Scaling Study of the Sun Performance Library Sparse Direct Solver

Mike Johnson (Sun ESP Performance Group), Rajat P. Garg (Sun Performance Library Group)

We investigated the performance of the Sun Performance Library Sparse Direct Solver, especially its scalability across multiple CPUs on the UltraSPARC-III based high-end server systems, namely Sun Fire 6800 and Sun Fire 15000 respectively. We also experimented with various memory configurations on the Sun Fire 15000 using the Solaris madvise library and examined the effects of memory placement policies on solver performance.

***Index Terms* — CPU/Thread Binding, Memory Access Policies, Memory Placement**

I. Introduction

THIS investigation involved a scalability study of sparse direct matrix solvers in the Sun Performance Library software.

Our main findings are summarized here :

- The single-CPU performance on Sun Fire 6800 is about 10% faster than the Sun Fire 15000.
- The parallel speedups on Sun Fire 6800 are better than Sun Fire 15000 and improve with increasing CPU count (in the range of 1 to 24 CPUs).
- Variability in the runtimes (from run-to-run) is substantially higher on the Sun Fire 15000 compared to Sun Fire 6800.
- On an average the 1.2GHz Sun Fire 15000 is 17% faster in runtime of the benchmark compared to the 900MHz Sun Fire 15000 system (in the range of 1 to 24 CPUs).
- The **access_many** memory placement policy shows better scaling than **access_default** and **access_lwp** policies respectively, but has inferior single-CPU performance.
- Process binding was found to result in 1-2% better single-CPU performance on the Sun Fire 15000. With no process binding, measurements using the collector/analyzer tool indicated remote E-cache misses occurring even in a single-CPU/single-thread run and accounting for about 7% of the total E-cache misses.

II. Benchmarks

Our benchmark program solves a large sparse system of linear equations with a symmetric positive definite matrix. The solver used is that which is integrated in the Sun Performance Library: SPSOLVE/LDLT (S1S8 version). All datasets are real, double precision and based either on public-domain tests or extracted from customer benchmarks.

A. Benchmark Workloads

The tests perform all the steps in the solution process, namely, matrix reordering, symbolic factorization, numerical factorization and triangular solves. However, the focus is on the numerical factorization step, since typically that is the most expensive step in the overall solution process. It is also the step in the solution process that has been parallelized (using OpenMP API). The internal timers in the Sun Performance Library measure and report the times in all the steps of the solution process. The internal timers are based on **gethrtime()** function call and the Solaris microstate accounting facility (that enables high precision timing) is enabled by running the benchmarks with the **ptime** command. The scalability results are based on the measured times in the numerical factorization step. An overview of the benchmark datasets is given below.

TABLE I
BENCHMARK DATA SETS

Dataset	Application/Background	Customer
18081	metal stamping (FEM)	Ford
46053	-do-	-do-
90153	-do-	-do-
tk30	structural analysis (FEM)	Harwell-Boeing datasets (public domain)
tk31	-do-	-do-
tk32	-do-	-do-
tk33	-do-	-do-
sun_ms_lpf	electromagnetics (FEM)	HP-EESof (now Agilent)
sun_rod_ant	-do-	-do-
gearbox	mech. analysis (FEM)	public domain
large	structural analysis (FEM)	CESAR (French customer)
pds-20	LP	public domain
dfl001	-do-	-do-
a15	LP	Ilog/Cplex dataset

The data sets are divided into three categories for analysis :

- big : large, gismondi, gearbox, weekly
- medium : a15, daily, 90153
- small : 18081, 46053, tk30, tk31, tk32, tk33, sun_ms_lpf, sun_rod_ant, pds-20, dfl001, finan512, daily2, 3xl-01

The big data sets are thought to scale to the highest CPU counts and we spend most of our efforts on the analysis of the benchmarks' performance on those data

Sun Microsystems Internal Technical Report, created under the Sun Mentor program and submitted to Sun Microsystems in 2003.

