Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines

Fabio Checconi, Fabrizio Petrini
IBM TJ Watson
Yorktown Heights, NY 10598
Email: {fchecco,fpetrin}@us.ibm.com

Jeremiah Willcock, Andrew Lumsdaine
CREST, Indiana University
Bloomington, IN 47405
Email: {jewillco,lums}@osl.iu.edu

Anamitra Roy Choudhury, Yogish Sabharwal
IBM India Research
New Delhi, DL 110070, India
Email: {anamchou,ysabharwal}@in.ibm.com

***Abstract*—In this paper, we describe the challenges involved in designing a family of highly-efficient Breadth-First Search (BFS) algorithms and in optimizing these algorithms on the latest two generations of Blue Gene machines, Blue Gene/P and Blue Gene/Q. With our recent winning Graph 500 submissions in November 2010, June 2011, and November 2011, we have achieved unprecedented scalability results in both space and size. On Blue Gene/P, we have been able to parallelize a scale 38 problem with 2^{38} vertices and 2^{42} edges on 131,072 processing cores. Using only four racks of an experimental configuration of Blue Gene/Q, we have achieved a processing rate of 254 billion edges per second on 65,536 processing cores. This paper describes the algorithmic design and the main classes of optimizations that we have used to achieve these results.**

I. Introduction

Graphs offer a natural representation for unstructured data from a variety of application areas, such as biology, transportation, complex engineered systems (such as the Internet and power grid), communication data and, more generally, various forms of relational data. The graphs may or may not be directed, weighted, typed, semantic, and have features on their vertices and edges. Queries on these graphs are often challenging due to the response time needed, ingestion of massive volumes of data, and dynamic updates.

The past decade has seen an exponential increase of data produced by online social networks, blogs, and micro-blogging tools. Many of these data sources and networks are best modeled as graphs, 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, individuals originating major conversations or activities in a community, and measuring the impact of an exchange or information update in a network. Moreover, due to 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 reasonable time by using an optimal amount of resources, and predicting macro- or microscopic activities in the network.

These data-intensive applications rely on a collection of search algorithms, and among them breadth-first search (BFS) is probably the most common, and a building block for a wide range of graph applications. For example, in the analysis of semantic graphs, the relationship between two vertices is expressed by the properties of the shortest path between them, given by a BFS search. Applications in community analysis need to determine the connected components of a semantic graph [1]–[4], and connected components algorithms [5] often employ a BFS search. BFS is also the basic building block for heuristic search algorithms (such as A*) used in motion planning for robotics [6], [7].

A. Parallelizing Graph Algorithms

Graph algorithms on massive data are becoming an increasingly important part of modern HPC workloads. These algorithms stress to the limit many architectural aspects of a parallel machine. They manipulate very large data-sets, with irregular access patterns and little spatial locality or data reuse. The amount of computation per loaded byte is very low, often involving byte or even bit manipulation. Pointer-chasing is often the norm, requiring heavy overlap of memory operations to hide latency. Classical architectural optimizations, such as hardware prefetching, provide little benefit to this class of algorithms. Conventional processors are optimized to execute complex floating point operations while graph algorithms mainly use integer and memory operations. Branches are also more common and less predictable in graph algorithms than in the more regular patterns of common scientific software. Similarly, the generated network traffic is composed of small packets sent to random destinations at a very high rate. Conventional HPC networks are built with explicit message passing in mind as the primary communication mechanism, and thus tend to be designed for coarse-grained, rather than fine-grained, communication.

B. Graph 500

Over the years the scientific community has converged towards a commonly accepted benchmark to rank supercomputers, High-Performance Linpack (HPL), and the Top500, a semi-annual list where HPL runs are collected. The Top500 provides

SC12, November 10–16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

an *apples-to-apples* way to compare machines and their capabilities for running dense-matrix-based numerical simulations. Both HPL and the Top500 have been pivotal in guiding the design of new supercomputers and assessing their usability and performance at scale.

The Graph 500 list (http://www.graph500.org/) was introduced in 2010 as an alternative to the Top500 list to rank computer performance on data-intensive applications. Graph exploration is critical for many of these applications, so it is not a coincidence that the Graph 500 has initially focused its attention on the BFS search of a particular class of graphs, the Recursive MATrix (R-MAT) scale-free graphs [8], [9]. The critical processing metric of the Top500, FLoating point OPerations per Second or FLOPS, is replaced by the Traversed Edges Per Second or TEPS in the Graph 500. While still in its infancy, the Graph 500 captures many essential features of data intensive applications, and has raised a lot of interest in the supercomputing community at large, both from the scientific and operational point of view: in fact, many supercomputer procurements are now including the Graph 500 in their collection of benchmarks. The importance of the Graph 500 has also been emphasized by a recent congressional report [10].

C. Contributions

The paper provides the following primary contributions:

- A detailed description of the Graph 500 benchmark and the BFS algorithms that we have used on Blue Gene/P and Blue Gene/Q. It is worth noting that even if the high-level design is similar and based on a bulk-synchronous 2D decomposition ([11]–[13]), the implementation strategies are considerably different. The first two editions of the Graph 500 have ranked their submissions by problem size, leading to the implementation of data compression algorithms and the successful utilization of Blue Gene/P machines with the largest memory footprint, ANL's Intrepid and Jülich's Jugene, in November 2010 and June 2011. In the November 2011 list, the ranking was changed to be based only on speed, giving a substantial performance advantage to a smaller Blue Gene/Q experimental system.
- A description of the architectural features of Blue Gene that are relevant to our parallelization, with emphasis on the innovative features of Blue Gene/Q.
- A detailed experimental evaluation of the algorithms on Blue Gene/P and Blue Gene/Q. We try to explain *how* the different algorithmic components are integrated together, and to provide insight on the performance impact of several architectural and software design choices.

The rest of this paper is organized as follows. Section II provides an overview of existing results in the literature in graph exploration algorithms. Section III describes the Graph 500 benchmark in more detail. Section IV discusses standard parallelizations of the BFS algorithm, while we show the Blue Gene-specific algorithms in Section V. In Section VI, we describe the architectural features of Blue Gene/Q relevant to our algorithmic design, e.g., the adoption of a custom designed, ultra-low latency communication layer and L2 atomic primitives. Section VII analyzes our performance results and provides insight into the various techniques that we used to optimize performance, including taking advantage of several innovative architectural mechanisms available on Blue Gene/Q. Finally, discussion and concluding remarks are given in Section VIII.

II. Related Work

Being a fundamental method in algorithmic graph theory, the BFS has received significant attention in the literature of various fields, not only limited to computer science. Recently, to satisfy the need generated by the increasing amount of parallelism available at the hardware level, several parallel BFS solutions have been proposed.

One of the main purposes of the work on BFS solutions is analyzing 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 proposals includes, for shared memory architectures, [14], [15], [20]–[23]; for commodity AMD or Intel processors, [18], [19]; and for GPUs, [24]–[27]. Our paper belongs to this same class, proposing a BFS solution on a distributed memory machine, similarly to what is done in [17] on Blue Gene/L. Examples of the second class, specialized hardware designs, include [28], [29]. Domain-specific languages for graph processing are presented in [30], [31]. There is also work focusing on parallelizing specific applications that use algorithms derived from BFS, as done for example with list ranking in [32], or with phylogenetic trees on the Cell BE in [33].

