFFEAE200000     952     952       -     952
read/write          [ anon ]
FFFFFFFEAE300000     512     264       -     264
read/write          [ anon ]
FFFFFFFEAE400000     952     952       -     952
read/write          [ anon ]
....
```

Since our data set was 9 GB, it seemed that the allocations could have been done more efficiently. After some minimal coercion from SAS R&D responsible for **SORT**, I was given a hint about an undocumented **SORT** option, *KEYBLK*=. For this given build, the default was set to 256 KB and we decided to try 2 MB (*KEYBLK=2048k*).

Note, V9 contains 2 **SORT** algorithms, the parallel **SORT** (referred to as PSORT) new in V9, and the standard pre−V9 **SORT**(referred to as SASSORT). Both **SORT**s have some options unique to each and as such, *KEYBLK* is unique to PSORT. *KEYBLK i*s not a documented nor exposed **SORT** option and could be modified, deprecated, or removed in future SAS releases. However, its worth mentioning because it exemplifies an inductive process used in characterizing memory usage. This paper is not recommending the usage or non−usage of this option.

SORT, **2GB data set, Memory Allocation Comparison**

	Total Memory Allocated	Allocation Sizes	# Allocations
Default *KEYBLK= 64k*	3.225 MB	952 KB 512 KB	Total: 4490 2245 of each size
KEYBLK= 2048k	2.645 MB	7520 KB 2056 KB	Total: 562 281 of each size

The allocation counts were obtained via a somewhat simplistic method of searching for all instances of the specific allocation and piping that output into **wc(1)**.
bash−2.03# cat /tmp/pmap.out | grep 952 | wc −l

From the table above, we see that the number of memory allocations was reduced from ~4500 to ~600. Although the size of the individual allocations was increased, the benefits of the increased efficiency was somewhat evident.

As a interesting side effect, note that the total memory allocation decreased when using he larger *KEYBLK*= value (3.2 MB −> 2.6 MB). (Since this paper was written prior to V9 going production, this might not always be true).

We also see that the overall performance gain was about 30 seconds in a 2.5 minute program. Perhaps it might not be worth the effort to go through the trouble to have discovered this.

LWP (Light Weight Process) CPU utilization

With *CPUCOUNT=4*, and 4 CPUs on the system, we see via ***prstat(1)***, that a particular SAS job has spawned 6 LWPs.

```
ctcsun8. prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE
TIME  CPU PROCESS/NLWP
  2156 maureen   140M  133M cpu0    10    0
0:00.48  92% sas/6
  2158 root     1512K 1256K cpu1    58    0
0:00.00 0.0% prstat/1
   949 root       13M 9104K sleep   58    0
0:00.12 0.0% jre/8
  1010 root     3288K 2848K sleep   58    0
0:00.00 0.0% mibiisa/12
   652 root     2056K 1472K sleep   50    0
0:00.00 0.0% cron/1
   662 root     3144K 2648K sleep   55    0
0:00.00 0.0% nscd/9
   634 root     3512K 2288K sleep   58    0
0:00.05 0.0% automountd/5
```

In most instances, SAS spawns 2−3 LWPs for administrative or housekeeping purposes. So LPWS=6, falls right in line with our expectation since *CPUCOUNT=4*.

prstat(1) −L provides a view on a per LWP basis:

```
ctcsun8. prstat -L -m -p 2156
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX
ICX SCL SIG PROCESS/LWPID
  2156 maureen  100 0.0 0.0 0.0 0.0 0.0 0.0 0.1   0
45   0   0 sas/3
  2156 maureen  100 0.0 0.0 0.0 0.0 0.0 0.0 0.4   0
54   0   0 sas/5
  2156 maureen   98 0.0 0.0 0.0 0.0 0.0 0.0 1.7   0
49   0   0 sas/4
  2156 maureen   98 0.0 0.0 0.0 0.0 0.0 0.0 2.2   0
55   0   0 sas/6
  2156 maureen  0.0 0.0 0.0 0.0 0.0 0.0 100 0.0   1
10  10   0 sas/2
  2156 maureen  0.0 0.0 0.0 0.0 0.0 0.0 100 0.0   0
0  12   0 sas/1
```

and shows that each LWP is realizing 100% (or near) CPU utilization.

The ***prstat(1)*** output below shows an example where there are more multi−threaded SAS jobs than CPUs to service them:

