RC25302 (WAT1208-065) August 22, 2012
Computer Science

IBM Research Report

Research Issues in Supporting Data Intensive Applications within an Exascale System

Abhirup Chakraborty, Dilma Da Silva
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Issues in Supporting Data Intensive Applications within an Exascale System

Abhirup Chakraborty [†], Dilma Da Silva [§]
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA 10598
[†]achakrab@us.ibm.com, [§]dilmasilva@us.ibm.com

ABSTRACT

Analyzing large graphs are crucial to a variety of applications domains, like personalized recommendations in social networks, search engines, communication networks, computational biology, etc. In these domains, there is a need to process aggregation queries over large graphs. Existing approaches for aggregation are not suitable for large graphs, as they involve multi-way relational over gigantic tables or repeated multiplication of large matrices.

In this report, we consider the top-K aggregation queries that involve identifying top-K nodes with highest aggregate values over their h-hop neighbors. We propose algorithms for processing such queries over large graphs in a shared nothing environment. We propose a hybrid algorithm that minimizes network loads is shuffling data across the processing nodes. The algorithm partitions the graph across the processing nodes, and uses Floyd-Warshall algorithm within each nodes. The nodes shuffles updates among themselves in iterative phases; such incremental iterative processing is similar to route discover in a large network. The algorithm needs only a few iterations to converge to an equilibrium state.

1. INTRODUCTION

Analyzing large graph data has seen renewed interest due to increased interests in a number of applications such as chemical data, biological data, XML data, social network data, communication networks, etc. In each of these applications, the underlying graphs are very big in size. There is current trends in advanced analysis of social network graphs aiming at evaluating the network values of customers [3, 6], link prediction [7] etc. In large graph analysis tasks all the vertices and edges of the entire graph are accessed multiple times in a random fashion. Examples include Page-rank [8], social network influence analysis [11], recommender systems [1], etc.

In the graphs relevant to social and biological networks, each node is often associated with an attribute set. The value of each attribute for a node represents some features of the entities represented by the node. For example, a node representing a Facebook user may have an attribute indicating his/her interest in a particular online game. Within the applications domains, there is a growing need to process standard queries efficiently over large graphs. For example, for each node find the aggregate value of an attribute for all its neighbor lying within h-hops in the graph. Such a query could identify the popularity of a game within one's social circle. Also, such queries could help finding the influential nodes (based on some attributes) in the social network and help in placing online advertisements.

2. PROBLEM

The aggregation queries over graphs are relevant to a number of emerging applications in online social communities such as book recommendation on Amazon, target marketing on Facebook, etc. These application can unified by a generalized aggregation query over a graph. In a graph $G(V, E)$, h-hop neighbors of a node v (i.e., aggregate score) can be given as

$$F(v) = \sum_{u \in S_h(v)} f(u)$$

Here, $S_h(v)$ is the h-hop neighbors of a node v, and $f(v)$ is a relevance function that assigns a score to each node. Top-k aggregation queries find the k nodes with the highest aggregate scores. Such a top-k aggregation query needs to solve three issues:

1. Evaluate individual relevance function $f(u)$ of a node. $f(u)$ could be as simple as 0/1 (e.g., if a user recommends a book or not), or it can be a classification function (e.g., how likely a user is a network expert), or simply a constant 1 (e.g., counting the number of neighbors).
2. Evaluate the aggregate score or collective strength of a node $F(u)$. Such an evaluation of an aggregation score of a node requires the identification of all nodes that lie within h-hops of the node u.
3. find the top-k nodes with the highest scores.

It is computationally expensive to perform aggregation over large graphs for various types of queries. If the average degree of nodes in the graph is m, the total number of edges to be accessed while computing h-hop aggregation for each node would be around $m^h|V|$. In applications with a large-scale graph(having billions of nodes) and high query workloads, such computational cost is prohibitive. Moreover, access patterns for the edges are usually random that causes high memory access overheads in a system. Also, for a large graph, it is impossible to store the whole graph in memory, which renders the aggregation over large scale graphs a challenging issue.

Using relational database queries to process aggregation over graph is very costly. If we choose to use tabular representations of graphs for processing $(h+1)$-hop aggregation queries, such an algorithm should compute h-way joins over the relational tables. For a large graph with billions of nodes such cost is prohibitive.

