Context Matters: Distributed Graph Algorithms and Runtime Systems

A Case Study of Distributed Graph Traversals

Jesun Sahariar Firoz
jsfiroz@iu.edu

Thejaka Amila Kanewala
thejkane@iu.edu

Marcin Zalewski
zalewski@iu.edu

Martina Barnas
mbarnas@indiana.edu

Andrew Lumsdaine
lums@iu.edu

Center for Research in Extreme Scale Technologies (CREST)
Indiana University, Bloomington, IN, USA

ABSTRACT

The increasing complexity of the software/hardware stack of modern supercomputers makes understanding the performance of the modern massive-scale codes difficult. Distributed graph algorithms (DGAs) are at the forefront of that complexity, pushing the envelope with their massive irregularity and data dependency. We analyze the existing body of research on DGAs to assess how technical contributions are linked to experimental performance results in the field. We distinguish *algorithm-level* contributions related to graph problems from *runtime-level* concerns related to communication, scheduling, and other low-level features necessary to make distributed algorithms work. We show that the runtime is an integral part of DGAs' experimental results. We argue that a DGA can only be fully understood as a combination of these two aspects and that detailed reporting of runtime details must become an integral part of scientific standard in the field if results are to be truly understandable and interpretable. Based on our analysis of the field, we provide a template for reporting the runtime details of DGA results, and we further motivate the importance of these details by discussing in detail how seemingly minor runtime changes can make or break a DGA.

CCS Concepts

•General and reference → Experimentation; Design;
•Theory of computation → Graph algorithms analysis; Shortest paths; Distributed algorithms;

Keywords

Distributed graph algorithms; Single source shortest paths; Runtimes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PASC '16, June 08–10, 2016, Lausanne, Switzerland

© 2016 ACM. ISBN 978-1-4503-4126-4/16/06...\$15.00

DOI: <http://dx.doi.org/10.1145/2929908.2929917>

1. INTRODUCTION

Large, irregular applications are gaining recognition as the future challenge in parallel computing. This is reflected by the Graph500 benchmark [22], the subject of which is the prototypical irregular problem of graph traversal. Graph traversal is a basic building block of many other graph algorithms. The current Graph500 benchmark is based on breadth-first search (BFS) with a proposal to extend the benchmark with single-source shortest paths (SSSP). In this paper, we concentrate on BFS and SSSP for the same reasons, i.e., as representatives of a class of irregular graph problems.

Research on distributed graph algorithms (DGA) is an emerging and active field. New algorithms, new approaches to distribute the data, and new performance results appear at most major distributed computing conferences. The Graph500 benchmark bears witness to the progress with the best results progressing from seven GTEPS (billions of traversed edges per second) in 2010 to 23 TTEPS in 2014. Many new algorithmic techniques have been developed, e.g., direction optimization [4,5], pruning [9], k-level asynchronous algorithm [17], hybrid algorithms [9], and distributed control [31]. A practitioner faces a multitude of published approaches, which are often vague on low-level details of implementations.

However, for DGAs, the low level details of implementation details profoundly affect performance. This is because DGAs exhibit little locality, rarely require any significant computation per memory access, and result in a high-rate of communication of small messages. Thus, unlike regular algorithms that are built on top of well-understood regular communication and memory access, graph algorithms interact with the entire software and hardware stack in a complex way due to their data-driven, fine-grained, irregular nature of tasks. Each piece of the stack, designed independently, from the algorithm level through the transport layer to the hardware layer and the topology of the physical network, interacts within the system. This makes designing DGAs an experimental endeavor, and this state of affairs will be only exacerbated as we move towards exascale computing. It should be noted that the complexity of the interactions between high-level algorithm and low-level runtime is not unknown to the practitioners. However, this knowledge is implicit, fragmented, and often sidelined in presentation of new techniques. Notably, Checconi and Petrini [11], who achieve

Table 1: Runtime-level aspects of DGAs. These aspects need to be disclosed when reporting results; for those that are quantifiable, numerical values should be reported for each experiment.

Transport		Sec. 2.1.1
Communication Paradigm	Point-to-Point ^[11,31,18] , Collectives ^[8,7,11,23]	
Request Tracking	One-Sided ^[18] , Active messages ^[31,9]	
Progression ^[11,31] :	Remotely synchronous, Asynchronous ^[31,19]	
• Asynchronous	Locally synchronous ^[19,11]	
• Synchronous	System threads ^[31] , User threads	
	Explicit progress ^[31] , Runtime scheduler ^[31]	
	Lightweight task ^[31]	
Bit Transport	System Processing Interface (SPI) ^[12,9,11]	
	Message Passing Interface (MPI) ^[7,11,8]	
	Active Pebbles ^[28] , ARMI ^[16,17] , Photon ^[19]	
Protocol	Eager, rendezvous ^[31] , completion ^[9,11]	
	Message reduction/caching ^[25,31,23]	
Optimization	Message coalescing ^[11,25,24]	
	Message compression ^[11]	
Message routing	2D, 3D ^[24,25] , ring ^[30] , hypercube, rook ^[14]	
Thread safety	Multi-threaded ^[31] , serialized, funneled	
Network Topology		Sec. 2.1.2
Physical Topology	3D ^[24,10] and 5D ^[10] torus, Dragonfly ^[31,8]	
Logical Topology	Skewness ^[8] , synthetic network ^[24,25]	
Job topology	Job allocation, rank mapping ^[8]	
Local Scheduling		Sec. 2.1.3
Threading	Pthreads ^[31] , OpenMP ^[7,8] , HPX-5 ^[18]	
Task management	Work stealing, FIFO/LIFO tasks ^[31]	
	Priority scheduling ^[18] , System threads ^[10]	
Termination	Quiescence detection ^[24] , SKR ^[31]	
Heuristics	End-epoch frequency ^[31]	
	Eager progress limit ^[31]	
Hardware effects	L2 atomics ^[9] , NUMA effects ^[8]	
Runtime Feedback		
Optimal K-level ^[17] , optimal Δ , sync to async switching ^[29]		

the top results in the Graph 500 benchmark in part due to direct access to the SPI (System Programming Interface) low-level primitives, provide an outstanding analysis of their evolving implementation, including a three-years timeline of changing conclusions and understanding. Unfortunately, this manner of reporting is not typical for the field.

