Software Routing and Aggregation of Messages to Optimize the Performance of HPCC Randomaccess Benchmark

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

Abstract

The HPC Challenge (HPCC) benchmark suite is increasingly being used to evaluate the performance of supercomputers. It augments the traditional LINPACK benchmark by adding six more 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 show that, on many systems, the bisection bandwidth of the network may be the performance bottleneck of this benchmark. We suggest an aggregation and software routing based technique that may be used to optimize this benchmark. We report the performance results obtained using this technique on the Blue Gene/L supercomputer.

1 Introduction

Traditionally the Linpack benchmark [3] has been used to evaluate the performance of supercomputers. This benchmark solves the linear system of equations of the form $Ax = b$ using LU factorization. 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 that include Linpack, matrix multiply, streams, matrix transpose, randomaccess, FFT and communication latency and bandwidth benchmarks.

The HPCC benchmark suite is gaining importance as more and more organizations are using HPCC performance figures to make costly procurement decisions. It is therefore very important to understand the significance of performance numbers reported by these benchmarks and whether they may be used to estimate the true application performance on supercomputing systems.

In this paper, we present a detailed analysis of bottlenecks in the HPCC Randomaccess benchmark, which measures the rate of integer updates to random memory locations. 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* [1] (which is also related to the bisection bandwidth of the network).

We also suggest a technique based on aggregation and software routing that may be used to optimize this benchmark. This technique may be applied to systems where the bisection bandwidth of the communication network grows sublinearly with the system size.

We implemented our technique on the Blue Gene/L supercomputer and obtained significant performance gains. The performance on a 64-rack Blue Gene/L system was 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 the specifics of 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. In Section 2, we describe the HPCC Randomaccess benchmark in more detail. The bottlenecks of this benchmark on are analyzed in Section 3. Our aggregation and software routing technique for improving the benchmark performance are described in Section 4. In Section 5, we give a brief overview of the Blue Gene/L architecture and discuss the bottlenecks of this benchmarks on this system. 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 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 [2]. 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.

Two types of performance numbers may be reported. The *baseline runs* are obtained by compiling and running the supplied code *as is.* 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.

3 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 aggregating a number of updates). The operation *receive* copies packets from the network/system buffer to application buffers. Operation *update* performs the local table update. 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.

3.1 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 $GUPS = 1/(t_g + t_u + t_o^s)$.

Since the local table should occupy approximately half the local memory, 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. In contrast, the operation *generate* can be carried out of cache and is likely to much faster.

Let t_s and t_r represent the average time per update to perform *send* and *receive* operations. 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

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

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

3.2 Network Bottleneck

The *xor* operation is associative, therefore the order in which the updates are performed has no impact on the final result. A node may pipeline and hide the network latency by asynchronously sending the remote update request over the network.

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 equal to the capacity of the corresponding link in the system. 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 amortized header overheads) by the source nodes.

Now consider a vertex in S. The expected number of updates from the vertex to other vertices in $\overline{S}$ in given by $U|\overline{S}|/N$. 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

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

The right hand side of above expression is very closely related to a well-studied problem in graph thoery called the *sparsest cut* [1]. 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. The smallest edge expansion is also related to the bisection bandwidth of the network. If B is the bisection bandwidth, it is easy to see that

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

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 assymptotically 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 $\alpha(G) = O(1)$.

4 Optimizing the Benchmark

In this section we describe a software routing and aggregation technique that may be used to optimize this benchmark. This technique is directly applicable to systems having an underlying grid/torus communication network. Moreover, this technique may be extended to other systems by arranging the processors into a logical grid topology and efficiently mapping this grid to the underlying interconnection network.

In the following discussion, we illustrate our technique for a 3-dimensional torus interconnection network. It is easy to see how this technique may be extended to networks with higher dimensions.

4.1 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. 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 3. 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.

4.2 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.

Consider 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 $X_{max} + Y_{max} + Z_{max} - 3$.

4.3 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. The reason for maintaining separate receive buckets for each dimension is to avoid deadlocks. For a more detailed discussion on the deadlock situations that arise and how this strategy prevents deadlocks, see [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 a 32 KB L1 cache, leaving enough for other temporary storage variables.

