Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms

Gurbinder Gill[1], Roshan Dathathri[1], Loc Hoang[1], Andrew Lenharth[1], and Keshav Pingali[1]

The University of Texas at Austin, TX 78712, USA
{gill,roshan,loc}@cs.utexas.edu,
andrewl@lenharth.org,pingali@cs.utexas.edu

Abstract. The trend towards processor heterogeneity and distributed-memory has significantly increased the complexity of parallel programming. In addition, the mix of applications that need to run on parallel platforms today is very diverse, and includes graph applications that typically have irregular memory accesses and unpredictable control-flow. To simplify the programming of graph applications on such platforms, we have implemented a compiler called Abelian that translates shared-memory descriptions of graph algorithms written in the Galois programming model into efficient code for distributed-memory platforms with heterogeneous processors. The compiler manages inter-device synchronization and communication while leveraging state-of-the-art compilers for generating device-specific code. The experimental results show that the novel communication optimizations in the Abelian compiler reduce the volume of communication by 23×, enabling the code produced by Abelian to match the performance of handwritten distributed CPU and GPU programs that use the same runtime. The programs produced by Abelian for distributed CPUs are roughly 2.4× faster than those in the Gemini system, a third-party distributed CPU-only system, demonstrating that Abelian can manage heterogeneity and distributed-memory successfully while generating high-performance code.

Keywords: Graph analytics · Heterogeneous computing · Distributed computing · Compilers · High performance computing.

1 Introduction

Graph analytics systems must handle very large data-sets with billions of nodes and trillions of edges [16]. Graphs of this size are too big to fit into the memory of a single machine, so one approach is to use distributed-memory clusters consisting of multicore processors. Writing efficient distributed-memory programs can be difficult, so a number of frameworks and libraries such as Pregel [18], PowerGraph [12], and Gemini [33], have been developed to ease the burden of writing graph analytics applications for such machines. New trends in processor architecture have made this programming problem much more difficult. To reduce energy consumption, computer manufacturers are turning to *heterogeneous* processor architectures in which each machine has a multicore processor and GPUs or FPGAs. To exploit such platforms, we must tackle the twin challenges of *processor heterogeneity* and *distributed-memory computing*. Frameworks like

Lux [15] and Gluon [10] permit graph analytics applications writers to use distributed GPUs, but they require writing platform-specific programs that are not portable.

Ideally, we would have a compiler that takes single-source, high-level specifications of graph analytics algorithms and automatically translates them into distributed, heterogeneous implementations while optimizing them for diverse processor architectures. This paper describes such a compiler, called *Abelian*. Application programs are generalized vertex programs written in the Galois programming model, which provides programming patterns and data structures to support graph applications [20]. Section 2 describes this programming model in more detail. The Abelian compiler, described in Section 3, targets the Gluon runtime [10], which implements bulk-synchronous execution. Unlike other systems in this space, this runtime supports a number of graph partitioning policies including edge-cuts and vertex-cuts, and the programmer can choose any of these policies. The compiler exploits domain-knowledge to generate distributed code, inserting optimized communication code. Back-end compilers generate optimized code for NUMA multi-cores and GPUs from the output of Abelian.

The experimental results presented in Section 4 show that the communication optimizations in Abelian reduce communication volume by $23\times$, enabling Abelian-generated implementations to match the performance of hand-tuned distributed-CPU and distributed-GPU programs on the same platform. In addition, the distributed-CPU implementations produced by Abelian yield a geometric mean speedup of $2.4\times$ over those in the standalone distributed-CPU system Gemini [33] on the same hardware. This shows that the flexibility of Abelian in compiling a high-level, shared-memory, single address space specification for heterogeneous and distributed-memory architectures does not come at the cost of performance, even when compared to integrated, homogeneous systems.

2 Programming model

The Abelian compiler supports a generalized vertex programming model [12, 18, 33] that is a restriction of the Galois programming model [20, 24]. Nodes and edges of the graph have labels that are updated iteratively by the program until some quiescence condition is reached. Updating of labels is performed by applying *operators* to *active nodes* in the graph; this is called an *activity*. A push-style operator uses the label of the active node to conditionally update the labels of immediate neighbors of that node while a pull-style operator reads the labels of the immediate neighbors and conditionally updates the label of the active node.

Abelian supports more general operators than other systems in this space. In particular, an operator is allowed to update the labels of both the active node *and* its immediate neighbors, which is useful for applications like matrix completion using stochastic gradient descent. In addition, Abelian does not require updates to node labels to be reduction operations. For example, k-core decomposition evaluated in Section 4 uses subtraction on node labels.

In addition to the operator, the programmer must specify how active nodes are found in the graph [19]. The simplest approach is to execute the program in rounds and apply the operator to every node in each round. The order in which nodes are visited is unspecified, and the implementation is free to choose whatever order is convenient.

These *topology-driven algorithms* [24] terminate when a global quiescence condition is reached. The Bellman-Ford algorithm for single-source shortest-path (sssp) is an example.

