HPCGraph: Benchmarking Massive Graph Analytics on Supercomputers

George M. Slota[*,†] Sivasankaran Rajamanickam[*] Kamesh Madduri[†]
[*]Sandia National Laboratories
[†]The Pennsylvania State University

1 Introduction

We propose HPCGraph, a new benchmark to spur algorithmic innovation in graph processing libraries and software frameworks for massive graph analytics. Several distributed-memory graph processing platforms have emerged in the past few years (reviewed in [1]), and each framework with its own performance-programmability tradeoffs. Novel algorithmic and implementation speedup strategies developed for the Graph500 [2] Breadth-First Search (BFS) kernel (for instance, direction-optimized traversal, use of bitmap data structures, a two-dimensional graph layout, high degree vertex adjacency partitioning, adaptive load balancing, fine-grained asynchronous messaging, etc.) have not yet been adopted in most graph processing frameworks. One reason why the Graph500 BFS optimizations are not commonplace in generic software frameworks is the vast diversity of modern graph analytics (i.e., the graph structures, typical sizes, and algorithms vary significantly from one application domain to another). In order to identify commonly-applicable performance optimizations in real-world settings, we constrain the test graph instance in HPCGraph to the 2012 Web Data Commons hyperlink graph (`http://webdatacommons.org/hyperlinkgraph/`), and describe seven graph processing routines ("kernels") that could be applied to this graph. We also provide reference implementations (C++/MPI/OpenMP-based) of these seven kernels. Implementation details and performance results on the NCSA Blue Waters supercomputer are discussed in [3]. Since this hyperlink graph is quite large (3.5 billion vertices and 128 billion edges), we typically require multiple compute nodes for executing the kernels. Our benchmark is similar to Graph500 in that it is textual specification-based and we do not want to restrict algorithms used for each kernel.

2 HPCGraph Kernels

A graph construction phase (Kernel 0) and six graph analytics constitute HPCGraph. We assume the input data is disk-resident in binary edge list format. The edge list file is nearly 1 TB. Each directed edge is represented by two vertices of 32-bit unsigned integer type $\langle v_0, v_1 \rangle$. The vertex identifiers are zero-indexed. The edges are sorted by start vertex. The goal of the graph construction kernel is to ingest the 1 TB file and construct an in-memory graph representation that will permit efficient execution of the next six kernels.

Since the hyperlink graph is directed with possibly many strongly connected components, we implement a routine to identify vertices belonging to the largest strongly connected component in this graph. This kernel is labeled **SCC**. Another related analytic is determining the edges and vertices that belong to the largest weakly connected component (**WCC**). When determining the largest WCC, we ignore the edge directivity and consider vertex connectivity through both incoming and outgoing edges. We then have two centrality measures, a *vanilla* implementation of PageRank (**PR**) and Harmonic Centrality (**HC**). We use the power iteration method for computing PageRank. This is an iterative method with the stopping criterion dependent on a user-defined tolerance setting. The inter-node data communication volume does not vary much across iterations in the vanilla PageRank implementation. We recommend fixing the iteration count to a small value, say 30, and reporting the average per-iteration time. Computing the harmonic centrality of a single vertex has an operation count that is linear in the number of edges and requires an augmented breadth-first search. Determining the harmonic centrality of all the vertices is thus prohibitively expensive for large graphs. In HC, we compute the harmonic centralities of the top 1000 vertices ranked by their vertex degree, and report the average time per vertex. The fifth analytic we implement is *approximate k-core computation* (**KC**). A k-core in a graph is a maximal connected subgraph in which all vertices have degree k or higher. If a vertex belongs to an l-core, but not an $l+1$-core, it is said to have a *coreness* of l. Using this approximate k-core analytic, we determine an upper bound for the coreness of every vertex in the graph in the following manner: we iteratively remove vertices that have degree less than 2^i, i ranging from 1 to 27, and determine the largest connected component in the pruned graph. The value 2^i thus gives a coreness upper bound for all vertices in the component. Our sixth analytic is a vanilla implementation of the Label Propagation (**LP**) community detection method. This is also an iterative method and the stopping criterion is typically user-defined. We can again fix the number of iterations beforehand, and use the average per-iteration time as the performance measure.

