DisNet: A Framework for Distributed Graph Computation

Ryan Lichtenwalter and Nitesh V. Chawla
Department of Computer Science
The University of Notre Dame
Notre Dame, IN 46556
Email: rlichten@nd.edu, nchawla@nd.edu
Phone: 1-574-631-7095

***Abstract*—With the rise of network science as an exciting interdisciplinary research topic, efficient graph algorithms are in high demand. Problematically, many such algorithms measuring important properties of networks have asymptotic lower bounds that are quadratic, cubic, or higher in the number of vertices. For analysis of social networks, transportation networks, communication networks, and a host of others, computation is intractable. In these networks computation in serial fashion requires years or even decades. Fortunately, these same computational problems are often naturally parallel. We present here the design and implementation of a master-worker framework for easily computing such results in these circumstances. The user needs only to supply two small fragments of code describing the fundamental kernel of the computation. The framework automatically divides and distributes the workload and manages completion using an arbitrary number of heterogeneous computational resources. In practice, we have used thousands of machines and observed commensurate speedups. Writing only 31 lines of standard C++ code, we computed betweenness centrality on a network of 4.7M nodes in 25 hours.**

I. Introduction and Related Work

DisNet is an architecture for achieving difficult feats of computation on large networks. It allows network scientists to develop and deploy new and existing algorithms to obtain results for intractable problems quickly. Architecturally, it is a master-worker framework where the master coordinates vertex-centric computation by distributing vertices to workers. In this paradigm, users must specify only how to compute results for a single vertex and how to combine computed results from two vertices. Each potentially multi-core worker machine has its own local in-memory copy of the network. DisNet *is not* an attempt at enabling the computation of any problem in any network. It *is* a recognition that many interesting forms of network analysis lend themselves to computation in a naturally parallel fashion that makes them feasible for the majority of interesting networks. While the rest of this paper will describe in detail the implementation and scaling properties of DisNet, from a user perspective the system is incredibly simple. Users do not need to deploy any complex architectures, learn any special primitives, or understand any principles of parallel processing. With a only a basic knowledge of network science or graph algorithms, users can leverage a wide variety of different computing resources, grids, and clouds to solve their problems. More concretely, by writing between 16 and 31 lines of standard C++ code with no special primitives or consideration of parallelization, we were able to leverage thousands of machines to compute algorithms such as exact diameter and betweenness centrality in under 25 hours for a 4.7M node network.

Because this work resides at the intersection of work in distributed computing and in graph algorithms, a wide variety of research has at least some bearing. We focus here on reasonable existing approaches for accomplishing rapid computation in large networks. In [1], the authors offer a much more complete survey of the challenges and options available. One solution is the application of approximation algorithms or specifically targeted parallel algorithms. For instance betweenness centrality is of such interest that researchers have developed a number of algorithms since Brandes' exact algorithm [2], including an approximation algorithm [3], a range-limited algorithm [4], and a parallel algorithm [5]. While such solutions are impressive and clever, each such approximation or parallel algorithm requires its own impressive cleverness. That is to say that the approach of devising these algorithms lacks generality in providing a solution to the explosion in the size of network analysis tasks. Further, these algorithms are often conceptually complex, may be difficult to implement, may still require substantial serial post-processing time, may provide unacceptable bounds on approximation accuracy, or may admit no error bounds at all.

Another approach is to grant the unmanageable computational complexity and devise better ways to attack the problem with greater computational resources. Barring significant breakthroughs in serial computational speed, we cannot hope for a single processor core to keep up with the size of the network data we wish to analyze. Instead, we must apply resources from computational grids and clouds to attack the problem. One solution involves distributed memory and message-passing, for example using MPI. The Parallel Boost Graph Library (BGL) [6] is designed primarily for user extension in achieving MPI computation on distributed graphs, but it supplies and supports parallel algorithms on traditional graphs. Writing parallel implementations in this manner is tricky, however, and to such a degree that the implementation and study of single-source shortest paths in the Parallel BGL was sufficiently novel to warrant publication [7]. In short,

while researchers can use existing parallel implementations in these tools, and even this requires knowledge of MPI, new implementations require additional knowledge above and beyond vanilla programming that many lack. Furthermore, message passing with distributed graph representations to exploit fine-grained parallelism is inefficient when problems present sufficient coarse-grained parallelism.

The authors of XWS [8] provide a nice framework with which users can more easily manage concurrency in fine-grained parallelism. Because it allows for fine-grained parallelism, the best it can do in terms of simplicity is provide a set of abstractions and primitives. It does not remove the burden of understanding the parallelism from its users. The simple alternative to such fine-grained parallel computation is coarse-grained distributed computation. MapReduce [9] provides an excellent framework for distributed computation when coarse-grained components of the computation are independent and divisible. For graph computation specifically, [10] describes a number of graph analysis techniques in terms of their MapReduce transformations. In recognition of some of the limitations and efficiency problems of MapReduce for many graph problems [11], the authors of [12] extend the framework with a data propagation primitive. In general, while we do not disagree that MapReduce is an excellent paradigm for computation in graphs, and indeed the mapping and reducing operations are analogous to the `process_vertex` and `combine_data` API in DisNet, [10] acknowledges that MapReduce transformations often require a significant rethinking about the computation in question.

