DisGCo : A Compiler for Distributed Graph Analytics

DOUBLE BLIND

Graph algorithms are widely used in various applications. Their programmability and performance have garnered a lot of interest among the researchers. Being able to run these graph analytics programs on distributed systems is an important requirement. Green-Marl is a popular Domain Specific Language (DSL) for coding graph algorithms and is known for its simplicity. However, the existing Green-Marl compiler for distributed systems (Green-Marl to Pregel) can only compile limited types of Green-Marl programs (in Pregel canonical form). This severely restricts the types of parallel Green-Marl programs that can be executed on distributed systems. We present *DisGCo*, the first compiler to translate any general Green-Marl program to equivalent MPI program that can run on distributed systems.

Translating Green-Marl programs to MPI (SPMD/MPMD style of computation, distributed memory) throws up many other exciting challenges, besides the issues related to differences in syntax, as Green-Marl gives the programmer a unified view of the whole memory and allows the parallel and serial code to be inter-mixed. We first present the set of challenges involved in translating Green-Marl programs to MPI and then present a systematic approach to do the translation. We also present a few optimization techniques to improve the performance of our generated programs. *DisGCo* is the first graph DSL compiler that can handle all syntactic capabilities of a practical graph DSL like Green-Marl and generate code that can run on distributed systems. Our preliminary evaluation of *DisGCo* shows that our generated programs are scalable. Further, compared to the state-of-the-art `DH-Falcon` compiler that translates a subset of Falcon programs to MPI, our generated codes exhibit a geomean speedup of 17.32×.

ACM Reference format:
Double Blind. 2020. DisGCo : A Compiler for Distributed Graph Analytics. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (June 2020), 25 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graph algorithms find applications in a variety of fields for various problems. Non-uniform distribution of the node degrees in the input graphs and unpredictable data access patterns in the algorithms, pose many exciting challenges in efficiently implementing these algorithms, especially for the emerging parallel systems. Considering the challenges in implementing parallel graph algorithms using traditional general-purpose high-level languages (for example, C++, Java, and so on), researchers have proposed languages/frameworks/libraries like GraphLab [35], PowerGraph[20], Gemini[58], Pregel [37], Green-Marl [27], `DH-Falcon` [13] that provide different APIs for writing parallel graph algorithms. Among these proposed approaches, Green-Marl and `DH-Falcon` are high-level domain-specific languages which allow graph algorithms to be expressed in an imperative programming style.

XXXX-XXXX/2020/6-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Though graph DSLs like Green-Marl and DH-Falcon present a convenient way to code graph algorithms, their usage in the context of distributed systems is still in its nascent stage. In this paper, we focus on the popular Green-Marl DSL, known for its speed on shared memory systems [27].

Currently, Green-Marl supports three backends: OpenMP [27], CUDA [48] and Pregel [28]. The OpenMP and CUDA backends generate code to be run on shared memory systems and GPGPUs, respectively. Even though the Pregel backend can be used to compile Green-Marl programs to be run on distributed systems, the backend can only translate programs in Pregel canonical form [28]: a small subset of possible Green-Marl programs. For example, of the 27 programs in the Green-Marl repository, only seven could be compiled by the existing Pregel backend. To the best of our knowledge, there is no existing compiler for Green-Marl that can translate any general Green-Marl programs to programs that can run on distributed systems. In this paper, we present the design of *DisGCo* (Distributed Green-Marl Compiler), the first compiler that can be used to compile arbitrary Green-Marl codes to equivalent C++ MPI code.

Message Passing Interface (MPI) [2] is a standard (library specification) for distributed programming, which is widely used for its portability and performance. MPI supports two models of programming; Point-to-Point (P2P) using *sends* and *receives* (*a.k.a.* two-sided communication model), and Remote Memory Access (RMA) [2, 26, 51] using *puts* and *gets* (*a.k.a.* one-sided communication model). The state-of-the-art DH-Falcon compiler translates a subset of programs written in DH-Falcon to MPI programs using P2P communication. Prior research [18, 24] has shown that RMA based communication can provide a better overlap of computation and communication by avoiding CPU intervention in communication and hence could provide better performance than point-to-point programs. Recently, the pioneering works of Li et al. [32, 33] show that RMA can be used for graph processing systems to improve their performance over P2P implementations significantly. In this paper, we leverage their experience to design a novel translation scheme from Green-Marl to an MPI RMA. To the best of our knowledge, ours is the first work that exploits the RMA based communication to translate graph algorithms and high-level data structures of a practical graph DSL.

There are many fundamental differences between Green-Marl and MPI that makes the translation quite challenging. These challenges arise due to the following three main factors. (i) Differences in syntax; (ii) The programmer view of data storage: In contrast to MPI, where the data can be distributed across different processes, Green-Marl views all data as available locally in the shared memory; (iii) Model of computation: In contrast to the SPMD/MPMD style of computation in MPI, the Green-Marl programs can have the parallel and serial code inter-mixed. We first identified the underlying challenges resulting from these differences, and developed novel approaches for mapping Green-Marl features onto MPI codes. Using the *DisGCo* compiler, the application developers can take advantage of the high-level programmability aspects of Green-Marl, while harnessing the capabilities of distributed systems to run complex graph algorithms on large input graphs. Though we develop our techniques in the context of Green-Marl, we believe these can be extended to other DSLs (like DH-Falcon and GraphIt [57]) that provide a shared memory view of data, to generate performant MPI RMA code.

Our Contributions:

• We present *DisGCo*, the first Green-Marl compiler which can compile arbitrary Green-Marl programs to MPI RMA programs that can run on a distributed set-up.

• We identified the challenges involved in compiling each Green-Marl construct, and then we devised and implemented an novel translation scheme in *DisGCo*. Considering the importance of the underlying data distribution scheme, we have designed *DisGCo* to admit any arbitrary programmer specified vertex partitioning scheme.

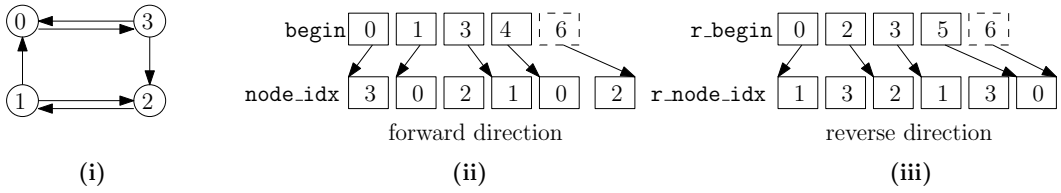


Fig. 1. An example representation in CSR.

• We have also implemented a set of optimizations in *DisGCo* to improve the performance of our generated codes.
• We present an evaluation of *DisGCo* over a set of five popular kernels; we found that our programs are scalable, and the optimizations bring-in impressive performance benefits.
• We compare the performance of our generated codes against the MPI codes generated by the state-of-the-art `DH-Falcon` compiler and show that the *DisGCo* generated programs lead to a geomean speedup of 17.32× over `DH-Falcon` generated codes.
The paper is organized as follows: In Section 2, we provide the reader with a brief background of the basics of Green-Marl and MPI RMA. In Section 3, we discuss some of the main challenges encountered in designing and implementing *DisGCo*. In Section 4, we illustrate the translation scheme used by *DisGCo* for handling various constructs in Green-Marl. In Section 5, we present some optimizations designed to improve the code generated by *DisGCo*, and in Section 6 we discuss some salient issues of *DisGCo*. In Section 7, we present a detailed evaluation of *DisGCo*. We discuss the related work in Section 8 and conclude in Section 9.

2 BACKGROUND

In this section, we briefly cover some of the background details required for this paper.

2.1 Green-Marl Features

We now briefly describe some of the features of Green-Marl and its existing runtime, relevant for this paper. More details can be found in the Green-Marl language specification [1].
Graph representation. The current Green-Marl runtime stores the input graph using a space-efficient representation called Compressed Sparse Row (CSR) format. In CSR format, for each graph (V, E), two arrays are maintained: `node_idx` and `begin`. The array `node_idx`, as the name indicates, stores the vertex-ids. The edges are grouped according to source vertices, and each group is then sorted based on the source vertex-ids to give us an ordered list of edges. The destination vertex-id of all the edges (in the ordered list) are stored in order in `node_idx`. The `begin` array contains $1 + |V|$ number of elements. For any vertex with vertex-id v, the outgoing edges of v will have edge-ids starting from `begin`$[v]$ to `begin`$[v+1]-1$. For $i \in$ [`begin`$[v] \cdots$ `begin`$[v+1]-1$], `node_idx`$[i]$ gives the $(i - \texttt{begin}[v])^{th}$ neighbor of v. Further, for a fast lookup for incoming edges, Green-Marl runtime uses a reverse lookup with two more arrays `r_begin` and `r_node_idx` (for reverse edges). For the graph shown in Figure 1(i), Figure 1(ii) shows the forward maps and Figure 1(iii) shows the reverse maps in CSR format representation.
Datatypes. Besides the scalar data types (such as `int`, `bool`, and so on), Green-Marl supports a few graph specific data-types such as `node`, `egde`, `DGraph` (representing directed graph), and `UGraph` (representing undirected graph). Nodes and edges can have associated properties of types `N_P` and `E_P`, respectively. In Green-Marl, graph elements can also be stored in sets, orders, or sequences.

Loops and Iterators. Besides the sequential `for`-loops, Green-Marl supports parallel-loops (using the `Foreach` construct). These looping constructs can be used to iterate over nodes, edges, or different collections. Each of these looping constructs can have an optional argument that sets up a filter to decide if the specific iteration has to be executed. For example, in Figure 2, at line 13, the condition ensures that the node `n` will be processed, only if the value of the node property `n.updated` is set to true.

Green-Marl also supports special iterators like `inBFS` and `inDFS` that allow nodes to be visited in BFS and DFS order, respectively. For example, the following example iterates over the nodes of the graph `G`, using `src` as the source node in BFS order. For each node in the BFS order, it executes `SB1`, and at the end of the BFS travel, `SB2` is executed for each node in the reverse BFS order.

```
1 inBFS (Iter_name : G.nodes from src)
2 { SB1; /* statement block 1 */}
3 inReverse
4 { SB2; /* statement block 2 */}
```

Green-Marl also provides iterators like `Nbr` to iterate over the neighbors of a node and `CommonNbrs` to iterate over the common neighbors of two nodes. For example, the Green-Marl code '`for (s in u.CommonNbrs(v)) {S1}`' executes `S1` for the common neighbors of `u` and `v`.