We contend that the advancements in the field are difficult to generalize and reconcile because the information reported is commonly incomplete. The low-level details of implementations are often vague or missing. Yet, these can have important impact.

In this paper we propose a template of runtime features, presented in Table 1, to aid authors in reporting their work. We arrive to this template by combining our own experience in co-designing runtime system and DGAs with findings from review of existing literature. For completeness, we also list algorithm-level aspects, Table 2. We invite others to extend and revise our suggestions, and to make this a community effort. Widespread adoption of our recommendation will enable transferability of lessons learned across the field, metastudies of the interaction of runtime and algorithmic concerns to potentially derive abstract models, and,

Table 2: Algorithm-Level aspects of DGAs. These aspects are usually disclosed in literature; we present them here for completeness.

Approach	
Ordered	Level-Synchronous ^[30,11]
	Bellman-Ford ^[23]
	Combinatorial BLAS ^[7,4,8]
Ordered/Unordered	Hybrid ^[9] , HSync ^[29]
Unordered	Distributed control ^[31]
	KLA ^[17] , Δ -stepping ^[13,23]
Algorithmic Considerations	
Data Distribution	1D ^[13,23,9,31,11]
	2D ^[30,7,8] , Edge list ^[24,25]
Optimizations	Ghosts ^[15,20,24,25] , Pruning ^[9]
	Direction optimization ^[9,4,11,8]
	Priority messages ^[31]
	Tree-based broadcast, reduction, and filtering ^[25]
Load Balancing	Per-thread work splitting ^[9] ,
	Random shuffling of vertex identifiers ^[7]
	Delegates ^[25] , Proxies ^[9]
Graph Representation	
CSR ^[7,4,23,31,8] , compressed coarse-index adjacency list ^[11] , skip list ^[11] , doubly-compressed sparse column (DCSC) ^[7,8]	
Data Structures (algorithm progress)	
Distributed async visitor queue ^[24,25] , thread-local priority queue ^[31] , dynamic-array buckets ^[23]	

with deepened, systematic understanding, improvement of performance.

Contributions:

An analysis of the field (Sec. 2), identifying, classifying, and discussing two levels of distributed graph algorithms: (i) Runtime-level aspects (Sec. 2.1) that authors do not explicitly consider a part of the algorithm but that play a crucial role in the overall performance; and (ii) Algorithm-level aspects (Sec. 2.2) that authors identify as the main algorithmic contributions of their research;

A template of runtime features (Sec. 2, Table 1) for authors to consider when describing their research;

Filled-out “report cards” (Table 3) for Distributed Control [31] and BlueGene/Q BFS [11], which constitute examples of how to use our proposed template; and

Demonstration of sensitivity of algorithm performance to runtime parameters (Sec. 4) where we show how seemingly minor changes may have significant effect.

2. ANALYSIS OF DGAs

In this section, we analyze and describe a sample of the existing research on distributed BFS and SSSP problems. Our motivation is two-fold. First, we want to get an overall feel for how complete (or incomplete) is the information about runtime part of the DGA stack that is presented in literature. Second, we aim to identify potential aspects of DGAs beyond those stemming from our own work. DGAs consist of complementary *runtime-level* (Sec. 2.1) and *algorithm-level* (Sec. 2.2) aspects. Algorithm-level aspects vary for different graph algorithms; however, the runtime-level aspects are

Figure 1: Overview of the runtime stack components

algorithm agnostic. For this reason, our analysis of runtime concerns is applicable to any DGAs. The purpose of our effort is to construct a blueprint for a more holistic treatment of DGAs. Tables 1 and 2 summarize the runtime-level and algorithm-level aspects of DGAs discussed in the remainder of the section. Moreover, Table 1 serves as a template for reporting runtime aspects of DGAs, and we use it as such to describe three DGA runtimes in Table 3.

2.1 Runtime-Level Aspects of DGAs

DGAs *runtime* (Fig. 1) has two major distinguishable components working in rapport: a *transport* (Sec. 2.1.1) and a *scheduler* (Sec. 2.1.3). The basic component of a transport is a *bit transport* that is a system-provided implementation of an interface for actually sending bits over the wire. Bit transport can support different *protocols* for exchanging messages such as eager or rendezvous protocols. The bit transport API may support levels of *thread safety*, impacting both the use and the internal working of the transport (hence the shared part in the figure). *Request tracking* is the mechanism for making and keeping track of communication requests. Building on bit transport, the runtime must use the request tracking mechanism in some way, according to thread safety, to initiate and complete requests. We call this process *progression*. The choice of *Communication paradigm* has reverberating impact both on the rest of the transport layer and the DGA itself. For example, collectives require more memory for communication results than point-to-point communication and the progression mechanism has to be designed with fewer but more heavyweight requests in mind. Transport may provide *logical topologies* that provide additional *routing* on top of *physical* and *job layout* topologies to improve performance (e.g., message reductions in logical topology) or to reduce memory requirements (e.g., fewer coalescing buffers). Finally, the transport may employ various optimizations such as message coalescing.

