Resource Scheduling for Parallel Query Processing on Computational Grids

Anastasios Gounaris Rizos Sakellariou Norman W. Paton Alvaro A.A. Fernandes

Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK
E-mail: {gounaris,rizos,norm,alvaro}@cs.man.ac.uk

Abstract

Advances in network technologies and the emergence of Grid computing have both increased the need and provided the infrastructure for computation and data intensive applications to run over collections of heterogeneous and autonomous nodes. In the context of database query processing, existing parallelisation techniques cannot operate well in Grid environments because the way they select machines and allocate tasks compromises partitioned parallelism. The main contribution of this paper is the proposal of a low-complexity, practical resource selection and scheduling algorithm that enables queries to employ partitioned parallelism, in order to achieve better performance in a Grid setting.

1. Introduction

Grid technologies have enabled the development of novel applications that require close and potentially sophisticated interaction and data sharing between resources that may belong to different organisations. Examples include bioinformatics labs across the world sharing their simulation tools, experimental results, and databases; as well as the use of the donated spare computer time of thousands of PCs connected to the Internet in order to solve computation intensive problems. The maturity of database technologies and their widespread use has led to many proposals that try to integrate databases with Grid applications (e.g., Spitfire (http://eu-datagrid.web.cern.ch/eu-datagrid/), OGSA-DQP [1]). In particular, query processors for Grid-enabled databases, such as [1], can provide effective declarative support for combining data access with analysis to perform non-trivial tasks, and are well suited for intensive applications as they naturally provide for parallelism. This is due to the fact that many complicated tasks or subtasks in a workflow can be effectively encapsulated and specified by database queries. However, for the efficient exploitation of parallelism in such query processors, one of the most challenging problems to be solved is the selection and scheduling of the resources that will participate in a potentially parallel query evaluation from a vast and heterogeneous pool.

In query processing, a query in a declarative language (typically SQL or OQL) is transformed into a query plan by successive mapping steps through well-established calculi and algebras. A query plan is represented by a tree-like directed acyclic graph (DAG), whose vertices denote basic query operators and its edges represent dataflow. Evaluation can be speeded up by processing the query plan in parallel, usually transparently to the user. The three classical forms of parallelism in database query processing are *independent*, *pipelined* and *partitioned (or intra-operator)*. Independent parallelism can occur if there are pairs of query subplans, in which one does not use data produced by the other. Pipelined parallelism covers the case where the output of an operator is consumed by another operator as it is produced, with the two operators being, thus, executed concurrently. In partitioned parallelism, a physical operator of the query plan has many clones, each of them processing a subset of the whole data. This is the most profitable form of parallelism, especially for data and/or computation intensive queries.

The resource scheduling problem in databases for the Grid is the problem of (i) choosing resources and (ii) matching subplans with resources. The problems of defining the execution order of subplans and exploiting pipelined parallelism are addressed by adopting well-established execution models, such as iterators [5], and thus, need not be part of query schedulers. Existing scheduling algorithms and techniques, either from the database or the Grid or the parallel research communities, seem inadequate for parallel query processing on the Grid basically because the way they select machines and allocate tasks compromises partitioned parallelism in a heterogeneous environment. For example, generic DAG schedulers (e.g., [11, 9]), and their Grid variants tend to allocate a graph vertex to a single machine, which leads to no partitioned parallelism. More comprehensive proposals (e.g., GrADS [3]) still rely on application-dependent "mappers" to map data and tasks to resources, and thus come short of constituting complete scheduling algorithms. Excellent proposals for mixed-parallelism

scheduling (e.g., [7]) and parallel database scheduling (e.g. [4]), are restricted to homogeneous settings. Our proposal effectively addresses the resource scheduling problem for Grid databases in its entirety, allowing for arbitrarily high degrees of partitioned parallelism across heterogeneous machines, by leveraging and adjusting existing proposals in a practical way. The practicality of the approach lies in the fact that it is not time-consuming, it is effective in environments where the number of available resources is very large, it is dependable, and minimises the impact of slow machines or connections.