Pregel [11] is similar in spirit to DisNet because of its vertex-centric approach and its own processing and combination primitives. Pregel employs a more complex set of abstractions than DisNet, provides for the modification of the input network, and employs message passing between computational workers. This grants it a greater flexibility and broader range of applicability than DisNet, but it again comes at the expense of an additional burden of code development and understanding on the user. Writing algorithms for Pregel requires writing code in terms of Pregel primitives and messaging operations. Pregel uses message passing with aggregation as part of the means of achieving its flexibility and must therefore occasionally wait for messages to continue processing. DisNet workers are bound only by the speed of their local processors and memories; all communication is performed while independent computations are running.

There are several fundamental differences between DisNet and all of these existing systems. DisNet represents the recognition that many interesting problems in network analysis involve computations that are inherently vertex-centrally independent. This paper offers two maxims. First, naturally parallel graph computation is most quickly run when it is unencumbered by synchronization or knowledge of unrelated computations. Second, graph algorithms are most quickly developed when the developer can focus on the complexities of the serial algorithm instead of the complexities of parallelism. Many researchers in biology, sociology, and other fields have minimal programming expertise. They can write serial graph processing code but cannot be expected to have the knowledge to write MPI or multi-threaded code or otherwise manage aspects of parallelism. DisNet presents many advantages: it is extremely easy to use and deploy, robust, fast, and can exactly solve a large set of problems of interest in the analysis of large graphs. If you can write the code describing a graph algorithm, you can use DisNet. It is even possible to debug DisNet code easily using standard debugging tools and techniques.

Some may be concerned about using localized in-memory representations, but we disagree that in-memory representation is problematic when using adjacency lists. An uncompressed space-efficient immutable adjacency list representation of a graph of $|V|$ vertices and $|E|$ edges theoretically requires only $|V| \cdot \lceil \log_2 |V| \rceil + |E| \cdot \lceil \log_2 |V| \rceil$ bits of space, because only $\log_2 |V|$ bits are necessary to address $|V|$ elements. To put this in a more concrete perspective, a large social network of 5 million nodes and 20 million edges requires less than 72 MB of space. Networks with 6 billion nodes and 36 billion edges, a possible network representation of all living humans, would require only 27.9 GB. Practical demands are somewhat higher, but not much. Meanwhile, many research universities have computing clusters with hundreds or thousands of available computational nodes, and services such as Amazon EC2 provide inexpensive computing time to those without direct access to such clusters.

II. Pertinent Problems

We provide here a small sample of the problems to which the framework nicely lends itself. The all-pairs shortest paths problem lies at the root of several of these. It may be computed in $O(V^3)$ with the Floyd-Warshall algorithm or in $O(V^2 \log(V) + VE)$ with Johnson's algorithm for sparse graphs [13]. Important centrality measures also require high computational time. Betweenness centrality, computed by Brandes' algorithm requires $O(VE)$ or $O(VE + V^2 \log V)$ in unweighted or weighted networks respectively [2]. This is intractable for large networks, and achieving results faster is of great interest [5]. Link prediction, especially with expensive path-based approaches, benefits greatly from distributed computation. In many cases, the problems listed are related conceptually or through transformation to a variety of other difficult problems. These include *all-pairs shortests paths*, *eccentricity distribution*, *radius and diameter*, *radiality*, *clustering coefficient*, *graph censuses and motif membership*, *centrality (e.g. random-walk, closeness, betweenness, edge betweenness)*, and *link prediction (e.g. Adamic/Adar, Katz, Rooted PageRank)*, just to name a few.

DisNet has already been used in a production setting by fellow researchers to: (1) compute the exact diameter of a social network of 6.5 million vertices (2) determine geodesic distances and node similarities for all pairs in a selection of 60,000 vertices of Twitter and (3) analyze the exact distribution of betweenness centralities for a demonstration of centrality scaling properties in a 4.7 million node communication network. DisNet easily supports node and edge attributes in

addition to purely topological information by including this information in the network representation.

III. DisNet User Interface

Design and implementation decisions are geared toward allowing users to complete the development phase as quickly and easily as possible so that they can finish the real work of computing the results they desire. From start to finish, typical users of DisNet must simply specify three things: the data type into which results are computed, how to compute results for a single vertex, and how to combine results from two vertices in the selected data type. Specifying the data type requires only selecting from a list of preprocessor macro options. Single-vertex result computation and data combination require writing code or calling graph library functions.

