Massive data analytics: The Graph 500 on IBM Blue Gene/Q

F. Checconi
F. Petrini

Graph algorithms are becoming increasingly important for biology, transportation, business intelligence, and a wide range of commercial workloads. Most graph algorithms stress to the limit various architectural aspects of conventional machines. The memory access patterns are irregular, with little spatial locality and data reuse. The amount of computation per loaded byte is very small, typically involving bit manipulation; pointer-chasing is often the norm. Likewise, the generated network traffic comprises small packets that are sent to random destinations at a very high messaging rate. With our recent winning Graph 500 submissions in November 2010, June 2011, and November 2011, we have demonstrated the versatility of the IBM Blue Gene® family of supercomputers and the possibility of using them to parallelize demanding data-intensive applications. In this paper, we describe the algorithmic techniques that we used to map the Graph 500 breadth-first search (BFS) exploration on the IBM Blue Gene®/Q, achieving a performance of 254 billion traversed edges per second.

Introduction

Graphs offer a natural representation for unstructured data from a variety of application areas such as biology, transportation, and security. The graphs may or may not be directed, weighted, typed, or semantic, with features on the vertices and edges. Queries on these graphs are often challenging because of the response time needed, ingestion of massive volumes of data, and dynamic updates to the graph.

The past decade has seen an exponential increase of data produced by online social networks, blogs, and microblogging tools. Many of these data sources are best modeled as networks, and there has been exhaustive research on interpretation of sociological processes and their temporal evolution through properties of the underlying network. Monitoring information propagation is a key aspect: Different facets of this problem are identifying individuals with a large following, identifying individuals originating major conversations or activities in a community, and measuring the impact of an exchange or information update in a network. Moreover, because of the evolving nature of social media, each of these tasks represents a continuing activity of growing complexity. This introduces a number of algorithmic challenges: discovering entities driving a social phenomenon as soon as it occurs, establishing a framework that will ensure coverage of the social phenomena within a reasonable time by using an optimal amount of resources, and predicting macroscopic or microscopic activities in the network.

Graph algorithms are increasingly important for high-performance computing (HPC) workloads, and most graph algorithms stress to the limit various architectural aspects of a conventional machine. The memory access pattern is irregular, with little spatial locality and data reuse. The amount of computation per loaded byte is very low, with IBM Blue Gene*/Q typically involving bit manipulation. Pointer-chasing is often the norm, with several logical threads that can be executed in parallel. Classical optimizations, such as hardware prefetching, have little benefit with this class of algorithms. Conventional processors are optimized to execute complex floating-point operations, for example, assisted by a rich set of vector operators, whereas graph algorithms rely mostly on the integer parts of the processor. Likewise, the generated network traffic

Digital Object Identifier: 10.1147/JRD.2012.2232414

©**Copyright** 2013 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied by any means or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of this paper must be obtained from the Editor.

0018-8646/13/$5.00 © 2013 IBM

is composed of small packets that are sent to random destinations with a very high messaging rate.

The main contributions of this paper are as follows: 1) a description of the Graph 500 benchmark; 2) a detailed description of the Blue Gene/Q implementation of the winning Graph 500 submission of November 2011; and 3) an analysis of the optimization techniques used. The remainder of this paper is organized as follows. The first section introduces the existing results on breadth-first search (BFS) graph exploration. The second section outlines the characteristics that are relevant to this paper. The third and fourth sections describe the specification of the Graph 500 benchmark and the algorithmic design behind our Graph 500 implementation, respectively. The fifth section describes the performance results and provides insight into the various techniques that we have used to optimize performance taking advantage of several innovative architectural mechanisms available on Blue Gene/Q. Finally, some concluding remarks are given.

Related work

One of the main purposes of the work on BFS solutions is to analyze the relationship between algorithms and hardware architectures, either describing how to design algorithms around existing architectures or proposing new architectural features specifically targeted at graph exploration. The first class of includes proposals for shared memory architectures [1–6], commodity AMD or Intel processors [7, 8], and graphic processing units (GPUs) [9–12]. Our paper belongs to this same class, proposing a BFS solution on a distributed memory machine, similar to what was done on IBM Blue Gene*/L [13]. Examples of the second class, specialized hardware designs, include references [14, 15]. Domain-specific languages for graph processing are presented in references [16, 17]. There is also work focusing on parallelizing specific applications that use algorithms derived from BFS, as is done for example with list ranking in reference [18] or with phylogenetic trees on the Cell Broadband Engine** in reference [19].

The two-dimensional (2D) data distribution and matrix-vector-multiplication-like approach for distributed BFS were initially introduced in reference [20]. Gilbert *et al.* also show the representation of graph algorithms using sparse linear algebra [21]; as part of their work in this area, Buluç and Gilbert address the storage of "hypersparse" (with many fewer edges than vertices) matrices and demonstrate that blocks of a large graph's adjacency matrix have this property [22]. They give a specialized representation and compression scheme for hypersparse matrices. The use of bitmaps for the queues in a BFS was used in the context of GPUs in reference [9]; the benefits of this approach are its predictable space requirements and the simplicity of atomically adding an element to the queue if it is not present.

Table 1 Blue Gene/Q.

Frequencies	
Processor clock	1.6 GHz
External memory	1.333 Gb/s
Torus in midplane	4.0 Gb/s
Power	
Bulk power supplies	64 kW
Node properties	
Processors per chip	16 × A2
Processor voltage	0.85 V
Coherency	SMP
L2/L3 cache size (shared)	32 MB
Main store/processor	8 GB/16 GB
Main store bandwidth	42.6 GB/s
Peak performance	204.8 GFLOPS/node
Torus network	
Bandwidth	10 × 2 × 2 GB/s = 40 GB/s
Off-box latency: hw/MPI (μs)	0.3/2.5
System properties	
Peak performance	20 PFLOPS (96 racks)
Total power	7.9 MW

IBM Blue Gene/Q

Blue Gene/Q [24–26] is the third generation of highly scalable, power-efficient supercomputers in the IBM Blue Gene* family, following Blue Gene/L [27] and Blue Gene*/P [28]. A full-size 96-rack, 20 peta-floating-point operations per second (PFLOPS) Blue Gene/Q system called *Sequoia* has been installed at the Lawrence Livermore National Laboratory, whereas a 48-rack configuration named *Mira* is being installed at the Argonne National Laboratory.

