Graph Partitioning using Parallel Clustering for Improving Performance of Distributed Databases

Project Report

Abdul Quamar  
abdul@cs.umd.edu

Jayanta Mondal  
jayanta@cs.umd.edu

Ethar Elsaka  
eelsaka@cs.umd.edu

1 Introduction

In the recent years there has been an explosion of the amount of data associated with applications which can be represented as graphs, e.g., social network data, web graph data. Processing, querying, storing and programming such large size graphs poses significant challenges and scaling out has emerged as natural solution to address these challenges effectively. Scaling out involves deploying the graph on a distributed memory system and hence involves graph partitioning. An efficient partitioning of the graph would involve load balancing of the computation and minimization of communication among the distributed set of machines.

Graph partitioning is a much researched problem which has applications in many fields including VLSI, social network applications, database partitioning, large scale web analytics, etc. The graph partitioning problem can be stated as follows: Given the graph representation of a problem, mapping it onto parallel processing elements to exploit concurrency and at the same time guarantee efficient as well as load balanced execution. The applications can be expressed as a graph in terms of nodes and edges. Typically for such representations the nodes represent computation and the edges represent communication. The weights associated with the nodes capture the cost of computation while on the other hand the weights associated with the edges capture the cost of communication.

Graph partitioning under the dual constraints of load balancing and minimizing communication is a hard problem and thus necessitates the employment of algorithms which use heuristics to address the challenges associated with these optimizations. A majority of existing graph partitioning algorithms have been motivated from VLSI applications due to the extensive research and utility in the area. These tools have been optimized for performance and quality over several years and have taken the form of general purpose graph partitioning tools.

However in the recent years a large number of applications which can be represented as graphs have emerged which have different characteristics, structure, computation and communication requirements as compared to VLSI applications. Moreover the increase in the size of the graphs pose several scalability issues which are not addressed by the current partitioning tools. In this project we address the issue that the one size fits all model may not work for all applications and also explore ways to address the scalability issues involved with large sized graphs. We propose a parallel shingle based clustering approach which preserves the neighborhood information of the graph while reducing its size substantially. The reduced size of the graph can then be efficiently partitioned using an off-the-shelf partitioning tool. Thus our proposed solution provides the benefits of scalability in terms of handling large graphs, performance improvement by employing the benefits of parallelism and improved partitioning quality as a consequence of structure preserving shingle based clustering technique.

We demonstrate the effectiveness of our technique on a database partitioning application used for scaling out large databases. We represent the database and its query workload as a hypergraph and use hypergraph partitioning to create a distributed database across several machines such that the load is balanced across the machines and min-
imize the number of distributed transactions. Our results indicate that database partitioning can be substantially improved by using parallel clustering techniques which can capture the nature of the query workload and handle large graph sizes. Use of these techniques improves the quality of partitioning by minimizing the number of distributed transactions and also address scalability issues effectively.

The rest of the report is organized as follows: Section 2 provides an overview of graph partitioning and the existing techniques used for the same. It further provides details about database workload representation and partitioning for scaling large databases. Section 3 discusses the system design and architecture of our approach to graph partitioning. Section 4 and 5 give the implementation details and experimental results. Section 6 talks about related work in the area and finally Section 7 concludes with pointers to future work.

## 2 Graph partitioning overview

For an efficient and load balanced system design, graph partitioning would involve partitioning data represented as a graph onto several parallel processing elements such that the computation on these elements is evenly balanced and the communication between the partitions is minimized to maximize the gains of parallelism reduce the overheads to a minimum. The k-way balanced min-cut of a graph gives k balanced partitions and the min-cut ensures that the communication between the k partitions is minimized. Since the k-way min-cut partitioning of a graph is known to be NP-Hard the existing partitioning tools such as the Metis family if partitioning tools use heuristics to compute the k-way min-cut.

The existing graph partitioning tools use a multilevel partitioning technique which employs different flavors of refinement algorithms such as the Kerninghan-Lin (KL) algorithm and the Feduccia-Mattheyses (FM) algorithm which employ recursive bisection techniques and node swapping for minimizing the cut. These refinement algorithms require the graph data to be memory resident and thus can operate on graph sizes that can fit into the memory for efficient computation. To address this issue the graph partitioning algorithms coarsen the input graph in several steps until the size is suitable for operation of algorithms such as KL and FM. Figure 1(a) shows the distinct phases of multilevel partitioning algorithms i.e., the coarsening phase, initial partitioning phase and refinement phase.