An alternative strategy is to track active nodes in the graph and apply the operator only to those nodes, which potentially creates new active nodes. These *data-driven algorithms* [24] terminate when there are no more active nodes in the graph. As before, the order in which active nodes are to be processed is left unspecified, and the implementation is free to choose whatever order is convenient. Chaotic relaxation sssp uses this style of execution. Tracking of active nodes can be implemented by maintaining a *work-list* of active nodes. Alternatively, this can be implemented by marking active nodes in the graph and making sweeps over the graph, applying the operator only to marked nodes; we call this approach *filtering*. Fine-grain synchronization in marking and unmarking nodes can be avoided by using Jacobi-style iteration with two flags, say *current* and *next*, on each node; in a round, active nodes whose *current* flag is set are processed, and if a node becomes active in that round, its *next* flag is set using an ordinary write operation. The roles of these flags are exchanged at the end of each round. In our programming model, data-driven algorithms are written using work-lists, but the compiler transforms the code to use a filtering implementation. The correctness of this transformation is ensured by the fact that active nodes can be processed in any order.

Implementation: This programming model is implemented in C++ using the Galois library [20]. Figure 1 shows a program for push-style data-driven algorithm of pagerank. A work-list is used to track active nodes. The **Galois::for_each** in line 30 populates the work-list initially with all nodes in the graph and then iterates over it until the work-list is empty. The operator computes the update to the pagerank of the active node, and it pushes this update to all neighbors of the active node. If the residual at a neighbor exceeds some user-specified threshold, that neighbor becomes active and is pushed to the work-list.

The semantics of the **Galois::for_each** iterator permit work-list elements to be processed in any order. In a parallel implementation of the iterator, each operator application must appear to have been executed atomically. To ensure this, the application programmer must use data structures provided in the Galois library which include graphs, work-lists, and accumulators. This permits the runtime to manage updates to distributed data structures on heterogeneous devices and allows the compiler to treat data structures as objects with known semantics, which enables program optimization and generation of parallel code from implicitly parallel programs as described in Section 3.

Restrictions on operators: Like in other programming models for graph analytics [12, 33, 26, 15] and compilers for data-parallel languages [30, 27, 3], operators cannot perform I/O operations. They also cannot perform explicit dynamic memory allocation since some devices (like GPUs) have limited support for this in their runtimes. The library data structures can perform dynamic storage allocation, but this is done transparently to the programmer.

```
struct NodeData{
  // data on each node
  unsigned int nout; // out-degree
  float rank;
  std::atomic<float> res; // residual
};

struct PageRank {
  Graph* g;
  PageRank(Graph* g) : g(g) {}
  void operator()(GNode src,
                  Worklist& wl) {
    auto& sd = g->getData(src);
    auto res_old=sd.res.exchange(0);
    // apply residual to self
    sd.rank += res_old;
    auto delta=res_old*alpha/sd.nout;
    for (auto e : g->getEdges(src)) {
      GNode dst = g->getEdgeDst(e);
      auto& dd = g->getData(dst);
      // update residual of dest
      dd.res += delta;
      if (dd.res > tolerance) {
        wl.push(dst);
      }
    }
  }
};

Galois::for_each(g, PageRank{g});
```

Fig. 1. Pagerank source program

```
struct Add_contrib {
 typedef float ValTy;
 static ValTy extract(NodeData& node){
  return node.contrib;
 }
 static bool reduce(NodeData& node,
                    ValTy y) {
  add(node.contrib, y);
  return true;
 }
 static void reset(NodeData& node) {
  node.contrib = 0;
 }
};

struct Bcast_contrib {
 typedef float ValTy;
 static ValTy extract(NodeData& node){
  return node.contrib;
 }
 static void setVal(NodeData& node,
                    ValTy y) {
  node.contrib = y;
 }
};
```

Fig. 2. Compiler-generated synchronization structures for field contrib in pagerank

```
struct NodeData {
  // data on each node
  unsigned int nout; // out-degree
  float rank;
  float res; // residual
  // compiler added field
  std::atomic<float> contrib;
};
DistributedAccumulator work_done;
... // field-specific bitvector, flags
... // field-specific sync structures
struct PageRank {
  Graph* g;
  const float &l_alpha, &l_tolerance;
  ... // copy constructor for members
  void operator()(GNode src) {
   auto& sd = g->getData(src);
   if(sd.res > l_tolerance) {
     work_done += 1; // do not terminate
     auto res_old = sd.res;
     sd.res = 0;
     sd.rank += res_old;
     Bitvec_rank.set(src);
     auto delta=res_old*l_alpha/sd.nout;
     for (auto e:g->getEdges(src)) {
      GNode dst = g->getEdgeDst(e);
      auto& dd = g->getData(dst);
      dd.contrib += delta;
      Bitvec_contrib.set(dst);
     } } }
};
struct PageRank_splitOp {
  Graph* g;
  PageRank_splitOp(Graph* g) : g(g) {}
  void operator()(GNode src) {
    auto& sd = g->getData(src);
    sd.res += sd.contrib;
    Bitvec_res.set(src);
    sd.contrib = 0;
  }
};
... // 1st round for all nodes in
     initial work-list
do { // subsequent rounds: predicate-
     based filter
  work_done.reset(); // for termination

  ... // sync res if required: readSrc
  Galois::do_all(g.getSources(),
    PageRank{&g,alpha,tolerance});
  Flag_rank.set_writeSrc();
  Flag_contrib.set_reduceDst();

  if (Flag_contrib.is_reduceDst()) {
    graph.sync<reduceDst, readSrc,
      Add_contrib, Bcast_contrib>
      (Bitvec_contrib); // executed
    Flag_contrib.reset_reduceDst();
  }else if(Flag_contrib.is_reduceSrc()){
    // sync contrib: reduceSrc, readSrc
  } else{...}// sync contrib if required
  Galois::do_all(g.getSources(),
                 PageRank_splitOp{&g});
  Flag_res.set_writeSrc();
} while(work_done.reduce());
```

