Low Level Architectural Characterization Benchmarks for Parallel Computers

Philip J. Mucci
Kevin S. London
mucci@cs.utk.edu
london@cs.utk.edu

May, 1998

Abstract

This paper 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 performed at the ARL and ASC MSRCs. The data and benchmarks presented here should be of interest to developers as a point of reference for application performance, modeling and scalability analysis.

1 BLASBench

1.1 Introduction

BLASBench is a benchmark designed to evaluate the performance of some kernel operations of different implementations of the BLAS or Basic Linear Algebra Subroutines. The BLAS are found in some form or another on most vendors' machines and were initially developed as part of LAPACK. Their goal was to provide a standardized API for common Vector-Vector, Vector-Matrix and Matrix-Matrix operations. A version of the BLAS is available from Netlib at `http://www.netlib.org/`. This version is subsequently referred to as the reference version. The reference BLAS are completely unoptimized Fortran codes intended as a reference for correctness to the vendors.

1.2 Goals of BLASBench

BLASBench aims to do 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.

1.3 Description

BLASBench currently benchmarks the three most common 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. The benchmark 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.

1.4 How it works

BLASBench is a C program that calls BLAS routines written in Fortran so that dynamic memory allocation can be performed. For each test, BLASBench allocates its memory in such a way that the total amount of memory taken up by each test is less than or equal to the nearest power of two. The rationale for this is that our cache sizes are always in a power of two. Once the memory is allocated, the arrays are initialized, and the test is run for a certain number of iterations. By default, the iteration count is not constant over each problem size. BLASBench trys to keep the amount of data "touched" by each run constant. This means that larger problem sizes run for fewer iterations. The effect of this is that the run time for each size is approximately constant. BLASBench provides an option to keep the iteration count constant across all tests. After each size is tested, the cache is flushed and BLASBench either proceeds to the next size or repeats that size, depending on the options given to the program. BLASBench allows you to repeat each size any number of times. This could be used to validate the numbers from each size on a time-shared system. By default, BLASBench calls the BLAS routines with the leading dimension of each array or vector set to the exact size of that vector. This means that the BLAS routines operate on the entire data set. Frequently, however, BLAS routines are called upon smaller portions of larger arrays and matrices, thus an option is provided to keep the leading dimension constant among every test. In this case, BLASBench allocates the largest possible data set, and simply changes the working set size passed to each BLAS routine.

1.5 Using BLASBench

1.5.1 Obtain the Distribution

Download the latest release from either of the following URLs:

`http://www.cs.utk.edu/~mucci/blasbench`
`ftp://cs.utk.edu/pub/mucci/blasbench.tar.gz`

Now unpack the installation using `gzip` and `tar`.

```
kiwi> gzip -dc blasbench.tar.gz | tar xvf -
kiwi> cd blasbench
kiwi>  ls
CVS/            Version         blasgraph.gp    index.html
Makefile        bb.c            conf/           make.def@
README          blasbench.html  doc/            samples/
```

1.5.2 Build the Distribution

First we must configure the build for our machine, OS and BLAS libraries. All configurations support the reference BLAS if available. Running `make` with no arguments lists the possible targets.

```
kiwi> make
Please use one of the following targets:

        sunos sunos4
        solaris sunos5
        sunmp
        alpha
        linux
        hppa
        sgi
        sgi-o2k o2k
        sgi-pca pca
        t3e
        t3d
        ibm-pow2 ibm-sp2 sp2 pow2
        ibm-pow pow
```

Configure the build. Here, we are on a Solaris workstation.

```
kiwi> make solaris
ln -s conf/make.solaris make.def

Please check the VBLASLIB variable in make.def and make sure
that it is pointing to the vendor BLAS library if one exists.
Then type 'make'.
```

Examine the `make.def` file to ensure proper compiler flags and paths to the different BLAS libraries. The BLASLIB variable should contain the absolute path to the reference BLAS library and the VBLASLIB variable should contain the absolute path to the vendor's BLAS library. If one or the other is not available, just leave it blank and that specific executable will not be generated.

Now build it. Depending on whether or not both BLASLIB and VBLASLIB are set, one or two executables will be generated.

```
kiwi> make
cc -fast -dalign -DREGISTER   -c bb.c -o bb.o
if [ -f "/src/icl/LAPACK_LIBS/blas_SUN4SOL2.a" ];
  then f77  -o blasbench bb.o /src/icl/LAPACK_LIBS/blas_SUN4SOL2.a ; fi;
if [ -f "/kevlar/homes/susan/bin/solaris/libsunperf.a" ];
  then f77  -o vblasbench bb.o /kevlar/homes/susan/bin/solaris/libsunperf.a ;
fi;

Now type 'make run'.
```

1.5.3 Running BLASBench

BLASBench can be run by hand, but it is intended to be run through the makefile. Running it via the makefile automates the collection and presentation process. By default, the makefile runs both executables with the arguments `-c -o -e 1 -i 1`. This says that the iteration count should be constant, the output should be reported in MFLOPS/sec, each size should be repeated only once and the iteration count should be set to one. You can change the default settings by changing the BBOPTS variable in the makefile after you have configured the distribution.

```
kiwi> make run
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -v > daxpy.dat; fi
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -a > dgemv.dat; fi
if [ -x blasbench ]; then blasbench  -e 1 -i 1 -c -o -t > dgemm.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -v > vdaxpy.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -a > vdgemv.dat; fi
if [ -x vblasbench ]; then vblasbench  -e 1 -i 1 -c -o -t > vdgemm.dat; fi
.
.
.
Now do a make datafiles.
```

At this point, depending on whether or not you have GNUplot installed on your system, you have the choice of either packaging up the datafiles for analysis on another machine, or generating the graphs in place.
First we must package up the datafiles.

```
kiwi> make datafiles
.
.
.
```

```
daxpy-kiwi.dat
dgemm-kiwi.dat
dgemv-kiwi.dat
vdaxpy-kiwi.dat
vdgemm-kiwi.dat
vdgemv-kiwi.dat
blasgraph.gp
compare.gp
custom.gp
vcustom.gp
kiwi.info

Now do a 'make graphs'.
If you don't have GNUplot, you can do this on another machine
using the kiwi-bp-datafiles.tar file.
```

Now make the graphs.

```
kiwi> make graphs
gnuplot < custom.gp > blasperf.ps
UTK BLAS graph is in blasperf.ps

gnuplot < vcustom.gp > vblasperf.ps
Vendor BLAS graph is in vblasperf.ps

gnuplot < compare.gp > compare.ps
Vendor BLAS graph is in vblasperf.ps
```

This will result in either 1 or 3 graphs. Each graph contains the performance in megaflops of all three operations. They are named as follows:

- blasperf.ps - Postscript file of the reference BLAS.
- vblasperf.ps - Postscript file of the vendor's BLAS.
- compare.ps - Comparison of the two.

1.5.4 Arguments to BLASBench

```
kiwi> blasbench -h
Usage: blasbench [-vatsco -x # -m # -e # -i #]
         -v AXPY dot product benchmark
```

```
        -a GEMV matrix-vector multiply benchmark
        -t GEMM matrix-matrix multiply benchmark
        -s Use single precision floating point data
        -c Use constant number of iterations
        -o Report Mflops/sec instead of MB/sec
        -x Number of measurements between powers of 2.
        -m Specify the log2(available physical memory)
        -e Repeat count per cache size
        -l Hold LDA and loop over sizes of square submatrices
        -d Report dimension statistics instead of bytes
        -i Maximum iteration count at smallest cache size

Default datatype   : double, 8 bytes
Default datatype   : float, 4 bytes
Defaults if to tty : -vat -x1 -m24 -e2 -i100000
Defaults if to file: -t   -x1 -m24 -e1 -i100000
```

1.6 Results on the CEWES MSRC Machines

The following graphs are taken from our runs on each of the CEWES MSRC machines during dedicated time. The machines are the SGI Origin 2000, the IBM SP and the Cray T3E. The peak MFLOPS is as reported by the vendor and is simply computed as a product of the clock speed times the number of independant FMA's that can be computed per cycle. The cache size and theoretical peak MFLOPS for each machine is listed as follows.

Machine	*Cache*	*Peak*
SGI Origin 2000	32K,4MB	390
IBM SP	128K	240
Cray T3E	8K,96K	900

1.6.1 DAXPY

$$y = y + \alpha b$$

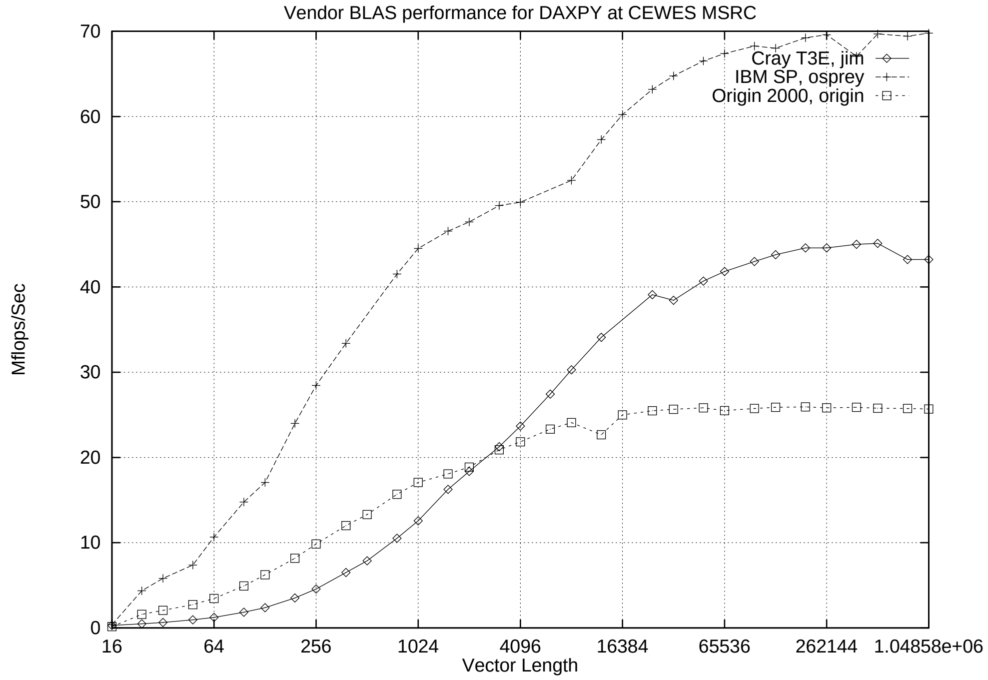

Figure 1: Performance of Vector-Vector Addition with Scale

The DAXPY operation is highly bound by the latency of cache and the throughput of the main memory subsystem. Latency is a factor because data is continuously being replaced in the cache. Every line that is loaded is used once and then discarded. The data from each vector is never *re*-used. Thus, for each cache line, we must incur the cost of flushing the dirty line and loading the new data. Because the data items are accessed sequentially, cache features like requested data first[1] line buffering and non-blocking[2] help very little.

