Early Experience with Integrating Charm++ Support to Green-Marl Domain-Specific Language

Alexander Frolov
JSC NICEVT
Moscow, Russia
frolov@nicevt.ru

Abstract

The paper presents the implementation of the code generation mechanism in the Green-Marl domain-specific language compiler targeted at the Charm++ framework. Green-Marl is used for parallel static graph analysis and adopts imperative shared memory programming model, while Charm++ implements a message-driven execution model. The description of the graph representation in the generated Charm++ code, as well as of the translation of the common Green-Marl constructs to Charm++, is presented. The evaluation of the typical graph algorithms (Single-Source Shortest Path, Connected Components, and PageRank) showed that the Green-Marl programs translated to the Charm++ have almost the same performance as their native implementations in Charm++.

Keywords: domain-specific language, data-driven computation models, graph computations

1 Introduction

A parallel static graph analysis on high-performance computing systems (supercomputers) is one of the recent application domains which is characterized by a domination of irregular data processing rather than massive bulks of floating point operations which are common in other scientific applications used for supercomputers. The key challenges of parallel graph processing are discussed in [1].

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: V. Voevodin, A. Simonov (eds.): Proceedings of the GraphHPC-2017 Conference, Moscow State University, Russia, 02-03-2017, published at http://ceur-ws.org.

The paper presents the results of the efforts to integrate a Charm++ support to the Green-Marl compiler. Green-Marl is an open-source domain-specific programming language (DSL) designed for static graph processing. Charm++ is an asynchronous message-driven execution model developed to execute parallel programs on multiprocessor computing systems. Charm++ relies on its runtime system to schedule computation, as well as perform dynamic load balancing.

For the translation of Green-Marl programs to Charm++ a code generation module has been developed in the Green-Marl compiler that uses the existing internal representation of a Green-Marl program for Charm++ code generation. For performance evaluation we used three well-known graph problems: Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank. We compared the performance of the generated code to the hand-coded reference implementations.

In the next section an overview of research efforts on asynchronous computation models and graph-specific DSLs is presented. In section 3 a brief description of the Green-Marl DSL is presented, section 4 contains an overview of the Charm++ programming model. Section 5 shows the main technical aspects of translating Green-Marl programs to Charm++. Section 6 presents results of the performance evaluation. The final section contains conclusion and future work.

2 Related Work

A usage of common approaches to the development of parallel graph applications for massive parallel, distributed memory systems such as MPI or Shmem in combination with OpenMP is complicated due to its limited support for expressing irregular parallelism and implicit orientation on statically balanced parallel bulk-synchronous problems, and, therefore, the complexity of implementing parallel graph algorithms

falls on the application developer. In contrast, the asynchronous data-flow computational models enable almost transparent mapping of the graph algorithms. The examples of such models which are built on top of *asynchronous active messages* concept are Charm++ [2, 3], Active Pebbles [4, 5], HPX [6]. The researched next generation parallel programming languages such as Chapel [7, 8, 9] and X10 [10, 11] also support active messages as one of the base principles of their programming models.

Even better productivity of the graph application development can be achieved with a usage of high-level domain-specific languages which enable to shift a major part of work on the control of parallel execution, including task spawning, communication, synchronization, etc. to the compiler and dynamic load balancing and a failure resiliency – to the runtime system. There are few DSLs specifically developed for a parallel static graph analysis such as Green-Marl [12,13,14], OptiGraph, Elixir [15], and Falcon [16]. However, none of them supports distributed memory massive-parallel HPC systems, rather the researchers focused their work on the shared memory systems, GPUs, and Big Data platforms.

3 Green-Marl

Green-Marl [12, 13, 14] is an open-source domain-specific programming language designed for a parallel static graph analysis in Stanford University (Pervasive Parallel Laboratory, PPL). The Green-Marl compiler supports translation to the following parallel programming models: OpenMP and Pregel [17]. The OpenMP backend allows to run Green-Marl programs on shared memory multi-processor systems, while the Pregel backends (there are two slightly different backends with Pregel implementations: GPS and Giraph) enable to use Green-Marl for distributed massive-parallel computing systems.

Green-Marl supports special types for declaration of graphs (`Graph`), vertices (`Node`), and edges (`Edge`), as well as declaration of vertex properties (`N_P<`*type*`>`) and edges properties (`E_P<`*type*`>`).

