Title: # Performance of the BLAS-1 and Other Mathematical Kernals on the SGI/CRAY ORIGIN 2000 Processor

Author(s): William Dearholt
Wayne Joubert

Submitted to: For external distribution as requested.

Los Alamos
National Laboratory

Performance of the BLAS-1 and Other Mathematical Kernels on the SGI/Cray Origin 2000 Processor

William Dearholt
Wayne Joubert

Parallel Architectures Team
CIC-19, Scientific Computing Group
Los Alamos National Laboratory
Los Alamos, NM 87545

August 1997

1. Introduction

The purpose of this paper is to explore issues related to the computation and communication performance of the Basic Linear Algebra Subroutines (BLAS-1) and related kernels on the SGI/Cray Origin 2000 parallel computer. Experiments are performed both on vendor-supplied mathematical library routines as well as hand-coded loops and array syntax. The goal of this study is to get a better understanding of performance issues pertaining to the Origin 2000 architecture.

2. Architecture

The Cray/SGI Origin 2000 supercomputer is based on the R10000 RISC processor with a clock speed of 195Mhz. Each CPU has 32KB of L1 cache, 4MB of L2 cache and 128MB of main memory. Two CPU chips and their associated hardware form one node that is connected to the rest of the machine via the databus in a fat-bristled hypercube topology.

Due to the two levels of cache memory, some explanation is in order regarding their usage and resulting performance. As a general rule-of-thumb, the closer data is to the CPU in the register/cache/RAM hierarchy, the faster the computations will be. When data is in the registers on the CPU chip, very little effort is needed to get the data processed. Once the registers are full or data is needed that is not already in registers, the L1 cache is searched. The latency period to load data from L1 cache to be processed is 2-3 clock cycles or roughly 15ns. When the data is in the L2 cache, 8-10 cycles are needed to retrieve it. If that data is out in the main memory on the local node, approximately 300ns or 60 clock cycles are needed. If the data is on another node's memory, up to 1100ns are required to fetch it depending on how many routers are encountered along the way.

It is clearly in the best interest of the code developer to utilize arrays that stay in the memory closest to the processor using the data. This of course cannot always happen and so the operating system must clear the cache out in order to move needed data in. This process is determined on a Least Recently Used (LRU) basis; the data that has not been used for the longest period of time is flushed out of the cache first. Because the L1 cache is 2-way set-associative, the data may reside in one of two places determined by the low-order bits of the data's address. If two pieces of data have the same low-order bits, both may be in the cache at once. If a third piece of data with the same low-order bits is needed, the least-recently used piece of data at that address is replaced. If much of the data is not in the cache or keeps getting cleared from the cache at the time it is needed, the program's performance can degrade substantially.

3. Description of the BLAS-1 Experiments

All of the experiments were done on the Origin 2000 machine at Los Alamos National Laboratory. It is currently configured with 128 processors but is expected to be expanded in the near future. All of the experiments were compiled with the command "`f90 -c -64 -mips4 -O3 -r10000 -OPT:roundoff=3 -OPT:IEEE_arithmetic=3`".

We tested the following double precision (64-bit) BLAS-1 operations (note: x and y are vectors and a is a scalar):

- call DAXPY(n,a,x,1,y,1): $y = y + ax$
- call DCOPY(n,x,1,y,1): $y = x$
- call DSCAL(n,a,y,1): $y = ay$
- a = DDOT(n,x,1,y,1): $a = x^T y$
- a = DNRM2(n,y,1): $a = \sqrt{\sum_i y_i \cdot y_i}$
- a = DASUM(n,y,1): $a = \sum_i | y_i |$

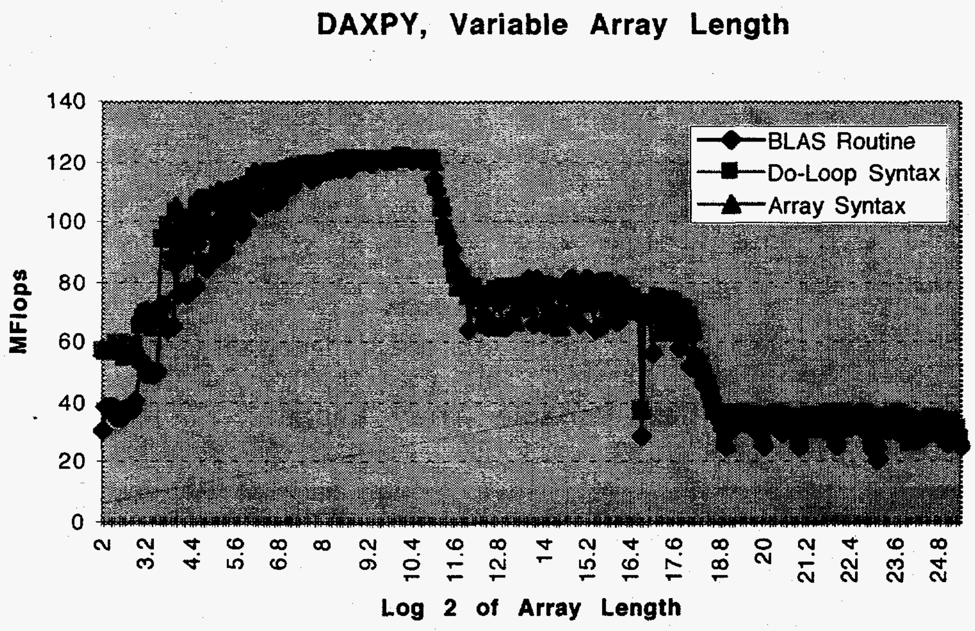
4. BLAS-1 Kernel Performance as a Function of Vector Length