Fig. 3. Compiler-generated pagerank program

3 Abelian Compiler

Figure 4 is an overview of how input programs are compiled for execution on distributed, heterogeneous architectures. The Abelian compiler (implemented as a source-to-source translation tool based on Clang's libTooling) analyzes the patterns of data accesses in operators, restructures programs for execution on distributed-memory architectures, and inserts code for optimized communication. The output of the Abelian compiler is a bulk-synchronous parallel C++ program with calls to the Gluon [10] communication runtime (Figure 3). Gluon transparently handles the graph partitioning while loading the input graph. The generated code is independent of the partitioning policy, but the partitioning policy determines which portions of this code are executed. This permits Gluon's optimization that exploits structural invariants in partitioning without recompiling the program. The Abelian compiler also generates IrGL [22] intermediate representation kernels corresponding to each Galois::do_all call in the C++ program and inserts code in the C++ program to switch between calling the Galois::do_all and the corresponding IrGL kernel depending on the configuration chosen for the host (these are not shown in Figure 3 for brevity). The C++ program and the IrGL intermediate code are then compiled using device-specific compilers. The output executable is parameterized by the graph input, the partitioning policy, and the number of hosts and their configuration (CPU or GPU). The user can thus experiment with a variety of partitioning strategies and heterogeneous devices with a single command-line switch.

3.1 Graph-data Access Analysis

The access analysis pass analyzes the fields accessed in an operator. The results of this analysis are used to insert required communication code. Field accesses are classified as follows:

- *Reduction:* The field is read and updated using a reduction operation inside an edge iterator within the operator (e.g., addition to *residual* in line 22 in Figure 1). This is a common and important pattern in graph analytics applications.
- *Read:* The field is read, and it is not part of a reduction (e.g., read from *nout* in line 17 in Figure 1).
- *Write:* The field is written, and it is not part of a reduction (e.g., write to *rank* in line 16, Figure 1).

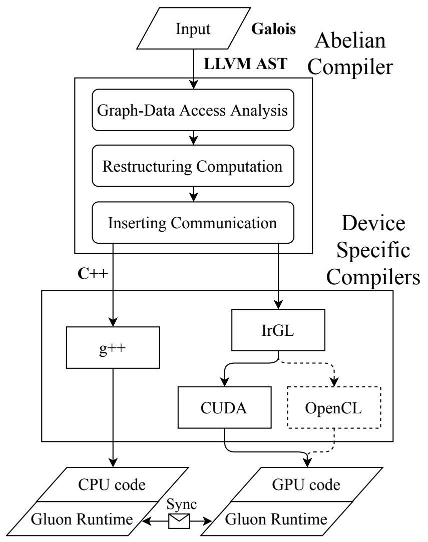

Fig. 4. System Overview

In addition, it is useful to abstract the context in which a field access is made.

- *At source:* The field is accessed at the source node of an edge.

- *At destination:* The field is accessed at the destination node of an edge.
- *At any:* The field is accessed at a node independent of any edge or at both endpoints of an edge.

3.2 Restructuring computation

The goal of computation restructuring is to bridge the semantic gap between the programming model, which has a single address space, and the execution model, which is distributed-memory and bulk-synchronous parallel. The semantics of Galois iterators permit iterations to be executed in parallel as long as each iteration appears to execute atomically. This fine-grain, iteration-level parallelism must be converted to round-based, bulk-synchronous parallelism by the Abelian compiler. This includes eliminating global variables (similar to *closure conversion* in functional languages) by adding them as members of the structure. This also requires two key transformations.

Splitting operators When active nodes are processed in parallel on a shared-memory machine, fine-grain synchronization may be needed for correct execution. This problem appears in a different guise on distributed-memory machines: if the two active nodes are on different hosts, proxies will be created on both hosts for the common neighbor, and it is necessary to reconcile the values pushed to these proxies so that the semantics of the program are respected. The bulk-synchronous execution model does not permit fine-grain synchronization, so these kinds of problems must be solved, in general, by breaking up the operator into phases if necessary and introducing `sync` calls between phases. There are a number of cases to consider depending on the type of field access as determined by the graph-data access analysis. We describe this for one such case.