Reduction. Green-Marl supports many reduction operations like `Sum, Product, Count, Min, Max,` and `Exists` for performing reductions inside a parallel region. For example, in Figure 2, at line 16, the property `s.dist_next` is set to the minimum of `n.dist + e.length` and `s.dist_next`, and if the value is updated, the property `s.updated_next` is set to true. Similarly, at line 22, the variable `fin` is set, if the property `updated` is set for any node.

We now briefly explain the usage of Green-Marl syntax using an example program shown in Figure 2, which implements the Bellman Ford's version [15] of SSSP. Here, the formal parameter, `dist` is an example of a node property, and `len` is an example of an edge property. The lines 6 to 9 show group assignments where the properties of all the nodes/edges are initialized to the value of the expression on the RHS. The main computation spans the two nested `Foreach` loops, which together iterate over all the edges in the graph and update the shortest distance of the edge's destination. The computation inside `Foreach` loops is repeated inside an outer `while-loop` until no node has its distance updated to a new value.

2.2 Programming in MPI RMA

We now cover an introduction to programming in MPI RMA.

Parallelism in MPI. We use the SPMD (single program multiple data) style of parallelism in our generated MPI programs. In SPMD programs, one or more processes are created to execute the same program; the data is distributed among all the processes. Each data item has a unique "owner" process. Each process works on its "local" data and may access (read/write) the data of remote processes, by communicating with the owner process of the remote data. In the context of graphs, each process owns a local sub-graph with local vertices, local edges, and local vertex/edge properties. Further, with each vertex and edge, we maintain a local-id and a global-id.

Programming in MPI RMA. Using MPI RMA (one-sided communication model), a process (called the origin process) can directly access the memory of a target process, without the target necessarily participating in the communication. This is achieved through a handle called *window*, to which any remotely accessible memory is attached. MPI supports different types of *windows*, of which, *DisGCo* uses *dynamic windows* to which any arbitrary memory can be attached to (or detached from). For any program generated, *DisGCo* maintains a single *dynamic window* (in a variable `win`), on which, we use `MPI_Get` (or `MPI_Put`) to read (or write) remote data. The calls `MPI_Get` and `MPI_Put` take

```
1  Procedure sssp(G:Graph, dist:N_P<Int>, len:E_P<Int>, root:Node) {
2   N_P<Bool> updated;
3   N_P<Bool> updated_nxt;
4   N_P<Int>  dist_nxt;
5   Bool fin = False;
6   G.dist = (G == root) ? 0: +INF;
7   G.updated = (G == root) ? True: False;
8   G.dist_nxt = G.dist;
9   G.updated_nxt = G.updated;
10
11  while(!fin) {
12   fin = True
13   Foreach (n: G.Nodes)(n.updated) {
14    Foreach (s: n.Nbrs) {
15     Edge e = s.ToEdge(); // the edge to s
16     <s.dist_nxt; s.updated_nxt> min=  <n.dist + e.len; True>;
17   } /*Foreach*/ } /*Foreach*/
18
19   G.dist = G.dist_nxt;
20   G.updated = G.updated_nxt;
21   G.updated_nxt = False;
22   fin = ! Exist(n: G.Nodes){n.updated}; } /*while*/ } /*procedure sssp*/
```

Fig. 2. SSSP written in Green-Marl.

three main arguments: the window handle, the offset of the memory in window, and the rank of the target process.

As suggested by Li et al. [32], to improve the communication-computation overlap, we use passive synchronization using the `MPI_Win_lock` and `MPI_Win_unlock` calls. The period between a lock and unlock is called an *epoch*, and any reads or writes to a remote process within an epoch are ensured to be over by the end of the epoch. Within an epoch, the MPI call `MPI_Win_flush` can be used to ensure the completion of a remote access. To reduce a value across different processes, MPI supports many reduction operations using the `MPI_All_reduce` call that makes the result available to all the processes.

For the ease of presentation, we use a set of concise macros (described in Figure 3), instead of the detailed MPI calls, in our further discussions.

3 CHALLENGES IN GREEN-MARL TO MPI RMA TRANSLATION

Considering the differences in the underlying design philosophies of Green-Marl and MPI, it is natural that there is no direct one-to-one correspondence between the constructs/features of both. In this section, we discuss the different challenges that we encounter when we go about translating the various Green-Marl constructs/features to MPI.

3.1 Parallelism Specification and Data Layout

Parallelism in Green-Marl is specified using explicit `Foreach` statements (see Figure 2, for example). However, MPI does not have any direct syntax for specifying such parallel-for-loops.

While writing a Green-Marl program, the programmer does not specify where the data is actually located (local or remote). In contrast, MPI allows the data to be distributed across different processes (possibly running on multiple nodes), thereby improving the scalability of the programs (to handle large data). Here, different processes can access the data of other processes via remote communication. Efficiently mapping and accessing the data structures of Green-Marl (such as graphs, maps, properties, and so on) from a shared-memory view to a distributed-memory view

Macro	Explanation
LOCK(r)	Locks win at rank r.
UNLOCK(r)	Unlocks win at rank r.
FLUSH()	Ensures completion of pending communication requests by executing processes.
OFFSET(A,idx,r)	Computes offset of A[idx] at rank r in win.
GET(var,offset,r)	Reads value at offset in win at r to var.
PUT(var,offset,r)	Writes value in var to memory specified by offset in win at r.
GETA(A,offset,r)	Reads an array at offset in win at r to the local array A.
ALLREDUCE(v1,v2,op)	Reduces value in v2 in all processes to v1 in all processes using op.
RANK(v)	Computes rank of the process that owns vertex v.
LId_v(v)	Computes local-id of vertex v.
BARRIER	Synchronizes all processes.
ADD(array, elem)	Appends elem to array.
NDOFFSET(A, s)	OFFSET(A,LId_v(s),RANK(s)).

Fig. 3. Macros used in the text. These macros wrap MPI specific functionalities.

poses interesting challenges. Further, the exact distribution for the input data (among the different processes) has a big impact on the load-balancing, and consequently on the performance of the generated code. Supporting arbitrary types of distributions remains an important challenge.

3.2 Presence of Mixed Parallel and Serial Codes

A Green-Marl code may contain a sequence of serial and parallel code parts (for example, see Figure 2, where lines 13-17 constitute the parallel part). In contrast, in MPI, many instances of the same program are executed by multiple processes, and there is no explicit syntax to support serial code. In the translated code, the iterations of the input Green-Marl code's parallel loop should be distributed to be executed by all the processes, whereas the serial-part of the input should not be naively executed by all the processes (as it may lead to incorrect execution). Further, there may be dependencies between the serial and parallel parts which need to be preserved in the translation.

3.3 Distributed Graph Representation

To exploit the SPMD parallelism in MPI, different parts of the data are associated with different processes (that is, "distributed data"). Efficient representation of data is an important aspect of performant distributed programs, and it becomes more challenging in the context of storing complex data structures like graphs. We need an efficient storage representation for the underlying graph data structures to generate scalable programs.

3.4 Translating Green-Marl Constructs

As discussed in Section 2, for improved programmability, Green-Marl provides a rich set of constructs such as iterators (for example, InBFS, InDFS, CommonNbrs, and so on), collections (for example, Maps, Sets, and so on), reductions, and so on. Each of these constructs needs to be translated in a space- and time-efficient manner. Translation of Green-Marl constructs depends on two main factors (i) whether the data accesses made in these constructs are remote or local, and (ii) whether the constructs are used in a sequential or parallel region. If the constructs are used in a sequential region of the Green-Marl code, multiple MPI processes executing the corresponding generated code should compute the same values consistent with a single sequential execution.

However, if constructs are used in a parallel region, then multiple MPI processes can operate independently (unless any explicit synchronization is specified in the Green-Marl code).

3.5 Efficiency of the Translated Code

In distributed systems, especially in the context of compiling graph algorithms, it is crucial that the generated code runs efficiently and scales up to large inputs, over multiple nodes. Naively translating individual Green-Marl constructs may lead to inefficient codes. Hence it is important to identify and implement various optimizations over the generated code.

4 GREEN-MARL TO MPI RMA TRANSLATION

In this paper, we present some of the underlying facets of the design of *DisGCo* (Distributed Green-Marl Compiler). It is the first compiler that can compile any general Green-Marl program to C++ MPI code. In this section, we mainly focus on the translation of some of the important and challenging Green-Marl features/constructs. We start by discussing how the generated codes store the graphs over distributed systems in a space-efficient manner. For the ease of presentation, in this section, we assume that all parallel parts of the code is present (in-line) within a single function. In Section 6, we discuss how we deal with general programs.

4.1 Distributed Graph Representation

For a space-efficient representation of the input graph, whose vertices and edges are distributed across different processes, we extend the popular and performant CSR format (see Section 2.1) for distributed systems. We start by distributing the underlying graph data structures like `begin` and `node_idx`. For the simplicity of discussion, we use a simple block partitioning for distributing vertices among processes. In Section 4.7, we discuss how *DisGCo* admits any arbitrary vertex partitioning scheme.

We assign all the forward and reverse edges of a vertex to the same process to which the vertex belongs. This enables any process to access the properties of the forward/reverse edges of a local vertex quickly (as it is stored locally), which is a desirable condition for many propagation-based graph analytics like SSSP where the value of 'distance' is propagated through neighbors. For each process, let `num_local_nodes` hold the number of (local) nodes. During the execution of our generated code, for each vertex v assigned to an MPI process, we maintain a partial function $\texttt{LId}_v$: Vertices $\rightarrow \{0, 1, \cdots, \texttt{num\_local\_nodes} - 1\}$ to return the "local" id of the vertex. A similar map is maintained for the edges, as well. Note that unlike the actual ids of the vertices and edges, the local ids are not unique across processes.