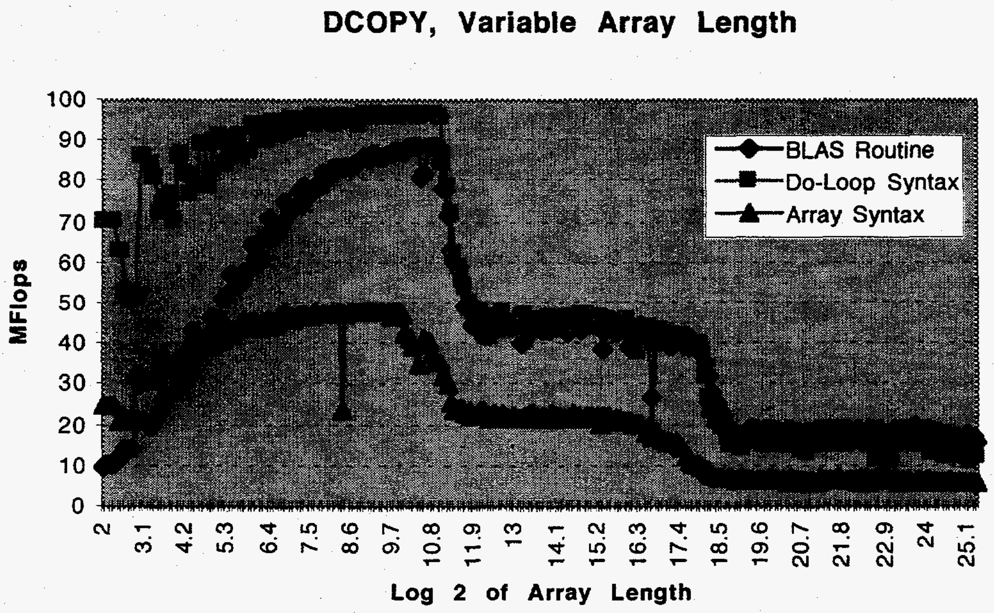
This section of the paper describes the performance of BLAS-1 kernels as a function of vector length. The experiments are done for vector lengths from 2^2 to $2^{25.5}$ words. The vectors are stored in a single array with some padding between where the first array ends and the second array begins to avoid performance problems with power-of-two multiples of array length. The array alignment in memory was kept constant for these experiments. Stride-1 arithmetic is used in all cases. In order to get accurate timings, the given operation was placed in a loop and executed many times in order to get the timing value. The `gettimeofday` system call was used to obtain the timings. The runs were performed in a dedicated batch queue on the Origin 2000.

The results of all six BLAS routines are similar: the performance starts out poorly and increases to a maximum until the array is too large for L1 cache at which time the performance drops significantly to a near constant level in the L2 cache. When the arrays become too large for the L2 cache, performance drops again while the machine has to use the main memory to store the data. The use of the array padding dramatically smoothed out the performance levels for the arrays within the L2 cache memory size.

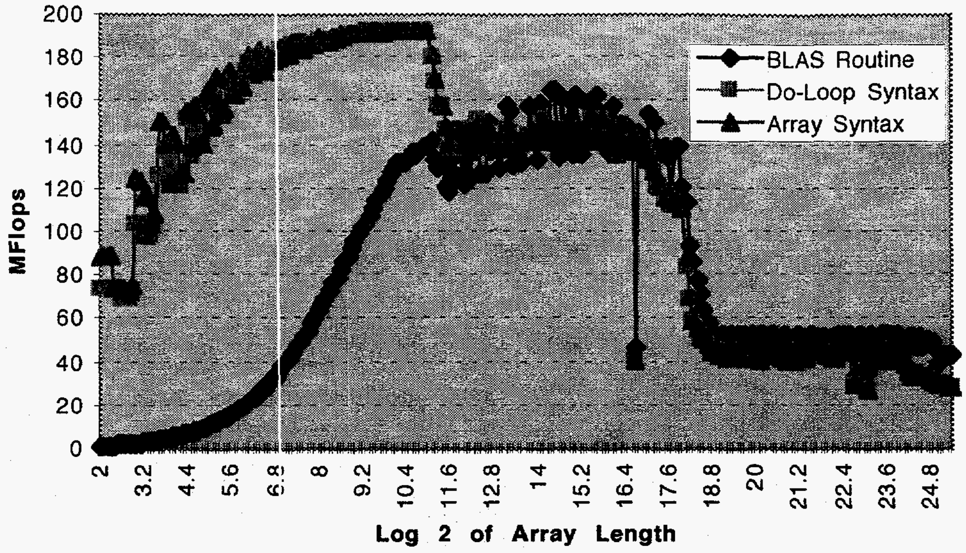
Each method of coding the function, whether it was the system BLAS function, the do-loop syntax or the array syntax, yielded plots of similar shape. The best performance came from the arrays with sizes between 500 and 2000 elements, in which case both vectors stay in L1 cache. In almost all cases, each of the three methods performed about equally well near the peak. The exceptions were the do-loop syntax on the DASUM code and the system routine on DNRM2. In both of those cases, they performed very poorly compared to the other methods. In addition, the array syntax on the DCOPY code performed at only about half the speed of the other two methods. For arrays of length less than about 32 elements, the system routine always suffered the worst performance, due mainly to the overhead of the subroutine call. Unfortunately, in some cases the BLAS routine underperformed the do-loop or array syntax versions, even for long vectors, in some cases significantly.