In the PageRank source code in Figure 1, the *residual* field is read (line 14) to update the *rank* field (line 16) and written (line 14 using *exchange(0)*) at the *source*, but it is also reduced (line 22) at the *destination*. Since different hosts could update the residual, the hosts reading it should have the reduced value. To handle this, the compiler splits any operator that has such a dependence into multiple operators (a form of *loop fission*): one with only Read and Write accesses to the field and another with only Reduction accesses, as shown in the `PageRank` and `PageRank_splitOp` operators (lines 12-41) respectively in Figure 3. This may involve introducing new fields to store the intermediate values (e.g., `contrib`). The compiler also transforms some non-reduction read-after-write operations (e.g., subtraction) to equivalent reduction operations (e.g., addition) in a similar way. After this transformation, `sync` calls are introduced between the parallel phases, as described in Section 3.3.

Eliminating work-lists The Abelian compiler eliminates work-lists by using *filtering*, as explained in Section 2: in a given round, all nodes in the graph are visited and the operator is applied to nodes whose *current* flag is set. This flag is reset, and if a node becomes active in that round, its *next* flag is set; the roles of the flags are exchanged at the end of each round.

In some algorithms, the predicate used in the source program to push an active node to the work-list can be used during filtering to check if the node is active. Extracting this predicate involves a form of *loop fission*, and it avoids introducing flags and synchronizing their accesses. For example, in Figure 1, the code in lines 23-24 adds active nodes to the work-list. In the generated code, this is eliminated, and a new operator is created to conditionally activate nodes as shown in line 18 in Figure 3. Another operator is created to execute all nodes that would have been on the initial work-list (line 42). Abelian can also directly take filter-based implementation of data-driven algorithms as an input, in which case this transformation is not required. Termination is detected using a distributed accumulator (lines 19 and 63) provided by Gluon.

3.3 Inserting communication

The final pass of the Abelian compiler inserts code for communication and synchronization. A simple approach is the following: in each round, every mirror sends its value to its master where these values are combined, and the result is broadcast back to all the mirrors. This is essentially the gather-apply-scatter model used by most systems in this space, and it can be implemented by inserting a Gluon [10] `sync` call after each operator for every field that might be updated by that operator. Compilers for heterogeneous systems, such as Falcon [30], Dandelion [27], LiquidMetal [3], and DMLL [6], take a similar approach since their granularity of synchronization is an object or field. This coarse-grained approach can be seen as a more elaborate version of the write-broadcast cache coherence protocol used in systems with hardware cache-coherence. Abelian implements a different, fine-grained communication protocol to reduce the communication volume: a host sends the value of a field to other hosts only if that field has been updated in the previous rounds and if this value will be read in the current round. Static analysis is not adequate to determine these properties, so instrumentation code is inserted to track this dynamically. The actual communication is performed by the Gluon runtime, and it is invoked by inserting sync calls into the code.

Fine-grained communication In graph analytics applications, each round typically updates the field of only a small subset of graph nodes. A device-local, field-specific bit-vector is used to track updates to nodes' fields that participate in communication. The analysis pass determines points in the operator where these fields might be updated, and the compiler inserts instrumentation code at those points to also update the node's bit in the bit-vector for that field (lines 23, 29, 38 in Figure 3). The Gluon sync interface permits this bit-vector to be passed to the runtime system, which uses it to avoid sending node values that have not been updated in the current round.

On-demand communication Using the bit-vector ensures only updated values are communicated, but it does not permit Gluon's communication optimization that exploits structural invariants in partitioning policies [10]. To do so, the domain-specific knowledge of abstract write and read locations for the last reduction access(es) and next read access of the field must be specified, respectively. If it is unspecified or imprecise, Gluon may conservatively perform some redundant synchronization. The Abelian compiler can only precisely identify the abstract locations of fields accessed within an operator and cannot be precise about the future accesses. Therefore, after an operator, it inserts code that sets or invalidates the sync-state invalidation flags for fields that could be written in the operator using its write location (lines 49, 50, 62 in Figure 3). Before an operator, it inserts the synchronization structures, as shown in Figure 2 (equivalent GPU functions generated for a vector of nodes are omitted for brevity), and the communication code for fields that could be read in the operator (lines 46, 52-59 in Figure 3). The code checks the field-specific sync-state flags and calls the Gluon sync routine with the precise write and read locations if the flag is invalidated.

3.4 Device-specific compilers

The Abelian compiler outputs C++ code that can be compiled using existing compilers like g++ to execute on shared-memory NUMA multicores using the Galois runtime [20]. A naive translation of this C++ code to CUDA or OpenCL is not likely to yield high-performance code because it will not exploit SIMD execution. We instead use the IrGL [22] compiler, which produces highly optimized CUDA and OpenCL code from an intermediate representation that is intended for graph applications. This compiler exploits nested parallelism, which is important when processing scale-free graphs. To interface with the IrGL compiler, the Abelian compiler generates IrGL intermediate code, translating data layout of fields from arrays of structures to structures of arrays.

