SV1ex Memory Upgrade Gives Greatest Boost to User Performance

Thomas J. Baring
Arctic Region Supercomputing Center
University of Alaska Fairbanks

ABSTRACT: CPU and memory upgrades to the Cray SV1 at ARSC were spaced about eight months apart. In all stages of this upgrade, performance data were collected on all user jobs using hpmflop, and controlled tests were made on four particular user codes. Analysis shows that the memory, rather than the CPU, upgrade was of greatest overall benefit to users. Codes with good inherent cache efficiency and reliance on scalar processing got speedup under the CPU upgrade and those with a high degree of vectorization obtained the most speedup under the memory upgrade. This follows from the observation that machine balance was improved for scalar-only codes by the CPU upgrade and for vector codes by the memory upgrade.

Introduction

ARSC's two step upgrade of its Cray SV1 to SV1ex provided an unusual opportunity to assess the performance impact of CPU and memory enhancements individually. This study uses two different types of data collected on the SV1 iterations, the SV1, SV1e, and SV1ex.

First, passive monitoring of all actual user jobs was done, and from this data, an attempt made to track both system-wide and individual code performance. There is admittedly insufficient data in these records to guarantee that each individual code tracked was compiled and run in a consistent manner across the two upgrades, but after some cleaning of the data, it is clear that the memory upgrade was of greater benefit to more individual codes and the overall user base than the CPU upgrade.

Second, to help understand the reasons behind the observations described above, four user codes with fixed input were obtained, and run as benchmarks across the three SV1 iterations. These data allow for controlled analysis which shows that a code's inherent degree of vectorization remains the key to its performance on the SV1 series, and that the upgrade which helped well-vectorized codes was the memory upgrade. Codes which use the SV1 cache effectively gained the most in the CPU upgrade, but unless they were also well vectorized, they remained underachievers.

The broader conclusions of this study are that memory bandwidth remains the key to vector performance and that, although some codes were unaided by the first, CPU upgrade, the combination of CPU and memory upgrades improved the performance of all user codes.

Background

ARSC

The Arctic Region Supercomputing Center (ARSC), located on the University of Alaska Fairbanks campus, operates a Cray SV1ex, Cray T3E, and IBM SP. ARSC's vector offerings prior to the SV1 series were a Cray J90 and Cray Y-MP M98.

SV1ex Upgrade in Stages

The basic features of the ARSC SV1 system remained unchanged in these upgrades: it is a 32-CPU, 4GW, shared-memory parallel-vector processor. Each SV1 processor has both scalar and vector functional units, with maximum scalar speed 1/20th of vector, two vector pipes, and a 256KB vector/scalar/instruction cache. The vector length is 64 8-byte words [1].

ARSC was the first SV1 user site, in both cases, to upgrade CPU and memory. The following table, in which "Peak speed" denotes the theoretical peak single-CPU performance, and "Mem. Bandwidth" denotes the actual peak CPU to memory bandwidth as measured on a single CPU by the STREAM benchmark [5], summarizes the upgrades:

System	Install or Upgrade Date	Clock speed MHz	Peak speed mflop/s	Mem. Bandwidth GB/s
SV1	Sep 30, 2000	300	1200	2.5
SV1e	Apr 11, 2001	500	2000	2.5
SV1ex	Dec 12, 2001	500	2000	3.4

Table 1. SV1-SV1ex Upgrade

System-Wide User Speedup From Upgrades

Enabling users to accomplish more work in less time is the purpose of an upgrade. To establish whether this goal was achieved, ARSC configured the SV1 to do passive performance monitoring of all user jobs, the results of which are readily available via the UNICOS "hpmflop" command. A two-part analysis of the hpmflop data across both SV1 upgrades shows that ARSC users benefited most from the faster memory. The first analysis includes runs of all user codes, even those which, for various reasons, were only run on one of the three SV1 iterations. The second analysis makes cross-platform comparisons of those individual codes which were run on two or all three of the SV1, SV1e, and SV1ex.

Raw hpmflop data

The raw data available through hpmflop includes an entry for every job run on the system which a) terminates normally, and b) accumulates more than five CPU-Seconds. The following four items are reported:

- user name,
- average MFLOPS for the entire run,
- file and path of executable (as specified by the user), and
- total CPU-Time.

The raw data set includes all ARSC SV1 and SV1e jobs, and all SV1ex jobs through April 16, 2002.

Data Preparation

This study is concerned with the performance of significant user work. Thus, all jobs run by ARSC staff members and those which accumulated less than 180 CPU-Seconds were immediately removed from the data set, as were runs of utilities, such as "ssh" and "ftp."