Table 1 shows a summary of the hardware features of Blue Gene/Q. Blue Gene/Q systems consist of compute nodes and I/O nodes. Applications run on the compute nodes, whereas file input/output (I/O) is handled by communicating with I/O nodes that send requests over PCI Express** (PCIe**) interfaces to a file system. Compute nodes are interconnected via a five-dimensional (5D) torus. There are 1,024 compute nodes in a Blue Gene/Q compute rack, as shown in **Figure 1**.

The A2 processor core implements the 64-bit Power* instruction set architecture (Power ISATM) and is optimized for aggregate throughput. It supports 2-way concurrent instruction issue: one integer, branch, or load/store instruction and one floating-point instruction. Each core supports four hardware threads, and within each thread, dispatch, execution, and completion are in-order. Of the 18 cores, 16 are exposed to user applications, 1 is used for system software functionality, and 1 is for yield purposes; i.e., 17 working cores exist per chip. A quad floating unit is associated with each core, allowing the chip to achieve 204.8 giga-FLOPS peak. Thus, at 1,024 chips per

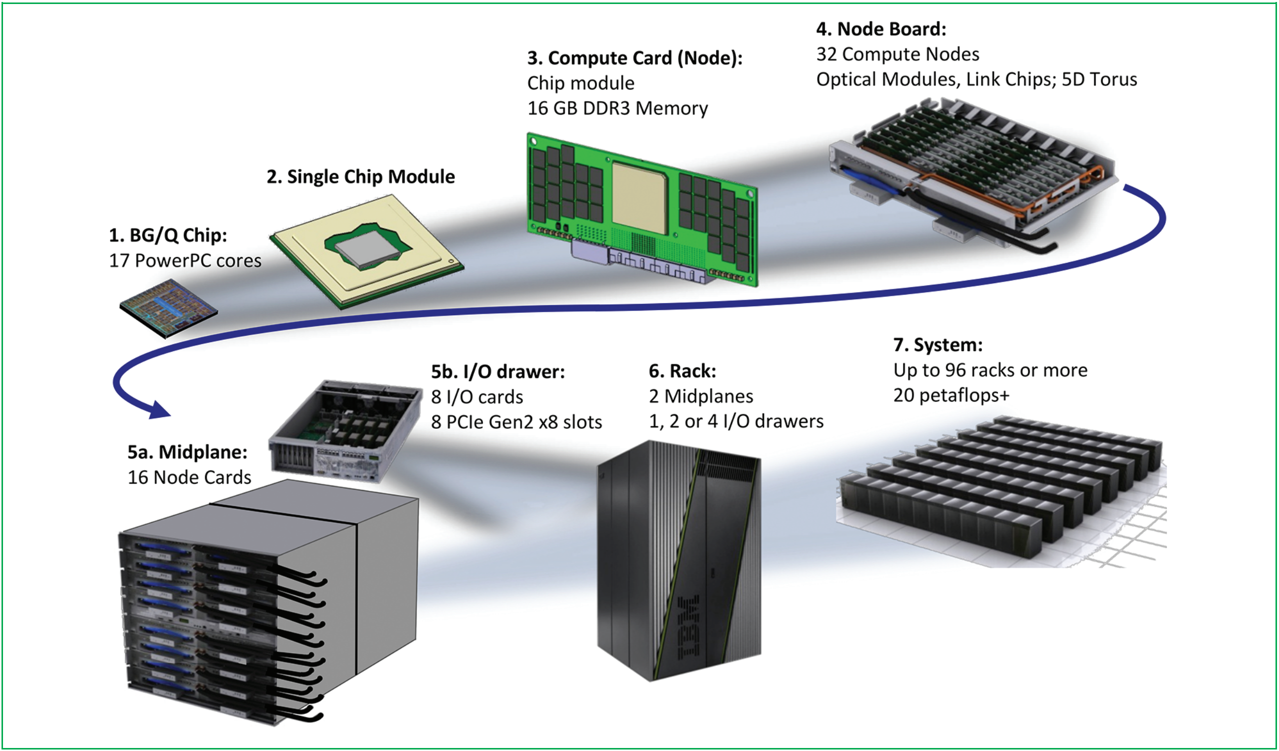


Figure 1

IBM Blue Gene/Q dense packaging hierarchy for massive scale-out. ©2012 Springer Verlag. Reprinted, with permission, from [23].

rack, the 96-rack LLNL Sequoia systems should achieve a peak performance of 20.1 PFLOPS. The first level (Level 1 [L1]) data cache of the A2 core is 16 KB, 8-way, set-associative, with 64-byte lines. The 32-byte wide load/store interface has sufficient bandwidth to support the quad floating-point unit. The L1 instruction cache is 16 KB, 4-way, set-associative.

Graph 500

The Graph 500 is a coordinated attempt at establishing a set of large-scale benchmarks characterizing data-intensive applications. Its specifications are the result of the efforts of a steering committee whose members come from academia, industry, governmental agencies, and national laboratories, and despite its recent introduction, the Graph 500 is becoming a widely recognized alternative to more traditional benchmarks that very often fail to capture the essence of big-data applications.

At the time of this writing, the Graph 500 specification defines one timed kernel consisting of a BFS search over a randomly generated graph. The benchmark is shown in **Algorithm 1** and is composed of the following steps:

Algorithm 1: The Graph500 Benchmark

```
Input: $SCALE$, problem scale
Data: $G$ : graph (edge list representation)
      $R$ : roots to be visited
      $G'$ : graph (internal representation)
      $root$ : current root
      $T$ : BFS tree
  /* Generate the graph.                          */
1 $G \leftarrow$ GENERATEGRAPH($SCALE$) ;
  /* Generate the roots.                          */
2 $R \leftarrow$ GENERATEROOTS($SCALE$) ;
  /* Convert to internal representation.          */
3 $G' \leftarrow$ KERNEL1($G$) ;
4 for $root \in R$ do
      /* Produce the BFS tree.                    */
5     $T \leftarrow$ KERNEL2($G', root$) ;
      /* Validate the result.                     */
6     VALIDATE($G, root, T$) ;
```

- Generate an edge list, where each thread randomly chooses a pair that represents an undirected edge between vertices.
- Convert the edge list representation of the graph to the data structures that will be used during the search. This conversion is called KERNEL1 in Graph 500 terminology, even though it is not timed.