The issue of processing aggregate queries over graphs has been addressed in [12]. However, this approach does not consider problem of finding h-hop neighbors of the nodes; it assumes that the h-hop neighbors of the nodes are already computed. Their proposed algorithms only save computations in calculating the aggregate score of each node by pruning nodes selectively. We observe that computing h-hop neighbors is a crucial operation in the graph aggregation, and the aggregate scores can be precomputed or materialized during neighborhood discovery; therefore, allowing a separate phase for aggregate computations appears to be redundant. Moreover, the proposed algorithms work only in a single node, and does not consider the issue of distributing aggregate processing loads over a shared nothing system. Parallelizing or distributing processing loads of the transitive closure algorithm (i.e., Floyd-Warshall Algorithm [10]) over a shared nothing system is infeasible due to a large number of synchronization barriers. Reference [9] improves the Floyd-Warshall algorithm by proposing a blocking algorithm to reduce random memory accesses and to increase cache utilization. Reference [5] extends the Floyd-Warshall algorithm in a GPU-based system using a block-based algorithm; the algorithm attempts to increase the instruction execution throughput using the GPU. However, the approaches does not show significant performance gains even in the multi-threaded shared memory system.

In this report, we consider large-scale graphs that are too large to fit in the memory of one processing machine. We consider the issue of parallelizing the aggregation queries over a shared-nothing system. In such a system, as with the centralized scenario stated earlier, using matrix multiplication or relation joins is too costly to be affordable. Though h-hop neighbors of nodes in a graph can be found using Floyd-Warshall algorithm [10], such an algorithm is difficult to parallelize even within a single node due to large synchronization barriers. In this report, we propose a hybrid scheme to compute h-hop aggregation over graphs. Such an algorithm partitions the graph across the computing nodes. Within each partition or processing node the algorithm applies Floyd-Warshall algorithm to find the h-hop neighbors of the nodes. At inter-partition level, the algorithm uses a technique similar to that with route discovery in a large network. The algorithm proceeds in iterations; and at the end of each iteration, it propagates the updates (observed within a partition) to the relevant partitions.

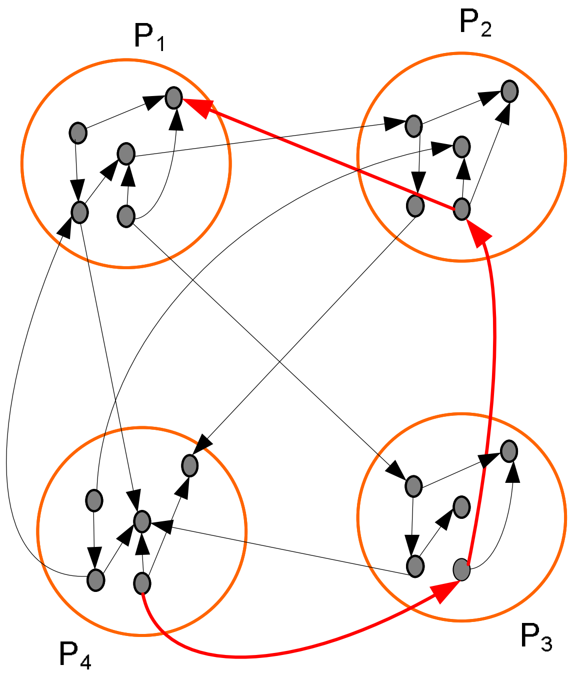

Figure 4: A partition-based hybrid approach to aggregation over a graph

3. THE HYBRID AGGREGATION ALGORITHM

As observed earlier, the naive way to answer h-hop aggregation over a distributed system is to process self joins or matrix multiplication (h-1) times. Such an approach is costly both in terms of computation and communication. We develop the hybrid approach that converges to equilibrium after a few iterations. This approach partitions the graph across the processing nodes. The algorithm proceeds in two phases: local-compute phase and incremental discovery phase. During the local-compute phase, each node applies Floyd-Warshall Algorithm locally over its partition. During an incremental discover phase, each partition (or node), sends the updates to relevant partitions.

