Very Large Graph Partitioning by Means of Parallel DBMS

Constantin S. Pan and Mikhail L. Zymbler

South Ural State University, Chelyabinsk, Russia

Abstract. The paper introduces an approach to partitioning of very large graphs by means of parallel relational database management system (DBMS) named PargreSQL. Very large graph and its intermediate data that does not fit into main memory are represented as relational tables and processed by parallel DBMS. Multilevel partitioning is used. Parallel DBMS carries out coarsening to reduce graph size. Then an initial partitioning is performed by some third-party main-memory tool. After that parallel DBMS is used again to provide uncoarsening. The PargreSQL's architecture is described in brief. The PargreSQL is developed by authors by means of embedding parallelism into PostgreSQL open-source DBMS. Experimental results are presented and show that our approach works with a very good time and speedup at an acceptable quality loss.

1 Introduction

Nowadays graph mining plays an important role in modeling complicated structures, such as chemical compounds, protein structures, biological and social networks, the Web and XML documents, circuits, etc. Being the one of the topical problems of graph mining, graph partitioning is defined as follows [9]. Given a graph $G = (N, E)$, where N is a set of weighted nodes and E is a set of weighted edges, and a positive integer p, find p subsets N_1, N_2, ..., N_p of N such that

- $\cup_{i=1}^{p} N_i = N$ and $N_i \cap N_j = \emptyset$ for $i \neq j$,
- $W(i) \approx W/p, i = 1, 2, \ldots, p$, where $W(i)$ and W are the sums of the node weights in N_i and N, respectively;
- the *cut size*, i.e. the sum of weights of edges crossing between subsets is minimized.

The usual way to do partitioning is through several recursive steps of bisection. So our approach was aimed at this particular kind of partitioning problem.

A very large graph, comprising of billions vertices and/or edges, is the most challenging case of partitioning because the graph being partitioned and all the intermediate data does not fit into main memory.

In this paper we introduce an approach to partitioning of very large graphs by means of parallel relational database management system (DBMS). The rest

© Springer-Verlag Berlin Heidelberg 2013

of the paper is organized as follows. Section 2 briefly discusses the related work. Section 3 provides a short intro to PargreSQL parrallel DBMS that we use for the graph partitioning. Section 4 presents our approach. The results of experiments are shown in section 5. Section 6 contains concluding remarks and directions for future work.

2 Related Work

A significant amount of work has been done in the area of graph-based data mining, including graph partitioning problem [1].

The classical algorithm based on a neighborhood-search technique for determining the optimal graph partitioning is proposed in [16]. Multilevel approach to graph partitioning is suggested in [14]. There are many sophisticated serial and parallel graph partitioning algorithms have been developed [9]. One of the first parallel graph partitioning algorithms based upon multilevel approach was proposed in [15]. An approach to graph partitioning based upon genetic alorighms investigated in [17]. In [7] authors suggested graph partitioning algorithm without multilevel approach.

A parallel graph partitioner for shared-memory multicore architectures is discussed in [28]. Parallel disk-based algorithm for graph partitioning is presented in [29]. In [26] distributed graph partitioning algorithm is suggested. Cloud computing-based graph partitioning is described in [6].

There are software packages implementing various graph partitioning algorithms, e.g. Chaco [12], METIS and ParMETIS [13], KaFFPa [25], etc.

In [3] ParallelGDB, a system to handle massive graphs in a shared-nothing parallel system is described. Pregel [20] is a Google's distributed programming framework, focused on providing users with a natural API for programming graph algorithms.

Existing graph data mining algorithms typically face difficulties with respect to scalability (storing a graph and its intermediate data in main memory). Mining by means of relational DBMS and SQL allows to overcome main memory restrictions. This approach supposes bringing mining algorithms to data stored in relational database instead of moving stored data to algorithms of third-party tools. Using relational DBMS for mining we have got all its services "for free": effective buffer management, indexing, retrieval, etc. However it demands uneasy "mapping" of graph mining algorithms onto SQL with cumbersome source code.