Distributed CSR Representation. The four main arrays in CSR representation (`begin`, `node_idx`, `r_begin`, and `r_node_idx`) are distributed such that each process maintains information using similar four local arrays (`l_begin`, `l_node_idx`, `l_r_begin`, and `l_r_node_idx`) by storing information about only the vertices and edges the process owns. We call it the distributed CSR (DCSR, in short) representation.

Example: Figure 4 explains how the graph shown in Figure 1 is represented in DCSR format, assuming two processes. Here, the four vertices `a, b, c`, and `d` are distributed such that `a` and `b` belong to process with rank 0, and `c` and `d` belong to process with rank 1. The forward/reverse edges are owned by the process owning the source vertex of the edge.

4.2 Translating for-loops

We divide the discussion on how we translate the for-loops into two parts, depending on whether it is a parallel or serial for-loops.

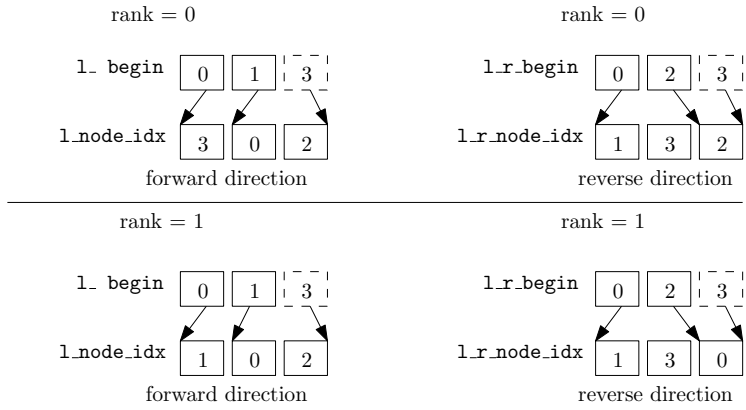


Fig. 4. An example representation in DCSR.

Parallel for-loops. As discussed in Section 2, we focus on the Green-Marl `Foreach`-loops that can iterate over ranges of the nodes and edges. As discussed above, the elements of these ranges may be distributed across multiple processes during the execution of the generated code. Thus, while translating a `Foreach`-loop to MPI, we emit code such that different processes handle their local data in parallel with each other. Like the current Green-Marl compiler, for nested `Foreach`-loops, we parallelize only the outer-most `Foreach`-loop.

Consider the SSSP code shown in Figure 2, where there is a parallel loop conditionally iterating over the nodes of the graph (line 13). In the generated MPI code, we emit a loop for each process to iterate over `num_local_nodes` (line 4 in Figure 5); the check in line 5 enforces the condition in the parallel loop.

Serial for-loops. In the generated MPI program, unlike the parallel loops, the complete serial-loop is executed by every process and not distributed across different processes.

Translating other node- and edge-ranges. As discussed in Section 2, the Green-Marl for-loops can range over many pre-defined ranges, all of which are handled by us using a similar scheme. We now illustrate the scheme using the translation of the Green-Marl expression `n.Nbrs`, which is used to obtain the neighbors of the node `n`.

Example: Consider the SSSP code (Figure 2, line 14), where there is an inner loop iterating over neighbors of the node `n`. If we look at the translated MPI code (Figure 5), we can see that `n` will always be a local node inside the outer for-loop at Line 4, and hence to iterate over the neighbors of `n` inside this outer for-loop, we can iterate over the locally stored graph data structures. *DisGCo* statically identifies that `n` is local and translates the inner loop to iterate from `l_begin[n]` to `l_begin[n+1]` (Line 6). The corresponding loop index, will be a local edge number (`s_i`) and the target of the edge (*a.k.a.* neighbor) is given by `l_node_idx[s_i]` (Line 7).

Note that unlike in the discussed SSSP code, if *DisGCo* cannot guarantee that the node `n` at the inner for-loop will always be a local node, then it will be treated as a remote-node and we will have to obtain the neighbor information from the process to which `n` may belong to, at runtime. In such a case, we will generate MPI code similar to the one shown in Figure 6, where `find_num_neighbors` computes the number of neighbors of the node `n` and `read_neighbors` returns the neighbors of `n`. At runtime, these functions will perform remote accesses to obtain neighbor information, if necessary (`n` is a remote node, that is).

```
void sssp(...)
{ ...
 while (!fin){
  for(node_t n=0; n<num_local_nodes; n++) {
   if (updated[n]) {
    for(edge_t s_i=l_begin[n];s_i<l_begin[n+1];s_i++){
     node_t s = l_node_idx[s_i]; // neighbor of n.
     e = s_i;
     int temp1 = dist[n] + len[e];
     LOCK(RANK(s));
     GET(temp2,NDOFFSET(dist_nxt,s),RANK(s));
     FLUSH();
     if(temp1 < temp2) {
      PUT(temp1,NDOFFSET(dist_nxt,s),RANK(s));
      PUT(true,NDOFFSET(updated_nxt,s),RANK(s));
     }
     UNLOCK(RANK(s));
     ...
  }}}
  ...
  bool temp = false;
  for(n=0; n<num_local_nodes; n++) { temp = temp||updated[n]; }
  ALLREDUCE(temp,fin,MPI_LOR);
} }
```

Fig. 5. Part of translated SSSP code in MPI RMA.

```
  for(node_t n=0; n<G.nodes(); n++) { ...
    int neighbors_size = find_num_neighbors(n);
    node_t *neighbors = read_neighbors(n);
    for(int i=0; i<neighbors_size; i++) {node_t s = neighbors[i]; ...}}
```

Fig. 6. Translating `Nbrs` with remote accesses.

4.3 Handling Property Accesses

In Green-Marl, programmers can associate the nodes (and edges) with different data, and such data are collectively referred to as the properties of the nodes (or edges). In the generated MPI code, for each property, we use arrays at the back-end to hold all the property-values across all the nodes/edges. Each process will store only information about the local node-properties (and/or edge-properties). When a process accesses a node- or an edge-property, it can be a local- or remote-access depending on where the corresponding node or edge resides.

If we can conservatively prove that the access is local (see the end of Section 4.3), then we emit code to access the local elements directly. Else the access is treated as remote access. We now explain how we translate remote accesses of the properties, using node properties as an example; the same can be extended to edge properties. The translation scheme is shown in Figure 7.

Remote read/write in a parallel loop. Consider the statements `n.x=...`, or `...=...n.x...`, where the remote property `n.x` is set or read, inside a parallel loop. To avoid data-races and ensure consistency of data, we emit code using MPI instructions `MPI_Put` and `MPI_Get`, inside locked regions, to write or read remote data. See rules 1 and 2 in Figure 7 for details.

Remote read/write outside a parallel loop. As per Green-Marl semantics, the execution of statements that are present outside a parallel-loop must respect the sequential semantics. In the generated MPI code, since many processes execute these serial parts of the code in parallel, to preserve semantics,

//1. *parallel remote write:*
`n.x = ...;`
	`emit("temp=...;//temp` *is local, set to RHS.*
	`r=RANK(n);`
	`LOCK(r); PUT(temp, NDOFFSET(x, n), r); UNLOCK(r); ");`

//2. *parallel remote read:*
`...=...n.y...;`
	`emit("r=RANK(n);`
	`LOCK(r); GET(temp, NDOFFSET(y, n), r); UNLOCK(r);`
	`...=...temp...; ");`

//3. *sequential remote write:*
`n.x=...;`
	`if(bNeededWr.contains(x)) {`
	`emit("BARRIER;"); clear(bNeededRd); clear(bNeededWr); }`
	`emit("r=RANK(n); if(my_rank ==r ){ x[LId`$_v$`(n)]=...; } ");`
	`bNeededRd.insert(x);`

//4. *sequential remote read:*
`...=...n.x...;`
	`if(bNeededRd.contains(x)) {`
	`emit("BARRIER;"); clear(bNeededRd); clear(bNeededWr);}`
	`emit("r=RANK(n);`
	`LOCK(r); GET(temp, NDOFFSET(x, n), r); UNLOCK(r);`
	`...=...temp...; ");`
	`bNeededWr.insert(x)`

//5. *sequential for:* `for(..){}`	`if(forhasaremoteaccess()) {` `emit("BARRIER") as the first statement of the for-loop}`

Fig. 7. Translating remote accesses.

we use barriers to synchronize, if there is a read-write dependency between remote accesses in these serial parts. Considering the overheads of barriers, we now describe a flow-sensitive scheme that tries to avoid emitting redundant barriers.

For checking the read-write dependency, we maintain two vectors at each statement: bNeededRd, the list of all properties whose reads need a preceding barrier, and bNeededWr, the list of all properties whose writes need a preceding barrier; both initialized to the empty set. When there is a read (or write) to a property x, we check if this read (or write) needs a preceding barrier, by checking whether x is present in bNeededRd (or bNeededWr) or not. If it is present, (i) we emit barrier before the read (or write) to ensure consistency, (ii) clear both bNeededRd and bNeededWr vectors, (iii) add x to bNeededWr (or bNeededRd) – to indicate that a following write (or read) needs a barrier, and finally, (iv) emit the code for read (or write). These updated vectors are used for the next statement. See rules 3 and 4 in Figure 7 for details.

Processing conditional and loop statements. On processing a conditional statement (if/if-else), we take the union of bNeededRd and bNeededWr at the join-point to derive the corresponding vectors for the following statement. In case of a sequential for-loop, if the for-loop has at least one remote access, to satisfy the inter-iteration read-write dependencies (if any), we conservatively emit a barrier as the first statement of the for-loop. See rule 5 in Figure 7 for details.

Before the start of a parallel-loop, if the vectors bNeededRd or bNeededWr have at least one entry, we emit a barrier and clear the two vectors. This ensures the 'completion' of the sequential region before the start of the parallel region.

Note that we do not have to insert barriers to preserve write-after-write dependencies inside a sequential region. When two writes occur to the same property indexed by same vertex/edge, they

will be executed by the same process (see rule 3 – in the generated code, the write is effectively non-remote). Hence the sequential order of writes to the property would be maintained without needing any barriers in between.

Proving accesses local. Each vertex in the distributed graph is mapped to a unique process. Independent of distribution, a parallel loop in Green-Marl iterating over the vertices of the graph will be translated to iterate over local vertices of a process. The vertex corresponding to the loop iteration (say i) in this case is always going to be a local vertex, and any access to the properties of the vertex i are going to be local unless the value of i is modified in the program. We keep track of the non modified loop index variables to identify the local accesses in *DisGCo*.