The *scheduler* part of the runtime schedules worker threads or tasks. One of the responsibilities of the local scheduler is to ensure transport progression. Local scheduler can also be responsible for checking termination of an algorithm.

In the following discussion, we categorize important runtime-level parameters based on the review of existing literature for DGAs. Table 1 summarizes the set of low-level runtime parameters.

2.1.1 Transport

The transport layer is the part of the stack responsible for sending and receiving bits. Important properties of transport include how message buffers are handled, which entity

manages them, and how frequently they need to be managed. The runtime needs to take several decisions regarding these.

The choice of *communication paradigm* can have a notable impact on performance of DGAs. Each paradigm imposes different tradeoffs in terms of memory constraints, synchronization overhead, and network latency. The *collectives* paradigm is used when large, low-overhead stages of all-to-all communication are needed, *point-to-point* paradigm allows for finer overlap between computation and communication at the expense of code complexity, and *active messages* are a refinement of point-to-point communication that adds an implicit execution of *handlers* on remote objects. Finally, *one-sided* paradigm provides remote memory operations (GET, PUT, etc.) which are very efficient but require remote memory management protocol.

For example, collectives are the base of BLAS approaches and level-synchronous approaches. However, Checconi and Petrini [11] show how using lightweight point-to-point communication may lead to improvements in traditionally synchronous approaches. They compare their point-to-point implementation using low-level BlueGene/Q’s System Processing Interface (SPI) to an MPI implementation using collectives. They note the large memory footprint required for collective buffers, which forces them to decrease the scale of the problem per node. Furthermore, collectives do not allow for easy interleaving of computation with communication. Active messages are based on point-to-point communication, and they display similar communication performance characteristics.

Request Tracking (RT) refers to how communication requests are made: a request is scheduled and completed separately (*asynchronous*), a request is made and completed at the same time (*locally synchronous*), or a request is made and the requester waits until it has been completely processed (*remotely synchronous*). As an example, *remotely synchronous* request tracking in MPI (e.g., by `MPI_Ssend` etc.) guarantees a small number of messages on the network, but it hinders parallelization. On the other hand, using *locally synchronous* RT (e.g., `MPI_Send`) makes it easy to reuse buffers and may allow parallelism if eager protocol is used. Finally, *asynchronous* RT uses interfaces such as `MPI_Isend/MPI_Irecv` to start requests along with interfaces such as `MPI_Testsome` to check for their completion, maximizing overlap (if the MPI implementation supports it) between computation and communication at the cost of more complex progression and request management.

Completing a round trip through transport requires book-keeping, performing bit moving, and delivering the results of completed requests—we call that *progression*. Progression

influences the timeliness and efficiency of transport delivery, and a wrong progression model can render an algorithm infeasible (Sec. 4.3). *Asynchronous* progression is performed periodically and is scheduled through dedicated resources such as *system* or *user* threads. For example, Cray MPI provides an option for starting progression pthreads that perform internal MPI progress in parallel with the algorithm threads. Progression through *user threads* is scheduled by the runtime explicitly, and, for example, calls MPI repeatedly to generate progress. In contrast, *synchronous* progression is initiated periodically from the runtime. In *explicit* progress, the algorithm can choose, bypassing the runtime scheduler (if any), when to call progress, enabling optimizations at the cost of added code complexity. For example, in [31], we employed explicit polling in our Distributed Control (DC) algorithm for SSSP, but observed a decrease in performance. In a task-based system, network progress can be scheduled as a *lightweight* task. For example, AM++ [28] implements network polling, buffer flushing, checking for termination, and executing pending handlers for received messages as tasks, on equal footing with algorithm tasks that run message handlers. HPX-5 [18] executed network progress in a similar fashion, but the more recent versions switched to explicitly initiating progress in the main *scheduler loop*, giving the runtime more control over when progression is executed. Most authors do not discuss progression and request tracking explicitly, but the choices made for these parameters may have a profound effect on performance (cf., Sec. 4.3).

Bit transport is the lowest-level network interface used by upper levels to deliver bits from one location to another. In efficient BlueGene/Q implementations [12,9,11,26], the *System Processing Interface (SPI)* communication layer serves as a bit transport (as described above). The majority of implementations in the literature use *Message passing Interface (MPI)* for their bit transport. SPI is a direct interface to hardware queues, while MPI is a complex framework with extra functionality and semantics. Direct interfaces such as SPI may yield more efficient communication, but are less or not at all portable, and may require more implementation effort. The third type of bit transport is based on remote method invocation (RMI) technique and is used in approaches based on STAPL [16,17], a generic parallel library for graph and other data structures and algorithms. STAPL uses the ARMI (Adaptive Remote Method Invocation) active-message communication library, based on RMI. ARMI supports automatic message coalescing but does not provide routing or message reductions natively. HPX-5 runtime has support for two types of bit transport: one is based on MPI, another one is based on Photon [19] RDMA middleware library. Photon is based on RDMA put and get *with completion*, where requests are completed asynchronously, and their completions are written to *ledgers* that can be read by higher level runtimes. HPX-5, AM++ and ARMI can use different bit transport backends, making the interface boundary between the bit transport and the runtime very clear.

Bit transport may employ different **protocols**. For example, MPI point-to-point communication may support *eager* protocols for small messages and *rendezvous* protocols for larger transfers, sending messages without or with, respectively, round-trip communication [3]. The choice of protocols may have a detrimental impact on algorithms (e.g., Sec. 4.2), and it may be difficult to control explicitly. For example, most MPI implementations provide extensive configuration

options, but these options are not standardized and often can only be fully utilized by experts.