B. Benchmark Hardware

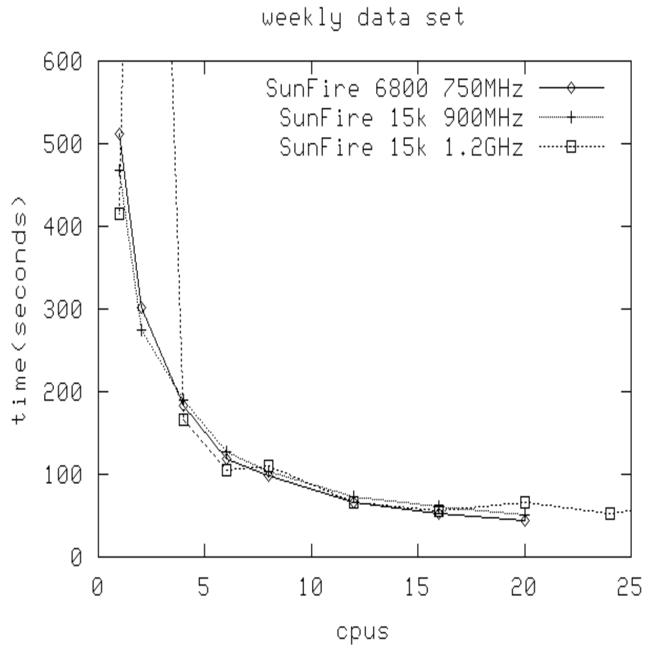
The following three machines were used in running all the tests :

- Sun Fire 6800 with 24 750MHz US-III+ CPUs, 20 GB of memory, Solaris 9u2 (or later update)
- Sun Fire 15000 with 72 900Mhz US-III+ CPUs, 144 GB of memory, Solaris 9u2 (or later update)
- Sun Fire 15000 with 72 1.2GHz US-III+ CPUs, 288 GB of memory, Solaris 9u4

III. Results

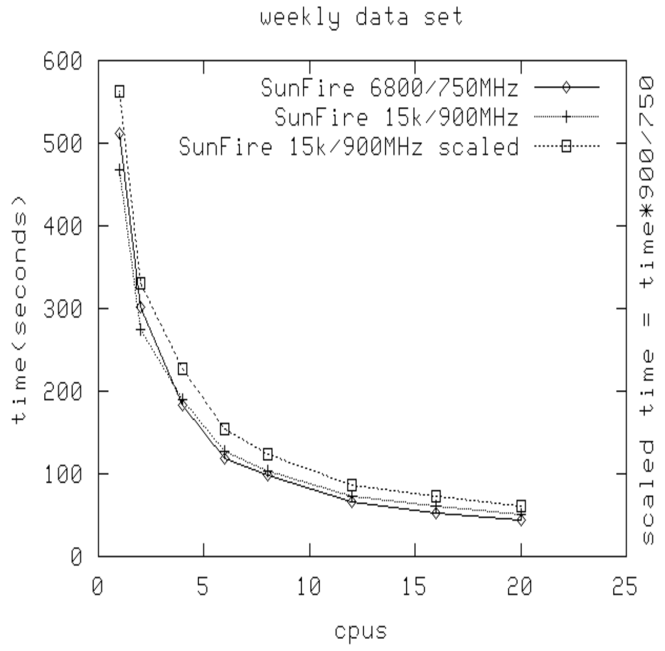
Although, performance results were acquired for all the 20 datasets, in what follows we will concentrate on only the 4 big tests, namely, gismondi, gearbox, large and weekly. Unless otherwise noted, in all the runs no effort was made to control thread placement (such as via use of the MT_PROCESSOR _BIND environment variable, process binding, processor sets etc.) and instead the default Solaris thread scheduling was allowed to take place.

The benchmark reports the wall time taken for each step in the matrix solution. Our performance measurements are taken from the numerical factorization step in this output. The performance for all three architectures is shown in the following figure.



In general we observed that run-to-run variation for the parallel cases (thread-count > 1) is much higher on the Sun Fire 15000 as compared to Sun Fire 6800 due to the stronger NUMA nature of Sun Fire 15000 and the fact that memory distribution and thread scheduling effects are more pronounced (compared to Sun Fire 6800). This might be the reason for the anomalous 2-CPU timings on the 1.2GHz Sun Fire 15000 in the above plot. Due to limited access to the system we could not perform repeat measurements under different load conditions and different Solaris kernel environment. So this was not investigated further. It was however observed to occur for different memory placement policies on that particular machine (so repeatable anomalous behavior was observed).

