Graphical Abstract

StarPlat: A Versatile DSL for Graph Analytics

Nibedita Behera, Ashwina Kumar, Ebenezer Rajadurai T, Sai Nitish, Rajesh Pandian M, Rupesh Nasre

Highlights

StarPlat: A Versatile DSL for Graph Analytics

Nibedita Behera, Ashwina Kumar, Ebenezer Rajadurai T, Sai Nitish, Rajesh Pandian M, Rupesh Nasre

- StarPlat, a DSL for graph analytics which allows users to write a high level algorithmic specification of their static graph processing, which captures the parallelism intentions but decouples the target architecture.
- An intermediate representation common across backends that captures all the essential information associated with constructs for their translation in the target architecture.
- A code-generation scheme to support translation of the intermediate representation into efficient codes for the multicore (using OpenMP), distributed (using OpenMPI), and many-core (using CUDA) backends.
- Performance analysis of the generated BC, PR, SSSP, and TC codes for different target hardware on a variety of popular graphs, and illustrating competitive performance against that of the hand-crafted frameworks.

StarPlat: A Versatile DSL for Graph Analytics

Nibedita Behera, Ashwina Kumar, Ebenezer Rajadurai T, Sai Nitish, Rajesh Pandian M, Rupesh Nasre

[a]*Department of Computer Science and Engineering, Indian Institute of Technology Madras, , Chennai, 600036, Tamil Nadu, India*

Abstract

Graphs model several real-world phenomena. With the growth of unstructured and semi-structured data, parallelization of graph algorithms is inevitable. Unfortunately, due to inherent irregularity of computation, memory access, and communication, graph algorithms are traditionally challenging to parallelize. To tame this challenge, several libraries, frameworks, and domain-specific languages (DSLs) have been proposed to reduce the parallel programming burden of the users, who are often domain experts. However, existing frameworks to model graph algorithms typically target a single architecture. In this paper, we present a graph DSL, named StarPlat, that allows programmers to specify graph algorithms in a high-level format, but generates code for three different backends from the same algorithmic specification. In particular, the DSL compiler generates OpenMP for multi-core systems, MPI for distributed systems, and CUDA for many-core GPUs. Since these three are completely different parallel programming paradigms, binding them together under the same language is challenging. We share our experience with the language design. Central to our compiler is an intermediate representation which allows a common representation of the high-level program,

Email addresses: cs20s023@cse.iitm.ac.in (Nibedita Behera), cs20d016@cse.iitm.ac.in (Ashwina Kumar), ebenezerrajadurai5@gmail.com (Ebenezer Rajadurai T), bsainitishkumar@gmail.com (Sai Nitish), mrprajesh@cse.iitm.ac.in (Rajesh Pandian M), rupesh@cse.iitm.ac.in (Rupesh Nasre)

URL: https://orcid.org/0000-0002-1563-8686 (Nibedita Behera), https://orcid.org/0000-0001-6425-7479 (Ashwina Kumar), https://orcid.org/0000-0003-4702-4678 (Rajesh Pandian M), https://orcid.org/0000-0001-7490-625X (Rupesh Nasre)

from which individual backend code generations begin. We demonstrate the expressiveness of StarPlat by specifying four graph algorithms: betweenness centrality computation, page rank computation, single-source shortest paths, and triangle counting. Using a suite of ten large graphs, we illustrate the effectiveness of our approach by comparing the performance of the generated codes with that obtained with hand-crafted library codes. We find that the generated code is competitive to library-based codes in many cases. More importantly, we show the feasibility to generate efficient codes for different target architectures from the same algorithmic specification of graph algorithms.

Keywords: Graph Algorithms, Domain-Specific Language, OpenMP, MPI, CUDA

---

1. Introduction

The graph data structure has become an integral component of many real-world applications for modelling relationships in their data today. Enormous growth of unstructured and semi-structured data has led to these graphs growing to billions of edges. Therefore, parallel graph analytic solutions are inevitable to scale to such large graph sizes. The last two decades have witnessed significant advances in hardware towards parallel processing, which has also resulted in major developments in the software support, be it in the form of libraries or programming languages. These hardware and software architectures are primarily suited for regular codes wherein the data access, control flow, and communication patterns are statically identifiable. As an example, tiling of regular matrix computations can now be performed automatically by the compiler for improved cache benefits or with minimal communication.

Unfortunately, existing widely-used compilers perform poorly in the case of graph algorithms. This is due to the inherent *irregularity* in sparse graph processing, wherein the access patterns are dependent on the input (which is unavailable at compile time). On the other hand, our community has shown that graph algorithms exhibit enough parallelism to keep the cores busy on several real-world graphs [**?**]. Unfortunately, manually exploiting this parallelism on various hardware is quite challenging. The programmer needs to be an expert in the application domain to exploit algorithmic properties, in high-performance computing (HPC) to model the computation to suit

the target hardware, and also in computing systems to optimize the overall application and the support software on the given infrastructure. Even if corporations and institutions can find and afford such an expert, the expert is unlikely to be a best-fit for another application domain. A sore practical reality is that we have domain experts who are not HPC or systems experts, and we have HPC experts who may not know enough about the underlying application domain.

A viable alternative towards improved productivity is a graph library or a domain-specific language, which allows domain experts to express their algorithm using API or high-level constructs, and the library or the DSL compiler taking care of generating high-performing code for the target hardware. A range of such frameworks exists today [**?** **?** **?** **?**]. Multiple of these frameworks partially or fully hide the parallelism intricacies, provide mnemonics for scheduling strategies, and perform program analysis to identify races to generate synchronization code. It is often seen that the amount of code one needs to write reduces considerably compared to that in a hand-crafted explicitly parallel code.

One of the limitations of the existing frameworks (libraries or graph DSLs) is that they target a specific hardware architecture. For instance, Ligra [**?**] is a graph processing frameworks specifically for shared memory systems. Greenmarl [**?**] primarily targets multicore devices and supports distributed implementation through Pregel API. Gunrock [**?**] is a CUDA library for graph processing on GPUs. GraphIt [**?**] is a DSL that targets shared memory and manycore systems. Rarely, a framework targets more than one backends (e.g., Kokkos [**?**]). For the best performance and considering the range of HPC hardware we use today, libraries often restrict themselves to a certain target. Multi-core parallelism permits data shared across all the running threads, while a cluster-level parallelism demands explicit communication, while further, many-core parallelism necessitates a hierarchical computation and SIMD execution for the best performance. While domain-experts are aware of these basic differences, they should be able to *specify* rather than *code up* these for an architecture to achieve desired performance. Therefore, it is ideal if the domain-specific language encompasses different backend peculiarities in its design. This bears the advantage of simplified and efficient code generation, and avoids *patching* a language originally designed for a different parallel architecture.

In this work, we propose a graph DSL named StarPlat which allows a user to provide an algorithm specification of graph problems using its high-

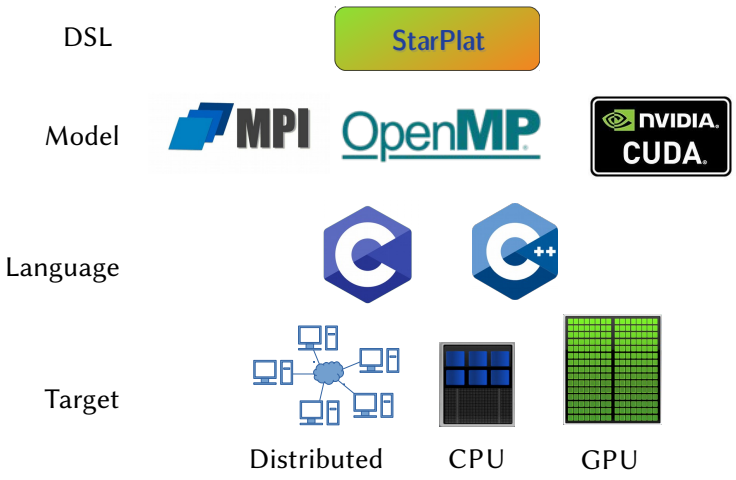


Figure 1: Overview of StarPlat compiler

level graph specific constructs and generates code for multiple backends from the same algorithmic specification (currently, multi-core, distributed, and many-core). The constructs are carefully designed to abstract the parallelization specific implementations from users, while ensuring generation of high performing graph codes. The StarPlat compiler moulds the high level information embedded in these constructs to different architecture specific implementations. The compiler translates the DSL code to C/C++. It uses OpenMP parallelism for the multicore setting, MPI for the distributed setup, and CUDA for the many-core architecture. Encompassing these three very different backends in the same DSL compiler is challenging, and we highlight these challenges in this manuscript. We proudly admit that we build upon the insights provided by the existing libraries and DSLs, borrow certain constructs from these frameworks (while providing our own), and generate code competitive to these frameworks in terms of performance. In particular, we illustrate the versatility of StarPlat with a discussion on four graph algorithms: Betweenness Centrality (BC), PageRank (PR), Single Source Shortest Paths (SSSP), and Triangle Counting (TC).

This work makes the following technical contributions:

- StarPlat[1], a DSL for graph analytics which allows users to write a high

[1] `https://github.com/nibeditabh/StarPlat`

level algorithmic specification of their static graph processing, which captures the parallelism intentions but decouples the target architecture.

- An intermediate representation common across backends that captures all the essential information associated with constructs for their translation in the target architecture.

- A code-generation scheme to support translation of the intermediate representation into efficient codes for the multicore (using OpenMP), distributed (using OpenMPI), and many-core (using CUDA) backends.

- Performance analysis of the generated BC, PR, SSSP, and TC codes for different target hardware on a variety of popular graphs, and illustrating competitive performance against that of the hand-crafted frameworks.

The rest of the article is organized as follows: Section 2 presents the language specification and the intermediate representation. Section 3 describes the code-generation scheme followed for the translation of the DSL code for each backend. Section 4 provides an overview of the backend-specific optimizations StarPlat employs for efficient code generation. The experimental evaluation of the generated code for each backend is discussed in Section 5. Section 6 discusses the related work for graph analytics. We summarise our experience and conclude in Section 7.