The 2-D data distribution and matrix-vector-multiplication-like approach for distributed BFS were initially introduced in [34]. Gilbert et al. also show the representation of graph algorithms using sparse linear algebra [35]; 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 [36]. 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 [24]; the benefit of this approach is its predictable space requirements and the simplicity of atomically adding an element to the queue if it is not present.

Table I shows an admittedly cursory comparison of the most relevant approaches in the literature. For each implementation, we can see the graph properties, the hardware on which the implementation was executed, and the performance obtained. Our Graph 500 submissions (the bold rows) are shown in the bottom part of the table. ANL's Intrepid submission, at scale 38, is the largest BFS search presented in the literature, while the Blue Gene/Q NNSA/SC Prototype II was the fastest in November 2011. This result has been superseded by our

Reference	Type	Performance Vertices	Edges	GTEPS	Processors	Base architecture
Bader, Madduri [14]	R-MAT	200M	1B	0.5	40	Cray MTA-2
	SSCA2v1	32M	310M	0.25	10	
	SSCA2v1	4M	512M	0.25	10	
Mizell, Maschhoff [15]	Uniformly random	64M	512M	0.21	128	Cray XMT
Scarpazza, Villa, Petrini [16]	Uniformly random	1M	256M	0.54	1 chip	IBM Cell/B.E.
Yoo, Chow, et al [17]	Uniformly random	Peak	*d*200	0.73	256	IBM Blue Gene/L
Xia, Prasanna [18]	8-Grid	1M	16M	0.22	Peak	Dual Intel X5580
Agarwal et al. [19]	R-MAT	1M	32M	1.3	4 sockets	Intel Nehalem EX
LBNL Hopper, April 2011 [11]	Custom Graph 500	2^{32}	2^{36}	17.8	40,000 cores	Cray XE6
Mirasol, November 2011[20]	Custom Graph 500	2^{28}	2^{32}	5.1	40 cores/80 threads	Intel Xeon E7-8870
ANL Intrepid, November 2010	Custom Graph 500	2^{36}	2^{40}	6.6	8,192 nodes/32,768 cores	IBM Blue Gene/P
ANL Intrepid, June 2011	Custom Graph 500	2^{38}	2^{42}	18.5	32,768 nodes/131,072 cores	IBM Blue Gene/P
Jülich Jügene, November 2011	Custom Graph 500	2^{37}	2^{41}	92.8	65,536 nodes/262,144 cores	IBM Blue Gene/P
NNSA/SC Prototype II, November 2011	Custom Graph 500	2^{32}	2^{36}	254.0	4,096 nodes/65,536 cores	IBM Blue Gene/Q

TABLE I: Performance of BFS approaches in the literature; adapted and expanded from Table 3 of [19].

improved submission in June 2012, a scale 38 32-rack run on LLNL's Sequoia.[1]

Data compression has been used in several ways in the context of graphs and the structures of sparse matrices, and plays a central role in our Blue Gene/P implementation. Two uses of compression have been explored: compressing graphs on disk for faster loading into memory, and compressing graphs in memory to fit larger data in memory and/or conserve memory bandwidth. The BFS implementation in this paper uses compression for the second purpose. Compression is used to reduce memory consumption for larger graphs, at the expense of processing time. Examples of compression algorithms for power-law graphs (such as links between Web pages) include [37]–[40]. The work of Blandford et al [41], [42] uses a delta-based coding scheme to improve performance of depth-first search. Hannah et al compare various compression techniques on Web graphs [43].

III. Graph 500

The Graph 500 benchmark (http://www.graph500.org/) was designed to measure the performance of computer systems on irregular applications. It was first announced in June 2010, with the first Graph 500 ranking released in November 2010; lists have been released biannually since. The benchmark is overseen by a steering committee of over 50 members from the high-performance computing community.

The benchmark uses a BFS on undirected Kronecker graphs as a kernel to represent graph algorithm performance.The benchmark also defines creation of the graph data structure from a list of edges as a timed kernel, but the current evaluation criterion is only the performance on BFS. Performance is measured as the number of graph edges traversed per second (TEPS). Reference implementations of the benchmark are provided for several platforms, including sequential, OpenMP, the Cray XMT, and MPI.

[1]We plan to release the new technical results in an upcoming publication.

An implementation of the Graph 500 benchmark must perform the following steps:

1) Graph data generation (un-timed): The initial graph data generation step consists of creating a list of random edges using a Kronecker graph generator [9]. The vertex numbers used must be permuted to reduce locality, and the edges must be generated in a random order. The graph generator is also responsible for finding a random set of roots for the searches to start from. Each root must be incident to at least one edge; the Kronecker graph generator generates vertices that fail this test and cannot be visited during searches [44]. Graph data sizes are chosen by the user; the size of the graph is represented by the base-2 logarithm of the number of vertices (known as *scale*) or the nominal data size (assuming 16 bytes of storage per edge). The average degree of the graph is defined to be 32, and the generator is allowed to create duplicate (parallel) edges and self-loops.

2) Data structure creation (timed): Graph 500 implementations are allowed to use any data structure to store their graph data for the BFS step, but conversion to that data structure is a timed kernel in the benchmark.

3) BFS iterations: The code must do a number of iterations of the BFS and validation steps, one for each root generated by the graph generator, each containing: *3.a) BFS (timed)*: The implementation must perform a BFS of the input graph starting from the root, creating a BFS tree. The tree is stored as an array mapping from each graph vertex to its parent, with special values for the root of the tree and for un-visited vertices. *3.b) Result validation (un-timed)*: Validating result correctness involves determining that each claimed BFS tree is a single tree, that all edges in the tree correspond to edges in the graph, and that all graph edges connect vertices at most one level apart in the tree.

IV. BFS Algorithms

A. Sequential BFS

We define a graph G as tuple (V, E), consisting of a set of vertices V and a set of edges E containing all the connections of G as pairs (u, v), with $u, v \in V$. In this paper we only consider *undirected* graphs, i.e., if $(u, v) \in E$, then $(v, u) \in E$ as well. The *size* of a graph is the number of vertices $|V|$. Given a vertex $v \in V$, we call the set of vertices connected to it by an edge in E, i.e., the set $\{w \in V \mid (v, w) \in E\}$, the *neighbors* of v.

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, producing a spanning tree containing the edges that compose the paths leading from each vertex to v_s (the root of the spanning tree). In the Graph 500 benchmark, the spanning tree is represented by a *predecessor map* P, which is a map from each vertex v to its parent $P(v)$ in the tree. The parent of the root v_s is itself, and vertices not reachable from v_s have no parent. The *level* of a vertex v in the spanning tree is the distance from v_s to v.

Algorithm 1: Sequential, level-synchronized BFS.

