HPCC Optimizations and Results for the Cray X1

Nathan Wichmann Cray Inc.

May 14, 2004

ABSTRACT: A new benchmark call HPCC has recently been proposed to evaluate High Performance systems. This paper discusses the porting of this new benchmark, what optimizations have been made so far and plans for future optimizations. Results are shown for both Cray and other machines. Finally, the presentation concludes by comparing the CrayX1 to the competition using HPCC.

Introduction:

HPCC is a recently introduced benchmark sponsored by DARPA and the Department of Energy consisting of six tests that are meant to characterizes the performance of HPC architectures using kernels with more challenging memory access patterns than LINPACK with the results posted to a web site. HPCC is a single program written in C which can easily be downloaded and ran on virtually any platform. Anyone submitting results must first submit a "base" or AS IS run and then a run that includes optimizations. It is intended that this benchmark will become part of numerous HPC proposals in the future.

The six main tests are as follows:
HPL – A LINPACK solver
PTRANS - Performs a global transpose of a matrix then adds it to another matrix
STREAM - Pure stride-1 memory bandwidth test
RandomAccess - The DARPA GUPs benchmark. Does a large, random table update. They have both a single cpu version and an MPI version.
MPI latency tests
MPI bandwidth test
Additional details can be found at: **http://icl.cs.utk.edu/hpcc/ .**

HPCC is really more than just the six tests, since some of these tests are run in up to eight different ways, such as with the STREAM case, and the results being reported for these tests also vary. At the moment, some of the statistics gathered by these benchmarks are quoted as single CPU numbers, some are "per" CPU numbers, while others are total system numbers. As a result the interpretation of these results may initially be confusing to some people. This paper will first review each test in greater detail and then present results and comparison with other platforms.

HPL:

The HPL test is basically another version of the LINPACK solver. However, it differs from the LINPACK used for TOP500 numbers in several key ways. This version is already a complete package that is built using MPI as its method of communication while the numbers that Cray submits for the TOP500 list were created using special purpose software which uses SHMEM and contains other optimization. While these optimizations were legal under the TOP500 rules, it does not seem possible with HPL and HPCC. This means that we cannot (easily) match the performance as reported in the TOP500 list. For example, in HPCC runs I have observed per MSP performance in the range of ~9.5 Gflops/MSP while in the TOP500 list we report numbers in excess of 11 Gflops/MSP. There have been no optimizations made to HPL so far except for choosing the best tuning parameters. Nonetheless, HPL scales well because there is little communication relative to the amount of computation.

Results for HPL for selected machines are shown below:

Machine Name- # CPUS	HPL- Tflops
Cray X1- 252	2.36
Cray X1- 124	1.18
Cray T3E- 1024	0.05
HP DEC Alpha- 484	0.62
IBM Power4- 504	0.90
Linux Networx- 256	1.03

PTRANS:

The PTRANS test is a relatively complex set of routines that creates two matrices, A and C, distributed over all the processors in a block-block layout, and then performs the computation: C=C+beta*transpose(A). It is the communication associated with the transpose of A that is suppose to stress the system, although there are other parts of the test that do local copies that take some time as well.

The transpose of A is done by copying blocks of A to a temp array, communicate that temp space using a routine called Cblacs_dSendrecv, and then copys it back into a usable format.

The performance of PTRANS is very chaotic and varies depending on problem size, blocking factor, the number of CPUs, the layout of the grid of CPUs in PTRANS, and perhaps even the order in which the tests are run! Performance can easily vary by 50% or more, and sometimes performance "falls off a cliff" by a factor of 10. To date I have no explanations why this is happening but it seems clear this is NOT a random effect, as the

results are repeatable from one run to the next, nor does it seem to be hardware related. At this time members of our libraries group are examining performance to see what can be done.

