Architecture Characterization of DoD MSRC HPC Platforms

Philip Mucci* and Kevin London
University of Tennessee
Knoxville, Tennessee 37996-1301

Abstract

This paper outlines the results for a set of low-level, architecture characterization benchmarks that measure the performance of dense numerical computations, access to the memory hierarchy and MPI message passing of three high performance architectures. The machines evaluated are the Cray T3E, IBM SP, and SGI Origin 2000 platforms at the CEWES Major Shared Resource Center. Verification of the results was done at the ARL and ASC MSRCs. The data presented here should be of interest to developers as a point of reference for application performance, modeling and scalability analysis.

Introduction

This paper describes three low-level, single processor benchmarks developed to measure some critical factors affecting application performance among the newest parallel machines at various DoD MSRCs. A brief summary of the results is also included here. For further details, the reader is referred to the full version of this paper that can be found at http://www.cs.utk.edu/~mucci/dodmsrc.html.

1. Introduction to CacheBench

CacheBench is a benchmark designed to evaluate the performance of the memory hierarchy of computer systems. Its specific focus is to parameterize the performance of possibly multiple levels of cache present on and off the processor. By performance, we mean raw bandwidth in megabytes per second. Of interest to us is the ability of the cache to sustain large, unit-stride, floating point workloads. The goal of this benchmark is to establish peak computation rate given optimal cache reuse. Many scientific applications in use have significant resource requirements in terms of memory footprint. High speedups of these applications are often achieved through exploiting the cache. This is especially true given the widening gap between processor speed and main memory. Thus this benchmark will provide us with a good basis for application performance modeling and prediction for those applications that have already been substantially tuned for cache reuse.

A detailed discussion of cache and processor architecture is well beyond the scope of this paper, but interested readers are referred to Hennessey and Patterson (1996). An example in this textbook serves as the basis for this benchmark.

1.2 Description of CacheBench

CacheBench currently incorporates eight different benchmarks. Each of the eight performs repeated access to data items on varying vector lengths. Timings are taken for each vector length over a number of iterations. Computing the product of iterations and vector length gives us the total amount of data accessed in bytes. In addition to this figure, the access time in nanoseconds per each data item is computed and reported. The eight tests are Cache Read, Cache Write, Cache Read/Modify/Write, Hand-tuned Cache Read, Hand-tuned Cache Write, Hand-tuned Cache Read/Modify/Write, memset(), and memcpy(). The first three of the tests are intended to provide us with information about how good our compiler is. They are very straightforward consisting of only a few lines of code. The second three are intended to

reflect portable, tuned code as found in production applications. Here the optimizer has little opportunity to enhance the code, and in fact, the numbers from these three tests often do not change very much given different levels of optimization. The last two tests are included as points of comparison. These routines are often heavily used in C applications, but vary greatly in efficiency. One would expect high performance out of these benchmarks in terms of memory bandwidth, but more often than not, the results have been disappointing.

1.3 Sample Results - Cache Read/Modify/Write

This benchmark is designed to provide us with read/modify/write bandwidth for varying vector lengths in a compiler-optimized loop. This benchmark generates twice as much memory traffic, as each data item must be first read from memory/cache to register and then back to cache. Each direction of transfer is counted in the computation of bandwidth. Bandwidth for this test is often a bit higher than the sum of the previous two tests. The benefit comes from the compilers' ability to better schedule operations and group memory accesses to amortize the cost of the store.

Cachebench results are shown and explained for the two versions of the read/modify/write benchmark on the SGI Origin 2000, IBM SP, and Cray T3E. Of interest in figure 1 and figure 2 is the difference in performance of the IBM SP. Note that in the hand-tuned version, performance averages about six hundred megabytes per second better than that of the compiler-optimized version. In the tuned version, the compiler is probably scheduling/aggregating memory access into double-word loads and stores, a unique feature of this architecture. This definitely happens in the compiler optimized version, but the fact that the compiler must also unroll the loop and optimize register usage seems to complicate its analysis. Also of interest is the better performance on the T3E in level two cache for the untuned version. Software pipelining, the mixing instructions from one iteration to another may be aiding this code to hide the latency of the level two cache misses.

