RI-05-010 February, 2006 Subject: Computer Science

IBM Research Report

Analysis and Optimization of the HPCC Randomaccess Benchmark on BlueGene/L Supercomputer : Extended Version

Rahul Garg
grahul@in.ibm.com

Yogish Sabharwal
ysabharwal@in.ibm.com

IBM Research Division,
IBM India Research Lab,
Hauz Khas, New Delhi - 110016. INDIA.
Phone: +91-11-2686-1100, Fax: +91-11-2686-1555

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Abstract

The performance of supercomputers has traditionally been evaluated using the LINPACK benchmark, 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. The HPCC benchmark suite consists of seven benchmarks, each designed to measure a specific aspect of the system performance.

In this paper, we analyze the HPCC Randomaccess benchmark which is designed to measure the performance of random memory updates. We study the performance of this benchmark on the BlueGene/L supercomputer. We describe the optimizations made to get optimal performance and discuss the inherent limits in scaling this benchmark to massively parallel systems.

1 Introduction

The top500 project [11] lists the 500 most powerful computers in the world. This list is updated twice every year and has been the most widely cited reference for comparing the supercomputer performance. The supercomputing systems are ranked on the basis of their floating point operations per second (FLOPS) performance while solving a dense system of linear equations of the form $Ax = b$. The benchmark, called high performance linpack (HPL) [4] uses a parallel implementation of LU factorization to solve the system of linear equations. The algorithm operates on a matrix of size $n \times n$, requires $O(n^3)$ floating point operations and has very high data locality. Thus, the reported performance is often very close to the theoretical peak floating point performance of the systems.

The HPC Challenge (HPCC) benchmark suite attempts to augment this benchmark by measuring additional aspects of system performance involving memory and network. It consists of seven benchmarks: (i) *HPL*, the high performance linpack benchmark (ii) *DGEMM*, which measures the floating point rate of execution of double precision real matrix multiplication (iii) *STREAM*, which measures sustainable memory bandwidth and the corresponding computation rate for four simple vector kernels, namely, *copy*, *scale*, *add* and *triad* (iv) *PTRANS*, which exercises the network by taking parallel transpose of a large distributed matrix (v) *Randomaccess* which 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* benchmarks which measure latency and bandwidth for a number of simultaneous communication patterns.

In this paper, we share our experiences in optimizing the HPCC Randomaccess benchmark on the BlueGene/L supercomputer. We analyzed the bottlenecks of this benchmark on the BlueGene/L system. We found that on large systems, the performance of this benchmark is inherently limited by the bandwidth of the interconnection network. We formally show that on a variety of systems, the performance of this benchmark is asymptotically limited by a well-studied graph-theoretic property of the interconnection network called the *sparsest cut* [2] (which is also related to the minimum balanced cut or the bisection bandwidth of the network).

We designed and implemented a novel technique based on aggregation and software routing that utilizes the network more efficiently. Our optimized implementation of this benchmark on a 64-rack Blue Gene/L system led to a performance of 35.5 GUPs (giga updates per second), the best known so far. This performance is approximately factor of two better than the optimized UPC-based implementation of the benchmark the same system and factor 4.5 better than the best performance obtained on any other system.

We also found that the performance of baseline code supplied with this benchmark is very sensitive to its implementation. In addition, the benchmark specifications allow systems with small number of nodes to amortize their fixed overheads more efficiently as compared to large systems. We suggest some additional characteristics of remote memory performance that might be of importance to applications on large parallel systems.

The rest of the paper is organized as follows. Section 2 gives an overview of the BlueGene/L architecture. In Section 3, we describe the HPCC Randomaccess benchmark in more detail. The bottlenecks of this benchmark on BlueGene/L are analyzed in Section 4. Our software routing technique for improving the benchmark performance are described in Section 5. The performance results of the baseline and optimized implementations of the benchmark are presented in Section 6. Section 7 concludes with discussions and suggestions for future work.

2 BlueGene/L Overview

The BlueGene/L is a massively parallel supercomputer that scales upto 65,536 dual-processor nodes [5]. The nodes themselves are physically small, allowing for very high packaging density in order to realize optimum cost-performance ratio.

Each node has two processor cores, allowing the system to run in two different modes. In the *coprocessor mode* one of the processors is dedicated to messaging and one is available for application computation. In the *virtual node mode* each node is logically separated into two nodes, each of which has a processor and half of the physical memory. Each processor is responsible for its own messaging. In this mode, the node runs two application processes, one on each processor.

Each of the two cores is an embedded `PPC440` core, designed to reach a nominal clock frequency of 700 MHz. The `PPC440` is a high-performance, superscalar implementation of the full 32-bit Book-E Enhanced PowerPC Architecture. It has a seven-stage, highly pipelined microarchitecture with dual instruction fetch, decode, out-of-order issue. It also has out-of-order dispatch, execution and completion. The independent 32-KB L1 instruction and data caches have a 32-byte line and 64-way associativity with round-robin replacement policy [8]. The L1 data cache supports write-back and write-through operations and is non-blocking with upto four outstanding load misses. There is an L3 cache on every node, composed of 4-MB embedded DRAM with cache lines of 128 bytes. The latency for an L1 cache access is 3 cycles and for L3 cache access is 28-40 cycles. The latency for main memory access is 86 cycles with L3 cache enabled.

The aggregate memory of the system is completely distributed in the style of a multicomputer with no hardware sharing between the nodes. Each node has 512 MB of physical memory which is shared by its two processors. Each `PPC440` processor has independent 16-byte read and write data buses and an independent 16-byte instruction bus.

The BlueGene/L uses five interconnect networks for I/O, debug, and various types of interprocessor communication. *Gigabit Ethernet* is used to support the file system. *Fast Ethernet* and *JTAG (IEEE 1149.1)* are used for diagnostics, debugging, and some aspects of initialization. The *three dimensional torus* [1] supports high-bandwidth, low latency point-to-point communication. The physical network comprises of a 3D cubic grid in which each node is connected to its six neighboring nodes along the three dimensions with high speed dedicated links. Communication with remote nodes involves transit through other nodes in the system. The *collective network* is designed to efficiently perform collective operations such as global summation. The *global barrier network* is designed to perform global barrier and global interrupt operations efficiently. This network has four independent channels that are optimized to exhibit very low latencies.

