HPCC RandomAccess Benchmark for Next Generation Supercomputers

Vikas Aggarwal, Yogish Sabharwal and Rahul Garg
IBM India Research Lab
Plot 4, Block C, Vasant Kunj Inst. Area
New Delhi 110070, India.
Email: vicky.aggarwal@gmail.com,
ysabharwal@in.ibm.com, grahul@us.ibm.com

Philip Heidelberger
IBM T. J. Watson Research Center
1101 Kitchawan Rd, Rt. 134
Yorktown Heights, NY 10598, USA.
Email: philiph@us.ibm.com

Abstract

In this paper we examine the key elements determining the performance of the HPC Challenge RandomAccess benchmark on next generation supercomputers. We find that the performance of this benchmark is closely related to the bisection bandwidth of the underlying communication network, performance of integer divide operation and details of benchmark specifications such as error tolerance and permissible multi-core mapping strategies. We demonstrate that seemingly small and innocuous changes in the benchmark can lead to significantly different system performance. We also present an algorithm to optimize RandomAccess benchmark for multi-core systems. Our algorithm uses aggregation and software routing and balances the load on the cores by specializing each of the cores for one specific routing or update function. This algorithm gives approximately a factor of 3 speedup on the Blue Gene/P system which is based on quad-core nodes.

1. Introduction

Future supercomputing systems are moving towards hybrid architectures comprising massively distributed systems with multi-core nodes. For example, the Blue Gene/P supercomputer [1] has 4 cores per node, RoadRunner [2] has a nine-core STI Cell chip per Opteron core, SGI Altix ICE [3], [4] has dual-core/quad-core Intel Xeon processors. This trend, primarily driven by increasing power consumption and the diminishing gains in processor performance from increasing operating frequency is also reflected in desktop and server processors [5], [6]. Due to cost considerations, the systems may be upgraded incrementally and the number of nodes in such systems may not be a perfect power of two. It is important to understand how these trends impact the performance of applications and benchmarks used to evaluate these systems.

The HPC challenge (HPCC) benchmark suite is becoming increasingly popular for evaluating the performance of supercomputers. In augments the traditional high-performance LINPACK (HPL) benchmark [7] by measuring additional aspects of system performance involving memory and network. One of the important benchmarks in this suite is the RandomAccess benchmark that measures the peak capacity of the memory subsystem while performing random updates to the system memory. In an earlier paper, [8], we showed that the HPCC RandomAccess benchmark measures the bisection bandwidth of the interconnection network, if the bisection bandwidth scales sub-linearly with the number of nodes. However, in case of shared-memory multi-core systems or in systems where the number of nodes is not a perfect power of two, the situation is different.

On multi-core systems, a parallel application may use the multi-core nodes either as shared-memory multiprocessors or as distributed-memory uniprocessors. In shared-memory mode the threads of an application have access to most of the node's memory. In the distributed-memory mode, the node's memory is partitioned and threads have access to only a fraction of the total node memory. There are eight modes in which the RandomAccess benchmark can be mapped to the cores of a multi-core node. The performance of this benchmark depends on the choice of the mapping mode and the optimization strategy.

In this paper we show how the multi-core mapping mode, cost of integer divide operation and error tolerance can influence the performance of the RandomAccess benchmark. Achieving high performance while guaranteeing zero error rate in the shared memory multi-core nodes is difficult because of the shared-memory and locking overheads. We describe a novel load-balancing technique that may be used along with the aggregation and software routing method [8] on multi-core systems. This technique specializes each of the cores for specific routing and update functions. This ensures that all the cores are kept busy and the load is approximately balanced. This also guarantees zero error rate in multi-core systems without having to use locks or other synchronization primitives. On quad-core Blue Gene/P system this technique gives close to a factor 3 speedup in comparison to the single core algorithm. Using this technique, we obtain a performance of 103 GUPs on a 32K node Blue Gene/P system – three times more than the currently best reported performance.

In Section 2 we give a brief introduction to the HPCC RandomAccess benchmark. In Section 3 we discuss the key elements determining the performance of the RandomAccess benchmark on massively parallel multi-core architectures. In Section 4, we present a load-balancing technique for the optimization of this benchmark on multi-core supercomputing systems. In Section 5 we present the performance results on the Blue Gene/P system followed by a brief summary and conclusion in Section 6.

2. The RandomAccess benchmark

The RandomAccess benchmark is motivated by a growing gap in performance between processor operations and random memory accesses. 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 large distributed table T of size 2^k, occupying approximately half of the total system memory. Each processor generates a pseudo-random sequence of 64 bit integers. For each random number (say a_i), the most significant bits are selected to index into the distributed table T. The selected entry in the table (which mostly resides on a remote node) is updated using a bit-wise *xor* with the random number a_i. The number of such updates performed by each node is four times the local table size. The benchmark specifications allow each node to look-ahead and store at most 1024 updates before they are applied to the table. The performance of the system is measured by the number of *giga updates per second* (GUPS) performed by the system.

The RandomAccess benchmark has undergone several improvements since its inception in the HPC Challenge benchmarks. In the original version of the benchmark, the initialization of the local tables on every node was also included in the performance timings. Moreover, the benchmark overestimated the time required to perform updates resulting in very small number of updates especially on large systems. As a result, the initialization section of the benchmark could end up taking a significant fraction of time leading to significantly poor performance figures. The latest release of the HPC Challenge benchmarks (version 1.2.0) has now been modified to exclude table initialization time in final performance measurements and also to perform a large number of updates (four times the local table size). These changes alone have boosted the performance of the 64 rack Blue Gene/L system to 75 GUPS from earlier reported performance of 35.5 GUPS. This also explains the gaps in the theoretically estimated and measured performance reported in [8].

