Large Scale Complex Network Analysis using the Hybrid Combination of a MapReduce cluster and a Highly Multithreaded System

Seunghwa Kang David A. Bader
Georgia Institute of Technology
Atlanta, GA, 30332, USA

***Abstract*—Complex networks capture interactions among entities in various application areas in a graph representation. Analyzing large scale complex networks often answers important questions—*e.g.* estimate the spread of epidemic diseases—but also imposes computing challenges mainly due to large volumes of data and the irregular structure of the graphs.**

In this paper, we aim to solve such a challenge: finding relationships in a subgraph extracted from the data. We solve this problem using three different platforms: a MapReduce cluster, a highly multithreaded system, and a hybrid system of the two. The MapReduce cluster and the highly multithreaded system reveal limitations in efficiently solving this problem, whereas the hybrid system exploits the strengths of the two in a synergistic way and solves the problem at hand. In particular, once the subgraph is extracted and loaded into memory, the hybrid system analyzes the subgraph five orders of magnitude faster than the MapReduce cluster.

***Keywords*-cloud computing; parallel algorithms; power-law graphs.**

I. Introduction

Complex Networks [12] abstract interactions among entities in a wide range of domains—including sociology, biology, transportation, communication, and the Internet—in a graph representation. Analyzing these complex networks solves many real-world problems. Watts and Strogatz [18] find that a small graph diameter, which is a common feature in many complex networks [18], has a significant impact on the spread of infectious diseases. Albert *et al.* [3] study the impact of a power-law degree distribution [2], [14] on the vulnerability of the Internet and the efficiency of a search engine. Bader and Madduri [7] compute the betweenness centrality of the protein-interaction networks to predict the lethality of the proteins, and Madduri *et al.* [17] apply an approximate betweenness centrality computation to the IMDb movie-actor network and find the important actors with only a small number of connections.

Analyzing large scale complex networks, however, imposes difficult computing challenges. Graphs that represent complex networks commonly have millions to billions of vertices and edges. These graphs are often embedded in the raw (and often streaming) data—*e.g.* web documents, e-mails, and published papers—of terabytes to petabytes. Extracting the compact representation of a graph or subgraph from large volumes of data is a significant challenge. The irregular structure of these graphs stresses traditional hierarchical memory subsystems as well. These graphs also tend not to partition well for multiple computing nodes; the partitioning of these graphs has significantly larger edge cuts than the partitioning of the traditional graphs which are derived from physical topologies [8]. This necessitates large volumes of inter-node communication. In this paper, we study the problem of extracting a subgraph from a larger graph—to capture the challenge in processing large volumes of data—and finding single-pair shortest paths in the subgraph—to capture the challenge in irregularly traversing complex networks.

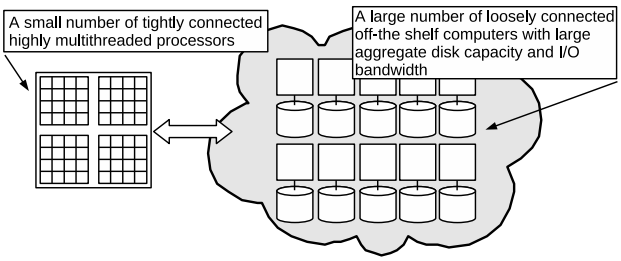


Figure 1: The hybrid combination of a MapReduce cluster and a highly multithreaded system.

We map our problem onto three different platforms: a MapReduce cluster, a highly multithreaded system, and the hybrid combination of the two. Cloud computing, using the MapReduce programming model, is becoming popular for data-intensive analysis. A Cloud computing system with the MapReduce programming model—or a MapReduce cluster—is efficient in extracting a subgraph via filtering. Finding a single-pair shortest path, however, requires multiple dependent indirections with irregular data access patterns. As we will show later in this paper, the MapReduce algorithm to find a single-pair shortest path is not work optimal and also requires high bisection bandwidth to scale on large systems. In our experiment, a single-pair shortest path problem runs five orders of magnitude slower on the MapReduce cluster than the highly multithreaded system with the single Sun UltraSparc T2 processor.

A highly multithreaded system—with the shared memory programming model—is efficient in supporting a large number of irregular data accesses all across the memory space. This system, however, often has limited computing power, memory and disk capacity, and I/O bandwidth and inefficient or even impossible to process very large graph data. Finding

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

a single-pair shortest path in the subgraph, in contrast, fits well with the programming model and the architecture.

The hybrid system (see Figure 1) exploits the strengths of the two different architectures in a synergistic way. A MapReduce cluster extracts a subgraph, and a highly multithreaded system loads the subgraph from the cluster and finds single-pair shortest paths. The subgraph loading time is less significant if the subgraph size is much smaller than the original data. The hybrid system solves our problem in the most efficient way in our experimentation.