A number of runtime-level **optimization** techniques have been proposed in the literature to reduce communication overhead and maximize throughput. AM++ provides *message reduction* (caching). Pearce et al. [25] used tree-based broadcast, reduction and filtering for communication involving high degree vertices. Panitanarak and Madduri [23] used local lookup arrays to track the tentative distance of every vertex, thus avoiding duplicate request being sent.

Increasing *message coalescing* (cf., Sec. 4.2) buffer size increases the rate at which small messages can be sent over a network at the cost of latency. Checconi and Petrini [11] use coalescing to pack together all the edges that would be sent to each destination separately and queue them in an intermediate buffer. Pearce et al. [25, 24] combined coalescing with routing to reduce dense communication.

Message routing constructs a logical topology to add intermediate targets for messages. Pearce et al. [24] implemented routing through a synthetic network to mimic the BG/P 3D torus interconnect topology. In a follow-up paper [25], the authors additionally embedded the delegate tree as a means for further communication reduction. AM++ [27] supports software routing and provides two predefined strategies: *rook routing* and *Hypercube routing*. Rook routing reduces the number of communicating buffers to $O(\sqrt{p})$ [14]. Yoo et al. [30] used ring communication in their optimized collective implementation and adjusted the diameter of the ring to achieve better performance.

Thread Safety A message passing framework can support different levels of thread safety. For MPI, there are 4 levels in total [2]: *single*, *funneled*, *serialized* and *multiple*. We used *multiple* as the threading level together with an asynchronous progress thread in our AM++ DC [31] implementation.

2.1.2 Network Topology

Computing resources are organized in several specific *physical topologies*: 3D torus, dragonfly, 5D torus, and so on. The physical topology impacts the efficiency of communication in a graph computation. For example, Cray MPI provides an all-to-all implementation that is optimized for Aries and Gemini systems. In another example, Checconi and Petrini [10] map parts of graph adjacency matrices onto the Blue Gene/Q 5D torus topology in such a way that neighboring parts of the matrix are also neighbors in the physical topology.

The *logical topology* is the layout of the data in physical topology. Buluç et al. [8], for example, found that *processor grid skewness*, i.e., the distribution and the shape of the blocks of an adjacency matrix, had significant impact on their results: the “tall skinny” grids (more blocks across the Y dimension of the matrix) performed faster, and “short fat” grids (more blocks across the X dimension of the matrix) performed worse than square grids.

Job scheduler for computing resources allocates nodes based on scheduling policy resulting in a *job topology*. Bhatele et al. [6] showed that job topology may have an impact on performance due to the distances among allocated nodes or due to contention on shared network.

2.1.3 Local Scheduling

Depending on the node-level threading mechanism, thread scheduling policies, and synchronization primitives, tasks

associated with a DGA can execute in different order with varying frequencies. For example, in an attempt to quickly spread good work, a message can be sent with priority and put the message handler in front of the task queue [31]. Supporting data structures, for example bitmaps in sync mode and global queue in async mode in [29], can be another way to implement local scheduling. Below, we discuss several thread-granularity and scheduling-related factors.

Threads (worker threads) can be used for intra-node threading. Buluç and Madduri [7] and Buluç et al. [8] used MPI for inter node processing and GNU OpenMP for intra-node threading. Zalewski et al. [31] used a combination of MPI and pthreads. HPX-5 uses suspendable lightweight threads with their own stacks and with cheap thread transfer.

Lightweight threads or tasks, implemented on top of kernel threads, can be scheduled differently. *Task management* mechanisms achieve load balancing by mechanisms such as *work stealing* and *FIFO/LIFO* schedulers.

2.2 Algorithm-Level Aspects of DGAs

Table 2 summarizes the set of parameters we identified as the algorithm-level aspects of DGAs, and divide them into four categories. The *approach* category is about the main algorithmic choices, the *algorithmic considerations* category covers the main aspects of the approach, and the categories of *graph representation* and *data structures* cover the data structures that are used.

3. OUR TEMPLATE IN PRACTICE

In this section we show how our proposed template can be applied in practice by comparing 3 different runtimes: AM++, HPX-5, and the IBM BlueGene/Q implementation. The characteristics of the runtimes that we consider are independent of any particular application. Our recommendation is to consider DGAs holistically, encompassing runtime and algorithmic concerns. Specifically, we apply the template to Distributed Control (*DC*) algorithm[31] for solving the SSSP problem implemented in the AM++ and HPX-5 runtimes and to BlueGene/Q BFS [11]. The BlueGene/Q BFS is a rare case where the authors disclosed enough information so that we can fill out a “report card” based on our template (Table 3). We choose *DC* because it is particularly well suited as a subject of inquiry into interplay of runtime and algorithmic concerns, as will be evident from next section devoted to experimental results.

3.1 The Runtime Report Card

The “report card” in Table 3 enumerates the runtime features of each of the two DGAs according to the template in Table 1. We summarize the runtimes in a table form for clarity, but we do not advocate that scientific publications use that exact format. The important task is to ensure that all relevant aspects of the runtime are adequately covered. Furthermore, every runtime feature must list relevant associated quantities that are necessary for complete interpretability of experimental results. For example, in presence of coalescing, performance results cannot be interpreted if the size of coalescing buffers is not given.