Besides the `While` and `Do-While` constructs used for defining sequential loops and `If` and `If-Else` for branches, Green-Marl provides statements for parallel execution such as `For` and `Foreach` loops. The semantics of the parallel loops assumes that all computations should be finished before the first operation after the loop is executed. However, the order of the iteration execution is not defined.

Green-Marl supports the following reduction operators `+=`, `*=`, `max=`, `min=`, `&&=`, `||=`. These operators can be used in parallel loops to produce reduction assignments.

```
1  Procedure sssp(G:Graph, dist:N_P<Int>, len:E_P<Int>,
2                 root: Node)
3  {
4    N_P<Bool> updated;
5    N_P<Bool> updated_nxt;
6    N_P<Int>  dist_nxt;
7
8    Bool fin = False;
9    G.dist = (G == root) ? 0 : +INF;
10   G.updated = (G == root) ? True: False;
11   G.dist_nxt = G.dist;
12   G.updated_nxt = G.updated;
13
14   While(!fin) {
15     fin = True;
16     Foreach(n: G.Nodes)(n.updated) {
17       Foreach(s: n.Nbrs) {
18         Edge e = s.ToEdge();
19         <s.dist_nxt; s.updated_nxt>
20           min= <n.dist + e.len; True, n>;
21       }
22     }
23     G.dist = G.dist_nxt;
24     G.updated = G.updated_nxt;
25     G.updated_nxt = False;
26     fin = ! Exist(n: G.Nodes){n.updated};
27   }
28 }
```

Figure 1: SSSP (Bellman-Ford algorithm) in Green-Marl

In Figure 1 an implementation of the Bellman-Ford algorithm for finding the shortest paths from the specified vertex to other vertices in the graph is presented. The program consists of a single procedure (`sssp`) with four parameters: `G` is a graph to be analyzed, `dist` is a vertex property storing the distance from the source vertex, `len` is an edge property storing the edge length (or weight), and `root` is a source vertex.

In the initialization step (lines 8-12) `dist` is set to `+INF` for all vertices except the root vertex which `dist` is set to zero. Also, the root vertex is marked by setting its `update` property to `true` which signifies that the vertex will be processed in the first iteration of the `While` loop (lines 14-17). Then the `While` loop is executed. It completes when `fin` is set to `true` that is there is no more updated vertices in the graph. In each iteration of the `While` loop a parallel `Foreach` loop is executed (lines 16-22), which scans vertices and for each previously updated vertex (i.e. its `updated` property is set to `true`) checks neighbour vertices by executing a relaxation operation: if $dist[v] + weight(v,u) < dist[u]$ for the (v,u) edge then $dist[u]$ is assigned to $dist[v] + weight(v,u)$ (lines 19-20). The number of iterations of the `While` loop is restricted to $|V| - 1$ (upper bound).

Therefore, Green-Marl is an imperative domain-specific programming language designed for a parallel static graph analysis. Green-Marl allows to significantly simplify the development of parallel graph applications by its builtin specialized high-level abstractions as well as the diverse set of the compiler

optimizations. However, it still lacks the support of the translation to effective programming platform for HPC systems with distributed memory.

4 Charm++

Charm++ [2, 3] is a parallel programming framework which is based on the object-oriented asynchronous message-driven execution model. Charm++ designed and developed in Illinois University of Urbana-Champaign (Parallel Programming Laboratory, PPL). As a framework, Charm++ includes the following core components: a compiler used to translate application's objects interfaces to the C++ code and a runtime system which drives the application execution process.

In Charm++ a program consists of a set of *chares*, implemented as C++ objects, that have an interface of predefined methods (*entry* methods) which can be used to transfer data between chares and initiating asynchronous computations, that is caller thread will proceed its execution. A mapping of chares to process elements (PEs) is done statically by the Charm++ runtime (by default) or by application developer. Typically, PE can be regarded as a CPU core (or an operating system process assigned to a CPU core). Additionally to stand-alone chares, single-dimensional and multi-dimensional chare arrays are supported.

For addressing chares special proxy objects are used that provide an abstraction of global address space and enable the Charm++ runtime to manage location and distribution of chares transparently for application. This allows to use dynamic load balancing by migrating chares from more loaded nodes to less loaded by the runtime system. A migratability of chares is another base concept of the Charm++ programming model.