The performance of this benchmark is highly dependent on cache line size, but independent of cache size. The reason is that the cache simply adds latency to memory accesses. As there is no data re-use, the cache is of zero benefit. The size of a cache line reflects the size of the unit of transfer between cache and main memory. Moving a

[1]The cache controller returns the missed-upon operand first and then the rest of the line.
[2]Multiple outstanding misses can be satisfied at once.

lot of data at once is a performance win because of high latency of accessing the main memory.

In figure 1 can been seen the results of performing vector-vector addition for increasing vector lengths of double words. Here we find that the SP far outperforms the other two machines. Interestly enough, the SP has the simplest memory subsystem and the largest line size at 128 bytes. Although, the T3E had the stream buffers enabled its performance was still poor. The Origin's non-blocking cache didn't help either.

1.6.2 DGEMV

$$C = \alpha Ax + \beta y$$

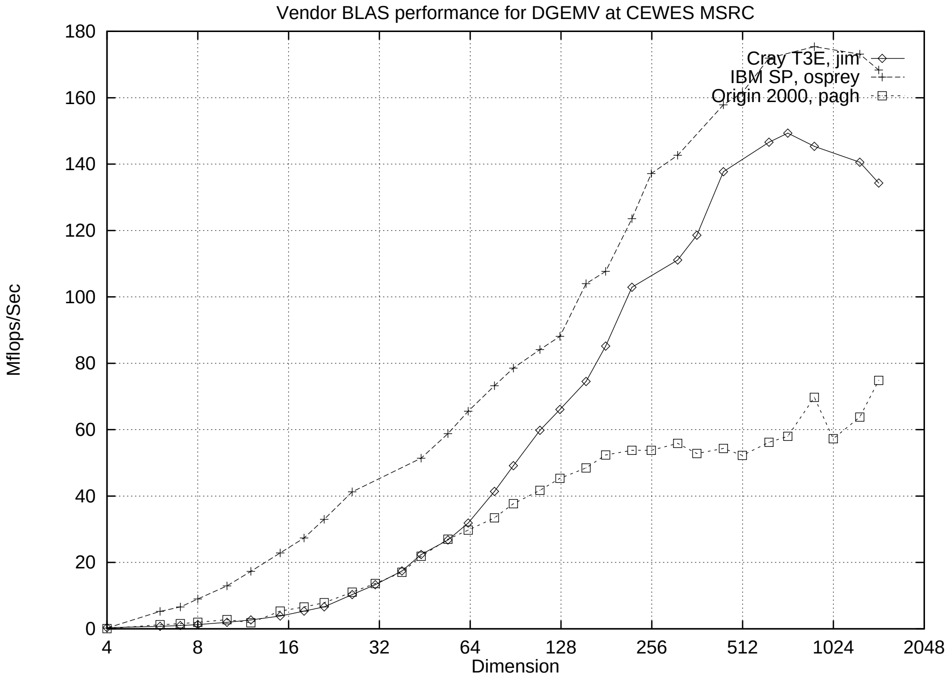

Figure 2: Performance of Matrix-Vector Multiplication with Scale

The DGEMV operation is an operation that stresses both the capacity and the latency of the cache. It provides us with some opportunity of cache re-use provided the implementation is blocked or tiled such that portions of the matrix and vectors remain in cache as long as possible.

Figure 2 shows the performance of this operation on the three machines under study. Note that while performance of the SP and T3E nearly tripled that for DAXPY, the Origin only doubled. We attribute again to its small line size of 64 bytes. The T3E experienced a large performance improvement because of its hardware prefetching. This time the prefetching is highly effective because data is re-used and the ratio of floating point operations to number of memory references is greater.

1.6.3 DGEMM

$$C = \alpha AB + \beta C$$

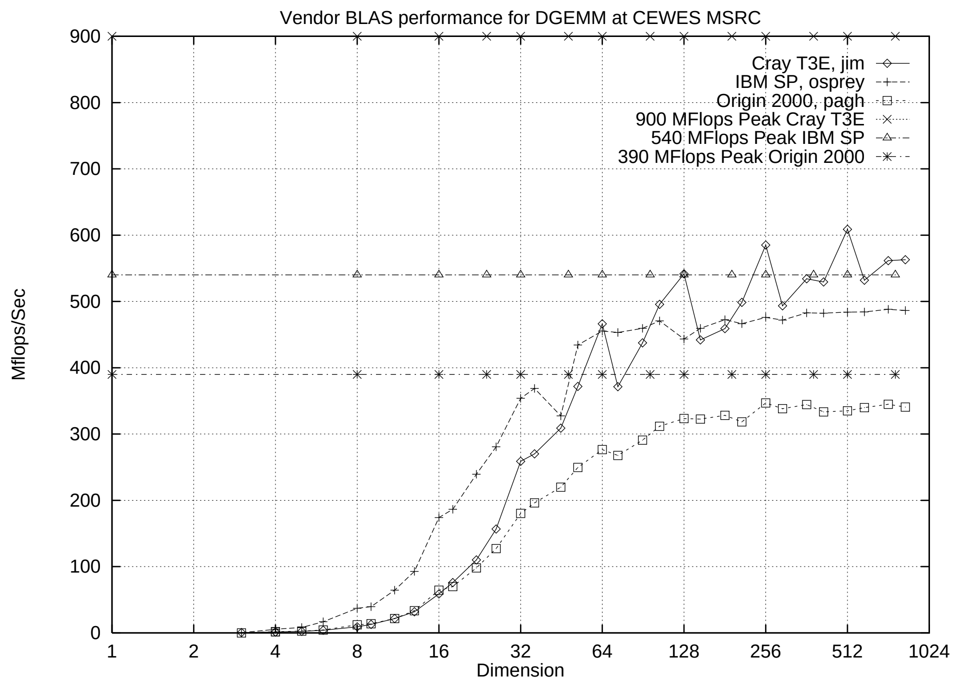

Figure 3: Performance of Matrix-Matrix Multiplication with Scale

Matrix-matrix multiplication performs well because it provides a lot of opportunity for cache re-use. By tiling the matrices, the working set can be reduced to the size of cache and thus only *capacity* misses are taken. For this reason, the performance of GEMM has long since served as a good indicator of a machine's *peak practical performance*. A machine with an adequate memory subsystem like the SP can achieve very high efficiencies, i.e. high percentage of the vendor's published MFLOP rating.

The reader should notice that clock speed and L2 cache size do not play as critical a role in the performance of this routine as one might think. 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 cache/line size of the T3E simply exaggerates the performance loss. The SP, with its large line size and ability to issue two multiply-add instructions as well as a load/store per cycle does quite well, reaching approximately 90 percent efficiency. The Origin appears to suffer from its small cache line size and its inability to issue a load/store every cycle. Like the Power2, the R10000 processor can issue two multiply-adds per cycle, however, its memory system does not appear to be able to supply the operands fast enough for peak performance.

1.7 Future work

- Provide an option for measuring specific problem sizes and ranges.
- Provide an option to specify the problem sizes in dimensionality.
- Provide an option to specify the starting problem size.
- Use specialized, high-resolution timers where available.
- Add additional BLAS routines `TRSM`, `TRSV`, and `SYR2K`.
- Add parameters to tune the placement and padding of the arrays.
- Standardize configuration with GNU *autoconf*.
- Grab machine configuration and store it with each run.
- Standardize data/graph naming scheme with timestamp.

2 CacheBench

2.1 Introduction

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.

2.1.1 Cache Architecture

Caches are essentially very small, high speed memories designed to speed computation among repeatedly accessed data. They are found on virtually all commercially available processors from small sixteen bit embedded microprocessors to the large, multi-million transistor RISC chips found in today's workstations and supercomputers. Caches exploit both *spatial* and *temporal* locality. Spatial locality is the concept that data items that are physically located near each other in main memory will likely be accessed together. Temporal locality is the concept that a data item that is frequently accessed will likely be accessed again in the near future.

When the processor wishes to operate on an item from main memory, it issues a load to the cache. If the item is resident in the cache, this is called a cache hit. If not, it is called a cache miss, and the load request is forwarded to main memory, which moves the data from main memory into a cache line. 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's, *Computer Architecture, A Quantitative Approach.* An example in this textbook serves as the basis for this benchmark.

2.1.2 Goals of CacheBench

The goal of this benchmark is to establish peak computation rate given optimal cache reuse and to verify the effectiveness of high levels of compiler optimization on tuned and untuned codes. 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.

2.2 How it works

CacheBench currently incorporates eight different benchmarks. Each one 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. This total is then divided by the total time to compute a bandwidth figure. This figure is in megabytes per second. Here we define a Megabyte as being 1024^2 or 1048576 bytes. In addition to this figure, the average access time in nanoseconds per each data item is computed and reported. The tests are as follows.