- Randomly determine 64 unique roots with degree of at least one.
- For each search key, compute the parent array, an array that tracks the parent of each vertex reachable from the root; this phase is called KERNEL2 and its run time is used to determine the final benchmark result, whose performance is measured in traversed edges per second (TEPS).
- Finally, each search is validated with a sequence of tests to check the correctness of the BFS tree.

The edge list generation is a highly parallel task that is mostly constrained by the speed of the random number generation. The graph generator is a Kronecker similar to the Recursive MATrix (R-MAT) scale-free graph generation algorithm [29]. This graph model recursively subdivides the adjacency matrix of the graph into four equal-sized partitions and distributes edges within these partitions with unequal probabilities.

The input values required to describe the graph are the SCALE, the base two logarithm of the number of vertices $v = 2^{\text{SCALE}}$, and the EDGE FACTOR, the ratio of the graph's edge count to its vertex count (i.e., half the average degree of a vertex in the corresponding directed graph).

The graph generator creates a small number of multiple edges between two vertices as well as self-loops. The algorithm also generates the data tuples with high degrees of locality. As a final step, vertex numbers are randomly permuted and then randomly shuffled. The scalable data generator runs before starting KERNEL1, and in our implementation, it stores its results in main memory.

KERNEL1 transforms the edge list in a collection of data structures that are used for the remaining kernels. The Graph 500 specifications leave to the implementation complete liberty over the internal representation used during the visit, under the condition that the time necessary to convert the graph into it, starting from the plain edge list created by the generation, is timed, and the time taken is reported as part of the final result (this time does not contribute to the main performance metric though).

The BFS of KERNEL2 starts with a single source vertex and then, in phases, finds and labels its neighbors, then the neighbors of its neighbors, etc. This is a fundamental method on which many graph algorithms are based. The benchmark's memory access pattern (internal or external) is data-dependent with a small average prefetch depth. As in a simple concurrent linked-list traversal benchmark, performance reflects an architecture's throughput when executing concurrent threads, each of low memory concurrency and high memory reference density.

The Graph 500 specifications require any compliant output to be validated. A full-scale validation using a known valid result or comparing the output to that of a reference implementation would be extremely expensive both in terms of time and resources, so the implementations are required to perform a small set of consistency controls, verifying the following:

- That all the edges in the BFS tree are present in the edge list generated at the beginning.
- That all the edges in the original edge list have been considered (this requires checking the distance between the ends of each vertex in the BFS tree).
- That the result is a valid BFS tree (again, this involves checking the levels of each child-parent pair in the tree for loops, invalid connections, etc.).

The visit and the validation of its result are repeated for each root in the input set.

BFS algorithm

This section introduces the BFS algorithm and describes how we implemented it on Blue Gene/Q.

Breadth-first search

A graph $\boldsymbol{G}$ can be defined as a tuple (V, E), where V is a set of vertices and E is a set of pairs (u, v), with $u, v \in V$, each denoting an edge joining vertices u and v. In this paper, we consider only *undirected* graphs, i.e., those where $(u, v) \in E$ implies that $(v, u) \in E$ as well. The *size* of a graph is the number of vertices $|V|$. The *neighbors* of a vertex $v \in V$ are the vertices connected to it by an edge in E, i.e., the elements of the set $\{w \in V | (v, w) \in E\}$.

Given a graph $G = (V, E)$ and a source (or root) vertex $v_s \in V$, a BFS explores all the vertices that can be reached from v_s and produces a spanning tree containing the edges that compose the paths leading from each reachable vertex to v_s (the root of the spanning tree). According to the Graph 500 specification, the spanning tree is represented as a *predecessor map* P, which stores the parent $P(v)$ of each node v in the tree. The *level* of a vertex v in the spanning tree is the distance from v_s to v.

Our BFS implementation is *level-synchronized*, i.e., it processes all the vertices at one level before moving to the next. The algorithm keeps a set of active vertices (*In* in the pseudocode), sometimes also called a *frontier*, and at each iteration explores all the vertices that can be reached from the frontier in one step, storing them in a secondary queue (*Out*) that will become the frontier for the next iteration. Each iteration corresponds to a BFS level. We also keep track of the set of vertices already encountered, in *Vis*.

Data layout

The *adjacency matrix* $\mathbf{A}$ of a graph is a $|V| \times |V|$ matrix, whose element $a_{i,j}$ is one if $(v_i, v_j) \in E$ and is zero otherwise.

To distribute the edges of the graph among the compute nodes, we use a block decomposition of **A**:

$$\mathbf{A} = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,\ell-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,\ell-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\ell-1,0} & A_{\ell-1,1} & \cdots & A_{\ell-1,\ell-1} \end{bmatrix}. \qquad (1)$$

A 2D partitioning scheme assigns each block, $A_{i,j}$ of the adjacency matrix (and consequently all the edges corresponding to the nonzero elements of it) to each node.

Under this partitioning of the graph, each node shares the responsibility of owning information about the neighbors of a set of vertices with other nodes, so it needs to communicate with them to keep track of the global contents of *In*, *Out*, and *Vis*. Different ways of handling the bookkeeping and the communication have been used in the past; more details about the existing solutions may be found in references [30] and [13].

Virtual processors

We define a virtual computing platform composed of a grid of $\ell \times \ell$ *virtual processors*, with ℓ a power of 2. We then go back to the block decomposition of the adjacency matrix given in Equation (1) and assign each block $A_{i,j}$ to a different virtual node.

Figure 2(a) shows the logical layout of the virtual processors, each represented by a square, and the blocks of the adjacency matrix assigned to them. We call $N_{i,j}$ the virtual node at coordinates $\langle i,j \rangle$, and we assign it block $A_{i,j}$.