In recent years a number of graph mining algorithms have been implemented using relational DBMS. In [22] a scalable, SQL-based approach to graph mining (specifically, interesting substructure discovery) is presented. A framework for quasi clique detection based upon relational DBMS is described in [27]. Frequent SQL-based subgraph mining algorithms proposed in [5,10]. Database approach to handle cycles and overlaps in a graph is investigated in [2,4]. We also have to mention non-DBMS oriented but disk-based system for computing efficiently on very large graphs described in [18].

Despite the benefits of using a DBMS, it still could be incapable of effectively processing *very large* graphs. Our contribution is applying a *parallel* DBMS to graph partitioning (what was done for the first time, to the best of our knowledge).

3 PargreSQL Parallel DBMS

Currently, open-source PostgreSQL DBMS [24] is a reliable alternative for commercial DBMSes. There are many research projects devoted to extension and improvement of PostgreSQL. PargreSQL DBMS is a version of PostgreSQL adapted for parallel processing. PargreSQL utilizes the idea of *fragmented parallelism*[1] [8] in cluster systems (see fig. 1). This form of parallelism supposes horizontal fragmentation of the relational tables and their distribution among the disks of the cluster.

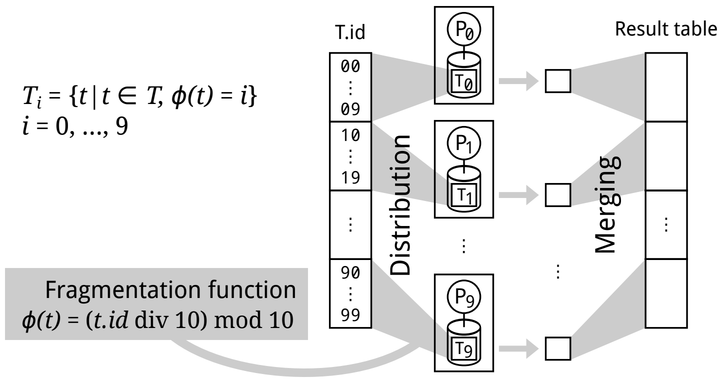


Fig. 1. Fragmented parallelism

The way of the table fragmentation is defined by a *fragmentation function*, which for each record of the table returns the ID of the processor node where this record should be placed.

A query is executed in parallel on all processor nodes as a set of parallel *agents*. Each agent processes its own fragment and generates a partial query result. The partial results are merged into the resulting table.

The PargreSQL architecture (see fig. 2) implies modifications to the source code of existing PostgreSQL's subsystems as well as implementation of new subsystems to provide parallel processing.

Modified PostgreSQL's engine instance is installed on every node of a cluster system. The *par_Storage* subsystem provides fragmentation and replication of tables among the disks of the cluster. The *par_Balancer* subsystem is responsible

[1] Also known as *partitioned parallelism*, but that term is not used in the paper to avoid possible confusion with *graph partitioning*.