The coarsening techniques used by the existing partitioning tools are motivated to suit the needs of the VLSI design community which extensively uses them. The Metis family of algorithms use a maximal matching technique for edge coarsening. The algorithms determine a maximal matching of edges using various techniques and collapse the edges to coarsen the graph. The different flavors of coarsening used are edge coarsening, hyperedge coarsening and modified hyperedge coarsening as shown in Figure 1(b). The problem of using such edge coarsening techniques is that they might lose the structural information inherent in the graph representation of application data which might be significant for many applications such as social network applications, large scale web graph analytics, database partitioning etc. and hence might benefit from different types of coarsening which can preserve structural information of the underlying graph.

So the question we address in this project is: Can we use other coarsening techniques for graph partitioning to suit the needs of different applications? The intuition behind this idea is to provide a certain degree of flexibility in the choice of graph coarsening or compression technique to improve the quality of partitioning. Another issue related to graph partitioning is the scaling of partitioning tools to handle large graphs. The current partitioning tools such as parMetis use a parallel partitioning technique on multiple processors to address this issue. However it uses the same maximal matching edge coarsening technique for graph compression and hence suffers from the same problem of loss of structural information leading to a poorer quality partitioning different applications.

To address the issues of good quality graph partitioning and scaling of partitioning tools to handle large graphs we have explored the use of a parallel shingle based clustering technique. The shingles determine node similarity using immediate neighbor information and hence preserve the structural information in the underlying graph. The parallelism in the technique helps it to scale to handle large graph
 sizes. We apply our technique for database partitioning for improving the performance of distributed databases.

2.1 Database Partitioning

Horizontal partitioning of databases to create a shared nothing distributed architecture has emerged as predominant mechanism for scaling out large databases and tapping the benefits of parallel query processing. However for transactional workloads, horizontal partitioning leads to distributed transactions which require a distributed consensus protocol and are therefore costly. It has been reliably ascertained that the number of distributed transactions need to be minimized in order to improve response time and throughput.

An approach to minimize the number of distributed transactions is to use query workload aware partitioning. This involves firstly to determine the database access patterns using the query workload, secondly creating a graphical representation of the query workload and finally obtaining a partitioning that minimizes the number of distributed transactions. We choose to represent the query workload as a hypergraph as firstly the cut of the hypergraph corresponds to the number of distributed transactions and secondly a hypergraph partition provides a better quality partition which we have substantiated experimentally and included the results for the same in the experimental result section.

**Representation of query workload:** The query workload can be represented as a hypergraph, $H = (V, E)$, where the nodes are the tuples or data items and each (hyper)edge $e \in E$ corresponds to a query in the workload which spans over a set of nodes $V' \subseteq V$. Figure 2 shows an illustrative example, where we have 4 queries over 6 data items, each of which is represented as a hyperedge over the data items. The query workload is represented by $Q = \{e_1 : \{v_1, v_2, v_3\}, e_2 : \{v_2, v_3\}, e_3 : \{v_3, v_5, v_6\}, e_4 : \{v_4\}\}$. The graph can be modeled as a weighted graph. The edge weight represents the number of transactions that span over a set of tuples and the node weight is a function of tuple size and the frequency of access of the tuple.

![Figure 2: Workload representation using a Hypergraph](image-url)
tecture details of our proposed parallel clustering based graph partitioning technique which we employ to carry out effective database partitioning.

3 System Design

The system architecture of our approach is depicted by the workflow shown in Figure 3. The input query workload hypergraph is partitioned into blocks and processed over multiple processors. Each processor computes the shingles for the nodes assigned to it in the graph and clusters them based on the shingle values using a clustering algorithm. The output of the different processors is merged together to form the coarsened hypergraph which is then partitioned using hMetis to obtain a partitioning of the coarsened graph. The partitioning is then refined in the unwrapping phase to obtain the partitioning of the original hypergraph.