The following properties of the Charm++ model: first, an entry method can only access data which belong to the chare that it is called on, and, second, that only single entry method can be executed on a chare at a time, guarantee the atomicity of modifications of chare's data in Charm++ that significantly simplifies application development.

5 Porting Green-Marl Compiler to Charm++

5.1 Compiler Overview

The Green-Marl compiler performs a source-to-source translation from Green-Marl to one of the possible equivalent representations: a sequential C++ code, a parallel code for shared memory systems in C++ with OpenMP pragmas, a parallel code for distributed memory systems in Java on top of Pregel [17] (namely, one of the two Java based implementations of Pregel: GPS [18] or Giraph [?, 19]), which is a vertex-centric programming model built on the Bulk Synchronous Parallel (BSP) execution model [20].

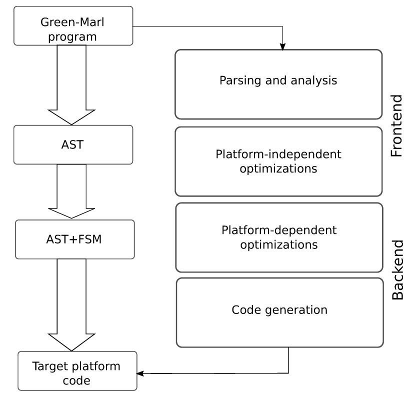


Figure 2: Main stages of Green-Marl compiler

The general scheme of Green-Marl is shown in Figure 2. The main stages of translation are the following: a lexical and syntax analysis (type checking, a syntactic sugar expansion etc.), platform independent optimizations (such as loop splitting and merging, loop-invariant code motion etc.), platform specific optimizations and a final code generation.

An internal representation of the program in the Green-Marl compiler includes an abstract syntax tree (AST) and a control-flow graph (a finite state machine, FSM), the nodes of the graph (or extended basic blocks, EBB) correspond to linear blocks of code in the translated program. Each EBB contains information about local variables, incoming and outgoing dependencies, code statements, etc.

The FSM states can be of two types: sequential (SEQ) and parallel (PAR). The sequential nodes correspond to the code blocks in which the statements should be executed in the strict order. The parallel nodes correspond to the code blocks which suggest execution for each vertex of the graph in parallel (`Foreach` and `For` loops).

5.2 Charm++ Code Generator

The developed Charm++ code generator is based on the GPS generator developed for translating Green-Marl programs to the Pregel execution model, a full description of the GPS generator can be found in [14]. However, the Charm++ programming model is more general than Pregel, which allows, first, to relatively easy implement the Pregel model and, second, extends the possibilities of Green-Marl translator, that is increase the variety Green-Marl programs which can be translated to Charm++.

At the same time, to prove the concept of the possibility to translate Green-Marl programs to Charm++ it was sufficient to implement the last stage of the com-

```
Procedure count(G:Graph, age:N_P<Int>, root: Int)
{
  Int S = 0;
  Int C = 0;
  Foreach (n : G.nodes) {
    If (n.age < K) {
      S += n.age;
      C += 1;
    }
  }
  Float val = (C == 0) ? S / (float) C;
}
```

(a)

```
module count {
  message __ep_state1_msg;
  readonly CProxy_count_master master_proxy;
  readonly CProxy_count_vertex vertex_proxy;
  chare count_master {
    entry count_master(const CkCallback & cb);
    entry void do_count(int K);
    entry [reductiontarget] void __reduction_S (int S);
    entry [reductiontarget] void __reduction_C (int C);
    entry void __ep_state0();
    entry void __ep_state1();
    entry void __ep_state2();
  }; // count_master
  array [1D] count_vertex {
    entry count_vertex();
    entry void __ep_state1(__ep_state1_msg *msg);
    entry void add_edge(const count_edge &e);
  }; // count_vertex
}; // count
```