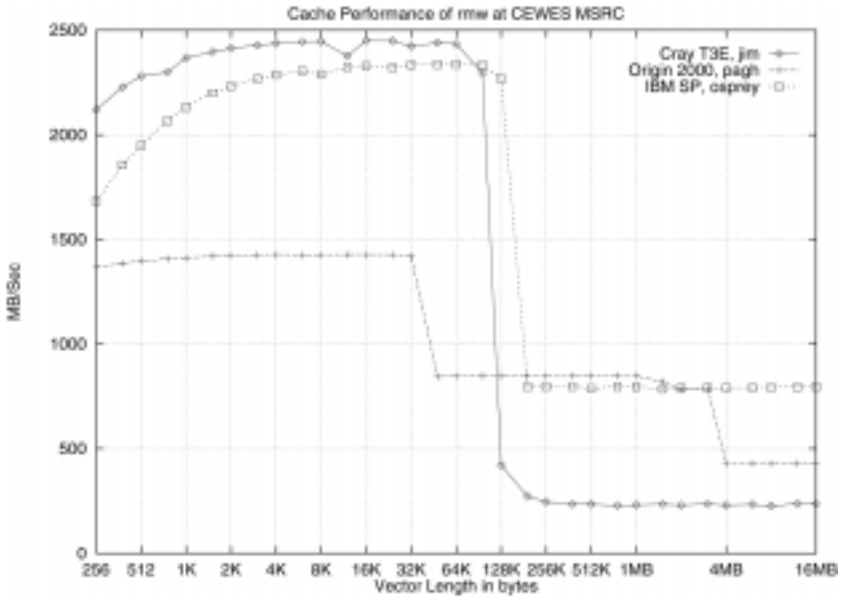

Figure 1. Read/Modify/Write performance for the memory hierarchy.

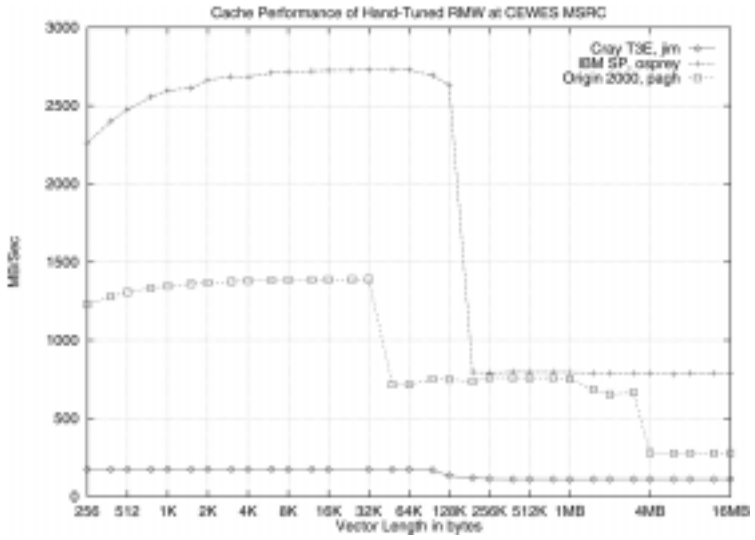

Figure 2. Hand-tuned Read/Modify/Write performance for the memory hierarchy.

2. Introduction to BLASBench

BLASBench is a benchmark designed to evaluate the performance of some kernel operations of the BLAS or Basic Linear Algebra Subroutines. These routines are found on all high performance architectures. The BLAS routines are designed to provide a standardized high performance API for common Vector-Vector, Vector-Matrix and Matrix-Matrix operations. These routines are expected to be the building blocks for larger numeric intensive algorithms and computations. As such, we expect them to be highly tuned for each specific architecture obtain as close to peak performance as is achievable.