The data is organized by treating the combination of "user name" and "file and path of executable" as a single user "code." For instance, "`smith:./ver2.3/vel.x`", would be one code. Every occurrence of the "code" in the hpmflop output is considered as one run, and runs on the three SV1 versions are pooled separately for plotting and statistical analysis. In total, 270 such "codes" were found.

Actual statistics extracted for one code are shown in table 2 as an example. The name, "`smith:./ver2.3/vel.x`" is fictitious, but we can pretend that these data were from runs by user "smith" of the executable program, "`./ver2.3/vel.x`".

	SV1	SV1e	SV1ex
N Runs	102	608	154
cpu-sec tot.	86790.2	68657	58806.7
mflops wavg	217.23	210.044	293.121
mflops avg	216.588	208.921	283.403
mflops sdev	28.0825	29.9498	42.9259

Table 2. Statistics for one actual, sample code

The quantity "mflops wavg" is the average mflops, weighted by run time, and is the "average" used throughout the study. Perl scripts were used to preprocess the 400,000 plus lines of hpmflop output, and the IgorPro data analysis application by Wavemetrics, Inc. was used for further data cleaning, computation of statistics, and plotting.

Overall Performance Improvements

The sample code of table 2 is, unfortunately, only one of eight codes which a user ran on all three SV1s, and thus, across both upgrades. Including those eight, about 20 codes were run across the SV1-SV1e upgrade and another 20 were run across the SV1e-SV1ex upgrade. The remaining 232 codes were run on one platform only. The graph in figure 1 includes all codes, regardless of overlap.

This "cumulative" plot was produced as follows. The total CPU Time accumulated by all codes on each platform was determined, and then the percent contribution to this total made by each job computed. The codes were sorted by MFLOPS and plotted, with each code's percent of total CPU time added to the cumulative sum of all codes with lower MFLOPS. Thus, looking at the red, SV1ex trace, we know that 100% of all user codes on the SV1ex ran at or below 760 MFLOPS. Similarly, 100% of all codes on the SV1e ran at or below 500 MFLOPS, and on the SV1, they ran at or below about 450.

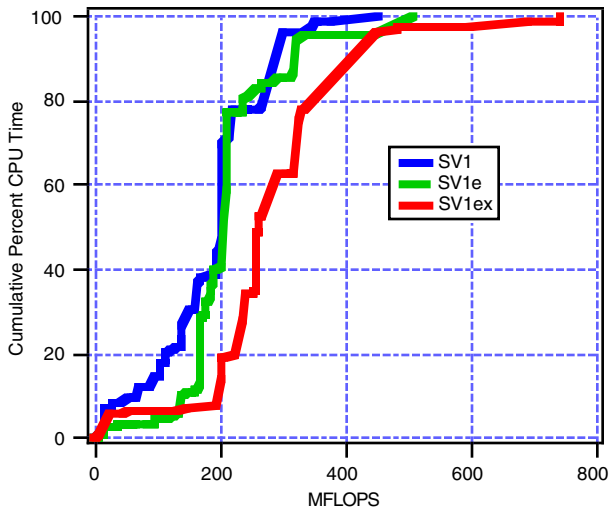

Figure 1: Cumulative Plot of MFLOPS

The importance of the memory upgrade to users is clear from this plot. On both the SV1 and SV1e, nearly 80% of all user codes ran at below 225 MFLOPS while on the SV1ex, only 20% run below that rate.

Note the gap between the SV1 and SV1e curves in the MFLOPS region below about 160 MFLOPS. Codes in this regions were helped most by the CPU upgrade while the better performing codes were helped more by the memory upgrade. This particular observation will recur throughout this study and, as shown later, can be explained quite simply: scalar-dominated codes were helped by the CPU upgrade, well-vectorized codes were helped by the memory upgrade.

The combined effect of the CPU and memory upgrades was a fairly uniform improvement across the entire range of MFLOPS values. Neither upgrade in isolation would have satisfied everyone.

Analysis of Individual Codes

For the most precise analysis of user code speedup across the two upgrades, each user would have, ideally, re-run each of his or her codes with identical input parameters and files and sent us the hpm output. This didn't happen, but with additional data cleaning and acceptance of some inaccuracies in the cross-platform comparisons, we can use the hpmflop data to estimate the gains for many of the individual codes. The method of cleaning the data is discretionary and a more algorithmic approach might be implemented for future studies. Some interesting patterns appear, however, and are worth mentioning.

Figures 2 and 3 show all runs of one particular user code (randomly numbered "162") on the SV1e and SV1ex. This code was not run on the SV1. In these graphs, using the right axis, the circular symbols mark each run as CPU Seconds versus MFLOPS and the single solid square marks the average CPU-Seconds of all runs against the weighted average of all runs. The histogram bars, using the left axis, mark the frequency of runs within each MFLOPS bin. Is it reasonable to conclude that this particular code did speed up from 283 to 313 MFLOPS, as shown in these graphs, as a result of the memory upgrade? Or is this an apparent speed up, caused by another factor?