4.4 Handling Reduction Statements

Consider a reduction statement of the form '`x op= expr`'. We term the LHS (left-hand side) as the target of the reduction statement. If the target is a property, then it is translated using the rules discussed in Section 4.3, such that in each process, both the read and write (of `x`) happen atomically. If the target is a variable (say `x`) then the translation of the reduction statement depends on the declared scope of `x`. (i) If `x` is declared in a serial part of the input code, then we emit code involving MPI reduction functions. Here, each process will compute a reduced local value, in parallel, and these values are reduced to a global consistent value by the underlying MPI functions. (ii) If `x` is declared in a parallel region of the input code (the operation is independent of the other processes), then the reduction can be replaced with the corresponding simple arithmetic or logical operations.

Example: We describe our translation of the two reduction statements in Figure 2 (Lines 16 and 22). (i) Min reduction at Line 16: Here, the target of the reduction is a property, and hence we use atomic updates to realize the reduction; see the generated MPI code (Figure 5), Lines 9-17. Note that `dist_next[s]` is updated to the minimum value atomically by taking locks. (ii) Exists reduction: Here, the target variable `fin` is declared in the serial part, and we use MPI reduction (with `MPI_LOR` as the operator) for the translation. We emit code so that all processes first locally compute the logical-OR of the locally stored values of the node-properties (given by `updated[n]`), before invoking the reduction. In Figure 5, lines 21-23 show the generated code.

4.5 Map Implementation

`Map` is a dynamically growing data structure in Green-Marl, to store/retrieve key-value pairs. We have extended the distributed linked-list implementation specified in the MPI report [2] to implement a dynamic concurrent `Map`.

We use a hash function (many-to-one) to map each key to the beginning of a linked-list, each of which is headed by a "dummy" node. We maintain a table of these dummy nodes, indexed by the hash values of the existing keys, at the master process (rank 0), and this table is replicated onto each process. Each linked list is a distributed singly directed linked list (headed by a dummy node), where the "next" field of each node in the linked list will hold the rank and address of the next node in the linked list. When a process tries to insert a key-value pair (k, v) into the `Map`, we first find the linked-list (say L) corresponding to k. If k is already present in a node n, in L, then we assign v to $n.val$; else we locally allocate a new node (for (k, v)) and insert it as the last node of L. The linked list is updated using `compare_and_swap` operations to provide concurrency.

For deletion, we follow a flag based deletion approach [36], where the node to be deleted will be marked as deleted (by setting a flag), The memory for these nodes is freed, when the table is deallocated.

To support Green-Marl `Map` related functions like `getMinKey`, `getMaxKey`, and so on, we maintain additional information in the master-process. We update these information on each insertion. On

a deletion, if any of this saved information is to be changed, then we mark that information to be invalid. The subsequent request to obtain that information would result in the traversal of the complete linked list, and updation of all the invalidated information. This scheme leads to fast (cached) accesses for each subsequent calls to `GetMinKey/GetMaxKey`, and so on, after the first computation. This leads to minor overheads during insertion and deletion, but improves the average performance of functions like `GetMinKey/GetMinKey` and so on. Without this optimization, computing the minimum key or maximum key from Map would require complete traversal of the map which may be a huge overhead when the distributed map grows large in size. In such scenarios, our generated code answers multiple queries to these functions at the cost of a single remote read.

4.6 Iterators

Green-Marl supports different types of iterators to conveniently traverse the underlying graphs. In this section, we highlight two of the iterators with interesting challenges for translation.

BFS Iterator. As discussed in Section 2, the `inBFS` constructor of Green-Marl is used to iterate over all the reachable nodes (may be distributed across multiple processes) of a source node in BFS order. The construct admits codes to be executed in the forward and reverse order. Consequently, with each `inBFS` construct, we use two functions `visit_fw()` and `visit_rv()` to hold the respective translated codes. Different instances of these functions are expected to be independent of each other. During the traversal of the graph (discussed below), these functions are called from the appropriate program points. We now discuss how forward and reverse traversals are handled by *DisGCo*. We first describe the case when `inBFS` is not called inside a parallel region in Green-Marl.

Forward traversal. In the generated MPI code, at each level of BFS, nodes at that level are processed by the associated (owner) processes. To realize this behavior, we maintain a per-process vector called the `local_vector`, which stores the local nodes owned by that process, in BFS order. Initially, only the source node is inserted into the `local_vector` of the process that owns the source node. With each process, we maintain a variable called `current_level` to store the current level in which the visit is happening. For each level, each process p also remembers the index of the last node in `local_vector` in that level, to identify the set of nodes for p in that level. At each level, all the processes will go over their `local_vectors` and visit each node in their `local_vectors` until all the nodes in `current_level` have been processed. When a node is first visited, each of its neighbors is added to the `local_vector` of the corresponding owning process (this may involve remote updates); after that, the function `visit_fw()` is invoked. Once, all the processes find that all nodes ($\in$ `local_vector`) in the `current_level` have been visited, they increment the `current_level` by 1. All processes synchronize at this point before proceeding to the next level.

Reverse traversal. Each process, visits the nodes stored in its `local_vector`, in reverse order, level by level (in decreasing order). At each level, it traverses the nodes in the reverse BFS order and invokes `visit_rv()` on each of those nodes; at the end of the level, all processes synchronize with each other.

When `inBFS` is called in a parallel region, *DisGCo* implements a serial version of the explained logic, such that a process executing the `inBFS` code maintains a vector holding all the vertices that need to be visited by the process in the next level. Here, the process may have to handle non-local vertices as well, and consequently, the neighbor information of such vertices has to be obtained remotely.

CommonNbrs Iterator. An interesting challenge in translating the common neighbor iterator (see Section 2) of Green-Marl, which iterates over the common neighbors of two nodes `u` and `v`, is that the process p_0 that executes the iterator, the processes p_u and p_v that own u and v, respectively, could all be different processes.

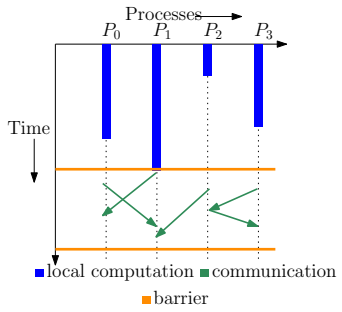


Fig. 8. Communication aggregation illustration.

As part of the translation, we emit code to (i) get the neighbors of u and v by doing a remote read (using GET), (ii) populate the common neighbors in a vector, and (iii) iterate over the vector. The following code shows the corresponding translation where the function `get_neighbors` is used to obtain the neighbors of the input node, and `intersect` returns a vector containing the common neighbors.

```
1 //Green-Marl code: 'for(s in u.CommonNbrs(v)){S1}'
2 int *u_nbrs=get_neighbors(u), *v_nbrs=get_neighbors(v);
3 vector<int> *common_nbrs = intersect(u_nbrs,v_nbrs);
4 for(auto& s : common_nbrs) {S1}
```

4.7 Graph Partitioning

The performance of *DisGCo* generated programs depends a lot on *how well the graph is load-balanced among the executing processes*. For better load balancing, the input graph has to be partitioned effectively among processes, and efficient partitioning schemes are explored by many prior works in the literature [5, 6, 8, 9, 41, 53, 54]. There are also standalone tools that partition the graphs for later use [29, 50]. Understanding the importance of the problem of partitioning, the challenges in identifying the best partition, and the orthogonal nature of the problem, instead of attempting a new partitioning scheme, *DisGCo* runtime provides a feature for programmers to specify any arbitrary vertex distribution. The programmer can specify (in a file) to which process each vertex belongs to, and *DisGCo* uses it as input for graph partitioning. If not specified, *DisGCo* uses a simple block distribution for partitioning graphs.

5 OPTIMIZATIONS

In this section, we discuss two optimizations that we implemented in *DisGCo* to improve the performance of code generated using the translation mechanism discussed in Section 4.

5.1 Communication Aggregation

Inspired by the conditions given by Hong.et.al. [28], we optimize Green-Marl programs with the following commonly occurring pattern: (i) `Foreach` loops iterate over only the edges or vertices of the program (ii) nesting of `Foreach` loops is allowed, but the maximum nesting depth is restricted to two; that is, at most doubly nested `Foreach` loops are present. Further, if nested, the outer `Foreach` loop iterates only over the vertices of the graph and the inner `Foreach` loop (if present) iterates only over the neighbors of each vertex. (iii) in terms of operations on properties, the `Foreach` loops can only have property writes or a reduction statement reducing a remote property.

The optimization has two passes. In the first pass, the function `check_for_comm_aggr` checks whether the `foreach` loops in Green-Marl can be optimized using communication aggregation or not. In the second pass, if the loop can be optimized, we collect relevant information about the loop like: (i) if there is a reduction present in the loop and if so its type, (ii) information about the local values written to the property and the local values used in the reduction, (iii) information about the remote property writes. In the second pass, we use this information to generate optimized code in BSP [11] style; example BSP style interaction shown in Figure 8. The function `emit_code` generates two for-loops, for each loop `f`, if `check_comm_aggr(f)` is `true`. In the first loop, information about remote writes is aggregated into one or more arrays, and in the second loop, actual write is performed. In the optimized code, unlike the code generated by the naive translation scheme, (i) all the processes synchronize after performing local computation and aggregating the messages for each destination process; (ii) after the synchronization, the processes read the aggregate messages for which they are the destinations/recipients and synchronize again. Here, message aggregation can greatly optimize the number of remote communications.

Figure 9 shows how the doubly nested loop (lines 13-17) in Figure 2 is translated to BSP model. It can be seen that the loop headers (lines 5 and 7) in Figure 9 are similar to that in Figure 5 (lines 4 and 6). To handle the writes to a remote property, we store the value to update as well as the node/edge id to which the write happens in two different arrays. After aggregating all writes into arrays, the processes communicate the data in arrays to the destination processes. The destination processes then perform the writes locally. In the case of SSSP, each process takes the value to update, and finds the minimum distance locally. Note that here, the number of remote communications reduces to $O(num\_process^2)$, from $O(|E|)$ of the naive translation scheme, where E = set of edges.