Results for PTRANS for selected machines are shown below:

Machine Name- # CPUS	GB/s
Cray X1- 252	96.1
Cray X1- 124	39.4
Cray T3E- 1024	10.3
HP DEC Alpha- 484	3.74
IBM Power4- 504	5.00
Linux Networx- 256	3.11

As you can see, even with chaotic performance the Cray X1 is far superior to anything else on the list. A 252 MSP Cray X1 is between 20 and 30 times faster than the cluster based machines listed here. While observed performance will vary on different clusters and configurations, it will not change the conclusion that the Cray X1 has more than 1 order of magnitude more network bandwidth than most other machines on the market.

STREAM:

There are a total of 8 STREAM runs, four of them are the "normal" single CPU STREAM runs of Copy, Add, Scale, and Triad. For the single CPU runs the program picks a processor at random, but not processor 0, and runs the STREAM benchmark while every other processor is sitting on a wait statement. The other four tests are equally interesting, they are what they call "*" or "Star" runs. For these runs, the processors are synchronized once at a very high level, and then all processors run the STREAM benchmarks "simultaneously". The numbers are still reported as a per CPU number. A comparison between the single CPU number and the * number is a good indication of what happens when you load up the system.

While the single verses Star comparison is very enlightening, an aggregate STREAM number is probably better for compare platforms. While this is basically embarrassingly parallel and uses ZERO interconnect bandwidth, it still will show off systems with high local memory bandwidth. The likes of IBM, XEON clusters, and anyone that packs in the CPUs compared to memory bandwidth will still be at a disadvantage.

Results for STREAM for selected machines are shown below:

Machine Name- # CPUS	Single CPU GB/s	Star CPU GB/s	Aggregate GB/s
Cray X1- 252	24.0	21.7	5478
Cray X1- 124	24.0	21.7	2697
Cray T3E- 1024	0.51	0.51	529
HP DEC Alpha- 484	1.66	1.38	672
IBM Power4- 504	1.99	1.71	864
Linux Networx- 256	1.64	0.77	198

There are a few observations that can be made from this chart. First is that the Cray X1 has more than an order of magnitude more bandwidth per cpu than the competition, a feature which is critical in the performance of many applications. Second is that the Linux Network XEON cluster take a massive hit in per cpu performance when the machine is loaded up, a good indication that observed sustained performance in a production environment will be much lower than implied by something like HPL. Finally, we see that the Aggregate Bandwidth of the Cray X1 is far superior to anything else, where 252 MSPs is more than 6 times as powerful as 504 POWER4s and more than 25 times as powerful a the Linux Cluster. This is in stark contrast to the HPL number which shown only a factor of 2 between the Cray and those machines.

There were two optimizations done to improve performance. The first optimization was to align the arrays to cache line boundaries so that the edges of vector loads and stores would not see bank conflicts as they shared a cache line. The other optimization was to add a no_cache_alloc directive so that the data for A, B and C would not be cached and the memory transfers would happen more efficiently. The result was a 50% improvement in performance.

RandomAccess:

There are three tests in RandomAccess. The first version is a single CPU table update, the second runs the single cpu version simultaneously on all processors while the third version solves a much larger single problem using all processors.

The basic kernel is relatively simple, even if its performance characteristics are not. Basically the kernel uses a random number generator to generate an index into a large Table and much smaller substitution table, or STable. The values in those two tables are loaded and then XOR'd together and the result is store back into Table. The kernel itself is a double nested loop where the inner loop has had it random number sequence initialized for STRIPSIZE different starting points. Performance is measured in GUPs, or "Giga UPdates per second", a unit which is relatively unique to this benchmark.

One unusual characteristic is that the kernel allows one to run the inner loop in parallel, even though there are potential conflicts and one can get some of the answers wrong. As long as the error rate is less than 1% of the total table the run is considered to be valid. It turns out that since the size of the Table is so large this is very easy to achieve even for highly parallel implementations.