The most significant of these interconnection networks is the $64 \times 32 \times 32$ three-dimensional torus that has the highest aggregate bandwidth and handles the bulk of all communication. The signaling rate for the nearest neighbor links is 1.4 Gbps in each direction. Each node supports six independent, bidirectional nearest neighbor links, with an aggregate bandwidth of 2.1 GB/s. The hardware latency to transit a node is approximately 100 ns. The torus network uses both dynamic (adaptive) and deterministic routing with virtual buffering and cut-through capability. Adaptive routing allows packets to follow any minimal path to the final destination, allowing packets to dynamically "choose" less congested routes. Four virtual channels are supported in the torus network, contributing to efficient deadlock-free comunication. The messaging is based on variable size packets, each $n \times 32$ bytes, where $n = 1$ to 8 "chunks". The first eight bytes of each packet contain link-level information, routing information and a byte-wide cyclic redundancy check (CRC) that detects header data

corruption during transmission. In addition, a 24-bit CRC is appended to each packet, along with a one byte valid indicator. The valid indicator is necessary, since packets can be forwarded before being entirely received. A robust error recovery mechanism that involves an acknowledgement and retransmission protocol is implemented across the physical links that connect two BG/L compute nodes.

3 Description of the HPCC Randomaccess Benchmark

The Randomaccess benchmark is motivated by the growing gap in the CPU and memory performance. Several architectural enhancements such as bigger caches, wider line sizes, complex pre-fetch and cache replacement policies tend to improve performance of applications with data locality and sequential access. However, these enhancements may impact the performance of applications that access the memory in random or unpredictable manner. This benchmark intends to measure the peak capacity of the memory subsystem while performing random updates to the system memory. The parallel version of the Randomaccess benchmark, called MPIRandomaccess measures the performance of the system while carrying out local as well as remote updates to the total system memory in parallel.

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 using the primitive polynomial $x^{63} + x^2 + x + 1$ over GF(2). For each random number (say a_i), the most significant bits are selected to index into the distributed table T. The bit-wise *xor* of the random number a_i and the selected entry in the table (which may also reside on a remote node) is computed, and stored back at the same location in the table. The number of such updates performed by each node is dynamically determined at the beginning in a way that the benchmark terminates in a reasonable period of time. The benchmark specifications allow each node to look-ahead and store at most 1024 updates before they are applied to the table.

Some implementations of the benchmark are provided by the committee [3]. This includes a basic and a vector/multi-threaded implementation of sequential Randomaccess in C. The vector implementation "looks ahead" the sequence of updates before actually carrying them out. This allows the updates to be pipelined. A MPI-based and a UPC based parallel implementation of Randomaccess is also supplied with the benchmark. In the MPI implementation, each node maintains a bucket containing pending updates for every other destination. The local updates are carried out immediately while the remote updates are added to the buckets corresponding to their destinations. As soon as the number of pending updates hit the limit of 1024, the bucket with largest number of updates is sent out to its destination. Upon receiving a packet, a node updates its local table using the random numbers stored in the packet.

The performance of the system is measured by the number of *giga updates per second* (GUPS) performed by the system. In addition to the total system performance, the benchmark also reports the performance of a single node version (where a node carries out updates to its local memory while other nodes are silent) and an embarrassingly parallel version (all the nodes independently update their local table in parallel).

Two types of performance numbers may be reported. The *baseline runs* are obtained by compiling and running the supplied code *as is*. In the baseline runs, no change maybe made to the supplied code. Suitable compiler options and vendor-supplied MPI library may be used while compiling and linking the benchmark.

An "optimized run" may be obtained by replacing the function that implements the benchmark with an

optimized version that may make use of system specific features. The optimized implementation must adhere to the basic definition of the benchmark i.e. (a) it should generate and carry out updates using the specified polynomial (b) the table T should approximately fill half the total system memory and (c) the look ahead and stored updates must be no more than 1024. Moreover, the optimized implementation must not reorder the stream of updates such that all updates become local [9].

4 Bottleneck Analysis

MPIRandomaccess performs four basic operations at all the nodes: *generate*, *send*, *receive* and *update*. The operation *generate* computes the next random number in the series using the specified generator polynomial. If the table update is to be carried out on a remote memory, then a *send* is performed (possibly after bunching a number of updates). Operation *send* involves copying the data into network/system buffers (if required) and initiating the data transfer. The operation *receive* copies packets from the network/system buffer to application buffers. Operation *update* performs the local table update by reading the required entry from the table, performing the *xor* operation with the random number received in the packet and writing the result back into the table. The bottleneck while running this benchmark could be either due to the CPU and memory subsystem or the communication network or a combination of both. We examine these possibilities in more detail.

4.1 CPU and Memory Subsystem Bottleneck

The performance of sequential Randomaccess benchmark depends on the time taken to perform *generate* and *update* operations. Let t_g and t_u respectively represent the average time to generate and perform an update. Let t_o^s be the overhead in executing these operations in a loop. The performance of the sequential randomaccess benchmark is given by $GUPS = 1/(t_g + t_u + t_o^s)$.

Operation *generate* requires one bit shift and one conditional xor operation on a 64-bit integer. Most of the current architectures have capability to perform these operations in small number of cycles. Also, with currently available optimizing compilers the loop overheads are likely to be small.

According to the benchmark specifications, the local table should occupy approximately half the local memory, which is expected to be bigger than the size of largest cache in the system. Therefore, random updates to the memory are likely to miss all the levels of cache hierarchy. Thus, the time taken for the *update* operation is likely to be dominated by the main memory latency.

On BlueGene/L the latency to access main memory is 86 cycles [8]. The observed performance of sequential Randomaccess on BlueGene/L is 0.0067 GUPS which translates into an average value of 104 cycles for $t_g + t_u + t_o^s$. Thus, the generate-xor operations including the loop overheads take only 18 cycles per update.