(b)

```
class count_master: public CBase_count_master {
  public:
    void __reduction_S (int S) { this->S = S; }
    void __reduction_C (int C) { this->C = C; }
    void __ep_state0() {
      S = 0;
      C = 0;
      thisProxy.__ep_state1();
    }
    void __ep_state1() {
      __ep_state1_msg *_msg = new __ep_state1_msg();
      _msg->K = K;
      vertex_proxy.__ep_state1(_msg);
      CkStartQD(CkIndex_count_master::__ep_state2(),
          &thishandle);
    }
    void __ep_state2() {
      val = (C == 0)?((float)S):(0 / ((float)C));
      done_callback.send();
    }
  private:
    CkCallback done_callback;
    int S, C, K;
    float val;
}; // count_master
class count_vertex: public CBase_count_vertex {
  public:
    struct vertex_properties {
      int age;
    };
  public:
    void __ep_state1 (__ep_state1_msg *msg) {
      int K = msg->K;
      delete msg;
      int S, C;
      if (this->props.age < K) {
        S = this->props.age;
        contribute(sizeof(int), &S,
          CkReduction::sum_int,
          CkCallback(CkReductionTarget(
            count_master, __reduction_S), master_proxy));
        C = 1;
        contribute(sizeof(int), &C,
          CkReduction::sum_int,
          CkCallback(CkReductionTarget(
            count_master, __reduction_C), master_proxy));
      }
    }
  private:
    std::list<struct count_edge> edges;
    struct vertex_properties props;
}; // count_vertex
```

(c)

Figure 3: Example of Green-Marl program (a) and its generated code in Charm++ (b, c)

piler – code generation from the internal program representation (IR). Further, a description of the translation methods is presented for main Green-Marl constructs. As an example, a program for calculating a mean age of all registered users of a social network whose age is less than K. The source code of the Green-Marl program and the generated code in Charm++ are shown in Figure 3.

Graph representation in the generated code

For a graph representation in the generated code in Charm++ the most simple and natural approach has been chosen: vertices of the graph are mapped to the chares (i.e. the elements of the chare array), which are defined in the code by the *name*_vertex class, *name* is a name of the translated Green-Marl procedure.

The vertex_properties data structure is used for storing vertex attributes, while edges is used for – an edge list of the vertex, each edge is a pair of a neighbour identifier and its attributes (edge_properties). As a 1-dimensional chare array (array [1D]) is used for storing graph vertices then the vertices are distributed in contiguous blocks over parallel processes (or cores) by the Charm++ runtime system.

In the example (see Figure 3) the vertices are represented by the count_vertex class (Figure 3 (b), lines 14–18, 3, lines 26–52). The vertices have a single attribute age, which is stored in the vertex_properties class. As the attributes for edges are not defined in the Green-Marl program then edge_properties does not have any field.

FSM construction

As it has already been mentioned, after the control-flow analysis the Green-Marl compiler creates a finite state machine (FSM). For managing the execution of FSM a special chare (*name*_master) is created, each state of FSM is mapped to one of the entry methods: __ep_state_*i*, where *i* is a number of the FSM state. The program starts with the execution of __ep_state_0 then the states are switched according to FSM. When PAR states are executed, an appropriate call is initiated to all chares of the *name*_vertex class (i.e. array elements). Then master (*name*_master) waits until all computation are done (by using a quiescence detection) and then an entry method correspondent to the next FSM state is executed. The process continues until the terminal (final) state is called.

In the example FSM consists of three states (see Figure 4) which correspond to the following entry methods of the count_master: __ep_state_0, __ep_state_1 and __ep_state_2. As the state 1 is parallel, then it has a correspondent entry method in the count_vertex class (__ep_state_1). In the terminal

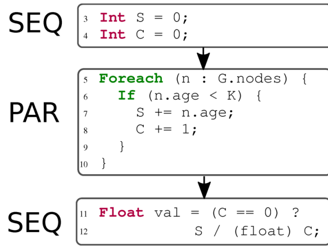


Figure 4: Fragment of Green-Marl program and its control-flow graph

state (state 2) the callback (`done_callback`) is triggered to move control flow to the boiler-plate code (or code of the Green-Marl program loader).

Global variables and reduction

In Green-Marl, all variables can be of two types: global and local (for vertices). The global variables are declared in the sequential code blocks and can be accessed from the parallel code blocks. All global variables are stored in the *name*`_master` class as its member variables. In case of a read access to the global variable from the `Foreach` loop, its value is passed as a parameter of the correspondent entry method to all vertices (*name*`_vertex`). In case of a write access then the correspondent entry method of *name*`_master` is called, however, when write accesses operations are performed to the same global variable from the multiple vertices then race conditions occur and the behaviour is undefined.

The Green-Marl compiler applies the `contribute` operation supported in Charm++ to implement reductions which can be used in Green-Marl programs. In Charm++ there are several supported reductions operations which can be applied to the chare array elements, the mechanism suggests that a specified entry method (of the specified chare) is called when reduction is completed which has to be marked with `reductiontarget` keyword and has a single parameter which will store the result of the reduction.

