Optimizing the HPCC Randomaccess Benchmark on Blue Gene/L Supercomputer

Rahul Garg
IBM India Research Lab
Block-I, IIT Delhi, Hauz Khas-16
New Delhi, India
grahul@in.ibm.com

Yogish Sabharwal
IBM India Research Lab
Block-I, IIT Delhi, Hauz Khas-16
New Delhi, India
ysabharwal@in.ibm.com

Categories and Subject Descriptors: C.4 [Performance of Systems]: Measurement techniques

General Terms: Measurement, Performance.

Keywords: High Performance Computing, Benchmarks, Randomaccess

1. INTRODUCTION

The performance of supercomputers has traditionally been evaluated using the LINPACK benchmark [3], which stresses only the floating point units without significantly loading the memory or the network subsystems.

The HPC Challenge (HPCC) benchmark suite is being proposed as an alternative to evaluate the performance of supercomputers. It consists of seven benchmarks, each designed to measure a specific aspect of the system performance. These benchmarks include (i) the high performance linpack (HPL) (ii) *DGEMM*, which measures the floating point rate of execution of double precision real matrix-matrix multiplication, (iii) *STREAM* that measures sustainable memory bandwidth and the corresponding computation rate for four simple vector kernels, namely, *copy*, *scale*, *add* and *triad* (iv) *PTRANS* that exercises the network by taking parallel transpose of a large distributed matrix (v) *Randomaccess* that measures the rate of integer updates to random memory locations (vi) *FFT* which measures the floating point rate of execution of a double precision complex one-dimensional Discrete Fourier Transform (DFT) and (vii) *communication bandwidth and latency* which measures latency and bandwidth of a number of simultaneous communication patterns.

In this paper we outline the optimization techniques used to obtain the presently best reported performance of the HPCC Randomaccess benchmark on the Blue Gene/L supercomputer.

2. DESCRIPTION OF THE HPCC RANDOMACCESS BENCHMARK

The Randomaccess benchmark is motivated by the growing gap in the CPU and memory performance. The benchmark operates on a distributed table T of size 2^k where k is largest integer such that 2^k is less than or equal to the size of total system memory. Each processor generates a random sequence of 64 bit integers. For each random number, (say a_i), the most significant bits are selected to index into the distributed table T. The entry (which may reside on a remote node) is *xor*'ed with the random number a_i. The performance of the system is measured by the number of *giga updates per second* (GUPS) performed by the system.

Two types of performance numbers are reported. A *baseline run* is obtained by compiling and running the supplied code without making any changes (except for linking certain optimized libraries such as ESSL). In an *optimized run*, the function that implements the benchmark may be replaced by an optimized, system-specific implementation. This implementation must (a) use the same random number generator as in the baseline code (b) ensure that at any stage, the number of pending updates stored at any node does not exceed 1024 and (c) pass the verification test which is performed outside the benchmark.

3. BOTTLENECK ANALYSIS

The potential bottlenecks for the Randomaccess benchmark are the CPU and memory subsystem, the communication network or a combination of both.

The CPU bottleneck is given by

$$GUPS \leq \frac{N}{t_g + t_u + t_o^s + ((N-1)/N) \cdot (t_s + t_r + t_o^p)} \quad (1)$$

where N represents the number of processors in the system, t_g and t_u respectively represent the average time to generate and perform an update, t_s and t_r represent the average time per update to perform *send* and *receive* operations and t_o^s and t_o^p represent additional overheads (see [5] for details).

It turns out that the network bottleneck is related to a graph-theoretic property of the underlying network called the *smallest edge expansion* [1], $\alpha(G)$, formally defined as

$$\alpha(G) = \min_{S \subset V : |S| \leq |V|/2} \frac{C(S, \overline{S})}{|S|}.$$

A related property of a communication network, called *bisection bandwidth*, B, formally defined as

$$B = \min_{S \subset V : |S| = |V|/2} C(S, \overline{S})$$

where $C(S, \overline{S})$ is the capacity of the cut separating S and $\overline{S}$, and V is the set of nodes in the system.

The network bottleneck is given by (see [5] for a proof)

$$GUPS \leq \frac{2N}{b} \alpha(G) \leq 4B/b \quad (2)$$

SIGMetrics/Performance'06, June 26–30, 2006, Saint Malo, France.
ACM 1-59593-320-4/06/0006.

where b is the average number of bytes per update (including fixed packet overheads).

For a 3-dimensional torus network, $\alpha(G) = O(N^{-1/3})$. Thus, for Blue Gene/L, which has a 3-d torus network [4], the network bottleneck scales as $O(N^{2/3})$, whereas the CPU bottleneck scales as $O(N)$. Hence, for very large number of nodes, the benchmark performance is expected to be limited by the network.