II. Complex Network Analysis

Complex networks are often embedded in large volumes of real-world data. Extracting a graph representation from the raw data is a necessary step in solving many complex network analysis problems. Analyzing the subgraphs of a larger graph is also an interesting problem. Costa *et al.* [12] survey measurement parameters for complex networks and present several examples in this direction. Subgraphs with the edges created in different time intervals along the growth of a network reveal the dynamic characteristics of the network. Rich-club coefficient measures interactions among only highly influential entities—or high degree vertices using terminologies in graph theory. Filtering only a certain type of vertices—*e.g.* finding a network of Atlantans or computer scientists in the larger Facebook network—also creates interesting subgraphs to investigate. A common operation involved in the above cases is filtering large input data. A large part of the raw data is unnecessary in generating a graph representation. Extracting a subgraph involves scanning large volumes of data and filtering out vertices and edges that are not part of the subgraph.

The size of an extracted graph is often significantly smaller than the size of the input data, and the memory requirement is less demanding with the sparse graph representation—*i.e.* the adjacency list format which stores the list of neighbors for every vertex. The sparse graph representation requires $O(n + m)$ memory space—for n vertices and m edges—with a small hidden constant. Graphs with multiple millions to billions of edges fit into the main memory of moderate size systems—*e.g.* Madduri *et al.* [17] compute the approximate betweenness centrality of the input graph of 134 million vertices and 1.07 billion edges using the Sun T5120 server with 32 GB of DRAM. For the graphs represented in the compact representation, a large fraction of practical complex networks and subnetworks is likely to fit into the memory capacity of moderate size systems, and if this is the case, the match between the graph analysis problems' computational requirements and the architectures' capability becomes more important than the mere capacity.

A Description of Our Problem. We study the problem of extracting a subgraph from a larger graph and finding single-pair shortest paths in the subgraph. We find shortest paths (one shortest path per pair) for up to 30 pairs, which are randomly picked out of the vertices in the subgraph. To extract a subgraph, we filter the input graph to include only the edges that connect the vertices in the subnetwork—we experiment with three subnetworks that cover approximately 10%, 5%, and 2% of the entire vertices—and create adjacency lists from the filtered edges. We also assume that the input graph is generated in advance and stored in the MapReduce cluster.

The input graph is an R-MAT graph [10] with 2^{32} (4.29 billion) vertices and 2^{38} (275 billion) undirected edges. Sampling from a Kronecker product generates an R-MAT graph which exhibits several characteristics similar to social networks such as the power-law degree distribution and the clustering structure. The graph has an order of magnitude more vertices than the Facebook network with a comparable average vertex degree—the Facebook network has over 250 million active users with 120 friends per user in average. We use the R-MAT parameters a=0.55, b=0.1, c=0.1, d=0.25. The graph size is 7.4 TB in the text format.

III. MapReduce

A MapReduce cluster is typically composed of a large number of commodity computers connected with off-the-shelf interconnection technologies. The MapReduce programming model frees programmers from the work of partitioning, load balancing, explicit communication, and fault tolerance in using the cluster. Programmers provide only map and reduce functions (see Figure 2). Then, the runtime system partitions the input data and spawns multiple mappers and reducers. The mappers apply the map function—which generates an output (key, value) pair—to the partitioned input data. The reducers shuffle and sort the map function output and invoke the reduce function once per each key with a key and all values associated with the key as input arguments.

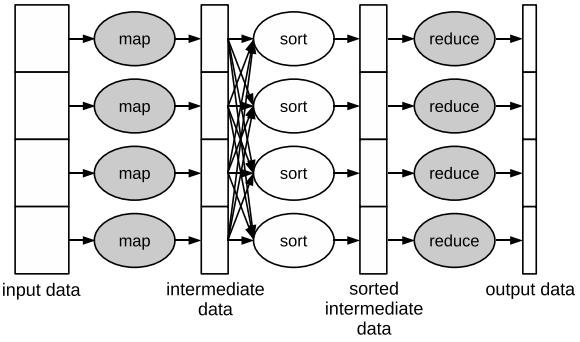


Figure 2: A MapReduce workflow. A programmer provides map and reduce functions (gray shading), and the runtime system is responsible for the remaining parts.

A. Algorithm Level Analysis

In this section, we provide a criterion to test the optimality of MapReduce algorithms based on a number of MapReduce iterations and amounts of work in each MapReduce iteration. Assume an edge list with m edges. To extract a subgraph from the list, we need to inspect the edges and include only the edges that belong to the subnetwork of interest. This is local computation which only uses data in a single edge representation, and a single invocation of a map function is sufficient to process one edge. Forming adjacency lists from the filtered edges requires global computation which accesses multiple edges at the same

time. All edges incident on a same vertex need to be co-located. A MapReduce algorithm co-locates the edges in the shuffle and sort phases via indirections with a key. This is different from $O(1)$ complexity random accesses, or indirections with an address, for the random access machine (RAM) model. One pair of the shuffle and sort phases can serve all independent indirections. As co-locating incident edges for one vertex is independent of co-locating incident vertices for other vertices, a single MapReduce iteration is sufficient for the subgraph extraction.