The remainder of the paper is structured as follows: Firstly, we present the problem followed by our proposed solution in Section 2. This solution is then evaluated in Section 3. We conclude in Section 4.

2. A practical query scheduler for the Grid

2.1. Problem Description

It is well understood that, even in homogeneous systems, choosing the maximum degree of parallelism not only harms the efficiency of resource utilisation, but can also degrade performance. This holds for heterogeneous systems as well. However, the problem of resource scheduling on the Grid is actually more complicated than choosing the correct degree of parallelism. Grid schedulers should decide not only how many machines should be used in total, but *exactly which these machines are*, and which parts of the query plan each machine is allocated. Note that the related and common problem of devising optimal workload distribution among the selected machines is out of the scope of this paper. Another difficulty has to do with the efficiency of parallelisation, which is of significant importance especially when the available machines belong to multiple administrative domains and/or are not provided for free. Thus, the aim is, on one hand to provide a scheduler that enables partitioned parallelism in heterogeneous environments with potentially unlimited resources, and on the other hand to keep a balance between performance and efficient resource utilisation. As the problem is theoretically intractable, effective and efficient heuristics need to be employed.

2.2. Solution Approach

The complexity of the problem of resource selection and scheduling on Grids justifies resorting to heuristics, as an exhaustive search for all the possible combinations of machines, workload distributions and query subplans is an obviously inefficient solution. An acceptable solution will be one that can scale well with the number of machines that are available. The algorithm proposed here starts from a valid query plan with minimum partitioned parallelism, and thus, such a query plan is unlikely to perform well for intensive computations. Traditionally, performance is improved by increasing the partitioned (or intra-operator) parallelism. This algorithm increases that by one degree (i.e., one additional machine) for one part of the query plan at a time, in line with [7]. After each step, it selects an available machine, allocates it to a particular query subplan and checks the predicted improvement in performance. If there is no improvement, or the improvement is below a certain threshold, it stops. The threshold is of considerable importance. The smaller the threshold, the closer the final point to the optimal point will be. However, this comes at the expense of higher compilation time. A bigger threshold may force the algorithm to terminate faster, but also to stop returning a number of nodes, which yields a final response time that can be improved much more, although it can still be much smaller than the initial.

The scheduler proposed requires a decoupled cost model which (i) assigns a cost to a parallel query plan, and (ii) assigns a cost to every physical operator of that query plan. Any such cost model is suitable, as the scheduler is not based on any particular one, following the approach of [3]. By decoupling the cost model and the scheduler algorithm, enhancements in both these parts can be developed and deployed independently. The cost model is also responsible for defining the cost metric, with query completion time being a typical choice.

2.3. The algorithm

The algorithm receives a query plan which is partitioned into subplans that can be evaluated on different machines. Each of the operators of the query plan is scheduled on at least one machine. After this initial resource allocation, which is at the lowest possible degree of partitioned parallelism, it enters a loop. In that loop, the algorithm estimates the cost of the query plan and of each physical operator of the query plan individually. Then, it takes the most costly operator that can be parallelised, and increases its degree of parallelism if that increase improves the performance of the query plan above a certain threshold. When no more changes can be made for that operator, the algorithm re-estimates the cost of the plan and the operators in order to do the same for the new most costly operator. The loop exits when no changes in the parallelism of the most costly operator can be made. *Exchange* operators encapsulate the parallelism and involve communication [5], and *operation-calls* are used to encapsulate user defined functions [1]. *Scans* entail I/O cost, *projects* denote data pruning, and the rest of operators, such as *joins*, incur computation cost. Thus, all kinds of cost (i.e., CPU, I/O, communication) and their combinations can be considered.