3.2 Algorithmic Concerns

The goal of *DC* is to remove the overhead of synchronization and global data structures by using only thread-local

Algorithm 1: Distributed Control Algorithm

Work loop

Input: Graph $G(V, E)$, source s , distances D

```

1: Init( $D$ )                                     {set distances to  $\infty$ }
2: if thread_id = 0 and owner( $s$ ) then
3:   relax( $s, 0$ )                                {relax source}
4: end if
5: while not terminated do
6:    $q$  is a (distributed) approximate priority queue
7:   if not  $q$ .empty() then
8:      $(v, d) \leftarrow q$ .pop()                    {next task to process}
9:      $d_v \leftarrow D(v)$                           {current distance for  $v$ }
10:    if  $d_v < d$  then
11:      continue
12:    else
13:      relax( $v, d$ )
14:    end if
15:  end if
16: end while

```

Relax handler

Input: Task (v, d)

1: q_t .push(v, d) {insert task into priority queue}

Relax

Input: Task (v, d) , distances D

```

1: if  $d < D(v)$  then
2:    $D(v) \leftarrow d$ 
3:    $\forall v_n \in \text{neighbors}(G, v)$  : send( $v_n, d_v + \text{weight}(v, v_n)$ )
4: end if

```

priority queues to select best work, obtaining an approximation of the global ordering. Pseudo code for a DC algorithm for SSSP is given in Algorithm 1. The algorithm consists of 3 parts: the work loop that processes tasks from the local priority queue, the message handler that receives tasks from other workers, and the relax function that updates distances and generates new work. The work loop in *DC* is preceded by initialization of the distance map and by relaxing the source vertex (Lines 2–4). In the loop, the work on the graph is performed by removing a task from the thread-local priority queue in every iteration and then relaxing the vertex targeted by the task (Lines 7–14). Vertex relaxation checks whether the distance is better than the distance already in the distance map, and it sends a relax message (task) to all the neighbors with the new distance. Relax handler receives the messages sent from the relax function, and its only purpose is to insert the incoming tasks into the thread-local priority queue. When a handler finishes executing, it is counted as finished in *termination detection*. Note that there is no synchronization barrier in the algorithm. The work loop in the algorithm description is an abstraction of a more general work scheduling mechanism. In our AM++ implementation of *DC*, work loop is very similar to the loop in the figure, but it also includes some heuristics to improve performance and additional code for explicit management of termination. In our HPX-5 implementation, the work loop is provided as a more general *priority scheduling* mechanism in the runtime, where the runtime promises that a best effort priority will be applied when scheduling work marked as priority work.

4. RUNTIME PARAMETERS OF DGA PERFORMANCE

DC is particularly sensitive to the runtime characteristics

Table 3: Properties of the AM++ [31], HPX-5 [18], and BlueGene runtimes [11] summarized in a template based on Table 1.

HPX [18]	AM++ [31]	BlueGene/Q BFS [11]
Transport		
<i>Paradigm</i>		
Parcels	AM++ [27] active messages.	One-sided (low-level active messages).
<i>Request Tracking</i>		
With Photon, sends are asynchronous, and receives are automatic with completion events. With MPI, asynchronous MPI interfaces are used, with limited number of active send and receive requests. Send requests are queued until an MPI send slot becomes available. Required: Send/receive limits (MPI), ledger size (Photon)	Sends and receives are scheduled with asynchronous MPI interfaces. The number of receive requests is kept constant, and send requests are created on demand with a flow-control mechanism to cap the number of outstanding requests. Required: number of receive buffers, flow control limit	Sends are locally synchronous (through SPI) with one buffer per destination (maximum one outstanding send per destination). Reception is asynchronous through polling of SPI counters.
<i>Progression</i>		
HPX-5 invokes progress explicitly in the scheduler loop. Progress is invoked by a worker thread if there is no local work left, and with the priority scheduler for DC, progression is invoked periodically based on flow control feedback. Progression for MPI is serialized between workers. Sends are processed first from a send queue, then receives are processed and reused as they complete. For Photon, progression can be run by multiple threads (Photon is thread safe). Completion events returned from Photon are processed. Required: Flow control parameters	AM++ uses shared coalescing buffers. Requests can be explicitly polled (with <code>MPI_Testsome</code> on an array of requests). AM++ also has special purpose user-level tasks that perform progression, executed from AM++ task queue when sends are performed, or when end-of-epoch tests are performed (during these tests AM++ tries to finish an epoch by processing remaining work). On Cray machines, distributed control performs the best when an asynchronous MPI progress thread is used. Required: number of poll tasks	A lightweight asynchronous communication layer on top of System Processing Interface (SPI) with separate FIFOs for injections and receptions (multiple queues per node, providing network-level parallelism). Messages are coalesced to per destination buffers where each buffer is exclusively owned and operated on by a single thread, eliminating locking and contention. Buffers are placed into injection queues when ready, and a thread will wait for completion when another message needs to be sent to a given destination. Required: number of FIFO queues
<i>Bit Transport</i>		
MPI (high-level), Photon (low-level).	MPI, a high-level portable interface.	SPI, a low-level BG/Q-specific interface.
<i>Protocol</i>		
Depends on coalescing sizes. MPI: depends on implementation. Photon: direct and rendezvous.	Depends on MPI implementation and coalescing buffer sizes (Sec. 4.2). Required: list of used protocols	FIFO injection queue send and receive in SPI.
<i>Optimization</i>		
Message coalescing (per destination or cumulative) and compression. Required: Coalescing buffer sizes	Message coalescing. AM++ is capable of reduction and caching, but these optimizations are not beneficial for DC. Required: Coalescing buffer sizes	Message coalescing and compression. Compression is beneficial only for large BFS wave fronts. Required: Coalescing buffer sizes
<i>Routing</i>		
No routing.	Rook, hypercube routing.	No routing.
<i>Thread Safety</i>		
MPI: thread serialized. Photon: multiple threads.	AM++ works with MPI thread serialized and thread multiple safety levels.	Multiple threads can progress SPI FIFO queues.
Network Topology		
<i>Physical Topology</i>		
Star (central switch).	3D torus Gemini and Dragonfly Aries.	BlueGene/Q 5D torus.
<i>Logical Topology</i>		
Global Address Space (PGAS or AGAS).	None.	None.
<i>Job Topology</i>		
Inconsequential because of star topology.	Unknown (execution time averaged from the same batch execution).	Unknown. Largest runs show significant variability.
Local Scheduling		
<i>Threading</i>		
Pthreads, lightweight user threads.	Pthreads.	Heavyweight worker threads.
<i>Task Management</i>		
LIFO and priority queues of <i>parcels</i> , which represent undone work or suspended threads.	FIFO queue with every coalesced buffer represented as a task.	None, no lightweight tasks.
<i>Termination</i>		
SKR termination detection (activity counts periodically reduced).	SKR termination detection (activity counts periodically reduced).	Unknown. “Termination check” is mentioned at least once.