2. StarPlat Language and Frontend

The high-level philosophy of StarPlat is to relieve the user of the parallelization constructs as much as possible, and to achieve performance competitive to hand-tuned codes by providing constructs and hints on aggregates. In rare cases, when it is a must to have a trade-off between abstraction and performance, we have taken a conscious decision to prioritize abstraction. This is because the language is meant primarily for domain-experts (rather than HPC experts).

From day one, the language was designed to abstract away the hardware. This was a challenge, since the backends are quite different. But we have found commonalities at the algorithmic level which we encode using specific constructs, which could be then translated to the appropriate backend code.

For instance, when a lock-based synchronization was required in OpenMP, it also demanded communication in MPI, and a lock-free synchronization in CUDA.

StarPlat provides various abstractions and data types relevant to static graph algorithm, such as `Graph`, `node`, `edge`, `propNode` (for node property) etc. The programmer writes in a procedural style and hints at the parallelization opportunities using aggregate iteration constructs such as `forall` along with other inherently parallel operations. The compiler decides whether to exploit this parallelization (e.g., nested `forall`). Several algorithms can be viewed as iterative procedures repeatedly executed until convergence. The convergence criteria is application dependent. Such an iterative procedure is described using a `fixedPoint` construct. StarPlat follows a "batteries included" approach and has several utility functions for the data types. Internally, each of the functions is implemented target-hardware-wise.

2.1. Compiler Overview

As in a general-purpose compiler, StarPlat's compiler is split into frontend and backend, as shown in Figure 2. The frontend is responsible for ensuring syntactic and semantic consistency of the StarPlat implementation. The frontend builds an abstract syntax tree (AST) of the supplied input code. The AST is common across all the backends and is populated with the metadata for each construct during the parsing stage of the compiler. It is then fed to the appropriate code generator depending upon the target code the user wishes to generate.

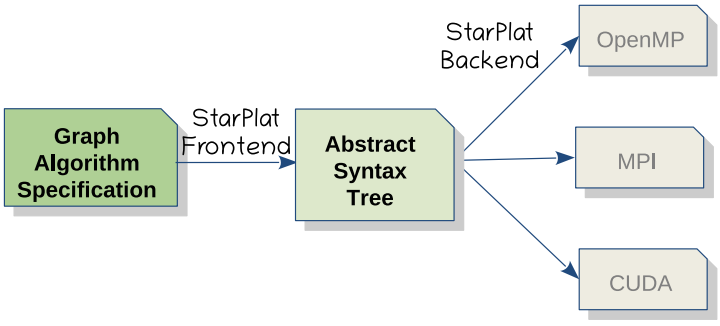


Figure 2: Process flow of the StarPlat compiler

2.2. Language Overview with an Example: SSSP

Single source shortest path computation (SSSP) finds a shortest path from a designated source node to every other node in the graph. Parallel

SSSP computation can be modelled as an iterative procedure, as illustrated in Figure 3. This is a variant of the Bellman-Ford's algorithm.

```
1  function computeSSSP (Graph g, node src) {
2      propNode<int> dist;
3      propNode<bool> modified;
4      g.attachNodeProperty(dist = INF, modified =
           False);
5      src.modified = True;
6      src.dist=0;
7      bool finished = False;
8      fixedPoint until (finished: !modified) {
9          forall (v in g.nodes().filter(modified
               == True)) {
10             forall (nbr in g.neighbors(v)) {
11                 edge e = g.getEdge(v, nbr);
12                 <nbr.dist, nbr.modified> =
                       <Min(nbr.dist, v.dist +
                       e.weight), True>;
13 }   }   }   }
```

Fig 3: SSSP computation in StarPlat

The function `compute_SSSP` takes two arguments: a graph g and source vertex src. Statement 2 declares a variable dist of type `propNode`. `propNode<int>` marks dist as an integer type attribute associated with each vertex. Statement 3 creates a `bool` type vertex attribute modified. In Statement 4, the function `attachNodeProperty` associates the two attributes with each vertex of g, and initializes the property values of dist to infinity (`INT_MAX`) and modified to false, as aggregate operations. In Statements 5–6, the dist and modified attributes for vertex src are assigned specific values. In particular, the distance of src is set to zero, and the vertex is also marked as modified. The `fixedPoint` construct in Statement 8 makes sure the iterative procedure executes until the variable finished becomes true. The variable finished is updated in each iteration, and decides if another iteration is necessary. A fixed-point is reached when no vertices are modified. This is achieved by or-ing the modified attribute of each node and assigning to finished. The iteration construct forall in Statement 9 specifies parallel iteration over the graph nodes. In addition, the optional filter clause allows processing a subset of the vertices meeting a criterion. The forall construct in Statement 10 specifies

iterating over all the neighbors of vertex `v` in parallel and for each neighbor nbr, the corresponding edge is relaxed (Statements 11–12). The multiple assignment construct in Statement 12 performs multiple tasks and is a suitable example of the abstraction provided by the DSL's constructs (borrowed from Green-Marl [**?**]). The rvalue `Min(nbr.dist, v.dist + e.weight)` is assigned to the dist attribute of `nbr` with the sum of `v.dist` and `e.weight` based on whether the alternative distance via v is smaller than the existing distance of `nbr`. If `nbr.dist` is updated, the modified attribute of `nbr` is set. Such a construct allows processing related statements together (as a critical section, to be precise) and translates to an update logic protected by some form of synchronization to prevent data races.

We highlight that the SSSP code structurally resembles the algorithm specified in a textbook, with a few changes. Hence, StarPlat eases implementation, analysis, and modification of graph algorithms from the context of parallelization. The multi-core translation of this code would follow a similar structure as the StarPlat code with OpenMP pragmas inserted; the distributed version differs in terms of complexities involved in scattering and gathering data across devices; while a many-core version need to pre-transfer the data being used inside `forall` from CPU to GPU.

We now delve deeper into various StarPlat constructs.

2.3. Language Constructs

We first discuss various data types, followed by iteration schemes and reductions.

2.3.1. Data Types

StarPlat supports graph theoretic concepts as first-class types such as Graph, node, edge, node attribute, edge attribute, etc. It also supports primitive data types: `int`, `bool`, `long`, `float`, and `double`.

The `Graph` data type encapsulates the operations and properties of a standalone graph. The properties include its nodes, edges, number of nodes, number of edges, etc. StarPlat stores the graph in Compressed Sparse Row (CSR) format, which provides the storage benefits of adjancency lists, and also allows seamless transfer across devices, due to the use of offsets. The data type also facilitates information gathering and manipulation at the node and the edge levels. Since the nodes and edges are tightly bound to the graph, it becomes convenient for Graph to support this through various library functions. For instance, as per the semantics of `neighbors(u)`, it returns

the outgoing neighbors in case of a directed graph and all the neighbors in case of an undirected one. For directed graphs, graph type also exposes a function that returns the incoming neighbours to a node, `nodesTo()`. This needs Graph to maintains a CSR representation for also its transpose, which becomes handy in algorithms which perform computation on a transposed version of the input graph (e.g., Betweenness Centrality).

```
1 function foo(Graph g, propNode<int> modified,
    propNode<int> data) {
2   SetN<g> nodeSet;
3   for(v in g.nodes().filter(modified)) { //
      Sequential loop
4     for(nbr in g.neighbors(v)) {
5       if(nbr.data > THRESHOLD) {
6         nodeSet.addNode(nbr);
7 }   }   }   }
```

Fig 4: Example program to illustrate various data types in StarPlat

A node and an edge can in themselves have properties associated with them. In the SSSP problem setting, a vertex's distance can be viewed as a node property, being computed by the corresponding algorithm. Similarly, in the BC computation, the betweenness centrality values of each node can be viewed as a property. The `propNode` datatype facilitates declaring property for nodes of a graph with the provision of specifying its type. The `attachNodeProperty` function provided by the Graph type binds this property to the graph's nodes and initializes the property values if provided. Line 2 in our SSSP code from Figure 3 specifies the declaration of a node property dist of type `int`. The `attachNodeProperty` binds dist to the graph and optionally, initializes the distance attribute for each vertex (e.g., to infinity in Figure 3). Similarly, `propEdge` datatype is associated with edges, and has otherwise the same semantics as that of `propNode`. The `attachEdgeProperty` function binds the property to the graph's edges.

StarPlat also provides collection types such as `List`, `SetN`, and `SetE`. `List` allows the presence of duplicates whereas `SetN` and `SetE` store unique nodes and edges respectively. Line 2 in Figure 4 shows an example usage. The separation of sets between nodes and edges enables choosing the relevant implementation in vertex-based vs. edge-based codes.

2.3.2. Parallelization and Iteration Schemes

`forall` is an aggregate construct in StarPlat which can process a set of elements in parallel. Its sequential counterpart is a simple for statement. Currently, StarPlat supports vertex-based processing[2]. The parallel `forall` supports various ranges it can iterate on (e.g., nodes in the whole graph or neighbors of a node, as shown in Figure 3, Lines 9 and 10).

The function `g.nodes()` called on a graph `g` returns a sequence of nodes which can be iterated upon. To iterate over the neighbors of a node `u`, the functions `g.neighbors(u)`, `g.nodesTo(u)` and `g.nodesFrom(u)` return a similar sequence. The `forall` body can be executed selectively for the nodes satisfying a certain boolean expression based on the node label or, node's property by including a `filter` construct. Line 3 in Figure 4 shows its usage using the node property `modified`.

Considering that several graph algorithms can be well represented using a single outer parallel loop, and that the analysis of nested parallel loops gets complicated, currently, StarPlat supports only outer level parallelism. Hence a nested `forall` in the DSL results in a parallel outer loop and a sequential inner loop in the target code. One may argue that an outer loop over vertices and an inner loop over neighbors can benefit from nested parallelism, and we agree. However, (i) such a processing can be well taken care of by an edge-based parallelism (to be supported), and (ii) since we target large graphs, even a vertex-based processing has enough parallelism to keep the resources busy.