4 Experimental results

To evaluate the performance of programs generated by the Abelian compiler, we studied a number of graph analytical applications: betweenness centrality (bc), breadth-first search (bfs), connected components (cc), k-core decomposition (kcore), pagerank (pr), single-source shortest path (sssp), and matrix completion using stochastic gradient descent (sgd). We specify the programs in Galois C++: pull-style topology-driven algorithm for pr, push-and-pull-style topology-driven algorithm for sgd, and push-style work-list-driven algorithms for the rest. The Abelian compiler analyzes the program, restructures the operators, and synthesizes precise communication. Unless otherwise noted, all optimizations are applied in our evaluation, including eliminating work-lists. The programs work with different partitioning policies. In our evaluation, we choose incoming edge-cut for pr, cartesian vertex-cut for sgd, and outgoing edge-cut for all other benchmarks. We have empirically found these policies to work well in practice; an exhaustive search to find the best policy is outside the scope of this work.

Table 1 shows the input graphs we used along with their properties. All the CPU experiments were done on the Texas Advanced Computing Center's [2] Stampede [28] KNL Cluster. For GPU experiments, the Bridges [21] supercomputer at the Pittsburgh Supercomputing Center [1, 29] was used. Table 2 shows

Table 1. Inputs and their key properties

	clueweb12 [25]	**kron30** [17]	**rmat28** [7]	**amazon** [13]
$\lvert V\rvert$	978M	1073M	268M	31M
$\lvert E\rvert$	42,574M	10,791M	4,295M	82.5M
$\lvert E\rvert/\lvert V\rvert$	44	16	16	2.7
max D_{out}	7,447	3.2M	4M	44557
max D_{in}	75M	3.2M	0.3M	25366

the configuration of these clusters used in our experiments. In all our experiments, we choose the max-degree node as the source for bc, bfs, and sssp. For kcore, we solve for $k = 100$. We present the mean execution time of 3 runs, excluding graph partitioning time. *We run pr and sgd for 100 and 50 iterations, respectively; all other algorithms are run until convergence.*

Table 2. Cluster configurations

	Stampede (CPU)	**Bridges (GPU)**
NIC	Omni-path	Omni-path
Machine	Intel Xeon Phi KNL	4 NVIDIA Tesla K80s
No. of hosts	32	16
Each host	272 threads	1 Tesla K80
Memory	96GB DDR4	128GB DDR5
Compiler	g++ 7.1	g++ 5.3

Table 3. Bridges: execution time (in seconds) on 16 GPUs for rmat28

	D-IrGL	**Abelian**
bc	9.6	9.6
bfs	1.1	1.2
cc	2.6	2.7
kcore	1.5	1.5
pr	32.9	30.5
sssp	2.5	2.5

4.1 Comparison with the state-of-the-art

We compare the performance of Abelian compiler-generated programs with handwritten D-Galois programs for CPU-only systems [10] and handwritten D-IrGL programs for GPU-only systems [10]. D-Galois and D-IrGL programs have explicit synchronization specified by the programmer; in contrast, synchronization in programs produced by the Abelian compiler is introduced automatically by the compiler. However, all these programs use Gluon [10], a communication substrate that optimizes communication at runtime by exploiting structural and temporal invariants in partitioning (Gluon uses LCI [9] for message transport between hosts). In addition, D-Galois and Abelian use the same Galois [20] computation operators on the CPU while D-IrGL and Abelian use the same IrGL [22] computation kernels on the GPU. Therefore, differences in performance between Abelian-generated code and D-Galois/D-IrGL code arise mainly from differences in how synchronization code is inserted by the Abelian compiler.

Table 4. Stampede: execution time (in seconds) (H: hosts)

		Gemini		**D-Galois**		**Abelian**	
		8H	**32H**	**8H**	**32H**	**8H**	**32H**
bc	**clueweb12**	-	-	OOM	430.4	OOM	437.6
	kron30	-	-	41.3	27.0	39.7	27.3
bfs	**clueweb12**	OOM	69.9	11.6	9.1	12.0	10.1
	kron30	5.1	7.1	5.1	4.0	5.2	4.2
cc	**clueweb12**	39.3	38.8	OOM	16.5	OOM	18.3
	kron30	15.8	14.8	7.6	4.6	7.7	4.0
kcore	**clueweb12**	-	-	OOM	290.4	OOM	289.1
	kron30	-	-	4.4	3.0	4.5	3.0
pr	**clueweb12**	OOM	257.9	395.1	248.0	402.1	277.4
	kron30	245.1	232.4	278.1	221.9	281.0	232.5
sssp	**clueweb12**	OOM	128.3	OOM	14.3	OOM	15.8
	kron30	14.0	14.9	9.4	8.2	9.3	8.2
sgd	**amazon**	-	-	1570.2	701.6	1570.2	696.2

We also compare Abelian-generated programs with distributed-CPU programs written in the Gemini framework [33] (Gemini does not have kcore and sgd; bc in Gemini uses bfs while that in Abelian uses sssp, so it is omitted). Gemini has explicit communication messages in the programming model, and it provides a third-party baseline for our study.