Some of the graphs denote a downward performance spike at vector length approximately $2^{16.7}$. This behavior was repeatable from run to run. We have no explanation for this behavior at present.

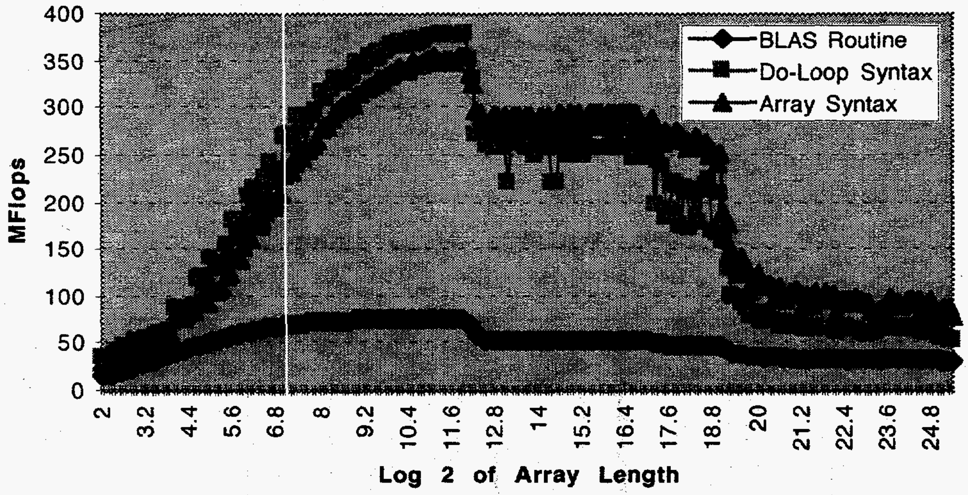




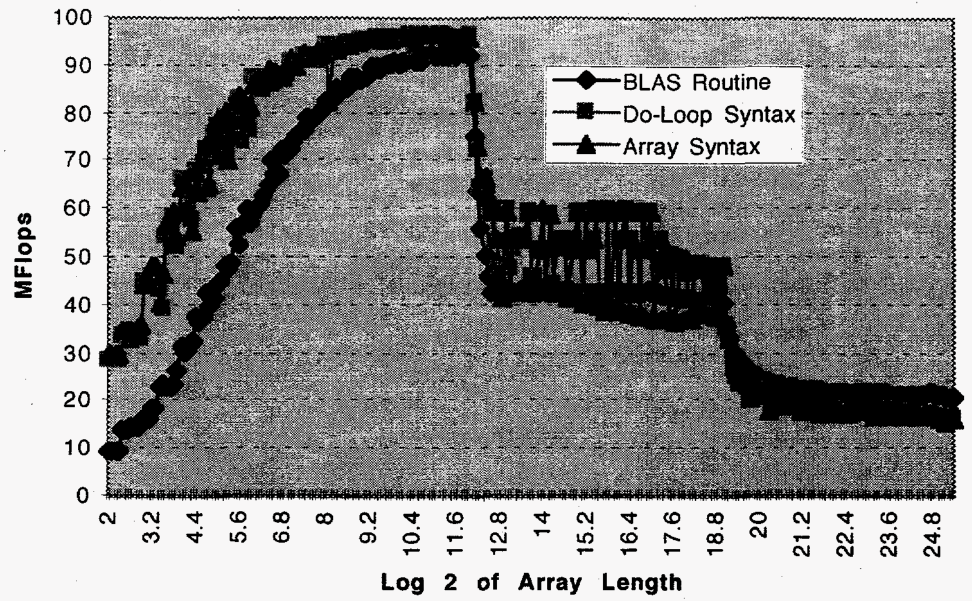
DDOT, Variable Array Length



DNRM2, Variable Array Length



DSCAL, Variable Array Length



5. BLAS-1 Kernel Performance as a Function of Vector Alignment

In this series of experiments, the locations of the x and y vectors were varied within a larger array to examine how alignment affected performance. The y array was started at the first location in the larger array and then the x array was moved at 4 word increments. Thus the relative starting locations of the two vectors was varied through a range of values modulo the size of an L1 cache image.