For a graph processing DSL, traversals become the fundamental building blocks. StarPlat provides breadth-first traversal as a construct, borrowed from GreenMarl [**?**].

```
iterateInBFS(v in g.nodes() from root) {...}
```

`iterateInBFS` performs a BFS traversal of the graph from the given root node. The underlying processing is level-by-level and iterates in parallel over the visited nodes in a specific level. On visiting a node in a level, it executes the body statements and forms the next set of visited nodes. The `filter()` construct can be utilized to explore the neighborhood of a visited node selectively. Similarly, `iterateInReverse` performs a reverse BFS traversal in a

[2]We are adding support for edge-based processing, which needs changes to the underlying data representation. Compressed Sparse Row (CSR) storage format is suited for vertex-based processing.

level-synchronous manner and extracts parallelism at each level in the computation of the body statements. Note that `iterateInBFS` is a prerequisite to use `iterateInReverse`, since the former builds the BFS DAG to be traversed through in the latter. The functions `neighbors`, `nodesTo` and `nodesFrom` have a subtle change in their meaning when used inside `iterateIn...` constructs: they correspond to the neighbors in the BFS DAG rather than the original graph `G`. We have found this semantics change to satisfy our natural inclination to write a code. We illustrate it in the BC computation (Appendix Appendix A).

2.3.3. Reductions

Reductions are one of the popular parallel programming primitives, and can be useful in achieving efficient computation. Specifying a reduction in the DSL also helps in conveying a necessity of synchronization. Unfortunately, it does not directly fit into the philosophy of StarPlat design to support reduction as a language construct. Therefore, as a golden-mid, StarPlat permits usage of certain relative C-operators (e.g., `+=`) to convey reduction. This "trick" allows us to retain the abstraction and still achieve efficiency of the generated code. The reduction operators supported by StarPlat are tabulated in Table 1.

Operator	**Reduction Type**
`+=`	Sum
`*=`	Product
`++`	Count
`&&=`	All
`||=`	Any

Table 1: Reduction operators in StarPlat

We illustrate the usage of reduction in Figure 5. The introduction of reduction (Line 7 in the code makes sure the `accum` variable has a deterministic result at the end of the parallel region. Note that Line 5 involves a thread-local variable `count` and does not need reduction. On the other hand, if nested parallelism was supported, `count` would also need a reduction. The reduction operators in StarPlat translate to library based implementations

of reduction in the target backend.[3]

```
1 int accum = 0;
2 forall(v in g.nodes()){
3     int count = 0;
4     forall(nbr in g.neighbors(v)){
5             count = count + nbr.A;    // regular
                assignment
6     }
7    accum += count; //sum reduction in assignment
       form
8 }
```

Fig 5: Reduction example

2.3.4. fixedPoint and Min/Max Constructs

Several solutions to graph algorithms are iterative, and converge based on conditions on node attributes. StarPlat provides a fixedPoint construct to specify this succinctly. Its syntax involves a boolean variable and a boolean expression on node-properties forming the convergence condition, as shown below.

```
fixedPoint until (var: convergence expression) {...}
```

Line 8 of Figure 3 in SSSP's specification uses the `fixedPoint` construct to define the convergence condition. The loop iterates till at least one node's `modified` property is set to true.

StarPlat provides constructs `Min` and `Max` which perform multiple assignments based on a comparison criterion. This can be useful in update-based algorithms like SSSP, where an update on node properties is carried out on a desired condition, while taking care of potential data races.

In the SSSP computation, the `Min` construct is used to encode the relaxation criteria in Line 12. The neighboring node's distance is updated if the alternative distance via vertex `v` is smaller than `nbr`'s current distance. The update of the `dist` property based on this comparison specified using `Min` results in an update of the `modified` property to `True`.

StarPlat also has aggregate functions `minWt` and `maxWt` to find the minimum and maximum edge weights.

[3]Currently, the onus of writing the correct operator is onto the DSL user. We are adding program analysis to StarPlat which would relieve the user of worrying about this.

2.4. Abstract Syntax Tree (AST)

Each meaningful non-terminal that builds the language are at the highest level denoted as `ASTNode`. The ASTNode is the parent class of all the nodes that are part of the abstract syntax tree. The ASTNode forges to child class nodes that signify specialization like `statement` Node, `Expression` Node. The `statement` node is the parent class of all statement types like assignment, declaration, control flow, procedure call, etc. Each of these nodes can sometimes be composed of other nodes, owing to the way the construct is defined in the parser. For instance, `forallStmt's` construct is composed of a body, an iteration space which is defined by an iterator and range (functions) and an optional filter expression on the iterated items. The body in forallStmt is represented as a statement node, iterator as an `Identifier` node, ranges as `proc_callExpr`, and filter expressions as *Expression* node in the AST. The other node types are `Identifier`, `iterateReverseBFS`, `Type`, `reductionCall`, `fixedPointStmt`, `formalParam` etc.

Most of the node's data are populated in the parsing stage itself. Data related to the type of the symbols are added during an additional pass through the already built AST. Given the target backend, the construct's AST are populated with additional information relevant for the code-generation in that specific backend. The analyzer phase before the code generation performs analysis for optimized and efficient code generation. Some of these optimizations are specific to the target backend. The separation of nodes into different subclasses allows maintaining handlers for different node types in the code generator. The code-generator forms the backend part of the compiler. Unlike the frontend which was primarily target-independent, backend is tightly coupled with the target parallelization platform.

3. StarPlat Code Generator

The code generation phase supports the generation of efficient code for primarily three target backends, shared memory, distributed, and many-core platforms. The generated code for each backend also makes use of specific backend libraries to handle the parallel processing intuitively specified in the StarPlat code. Shared memory implementation uses OpenMP as the threading library, distributed memory implementation uses MPI, and many-core implementation uses CUDA.

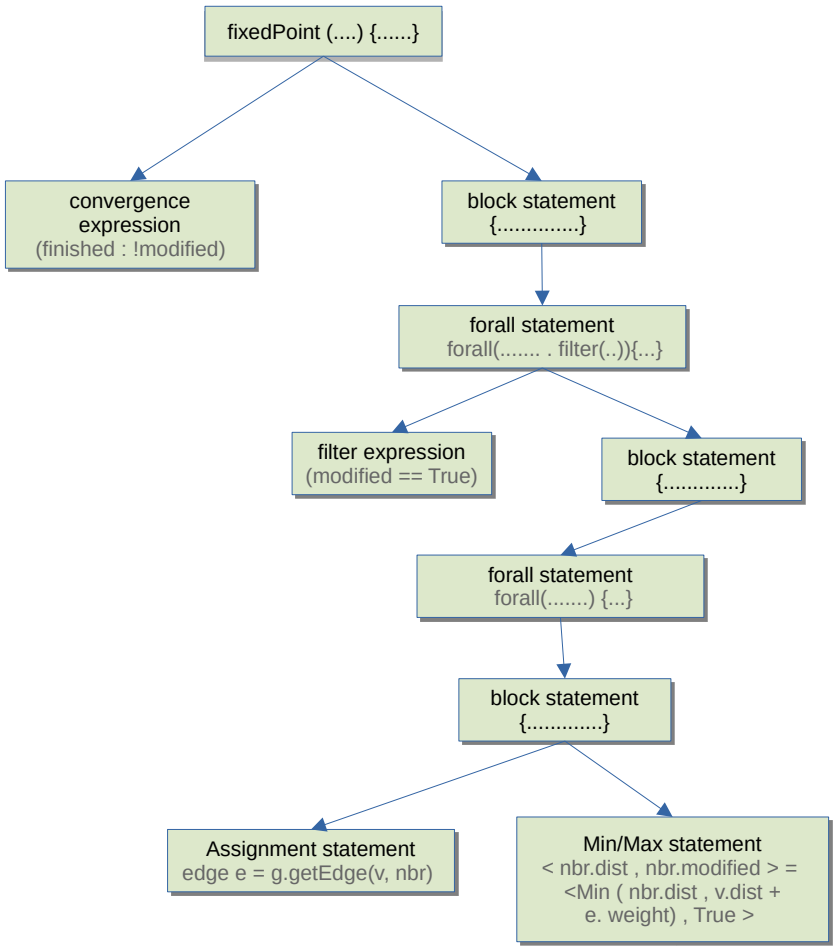


Figure 6: AST for SSSP's `fixedPoint` construct in Figure 3

3.1. Graph Representation and Storage

While there are several formats in which graphs can be stored, we preferred a format which:

- can preferably work across all the backends
- works well with vertex-centric algorithms, common in graph processing
- can be easily split

The last preference is not only due to the MPI backend, but also due to our future plans to extend StarPlat for multi-GPU and heterogeneous backends. Since offset-based formats fit our requirements, we had the options of compressed sparse row (CSR) and coordinate list (COO) formats, of which CSR fits the second preference. Therefore, we chose to use it. CSR is also popularly used in sparse graph processing frameworks. StarPlat has a backend graph library which takes care of loading a graph from the input file, and storing it in the CSR format.

The OpenMP backend does not pose any issues with respect to storing the graphs. In case of the MPI backend, the graph is loaded by rank 0 process and distributed using a simple block partitioning approach among the MPI processes, resulting in *local* and *remote* vertices for each process. The information of only the local vertices is available for direct access, while that of the remote vertices needs to be communicated with explicit send-receives. The MPI backend maintains a mapping between the local vertex id and its corresponding global vertex id, for correct processing and for sending and receiving data. This is because, for instance, a large global vertex id may still map to a local id of 0 in an MPI process. The CUDA backend needs to transfer the graph and the associated attributes fully from the host to the GPU (using multiple `cudaMemcpy` calls). Since the algorithms work with static graphs, only the modified attribute information needs to be brought back from the GPU to the host, at the end of the processing.

Seamless management of both the graph representation and the attribute storage across various backends relieves the domain-user of complex and erroneous graph handling.