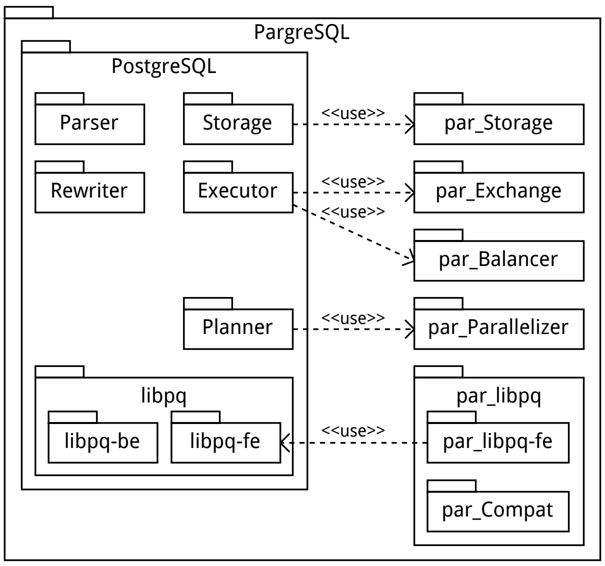


Fig. 2. PargreSQL architecture

for load balancing of the engine instances. The *par_Exchange* subsystem provides the EXCHANGE operator [11,19], which is in charge of data transmission among PostgreSQL's engine instances during the query execution and encapsulates all the parallel work. The *par_Parallelizer* subsystem inserts EXCHANGEs into appropriate places of the query execution plan made by PostgreSQL's engine instance. The *par_libpq* provides an API that is transparent to PostgreSQL applications.

4 Applying PargreSQL to Graph Partitioning

This section describes an approach to bisecting of very large graphs with PostgreSQL parallel DBMS. *Bisection* of a graph is a basic case of graph partitioning problem ($p = 2$, partitioning into two sets). Multi-way partitioning is performed by recursively applying bisection.

Graph partitioning with PargreSQL is depicted in fig. 3. We use the multilevel partitioning scheme [14], which includes three steps.

On the first step, *coarsening*, we reduce the input graph to the size that existing graph partitioning tools could deal with. A relational table (list of edges) that represent the graph is split into horizontal fragments and PargreSQL executes SQL-queries which reduce its size. During the coarsening step we collapse the heaviest edges of the graph, reducing excessive vertices and edges and aggregating their weight. Coarsening can be repeated multiple times to get smaller graphs. This step is implemented in PargreSQL by means of representing the input graph as a relational table and quering this table using SQL.

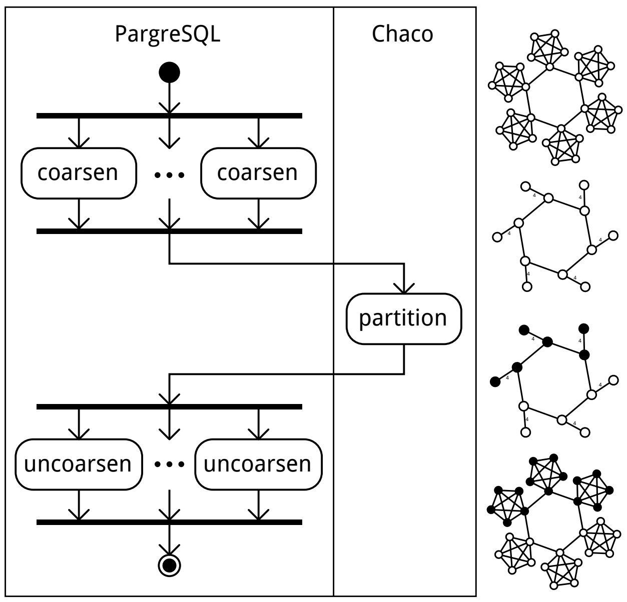


Fig. 3. Graph partitioning scheme

On the second step, *initial partitioning*, we export the graph from the database and feed it to a third-party graph partitioning utility (Chaco [12]) which performs the initial partitioning using one of the well-known algorithms (Kernighan-Lin, inertial, spectral, etc.). The result of the initial partitioning is imported into the database as a relational table containing the list of the graph's vertices with their partitions assigned.

Table 1. Data structure for graph partitioning

Relational table	Description
GRAPH(A, B, W)	The original fine graph A, B: edge's ends, W: weight
MATCH(A, B)	The matching of the fine graph A, B: edge's ends
COARSE_GRAPH(A, B, W)	The coarse graph A, B: edge's ends, W: weight
COARSE_PARTITIONS(A, P)	The partitions of the coarse graph A: vertex, P: partition
PARTITIONS(A, P, G)	The partitions of the fine graph A: vertex, P: partition, G: gain