Before we delve deeper, it is important to consider applications that might have motivated this benchmark. Two applications that resemble this benchmark are distributed in-memory databases [9], [10] and large distributed hash tables [11]. In these applications, the database or the hash table is very large and therefore partitioned into the local memory of a large number of nodes. Each node also gets a random sequence of requests to read or update entries (which are usually small). To service the update requests the nodes need to communicate with other nodes in a way similar to the MPIRandomaccess benchmark. While discussing the issues of multi-core mapping mode and error tolerance, one must keep similar applications in mind and examine if the performance of this benchmark on a system will truly reflect the performance of such applications.

3. Issues with MPI RandomAccess benchmark

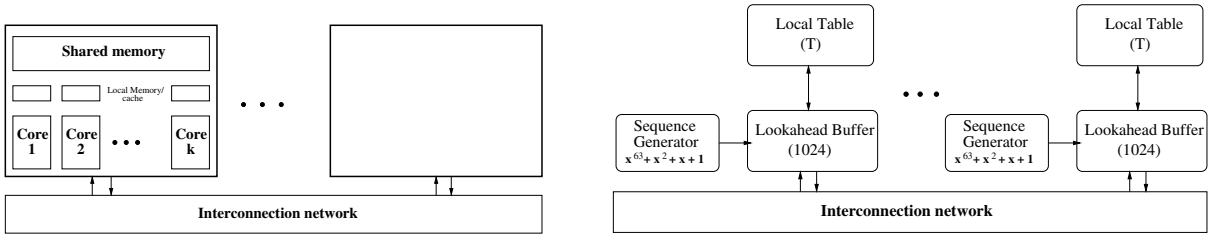
Next generation supercomputers are moving towards hybrid architectures based on large numbers of multi-core nodes [1] comprising large number of cores [6] at every node. Such systems are typically modular and allow the users to upgrade the sizes incrementally. The bisection bandwidth of the network in these systems may not scale linearly with the number of nodes. This poses several challenges in optimizing as well as interpreting the performance results reported by this benchmark. In this section, we identify the key elements that influence the performance of this benchmark on next generation supercomputers.

3.1. Bottleneck analysis

It was shown in [8] that the GUPS performance of a any single-core system is bounded by the following:

$$GUPS \leq Min(\frac{N}{t_g + t_u + t_s^o + \frac{(N-1)}{N}(t_s + t_r + t_p^o)}, \frac{4B}{b}) \quad (1)$$

where the first term in the parenthesis corresponds to CPU bottleneck and the second term corresponds to network bottleneck. Here N represents the number of processors in the system, t_g and t_u represent the average time for performing a *generate* and an *update* operation respectively, t_s^o represents the loop overheads, t_s and t_r represent the average send and receive time per update, t_p^o corresponds to additional overhead involved in performing the send/receive operations, B is the bisection bandwidth of the system, and b is the average number of bytes (amortized) sent per update over the network. It is evident that on systems where the bisection bandwidth does not increase linearly with the number of nodes, the performance bottleneck will eventually be determined by the bisection bandwidth B and the average number of bytes b needed to send an update over the network.



(a) A massively parallel multi-core system (b) The MPI RandomAccess benchmark

Figure 1. Logical views

3.2. Multi-Core mapping modes

A typical parallel multi-core system architecture is shown in Figure 1(a). It consists of large number of multi-core nodes connected by a high speed interconnection network. Each multi-core node has a large amount of memory accessible to each of its core. This memory could be organized into memory-banks with core affinity (such as in NUMA systems [12]) or could be designed to provide a truly uniform access to each of the cores [1], [13]. Besides this, each core typically has its own cache which, depending on the design (or mode of operation) may be coherent or incoherent with other cores. In addition, each core may have a private local memory accessible only to itself. The sizes of private memory, cache and shared memory (as well as their coherence semantics) may vary across different systems, but the overall architecture remains mostly the same [5], [13], [14].

Fig 1(b) shows a logical view of the random-access benchmark. It consists of a large globally distributed table split equally across multiple nodes, a random sequence generator associated with each node and an application look-ahead buffer (see Section 2) of size 1024 which may be used to store table updates to be sent to (or received from) other nodes.

The *multi-core mapping mode* determines how the various components of this benchmark are mapped on the different cores of a multi-core system. A simple mapping of the RandomAccess benchmark to a multi-core system is shown in Figure 2(a). Here, each core has its own private table, its own random update sequence generator and its own look-ahead buffer of size 1024. As the multi-core systems continue to evolve, they may have more than 100's of cores at each node [6]. As a consequence, the amount of memory per core may be much smaller as compared to the total memory available at a node. Partitioning node's memory into disjoint private memory for each core is not likely to be the preferred mode of operation for many real applications.

The other extreme, as illustrated in Figure 2 (c), has a single large table at every node shared across its cores. In this case, there is only one sequence of random updates generated by a node and there is a single look-ahead buffer of size 1024 at each node. Under such a mapping, only one of the cores of a node may generate the random sequence of table updates. Another way to map this benchmark to a multi-core system is shown in Figure 2 (b). Here, the table is shard across the cores, but each core generates its own sequence of updates and has its own look-ahead buffer of size 1024.

In general, there are eight possible mapping of this benchmark on a multi-core system, depending on whether the table, the random sequence generation or the look-ahead buffer limit of 1024 is applied on a per-core or per-node basis. These mapping modes are listed in Table 1. Some of these modes may not be meaningful – for instance, if the table is defined at the core level, then the generation of updates and the 1024 lookahead limit should both be applicable at the core level and not the node level. This rules out combinations 2, 3 and 4.

The present specifications of the benchmark allow mapping modes 1 (1 MPI process per core), 6 and 8 (one MPI process per node). However, a real application such as a distributed in-memory memory database system [9] or distributed hash tables [11] may require other mapping (such as mode 5) for optimal performance. The performance of this benchmark critically depends on the mapping mode and it is important to carefully study the impact of these mapping modes on system performance.