3.2. Overall Flow

In the OpenMP backend, the outermost `forall` gets translated to have `#pragma omp parallel for` as a header. Input graphs are parsed and stored

as C++ classes in the generated code, while the nodes and the edges are represented as integers (for their ids). The node properties and edge properties are translated as arrays of types specified in the property declaration. We have found that generating OpenMP code is relatively easier compared to the other two backends due to (i) a single device, (ii) shared memory paradigm wherein all the variables are present in the same memory address space for all the threads, and (iii) pragma based processing which does not need to change the sequential code.

The MPI backend generates code that follows a bulk-synchronous processing (BSP) style, which involves a computation step followed by a communication step in every iteration. During the computation step, various MPI processes execute the main-loop code, update the attributes of their local vertices, and mark those for sending in a send-buffer for the corresponding global vertices. During the communication phase, processes receive these updates from other processes, convert those to their local vertices, and update the attributes. Similar to OpenMP, the node edge attributes / properties are translated as arrays.

Unlike OpenMP and MPI, the CUDA backend needs to convert the `forall` loop into a GPU kernel, which has a different scope. This makes certain variables unavailable, which need to be explicitly allocated (`cudaMalloc`) and transferred (`cudaMemcpy`). In addition, the `forall` loop body may contain local variables, which get translated to thread-local variables in CUDA. To distinguish between these two kinds of variables, the CUDA backend performs a rudimentary analysis of the AST.[4] Apart from each outermost `forall`, the initialization to attribute values needs to be converted to a separate kernel (which is a loop in OpenMP and MPI). The kernel launch configuration is set to a fixed number of threads per block (1024 in the generated code, which performs the best on an average in our experiments) and the number of blocks proportional to the number of vertices.

Apart from this, the boilerplate code consists of calls to timing functions to time the initialization, the fixed-point processing, and the data transfer where applicable.

[4] A full-fledged program analysis of the algorithm at the AST level is an ongoing work, which involves identifying races and addressing those with proper synchronization or communication.

3.3. Neighborhood Iteration

Iterating over the neighborhood is typically nested inside iterating over the graph vertices (possibly, with a filter, as in Figure 3). In the case of the OpenMP backend, the generated code snippet is as below.

```
1 #pragma omp parallel for
2 for (int v = 0; v < g.num_nodes(); v++) {
3     if (!modified[v]) continue;
4     for (int edge = g.indexofNodes[v]; edge <
          g.indexofNodes[v + 1]; edge++) {
5         int nbr = g.edgeList[edge];
6         ...
7 }   }
```

Fig 7: OpenMP code generated for neighborhood iteration

The MPI code is very similar, but goes over a predefined range based on the process rank.

```
1  mpi::communicator world;
2  myrank = world.rank();
3  np = world.size();
4
5  part_size = ceil(g.num_nodes() / (float)np);
6  startv = myrank * part_size;
7  endv = startv + part_size - 1;
8  ...
9  for (int v = startv; v <= endv; v++) {
10     for (int edge0 = local_index[v-startv];
           edge0 < local_index[v-startv+1]; edge0++)
           {
11             int nbr = local_edgeList[edge0];
12         ...
13 } }
```

Fig 8: MPI code generated for neighborhood iteration

The CUDA code needs to separate the kernel call and kernel processing. The kernel call sets up the launch configuration (number of thread-blocks and the size of each thread-block) and passes appropriate parameters (graph, attributes, algorithm specific variables such as the source node in SSSP, and

other miscellaneous variables used internally). The kernel uses 1D thread id, and utilizes the CSR copied to the GPU to go over the neighbors. Note that unlike OpenMP and MPI, there is no loop over the vertices, which is handled by parallel threads with which the kernel is launched.

```
1  __global__ void computeSSSP(...) {
2      unsigned id = blockIdx.x * blockDim.x +
           threadIdx.x;
3      ...
4      for (int edge = gpu_OA[id]; edge < gpu_OA[id
           + 1]; edge++) {
5          int nbr = g.gpu_edgeList[edge];
6          ...
7      }
8      ...
9  }
10 ...
11 computeSSSP<<<nblocks, blocksize>>>(...);
12 cudaDeviceSynchronize();
```

Fig 9: CUDA code generated for neighborhood iteration

The loop bound functions change appropriately for directed vs. undirected graphs, and for in-neighbors (`g.nodesTo(v)`) which demand using reverse adjacencies (`g.revIndexofNodes[v]`).

3.4. Reductions

Recall the reduction supported in StarPlat (e.g., Line 7). In the OpenMP backend, the reduction on a scalar translates to OpenMP's `reduction` clause as below. OpenMP reduction implementation creates and updates a private copy of the reduction variable per thread. At the exit of the parallel region, the local changes are accumulated in the global variable. This significantly reduces the data race handling costs required for the correctness of operation on a shared variable.

```
#pragma omp parallel for reduction(+:accum)
```

The situation in CUDA gets complicated due to limitations on the number of resident thread-blocks, which can block the kernel. One option to counter this is to use a reduction from the host as a separate kernel (e.g., using `thrust::reduce`). But this demands terminating the kernel, coming back to the host, calling reduce, coming back to the host, and then calling

another kernel to perform the rest of the processing in the `forall` loop. This not only adds complication to the code generation, but also makes the overall processing inefficient. Therefore, we rely on atomics to generate the functionally equivalent code (e.g., using `atomicAdd` in the example in Figure 5).

```
atomicAdd(&accum, prop[nbr]);
```

3.5. BFS Traversal

The `iterateInBFS` construct gets expanded to a parallel BFS. This single construct expands to around 40 lines of OpenMP code, 60 lines of MPI code, and 40 lines of CUDA code. Therefore, we show and discuss only crucial parts of the snippets below.

The OpenMP backend maintains a vector of vectors for tracking vertices level-wise as shown in the code snippet in Figure 10. The BFS has an outer `while` loop and two inner loops: one to go over the vertices in a level (this is parallelized at Line 6) and the other to add the explored vertices in a level to the appropriate level-vector (Line 18), which acts as a frontier. The parallel loop examines a vertex's neighbors and checks if their current distances are not set (value -1) and sets those to 1 + current vertex's distance. To avoid data races, this update needs to be done atomically using a CAS (Line 9).

```
1  levelNodes[phase].push_back(root);
2  ...
3  while (bfsCount > 0) {
4      ...
5      #pragma omp parallel for
6      for(int l = 0; l < prev_count ; l++) {
7          int v = levelNodes[phase][l];
8          ...
9          dnbr =
               __sync_val_compare_and_swap(&bfsDist[nbr],
               -1, bfsDist[v] + 1);
10         if (dnbr < 0) {
11           int num_thread = omp_get_thread_num();
12           levelNodes_later[num_thread].push_back(nbr);
13         }
14         // body of iterateInBFS
15     }
16     phase = phase + 1;
```

```
17
18      for (int i = 0; i < omp_get_max_threads();
            i++) {
19
                levelNodes[phase].concat(levelNodes_later[i]);
20          ...
21      }
22  }
```

Fig 10: OpenMP code generated for the `iterateInBFS` construct

The MPI backend runs a similar outer `while` loop and goes over the vertices level by level, but only for the set of local vertices. Once a vertex's neighbor is updated, it is pushed to the send buffer (Line 14). At the end of each level, an all-to-all call exchanges the level values of local and remote vertices (Line 17). Different levels are separated by an MPI barrier in a BSP style (Line 23).

```
1  while(is_finished(startv, endv, active)) {
2      ...
3      while (active.size() > 0) {
4          int v = active.back();
5          active.pop_back();
6          ...
7          for (int j = local_index[v - startv]; j
               < local_index[v - startv + 1]; j++) {
8              int w = local_edgeList[j];
9              //If local
10             if (d[w-startv] < 0) {
11                 active_next.push_back(w);
12                 d[w-startv] = d[v-startv] + 1;
13             } else
14                 // send d[v], w, and v
15         ...
16     }
17     all_to_all(world, sendbuf, receivebuf);
18     for (int t = 0; t < np; t++) {
19         // process receives
20         if (d[w-startv] == d_v + 1)
21             //Body of iterateInBFS
```

```
22      }
23      MPI_Barrier(MPI_COMM_WORLD);
24      phase = phase + 1 ;
25  }
```

Fig 11: MPI code generated for the `iterateInBFS` construct

The CUDA backend poses a challenge for the `iterateInBFS` construct because unlike the other two backends, the code needs to be generated to be running on both the host and the device. The outer `do-while` loop runs on the host (Line 1), which internally calls the level-wise BFS kernel on the GPU (Line 13). Since the loop is on the host, for ensuring progress with the loop's condition, the flag (`finished`) needs to be repeatedly copied across devices (Lines 2 and 8), passed as a parameter to the kernel, and updated in the kernel whenever there is change of a vertex's level.

```
1  do {
2      H2D(...);
3
4      BFS<<<...>>>(...)
5      cudaDeviceSynchronize();
6      ++hops_from_source;
7
8      D2H(...);
9      ...
10
11 } while (!finished);
12
13 __global__ void BFS(...) {
14     ...
15     if (d_level[u] == *d_hops_from_source) {
16         ...
17         for (unsigned i = d_offset[u]; i < end;
               ++i) {
18           unsigned v = ... if (d_level[v] ==
                 -1) {
19             d_level[v] = *d_hops_from_source
                   + 1;
20             *d_finished = false;
21           }
```

```
22                  // Body of iterateInBFS
23 }    }    }
```

Fig 12: CUDA equivalent code for the `iterateBFS` construct

3.6. `Min/Max` Constructs

Recall the `Min` construct from SSSP (Line 12). In OpenMP, the conditional update on the node property is achieved through atomic implementations of these operations built upon atomic CAS (Compare and Swap), made available as a StarPlat library. The generated OpenMP code for the `Min` construct in SSSP is as follows.

```
1  int dist_new = dist[v] + weight[e];
2  bool modified_new = true;
3
4  if (dist[nbr] > dist_new) {
5      int oldValue = dist[nbr];
6      atomicMin(&dist[nbr], dist_new);
7
8      if(oldValue>dist[nbr] {
9          modified_nxt[nbr] = modified_new;
10 }    }
```