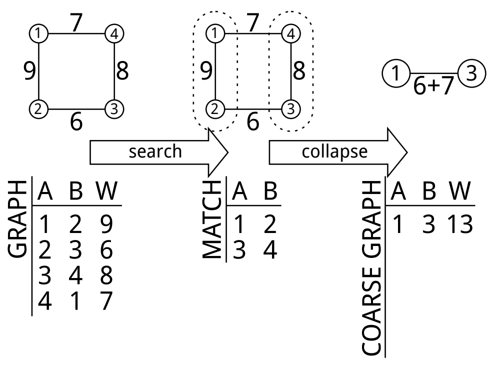
On the final step, *uncoarsening*, we refine the coarse results using another sequence of SQL queries. This step is repeated as many times as the coarsening was performed. During this step we first map coarse results on the larger graph, and then apply a local optimization to improve the results. PargreSQL calculates a table of two columns: vertex and its partition.

Since we apply a relational DBMS, our algorithms suppose that the data is stored in relational tables (see tab. 1). These tables are uniformly distributed among cluster nodes by means of $\psi(t) = \lfloor \frac{t.A \times n}{|E|} \rfloor$ fragmentation function, where n is the number of nodes in the cluster and $t.A$ is used as the fragmentation attribute. Every instance of the PargreSQL's engine processes its own set of fragments of these tables.

Since our approach does not implement the initial partitioning itself, we will discuss further only the coarsening and uncoarsening steps.

4.1 Coarsening

We divided the coarsening problem into two subproblems: **find a matching** and **collapse the matching** (see fig. 4).



GRAPH

A	B	W
1	2	9
2	3	6
3	4	8
4	1	7

MATCH

A	B
1	2
3	4

COARSE GRAPH

A	B	W
1	3	13

Fig. 4. An example of **coarsening**

During the **find** substep we search for the heaviest matching (independent edge set) in the graph. Since we use a simple greedy algorithm for that, the matching could turn out not to be actually the heaviest. This matching is stored into a database table as a list of edges for later use.

The **collapse** substep implements the removal of all the edges found on the previous substep. The edges in question get collapsed into vertices (see fig. 5).

4.2 Uncoarsening

The uncoarsening step is also implemented inside PargreSQL as a series of relational operations. The implementation consists of three substeps: **propagate the partitions**, **calculate the gains** and **refine the gains** (see fig. 6).

```
select least(newA, newB) as A, greatest(newA, newB) as B, sum(W) as W
from (
  select
    coalesce(match2.A, GRAPH.A) as newA,
    coalesce(MATCH.A, GRAPH.B) as newB,
    GRAPH.W
  from
    GRAPH, left join MATCH on GRAPH.B=MATCH.B
    left join MATCH as match2 on GRAPH.A=match2.B)
where newA<>newB group by A, B
```

Fig. 5. The **collapse** substep implementation

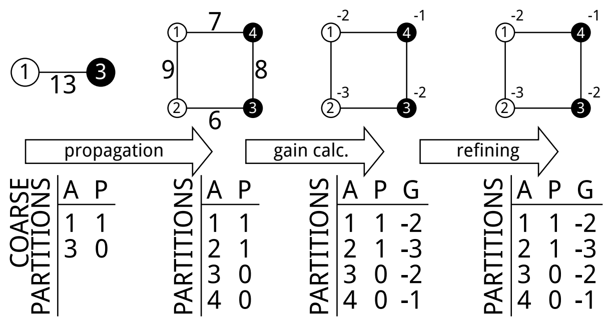


Fig. 6. An example of **uncoarsening**

The idea of the **propagate** step is to map coarse partitions onto the finer version of the graph. This is the opposite of the **collapse** substep of the coarsening process. The matching saved earlier gets used here to uncollapse the edges and mark their ends as belonging to the corresponding partitions (see fig. 7).

```
select a, p from COARSE_PARTS
union
select match.b, part.p
from MATCH as match, COARSE_PARTS as part
where match.a = part.a
```

Fig. 7. The **propagate** substep implementation

Right after the partition propagation we could end up with a nonoptimal solution. So we need to know, which vertices should go to the opposite partition. This is calculated for each vertex as the *gain*, which is the difference between the total weight of the edges connecting the vertex with ones from the other partition and the total weight of the edges connecting the vertex with ones from the same partition.