2.1 Goals of BLASBench

BLASBench aims to provide the following:

- Evaluate the performance of the BLAS routines in MFLOPS/sec.
- Provide information for performance modeling of applications that make heavy use of the BLAS.
- Evaluate compiler efficiency by comparing performance of the reference BLAS versus the hand-tuned BLAS provided by the vendor.
- Validate vendor claims about the numerical performance of their processor.
- Compare against peak cache performance to establish bottleneck, memory or CPU.

2.2 Description of BLASBench

BLASBench currently evaluates the performance of the three most frequently used routines in the BLAS. They are:

- AXPY: Vector addition with scale
- GEMV: Matrix-vector multiplication with scale
- GEMM: Matrix-Matrix multiplication with scale

The benchmark can run in either single or double precision. This is important as many systems cannot sustain the additional memory bandwidth required by using double precision data. The test space is highly tunable, as are some of the metrics BLASBench reports. It reports its results in MFLOPS/sec and MB/sec. These numbers are not computed from hardware statistics, but rather from the absolute operation and memory reference count required by each algorithm.

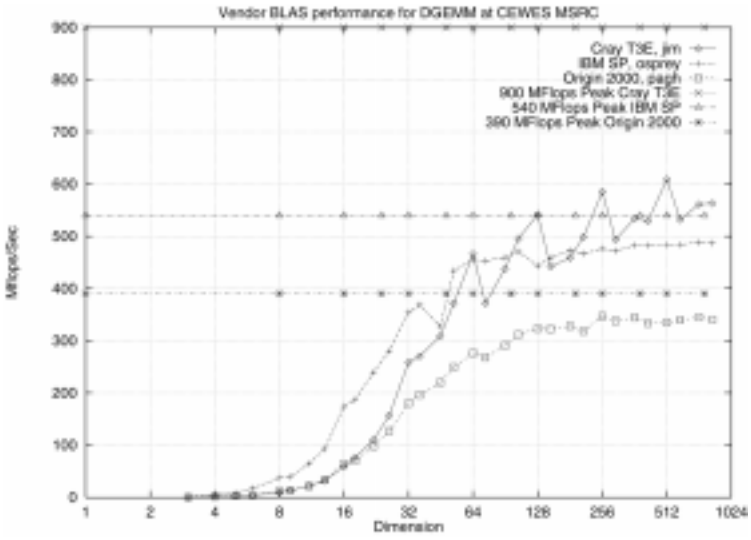

Figure 3. BLAS DGEMM performance.

2.3 Sample Results - Matrix Multiplication

Matrix multiplication provides a lot of opportunity for cache reuse due to the fact that the number of computations is an order of magnitude larger than the number of unique memory references. By tiling the matrices appropriately, the working set of the problem can be reduced to the size of cache. Thus a data item is only loaded into the cache once and discarded once that portion of the matrix has been finished. Thus all cache misses become capacity misses and conflict misses are eliminated entirely. The performance of the GEMM routine has long served as a good indicator of a machine's peak practical performance. A machine with an adequate memory subsystem and a well-blocked implementation, can achieve very high efficiencies, that is a high percentage of the processors peak MFLOP rating.

BLASBench results are presented for double precision matrix multiply (DGEMM) for the SGI Origin 2000, IBM SP, and Cray T3E. In figure 3, the spikes in the T3E's performance curve are due to the matrix dimension being a multiple of the block size. For other cases, the GEMM routine must engage in rather lengthy cleanup code. The small direct-mapped 8K cache with its 32 byte line size of the T3E exaggerates the performance loss. The Origin also suffers from a 32 byte line size and its inability to issue a load/store every cycle. The R10000 processor, like the Power2, can also issue two multiply-adds per cycle, but it appears to have great difficulty keeping those functional units busy due to stalls in the memory pipeline. The SP, with its large line size of 256 bytes and ability to issue two multiply-add instructions as well as a load/store per cycle, does quite well, reaching approximately 90 percent efficiency.

3. Introduction to MPBench

MPBench is a benchmark to evaluate the performance of MPI and PVM on MPP's and clusters of workstations. It uses a flexible and portable framework to allow benchmarking of any message passing layer with similar send and receive semantics. The goal of including this benchmark is to spot poor MPI implementations so that the application engineer might be able to restructure his communication patterns. MPBench currently measures six different types of message passing performance; Application latency, Roundtrip time Send, Broadcast, Reduce and AllReduce bandwidth.