Let t_s and t_r represent the average time per update to perform *send* and *receive* operations. The time taken by the *send* operation corresponds to the time taken to initiate the data transfer. It does not include the time taken to actually perform the data transfer (which may be done asynchronously). Similarly the time taken by the *receive* operation is given by the time taken to get the data into application memory (from network/system buffers) rather than to receive the data from the network. These times typically depend on the overheads of the communication library used. Moreover, if multiple updates are communicated using a single call to send/recv functions, the corresponding overheads get divided by the number of updates transmitted.

Let N be the number of processors in the system. In the case of MPIRandomaccess, the expected fraction of local and remote updates will be $1/N$ and $(N-1)/N$ respectively. Each remote update must be communicated to the remote node using a *send* and a *receive* operation. Therefore, the CPU bottleneck will restrict the performance of MPIRandomaccess to

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

where t_o^p represents the additional overheads in performing these operations.

4.2 Network Bottleneck

The global table is updated using a *xor* operation which is associative. The order in which the updates are performed has no impact on the final result. A node updating remote memory need not wait for the operation to complete before generating the next update. It may asynchronously send the remote update request over the network and proceed with further updates. As a result, the network latency can be completely hidden by pipelining. If implemented carefully, the network latency should have no impact on the performance of this benchmark.

For analyzing the impact of network bandwidth on the benchmark performance, we model the communication network as a directed graph $G = (V, E)$ with capacities on the edges. Vertices V in the graph represent compute nodes and the edges E represent communication links. The number of nodes in the system is given by $N = |V|$. The capacity of an edge is equal to the capacity of the corresponding link in the system (1.4 Gbps in the case of BlueGene/L). Let S be a subset of vertices in this graph and $\overline{S}$ represent $V - S$. Let $C(S, \overline{S})$ represent the sum of the capacity of edges from S to $\overline{S}$. Let U represent the average number of updates generated by a node and b represent the average number of bytes sent per update (including the header overheads) by the source nodes. A single packet may contain multiple updates. In this case header overheads are shared equally by all the packets.

Now consider a vertex in S. The expected number of updates from the vertex to other vertices in $\overline{S}$ in given by $U|\overline{S}|/N$ (since the destinations are chosen uniformly at random). Therefore the expected total number of updates from S to $\overline{S}$ is given by $U|S||\overline{S}|/N$. Let $t(S)$ represent the time taken by updates in S to reach $\overline{S}$. The time $t(S)$ is bounded by the number of updates crossing the cut $(S, \overline{S})$ and the capacity of the cut $C(S, \overline{S})$ as

$$t(S) \geq \frac{bU|S||\overline{S}|}{NC(S, \overline{S})}$$

The total number of updates performed by the system is given by NU. Thus the MPIRandomaccess performance is bounded by

$$GUPS \leq \frac{N^2 C(S, \overline{S})}{b|S||\overline{S}|}$$

The above expression is true for each set S of V. Thus

$$GUPS \leq \min_{S \subset V} \frac{N^2 C(S, \overline{S})}{b|S||\overline{S}|}$$

The right hand side of above expression is very closely related to a well-studied problem in graph thoery called the *sparsest cut* [2]. Formally, the *smallest edge expansion* $\alpha(G)$ of a graph G is defined as

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

The cut that achieves the smallest edge expansion is called the sparsest cut. It is easy to see that

$$GUPS \leq \frac{2N}{b}\alpha(G)$$

The smallest edge expansion is also related to the bisection bandwidth of the network. Formally, the bisection bandwidth B is the capacity of minimum balanced cut.

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

It may be noted that $\alpha(G) \leq 2B/N$. Thus,

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

4.3 Bottleneck on BlueGene/L

From (1) it is apparent that the CPU bottleneck for MPIRandomaccess scales linearly with the number of nodes. However, the network bottleneck depends on the smallest edge expansion of the underlying communications network. If the smallest edge expansion diminishes with the number of nodes then the network is expected to become the bottleneck for large number of nodes. For a d-dimensional grid, the smallest edge expansion is given by $\alpha(G) = O(N^{-1/d})$, for a hypercube network, it is given by $\alpha(G) = O(1)$.

For the 3-dimensional torus network of BlueGene/L the smallest edge expansion is $O(n^{-1/3})$. Therefore, the network is expected to limit the performance of MPIRandomaccess when the number of nodes is very large. However, to determine the bottleneck on a small number of nodes, one needs to substitute the actual values of parameters $t_g, t_u, t_o^s, t_s, t_r, t_o^p, b, \alpha(G)$.

The value of $t_g + t_u + t_o^s$, as estimated from the performance of the sequential randomaccess benchmark is found to be 104 cycles (see Section 4.1).

4.3.1 Using MPI library

In order to estimate t_s and t_r we wrote some micro-benchmarks. For measuring t_s, the benchmark designates one node in the system as a sender another as a receiver. Both the nodes first synchronize using a barrier. The receiver node then posts a receive using the `MPI_Recv` function call. The sender node waits for a sufficiently long duration to ensure that the receiver has posted a receive. It then posts a message to the receiver using `MPI_Send` and measures the time taken by the function call. Both the nodes synchronize again using a barrier. The same sequence repeats multiple times. First few observations are discarded to factor out the effects of cache warm-up and protocol setup latency.

To estimate t_r, the second benchmark also selects a sender and a receiver node as before. Both these nodes first synchronize using a barrier. The sender posts a message using the `MPI_Send` function. The receiver waits for a sufficiently long duration to ensure that the message has arrived at its destination. It then posts a `MPI_recv` and measures the time taken by the function call. The same sequence is repeated multiple times and first few observations are discarded.

Table 1 shows the mean and variance of the number of CPU cycles taken by `MPI_Send` and `MPI_Recv` function calls for different payload sizes. The above two micro-benchmarks were executed for different pairs of nodes. The variation in mean time was found to be within 5%.