Figure 2: Execution time of Δ -stepping and DC on HPX-5 with two different bit transports (ISIR and PWC) and two different interconnects: InfiniPath QLE7340 (qib) and Mellanox ConnectX-3 EN (mellanox).

such as timing of task execution, communication latency, and buffering, and we use it to illustrate the impact that the runtime may have on the performance of DGAs. In the remainder of this section, we discuss the characteristics of the runtime used in our experiments and the impacts we observed. Our experiments were run at varying times on different machines. The experiments we discuss here were run on Big Red 2 (BR2) at Indiana University [1], on Edison at NERSC, and on our own cluster Cutter consisting of 16 nodes with 16 Haswell cores each and equipped with InfiniPath and Mellanox interconnects. The most important differences between the two Cray machines are the topologies – 3-D torus on Big Red 2 vs. Dragonfly on Edison – and the MPI implementations – Cray’s Message Passing Toolkit (MPT) 6.2.2 on Big Red 2 vs. MPT 7.1.1 on Edison. On Cutter, we use OpenMPI 1.8.8 for MPI runs and Photon [19] otherwise. On Cutter, we also vary the interconnect between InfiniPath and Mellanox. All experiments were run on Graph500 graphs. All execution times are reported in by taking the average of executing multiple problem instances (over the same batch job execution).

4.1 Effect of Bit Transport and Interconnect

Figure 3: Impact of send limit for ISIR network on the InfiniPath interconnect with 16 nodes and scale 23 graph for Δ -stepping algorithm.

Figure 2 demonstrates how different bit transports (MPI ISend-IREcv based ISIR transport and Photon Put-With-Completion (PWC) [19] transport) can affect the performance of an algorithm. We have ran Δ -stepping and DC with HPX-5 on cutter for these experiments. In terms of performance and scaling, DC with PWC transport and Mellanox interconnect performs the best. In fact DC shows better scaling for 8 and 16 nodes with Mellanox than with InfiniPath for both PWC

Figure 4: Effect of coalescing on a scale 31 graph (BR2).

Figure 5: Effect of coalescing size on DC SSSP algorithm on a scale 31 graph (Edison).

and ISIR transports. Our one node performance is taken with networking turned on; DC performs much worse than Δ -stepping, but it quickly improves with scale. Δ -Stepping does not show good scaling behaviour altogether.

While experimenting with ISIR transport, we have tried different limits for the number of MPI Isend calls that HPX-5 spawns concurrently. Figure 3 shows how varying the send limit changes the performance of Δ -Stepping algorithm for one of the scales. Table 4 shows the optimal send-limits for Δ -stepping algorithm with ISIR transport. This experiment illustrates how a runtime-level bit transport parameter can make an impact on the performance of an algorithm.

4.2 Effect of Coalescing Size on Transport Protocol

To increase bandwidth utilization, AM++ performs *message coalescing*, combining multiple messages sent to the same destination into a single, larger message. Messages are appended to per-destination buffers. To handle partially filled buffers, a periodic check is performed to check for activity. In the case of DC SSSP, a single message consists of a tuple of a destination vertex and distance, 12 bytes in total. With such small messages, coalescing has great impact on the performance, but finding the optimal size is difficult.

Table 4: Optimal send limits for weak scaling experiment in Fig. 2 for InfiniPath (qib) and Mellanox (mel) interconnects.

n	qib	mel
1	4	8
2	8	32
4	8	128
8	16	256
16	32	64

We investigated the impact of coalescing in Graph500 scale 31 graphs when running DC SSSP with max edge weight of 100 (Figs. 4a and 5). Figure 4a shows the large impact of a small change in the coalescing size, which is measured by the

number of SSSP messages per coalescing buffer. Changing the coalescing size by less than 2% causes over 50% increase in the run time. This unexpected effect is caused by the specifics of Cray MPI protocols. At the smaller coalescing size, full message buffers fit into rendezvous R0 protocol that sends messages of up to 512K using one RDMA GET, while the larger buffers hit R1 protocol that sends chunks of 512K using RDMA PUT operations. At the size of 44,000, the bulk of the message fits into the first 512K buffer, and the small remainder requires another RDMA PUT, causing overheads. The sizes 43,000 and 86,000 fill out 1 and 2 buffers, respectively, achieving similar performance. The larger size, 86,000, results in better scaling properties.

Figure 6: Effect of asynchronous progress on BR2.

show that adjusting other parameters is less and less important as the coalescing buffer size increases.