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. For a more detailed discussion, see [5].

```
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 next hop router for this update
        If ( no bucket is full corresponding to
            the dimension of the router )
            Insert update in the bucket of the router
            Remove update from the receive buffer
        Else
            Return
        End-If
    End-If-Else
End-For
```

Figure 2: The RouteUpdates sub-routine

5 Bottleneck on the Blue Gene/L Supercomputer

In this section, we start by giving a brief overview of the Blue Gene/L Supercomputer. We then describe our experiments for estimating various parameters used in inequalities (1) and (2) to determine the performance bottleneck.

5.1 Blue Gene/L Overview

The Blue Gene/L is a massively parallel supercomputer that scales upto 65,536 dual-processor nodes [4]. The nodes themselves are physically small, allowing for very high packaging density in order to realize optimum cost-performance ratio.

Each node has two embedded 770 MHz PPC440 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 node has 32-KB L1 instruction and data caches and a 4-MB embedded DRAM L3 cache. The latencies for an L1 cache, L3 cache and main memory access are 3 cycles, 28-40 cycles and 86 cycles respectively. Each node has 512 MB of physical memory which is shared by its two processors.

The Blue Gene/L uses five interconnect networks for I/O, debug, and various types of interprocessor communication. 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. Each node supports six independent 1.4 Gbps bidirectional nearest neighbor links, with an aggregate bandwidth of 2.1 GB/s. The torus network uses both dynamic (adaptive) and deterministic routing with virtual buffering and cut-through capability. 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 32-bit trailer is appended to each packet that includes a 24-bit CRC.

5.2 Measuring Blue Gene/L Specific Parameters

On the Blue Gene/L system, the latency to access main memory is 86 cycles [8]. The observed performance of sequential Randomaccess on Blue Gene/L is 0.0067 GUPS which translates into an average value of 104 cycles for $t_g + t_u + t_o^s$ (see (1)). Thus $t_g + t_o^s$ is only 18 cycles.

Since the network in Blue Gene/L is a 3-dimensional torus, it is expected to limit the performance of MPIRandomaccess when the number of nodes is very large (see inequalities (1) and (2)). However, to determine the bottlenecks, we needs to estimate the values of the parameters $t_g, t_u, t_o^s, t_s, t_r, t_o^p, b, \alpha(G)$ used in these equations.

With the bucketing technique of the baseline code, the average number of updates per packet approaches 1 as the system size increases. Therefore, we estimate the above parameters assuming each packet carries a single update.

Our measurements on Blue Gene/L indicated MPI function call latencies between 1270 and 12,000 cycles for payload sizes between 8 and 1024 bytes. This was unacceptably high for this benchmark. We therefore decided to use the raw device interface for communication.

Performance of the raw device interface is presented in Table 1. 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,

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

N	Dimension	$\alpha(G)$	Bottleneck	
	(XxYxZ)	/ C[1]	CPU	N/W
32	4x4x2	1/2	0.13	0.12
64	8x4x2	1/4	0.25	0.12
128	8x4x4	1/4	0.51	0.24
256	8x4x8	1/4	1.03	0.48
512	8x8x8	1/2	2.06	1.95
1024	8x8x16	1/4	4.11	1.95
2048	16x8x16	1/4	8.23	3.89
4096	8x32x16	1/8	16.47	3.89
8192	16x32x16	1/8	32.94	7.79
16384	32x32x16	1/8	65.88	15.58
32768	32x32x32	1/8	131.77	31.16
65536	64x32x32	1/16	263.53	31.16

Table 2: CPU and network bottleneck GUPS for Blue Gene/L using raw network device

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 $b = 32 + 14 = 44$.

In order to compute the network bottleneck, we need to know its sparsest cut. Finding the sparsest cut for general graphs is a NP-hard problem [9]. However, for Blue Gene/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 Blue Gene/L partition of different sizes and shapes.

It may also be observed that the bound obtained in (2) is tight for any regular d-dimensional torus network (including the Blue Gene/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 Blue Gene/L.

Table 2 also lists the CPU and network bottlenecks (in units of GUPS) for the Blue Gene/L system assuming send/recv latencies of Table 1 and $b = 44$ bytes. With these assumptions, it is apparent that the network bandwidth is the bottleneck for Blue Gene/L systems of all sizes.

Fundamentally, for each update only 8 bytes need to be communicated while an average of 44 bytes per update are communicated if we send only one update per packet. Therefore our aggregation and software routing technique should lead to significant performance improvements. Since the primary communication network in Blue Gene/L is a 3-dimensional torus, the logical grid topology in our algorithm can be the same as the physical topology.