The figure also shows the *ranges* of vertices covered by the virtual processors: we define a range as a set

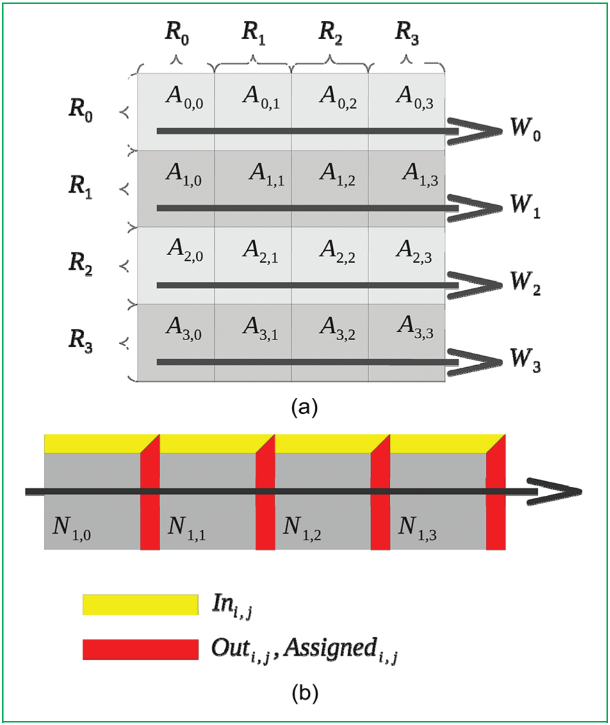
$$R_i = \left\{ v : \frac{|V|}{\ell} i \le v < \frac{|V|}{\ell}(i+1) \right\}$$

of contiguous vertices corresponding to the i-th row (or column) in the block decomposition of **A**. Using the notion of range we can see how each block $A_{i,j}$ represents the edges $\{(u,v) : u \in R_i \land v \in R_j\}$, and that the virtual nodes in the same row share the same R_i, while those in the same column share the same R_j. This simple observation bears a considerable influence on the communication patterns of our implementations, as we show later in this paper.

Data representation

The vertex sets *In*, *Out*, and *Vis* are stored as bitmaps; each vertex set corresponds to a range R_i spanning $|V|/\ell$ vertices, with the presence or absence of each element in that range represented by a bit.

To represent the edges, we use adjacency lists. Although the sparsity of the input graph makes this choice quite inefficient in terms of space (see [22] for an analysis), in this work we focus primarily on the speed of the visit and thus prefer to accept the extra space in exchange for fast access to the neighbors of each vertex.



R_0
R_1
R_2
R_3
R_0
R_1
R_2
R_3
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
W_0
W_1
W_2
W_3
(a)
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$In_{i,j}$
$Out_{i,j}, Assigned_{i,j}$
(b)

Figure 2

Virtual nodes.

Algorithm outline

Algorithm 2 shows the BFS as executed from the virtual processor at coordinates $\langle i,j \rangle$. The visit begins with the initialization of the main variables, in lines 1–4; each node sees only a portion of them. For example, $In_{i,j}$ indicates the portion of the frontier available on node $N_{i,j}$. $Out_{i,j}$ indicates the portion of *Out* on node $N_{i,j}$, and so on.

Figure 2(b) shows the ranges covered by *In*, *Out*, and *Assigned* on a row of virtual processors. It is important to note that all the processors in a column c see the same $In_{i,c}$, and that all the $Out_{r,j}$ in a row r work on the same range R_r. We use a 1D distribution for the predecessor map, as in [30], over virtual processors. We denote by $P(n,v)$ the predecessor of vertex v that is owned by node n. We use the notation $R^{1D}_{i,j}$ to denote the range of vertices assigned to $N_{i,j}$ in the 1D distribution. According to the Graph 500 specifications, the root v_s is its own parent; the node responsible for v_s conforms to this requirement by performing the assignment in line 4.

Each iteration of the loop at line 5 begins the exploration of a new BFS level. Line 6 initializes $Out_{i,j}$ to the empty set, whereas the loop at lines 7 to 9 fills it with all the vertices directly reachable from $In_{i,j}$. Using an adjacency list to

Algorithm 2: KERNEL2 Implementation.

```
   Input: $v_s$: source vertex;
          $\langle i, j\rangle$: $\langle row, col\rangle$ position of virtual node.
   Output: $P$: predecessor map.
   Data: $In$: frontier;
         $Out$: output vertices;
         $Vis$: visited vertices;
         $Assigned$: vertices to be updated;
         $Marks$: edge marks (Blue Gene/P only).
1  $In_{i,j} \leftarrow \begin{cases} \{v_s\} & \text{if } v_s \in R_j \\ \emptyset & \text{otherwise} \end{cases}$ ;
2  $Vis_{i,j} \leftarrow In_{i,j}$ ;
3  $P(N_{i,j}, v) \leftarrow \perp$ for all $v \in R^{1D}_{i,j}$ ;
4  if $v_s \in R^{1D}_{i,j}$ then $P(N_{i,j}, v_s) \leftarrow v_s$ ;
5  repeat
       // Find reachable edges.
6      $Out_{i,j} \leftarrow \emptyset$ ;
7      for $u \in In_{i,j}$ do
8          for $v : (u, v) \in E$ do
9              $Out_{i,j} \leftarrow Out_{i,j} \cup \{v\}$ ;
       // Exclude the visited vertices.
10     $Out_{i,j} \leftarrow Out_{i,j} \setminus Vis_{i,j}$ ;
       // Check for termination.
11     $done \leftarrow \bigwedge_{0 \le k,l < \ell} \left(Out_{k,l} = \emptyset\right)$ ;
12     if $done$ then
13         exit loop;
       // Wave.
14     if $j = 0$ then
15         $prefix_{i,j} \leftarrow \emptyset$ ;
16     else
17         receive $prefix_{i,j}$ from $N_{i,j-1}$ ;
18     $Assigned_{i,j} \leftarrow Out_{i,j} \setminus prefix_{i,j}$ ;
19     if $j \neq \ell - 1$ then
20         send $prefix_{i,j} \cup Out_{i,j}$ to $N_{i,j+1}$ ;
       // Broadcast $prefix_{i,\ell-1} \cup Out_{i,\ell-1}$ back to row $i$
21     $Out_{i,j} \leftarrow \bigcup_{0 \le k < \ell} Out_{i,k}$ ;
       // Write predecessors.
22     for $u \in Assigned_{i,j}$ do
23         for $v : (u, v) \in E$ do
24             if $v \in In_{i,j}$ then
                   // Write the predecessor using RDMA.
25                 $owner \leftarrow$ OWNER1D$(u)$ ;
26                 $P(owner, u) \leftarrow v$ ;
27                 exit loop;
       // Prepare for next iteration.
28     $Vis_{i,j} \leftarrow Vis_{i,j} \cup Out_{i,j}$ ;
29     $In_{i,j} \leftarrow Out_{j,i}$ ;
```