Input: $G = (V, E)$: graph representation;
v_s: source vertex;
In: current level input vertices;
Out: current level output vertices;
Vis: vertices already visited.
Output: P: predecessor map.

```
1  In ← {v_s} ;
2  Vis ← {v_s} ;
3  P(v) ← ⊥ ∀v ∈ V ;
4  while In ≠ ∅ do
5      // Find the reachable edges.
6      Out ← ∅ ;
7      for u ∈ In do
8          for v | (u, v) ∈ E do
9              if v ∉ Vis then
10                 Out ← Out ∪ {v} ;
11                 Vis ← Vis ∪ {v} ;
12                 P(v) ← u ;
13     // Prepare for next level.
14     In ← Out ;
```

Pseudocode for a sequential BFS is shown in Algorithm 1. Parallel implementations add complexity to this formulation, but do not change its fundamental structure. The variation of BFS we show is *level-synchronized*: all vertices at one level are processed before any vertices further from the root. The algorithm keeps a set of active vertices, 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 that will become the frontier for the next iteration. Each iteration corresponds to a BFS level. In the pseudocode, the frontier is represented by the set *In*, while the set of vertices that can be reached from the frontier is stored in *Out*; to avoid visiting a vertex more than once we also keep a set of visited vertices *Vis*.

The exploration begins initializing the frontier with the source vertex, at line 1; to prevent further visits to the root, we also put it in *Vis*. The external loop, at lines 4–12, explores one level at a time, terminating when the frontier becomes empty. The inner loop at lines 7–12 iterates over the vertices in *In*, exploring the neighbors of each of them. Each neighbor v of u that has not yet been visited (lines 8 and 9) is added to *Out* and *Vis*, while building the spanning tree P. Finally, we prepare the next iteration, updating the current frontier (line 14).

B. Parallel BFS

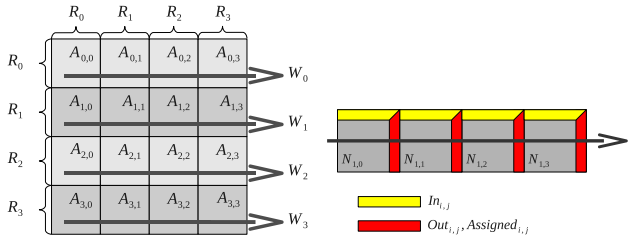
Of the many choices in implementing a parallel BFS, we will go into some detail only about that between *1-D* and *2-D partitioning* of the graph. A 1-D partitioning distributes the vertices among the compute nodes; each compute node is assigned a set of vertices (we say it *owns* them), and it is the sole node responsible for all the information related to them. The BFS traversal follows the steps of Algorithm 1 closely, with the outer loop at lines 4–12 executed on all nodes and the iterations of the loop at lines 7–12 distributed across the nodes. In particular, each node only has the part of *In* containing vertices that it owns, and thus only processes those vertices. For any edge (u, v) adjacent to u determined in line 8, the current node may or may not own v. Thus, a message containing u and v must be sent to the owner of v for it to update *Out*, *Vis*, and P. When the message arrives, the receiver checks that the node v is not already visited in line 9 and then updates *Out*, *Vis*, and P with the received values of u and v. The termination check at line 4 requires communication to count the elements in *In* across the nodes.

In a 2-D partitioning, the primary entities being distributed are the graph edges. To show how edges are assigned to compute nodes, we introduce the notion of *adjacency matrix* of a graph: a square matrix $\mathsf{A} = [a_{u,v}]_{u,v \in V}$ with $a_{u,v} = 1$ if $(u, v) \in E$, and $a_{u,v} = 0$ otherwise. Each of A's rows and columns corresponds to a vertex; if we consider a block decomposition of A on $\ell \times \ell$ nodes:

$$\mathsf{A} = \begin{bmatrix} \mathsf{A}_{0,0} & \mathsf{A}_{0,1} & \cdots & \mathsf{A}_{0,\ell-1} \\ \mathsf{A}_{1,0} & \mathsf{A}_{1,1} & \cdots & \mathsf{A}_{1,\ell-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{A}_{\ell-1,0} & \mathsf{A}_{\ell-1,1} & \cdots & \mathsf{A}_{\ell-1,\ell-1} \end{bmatrix}, \tag{1}$$

a 2-D partitioning scheme assigns blocks of the adjacency matrix (and consequently all the edges corresponding to the nonzero elements of those blocks) to the compute nodes.

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*. The details of the bookkeeping and of the communication may vary from algorithm to algorithm; more details on the state of the art and a comparison of the 1-D and 2-D approaches can be found in [11] and [17].



(a) (b)

Fig. 1: Virtual Nodes.

V. Blue Gene Implementations

Our implementations of BFS for Blue Gene/P and Blue Gene/Q use similar approaches, so we start their description with a common overview, based on an abstraction of the underlying computing platform, then proceed to give the details of each of the two implementations.

A. Virtual Processors

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

Figure 1a 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 $\mathsf{A}_{i,j}$.

The figure also shows the *ranges* of vertices covered by the virtual processors: we define a range as a set $R_i = \left\{v : \frac{|V|}{\ell} i \leq v < \frac{|V|}{\ell}(i+1)\right\}$ 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 $\mathsf{A}_{i,j}$ represents the edges $\{(u, v) : u \in R_i \wedge v \in R_j\}$, 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 will see very soon.

B. 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 1b 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 1-D distribution for the predecessor map, as in [11], 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 1-D distribution. According to the Graph 500 specification, 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. We first invoke ExploreFrontier; this routine returns the set of vertices $Out_{i,j}$ reachable in one step from $In_{i,j}$ on the local node; this routine is implemented in different ways depending on the platform under consideration. On the Blue Gene/P system, this routine also returns a set of edges $Marks_{i,j}$ that will be described later. To keep the notation simple, we avoid mentioning the parameters to ExploreFrontier explicitly, as most of the state is kept in global variables accessible to its implementation.

Line 9 uses a reduce-to-all operation to determine if *Out* is completely empty; this happens when all 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 13–19. We refer to this communication as a *wave*; this is because it proceeds independently along the rows of the virtual processors grid, as shown in Figure 1a. 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 13–16) and we subtract its result from the vertices in *Out* (line 17) 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. This extra step reduces the number of predecessor map updates sent from 160×10^9 to 11.7×10^9 (92.7%) on 32,768 nodes (256 per row) and a scale 35 graph.

The union of all the $Out_{i,j}$ is now collected on each node of the same row, in line 21; the implementation of the broadcast depends on the platform and will be detailed later.

The next step consists of updating the predecessor map. The actual procedure varies depending on the implementation, and is indicated by a call to WritePred in line 23. This function uses $Assigned_{i,j}$ and $Marks_{i,j}$ to generate the required predecessor updates and then performs these updates.

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 25, $Vis_{i,j}$ is updated, adding to it the newly found vertices from $Out_{i,j}$, while in line 26 the new frontier replaces the old one. Note that $In_{i,j}$ is replaced by $Out_{j,i}$, requiring a transposition on the virtual grid. We will see later that this

can be avoided by doing a careful mapping of the virtual processors to the physical hardware.

Algorithm 2: BFS on Blue Gene.