3.3. Error tolerance

The specifications permit up to 1% errors in the final table computed by the benchmark. There are two potential sources of such errors. Firstly, if the cache of different cores are not

Mode	1	2	3	4	5	6	7	8
Local table	C	C	C	C	N	N	N	N
Generation	C	C	N	N	C	C	N	N
1024 limit	C	N	C	N	C	N	C	N

Table 1. Mapping modes: $C \Rightarrow$ one per core; $N \Rightarrow$ one per node

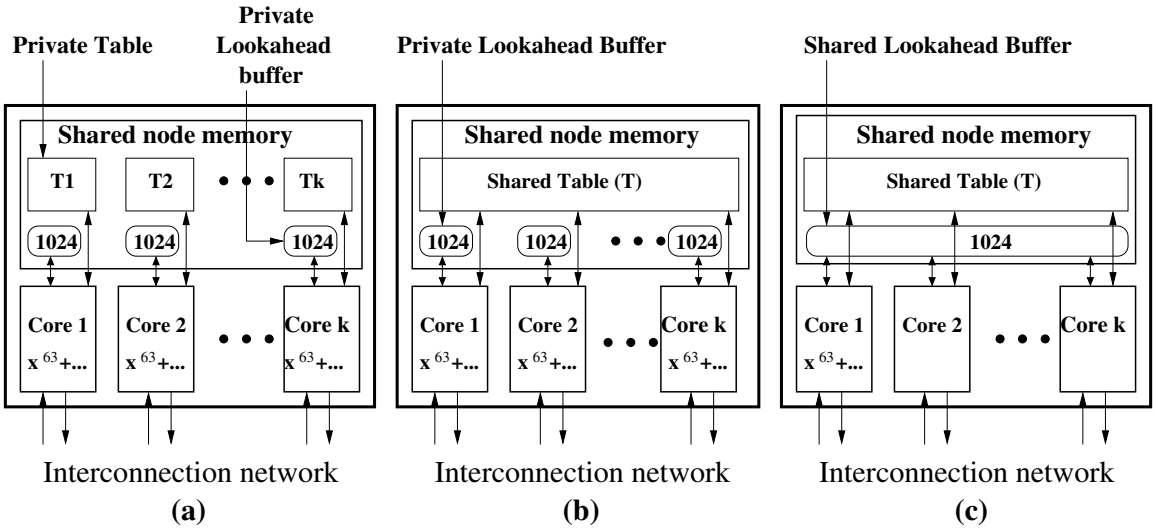


Figure 2. Mapping modes -(a) Mode 1, (b) Mode 5, (c) Mode 8

coherent, then update performed by one core may just sit in its cache for a long time and not be seen by other core performing the update of the same table location.

Secondly, the xor operation consists of a read-modify-update sequence which is not atomic. If two cores read the same table location at exactly the same time, the update done by the core that writes back the result to the memory first will be overwritten by the result of the other core. Since the modify-write operation is typically very fast the probability of such a race condition is extremely small, but non-zero. As the number of cores in a node increase, this probability is expected to increase.

The number of errors observed while performing (pseudo) random updates to the shared table is a random variable. Table 2(b) shows the ratio of the standard deviation to the mean error rate for random updates to a table of size 16M entries (128 MB). As the number of updates is increased, this ratio decreases due to the law of large numbers. When the number of updates is four times the table size (as required by the benchmark) this ratio is 0.45. Therefore, one may expect a significant variation in error rates for different runs of the benchmark (in one run it may be less than 1% while in another it may be more than 1%).

Obtaining *zero error guarantee* in multi-core systems can be expensive (in cost or performance). It requires either transactional memory or mutex locking. We performed some experiments to determine the effect of locking using mutexes on Blue Gene/P. The results are given in Table 2(b). Utilizing all the four cores to carry out updates to the table results in only a factor 2.58 performance improvement. However, if a single lock is used to ensure consistency, the performance degrades by a factor 1.2 (= 1/0.83). The relative performance improvement while using 16 or 32 locks is only 1.83 and 1.87 respectively.

In the current generation multi-core systems, the cache coherency and locking overheads are not exorbitant and the error rates are well below the permitted 1% limit. However, in future systems with many more cores, the cache coherency and locking overheads will be higher. This may provide strong incentives to avoid locking and work in a "cache-incoherent mode" while running this benchmark. This may lead to error rates that are comparable to the 1% limit. Since the error rate is a random variable the definition of a 1% error tolerance becomes ambiguous. It is not clear if the error should be less than the prescribed limit in "every possible run" of the benchmark, "one of the run" or "on the average". Moreover, it is impossible to provably verify if an implementation meets the required tolerance with any of the definitions above.

3.4. When number of nodes is not a power of two

The global table maintained by the RandomAccess benchmark has a size, that is a power of two. This makes it easy to determine the global offset for a table entry from a randomly generated number by using simple bit-arithmetic. When the number of nodes is not a power of two, computing the node id and the offset into table requires an integer division operation. On many architectures, integer divides can take 1-2 orders of magnitude more time than other operations. This can significantly impact the performance of systems. Therefore adding more nodes to the system (when it already has power-of-two nodes) can significantly degrade the performance of this benchmark.

Num Updates	Mean error rate (10^{-6})	Std. deviation (10^{-6})	Std./ mean
32M	2.92	1.49	0.51
64M	3.46	1.55	0.45
128M	6.14	2.50	0.41
256M	8.52	1.91	0.22
512M	19.13	3.27	0.17

Benchmark	Relative GUPS performance
Serial – 1 core	1
Shared – 4 cores	2.58
Shared – 4 cores, 1 lock	0.83
Shared – 4 cores, 16 locks	1.83
Shared – 4 cores, 32 locks	1.87

Table 2. (a) Error rates for different number of updates, (b) Relative performance of serial benchmark with different locking strategies

4. Optimization on the Blue Gene/P system

We determined the bottlenecks for different sizes of the Blue Gene/P system using Eq.(1). For small system sizes, the CPU is the bottleneck and for large system sizes, the network is the bottleneck. In co-processor mode, we determined that the crossover happens beyond 256K nodes and therefore will not happen for a real Blue Gene/P system. However, for virtual-node mode, the crossover happens at 8K nodes. This warrants the use of optimization techniques based on aggregation and software routing presented in [8] that alleviates this problem by packing more updates in a packet, thereby reducing b.