Payload	`MPI_Send`		`MPI_Recv`	
Size	mean	variance	mean	variance
8	1544	953	1271	1754
16	1532	922	1264	1845
32	1546	957	1293	2249
64	1553	867	1298	2206
128	1584	1123	1347	1825
256	2059	1248	1943	2357
512	2499	1069	2463	2386
1024	12270	13298	10306	6689
2048	15255	26277	14480	7117
4096	23725	13189	23957	7101
8192	43972	11323	77908	6948

Table 1: Mean and variance of number of CPU cycles taken by MPI send and receive calls

The packet size on the BlueGene/L network varies from 32 bytes to 256 bytes in multiples of 32 bytes. The total header overhead (including the hardware packet overhead) when using MPI is 48 bytes. Therefore the size of an MPI packet carrying a single 8-byte update will be 64 bytes. In addition, we have observed a packet communication overhead in the torus network equivalent to 14 bytes. Therefore the equivalent packet size b of a MPI packet containing a single update is 78 bytes.

Finding the sparsest cut for general graphs is a NP-hard problem [10]. However, for BlueGene/L 3-dimensional torus network, sparsest cut can be computed by examining the face with smallest area. Table 2 lists the smallest edge expansion (in units of link capacity C) for BlueGene/L partition of different sizes and shapes.

N	Dimension $(X \times Y \times Z)$	Torus/ Mesh	Smallest face area	$\alpha(G)/C$
32	$4 \times 4 \times 2$	M	8	1/2
64	$8 \times 4 \times 2$	M	8	1/4
128	$8 \times 4 \times 4$	M	16	1/4
256	$8 \times 4 \times 8$	M	32	1/4
512	$8 \times 8 \times 8$	T	64	1/2
1024	$8 \times 8 \times 16$	T	64	1/4
2048	$16 \times 8 \times 16$	T	128	1/4
4096	$8 \times 32 \times 16$	T	128	1/8
8192	$16 \times 32 \times 16$	T	256	1/8
16384	$32 \times 32 \times 16$	T	512	1/8
32768	$32 \times 32 \times 32$	T	1024	1/8
65536	$64 \times 32 \times 32$	T	1024	1/16

Table 2: Smallest edge expansions for BlueGene/L torus network

It may also be observed that the bound obtained in (2) is tight for any regular d-dimensional torus network (including the BlueGene/L torus network). Thus, if the network becomes the bottleneck, the performance of $(2N/b) \cdot \alpha(G)$ is theoretically achievable (assuming 100% link utilization) by dynamic shortest path routing of BlueGene/L.

N	Dimension ($X \times Y \times Z$)	$\alpha(G)$ / C	CPU Bottleneck	N/W Bottleneck
32	$4 \times 4 \times 2$	1/2	0.0077	0.0718
64	$8 \times 4 \times 2$	1/4	0.0152	0.0718
128	$8 \times 4 \times 4$	1/4	0.0302	0.1436
256	$8 \times 4 \times 8$	1/4	0.0602	0.2872
512	$8 \times 8 \times 8$	1/2	0.1202	1.1487
1024	$8 \times 8 \times 16$	1/4	0.2402	1.1487
2048	$16 \times 8 \times 16$	1/4	0.4802	2.2974
4096	$8 \times 32 \times 16$	1/8	0.9601	2.2974
8192	$16 \times 32 \times 16$	1/8	1.9200	4.5949
16384	$32 \times 32 \times 16$	1/8	3.8398	9.1897
32768	$32 \times 32 \times 32$	1/8	7.6793	18.3795
65536	$64 \times 32 \times 32$	1/16	15.3585	18.3795

Table 3: CPU and network bottleneck GUPS for BlueGene/L with MPI

Table 3 lists the CPU and network bottlenecks (in unit of GUPS) for the BlueGene/L systems using inequalities (1) and (2) and the data presented earlier. These bottleneck calculations assume that each remote update is sent/received using a single `MPI_Send/MPI_Recv` call. It is apparent that with these assumptions, the MPI function call latencies are the source of bottleneck on BlueGene/L system of size upto and including 65,536.

4.3.2 Using the raw device interfaces

The MPIRandomaccess benchmark does not require the complete functionality of MPI-2. It is possible to reduce t_s and t_r using a light-weight communication library. In order to estimate the inherent bottleneck in the BlueGene/L system, we experimented with raw network device interface. We wrote micro-benchmarks to measure the latencies of writing/reading directly to/from the network device. The results are presented in Table 4. On the send side there is a base overhead of 6 cycles in addition to 0.25 cycles per byte for copying the data. On the receive side, the fixed overhead is 34 cycles and the variable overhead is 0.625 cycles per byte for copying data.

If the raw network device interface is used, there is no need for software and MPI headers. Therefore an 8-byte update can easily be packed in a 32-byte packet. This gives $b = 32 + 14 = 44$. Table 5 lists the CPU and network bottlenecks (in units of GUPS) for the BlueGene/L system assuming send/recv latencies of Table 4 and $b = 44$ bytes. With these assumptions, it is apparent that the network bandwidth is the bottleneck for BlueGene/L systems of all sizes.

Fundamentally, for each update only 8 bytes need to be communicated while the calculations above assume 44 bytes per update. It might be possible to get a significant improvement in the network bottleneck on the

Payload	Send		Recv	
Size	mean	variance	mean	variance
32	14	0	54	0.32
64	22	0	98	0.16
96	28	0	118	0.18
128	36	0	120	0.34
160	50	0	140	0.27
192	64	0	160	0.30
224	70	0	172	0.15
256	70	0	192	0.43

Table 4: Mean and variance of number of cycles taken by send and receive calls using raw network device

N	Dimension $(X \times Y \times Z)$	$\alpha(G)$ / C	CPU Bottleneck	N/W Bottleneck
32	$4 \times 4 \times 2$	1/2	0.1303	0.1217
64	$8 \times 4 \times 2$	1/4	0.2590	0.1217
128	$8 \times 4 \times 4$	1/4	0.5163	0.2435
256	$8 \times 4 \times 8$	1/4	1.0310	0.4870
512	$8 \times 8 \times 8$	1/2	2.0604	1.9478
1024	$8 \times 8 \times 16$	1/4	4.1193	1.9478
2048	$16 \times 8 \times 16$	1/4	8.2369	3.8957
4096	$8 \times 32 \times 16$	1/8	16.4722	3.8957
8192	$16 \times 32 \times 16$	1/8	32.9428	7.7913
16384	$32 \times 32 \times 16$	1/8	65.8840	15.5826
32768	$32 \times 32 \times 32$	1/8	131.7663	31.1652
65536	$64 \times 32 \times 32$	1/16	263.531	31.1652

Table 5: CPU and network bottleneck GUPS for BlueGene/L using raw network device

BlueGene/L system by packing more than one update in a packet. In the next section we describe our software routing technique that achieves this goal even on large systems.