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**
6 *// Find reachable edges.*
7 $\left(Out_{i,j}, Marks_{i,j}\right) \leftarrow$ ExploreFrontier () ;
8 *// Check for termination.*
9 $done \leftarrow \bigwedge_{0 \le k,l < \ell} \left(Out_{k,l} = \emptyset\right)$;
10 **if** *done* **then**
11 **exit loop**;
12 *// Wave.*
13 **if** $j = 0$ **then**
14 $prefix_{i,j} \leftarrow \emptyset$;
15 **else**
16 **receive** $prefix_{i,j}$ **from** $N_{i,j-1}$;
17 $Assigned_{i,j} \leftarrow Out_{i,j} \backslash prefix_{i,j}$;
18 **if** $j \neq \ell - 1$ **then**
19 **send** $prefix_{i,j} \cup Out_{i,j}$ **to** $N_{i,j+1}$;
20 *// Broadcast* $prefix_{i,\ell-1} \cup Out_{i,\ell-1}$ *back to row* i
21 $Out_{i,j} \leftarrow \bigcup_{0 \le k < \ell} Out_{i,k}$;
22 *// Write predecessors.*
23 WritePred () ;
24 *// Prepare for next iteration.*
25 $Vis_{i,j} \leftarrow Vis_{i,j} \cup Out_{i,j}$;
26 $In_{i,j} \leftarrow Out_{j,i}$;

The outline of the algorithm we have given so far covers both our Blue Gene/P and Blue Gene/Q implementations. We now take a closer look at the ExploreFrontier and WritePred procedures and provide specific details for each platform.

C. Blue Gene/Q

Algorithm 3 shows the Blue Gene/Q versions of the two functions; in this section we explain how they work. Before that, we introduce the data structures used to represent the graph and the vertex sets.

1) Data Representation: On both Blue Gene/P and Blue Gene/Q, vertex sets are stored as bitmaps. Each vertex set corresponds to a range R_i spanning $\frac{|V|}{\ell}$ vertices, with the presence or absence of each element in that range represented by a bit. The $In_{i,j}$, $Out_{i,j}$, and $Assigned_{i,j}$ sets are represented by similar bitmaps.

Our Blue Gene/Q implementation represents the graph using adjacency lists. Though the sparsity of the input graph makes this choice quite inefficient in terms of space (see [36] for an analysis), the Blue Gene/Q implementation is focused primarily on the speed of the visit and therefore we prefer to accept the extra space in exchange of fast access to the neighbors of each vertex.

2) Frontier Exploration: In ExploreFrontier, line 4 initializes $Out_{i,j}$, while the loop at lines 5–7 fills it with all the vertices directly reachable from $In_{i,j}$. Using an adjacency list to 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 also 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. Since we add all the reachable vertices without checking if they have already been visited, we need to exclude $Vis_{i,j}$ from $Out_{i,j}$ in line 9, so that the $Out_{i,j}$ we return only contains the newly discovered vertices.

3) Waves: We implement the waves with a slight modification in order to take advantage of the bi-directional torus links constituting the physical network. We divide the data into two equal parts. For the first part, we implement the prefix-sum and broadcast as described in the outline. For the second part, we implement a prefix sum and broadcast similar to the first part—the only difference being that the data flows in the opposite direction. Each node can then calculate the overall union of the output sets as the union of the two prefixes.

4) Predecessor Map Update: The simplicity of ExploreFrontier has a downside: when we invoke WritePred, we know whose predecessors we need to write, but we have lost track of the actual predecessors. In WritePred we need to find them again. This requires iterating through the adjacency lists again. Although this may seem like duplicated effort, note that that this time we have a very small number of adjacency lists to scan (the ones relative to the vertices in $Assigned_{i,j}$; we can therefore afford a slightly more complicated inner loop.

Lines 13–19 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 the relevant entry of the predecessor map on the node that owns it (found via a call to Owner1D in the 1-D decomposition of P.

D. Blue Gene/P

1) Data Representation: On Blue Gene/P, we use more sophisticated data representations to reduce the amount of data to store, while still allowing efficient access:

Algorithm 3: Blue Gene/Q.

```
1  function ExploreFrontier
2  begin
3      // Find the reachable edges.
4      Out_{i,j} ← ∅ ;
5      for u ∈ In_{i,j} do
6          for v : (u, v) ∈ E do
7              Out_{i,j} ← Out_{i,j} ∪ {v} ;
8      // Exclude the visited vertices.
9      Out_{i,j} ← Out_{i,j} \ Vis_{i,j} ;
10     return (Out_{i,j}, ∅) ;
11 function WritePred
12 begin
13     for u ∈ Assigned_{i,j} do
14         for v : (u, v) ∈ E do
15             if v ∈ In_{i,j} then
16                 // Write the predecessor using RDMA.
17                 owner ← Owner1D(u) ;
18                 P(owner, u) ← v ;
19                 exit loop;
```

a) Graph Data Structure: Within each virtual process, the graph is broken into a matrix of blocks, typically 32×32 at larger scales. These blocks are used to distribute matrix-vector multiplications across the cores and for higher locality during frontier exploration.

b) Edge Storage: Within each block of the graph data, edges are stored in coordinate format (lists of source and target vertices). The adjacency matrix for each block is first split into $65,536 \times 65,536$ sub-blocks, in a manner similar to Compressed Sparse Blocks (CSB) [45] and for similar reasons—to provide locality in the vertex bitmaps for both normal and transposed matrix-vector multiplication. Unlike CSB, our approach stores the edge counts in the sub-blocks in a dense array; since most sub-blocks have no or few edges at large scales, a single byte is used for each sub-block size, with an escape code to handle overflows. Two copies of the edge data are stored, each with the edges in the same order. One copy is similar to the edge storage in CSB, storing only the lowest 16 bits of each source or target, i.e., its position within its sub-block. The other copy stores the source and target indices relative to the whole block. The smaller copy is used for frontier exploration; the second copy is used during predecessor map generation and could be removed with a small loss of performance. A format with faster access such as compressed sparse row is not used for the reasons described by Buluç and Gilbert in their work on hypersparse matrices [36]: there are so few edges in a block relative to the number of vertices that CSR would be less efficient than coordinate format. Some of the advantages of CSR for multiplying by very sparse vectors are obtained by skipping sub-blocks corresponding to empty parts of the input vector.

c) June 2011 Version: The June 2011 implementation of BFS on BG/P uses a different edge storage scheme, designed for space rather than access efficiency. In this approach, the edges within a block are first encoded into 64-bit words by concatenating the indices of the source and target vertices within the block. First, they are each padded to the correct number of bits based on the size of the block. Then, these values are sorted for the block. Finally, the lower 32 bits of each value is stored, along with the number (encoded in unary) of edges with each value for the high 32 bits.

Algorithm 4: Blue Gene/P.