In the current implementation, typical users should modify one file, user.cpp, to make these three modifications. This file does not contain any DisNet architectural code or graph library code. Instead, its sole purpose is to serve as the one location where users write the small code segments necessary to specify the computations they want DisNet to perform.

This implementation of the DisNet framework employs C/C++ for reasons of memory efficiency and execution speed. The framework will scale to million-node networks even with commodity resources and billion-node networks with available cluster resources. Users unfamiliar with C/C++ need not fret as actual coding requirements are minimal. Only a basic knowledge of C/C++ is required to achieve computation using the provided graph library, and the user need not understand any details of the DisNet implementation itself. Despite its current roots in C/C++, the ideas that form the DisNet architecture are entirely separate from this particular implementation. DisNet may be implemented in or employ modules from many programming languages including Perl, Python, Java, and R.

We will briefly describe a few artifacts in the code segments below for reasons of clarity. First, the object variable network is available globally in user.cpp. It contains the single, immutable local representation of the network, which is shared among all worker computational threads. The network object supports several self-analytical functions and simple functions to support arbitrary computation over the topology, such as vertexCount(), which indicates the number of vertices in the network. vertex_t is a type definition for vertex identifiers. Likewise, neighbor_set_t is a type definition for the container class that holds adjacent vertex_t neighbors for each vertex.

A. Arbitrary Data Types

Users may select whatever data type they like to contain their data. The specification involves modifying one line in user.cpp to select from the provided data types. Existing selectable data types include primitives, strings, and several STL containers including vector, map, and set. Adding additional data types is simple, requiring the specification of only a few lines of code in the file macros.h and potentially a basic ASCII serialization routine.

TABLE I
THE OPTIMAL DATA STRUCTURE AND REQUIRED SLOC FOR THE (A) PROCESS_VERTEX AND (B) COMBINE_DATA ROUTINES.

Problem	Data Structure	SLOC(A)	SLOC(B)
Betweenness	vector<double>	28	3
Closeness	map<vertex_t,double>	24	3
Eccentricities	map<vertex_t,unsigned int>	15	3
Graph Diameter	unsigned int	15	1
Link Prediction - Katz	string	16	1

The initialization of these data types is important because the result of data type initialization serves as the initial value passed into the combination function with the first results. In the current implementation, all data types including primitives will be initialized by their default constructor except for random-access data structures, which contain $|V|$ objects each initialized by their default constructor. For example, a double is initialized to 0.0, a map<unsigned int,bool> is initialized to a map with no existing key-value pairs, and a vector<string> is initialized to $|V|$ empty strings. These defaults are reasonable because they serve as identities, either additive identities or set union identities or otherwise. Consider, for instance, betweenness, which involves attributing partial sums to each vertex in the network. In this case, the initialization of a vector of values to the additive identity makes perfect sense. In the rare case when the default initialization behavior for an existing type is undesirable, users may easily modify it by changing one line of code in macros.h.

Often, many different data types will get the job done. For some problems particular data structures are required. For others selecting a particular data structure may reduce performance or increase network or disk demands. Consider the case of centrality computations. Computing closeness centrality for a single vertex results in a single number representing the average distance of that vertex from all others. This number could be placed in a vector<vertex_t,double>, but then each combination requires $|V|$ summations and every transmission to the master must send the entire vector. If a map is selected then the combination step is merely adding an entry to the map, and transmission only involves as many vertices as have been computed. On the other hand, the betweenness centrality computation for each vertex produces the contribution of that vertex in shortest paths to other vertices. The computation requires storage and fast access for all vertices, and all vertices must be transmitted for accumulation of partial sums. In this case, a vector<double> is the most efficient choice. Finally, data types may contain multiple information elements such as a map<vertex_t,pair<int,double>> so that problems with overlapping solutions like eccentricity and closeness can benefit from the same computation.

B. Vertex Processing

This is the first of two entirely isolated locations where users must supply code. The function process_vertex in user.h is constructed so that users can forget about every aspect of the framework except how to compute results for a single vertex. User code can either employ the API in the network library to achieve arbitrary unanticipated analysis,

or it can call library procedures. For example, the included library already implements closeness centrality so computing closeness for all vertices requires writing only a one-line call. In the case when users want to experiment with new analytical techniques or implement new functionality, most of the coding requirements for the framework will rest in the vertex processing task.

C. Data Combining

This is the second of two entirely isolated locations where users must supply code. The function `combine_data` in `user.h` is constructed so that users can forget about every aspect of the framework except how sets of results from two vertices should be combined. This snippet of code will usually be extremely minimal. It might involve merging maps, summing two vectors, concatenating two strings, or computing the maximum of two numbers. These tasks are usually straightforward, involving at most a loop expressed in three lines of code. The combination function does not have any theoretical constraints. It may be any of idempotent, associative, or commutative depending on what the user wants to accomplish. If the combination function involves generating the union of one set with another, all three properties pertain. If the combination function involves the concatenation of two strings, none of the three pertain.