The gain is calculated as

$$\text{gain}(v) = \text{ext}(v) - \text{int}(v),$$

where

$$\text{ext}(v) = \sum_{(v,u)\in E, P(v)\neq P(u)} w(v,u),$$

$$\text{int}(v) = \sum_{(v,u)\in E, P(v)=P(u)} w(v,u).$$

Basically, the gain means how much better the overall solution would be if we excluded this vertex v from its current partition $P(v)$. In case $\text{gain}(v) > 0$ we would want to move v to the opposite partition. The gain calculation implemented in PL/pgSQL as shown in fig. 8.

```
select PARTITIONS.A, PARTITIONS.P, sum(subgains.Gain) as Gain
from
  PARTITIONS left join (
    select GRAPH.A, GRAPH.B,
     case when ap.P = bp.P then -GRAPH.W
      else GRAPH.W end as Gain
    from
    GRAPH left join PARTITIONS as ap on GRAPH.a = ap.A
    left join PARTITIONS as bp on GRAPH.b = bp.A
 ) as subgains
  on PARTITIONS.A = subgains.A or PARTITIONS.A = subgains.B
group by PARTITIONS.A, PARTITIONS.P
```

Fig. 8. The gain calculation

```
select * from PARTITIONS
where
  P = current and
  G = (
    select max(G)
    from PARTITIONS
    where P = current)
limit 1
into V
```

(a) Vertex picking

```
update PARTITIONS
  set G = G + W * (case when P = V.P then 2
   else -2 end)
from (
  select case when A = V.A then B else A end,
  W from GRAPH
  where B = V.A or A = V.A) as neighbors
where neighbors.A = PARTITIONS.A;

update PARTITIONS
  set G = -G, P = 1 - P
where A = V.A;
```

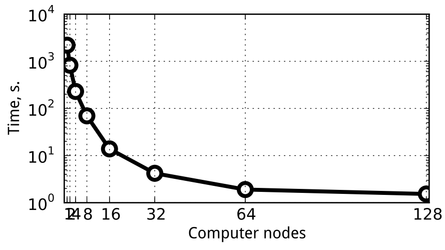
(b) Vertex moving

Fig. 9. Refining

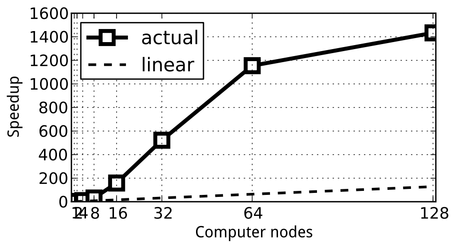
During the **refine** substep we switch back and forth between the partitions, each time picking a vertex which has the largest positive gain and moving it to the opposite partition. This repeats until we cannot find such vertices any more. This method is based on the heuristic proposed in [16]. Its implementation in PL/pgSQL is shown in fig. 9.

5 Experiments

We have conducted a series of experiments using SKIF-Aurora supercomputer of South Ural State University [21]. We tried to partition a street network graph[2] of 10^5 vertices and we got an embarassingly parallel implementation where every fragment of the graph's database represenation gets processed independently of the other fragments. Since the problem has the complexity of $\underline{O}(n^3)$, this kind of "trick" gives us the superlinear speedup that you can see on fig. 10, but at a price of reduced accuracy of the results.



(a) Total execution time



(b) Speedup with PargreSQL

Fig. 10. Execution time with PargreSQL

Relative execution time of coarsening and uncoarsening did not show any dependency on configuration (see fig. 11).