represent the graph allows us to retrieve the beginning of the neighbor list of any active vertex in constant time and to scan it sequentially from an array. This speeds up the internal loop and permits a fast scan of the bitmap representing $In_{i,j}$, skipping all the regions with no bits set.

A subtle factor contributing to the effectiveness of the inner loop is that vertices are added unconditionally. The ratio between the number of updates to the bitmaps and the number of vertices actually inserted can be very high, so a branch inside the inner loop would be extremely costly in terms of performance. Because we add all the reachable vertices without checking whether they have already been visited, we need to exclude $Vis_{i,j}$ from $Out_{i,j}$ in line 10 so that the $Out_{i,j}$ we return contains only the newly discovered vertices.

Line 11 uses a reduce-to-all operation to determine if Out is completely empty. This happens when all of the vertices reachable from In have already been visited and we can therefore terminate the BFS.

We then get into the communication phase, in lines 14 to 20. We refer to this communication as a *wave*, because it proceeds independently along the rows of the virtual processors grid, as shown in Figure 2(a). The arrows in the figure traverse the virtual processors involved in each wave W_i. The task of each wave is to gather the information for all the vertices that has been computed and placed in Out. The wave also computes on every node, a set $Assigned_{i,j}$ that contains the vertices that $N_{i,j}$ is the first to discover in the wave W_i of row i. To implement the wave, we compute a prefix sum (scan) on each row (lines 14–17), and we subtract its result from the vertices in Out (line 18) to obtain $Assigned_{i,j}$. After this step the sets $Assigned_{i,j}$ are disjoint, and their union is equal to the set of all the vertices that were discovered at the current level. We use this partition of the output vertices to ensure that at most one update is performed to the predecessor map for every vertex; for each vertex $v \in Out$, the node that has v in its $Assigned_{i,j}$ set is responsible for updating the predecessor map for v.

The union of all the $Out_{i,j}$ is now collected on each node of the same row, in line 21. We take advantage of the Blue Gene/Q toroidal network implementing the broadcast as two parallel prefixes, one calculated from left to right (the one described above) and one from right to left; each node combines the two prefixes and its local copy of the bitmap to calculate the union of all the $Out_{i,j}$ in its row (i.e., the value that would have been broadcast by the node in the last column).

The simplicity of the frontier exploration has a downside: when we need to update the predecessor map, we only know whose predecessors we need to write, but we have lost track of the actual predecessors. This means that we need to find them again. This requires iterating through the adjacency lists again. Although this may seem like duplicated effort, note that this time we have a very small number of adjacency lists to scan (the ones relative to the vertices in $Assigned_{i,j}$), and we can therefore afford a slightly more complicated inner loop.

Lines 22 to 27 scan the bitmap representing the vertices we have to update; for each of them, we look for the first possible predecessor (i.e., neighbor belonging to the current frontier). When we find it, we use a remote write to assign

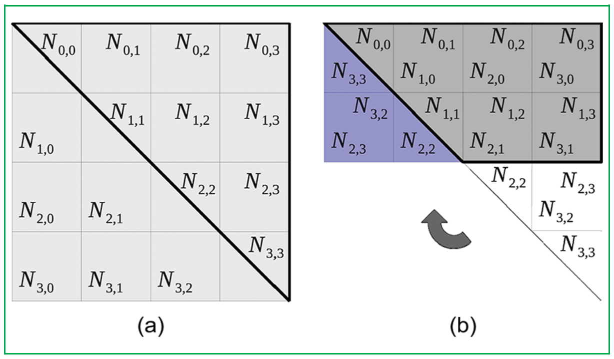


Figure 3

Virtual node mapping.

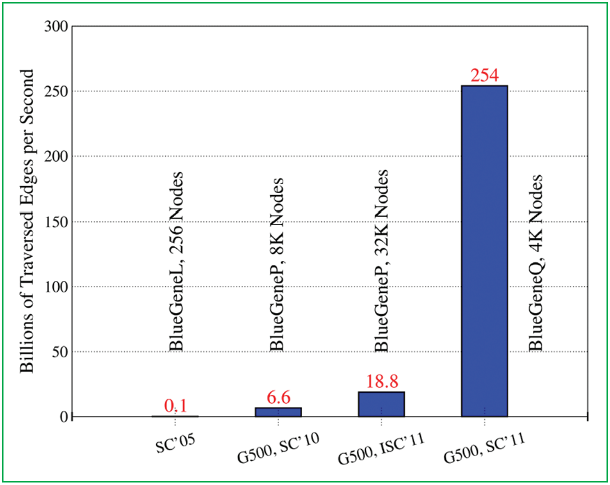


Figure 4

Performance improvements of BFS over several generations of IBM machines.

the relevant entry of the predecessor map on the node that owns it (found via a call to OWNER1D in the 1D decomposition of $\boldsymbol{P}$.

The algorithms described in this paper compute updates to the predecessor map at each level of the BFS. However, by making the sets of assigned vertices and edge marks cumulative (rather than computing them separately for each level), as well as storing each vertex's level number during the BFS, the predecessor map can be computed in a single pass after all levels have completed.

Each iteration ends by preparing for the next level. In line 28, $Vis_{i,j}$ is updated, adding to it the newly found vertices from $Out_{i,j}$, whereas in line 29, the new frontier replaces the old one. Note that $In_{i,j}$ is replaced by $Out_{i,j}$, requiring a transposition on the virtual grid. We show later that this can be avoided by carefully mapping the virtual processors to the physical hardware.