5 Optimizing the Benchmark

5.1 Bucketing

The HPCC rules allow each processor to store upto 1024 updates. This allows for more updates to be packed in the messages that are communicated between the nodes, therefore increasing the performance due to the following reasons:

- The packet send and receive overheads are divided over more updates, thereby reducing the processing time per update. This results in increased performance when CPU processing is the bottleneck.

- The packet header/trailer overheads are distributed over more updates. Moreover, on some systems there is a minimum packet size, that is generally large compared to the size of the update. For instance, the BlueGene/L has a minimum packet size of 32 bytes, whereas an update is only 8 bytes long. This improves bandwidth utilization of the communication network and therefore results in increased performance when the network bandwidth is the bottleneck.

Recall that the baseline code maintains one bucket for every destination node. As soon as the limit of 1024 pending updates is reached, the largest bucket is dispatched to its destination. We wrote a simple simulator to estimate the average bucket size for different number of nodes. The results are presented in Table 6. As the number of processing nodes increase, the average number of updates that are packed in a message decreases. This indicates that only systems with small number of nodes can take advantage of the bucketing. On large systems, bucketing only adds to the processing overheads.

5.2 Software Routing

In order to retain the advantages of bucketing updates when the number of processing nodes is large, we route the updates via intermediate nodes. We club together updates for a group of destination nodes and send them to an intermediate processing node, called the routing node. Thus, each node, in addition to generating and processing updates, is also responsible for routing updates received from other nodes. This limits the number of communicating node pairs, therefore reducing the number of buckets and hence allowing more updates to be packed in each message.

The BlueGene/L has a 3-dimensional torus interconnection network. 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. 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, 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. We call these routers the x-routers, y-routers and z-routers of the node respectively. Let X_{max}, Y_{max} and Z_{max} denote the size of the torus dimension along the x, y and z dimensions respectively. Thus any node communicates with $X_{max} - 1$ nodes along its x dimension, $Y_{max} - 1$ nodes along its y dimension and $Z_{max} - 1$ nodes along its z dimension. The total number of routing nodes for a processing node is $X_{max} + Y_{max} + Z_{max} - 3$.

5.3 The algorithm

The algorithm maintains a bucket for each of its routing nodes. It has $X_{max} - 1$ buckets for x-routers, $Y_{max} - 1$ buckets for y-routers and $Z_{max} - 1$ buckets for z-routers. In addition, it has three buckets that are used for receiving updates (one from each dimension). Thus, there are a total of $X_{max} + Y_{max} + Z_{max}$ buckets at every node. Every bucket has a capacity of $1024/(X_{max} + Y_{max} + Z_{max})$. Each update is accumulated in the bucket corresponding to its next hop software router. When a bucket becomes full, it is sent to the corresponding software routing node.

Each packet is received into the receive bucket corresponding to the dimension from which it arrives. The next packet from that dimension is received only when the algorithm finishes processing all the previous updates in the corresponding receive bucket. This ensures that the total number of updates pending in all the buckets put together on a processing node never exceeds 1024.

Note that this approach allows for packing more updates even when the number of processing nodes is very large. For instance, on a $64K$ node BlueGene/L system having a $64 \times 32 \times 32$ configuration, the number of unique destinations for each node is only 125, allowing upto 8 updates to be packed in a single packet. The reason for maintaining separate receive buckets for each dimension is to avoid deadlocks. This will be discussed in more detail in Section 5.5.

The pseudo-code for the algorithm is presented in Figures 1 and 2. The main loop is executed until all the updates have been formed and sent.

In the main loop, the algorithm first sends packets that are ready to be sent (corresponding to a full bucket) along each dimension. The algorithm maintains the invariant that there is only at most one full bucket for the routers along a given dimension. This allows a single variable to maintain which bucket is full (if any) along each dimension, avoiding an exhaustive search to determine which buckets need to be sent. Note that the algorithm does not explicitly maintain a pending updates counter, as the sum of the sizes of the buckets is guaranteed not to exceed the limit of 1024 by our choice of bucket capacities.

The algorithm then tries to receive and process available packets. Available packets are received into the receive buffer, corresponding to the dimension from which they arrive. The *routeUpdates* sub-routine is called to process the updates in the receive buffer. If the update is local, it updates the local table, otherwise it inserts the update in the bucket corresponding to the next-hop software router, if possible. The algorithm tries to repeatedly receive packets from the network and process the updates untill either no more packets are available or a received update cannot be inserted into its corresponding bucket.

Finally, the algorithm forms new updates and routes them towards the destination. The update is only formed if no bucket if full. This ensures that the *routeUpdates* subroutine does not fail; it necessarily either updates the table in the local memory or inserts the update in some bucket.

The algorithm uses a lookup table to quickly determine the next-hop router given the destination node for an update. The size of the lookup table is N bytes, where N is the number of processing nodes in the system. Each entry stores a 1 byte index of the corresponding routing node. Additionally, 32 bytes of information (packet header, etc.) are maintained for each destination router. 1K is taken up by the buckets to store updates. Therefore, on a 16K system in a $32 \times 32 \times 16$ configuration, the algorithm requires 16KB for the routing table, less than 3KB for the routing nodes information and 8 KB for the updates. This sums up to 27K, which fits into the 32 KB L1 cache, leaving enough for other temporary storage variables.

5.4 Termination Detection

Note that even if a processing node has finished forming its updates, other nodes may still be generating updates. Therefore, the node has to continue performing the software routing and local updates. Thus, a termination detection mechanism is required so that the nodes can determine when they are not required to perform any more software routing and can therefore terminate.