The inputs to the algorithm are:

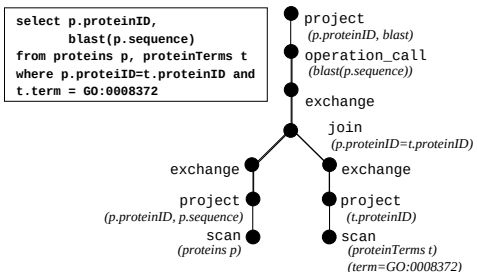


Figure 1. An example query plan.

repeat
  $Operator\ Op = getCostlierParallelisableOp()$
  $Criteria[]\ L = getCriteria(Op)$
  $float\ a = getThreshold()$
  repeat
    $Machines[]\ M = getAvailMachines()$
    $checkMoreParallelism(Op, M, L, a)$
  until no changes
until no changes

Figure 2. The steps of the scheduling algorithm after the initial resource allocation.

- A partitioned single-node optimised plan, with *exchanges* placed before *attribute sensitive* operators (e.g., joins) and *operation-calls* (see Fig. 1). *Attribute sensitive operators* are those that, when partitioned parallelism is applied, the data partitioning among the operator clones depends on values of specific attributes of the data.

- A set of candidate machines. For each node, certain characteristics need to be available. The complete set of these characteristics depends on the cost model and its ability to handle them. However, a minimum set that is required by the algorithm consists of the available CPU power, the available memory, the I/O speed, the connection speed, and proximity information with regard to the data and computational resources employed in the query. Such metadata can be provided by MDS [2] and NWS [12], as in GrADS.

- A threshold a, referring to the improvement in performance. This improvement is caused by transformations of the query plan. The improvement ratio is given by $\frac{t_{old} - t_{new}}{t_{old}}$, where t_{old} and t_{new} are the time costs before and after the transformation respectively. The cost model is responsible for computing these costs. The partitioned parallelism is increased only when the improvement ratio is equal to or greater than the threshold.

The algorithm consists of two phases. In the first phase, a query plan without partitioned parallelism is constructed. The resource allocation in this phase is mostly driven by data locality. E.g., the *scans* are placed where the relevant data reside and the *joins* are placed on the node of the larger input, unless more memory is required [8]. As there already exists a significant number of proposals for resource scheduling without partitioned parallelism (e.g., [6]), this phase is not covered in detail.

In the second phase, which is the main contribution of this work in its own right, the most costly operator that has a non empty set of candidate machines is selected. The set of candidate machines consists of the nodes that (i) are capable of evaluating the relevant operator, (ii) have not yet been assigned to that operator, and (iii) have either been started up, or have a start-up cost that permits performance improvement larger than the relevant thresholds. For this operator the *checkMoreParallelism* function is repeatedly called until the query plan cannot be modified any more. Each call can increase the partitioned parallelism by one degree at most, as only one machine can be added to an operator at a time.

checkMoreParallelism() is the basic function that checks whether the addition of one machine for a specific operator in the query plan is profitable. The list of choice criteria for machines, which is one of the function's parameters, defines which machine is checked first. L symbolises the list of choice criteria for machines. The criteria can either be in the form of the standard machine properties (e.g., available CPU) or combinations of them. For example, if $L = [mem, CPU \times ConSpeed]$, this corresponds to two criteria. The first is the available memory, and the second is the product of the available CPU speed with the available connection speed. The machines with high disk I/O rate are preferred for retrieving data, the machines with high connection speeds are preferred when the query cost is network-bound, the machines with large available memory are chosen for non-CPU intensive tasks, like *unnests*, and the machines with high CPU capacity are selected for the rest, CPU-intensive operations. *checkMoreParallelism()* also evaluates the achieved improvement ratios with the help of the cost model. If the improvement ratios are above the threshold, then the query plan is modified accordingly. Otherwise, the function iterates after removing the first element of the list of choice criteria for machines.

The algorithm comprises two loops. The outer loop can be repeated up to n times, where n is the number of physical operators in the query plan. The inner loop can be repeated up to m times, where m is the number of available machines. So, the worst-case complexity of the algorithm is of $O(n \times m)$, which makes it suitable for complex queries and when the set of available machines is large.