5.2 Common Sub-expression Elimination

Our translation scheme, discussed in Section 4, generates expression to compute rank and local_id of nodes/edges at many program points (may lead to redundant computation). We avoid such re-computations by substituting pre-computed variables for rank and local_id at each re-computation.

6 DISCUSSION

In this section, we discuss some of the salient points about our proposed *DisGCo* compiler.

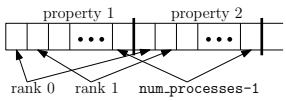
Other Green-Marl constructs handled. (i) The `inDFS` iterator that follows a similar syntax to that of `inBFS`, is used to iterate over nodes of the graph in DFS order. Our DFS implementation (like the existing implementation) is inherently sequential, and we use a stack-based implementation for it. (ii) For implementing the collections types `sequence` and `order` of Green-Marl (see Section 2), we use distributed concurrent linked lists. Similarly for implementing the `set` collection type, we use distributed bitmaps. (iii) For utility functions like `pickRandom`, `uniform`, and so on, we have developed the corresponding low-level calls to be used as translations.

Implementation Details for remote and property accesses. *Remote accesses.* As discussed in Section 2, any process can remotely access data in a target memory location which is attached to the shared `win` object. This access can be done by providing the target's rank, window handle (`win`), and the offset (= address of the memory location).

Handling property accesses. Corresponding to any property x, in each process, we attach a local array, (say A_x) to `win`. We emit code so that, once A_x is attached to `win`, the process which attaches the array to `win`, broadcasts the base address of A_x to all other processes. We use an array called `base_address` to store the starting location of the attached arrays. Note that the address of any element in A_x can be calculated by adding appropriate shifts to the base address (to compute the offset). In essence, at each process, corresponding to each property, `base_address` will have

```
void sssp(...)
{ ...
 int *ind = 0;
 while (!fin){
  for(node_t n=0; n<num_local_nodes; n++) {
   ...
   for(edge_t e=l_begin[n];e<l_begin[n+1];e++){
    node_t s = l_node_idx[e];
    int temp1 = dist[n] + len[e];
    int r1 = GET_RANK(s);
    ADD(nd_to_update[r1], LId_v(s));
    ADD(val_to_update[r1], temp1);
  } }
  BARRIER; ...
  for(i=0; i<num_processes; i++) {
   GETA(n_t_u,OFFSET(nd_to_update,my_rank,i),i);
   GETA(v_t_u,OFFSET(val_to_update,my_rank,i),i);
   for(int i=0; i<n_t_u_size; i++) {
    int n = n_t_u[i];
    int v = v_t_u[i];
    if(v < dist_nxt[n]) {
      dist_nxt[n] = v;
      updated_nxt[n] = true;
  } } }
  BARRIER; ...
} }
```

Fig. 9. Part of optimized SSSP code in MPI RMA.



(a) base_address array.

```
void win_attach(int *array, int array_size) {
// attaches the whole array to win.
  MPI_Win_attach(win,array,array_size);
  int index = property_index*num_processes;
  base_address[index + my_rank] = array;
  property_index++;
  for(int i=0; i<num_processes; i++) {
    // broadcasts base address of the array
    // to all processes with source as i.
    MPI_Bcast(&base_address[index + i],···,i,···);
  } }
```

(b) Code for attaching an array to win.

Fig. 10. *DisGCo* Implementation details.

num_processes entries, as shown in Figure 10a. The code for attaching an array to win is shown in Figure 10b; the variable property_index stores the count of the number of properties currently attached to the base_address.

Programs with Multiple Functions. Though the translation scheme presented in Section 4 assumes the complete parallel part of the code is present within a single function, it can be easily extended to a general program with multiple functions. As discussed in Section 4, the translation of different constructs differs based on whether the construct appears within a parallel region or not. We extend this idea to codes with multiple functions, by marking each function with a flag (mayRunInParallel) to indicate if it may be invoked inside a parallel region or not; this is done

in a pre-pass. During the translation of a function, if its `mayRunInParallel` flag is set, then the function body is translated by assuming it to be present inside a parallel region.

Sources of Efficiency of the *DisGCo* generated code. We now highlight some of the important design decisions that lead to performant MPI code. (i) *Using D-CSR format.* Many graph algorithms work by propagating information through forward or reverse edges. It is desirable for such algorithms to access incoming/outgoing edges (distributed over nodes) in constant time. In D-CSR format, we ensure this by storing forward and reverse edges with the same process. (ii) *Designing distributed versions of the Green-Marl collections.* We have designed the distributed counterparts of the Green-Marl collections like `Map` and `NodeSequence`, and avoided centralized storage for them. Moreover, these dynamically growing data structures are implemented as concurrent data structures, to allow concurrent parallel access to them there by improving efficiency. (iii) *Identifying local accesses.* We keep track of locally accessed vertices and edges, and we disallow remote access on them if a vertex/edge is identified as local. (iv) *Data distributions.* We provide a mechanism to admit arbitrary data distributions that can lead to efficient program execution. (v) *Identifying and implementing different optimizations.* We have identified a few potential optimizations and implemented them in *DisGCo* to improve the efficiency of the generated code. Considering the efficiency of the BSP model, we generate programs in a BSP like model using Communication Aggregation optimization, if the input programs satisfy certain conditions. (vi) Importantly *DisGCo* generates code that can take advantage of the efficient RMA based remote communication. These schemes and the Communication Aggregation optimization are general and may be extended to any graph DSL.

Comparison with shared memory backend of Green-Marl. Considering the overheads associated with MPI processes and distributed communication, it is natural that compared to a parallel MPI code, the corresponding OpenMP code would run much faster, when both the codes are run on a shared memory system and provided that the input data fits in the memory of the shared memory system. This was also visible in our brief comparison, where across the five benchmarks discussed in Section 7, for the smallest input (Amazon), the handwritten OpenMP programs showed a Geo-mean speedup of 10.68× over the generated MPI programs, when ran on 16 cores.

7 EVALUATION

We have implemented *DisGCo* as an extension to the existing Green-Marl compiler, which otherwise translates Green-Marl programs to OpenMP. Our compiler can translate all the programs in the Green-Marl repository. We did our evaluations on a 16 node IBM cluster, where each node has 2 Intel E5-670 2.6GHz processors, 8 cores/processor, and 64GB RAM. These nodes are connected using an RDMA enabled FDR10 Infiniband interconnect, with low-powered Mezzanine adapters [3]. The codes generated by *DisGCo* are compiled using MPICH [4] and are evaluated on five different graph benchmark kernels; Single Source Shortest Path using BellmanFord's algorithm (SSSP), Breadth First Search (BFS), Connected Components (CC), Average Teen Count (ATC) and Pagerank (PR). As an additional check, we have verified that the output of each *DisGCo* compiled code matches that of the existing Green-Marl compiler.

We divide our evaluation into six parts. (i) We compare the various static characteristics of code generated by *DisGCo* against the code generated by `DH-Falcon`. (ii) We discuss the scalability of the *DisGCo* generated codes over five input graphs; a synthetic graph[1] *Random30*, and four real-world graphs - *Amazon* (TWEB), *LiveJournal*, *Orkut* and *Youtube*; see Figure 11 for some details of the input graphs. (iii) We present a performance comparison against the state-of-the-art `DH-Falcon`, the only graph DSL compiler that generates MPI codes. (iv) We evaluate the *DisGCo* generated

[1]We use the random graph generator provided by the Green-Marl distribution that uses a uniform distribution.

Graph	#V	#E	*DisGCo* running times for 1×1				
			SSSP	BFS	CC	ATC	PR
Random30	30M	300M	166.20	34.77	31.16	23.34	166.91
Amazon (TWEB)	0.4M	3M	2.18	0.41	6.6	0.19	3.02
LiveJournal	4M	68M	39.57	5.62	71.20	3.60	42.29
Orkut	3M	117M	40.05	6.86	50.19	6.10	96.89
Youtube	3M	9M	2.57	1.21	11.49	0.52	4.99

Fig. 11. Input details and *DisGCo* runtimes for running times for a single node with single core (1x1).

Graph	SSSP			BFS			CC			PR		ATC	
	D	F	M	D	F	M	D	F	M	D	M	D	M
LOC	171	284	140	168	194	141	163	343	134	173	131	157	126
#Sends/Receives	0	10	0	0	6	0	0	6	0	0	0	0	0
#Gets	3	0	3	3	0	3	3	0	3	3	3	3	2
#Barriers	4	3	1	4	3	1	4	3	1	6	1	4	1
#Reductions	1	1	1	1	1	1	1	1	1	1	1	1	1
#locks/unlocks	3	0	3	3	0	3	3	0	3	3	3	3	3

Fig. 12. Static characteristics of the code generated by *DisGCo*, `DH-Falcon`, and the manually tuned codes (D=*DisGCo* generated, F=`DH-Falcon` generated, M=Manually tuned)

codes over different distributions to study the impact thereof. (v) We discuss the impact of the proposed optimizations on the generated codes. (vi) We present a comparison of *DisGCo* generated codes against the manually tuned versions thereof. Except for (iii), we use a blocked-distribution for partitioning the input.