3.1 Local-Compute Phase

In local-compute phase, each node uses Floyd-Warshall Algorithm to calculate the transitive closure within the partition with a given threshold(h). The algorithm updates or adds a neighbor, if its hop-count is bounded by the given threshold (h). The code for this algorithm is given in Figure 1. *adj_list* is the adjacency-based representation of partition. The array *arr* keeps the mapping from index to node ID. Here, Index i in the *adj_list* stores the neighbors of node arr[i] and their hop-counts.

3.2 Incremental Discovery Phase

In incremental discovery phase, each partition sends the recent updates to all the relevant partitions. Each partition maintains, for each other partitions, two data-structures: an *out-table* and an *in-node-list*. Within partition P, the *out-table* for partition Q stores all inter-partition(cut) edges that emanates from the partition P and are incident to

```
//apply Floyd Warshall algorithm locally
// for local nodes, insert   'index' values in the adjacency list
// hval = maximum hop length
void Partition::local_compute( int hval){
    int mik, mij, mkj;

    for(int k=0; k< this->N; k++){
        for(int i=0; i<this->N; i++){

            if(i==k) continue;
            for(int j=0; j<this->N; j++){
                if(j==i || j==k) continue;
                if(!adj_list->existPath(i, this->arr[j], &mij)){
                    mij = INT_MAX; // no path
                } //end if
                //if a path with a shorter hop is  found
                if( adj_list->existPath(i, this->arr[k], &mik) &&
                  adj_list->existPath(k, this->arr[j], &mkj) ){
                    if(mik+mkj<=hval && mik+mkj<mij)
                        adj_list->insert(i, this->arr[j], mik+mkj);
                }//end if
            } //end for
        }//end for
    }//end for
} //end method
```

Figure 1: Finding all-to-all transitive closure (based on a threshold) within a partition

partition Q. On the other hand, each partition P maintains one in-node-list for every other partition (Q). This list stores all nodes in partition Q that have an outgoing edge incident to any node within partition P. Therefore, while propagating updates, each partition should accumulate the updates/changes in the neighbors of the nodes in the in-node-list of other partitions, and send the updates to the respective partitions. Thus an incremental discovery phase should carry out an all-to-all propagate task that shuffles the updates across the participating nodes. Each node, after receiving updates from a remote partition processes to updates to find new neighbors or change their hop-counts. This task requires a join operation between received updates and the corresponding out-table, and changes the neighbors of all intra-partition node that might have a route to the nodes in update list. Figure 2 shows the all-to-all propagation of updates within the partitions. Updates are transferred between nodes using a bounded buffer to prohibit buffer overflow in case the aggregate updates across all the partitions become very large.

Incremental phase converges when no node in system has updates to send. This algorithm discovers all the neighbors with proper distance after once the convergence is achieved. Is should be noted that the maximum number of iterations in the incremental discovery phase is bounded by a parameter p. Suppose p' indicates the number of partitions spanned by a shortest path (with hop-count less than h) between any two nodes. The parameter p maximizes p' over all pairs of nodes within the whole graph. The maximum iterations before convergence is given by $log(p)$. Figure 4 shows the partitions of a graph with the inter-partition(cut) and intra-partition edges. Here, the shortest path (among all the node pairs) spanned by the maximum number of partitions is given by the path shown in red color. The path spans 4 partitions. So, the algorithm will converge after 2 iterations. Although there are $log(p)$ total iterations before the convergence, total volume of updates shuffled across the nodes will decrease drastically after the first few iterations, thus saving a significant network loads.

In the code *propagate_updates*, we use a ring-based algorithm as proposed in [2]. In exchanging data logs, we need to exchange meta-data (e.g., total number of update entries) and serialize the update list over MPI. At the receiving end, the node extract the updates and convert it to a list that can be joined with the respective out-table.

3.3 Top-K collection

Once the convergence of the iterative discovery phase is achieved, each node computes top-k results locally. To compute global top-K over all the nodes, we use a tree-based collection mechanism where all the nodes are organized in a binary tree structure. Each node receives top-K values from its two children and computes top-K nodes over the nodes received from the children and its own local top-K. The node then sends the final local top-K to its parent. The global top-K over the whole graph can be found at the root node. The code for in-network processing of top-K is given in figure 5.