- Cache Read
- Cache Write
- Cache Read/Modify/Write
- Hand tuned Cache Read
- Hand tuned Cache Write
- Hand tuned Cache Read/Modify/Write
- memset() from the C library
- memcpy() from the C library

The first six of these tests access their data through arrays of a predefined base type. This type is set at compile time and defaults to `double`. The rationale for this is that some systems perform memory access differently depending on the functional unit that generated the miss. The default data-type can be altered by setting the `USE_<type>` compiler definition in the `Makefile`. Currently `USE_CHAR`, `USE_INT`, `USE_FLOAT` and `USE_DOUBLE` are supported.

The first three of the tests are intended to provide us with information about how good the 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.

All of these benchmarks runs for a fixed amount of time, which is tunable at run-time. The rationale for this is the widely varying performance of processors these days. CacheBench intends to provide the user with relatively quick feedback about the memory performance of the machine in use. However, this timing restriction limits the accuracy with which we can report the results. A faster machine that runs the test for a higher number of iterations has less relative error. This makes accurate, statistical analysis difficult but it will be fixed in the next release.

2.2.1 Cache Read

This benchmark is designed to provide us with read bandwidth for varying vector lengths in a compiler optimized loop. For the cases where the vector length is less than the cache size, the data will come completely from cache and the resulting bandwidth will be much higher.

The pseudo code for this test is as follows:

```
for all vector length
  timer start
  for iteration count
    for I = 0 to vector length
      register += memory[I]
  timer stop
```

2.2.2 Cache Write

This benchmark is designed to provide us with write bandwidth for varying vector lengths in a compiler optimized loop. This benchmark is greatly affected by architectural peculiarities in the memory subsystem. Replacement policy, associativity, blocking and write buffering all play important factors in the performance of this benchmark. For example, a *write-back* cache will show a much higher bandwidth because it frequently avoids unnecessary references to main memory. In addition, many systems coalesce and buffer multiple writes to cache/memory. This can hide much of the latency of the underlying hardware.

```
for all vector length
  timer start
  for iteration count
    for I = 0 to vector length
      memory[I] = register++
  timer stop
```

2.2.3 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 compilers' ability to better schedule operations and group memory accesses to amortize the cost of the store.

```
for all vector length
  timer start
  for iteration count
    for I = 0 to vector length
      memory[I]++
  timer stop
```

2.2.4 Hand Tuned Versions

A full description of the hand tuned versions of these codes is beyond the needs of this paper. However, to provide some background, the following optimizations were applied:

- Degree eight unrolling. Each loop now references eight memory elements instead of one.
- Dependency analysis. Each operation is independent of the previous seven.
- Register re-use. Registers are allocated to memory locations and reused whenever possible.

The optimizations reflect what a *minimally good* compiler should be doing on these simple loops. In CacheBench, if we see our compiler loops not reaching the performance of our tuned loops, we can conclude that our compiler is poor. The complexity of these loops is minimal and any compiler should be able to optimize them. It is possible, that our compiler optimized loops will outperform our hand-tuned loops, if the compiler inserts prefetching and coalesces memory operations into block transfers.

2.2.5 Memory Set

The C library provides us with the function `memset()` to initialize regions of memory. This function is often highly optimized as it is widely used both in and outside of the operating system. Often, this function is either assembly code placed *inline* in the executable from a header file, or it is an *intrinsic* function that the compiler recognizes and replaces automatically. Some systems have additional hardware on chip to perform this operation, specifically when the value to be set to is zero. This benchmark allows us to compare the numbers from our two formulations of memory write with this version. More often than not, we find that *both versions outperform a call to this routine.*

```
for all vector length
  timer start
  for iteration count
    for I = 0 to vector length
      memset(vector1,0xf0,length)
  timer stop
```

2.2.6 Memory Copy

The C library also provides us with the function `memcpy()` to copy regions of memory. It is also usually an intrinsic or inline assembler function. This benchmark allows us to compare the numbers from our two versions of memory read/modify/write with this version. Frequently we find that `memcpy()` is not as fast as it should be. While this function may not appear explicitly in Fortran application codes, it is used by many of the supporting libraries, like MPI.

```
for all vector lengths
  timer start
  for iteration count
    for I = 0 to vector length
      memcpy(dest,src,vector length)
  timer stop
```

2.3 Using CacheBench

2.3.1 Obtain the distribution

Download the latest release from either of the following URLs:

`http://www.cs.utk.edu/~mucci/cachebench`
`ftp://cs.utk.edu/pub/mucci/cachebench.tar.gz`

First, we must unpack the installation using `gzip` and `tar`.

```
kiwi> gzip -dc cachebench.tar.gz | tar xvf -
kiwi> cd cachebench
kiwi>  ls
```

```
CVS/              Version           cachegraph.gp     index.html
Makefile          cachebench.c      conf/             make.def
README            cachebench.html   doc/              samples/
```

2.3.2 Build the distribution

First we must configure the build for our operating system. Running `make` with no arguments lists the possible targets.

```
kiwi> make
Please use one of the following targets:

        sunos sunos4
        solaris sunos5
        sunmp
        alpha
        linux
        hppa
        sgi-r4k
        sgi-r5k
        sgi-r8k
        sgi-r10k
        sgi-o2k o2k
        sgi-pca pca
        t3e
        t3d
        ibm-pow2 ibm-sp2 sp2 pow2
        ibm-pow pow
```

Configure the build. Here, we are on a Solaris workstation.

```
kiwi> make solaris
ln -s conf/make.solaris make.def
```

Examine the `make.def` file to ensure that the proper compiler flags are being used. Full optimization should be enabled by default. Some machines have model specific flags that can significantly affect the performance of this benchmark. Some of the `make.def` files have these options commented out. The user should examine his system and be sure that the appropriate options are enabled.

```
kiwi> make cachebench
cc -fast -dalign -DREGISTER -DUSE_DOUBLE -o cachebench cachebench.c
```

2.3.3 Running CacheBench

While CacheBench can be run from the command line, it is designed executed through use of the `Makefile`. The resulting datafiles for each of the runs will be left in the file: `tmp/<test>-<HOSTNAME>-<DATATYPE>.dat`.

Immediately after running, the `Makefile` will attempt to graph the results. If `GNUPlot` is not available on this system, simply copy `cacheperf-<HOSTNAME>-<DATATYPE>.tar` to another machine that has `GNUPlot`, extract the tar file and process each `GNUPlot` script file with `gnuplot < <HOSTNAME>.gp > <file>.ps`.

```
kiwi> make run
Measuring Read...
Measuring Write...
Measuring RMW...
Measuring Tuned Read...
Measuring Tuned Write...
Measuring Tuned RMW...
Measuring memcpy()...
Measuring memset()...
.
[commands deleted for brevity].
.
```

2.3.4 Arguments to CacheBench

```
Usage: cachebench -rwbtsp [-x #] [-m #] [-d #] [-e #]
         -r Read benchmark
         -w Write benchmark
         -b Read/Modify/Write benchmark
         -t Use hand tuned versions of the above
         -s memset() benchmark
         -p memcpy() benchmark
         -x Number of measurements to take between powers of 2
         -m Specify the log base 2 of the available physical memory
         -d Number of seconds per iteration
         -e Number of times to repeat test for each vector size

Datatype used is double, 8 bytes
Defaults if  tty: -rwbsp -x1 -m24 -d5 -e2
Defaults if file: -b   -x1 -m24 -d5 -e1
```

Note the fact that the defaults are different depending on whether or not the output is directed to a TTY or a file. Again, the best way to run cachebench is with the `Makefile`.

2.4 Results on the CEWES MSRC Machines

The following graphs are taken from our runs on each of the CEWES MSRC machines during dedicated time. Those machines are the SGI Origin 2000, the IBM SP and the Cray T3E. The cache size and theoretical peak MFLOPS for each machine are listed as follows. The peak MFLOPS is as reported by the vendor and is simply computed as a product of the clock speed times the number of independent FMA's that can be computed per cycle.

Machine	*Cache*	*Peak*
SGI Origin 2000	32K,4MB	390
IBM SP	128K	240
Cray T3E	8K,96K	900

2.4.1 Cache Reads

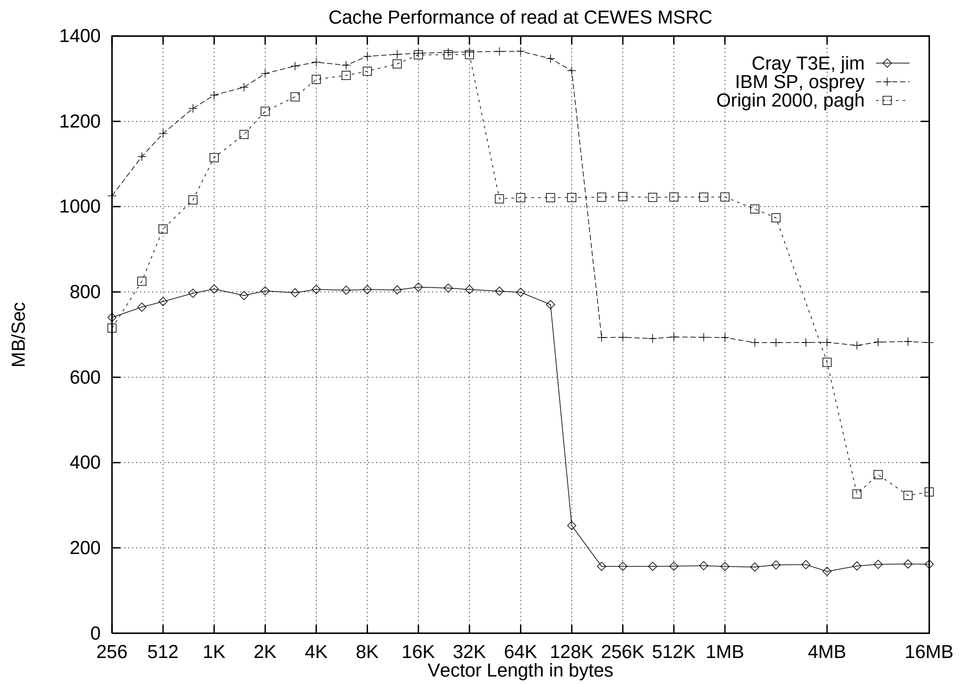

Figure 4: Performance of Compiler Optimized Memory Read

In figures 4 and 5, we notice that the read performance of the Cray T3E is much lower for the hand-tuned version. For the compiler optimized version, we find a two to threefold improvement for vector sizes that lie in cache. The Cray compiler seems to have a very difficult time recognizing what optimized code is doing. This means that tuned applications ported to the Cray *might not perform very well.* For the SP2 and the Origin 2000, the only difference we find is the steepness of the portion of the curve lying substantially below the cache size. Here, we are seeing the overhead of the compiler's code that handles the special cases where the vector length is not a multiple of the degree of unrolling. In the tuned version, this *residual* code does not exist and thus there are no branches in the underlying assembly language. The SP has a hardware loop capability allowing zero cycle branches. For the hand-tuned version, there is no residual code, so the compiler simply sets up the hardware loop and lets it run with no overhead. Thus, we see no performance falloff at smaller vector lengths.

