Traversing Trillions of Edges in Real-time: Graph Exploration on Large-scale Parallel Machines

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

***Abstract*—The world of Big Data is changing dramatically right before our eyes—from the amount of data being produced to the way in which it is structured and used. The trend of "big data growth" presents enormous challenges, but it also presents incredible scientific and business opportunities. Together with the data explosion, we are also witnessing a dramatic increase in data processing capabilities, thanks to new powerful parallel computer architectures and more sophisticated algorithms. In this paper we describe the algorithmic design and the optimization techniques that led to the unprecedented processing rate of 15.3 trillion edges per second on 64 thousand BlueGene/Q nodes, that allowed the in-memory exploration of a petabyte-scale graph in just a few seconds. This paper provides insight into our parallelization and optimization techniques. We believe that these techniques can be successfully applied to a broader class of graph algorithms.**

I. Introduction

Every day, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone. This data comes from everywhere: sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals, to name a few.

Effectively sifting through massive data can provide nearly limitless opportunities. Decoding data can help scientists share information from a single, hard-to-find specimen, making new discoveries more possible and frequent. Understanding big data can yield advances in the study of the human brain, offering the prospect, perhaps someday soon, to be able to help doctors and scientists to find a cure for critical diseases.

A. Big Data and Graph Exploration

The nature of big data, often consisting of connections, relations, and interactions among entities, frequently lends itself well to graph-based models. As a consequence the huge demands on the processing systems translate directly into a renewed need for efficient and scalable graph algorithms.

The fields in which graphs are proving themselves as a natural modeling abstraction cover a number of application areas, such as biology, transportation, complex engineered systems (such as the Internet and power grids), communication data, social networks and, more generally, various forms of relational data. The challenges posed by processing huge graphs include, but are not limited to, how to store them, how to keep up with updates that could have very high frequencies, how to monitor those updates, how to classify entities, and in general how to answer queries that have very little structure to exploit for optimization purposes.

Graph traversals, when not a solution in themselves, are often used as basic components of more sophisticated methods. For example, connected component algorithms [1], algorithms to calculate centrality measures, or heuristic search algorithms, such as A* [2], [3] use graph traversal as a building block. With huge data sets many of the locality assumptions current architectures are optimized for are not satisfied anymore. Very often the access patterns are irregular, limiting the beneficial effects of caching or prefetching, and there is very little computation per byte loaded, so that latencies are harder to hide.

This has consequences both in terms of algorithmic and of architectural design. On the software side, algorithmic design has to include parallelism at all levels, from the pipelining of the individual memory accesses to the data distribution on the nodes of a parallel machine. On the hardware side, conventional High Performance Computing (HPC) architectures are showing their limits, being optimized for floating point operations and more regular computation patterns. Even network traffic patterns are different, with small packets sent to random destinations at a very high rate. Conventional HPC networks are built for explicit message passing as the primary communication mechanism, and, as a consequence, efficiently support coarse-grained, rather than fine-grained, communication.

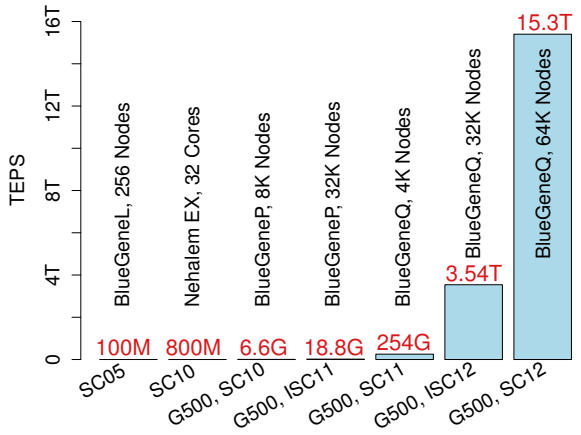

Fig. 1: Evolution of the BFS search and Graph500 on distributed memory machines. The graph exploration performance is expressed in Traversed Edges Per Second (TEPS).

B. Recent Advances in Big Data Exploration

Together with the big data explosion, we have witnessed an almost exponential increase in processing speed of massive-scale graphs. Figure 1 reports the evolution of parallel Breadth-First Search (BFS) from 2005 until the end of 2012, with emphasis on the winning Graph500 submissions from December 2010 to December 2012. The graph shows a giant leap forward, from hundreds of millions of Traversed Edges Per Second (TEPS), to more than 15 trillion TEPS on a graph with trillions of vertices and tens of trillions of edges. Just a few years ago, the general consensus in the scientific community was that graphs were "too hard" for a distributed memory

machine, due to the potential load imbalance, fine grained access granularity of the data structures and poor temporal and spatial locality. The results of the first Graph500 in November 2010 demonstrated that conventional supercomputers can also parallelize sparse problems achieving competitive results. In the following two years the performance improved by three orders of magnitude. This unexpected performance improvement has started a debate on whether this is the end result of better algorithmic design or the exploitation of newer and more scalable architectures, such as BlueGene/Q.

	BFS SC10	Graph500 Nov 2010	Graph500 Jun 2011	Graph500 Nov 2011	Graph500 2012/2013
Algorithms					
2D decomposition	·	⊛	⊛	⊛	·
1D decomposition	·	·	·	·	◎
Direction optimization	·	·	·	·	◎
Load rebalancing	·	·	·	·	◎
Bit-level wavefronts	·	·	◎	◎	·
Bit-level indexes	◎	·	·	·	⊛
Bit matrix-vector multiply	⊛	·	·	·	·
Speculative data scan	·	·	·	•	·
Data Compression	·	⊛	⊛	·	⊛
Wavefront algorithms	·	·	◎	◎	·
Lock-free queues	◎	◎	◎	◎	◎
Loop unrolling	◎	·	·	•	·
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	·	·	·	◎	◎

• ⊛ ◎ Minimal, medium, good measured impact.

TABLE I: Classes and impact of parallelization techniques and optimizations.