4. BASE-LINE ALGORITHM

As a base-line algorithm over distributed nodes, we use top-K aggregation using relational joins. To process relational joins within distributed nodes, we use Cycle-join [4] algo-

```
int  Partition::alltoall_propagate__process( int hval){
    MPI::COMM_WORLD.Barrier(); // synchronization
    if(no updates to send in any partition) return 0;

      //for the first time, change_list is empty and get changes from adj_list
    if(this->iterations ==0)
        src_list = this->adj_list->entry;
    else src_list = this->change_list;

    this->change_flag=false; // reset
    rcv_rank = (this->rank-1)>=0? this->rank-1:this->procs-1;
    for(int i= (rank+1)%procs; i!= this->rank; i= (i+1)%this->procs){

        scount =0; rcv_flag = true; fill_flag=false;
        for(set<int>::iterator sit = this->in_node_list[this->mapper[i]].begin();
          sit!=in_node_list[this->mapper[i]].end(); ++sit){
            if(this->iterations==0)
                sbuf[scount++] = Entry(*sit, 0); // mark the node with a valid hop count(so that the
            else sbuf[scount++] = Entry(*sit, -1); // mark the node
            if(scount==BUF_S) fill_flag=true;
            for(map<int, int>::iterator mit=src_list[this->node_to_index(*sit)].begin();
              mit!=src_list[this->node_to_index(*sit)].end(); ++mit){
                if(!fill_flag){
                    sbuf[scount++] = Entry(mit->first, mit->second);
                    if (scount==BUF_S) fill_flag=true;
                } else{
                    MPI::COMM_WORLD.Send(&scount, 1, MPI::INT, i, TAGS::META);
                    req=MPI::COMM_WORLD.Isend(sbuf, scount, MIX, i, TAGS::DATA);

                    if(rcv_flag) { // receive from the remote process
                        MPI::COMM_WORLD.Recv(&rcount, 1, MPI::INT, rcv_rank, TAGS::META);
                        MPI::COMM_WORLD.Recv(&rbuf, rcount, MIX, rcv_rank, TAGS::DATA);

                        changes.insert(changes.end(), rbuf, rbuf+rcount); // insert into the vector list
                        if( rcount<BUF_S ) rcv_flag=false; // no data to receive from this MPI process
                    } //end if

                    req.Wait();
                    scount=0;
                    sbuf[scount++] = Entry(mit->first, mit->second); // BUF_S >1
                    fill_flag = false;
                } //end else
            } //end for
        } //end for
        //send the remaining buffer content
        MPI::COMM_WORLD.Send(&scount, 1, MPI::INT, i, TAGS::META);
        req=MPI::COMM_WORLD.Isend(sbuf, scount, MIX, i, TAGS::DATA);

        if(rcv_flag){ // receive the remaining data
            do{
                MPI::COMM_WORLD.Recv(&rcount, 1, MPI::INT, rcv_rank, TAGS::META);
                MPI::COMM_WORLD.Recv(&rbuf, rcount, MIX, rcv_rank, TAGS::DATA);
                changes.insert(changes.end(), rbuf, rbuf+rcount); // insert into the vector list
            } while(rcount==BUF_S);
        } //end if

        req.Wait();

        this->incr_discovery(this->mapper[rcv_rank], changes, hval);
        rcv_rank = (rcv_rank-1)>=0? rcv_rank-1:this->procs-1;
    } //end for
    this->iterations++;
    return 1;
} //end method
```