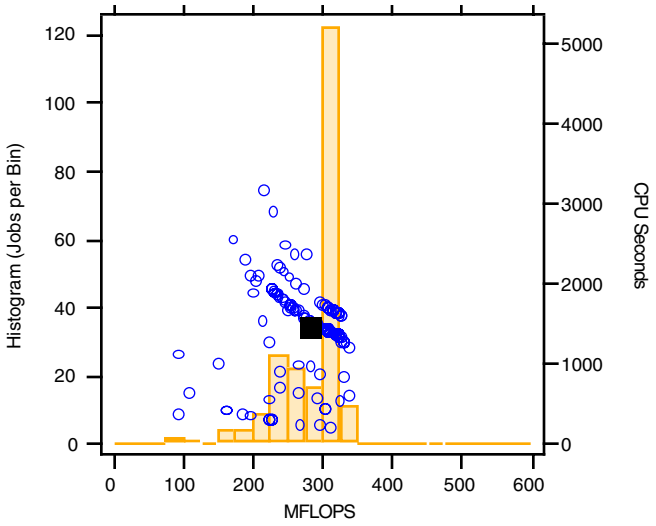

Figure 2: SV1e runs of user code "162"

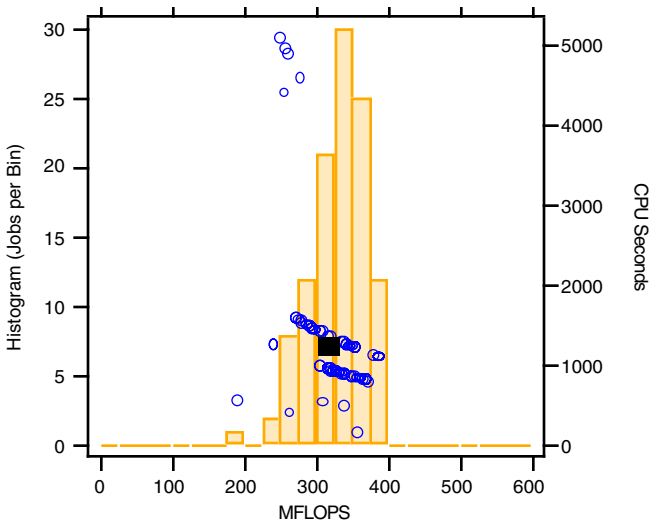

Figure 3: SV1ex runs of user code "162"

Ultimately, nine user codes were eliminated from the comparison study by visual inspection of graphs like these. The codes with the 50 largest standard deviation and/or skewness were plotted and the decision of which to reject made by inspection. In the process, codes "162" (above) and "155" (below), for example, were retained, as the computed average values seemed to adequately represent the overall performance of the codes, while "131" (below) was rejected.

The runs of "131" plotted in figure 4 are clearly bimodal, and the "average" is not representative of either mode. Several hypothetical explanations are available for the two modes. The user may have:

- switched to a different code of the same name,
- switched to a fundamentally different data set,
- started running on some N rather than M multitasked CPUs, or
- done something else.

This is an extreme example, but several codes had histories like this, and were simply thrown out.

The other interesting feature of "131," which also appears in both runs of "162" are the two gently sloping curves defined by subsets of the runs. Multiplying the CPU-Seconds and MFLOPS for a single job yields the total number of floating point operations, and this product is very similar for all jobs in each trend. This suggests that each "curve" is produced by a suite of nearly identical jobs (same executable, same size data sets) which ran at slightly different rates, due, for instance, to system load during the run. The observed inverse relationship between MFLOPS and CPU-Seconds is expected, of course, if the total number of operations is constant. The weighted average MFLOPS values for most codes showing such "suites" of runs seem to adequately represent the actual ranges of performance, and were not eliminated from the study.

An interesting pair of groups is shown by code "155" which made over 1,200 short, slow runs, and a small number of long, faster runs. These were, perhaps, testing and production phases of the user's project. The weighted

average MFLOPS value seems to catch the true performance of this code, so it was retained.