In Table I we try to provide a visual classification of the various techniques and algorithms that we have utilized across the various editions of the Graph500. The first observation is that we use many algorithmic techniques and take advantage of architectural improvements: graph algorithms are much more demanding than traditional numerical simulations. We have adopted numerous techniques to deal with a thread-based programming model, bit-manipulation, load imbalance, meta-data optimization, data compression and lock-elimination, just to cite a few. Load imbalance and irregular access pattern proved to be the most challenging problems at single node level. Big data problems can take advantage of a number of optimizations at network level, due to the fine grained communication and the irregular communication pattern. Unlike regular computations where the communication pattern is static with well defined senders and receivers, graph algorithms are more naturally mapped on anonymous mailbox-communication.

C. Contributions

The paper provides the following primary contributions:

- A detailed description of the second generation of BFS algorithms for BlueGene/Q, the submissions of June and November 2012, and June 2013. After a first generation of BlueGene/P and BlueGene/Q [4], we moved from a 2D to a 1D decomposition, and in this paper we describe how, with the new data distribution and a rich set of novel algorithmic optimizations, we were able to obtain much improved scalability and performance. Table II highlights the main differences between the two generations of algorithms.
- A detailed experimental evaluation of the algorithms and design choices adopted on BlueGene/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. A key technique is the design of a lightweight asynchronous communication layer that provides a 5X speedup with respect to an equally optimized MPI implementation.
- In the largest configuration, we have been able to explore a scale 40 R-MAT (Recursive-MATrix) graph with 1 trillion vertices and 32 trillions of undirected edges using 64 thousand BlueGene/Q nodes (4 million threads) in just a few seconds. This is a groundbreaking result that proves that massive-scale graphs can be analyzed in real-time on existing leadership-class supercomputers, such as LLNL's Sequoia and ANL's Mira.
- We believe that many techniques that we have used for BFS can be applied to a larger class of graph algorithms. For this reason the paper characterizes the most effective techniques that we have adopted and their impact on the overall performance. Table II shows the main classes of optimizations and how they are presented in the rest of the paper.

Challenge	Strategy	Section
Algorithms		
Data decomposition	1D/2D	II-B
Search pruning	Direction optimization	II-C
Metadata	Bitmaps/shortcut lists	II-D
Load balancing	Node/thread	II-E,III-C
Comp/comm equalization	Compression	II-F,III-B
Implementation		
Irregular communication	Mailboxing/coalescing	III-B
Synchronization overhead	Lock-free queues	III-B
QoS	Private queues	III-B
Global decisions	Async collectives	III-C

TABLE II: Organization of the paper.

D. Graph 500

As data intensive workloads keep pushing conventional architectures towards their limits, the commonly accepted benchmarks used to rank supercomputers appear increasingly inadequate to characterize their performance. High-Performance Linpack (HPL) and the Top500 make it possible to fairly compare the performance of different machines on traditional HPC workloads, and have played a major role in guiding the design of the supercomputers of the past.

The Graph 500 list (http://www.graph500.org/) was introduced in 2010 as the equivalent to the Top500 list to rank computer performance on data-intensive computing applications. Graph traversals are critical for many such applications, so it is not a coincidence that the Graph 500 has initially focused its attention on BFS. Recursive MATrix (R-MAT) scale-free graphs [5]–[7] have been chosen as the input data because of

their occurrence in many real-world applications. 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.

In detail, the benchmark is specified by the following steps:

1) a graph must be created, using a random Kronecker generator with given coefficients. The size of the graph is specified by the *SCALE* parameter; the graph has 2^{SCALE} vertices and $16 \cdot 2^{SCALE}$ edges. The edge is represented as a set of tuples, each representing one edge.
2) The implementation is free to redistribute the edges across the nodes and to represent the graph in any suitable way. The time taken to convert the graph to the implementation-dependent representation is measured and is part of the benchmark output.
3) The implementation has to perform a number of BFS explorations of the graph, each time starting from a different source vertex. The duration of each exploration has to be measured, and the generated spanning tree has to be validated. Given the number of (undirected) edges explored and the time taken by the exploration each iteration gives a TEPS rate. At the end the metric according to which the systems are ranked is the harmonic mean of the TEPS across all the explorations.

II. Algorithms

A. Preliminaries

We consider a breadth-first search on an unweighted and undirected graph $G = (V, E)$. V is the set of vertices of G, and E is the set of all the pairs (u, v), with $u, v \in V$ such that u and v are connected by an edge in G. Because the graph is undirected, $(u, v) \in E$ implies $(v, u) \in E$.

Given a source vertex $v_s \in V$, a BFS is a full exploration of G that produces a spanning tree of the graph, containing all the edges that can be reached from v_s, and the shortest path from v_s to each one of them. Following the Graph500 specifications, we represent the spanning tree as a predecessor map P, containing the predecessor $P(v)$ of each $v \in V$. We adopt the conventions of assigning its own self as the predecessor of the root, i.e., $P(v_s) = v_s$, and assigning no predecessor at all to the vertices that cannot be reached from v_s. Finally, we define the level of a vertex as its distance from the root in the spanning tree.

Algorithm 1 outlines a sequential BFS implementation. The visit proceeds in steps, examining one BFS level at a time. It uses three sets of vertices to keep track of the state of the visit: *In* contains the vertices that are being explored at the current level (this set is usually referred to as the *frontier* of the exploration), *Out* the ones that can be reached from *In*, and *Vis* the ones reached so far.

In their initial state (lines 1–3), the sets *In* and *Vis* only contain v_s, and $P(v_s)$ is set to the root itself. The loop at line 5 is executed once for each BFS level. The body of the loop (5–13) first resets *Out*, then scans the frontier (8), exploring all the vertices that can be reached from it (9). All the vertices that have not yet been visited are then claimed and added to *Out* and *Vis* (10–13). The visit of each level terminates updating the frontier with the newly discovered vertices (15).

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

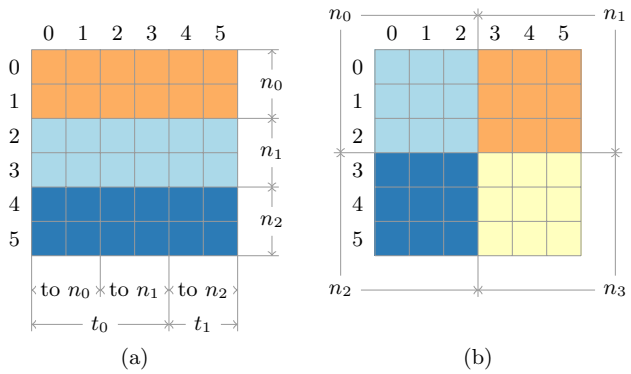

Fig. 2: Adjacency matrix decompositions: 1D (a) and 2D (b).