Mapping the virtual processors

The mapping of the virtual processors on the physical compute nodes varies depending on the number of nodes. In what follows, we describe the case when the number of compute nodes m can be written as $m = 2^{p+1}$, i.e., when m is a power of 2 but not a power of 4.

Figure 3(a) shows the grid of virtual nodes $N_{i,j}$. Since our graphs are undirected, their adjacency matrices are symmetric, and so $A_{i,j} = A^T_{j,i}$: each virtual node has the same edges as its reflection across the diagonal of the grid. We exploit this symmetry and store $N_{i,j}$ and $N_{j,i}$ on the same physical node. This can be visualized as the lower triangle in Figure 3(a) "folding" along the diagonal, ending up on top of the upper triangle. Considering that the diagonal blocks $\mathbf{A}_{i,j}$ are symmetric, we only need to store half of them; after the folding this would leave us with $\ell^2 + \ell/2$ nodes of which ℓ are on the diagonal and are only half-full. We then proceed to a second transformation, shown in **Figure 3(b)**, where half of the diagonal nodes are compacted on the remaining half, "rotating" the lower half of the triangle to form a rectangle of $\ell^2/2$ elements; a modification of this mapping is used on Blue Gene/P.

Because of the symmetry of the graph storage, all pairs of rows in the process grid intersect, and so all collectives across individual rows must be nonblocking to avoid sequentializing them because of the overlaps between rows. The folded triangle of virtual processes is embedded into the 5D torus with unit dilation; neighbors in the triangle become neighbors in the physical system topology.

The mapping used on Blue Gene/Q when m is a power of 4 are a bit-convoluted. We limit our discussion here to the fact that it is based on the observation that instead of storing only the edges (u, v) with $v < u$, as in the triangular case, we can store (u, v) with probability 0.5 or (v, u) with probability 0.5. This gives us two triangular configurations that allow us to make full use of the $\ell \times \ell$ physical nodes by re-routing the waves. This could be visualized as having two layers in Figure 3(a), folding one of them onto the upper triangle and the other one onto the lower triangle.

Experimental results

Figure 4 shows how the IBM algorithmic design for BFS searches has evolved over the last few years over several generations of supercomputers. From the initial results on Blue Gene/L at 100 million traversed edges per second (MTEPS) with 256 processors [13], to three winning Graph 500 submissions: 6.6 billion traversed edges per second (GTEPS) in November 2010, 18.8 GTEPS in June 2011, both on Blue Gene/P, and 254 GTEPS in November 2011 on Blue Gene/Q. To deliver this exponential

Table 2 Optimization techniques.

	BFS SC10	Graph500 Nov 2010	Graph500 Jun 2011	Graph500 Nov 2011
NETWORK				
Adaptive routing	.	.	.	●
HW-offloaded collectives	.	.	.	●
NETWORK INTERFACE				
Non-blocking collectives	.	.	.	◎
Cache injection	.	.	.	◎
Batched message injection	.	.	.	⊛
Short payload embedding	.	.	.	●
Private network queues	.	.	●	◎
Message combining	.	.	.	●
Non-blocking send	.	.	.	◎
Non-blocking put	.	.	●	⊛
PROCESSING NODE	.	.	.	.
Integer vector	.	●	●	●
Floating point vector	.	.	.	.
Bit operations	●	⊛	⊛	◎
L1/L2 atomics	.	.	.	◎
Gather and scatter	.	.	.	.
MAPPING				
Wavefront square	.	.	.	◎
Wavefront triangular	.	⊛	⊛	⊛
Limited dimensionality	.	.	.	◎
ALGORITHMIC				
Lock-free queues	◎	◎	◎	◎
Bit-level wavefronts	.	.	◎	◎
Bit-level indexes	◎	.	.	.
Bit matrix-vector multiply	⊛	.	.	.
Speculative data scan	.	.	.	●
Data Compression	.	⊛	⊛	.
2D decomposition	.	⊛	⊛	⊛
1D decomposition	.	.	.	.
Wavefront algorithms	.	.	◎	◎
Loop unrolling	◎	.	.	●

● ⊛ ◎ Minimal, medium, good measured impact.

performance increase, we have developed algorithms that take into account the underlying hardware capabilities of the machines and exploit their peculiarities, dealing effectively with the most significant bottlenecks. In just a few years, IBM has delivered a performance improvement of 3 orders of magnitude. More importantly, IBM is now able to process, in real-time, highly irregular (R-MAT) graphs that are rapidly approaching trillions of edges, a result that only a few years ago was not even considered within reach.

Blue Gene/Q offers a large number of performance and scalability opportunities to run data-intensive applications [24, 25]. **Table 2** lists the main classes of optimizations that we have used in the last few years and compares the original Blue Gene/L implementation with the three winning Graph 500 submissions in November 2010, June 2011, and November 2011.

The table lists a collection of design characteristics and the importance of those characteristics for the corresponding algorithms. We do not rely on a single, or even limited, set of techniques, such as using a specific communication library rather another. In a first and admittedly cursory analysis, we have identified almost 30 distinct classes of parallelization and optimization techniques.

At the network level, adaptive routing (an intelligent form of routing that is able to deal effectively with congested network areas at the cost of a more complex packet re-assembly at the destination) and hardware-offloaded collectives can have a significant impact. In fact, many HPC

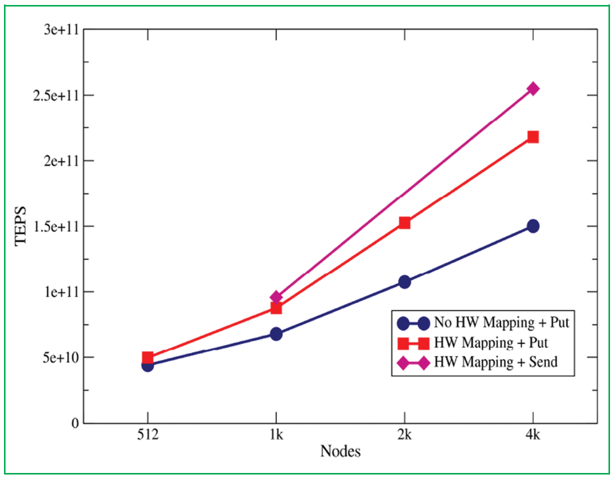


Figure 5

Effect of some optimization techniques.