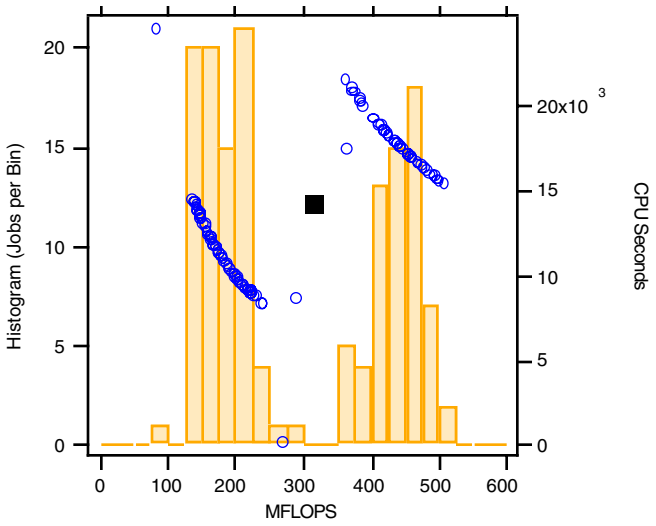

Figure 4: SV1e runs of user code "131"

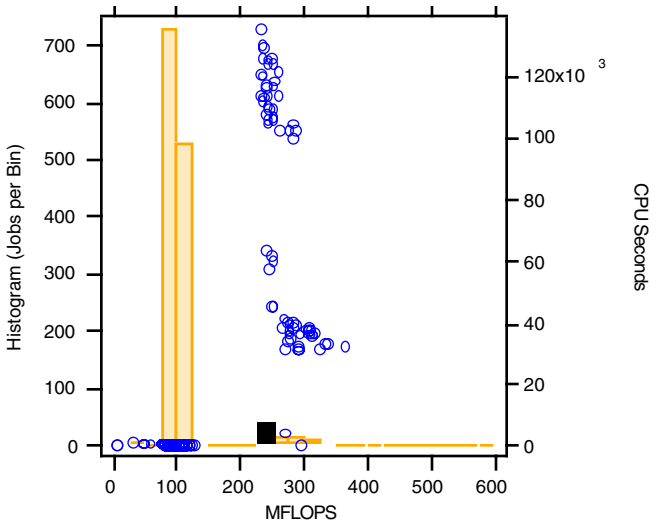

Figure 5: SV1ex runs of user code "155"

Cross-Platform Comparisons by Code

Approximately 260 codes remained in the hpmflop data set after the process of "cleaning," and, as mentioned earlier, 20 of those codes were run on both the SV1 and SV1e. Each of these codes appears as a separate point in figure 6, with improvement in performance plotted against the (weighted) average MFLOPS.

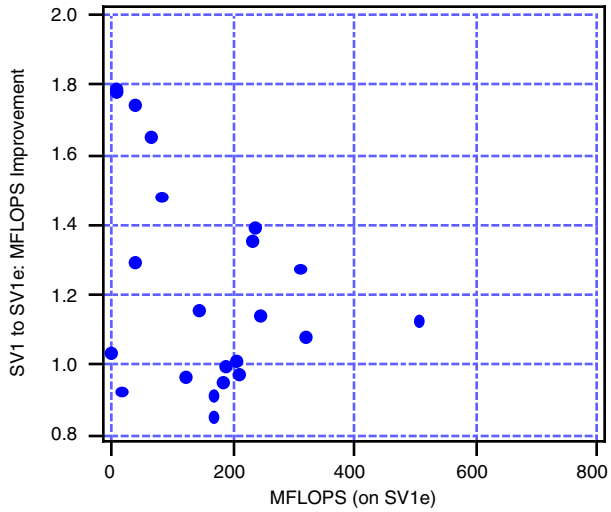

Figure 6: MFLOPS Improvement of Individual User Codes, SV1 to SV1e

This comparison again suggests that the CPU Upgrade helped the slower, non-vector codes. In particular, note the apparent inverse relationship between speedup and MFLOPS: slower, scalar codes did better. The slow-down of some codes ("improvements" below 1.0) can't be explained by the CPU upgrade, and must be due to factors not tracked by hpmflop. It reminds us of the inaccuracies inherent in this portion of the study.

Similarly, the 20 codes which were run on both the SV1e and SV1ex are plotted in figure 7, at the same scale.

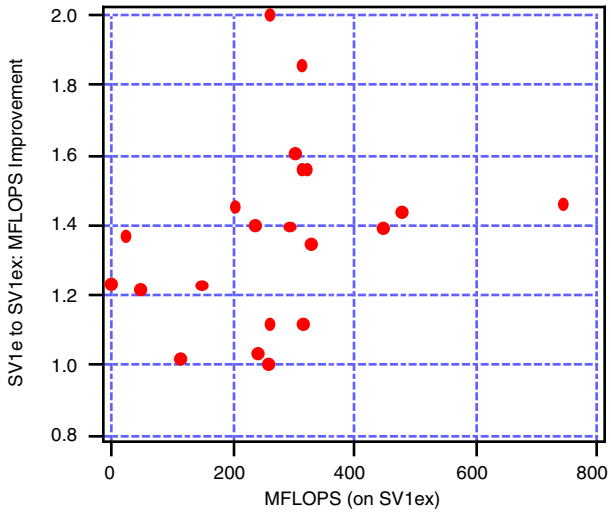

Figure 7: MFLOPS Improvement of Individual User Codes, SV1e to SV1ex

The average speedup for these sets of codes from the CPU upgrade was 1.24 and from the memory upgrade, 1.37. Visually, the center of mass of points in the SV1e to SVex graph (figure 7) is higher and to the right, and thus, once again, higher MFLOPS codes were generally helped more by the memory upgrade. The comparison between figures 6 and 7 is not completely reliable because, except for the eight

codes which overlapped both upgrades, the codes plotted are different.