```
ctcsun8. prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE
TIME  CPU PROCESS/NLWP
  2236 allen     140M  133M run     52    0
0:01.28  26% sas/5
  2238 maureen   140M  133M cpu2    52    0
0:01.18  24% sas/6
  2242 maureen   280M  273M cpu1    52    0
0:01.25  23% sas/6
  2248 mike      140M  133M cpu3    51    0
0:01.24  22% sas/5
  2267 margo     140M  132M run     41    4
0:00.02 3.5% sas/9
  2250 sasmau   1528K 1272K cpu0    58    0
0:00.00 0.0% prstat/1
  2249 william   140M  133M cpu3    51    0
0:01.24  22% sas/5
   949 root       13M 9104K sleep   58    0
0:00.10 0.0% jre/8
  1010 root     3288K 2848K sleep   58    0
0:00.00 0.0% mibiisa/12

ctcsun8. prstat -L -m -p 2338
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX
ICX SCL SIG PROCESS/LWPID
  2238 maureen   33 0.0  -   -   -   -  0.0  -    0
71   0   0 sas/4
  2338 maureen   24 0.0  -   -   -   -  0.0  -    0
24   0   0 sas/5
  2238 maureen   22 0.0  -   -   -   -  0.0  -    0
21   0   0 sas/6
  2238 maureen   21 0.0  -   -   -   -  0.0  -    0
25   0   0 sas/3
  2238 maureen  0.0 0.0  -   -   -   -  0.0  -    0
0   0   0 sas/2
  2238 maureen  0.0 0.0  -   -   -   -  100  -    0
0   0   0 sas/1
```

It's useful to realize the "optimal example" is the exact same program as the "overloaded system" example. However, in the latter, we can see that the CPU utilization is no longer at (or close to) 100% but rather averaging around 25%. In this case, you have approximately 6X the number of LWPs competing for the same resources as was available when a single SAS job was running.

When you see CPU utilization on the lower end (<50%), check for system wide cumulative CPU utilization. 2 methods are shown here, ***se mpvmstat*** and ***mpstat(1)***:

```
ctcsun8. se mpvmstat 3
+ /opt/RICHPse/bin/se mpvmstat.se 3
mpvmstat.se  3 second intervals starting at Sat Jan
12 15:12:26 2002
cpu r  b  w  pi  po  sr smtx  in    sy   cs  us  sy
id   swap   free
 0 20  0  0   0   0   0    0  255    0   23 100   0
0 9951936 6795392
 1 20  0  0   0   0   0    2   29   76   38 100   0
0
 2 20  0  0   0   0   0    0   24   26   29 100   0
0
 3 20  0  0   0   0   0    2   32  176   53  97   3
0

ctcsun8. mpstat 3
CPU minf mjf xcal  intr ithr  csw icsw migr smtx
srw syscl  usr sys  wt idl
  0    0   0   61   225  120   23   22    4    0
0    16  100   0   0   0
  1    0   0    4    20    0   27   20    5    1
0    20  100   0   0   0
  2    2   0    3    23    0   29   23    4    0
0     6   97   3   0   0
  3    0   0    2    23    0   24   23    5    0
0     0  100   0   0   0
```

If the per user LWP cpu utilization is not high and the overall system cpu wide utilization is not pegged, then the application is either I/O bound or possibly waiting on lock synchronization.

Memory Utilization

Earlier we introduced a potential performance anomaly resulting from TLB (Translation Lookaside Buffer) miss issues. The function of the TLB is to map virtual memory addresses into actual hardware addresses. TLB misses are caused by scanning through large chunks of memory across wide strides.

One example of this was a case where a very CPU intensive IML program ran in about 70 minutes on an UltraSPARC III based systems and about the same time as an UltraSPARC II system. Needless to say, the performance was disappointing at best since the UltraSPARC III CPUs clock in at almost twice the speed as UltraSPARC II.

Running in 4+ hours, another example using **PROC LOESS** exhibited the same problem.

To determine effects of TLB misses, we use the tool,

trapstat(1M) new to Solaris 9 (or available from your friendly Sun technical advocate for Solaris 8).

The −t option to trapstat will give you % time servicing the TLB misses .