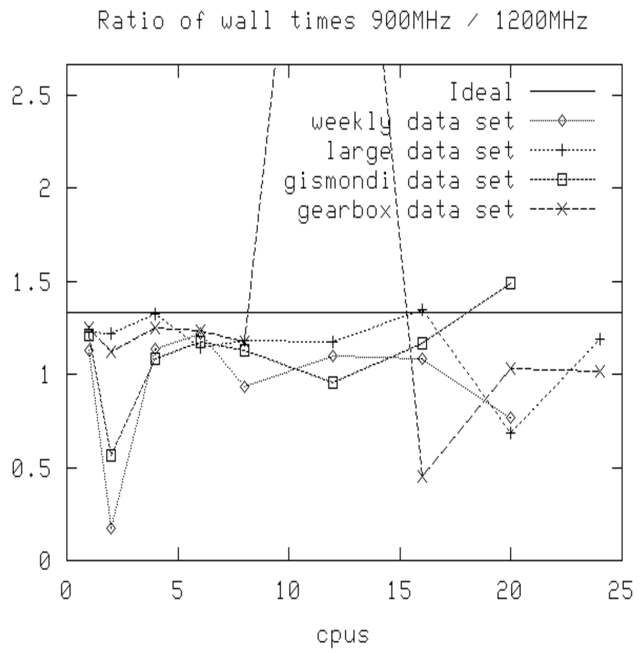
Direct comparisons between the Sun Fire 6800 and Sun Fire 15000 architectures are not possible here because of the differing CPU speeds of our machines, but the figure below shows what we might expect the results to look like on systems with identical CPU speeds by scaling the results of one of the machines by the ratio of the CPU speeds.

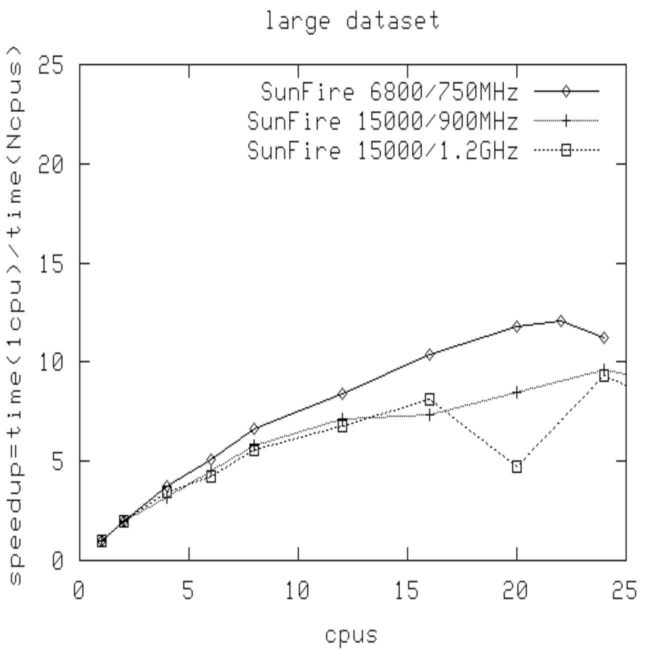


There are several noteworthy observations that can be made from the above plot.

Comparing the single-CPU results for Sun Fire 6800 and Sun Fire 15000 (scaled), we see that the Sun Fire 15000 is about 10% slower than Sun Fire 6800. This is expected since the memory latency (local) on Sun Fire 15000 is slower than Sun Fire 6800 (~350ns vs. ~230ns for the 150MHz interconnect). From the figure one can also discern that the scaling is better on the Sun Fire 6800 and in fact for 4 CPUs (and higher) the 750mHz Sun Fire 6800 has a better performance than the 900MHz Sun Fire 15000. A consequence of this is that the performance delta at single-CPU (about 10%) shows an increase with increasing CPU-count. An important conclusion then is that for this benchmark, the higher memory latency of Sun Fire 15000 as well as its stronger NUMA nature result in both poorer single CPU performance as well as poorer scaling.

The figure below shows the ratio of the wall time for the 900MHz Sun Fire 15000 to the 1.2GHz machine. Ideally, this would be 4/3 based on clock speed scaling.