We also wanted to achieve a guaranteed zero error rate. This would require the use of locks if all the cores were to simultaneously update the shared table (i.e. mapping modes 5 or 6 were used). Our measurements indicated significant overheads in using locks (see Table 2). Moreover, the use of the aggregation technique requires additional CPU overheads in carrying out the software routing of updates. We did not anticipate any performance gains of multiple cores by using the mapping modes 5 or 6. The only realistic choices left were modes 1 and 8.

The best random-access performance was expected in mapping mode 1. In this mode all the cores are perfectly load balanced, there are no cache consistency overheads, and the 1024 lookahead buffer is available to every core. Moreover, the table on a core can be allocated on a bank that has affinity to the core (in case of NUMA access). Lastly, inter-node traffic is expected to dominate the inter-core traffic. However, this mode is not representative of the way real applications may be run on such systems. So we decided to use the mapping mode 8 for this benchmark.

Load-balancing among the cores was the most important issue in this mode. Since there is only one sequence of random numbers per node, three out of four cores may not have any work. We specialized each of the four cores of the Blue Gene/P nodes for specific routing, generation, send or update operations. This ensured that all the cores are kept busy and load is approximately balanced. Since only one of the four cores performed local table update, use of locks was also avoided.

The details of our load-balancing scheme is given in the remainder of this section. We start by giving a brief description of the Blue Gene/P system and the aggregation and software routing technique [8]. We then describe our multi-core algorithm for the Blue Gene/P system in more detail. Finally, we describe our technique to handle the case when the number of nodes is not a power of two.

4.1. Blue Gene/P system overview

The Blue Gene/P [1] is IBM's next-generation massively-parallel supercomputers which has evolved from the Blue Gene/L architecture. Each node in a Blue Gene/P system consists of four 850 MHz PowerPC 450 processor core and 2GB of physical memory. Each node has 32-KB L1 instruction and data caches. An 8-MB embedded DRAM L3 cache is shared by all the cores. The latencies for an L1 cache, L3 cache and main memory access are of the order of 3 cycles, 50 cycles and 104 cycles respectively.

The nodes are interconnected through five networks, the most important one of which connects the nearest neighbors into a 3-dimensional torus. The torus network handles the bulk of the communication data from an application and offers the highest bandwidth in the system. Each node supports 850 MB/s bidirectional links to each of its nearest neighbors for a total of 5.1GB/s bidirectional bandwidth per node.

Injection and reception Direct Memory Access (DMA) first come first serve (FIFO) buffers are provided for transferring packets to and from the network (torus) buffers and between the cores. Blue Gene/P provides low level APIs, Deep Computing Messaging Framework (DCMF) [15] for generalized message passing. The messaging is based on variable size packets, which are multiples of 32 bytes in size with a maximum packet size of 256 bytes. The messages carry a hardware header and software header of 8 bytes each, resulting in a total header of 16 bytes. Thus there can be a maximum of 240 bytes of payload per packet. There is an additional 14 byte overhead per packet due to acknowledgments. Our experiments with the raw device interface indicated that send and receive function call latencies are fairly constant (150 and 600 cycles respectively) over

different packet sizes. These numbers are more than the Blue Gene/L send/receive times (which range from 14-70 cycles for send and 50-190 cycles for receives depending on the packet size) primarily due to the DMA overheads. The Blue Gene/P send/receive times are not dependent on data size as they are DMA based [1].

4.2. The aggregation & software routing algorithm

In this approach [8], the updates for a group of destination nodes are aggregated and sent to intermediate processing node, called routing nodes. An update is routed along the nodes of the 3D torus of the Blue Gene interconnect in dimension order. As an example, consider an update that is generated on node $\langle x_i, y_i, z_i \rangle$ for a table entry that resides on node $\langle x_j, y_j, z_j \rangle$. This update is first sent along the x dimension of the 3D-torus to the node $\langle x_j, y_i, z_i \rangle$. It is then routed along the y dimension to node $\langle x_j, y_j, z_i \rangle$ and finally along the z dimension to the destination node $\langle x_j, y_j, z_j \rangle$. This is illustrated in Figure 3(a). As far as possible (constrained by the 1024 look-ahead limit), updates that are traveling in the same direction are aggregated and sent together. For instance, the originating node $\langle x_i, y_i, z_i \rangle$ clubs together updates that are destined for nodes with x co-ordinates equal to x_j and sends them to $\langle x_j, y_i, z_i \rangle$. Similarly, the routing nodes aggregate updates received from other nodes traveling in the same direction. This can be generalized to any number of dimensions [8]. Thus in a k-dimensional grid with N^d nodes, each node needs to send packets only to $d(N-1)$ other nodes (i.e. only to nodes along each of its axes) as opposed to $N^d - 1$ nodes. Thus, $1024/(dN)$ updates can be aggregated in a single function call (or a packet) amortizing the overheads by the same factor. The aggregation and software routing approach has also been used to optimize performance in other domains such as short-message all-to-all communication [16] and FFT computation [17].

4.3. Load balancing on Blue Gene/P cores

Figure 3(b) presents a functional block diagram representing the work-load distribution across the cores. *Core 0* is responsible for generation of updates and routing the updates that need to be sent along the X dimension. *Core 1* is responsible for receiving updates along the X dimension and routing the updates along the Y dimension. *Core 2* is responsible for receiving updates along the Y dimension and routing the updates along the Z dimension. *Core 3* is responsible for receiving updates along the Z dimension (i.e. updates destined for local node) and performing the local table updates. In addition, each core sends and receives messages to other three cores through local FIFO buffers.