```
bash-2.03# trapstat -t 3
cpu | itlb-miss %tim itsb-miss %tim| dtlb-miss %tim
dtsb-miss %tim | %tim
-----+-------------------------------+-------------
---------------+-----
  0 k|         15  0.0          0  0.0 |      10934  0.1
849  0.1 |  0.3
  0 u|         30  0.0          0  0.0 |      14763  0.2
  0  0.0 |  0.2
-----+-------------------------------+-------------
-----------------+-----
  1 k|         16  0.0          0  0.0 |       3884  0.0
 98  0.0 |  0.0
  1 u|         47  0.0          0  0.0 |        656  0.0
  0  0.0 |  0.0
-----+-------------------------------+-------------
-----------------+-----
  2 k|          0  0.0          0  0.0 |       2536  0.0
  0  0.0 |  0.0
  2 u|          0  0.0          0  0.0 |    2016249 31.0
  0  0.0 | 31.0
-----+-------------------------------+-------------
-----------------+-----
  3 k|         17  0.0          0  0.0 |       5673  0.0
207  0.0 |  0.1
  3 u|         18  0.0          0  0.0 |        165  0.0
  0  0.0 |  0.0
=====+===============================+=============
==================+=====
ttl|        143  0.0          0  0.0 |    2054860  7.9
1154  0.0 |  7.9
```

Note: This example is running V8.2 so that we can limit the process to a single CPU to demonstrate the effects. We would have seen the same thing for V9 with the *NOTHREADS* SAS option or if this particular PROC was not modified to take advantage of the new TK subsystem.

From the ***trapstat(1M)*** −t output, we are showing ~30% time to service TLB misses which is very high. This is from **PROC LOESS** while the **PROC IML** case was showing ~20%. Typically, anything in the 5−10% or higher range on an UltraSPARC III based system is likely to be a problem.

Continuing with this example, we also ran ***trapstat*** without the −t option. Note, that due to probing differences, the −t option will tend to show you numbers that are lower in value.

The output of ***trapstat*** is shown because it matches what ***cputrack(1)*** will output. If you are unable to obtain ***trapstat(1M), cputrack(1)*** is bundled with the Solaris OE.

Don't get too hung up on the numbers, these tools are discussed here as performance problem indicators. Using ***cputrack(1)*** without the −t option, seeing more than ~1M/sec could be a problem.
Note that arguments ***cputrack(1M***) are chip specific and that TLB counters are not accessible via ***cputrack(1M)*** on UltraSPARC II systems.

```
bash-2.03# trapstat 3
vct  name            |    cpu0    cpu1    cpu2    cpu3
---------------------+-----------------------------
 20  fp-disabled      |       0       0       0       0
 24  cleanwin         |       4      27       0       1
 41  level-1          |       4       0       0       0
......
 68  dtlb-miss        |   21863    9012 3508575    7399
 6c  dtlb-prot        |      33      84       0      67
......
127  gethrtime        |       1       2       0       1
140  syscall-64       |       0       3       0       0
---------------------+-----------------------------
ttl                   |   50947   16216 3510468   13548
```

```
bash-2.03# cputrack -c EC_rd_miss,DTLB_miss -p 1933
   time lwp      event      pic0      pic1
  1.490   1       tick   1371206   3735327
  1.420   2       tick         0         0
  0.301   3       tick         0         0
  0.301   4       tick         0         0
  2.510   1       tick   1163973   3411860
  2.420   2       tick         0         0
  0.301   3       tick         0         0
  0.301   4       tick         0         0
```

In summary, the various tool output above was showing:

- ***trapstat −t*** : ~2 M TLB misses / sec
- ***trapstat*** : ~3.5 M TLB misses / sec
- ***cputrack*** : ~3.5 M TLB misses / sec

OK− so we're seeing TLB service time at 30% for **LOESS** and 20% in our **IML** program, now what?

In both these cases, SAS R&D re−coded their algorithms to access memory in a more optimal fashion with dramatic results. One of the program changes was to simply transpose 1 line of code so that matrices iterated over a row basis instead of column basis.

While it may the most optimal solution to wait for a SAS Hot Fix, that may not be timely or practical. There is an "experimental" approach which has been shown to produce very good results in most situations. Called ISMMAP, it is a very simple approach to conceptually simulate the new Solaris 9 feature, MPSS (Multiple Page Size Support).

ISMMAP is built as a shared library which you pre−load on the SAS command line invocation. Utilizing the standard Solaris ISM (Intimate Shared Memory) capability, ISMMAP intercepts certain sized calls to anonymously mmap(3)'ed memory and gives back "large" pages. This results in many, many fewer page address translations that have to be made.

To take advantage of ISMMAP, the application needs to utilize large anonymously mapped segments >2−4 MB in size. An excerpt for pmap(1) again shows mappings on the order of 90 MB, 42 MB, 9 MB, 5MB, etc:

```
         Address   Kbytes Resident Shared Private
Permissions       Mapped File
...
FFFFFFFF61C00000   89856   89848       -   89848
read/write          [ anon ]
FFFFFFFF6D800000    9216    9176       -    9176
read/write          [ anon ]
FFFFFFFF71000000   41472   41376       -   41376
read/write          [ anon ]
FFFFFFFF74800000   89856   89856       -   89856
read/write          [ anon ]
FFFFFFFF7A000000    5120    5120       -    5120
read/write          [ anon ]
.....
```

ISMMAP Results

	Orig Time	Trapstat −t	New Time w/ ISMMAP
LOESS	~4.5 hrs	~30%	~1.5 hrs
IML	~70 min	~20%	~38 min

For both cases, TLB misses fell to less than 2−3%. The downside to this is that performance can actually get worse if there is not enough free, coalesced memory. In this example, LOESS was a better candidate for ISMMAP since its memory requirement was in the 140 MB range compared to ~600 MB for IML; thus it was less subject to the coalescing issue.

ISMMAP is a technology for specifically addressing large numbers of TLB misses. Before looking at this as a potential performance workaround, verify the following:

- CPU bound application (***prstat(1)***)
- Utilizing a fair amount of memory (***prstat(1)***)
- Memory allocations 2−4 MB and up (***(pmap (1))***
- ***trapstat −t*** showing >5−10%

This experimental technology is available from your Sun advocate. Additionally, ISMMAP results are not as dramatic on UltraSPARC II based systems.

Managing Scalability Expectations

Using an 8 way system, **PROC LOESS** realized solid scalability as results went from ~20 min (*CPUCOUNT=1)* to ~4 min (*CPUCOUNT=8*).

#CPUS	**PROC LOESS** Job Times
1	21:31:00
2	12:16:00
4	06:34:00
6	05:06:00
8	04:04:00

Thes results are excellent but what happens when you start loading multiple *CPUCOUNT=8* applications? Chances are slim that you are the exclusive user of an 8 way configuration.

Here we demonstrate the robustness of the Solaris threading implementation by loading the system to determine if the performance degradation occurs predictably.

The first test series consisted of simultaneously running a varying number of *CPUCOUNT=8* jobs. Recall that just **1** job can/will efficiently utilize all the CPUs in an 8 CPU system:

Simultaneous CPUCOUNT=8 Applications on an 8 way system

	Avg Job Time		Addt'l CPUs Needed
concurent programs			
1	04:04:00		0
2	06:47:00		8
3	09:38:00		16
4	12:26:00		24
8	23:40:00		56
	8 programs, running 8 way, time=23:40 roughly equal ==>	1 program, running 1 way, time=21:31 < ==roughly equal	

This speaks very well of the Solaris threading model and from the SAS user perspective, the cost of threading overhead is low. If overloading the system by 7X its CPU capacity
wasn't enough of a load stress test, what would happen if we wildly overloaded the system? Can we predict how application performance will be affected ? Would performance continue to degrade linearly?
From the pattern above, we see that average time increases ~3 minutes for each additional application added. In that test, we stopped at 8 simultaneous running SAS applications.

In a sheer moment of sadistic inquisitiveness, 63 of these *CPUCOUNT=8* jobs were invoked simultaneously. This represents a workload greater than **60 times** the CPU configuration. At perfect linear degradation, we can, based on the above data, extrapolate the average program time to be :

Time @ 2 programs + (3 min X (Total #Apps − Baseline #Apps))

= 6:47 + (3 X (63 − 2))
= 6:47 + 183
= ~189 min or 3 hours, 9 minutes

All 63 programs, wrote their final real time value via the *−fullstimer* option to the SAS log file at 3 hours, 9 minutes and thus finished to the minute (out of 180 minutes!) ***EXACTLY*** what we predicted. This clearly demonstrates the robustness of the Solaris threading model.

What does this imply for sizing considerations? It makes sizing a little simple rbecause the response time degradation

for CPU intensive applications is very predictable. You basically pay more to wait less. Limiting CPUCOUNT to a known quantity (and assuming SAS users abide to that) provides an upper bound to number of LWPs. The trick is to then determine the average number of simultaneous users.

Looking at the table above, there is still great benefit from 2 users both setting *CPUCOUNT=8* as time is only slightly increased. In the pre−V9 implementation, your performance would be no better than 21 minutes since the application advantage of unused capacity. And when there is no unused capacity, performance does not degrade worse in the threaded path.

Performance expert, Adrian Cockcroft, through his ***virtual_adrian*** rules specify CPU overload thresholds upon which ***se zoom*** depends.
His cutoffs are "GREEN" for RunQ values up to 2, "AMBER" for RunQ values up to 5 and "RED" for anything over.

On a 4 way system, we ran 8 *CPUCOUNT=4* SAS jobs. Prior runs have told us that at 4 LWPs, this program can maintain a very high CPU utilization on a per LWP basis. So, we have (# SAS apps x *CPUCOUNT*) or (8 X 4)=32 LWPs being queued up. It's then no surprise that any any given time , we see run queue in the range of 28 (32 LWPs − 4 CPUs):