B. Data Decomposition

Distributed BFS approaches can be classified on the basis of how data is split among the nodes; the two commonly adopted approaches are 1D and 2D decompositions.

1) 1D Decomposition

In a 1D decomposition the vertices are partitioned among the m nodes $n_0, \ldots, n_{m-1}$ in the system. The edges incident on the vertices assigned to a node n_i are all stored on n_i, and visiting an edge (u, v) involves the cooperation and communication between the owners of u and v, which are, in general, different.

The adjacency matrix of a graph G is a square matrix defined as $A = [a_{u,v}]_{u,v \in V}$, with $a_{u,v}$ equal to one if $(u, v) \in E$, zero otherwise. Figure 2a shows an example of 1D decomposition of six vertices among three nodes.

The formulation of a distributed BFS on a 1D decomposition is straightforward, with each edge generating a message from the owner of the source vertex to the owner of the destination. This extreme simplicity is one of the main advantages of a 1D decomposition, but issues like load imbalance and the efficient utilization of network resources need to be addressed in order to obtain good scalability and performance. The next sections describe these issues and our approaches to their solution.

2) 2D Decomposition

A different approach consists in partitioning the edges of the graph among the nodes. This has the same effect of using a 2D block decomposition of the adjacency matrix, with each block of the matrix being assigned to a different node, as shown in Figure 2b. The resulting layout tends to be more balanced, because the longer edge lists are partitioned across multiple nodes, and makes use of more regular communication patterns. The complexity of the approach is its main downside. We went into the details of efficient and scalable 2D implementations in [4] (and other references are given in Section V), therefore we will limit the considerations on 2D decompositions to a minimum.

C. Search Pruning

While in the general case the BFS is implemented visiting all the edges of a graph, on low diameter graphs such as the R-MATs used in Graph 500, it is possible to avoid traversing some of the edges. This is done through a technique, which we will call *direction optimization*, that was introduced in [8], and consists in alternating the traditional top-down BFS with a symmetrical bottom-up search phase. In a bottom-up phase, each vertex that has not yet been reached probes its parent, checking whether it belongs to the current frontier; at some levels of the visit, depending on the topology of the graph, this allows a reduction of one order of magnitude of the edges that need to be considered.

Direction-optimization is an extremely powerful algorithmic technique, but exploiting its full potential requires careful implementation. In a distributed 1D decomposition, a bottom-up phase requires one or two messages for each traversed edge: one for the child to probe the potential parent, and, eventually, one from the parent to try to claim the child. This scheme creates potential loops and contributes to the irregularity of the communication pattern; it is also sensitive to the round-trip delay experienced by the probe message, because a child, once claimed, can stop sending out probes.

Our implementation uses low-latency communication primitives and small messages, to reduce the round-trip times; we use separate FIFOs for probes and responses, to simplify the handling of the messages. Also, partitioning the edge lists of each vertex across multiple threads, and staggering the initial vertex of each thread, allows us to further reduce the number of probes to send, because the last threads to visit a vertex will, with good probability, find that it has already been claimed.

D. Data Structures

After exploring several graph representations, we opted for a compressed adjacency list with a coarse index. A sample graph and the corresponding representation are shown in Figure 3b. Each source vertex is stored once, and is followed by its neighbors. Shortcuts from one source vertex to the following allow us to skip edges that do not need to be visited.

A coarse index, with one entry every 64 source vertices, allows us to skip large portions of the adjacency list during the initial and the final levels of the visit, where only a few vertices are active, and, consequently, a few edges need to be visited. The choice of one entry every 64 vertices allows us to check one full word in the *In* bitmap and skip all the corresponding vertices if none of its bits is set.

We don't need a full 64 bit integer to store the source and destination vertices. The most significant bits of source vertices can be derived from the rank of the node they belong to; the remaining bits are used to store the shortcut, which also serves as an edge count during the traversal. Destination vertices need to be represented in full, but since the maximum scale we support is 40, there are spare bits that we use for load balancing purposes, as we will see shortly.

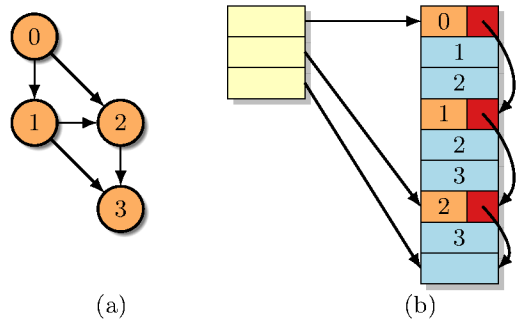

Fig. 3: Edge representation: the graph in (a) is represented by a coarse index and an array of edges (b).

E. Load Balancing

At large scales we experienced load balancing issues, due in the most part to the irregular nature of the graphs being visited, and to the presence of vertices with a huge number of neighbors. We took advantage of the efficient collective operations available on BlueGene/Q, and implemented a load-balancing infrastructure that handles vertices with long edge lists globally, only using local bitmap updates and collectives for coordination.

Denoting with LB the number of vertices we want to rebalance, during the data distribution phase we find the set H containing the LB vertices with the highest degrees, using all-reduces and histogramming on $O(LB)$ storage. The vertices of H do not follow the 1D distribution of the other vertices, but their edges are distributed among all the nodes in the system, and are visited globally (the details of the global visit are detailed in III-C, together with the interactions with the plain 1D visit).

The global distribution of the edges adjacent to H offers an additional optimization opportunity, making it possible to place them in a way that doesn't require communication when they are visited. We do this assigning the edge (u, v), with $u \in H$, to the 1D-owner of v. The subset of the frontier carrying information on the vertices in H is shared among all the nodes, and updated using collective operations, while the visit of the edges adjacent to H is carried out locally.

F. Computation and Communication

The performance of a search algorithm depends at least in part on the efficiency with which it utilizes the computation and communication resources available. Our design, based on a fully asynchronous communication paradigm, is centered around the complete overlap of communication and computation.