3.1 Shingle based clustering

We employ a shingle based ordering heuristic algorithm to coarsen the input graph to explore the properties of locality and similarity captured by the shingle ordering heuristic. In earlier work on compressing social networks [3], Ravi Kumar et al. have shown that using shingle based ordering they are able to compress different types of social networks efficiently. The heuristic is based on obtaining a fingerprint of the neighbors of each node and ordering the nodes according to this fingerprint. The fingerprints express the similarity between nodes through the Jaccard coefficient measure $J(A,B) = |A \cap B| / |A \cup B|$, a natural notion of similarity of two sets. Let $\sigma$ be a random permutation of the elements in $A \cup B$. For a set $A$, let $M\sigma(A) = \sigma^{-1}(\min_{a \in A} \sigma(a))$, the smallest element in $A$ according to $\sigma$ is called the shingle. The intuition behind the heuristic is to treat the neighbors of a node $u$ as a set and compute the shingle of this set for a suitably chosen permutation. The nodes in the graph can then be ordered by the shingles. By the property stated above, if two nodes have significantly overlapping out neighbors, i.e., share a lot of common neighbors, then with high probability they will have the same shingle and hence be close to each other in a shingle-based ordering. Similarly, a set of nodes will be close to each others in this ordering if they share many of their neighbors.

In order to calculate the shingles for each node in the graph, a number of pairwise-independent hash functions are used. The number of hash functions is equal to the number of shingles. For each node in the graph, each hash function is applied to all its neighbors, and the neighbor with lowest hash value is selected as the shingle of this node. Performing that for all hash functions, we obtain the required number of shingles for each node. The input to this function is a graph, and the output is a list of graph nodes with its corresponding shingles sorted by shingles. The detailed algorithm to compute the shingles is given in Algorithm 1.

The clustering of the graph is obtained using vertex collapsing based on the shingle ordering. The idea is to collapse or cluster the vertices that have similar shingle values together, and collapse the edges accordingly. The shingle ordering and clustering phases run in a decentralized fashion leading to significant parallelism.

Input : A graph $G$, A family of pairwise-independent hash functions $H$
Output: The shingles of every node in the graph $G$

for $v \in V(G)$ do
  \[v\] ← $\phi$;
  for $h \in H$ do
    $s$ ← $\arg\min_{n \in \text{Neighbors}(v)} h(n)$;
    \[v\] ← \[v\] $\cup s$;
  end
end

return \[v\];

Algorithm 1: The algorithm to compute the shingles of the graph nodes

3.2 Parallel clustering problem formulation

The aim of the problem decomposition is to parallelize the computation of the shingles of each node and the node clusters. Our input is a hypergraph that represents the graph workload. This hypergraph is represented by a list of edges. In the problem decomposition phase, the graph nodes are
equally distributed across number of threads. Each thread is individually responsible of computing the shingles of each node and the node clusters based on shingle values. Clustering is done by applying the K-Means clustering algorithm. The output of the shingle ordering and clustering phase is a group of clusters equal to the number of threads. Each group contains the clusters of their corresponding nodes which is then used to generate the coarsened graph.

4 Implementation details

This section provides the implementation details of the various states of our parallel clustering and partitioning framework.

4.1 Parallel shingle based clustering implementation

Our parallel shingle based clustering implementation is based on a shared memory model. We use Java threads to implement this type of parallelism. As opposed to the serial implementation, where the shingles for the nodes are calculated for one node-at-a-time followed by the clustering of the entire set of nodes, in the parallel implementation, each thread is assigned a group of graph nodes and it computes their shingles and does clustering in parallel with other threads. The k-means clustering algorithm uses the euclidean metric to compute the distance between the data points and the centroids. To achieve load balancing, we divide the number of graph nodes equally on the number of threads. The operation of calculating shingles for the graph nodes is very suitable to be implemented in parallel because shingles of every node is totally independent of other nodes and at the same time it is a read-only operation, and therefore communication and synchronization cost is virtually zero.

4.2 Constructing the coarsened graph from the clusters

Once we have the clustering done at each processors, we are set to join them to construct the final coarsened graph wherein we treat each cluster as a node and add hyperedges between these nodes conforming to the original hypergraph. We first assign a unique global id to all the clusters generated from different processors. These unique ids are the final node ids in the coarsened hypergraph. We also maintain a node to global cluster id mapping for each partition.