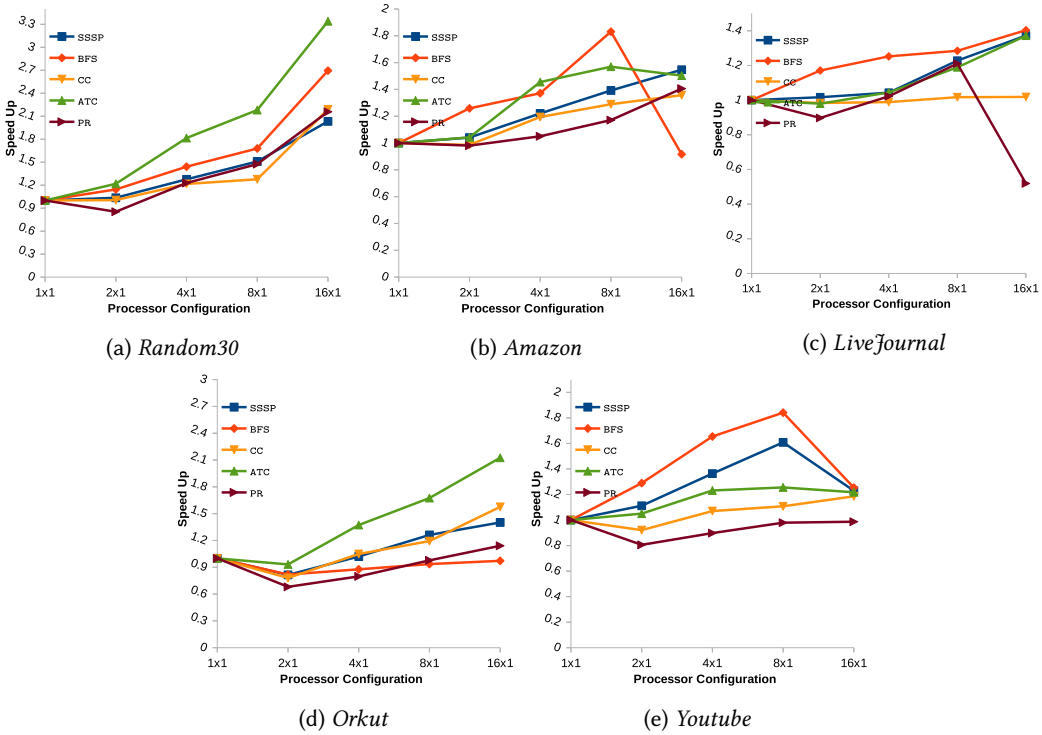
7.1 Static Characteristics

Figure 12 (columns labeled "D" and "F") shows some static characteristics of the *DisGCo*, and `DH-Falcon` generated programs. Comparing the static characteristics of *DisGCo* and `DH-Falcon`, we can see that *DisGCo* generated codes are better than `DH-Falcon` generated codes in terms of the number of lines of code and remote communications used. However, we see that in terms of the number of barriers, the *DisGCo* generated codes show more barriers than the `DH-Falcon` generated code. This is because in the `DH-Falcon` generated codes, the blocking-receives produce implicit synchronizations (not counted in Figure 12).

7.2 Scalability

We show the scalability results of *DisGCo* in Figures 13 and 14, for varying hardware configurations. Figure 13 shows the scalability result when evaluated in a purely distributed (DS) set-up by varying number of nodes (from 1 to 16), whereas Figure 14 shows the scalability result when evaluated on a single node multi-core (MC) system by varying the number of processes (from 1 to 16) for all five graphs. For reference, the execution times for the configurations 1×1 for different input graphs are shown in Figure 11.

Figures 13 and 14 show that the *DisGCo* generated programs scale well on both DS and MC systems. Naturally, the scaling on the MC system is higher than on the DS system, owing to the relatively lower communication overheads in the MC systems. We see that in some of the plots, the scaling reduces after a certain point - this is expected as for any graph application, for a given input, there is an optimum configuration where we get the maximum performance and after which,



(a) *Random30* (b) *Amazon* (c) *LiveJournal*

(d) *Orkut* (e) *Youtube*

Fig. 13. Strong Scalability of *DisGCo* on multi node systems. Speedup on a $N\times1$ system = (Exec time on $N\times1$ system)÷ (Exec time on 1×1 system). A configuration $N\times C$ is a system with N nodes, with C cores per node.

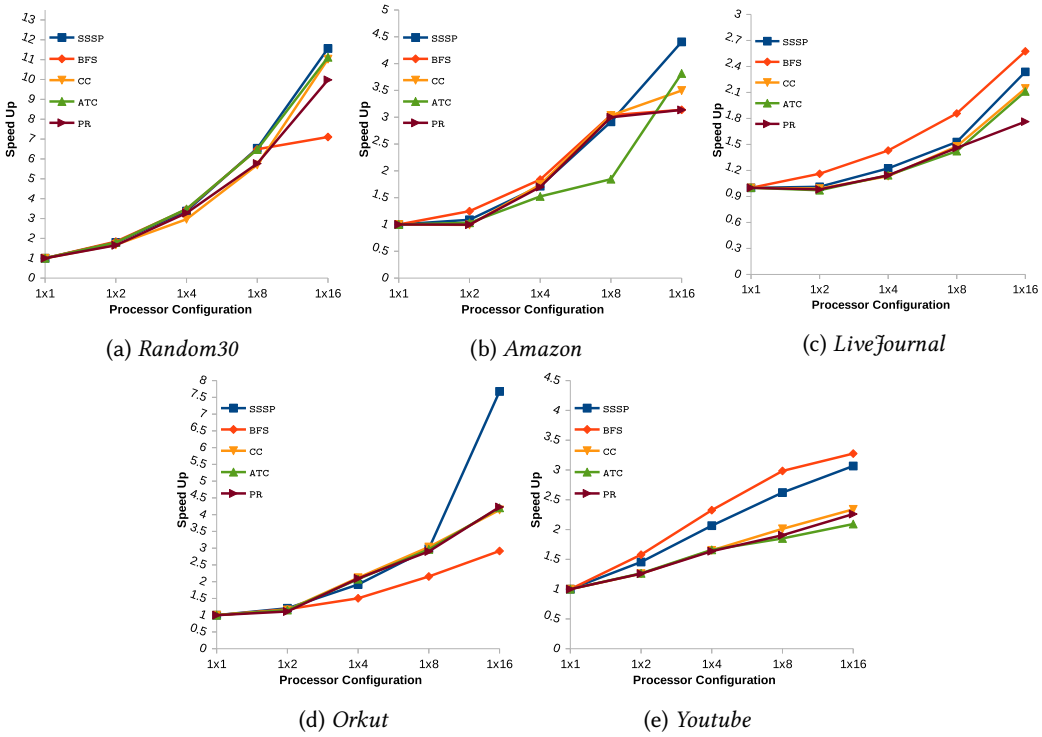
the performance starts degrading as the increase in the communication overheads overshadow the gains due to parallelism. For instance, for the *Amazon* graph (Figure 13), for BFS, 8×1 is the optimum configuration and the performance degrades for 16×1. Similarly, for the *LiveJournal* input , for the PR kernel, the optimum performance was observed for the 8×1 configuration and for the *Youtube* graph, for SSSP and BFS, the optimum performance was observed for the 8×1 configuration.

Summary: We see that *DisGCo* can produce scalable MPI codes. The actual amount of scaling depends on the size of the input, amount of local computation, and resulting trade-offs between the distribution of the computation and the increase in communication.

7.3 Comparison with `DH-Falcon`

The `DH-Falcon` compiler converts programs written in `DH-Falcon` (a domain specific language) to MPI programs (using two-sided communication) in the efficient BSP model. We got the `DH-Falcon` codes for SSSP and BFS from the authors of `DH-Falcon` and we ourselves wrote the Falcon codes for CC, ATC, and PR. We found the `DH-Falcon` compiler could only compile SSSP, BFS, and CC, but crashed (threw a segmentation-fault) on ATC and PR. We tried talking to the developers of `DH-Falcon`, but they could not fix the issues. Further, we found that despite our best efforts, we could not run these three kernels on real-world inputs - the programs crashed. Hence, in this evaluation, we only discuss the comparison of these three kernels (SSSP, BFS, CC) using the *Random30* input.

Figure 15 shows the speedups of the *DisGCo* generated codes over the `DH-Falcon` generated codes for *Random30*, for varying hardware configurations. The `DH-Falcon` generated code crashed



(a) *Random30* (b) *Amazon* (c) *LiveJournal*

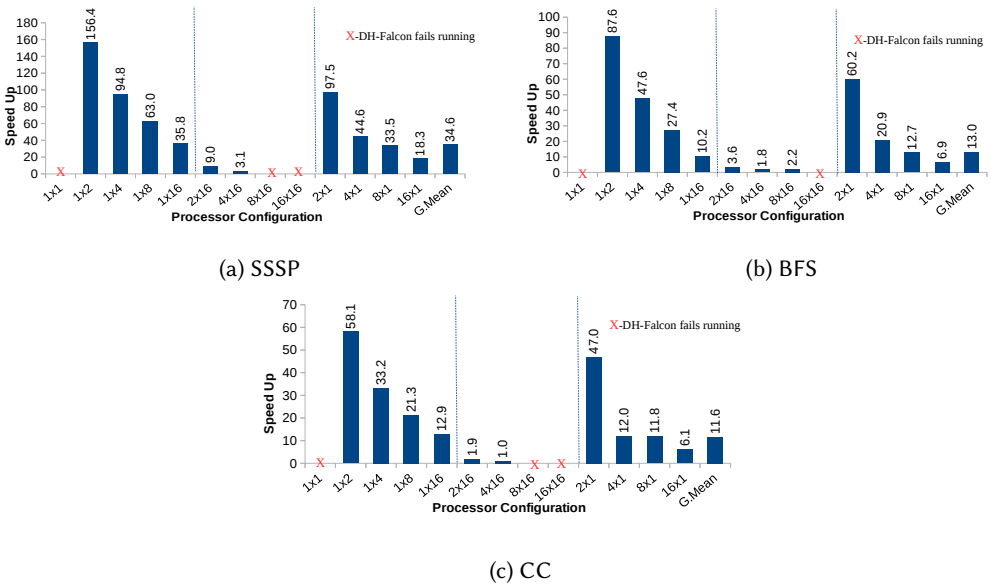
(d) *Orkut* (e) *Youtube*

Fig. 14. Strong Scalability of *DisGCo* on multi-core systems. Speedup on a 1×N system = (Exec time on 1×N system)÷ (Exec time on 1×1 system). A configuration N×C is a system with N nodes, with C cores per node.

for a couple of configurations, and we skip the corresponding speedup entries (shown with an X). The graphs show speedups on three types of configurations: single-node-multi-cores, multi-node-max-cores, and multi-node-single-core. Across the three types of configurations, we see that the overall, compared to `DH-Falcon`, *DisGCo* leads to a geomean speedup of 17.32×, across the three benchmarks. These speedups are chiefly due to our design choices for efficiency (see Section 6) including the use of one-side communication (RMA), in contrast to the two-side communication employed by the `DH-Falcon`.