To understand the routing path traversed by an update, consider an update from source node $i = \langle x_i, y_i, z_i \rangle$ to destination node $j = \langle x_j, y_j, z_j \rangle$. It is routed along the x-dimension by *core 0* on the source node. *Core 1* on intermediate routing-node $\langle x_j, y_i, z_i \rangle$ receives the update and routes the update on y-dimension to the *core 2* on routing-node $\langle x_j, y_j, z_i \rangle$, which in turn routes the update along z-dimension to *core 3* on the destination node. *Core 3* on the destination node applies the update to the memory. Moreover, *core 3* only receives updates destined for the local node and hence does not perform any routing functions.

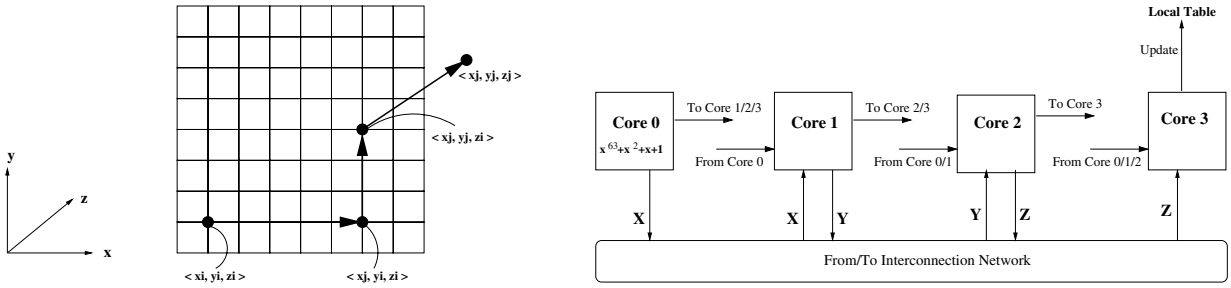
If some co-ordinate for source and destination nodes is same, the routing hop for the corresponding dimension is handled by transferring the update to the core (on the same node) which handles communication corresponding to the update's next hop over the network. In our previous example, if node i and j had the same coordinate on x-dimension and different y and z dimensions, *core 0* on the source node will send the update to *core 1* on its local node instead of sending the data over the network. The communication of updates between any two cores on a local node is performed by exchanging packets over local FIFO buffers. The update, then follows its normal path from *core 1* on the source node to *core 3* on the destination node along the y and z dimensions of the torus. All updates in transit between the cores on local buffers are accounted within the limit of 1024 updates.

This simple partitioning scheme has several advantages:

- It distributes the routing workload over the different cores. All the cores are involved in communication, i.e., the send and receive operations are well distributed amongst the cores. This leads to good load-balance between the nodes (as indicated by our experiments in section 5.3).
- Since the routing logic for any dimension is independent of other dimensions, each core can maintain their individual update buckets and routing information without having to interact with other cores. This reduces the synchronization overheads, that could have potential performance impact, to a minimal.
- Only one core updates the memory therefore ensuring memory consistency This removes need for synchronization amongst the cores for ensuring consistency.

4.3.1. Update Buckets. The updates are transferred between nodes in units of buckets. The number of updates that can be packed in a bucket is determined by the total number of buckets that must be maintained by each node. The 1024 update limit is distributed equally amongst the total number of buckets. The buckets maintained by each node consist of

1) *Buckets consisting of updates to be sent to other nodes.* The total number of nodes that each node communicates with is $X_{max} + Y_{max} + Z_{max} - 3$. Therefore these many buckets are required for sending updates to remote nodes.



(a) (b)

Figure 3. (a) Illustration of the routing path for an update, (b) Function block diagram demonstrating the load distribution across the BG/P quad-cores

2) *Buckets consisting of the updates received from other nodes which have not been fully processed yet.* As described in Section 4.1, the latency of send/receive function calls are high on Blue Gene/P due to memory related synchronizations related to DMA setup. These latencies are better amortized when more than one packet are sent/received together. Our experiments demonstrate a reduction in average receive function call latency per packet from 600 cycles to 330 cycles by employing the above technique when four packets are available. Therefore, we receive up to 4 packets from the memory buffers in a single receive operation. As a result, the algorithm now requires 12 additional buckets for receiving updates from remote nodes (four for each dimension).
3) *Buckets consisting of updates to be sent/received to/from other cores on the same node.* Since the proportion of updates traversing the local buffers is low on large systems, only one bucket is maintained for each other core. Core 0 may send updates to cores $1, 2$ and 3, core 1 may send updates to cores 2 and 3 and core 2 may send updates to core 3. Thus, there are 6 communicating core pairs on every node, and therefore 6 buckets are required for sending inter-core updates. Similarly 6 buckets are maintained (one for each communicating pair) for receiving of local updates from other cores.

Therefore, the total number of buckets maintained on each processing node is

$$\begin{aligned} NumBuckets &= X_{max} + Y_{max} + Z_{max} \\ &\quad -3 + 12 + 6 + 6 \\ &= X_{max} + Y_{max} + Z_{max} + 21 \end{aligned}$$

where X_{max}, Y_{max} and Z_{max} represent the dimensions of the 3D-torus.

The capacity of each bucket is then determined as $\lfloor 1024/NumBuckets \rfloor$. The capacity of each bucket can be further optimized by noting that, the algorithm allows at most one full bucket for routing nodes along each dimension at any point of time.