3. Evaluation

In this section we evaluate the scheduler proposed against existing and other common-sense techniques from distributed databases that do not employ, or employ only limited, partitioned parallelism, and against techniques from parallel databases that use all the available nodes. We want to compare the efficiency of our proposal for resource selection and allocation to subplans. The results enable us to claim that the scheduling proposal can significantly improve the performance of distributed queries evaluated in heterogeneous environments.

For the evaluation of the proposed scheduler we use simulation; we built the simulator by extending the Grid-enabled query compiler in [1]. Queries are executed according to the *iterator* model [5], which minimises the amount of intermediate data that needs to be stored and enables the operators comprising the query plan to run concurrently through pipelined parallelism. The parallel execution of operators is always load balanced. The cost model in [10], which is a detailed and validated one developed for parallel object database systems, has been adapted to operate in a distributed and autonomous environment and has been incorporated in the query engine. This model estimates the query completion time, by estimating the cost of each operator separately in time units.

Two intensive queries with one and five joins, respectively, are used for the evaluation. These queries retrieve data from two and six remote tables, respectively. Each table contains 100,000 tuples. We use two datasets. In the first one, *setA*, the average size of a tuple is 100 bytes, whereas in the second, *setB*, it is 1Kbyte. All the joins have a low selectivity of 10^{-5}. The joins are implemented by single-pass *hash joins*, which is the most efficient join algorithm for this case. The initial machine characteristics are set as follows: those machines that hold data (2 in the first query and 6 in the second) are able to retrieve data from their store at a 1MB/sec rate. The average time, over all machines participating in the experiment, to join two tuples depends on the CPU power of the machines and it is 30 microseconds. On average, data is sent at a connection speed of 600KB/sec.

For such configurations and datasets, the two queries are computationally intensive. Also, we assume that there are no replicas, so the *scans* cannot be parallelised. These queries are essentially CPU-bound. Similar experiments for network and disk I/O-bound queries in presence of replicas, and different query operators and data sizes are not presented due to space limitations and because their results neither contradict nor contribute significantly more than these results.

3.1. Performance evaluation

In this experiment, we evaluate the two example queries when the number of extra nodes (i.e., the machines that do not store any base data) varies between 0 and 20. We expect results not to vary significantly if this number is much higher; however, typical database queries on the Grid do not require to contact that many remote machines, especially if some of them have multiple processors. From the extra machines, 25% have double the CPU power and connection speed of the average (i.e., they evaluate a join between two tuples in 15 microseconds and transmit data at 1.2MB/sec), 25% have double CPU power and half connection speed (i.e., they evaluate a join between two tuples in 15 microseconds and transmit data at 300KB/sec), 25% have half CPU power and double connection speed (i.e., they evaluate a join between two tuples in 60 microseconds and transmit data at 1.2MB/sec), and 25% have half CPU power and connection speed (i.e., they evaluate a join between two tuples in 60 microseconds and transmit data at 300KB/sec). We compare two configurations of our proposal, one with lower improvement ratio threshold and one with higher, against six other simpler approaches: (i) using all the available nodes as we tend to do in parallel systems; (ii) not employing partitioned parallelism and placing the *hash joins* on the site where the larger input resides in order to save communication cost; (iii) employing limited partitioned parallelism and placing the *hash joins* on the nodes that already hold data participating in the join, i.e., not employing nodes that have not been used for scanning data from store; (iv) using only the two most powerful from the extra machines to parallelise all the query operators; (v) using all machines evenly, which is the only case where the number of tuples assigned to each machine is equal (i.e., the workload is not inversely proportional to its CPU power and *hash join* evaluation speed); and (vi) fully parallelising the most costly operator (this applies only to the multi-join query). Figures 3a-d show the results when the two queries are applied to the two datasets. Note that the thresholds are different in the two queries. The dashed lines in the charts depict the non parallelisable cost of the queries, i.e., the cost to retrieve data from the non-replicated stores. The numbers above the bars representing the performance of our algorithm, show how many machines are chosen. In all the other cases, the number of machines used is a result of the approach applied.