D. Development

To illustrate just how little effort is necessary, and to provide an entirely concrete example, we include a complete user specification for a working DisNet deployment in Listing 1. To compute the diameter of a graph, we need only to select the largest among all vertex eccentricities. Maintaining the diameter we have discovered so far requires a single integer, so we specify the data type as a single integer. For each vertex, we must perform a breadth-first search, to determine the maximum distance of any vertex from a source vertex. To combine results, we choose the largest eccentricity.

Table I provides summary details of the development requirements for a few other research problems. We use physical source lines of code (SLOC) as a measurement of the difficulty of the development task. The SLOC determination is based on a standard count of semicolons.

```
1  #define DATA_TYPE DATA_TYPE_UNSIGNED_INT
2
3  void process_vertex( void* dataPtr, const Network& network, vertex_t vertex ) {
4  // automatic cast and declaration of ``data'' variable from dataPtr
5  vector<bool> found(network.vertexCount());
6  vector<vertex_t> v1;
7  found.at(vertex)=true;
8  v1.push_back(vertex);
9  for(data=0;!v1.empty();data++) {
10   vector<vertex> v2;
11   for(vector<vertex_t >::const_iterator vIt=v1.begin();vIt!=v1.end();++vIt) {
12     const neighbor_set_t& n=network.outList.at(*vIt);
13     for(neighbor_set_t::const_iterator nIt=n.begin();nIt!=n.end();++nIt) {
14       if(!found.at(*nIt)) {
15         found.at(*nIt)=true;
16         v2.push_back(*nIt);
17       }
18     }
19   }
20   v1.swap(v2);
21 }
22 }
23
24 void combine_data( const Network& network, void* dataPtr, const void*
       vertexDataPtr ) {
25 // automatic case and declaration of ``data'' and ``vertexData'' variables
26 data = data >= vertexData ? data : vertexData;
```

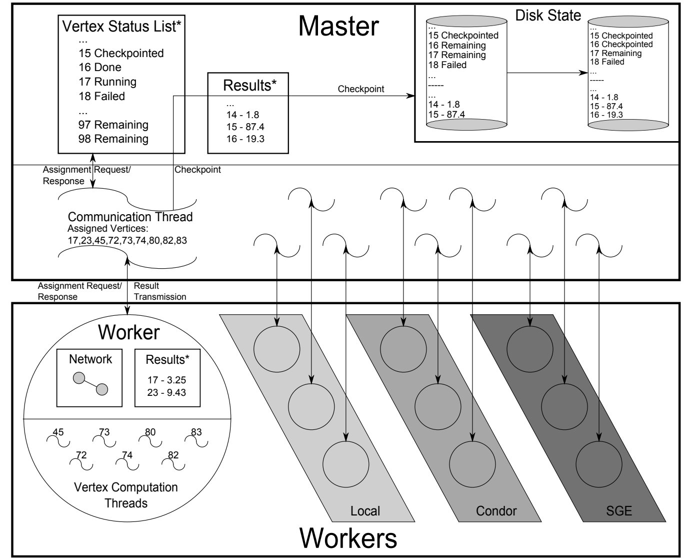


Fig. 1. An architectural overview of DisNet. Data structures with synchronized access are marked with *.

```
27 }
```

Listing 1. All required user code for graph diameter.

IV. Architectural Design

The framework employs the master-worker paradigm to accomplish computation. The master coordinates work among the workers and maintains the consistency of the results. Workers accept vertices as basic independent units of the computational task and run the same user-specified routine for each. Workers communicate only with the master, never with each other. Figure 1 is an illustration of these basic components, their construction, and their interactions. Each master communication thread and each worker share the same data as their exemplars on the left.

The only requirement for a machine to serve as a DisNet master is that it have an accessible port on which to listen for contact from new worker machines. Workers may be instantiated on local or remote machines directly operated by the user, within computing grids such as Condor [14] or the Sun Grid Engine (SGE), or even on cloud resources such as Amazon EC2. DisNet is not inherently married to any particular underlying grid or cloud management system. Any machine or virtual machine that can send and receive data using TCP/IP may serve as a worker. Any grid or cloud management system that allows user-specified programs and network communication can host pools of workers. In practice, we have simultaneously employed locally-operated servers, Condor, SGE, and Globus. DisNet includes batch submission scripts for all of these.