The base version performs poorly only because the authors commented out the "pragma ivdep" that was on the inner loop, inhibiting clean vectorization and streaming. Increasing the STRIPSIZE to 1024 and adding a "pragma concurrent" directive on the inner loop easily optimized the kernel. This allows the compiler to vectorize and stream the loop and sustain about 0.2 GUPS/MSP.

For the Global RandomAccess test case the problem being solved remains the same, but the size and algorithm used to solve it differs substantially. The base version uses an algorithm that first sorts the updates into different buckets for different CPUs, sends those buckets around the machine through a huge mpi_alltoall, and then uses the information in the buckets to do a final, local update of Table. Furthermore, since this problem is only defined to work on a power of 2 Tablesize but we want to run on non-power of 2 CPUs, an if test, integer divide, and an integer mod are required to calculate the "Whichpe" and the "LocalOffset", all done in scalar mode on the Cray X1. All together this makes for a very slow implementation indeed.

Two different avenues were taken to optimize Global RandomAccess: First is to optimized the operations but still use the MPI based algorithm, second is the change the code to use Unified Parallel C.

To optimize the MPI version, three basic changes were implemented. First Table was redistributed in a manner which allow the critical if statement to be removed, leaving only loop invariant if's inside the loop, which the compiler could easily optimize away. Second, it was realized that both the numerator and the denominator of the integer divide were less than 52 bits in size, this means that they could both be cast to floating point values and the operation could be done using the much faster floating point divide. Furthermore, since the denominator is a constant, the compiler was able to take this new code and turn it into a multiple by reciprocal, further increasing performance. The third optimization made was to implement multiple buckets for every cpu and vectorizing across these "extra buckets". The optimization is very similar to vectorization in the single cpu version and the performance improvement is substantial. All of the optimized numbers submitted to the HPCC website contain this optimized kernel.

Results for STREAM for selected machines are shown below:

Machine Name- # CPUS	GUPs
Cray X1- 252	1.10
Cray X1- 124	1.52
Cray T3E- 1024	0.25
Cray X1- 60	1.31
Cray X1- 32	1.06
HP DEC Alpha- 484	0.45
IBM Power4- 504	0.18
Linux Networx- 256	0.31

As you can see, while the Cray X1 numbers are better than anyone else, performance in terms of GUPs. Large machine allows one to solve large problems, but GUPs rate does not improve as quickly. Indeed, a 32 MSP system already achieves 1 GUP. The poor scaling is no doubt a result of the algorithm used to solve the problem, that is a bin sort followed by a AlltoAll, followed by the actually updates. This does not overlap communication and computation and requires many synchronizations and as a result is likely suffering from load imbalance problems.

The second avenue of optimization was to replace the MPI algorithm with one based on UPC. While the MPI version works, it has a number of disadvantages. First is that it breaks the code into three separate sections, in a way tripling the amount of work that needs to be done. Second it does a very large alltoall, which is very network intensive and does not overlap with any operations. Third is that the alltoall must be done any time that one of the buckets is full, this makes for many, many synchronizations even though the problem does not call for any. Couple this last point with different CPUs filling up different buckets at different rates and you have the makings of a load balance problem. All of these issues beg for a different solution.

The UPC implementation is very simple. Allocate a shared array Table which is cyclically distributed across all CPUs and then use the single cpu algorithm on each cpu to do its share of the updates. The result is a double nested loop while is vectorizable, overlaps computation and communication, and does not do any synchronization for tens of seconds. The only additional modification was to split the one dimensional Table into two dimensions so that I could do the WhichPE and LocalOffset calculations myself, the same way as described above for the MPI version. The end result is that the UPC version runs much, much faster while being fewer lines of code and easier to understand. A UPC run performed on 252 MSPs showed it sustaining approximately 3.5 GUPs, more than 3 times faster than the MPI version, with the possibility of further optimizations.