Figure 2: All-to-all propagation of updates within the partitions

```
void Partition::process_updates(int part,  vector<Entry> &changes, int hval ){
    vector<Edge>::iterator out_it;
    map<int, int> position;
    map<int, int>::iterator it_a, it_b;
    vector<Entry>::iterator it;
    int nodeI, index, nodeB;
    int node_id;
    bool flag, zero_flag;
    Entry ent, tmp_ent;
    int c, count;
        // create the structure to map from node to position in 'changes' list
    c=0;
    for(it=changes.begin(); it!=changes.end(); ++it, c++){
        if(it->hop == 0) position.insert( pair<int, int>(it->dest, c));
        else if(it->hop == -1) position.insert( pair<int, int>(it->dest, c+1));
    }//end for

    for( out_it=this->out_tables[part].begin(); out_it<this->out_tables[part].end(); ++out_it){
        nodeI = node_to_index(out_it->src); // index of 'src'  (in the current partition)
        it_a = position.find(out_it->dest);
        if(it_a==position.end()) continue; //not found

        count =0;
        for(int pp=it_a->second; pp<changes.size(); pp++){
            ent=changes.at(pp); //get the change entry
            if (ent.hop==-1) break; //end of the list for the 'dest' node
            if(ent.hop==0) count ++;
            if (count==2) break; // consider the first node with hop 0, but not the next one
            ent.hop++; // add one: to get the distance from  the nodeI
            if(ent.hop>hval) continue; // discard  distant  neighbors

            it_a = this->adj_list->entry[nodeI].find(ent.dest); // search for the node
            flag = this->adj_list->insert(nodeI, ent.dest, ent.hop);

            if (!flag) continue;
            add_change_list(nodeI, ent); //add to the change list
                 // propagate the changes
            for(it_a=this->adj_list->entry[nodeI].begin();
                   it_a!=this->adj_list->entry[nodeI].end(); ++it_a){
                //node_id = it_a->first; //get the node id
                index = this->node_to_index(it_a->first); //get the index
                //the node  not in current partition or  hop-count exceeds the threshold
                if(index<0 || (it_a->second+ent.hop)>=hval) continue;

                flag = this->adj_list->insert(index, ent.dest,  it_a->second+ent.hop);
                if(flag){ // inserted (the dest node)  or change in the hop count
                    ent.hop += it_a->second;
                    add_change_list(index, ent); //add to the change list
                }
            } //end for
        } //end for
    } //end for
} //end method
```

Figure 3: Processing the incoming updates during the local-discovery phase

```
void  Partition::in_network_topK(int kval){
    Entry  *left_topk, *right_topk;
    int  min_v;

    this->loc_topk=new Entry[kval];
    left_topk=new Entry[kval];
    right_topk=new Entry[kval];

    compute_local_topK(kval);
    print_loctopk(kval); // debug;
    if((this->rank*2+1) < this->procs ){ //receive from left child
        MPI::COMM_WORLD.Recv(left_topk, kval, MIX, this->rank*2+1, TAGS::RES);

        for(int i=kval-1; i>=0; i--){ // left_res
                //if a larger element encountered
            if( left_topk[i].metric > loc_topk[0].metric){
                loc_topk[0].node = left_topk[i].node;
                loc_topk[0].metric= left_topk[i].metric  ; //add the number
                heapify<Entry>(loc_topk, 0, kval-1);    //adjust the min-heap
            }  else break;  // stop scanning
        } //end for
    } //end if

    if((this->rank*2+2) < this->procs ){ // receive from right child
        MPI::COMM_WORLD.Recv(right_topk, kval, MIX, this->rank*2+2, TAGS::RES);

        for(int i=kval-1; i>=0; i--  ){ //right_res
            if( right_topk[i].metric > loc_topk[0].metric){
                loc_topk[0].node= right_topk[i].node;
                loc_topk[0].metric = right_topk[i].metric; // add the number
                heapify<Entry>(loc_topk, 0, kval-1);    //adjust the min-heap
            }  else break;
        } //end for
    }
    if(this->rank !=0) {
        MPI::COMM_WORLD.Send(loc_topk, kval, MIX, (int) ((this->rank-1)/2), TAGS::RES);
    }  else {
         output final top-K
    }
} //end method
```

Figure 5: In-network computation of top-k aggregates

rithm that partitions the dataset across the nodes. The nodes are organized in a ring-shaped network. Each node receives a partition from its predecessor node and joins with its local partition. Each node circulates the partitions received from its predecessor to its successor node. So, once a partition has traversed all the nodes in the system, that partition get joined with all the partitions of the dataset. For processing top-K aggregation with hop h, we invoke the join algorithm $h - 1$ times (i.e., (h-1)-way self-join). The code for processing joins is a distributed system using MPI is given in Appendix. To compute h-way join we invoke the join processing code h-times within a loop.