The concluding graph, below, shows the eight user codes which were run on the SV1, SV1e, *and* SV1ex and thus, for which improvement due to both upgrades could be compared directly . The x-axis gives improvement from the CPU upgrade and the y-axis, the improvement from the memory upgrade. The two axes are at the same scale.

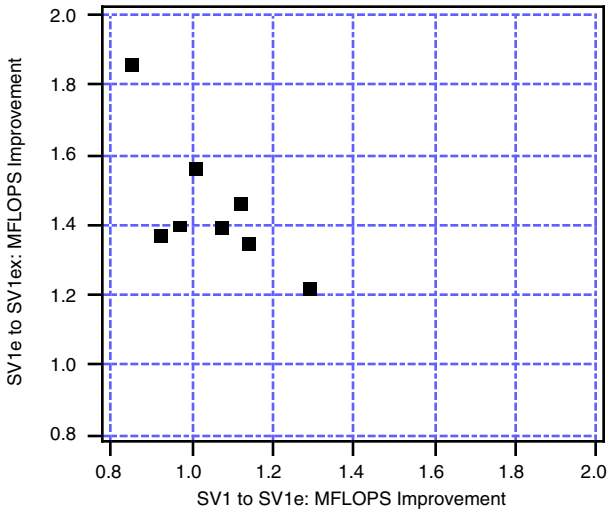

Figure 8: MFLOPS Improvement of Individual User Codes, Both Upgrades

This plot reiterates both major conclusions of this study.

First, the inverse relationship suggested by the downward sloping trend shows that the attributes of an individual code which made it respond well to one upgrade hindered it in the other. These attributes are the dominance of scalar versus vector processing which are, of course, inversely related.

Second, the codes' center of mass is high on the memory upgrade (y-) axis but low on the CPU upgrade (x-) axis. For these eight codes, the average MFLOPS improvement from the CPU upgrade was 1.05 while from the memory upgrade it was 1.45. Again, the memory upgrade gave users the bigger boost.

Speedup of Benchmark User Codes

Performance Metrics for SV1 Codes

To better understand the effects described above, four user codes were evaluated in more detail using three performance metrics.

An executable program is a fixed implementation of an algorithm, under the treatment of a compiler, which, when run on similar architectures against similar data sets will retain some relatively constant performance attributes. In comparing runs of the same program on the SV1, SV1e, and SV1ex, the following three inherent attributes are of particular interests:

- Degree of Vectorization (or "Vectorization")
- Computational Intensity
- Cache Efficiency

These attributes can be treated as metrics, or ways of "measuring" codes, to aid in understanding their performance or even predicting how they might perform on future, similar architectures.

On a Cray vector system, values of these metrics may be derived from data collected by the hardware performance monitor (hpm). HPM is a tool for accessing special hardware counters. It averages results across the execution of the entire program, and running it has no impact on the performance of the code [3]. For this study, the following fields were used from hpm groups 0 and 3:

hpm group 0:

```
Million inst/sec (MIPS)
Cache hits/sec
CPU memory references/sec
Floating ops/CPU second
```

hpm group 3:

```
Percent of all instructions by type
(000-017)jump/special
(020-077)scalar functional unit
(100-137)scalar memory
(140-157,175)vector integer/logical
(160-174)vector floating point
(176-177)vector load and store
```

The *degree of vectorization* metric can be estimated as the hpm fields "Floating ops/CPU second" divided by "Million inst/sec (MIPS)" (or MFLOPS/MIPS) [3]. This report uses the MFLOPS/MIPS measure throughout because only hpm group 0 results were collected for all programs on all platforms. Where available, however, these results were tested against the more precise group 3 vectorization statistics and no significant qualitative differences were found. A high value indicates that the executable is making good use of the SV1's vector functional unit.

Computational intensity can be considered the amount of work done per memory reference, and is computed as "MFLOPS" divided by "M memory references per sec." A high value indicates that the code is performing several computations for each relatively expensive memory reference. Data fetched from memory can be cached in registers or in the SV1 cache for reuse.

Cache efficiency is computed as "cache hits per sec" divided by "CPU. Mem. References/sec", and is the fraction of all memory references which were satisfied from cache.