A technique that improved the balance between these two components of the search was message compression. Based on the observation that, on large systems, where the bisection bandwidth of the network becomes the limiting factor, we used the available processing power to compress the messages sent, reducing the communication time.

III. Implementation

A. Optimized BFS

The pseudocode for our optimized BFS is shown in Algorithm 2. Vertex sets are represented as bitmaps and partitioned among the nodes, we indicate the portion of the bitmap B assigned to node i with B_i. P is partitioned in the same way, so that each node is responsible for the predecessors of the vertices it owns.

The visit begins with the initialization of In, Vis, and P on the owner of the root vertex v_s. Every node then enters a loop,

and on each iteration explores a new level. The loop goes on until the frontier In becomes (globally) empty. On each step the direction of the visit is decided according to a heuristic function `CalcDirection()`, that counts how many vertices have not been visited yet, and how many edges are reachable from the frontier; if the edges reachable from the frontier outnumber the ones to be visited we switch to a bottom-up visit.

The body of the functions taking care of the actual visit is shown in Algorithm 3. `ForwardStep()` performs a top-down visit step, scanning the frontier and sending one message per edge to the owner of the visited neighbor. `BackwardStep()` goes the other way round, scanning the complement of the frontier and trying to reach for a possible parent. Algorithm 3 also shows the handler for the received messages. In the actual implementation the two receive paths are separated, and messages in the two directions are sent using different logical channels.

When receiving a forward packet we check if the destination vertex has already been visited, if not we update our bitmaps and the predecessor tree. When receiving a backward packet we check if the source vertex belongs to the frontier, and if this is the case, we send back to the owner of the source vertex a forward message, that is interpreted in the same way as any other equivalent message.

Algorithm 2: Parallel 1D BFS.

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

$In_r \leftarrow \begin{cases} \{v_s\} & \text{if } v_s \in R_r \\ \emptyset & \text{otherwise} \end{cases}$;
$Vis_r \leftarrow \{v_s\}$;
$P(v) \leftarrow \perp \forall v \in V$;
if $v_s \in R_r$ **then** $P(v_s) \leftarrow v_s$;
while $In \neq \emptyset$ **do**
  $dir \leftarrow$ `CalcDirection()` ;
  if $dir = FORWARD$ **then**
    `ForwardStep()` ;
  else
    `BackwardStep()` ;
  $In \leftarrow Out$;

B. Communication

From a logical point of view, a 1D algorithm sends edges individually. This proves to be extremely inefficient, due to the high overhead it imposes on the networking subsystem, as shown in Figure 4. To prevent this degradation of the network performance, we pack together all the edges that would be sent to each destination separately, queueing them into an intermediate buffer as long as they are produced by the visit. We keep one open (i.e., accepting new edges) packet per destination, and we allow multiple packets to be in flight at any given time.

This buffering scheme is hidden behind the interface shown in the pseudocode: a send operation doesn't actually send individual messages, but queues the edges as long they fit in the current packet, and eventually hands it down to the network interface. Before starting to write to a new packet, **send** checks that the memory occupied by the buffer space is not still in use by a previously sent packet, and, if necessary waits until that packet has left the node and the buffer is available.

Algorithm 3: Parallel 1D BFS: support functions.

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

1 **function** `ForwardStep` ()
2 **begin**
3 **for** $u \in In_n$ **do**
4 **for** $v : (u, v) \in E_n$ **do**
5 **send** $(u, v, FORWARD)$ **to** `Owner` (v) ;
6 **function** `BackwardStep` ()
7 **begin**
8 **for** $v \in \neg Vis_n$ **do**
9 **for** $u : (u, v) \in E_n$ **do**
10 **send** $(u, v, BACKWARD)$ **to** `Owner` (u) ;
11 **function** `Receive` (u, v, dir)
12 **begin**
13 **if** $dir = FORWARD$ **then**
14 **if** $v \notin Vis_n$ **then**
15 $Vis_n \leftarrow Vis_n \cup \{v\}$;
16 $Out_n \leftarrow Out_n \cup \{v\}$;
17 $P(v) \leftarrow u$;
18 **else**
19 // *Backward.*
20 **if** $u \in In_n$ **then**
21 **send** $(u, v, FORWARD)$ **to** `Owner` (v) ;

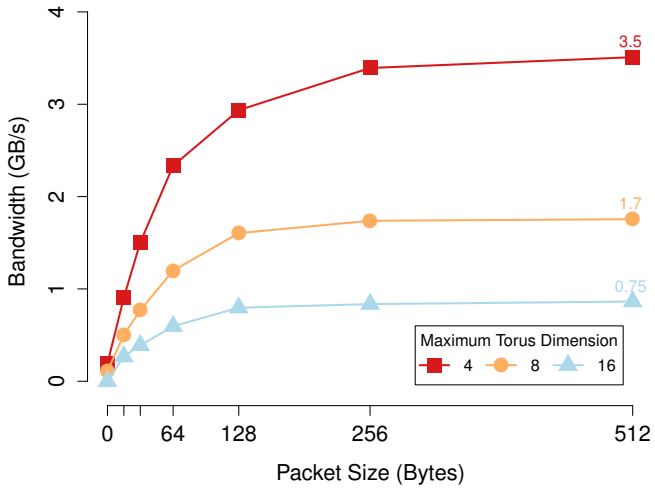

Fig. 4: Network bandwidth as a function of packet size, for various torus dimensions.

Going back to Figure 2a, we note how the edge list of each vertex is partitioned among the threads of the node that owns the vertex. This has beneficial effects on load balancing and on direction optimization, as we have already pointed out; it also allows us to avoid locks, atomic operations, and memory synchronization on the locations occupied by the buffers when the packets are built. In the example in the figure, each of the three nodes runs two threads. The adjacency matrix is sliced into $m = 3$ rectangles, each of dimensions $|V| \times \frac{|V|}{m}$, and each slice is split equally among the threads on the node, with each thread working on edges belonging to a partition of vertices, such that to different threads correspond different remote nodes.

Each block $A_{i,j}$ of the adjacency matrix identifies a partition of size $\frac{|V|}{m}$ of the vertex set, and a partition of their neighbors, of size $\frac{|V|}{|T|}$. Each block is assigned to a different thread, and all the threads can handle messaging independently of each other.