In the example (see Figure 3) three global variables are used (`S`, `C`, `K`). The `K` value is used inside the `Foreach` loop, therefore `__ep_state_2` in `count_vertex` receives `K` as one of the parameters of the `__ep_state2_msg` structure, which is a Charm++ message (`message`). `S` and `C` are used for the reduction results (lines 7, 8). In the generated code there are two methods in the `count_master` class: `__reduction_S` (line 3) and `__reduction_` (line 4) which are used for passing the reduction results to the `S` and `C` variables.

In conclusion of this section we can say that for implementing Charm++ support in the Green-Marl compiler it was sufficient to add a new code generation module to the existing backend chain for GPS code generation. However, as it has been mentioned, Charm++ has significantly more flexibility and generality than Pregel that is why it provides wider possibilities to the compiler as well as to the domain-specific language itself. Using of these possibilities is the topic of further research.

6 Performance Evaluation

For benchmarking a 32-node Angara-K1 cluster has been used, we used its 24-node segment with dual 6-core Intel Xeon E5-2630 processors and 64 GB of memory in each node. The nodes are connected by a custom 4D-torus Angara interconnect [21]. Eight CPU cores have been used per each node in the test runs to keep total number of processes equal to power of two, however, it may not be necessary in the general case.

The benchmarks used for the performance evaluation include: Single-Source Shortest paths (SSSP), Connected Components (CC), and PageRank. We compared the performance of Green-Marl programs to their hand-written Charm++ invariants (or reference implementations).

We used two types of synthetic graphs: RMAT [22] and Random. RMAT graphs have been designed to simulate many real-world graphs, which are characterized by a power law of degree distribution (for example social networks, etc.). For RMAT graphs we used generator from the Graph500 test. Random graphs have a random uniform distribution of edges over graph vertices.

The performance results are shown in Figure 5. In all plots the graph size is 2^{22} vertices. For SSSP and PageRank directed graphs have been used, while for CC – undirected.

For SSSP (as well as for CC) the two different reference implementations in Charm++ have been used. The first is `sssp-async` (`cc-async`) and it implements a fully asynchronous algorithm which employs a label updating approach. For SSSP the updated label is a distance from the root vertex (`dist`) in each vertex, while for CC – identifier of connected component (`CC`). The computation process continues until any update is possible. When there is no more updates in the graph then the algorithm finishes, this is controlled by a quiescence detection mechanism, supported in Charm++. Another reference implementation `sssp-adapt` (`cc-adapt`) is similar to the first except that a global synchronization is used to control label propagation (the new labels are sent only to the neighbour vertices of the front and then a global synchronization is performed). In Green-Marl implementations of the tests it is achieved by the outer loop `While`.

As can be seen from Figure 5, for SSSP and CC

the performance of Green-Marl implementations is very close to the performance of the reference implementation with a partially synchronized algorithm (`sssp-adapt` for SSSP and `cc-adapt` for CC). For CC the adapted reference implementation shows even better performance than asynchronous one. It can look unexpected, but the following reasons of such behaviour are possible: first, while the partially synchronized algorithms (`sssp-adapt` and `cc-adapt`) restrict the amount of available parallelism, at the same time the number of active messages sent during execution of the programs is also significantly restricted, thus resulting in less overheads from the runtime system and less memory consumption, and, second, the fully asynchronous implementations (`sssp-async` and `cc-async`) generate many speculative computations (speculative wavefronts), for example propagating of the distance which is not minimal for SSSP. It results in large amount of unnecessary computations and degradation of performance. For the partially synchronized implementations the amount of speculative computations is significantly lesser.

The evaluation of PageRank has not revealed any significant difference in performance between reference and Green-Marl versions. In the first order, it is explained by the fact that for both cases the algorithms are very close (almost the same).

7 Conclusion and Future Work

In the paper the Charm++ code generator developed for the compiler of the domain-specific language Green-Marl is presented. Therefore, the Green-Marl DSL is extended to HPC clusters with distributed memory: Charm++ is added to already supported OpenMP, GPS, and Giraph target platforms.

The performance evaluation shows that Green-Marl programs compiled to Charm++ have the same performance as native Charm++ programs assuming that the same algorithms are used in the generated code and the reference implementation. In the future we plan to add a support of the Topological Routing and Aggregation Module (TRAM) [23] to the Charm++ code generator implemented in the Green-Marl compiler.