To be useful and effective, the framework must scale to provide service for at least thousands of workers, reliably handle worker failures, and efficiently make use of disk and network resources. Because systems may provide multiple cores to a worker, the worker abstraction should handle

these resources optimally. Some high-throughput systems like Condor employ a wide variety of architectures and operating systems with heterogeneous resources, so the framework must be constructed in a way that is portable across operating systems and architectures both in terms of how it is constructed and how it transmits information. Given that computation may require years of CPU time, the framework must consider computations to be precious and must be able to recover from almost any type of error or failure in a consistent state and with minimal loss of data. The data involved may place arbitrarily severe demands on the disk and network, but the machines involved may be arbitrarily prone to failure. To allow for appropriate trade-offs in this environment, the framework must allow for simple tuning. Perhaps most importantly, the framework must allow researchers to compute results on large networks quickly and with minimal effort. We can encapsulate these goals tersely: portability, efficiency, scalability, reliability, customizable tuning, and ease of use. We will discuss the final point, ease of use, in Section III.

A. Portability

DisNet achieves portability through the use of Bourne shell scripts and remote compilation. When `master.sh` is invoked, it combines and compresses all worker source code and creates a compressed representation of the network. When `worker.sh` is invoked, it first requests worker source code from the master, which it then compiles. Remote compilation is performed once at each physical worker machine and requires only a couple seconds. Next, the worker requests the network from the master unless the network file is specified to be available locally. Transmission of compressed million-node social networks requires less than 3 seconds on a 100 Mbps network. After these preliminary operations have been successfully completed, the worker Bourne script invokes the C++ worker binary. The setup time is negligible compared to the time required to compute the difficult results for which DisNet is designed.

B. Efficiency

DisNet workers load the network into memory once when they are started and use the same immutable representation for the duration of their lifetime. The immutable nature of the network also allows for a single read-only representation of the network that all worker threads can share. Local workers on multi-core machines can employ any number of those cores for computation. As processors offer increasingly many cores, workers will enjoy increasing speed and lower overall ratios of memory requirements to core count. Transient threads arise to handle each vertex and require only the additional memory necessary for running the computing algorithm. The main worker thread also contains a copy of all data combined so far. When each transient thread completes vertex processing, the result is combined with the existing data under a mutex lock.

Time with the master is a precious resource, so workers accept as a parameter a number of vertex results to combine locally before sending data to the master. As we will demonstrate theoretically and practically below, this combination parameter effectively eliminates the master as a bottleneck without the complexity of having multiple masters in a distributed or hierarchical arrangement. It simultaneously reduces processing demands on the master and reduces the bandwidth required by both the master and the worker. Transient threads spawned by the worker for additional vertex assignments continue running while the worker communicates with the master. The combined vertex data is locked until communication finishes at which time it is cleared and again becomes available to the transient threads for update.

C. Scalability

One of the principal concerns of DisNet is scalability. By isolating workers from each other entirely, the architecture guarantees that the only potential bottleneck is the master. The master must be able to handle many simultaneous connections, deserialize results, combine results with the combination function, and checkpoint all as quickly as the workers can supply results. Whereas workers are likely only to encounter CPU limitations, the master may encounter limitations due to the CPU, the disk, or the network. CPU and network limitations are both due to frequent and voluminous client communication, which necessitates equally frequent deseralization and combination. By designing workers that combine results as they produce them, most of the burden on these resources can be reduced to allow for virtually arbitrary scalability. Scalability problems are reduced to tradeoffs between speed and waste, which we can leave to users based on their level of confidence in the stability of their systems.

We implemented the master with a thread-per-worker communication paradigm. Although we are aware of theoretical arguments against such a paradigm, in practice we did not observe thread scheduling problems or network saturation. The framework does not currently provide for hierarchical master-worker relationships where workers are themselves masters for a broader array of workers. Such a provision would indeed decrease load on the master, but at the cost of a decrease in CPU and network efficiency. Load on the master is not problematic in practice. Even if it were, the master can itself utilize multiple cores, and result deserialization is often substantially more expensive than data combination in the mutex, reducing the probability of encountering a real bottleneck.

D. Reliability

The most efficacious approach to achieving reliability is to acknowledge that workers are inherently unreliable and to treat them accordingly. Workers may experience software or hardware failures for innumerable reasons, most of which are entirely beyond our control. The master ensures the appearance of reliability to the user through failover. The master must be able to detect when workers fail so that it can reassign the relevant vertices. It accomplishes this using keepalive probes. When a worker fails to respond to a number of successive

keepalive probes, the master considers the worker dead, closes the connection, and moves all outstanding assigned vertices from the RUNNING state to the FAILED state.

We take the approach that the system should successfully complete as much computation as possible as quickly as possible in the event that errors occur. Because some worker failures may actually result from a problem processing a particular vertex, vertices in the FAILED state are reassigned only when there are no longer REMAINING vertices to assign to waiting workers. This implicitly assumes that the remainder of results are still valuable and avoids spinning on the assignment of problematic vertices that consume worker time only to probably or certainly fail. After all vertices have been moved from the REMAINING state through assignment to workers, FAILED vertices are reassigned repeatedly until all vertices are finished or the user terminates the master. At no time are resources idle, because FAILED vertices occupy free resources as soon as there are no longer REMAINING vertices to assign. Alternatively, FAILED vertices might be retried immediately with a threshold number of attempts before reporting the failure to the user.