5. CONCLUSION

In this report, we present an approach to process aggregation over graphs that can be deployed over a large number of processing nodes. The hybrid algorithm eliminates multiple pass over local partition by using Floyd-Warshall algorithm. The algorithm incrementally propagates the updates in a separate phase. Such incremental propagation reduces network loads. Also, the algorithm reduces synchronization barriers by allowing each node proceed with the local computation independent of others. It is observed that though the number of iterations is bounded by the maximum partitions spanned by the shortest paths in the graph, the total volume of update data shuffled across the nodes would be very low. There exists a number of open issues that we plan to investigate further. Firstly, developing a partitioning algorithm that minimizes the parameter p (maximum number of partitions spanned by the shortest paths in the graph) is an open research issue. Such an algorithm would decrease total iterations before convergence, reducing the synchronization overheads in a system with a large number of processing nodes. Secondly, in a distributed environment with a large number of nodes, allowing synchronization barriers at the onset of each iteration would result in high overheads due to unbalanced computation within nodes or noises within computing nodes or networks. Thus, devising a lazy protocol for propagating updates eliminating barrier synchronization at the end an iteration is necessary for such a system. Such a lazy protocol will propagate or shuffle updates more frequently across partitions having larger number of intra-partition edges (i.e., higher out-table size) that those will lower out-tables sizes. Such an algorithm is feasible based on the redesign of the update propagation scheme. Also, devising a distributed partitioning algorithm to partition large graphs (i.e., tera bytes) is an open research issue.

APPENDIX

```
void   Base::alltoall_propagate_process(int hval){
    int rcv_rank, send_rank,*snum, *rnum, rcount, scount;
    vector<Edge> rcv_changes, send_changes;  //
    Edge sbuf[BUF_S], rbuf[BUF_S];
    MPI::Request req;

    rcv_rank = (this->rank-1)>=0? this->rank-1: this->procs-1;
    send_rank = (this->rank+1)%this->procs;
    send_changes = this->own_changes; // copy the local table
      // joins with its own table before  sending to 'send_rank'
    this->join_buffer(send_changes, hval);
    for(int iter=0; iter<this->procs-1; iter++){
        MPI::COMM_WORLD.Barrier(); // synchronization

        scount =0; rcv_flag = true;
        for(vector<Edge>::iterator vit = send_changes.begin(); vit !=send_changes.end(); ++vit){
            sbuf[scount].src= vit->src; // copy to the array
            sbuf[scount].dest = vit->dest;
            scount++;

            if(scount==BUF_S){
                MPI::COMM_WORLD.Send(&scount, 1, MPI::INT, send_rank, TAGS::META);
                req=MPI::COMM_WORLD.Isend(sbuf, scount, MIX, send_rank, TAGS::DATA);

                if(rcv_flag) { // receive from the remote process
                    MPI::COMM_WORLD.Recv(&rcount, 1, MPI::INT, rcv_rank, TAGS::META);
                    MPI::COMM_WORLD.Recv(&rbuf, rcount, MIX, rcv_rank, TAGS::DATA);
                     // insert into the vector list
                      rcv_changes.insert(rcv_changes.end(), rbuf, rbuf+rcount);
                    if( rcount<BUF_S ) rcv_flag=false;
                } //end if

                req.Wait();
                scount=0; //reset the scount
            } //end else
        } //end for
        //send the remaining buffer content
        MPI::COMM_WORLD.Send(&scount, 1, MPI::INT, send_rank, TAGS::META);
        req=MPI::COMM_WORLD.Isend(sbuf, scount, MIX, send_rank, TAGS::DATA);

        if(rcv_flag){ // receive the remaining data
            do{
                MPI::COMM_WORLD.Recv(&rcount, 1, MPI::INT, rcv_rank, TAGS::META);
                MPI::COMM_WORLD.Recv(&rbuf, rcount, MIX, rcv_rank, TAGS::DATA);
                rcv_changes.insert(rcv_changes.end(), rbuf, rbuf+rcount);
            } while(rcount==BUF_S);
        } //end if

        req.Wait();
        this->join_buffer( rcv_changes, hval); // joins with adj_list
        send_changes.clear();
        send_changes=rcv_changes; // now have to send 'rcv_changes' to 'send_rank'
        rcv_changes.clear(); // empty the receive buffer to be ready to receive
    } //end for
} //end method
```

Figure 6: Base-line algorithm for processing joins within distributed nodes