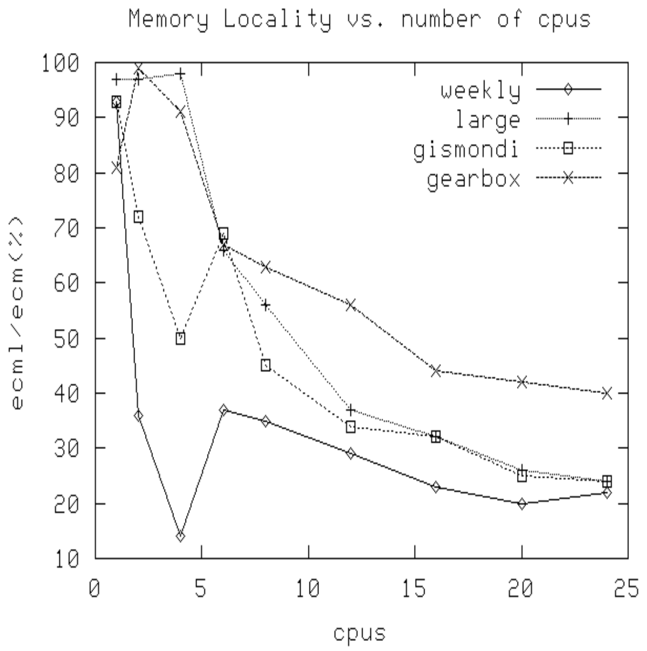
This is an interesting way to look at how the clock speed scaling varies as a function of CPU count. Any valid data point above 1.33. implies a faster than clock speed scaling while data points below 1.0 imply a slowdown on the 1.2GHz machine compared to the 900MHz machine. We see a cluster of points in the range of 1 to 1.33. When the average is computed (after excluding the outliers and only considering the data points falling between 1 and 1.33, we obtain a value of ~1.17. Thus the results show that on the 1.2GHz system we are getting only half as much of the theoretical clock speed boost (compared to the 900MHz system). This can primarily be attributed to the cost of memory transactions (which comprise of both local as well as remote accesses) since the memory system is same (in terms of latency and interconnection speed) on the two systems. The large outliers are indicative of the run-to-run variation (due to the fact that with the Solaris MPO kernel, the memory distribution and thread placement can vary from one run to another).

A. Scaling

The scaling of the benchmark with respect to the number of CPUs is shown for all three architectures in the following figure.

It is clear from the above plot (as well as earlier ones) that scaling is better on Sun Fire 6800 and keeps getting better with increasing CPU count (in the range tested here). Also note that the two starcats are roughly comparable, even though we might expect the 1.2GHz machine to scale poorer because the memory system is same between the two machines.

To help us understand the role of memory locality in the performance of this solver, we used collector/ analyzer tool to profile the benchmark and obtain hardware-counter statistics for remote and local cache misses. The CPU counters ecm (EC_misses) and ecml (EC_local_misses) measure the total external cache misses and the external cache misses that are filled from local memory, respectively. By dividing local misses by total misses, we find an approximation for what fraction of memory references are made to co-local memory. This approximation is not perfect because of the granularity of the counters and side effects of the counter monitoring. We ascertained the accuracy of the measurements by repeating select tests with higher precision setting in the collector (hi setting in collect tool and a smaller overflow interval) and ensuring that the numbers are repeatable in a statistical sense.