In the case of master machine failure, recovery is accomplished via checkpointing. Checkpoints are created by writing a single temporary file that contains the vertex status information and the computed results. After writing a temporary file, the master atomically renames the file to create the latest checkpoint and then updates the in-memory vertex status information moving vertices from the DONE state to the CHECKPOINTED state. When all vertices reach the CHECKPOINTED state, the master indicates successful completion and terminates. Until then, the master sends deferral messages to unoccupied workers in case straggling computations fail. The master creates a checkpoint in one of three circumstances: (1) every time a specified number of vertices has been successfully processed, (2) when all vertices have been successfully processed, or (3) upon detecting Ctrl+C. When the master is restarted after failure, it loads the vertex status list and result data structure from the latest existing checkpoint and listens for connections from new workers.

E. Customizable Performance Tuning

Tuning is achievable with a data combination parameter, c, which is accepted by master and worker alike. On the master, this parameter designates the number of vertex results, not worker transmissions, to accept and combine before initiating a checkpoint. Decreasing the value of this parameter results in more frequent checkpoints and less wasted worker time if the master fails; increasing it lowers the disk throughput requirements on the master. On workers, the combination parameter designates the number of vertices to finish and combine before sending data to the master. In this case, a lower parameter value decreases wasted worker time when the worker fails; higher values reduce the computational and network bandwidth demands on the master. The parameter c can be set differently on the master, each worker, or on each batch of workers submitted to a computational grid.

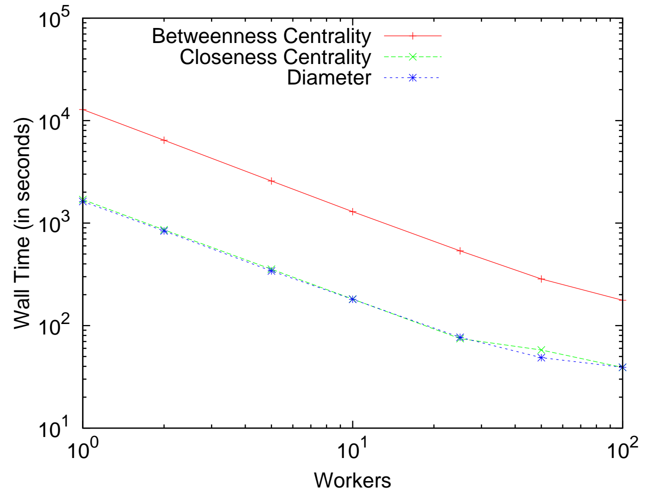


Fig. 2. Strong scaling behavior.

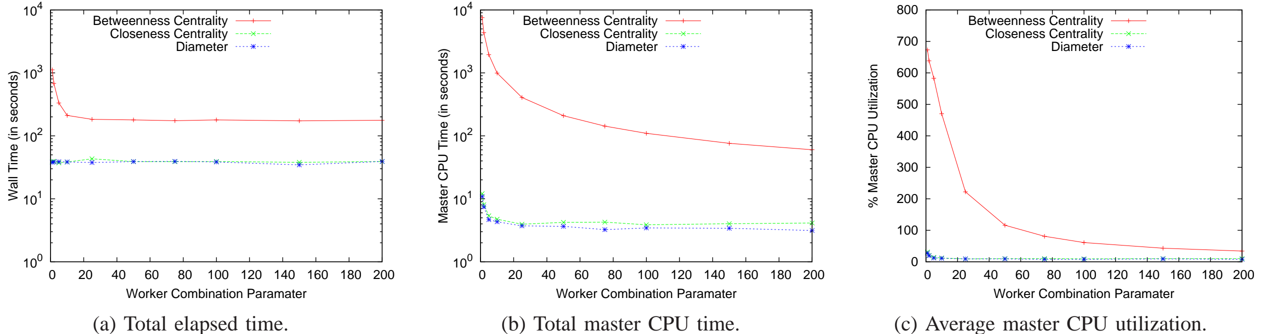
V. Efficiency and Scalability

Analyzing the efficiency of non-parallelized algorithms on real social networks of the size for which DisNet is designed is intractable because they simply take too long to run. DisNet does not reduce the total computational time to complete long-running algorithms on large graphs, but simply makes it extremely easy to distribute this time across many heterogeneous resources. For comparative experiments, we extract the largest strongly-connected component from Erdös-Rényi $(n, M) : M = 4n$ model random graphs of different sizes to gain an idea of the efficiency and scalability of the distributed system. All experiments are performed with betweenness centrality, closeness centrality, and diameter using code compiled with the -O3 flag to the GNU g++ compiler. The master is a dual quad-core 3.0 GHz Xeon machine with 1 Gbps network bandwidth. We report CPU utilization such that 100% utilization means complete occupation of a single core.