Table 3 and Table 4 show the distributed-GPU and distributed-CPU results. Abelian programs match the performance of D-Galois and D-IrGL programs; the difference is not more than 12%. Gemini is 15% faster than Abelian for pr with kron30 on 8 hosts. In all other cases, Abelian matches or outperforms Gemini. The geometric mean speedup of Abelian over Gemini on 32 KNL hosts is $2.4\times$. These results show that Abelian is able to compile a high-level, shared-memory, single address space specification into efficient implementations that either match or beat the state-of-the-art graph analytics platform. Although the Abelian compiler produces code for heterogeneous devices, we do not report numbers for distributed CPU+GPU execution because the 4 GPUs on a node on Bridges outperform the CPU by a significant margin.

4.2 Impact of communication optimizations

We analyze the performance impact of the communication optimizations in Abelian (Section 3.3) by comparing three levels of communication optimization.

1. *Unoptimized* (UO): the Gluon sync call is inserted for a field after an operator if it could be updated in that operator. The bit-vector as well as the abstract write and read locations are left unspecified, so all elements in the field are synchronized. Existing compilers for heterogeneous systems like Falcon [30], Dandelion [27], and Liquid Metal [3] do similar field-specific, coarse-grained synchronization.
2. *Fine-grained communication optimization* (FG): the compiler instruments the code to use a bit-vector that dynamically tracks updates to fields. The Gluon sync call used is the same as in UO, but it only synchronizes the elements in the field that have been updated using the bit-vector. This is similar to existing graph analytical frameworks [12, 8, 33] that synchronize only the updated elements.
3. *Fine-grained and on-demand communication optimization* (FO): this (default of Abelian compiler) uses on-demand communication along with fine-grained optimization. It instruments invalidation flags to track fields that have been updated and inserts Gluon sync calls before an operator for fields that could be read in the operator, thereby precisely identifying both the abstract write and read locations. This enables Gluon's communication optimization that exploits structural invariants in partitioning policies.

We compare these three communication optimization variants with hand-tuned (HT) programs written in D-Galois and D-IrGL on distributed CPUs and distributed GPUs respectively. In these programs, the programmer (with global control-flow knowledge) specified the precise communication using Gluon sync calls.

Figure 5 and Figure 6 present the comparison results on 32 KNL hosts of Stampede and 16 GPU devices of Bridges respectively. Each bar in the figures shows the execution time (maximum across hosts). We measure the maximum computation time across hosts in each round and take their sum, which is the total computation time (top). The rest of the execution time is non-overlapped communication time (bottom). We also measure the total communication volume across all rounds, shown in text on the bars.

The trends are clear in the figure. Each optimization reduces communication volume and time, improving execution time further. FG significantly reduces communication volume and time over UO, with the exception of pr. FG performs atomic updates to the bit-vector, which could be overhead when the updates are dense, like in pr. FO optimizes the communication volume and time further to match the performance of HT. FO reduces communication volume by $23\times$ over UO, yielding a geometric mean execution

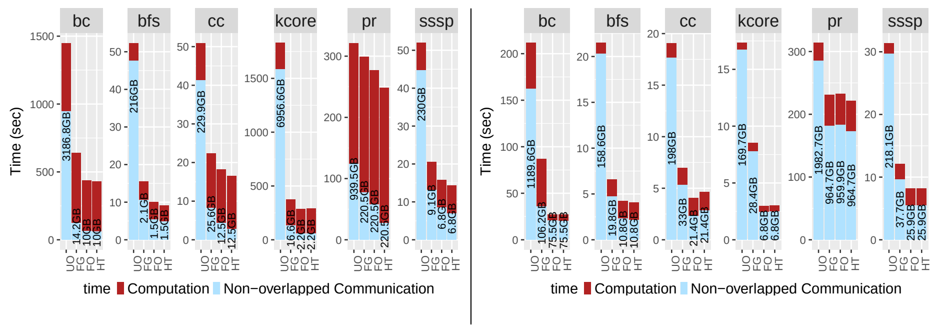

Fig. 5. 32 KNL hosts on Stampede: clueweb12 (left) and kron30 (right). Different variants are: UnOpt (UO), Fine-Grained opt (FG), Fine-grained+On-demand opt (FO), Hand-Tuned(HT)

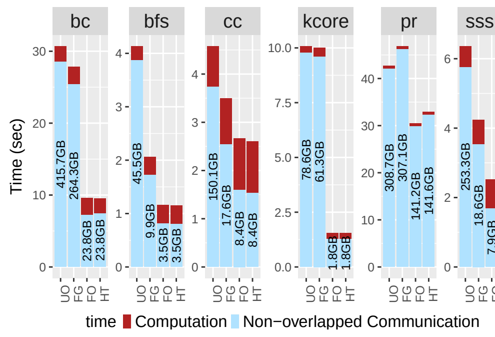

Fig. 6. 16 GPU devices on Bridges: rmat28

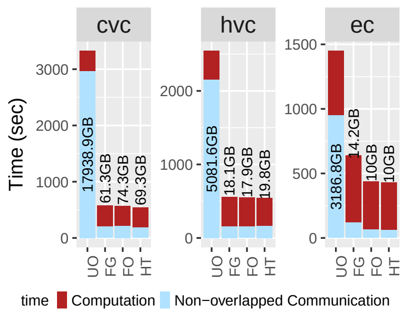

Fig. 7. 32 KNL hosts on Stampede: partitionings for bc on clueweb12

time speedup of 3.4×. Fine-grained and on-demand communication optimizations (FO) are thus essential to match the performance of HT on both CPUs and GPUs.