4. OPTIMIZING THE BENCHMARK

The HPCC rules allow each processor to store upto 1024 updates. The baseline code maintains one bucket for every destination node. It generates updates and stores them in the corresponding bucket until there are 1024 pending updates in the buckets. It then determines the largest bucket and dispatches the updates of this bucket to its destination. For small number of nodes ($N \leq 1024$), bucketing of updates improves the performance of the benchmark as the packet send and receive overheads are divided over more updates, thereby reducing the average time per update. However, as the number of nodes becomes large ($N >> 1024$), the average number of updates packed in a packet decreases significantly, reducing to almost 1 for $N > 8192$. Therefore, bucketing may not give any performance benefits for large number of nodes.

The profile of the baseline code indicated that most of the time was being spent in MPI function calls and bucketing logic. We then profiled the raw device interface and found that its use would eliminate most of the MPI overheads. We then calculated the bottleneck using the raw device interface performance measurements, Blue Gene/L torus network bandwidth and packet size. We found the network to be the bottleneck for all configurations of nodes ($N \geq 32$).

In order to retain the advantages of packing more updates even when the number of processing nodes is large, we designed a software-based dimension ordered routing technique. Let the triplet $\langle x_i, y_i, z_i \rangle$ denote the co-ordinates of a processing node i on the 3-d torus network of Blue Gene/L. An update from processing node i to j is routed along a fixed path in a dimension ordered manner. It is first routed along the x-dimension, then along the y-dimension and finally along the z-dimension to its destination. Let $i = \langle x_i, y_i, z_i \rangle$ be the source node and $j = \langle x_j, y_j, z_j \rangle$ be the destination node. Suppose that $x_i \neq x_j$, $y_i \neq y_j$ and $z_i \neq z_j$. Then the first and second software routers in the path of the update from i to j are $\langle x_j, y_i, z_i \rangle$ and $\langle x_j, y_j, z_i \rangle$ respectively. If a co-ordinate of the source and destination nodes is same along a dimension, the software routing hop corresponding to that dimension is not required. Therefore any update is routed in the software at most twice. As can be observed, a processing node only sends updates to another processing node if it lies along its x, y or z dimension on the torus. This limits the number of nodes that a node communicates with, therefore reducing the number of buckets and hence allowing more updates to be packed in each packet.

5. PERFORMANCE RESULTS

Table 1 lists the performance of the baseline and optimized MPIRandomaccess implementations.

The baseline code is limited by the processing required for each update, not by the network bandwidth. Its performance increases steadily (though not linearly) till $N = 16$K.

Number of Nodes	Dimensions ($X \times Y \times Z$)	Base GUPs	Opt GUPs	Bottleneck CPU	Bottleneck N/W
32	$4 \times 4 \times 2$	0.022	0.062	0.06	0.6015
128	$8 \times 4 \times 4$	0.071	0.226	0.21	1.2030
512	$8 \times 8 \times 8$	0.190	0.844	0.81	9.6237
1024	$8 \times 8 \times 16$	0.290	1.780	1.60	9.6237
2048	$16 \times 8 \times 16$	0.450	3.300	3.14	18.8235
4096	$8 \times 32 \times 16$	0.680	5.830	6.23	17.5081
8192	$16 \times 32 \times 16$	1.080	10.990	12.25	32.8113
16384	$32 \times 32 \times 16$	1.274	18.030	24.30	58.6473
65536	$64 \times 32 \times 32$	0.065	35.471	95.97	91.8974

Table 1: MPIRandomaccess Performance Results on BG/L for Optimized code

There is a degradation of the performance for $N = 64$K. The speedup is not linear because the average number of updates per packet decreases and the depth of the heap to be traversed for maintaining the buckets increases as the number of nodes increase. We suspect that the performance degradation at $N = 64$K is due the size of the working set of the baseline implementation exceeding the L3 cache size.

For the optimized code, we projected the CPU bottleneck using performance numbers from 32 nodes. We measured the generate and update time by running the benchmark on a single processor. We back-calculated the software routing overheads from the 32 node performance numbers and then projected the performance numbers for larger number of nodes. The theoretical network bottleneck (assuming 100% network utilization) was calculated using (2).

The optimized code scales almost linearly with number of nodes (for $N \leq 8192$). Up to 8192 nodes, the performance is limited by the CPU (within 10% of the projected CPU bottleneck). Our experiments indicated that on 16K nodes or less, the network was almost always ready to accept packets when requested, whereas, on 64K nodes, the network was not ready 90% of the time. However, according to our calculations, the network should not have become the bottleneck. We suspect that a combination of temporary hot-spots in the network and blocking at individual nodes is the cause of the observed performance gap.

6. REFERENCES

[1] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 222–231, New York, NY, USA, 2004. ACM Press.

[2] J. Dongarra and P. Luszczek. Introduction to the hpc challenge benchmark suite. Technical Report ICL-UT-05-01, ICL, 2005.

[3] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[4] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49:195–212, 2005.

[5] R. Garg and Y. Sabharwal. Analysis and optimization of the hpcc randomaccess benchmark on bluegene/l supercomputer : Extended version. Technical Report RI-05-010, IBM, 2006.