Fig 13: OpenMP code generated for the `Min` construct

In the MPI backend, the neighbor could be local or remote. If it is local, each process can update its property without synchronization. If it is remote, the property value to be updated is stored in the send buffer, which is eventually sent to the owning process (during the communication phase).

```
1  if (nbr >= startv && nbr <= endv) { // local
2      int e = edge0 + startv;
3      int dist_new = dist[v-startv] +
           weight[e-startv];
4      bool modified_new = true;
5
6      if (dist[nbr-startv] > dist_new) {
7          dist[nbr-startv] = dist_new;
8          modified[nbr-startv] = modified_new;
9      }
```

```
10 } else {  // remote
11     sendbuf[owner(nbr)][nbr] = dist[v-startv] +
           weight[e-startv];
12 }
13 ...
14 all_to_all(world, sendbuf, receivebuf);
```

Fig 14: MPI code generated for the `Min` construct (comments are not generated)

The CUDA backend handles Min/Max constructs exactly similar to the OpenMP backend, except (i) the variables used are the GPU copies, and (ii) the atomic instructions are readily supported in CUDA (`atomicMin` and `atomicMax`).

```
1 int nbr = gpu_edgeList[edge];
2 int e = edge;
3 int dist_new = gpu_dist[v] + gpu_weight[e];
4
5 if (gpu_dist[nbr] > dist_new) {
6     atomicMin(&gpu_dist[nbr], dist_new);
7     gpu_modified_next[nbr] = true;
8     gpu_finished[0] = false;
9 }
```

Fig 15: CUDA code generated for the `Min` construct

3.7. `fixedPoint` Construct

The `fixedPoint` construct translates to a while loop conditioned on the fixed-point variable provided in the construct. Typically, the convergence is based on a node property, but it can be an arbitrary computation. This code is generated on the host, so it is similar in template for all the three backends. However, as before, the MPI code needs to gather this information for setting the flag `finished` from all the processes based on the vertex ids they operate on. Similarly, the CUDA code updates a copy of the `finished` flag on the GPU, which is `cudaMemcpy`'ed to the host.

```
1 while (!finished) {
2     finished = true;
3
4     if (...) { // Min-construct expansion
```

```
5            ...
6            finished = false; // fixedPoint
                identifier updation
7  }    }
```

Fig 16: Code generated for the `fixedPoint` construct across all the backends

4. Optimizations

There are certain optimizations which a hand-crafted code can embed, which StarPlat's code generator cannot do. For instance, if variants of an algorithm perform better on different backends, we cannot auto-generate the variants from the same algorithmic specification (e.g., push versus pull based processing [**?**]), although the DSL is capable of specifying the variants separately. At the implementation level, persistent kernels in CUDA [**?**] can improve execution time of algorithms on large diameter graphs. However, StarPlat generates the code for the `fixedPoint` construct on the host – across backends. Despite this, our observation has been that many optimizations can be readily embedded into the code generated by StarPlat:

- The algorithmic optimizations can be embedded into the DSL specification itself, which forms a common input to all the backends. As an illustration, Figure 21 in the Appendix presents a pull-based SSSP.
- A backend-specific optimization can be embedded into that specific backend, since such an optimization will not be explicitly mentioned in the DSL specification and will be abstracted away.

We discuss below the backend-specific optimizations.

4.1. OpenMP

Avoiding false sharing in `iterateInBFS`. In the case of BFS abstractions, nodes discovered by multiple threads at a particular level are pushed to a data structure required for graph traversal. The false sharing due to shared usage of the same array in this scenario has been alleviated by assigning local update space to each thread.

Efficient fixed-point computation. The fixedPoint construct converges on a specific condition on a single boolean node property. The change of convergence is tracked through a boolean fixed-point variable ideally needs

to be updated after analyzing the property values for all nodes. The update procedure has been optimized by updating the fixed-point variable along with the update to the property value for any node. Since, updates to a boolean variable by multiple threads are atomic by hardware, this does not lead to a performance loss.

Using built-in atomics. Atomics has proved to be lightweight. Hence, atomics implemented using built-in functions provided by the GCC compiler is being used in generated code for achieving exclusive access to a single statement by a thread instead of locks.

4.2. MPI

Communication aggregation. The communication aggregation optimization is used by the MPI code while updating the distance of remote neighbor vertices. Instead of sending multiple messages to a remote vertex by a processing node, a single message with local minimum value is sent to the remote vertex.

Using Boost. The packing/unpacking mechanism provided by Boost MPI library gives better performance in sending and receiving STL containers such as vectors, maps etc., and user-defined data types than MPI library calls.

Quick index-based partitioning. StarPlat currently uses an index based partitioning to distribute the graph among various MPI processes. Compared to using a partitioner such as Metis, ours has the down-side of increased communication due to several inter-partition edges. However, such a partitioning is quick, and also allows us to partition the graph arbitrarily based on the number of MPI processes. Presently, the implementation assumes that all the processes have equal number of vertices. Hence, we pad temporary vertices for the last process.[5]

4.3. CUDA

Optimized host-device transfer. We perform a rudimentary program analysis of the AST to identify variables that need to be transferred across devices. For instance, since graph is static, it need not be copied back from

[5]We definitely plan to exploit Metis and its parallel variants for MPI, multi-GPU, and heterogeneous backends.

GPU to CPU at the end of the kernel. In contrast, the modified properties need to be transferred back. Similarly, the `finished` flag is set on CPU, conditionally set on GPU, and read on the CPU again in the fixed-point processing. Therefore, the variable needs to be transferred to-and-fro. The `forall`-local variables are generated as device-only variables.

Memory optimization in OR-reduction. The way we write the `fixedPoint` construct, a `modified` property is used in computing the fixed-point. At a high-level, another iteration is necessary if any of the vertices' `modified` flag is set. This is essentially a logical-OR operation. StarPlat takes advantage of this to generate a single flag variable which is set by threads in parallel (with dependence on hardware atomicity for primitive types). Managing this flag is cheaper than transferring arrays of the `modified` flags across devices, both in terms of time and memory.

5. Experimental Evaluation

To quantitatively assess StarPlat, we generate four popular algorithm implementations: Betweenness Centrality (BC), PageRank (PR), Single Source Shortest Path (SSSP), and Triangle Counting (TC). These are also coded and optimized using the other frameworks we compare against. The StarPlat code for SSSP is presented in Figure 3, while the other three are presented in the Appendix. Each codes fits in about 30 lines, and with that effort, a domain-expert or a student can generate efficient implementations of the four algorithms for a multi-core setup, a distributed system, and a GPU.[6]

Since the DSL codes are rather short, the compilation is immediate. Therefore, we focus on analyzing the efficiency of the generated codes for the various backends. We compare the performance of the backend-specific code in StarPlat against the existing state-of-the-art graph analytic solutions (qualitative comparison is discussed in Section 6). StarPlat's OpenMP backend is compared against Galois [**?**], Green-Marl [**?**], and Ligra [**?**]; its MPI backend is compared against Gluon [**?**]; and its CUDA backend is compared against GunRock [**?**] and LonestarGPU [**?**]. Galois is a C++-based framework for graph analytics, while Ligra is C-based. In addition, Galois also supports graph mutation, which is not the focus of other frameworks. Similar to StarPlat, Green-Marl is a graph DSL designed for the multi-core

[6] We believe this is remarkable, and would be appreciated by the community.

backend. Gluon is a distributed version of Galois built as a C++ framework. LonestarGPU is a manually optimized collection of CUDA codes, while Gunrock is a CUDA-based library providing data-centric API for graph analytics, such as *advance*, *compute*, and *filter*. So, except for Green-Marl, we compare StarPlat-generated codes against manually optimized ones.

We use ten large graphs in our experiments, which are a mix of different types. Six of these are social networks exhibiting the small-world property, two are road networks having large diameters and small vertex degrees, while two are synthetically generated. One synthetic graph has a uniform random distribution (generated using Green-Marl's graph generator), while the other one has a skewed degree distribution following the recursive-matrix format (generated using SNAP's RMAT generator with parameters a = 0.57, b = 0.19, c = 0.19, d = 0.05). They are listed in Table 2, sorted on the number of edges in each category. For unweighted graphs, we assign edge-weights selected uniformly at random in the range [1,100] (for SSSP).

Graph	**Acronym**	**Num. Vertices** (million)	**Num. Edges** (million)	**Avg. Degree**	**Max. Degr**
twitter-2010	TW	21.2	265.0	12	302,7
soc-sinaweibo	SW	58.6	261.0	4	4,0
orkut	OK	3.0	234.3	76.2813	33,3
wikipedia-ru	WK	3.3	93.3	55.4067	283,9
livejournal	LJ	4.8	69.0	28.257	22,8
soc-pokec	PK	1.6	30.6	37.5092	20,5
usaroad	US	24.0	28.9	2	
germany-osm	GR	11.5	12.4	2	
rmat876	RM	16.7	87.6	5	128,3
uniform-random	UR	10.0	80.0	8	

Table 2: Input graphs

All our experiments, including the baselines we compare against, were run on IIT Madras AQUA cluster. The configuration of each compute node as follows: Intel Xeon Gold 6248 CPU with 40 hardware threads spread over two sockets, 2.50 GHz clock, and 192 GB memory running RHEL 7.6 OS. All the codes in C++ are compiled with GCC 9.2, with optimization flag -O3. Various backends have the following versions: OpenMP version 4.5, OpenMPI version 3.1.6, CUDA version 10.1.243 and run on Nvidia Tesla

V100-PCIE GPU with 5120 CUDA cores spread uniformly across 80 SMs clocked at 1.38 GHz with 32 GB global memory and 48 KB shared memory per thread-block.