MPBench measures messages from 4 bytes to 16 Megabytes, in powers of two for 500 iterations. The tests are repeated to make sure the cache is pre-loaded. This is because applications typically communicate data soon after computing on it. This results in all or a portion of data residing in cache.

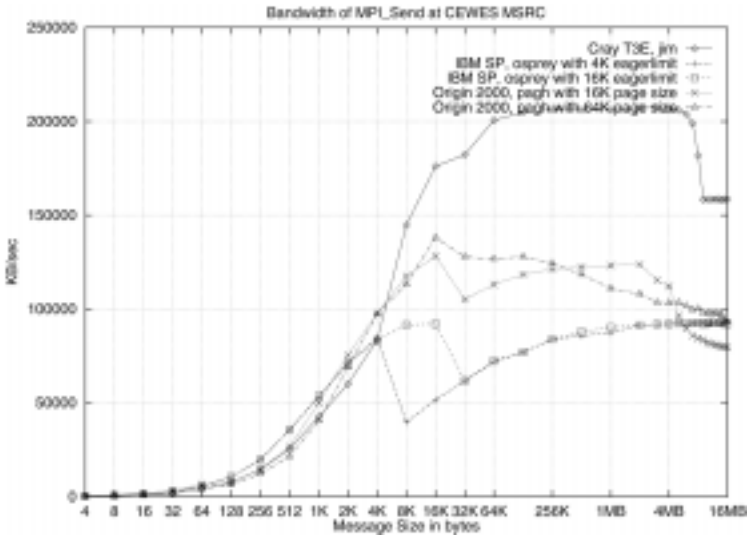

Figure 4. MPI_Send bandwidth

3.1 Sample Results - Send Bandwidth

MPBench measures bandwidth with a doubly nested loop. The outer loop varies the message size, and the inner loop measures the send operation over the iteration count. After the iteration count is reached, the slave process acknowledges the data it has received by sending a 4 byte message back to the master. This informs the sender when the slaves have completely finished receiving their data and are ready to proceed. This is necessary, because the send on the master may complete before the matching receive does on the slave. This exchange does introduce additional overhead, but given a large iteration count, its effect is minimal.

Figure 4 shows the bandwidth curve of MPI_Send for the SGI Origin 2000, IBM SP, and Cray T3E. We note the dramatic effect of MPI's rendezvous protocol on all three machines. When transferring messages larger than a certain size, MPI waits for an acknowledgment by the receiver that contains the address of the destination buffer as passed to MPI_Recv. Messages smaller than this size are sent "eagerly", without delay. This guarantees that the reciever has enough space fo the message and is ready to receive it. The default size of this limit is 4Kb for the SP and 16Kb for the T3E and the Origin. We have found that increasing the eager limit results in much better performance on the SP and the Origin. On the SP, this parameter is passed to the "poe" MPI environment via an environment variable or command line option. On the SGI, this is accomplished by changing the page size of the executable with "dplace" as message transfers occur via distributed shared memory. Note the severe performance loss at the 8MB mark on the T3E. This result is completely repeatable and is a factor in every measurement taken by MPBench. An explanation for this has yet to be found, thus we recommend that all transfers larger than 8MB be fragmented by the user into multiple calls to MPI_Send.

Conclusion

Benchmarking for performance characterization has always been a point of contention. Arguments can be made both for and against benchmarking and its relation to overall machine performance. Our goal in producing these benchmarks is not to conclude that one machine "is better" than another. The platforms mentioned here are highly complex and a full understanding of their characteristics and often beyond the scope of application developers on a deadline. The goal of these benchmarks is to provide concise and relevant information about these machines to help the users obtain better application performance.

References

1. Patterson, D.A., Hennessy, J.L., Goldberg, D., *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishing, Stanford, CA, 1996.

Keywords

benchmarking, cache, performance, MPI, message passing, BLAS, MFLOPS, processor, bandwidth, architecture