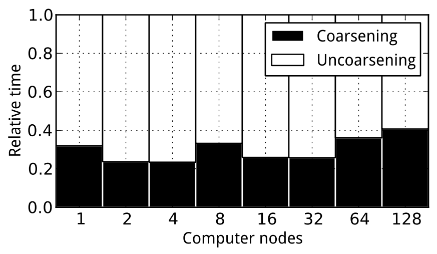
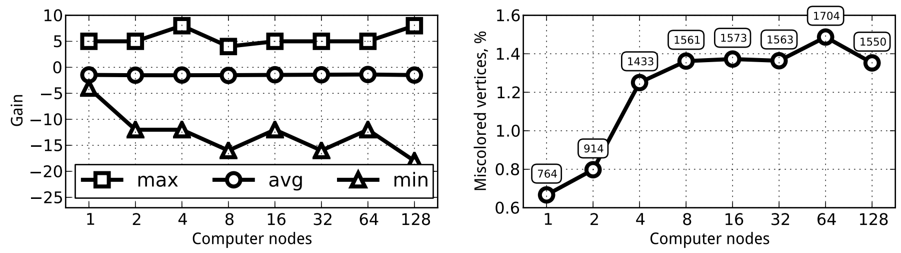


Fig. 11. Relative execution time

[2] Luxemburg street map from `http://www.cc.gatech.edu/dimacs10/archive/streets.shtml`



(a) Statistics on the gain values (b) Amount of miscolored vertices

Fig. 12. The quality of the partitioning

We have investigated exactly how much worse the results would get as we increased the number of computer nodes in the system. The statistics on the quality of partitioning is shown in fig. 12.

6 Conclusions

The paper introduces an approach to partitioning of very large graphs, comprising of billions of vertices and/or edges. Most of the existing serial and parallel algorithms suppose that the graph being partitioned and all the intermediate data fit into main memory, so they cannot be applied directly for very large graphs.

Our approach assumes using PargreSQL parallel relational DBMS. A very large graph is represented as a relational table (list of edges). PargreSQL DBMS carries out the coarsening of the graph. The coarsened graph and the intermediate data generated during the process fit into main memory, so its initial partitioning could be performed by some third-party tool (e.g. Chaco). In the end PargreSQL performs the uncoarsening of the coarse partitions.

PargreSQL is implemented on the basis of PostgreSQL open-source DBMS. PargreSQL utilizes the idea of fragmented parallelism and implies modifications in the source code of the existing PostgreSQL's subsystems as well as implementation of new subsystems to provide parallel processing.

Because of using a DBMS our approach will work even in cases when traditional tools may fail due to memory limits. Parallel query processing provides us with a very good time and speedup of graph partitioning at an acceptable quality loss.

In the future we would like to explore applying PargreSQL to other problems connected with mining very large graphs from different subject domains and try some more sophisticated partitioning schemes, not only the bisection.

Acknowledgment. The reported study was partially supported by the Russian Foundation for Basic Research, research projects No. 12-07-31217 and No. 12-07-00443.