5.1. OpenMP

Table 3 presents the running times of the codes in various frameworks for the four algorithms. The execution time refers to only the algorithmic processing and excludes the graph reading time, and is an average over three runs (for each backend). The Galois framework failed to load the largest graph TW (exited with a segfault). The framework also failed for BC computation on 150 sources for US graph and exited by giving a PTS out of memory error.

Betweenness Centrality. BC resembles all-pairs shortest paths, which can be implemented by running SSSP from each vertex as a source. Complete execution of BC on large graphs takes several hours, sometimes days. Depending upon how an implementation stores BC results for each source, the memory requirement can also increase proportional to $\sim V \sim$ per source. Therefore, literature presents results of running BC from one or only a few vertices. We tabulate results for BC for number of source vertices as {20, 80, 150}. The source list for each graph is generated using a random generator and fixed across frameworks for consistent comparison. StarPlat's BC code outperformed Galois code for the road network graphs and some of the social network graphs. Galois framework involves a scheduling policy that ensures better load balance among threads. But for jobs with even workloads, the scheduling adds to the runtime overhead. Ligra has an optimized forward BFS pass that bypasses distance computation for nodes, and computes only the number of shortest paths for each node, thus saving on the synchronization required for concurrent distance computation. This optimization results in Ligra outperforming StarPlat and Galois. Interestingly, the Green-Marl-generated code outperforms the other three frameworks on most inputs. Green-Marl's forward BFS traversal is optimized and chooses the nature of traversal at each level based on specific criteria.

PageRank. StarPlat-generated PR code is competitive to that of Green-Marl across inputs. Ligra, interestingly, has considerably slower implementation and takes significantly longer. This is primarily due to the loop separation between the computation of the PR values and the difference of successive PR values for each vertex. Overall, Galois (about 2× faster than

Algo.		Framework	TW	SW	OK	WK	LJ	PK	US	GR	RM	UI
BC	20	Galois	-	**1.840**	8.272	3.377	3.209	1.477	38.488	15.368	**2.861**	29.52
		Ligra	11.200	13.130	5.936	3.470	4.060	**1.220**	**18.160**	10.430	4.250	4.736
		GreenMarl	**4.289**	3.492	**5.504**	**2.444**	**2.218**	1.173	18.823	**9.602**	3.187	**3.573**
		StarPlat	6.530	11.060	7.650	3.300	4.422	1.777	22.250	11.740	5.580	12.83
	80	Galois	-	**7.269**	50.171	13.548	12.139	5.596	258.447	75.969	**7.435**	85.534
		Ligra	48.700	48.760	24.100	14.230	16.160	4.550	**73.230**	40.500	10.900	19.130
		GreenMarl	**18.189**	12.172	**21.094**	**9.353**	**8.578**	**4.107**	75.450	**39.704**	7.863	**14.184**
		StarPlat	33.240	44.270	32.150	13.050	16.550	6.810	86.400	47.290	18.030	50.260
	150	Galois	-	**14.598**	78.632	25.403	23.816	11.132	-	138.052	12.365	116.70
		Ligra	83.400	90.360	44.630	26.260	31.500	8.996	**124.66**	76.530	17.700	35.800
		GreenMarl	**32.990**	23.110	**38.612**	**17.709**	**17.231**	**8.184**	139.090	**74.506**	**11.069**	**26.186**
		StarPlat	61.625	84.430	57.664	24.520	32.290	13.553	160.865	80.111	32.136	94.540
PR		Galois	-	**0.510**	**0.647**	**0.371**	**0.474**	**0.156**	**0.607**	**0.224**	**0.324**	**0.443**
		Ligra	25.600	162.660	5.050	3.930	3.623	0.836	2.050	0.822	5.880	0.942
		GreenMarl	**0.585**	7.211	1.437	0.512	0.585	0.263	1.235	0.525	0.821	0.688
		StarPlat	1.752	9.002	1.213	0.473	0.509	0.236	1.600	0.667	0.883	0.619
SSSP		Galois	**0.522**	**0.132**	**0.404**	**0.203**	**0.205**	**0.099**	**19.387**	**6.798**	**0.133**	**0.480**
		Ligra	10.7	0.148	5.136	1.846	3.89	1.683	283.000	9.043	2.74	10.40
		GreenMarl	2.182	0.891	1.048	1.16	0.761	0.292	193.548	48.349	0.464	1.361
		StarPlat	5.831	1.437	1.850	1.759	2.412	0.846	294.303	53.395	1.369	5.237
TC		Galois	-	**56.432**	33.110	**46.168**	9.811	3.008	0.061	**0.020**	184.260	2.350
		Ligra	2103.333	188.660	**22.800**	97.360	10.460	1.926	0.147	0.0698	**130.330**	1.706
		GreenMarl	11611.029	4257.498	137.559	4564.568	29.426	12.705	0.065	0.021	5647.156	1.435
		StarPlat	1414.323	59.925	23.420	111.430	**7.544**	**1.559**	**0.059**	0.024	158.760	**1.170**

Table 3: StarPlat's OpenMP code performance comparison against Galois, Ligra and Green-Marl (20 Threads). All times are in seconds. BC is run with the number of sources as {20, 80, 150}.

SSSP	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR
	4.127	0.127	1.503	0.633	2.315	0.822	72.654	9.641	1.319	4.477

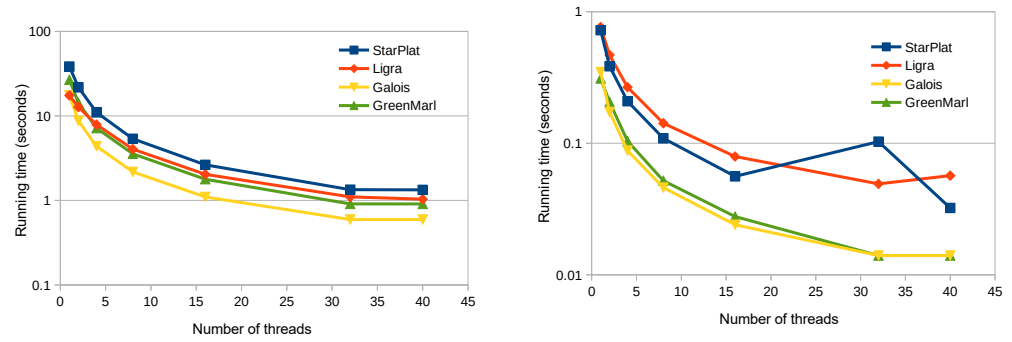
Table 4: StarPlat's SSSP OpenMP code running times (seconds) with `static` scheduling

StarPlat-generated code) outperforms all the other benchmarks. This is due to the in-place update of the PR values for vertices, which leads to faster convergence. StarPlat, Green-Marl, and Ligra follow a similar processing of updating the PR values using double buffering.

Single-Source Shortest Paths. We find that Galois SSSP is faster than StarPlat's code and other compared benchmarks. This is due to application-specific prioritized scheduling in the Galois framework [**?**]. For instance, processing tasks in the ascending distance order reduces the total amount of extra work done. GreenMarl and StarPlat have nearly similar implementations for SSSP calculation. Both follow a dense push configuration for vertex processing which requires iterating over all the vertices to check if they are active. This is typically costly for road networks which have a smaller frontier active in each iteration. In addition, this additional cost accumulates over several iterations due to the large diameter of road networks. However, GreenMarl performs better than StarPlat for nearly all graphs. GreenMarl uses spin-lock implementation with the back-off strategy to save on unnecessary CPU cycles. StarPlat also uses a lock-free atomic-based implementation but the unnecessary updates are more profound leading to performance degradation due to false sharing. Ligra's performance is not very competitive with other benchmarks. Ligra switches between sparse and dense edge processing based on the frontier size. But this direction optimization does not lead to considerable performance improvement for the given graph suite except for the road networks.

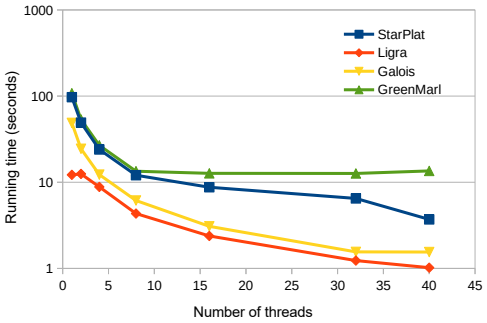
By default, StarPlat generates OpenMP code with `dynamic` scheduling. This largely works well across various algorithms and graph types. However, SSSP code seems to overall perform better with static scheduling as shown in Table 4. The difference is pronounced for large diameter graphs US and GR wherein the execution times reduce from over a minute to a few seconds.

Triangle Counting. Galois, StarPlat, and Green-Marl follow a node-iterator pattern in TC. On the other hand, Ligra follows an edge-iterator based version, which is supposed to work better for skewed degree graphs, since the edge-based version has better load balance. We observe that for different graphs, different frameworks outperform. Interestingly, performance of the Green-Marl-generated code is significantly poorer.

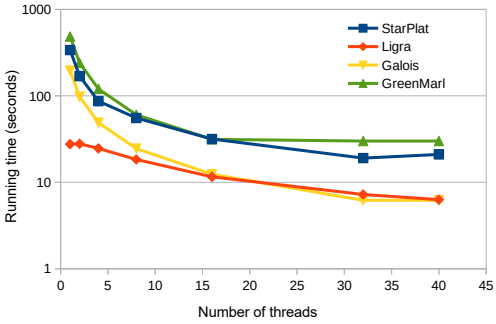


(a) UR

(b) GR



(c) PK



(d) LJ

Figure 17: Log plots of running times of various types of graphs with varying number of threads (note different y-axis scales).