The compression mechanism sends the edges using a differential encoding scheme. Each packet contains a header specifying where the packet comes from, how many bytes of the packet are actually in use, and a base vertex. Encoding each vertex requires at most six bytes, 24 bits for the local part of the destination vertex (whose most significant bits can be retrieved from the rank of the destination), and 24 bits for the local part of the source vertex (whose most significant bits can be retrieved from the rank of the source, specified in the header). The messages sent in the bottom-up phase when direction optimization is in use are encoded with the same scheme, swapping the roles of source and destination vertices.

When possible we used a reduced format, using only four bytes, specifying the source vertex in full (24 bits) and using 7 bits to encode the difference between the destination vertex and the one immediately preceding it. For the first one in the packet we use the base vertex from the header.

C. Load Balancing

Algorithm 4 shows the parts of *ForwardStep()* and *BackwardStep()* that deal with load balancing. They extend the implementations given in Algorithm 3 for generic vertices, and operate only on the vertices belonging to the load-balanced set H.

We use the superscript H to denote the versions of the data structures dedicated to load balancing. All the nodes share the same contents for In^H; a forward visit is logically equivalent to the generic case, and distinguishes between two cases. When an edge joining two heavy vertices is visited (lines 7–9) the local copies of Vis_n^H, Out_n^H, and P_n^H are updated. In the case of an edge joining a heavy and a light vertex (lines 11–13) the light vertex is guaranteed to be local, by virtue of the distribution of the load-balanced edges.

The Vis^H and Out^H bitmaps are synchronized across the nodes at the end of each level, while the local copies of P^H are synchronized with an all-reduce at the end of the visit, selecting only one of the candidate predecessors. The backward step behaves in a similar way.

Not shown in Algorithm 2, the generic visit needs to take care of the case of light vertices with heavy neighbors; in this case no message is sent and the local Vis^H, Out^H, and P^H are updated. The termination check is extended to consider In^H.

As anticipated in Section II-D, we only need 40 bits to store the vertex IDs; we use the available 24 bits to identify heavy vertices: if the neighbor is heavy its highest bits are set to the index into H, making it possible to identify in constant time the vertices subject to load balancing.

IV. Experimental Results

In this section we present an experimental evaluation of our BFS algorithm, together with some of the measurements that guided its design and development.

A. The Input Data

The topology of the input graph determines the kind of work that needs to be done in order to visit it. In [4] we showed how, in a 2D implementation, the redundant updates to the *Out* and *Vis* sets could be exploited to reduce the amount of work done for each edge. In our 1D implementation we make use of the direction-optimizing BFS introduced in [8], applying a similar principle to reduce the number of edges visited, and, consequently, messages exchanged.

Algorithm 4: Parallel 1D BFS: load-balancing support functions.

Input: $G = (V, E)$: graph representation;
n: rank;
In^H: current level heavy input vertices;
Out_n^H: current level local heavy output vertices;
Vis_n^H: vertices already visited.
Output: P_n^H: local predecessor map for heavy vertices;
P: predecessor map.

```
1  function ForwardStep ()
2  begin
3      Out_n^H ← ∅ ;
4      for u ∈ In^H do
5          for v : (u, v) ∈ E_n do
6              if v ∈ H then
7                  Vis_n^H ← Vis_n^H ∪ {v} ;
8                  Out_n^H ← Out_n^H ∪ {v} ;
9                  P_n^H(v) ← u ;
10             else
11                 Vis_n ← Vis_n ∪ {v} ;
12                 Out_n ← Out_n ∪ {v} ;
13                 P(v) ← u ;
14     allreduce (Vis_n^H, OR) ;
15     allreduce (Out_n^H, OR) ;
16     In^H ← Out_n^H ;
17 function BackwardStep ()
18 begin
19     Out_n^H ← ∅ ;
20     for v ∈ ¬Vis_n^H do
21         for u : (u, v) ∈ E_n do
22             if u ∈ H then
23                 if u ∈ In_n^H then
24                     Vis_n^H ← Vis_n^H ∪ {v} ;
25                     Out_n^H ← Out_n^H ∪ {v} ;
26                     P_n^H(v) ← u ;
27                 else if u ∈ In_n then
28                     Vis_n ← Vis_n ∪ {v} ;
29                     Out_n ← Out_n ∪ {v} ;
30                     P(v) ← u ;
31     allreduce (Vis_n^H, OR) ;
32     allreduce (Out_n^H, OR) ;
33     In^H ← Out_n^H ;
```

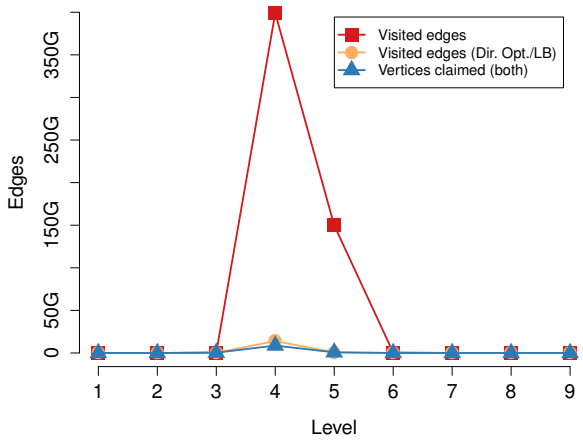

Fig. 5: Graph properties at each exploration level.

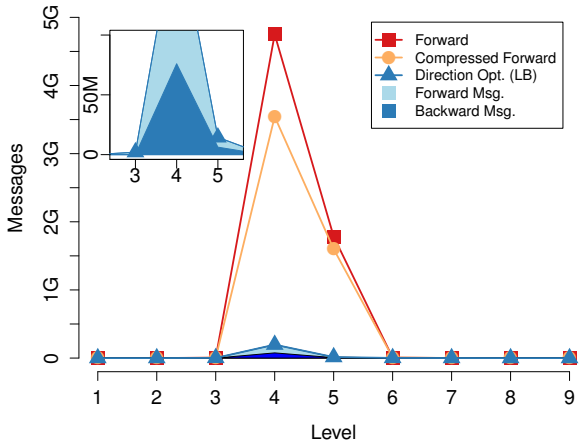

Fig. 6: Message counts at each level.