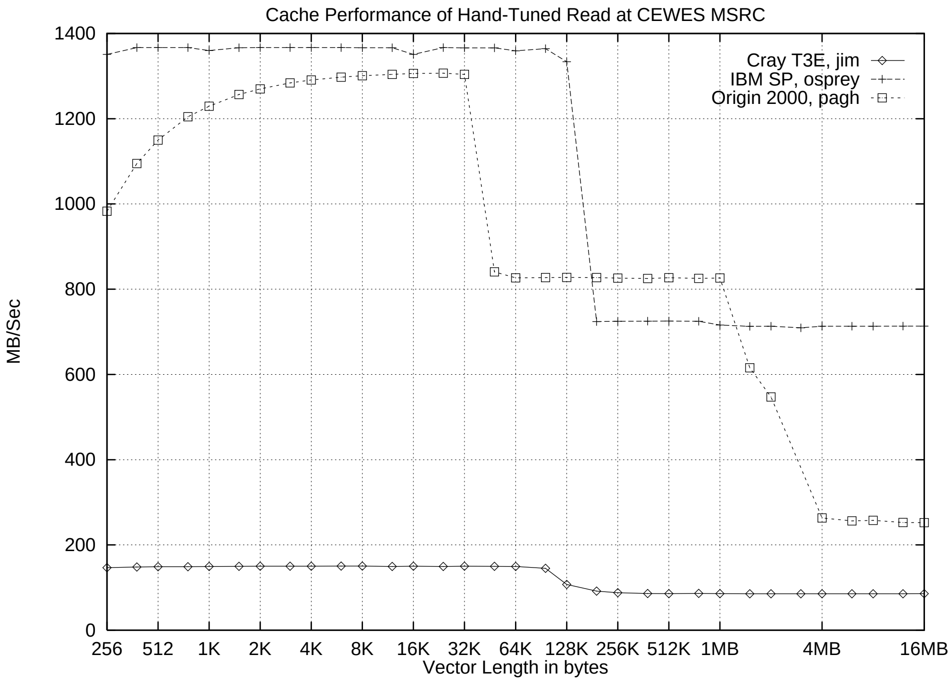

Figure 5: Performance of Hand-tuned Memory Read

2.4.2 Cache Writes

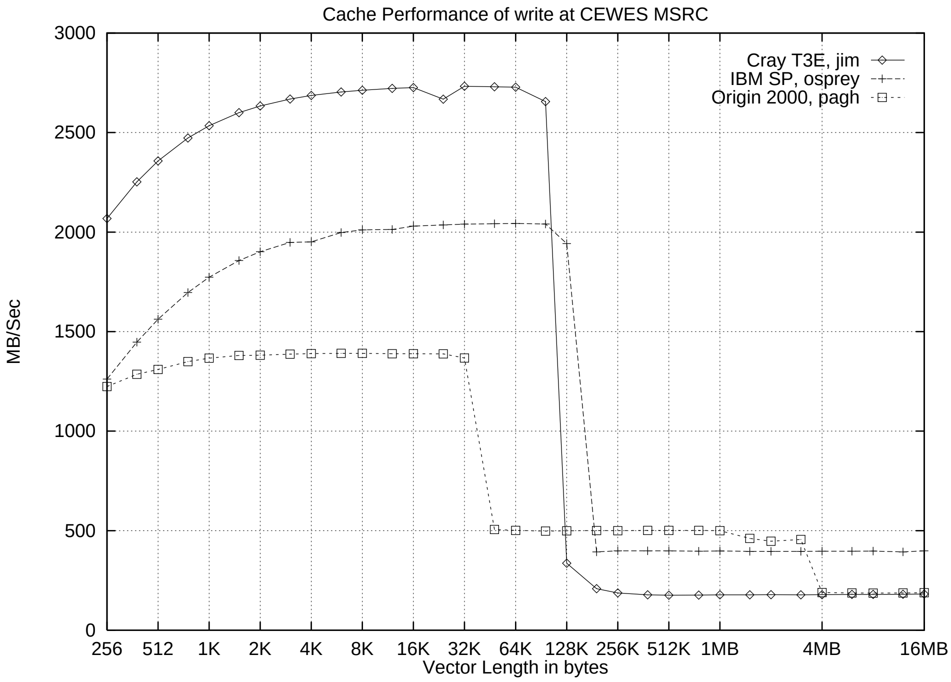

Figure 6: Performance of Compiler Optimized Memory Write

In figures 6 and 7, we can see that the performance of the compiler optimized loop is equal to or greater than that of the hand tuned loop as is the case for reads. The reader will notice that for vectors residing completely in L1 cache, the write bandwidth is equal to or greater than the read bandwidth. On the Origin, the L2 cache is significantly slower to write to than to read from. We infer that the compiler is probably prefetching on the read case and that there is inadequate pipelining between L2 cache and memory. For the T3E, we again notice how poorly the compiler does on the optimized code.

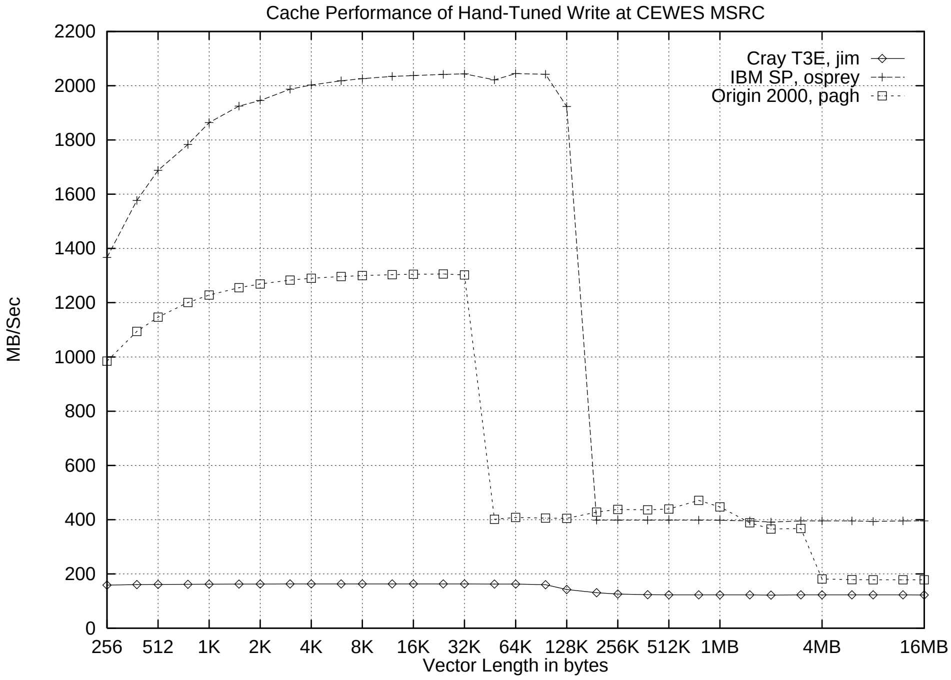

Figure 7: Performance of Hand-tuned Memory Write

2.4.3 Cache Read/Modify/Write

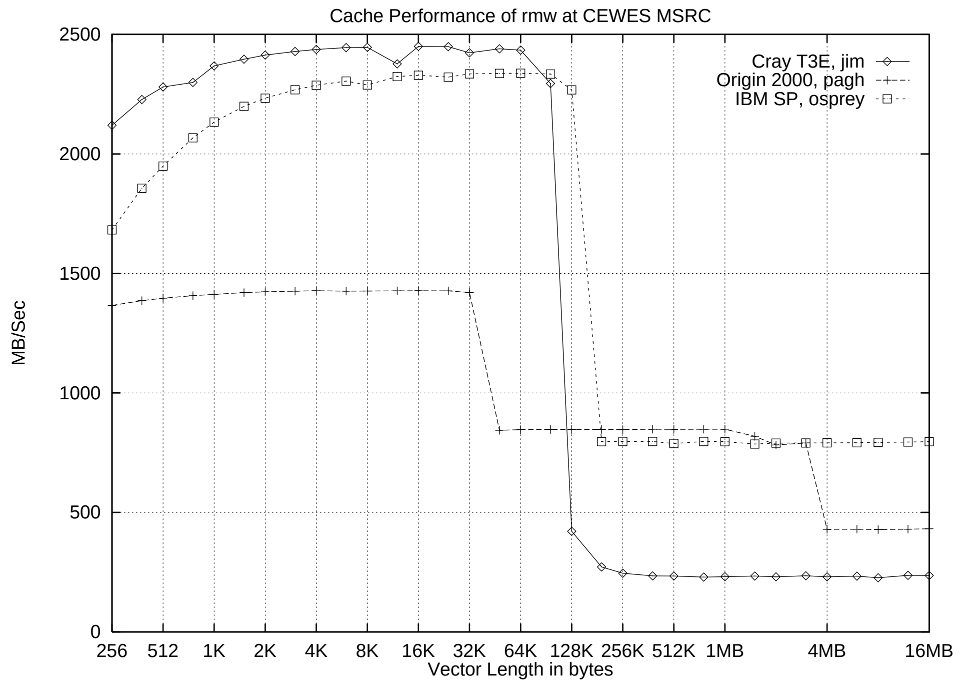

Figure 8: Performance of Compiler Optimized Memory Read/Modify/Write

Of interest in figure 8 and 9 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 probably 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. We are seeing this behavior in the case for reads and writes as well.