Our scheduling approach is tunable, and we can always achieve better execution times by using a smaller threshold. However, this benefit comes at the expense of employing more machines. From the figures we can see that (i) the proposed scheduler manages to reduce the parallelisable cost; (ii) techniques with no, or limited, partitioned parallelism (e.g., employing only the machines that store the databases,

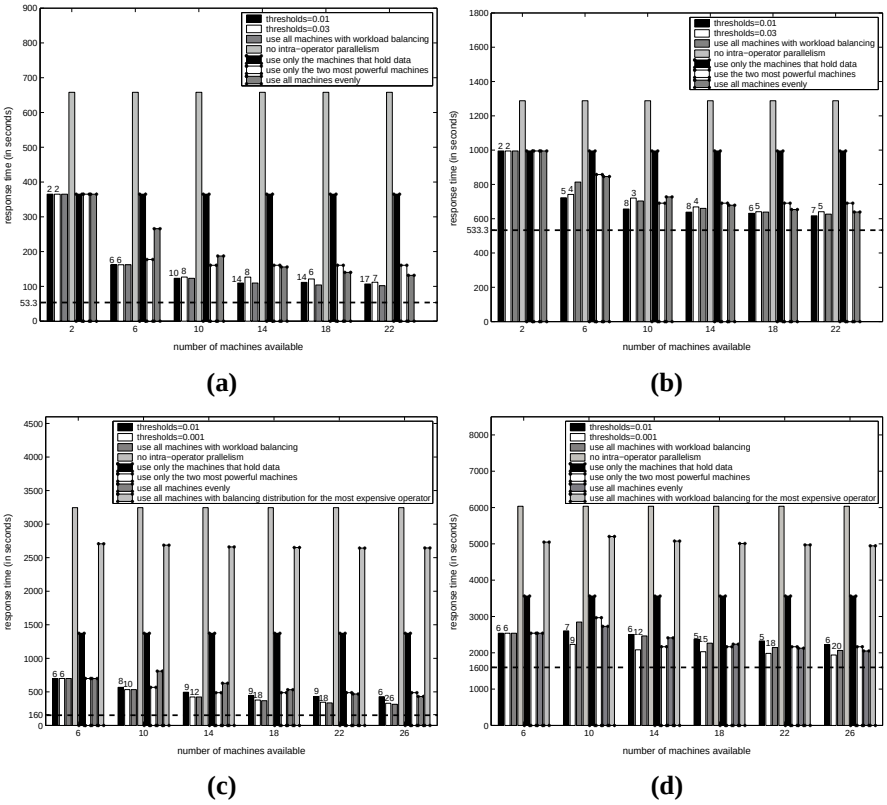
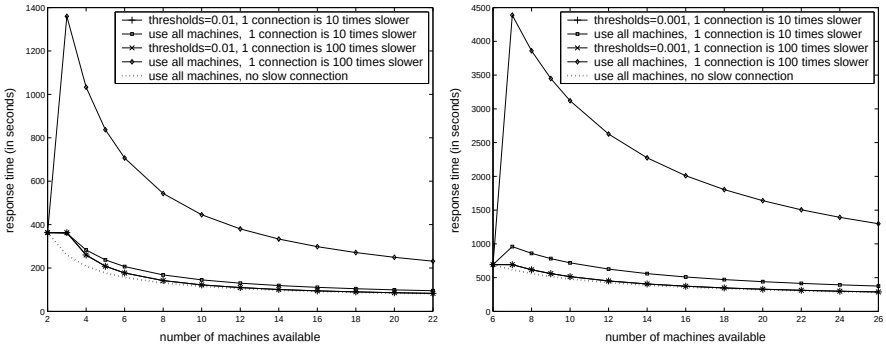


Figure 3. Comparison of different schedulings for (a) the 1-join query for setA, (b) the 1-join query for setB, (c) the 5-join query for setA, (d) the 5-join query for setB.

parallelising only the most costly operator) yield significantly worse performance than our proposal; (iii) policies that use only a small set of powerful nodes or do not try to perform workload balancing (i.e., the workload distribution is not according to the machine capabilities) are also clearly outperformed by the scheduler proposed, provided that the threshold is not relatively high for complex queries; (iv) relatively high thresholds can operate well when the cost of the query is concentrated on a single point in the query plan (Fig. 3a,b); (v) as expected, using all nodes and applying workload balancing can operate very well for specific machine configurations, provided that it is easy to calculate the appropriate workload distribution as in the queries over *setA*. In such cases, employing all nodes gives a very good indication of the lower bound on the performance. However, such a policy has severe drawbacks which are presented later.