The *FinishLogic* routine carries out this function. For routing, it executes a loop similar to the main loop. However, it does not wait for the buckets to become full. It flushes out the largest bucket available in each

iteration of the main loop corresponding to which it can send a packet in the corresponding dimension.

For termination detection, it first sends a finish packet to all the routers along its x dimension indicating that it does not have any updates to be routed to them. Note that since the routing is done in dimension ordered manner (first in x dimension, followed by y and z), once a node has finished forming updates, it will never send updates to routers along its x dimension. When a node receives finish packets from all the other nodes along its x dimension, it sends finish packets to all the routers in its y dimension indicating that it does not have any more updates to route to them. Note that the fact that it has received finish packets from all the nodes along the x dimension ensures that it will not receive any updates to be routed to nodes along the y dimension. Similarly, when it receives finish packets from all the nodes along the y dimension, it sends finish packets to nodes along the z dimension. Once a node has received finish packets from all the nodes along the z dimension, it can stop routing and return from the subroutine.

5.5 Preventing Deadlocks

All the updates pending with a node, those generated locally as well as those being routed, must not exceed 1024. This leads to various dependencies amongst the receive buffers (buckets) and send buckets. It is necessary to ensure that no deadlock conditions arise due to these dependencies. We describe below a deadlock situation that may arise when only one receive bucket is maintained on every node. We then describe a scheme for preventing deadlocks by using one receive buffer for every dimension.

Consider the case when there is one common receive buffer, i.e., a node receives the next buffer from the network/system only when it finishes processing all the updates in its receive buffer. Let A and B be two routing nodes along the Y dimension. A deadlock situation is illustrated in figure 3 (a). In this scenario, node A has a non-empty receive buffer such that the next update of the receive buffer meant to be routed to node B. It cannot process this update because the bucket corresponding to destination B is full. This bucket cannot be dispatched, since node B has a non-empty receive buffer. In turn, the next update of node B's receive buffer is meant to be routed via A (note that this receive buffer could have been received from some other node). This update cannot be processed, because B's bucket corresponding to destination A is also full. Node B cannot dispatch this bucket to node A because A's receive buffer is non-empty.

Our algorithm prevents such deadlocks by maintaining 3 receive buckets, one for each dimension for receiving update buffers from routers along that dimension. Figure 3 (b) illustrates dependence of resources. A newly generated update may require a resource in the x-buckets, y-buckets or the z-buckets, depending on the destination of the update. An update in the x-dimension receive buffer (i.e., received from a router along the x-dimension) may require a resource in the y-buckets or z-buckets. Similarly an update in the y-dimension receive buffer may require a resource in the z-buckets. Note that an update in the z-dimension receive buffer does not require any resources, as our algorithm guarantees that it is destined for the current node. Hence, the updates in the z-dimension receive buffer can be updated in the local memory table without blocking. Note that updates in the x and y dimension receive buffers may also be destined for the local memory table. The x-buckets (y-buckets and z-buckets resp.) may be blocked waiting for the x dimension (y dimension and z dimension resp.) receive buffer of the corresponding destination nodes to become free. A deadlock is not possible since there is no cycle in this resource dependency graph. Receive buffer for z dimension cannot be blocked. This will eventually free up the z dimension receive buffers, allowing nodes to send out their z-buckets. This will in turn allow nodes to free up their y and x receive buffer and in turn the y and x buckets of other nodes.

6 Performance Results

6.1 Baseline Code

N	GUPs	Cycles / Update	Mean Bucket Size	SBF
32	0.022	1018	62.88	0.00085
64	0.041	1093	32.42	0.00212
128	0.071	1262	16.83	0.00234
256	0.122	1469	8.93	0.00027
512	0.190	1886	4.97	0.00070
1024	0.290	2472	2.97	-
2048	0.450	3186	1.98	-
4096	0.680	4216	1.33	-
8192	1.080	5310	1.14	-
16384	1.274	9002	1.07	-
65536	0.065	705772	1.02	-

Table 6: MPIRandomaccess performance results on BlueGene/L for the baseline code

Table 6 shows the performance of baseline code for different number of nodes. For 32 nodes, the performance of 0.022 GUPS translates into an average of 1018 cycles per update. The corresponding figure for sequential Randomaccess is 104. Therefore, the overhead introduced by MPI calls and bucketing code amounts to 914 cycles per update. This overhead increases to 8900 for $N = 16382$.

The implementation uses asynchronous MPI calls (`MPI_Isend, MPI_Irecv, MPI_Test`) for communication. A call to `MPI_Isend` is made only if `MPI_Test` returns success, indicating that the earlier `MPI_Isend` has completed successfully. Our calculations indicate that the baseline code is limited by the processing required for each update, not by the network bandwidth. Define the *send blocking factor* (SBF) as the ratio of number of times `MPI_Test` reports that the earlier send request has not finished to the number of times it is called. A blocking factor of 0 indicates that a packet sent earlier never blocks the way of subsequent packets. A blocking factor close to 1 indicates a high network congestion.

To verify our hypothesis, we instrumented the code to measure the send blocking factor. Table 6 reconfirms our calculations that the network is not the bottleneck on BlueGene/L for the baseline version of MPIRandomaccess. We were unable to run the instrumented code on large BlueGene/L systems as the time available for experimentation on large system was limited.

We profiled the code using gprof [6]. After the analysis of the code and profiling data, we identified the following factors contributing to the overheads and its growth as the number of nodes increase.

Frequent calls to `MPI_Test`: The code ends up performing at least one `MPI_Test` for each update which adds a significant overhead. When the base code is modified to invoke `MPI_Test` 1 in 60 times (corresponding to the average bucket size), then the performance on 32 node BlueGene/L system improves to 0.0515 GUPS – more than a factor 2 improvement.

Heap Operations: Every remote update is inserted into a heap, which is an expensive operation even if implemented efficiently. The size of the heap is equal to the number of destinations $N-1$. The time taken for each heap operation is $O(\log N)$ which increases with the number of nodes.