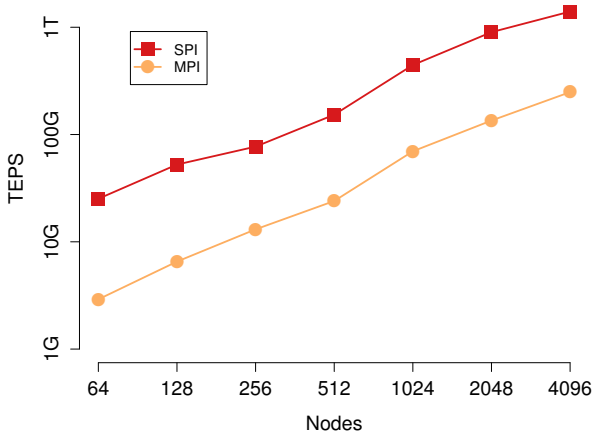

Fig. 7: MPI vs. SPI performance.

The first two or three levels of the BFS visit a relatively small number of vertices, with increasing degree. At the end of level three usually most of the vertices have been reached, and in a traditional BFS all their edges have to be checked. This leads to a fourth level exploring the majority of the edges of the graph, as shown in Figure 5, with most of those edges pointing back to vertices that have already been visited.

The brilliant intuition behind the direction-optimizing BFS is that, after entering the big connected component of the graph, a visit that starts from the vertices that have yet to be visited and stops as soon as a predecessor is found leads to greatly reduced work.

Figure 5 also shows the difference between what happens before and after applying the direction-optimizing technique. Most of the edges are encountered in the central levels of the explorations (levels three or four with the scales we used), determining the amount of work and data exchange necessary at those levels.

The edge check reduction we obtain is similar to what shown in [8], with roughly 20 times fewer edges visited than a plain BFS. Considering that we need to visit at least one edge for each vertex in the spanning tree, we can see from the figure how close we are to the theoretical minimum, with a very small number of redundant edges visited. As a consequence, the number of messages that have to be exchanged decreases drastically, as shown in Figure 6.

B. Impact of Optimizations

Our algorithmic design is centered on low-latency asynchronous messaging, which, on BlueGene/Q, translates to direct access to the SPI (System Programming Interface[1]) interface. It is not easy to quantify the impact of a design choice so pervasive to the architecture of the application, but in Figure 7 we try to estimate it. We implemented an MPI version of the code, using a bulk-synchronous two-phase approach. At each BFS level the code performs the following steps:

1) each thread scans its input vertices and counts how many messages it would send to each destination;
2) the nodes exchange the message counts via an all-to-all and allocate the destination buffers;
3) each thread scans again its input vertices, writing the edges to be sent into the source buffers;
4) the nodes exchange the edges using an all-to-all;

[1]The architecture's interface to the hardware communication FIFOs.

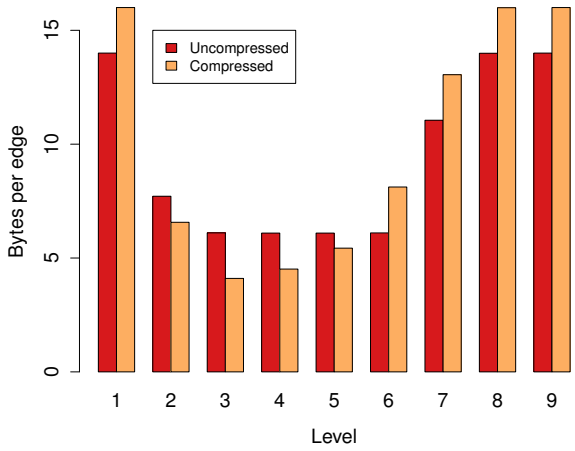

Fig. 8: Compression: average no. of bytes sent per edge.

5) each thread processes the messages coming from a subset of the possible sources.

The SPI and MPI implementations are compared using the same degree of single-node optimizations; in particular both implement direction optimization. Further optimization of the MPI code would have required a more substantial redesign of the application (e.g. introducing multiple passes on smaller buffers, or double-buffering the communication). The maximum scale that fits in one node with MPI is 23, one less than its SPI counterpart, because of the requirement for the final all-to-all to store at the same time the incoming and the outgoing messages. The SPI implementation outperforms the MPI one by a factor of about 5.

Direction-optimization is probably the most effective technique we have employed, but we went a bit further and explored other optimizations that could help us obtaining a better performance.

Figure 6 shows the effect of edge compression on the number of messages sent and how many of the messages are sent in each direction, comparing them to the values observed without edge compression and direction optimization.

Even without direction optimization, on small system configurations, the workload is CPU bound, due to the higher bisection bandwidth. On larger systems, especially on the central levels, where more edges are exchanged, compression helps improving performance. Figure 8 shows the bytes used on average for

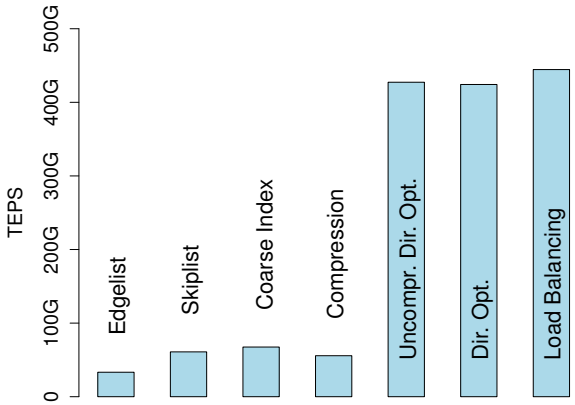

Fig. 9: Impact of optimizations.