Figure 3: Shingle based parallel partitioning flow
Once, that is done we go through the original hypergraph one hyperedge at a time and map these to construct the hyperedges for the coarsened graph. The process is straight-forward. For a hyperedge in the original graph we look at the node ids and using the node to cluster mappings created in the previous step we create a hyperedge in coarsened graph using those cluster ids as node ids. It's important to note here that for each hyperedge in the original graph we have a hyperedge in the coarsened graph. While building the coarsened graph we also maintain a hash of whether any hyperedge is being repeated or not (this will happen as the coarsened graph has much lower number of nodes). If any hyperedge is repeated, an appropriate weight is assigned to the hyperedge, the weight being the number of times it has been repeated. When hMetis does the hypergraph partitioning of the weighted graph it takes into account the edge weights and reports the partitioning and min-cut value accordingly.

5 Experimental results

In this section, we present a comprehensive experimental evaluation using our prototype system. We present our findings under three broad headings. Section 5.3 describes the experiments which provide a comparison of the workload representation as a graph Vs a hypergraph. Section 5.4 provides details of the speedup achieved by our parallel clustering implementation as we vary different parameters like number of clusters, number of shingles etc. Finally in section 5.5 we compare our technique with hMetis and parMetis with respect to runtime as well as quality of partitioning.

5.1 Dataset

We have used the TPC-C benchmark to generate the query workload. We have installed PostgreSQL 8.4 for running the benchmark and used the postgres logs to extract the queries executed by the benchmark. We have transformed the benchmark queries to enable us to extract the primary keys of the tuples accessed by each query. We have created a mapping mechanism to map each set of extracted primary keys to a unique tuple-ID. We use this information to generate a hypergraph with queries representing hyperedges and nodes are the tuple-IDs accessed by the query. The hypergraph was converted to an equivalent clique and we used its adjacency list representation as an input the shingle based clustering algorithm.

5.2 Experimental setup

We ran our experiments on a shared memory machine Intel machine with Linux kernel version 2.6.18. The machine has 48 GB of RAM (shared memory), 12 MB total cache size, 6 nodes each with a two processors, making it a 12 processor machine.

5.3 Comparison of workload representation: Hypergraph Vs Graph

The query workload for a database system can be represented as a hypergraph intuitively wherein the hyperedges represent queries and the tuples accessed by the query represent the nodes in the hypergraph. Alternatively the query workload can be represented as a graph wherein each query is represented by a clique in the graph and the nodes in the clique represent the tuples accessed by the query. The partitioning quality for each representation is determined by the number of distributed transactions. For a hypergraph the min-cut directly corresponds to the number of distributed transactions. However for a graph, the min-cut simply gives the edge count across the cut. Therefore the number of distributed transactions needs to be determined using an algorithm which counts the number of transactions spanning more than one partition. We have written a java program which implements this algorithm.

We carried out detailed experiments to ascertain the quality of partitioning obtained by partitioning a hypergraph using hMetis and an equivalent graph using Metis. We generated various size of hypergraphs form the query load and converted them into equivalent graphs for our experiments. Figure 4 shows the results of our experiments which ascertain the fact hypergraph partitioning using hMetis consistently provides a better quality of partition in terms of the min-cut as compared to graph partitioning using Metis.

Although the running time of Metis is much smaller as compared to hMetis there are two reasons
that favor hypergraph representation of the query workload. First, the quality of partitioning using a hypergraph is much better which would improve the run time performance of queries as compared to graph partitioning. Second, since partitioning is an off-line process the increased run time for hypergraph partitioning would not hurt the run time performance of query processing. In addition to this generating a hypergraph from the query workload trace is extremely efficient ($O(n)$) and simple as compared to generation of an equivalent graph (clique) representation which much more complex ($O(n^2)$) and time consuming which offsets the benefits of faster partitioning.

5.4 Speed up of parallel clustering

In this section we first analyze the effect of number of clusters and the number of shingles per data point on clustering time. Subsequently we provide details for the speed up obtained for our parallel clustering implementation as we vary the number of processors.