Average Bucket Size: The limit on 1024 pending updates gets divided into N buckets on a system with N nodes. Therefore, the average bucket size (i.e. the number of updates sent per packet) decreases with number of nodes. We wrote a simple simulator to measure this for different number of nodes. The results are presented in Table 6. The MPI function call overheads and the packet header overheads get divided by the average number of updates sent in a single MPI call. Therefore, as the number of nodes increase, the overheads per update increase.

Cache Miss Rate: The data structure maintained by the baseline code requires 20 bytes per destination and 16 bytes per update. In addition the MPI data structures also maintain state for each destination. On a 32KB L1 cache of BlueGene/L, 16K is taken up by the 1024 pending updates. Thus, L1 miss rate starts impacting performance of systems with size $N \geq 512$. As N increases, the L1 miss rate increases thereby adding to the average overhead per update.

Observe from Table 6 that for $N = 65536$, the performance of the baseline code becomes significantly worse than that for $N = 16384$. We suspect that this might be due to the working set of the program exceeding the size of the L3 cache. We were unable to carry out any experimentation on the system to analyze this further.

6.2 Optimized Code

N	GUPs	Bytes / Update	Cycles / Update	Bottleneck	
				CPU	Network
32	0.062	9.31	360	0.062	0.6015
128	0.226	9.31	360	0.211	1.2030
512	0.844	9.31	425	0.815	9.6237
1024	1.780	9.31	403	1.600	9.6237
2048	3.300	9.52	434	3.14	18.8235
4096	5.830	10.24	492	6.23	17.5081
8192	10.990	10.92	522	12.25	32.8113
16384	18.030	12.22	636	24.30	58.6473
65536	35.471	15.6	1293	95.97	91.8974

Table 7: MPIRandomaccess performance results on BlueGene/L for optimized code

Table 7 lists the performance of optimized MPIRandomaccess implementation described in Section 5.3. Unlike the baseline code, the optimized code scales very well for $N \leq 16384$. Figure 4 shows a log-log plot of the GUPS performance as a function of the number of nodes.

The software routing logic adds significant overheads to each update. For a 32-node system, the average number of cycles taken by an update is 360. Out of this 104 cycles can be attributed to the generate and update operation (see Section 4.1). As a first order approximation, we attribute the rest to the software routing. On a 4x4x2 configuration, the expected fraction of updates that need to be routed along the X, Y and Z dimension will respectively be $3/4, 3/4$ and $1/2$. Since the routing decision along each of dimensions are independent, the

expected number of software routing hops an update needs to travel is $2(= 3/4 + 3/4 + 1/2)$. Thus the average per-hop routing on an update may be approximated as 128 cycles on a 32 node system.

Assuming that average per-hop routing cost does not increase with the number of nodes, we compute the CPU bottleneck for systems of larger sizes. The results are listed in Table 7. The same data is plotted on a log-log scale in Figure 4. The CPU bottleneck projections match very well with the observed performance of optimized code for $N \leq 16384$. The projected CPU bottleneck is within 15% of the observed performance numbers[1] for $n \leq 8192$. For $N = 16384$, this gap is about 35% and for $N = 65536$ the projected performance is a factor 2.7 of the observed performance.

In the same table, we also list the theoretical network bottleneck (assuming 100% network utilization) calculated using (2). The network bottleneck is also significantly larger than the observed performance at $N = 65536$. To resolve this, we used the virtual-node mode of BlueGene/L on the 16K node system.

Recall that, in virtual-node mode both the processors of a node are available for computations. In this mode, the number of processors available on a 16-rack system is 32768. Our optimized implementation also works in the virtual-node mode. For supporting this, an additional software routing hop was added after the X, Y and Z routing hops. This adds extra routing overhead, but also doubles the number of processors carrying out routing and updates.

We observed about 70% performance improvement in virtual node mode over co-processor mode for $N \leq 8192$, confirming the hypothesis that the benchmark was CPU bound. However, at $N = 16384$, switching to virtual node gave us only a 10% improvement in performance. Moreover the send blocking factor observed was close to 0.01 in co-processor mode and 0.42 in virtual node mode suggesting that the network bottleneck has been hit at system of this size.

For $N = 65536$, the send blocking factor was 0.9, but it is difficult to pinpoint the causes of this performance gap on 64K node Blue Gene/L system primarily due to the limited availability of the system. We suspect that a combination of instantaneous congestion in the network and blocking at individual nodes is the cause of the observed performance gap. However, it will require more experimentation and analysis to verify this.

7 Discussions, Conclusions and Future Work

We analyzed the bottleneck of the HPCC Randomaccess benchmark on the BlueGene/L system. We designed and implemented a novel software routing technique that gave us a significant improvement in the performance of this benchmark. This technique is generic and may be used to improve the performance of this benchmark on any system.

Our experience with analysis and optimization of the HPCC Randomaccess benchmark can be summarized into the following key observations.

- The sequential Randomaccess benchmark measures the random memory access performance reasonably well.
- The baseline performance numbers for MPIRandomaccess are very sensitive to the specifics of the supplied implementation. A single line change in the code can result in more than a factor 2 change in its

[1] Note that for some entries, the observed performance is better than the projected bottleneck. This is not an anomaly because the bottleneck is computed by a first order approximation using the observed performance figures at $N = 32$.

performance. Heap operations and MPI function calls are significant factors influencing its performance. It is hard to characterize the impact of memory subsystem on the performance of the beseline benchmark code.

- Our analysis indicates that network latencies have little impact on the performance of this benchmark.
- The fixed 1024 pending updates limit specified in the benchmark rules allows systems with small number of nodes to amortize their fixed overheads (in packet headers/MPI function calls) more effectively than large systems, giving an illusion of better normalized system performance.
- The performance of any optimized implementation of this benchmark on systems with sub-linear bisection bandwidth is asymptotically limited by the *smallest edge expansion* of the underlying communication network. This limit may actually be achieved on massively parallel systems such as BlueGene/L.