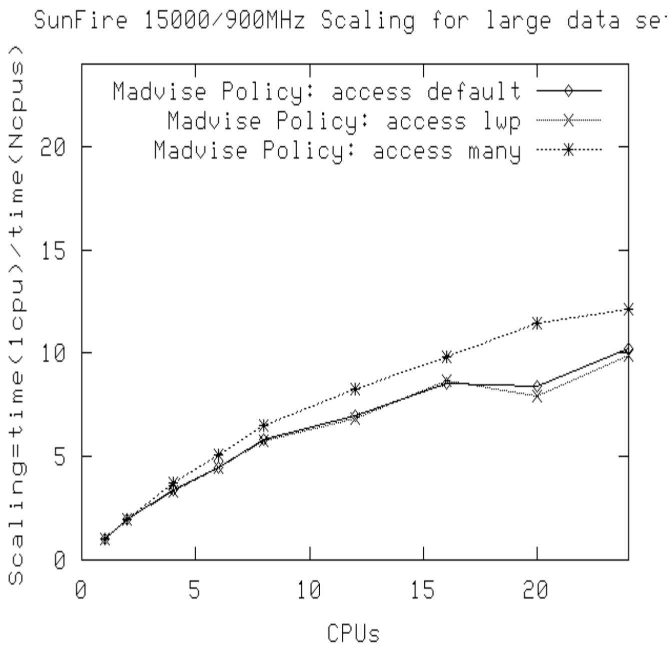


The distribution of the ratio of remote E-cache misses to the total E-cache misses as a function of number of CPUs is shown in above plot for the four different datasets (on the 900MHz Sun Fire 15000 machine). The utility of this plot is to make assessment on the overall characteristics and not to pinpoint the data variation at individual collection points. It is clear from the plot that (as expected) with increasing processor count, the fraction of total misses that are remote increases. In fact, for three out of the four datasets shown above, nearly 70-80% of misses are due to remote memory accesses at 24-CPUs. The other interesting aspect of above plots is the variation that can occur due to the NUMA nature of the Sun Fire 15000 architecture. The machine had 18 lgroups (72 CPUs across 18 boards). A 2-CPU run then can be scheduled such that the threads reside on the same board or on two different boards. Similarly a 4-CPU run could be scheduled by the kernel such that the threads reside on the same board, two, three or even four different boards. Threads colocated on the same board lead to less remote memory accesses and hence better memory access latency. On the other hand, threads located on different boards can utilize the memory bandwidth of the interconnect better than threads located on the same board, though they suffer from increased remote memory transactions. With default scheduling, the placement of threads can vary from run-to-run or could be different as the number of threads are varied for the benchmark tests. Thus this leads to the kind of variability we see in the results presented above as well as elsewhere in this report. Use of thread binding or processor sets to control thread placement is subject of future work and was not considered in the present study.

B. *Memory Placement*

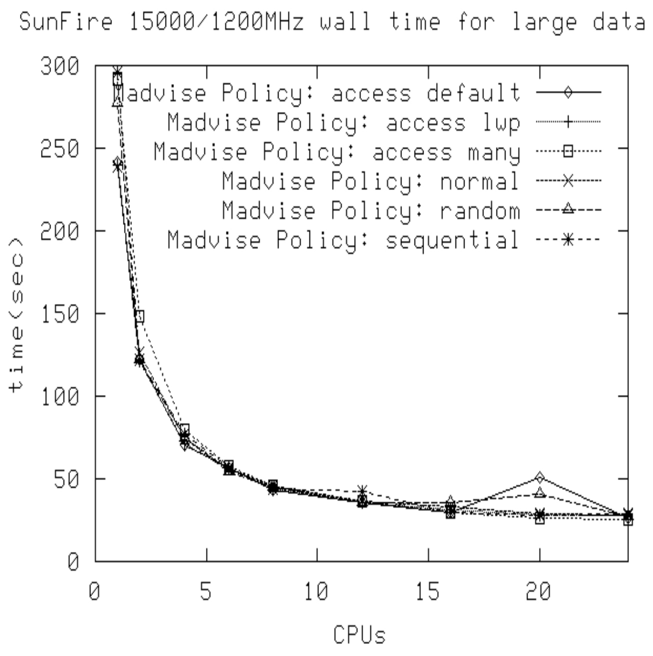
On NUMA systems, it is prudent to examine the effects of memory distribution on the benchmark performance. Using the madvise library (see madv.so.1 (1) and madvise (3c)) on Solaris 9, we can examine the affects of different memory allocation policies on performance.

The affects of the three policies that relate most closely to heap allocation, access_default, access_lwp, and access_many, are shown in the next figure. The access_lwp policy prompts the OS to allocate memory collocal to the CPU running the thread which first tries to access that memory, which is presumed better if that thread is going to make most of the accesses to that memory region throughout the life of the application. This is also the default policy. The access_many policy prompts the OS to spread memory out across lgroups, which is presumed better if all threads will be accessing memory randomly.



The sparse solver has an irregular memory access pattern such that each thread may access and touch potentially all the pages of the primary data-structure, namely the cholesky factor of the input matrix. Thus first-touch may not be the best memory placement policy for this solver and access_many (which distributes the memory pages amongst many different lgroups) might perform better. The scaling results presented above seem to bear this out, where we see that access_many has a substantially better scaling compared to access_lwp policy. It is important however to examine these results alongwith actual wall-clock time measurements, which are presented in the next figure. In that figure, timings for three other policies: random, normal and seqeuntial are also include. It should be noted that these are more applicable to placement of memory pages containing data being paged in and out from filesystem (the Solaris MPO documentation is not quite clear on how memory from normal, random, and sequential are distributed across lgroups).

Looking at the wall-clock times, we can see that although access_many policy shows better scaling, at 1-CPU, it's performance is quite inferior to access_lwp. This is not too surprising because access_many distributes the memory pages on many boards thereby causing increased remote memory traffic even in the single-CPU run. For this benchmark, though, overall, in the range studied, the access_default performs as well as (or better than) both access_lwp and access_many policies. Thus while the default policy turns out to be quite optimal, at higher CPU counts, these results indicate, that access_many policy might perform better than the other policies. Note, we will not remark on the results with normal, random and sequential since not much is documented about these in Solaris documentation and NUMA group papers.



C. Effects of Process-Binding to Single CPU

To study the effects of thread migration between lgroups (i.e. memory/CPU uniboard) on the performance of a single-CPU/single-thread run, we performed runs with and without process binding enabled.