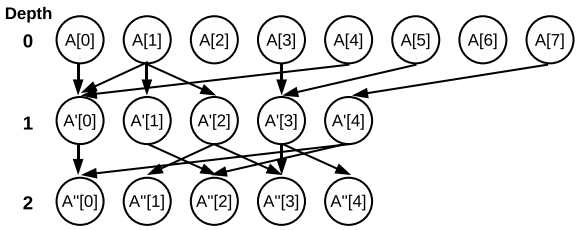


Figure 3: A directed acyclic graph (DAG) for MapReduce computation.

Given a pair of vertices s and t, we find a shortest path from s to t by running two breadth-first searches from s and t until the frontiers of the two meet each other. Each breadth-first search first inspects every vertex one hop away from the source vertex, then visits the vertices one hop away from the neighbors of the source vertex, and so on. This expands the frontier by one hop in each step, and there is dependency between indirections involved in the successive steps. Assume a directed acyclic graph (DAG) is constructed with an element (say $A[i]$) of the input data (say A) as a vertex and the indirections as edges (see Figure 3)—one adjacency list of the input adjacency lists is mapped as a vertex in our case. Local computation, which accesses only $A[i]$, is ignored in constructing the DAG. The longest path of dependent indirections determines the depth of the DAG and accordingly a number of required MapReduce iterations. $\lceil d/2 \rceil$, where d denotes the distance from s to t, sets the minimum number of MapReduce iterations to find the shortest path in our problem.

Assume the input data A has n elements, and each element of A is smaller than a certain constant that is much smaller than the input data size. Then, amounts of work in the map, shuffle, sort, and reduce phases and a single MapReduce iteration—W_{map}, $W_{shuffle}$, W_{sort}, W_{reduce}, and $W_{iteration}$, respectively—becomes the following.

$$
\begin{aligned}
W_{map} &= O(n(1+f)) \\
W_{shuffle} &= O(nf) \\
W_{sort} &= p_r \times Sort\left(\frac{nf}{p_r}\right) \\
W_{reduce} &= O(nf(1+r)) \\
W_{iteration} &= O(n+nf+nfr) + p_r \times Sort\left(\frac{nf}{p_r}\right)
\end{aligned}
$$

where $f = \frac{map\ output\ size}{map\ input\ size}$, $p_r = a\ number\ of\ reducers$, and $r = \frac{reduce\ output\ size}{reduce\ input\ size}$. Each reducer sorts its input data sequentially, and $Sort(n)$ is $O(n)$—with bucket sort—assuming a finite key space and $O(n\ log\ n)$ otherwise. If solving a problem requires k iterations, the amount of work to solve the problem on a MapReduce cluster, or $W_{MapReduce}$ becomes

$$
W_{MapReduce} = \sum_{i=1}^{k} \left(O(n_i + n_i f_i + n_i f_i r_i) + p_r \times Sort\left(\frac{n_i f_i}{p_r}\right)\right)
$$

where $n_1 = n$ and $n_{i+1} = n_i f_i r_i$. f_i and r_i are f and r for the ith iteration, respectively. A parallel random access machine (PRAM) algorithm is optimal if $W_{PRAM}(n) = \Theta(T^*(n))$ [16], where W_{PRAM} is an amount of work for the PRAM algorithm and T^* is the time complexity of the best sequential algorithm assuming the RAM model [16]. We extend this optimality criterion for MapReduce algorithms, and a MapReduce algorithm is optimal if $W_{MapReduce}(n) = \Theta(T^*(n))$. Every phase in a MapReduce iteration is trivially parallel, and the time complexity of MapReduce computation to solve a problem with p nodes, or $T_{MapReduce}(n,p)$, is $W_{MapReduce}(n)/p$ assuming $p \ll n$. To realize this time complexity in physical systems, communication in the shuffle phase needs to be minimized and overlapped with the preceding map phase unless bisection bandwidth of the MapReduce cluster grows in proportion to the number of compute nodes.