Figure 4b shows the effects of coalescing on a *DC* BFS, which is SSSP with maximum weight of 1. Surprisingly, increasing the coalescing size impacts performance negatively. We suspect that with smaller weights the possibility of reward from optimistic parallelism in *DC* decreases, and the added latency of coalescing has a much larger effect than with larger weights. All three cases shown in Figs. 4a, 4b and 5 show that adjusting the coalescing size is important, and the optimal value is not static. Rather, it depends on algorithmic concerns such as reward from optimistic parallelism.

4.3 Transport Progress

At first, when we experimented with *DC* on Big Red 2 with AM++, we found out that it was performing worse than Δ -stepping algorithm [21]. This raised a concern that the *DC* approach may not be practical. We suspected the possibility of message latencies being a culprit; so, upon researching MPT, we decided to experiment with *asynchronous progress*, which uses separate threads to perform progress in certain situations. Despite Cray’s warning at the time that thread-multiple progress required for asynchronous progress “is not considered a high-performance implementation,” we observed significant gains for *DC*, shown in Fig. 6. We ran the experiment on Graph500 scale 31 optimal strong scaling results. Without asynchronous progress, performance decreased with the increased number of nodes (with an unexplained anomaly at 112 nodes). (Note that all our experiments are averaged; thus, large anomalies are indicative of unexpected circumstances.) With asynchronous progress thread, the performance of *DC* has improved more than tenfold with growing node counts, entirely changing the viability of the approach. This dramatic effect illustrates how deeply an algorithm interacts with the runtime and how a gap in

We ran a more extensive suite of benchmarks on Edison. Figure 5 shows the coalescing buffer size experiments on Edison. The results are similar, with a periodic increases in the minimum run time as protocol buffers mismatch the coalescing buffers. The maximum run times signify the worst run time, as other parameters related to bit transport than coalescing are adjusted. The results

parameter space may lead to incorrect conclusions about DGA approaches. Interestingly, we did not observe a similar effect on Edison, where two different asynchronous progress and the standard progress modes perform similarly. We are unable to explain why it is so. This unpredictability and the difficulty of analyzing the performance shows how important it is to document the specific runtime in which DGAs are executed.

4.4 Distributed Control Progress Heuristics

In addition to transport layer progress, AM++ performs its own internal progress when AM++ interfaces are called. Since AM++ *DC* is built around a loop that empties the local priority data structure (Lines 5–16 in Algorithm 1), it must occasionally, with some frequency, call into the appropriate AM++ interfaces that perform progress. This frequency is controlled by two parameters: the end-epoch test frequency (EE) and the eager progress limit (EL). EE controls how many iterations of the *DC* loop run before AM++ progress is invoked. The eager limit is a threshold of outstanding *DC* tasks below which AM++ progress is performed during every iteration of the *DC* loop.

Figure 7 shows the effects of progress parameters, using performance data averaged over multiple runs while varying orthogonal parameters and choosing the best performing variant, which isolates the effects EE and EL parameters. Edison shows a significant sensitivity to the EE parameter. Smaller values are better, with 22 being the best of the ones tested. This suggests that latency may be a limiting factor on Edison. On Big Red 2, the results of varying the EE parameter are less pronounced, but the average of multiple experiments that we show here still suggests some sensitivity with the optimal value similar to that on Edison. Altogether, the results show that the performance of *DC* depends on the progress model.

4.5 Buffering and Work Efficiency

The prerogative of coalescing in AM++ is to decrease the overhead by sending as many full coalescing buffers as possible. Partially filled buffers are only sent when no more messages are being inserted. Figure 8 shows *DC* results on Edison for coalescing buffer size of 100,000. We found that the best predictor of performance is the amount of partial buffers (fewer is better) followed by full buffers (more is better). Partial buffers indicate periods of a lack of work, and this, in turn, indicates that the local priority queues are getting depleted more often, decreasing overall performance. AM++ was originally optimized for algorithms like BFS and Δ -stepping, which benefit from eager optimization of communication overhead and are not sensitive to work

Figure 7: Effect of AM++ progress parameters.

Figure 8: Partial and full buffer statistics for 40 fastest executions with coalescing size fixed at 100000 (on Edison, across different batch jobs).

Figure 9: Work statistics for DC-SSSP and Δ -stepping algorithm in AM++. (a) Useful, invalidated, and rejected work. (b) Useless work.

imbalance. Our example shows that optimization of runtime for a seemingly worthy goal can negatively impact algorithms that have other needs not anticipated by runtime developers.

4.6 Work vs. Overhead

Performance of an algorithm depends on the amount of *work* it performs and on the amount of *overhead* that this work incurs in a given runtime. Figure 9 shows the work statistics comprising of useful work (vertex distance was updated), useless work (distance was not updated), rejected work (distance updated but neighbors are not visited) and invalidated work (useful work overwritten by a better distance) for DC and our implementation of Δ -stepping in AM++ with scale 31 graph. Although DC performs better than Δ -stepping, DC always executes more work than Δ -stepping in the most efficient configurations of both of the algorithms. Despite consistently performing 10%-25% more work, DC performs better in all instances of tests at scale (3-6 times speedup). This shows that synchronization and uneven distribution of work have an important effect on the performance of DGAs. Although one can attempt to mitigate the work imbalance with algorithmic techniques, the cost of synchronization is hard to control and eliminate. In this regard, an underlying runtime can have a significant impact. The more an algorithm depends on keeping global information about the runtime (e.g., for load balancing), the higher the costs of synchronization necessary to maintain that information. In Figure 9 we count a task as rejected when the vertex distance it delivers is higher than what is already recorded and, consequently, the task is not inserted into the priority queue of DC or a bucket of Δ -stepping. Invalidated tasks

are similar to rejected tasks, but their distance expires while they wait in priority queue.