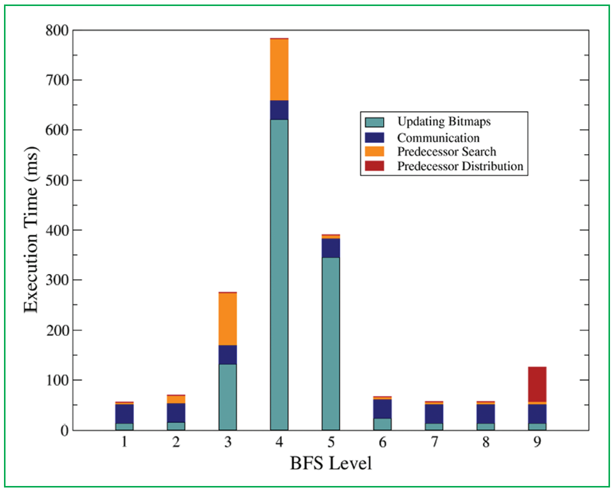


Figure 6

Performance breakdown for each KERNEL2 exploration level on 4,096 Blue Gene/Q nodes.

networks provide these features in a native form, but they are rarely exported to the application developer. In particular, hidden under several layers of system software is the gather/scatter capability to move multiple operands to and from a vector register.

We found that cache-level atomics, in particular those operating on bitmasks, are particularly useful to implement concurrent data updates, and we have observed experimentally and through performance models that extending these primitives to the main memory using non-cache-polluting operations is very beneficial.

All IBM's Graph 500 submissions rely on provably optimal mappings that can support wavefront algorithms for both directed and undirected graphs. Using these mapping strategies, we can efficiently support wavefront algorithms with both square and triangular mappings. **Figure 5** shows the impact of this hardware-mapping technique, which on 4,096 nodes brings performance from 150 GTEPS to almost 210 GTEPS. Finally, we rely on a collection of programming techniques ranging from lock-free queues to loop unrolling to optimize the sequential code. We found that most algorithms can be expressed with 1D and 2D decompositions and that bit operations are almost ubiquitous in graph algorithms.

In **Figure 6** we can see the performance breakdown of the Graph 500 running a scale 34 problem on 4,096 Blue Gene/Q nodes. Each bar identifies the run time during one level of exploration. We can observe that the runtime and the performance breakdown of each level vary dramatically from iteration to iteration. Initially, there is only an active vertex, the root, and the run time is only a few tens of milliseconds. The central levels are always the most computationally intensive, and then the final levels follow a limited number of "chains," sequences of vertices that are isolated and are connected to the central body of the graph with a limited number of edges.

The wavefront communication time (the dark blue part on each bar) is constant at every level, thanks to our mapping algorithms, and it takes approximately 19 ms. The major advantage of this approach is the deterministic run time, which is almost counter-intuitive for a completely irregular and unbalanced problem such as a BFS exploration. The major disadvantage is the speculative nature of the communication that sends the presence bit of each vertex at every iteration (even if the vertex is not active), and this becomes the predominant part of the algorithm for very large-scale configurations. The sequential parts, the bitmap updates (light blue), and the predecessor search consume most of the time in the central "heavy" levels. This clearly highlights the importance of the memory hierarchy. In fact most of the time is spent generating random memory accesses to main memory and atomic updates to the L2 cache.

Finally, the predecessor distribution in this specific configuration is batched and executed in the final level. The updates are grouped in packet-sized messages that are delivered according an all-to-all personalized exchange communication pattern. Depending on the problem size, it may be more efficient to overlap the updates at finer granularity, taking advantage of the remote atomic update capability of the Blue Gene/Q network, or limit the all-to-all on subsets that map directly on network hyperplanes. In our experience, with fewer than 1,024 nodes, predecessor updates were faster using RDMA writes, whereas on larger configurations FIFO batching obtained better results. This can be seen in Figure 5 (the sampling points for FIFO batching on triangular configurations are missing because we did not implement FIFO batching on that mapping).

Even if not shown directly in the graph, other collective operations that are necessary to check the completion of the phases and the separation of each phase are implemented with non-blocking collectives that are almost entirely overlapped during the execution and therefore are not reported in the performance breakdown.

Conclusion

In this paper, we have presented some of the techniques and algorithms used in the winning submissions of the first three editions of the Graph 500 benchmark. We have described the main kernels of the benchmark, the mapping strategies that we have adopted on Blue Gene/Q, and we have provided insight into the optimization techniques and the performance of each phase of the algorithm. Our design and experimental evaluation demonstrate that Blue Gene/Q can efficiently support data-intensive applications, reaching the impressing processing rate of 254 GTEPS on a configuration with 4,096 nodes. We believe that the work presented here and the lessons learned during the algorithmic development can be successfully used in the design of the upcoming exascale supercomputers.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Sony Computer Entertainment Corporation or PCI-SIG in the United States, other countries, or both.