The impact of the choice of RMA for communication can be seen from the speedups obtained by the *DisGCo* codes for smaller hardware configurations, where the `DH-Falcon` generated results incur large performance overheads due to the high cost of communication delays. We can see that as we increase the amount of parallelism, more communications happen in parallel, which in turn, reduces the overheads of `DH-Falcon` generated codes, and hence the speedups reduce. However, it can be seen that even at higher configurations, the performance of the *DisGCo* generated codes is still significantly higher than that of the `DH-Falcon` generated codes (attesting to the efficiency of our translation scheme). Further, note that for the three studied kernels, the `DH-Falcon` generated codes do not scale beyond 4×16 (either takes more time than lower configurations or crashes).

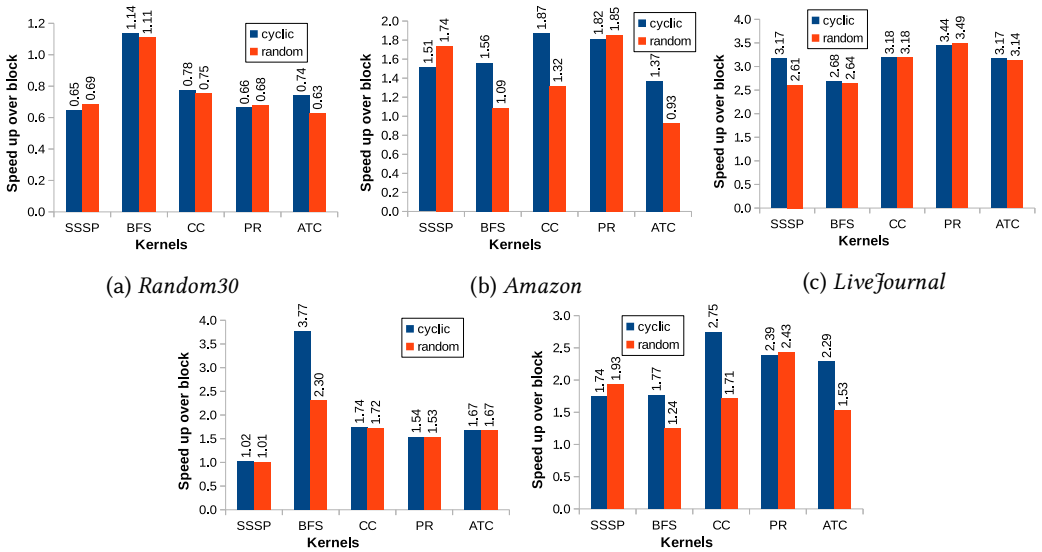
Summary: From the results, it is clear that *DisGCo* compiler generates more efficient MPI codes than those generated by the state-of-the-art `DH-Falcon` compiler.



(a) SSSP (b) BFS

(c) CC

Fig. 15. Speedup of *DisGCo* over `DH-Falcon`, for varying hardware configurations. Input graph: *Random30*.



(a) *Random30* (b) *Amazon* (c) *LiveJournal*

(d) *Orkut* (e) *Youtube*

Fig. 16. Speedup of *DisGCo* programs using cyclic and random vertex distributions over block distribution.

7.4 Performance of *DisGCo* Programs for Varying Distributions

Considering the importance of vertex-distribution, and the challenges in designing the most optimal distribution schemes, *DisGCo* provides a way for the programmer to specify any arbitrary vertex-distribution. To see the impact of distribution on the performance, we have tested the programs using three distributions: blocked (default), cyclic, and random. Figure 16 shows the performance of *DisGCo* generated programs executed using cyclic and random distribution of vertices as a speedup over the *DisGCo* programs executed using block distribution of vertices; we use the largest distributed system (DS) 16×1 for this evaluation.

Summary: We can see that for best performance the distribution should be chosen depending on both the specific application under consideration, and the input graph.

7.5 Effectiveness of the Optimizations

To demonstrate the importance of our proposed optimizations (see Section 5), we evaluated their impact on the generated codes. Figure 17 shows the performance improvement (with respect to the unoptimized code) we obtained due to Communication Aggregation Optimization, when run on the 16×1 configuration. The reduction in the number of communication operations due to Communication Aggregation Optimization resulted in significant speedup. Similarly, Figure 18 shows the speedups obtained due to Common Subexpression Elimination (baseline = unoptimized code), when run on the 16× configuration. The optimization was applicable only to SSSP, BFS and CC and their relative speedups are reported. We can clearly see that since the Communication Aggregation Optimization reduces the remote communicate its impact is very high. In contrast, since the CSE optimization reduces the number of computation steps and hence the gains are relatively smaller (significant nevertheless). Overall, we find that the proposed optimizations are very effective in substantially decreasing the associated overheads.

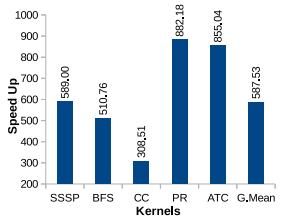
7.6 Speedup of manually tuned programs

Our current code generation scheme emits barriers at the end of all parallel for-loops, which can be eliminated if there are no dependencies to be preserved with the downstream code. During the manual tuning, we elided such redundant operations. Similarly, we performed another minor tuning wherein we eliminated the instructions that attach properties to windows, at the beginning of each function, if those properties are unused. We also identified some remote reads that are guaranteed to return the same value (depending on the program logic) and combined them into a single remote read. The resulting differences in terms of the static characteristics are shown in the columns labelled "M" in Figure 12. Naturally, the manually tuned codes have fewer barriers, and reads (Get operations) than the *DisGCo* generated codes. Figure 19 shows the speedup of manually tuned codes over the *DisGCo* generated codes. It can be seen that compared to the manually tuned codes the *DisGCo* generated code run slower (average 28%). We leave it as a future work to cover this gap automatically.

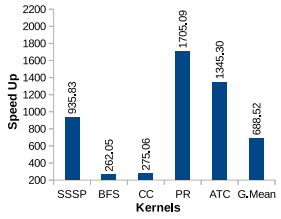
Overall summary of evaluation: The *DisGCo* compiler can compile arbitrary Green-Marl syntax, and can generate MPI RMA codes that are scalable and perform significantly better than those generated by the state-of-the-art `DH-Falcon` compiler. Further, the proposed optimizations are important to realize the performance gains.

8 RELATED WORK

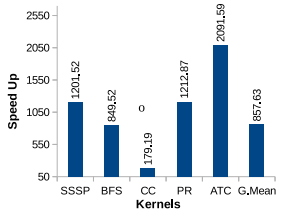
Graph DSLs for distributed systems. `DH-Falcon` [13] is a state-of-the-art Domain Specific Language that extends the C language to encode graph algorithms. Its compiler translates the input programs to MPI programs in BSP model. Hong et.al [28] translate a restricted subset of Green-Marl



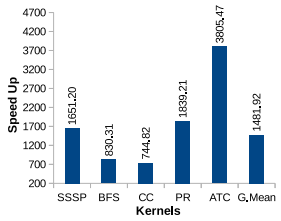
(a) *Random30*



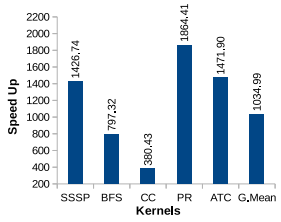
(b) *Amazon*



(c) *LiveJournal*

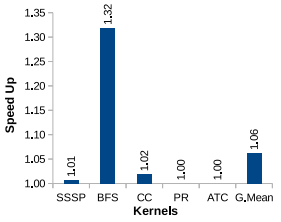


(d) *Orkut*

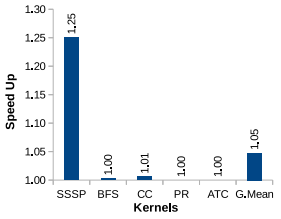


(e) *Youtube*

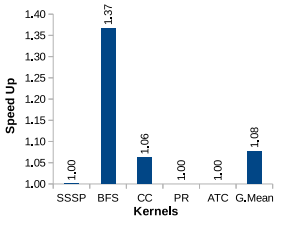
Fig. 17. Speedup due to Communication Aggregation Opt.



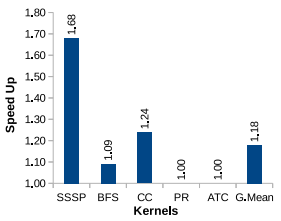
(a) *Random30*



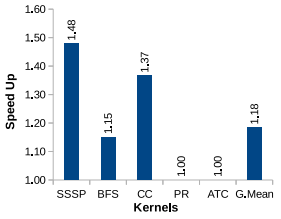
(b) *Amazon*



(c) *LiveJournal*

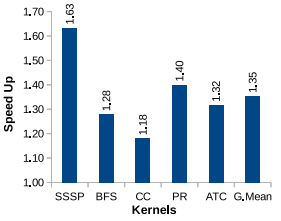


(d) *Orkut*

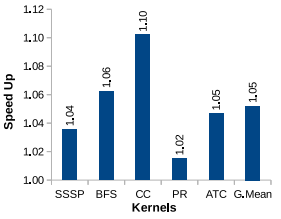


(e) *Youtube*

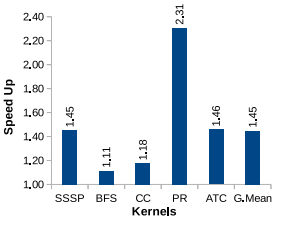
Fig. 18. Speedup due to Common Subexpression Elim.



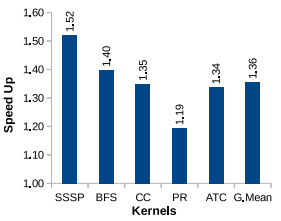
(a) *Random30*



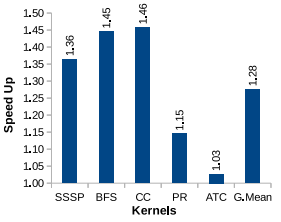
(b) *Amazon*



(c) *LiveJournal*



(d) *Orkut*



(e) *Youtube*

Fig. 19. Speedup of Manually Tuned Programs.

to generate distributed Pregel code. In contrast, the *DisGCo* compiler has no such restriction, and leads to more performant codes than those generated by `DH-Falcon`.