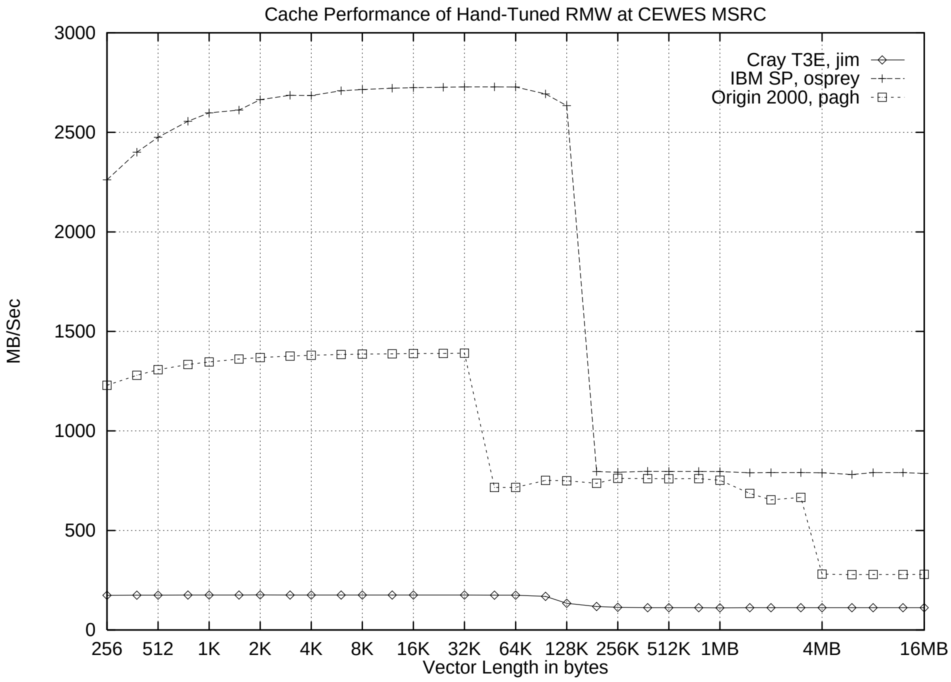

Figure 9: Performance of Hand-tuned Memory Read/Modify/Write

2.4.4 `memset()`

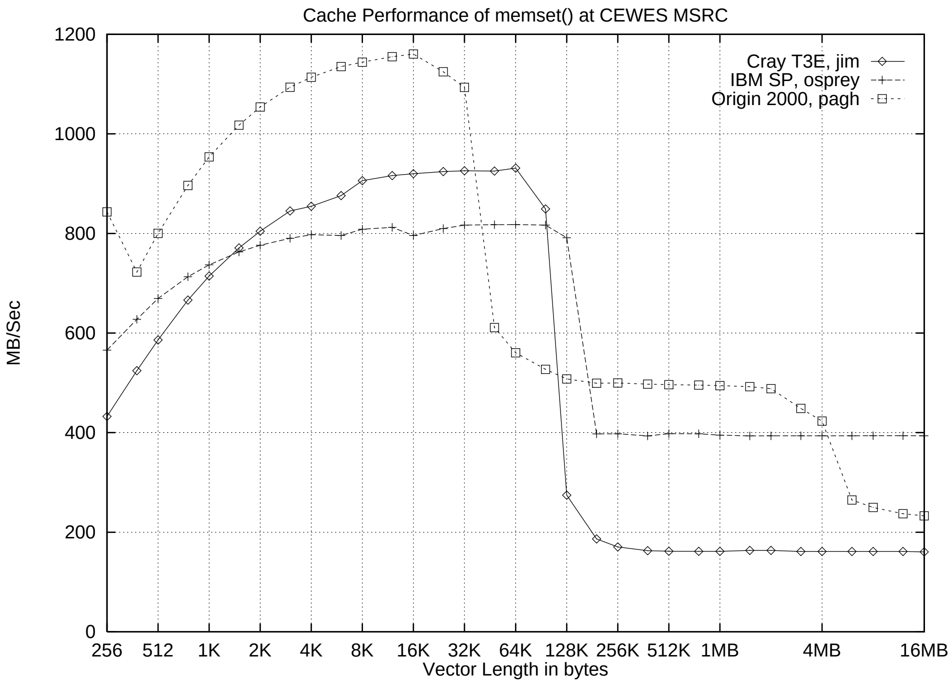

Figure 10: Performance of `memset()`

2.4.5 `memcpy()`

Figures 10 and 11 are provided as reference. The performance of these two routines, when compared with the write and read-modify-write benchmark, clearly indicates that the user would be better off using a *typed* version coded in C or Fortran rather than these library calls. The reason for this is that they are often coded at the byte level for maximum flexibility, not performance. By knowing the type and the alignment of the data ahead of time, the user could easily write a simple loop, let the compiler optimize it and still see much better performance. The only exception is the case where the vector is smaller than L2 cache on the T3E.

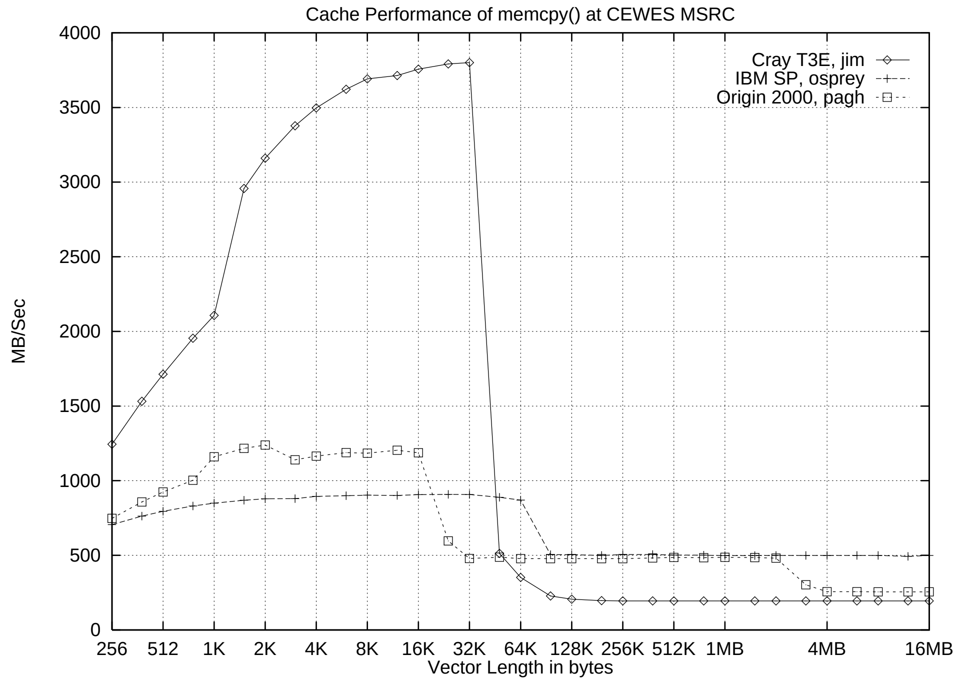

Figure 11: Performance of `memcpy()`

2.5 Future work

- Provide option for measuring specific vector lengths.
- Use specialized, high-resolution timers where available.
- Add benchmark for pointer traversal to measure latency of cache hit and miss.
- Add parameters to tune the placement and padding of the vectors.
- Change from constant run-time to constant iterations.
- Add unoptimized, untuned case for a baseline.
- Standardize configuration with GNU *autoconf*.
- Grab machine configuration and store it with each run.
- Standardize data/graph naming scheme with timestamp.

3 MPBench

3.1 Introduction

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. It generates two types of reports, consisting of the raw data files and Postscript graphs. No interpretation or analysis of the data is performed, it is left entirely up to the user.

3.2 How it works

MPBench currently tests seven different MPI and PVM calls. MPI provides much richer functionality that PVM does, so some of the benchmarks have been implemented in terms of lower level PVM functions. The following functions are measured.

Benchmark	*Units*	*Num Procs*
Bandwidth	Megabytes/sec	2,3
Roundtrip	Transactions/sec	2,3
Application Latency	Microseconds	2,3
Broadcast	Megabytes/sec	16
Reduce	Megabytes/sec	16
AllReduce	Megabytes/sec	16

AllReduce is a reduction operation in which all tasks receive the result. This function is not available in PVM, so it is emulated by performing a *reduce* followed by a *broadcast*.

All tests are timed in the following manner.

1. Set up the test.
2. Start the timer.
3. Loop of operations over the message size as a power of two and the iteration count.
4. Verify that those operations have completed.
5. Stop the timer.
6. Compute the appropriate metric.

In MPBench, we avoid calling the timer around every operation, because this often results in the faulty reporting of data. Some of these operations on MPP's take so little time, that the accuracy and latency of accessing the system's clock would significantly affect the reported data. Thus it is only appropriate that we perform our timing operations *outside the loop*. Some MPP's and workstations have the capability to access the system's timer registers, but this is not portable and would introduce unnecessary complexity into the code to compensate for situations where the timing routines were not efficient.