Abelian compiler-generated programs can support different partitioning policies, and we study whether they can fully exploit Gluon's partition-aware optimizations like HT. Figure 7 presents the comparison results for bc on clueweb12 using different partitioning policies namely, cartesian vertex cut [5] (cvc), hybrid vertex-cut [8] (hvc), and outgoing edge cut (ec). This shows that FO matches the performance of HT, although FG does not. This shows that the compiler can capture sufficient domain-specific knowledge to aid the Gluon runtime in performing partition-aware optimizations.

5 Related work

Distributed graph processing systems: Many frameworks [18, 12, 31, 8, 33, 15, 10] exist which provide a runtime to simplify writing distributed graph analytics algorithms. Like Abelian, these systems use a vertex programming model and bulk-synchronous parallel (BSP) execution. Abelian is the first compiler that synthesizes the required communication. Our evaluation shows that the programs generated by the Abelian compiler that use the Gluon [10] runtime match hand-tuned programs in the Gluon system and outperform those in the Gemini [33] system.

Single-host heterogeneous graph processing systems: There are several frameworks for graph processing on a single GPU [22], multiple GPUs [4, 23, 32] and multiple GPUs with a CPU [11]. All of these are restricted to a single physical node that connects all devices unlike our system, and consequently, they cannot handle graphs as large as the ones our system can. Abelian leverages the throughput optimizations in the IrGL [22] compiler that are essential for performance on power-law graphs. Unlike IrGL, which compiles an intermediate-level program representation to CUDA, the Abelian compiler not only generates this from a high-level C++ program but also synthesizes synchronization code to execute the compiled code on multiple devices in multiple hosts.

Compilers for distributed or heterogeneous architectures: Liquid Metal [3] compiles the Lime language to heterogeneous CPUs, GPUs, and FPGAs. Dandelion [27] compiles high-level LINQ programs to distributed heterogeneous systems. Green-Marl [14] is a DSL that is compiled to Pregel. Brown et al. [6] compile a data-parallel intermediate language DMLL to multicores, clusters, and GPUs. Upadhyay et al. [30] compile a domain-specific language, Falcon, to Giraph code for CPU clusters and MPI+OpenCL code for GPU clusters, but it does not do GPU-specific computation restructurings like nested parallelism which Abelian compiler does using IrGL. In all these compilers, the granularity of communication is an object or field, whereas Abelian identifies fine-grained elements of a label-array and communicates them precisely using the Gluon runtime. Moreover, none of the existing compilers use domain-specific analysis and computation restructurings for graph analytical applications like Abelian.

6 Conclusions

Abelian is the first graph analytics compiler that can produce high-performance, distributed, heterogeneous implementations from high-level, shared-memory, single address space specification of graph algorithms. It splits operators and eliminates worklists to make the programs bulk-synchronous. The fine-grained, on-demand communication optimizations in Abelian yield a speedup of 3.4$\times$ over field-specific, coarse-grained communication code generated by existing compilers. This enables the generated implementations to match the performance of hand-tuned implementations for distributed CPUs and distributed GPUs in the state-of-the-art Gluon system using the same computation engines on the same hardware. The distributed-CPU implementations produced by Abelian also yield a geometric mean speedup of 2.4$\times$ over programs in the distributed CPU-only system Gemini on the same hardware. This shows that the Abelian compiler can manage heterogeneity and distributed-memory successfully while generating high-performance code, even in comparison to homogeneous systems.

Acknowledgments

This research was supported by NSF grants 1337217, 1337281, 1406355, 1618425, 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Bridges system, supported by NSF award number ACI-1445606 at the Pittsburgh Supercomputing Center, and the Stampede system at the Texas Advanced Computing Center at The University of Texas at Austin.