References

1. Aggarwal, C.C., Wang, H.: Managing and Mining Graph Data, 1st edn. Springer Publishing Company, Incorporated (2010)
2. Balachandran, R., Padmanabhan, S., Chakravarthy, S.: Enhanced DB-subdue: Supporting subtle aspects of graph mining using a relational approach. In: Ng, W.-K., Kitsuregawa, M., Li, J., Chang, K. (eds.) PAKDD 2006. LNCS (LNAI), vol. 3918, pp. 673–678. Springer, Heidelberg (2006)
3. Barguñó, L., Muntés-Mulero, V., Dominguez-Sal, D., Valduriez, P.: *ParallelGDB*: a parallel graph database based on cache specialization. In: Desai, B.C., Cruz, I.F., Bernardino, J. (eds.) IDEAS, pp. 162–169. ACM (2011)
4. Chakravarthy, S., Beera, R., Balachandran, R.: DB-subdue: Database approach to graph mining. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 341–350. Springer, Heidelberg (2004)
5. Chakravarthy, S., Pradhan, S.: DB-FSG: An SQL-based approach for frequent subgraph mining. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 684–692. Springer, Heidelberg (2008)
6. Chen, R., Yang, M., Weng, X., Choi, B., He, B., Li, X.: Improving large graph processing on partitioned graphs in the cloud. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC 2012, pp. 3:1–3:13. ACM, New York (2012)
7. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.F.: Graph partitioning with natural cuts. In: IPDPS, pp. 1135–1146. IEEE (2011)
8. DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. Commun. ACM 35(6), 85–98 (1992)
9. Fjallstrom, P.: Algorithms for graph partitioning: A survey (1998)
10. Garcia, W., Ordonez, C., Zhao, K., Chen, P.: Efficient algorithms based on relational queries to mine frequent graphs. In: Nica, A., Varde, A.S. (eds.) PIKM, pp. 17–24. ACM (2010)
11. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: Garcia-Molina, H., Jagadish, H.V. (eds.) SIGMOD Conference, pp. 102–111. ACM Press (1990)
12. Hendrickson, B.: Chaco. In: Padua (ed.) [23], pp. 248–249
13. Karypis, G.: Metis and parmetis. In: Padua (ed.) [23], pp. 1117–1124
14. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: ICPP (3), pp. 113–122 (1995)
15. Karypis, G., Kumar, V.: A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. J. Parallel Distrib. Comput. 48(1), 71–95 (1998)
16. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal 49(1), 291–307 (1970)
17. Kim, J., Hwang, I., Kim, Y.-H., Moon, B.R.: Genetic approaches for graph partitioning: a survey. In: Krasnogor, N., Lanzi, P.L. (eds.) GECCO, pp. 473–480. ACM (2011)
18. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood (October 2012)
19. Lepikhov, A.V., Sokolinsky, L.B.: Query processing in a dbms for cluster systems. Programming and Computer Software 36(4), 205–215 (2010)
20. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Elmagarmid, A.K., Agrawal, D. (eds.) SIGMOD Conference, pp. 135–146. ACM (2010)

21. Moskovsky, A.A., Perminov, M.P., Sokolinsky, L.B., Cherepennikov, V.V., Shamakina, A.V.: Research Performance Family Supercomputers 'SKIF Aurora' on Industrial Problems. Bulletin of South Ural State University. Mathematical Modelling and Programming Series 35(211), 66–78 (2010)
22. Padmanabhan, S., Chakravarthy, S.: HDB-subdue: A scalable approach to graph mining. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 325–338. Springer, Heidelberg (2009)
23. Padua, D.A. (ed.): Encyclopedia of Parallel Computing. Springer (2011)
24. Pan, C.: Development of a parallel dbms on the basis of postgresql. In: Turdakov, D., Simanovsky, A. (eds.) SYRCoDIS. CEUR Workshop Proceedings, vol. 735, pp. 57–61. CEUR-WS.org (2011)
25. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
26. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: Bader, D.A., Mutzel, P. (eds.) ALENEX, pp. 16–29. SIAM/Omnipress (2012)
27. Srihari, S., Chandrashekar, S., Parthasarathy, S.: A framework for SQL-based mining of large graphs on relational databases. In: Zaki, M.J., Yu, J.X., Ravindran, B., Pudi, V. (eds.) PAKDD 2010, Part II. LNCS, vol. 6119, pp. 160–167. Springer, Heidelberg (2010)
28. Sui, X., Nguyen, D., Burtscher, M., Pingali, K.: Parallel graph partitioning on multicore architectures. In: Cooper, K., Mellor-Crummey, J., Sarkar, V. (eds.) LCPC 2010. LNCS, vol. 6548, pp. 246–260. Springer, Heidelberg (2011)
29. Trifunovic, A., Knottenbelt, W.J.: Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning. In: IPDPS. IEEE Computer Society (2004)