For simplicity purposes, we will refer to two different types of tasks in MPBench, the master of which there is only one, and the slaves of which their may be any number. The point-to-point tests only use two tasks, a master and a slave. The other tests run with any number of slaves, the default being sixteen.

MPBench averages performance over a number of iterations. The user should be aware that MPBench will use a lower number of iterations than the one specified for certain situations. This should not effect the accuracy of the results, as the iteration count is only changed when the message lengths are prohibitively large.

By default, MPBench measures messages from 4 bytes to 16 Megabytes, in powers of two for 500 iterations. We iterate to make sure that the cache is "warmed" with the message data. This is done because applications typically communicate data soon after computing on it.

The previous version of MPBench did not pay attention to the placement of slave tasks. This caused the user to make false claims about the performance of NUMA multiprocessors like the Origin 2000 where a two task job will be scheduled on processors on the same physical board. MPBench now measures point-to-point performance on MPI jobs with 2 and 3 tasks. For the 3 task case, task 1 (of 0, 1 and 2) remains idle until the end of the run. In fact, this arrangement does not guarantee that the remote processor is offboard, as that is dictated by the MPI environment. However, our measurements seem to indicate that task 2 in a three task job always performs slightly than task 1 of a two task job. We attribute this to extra time required to arbitrate and negotiate the link.

3.2.1 Notes on PVM

PVM has a number of options to accelerate performance. For this benchmark, we are interested in the optimal performance of the machine. In order to do so, we use the following options where possible:

- *Direct Routing* - This option sets up direct TCP/IP connections between processes as opposed to forward messages via daemon processes running on each node. It is established by calling `pvm_setopt(PvmRoute, PvmRouteDirect)` .

- *In-place Packing* - All messages in PVM must be *packed* into PVM buffers before transmission. This option tells PVM not to perform any additional copies of the data, but to transmit the data directly out of the application's buffer. This option precludes the use of any data-translation with PVM. By default, in-place packing is used whenever `pvm_psend()` is called, which is what is used in this program.

3.2.2 Notes on MPI

There are many different send and receive calls in MPI each with different semantics for usage and completion. Here we focus on the *default* mode of sending. This means we are not using any *nonblocking* or *immediate* communication calls. Each MPI implementation handles the default mode a bit differently, but the algorithm is usually a derivative of the following.

```
send first chunk of message
if message is larger than size N
  wait for reply and destination address Y
    send rest of message directly to address Y
else
  if more to send
    send rest of message
endif
```

MPI does this to avoid unnecessary copies of the data, which usually dominates the cost of any communication layer. The receiving process will buffer a limited amount of data before informing the sender of the destination address in the application. This way, a large message is received directly into the application's data structure rather than being held in temporary storage like with PVM. The problem with this is that for large messages, `sends` cannot complete before their corresponding `receives`. This introduces possibly synchronization and portability problems.

3.2.3 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 four 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.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  recv(4)
  stop timer
```

The slave's pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  send(4)
  stop timer
```

3.2.4 Roundtrip

Roundtrip times are measured in much the same way as bandwidth, except that, the slave process, after receiving the message, echoes it back to the master. This benchmark is often referred to as *ping-pong*. Here our metric is transactions per second, which is a common metric for database and server applications. No acknowledgment is needed with this test as it is implicit given its semantics.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
    recv(message size)
  stop timer
```

The slave's pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
```

```
    send(message size)
  stop timer
```

3.2.5 Application Latency

Application latency is something relatively unique to MPBench. This benchmark can properly be described as one that measures the time for an application to issue a `send` and continue computing. The results for this test vary greatly given how the message passing layer is implemented. For example, PVM will buffer all messages for transmission, regardless of whether or not the remote node is ready to receive the data. MPI on the other hand, will not buffer messages over a certain size, and thus will block until the remote process has executed some form of a `receive`. This benchmark is the same as bandwidth except that we do not acknowledge the data and we report our results in units of time. This benchmark very much represents the time our application will be waiting to do useful work while communicating.

The master's pseudo code for this test is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    send(message size)
  stop timer
```

The slave's pseudo code is as follows:

```
do over all message sizes
  start timer
  do over iteration count
    recv(message size)
  stop timer
```

3.2.6 Broadcast and Reduce

The two functions are very heavily used in many parallel applications. Essentially these operations are mirror images of one another, the difference being that reduce reverses the direction of communication and performs some computation with the data during intermediate steps. Both of these benchmarks return the number of megabytes per second computed from the iteration count and the length argument given to function call.

With PVM, broadcast and reduce will not complete unless the application has performed a barrier immediately prior to the operation. Thus, with PVM both of these tests include the cost of a barrier operation every iteration.

Here is the pseudo code for both the master and the slave:

```
do over all message sizes
  start timer
  do over iteration count
    reduce or broadcast(message size)
  stop timer
```

3.2.7 AllReduce

AllReduce is a derivative of an all-to-all communication, where every process has data for every other. While this operation could easily be implemented with a reduce followed by a broadcast, that would be highly inefficient for large message sizes. The PVM version of this test does this exactly, plus an additional barrier call. The goal of including this benchmark is to spot poor implementations so that the application engineer might be able to restructure his communication.

Here is the pseudo code for both the master and the slave:

```
do over all message sizes
  start timer
  do over iteration count
    allreduce(message size)
  stop timer
```

3.3 Using MPBench

3.3.1 Obtain the Distribution

Download the latest release from either of the following URLs:

`http://www.cs.utk.edu/~mucci/mpbench`
`ftp://cs.utk.edu/pub/mucci/mpbench.tar.gz`

Now unpack the installation using `gzip` and `tar`.

```
pebbles> gzip -dc mpbench.tar.gz | tar xvf -
pebbles> cd mpbench
pebbles>  ls
CVS/            README          index.html*     make.def        mpbench.c
Makefile        conf/           lib/            make_graphs.sh* samples/
```

3.3.2 Build the distribution

First we must configure the build for our machine, OS and release of MPI. All configurations support PVM. Before configuration `make` with no arguments lists the possible targets.

```
pebbles> make
Please configure using one of the following targets:

sp2
t3e
pca-r8k
o2k
sgi32-lam
sgi64-lam
linux-lam
linux-mpich
solaris-mpich
sun4-mpich
alpha
```

Configure the build. Here, we are on a SunOS workstation, using MPICH as our MPI implementation.

```
pebbles> make sun4-mpich
rm -f make.def
ln -s conf/make.def.sun4-mpich make.def
```

Now look at the available targets, and build one.

```
pebbles:mpbench> make
Please use one of the following targets:

mpi,pvm,all
run-mpi,run-pvm,run
graph-mpi,graph-pvm,graphs

pebbles> make all
gcc   -O2 -DINLINE -I/src/icl/MPI/mpich/include -DMPI -c mpbench.c -o mpibench.o
gcc   -O2 -DINLINE ./mpibench.o -o mpi_bench -L/src/icl/MPI/mpich/lib/sun4/ch_p4 -lmpi
sed -e "s:MPIRUNCMD:mpirun:g" < lib/mpibench.sh | \
sed -e "s:MPIRUNOPTS::g" | \
```

```
sed -e "s:MPIRUNPROCS:-np:g" | \
sed -e "s:MPIHOSTFILE:-machinefile:g" | \
sed -e "s:MPIRUNPOSTOPTS:mpi_bench:g" > mpi_bench.sh
chmod +x mpi_bench.sh
gcc   -I/shag/homes/mucci/pvm3/include  -O2 -DINLINE -DPVM -c mpbench.c -o pvmbench.o
gcc   -O2 -DINLINE ./pvmbench.o -o /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench \\
\ -L/shag/homes/mucci/pvm3/lib/SUN4 -lpvm3 -lgpvm3
cp lib/pvmbench.sh /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench.sh
chmod +x /shag/homes/mucci/pvm3/bin/SUN4/pvm_bench.sh
```

3.3.3 Running MPBench

While MPBench can be run from the command line, it is designed to be run from via the `Makefile`. When running MPI, sometimes it is required that you set up a hostfile containing the names of the hosts on which to run the processes. If your installation requires a hostfile, MPBench will tell you. If that happens, please check your **mpirun** man page for the format. The resulting datafiles for each of the runs will be left in `mpbench/results/<OS>-<HOSTNAME>_<API>_<test>.dat`.