**Varying the number of clusters:** For these set of experiments we have kept the number of processors fixed to 12 processors and used 5 shingles per node. Figure 5 shows the effect on the runtime of the clustering algorithm as we increase the value of $k$ in our $k$-means algorithm. The runtime increases significantly with the increase of number of clusters. This was expected, considering the inherent nature of the $k$-means algorithm which takes more time to cluster as the number of clusters increases as it has more number of points to iterate.

**Varying the number of shingles:** Figure 6 shows the impact of the number of shingles on the runtime. For these set of experiments we have used a fixed number of processors (12) and clusters (50) per processor. The results show that as we increase the number of shingles the runtime increases which is commensurate to our expectations as the $k$-means algorithm considers each shingle as a separate dimension of the data points. For example if we use $d$ different shingles then each node could be represented as a $d$-dimensional data-point. Thus it is trivial to see that as we increase $d$ i.e. the number of shingles the computation time for $k$-means also increases accordingly.

**Varying the number of processors:** To determine the speedup achieved by our parallel shingle based clustering technique we vary the number of processors. The other parameters have been kept fixed (the number of shingles per node: 5; total number of clusters: 120). To study the strong scaling characteristics of our implementation, we keep the problem size in terms of the total number of clusters fixed (120) and compute the number of clusters per processor $p$ as $120/p$. This is fed as input to the $k$-means algorithm to compute the appropriate number of clusters.

Figure 7 shows the results of the speedup achieved...
by our technique. As we can see, our parallel implementation achieves a super-linear speedup with the increase in the number of processors (potentially due caching of data and instructions at the processor level). A linear speedup was expected as our problem decomposition was embarrassingly parallel. There is no communication across the processors. The cut edges between processors (during decomposition) are taken into account when building the coarsened graphs from the clusters. Since for the boundary nodes, we do not consider the neighboring nodes on other processors for the shingles computation, one might argue that for these nodes the shingle calculation is not accurate. However, since the shingles computation is itself a heuristic, the return on investment would have been minimal as it would have increased the communication and synchronization costs significantly. The precise cost-benefit analysis of considering the cut-edges, has been left as future work.

5.5 Comparison with hMetis and parMetis

In this section draw a comparison of our techniques with hMetis and parMetis with respect to runtime and quality of partitioning achieved as we vary different parameters of the system.

Varying the number of clusters: Figure 8 shows the quality of the cut achieved as we vary the number of clusters compared to the cuts given by hMetis and parMetis. It is easy to see that as we increase the number of clusters the partitioning quality increases (min-cut decreases). This is intuitive to see, as when we increase the number of clusters we are going towards a more fine grained representation of the initial graph (i.e. with more nodes) and hence the min-cut decreases with the increase in the number of clusters. On the other hand, the coarser the graph, the more the loss of significant structural information leading to an inferior quality of partitioning.

Comparing the min-cut results with those obtained by direct partitioning using hMetis and parMetis, we find that hMetis provides a much better min-cut. This is expected as it operates on the entire graph without any loss of structural information. Technically, hMetis is similar to our implementation where the total number of nodes is equal to the total number of clusters (finest granularity). On the other hand it was surprising to see that parMetis provides a much worse quality of partitioning. This is partly because of the fact that parMetis works on an equivalent graph and not the hypergraph, and we have seen that hypergraph produces better quality cuts. The second reason may be attributed to the fact that parMetis might lose some structural information while decomposing the problem. However it is difficult to reason accurately on the quality of partitioning obtained by parMetis in the absence of information on the problem decomposition algorithms used by parMetis.
A key point to note here is that even with a significant amount of compression we do not lose much on the quality of partitioning in terms of the variation of min-cut. Even at 80% compression the variation of min-cut is under 15% as compared to hMetis. Thus the significant take away is that we can take significant advantage of graph compression without sacrificing much in terms of quality of partitioning which makes our partitioning scheme scalable in handling large graph sizes while minimizing the associated overheads.