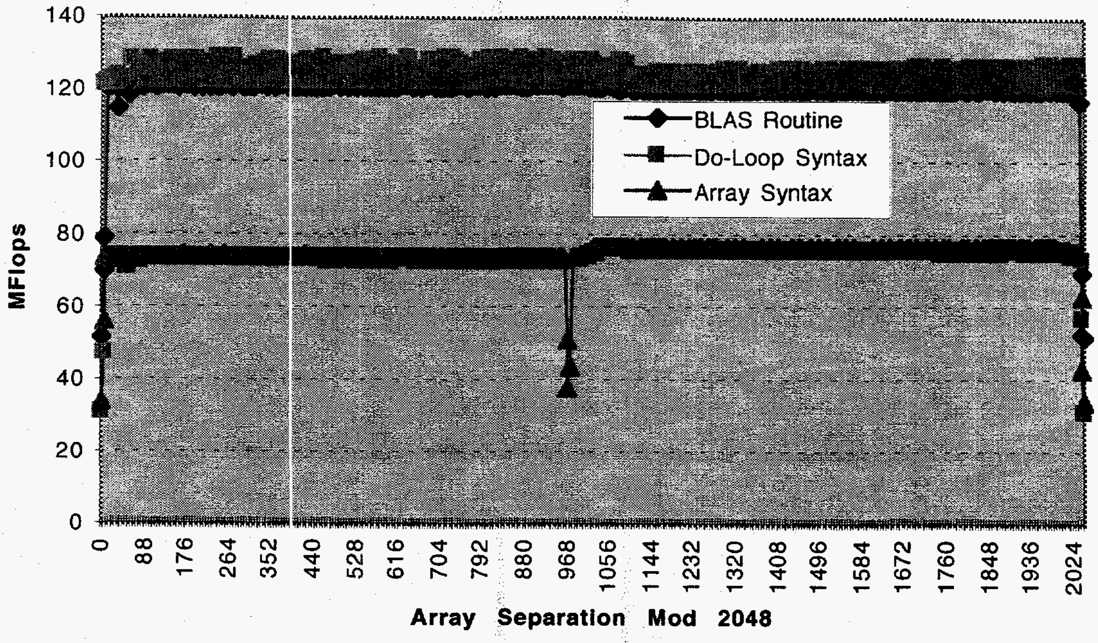
The length of the x and y arrays were kept constant at 1024 elements each. Three methods of coding each BLAS function were run; the BLAS system routine, the do-loop syntax and the array syntax.

The performance of the operations requiring the use of just one array (DSCAL, DASUM and DNRM2) remained constant. Their performance is shown in the following table.

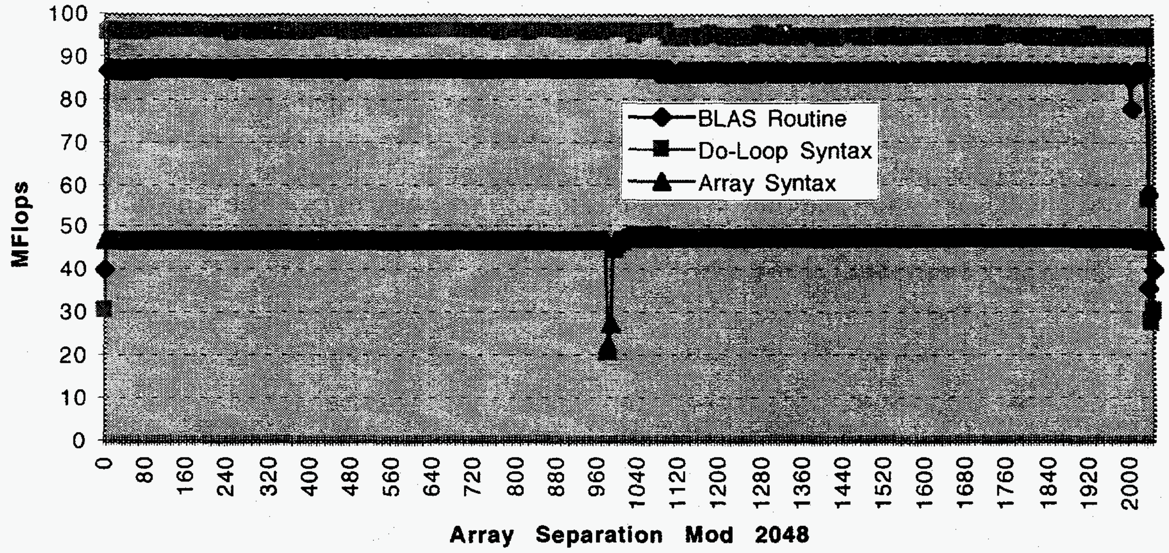
	Performance in MFlops		
Function Name	BLAS Routine	Do-Loop Syntax	Array Syntax
DSCAL	181	181	181
DASUM	30	85	85
DNRM2	76	48	377

The performance of the BLAS functions requiring two arrays is considerably more interesting. As the array locations in memory start to overlap in cache, the performance drops considerably as the cache is thrashed. The following three graphs illustrate these results.