Algo.		Framework	TW	SW	OK	WK	LJ	PK	US	GR	RM	UF
BC	20	Galois	2814	3502	1733	54776	2962	2331	>95 Hrs	>12 Hrs	1006	261
		StarPlat	**442**	**13**	**681**	**244**	**197**	**52**	**917**	**353**	**437**	**7**
	80	Galois	12375	14043	6999	>12 Hrs	11320	9035	>12 Hrs	>12 Hrs	2479	1024
		StarPlat	**2102**	**49**	**2705**	**978**	**764**	**194**	**3741**	**1489**	**975**	**29**
	150	Galois	NA	NA	NA	NA	NA	NA	NA	NA	NA	N
		StarPlat	**3816**	**94**	**4973**	**1829**	**1498**	**382**	**6782**	**2713**	**1442**	**55**
PR		Galois	398	761	314	376	551	479	257	296	244	26
		StarPlat	**57**	OOM	**23**	**11**	**8**	**2**	**2**	**0.3**	**44**	
SSSP		Galois	**33**	11	**30**	63	**27**	24	**1933**	82	**20**	4
		StarPlat	152	**0.4**	104	**35**	47	**15**	2302	**27**	70	**1**
TC		Galois	**71275**	**436**	**327**	**884**	**174**	**19**	7	9	**1238**	**1**
		StarPlat	>24 Hrs	>72 Hrs	64114	>24 Hrs	5741	2420	**4**	**0.6**	>24 Hrs	3

Table 5: StarPlat's MPI code performance comparison against Galois (96 processes). All times are in seconds. BC is run with the number of sources as {20, 80, 150}. StarPlat goes out-of-memory on SW in PR due to double buffering.

5.2. MPI

The comparison of execution time of StarPlat's MPI code against the distributed Galois framework for four algorithms is given in Table 5. The execution time does not include the graph reading and distribution time. OpenMPI 3.1.6 is used for execution and the algorithms are executed with 96 number of processes.

Betweenness Centrality. From the table, we can see that, StarPlat's BC code takes much lesser time for execution compared to the Galois framework for all the ten graphs. When 20 number of sources are used, the execution of Galois code for two graphs TW and GR is not completed even after 96 hours and 12 hours respectively. Similarly, for 80 number of sources, the Galois code does not give result for three graphs WK, US and GR till 12 hours of execution. The execution of Galois code for all the 10 graphs with 150 number of sources is not tried as it is expected to take much more time than StarPlat's BC code for all the ten graphs based on the execution time for 20 and 80 number of sources.

PageRank. StarPlat-generated PR code outperforms the Galois PR code for all the graphs except SW. StarPlat's PR on SW goes out of memory due to double-buffering. We are currently investigating a way to fix it.

Single-Source Shortest Paths. From the comparison of execution time

Algo.		Framework	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR
BC	1	LonestarGPU	-	-	-	-	-	-	-	-	-	-
	1	Gunrock	2.122	4.237	0.525	0.535	0.548	0.317	**2.811**	**1.750**	1.238	0.944
	1	StarPlat	**0.002**	**0.004**	**0.149**	**0.153**	**0.078**	**0.029**	17.656	6.359	**0.225**	**0.079**
	20	StarPlat	6.992	2.279	2.762	3.014	1.298	0.534	369.701	126.485	2.949	1.593
	80		28.179	9.332	11.331	12.050	4.886	1.907	1444.656	518.968	6.509	6.372
	150		55.548	MLE	21.241	27.271	9.609	3.794	2636.453	978.758	9.912	11.957
PR		LonestarGPU	-	**0.240**	0.363	**0.104**	**0.225**	**0.240**	**0.832**	**0.294**	**0.240**	**0.240**
		Gunrock	15.230	36.910	2.430	2.460	2.952	1.085	13.345	6.499	9.170	5.487
		StarPlat	**4.081**	7.112	**0.256**	1.780	1.300	0.257	3.420	0.679	0.891	0.257
SSSP		LonestarGPU	0.045	0.077	0.217	0.058	0.084	0.037	**0.162**	**0.091**	0.129	0.183
		Gunrock	2.272	4.057	0.616	0.556	0.562	0.311	1.283	1.140	1.034	0.915
		StarPlat	**0.001**	**0.002**	**0.078**	**0.044**	**0.027**	**0.012**	1.667	0.695	**0.120**	**0.028**
TC		LonestarGPU	-	31.990	2.998	2.771	**0.110**	**0.039**	11.874	5.695	**1.270**	0.499
		Gunrock	**67.718**	7.369	**0.843**	**0.997**	0.850	0.404	1.490	0.712	3.200	1.040
		StarPlat	10540.002	**1.410**	46.700	4.009	3.006	0.655	**0.001**	**0.001**	824.620	**0.034**

Table 6: StarPlat's CUDA code performance comparison against LonestarGPU and Gunrock. All times are in seconds. LonestarGPU does not have BC implemented and fails to load the largest graph TW.

of SSSP code of StarPlat with Galois, we can observe that, Galois performs better for 5 graphs TW, OK, LJ, US and RM while StarPlat performs better for the remaining five graphs. The number of iterations taken by StarPlat is fewer than that by Galois. This is due to multiple levels of distance update within the local nodes of a process in a single outer loop iteration.

Triangle Counting. The TC code generated by StarPlat is less efficient as it iterates through neighbours of each node multiple times and check for neighbourhood among these nodes. Due to this, StarPlat's TC code takes much more time than Galois. For road graphs US and GR, StarPlat's TC takes lesser time than Galois code. For some graphs such as TW, SW, WK and RM StarPlat's code did not give result even after 24 hours.

5.3. CUDA

Table 6 presents the absolute running times of the four algorithms on our ten graphs for the three frameworks: LonestarGPU, Gunrock, and StarPlat. The running times include CPU-GPU data transfer.

Betweenness Centrality. StarPlat-generated code outperforms Gunrock's library-based code on eight out of ten graphs. On the two road

networks, Gunrock is considerably better than StarPlat. We use CUDA's `grid.synchronization` to our benefit, whereas Gunrock relies heavily on this bulk-synchronous processing. Gunrock's Dijkstra's algorithm works very well for road networks. On the other hand, on social and random graphs, our implementation fares better. We also illustrate performance with multiple sources of different sizes (20, 80, and 150). Except for soc-sinaweibo, StarPlat-generated code is on par with or better than the other frameworks. Finally, unlike Gunrock and LonstarGPU, StarPlat has the provision to execute BC from a set of sources.

PageRank. We observe that the three frameworks have consistent relative performance, with hand-crafted LonestarGPU codes outperforming the other two and StarPlat outperforming Gunrock. StarPlat exploits the double buffering approach to read the current PR values and generate those for the next iteration. This separation reduces synchronization requirement during the update, but necessitates a barrier across iterations. LonestarGPU uses an in-place update of the PR values and converges faster.

Single-Source Shortest Paths. Gunrock uses Dijkstra's algorithm for computing the shortest paths using a two-level priority queue. We have coded a variant of the Bellman-Ford algorithm in StarPlat. Hence, the comparison may not be most appropriate. But we compare the two only from the application perspective – computing the shortest paths from a source in the least amount of time. LonestarGPU and StarPlat outperform Gunrock on all the ten graphs. Between LonestarGPU and StarPlat, there is no clear winner. They, in fact, have quite similar execution times.

Triangle Counting. Unlike other three algorithms, TC is not a propagation based algorithm. In addition, it is characterized by a doubly-nested loop inside the kernel (see Figure 20). Another iteration is required for checking edge (Line 7) which can be implemented linearly or using binary search if the neighbors are sorted in the CSR representation. Due this variation in the innermost loop, the time difference across various implementations can be pronounced, which we observe across the three frameworks. Their performances are mixed across the ten graphs, and no one emerges as an overall winner.

6. Related Work

We divide the relevant related work based on the target backend in the next three subsections, and discuss generic frameworks in Section 6.4. .

6.1. OpenMP

Ligra [**?**] is a lightweight graph processing framework that is specific for shared-memory parallel/multi-core machines, which makes graph traversal algorithms easy to write. The framework has two very simple routines, one for mapping over edges and one for mapping over vertices. The routines can be applied to any subset of the vertices, which makes the framework promising for many graph traversal algorithms operating on subsets of the vertices. In abstraction, a vertex subset is maintained as a set of integer labels for the included vertices, and the routine for mapping over vertices applies the user-supplied function to each integer.

A lightweight infrastructure for graph analytics, Galois [**?**] supports a more general programming model which is more expressive than restrictive. The infrastructure supports autonomous scheduling of fine-grained tasks with application-specific priorities and a library for scalable data structures. Existing DSLs can be implemented on top of the Galois system to achieve high throughput. One can think of a Galois backend for StarPlat.

Green-Marl [**?**] is a DSL for shared memory graph processing. Programs depending on BFS/DFS traversals can be written concisely with the corresponding constructs in the language. However, for other graph algorithms, the user has to define the iteration over vertices or edges explicitly. So, the algorithmic description can be written intuitively using the constructs provided by the language, while exposing the inherent data-level parallelism. StarPlat borrows a few ideas from the language, and also quantitatively compares its performance against Green-Marl, Ligra, and Galois.

GraphIt [**?**] is a DSL for graph analysis that achieves high performance by enabling programmers to easily find the best combination of optimizations for their specific algorithm and input graph. It essentially separates computation from scheduling. The algorithms are specified using an algorithm language, and performance optimizations are specified using a separate scheduling language.

6.2. MPI

D-Galois, a distributed graph processing system, was formed by interfacing Gluon, a communication optimization technique with Galois shared

memory graph processing system [**?**]. We compare against Gluon in our experiments.

Pregel and Giraph graph processing frameworks were inspired by the BSP programming model [**?** **?**]. Pregel decomposes the graph processing computation as a sequence of iterations in which the vertices receive data sent in the previous iteration, changes its state, and sends messages to its out neighbors which will be received in the next iteration. GPS provided additional features such as enabling global computation, dynamic graph repartitioning, and repartitioning of large adjacency lists to the Pregel framework [**?**].

A compiler automatically generates Pregel code from a subset of programs called Pregel canonical written using Green-Marl DSL [**?**]. It also applies certain transformations such as edge flipping, dissecting nested loops, translating random access, transforming BFS and DFS traversals to convert non-Pregel canonical pattern into Pregel canonical. Another compiler named DisGCo can translate all the Green-Marl programs to equivalent MPI RMA programs [**?**]. It handled the challenges such as syntactic differences, differences in memory view, intermixed serial code with parallel code, and graph representation while translating a Green-Marl program to MPI RMA code.