The table below presents single-CPU results for two data-sets using process binding.

Data Set	Total External Cache Misses	External Cache Misses Resolved Locally	Local/Total
gismondi / run 1	199636816 137.128 sec.	199395237	99.9%
weekly / run 1	1014440889 424.339 sec.	1013330579	99.9%
gismondi / run 2	202525134 138.041 sec.	202258432	99.7%
weekly / run 2	1017651109 422.210 sec.	1016702950	99.9%

The table below contains single-CPU results for two data-sets using process binding.

Data Set	Total External Cache Misses	External Cache Misses Resolved Locally	Local/Total
gismondi / run 1	203465024 138.678 sec.	189309677	93.0%
weekly / run 1	1005357644 426.077 sec.	943252853	93.8%
gismondi / run 2	199119159 138.428 sec.	185582590	93.2%
weekly / run 2	1010507699 426.157 sec.	948117535	93.8%

It is interesting to note that without process binding, even the single-CPU (i.e. single-thread) run exhibits some remote misses (about ~7 %). This behavior was observed on both the Sun Fire 15000 machines and observed in multiple runs. Using the tool vtop, we observed that in this case, some memory pages are placed such that they fall in a different lgroup than the one where the process is running. On the other hand, if pbind command is used to bind the process to a given processor, no remote misses are observed and vtop shows all memory assigned to that single board. The run-times for the case without process binding are consistently higher (albeit by a tiny, 1-2%, amount) compared to the case with process binding enabled. This is a rather peculiar behavior which needs to be investigated further but due to lack of machine access, currently we could not pursue it much further. Our speculation is that this could be due to either Solaris MPO kernel's memory placement policy or due to thread-affinity and scheduling policies which cause the thread to be scheduled on a processor lying in a different lgroup thereby causing some remote E-cache transactions to occur.

IV. Conclusions and Future Work

In this work we examined the performance and scalabilty of Sun Performance Library sparse direct matrix solver on the Sun Fire 6800 and Sun Fire 15000 platforms, respectively. The summary of our main findings is as follows:

- The single-CPU performance on Sun Fire 6800 is about 10% faster than the Sun Fire 15000.
- The parallel speedups on Sun Fire 6800 are better than Sun Fire 15000 and improve with increasing CPU count (in the range of 1 to 24 CPUs).
- Variability in the runtimes (from run-to-run) is higher on the Sun Fire 15000 compared to Sun Fire 6800.
- An average improvement of about 17% is seen (over the range of 1 to 24 CPUs) between the 1.2GHz and 900MHz Sun Fire 15000 systems. This is about half of the clock speed gain (which is 33%).

- The effects of different memory placement policies in Solaris 9 MPO kernel were examined on the Sun Fire 15000 systems. The results show that access_many policy results in a better speedup compared to access_lwp policy but has inferior single-CPU performance. Overall, in terms of wall-clock time, the default policy performs as well as access_lwp and access_many for these tests.
- Process binding was found to result in 1-2% better single-CPU performance on the Sun Fire 15000. With no process binding, measurements using the collector/analyzer tool indicated remote E-cache misses (about 7% of total E-cache misses) occurring in the runs. We conjecture that this is due to Solaris process scheduler moving the process from its home lgroup during the course of the run causing remote accesses to occur and slightly degrading performance. The lack of stronger processor affinity in Solaris MPO kernel is surprising.

Based on above results, there are several topics that need further investigation. First and foremost is further investigation into the variability in results on the Sun Fire 15000. One way to explore this is by binding the threads via use of MT_PROCESSOR_BIND environment variable. This will also enable to experiment with different thread placement strategies across the lgroups in the system. We also need to investigate the effects of memory placement strategies on the Sun Fire 6800.

Our expectation is that on Sun Fire 6800 the impact of memory placement strategies will be smaller but nevertheless it will be interesting to study it and correlate with the Sun Fire 15000 results. In order to better understand the effect of memory policies on the Sun Fire 15000, we need to obtain remote and local E-cache miss statistics and study them as a function of CPU count. We were not able to collect this data (for all the policies) in the present study due to machine time limitations. Finally, further investigation is required to understand the reason why with no process binding remote E-cache misses are observed even for serial runs.

References

Sun Microsystems Internal Links:

[1] Sun Performance Library Reference Manual (linked through http://docs.sun.com/

[2] Solaris NUMA group webpage (http://lgroup.eng/)

[3] vtop tool (http://posweb.east/pub/wildcat/tools/)