Input: *Src*: each edge's source within sub-block;
Tgt: each edge's target within sub-block;
SubblockSize: number of vertices in a sub-block;
SubblockCounts: edge count for each sub-block;
In: frontier.
Output: *Marks*: edge marks;
Out: visited vertices.
Data: *i*: position in list of edges;
bsrc, *btgt*: sub-block coordinates;
s, *t*: source and target of current edge;
notEmpty: set of non-empty sub-blocks in *In*.

```
1  function ExploreFrontier ()
2  begin
3      Out ← Vis ;
4      Marks ← ∅ ;
5      i ← 0 ;
6      notEmpty ← {v/SubblockSize | v ∈ In} ;
7      for ⟨bsrc, btgt⟩ ∈ sub-blocks of this block do
8          if bsrc ∈ notEmpty then
9              for j from 0 to SubblockCounts(bsrc, btgt) do
10                 s ← bsrc × SubblockSize + Src(i + j) ;
11                 t ← btgt × SubblockSize + Tgt(i + j) ;
12                 if s ∈ In ∧ t ∉ Out then
13                     Marks ← Marks ∪ {i + j} ;
14                     Out ← Out ∪ {t} ;
15         i ← i + SubblockCounts(bsrc, btgt) ;
16     return (Out, Marks) ;
17 function WritePred ()
18 begin
19     foreach n ∈ Marks_{i,j} do
20         s ← Src(n) ;
21         t ← Tgt(n) ;
22         if t ∈ Assigned_{i,j} then
23             owner ← Owner1D(tgt) ;
24             // MPI_Alltoallv and manual array update.
25             P(owner, tgt) ← src ;
```

2) Frontier Exploration: On BG/P, frontier exploration is similar to that used on BG/Q, except that an extra output array is produced. Whenever a new vertex is discovered as the result of processing a particular edge, that edge is marked. These marks are stored in a bitmap with one element per directed edge. The basic operation is otherwise similar: a matrix-vector multiplication is performed on each local part of the graph and a dense bitmap (the incoming queue); the result is then accumulated into a new dense bitmap (the outgoing queue,

initialized with the set of all vertices visited so far to prevent them from being rediscovered). The algorithm used is shown in Algorithm 4, labeled ExploreFrontier.

As each block of the adjacency matrix is stored in coordinate format, the multiplication algorithm is straightforward, except that sub-blocks of the matrix are skipped when all the corresponding elements of the incoming queue are absent; the dense matrix of sub-block element counts is used to determine how many elements to skip.

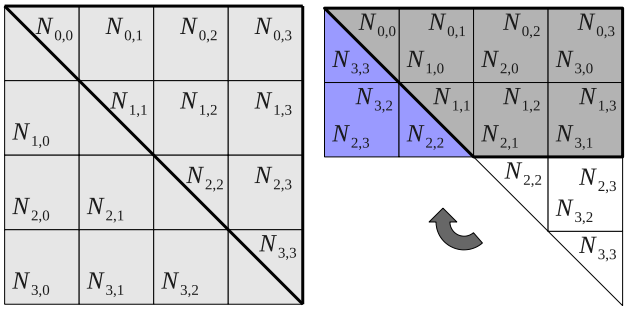
All threads participate in frontier exploration; the work is distributed by chunks of the outgoing queue, allowing each queue element and edge mark to be written from only one thread. This removes the need for atomic operations or thread synchronization during this step of the algorithm. Each thread iterates those edge blocks that correspond to its part of the outgoing queue as per the algorithm described above. Since only half of the symmetric adjacency matrix is stored, each block must have both itself and its transposed multiplied by (possibly different parts of) the incoming queue. Separate sets of edge marks are kept for the normal and transposed multiplications. Only non-transposed multiplication is shown here; the transposed case is analogous.

3) Waves: Waves on Blue Gene/P follow the general description above, using message pipelining to increase concurrency and a tree-based broadcast.

As the queues for the first and last few BFS levels contain very few elements, the bitmaps in the scan and broadcast steps are compressed to reduce data traffic. The bitmaps are first split into chunks for message pipelining, then each bitmap is compressed. The approach used is to keep a bitmap of those words within the chunk which are non-zero and to only send these non-zero words from the chunk; empty chunks are encoded specially. Compression is disabled when it would not reduce data size. Each process in the scan (other than the first in each row) takes the union of its incoming compressed bitmap and its uncompressed set of newly-discovered vertices, producing a compressed output. The broadcast step communicates compressed bitmaps, decompressing each one on each node after it has been received. On a scale 35 graph processed by 32,678 nodes, the average compression ratio was 76.5%, leading to a larger improvement in network traffic since the broadcast step sends the same compressed bitmap many times.

4) Predecessor Map Update: The predecessor map update step on BG/P is similar to that on BG/Q, except that edge marks are used to skip unnecessary predecessor map updates and that MPI collective operations are used to send updates rather than the RDMA used on BG/Q. On BG/P, each process iterates through each of its marked edges, generating an update when the edge's target has been assigned to that process.

Unlike the BG/Q implementation, the BG/P implementation uses *MPI_Alltoallv* to send $\langle vertex, parent, level \rangle$ triples to update the predecessor/level map. The number of discovered vertices for each owning node is first counted and sent (using *MPI_Alltoall*), allowing the updates to be sorted by destination as they are generated. Vertex numbers and predecessor map elements are then sent using two *MPI_Alltoallv* operations. An alternative would be to use one of the approaches for dynamic sparse data exchange from [46].



(a) (b)

Fig. 2: Virtual Node Mapping.

E. Mapping the Virtual Processors

How we map the virtual processors to the actual compute nodes depends on both the architecture and the number of nodes. When the number of nodes m is a power of two but not a power of four, i.e., when $m = 2^{p+1}$ for a positive integer p, both implementations use a similar approach, based on the symmetry of the adjacency matrix. When m is a power of four, on Blue Gene/P we use two MPI processes per physical processor, so that we fall into the previous case, while on Blue Gene/Q we use a slightly more complicated approach that we sketch out later.

Figure 2a shows the grid of virtual nodes $N_{i,j}$; since our graphs are undirected, their adjacency matrices are symmetric, and so $\mathsf{A}_{i,j} = \mathsf{A}_{j,i}^T$: 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 2a "folding" along the diagonal, ending up on top of the upper triangle. Considering that the diagonal blocks $\mathsf{A}_{i,i}$ are symmetric, we only need to store half of them; after the folding this would leave us with $\frac{\ell^2+\ell}{2}$ nodes of which ℓ are on the diagonal and are only half-full. We then proceed to a second transformation, shown in Figure 2b, where half of the diagonal nodes are compacted on the remaining half, "rotating" the lower half of the triangle to form a rectangle of $\frac{\ell^2}{2}$ elements; a modification of this mapping is used on BG/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 non-blocking to avoid sequentializing them because of the overlaps between rows. On Blue Gene/Q, the folded triangle of virtual processes is embedded into the 5-D torus with 1-dilation-neighbors in the triangle become neighbors in the physical system topology. Because Blue Gene/P has only a 3-D torus, the adjacency matrix cannot be mapped onto its topology with 1-dilation. Thus, some pairs of processes that are neighbors in the triangle are several nodes apart in the Blue Gene/P torus; a mapping is used to reduce these distances.

Due to space limitations, we cannot give the full details of the mapping used on Blue Gene/Q when m is a power of four; we note here 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.

VI. Architectural Description

Blue Gene/Q (BG/Q) [47]–[49] is the third generation of highly scalable, power efficient supercomputers in the IBM Blue Gene line, following Blue Gene/L and Blue Gene/P. A full-size 96 rack, 20 petaflops, Blue Gene/Q system called *Sequoia* has recently been delivered to the Lawrence Livermore National Laboratory, while a 48 rack configuration named *Mira* to the Argonne National Laboratory.