The specification and implementation of morph graph algorithms and non-vertex centric graph algorithms for heterogeneous distributed environments was provided by the DH-Falcon DSL and its compiler [**?**]. A hybrid approach that uses both dense and sparse mode processing by utilizing pull- and push-based computation approaches was provided by a distributed graph analytics scheme Gemini [**?**]. Additionally, it used optimized graph representation and partitioning methods for intra-node as well as inter-node load balancing.

GraphLab provides a fault-tolerant distributed graph processing abstraction that can perform a dynamic, asynchronous graph processing for machine learning algorithms [**?**]. Another parallel distributed graph processing abstraction for processing power law graphs with a caching mechanism and a distributed graph placement mechanism was introduced by PowerGraph [**?**]. GraphChi supports asynchronous processing of dynamic graphs using a sliding approach on the disk-based systems [**?**].

6.3. CUDA

Gunrock [**?**] is a graph library which uses data-centric abstractions to perform operations on edge and vertex frontier. All Gunrock operations are bulk-synchronous, and they affect the frontier by computing on values within

it or by computing a new one, using the following three functions: *filter*, *compute*, and *advance*. Gunrock library constructs efficient implementations of frontier operations with coalesced accesses and minimal thread divergence.

LonestarGPU [**?**] is a collection of graph analytic CUDA programs. It employs multiple techniques related to computation, memory, and synchronization to improve performance of the underlying graph algorithms. We quantitatively compare StarPlat against Gunrock and LonestarGPU.

Medusa [**?**] is a software framework which eases the work of GPU computation tasks. Similar to Gunrock, it provides APIs to build upon, to construct various graph algorithms. Medusa exploits the BSP model, and proposes a new model EVM (Edge Message Vertex), wherein the local computations are performed on the vertices and the computation progresses by passing messages across edges.

CuSha [**?**] is a graph processing framework that uses two graph representations: G-Shards and Concatenated Windows(CW). G-Shards makes use of a recently developed idea for non-GPU systems that divides a graph into ordered sets of edges known as shards. In order to increase GPU utilisation for processing sparse graphs, CW is a novel format that improves the use of shards. CuSha makes the most of the GPU's processing capability by processing several shards in parallel on the streaming multiprocessors. CuSha's architecture for parallel processing of large graphs allows the user to create the vertex-centric computation and plug it in, making programming easier. CuSha significantly outperforms the state-of-the-art virtual warp-centric approach in terms of speedup.

MapGraph [**?**] is a parallel graph programming framework which provides a high level abstraction which helps in writing efficient graph programs. It uses SOA(Structure Of Arrays) to ensure coalesced memory access. It uses the dynamic scheduling strategy using GAS(Gather-Apply-Scatter) abstraction. Dynamic scheduling improves the memory performance and dispense the workload to the threads in accordance with degree of vertices.

6.4. Generic Frameworks

Kakkos 3 [**?**] is a programming model in C++ for writing performance portable applications targeting all major HPC platforms (CUDA, HIP, SYCL, HPX, OpenMP and C++ threads as backend). It provides abstractions for both parallel executions of code and data management. However, Kokkos kernels are used mainly on matrix-based operations or algorithms, and it targets coloring algorithms only recently, and many other complex

algorithms are not supported (as of date). Philosophy-wise, Kokkos and StarPlat have similarity, while the former is generic and is not a DSL.

Graphit [**?**] adds support for code generation on GPUs along with CPUs in their recent version. It expands the optimization space of GPU graph processing frameworks with a novel GPU scheduling language, and a compiler which enables various optimizations such as combining load balancing, edge traversal direction, active vertex set creation, active vertex set processing ordering, and kernel fusion.

Julia [**?**] is a high-level, dynamic programming language designed for high performance. It is well suited for numerical analysis and computational science. It supports concurrent, parallel, and distributed computing (with and without MPI). Circle [**?**] is a new C++ 20 compiler. It extends C++ by adding many novel language features. Circle is a heterogeneous compiler that supports GPU compute and shader programming as well. With Circle, one can write a device-portable GPGPU code with Vulkan compute (using real pointers into physical storage buffers), a powerful feature only implemented by the Circle compiler. Carbon [**?**] is an experimental successor language of C++ in development. One of its main goals is to support modern OS platforms, hardware architectures, environments, and fast & scalable development (similar to Python and Rust). Odin [**?**] is again a general-purpose programming language with distinct typing built especially for high-performance, modern systems and data-oriented programming. It is under development and supports structure of arrays (SoA) data types and array programming. Chapel [**?** **?**] is a modern parallel programming language aimed at portability and scalability. The primary goal of Chapel is to support general parallel programming and also make parallel programming at a scale far more productive.

The recent sprouts of the above languages motivate the need for a simple language that is portable across different parallel programming platforms and architectures without compromising the productivity of the users, irrespective of the domain they work for.

In a similar spirit, Intel [**?**] and Nvidia [**?**] are venturing into heterogeneous programs which converts simple C++ program to parallel programs running on various architectures (multi-cores, GPUs, FPGAs and even Arm Servers). However, such programs are parallelized only if the program uses STL algorithms. Their compiler replaces the standard C++'s STL algorithms with the corresponding parallel versions based on the target. Intel OneAPI (which includes DPC++ and TBB) uses C++ STL, Parallel STL

(PSTL), Boost Compute, and SYCL for parallelization. Nvidia's `nvc++` uses a different linker for the target processors using command-line arguments. `nvc++` supports ISO C++17, and targets GPU and multicore CPU programming with C++17 parallel algorithms, OpenACC, and OpenMP.

We observe that Gunrock's Essentials-cpp and Essentials [**?**] are along similar lines and use modern C++, `std` parallelism constructs and lambdas built with their bulk-synchronous-asynchronous, data-centric abstraction model. They currently implement BFS and SSSP.

7. Conclusion

We presented StarPlat, a DSL for implementing graph analytic algorithms. The language model provides various constructs for expressing graph problems at a high level without the complexities of implementing the commonly occurring patterns from scratch. This also facilitates providing optimized solutions both in terms of memory and time for these patterns. Currently, we support development across multicore, distributed, and manycore backends. We discussed the translation of StarPlat's constructs for various backends. With a range of large graphs and four different algorithms, we experimentally validated that the performance of the generated code is comparable to the hand-tuned and generated implementations of existing frameworks and libraries.

While there are multiple software solutions available in the form of programming languages and libraries for the development of graph algorithms, an important next step is to build solutions that are versatile to support multi-platform development. StarPlat is an effort in this direction. The future direction of StarPlat is to improve the portability of the solution to support OpenACC and OpenCL backends, and to enable automatic code generation for heterogeneous architectures.

Acknowledgments

We gratefully acknowledge the use of the computing resources at HPCE, IIT Madras. This work is supported by India's National Supercomputing Mission grant CS1920/1123/MEIT/008606.

Appendix A. Graph Algorithms in StarPlat

StarPlat specifications for Betweenness Centrality, PageRank, and Triangle Counting are shown below in Figures 18, 19, and 20 respectively. In addition, Figure 21 shows pull-based SSSP.

```
function computeBC(Graph g, propNode<float> BC,
    SetN<g> sourceSet) {
  g.attachNodeProperty(BC = 0);
  for (src in sourceSet) {
      propNode<double> sigma;
      propNode<float> delta;
      g.attachNodeProperty(delta = 0);
      g.attachNodeProperty(sigma = 0);
      src.sigma = 1;

      iterateInBFS(v in g.nodes() from src) {
          for (w in g.neighbors(v)) {
              w.sigma = w.sigma + v.sigma;
      }   }

      iterateInReverse(v != src) {
          for (w in g.neighbors(v)) {
              v.delta = v.delta + (v.sigma /
                  w.sigma) * (1 + w.delta);
          }
          v.BC = v.BC + v.delta;
}   }   }
```

Fig 18: BC computation in StarPlat

```
function computePR(Graph g, float beta, float
    delta, int maxIter, propNode<float> pageRank)
    {
  float num_nodes = g.num_nodes();
  propNode<float> pageRank_nxt;
  g.attachNodeProperty(pageRank = 1 /
      num_nodes);
  int iterCount = 0;
  float diff;
```

```
    do {
        diff = 0.0;

        forall(v in g.nodes()) {
            float sum = 0.0;

            for (nbr in g.nodes_to(v)) {
                sum = sum + nbr.pageRank /
                    g.count_outNbrs(nbr);
            }

            float val = (1 - delta) / num_nodes
                + delta * sum;
            diff += val - v.pageRank;
            v.pageRank_nxt = val;
        }

        pageRank = pageRank_nxt;
        iterCount++;

    } while ((diff > beta) && (iterCount <
        maxIter));
}
```

Fig 19: PR computation in StarPlat

```
function computeTC(Graph g) {
    int triangle_count = 0;

    forall(v in g.nodes()) {
        forall(u in g.neighbors(v).filter(u <
            v)) {
            forall(w in g.neighbors(v).filter(w
                > v)) {
                if (g.is_an_edge(u, w)) {
                    triangle_count += 1;
}   }   }   }   }
```

Fig 20: TC computation in StarPlat

```
1  function computePullSSSP (Graph g, node src) {
2      propNode<int> dist;
3      propNode<bool> modified;
4      g.attachNodeProperty(dist = INF, modified =
           False);
5      src.modified = True;
6      src.dist=0;
7      bool finished = False;
8      fixedPoint until (finished: !modified) {
9          forall (v in g.nodes()) {
10             forall (nbr in
                   g.nodes_to(v).filter(modified ==
                   True) {
11                 edge e = g.getEdge(v, nbr);
12                 <v.dist, v.modified> =
                       <Min(v.dist, nbr.dist +
                       e.weight), True>;
13 }   }   }   }
```

Fig 21: SSSP-Pull computation in StarPlat