The MPIRandomaccess essentially measures the performance of asynchronous random writes to remote memory. Other characteristic of distributed memory performance, such as performance of random read, synchronous random write are missing. It may also be a good idea to measure the latency as well as the bandwidth of remote memory operations using sequential and strided accesses pattern (in addition to the random access pattern).

The baseline perfomance number of this benchmark is very sensitive to the details of its implementation. Another approach in designing such a benchmark could be to use a well-defined set of remote memory access primitives (such as ARMCI get/put [7]) in the benchmark code. The performance could be reported on a sequence of remote memory operations using these primitives. The system vendors could be requested to supply optimized implementation of these primitives (similar to the way vendor supplied BLAS and MPI libraries may be linked). This would lead to a more objective assessment of system performance and encourage vendors to supply optimized communication library that may also be used by real applications.

References

[1] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue gene/l torus interconnection network. *IBM Journal of Research and Development*, 49:265–276, 2005.

[2] 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.

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

[4] 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.

[5] 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.

[6] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.

[7] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[8] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski. Blue gene/l compute chip: Memory and ethernet subsystem. *IBM Journal of Research and Development*, 49:255–264, 2005.

[9] Randomaccess rules. `http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/`.

[10] D. Shmoys. Cut problems and their applications to divide-andconquer. in d. hochbaum, editor, approximation algorithms for np-hard problems, pages 192–235. pws publishing, 1996., 1996.

[11] Top500 supercomputer sites. `http://www.top500.org`.

```
Algorithm OptimizedRandomAccess

bktsize = 1024/(X_max + Y_max + Z_max)
/* Main loop */
While ( more updates are to be formed )
    /* Send logic */
    For dim = 1 to 3 do
        If ( some bucket is full in dim
                and can send a packet on dim )
            send the bucket along dim to its destination
            free bucket
            pendingUpdates -= bktsize
        End-If
    End-For
    /* Receive logic */
    For dim = 1 to 3 do
        While ( packet available on dim )
            If ( recvbuf[dim] is empty )
                recv packet from dim into recvbuf[dim]
                pendingUpdates += bktsize
            End-If
            routeUpdates( recvbuf[dim] )
            If ( recvbuf[dim] is not empty )
                break out of While-loop
            End-If
        End-While
    End-For
    /* Form new updates logic */
    If ( no bucket is full )
        x = nextUpdate()
        pendingUpdates++
        routeUpdates( x )
    End-If
End-While
FinishLogic()
```

Figure 1: The Optimized Randomaccess Algorithm

```
Subroutine routeUpdates( updateList )

For each update in updateList
   If ( update is for this node )
      Update corresponding table entry in local memory
   Else
      Determine the next hop router for this update
      If ( no bucket is full corresponding to
         the dimension of the router )
         Insert the update in the bucket of the router
         Remove the update from the receive buffer
      Else
         Return
      End-If
   End-If-Else
End-For
```

Figure 2: The RouteUpdates sub-routine

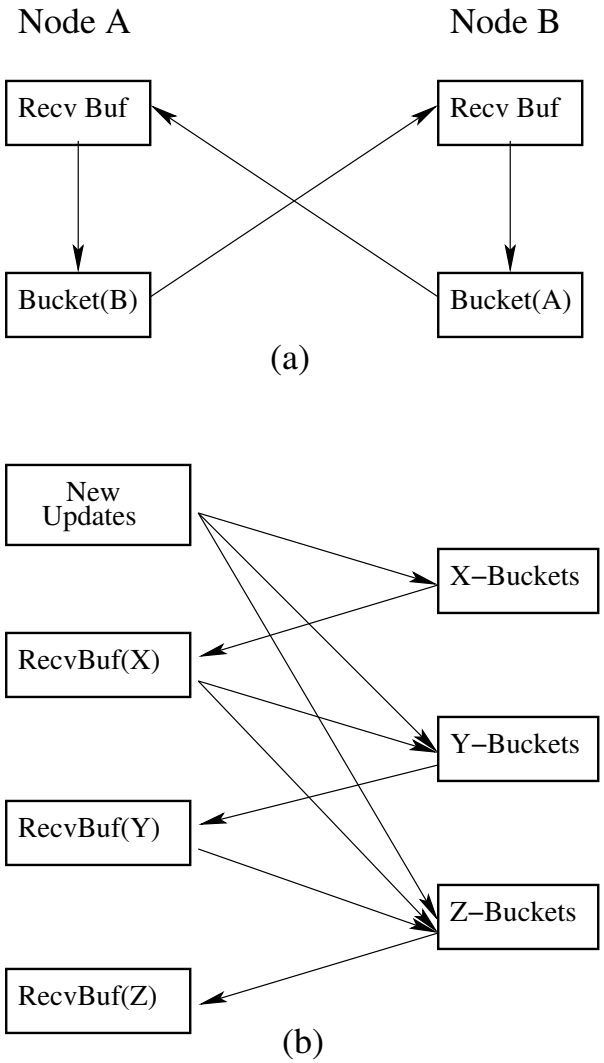

Figure 3: (a) Deadlock scenario with one receive buffer, (b) Preventing deadlocks with multiple receive buffers

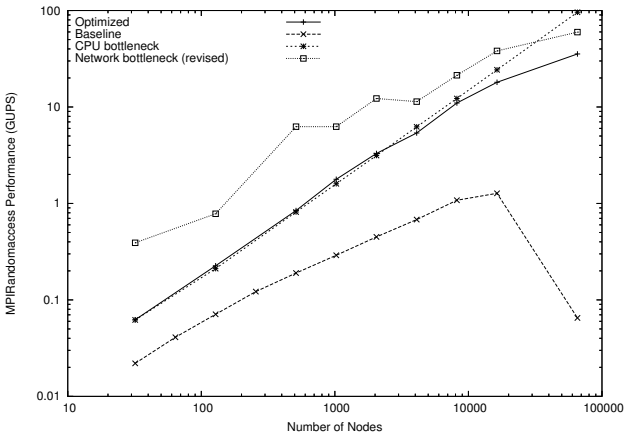

Figure 4: Log-log plot of CPU and network bottlenecks along with the performance of baseline and optimal implementations of MPIRandomaccess.