Selected User Codes

The four codes used as benchmarks were run against consistent data sets, in the same way, on the three SV1 architectures.

GAMESS

GAMESS is a well-known quantum chemistry package, and on the ARSC SV1 is run on one CPU. The run-time characteristics of GAMESS changes radically depending on the input problem and while it is most frequently run on MPP platforms, for the problems run on the SV1, it requires large memory and does not parallelize. The GAMESS data for this report was supplied by an ARSC user, all runs were on 1-CPU.

FLAPW

This is a well-vectorized quantum physics code. Due to a problem which took several months to understand, performance data for the original SV1 is not available for 1-CPU runs, and thus, data from 2-CPU runs is used consistently for this study. This is unfortunate, as this is by far the highest performing code in the suite, achieving sustained performance of 1/3rd to 2/5ths of peak theoretical performance when run on a single CPU. It doesn't scale well, and performance on 2 or more CPUs is not as exhilarating.

Tsunami

This is a finite difference code, written with traditional Fortran 77 style DO loops, and achieves reasonable vector performance. It solves the shallow-water wave equations for nested grids of increasing resolution.

POLAIR

This is a coupled ocean/land/atmosphere/ice model, and is unique in having been written using Fortran 90 constructs, such as WHERE statements and array syntax to the elimination of DO loops.

Measured Performance Under Metrics

Each of the values presented in the following graphs is the average obtained from between 2 and 6 runs, with an effort to run on mainly idle machines. An exception are the runs of GAMESS for which only one run per data point was made. To simplify this analysis, data for multi-processor runs is not considered (with the exception of the FLAPW code, for which data from autotasked, 2-CPU runs, is used).

As noted previously, the metrics are characteristic of the codes themselves, and the values of the metrics are nearly identical on the three different architectures. This is supported by a similar study performed last year [2] in which autotasked runs on 1, 2, 4, and 8 processors across two different compiler versions on the SV1 and SV1e were combined and, again, uniform values for each metric for each code observed. Values of the metrics are given below (Table 3), preceded by plots of the raw hpm values used in their computation.

Degree of Vectorization Metric

Laying aside questions of algorithm choice and effective implementation, the MFLOPS axis in figure 9 shows that Tsunami and FLAPW (even multitasked) rank highest in this suite. The axis maximum of 350 MFLOPS is 17.5% of the SV1e and SV1ex peak and quite respectable for sustained performance.

Vectorization is computed as MFLOPS/MIPS, the ratio of the y-axis to x-axis values in this plot. GAMESS is at the high end of the MIPS axis and low end of the MFLOPS axis, and thus is poorly vectorized and demands far more instructions per result.

In figures 9-11, the symbols, "1", "E", and "X" mark data collected on the SV1, SV1e, and SV1ex, respectively.

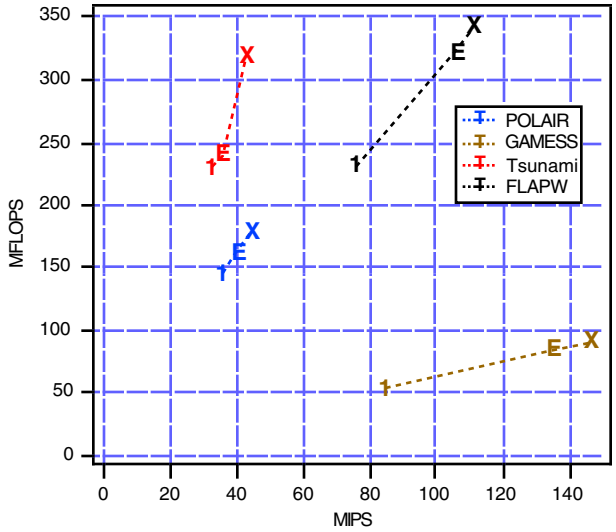

Figure 9: Degree of Vectorization

Computational Intensity Metric

FLAPW shows its memory efficiency in figure 10. Each transfer between CPU and memory performed by FLAPW yields about twice as many "results" as Tsunami and six times as many as POLAIR.