LATENCY TEST:

The HPCC latency test consists of two runs. The first is a ”not so simple” version of the "ping-pong" test where the test does a significant number of sends and receives between a number of different processor pairs, reporting the maximum latency found from of all the results. The second run performs a "random ring" test where it passes 8 byte messages between processors in a ring, once using MPI_Sendrecv, and once using non-blocking MPI routines, they take the minimum of the two methods and report the "per CPU" latency, which I take to mean as an average latency.

While it tests the latency of a large number of processor pairs, the end result is their “ping pong” results are very similar to a simple MPI “ping pong” test on the CrayX1. This is probably due to the fact that the X1 has a global address with a low latency network and most of the latency is in the MPI library itself. Their ring test is more complex but our results seem to be just two times the “ping pong” latency. This actually makes sense when you count the number of MPI routines each processor must go through.

The story is very different when looking at some of our competitors. For example, they report a “ping pong” latency of 70 microseconds on the IBM. The ring test is a little more "stable", but they still report a latency of 70 microseconds on the IBM and 30 microseconds on two HP machines. This is MUCH longer than the normally quoted latencies of the low single digits. This might be because the times normally quoted are between processors that are "close" together on an otherwise quiet network. Real life is different. But this is only speculation. Only SGI with the SGI supplied MPI is faster then the CrayX1, again probably because they have a global address space.

Regardless of why our competitors are the speed they are, the CrayX1 actually turns out looking pretty good. Only SGI is faster and we appear substantially faster than IBM.

By far the latency is the most difficult measurement to interpret when trying to compare machines. The range of values on the web site varies by a factor of 50! One problem with the latency test is: How much better is 3 μsec latency vs. 10 μsec vs. 100 μsec? After all, one could argue that low latency never really helps you, instead high latency hurts you, and what is considered high latency is problem dependent. Also latency does not improve as the machine gets bigger, instead, it almost always gets worse, in some way making it look like larger machines are “less powerful” than small machines.

Results for the Latency test for selected machines are shown below:

Machine Name- # CPUS	per CPU μsec	SM Band MB/s
Crav X1- 252	22.6	89.0
Crav X1- 124	20.8	47.6
Crav T3E- 1024	12.1	677
HP DEC Alpha- 484	39.9	97.0
IBM Power4- 504	367	11.0
Linux Networx- 256	22.3	92.0

From this chart you can see that there are 2 outliers. The first outlier is the latency of the IBM machine, at 367 μsec is 10 times worse than anything else listed and 15 times worse than the Cray X1. The second outlier is the Small Message bandwidth of the Cray T3E, at 677 MB/s is it almost 7 times larger than anything listed. This is a function of the low latency network of the T3E even at a relatively high processor count of 1024.

No optimization results for this test have been submitted because none were allowed under the current rules. However, I did optimize the code using UPC with the results presented in a section below. Ideally, I would like to see a test that focus on testing how, and how fast communications can be done, instead of simply testing the speed of some MPI routines. This would allow us to submit the UPC results. Both UPC and Co-Array Fortran extensions are becoming more widely available, and CAF has even been proposed to go into the next Fortran standard. Either of these communication methods is much faster and easier to write.

Network Bandwidth Test:

The network bandwidth test appears to use exactly the same software and algorithms at the Latency test, the only difference being that the bandwidth test uses message sizes of 2,000,000 bytes.

Results for the Natural Ring Bandwidth test for selected machines are shown below:

Machine Name- # CPUS	per CPU GB/s	LM Aggr Band GB/s
Cray X1- 252	2.60	654.3
Cray X1- 124	4.12	510.7
Cray T3E- 1024	0.15	149.2
HP DEC Alpha- 484	0.091	44.1
IBM Power4- 504	0.16	79.3
Linux Networx- 256	0.054	13.7

The results make the CrayX1 look very good. For the “ping pong” test we are approximately an order of magnitude faster than HP and SGI in most of the cases and 30 times faster than the IBM. For the ring test they use both a random and natural arrangement. The natural ring test is normally faster, and this is especially true for our competition. To help compare between machines and processor counts, I computed a Cumulative Natural Ring Bandwidth. Even using this number the CrayX1 looks very

good. Our 252 MSP number is more than 8 times faster than a 504 processors IBM system and 50 times faster than the Linux Network cluster.