In order to achieve a very high processing rate, we have utilized three important optimizations: *1) SPI communication layer*: inter-node communication is implemented at the SPI level, a thin software layer allowing direct access to the injection and reception DMA engines of the network interface. Each thread is guaranteed private injection and reception queues and communication does not require locking. Threads can communicate with very little overhead, of the order of a few hundred nanoseconds, with a base network latency of half a microsecond in the absence of contention. The SPI interface is also capable of delivering several tens of millions of messages per second [49]. *2) L2 Atomics*. Our bitmap-setting algorithms rely heavily on the efficient implementation of a set of atomic operations in the nodes' L2 caches. Each core can issue an atomic operation every other clock cycle, providing a considerable aggregate update rate. *3) Non-blocking collectives*: Barriers and all-reduces are completely overlapped with program execution. These take advantage of the collective acceleration units in the network routers that can execute the main reduce operations at line rate, with combining and broadcast capabilities.

VII. Experimental Results

In this section we present an experimental evaluation of the algorithms described in this paper. Our analysis focuses on the characterization of the algorithms and on how they are influenced by the structural properties of the input graphs, considering the effect of various optimizations and of the scale of the problem.

The Blue Gene/P version of the code is written using MPI for communication and OpenMP for on-node parallelism; most of the code is written in C++ compiled using GCC 4.3.2, with some sequential kernels written in C and compiled with IBM's XL C compiler. The graph generator (based on the reference implementation) is also in C, compiled with GCC. The system used for these experiments is Argonne National Laboratory's Intrepid system; although the system has 40,960 nodes, only up to 32,768 of them are used for our experiments.

The Blue Gene/Q implementation is entirely written in C and uses Pthreads for on-node threading and SPI for communication; the compiler used is GCC 4.4.6. The data in the experimental section were collected on the Blue Gene/Q NNSA/SC Prototype II.

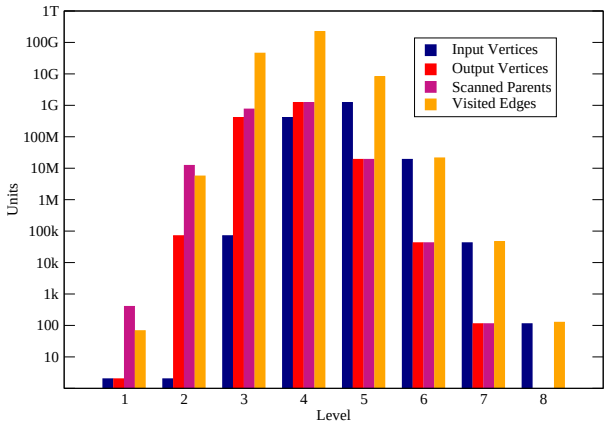


Fig. 3: Graph Statistics on 4096 BG/Q nodes, 2^{32} Vertices.

A. Graph Statistics per Level

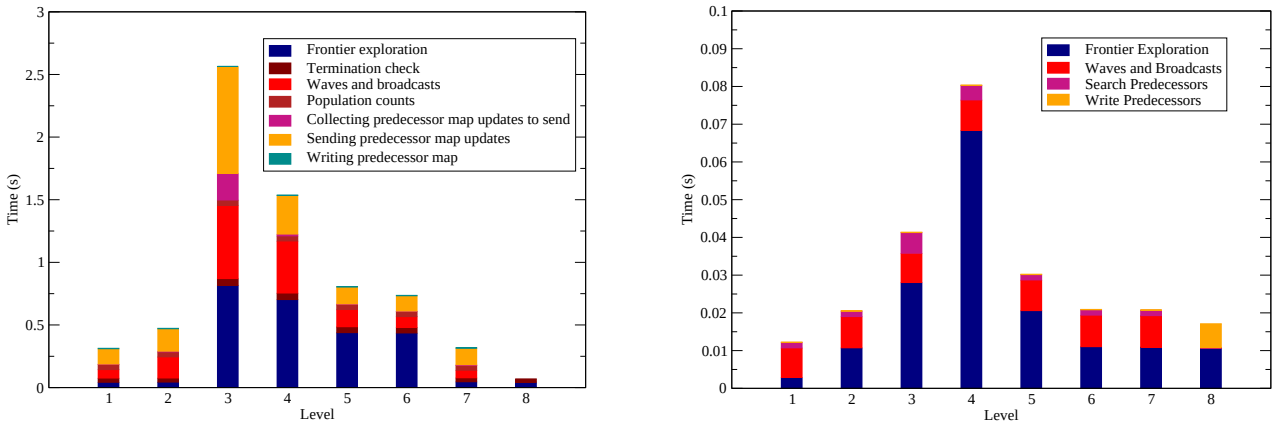
Figure 3 shows various statistics representing the amount of work at each BFS level (note that the Y axis is logarithmic). This figure highlights the fact that the bulk of the vertices are processed in one level in the middle of the BFS, with the numbers of vertices falling off rapidly before and after that. The numbers of visited edges and output vertices show how the visit starts from the periphery of the connected component, quickly moving to the highly connected vertices, that are primarily visited at levels 3 and 4 (the high ratio between visited edges and input vertices is caused by the high degree of the visited vertices). After levels 4 and 5, the visit moves to the rest of the periphery of the connected component, visiting edges with low degrees. The ratio between visited edges and output vertices shows how many of the visited edges link back to the section of the connected component that has already been visited. The last level has no visited edges.

Another important fact that emerges from the plot is how delaying work to the backward search of the predecessors allows us to keep the most iterated loops simple, moving the complexity to a later stage. In level 4, for example, the number of visited edges (hence the number of iterations in ExploreFrontier) is 1.3×10^{11}, but only 1.2×10^{9} new vertices are reached; this requires that we keep the amount of work done per edge very small.

The number of scanned parents in WritePred is usually proportional to the number of newly discovered vertices rather than to the visited edges. Still looking at level four, the number of edges scanned in the backward phase is equal to the number of output vertices up to the second decimal digit. The situation reversed only at the first two levels, but the number of iterations is smaller by orders of magnitude and does not affect the overall performance.

B. Time Breakdown

Figure 4 shows times for various steps of the BFS at each level. Figure 4a shows data for 32,768 BG/P nodes running on a graph with 2^{35} vertices and 2^{39} undirected edges, while



(a) Blue Gene/P (32,768 nodes, 2^{35} vertices)

(b) Blue Gene/Q (4096 nodes, 2^{32} vertices)

Fig. 4: Times by Algorithm Step and BFS Level.

Figure 4b shows data for 4096 BG/Q nodes, 2^{32} vertices, and 2^{36} undirected edges. These times are for a single representative graph and BFS root, and are averages across all nodes.

These figures show that different levels take dramatically different amounts of time, with two levels in the middle of the BFS accounting for over half of the total time on each platform. Some parts of each step take time proportional to the number of vertices at each level, while others are nearly constant. Some phases of the algorithm are skipped at the last level (level 8) as termination of the BFS is detected in the middle of the algorithm on both platforms.