4.4. Handling non-power of two nodes

Let ts denote the global table size (always power-of-two) and g the global offset in the table determined from a randomly generated number. Let pe be the node storing the table entry corresponding to the index g. When the number of nodes, n, is not a power of two, the benchmark distributes $mlts + 1$ entries to the first rem nodes $(0..rem - 1)$ and $mlts$ entries to the remaining nodes $(rem..n - 1)$ where $mlts = \lfloor ts/n \rfloor$ and $rem = ts - mlts \times n$. Now, in order to determine the pe from the global offset g, the benchmark checks if g is contained in the first rem nodes or not (this is easily done by storing the global offset of the first entry of the $(rem + 1)^{th}$ node). If so, then the destination PE, $pe = g/(mlts + 1)$. If not, then $pe = (g - rem)/mlts$. This requires division operations that are typically slower operations. On many architectures, integer divides can take 1-2 orders of magnitude more cycles than other operations. Recognizing this problem, the benchmark has a separate code path for handling power-of-two nodes and non-power-of-two nodes. When using the routing technique, this division is performed on every routing node as well.

We propose a node prediction technique that completely avoids the division operator and determines the pe using only multiplication and bit operations. The main idea of the technique is as follows. $pe \approx g/mlts$ and $mlts \approx ts/n$. Therefore pe can directly be approximated by $\lfloor (g \times n)/ts \rfloor$. The division by ts can be performed quickly using bit operations, since ts is a power-of-two. Of course, since we have discounted floors in the approximations, the resulting pe may not be correct. The Theorem below shows that the pe as predicted above can only be off by 1 under a reasonable assumption that $n < mlts$. This assumption is fairly safe for all systems today and most systems envisaged in the near future. Even with a modest local table size of 256M per processing element, this technique can scale up to 32 million

processing elements. Correction of the predicted pe can be performed by comparing the global offset with the boundary offset of the predicted node (which is straightforward to compute).

Theorem 4.1. *Let* $top = rem \times (mlts + 1)$. *If* $n < mlts$, *then*

$$p \leq \lfloor (g \times n)/ts \rfloor \leq p + 1$$

where $p = \lfloor g/(mlts + 1) \rfloor$ *if* $g < top$ *and* $p = \lfloor (g - rem)/mlts \rfloor$ *if* $g \geq top$.

In the inequalities stated in the theorem above , $\lfloor (g \times n)/ts \rfloor$ is the node predicted by our technique and p is the actual node computed by the HPCC base code (depending on the comparison of global offset, g, and top).

Proof: First suppose that $g < top$.

Note that

$$ts < (mlts + 1) \times n$$

Therefore,

$$\begin{aligned} \frac{g \times n}{ts} &> \frac{g}{mlts + 1} \\ \Rightarrow \left\lfloor \frac{g \times n}{ts} \right\rfloor &\geq \left\lfloor \frac{g}{mlts + 1} \right\rfloor \end{aligned} \tag{2}$$

Now,

$$\begin{aligned} \left\lfloor \frac{g}{mlts} \right\rfloor &= \left\lfloor \frac{g}{(mlts + 1)} \cdot \frac{(mlts + 1)}{mlts} \right\rfloor \\ &= \left\lfloor \frac{g}{(mlts + 1)} \cdot \left(1 + \frac{1}{mlts}\right) \right\rfloor \\ &= \left\lfloor \frac{g}{(mlts + 1)} + \frac{g}{mlts(mlts + 1)} \right\rfloor \\ &\leq \left\lfloor \frac{g}{(mlts + 1)} \right\rfloor + \left\lfloor \frac{g}{mlts(mlts + 1)} \right\rfloor + 1 \\ &= \left\lfloor \frac{g}{(mlts + 1)} \right\rfloor + 1 \end{aligned} \tag{3}$$

since $g < ts = mlts \times n$ and $n < mlts$ imply that $g < mlts(mlts+1)$ – hence the middle term vanishes. Therefore, inequalities 2 and 3 imply that

$$\left\lfloor \frac{g}{mlts + 1} \right\rfloor \leq \left\lfloor \frac{g \times n}{ts} \right\rfloor \leq \left\lfloor \frac{g}{mlts + 1} \right\rfloor + 1$$

Now we focus on the case when $g \geq top$. Note that

$$ts \geq mlts \times n$$

Therefore,

$$\begin{aligned} \frac{g \times n}{ts} &\leq \frac{g}{mlts} \\ \Rightarrow \left\lfloor \frac{g \times n}{ts} \right\rfloor &\leq \left\lfloor \frac{g}{mlts} \right\rfloor \\ &\leq \left\lfloor \frac{(g - rem)}{mlts} + \frac{rem}{mlts} \right\rfloor \\ &\leq \left\lfloor \frac{g - rem}{mlts} + 1 \right\rfloor \quad \text{since } rem < mlts \\ &\leq \left\lfloor \frac{g - rem}{mlts} \right\rfloor + 1 \end{aligned} \tag{4}$$

To prove the other side of the inequality, let p $(0 \leq p < n)$ denote the PE on which the global offset g falls and let o denote the offset of the g^{th} entry in the local table of PE p. Then,

$$\begin{aligned} g &= (mlts + 1) \times rem + mlts \times (p - rem) + o \\ &= p \times mlts + rem + o \end{aligned}$$

since the first rem nodes contain $mlts + 1$ entries and the remaining $mlts$ entries.

Then,

$$\begin{aligned} \left\lfloor \frac{g \times n}{ts} \right\rfloor &= \left\lfloor \frac{(p \times mlts + rem + o) \times n}{ts} \right\rfloor \\ &= \left\lfloor \frac{p \times (n \times mlts + rem)}{ts} + \right. \\ &\qquad \left. \frac{(rem + o) \times n - p \times rem}{ts} \right\rfloor \\ &= \left\lfloor p + \frac{rem \times (n - p) + n \times o}{ts} \right\rfloor \\ &\geq p \quad \text{(since } n - p > 0 \text{ and} \\ &\qquad \text{all terms are positive)} \\ &= \left\lfloor \frac{g - rem}{mlts} \right\rfloor \end{aligned} \tag{5}$$

Therefore, inequalities 4 and 5 imply that

$$\left\lfloor \frac{g - rem}{mlts} \right\rfloor \leq \left\lfloor \frac{g \times n}{ts} \right\rfloor \leq \left\lfloor \frac{g - rem}{mlts} \right\rfloor + 1$$

□

5. Performance analysis and results

We now present the performance of the multi-core algorithm on the Blue Gene/P system and compare it with the single-core performance. The single-core version is obtained by porting the Blue Gene/L optimized algorithm based on aggregation and software routing (which currently holds the record for best performance at 35.5 GUPS on a 64K node system [8]) to the Blue Gene/P system.