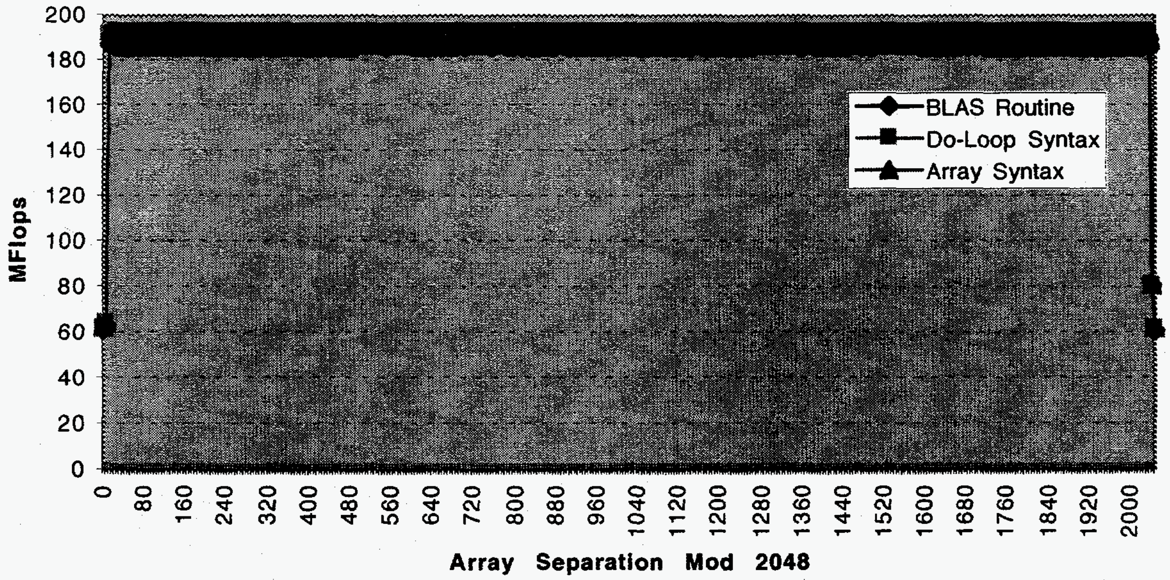
DAXPY, Variable Array Alignment



DCOPY, Variable Array Alignment



DDOT, Variable Array Alignment

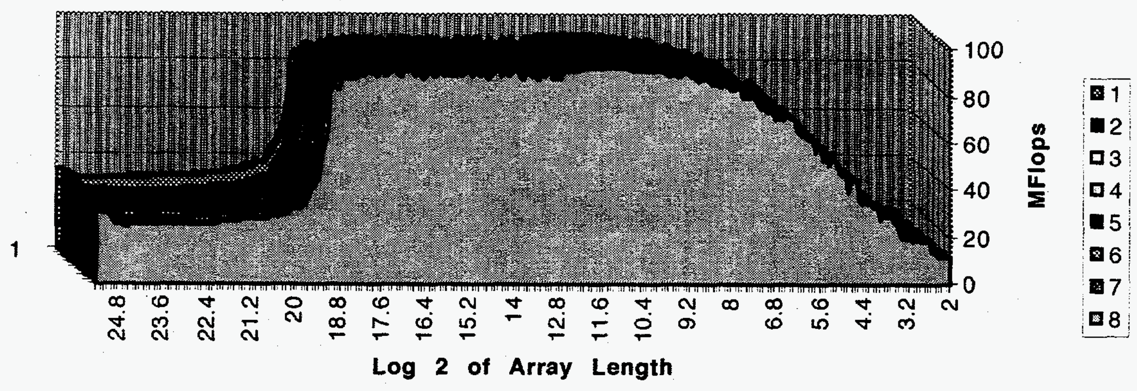


Both the L1 cache and the L2 cache on the Origin 2000 processor are two-way set associative; thus, it should be possible to store two vectors in the cache without any performance hit. However, the L2 cache behaves in such a way that whenever a switch is made between lines in a cache set, a stall is incurred. Nonetheless, the above experiments are for short vectors that fit into L1 cache; thus, we cannot fully explain the performance dropoff when the vectors line up evenly in the L1 cache.

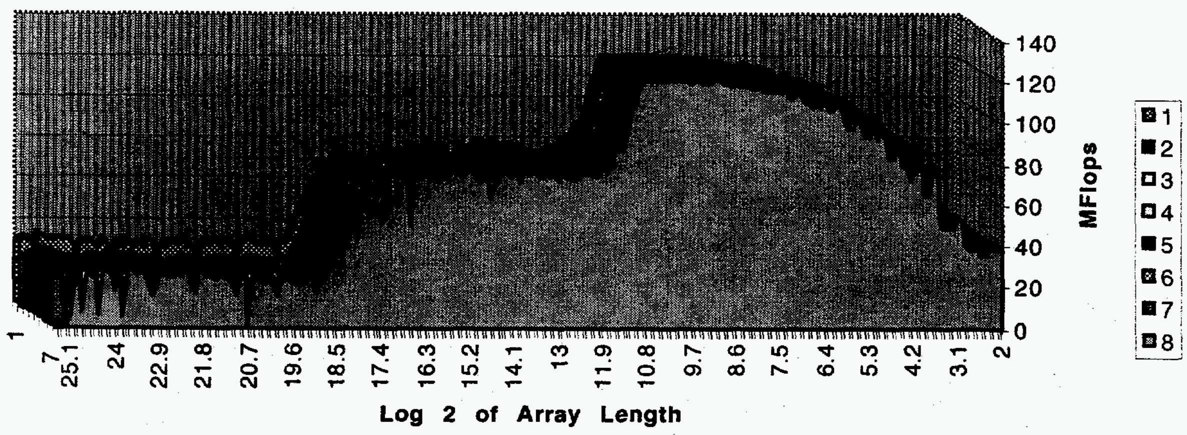
6. BLAS-1 Routines on Multiple CPUs

The next set of experiments was done with the same six BLAS-1 routines but involved 1-8 CPUs. The array length was varied from 2^2 to $2^{25.5}$ elements. The array alignment was not a variable in these tests. By increasing the number of CPUs, we could see the effect of the increasing memory demands on the bus even though the tests did not involve any communications. The results for these series of tests are shown on the following six 3-dimensional graphs. Overall, the greatest effect of increasing the number of CPUs seems to have occurred in going from one to two processors. This is due to the fact that there are two CPUs on a hub and they share a bus to their memory units. The slowdown is greatest in the part of the test that required access to RAM; computations with short arrays did not show any substantial slowdown.