UPC versions of the Network Latency and Bandwidth Test:

As mentioned above, the network bandwidth and latency test were implemented using MPI, specifically either MPI_Sendrecv or Isends and Irevcs. While this is a common method of communication it is not the faster form of communication available nor the most productive syntax to write.

I wanted to see what would happen if I replaced those calls to MPI with the equivalent UPC code. First I had to make a modification to the test from communicating bytes to communicating longs. This change not only seemed reasonable since the same amount of data was being transferred, but it seemed more realistic since the vast majority of time people transfer whole words, either longs or doubles, rather than bytes. After that the modifications were very simple, indeed most of the time was spent trying to understand what the sendrecv was doing rather than coding up the upc code. To demonstrate the difference the MPI and UPC codes segments are copied below.

MPI version of the ring test:

```
MPI_Sendrecv(  sndbuf_right, msglenw, MPI_LONG, right_rank, TO_RIGHT,
               rcvbuf_left, msglenw, MPI_LONG, left_rank, TO_RIGHT,
               MPI_COMM_WORLD, &(statuses[0]) );
MPI_Sendrecv( sndbuf_left, msglenw, MPI_LONG, left_rank, TO_LEFT,
              rcvbuf_right, msglenw, MPI_LONG, right_rank, TO_LEFT,
              MPI_COMM_WORLD, &(statuses[1]) );
```

UPC version of ring test:

```
upc_barrier;
for(i = 0; i < msglenw; i++ ){
     upc_recvbuf_left[i][right_rank] = sndbuf_right[i];
     upc_recvbuf_right[i][left_rank] = sndbuf_left[i];
     }
upc_barrier;
```

The results were rather dramatic. Below is a comparison of results on 252 MSPs:

MPI version:
RandomlyOrderedRingLatency_usec = 21
NaturallyOrderedRingBandwidth_Gbytes = 2.6
RandomlyOrderedRingBandwidth_Gbytes = 0.44

UPC version:
RandomlyOrderedRingLatency_usec = 8.67
NaturallyOrderedRingBandwidth_Gbytes = 6.06474
RandomlyOrderedRingBandwidth_Gbytes = 0.837

Basically all the performance for all of the test improved by a factor of 2-2.5X, with the latency being the best of any machine that uses more than 32 CPUs. Furthermore we expect the performance could be even better. Right now the algorithm does a upc_barrier to synchronize with all CPUs, a much larger hammer than what is required. Instead, if we had a mechanism that would synchronize only with its neighbors, effective latency could go down further.

Methods to Compare Machines using HPCC:

While trying to answer the question "How well do we do on HPCC compared to the competition?" I realized that the volume of numbers being produced was overwhelming. The web site has a total of 25 entries, each with 6 columns of data. To really try to understand how one machine was performing versus another is was necessary to combine all of these numbers into a single score.

Several steps were taken to compare machines. First, the list provided on the web site was exported to a spreadsheet and "similar" entries were removed. That means that if there was ever a particular architecture with ~N CPUs additional copies were removed. If there was ever a base and an optimized version in the list, the optimized version was use. A Cray X1 using 32 was also added to the list for comparison purposes. This resulted in a list of 17 machines total. It is important to note that this list includes only those machines listed on the HPCC web site. Currently no results have been submitted for any machines in the top 10 positions of the TOP 500 list.