Varying the number of shingles: In the next experiment we have compared the min-cut quality of our approach with hMetis and parMetis as we vary the number of shingles. Figure 9 shows the results for this experiment. It was surprising to see that our approach provides better quality cuts for less number of shingles and the min-cut increases as we increase number of shingles. This could be explained by analyzing the shingle heuristic in a little more detail. Shingle ordering gives an ordering of the nodes according to neighborhood similarity. If there is a tie in the shingle value using the first shingle, we use a second shingle to break the ties. And similarly is there is a tie using the second shingle we use the third one and so on. However in our k-means clustering set up we do not prioritize the shingles thereby diluting the importance of the first and subsequent shingles. A potential fix could be assigning weights to the shingles with the first shingles getting the highest weight and then running the k-means clustering algorithm. Intuitively, that was the reason why one shingle performed the best (i.e it was similar to assigning 100% weight to the first shingle and 0% weight to the rest.)

Partitioning time comparison: Finally, we compare the runtime of our technique with hMetis and parMetis. Figure 10 shows the related results. Here runtime for our technique includes the time of problem decomposition, parallel clustering, coarsened graph construction and hMetis partitioning time of the coarsened graph. On the other hand for parMetis we only have reported the equivalent graph partition time. The comparison is biased in favor of parMetis as it operates on an equivalent graph and it is known that graph partitioning is much faster that hypergraph partitioning (almost 1000 times faster for a equivalent graph for a given hypergraph). Moreover the runtime here does not include the substantial amount time required to generate the equivalent graph representation for parMetis.

The results show that parMetis gives the fastest partitioning time, however as discussed earlier it gives a much worse quality of partitioning. On the other hand it is interesting to see that our approach runs much faster than hMetis on the original hypergraph, as it uses parallel clustering for compressing the graph and then uses hMetis to partition...
6 Related Work

The scale-out of large databases to meet the growing demands of efficient processing and querying of data requires partitioning of data onto multiple machines to afford parallelism. The speed-up achieved depends on the partitioning scheme and the amount of communication and coordination required between the partitions while processing queries online. Curino et al. [4] propose a workload aware approach: Schism for database partitioning and replication which aims at minimizing the number of distributed transactions while balancing the workload distribution amongst the partitions. Schism uses a graph for the representation of the query workload which is subsequently partitioned to minimize the distributed transactions and balance the load. Graph partitioning and replication to minimize cuts has been looked into extensively by the VLSI research community. Hwang et al. [5] and El Gamal et al. [6] analyse the components of a k-way partition and determine replication sets to minimize the cuts further, which in-turn helps minimize the circuit layout. Nehme et al. [7] have developed a system to automatically partition the database based on the expected workload. Their approach is tightly integrated with the query optimizer which relies on database statistics.

Most of the graph partitioning based systems discussed so far try to minimize an objective function such as the min-cut or minimizing the number of cross edges while trying to come up with a set of partitions. Our approach differs from these as we use a parallel approach for shingle based clustering of the graph using local information available to each process involved in the parallel computation. We then use those clusters to come up with a graph that represents the workload and carry out min-cut partitioning of the graph. There are a number of graph partitioning tools that use heuristics to efficiently compute a k-way balanced min-cut partitioning of a hypergraph such as hMetis [1] and PaToH [2]. We have used hMetis to carry out the hypergraph partitioning of our coarsened graph.

7 Conclusion and Future work

In this project we have explored alternatives to graph coarsening techniques used by standard graph partitioning algorithms, to suit the needs of different applications. In particular we have used a parallel shingle based clustering technique for graph coarsening which when used in conjunction with hMetis provides a scalable alternative to existing graph partitioning algorithms. We find that hypergraph coarsening using shingle based clustering is an effective technique which improves scalability by reducing the graph size and associated overheads and provides a much better quality partitioning by capturing the neighborhood similarity information in the graph.

Our technique reduces partitioning time considerably without significantly compromising on the partitioning quality. For database workload aware partitioning, we have achieved a partitioning quality better than parMetis for up to about 60% graph compression which is a significant achievement con-
sidering that parMetis has been a much optimized tool for a considerable amount of time. On the other hand hMetis provides a better quality partition but is not scalable for handling large size graphs and has a considerably larger partitioning time which makes it unsuitable for frequent repartitioning as the query workload changes. We note that there is further scope of improvement by refining the clustering algorithm and tuning to application needs which we leave as future work.

References