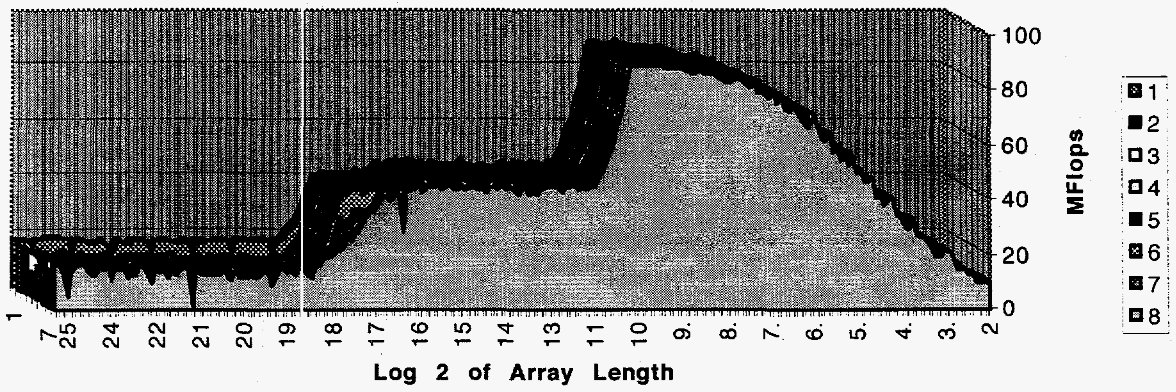
DASUM, Multiple CPUs



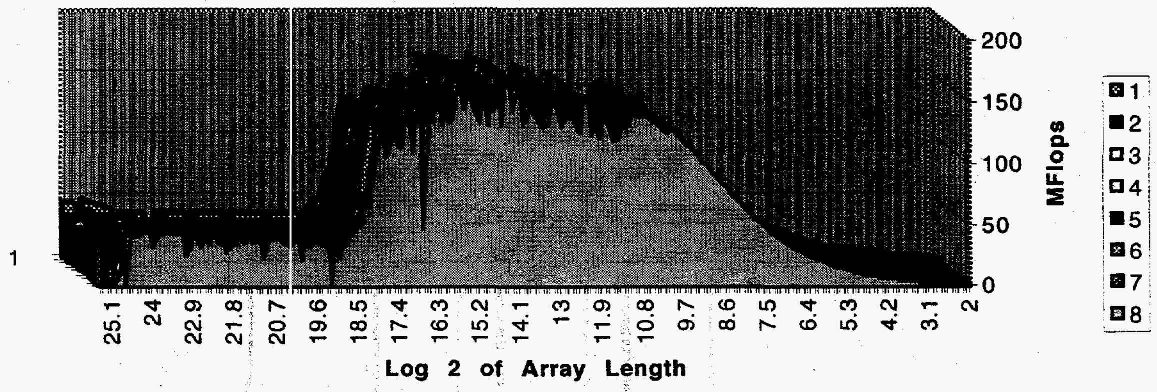
DAXPY, Multiple CPUs



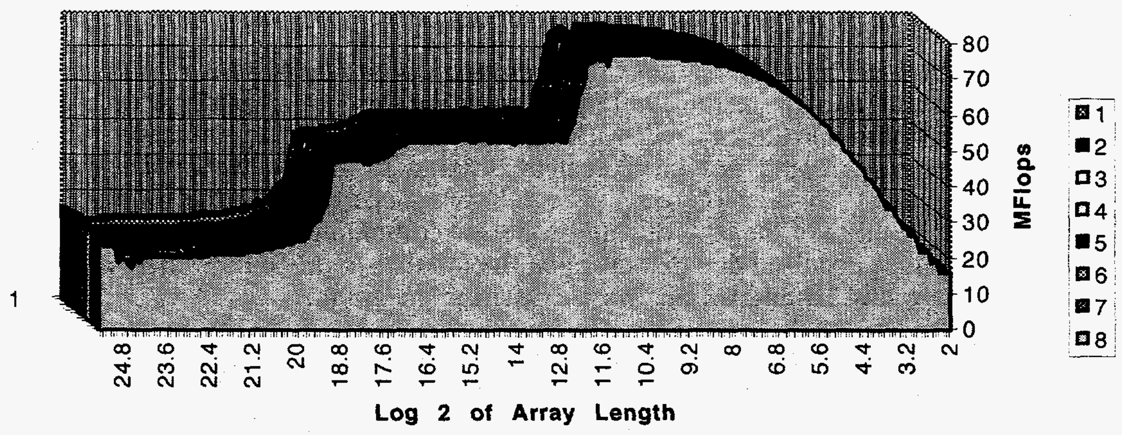
DCOPY, Multiple CPUs



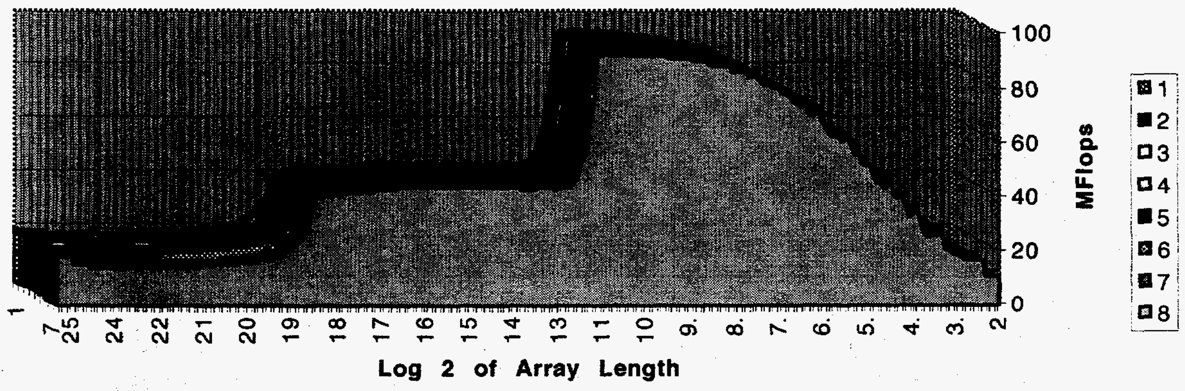
DDOT, Multiple CPUs



DNRM2, Multiple CPUs



DSCAL, Multiple CPUs



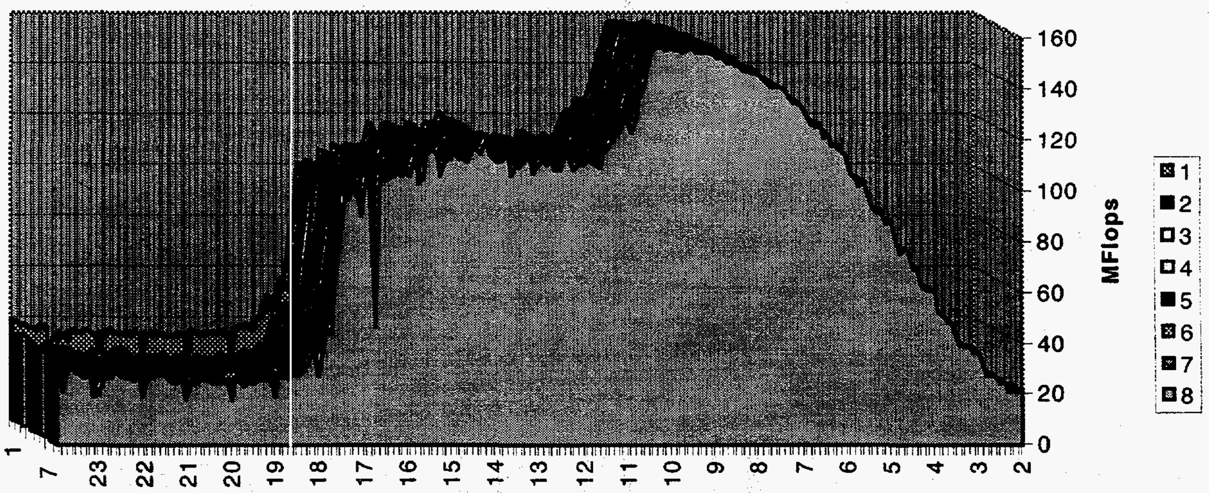
7. Rank-1 Update with Local Summations

This set of tests examines the performance of the computer performing the following computation:

$$w = w + u\left[\frac{v \cdot u}{v \cdot w}\right],$$

where u, v and w are all vectors. This requires two dot products (DDOT) and a DAXPY. In this experiment, all summations were performed on the local memory so there were no interprocessor communications needed. The length of the arrays were varied from 2^2 to 2^{24} words. The arrays could not be allocated as large as before since there were three allocations necessary instead of two. The following chart shows the results from this computation. Because of the lack of interprocessor communication, very little degradation in the performance occurred due to increasing the number of processors. The effects of the arrays running out of L1 and L2 cache are evident in the graph.