On Blue Gene/P, the three parts of the algorithm accounting for the most time are (i) the frontier exploration, (ii) the waves used for vertex assignment and union of outgoing queues, and (iii) the sending of predecessor map updates. Of these, frontier exploration is purely local, and proportional to the number of vertices and edges processed on each node at each level. The vertex assignment step (implemented using prefix sum and broadcast communications) varies similarly, except with fixed overhead at each level. Without bitmap compression, this step would not take less time at the beginning and end of the search than in the highest-traffic middle part. The final part of the algorithm that takes a substantial amount of time is sending updates corresponding to the predecessor map; this step has a high fixed overhead and also varies based on the number of vertices processed.

On Blue Gene/Q, updating the bitmaps dominates the slower levels – this is a direct consequence of the large number of edges that are visited (e.g., at level four). The communication phase has constant duration, as the only factors influencing it are the size of the bitmaps exchanged and the number of hops to traverse. The figure clearly shows the benefits of splitting the visit into a forward and a backward phase, as the predecessor search depends mostly on the number of output vertices. We noticed that the cost of assigning predecessors using RDMA grows when the number of nodes increases; to counteract this increased cost we switched to buffering the vertex/parent pairs $(v, P(v))$ and sending them at the end using an all-to-all personalized communication.

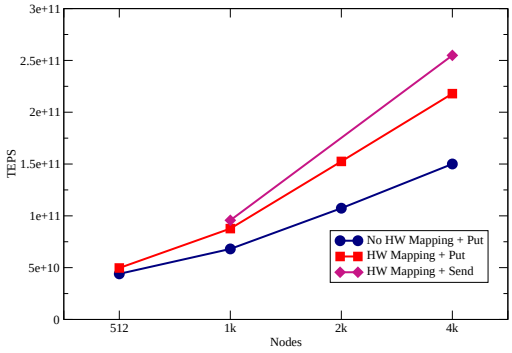


Fig. 5: Blue Gene/Q: Impact of Optimizations.

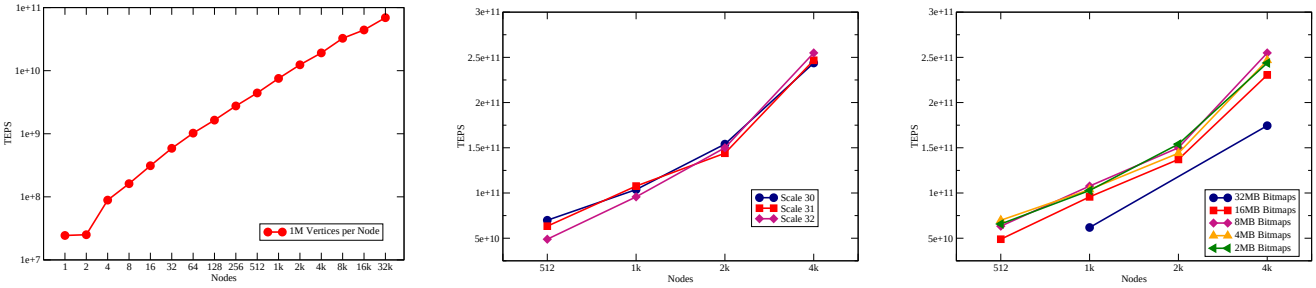
C. Optimizations

Figure 5 shows the impact of various optimizations on the overall Blue Gene/Q TEPS rate for a problem of scale 32 for various node counts. The baseline, the unoptimized version of the code, does not go beyond 150 GTEPS on 4096 nodes, and we can see the huge impact of the hardware mapping, that brings the performance up to more that 200 GTEPS; the benefits of the optimized mapping increase as the number of nodes becomes larger.

An additional optimization that had an impact on the performance was reorganizing the predecessor exchange, making it use a FIFO-based all-to-all communication pattern instead of RDMA remote writes. For small configurations, RDMA proved to be more effective, while on larger configurations the all-to-all was faster. Note that for triangular configurations (512 and 2048 nodes), the data points for the all-to-all version are missing. This optimization relies on the fact that the all-to-all messages are sent to nodes on the same row of the 2-D decomposition. While a similar effect could still be achieved in a triangular configuration, it would require extra bookkeeping and complicate the code, making it unclear if the benefits would be worth the extra effort.

D. Scaling

We evaluate the scalability of our algorithm on BG/P by showing weak scaling, with 2^{20} vertices and 2^{24} undirected edges per node, in Figure 6a. With this form of weak scaling, the memory usage per node increases as the scale becomes



(a) Blue Gene/P, fixed vertex/edge count per node (weak scaling)

(b) Blue Gene/Q, fixed overall vertex/edge count (strong scaling)

(c) Blue Gene/Q, fixed bitmap size per node

Fig. 6: Scaling.

larger. On BG/Q, we present two series of results: a variable number of nodes with a fixed scale (strong scaling, Figure 6b), and a fixed bitmap size (i.e., number of vertices per node in the 2-D decomposition, Figure 6c).

BG/P achieves good weak scaling as the size increases, but performance starts to diminish for the largest node counts. The TEPS rate is proportional to $nodes^{0.78}$ over the whole range of sizes, but is proportional to $nodes^{0.87}$ on the smallest sizes.

With the 2-D decomposition and full-size bitmaps storing $\frac{|V|}{\ell}$ vertices used on BG/Q, linear scaling is not expected, as part of the work is $O(\ell)$, and ℓ grows in proportion to $\sqrt{N}$. Fixing the scale, the pressure on the compute nodes' caches decreases as the number of nodes grows. The performance improvement when going from 2048 to 4096 nodes can be explained as a consequence of using the FIFO-based all-to-all to exchange predecessors.

To keep the workload balanced across different data points, we vary the number of nodes but keep the number of vertices per node constant (i.e., fix $\frac{|V|}{\ell}$), as shown in Figure 6c. A small bitmap size does not allow the system to reach its full potential, and performance drops significantly when the bitmaps exceed the size of the L2 cache (32 MiB). The active working set is determined for the most part by the size of the two output bitmaps (one per range of vertices) that are written randomly in the first phase. Implementation reasons prevented us from using bitmaps of 32 MiB on triangular configurations.

VIII. Conclusion

We have implemented efficient versions of BFS, and the Graph 500 benchmark, for the IBM Blue Gene/P and Blue Gene/Q systems. In particular, we applied several techniques, including bitmap storage, efficient graph representations, 1-dilation mapping, removal of redundant predecessor map updates, and compression during communication to achieve higher performance on large graphs than a standard BFS algorithm.

We believe that techniques similar to those in this paper can be applied to the single-source shortest path (SSSP) problem, especially when edge lengths are positive integers. Algorithms such as Δ-Stepping [50] use iterative searches similar to BFS, but usually with many fewer vertices at each level and a more complicated update operation. A similar type of algorithm to that used for BFS may then be applicable, but the bitmaps involved would be much sparser since there are so many fewer vertices at a given distance from the source.

References

[1] A. Clauset, M. E. J. Newman, and C. Moore, "Finding Community Structure in Very Large Networks," *Physical Review E*, vol. 6, no. 70, p. 066111, December 2004.