The $64K$ node Blue Gene/L system has a $64 \times 32 \times 32$ configuration. Thus, using our technique, the number of unique destinations for updates from each node is 125, allowing upto 8 updates to be packed in a single packet.

In the next section, we describe the performance results of the baseline code and our optimized implementation on Blue Gene/L.

6 Performance Results

6.1 Baseline Code

Table 3 shows the performance of baseline code for different number of nodes. For 32 nodes, the

[1]Note that upto 256 nodes, the network configuration is a 3D-Mesh and not a 3D-Torus. The edge expansions are calculated accordingly.

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 3: MPIRandomaccess performance results on Blue Gene/L for the baseline code

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 cycles 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 3 reconfirms our calculations that the network is not the bottleneck on Blue Gene/L for the baseline version of MPIRandomaccess. We were unable to run the instrumented code on large Blue Gene/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 Blue Gene/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 3. 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 Blue Gene/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 3 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	Bottleneck N/W
32	0.06	9.31	360	0.06	0.60
128	0.22	9.31	360	0.21	1.20
512	0.84	9.31	425	0.81	9.62
1024	1.78	9.31	403	1.60	9.62
2048	3.30	9.52	434	3.14	18.82
4096	5.83	10.24	492	6.23	17.50
8192	10.99	10.92	522	12.25	32.81
16384	18.03	12.22	636	24.30	58.64
65536	35.47	15.6	1293	95.97	91.89

Table 4: MPIRandomaccess performance results on Blue Gene/L for optimized code

Table 4 lists the performance of optimized MPIRandomaccess implementation described in Section 4.3. Unlike the baseline code, the optimized code scales very well for $N \leq 16384$. Figure 3 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 5.2). 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

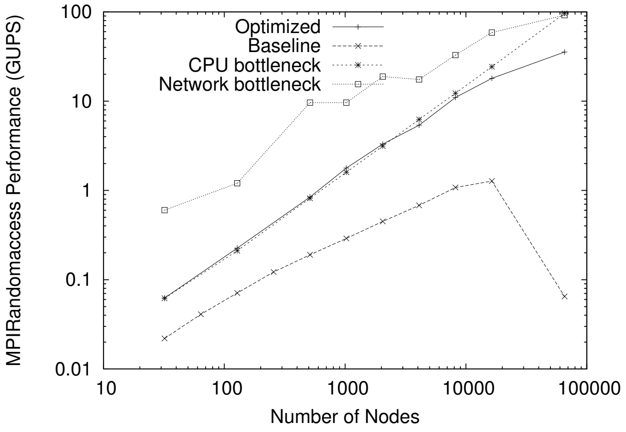

Figure 3: Log-log plot of CPU and network bottlenecks along with the performance of baseline and optimal implementations of MPIRandomaccess.

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 overhead 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 4. The same data is plotted on a log-log scale in Figure 3. 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[2] 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

[2]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 $N = 32$.

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 Blue Gene/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 fundamental bottlenecks in the HPCC Randomaccess benchmark and suggested a novel aggregation and software routing technique that leads to significant performance improvements. On the Blue Gene/L supercomputer, this technique gave the best-known performance of 35.5 GUPs. In the process of analysis of this benchmark and its implementation on the Blue Gene/L supercomputer, we gained the following insight.

- The sequential Randomaccess benchmark measures the random memory access performance reasonably well.
- 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 systems such as Blue Gene/L.
- 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 baseline performance numbers for MPI-Randomaccess 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 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.

This benchmark can be improved in the following ways. Currently, it only measures the performance of asynchronous random writes to remote memory. It is desirable to include other characteristic of distributed memory performance, such as random read, synchronous random write, latency and bandwidth of sequential and strided accesses patterns.

Since the baseline perfomance number of this benchmark is very sensitive to the details of its implementation, it is desirable to use a well-defined set of remote memory access primitives (such as ARMCI get/put [7]) in the benchmark code. 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.

8 Acknowledgements

We would like to thank several people who helped us with this work in many ways. Without their valuable help, this work could not have been accomplished. We thank John A. Gunnels, who closely worked with us on this and other HPC Challenge benchmarks, Tom Spelce, who got us time on the 64 rack Blue Gene/L system and carried out many runs for us on this system, Gheorghe Almasi and Charles Archer, for sharing their knowledge and experience of MPI on Blue Gene/L, Phil Heidelberg, for help with the raw device interface on Blue Gene/L and Manish Gupta, for providing over all support for this activity.

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.

[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] 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.