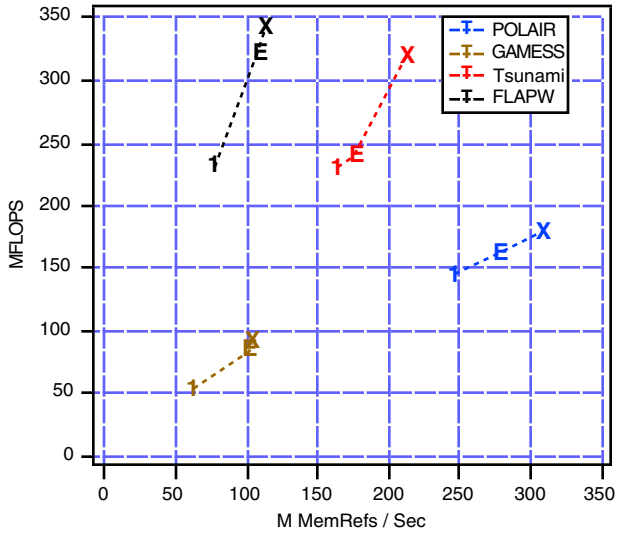

Figure 10: Computational Intensity

Cache Efficiency Metric

The best and worst performers (FLAPW and GAMESS) are strikingly similar, to judge by figure 11. On

the SV1e and SV1ex, they have the highest cache hit rates and lowest memory reference rates. POLAIR is the opposite.

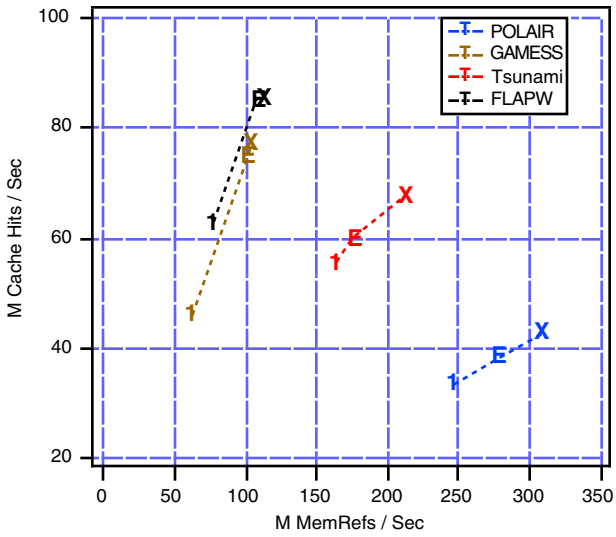

Figure 11: Cache Efficiency

Computed Values of Metrics

For each metric, larger values are better. Averaging across the three platforms for each code reveals that none of the codes ranks highest or lowest in every metric (table 3). To determine their relative importance, we must consider speedup.

	POLAIR	GAMESS	Tsunami	FLAPW
Vect.	3.99	0.61	7.00	3.03
Comp.Ints.	0.58	0.83	1.41	2.96
Cache Eff.	0.15	0.73	0.33	0.78

Table 3. Values of Metrics averaged across SV1, SV1e, and SV1ex

Speedup

Computations of speedup in this study are based on CPU-Time:

```
SV1e Speedup  = SV1 CPU-Time /SV1e CPU-Time
SV1ex Speedup = SV1e CPU-Time/SV1ex CPU-Time
```

In the following two figures, speedup is plotted against one of the metrics, there are two data points for each of the four user codes: one point is the speedup obtained in the CPU upgrade (marked with the symbol, "E") and the second is the speedup from the memory upgrade ("X").

Cache efficiency is well correlated to speedup from the CPU upgrade (figure 12). Ignoring the "X's" in this graph, the SV1e values ("E's") of GAMESS and FLAPW are clearly high and to the right of Tsunami and POLAIR.

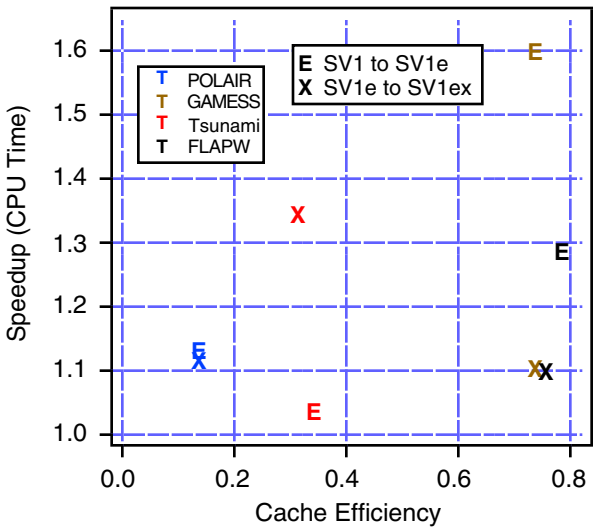

Figure 12: Speedup versus cache efficiency

These pairings based on cache efficiency were noted earlier in figure 11. Realizing that the CPU upgrade increased the cache to CPU bandwidth by 66% (along with the clock speed increase of 66%, table 1), the observed speedups are reasonable.

The SV1ex speedups ("X's") don't show a similar grouping on this plot. Cache efficiency wasn't a factor in gaining speedup from the memory upgrade.