References

1. Pittsburgh Supercomputing Center (PSC) (2018), https://www.psc.edu/
2. Texas Advanced Computing Center (TACC), The University of Texas at Austin (2018), https://www.tacc.utexas.edu/
3. Auerbach, J., Bacon, D.F., Burcea, I., Cheng, P., Fink, S.J., Rabbah, R., Shukla, S.: A compiler and runtime for heterogeneous computing. DAC (2012). https://doi.org/10.1145/2228360.2228411, http://doi.acm.org/10.1145/2228360.2228411
4. Ben-Nun, T., Sutton, M., Pai, S., Pingali, K.: Groute: An asynchronous multi-gpu programming model for irregular computations. PPoPP (2017). https://doi.org/10.1145/3018743.3018756, http://doi.acm.org/10.1145/3018743.3018756
5. Boman, E.G., Devine, K.D., Rajamanickam, S.: Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In: 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 1–12 (Nov 2013). https://doi.org/10.1145/2503210.2503293
6. Brown, K.J., Lee, H., Rompf, T., Sujeeth, A.K., De Sa, C., Aberger, C., Olukotun, K.: Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. CGO (2016). https://doi.org/10.1145/2854038.2854042, http://doi.acm.org/10.1145/2854038.2854042
7. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining, pp. 442–446 (2004). https://doi.org/10.1137/1.9781611972740.43
8. Chen, R., Shi, J., Chen, Y., Chen, H.: PowerLyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys (2015). https://doi.org/10.1145/2741948.2741970, http://doi.acm.org/10.1145/2741948.2741970
9. Dang, H.V., Dathathri, R., Gill, G., Brooks, A., Dryden, N., Lenharth, A., Hoang, L., Pingali, K., Snir, M.: A lightweight communication runtime for distributed graph analytics. IPDPS (2018)
10. Dathathri, R., Gill, G., Hoang, L., Dang, H.V., Brooks, A., Dryden, N., Snir, M., Pingali, K.: Gluon: A communication optimizing framework for distributed heterogeneous graph analytics. PLDI (2018)
11. Gharaibeh, A., Beltrão Costa, L., Santos-Neto, E., Ripeanu, M.: A yoke of oxen and a thousand chickens for heavy lifting graph processing. PACT (2012)
12. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed graph-parallel computation on natural graphs. OSDI (2012), http://dl.acm.org/citation.cfm?id=2387880.2387883
13. He, R., McAuley, J.: Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. WWW (2016). https://doi.org/10.1145/2872427.2883037, https://doi.org/10.1145/2872427.2883037
14. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. ASPLOS (2012). https://doi.org/10.1145/2150976.2151013, http://doi.acm.org/10.1145/2150976.2151013
15. Jia, Z., Kwon, Y., Shipman, G., McCormick, P., Erez, M., Aiken, A.: A distributed multi-gpu system for fast graph processing. Proc. VLDB Endow. (Nov 2017). https://doi.org/10.14778/3157794.3157799, https://doi.org/10.14778/3157794.3157799

16. Lenharth, A., Nguyen, D., Pingali, K.: Parallel graph analytics. Commun. ACM **59**(5), 78–87 (Apr 2016). https://doi.org/10.1145/2901919, http://doi.acm.org/10.1145/2901919
17. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: An approach to modeling networks. J. Mach. Learn. Res. **11**, 985–1042 (Mar 2010), http://dl.acm.org/citation.cfm?id=1756006.1756039
18. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. SIGMOD (2010). https://doi.org/10.1145/1807167.1807184, http://doi.acm.org/10.1145/1807167.1807184
19. Nasre, R., Burtscher, M., Pingali, K.: Data-driven versus Topology-driven Irregular Computations on GPUs. In: Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium. IPDPS '13, Springer-Verlag, London, UK (2013)
20. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. SOSP (2013), http://doi.acm.org/10.1145/2517349.2522739
21. Nystrom, N.A., Levine, M.J., Roskies, R.Z., Scott, J.R.: Bridges: A uniquely flexible HPC resource for new communities and data analytics. In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure. pp. 30:1–30:8. XSEDE '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2792745.2792775, http://doi.acm.org/10.1145/2792745.2792775
22. Pai, S., Pingali, K.: A compiler for throughput optimization of graph algorithms on GPUs. In: OOPSLA (2016)
23. Pan, Y., Wang, Y., Wu, Y., Yang, C., Owens, J.D.: Multi-gpu graph analytics. IPDPS (May 2017). https://doi.org/10.1109/IPDPS.2017.117
24. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X.: The TAO of parallelism in algorithms. In: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation. pp. 12–25. PLDI '11 (2011), http://doi.acm.org/10.1145/1993498.1993501
25. Project, T.L.: The clueweb12 dataset (2013), http://lemurproject.org/clueweb12/
26. Prountzos, D., Manevich, R., Pingali, K.: Synthesizing parallel graph programs via automated planning. In: Programming Language Design and Implementation. PLDI'15 (2015)
27. Rossbach, C.J., Yu, Y., Currey, J., Martin, J.P., Fetterly, D.: Dandelion: A compiler and runtime for heterogeneous systems. SOSP (2013). https://doi.org/10.1145/2517349.2522715, http://doi.acm.org/10.1145/2517349.2522715
28. Stanzione, D., Barth, B., Gaffney, N., Gaither, K., Hempel, C., Minyard, T., Mehringer, S., Wernert, E., Tufo, H., Panda, D., Teller, P.: Stampede 2: The evolution of an XSEDE supercomputer. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact. pp. 15:1–15:8. PEARC17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3093338.3093385, http://doi.acm.org/10.1145/3093338.3093385
29. Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., et al.: XSEDE: accelerating scientific discovery. Computing in Science & Engineering **16**(5), 62–74 (2014)
30. Upadhyay, N., Patel, P., Cheramangalath, U., Srikant, Y.N.: Large scale graph processing in a distributed environment. In: Euro-Par 2017 Parallel Processing Workshops (2018)
31. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: A resilient distributed graph system on spark. GRADES (2013)
32. Zhong, J., He, B.: Medusa: Simplified graph processing on gpus. IEEE TPDS (2014). https://doi.org/10.1109/TPDS.2013.111
33. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: A computation-centric distributed graph processing system. OSDI (2016), http://dl.acm.org/citation.cfm?id=3026877.3026901