A. Strong Scaling

Figure 2 shows the strong scaling of the system, how it behaves as additional workers are added. We select a random graph with 125,871 vertices and 503,411 edges. To obtain these results, we use only 2.53 GHz core-clock Nehalem workers on the SGE. The master was set to checkpoint only after completion and the worker combination parameter was fixed at 200.

The figure indicates excellent strong scaling, almost perfect scale-free behavior. Doubling the number of workers results in 50-55% of the wall time at any number of workers. From 50 to 100 workers, the time to complete the computation becomes so small that the time to setup worker connections, send source code, and distribute vertices becomes a significant factor. To illustrate that strong scalability continues to pertain with an ever-increasing number of workers, we computed the two more challenging problems on a SCC with 503,758 nodes and 2,014,746 edges. Closeness required 1186 seconds of wall time with 50 workers and 602 seconds with 100 workers, 51%



(a) Total elapsed time. (b) Total master CPU time. (c) Average master CPU utilization.

Fig. 3. Performance and master burden with respect to the worker combination parameter.

as much time. Betweenness required 6319 seconds of wall time with 50 workers and 3416 seconds with 100 workers, 54% as much time.

B. *Parameter Manipulation*

We also report burden on the master in terms of CPU time with different values of the combination parameter, c. We use the same random graph with the same parameters as for the strong scaling experiments, but the number of machines is fixed at 100. Figure 3 illustrates results. Theoretically, the three problems are representatives of different classes of difficulty in terms of their network bandwidth requirements and deserialization demands on the master. Betweenness centrality scales according to $O\left(\frac{|V|^2}{c}\right)$, closeness centrality according to $O(|V|)$, and diameter according to $O\left(\frac{|V|}{c}\right)$.

To understand the reason for these different classes of scaling, one must consider what information must be sent for each vertex and how that information interacts with or is independent of c. Betweenness centrality requires calculating and summing partial sums that result from one computation initiated at each vertex. Each transmission requires sending $|V|$ values, and $|V|$ transmissions are required, but workers can effectively reduce the number of transmissions by a factor of c by summing together individual vertex results. Closeness centrality requires sending $O(1)$ information for each of the $|V|$ vertices. Worker combination can aggregate this data for fewer transmissions, but ultimately all $|V|$ values must be transmitted. Finally, diameter requires only a single integer. This integer must be selected from the maximum eccentricity of each of the $|V|$ vertices, but worker combination can reduce the amount of data transmitted by selecting the maximum integer from all its results and discarding the rest.

Practical results align well with theory. Betweenness places much greater demands on the master. Nevertheless, we observe that the time required to finish the computation stops decreasing after the combination parameter reaches 25, which indicates that for the selected graph size and 100 workers, the master is no longer a bottleneck. We also note that increasing graph size for betweenness centrality computations increases the size of the data to transmit to the master, but computation for individual vertices also takes longer. The $O\left(\frac{|V|^2}{c}\right)$ requirements above are overall requirements. Per unit time, the number of workers involved is more important in selecting a combination parameter balanced against the reliability of the worker machines in question.

The substantially lighter demands of the closeness and diameter computations are apparent in their lower absolute time requirements, but also in terms of their flatness. Even with workers sending results upon finishing each vertex, the master is no bottleneck, but CPU utilization is slightly higher for low values of c. Since closeness centrality data volume is independent of c, the smaller, more frequent transactions are causing the higher utilization by virtue of their frequency alone. In employing DisNet to achieve our own results for a large social network the master was not a bottleneck. Speeds are instead bound to the number of accessible workers.

C. *Problem Size*

Betweenness centrality is among the most stressing problems of those listed in Section II because the data structure must always contain information for every vertex. For this reason, it serves as a conservative benchmark for DisNet. Figure 4 illustrates results for several problem sizes. In line with our scalability claims, we observe a monotonic decrease in average master CPU utilization with increasing network size. DisNet does not reduce the total computational time required to complete a particular task, so the functional form of the growth in Figure 4 is the same regardless of how the computation is achieved. Nonetheless, when the graph size surpasses 500,000 vertices, a week of serial computation is no longer sufficient to reach a solution. Even with only 100 cores, DisNet requires under two hours, and adding more computational power is simple.