B. System Level Analysis

Here, we analyze the architectural features of a MapReduce cluster and their impact on MapReduce algorithms' performance. The MapReduce programming model is oblivious to the mapping of specific computations to specific computing nodes, and the real-world implementations of the programming model—Google MapReduce [13] and open-source Hadoop [1]—provide very limited control over this mapping. Thus, the communication patterns in the shuffle phase are arbitrary, and with arbitrary communication patterns, approximately one half of the map phase output crosses the worst case bisection of a MapReduce cluster. Dividing the map phase output by the bisection bandwidth of a MapReduce cluster calculates execution time for the shuffle phase, or $T_{shuffle}$ as a result. The shuffle phase, however, can be overlapped with the preceding map phase. Say T_{map} is execution time for the map phase. Then, execution time for the map and shuffle phases is $max(T_{map}, T_{shuffle})$. $T_{shuffle}$ does not affect the overall execution time as long as $T_{map} \geq T_{shuffle}$. T_{map} involves only local computations and scales trivially. Scaling $T_{shuffle}$ to a large number of nodes is much more demanding as this requires linear scaling of the bisection bandwidth to the number of nodes—the network cost increases superlinearly to the number of nodes to scale the bisection bandwidth proportional to p. Disk I/O overhead—the representative MapReduce runtime systems store intermediate data in disks instead of DRAM—increases T_{map} and lowers the bisection bandwidth requirement. If

$f \ll 1$, the bisection bandwidth requirement is even lower. If f is large, however, the bisection bandwidth is likely to become a bottleneck if the system size becomes very large.

Disk I/O overhead is unavoidable for workloads that overflow the aggregate DRAM capacity of a MapReduce cluster—*e.g.* to store the large input graph in our problem. If workloads' memory footprint fits into the aggregate DRAM capacity, however, the relatively low disk I/O bandwidth compared to the DRAM bandwidth incurs a significant performance overhead—*e.g.* finding a shortest path in the smaller subgraph.

C. Finding Shortest Paths in the Subgraph

To find shortest paths in the subgraph, we first need to extract the subgraph from the larger input graph (see Algorithm 1). For the subgraph extraction, $f \leq 0.01 \ll 1$ in our problem, and a single MapReduce iteration is sufficient as discussed in Section III-A. W_{map} dominates the execution time as $f \ll 1$, and W_{map} is $O(m)$, where m is a number of edges in the input graph. The best sequential algorithm also requires the asymptotically same amount of work, and the MapReduce algorithm is optimal under our optimality criterion. The bisection bandwidth requirement is also low as the volumes of communication in the shuffle phase—$O(mf)$—is much smaller than amounts of the disk I/O and the computation in the map phase—$O(m)$. The disk I/O overhead in reading the input graph is unavoidable, and amounts of the disk I/O in the following phases are much smaller.

```
Input: an edge list for the input graph
Output: adjacency lists for the subgraph
map(key/* unused */, edge/* connects head and tail */) {
  if (edge belongs to the subnetwork of interest)
    output (head, tail);/* a (key, value) pair */
    output (tail, head);
  }
}
reduce(vertex, adjacent_vertices) {
  degree = number of vertices in adjacent_vertices;
  output (vertex, degree + '' '' + adjacent_vertices);
}
```

Algorithm 1: Extracting a subgraph using MapReduce.

Once we extract the subgraph, the next step is finding a single-pair shortest path in the subgraph. This requires multiple MapReduce iterations (say k iterations) as analyzed in Section III-A. Our shortest path algorithm extends the breadth-first search algorithm designed for the MapReduce programming model (see [9]) and sets initial distances from the two vertices in the pair (say s and t) first—0 for the source vertex and ∞ for the others. Then, our implementation invokes Algorithm 2 repeatedly to expand breadth-first search frontiers from s and t by one hop in each invocation.

Assume m/n is constant, where n is a number of vertices and m is a number of edges. Then, setting initial distances from s and t requires only the map phase and costs $O(n)$ work. For Algorithm 2, $f \geq 1$, $fr \simeq 1$, and f varies only slightly during k iterations. Each invocation of Algorithm 2 costs $O(n+nf)+p\times Sort(\frac{nf}{p})$. An amount of work to find the shortest path becomes $(O(n+nf)+p\times Sort(\frac{nf}{p}))\times k$.

```
Input: adjacency lists for the subgraph with distances from
s and t (distance_s and distance_t).
Output: adjacency lists for the subgraph with updated
distances from s and t.
map(key/* unused */, adjacency_list) {
  parse adjacency_list to find vertex, adjacenct_vertices,
  distance_s and distance_t;
  remove vertex from adjacency_list;
  output (vertex, adjacency_list);
  if(distance_s = iteration_number) {
    output (neighbor, 's') for every element neighbor of
    adjacenct_vertices;
  }
  if(distance_t = iteration_number) {
    output (neighbor, 't') for every element neighbor of
    adjacenct_vertices;
  }
}
reduce(vertex, values) {
  new_distance_s = ∞;
  new_distance_t = ∞;
  while (values is not empty) {
    value = remove values' next element;
    if(value is 's') new_distance_s = iteration_number + 1;
    else if(value is 't') new_distance_t = iteration_number
    + 1;
    else adjacency_list = value;
  }
  parse adjacency_list to find distance_s and distance_t;
  if(distance_s < new_distance_s) new_distance_s = distance_s;
  if(distance_t < new_distance_t) new_distance_t = distance_t;
  replace distance_s and distance_t in adjacency_list with
  new_distnace_s and new_distance_t;
  output (vertex, adjacency_list);
}
```

Algorithm 2: Expanding breadth-first search frontiers from s and t using MapReduce. $iteration\_number$ (starts from 0) is passed as a command line argument.