The next step was to normalize the results from each category to create a unitless number. This was necessary because each category had different units with drastically different orders of magnitude, the only way to combine them would be to turn them into normalized unitless numbers. Several normalization methods were evaluated include dividing by the most powerful result in that column or dividing by the range of values. The first method was reject because it made the scoring too sensitive to the speed of the largest machine while the second took what was potential small insignificant difference and made them large. The final method chosen was to divide by the combined power of all the machines in the column. This is in fact equal to the percentage of total power contained in machine M. This is not sensitive to any one machine as the combined "inertia" of the list is large for large list and is does not turn small difference into large ones. An interesting side effect was that this normalization already made the results easier to interpret. Since all of the numbers were unitless and of the same order of magnitude, one could easily tell when a machine did particularly well or particularly poorly on a test.

The next step was to combine the results into a single score. For this I decided to examine 3 different weightings. The first was to just use 100% HPL, very similar to the TOP 500 list. The second was to use 50% HPL – 50% for the other 5 test, 10% per test. This was a halfway point to the third weighting. The final weighting was to simply

weight every test equally. The philosophy behind this was that is what not a matter of how to weight each test but which tests were included. It the authors of HPCC determined one feature was not well represented, all they had to do was to add a test exercising that feature.

Most powerful machines using 100% HPL:

Machine Name- #CPUS	Tflops
Cray X1- 252	2.35
Cray X1- 124	1.18
Linux Networx- 256	1.03
IBM Power4- 504	0.903
IBM Power4- 256	0.654
HP DEC Alpha- 484	0.618
Cray X1- 60	0.58
SGI Altix- 128	0.52

Using only HPL we see that while the Cray X1 using 252 MSPs is at top and 124 MSP is second, a Xeon Linux Networx cluster using third just barely less than half the power of the top machine. The rest of the list include 2 IBMs, an HP based on the Alpha chip, another Cray X1 using 60 MSPs and an SGI Altix.

Things already start to look very different if we have a 50% HPL, 50% other weighting. Machines that moved up in the list are colored blue while those that moved down are colored in red. The position using only HPL is included for comparison purposes.

Most powerful machines using 50% HPL – 50% other:

Machine Name- # CPUS	HPCC Score	HPL Order
Cray X1- 252	25.6	1
Cray X1- 124	14.7	2
Cray X1- 60	8.23	7
Linux Networx- 256	6.63	3
Cray T3E- 1024	6.32	16
IBM Power4- 504	6.21	4
HP DEC Alpha- 484	5.27	6
Cray X1- 32	5.00	10

One can see that not only did the 60 MSP X1 move ahead of the Linux Network cluster, the score for that cluster says it is about 1/4 the power of the top machine rather than 1/2.

Also a T3E and a 32 MSP Cray X1 has moved into the top 8 displacing an IBM and the Altix.

Finally, we see what the list looks like with equal weighting.

Most powerful machines using equal weighting:

Machine Name- # CPUS	HPCC Score	HPL Order
Cray X1- 252	26.5	1
Cray X1- 124	16.4	2
Cray T3E- 1024	10.2	16
Cray X1- 60	9.75	7
Cray X1- 32	6.43	10
HP DEC Alpha- 484	4.54	6
IBM Power4- 504	4.15	4
Linux Networx- 256	3.99	3

Clearly the Cray machines dominate the list when all tests are considered equal. Of particular interest is the position and score of the Linux cluster. Is has dropped from 3rd to 8th and the score now shows that it is less than 1/6 the power of the top machine.

Conclusions:

While LINPACK is the most popular way to compare machines across architectures and sites, it is apparent that it emphasizes machine characteristics that work well on any platform and it is not representative of today's HPC workload. HPCC is a new benchmark that examines those other features that matter to HPC today, but it results in too many numbers to be used directly. Instead, it is highly desirable to combine those numbers into a single score so one can answer the question, "How well did you do one HPCC". One that score is calculated and a list generated, one can go back to further examine the results of machines with similar scores.

In the end, the HPC Challenge is a powerful new tool which is easy to use but yet extremely useful for comparing the machines built to solve the worlds most difficult problems.

About the Authors:

Nathan Wichmann has been a benchmark and applications engineer with Cray for six years. He has experience with a variety of codes and is frequently involved with defining feature requirements for compilers and hardware.