D. *Extreme Performance*

Finally, we are interested in the most grandiose performance we can muster with DisNet and the computational resources available to us. We compute the betweenness of a real-world social network with 4,773,246 vertices and 29,393,714 edges. The run required 25 hours. The master interacted with 2368 distinct workers and successfully handled 442 worker failures, most due to Condor evictions. The most highly multi-threaded worker ran 28 threads on a 32-core machine, achieved nearly

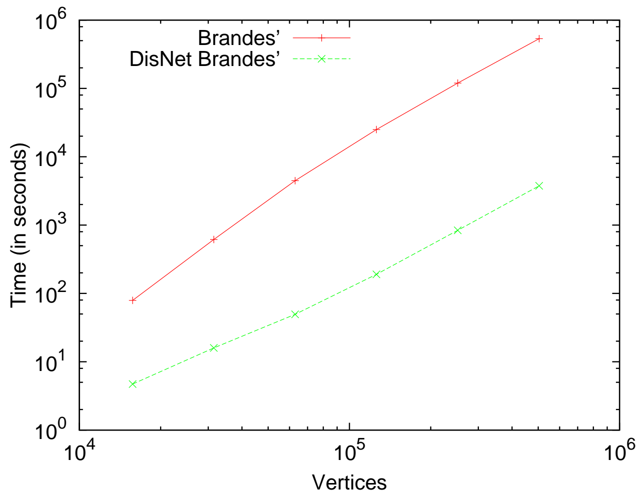


Fig. 4. Speedup as a function of problem size.

constant 2800% utilization, and required only 10 GB of memory. Local workers were run with a combination parameter of 1000, SGE workers with 500, and Condor workers with 250. The master was set to checkpoint every 100,000 vertices. With logging overhead, peak master CPU utilization was 609%, average master CPU utilization was 307%, and total master CPU time was 76.1 hours.

VI. Conclusion

DisNet represents a new avenue for approaching computation in large networks. With recognition of the natural parallelism in most network computation through a vertex-centric approach, a framework can achieve high efficiency and allow for easy development. We have illustrated how computationally complex measures like graph diameter, closeness centrality, and betweenness centrality can be calculated in a matter of hours on a representative of one of the largest social networks under active study, results that would have required years of computation with serial algorithms. Researchers can achieve the same complex computations in a reasonable time given widely available grid resources. Most importantly, the system is easy to use and understand. DisNet abstracts away all details of parallelism so that users only need to consider the fundamentals of the problem they are trying to solve in natural, untransformed terms and can ignore virtually all aspects of how the computation is achieved.

We recognize that some networks under study contain 10^9 vertices or more. So long as there is a reasonable pool of workers with sufficient memory, DisNet is not limited to networks of any particular size. Nonetheless, for these extremely large networks, duplicate local representations of the network in general, and DisNet in particular, may not be suitable. We contend that the selection of interesting algorithms, so many of which are $O(|V|^2)$ or $O(|V|^3)$, that can reasonably be applied to such networks is currently severely constrained under any system even with world-class computational resources. In short, DisNet is not an attempt to allow the computation of anything conceivable on any conceivable network. It is an attempt to make harnessing many heterogeneous computational resources easy and efficient in the pursuit of computing many interesting computationally complex algorithms on most networks under study.

DisNet includes a skeleton network library for rapid development with the framework as is, but the framework exists independent of the library. Existing graph libraries such as the BGL or home-built code can be substituted with few changes to the core software. DisNet is available at http://nd.edu/~dial/software.html.

Acknowledgments

Research was sponsored in part by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053 and in part by the National Science Foundation (NSF) Grant BCS-0826958. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, and J. Guest Editors, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[2] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Math. Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[3] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Algorithms and Models for the Web-Graph*, ser. LNCS, vol. 4863. Springer-Verlag, 2007, pp. 124–137.

[4] M. Ercsey-Ravasz and Z. Toroczkai, "Centrality scaling in large networks," *arXiv e-print 1003.0692*, 2010.

[5] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. of the 2009 IEEE Intl. Symp. on Parallel and Distributed Processing*. IEEE, 2009, pp. 1–8.

[6] D. Gregor and A. Lumsdaine, "The Parallel BGL: A generic library for distributed graph computations," in *Proc. of the Fourth Workshop on Parallel Object-Oriented Scientific Computing*, July 2005.

[7] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single-source shortest paths with the parallel boost graph library," *The 9th DIMACS Impl. Challenge: The Shortest Path Problem*, 2006.

[8] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Proc. of the 2008 37th Intl. Conf. on Parallel Processing*. IEEE Computer Society, 2008, pp. 536–545.

[9] J. Dean and S. Ghemawat, "Map reduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–114, 2008.

[10] J. Cohen, "Graph twiddling in a MapReduce world," *Comp. in Science and Eng.*, vol. 11, pp. 29–41, 2009.

[11] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of the 2010 Intl. Conf. on Mgmt. of Data*. ACM, 2010, pp. 135–146.

[12] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," in *Proc. of the 2010 Intl. Conf. on Mgmt. of Data*. ACM, 2010, pp. 1123–1126.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. The MIT Press, 2001.

[14] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proc. of the 8th Intl. Conf. on Dist. Computing Systems*, vol. 43, 1988.