```
ctcsun8. se mpvmstat 4
cpu r  b  w  pi  po  sr smtx  in    sy   cs  us  sy
id   swap   free
 0 28  0  0   0   0   0    0 221   22   33 100   0
0 9686024 6538336
 1 28  0  0   0   0   0    2  26    0   28 100   0
0
 2 28  0  0   0   0   0    0  24    0   24 100   0
0
 3 28  0  0   0   0   0    0  22    1   24  98   2
0
```

For this job mix, if we wanted to meet Adrian's recommendation, we would need another 24 CPUs to meet the service level expectation. Again, the cost of that additional CPU power to provide an *optimal* might well be prohibitive.

Summary

With the advent of 64−bit addressing capabilities and multi−threaded functionality within the SAS system, great performance benefits can be realized on multi−CPU systems with some tradeoffs. In large multi−threaded applications, demand on the system can grow exponentially as users of such programs are added.

To a much greater extent, it becomes more critical for SAS users to understand their applications' basic resource consumption and provide feedback to their systems administrators. It's also much more critical for systems administrators to understand basic SAS application resource characteristics such as *−fullstimer* & *CPUCOUNT* SAS options, how to invoke SAS programs, where data comes from and goes to, etc. They need to help SAS users understand the resource requirements they consume. We often are asked "How should we size a hardware configuration to run the SAS system? Or "What is the best system configuration for the SAS System?" The answer depends on the coding of the SAS user application, the exact PROC and data steps being called, the sequence they are called in, and the size of the data and problem. One seemingly innocuous change (line of code addition, deletion, change), PROC or global option change, can dramatically change the dynamics of an application in terms of results returned or time to completion.

This paper hopefully demonstrated :

- SAS users can/should understand:
  - # active LWPs(threads) and CPU utilization of them in their application
  - amount of memory consumed
  - *CPUCOUNT* tradeoff−The benefit of much faster results turnaround vs. the effects of potentially consuming resources in a multiplicative fashion. As you increase *CPUCOUNT*, if the decrease in real time is less than the increase in user CPU time, you should consider backing *CPUCOUNT* down so that the greater good of all the systems users could benefit.

- Systems Managers can/should understand:
  - How to run SAS programs, how to get timing information, and performance monitoring basics.
  - Large multi−CPU servers, with large numbers of SAS Users should limit *CPUCOUN*T in V9. Individual SAS users can override this should they have the need to. Resource Management software (ie: Solaris Resource Manager and Solaris Bandwidth Manager) can be put in place if there are enterprise wide IT policies which need to be enforced. For instance, the marketing analysis department cannot consume more than 6 CPUs of the the 48 CPU server. Solaris Resource Manager can be implemented such that this group will only utilize 6 CPUs regardless of the SAS *CPUCOUNT* setting, but if the system happens to be quiet, they will be able to take advantage of the additional idle capacity.

All tests were run on a Sun Enterprise Midframe Server, Sun Fire 3800
with either 8x750 MHz/8GB RAM or 4x750 MHz/4GB RAM

Please see
http://www.sas.com/partners/directory/sun/sugi27−paper.pdf for a full version (including references) of this paper
maureen.chew@sun.com