3.2. Presence of slow connections

In Figures 4a,b we compare our approach with the approach of employing all the nodes when the two queries, which have one and five joins, respectively, run over *setA*, and there is just one machine with a slow connection. Two cases are considered: in the first, the slow connection is ten times slower than the average (i.e., 60 KB/sec); and in the second, it is 100 times slower. All the other resources are homogeneous, i.e., the *hash join* evaluation speed is 30 microseconds for each machine, and the connection speed is 600 KB/sec for each machine apart from the one with the slow connection. In an homogeneous setting, using all the machines and applying workload balancing yields the optimal performance, provided that the workload granularity is large enough so that start-up costs are outweighed. Our approach has high dependability and is not affected by the presence of a slow connection. From the figure, we can see that our algorithm behaves exactly the same in both cases and does not employ the machine with the slow connection. Moreover, it is very close to the optimal behaviour (i.e., the behaviour if all machines are used and there is no machine with slow connection - see dotted line in the figures). To the contrary, the performance degrades significantly when all nodes are used. These results show the merit of approaches that avoid utilisation of all the available nodes as they allow the performance to degrade gracefully when the available machines are slow or they have slow connections.



(a) **(b)**

Figure 4. Comparison of different scheduling policies in the presence of a slow connection for (a) the single-join query, and (b) the 5-join one.

4. Conclusions

Current distributed database applications operating in heterogeneous settings, like computational Grids, tend to run queries with a minimal degree of partitioned parallelism, with negative consequences for performance when the queries are computation and data intensive. Also, naive adaptations of existing techniques in the parallel systems literature may not be suitable for heterogeneous environments for the same reasons. The main contribution of this work is the proposal of a low complexity resource scheduler that allows for partitioned parallelism to be combined with the other well-established forms of parallelism (i.e., pipelined and independent) for use in a distributed query processor over the Grid. To the best of our knowledge, this is the first such proposal. The evaluation showed that the approach yields performance improvements when no, or limited, partitioned parallelism is employed, and can outperform extensions from parallel databases that use all the resources available. It can also mitigate the effects of slow machines and connections. This paper has contributed: (i) an analysis of the limitations of existing parallel database techniques to solve the resource scheduling problem in Grid settings; (ii) an algorithm that aims to address the limitations characterised in (i); and (iii) empirical evidence that the algorithm in (ii) meets the requirements that led to its conception in an appropriate manner and is thus of practical interest.

References

[1] N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the grid. In *Proc. of ICSOC*, pages 467–482, 2003.

[2] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *10th IEEE Symp. On High Performance Distributed Computing*, 2001.

[3] H. Dail, O. Sievert, F. Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the grid application development software project. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid resource management: state of the art and future trends*. Kluwer Academic Publishers Group, 2003.

[4] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *Proc. of the 1978 ACM SIGMOD Conf.*, pages 169–180, 1978.

[5] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[6] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. of the 12th VLDB Conf.*, pages 149–159, 1986.

[7] A. Radulescu and A. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *Proc. of 2001 International Conference on Parallel Processing (30th ICPP'01)*, Valencia, Spain, 2001.

[8] E. Rahm and R. Marek. Dynamic multi-resource load balancing in parallel database systems. In *21th VLDB Conf.*, pages 395–406, 1995.

[9] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *13th HCW Workshop*. IEEE Computer Society, 2004.

[10] S. Sampaio, N. W. Paton, J. Smith, and P. Watson. Validated cost models for parallel OQL query processing. In *Proc. of OOIS*, pages 60–75, 2002.

[11] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, 2002.

[12] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.