The best sequential algorithm for breadth-first search runs in $O(n)$, and the work required to find the shortest path—by running breadth-first searches from both s and t with the heuristic of expanding the smaller frontier of the two—is even lower as the algorithm visits only a portion of the graph.

Retrieving the shortest path from the output is also problematic as the MapReduce programming model lacks a random access mechanism. In the RAM model, visiting at most k vertices and their neighbors in a backward direction towards s and t is sufficient to retrieve the shortest path. In the MapReduce programming model, in contrast, retrieving the shortest path requires additional MapReduce iterations and scanning the entire graph in each iteration. Thus, the MapReduce algorithm to find a single-pair shortest path is clearly suboptimal under our optimality criterion.

Bisection bandwidth is likely to become a bottleneck for large systems as volumes of communication is comparable or larger than amounts of work in the map phase, or $f \geq 1$. The overhead of the MapReduce runtime system and disk I/O exacerbates the situation as the workload incurs multiple MapReduce iterations with large volumes of the intermediate data. Cohen [11] presents several graph algorithms under the MapReduce programming model, and these algorithms also have large f and require multiple MapReduce iterations. Many of the algorithms require more work than the best sequential algorithm and necessitate a large bisection bandwidth to scale in large systems. The performance of

these algorithms may not be good on large systems with limited bisection bandwidth as the author also comments.

IV. A Highly Multithreaded System

Highly multithreaded systems support the shared memory programming model with the efficient latency hiding mechanism via multithreading and relatively large memory bandwidth and network bisection bandwidth for the system size. The shared memory programming model provides a randomly accessible global address space to the programmers.

A. Algorithm Level Analysis

Highly multithreaded architectures adopt the shared memory programming model with a randomly accessible global address space, which matches well with irregular data access patterns over entire data. A memory reference requires only a comparable number of instructions to the ideal RAM model. Memory access latency is higher than the ideal RAM model, but a large number of threads efficiently hide the latency assuming sufficient parallelism.

Bader *et al.* [4] provide a complexity model for highly multithreaded systems by extending Helman and JáJá's work [15] for symmetric multiprocessors (SMPs). This model encompasses the architectural characteristics of highly multithreaded systems. For SMPs, running time to solve a problem of size n with p processors, or $T(n,p)$ is expressed by the triplet $\langle T_M(n,p),\ T_C(n,p),\ B(n,p)\rangle$, where $T_M(n,p)$ is the maximum of a number of non-contiguous memory accesses by any processor, $T_C(n,p)$ is the maximum of local computational complexity of any processor, and $B(n,p)$ is a number of barrier synchronizations. This model penalizes non-contiguous memory accesses with higher latency and a large number of barrier synchronizations. Bader *et al.* [5] extend this model for multicore architectures as well. For highly multithreaded architectures, memory access latency is efficiently hidden, and a non-contiguous memory access costs $O(1)$. Multithreading also reduces $B(n,p)$. Bader *et al.* [4] conclude that considering only $T_C(n,p)$ is sufficient for highly multithreaded systems. An algorithm with $T_C(n,p)$ time complexity has at most $p \times T_C(n,p)$ work complexity.

B. System Level Analysis

Highly multithreaded systems are known to be efficient for applications with a large number of irregular data accesses [4], [6], [7], [17] and also lower programming burden to consider data locality. These are mainly due to relative large memory bandwidth—for single-node systems—or bisection bandwidth—for multi-node systems—to the system size in addition to the efficient latency hiding mechanism.

Traditional microprocessors with powerful integer and floating point execution units—but only one thread per core—suffer from low processor utilization for memory latency bounded workloads. Highly multithreaded architectures, in contrast, achieve high processor utilization for those workloads [6]. Thus, highly multithreaded systems can perform same amounts of computation using a smaller number of processors than a system with the traditional microprocessors. A typical highly multithreaded system consists of a small number of tightly integrated nodes with large memory bandwidth and network bisection bandwidth, and this addresses the communication issue in the memory bandwidth bounded applications with random access patterns as well. However, these systems have limited aggregate computing power, memory and disk capacity, and I/O bandwidth as a downside due to the small number of nodes in the system.

C. Finding Shortest Paths in the Subgraph

In order to extract the subgraph from the multiple terabytes of the input graph, we first need to store the input graph in a highly multithreaded system. For a single node system with a small number of disks, even storing the input data is not possible, and our problem cannot be solved. If the input graph fits into the capacity, then we can find shortest paths in the subgraph as the following.