5.1. Bottleneck of the multi-core algorithm

Based on the functions performed by each of the cores, the CPU time spent by individual cores for every update can be specified by the following set of equations.

$$T_0 = t_g + t_s + t^o_{p0}, \qquad T_1 = t_s + t_r + t^o_{p1},$$
$$T_2 = t_s + t_r + t^o_{p2}, \qquad T_3 = t_r + t_u + t^o_{p3} \tag{6}$$

Where, t^o_{pi} represents the overhead involved in performing the required operations on i^{th} core. These processing overheads are different, depending on the functions performed on the corresponding core. For instance, *core 3* receives updates that are destined for the local node, and hence doesn't incur any routing overheads. The overall performance of the system will be limited by the slowest component. Therefore, the CPU bottleneck of the system can then be determined by

$$GUPS \leq 1/max(T_0, T_1, T_2, T_3) \tag{7}$$

Based on the fact that core 3 is involved in updating the memory as well as communication over the network, we expect core 3 to be more heavily loaded. Core 1 on the other hand does not perform any routing or memory operations and is therefore expected to be least loaded. Cores 1 and 2 are expected to be equally loaded based on their functionality. We conducted experiments to determine the load on each core by progressively disabling processing (discarding the received updates without forwarding/updating) on the cores one at a time from core 3 down to core 1. On a 64-node system, this resulted in average load of 114, 112 and 95 cycles per update when core 3, 2 and 1 were disabled respectively down from 146 cycles in the full run. These numbers are suggestive of a performance load of 95, 112, 114 and 146 cycles per update for cores 0, 1, 2 and 3 respectively. To confirm this we introduced artificial delays on these cores observing a corresponding increase in the load. These experiments indicate that the multi-core algorithm achieves reasonably good load-balancing amongst the four cores of the Blue Gene/P system.

5.2. Performance model

The theoretical CPU bottlenecks for the multi-core algorithm were calculated using equations 6 and 7. Core 3 is the slowest of the cores (as determined above) and thus determines the bottleneck in equation 7. Therefore, to calculate the CPU bottleneck from equation 6, we require the Blue Gene/P specific parameters, t_r and t_u and the processing overhead (t^o_{p3}) on core 3. Recollect from section 4.1, the time to perform the update operation is about 104 cycles. Pipelining of two memory update operations reduces the update time to an average of 90 cycles per update. Therefore, we take $t_u = 90$. The processing overhead of receive operation per update, t_r is governed by the number of updates sent in each bucket. It is calculated as $t_{pkt\_rcv}/BktSize$ where $t_{pkt\_rcv} = 330$ cycles is the receive function call latency per packet when amortized over 4 packets (see Section 4.3.1) and $BktSize$ is the number of updates per bucket. The method for calculating $BktSize$ is described in Section 4.3.1. These parameters are listed in Table 4 for the different system configurations. The processing overhead on core 3 (t^o_{p3}) is estimated from the 64 node system performance. The performance of 0.38 GUPS on a 64-node system, corresponds to an average of about 143 cycles per update on *core 3*. Discounting memory update time of $t_u = 90$ cycles per update and average receive time per update $t_r = 11.3$ cycles, leads to an average processing overhead t^o_{p3} of 41.5 cycles per update. Plugging these values into equations 6 and 7, we obtain the CPU bottlenecks listed in Table 4.

N	Division based (GUPs)	Optimized (GUPs)
96	0.34	0.57
192	0.66	1.11
384	1.32	2.05
768	2.48	4.11

Table 3. Performance of RandomAccess Algorithm for non-power of two nodes

To evaluate the CPU bottleneck of the single core version, we estimate the routing overheads induced by the algorithm from a 64 node system. The performance results of a 64-node system, lead to an average of 457 cycles per update. We determined that the parameters $t_g + t_u + t^o_s$ attribute to about 92 cycles using a sequential version of the algorithm modified to pipeline 2 updates. Therefore, the remainder of 365 cycles can be attributed to the overheads induced by software routing ($t_s + t_r + t^o_p$) as a first order approximation. On a 64 node system with $4 \times 4 \times 4$ configuration, approximately $3/4^{th}$ of updates need to be routed along the X, Y and Z dimension. Thus, the expected number of software routing hops for an update is 2.25(= $3/4 + 3/4 + 3/4$). The average per-hop routing overhead for an update may be approximated to 162 cycles on a 64 node system. Although the average send and receive times per update vary based on the packet size, we assume that the processing overhead per update involved in performing the routing operation (t^o_p) remains unchanged with increase in system size.

We also calculated the network bottlenecks and determined that the crossover would happen at a system of size 64K for the multi-core algorithm and beyond that for the single-core algorithm. Therefore, for all system sizes under consideration, the CPU is the bottleneck for the aggregation and software routing-based multi-core algorithm.