5. CONCLUSIONS

We demonstrate that the algorithm-level parts of DGAs that are reported as major contributions do not constitute a complete description of a DGA. A DGA consists of two equally important layers: the algorithm-level aspects and the runtime-level aspects, which respectively represent the top and the bottom of the software/hardware stack. Based on analysis of a representative sample of DGAs, we further subdivide the layers into categories. We propose a template for reporting research design and results and we demonstrate how to use it. Altogether, the goal is to make research results in DGAs more accessible, general, and congruent.

Our Tables 1 and 2 serve as a map for *reporting the design features* and the related quantities relevant for interpretability of experiments. Some runtime aspects may remain “buried in the stack”, their impact unknown (e.g., the effects of job placement as in Sec. 2.1.2 are not usually investigated), and some may not be relevant in a given situation. A complete “report card” helps one understand which parts of the parameter space are covered and which are not. Our reporting template helps both consumers and authors of research, the former to understand and the latter to present contributions.

Our analysis and guidelines are the first step in unifying the field. We posit that the DGA research community should collectively develop a set of standards expected from top notch research, acknowledging that DGAs exhibit particularly strong interaction with the software/hardware stack due to their irregularity. Thus we appeal to the wider community to help develop standards for more explicit incorporation of runtime interactions in future research results and by collaboration on a continuously updated consensus on what constitutes the runtime of a DGA.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No(s). 1111888 and 1319520 and by the Department of Energy, National Nuclear Security Administration, under award number DE-NA0002377 and by in part by Lilly Endowment, Inc.

7. REFERENCES

- [1] Big Red II at Indiana University. <http://rt.uits.iu.edu/ci/systems/BR11.php#info>. Accessed: 2016-01-15.
- [2] MPLInit_thread. http://www.mpich.org/static/docs/v3.1/www3/MPLInit_thread.html, . Accessed: 2016-01-15.
- [3] MPI Performance Topics. https://computing.llnl.gov/tutorials/mpi_performance, . Accessed: 2016-01-15.
- [4] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. *Scientific Programming*, 21(3-4):137–148, 2013.
- [5] S. Beamer, A. Buluç, K. Asanović, and D. Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1618–1627, May 2013.

- [6] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 41:1–41:12. ACM, 2013.
- [7] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.
- [8] A. Buluç, S. Beamer, K. Madduri, K. Asanović, and D. Patterson. Distributed-Memory Breadth-First Search on Massive Graphs. In *Parallel Graph Algorithms, D. Bader (editor), CRC Press*. 2015. To appear.
- [9] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014.
- [10] F. Checconi and F. Petrini. Massive data analytics: The Graph 500 on IBM Blue Gene/Q. *IBM J. of Research and Devel.*, 57(1/2):10:1–10:11, Jan. 2013.
- [11] F. Checconi and F. Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proc. 28th International Parallel and Distributed Processing Symposium*, pages 425–434. IEEE, 2014.
- [12] 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 *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 13:1–13:12. IEEE, 2012.
- [13] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Nov. 2006.
- [14] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing Graph Algorithms Using Generalized Active Messages. In *Proc. 27th International Conference on Supercomputing*, pages 283–292. ACM, 2013.
- [15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.
- [16] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In H. Kasahara and K. Kimura, editors, *Languages and Compilers for Parallel Computing*, number 7760 in LNCS, pages 46–60. Springer Berlin Heidelberg, 2013.
- [17] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proc. 23rd Internat. Conf. on Parallel Architectures and Compilation*, pages 27–38. ACM, 2014.
- [18] HPX-5 at the Center for Research in Extreme Scale Technologies (CREST). <http://hpx.crest.iu.edu/>. Accessed: 2016-01-11.
- [19] E. Kissel and M. Swamy. Photon: Remote Memory Access Middleware for High-Performance Runtime Systems. In *First Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, IPDRM, 2016.
- [20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in The Cloud. *Proc. VLDB Endowment*, 5(8):716–727, 2012.
- [21] U. Meyer and P. Sanders. Δ -stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1): 114–152, 2003.
- [22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500 benchmark. *Cray User’s Group (CUG)*, 2010.
- [23] T. Panitanarak and K. Madduri. Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory System. In *Proc. Sixth SIAM Workshop on Combinatorial Scientific Computing*, page 60, 2014.
- [24] R. Pearce, M. Gokhale, and N. M. Amato. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *Proc. 27th IEEE International Symposium on Parallel and Distributed Processing*, Los Alamitos, CA, USA, 2013. IEEE.
- [25] R. Pearce, M. Gokhale, and N. M. Amato. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 549–559. IEEE, 2014.
- [26] X. Que, F. Checconi, F. Petrini, X. Liu, and D. Buono. Exploring network optimizations for large-scale graph analytics. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC ’15, pages 26:1–26:10. ACM, 2015. ISBN 978-1-4503-3723-6.
- [27] J. J. Willcock, T. Hoefer, N. G. Edmonds, and A. Lumsdaine. AM++: A Generalized Active Message Framework. In *Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 401–410. ACM, 2010.
- [28] J. J. Willcock, T. Hoefer, N. G. Edmonds, and A. Lumsdaine. Active pebbles: a programming model for highly parallel fine-grained data-driven computations. In *Proc. 16th ACM symposium on Principles and practice of parallel programming*, pages 305–306. ACM, 2011.
- [29] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proc. 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015.
- [30] 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. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 25–25, Nov. 2005.
- [31] M. Zalewski, T. A. Kanewala, J. S. Firoz, and A. Lumsdaine. Distributed Control: Priority Scheduling for Single Source Shortest Paths Without Synchronization. In *Proc. of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 17–24. IEEE, 2014.