1) Read the graph data from the disks and filter the edges in parallel and in a streaming fashion, store only the filtered edges in the main memory.
2) Transform the edge list with the filtered edges to the adjacency lists.
3) Run the single-pair shortest path algorithm optimized for highly multithreaded systems and complex networks (see [6]).

For the first step, the aggregate disk I/O bandwidth or the network bandwidth to the file server—when the input graph is stored in the separate file server—limits the data read rate. As typical highly multithreaded systems have limited disk I/O bandwidth or the network bandwidth to the separate file server, reading the entire input graph data takes a significant amount of time. Transforming the edge list to the adjacency lists is straightforward with the random access capability. Bader and Madduri [6] design an efficient algorithm to find the shortest path on the Cray MTA, and the SNAP package [8], which is portable to shared memory architectures, also provides a breadth-first search implementation that matches well with highly multithreaded architectures and complex networks. We tune this breadth-first search implementation to find a single-pair shortest path with the heuristic of expanding the smaller frontier (see [6]). This algorithm has the asymptotically same work complexity compared to that of the best sequential algorithm.

V. The Hybrid System

Analyzing a large scale complex network imposes distinct computational challenges (see Section II), which cannot be efficiently served with a MapReduce cluster or a highly multithreaded system on its own (see Section III and IV).

We design a novel hybrid system to address the computational challenges in large scale complex network analysis. Our hybrid system tightly integrates a MapReduce cluster and a highly multithreaded system and exploits the strengths of both in a synergistic way. In each step of the complex network analysis, we select a MapReduce cluster or a highly multithreaded system—based on the match between the computational requirements and the architectural capabilities—and perform the computation for the step. We transfer the data from one system to the other if we switch from one to the other in the consecutive steps, and the tight integration of the two systems reduces the data transfer time.

System	MapReduce cluster	highly multithreaded	hybrid
Nodes (# nodes, type)	(4, IBM System x3755)	(1, Sun SPARC T5120)	MapReduce cluster + highly multithreaded
Processors (# processors, type, power) / node	(4, AMD Opteron 2.4 GHz 8216, 95 W / processor)	(1, Sun UltraSparc T2 1.2 GHz, 91 W / processor)	
DRAM size / node	8 GB	32 GB	
DRAM bandwidth / node	42.8 GB/s	60+ GB/s	
Interconnect	Dual-link 1 Gb/s Ethernet	N/A	
Disk Capacity	96 disks $\times$ 1 TB / disk	2 disks $\times$ 146 GB / disk	
Software	Hadoop 0.19.2, Sun JDK 6	Sun Studio C compiler 5.9	

Table I: Technical specifications for the test platforms

A. Algorithm Level Analysis

Solving our problem, which finds single-pair shortest paths in the subgraph, requires filtering the larger input graph; transforming the edge list with the filtered edges to the adjacency lists; and running the shortest path algorithm on the subgraph. The MapReduce algorithm in Section III-C finishes the first two steps in a single MapReduce iteration, and this algorithm is optimal (under our optimality criterion for MapReduce algorithms) assuming $f \ll 1$. A highly multithreaded system, in contrast, suffers from limited disk capacity and the disk I/O bandwidth to finish the first step. For the last step, the MapReduce algorithm necessitates a significantly larger amount of work than the best sequential algorithm, and the disk I/O and runtime system overhead further exacerbates the performance. The extracted subgraph—in the compact representation—has higher chance to fit into the main memory of a highly multithreaded system, and if this is the case, a highly multithreaded system can find shortest paths in an efficient way. Using a MapReduce cluster for the first two steps and a highly multithreaded system for the last step exploits the strengths of both architectures, and we need to transfer the output of the second step from a MapReduce cluster to a highly multithreaded system in this case.

We generalize the above discussion to design a model that estimates the time complexity on the hybrid system. Assume the computation requires l steps. Then, execution time on the hybrid system is

$$T_{hybrid} = \Sigma_{i=1 \text{ to } l} \; min(T_{i,\ MapReduce} + \Delta, T_{i,\ hmt} + \Delta)$$

where $\Delta = \frac{n_i}{BW_{inter}} \times \delta(i-1, i)$. $T_{i,\ MapReduce}$ and $T_{i,\ hmt}$ are time complexities for the ith step on a MapReduce cluster and a highly multithreaded system, respectively. n_i is input data size for the ith step, and BW_{inter} is bandwidth between a MapReduce cluster and a highly multithreaded system. $\delta(i-1, i)$ is 1 if selected platforms for the $(i-1)$th and ith steps are different and 0 otherwise. As the input data resides in a MapReduce cluster, $\delta(0, 1)$ is 0 for a MapReduce cluster and 1 for a highly multithreaded system.