[2] J. Duch and A. Arenas, "Community Detection in Complex Networks Using extremal Optimization," *Physical Review E*, vol. 72, January 2005.

[3] M. E. J. Newman, "Detecting Community Structure in Networks," *European Physical Journal B*, vol. 38, pp. 321–330, May 2004.

[4] ——, "Fast Algorithm for Detecting Community Structure in Networks," *Physical Review E*, vol. 69, no. 6, p. 066133, June 2004.

[5] M. E. J. Newman and M. Girvan, "Finding and Evaluating Community Structure in Networks," *Physical Review E*, vol. 69, no. 2, p. 026113, February 2004.

[6] L. Zhang, Y. J. Kim, and D. Manocha, "A Simple Path Non-Existence Algorithm using C-Obstacle Query," in *Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR'06)*, New York City, July 2006.

[7] A. Sud, E. Andersen, S. Curtis, M. C. Lin, and D. Manocha, "Real-time Path Planning for Virtual Agents in Dynamic Environments," in *IEEE Virtual Reality*, Charlotte, NC, March 2007.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of 4th International Conference on Data Mining*, April 2004, pp. 442–446.

[9] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, Feb. 2010, http://jmlr.csail.mit.edu/papers/v11/leskovec10a.html.

[10] Presidents Council of Advisors on Science and Technology (PCAST), "Report to the President and the Congress. Designing a Digital Future: Federally Funded Research and Development in Networking and Information Technology," Executive Office of the President, Tech. Rep., December 2010. [Online]. Available: http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf

[11] A. Buluç and K. Madduri, "Parallel Breadth-First Search on Distributed Memory Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Seattle, WA, November 2011.

[12] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[13] J. Fernández, E. Frachtenberg, and F. Petrini, "BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers," in *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SuperComputing'03)*, Phoenix, AZ, Nov. 2003.

[14] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *Proc. 35th Intl. Conf. on Parallel Processing (ICPP'06)*. Columbus, OH: IEEE Computer Society, August 2006, pp. 523–530.

[15] D. Mizell and K. Maschhoff, "Early Experiences with Large-scale Cray XMT Systems," in *Proc. 24th Intl. Symposium on Parallel & Distributed Processing (IPDPS'09)*. Rome, Italy: IEEE Computer Society, May 2009.

[16] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381–1395, 2008.

[17] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'05)*. Seattle, WA: IEEE Computer Society, 2005.

[18] Y. Xia and V. K. Prasanna, "Topologically Adaptive Parallel Breadth-First Search on Multicore Processors," in *Proc. 21st Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'09)*, Cambridge, MA, November 2009.

[19] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, November 2010.

[20] 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 Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, Nov 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-117.html

[21] D. A. Bader, G. Cong, and J. Feo, "On the Architectural Requirements for Efficient Execution of Graph Algorithms," in *Proc. Intl. Conf. on Parallel Processing (ICPP'05)*, Georg Sverdrups House, University of Oslo, Norway, June 2005.

[22] G. Cong and D. A. Bader, "Designing Irregular Parallel Algorithms with Mutual Exclusion and Lock-free Protocols," *J. Parallel Distrib. Comput.*, vol. 66, pp. 854–866, June 2006, http://dx.doi.org/10.1016/j.jpdc.2005.12.004.

[23] G. Cong, G. Almasi, and V. Saraswat, "Fast PGAS Implementation of Distributed Graph Algorithms," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, New Orleans, LA, 2010.

[24] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *HiPC*, ser. LNCS. Springer Berlin / Heidelberg, 2007, vol. 4873, pp. 197–208, http://dx.doi.org/10.1007/978-3-540-77220-0_21.

[25] E. Saule and Ümit Çatalyürek, "An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture," in *Workshop on Multithreaded Architectures and Applications (MTAAP 2012), in conjunction with the 26th IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 2012.

[26] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in *20th International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*, Galveston Island, TX, October 2011.

[27] L. Luo, M. Wong, and W.-m. Hwu, "An Effective GPU Implementation of Breadth-First Search," in *Design Automation Conference*. New York, NY, USA: ACM, 2010, pp. 52–55, http://doi.acm.org/10.1145/1837274.1837289.

[28] 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 *Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, 2006.

[29] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient Multi-Context Graph Processors," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, ser. LNCS, vol. 2438. Springer Berlin / Heidelberg, 2002, pp. 915–924.

[30] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, February 2011.

[31] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for Easy and Efficient Graph Analysis," in *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, London, UK, March 2012.

[32] 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," *Parallel and Distributed Processing Symposium, International*, pp. 76–, 2007.

[33] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos, "RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine," in *Intl. Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, CA, March 2007.

[34] E. Chow, K. Henderson, and A. Yoo, "Distributed Breadth-First Search with 2-D Partitioning," LLNL, Tech. Rep. UCRL-CONF-210829, 2005.

[35] J. R. Gilbert, S. Reinhardt, and V. Shah, "High-Performance Graph Algorithms from Parallel Sparse Matrices," in *PARA*, 2006, pp. 260–269.

[36] A. Buluç and J. R. Gilbert, "On the Representation and Multiplication of Hypersparse Matrices," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008, http://gauss.cs.ucsb.edu/publication/hypersparse-ipdps08.pdf.

[37] M. Adler and M. Mitzenmacher, "Towards Compressing Web Graphs," in *Data Compression Conference*, 2001, pp. 203–212.

[38] S. Raghavan and H. Garcia-Molina, "Representing Web Graphs," in *IEEE Intl. Conference on Data Engineering*, Mar. 2003.

[39] T. Suel and J. Yuan, "Compressing the Graph Structure of the Web," in *Data Compression Conference*, 2001, pp. 213–222.

[40] P. Boldi and S. Vigna, "The WebGraph Framework I: Compression Techniques," in *World Wide Web Conference*. ACM Press, Jun. 2004, pp. 595–601.

[41] D. K. Blandford, G. E. Blelloch, and I. A. Kash, "Compact Representations of Separable Graphs," in *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[42] ——, "An Experimental Analysis of a Compact Graph Representation," in *Sixth Workshop on Algorithm Engineering and Experiments and First Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALC)*, Jan. 2004, pp. 49–61.

[43] D. Hannah, C. Macdonald, and I. Ounis, "Analysis of Link Graph Compression Techniques," in *Advances in Information Retrieval*, ser. LNCS, 2008, vol. 4956, pp. 596–601, http://dx.doi.org/10.1007/978-3-540-78646-7_62.

[44] C. Seshadhri, A. Pinar, and T. G. Kolda, "An In-depth Study of Stochastic Kronecker Graphs," in *International Conference on Data Mining*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 587–596.

[45] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244, http://doi.acm.org/10.1145/1583991.1584053.

[46] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable Communication Protocols for Dynamic Sparse Data Exchange," in *Principles and Practice of Parallel Programming*. ACM, Jan. 2010, pp. 159–168.

[47] 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, January/February 2012.

[48] 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, March/April 2012.

[49] 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 *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, Seattle, WA, November 2011.

[50] U. Meyer and P. Sanders, "Δ-Stepping: A Parallelizable Shortest Path Algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.