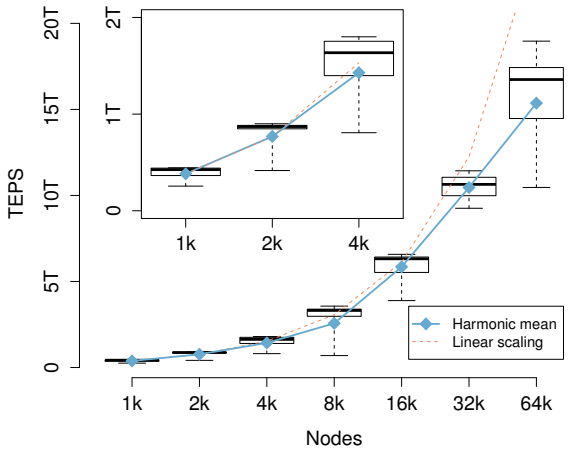

Fig. 10: Weak scaling.

each edge sent on the network. The compression mechanism relies on the edges' endpoints to be consecutive to work well, this is why at lightly loaded levels the 8 byte header dominated the per-edge cost. We should note that, when no compression is used, each edge takes 6 bytes, while, with compression, edges take 4 or 8 bytes: this explains why, in absence of compression, we may send fewer bytes per edge. As soon as we enter level three or four, where we have substantially more traffic flowing through the network, compression starts working as expected, and the average is between four and five bytes per edge.

Figure 9 shows the impact of each of the optimization techniques we have used on 1024 nodes. It is clear how direction optimization is responsible for the largest speedup, almost an order of magnitude more than what other optimizations can do. Due to the scale and the machine size available at the time of writing, the values for compression and load balancing are not representative of their actual impact at scale. A machine size of 1024 nodes implies a maximum dimension for the 5D torus of eight, resulting in a bisection bandwidth two times higher than that of higher sizes: this reduces the cycles available for compression, and the CPU load becomes the bottleneck. With respect to imbalance, at scale 34 the longest edge lists are not yet long enough to create substantial imbalance in the 2^{24} vertex partitions that we use.

C. Scaling

Figure 10 shows the performance measured on system configurations ranging from one rack (1024 nodes) to 64 racks (65,536 nodes). The box plots show the minimum, maximum, median, first, and third quartile values (the ones reported in the Graph500 submissions), together with the harmonic means (the blue line). For reference, the ideal scaling is reported as a dashed line. The scaling up to 16 racks is quite close to ideal, while it gets worse on larger configurations. One effect that we noticed, and that we tried to mitigate in part with our load balancing scheme, is the increasing imbalance in the edge distributions as the scale grows higher.

The other effect we noticed is how the minimum values have a stronger impact on the overall performance. Those values tend to be determined by the topology of the graph, with the worst results given by graphs where the third level explores a great amount of edges. Neither a fully top-down or fully bottom-up approach works well. We didn't experiment extensively with switching from a top-down to a bottom-up visit, because of the additional bookkeeping it requires, but that could be a possible way of dealing with the problem.

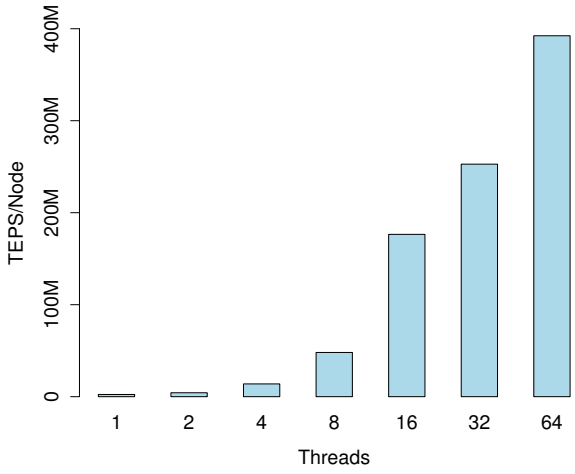

Fig. 11: Thread scaling.

Figure 11 shows how multi-threading helps increasing performance. Using the same configuration (scale 34 on 1024 nodes), the figure shows performance using an increasing number of threads per node. Each BlueGene/Q node has 16 A2 cores, each one capable of supporting up to four hardware threads. This, together with the thread allocation policies of BlueGene/Q, implies that hardware multi-threading enters the picture only after 16 threads in figure. We can see how it helps, in particular to hide the latencies from the memory subsystem.

D. 1D vs. 2D

Figure 12 shows a comparison between the weak scaling performance of our previous 2D BlueGene/Q implementation [4] with the current 1D. The 1D version is faster on all configurations. At the largest size for which our 2D code was optimized, 4096 nodes, the 1D version reaches 1.4 TTEPS, more than 5x the 254 GTEPS obtained by the 2D implementation, and it does so on a larger problem scale (36, against the 32 of the 2D case).

Our 2D implementation doesn't take advantage of direction optimization, and does not implement techniques such as compression and overlapping of communication and computation. Scaling is limited by the cache thrashing caused by the bitmaps growing at a rate $O(\frac{|V|}{\sqrt{m}})$, which leads them to quickly exceed the L2 cache size.

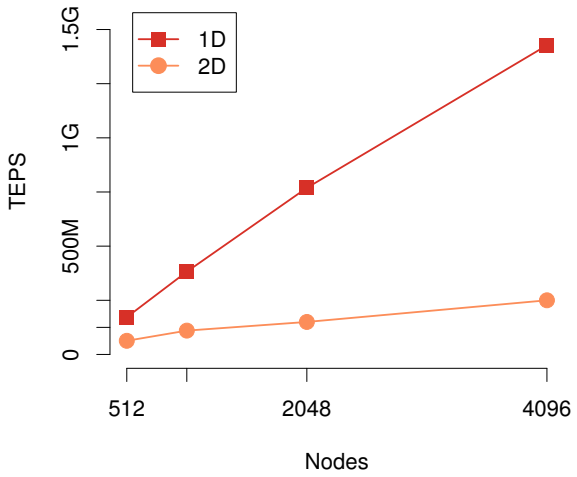

Fig. 12: Weak scaling: 1D and 2D.

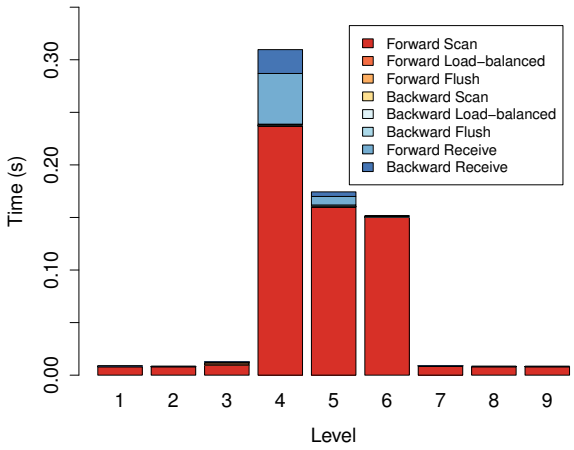

Fig. 13: Level breakdown: time taken by each phase of the visit.