BSP based Distributed Graph Processing Systems. Many works [12, 17, 23, 25, 30, 37, 45] in the literature exploit the efficient BSP model to bring out abstractions for programming, as well as translation schemes. However, writing programs by focussing on the BSP model, can be both restrictive and also arguably non-intuitive for some programmers. In contrast, *DisGCo* translates programs written in Green-Marl (a high-level graph DSL) to MPI RMA code and has optimizations built in to identify opportunities to translate Green-Marl code to codes in the efficient BSP model.
Distributed Graph Processing systems. There are many frameworks [17, 20, 34, 38, 43, 44, 46, 47, 58] that help encode different types of graph algorithms for distributed systems. For example, Distributed GraphLab [34] offers suitable abstractions for challenging parallel machine learning algorithms. PowerGraph[20] focusses on the challenges of power-law graphs where the programmer needs to provide the implementations for Gather, Apply, and Scatter functions to code any graph algorithm. Plimpton et.al. [43] proposed a library called MR-MPI, which helps to write MPI programs for graphs in Map-Reduce format. SnuCL [31] extends OpenCL to admit programs that can be run on heterogeneous systems. The SnuCL runtime has a dedicated command scheduler thread to schedule and to ensure completion of execution of kernels on multiple nodes. The work-items for each node are transferred using MPI point-to-point communication routines. Similarly, Gluon [17] is a graph analytic substrate proposed for distributed heterogeneous systems, which can be used to enable distributed memory execution for many shared memory applications, using an interface code. To interface with Gluon, a blocking synchronization call is inserted between successive parallel rounds, which therefore restricts the input programs only to be in BSP model. Galois programming model [39] is an efficient model for encoding graph analytics for shared memory systems. Besides Gluon, there are other prior works that propose various compilers for converting BSP style programs written in Galois model to Distributed systems [19] and on to GPUS [42]. Many shared memory frameworks [35, 40, 42, 49, 57, 58] have also been proposed for graph analytics.

In contrast, our proposed *DisGCo* compiler transforms a high-level imperative DSL code (not just restricted to BSP style code) into low-level MPI code that can run on distributed systems. We believe that many of the optimizations discussed in the above discussed prior works (for example, direction optimization, efficient graph partitioning, load balancing, and so on) can be extended to/implemented in *DisGCo*. We leave such extensions as future works.
SPMDization. There have been many prior works [7, 10, 16, 52] that translate fork-join style code to SPMD code. We use a similar approach and generate MPI SPMD code from Green-Marl.
Lock Optimization. Optimizing the overheads of synchronization is a hot area of research to improve the performance of parallel programs [14, 21, 22, 55, 56]. These works can be used to further speedup the the *DisGCo* generated programs in an orthogonal way.

9 CONCLUSION

In this paper, we present *DisGCo*, the first compiler that can translate any Green-Marl program to MPI RMA program. We show that *DisGCo* generated programs perform better than programs generated by the existing state-of-the-art `DH-Falcon` compiler. Unlike the existing Green-Marl compiler (targeting distributed systems) that admits only a subset of Green-Marl programs, *DisGCo* is general enough to handle all graph algorithms expressible in Green-Marl.

Future work. The proposed *DisGCo* compiler can be extended in various directions, such as (i) implementing all the library functions in the BSP model, (ii) reducing the overheads due to locks/unlocks in the RMA based codes, (iii) identifying optimal graph distributions, and (iv) extending *DisGCo* to generate hybrid MPI+OpenMP code, targeting heterogeneous systems.
Acknowledgements: This work is partially supported by SERB CRG grant (sanction number CRG/2018/002488) and NSM research grant (sanction number MeitY/R&D/HPC/2(1)/2014).

REFERENCES

[1] 2015. Green-Marl Language Spec. https://docs.oracle.com/cd/E56133_01/1.2.0/Green_Marl_Language_Specification.pdf. (2015).

[2] 2015. MPI3.1 documentation. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. (2015).

[3] 2016. Mezzanine Apapters. http://www.mellanox.com/related-docs/user_manuals. (2016).

[4] 2019. MPICH Home Page. http://www.mcs.anl.gov/mpi/mpich2. (2019).

[5] A. Abdolrashidi and L. Ramaswamy. 2016. Continual and Cost-Effective Partitioning of Dynamic Graphs for Optimizing Big Graph Processing Systems. In *IEEE International Congress on Big Data (BigData Congress)*. 18–25.

[6] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. 2013. Distributed Large-Scale Natural Graph Factorization. In *WWW*. 37–48.

[7] S. P. Amarasinghe and M. S. Lam. 1993. Communication Optimization and Code Generation for Distributed Memory Machines. In *PLDI*. 126–138.

[8] K. Andreev and H. Räcke. 2004. Balanced Graph Partitioning. In *SPAA*. 120–124.

[9] A. Bader and K. Madduri. 2008. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS*. 1–12.

[10] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. 2009. Efficient, Portable Implementation of Asynchronous Multi-place Programs. In *PPoPP*. 271–282.

[11] R. C. Calinescu. 2000. *The Bulk-Synchronous Parallel Model*. Springer London, London, 5–12. https://doi.org/10.1007/978-1-4471-0763-7_2

[12] A. Chan and F. Dehne. 2003. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 117–125.

[13] U. Cheramangalath, R. Nasre, and Y. N. Srikant. 2017. DH-Falcon: A Language for Large-Scale Graph Processing on Distributed Heterogeneous Systems. In *CLUSTER*. 439–450.

[14] S. Cherem, T. Chilimbi, and S. Gulwani. 2008. Inferring Locks for Atomic Sections. In *PLDI*. 304–315.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[16] R. Cytron, J. Lipkis, and E. Schonberg. 1990. A Compiler-assisted Approach to SPMD Execution. In *Supercomputing*. 398–406.

[17] R. Dathathri, G. Gill, L. Hoang, H. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[18] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. 2016. An Implementation and Evaluation of the MPI 3.0 One-sided Communication Interface. *Concurr. Comput. : Pract. Exper.* 28 (Dec 2016), 4385–4404. https://doi.org/10.1002/cpe.3758

[19] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali. 2018. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *EuroPar*. 249–264.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[21] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*.

[22] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of Locks in a Shared Data Base. In *VLDB*. 428–451.

[23] D. Gregor and A. Lumsdaine. 2005. Lifting Sequential Graph Algorithms for Distributed-memory Parallel Computation. In *OOPSLA*. 423–437.

[24] W. D. Gropp and R. Thakur. 2007. Revealing the Performance of MPI RMA Implementations. In *PVM/MPI*. 272–280.

[25] F. Hielscher and P. Gottschling. 2004. ParGraph. http://pargraph.sourceforge.net/. (2004).

[26] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* 2, Article 9 (Jun 2015), 26 pages. https://doi.org/10.1145/2780584

[27] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*. 349–362.

[28] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 208, 11 pages. https://doi.org/10.1145/2581122.2544162

[29] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20 (Dec 1998), 359–392.

[30] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys*. 169–182.
[31] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *ICS*. 341–352.
[32] M. Li, X. Lu, K. Hamidouche, J. Zhang, and D. K. Panda. 2016. Mizan-RMA: Accelerating Mizan Graph Processing Framework with MPI RMA. 42–51.
[33] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda. 2014. Scalable Graph500 design with MPI-3 RMA. In *IEEE International Conference on Cluster Computing (CLUSTER)*. 230–238.
[34] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. 2012. Proc. VLDB Endow. 5 (Apr 2012), 716–727.
[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. 2010. GraphLab: New Framework for Parallel Machine Learning. *CoRR* abs/1006.4990 (2010). arXiv:1006.4990 http://arxiv.org/abs/1006.4990
[36] T. Maier, P. Sanders, and R. Dementiev. 2016. Concurrent Hash Tables: Fast and General?(!). In *PPoPP*. 3:41–3:42.
[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*. 135–146.
[38] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *USENIX ATC*. 291–305.
[39] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *SOSP*. 456âĂŞ471.
[40] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *SOSP*. 456–471.
[41] J. Nishimura and J. Ugander. 2013. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *KDD*. 1106–1114.
[42] S. Pai and K. Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *OOPSLA*. 1–19.
[43] S. J. Plimpton and K. D. Devine. 2011. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Comput.* 37 (Sep 2011), 610–632. https://doi.org/10.1016/j.parco.2011.02.004
[44] L. Rauchwerger, F. Arzu, and K. Ouchi. 1998. Standard Templates Adaptive Parallel Library (STAPL). In *Languages, Compilers, and Run-Time Systems for Scalable Computers*. 402–409.
[45] S. Salihoglu and J. Widom. 2013. GPS: A Graph Processing System. In *SSDBM*. 22:1–22:12.
[46] J. Seo, J. Park, J. Shin, and M. S. Lam. 2013. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.* 6 (Sep 2013), 1906–1917.
[47] B. Shao, H. Wang, and Y. Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*. 505–516.
[48] G. Shashidhar and R. Nasre. 2017. LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs. In *LCPC*. 235–249.
[49] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*. 135–146.
[50] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. 2017. Partitioning Trillion-Edge Graphs in Minutes. In *IPDPS*. 646–655.
[51] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. L. Träff. 2009. Investigating High Performance RMA Interfaces for the MPI-3 Standard. In *ICPP*. 293–300.
[52] C. Tseng. 1995. Compiler Optimizations for Eliminating Barrier Synchronization. *SIGPLAN Not.* 30 (Aug 1995), 144–155.
[53] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *WSDM*. 333–342.
[54] R. Wang and K. Chiu. 2013. A stream partitioning approach to processing large scale distributed graph datasets. In *IEEE International Conference on Big Data*. 537–542.
[55] T. Yu and M. Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *ISSTA*. 389–400.
[56] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. 2007. Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections. In *PPoPP*. 146–147.
[57] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. 2018. GraphIt: A High-performance Graph DSL. *Proc. ACM Program. Lang.* 2, Article 121 (Oct 2018), 30 pages. https://doi.org/10.1145/3276491
[58] X. Zhu, W. Chen, W. Zheng, and X. Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*. 301–316.