B. System Level Analysis

In the hybrid system, data transfer between a MapReduce cluster and a highly multithreaded system is necessary in addition to the computation on each. The hybrid system becomes more effective if this data transfer time is minimized, and the data transfer time reduces as BW_{inter} increases. Minimizing the distance between a MapReduce cluster and a highly multithreaded system and tightly coupling the two is important to provide large BW_{inter} and to maximize the effectiveness of the hybrid system.

C. Finding Shortest Paths in the Subgraph

We find shortest paths in the subgraph using the hybrid system through the following steps.

1) Extract the subgraph using the MapReduce cluster with the algorithms described in Section III-C.
2) Switch to the highly multithreaded system, and load the extracted graph from the MapReduce cluster to the highly multithreaded system's main memory.
3) Find shortest paths using the highly mutltithreaded system by running the algorithm described in Section IV-C multiple times.

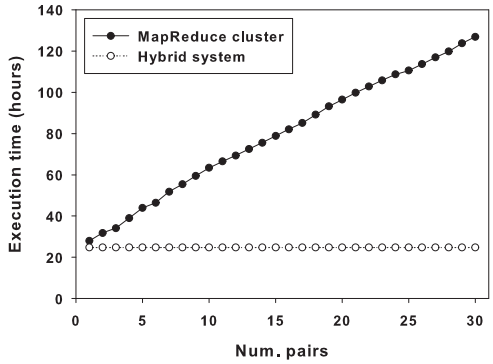
As $f \ll 1$ in our problem, the size of the extracted graph is much smaller than the input graph. Thus, the subgraph has higher chance to fit into the main memory of a highly multithreaded system. The MapReduce algorithm to solve the shortest path problem incurs significantly more work than the best sequential algorithm while the algorithm for a highly multithreaded system incurs only a comparable amount of work to the best sequential algorithm. For the graph that fits into the main memory, the disk I/O and runtime system overhead even widens performance gap between a MapReduce cluster and a highly multithreaded system. If the difference between execution time on a MapReduce cluster and a highly multithreaded system to find single-pair shortest paths is larger than the subgraph loading time, using a highly multithreaded system for the third step reduces the execution time to solve our problem.

VI. Experimental Results

Table I summarizes the test platforms. Hadoop is configured to spawn up to 8 mapper processes and 3 reducer processes per node. Hadoop Distributed File System (HDFS) is configured to create one replica per block. We add an additional system with the Intel Xeon processors to coordinate the map and reduce processes for the MapReduce cluster.

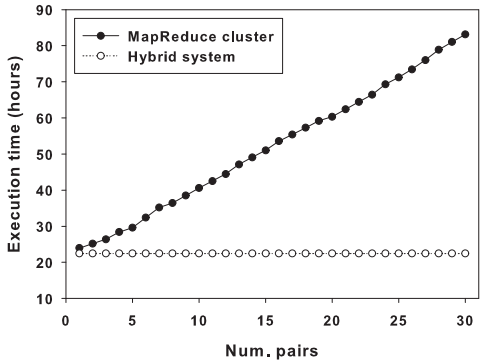
Figures 4, 5, and 6 summarize the experimental results in finding shortest paths in the subgraphs which cover 10%, 5%, and 2% of the vertices in the input graph, respectively. The extracted subgraphs have 351 million vertices and 2.75 billion undirected edges, 178 million vertices and 1.49 billion undirected edges, and 44.9 million vertices and 109 million undirected edges (excluding isolated vertices), respectively. We find single-pair shortest paths in the subgraphs for up to 30 pairs.

The MapReduce cluster successfully extracts the subgraphs, while the UltraSparc T2 blade fails to solve the problems on its own as the input graph size overflows the system's disk capacity. Using a network file system with a larger disk capacity is another option for the highly multithreaded



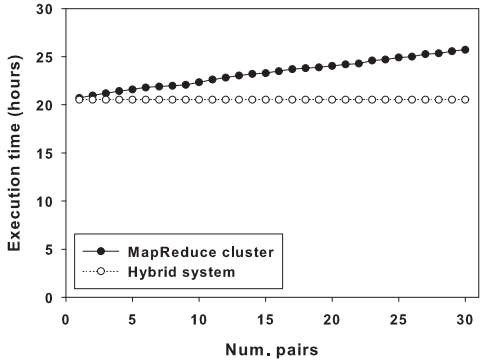
	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	23.9	23.9
memory loading	-	0.832
shortest paths (for 30 pairs)	103	0.000727

Figure 4: The execution time to extract the subgraph (which covers 10% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.



	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	22.0	22.0
memory loading	-	0.418
shortest paths (for 30 pairs)	61.1	0.000467

Figure 5: The execution time to extract the subgraph (which covers 5% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.