Acknowledgment

This work is partially supported by Russian Foundation for Basic Research (RFBR) under Contract 15-07-09368.

References

[1] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007. `http://dblp.uni-trier.de/rec/bib/journals/ppl/LumsdaineGHB07` (accessed: 10/23/2017).

[2] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, pp. 91–108, Oct. 1993. `http://doi.acm.org/10.1145/167962.165874` (accessed: 10/23/2017).

[3] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *2010 39th International Conference on Parallel Processing Workshops*, pp. 436–444, IEEE, 2010. `https://charm.cs.illinois.edu/newPapers/10-08/paper.pdf` (accessed: 10/23/2017).

[4] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: parallel programming for data-driven applications," in *Proceedings of the international conference on Supercomputing*, pp. 235–244, ACM, 2011. `http://doi.acm.org/10.1145/1995896.1995934` (accessed: 10/23/2017).

[5] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: a programming model for highly parallel fine-grained data-driven computations," in *ACM SIGPLAN Notices*, vol. 46, pp. 305–306, ACM, 2011. `http://doi.acm.org/10.1145/1941553.1941601` (accessed: 10/23/2017).

[6] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, p. 6, ACM, 2014. `http://doi.acm.org/10.1145/2676870.2676883` (accessed: 10/23/2017).

[7] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, pp. 52–60, IEEE, 2004. `http://ieeexplore.ieee.org/document/1299190/` (accessed: 10/23/2017).

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007. `http://dx.doi.org/10.1177/1094342007078442` (accessed: 10/23/2017).

[9] R. Haque and D. Richards, "Optimizing pgas overhead in a multi-locale chapel implementation of comd," in *Proceedings of the First Workshop on PGAS Applications*, pp. 25–32, IEEE Press, 2016. https://doi.org/10.1109/PAW.2016.9 (accessed: 10/23/2017).

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, pp. 519–538, ACM, 2005. http://doi.acm.org/10.1145/1103845.1094852 (accessed: 10/23/2017).

[11] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, M. Vaziri, and W. Zhang, "X10 and apgas at petascale," *ACM Transactions on Parallel Computing*, vol. 2, no. 4, p. 25, 2016. http://doi.acm.org/10.1145/2894746 (accessed: 10/23/2017).

[12] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: a dsl for easy and efficient graph analysis," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 349–362, ACM, 2012. http://doi.acm.org/10.1145/2189750.2151013 (accessed: 10/23/2017).

[13] S. Hong, J. Van Der Lugt, A. Welc, R. Raman, and H. Chafi, "Early experiences in using a domain-specific language for large-scale graph analysis," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 5, ACM, 2013. http://doi.acm.org/10.1145/2484425.2484430 (accessed: 10/23/2017).

[14] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying scalable graph processing with a domain-specific language," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p. 208, ACM, 2014. http://doi.acm.org/10.1145/2544137.2544162 (accessed: 10/23/2017).

[15] D. Prountzos, R. Manevich, and K. Pingali, "Elixir: A system for synthesizing concurrent graph programs," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 375–394, 2012. http://doi.acm.org/10.1145/2398857.2384644 (accessed: 10/23/2017).

[16] R. Nasre, Y. Srikant, *et al.*, "Falcon: a graph manipulation language for heterogeneous systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 54, 2016. http://doi.acm.org/10.1145/2842618 (accessed: 10/23/2017).

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010. http://doi.acm.org/10.1145/1807167.1807184 (accessed: 10/23/2017).

[18] S. Salihoglu and J. Widom, "Gps: a graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, p. 22, ACM, 2013. http://doi.acm.org/10.1145/2484838.2484843 (accessed: 10/23/2017).

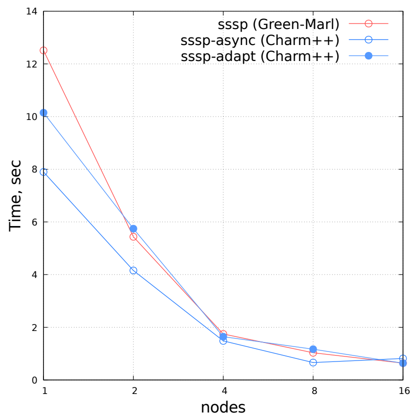
[19] S. Schelter, "Large scale graph processing with apache giraph," *Invited talk at GameDuell Berlin 29th May*, 2012.