```
pebbles> make run-mpi
Testing mpirun...
Current value is: mpirun  -machinefile \\
\ /shag/homes/mucci/mpibench-hostfile -np 2 mpi_bench
%Measuring barrier for 500 iterations with 2 tasks...
%Measuring barrier for 500 iterations with 4 tasks...
%Measuring barrier for 500 iterations with 8 tasks...
%Measuring barrier for 500 iterations with 16 tasks...
Measuring latency for 500 iterations...
Measuring roundtrip for 500 iterations...
Measuring bandwidth for 500 iterations...
Measuring broadcast for 500 iterations with 16 tasks...
Measuring reduce for 500 iterations with 16 tasks...
Measuring allreduce for 500 iterations with 16 tasks...
```

Now we plot the results with GNUplot. If GNUplot is not available on your system, perform the following.

- Unpack the distribution on a machine that does.
- Copy your results files to the new machine in the MPBench directory.
- Execute make_graphs.sh with the common prefix of your datafiles.

Normally, we can just do one of the following:

```
make graphs
make graph-mpi
make graph-pvm
```

```
pebbles> make graph-mpi
```

The graphs will be left in the `results` directory.

3.4 Results on the CEWES MSRC Machines

The following graphs are taken from our runs on each of the CEWES MSRC machines during dedicated time. Those machines are the SGI Origin 2000, the IBM SP and the Cray T3E.

Machine	*Cache*
SGI Origin 2000	32K,4MB
IBM SP	128K
Cray T3E	8K,96K

3.4.1 Latency

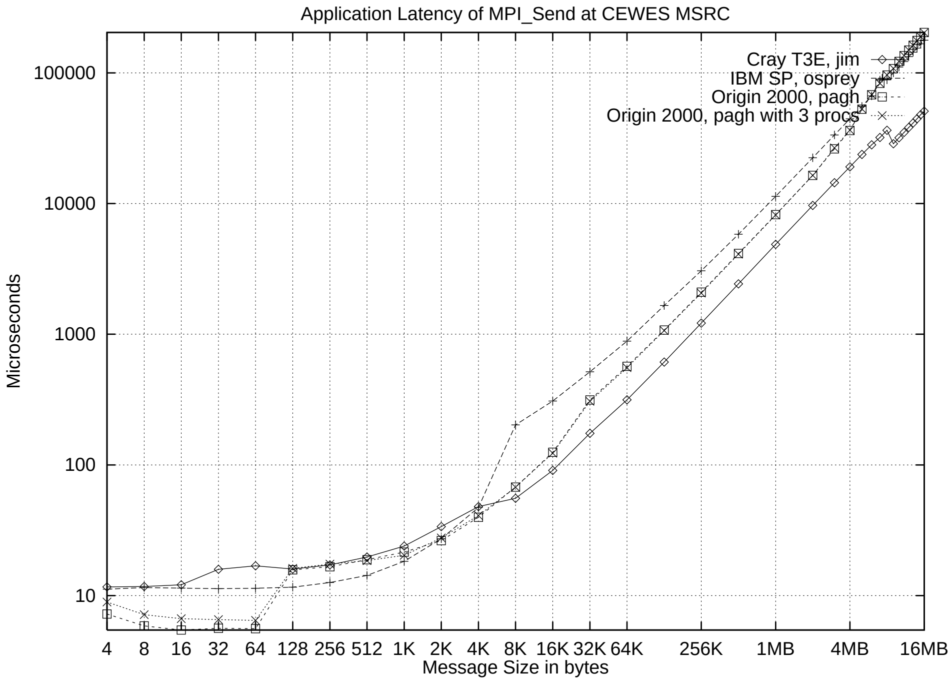

Figure 12: Application Latency of Send

In the figure 12 we see three interesting performance variations. First note the jump in latency of the O2K when the message is greater than 64 bytes. This is likely due to space allocated in the header exchanged between two processes. Many message passing systems allocate space in the header for a small payload so only one exchange is required. Next we note the jump in latency on the SP for messages larger than 4096 bytes. This is the point where IBM's MPI switches to a rendezvous protocol. This is tunable from the command line for `poe` IBM's version of `mpirun` with the `-eagerlimit` argument. It is also tunable with the `MP_EAGERLIMIT` environment variable. We recommend setting this to 16384 bytes for all runs. In fact, IBM does this when running parallel benchmarks. Also, note the falloff in performance at 8MB on the T3E. This is found throughout all our communication graphs and we are currently unable to explain it. On the Origin we see a steady increase in the latency corresponding to the message size. The Origin exhibits extremely low latencies until they exceed a cache size. These latencies suggest that the Origin might be suitable for applications that do a lot of data exchange in small quantities. When the message exceeds the 64 byte cache line on the Origin, it appears that some expensive routine is being called increasing the latency more than twenty-fold. As far as the differences between Origin processors on and off the node, it appears that the only time this is a factor is when messages are smaller than the cache line size. Otherwise, the transmission time appears to be dominated by the memory controller not the communication link.

3.4.2 Roundtrip

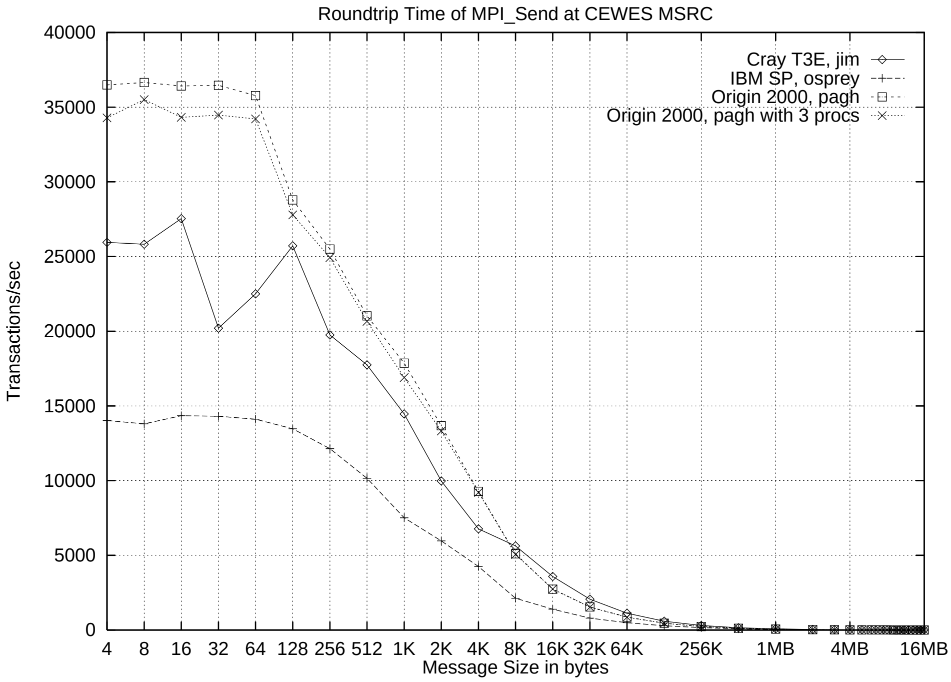

Figure 13: Roundtrip Time of Ping-Pong

Figure 13 is a graph of the average roundtrip time for a message exchange. This test is also commonly referred to as ping-pong. For small messages, roundtrip time is largely dominated by protocol overheads and the method of access to the network hardware. Notice in figure 14 that while the bandwidth for the T3E are higher than for the Origin, the Origin still outperforms it. An inversion takes place at 8K messages between the Origin and the T3E. We deduce that the Origin with its distributed shared memory hardware provides a very lightweight method of accessing remote memory. 8K is the page size of the Origin 2000, so it is not surprising that a penalty is paid after crossing that boundary. For the 3 task test case it appears that only very small message sizes are affected. At larger messages, the raw link speed of the T3E clearly dominates, while the performance of the SP and the Origin falters.

3.4.3 Bandwidth

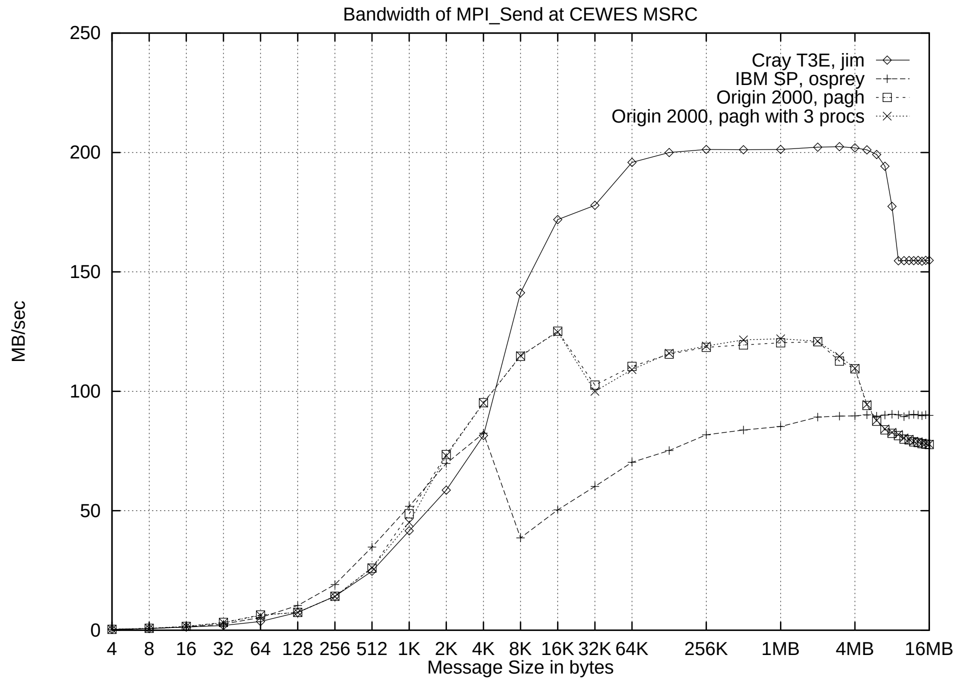

Figure 14: Bandwidth of Send