E. Level Breakdown

Figure 13 shows how time is spent during each BFS level. The first observation is that the central levels account for most of the visit time. The composition of top-down and bottom-up levels is different, nonetheless scanning the edges takes the most time in both cases. Due to the almost complete overlap of communication and computation we can achieve, thanks to the use of the low level SPI primitives, the receive times shouldn't be summed to the scan time. We see that, in the levels dominated by the communication time, the flush times show up (this is because, at the end of the computation we wait for all the messages to be sent, though, most likely, the computation would already have had to wait for the communication when trying to write to buffers yet to be emptied). Another important consideration is the impact of the efficiency of the full scan of the vertices at the lightly loaded levels. The amount of useful work done at those levels is very little, the input bitmap is basically empty, but an improper edge representation could force us to scan portions of the edge list anyway (e.g., without an index), and that cost would have to be payed at four or five levels.

The effect of different edge representations on performance is shown in Figure 14. The three representations considered are:

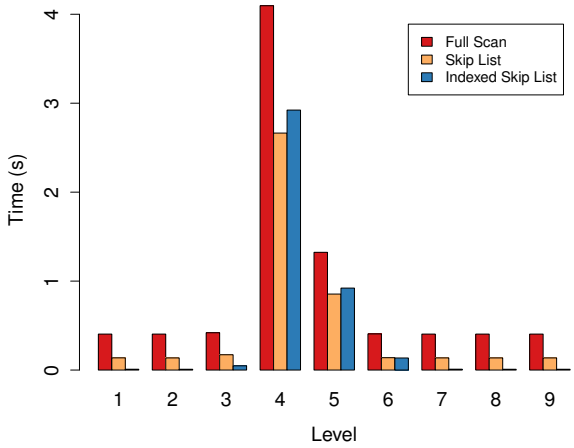

Fig. 14: Impact of graph representation on the level times.

1) a full edge list, representing the edges as pair of vertices; 2) a *skip list*[2], i.e., the data structure represented in Figure 3, but without an index, and 3) the same representation, with the coarse index described in Section II-D. The measurements are taken disabling direction-optimization.

In the central levels the version using a coarse index performs slightly worse than the version not using it. This difference, which is further reduced when direction-optimization is employed, is compensated by the higher fixed cost of the no-index version in the light levels; although small, that fixed cost is paid multiple times.

V. Related Work

The optimization of graph traversal on parallel and distributed systems has received a great deal of attention recently. In this section we to provide an overview of the works that are closer to what is presented in this paper; for a more complete synopsis of the existing literature the reader is referred to [10].

The works we are most indebted to are [8], and previously [11], which introduced the concept of direction optimization. We have already discussed the technique and its benefits, it is worth noting that it does not apply only to R-MAT graphs, but, with varying degrees of efficacy, to different classes of low diameter graphs.

In [12] the authors describe their energy-efficient approach to graph traversal. In common with them we use fast data compression and overlapping of communication and computation, but their focus is mainly on single-node performance, while our aim is optimizing the visit on large distributed machines. In [13] the authors provide an analysis of the challenges involved in designing algorithms for large scale-free graphs, and present their solution, based on a 1D decomposition extended with ghost vertices. Their load balancing scheme splits the edge list of each heavy vertex across a group of nodes who cooperate during the traversal of the edges; compared to our load balancing scheme, theirs requires finer grained synchronization among the nodes in the groups responsible for each heavy vertex, and does not take advantage of collective operations. On the other hand it is not limited by the size of the collective and can support a larger number of rebalanced vertices (something that could be useful with the smaller partition sizes available on BG/P, the reference architecture for the paper). They present a scaling study on both DRAM and external memory.

[2]We refer to it as a skip list because it could be seen as a simplified version of the data structure introduced in [9].

Much of the work on large distributed systems has been based on 2D decompositions. They were introduced in [14], where the authors also give a comparison between 1D and 2D partitioning. Other works using 2D partitioning include the results on BlueGene/L described in [15] and the already mentioned [10]. One of the problems with using a 2D decomposition is that the representation may become hypersparse, i.e., there may be nodes with more edges assigned than vertices. In [16], as part of their work on sparse linear algebra, the authors introduce a compression scheme suitable for dealing with the issue.

Another line of research has focused on single-node system shared memory graph algorithms, using CPUs, as in the case of [17]–[20], or manycore architectures, like [21]–[23]. We make extensive use of bitmaps to represent vertex sets; to the best of our knowledge this technique was first introduced in [24] in a GPGPU context.

Other interesting work has been done on multithreaded shared memory architectures, like the Cray MTA-2 [25], and the Cray XMT [26], as well as on specialized hardware designs [27], [28].

VI. Conclusion

We presented an efficient and highly scalable BFS implementation, making use of a combination of state-of-the-art techniques, such as direction optimization, and careful design targeting the needs and capabilities of the BlueGene/Q platform.

VII. Acknowledgement

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. IBM also wishes to acknowledge the National Nuclear Security Administrations Advanced Simulation and Computing Program (ASC) and Lawrence Livermore National Laboratorys Livermore Computing (LC) department for allowing the execution of part of the experiments presented here on the Sequoia supercomputer. We would like to thank Susan Coghlan, Kalyan Kumaran, and Ray Loy at ANL and David Fox and Scott Futral at LLNL for their help in the access to Mira and Sequoia.

References

[1] 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.

[2] 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.

[3] 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.

[4] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 13:1–13:12.

[5] 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.

[6] 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.

[7] 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.

[8] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-first Search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10.

[9] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990.

[10] 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.

[11] 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.

[12] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale Energy-efficient Graph Traversal: a Path to Efficient Data-intensive Supercomputing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 14:1–14:11.

[13] R. Pearce, M. Gokhale, and N. M. Amato, "Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory," in *Proc. 24th Intl. Symposium on Parallel & Distributed Processing (IPDPS'13)*, May 2013.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[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] 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.

[26] 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.

[27] 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.

[28] 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.