[20] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, "Bulk synchronous parallel computinga paradigm for transportable software," in *Tools and Environments for Parallel and Distributed Systems*, pp. 61–76, Springer, 1996.

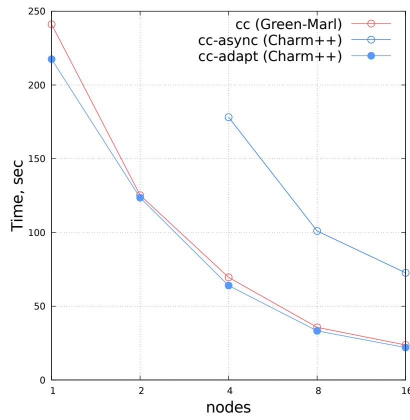
[21] A. Agarkov, T. Ismagilov, D. Makagon, A. Semenov, and A. Simonov, "Performance evaluation of the Angara interconnect," in *Proceedings of the International Conference Russian Supercomputing Days*, pp. 626–639, 2016. http://www.dislab.org/docs/rsd2016-angara-bench.pdf (accessed: 11.10.2017).

[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining.," in *SDM*, vol. 4, pp. 442–446, SIAM, 2004. https://faculty.mccombs.utexas.edu/deepayan.chakrabarti/mywww/papers/siam04.pdf (accessed: 10/23/2017).

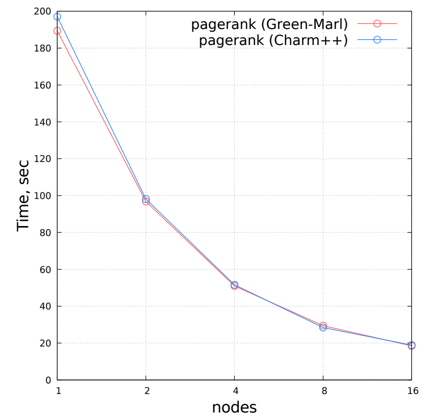
[23] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages," in *Proceedings of the International Conference on Parallel Processing*, ICPP '14, (Minneapolis, MN), September 2014. http://charm.cs.illinois.edu/newPapers/14-18/paper.pdf (accessed: 10/23/2017).



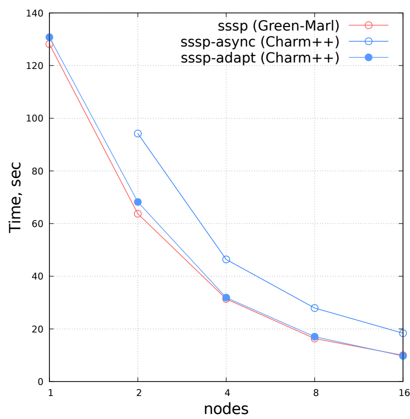
(a) SSSP, RMAT-22



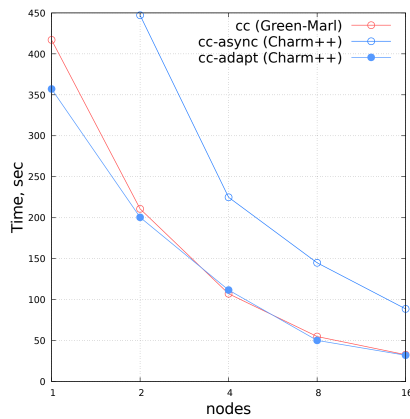
(b) CC, RMAT-22



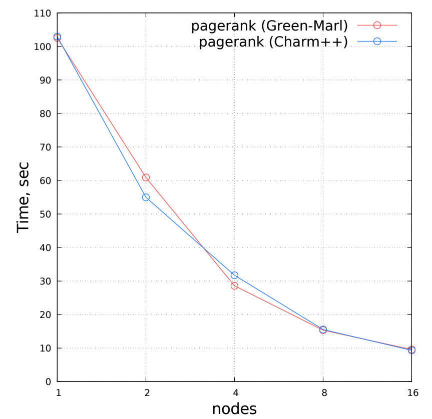
(c) PageRank, RMAT-22



(d) SSSP, Random-22



(e) CC, Random-22



(f) PageRank, Random-22

Figure 5: Performance results of SSSP, CC, and PageRank