3 Reference Implementation and Preliminary Performance Results

Our reference implementations use 1D distributed graph representation of $G(V, E)$, where each MPI task owns some subset of vertices V and the set E of edges belonging to those vertices. We see two main algorithmic patterns in our reference implementations of the six kernels. These algorithmic patterns differ in how per-vertex information propagates along edges in the graph. In the first pattern, given in Algorithm 1, the stored per-vertex information $D(v)$ for a given vertex v is potentially *pushed* to all neighbors u of v. We consider breadth-first search (BFS) to be a prototypical example. In BFS, we expand a frontier of vertices, held in Q, by examining all neighbors, potentially marking them as visited or assigning a level or parent (with $D(u) \leftarrow$ update()), and then adding them to the next-level frontier Q_{next}. For each iteration, updates to the frontier are exchanged among all MPI tasks for use in subsequent iterations. Using this algorithmic pattern, we implement HC, KC, (part of) WCC, and SCC.

Algorithm 1 *Push*-Based algorithmic pattern in our reference implementations.

```
1: procedure BFS-Like($G(V, E)$)                    ▷ Task Parallel
2:     for all $v \in V$ do                          ▷ Thread Parallel
3:         $D(v) \leftarrow$ init()
4:         if addToQ($v$) then
5:             $Q_{next} \leftarrow \langle v, D(v) \rangle$
6:     while $Q_{next} \neq \varnothing$ do
7:         $Q, D \leftarrow$ AllToAllExchange($Q_{next}$)
8:         $Q_{next} \leftarrow \varnothing$
9:         for all $v \in Q$ do                      ▷ Thread Parallel
10:            for all $\langle v, u \rangle \in E$ do
11:                $D(u) \leftarrow$ update()
12:                if addToQ($u$) then
13:                    $Q_{next} \leftarrow \langle u, D(u) \rangle$
14:    return $D$
```

In our second abstraction, each vertex v updates its own per-vertex information $D(v)$ by *pulling* information from all of its neighbors u. We consider the vanilla PageRank power iteration method to be the prototypical example, where on each iteration, every vertex updates its PageRank value based on the values of its neighbors. If the value of v is updated, this information is propagated to all tasks that own an adjacency of v. The kernel implementations exhibiting this pattern include PR, LP, (part of) WCC, and our PuLP partitioning algorithm [4]. Note that graph partitioning could be used as a preprocessing strategy in the graph construction kernel.

Table 1: Execution times (in seconds) on 256 nodes of *Blue Waters* with various 1D graph layout strategies, including with PuLP partitioning.

	Partitioning Strategy			
Analytic	PuLP	1DVert	1DEdge	1DRand
+PuLP	105	-	-	-
SCC	181	184	**108**	184
WCC	**39**	88	63	112
PR	**55**	87	111	227
HC	54	54	**46**	101
KC	375	445	**363**	583
LP	**59**	400	435	367
Total	**868**	1258	1126	1574

In Table 1, we give execution time in seconds for the reference implementations on 256 nodes of the *Blue Waters* supercomputer. We try four 1D partitioning schemes (PuLP: using a partitioner for vertex identifier reordering and reducing edge cut; 1DVert: using a balanced vertex distribution; 1DEdge: balanced edge distribution; 1DRand: 1D vertex with randomly shuffled vertex identifiers). We observe that the 1DEdge strategy benefits push-based kernels, and PuLP preprocessing improves performance of pull-based kernels.

Our reference implementations will be open-sourced. We hope the CSC and Supercomputing communities find HPCGraph challenging and stimulating.

Acknowledgments

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the NSF (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the UIUC and the NCSA. This work is also supported by NSF grants ACI-1253881, CCF-1439057, and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] R. R. McCune, T. Weninger, and G. Madey, *Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing*, ACM Comput. Surv. 48, 2, Article 25 (October 2015), 39 pages.

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, J. A. Ang, *Introducing the Graph 500*, Proc. CUG 2010. `http://www.graph500.org/`.

[3] G. M. Slota, S. Rajamanickam, and K. Madduri, *A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization*, Proc. IPDPS 2016.

[4] G. M. Slota, K. Madduri, and S. Rajamanickam, *PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks*, Proc. BigData 2014.