	Blue Gene/P						Blue Gene/L	
	Multi-core				Single-core		Single-core	
N	BktSize	t_r	Bottleneck	GUPs	Bottleneck	GUPs	Bottleneck	GUPs
64	29	11.3	0.38	0.38	0.12	0.11		
128	27	12.2	0.75	0.77	0.23	0.22	0.21	0.22
256	25	13.2	1.49	1.48	0.44	0.44		
512	23	14.3	2.94	2.91	0.83	0.85	0.81	0.84
1024	19	17.3	5.68	5.53	1.60	1.67	1.60	1.78
2048	17	19.4	10.91	10.51	3.47	3.21	3.14	3.3
4096	15	22.0	20.39	17.02	6.70	5.60	6.23	5.83
8192	12	27.5	43.8	36.89			12.25	10.99
16384	10	33.0	84.6	64.99			24.30	18.03
32768	9	36.7	164.6	103				(41.35)
65536							95.97	35.47 (75)

Table 4. Theoretical CPU bottlenecks and actual performance of RandomAccess algorithms

5.3. Performance results

Table 3 shows performance on BG/P using the optimization for non-power of two nodes against using the division based operation for determining the destination PE. The number of nodes is selected to be halfway between consecutive powers of 2. It can be seen from Tables 3 and 4 that the performance for non-power of two nodes degrades even if 50% more nodes are added to a system with power-of-two nodes, e.g. performance on 768 nodes is less than that on 512 nodes. The node prediction based mechanism leads to about 60% improvement in performance. These results are the average of the smaller and larger power-of-two systems. This is because using the optimized technique, core 3 continues to be the bottleneck as in power-of-two node systems.

The performance of the multi-core algorithm on the Blue Gene/P system for different system configurations is presented in Table 4 along with the performance of the single-core version. Following observations can be made from this table: (a) the performance predicted by our model (under the head "Bottleneck") is within 70% percent of the observed performance (90% for small systems). (b) Performance of single-core implementation on Blue Gene/L is slightly better than that on Blue Gene/P. The system clock speed and network bandwidth is higher in BG/P as compared to BG/L. However, the BG/P system incurs additional DMA overheads that are absent in the BG/L system. This results into an overall performance degradation. (c) The multi-core algorithm on BG/P gives approximately a factor of 3 boost in performance as compared to the single core algorithm. (d) The performance on a 32 racks BG/P system is 103 GUPs beating the current record of 35.5 GUPs by almost a factor of 3.

The performance on Blue Gene/L is taken from [8]. New performance numbers of 41.35 GUPS and 75 GUPs (in parenthesis) have been obtained on the 32 rack and 64 rack system using HPCC version 1.2.0.

6. Conclusion

In this paper, we identified factors impacting the performance of the RandomAccess benchmark on next generation supercomputers. In addition to the bisection bandwidth, the permitted multi-core mapping modes, error tolerance and the performance of integer divide operation impact the performance of this benchmark. We present a multi-core algorithm that distributes the benchmark work-load on the quad-cores of each Blue Gene/P node. We obtain speedup of a factor of close to 3 using our optimizations when compared with a single core version. Our load-balancing algorithm is general and can be applied to any system where the number of cores and number of dimensions do not match. Consider a d-dimensional virtual torus topology mapped onto nodes with k cores. Based on the bottlenecks, a set of k' cores can be dedicated for software routing of the updates and the remaining $(k - k')$ cores can be dedicated for performing memory updates (of disjoint entries). Therefore, routing along the d dimensions can be divided amongst the k' cores with each core routing along d/k' of the dimensions.

Acknowledgment

The authors would like to thank James Sexton, John Gunnels and Saurabh K. Garg for their valuable help on this and other HPC Challenge benchmarks. We would also like to thank Tom Spelce of Lawrence Livermore National Laboratory and Kalyan Kumaran and Scott Parker of Argonne National Laboratory for helping us run the benchmark on the systems.

References

[1] IBM Blue Gene Team, "Overview of the IBM Blue Gene/P project," *IBM J. Res. Dev.*, vol. 52, no. 1/2, pp. 199–220, 2008. [Online]. Available: http://www.research.ibm.com/journal/rd/521/team.html

[2] Los Alamos Lab, "High-performance computing: Roadrunner." [Online]. Available: http://www.lanl.gov/roadrunner

[3] S. Saini, D. C. Jespersen, D. Talcott, J. Djomehri, and T. Sandstrom, "Application-based early performance evaluation of SGI Altix 4700 systems for SGI systems," in *CF '08: Proceedings of the 2008 Conference on Computing Frontiers*. ACM, 2008, pp. 113–114.

[4] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, and R. Biswas, "Scientific application-based performance comparison of SGI Altix 4700, IBM POWER5+, and SGI ICE 8200 supercomputers," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 2008, pp. 1–12.

[5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005. [Online]. Available: http://www.research.ibm.com/journal/rd/494/kahle.html

[6] "Intel's teraflops research chip, advancing multi-core technology into the tera-scale era." [Online]. Available: http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf

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

[8] R. Garg and Y. Sabharwal, "Software routing and aggregation of messages to optimize the performance of HPCC Randomaccess benchmark," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, 2006, p. 109.

[9] C. Jacobi, II and C. Lichtenau, "Highly concurrent locking in shared memory database systems," in *Euro-Par '99: Proceedings of the 5th International European Conference on Parallel Processing*. Springer-Verlag, 1999, pp. 477–481.

[10] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, 1992.

[11] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *IPTPS '03: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003, pp. 98–107.

[12] G. Gostin, J.-F. Collard, and K. Collins, "The architecture of the HP Superdome shared-memory multiprocessor," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*. ACM, 2005, pp. 239–245.

[13] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 2007, pp. 339–350.

[14] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM J. Res. Dev.*, vol. 51, no. 6, pp. 639–662, 2007.

[15] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: Generalized scalable message passing on the Blue Gene/P supercomputer," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 2008, pp. 94–103.

[16] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, "Optimization of all-to-all communication on the Blue Gene/L supercomputer," in *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing*. IEEE Computer Society, 2008, pp. 320–329.

[17] Y. Sabharwal, S. K. Garg, R. Garg, J. A. Gunnels, and R. K. Sahoo, "Optimization of fast fourier transforms on the Blue Gene/L supercomputer," in *HiPC '08: Proceedings of the 15th International Conference of High Performance Computing*, 2008, pp. 309–322.