	MapReduce cluster (hours)	hybrid (hours)
subgraph extraction	20.5	20.5
memory loading	-	0.0381
shortest paths (for 30 pairs)	5.22	0.000191

Figure 6: The execution time to extract the subgraph (which covers 2% of the vertices) and find single-pair shortest paths (left) and the decompositions of the execution time on the MapReduce cluster and the hybrid system (right). The highly multithreaded system fails to solve the problem on its own as the input graph overflows its disk capacity.

system, but even in this case, the network bandwidth to the file system or the limited computing power of the highly multithreaded system is likely to become a performance bottleneck. Pre-processing the data before the transfer using a file system with a filtering capability—as is the case of the hybrid system—provides a better mechanism to work around these limitations. Finding single-pair shortest paths in the extracted subgraphs is significantly slower on the MapReduce cluster than the shared memory system. The hybrid system outperforms the MapReduce cluster in overall throughout the experiments. The performance gap widens as the subgraph size or a number of the input pairs to find a shortest path increases. Adopting a faster interconnection technology between the MapReduce cluster and the highly multithreaded

system will further widen the gap. This justifies the use of the hybrid system, especially when the analysis of the extracted subgraph requires a large amount of work.

The MapReduce cluster runs significantly slower than the highly multithreaded system in finding shortest paths. *This performance gap is mainly due to the match between computational requirements of the problem and the programming model and the architectural capability of the two different systems*. The MapReduce algorithm for the problem is clearly suboptimal under our optimality criterion while the algorithm for the highly multithreaded system executes only a comparable number of operations to the best sequential algorithm. The input pairs to find shortest paths are 6.04 hops away from each other in average (in the case of the 10% subnetwork, excluding the disconnected pairs) and the UltraSparc T2 blade visits approximately 100 thousand vertices in average—owing to the random access mechanism—with the heuristic of expanding the smaller frontier. The MapReduce algorithm needs to visit the entire vertices multiple times. The tightly connected network for the highly multithreaded system and the highly multithreaded system's latency hiding mechanism naturally match with the workload's irregular data access patterns, while the disk I/O and the runtime system overhead of the MapReduce cluster becomes more prominent with the graph data that fits within the main memory capacity.

VII. Conclusions and Future work

Performance and programming complexity is highly correlated with the match between the workload's computational requirements and the programming model and the architecture. Thus, a careful analysis of the workload's computational requirements and using the right tool in the right place have significant importance. We study the synergy of the hybrid system in the context of complex network analysis and show that the hybrid system can efficiently address these challenges, while a MapReduce cluster and a highly multithreaded system reveals the limitations on its own.

In this work, we use the highly multithreaded machine with a Sun UltraSparc processor and the 32 GB main memory. Even though we successfully fit the graph with 2.75 billion undirected edges into the 32 GB memory, there are larger complex networks that require more than 32 GB of the main memory—such as the entire Facebook network. Another massively multithreaded platform, the Cray XMT, provides terabytes of the main memory which is large enough to handle most practical complex networks. As future work, we will use the Cray XMT to analyze extreme scale complex networks.

Acknowledgments

This work was supported in part by NSF Grants IIP-0934114 and CNS-0708307 and Pacific Northwest National Laboratory Center for Adaptive Supercomputing Software for Multithreaded Architecture (CASS-MT). We acknowledge Sun Microsystems for their Academic Excellence Grant and donation of the UltraSparc systems used in this work.

References

[1] Welcome to Apache Hadoop!, 2009. http://hadoop.apache.org/core.

[2] R. Albert, H. Jeong, and A.-L. Barabasi. The diameter of the world wide web. *Nature*, 401:130–131, 1999.

[3] R. Albert, H. Jeong, and A.-L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 406:378–382, 2000.

[4] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, Oslo, Norway, Jun. 2005.

[5] D. A. Bader, V. Kanade, and K. Madduri. SWARM: A parallel programming framework for multi-core processors. In *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, Long Beach, CA, Mar. 2007.

[6] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, Aug. 2006.

[7] D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Computing*, 34(11):627–639, 2008.

[8] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Miami, FL, Apr. 2008.

[9] C. Bisciglia, A. Kimball, and S. Michels-Slettvet. Lecture 5: Graph Algorithms & PageRank, 2007. http://code.google.com/edu/submissions/mapreduce-minilecture/lec5-pagerank.ppt.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. SIAM Int'l Conf. on Data Mining (SDM)*, Lake Buena Vista, FL, Apr. 2004.

[11] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[12] L. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances In Physics*, 56(1):167–242, 2007.

[13] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th USENIX Symp. on Operating System Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proc. ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, Cambridge, MA, Aug. 1999.

[15] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *J. of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[16] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[17] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. Technical Report LBNL-1703E, Lawrence Berkeley National Laboratory, Apr. 2009.

[18] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.