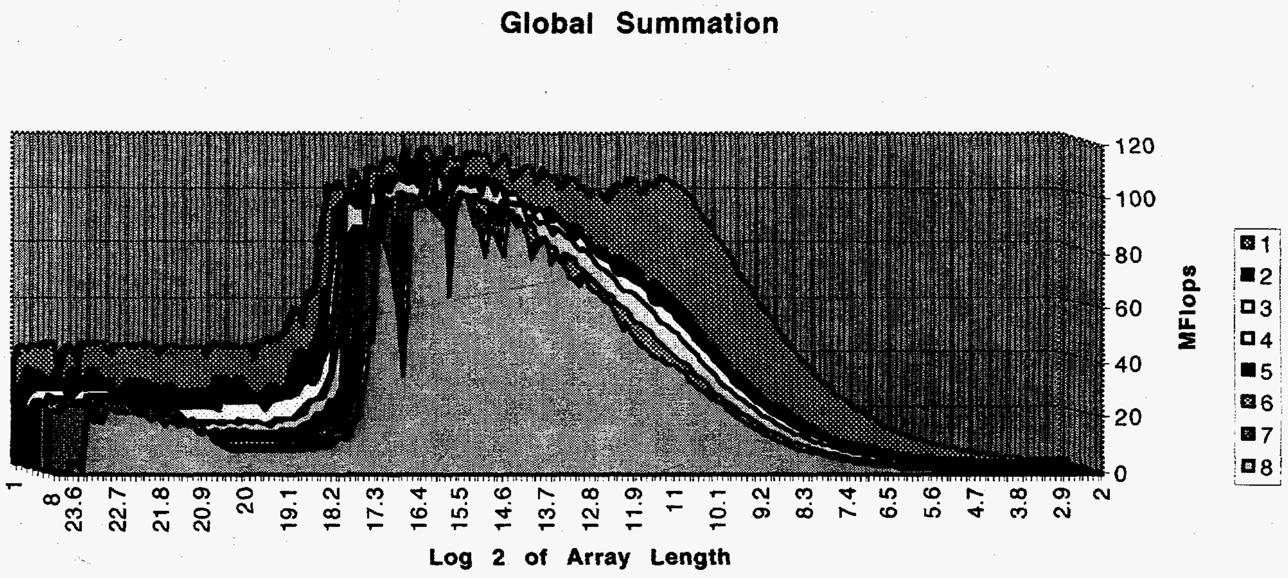
Local Summation, Rank-1 Update



Log 2 of Array Length

8. Rank-1 Update with Global Summations

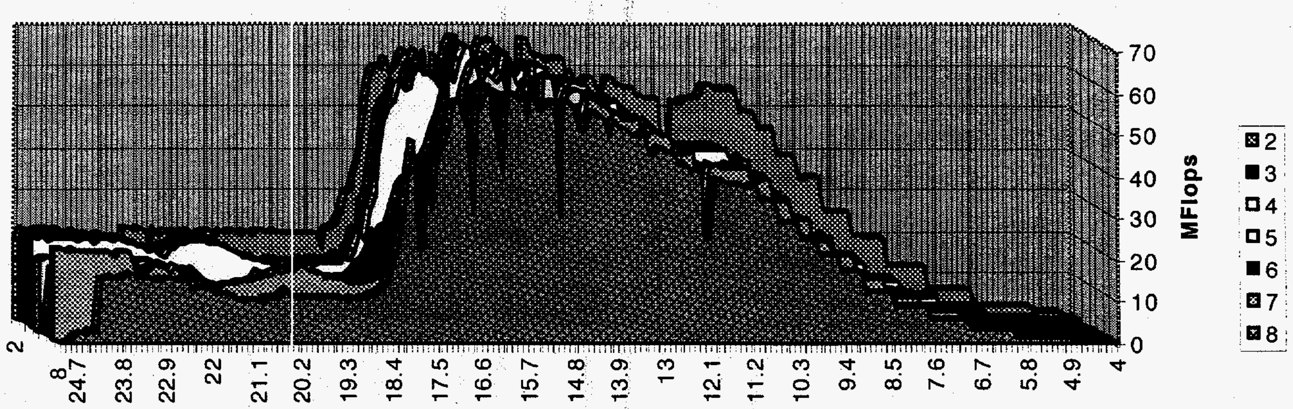
This experiment is very similar to the last one except for the need for each CPU to get summation data off of each of the other CPUs. This was implemented with the MPI subroutine `mpi_allreduce` which made use of the pre-defined operation `MPI_SUM`. The insertion of the `mpi_allreduce` subroutine call caused some performance degradation. This is because each CPU had to go around to other CPUs and get data. This apparently ties up a large amount of bandwidth on the data bus and slows the computations down. Unfortunately, this slowdown is extremely severe, even for very long vectors, in which case the global sum operation of one scalar value per processor should be small compared to the on-processor computation.



9. Face Communications on 2-8 CPUs

The last tests performed on the Origin 2000 were face communications. Each CPU had a cube-shaped 3-D array allocated on it and then one face of the array was sent to the adjacent CPU. That processor received the data and then performed a DAXPY calculation with another 3-D array already resident in its memory. As the program ran, the arrays were allocated to be larger and larger. Then the number of CPUs were also increased to see the effect of increased communication requirements. This experiment is intended to show performance that can be expected from kernels used in large-scale 3-D simulations. The results show some amount of drop of performance relative to the communication-free DAXPY results given earlier in the paper.

Face Communications, 2-8 CPUs



Log 2 of Allocated Memory

10. Conclusions

Several issues need to be taken into account in order to obtain maximum performance from the Origin 2000. The first and possibly the easiest way to gain speed is to block all of the arrays and loop computations so that they do not exceed L1 cache size, whenever possible. Performance in all of the functions was highest in places where the arrays were not large enough to overflow L1 cache. While the performance is not very close to the advertised peak CPU speed, performance is best for arrays of this length. When arrays that are this short are not possible, L2 cache still affords good performance. Arrays that ran off of the end of the RAM and had to be stored on other CPUs' memory afforded poor performance in some cases and should be minimized when possible.

Array alignment played an important role when more than one data structure was involved. If the code developer takes into account the alignment of the arrays in memory, loses in performance can be avoided. This may be done by avoiding powers of two or by padding, for example.

When interprocessor communication becomes an issue, from one standpoint the best performance is reached for smaller arrays. In this case, data is kept in cache, improving performance. On the other hand, larger arrays are useful for reducing the communication-to-computation ratio, thus improving performance. The combination of having to do interprocessor communication in addition to storing array data on far CPUs will cause serious slowdowns on the data bus. Also, in the current version of the system software, global sums incur a much larger performance penalty than face communications.

Acknowledgements

This work was supported in part by the Department of Energy through grant W-7405-ENG-36, with Los Alamos National Laboratory. This research was performed in part using the resources located at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545.