References

1. D. A. Bader, G. Cong, and J. Feo, “On the architectural requirements for efficient execution of graph algorithms,” in *Proc. ICPP*, Oslo, Norway, 2005, pp. 547–556.
2. D. Mizell and K. Maschhoff, “Early experiences with large-scale Cray XMT systems,” in *Proc. 24th IPDPS*, 2009, pp. 1–9.
3. D. A. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2,” in *Proc. 35th ICPP*, 2006, pp. 523–530.
4. G. Cong and D. A. Bader, “Designing irregular parallel algorithms with mutual exclusion and lock-free protocols,” *J. Parallel Distrib. Comput.*, vol. 66, no. 6, pp. 854–866, Jun. 2006.
5. S. Beamer, K. Asanovic, and D. A. Patterson, “Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500,” EECS Dept., Univ. California, Berkeley, CA, Tech. Rep. UCB/EECS-2011-117. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-117.html
6. G. Cong, G. Almasi, and V. Saraswat, “Fast PGAS implementation of distributed graph algorithms,” in *Proc. ACM/IEEE Int. Conf. SC*, 2010, pp. 1–11.
7. Y. Xia and V. K. Prasanna, “Topologically adaptive parallel breadth-first search on multicore processors,” in *Proc. 21st Intl. Conf. PDCS*, Cambridge, MA, 2009. [Online]. Available: http://halcyon.usc.edu/~pk/prasannawebsite/papers/topoBFS_pdcs09_cr_Xia.pdf
8. V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, “Scalable graph exploration on multicore processors,” in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2010, pp. 1–11.
9. P. Harish and P. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *Proc. HiPC*, Berlin, Germany, 2007, vol. 4873, pp. 197–208, ser. LNCS, Springer-Verlag. [Online]. Available: http://rd.springer.com/chapter/10.1007%2F978-3-540-77220-0_21
10. E. Saule and Ü. Çatalyürek, “An early evaluation of the scalability of graph algorithms on the Intel MIC architecture,” in *Proc. Workshop MTAAP, Conjunct. 26th IEEE Int. Parallel Distrib. Process. Symp.*, Shanghai, China, 2012, pp. 1629–1639. [Online]. Available: http://bmi.osu.edu/hpc/papers/Saule12-MTAAP.pdf
11. S. Hong, T. Oguntebi, and K. Olukotun, “Efficient parallel graph exploration on multi-core CPU and GPU,” in *Proc. 20th Int. Conf. PACT*, 2011, pp. 78–88.
12. L. Luo, M. Wong, and W.-M. Hwu, “An effective GPU implementation of breadth-first search,” in *Proc. Des. Autom. Conf.*, Anaheim, CA, 2010, pp. 52–55. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837289
13. A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, “A scalable distributed parallel breadth-first search algorithm on Blue Gene/L,” in *Proc. Int. Conf. SC*, Seattle, WA, 2005, p. 25.
14. M. deLorimer, N. Kapre, N. Metha, D. Rizzo, I. Eslick, T. E. Uribe, T. F. J. Knight, and A. DeHon, “GraphStep: A system architecture for sparse-graph algorithms,” in *Proc. Symp. Field-Program. Custom Comput. Mach.*, Los Alamitos, CA, 2006, pp. 143–151.
15. O. Mencer, Z. Huang, and L. Huelsbergen, “HAGAR: Efficient multi-context graph processors,” in *Proc. Field-Program. Logic Appl., Reconfig. Comput. Going Mainstream*, Berlin, Germany, 2002, vol. 2438, pp. 915–924, ser. LNCS, Springer-Verlag.
16. S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA graph algorithms at maximum warp,” in *Proc. 16th Annu. Symp. Princip. Pract. Parallel Program.*, 2011, pp. 267–276.
17. S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: A DSL for easy and efficient graph analysis,” in *Proc. 17th Int. Conf. ASPLOS*, 2012, pp. 349–362.
18. D. A. Bader, V. Agarwal, and K. Madduri, “On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking,” in *Proc. Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10.
19. F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos, “RAxML-Cell: Parallel phylogenetic tree inference on the cell broadband engine,” in *Proc. IPDPS*, 2007, pp. 1–10.
20. E. Chow, K. Henderson, and A. Yoo, “Distributed breadth-first search with 2-D partitioning,” Lawrence Livermore Nat. Lab., Livermore, CA, Tech. Rep. UCRL-CONF-210829, 2005.
21. J. R. Gilbert, S. Reinhardt, and V. Shah, “High-performance graph algorithms from parallel sparse matrices,” in *Proc. 8th Int. Conf. Appl. Parallel Comput.—State Art Sci. Comput.*, 2006, pp. 260–269.
22. A. Buluç and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” in *Proc. IEEE IPDPS*, 2008, pp. 1–11. [Online]. Available: http://gauss.cs.ucsb.edu/publication/hypersparse-ipdps08.pdf
23. The Blue Gene team, “Blue Gene/Q: by Co-design,” in *Proc. ISC*, 2012. [Online]. Available: http://rd.springer.com/article/10.1007/s00450-012-0215-3
24. D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, “The IBM Blue Gene/Q interconnection fabric,” *IEEE Micro*, vol. 32, no. 1, pp. 32–43, Jan./Feb. 2012.
25. R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, P. A. Boyle, N. H. Christ, C. Kim, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, and G. L. Chiu, “The IBM Blue Gene/Q compute chip,” *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar./Apr. 2012.
26. D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, “The IBM Blue Gene/Q interconnection network and message unit,” in *Proc. Int. Conf. SC*, Seattle, WA, 2011, pp. 26:1–26:10.

27. 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, The Blue Gene team, "Overview of the Blue Gene/L system architecture," *IBM J. Res. & Dev.*, vol. 49, no. 2, pp. 195–212, Mar. 2005.
28. IBM Blue Gene team, "Overview of the IBM Blue Gene/P project *IBM J. Res. & Dev.*, vol. 52, no. 1/2, pp. 199–220, Jan. 2008.
29. D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Data Mining*, 2004. [Online]. Available: http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf
30. A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. Int. Conf. SC*, 2011, pp. 65:1–65:12.

Received March 22, 2012; accepted for publication August 7, 2012

Fabio Checconi *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (fchecco@us.ibm.com).* Dr. Checconi is a Postdoctoral Researcher with the Deep Computing Systems Department at the IBM T. J. Watson Research Center. He received his Laurea degree in computer engineering from the University of Pisa, and his Ph.D. degree from the Scuola Superiore S. Anna, Pisa, Italy. His current research focuses on data-intensive computing, while his interests also include real-time operating systems and resource scheduling.

Fabrizio Petrini *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (fpetrin@us.ibm.com).* Dr. Petrini is a Research Staff Member in the Deep Computing Systems Department at the IBM T. J. Watson Research Center. He received his Laurea and Ph.D. degrees in computer science from the University of Pisa, Italy. His research interests include various aspects of multi-core processors and supercomputers, including high-performance interconnection networks and network interfaces, as well as data-intensive computing applications. He served as associated editor of *IEEE Transactions on Parallel and Distributed Processing* from 2006 to 2012, and conference and technical chair of several editions of IEEE Hot Interconnects.