The relationships between speedup and vectorization (figure 13) are quite revealing. First, consider the CPU upgrade (the "E" symbols) only. SV1e speedup increases dramatically from right to left, as the degree of vectorization decreases (and thus, as the scalar percentage of a code increases).

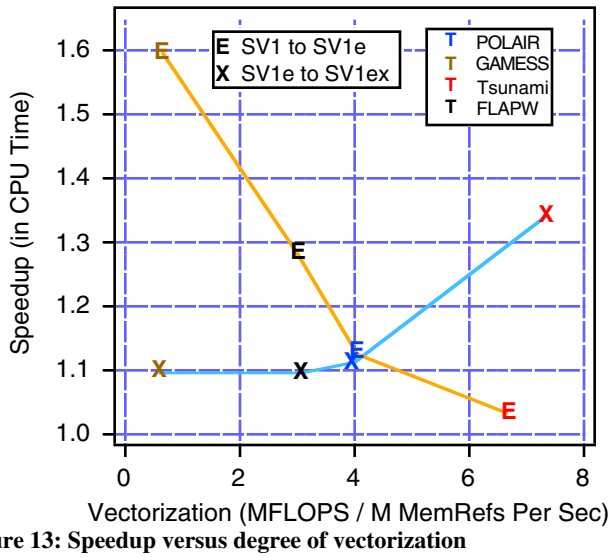

Figure 13: Speedup versus degree of vectorization

To help understand this effect, consider an analog of the computational intensity metric for codes, the "machine balance" metric for computer systems. "Machine Balance"

is the peak theoretical performance of a machine divided by the peak system memory to CPU bandwidth observed [4]. A value of 1.0 is best, and suggests the machine is "balanced"; values greater than 1.0 suggest that memory is too slow for the CPUs; values less than 1.0, that the CPUs are too slow. Since this study is concerned with single CPU performance, and the test codes were generally run on one CPU, balance values (table 4) for a single CPU are relevant. (This single-CPU balance is different from "machine balance," as normally defined [4].)

The column, "Scalar-only Balance," gives artificial values that would exist if the SV1 CPU had only a scalar, and no vector, unit, or, put another way, if a given code had no vector component or were compiled with vectorization off.

A shift in balance closer to unity implies an improvement. Thus, it follows from the values in table 4 that the CPU upgrade was an improvement for scalar codes and the memory upgrade was an improvement for vector codes.

	Balance	(Scalar-only Balance)
SV1	3.8	(0.19)
SV1e	6.4	(0.32)
SV1ex	4.7	(0.24)

Table 4. CPU versus Memory Balance for one CPU

The observation from the benchmarking tests (figure 13) that SV1e speedup increases with scalar percentage is reasonable considering this table. On the original SV1, the bottleneck for "scalar-only" codes was the CPU. The SV1e reduced this bottleneck by moving the scalar-only balance from .19 to .32, and could thus have been expected to improve the performance of scalar codes.

Similarly, consider the effect of the memory upgrade as shown in figure 13 (the "X" symbols). SV1ex speedup increases with better levels of vectorization. This, again, is as expected since (from the "Balance" column of table 4) the bottleneck for a vector code is the memory subsystem. The SV1ex upgrade reduced the memory bottleneck by moving the balance from 6.4 to 4.7. A memory upgrade could thus be expected to improve the performance of vector codes.

The observation made earlier that the CPU upgrade aided the slower codes in the ARSC user base (figure 1) can similarly be explained by noting that the slower codes are likely scalar-dominated. The memory upgrade aided the faster codes in the user base because they are, by definition, vectorized. Given that most of the codes running on the system are vectorized, it's not surprising that the memory upgrade gave the bigger overall boost to the user base.

References

[1]: Maynard Brandt, Jeff Brooks, Margaret Cahir, Tom Hewitt, Enrique Lopez-Pineda, Dick Sandness.. "The Benchmarker's Guide for CRAY SV1 Systems", Cray Inc., July 20, 2000

[2]: Thomas Baring, Jeff McAllister, "SV1e Performance of User Codes," CUG 2001 Proceedings, May 2001.

[3]: "Optimizing Application Code on UNICOS Systems", Cray online Software Publications, 004-2192-003

[4]: John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Technical Committee on Computer Architecture newsletter, December 1995.

[5]: "STREAM: Sustainable Memory Bandwidth in High Performance Computers", Web site URL: "http://www.cs.virginia.edu/stream/", Maintained by John D. McCalpin, Department of Computer Science School of Engineering and Applied Science University of Virginia, Charlottesville, Virginia

Acknowledgements

My thanks to the ARSC users who were willing to share their codes and results with me, and to Jeff McAllister and Guy Robinson of ARSC for ideas and feedback.

Author

Tom Baring is a User Consultant with six years of experience at the Arctic Region Supercomputing Center. He may be reached at: `baring@arsc.edu`.