In figure 14, we note the dramatic effect of MPI's rendezvous protocol on all three machines. The tradeoff is latency for bandwidth, but it doesn't always appear to be valid. As mentioned, the SP has a rather small limit of 4K, thus responsible for the fifty percent falloff at larger message sizes. The Origin and the T3E both have an eager limit set to 16K, with only the Origin suffering a loss in performance. Also of interest is the effect that caching has on the Origin. As mentioned, these tests are repeated a number of times, so most of the data will lie in the Origin's large 4MB level two cache. Note that for larger sizes, its performance suffers severely. We recommend that users raise the default eager limit to 32K for their runs.

3.4.4 Broadcast

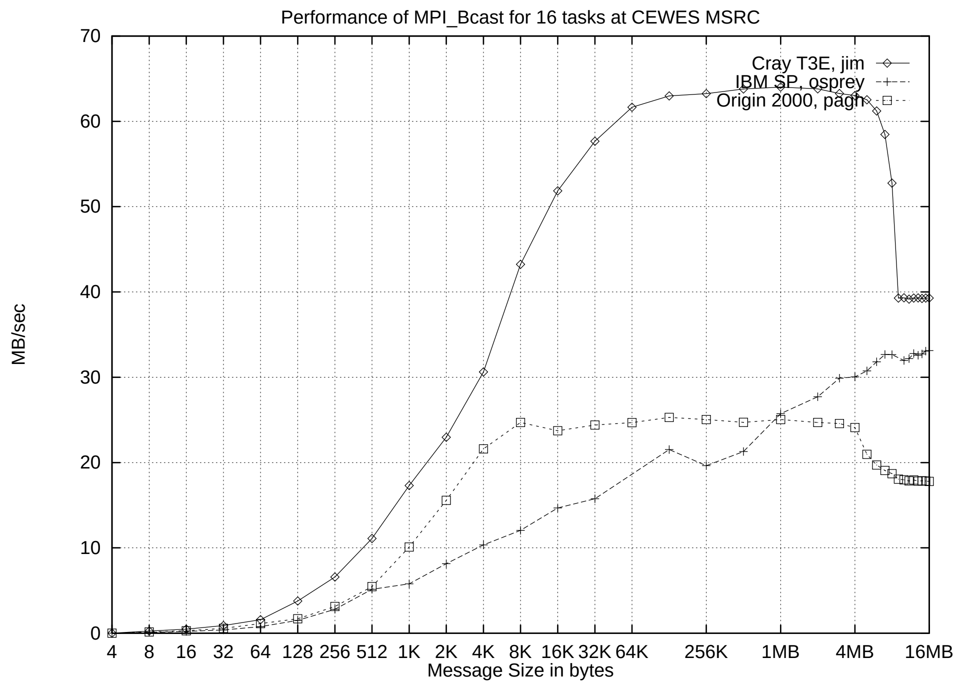

Figure 15: Broadcast Performance

For figure 15, we again note the dramatic drop-off found at the 8MB message size on the T3E. For the Origin, we also notice the effect of cache. The user should be aware that this test also includes the time for every task to send an acknowledge back to the master. In figure 16 we show the time steps for this test on an eight processor system. If we assume a left-to-right ordered binary tree distribution algorithm, we receive our first acknowledgement after log2(8)-1 or 2 full sends. Our last acknowledgement arrives on the rightmost branch after log2(8)+1 or 4 full sends have completed. Note that in this test, the timer does not stop until we receive the last acknowledgment. Relating our broadcast performance to our bandwidth requires understanding that we must execute at least one send to transmit the data. Thus to compare broadcast to bandwidth performance we must take into account the first send. Therefore broadcast performance should be 1/log2(p) that of bandwidth. This only holds true for machines that have no hardware assisted broadcast. Our results seem to agree with this model.

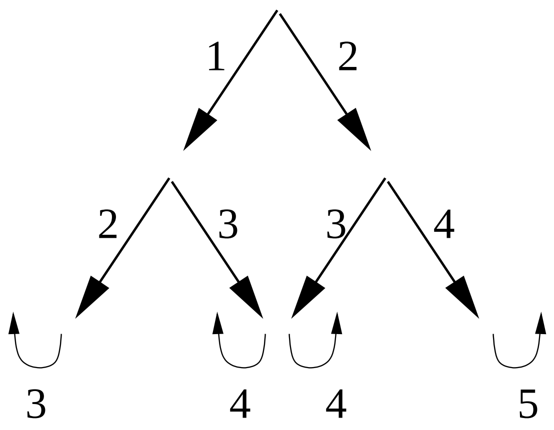

Figure 16: Message time steps tree for 8 nodes

3.4.5 Reduce

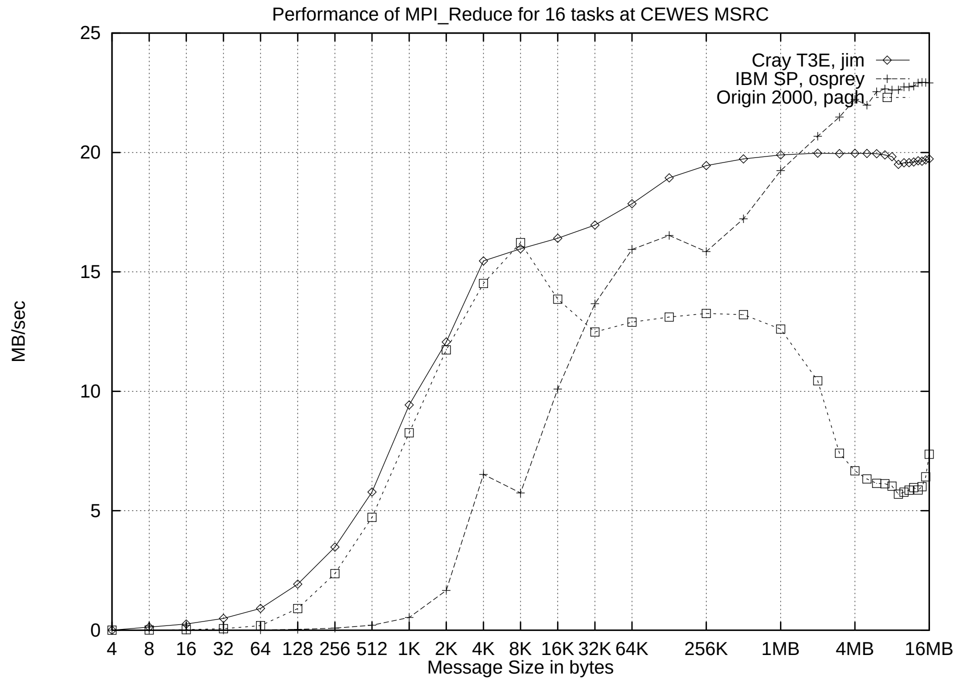

Figure 17: Reduce Performance

In figure 17 we plot the performance of a reduction operation. as due to a change in the MPI protocol, we would expect a graduate increase in performance after the initial drop. However, because this is an iterated test, the cache may be hiding the effects of the MPI protocol and exaggerating the cost of a distributed page fault. The large hump is where the message fits into the level two cache. For the SP, the dip at the 4K message size is again related to the small eager limit. Performance of the T3E increases steadily and levels off around 20MB/sec. Notice the lack of a significant falloff at larger messages on the T3E. Also notice how poorly the T3E performs in relation to figure 15.

3.4.6 AllReduce

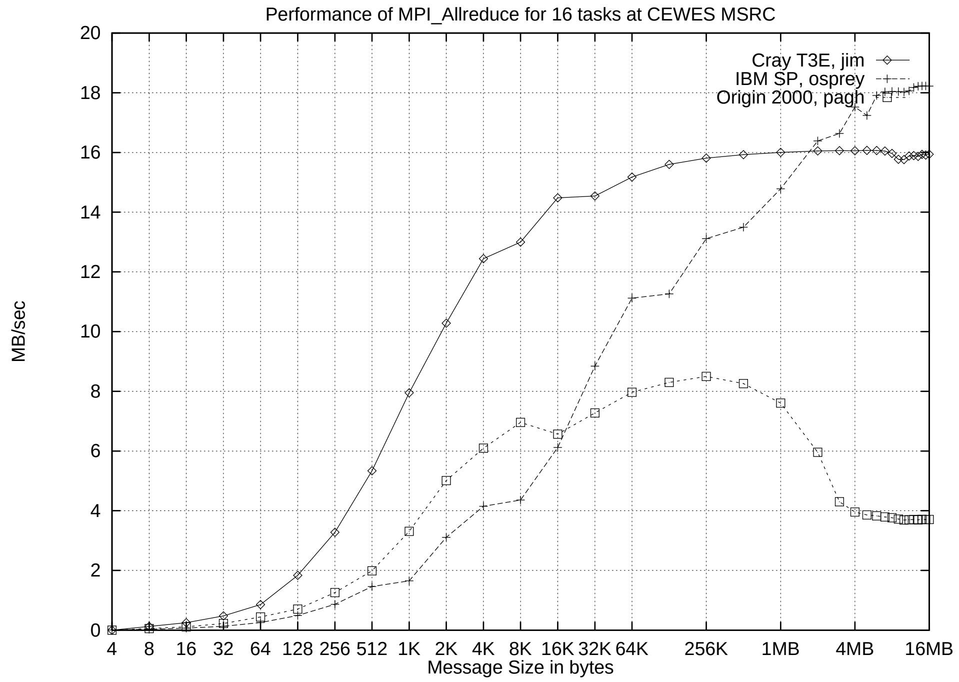

Figure 18: AllReduce Performance

For figure 18, we again notice the dramatic effect caching has on the Origin with performance falling off around the 8MB mark. Comparing this graph with that of figure 17, we note that the SP2 and the T3E perform about twenty percent worse on Allreduce than on Reduce. The Origin performs more than thirty percent worse, which is perhaps an architectural problem related to network contention.

3.5 Future work

- Provide option for adjusting the test space.
- Provde the option for measuring a specific message size.
- Provide an option for cache flushing between transmission.
- Use specialized, high-resolution timers where available.
- Add benchmarks for `MPI_Isend`, `MPI_Irecv`, `MPI_Irsend` and `MPI_Alltoall`.
- Standardize configuration with GNU *autoconf*.
- Grab machine configuration and store it with each run.
- Standardize data/graph naming scheme with timestamp.

4 References

Computer Architecture, A Quantitative Approach by David A. Patterson, John L. Hennessy, David Goldberg, Softcover, 1050 Pages, Published by Morgan Kaufmann Publishing, 01/1996, ISBN: 1558603298

The Science of Computer Benchmarking (Software, Environments, Tools) by Roger W. Hockney, Softcover, 600 Pages, Published by Society for Industrial and Applied, 6/1996, ISBN: 0898713633

PVM - Parallel Virtual Machine Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, MIT Press, 1994

MPI - The Complete Reference Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, Jack Dongarra, MIT Press, 1996

AIX Version 4 Optimization and Tuning Guide for Fortran, C and C++, Published by Internation Business Machines Corporation, 1992, 1996