Distrib Parallel Databases (2006) 19:87–106
DOI 10.1007/s10619-006-8490-2

A novel approach to resource scheduling for parallel query processing on computational grids

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

Published online: 25 May 2006

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. The evaluation results show that the scheduler proposed outperforms current techniques without sacrificing the efficiency of resource utilisation.

Keywords Resource scheduling · Resource allocation · Grid · Parallel queries · Heterogeneous databases

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

Recommended by: Ioannis Vlahavas

A. Gounaris (✉) · R. Sakellariou · N. W. Paton · A. A. A. Fernandes
Department of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, UK
e-mail: gounaris@cs.man.ac.uk

R. Sakellariou
e-mail: rizos@cs.man.ac.uk

N. W. Paton
e-mail: norm@cs.man.ac.uk

A. A. A. Fernandes
e-mail: alvaro@cs.man.ac.uk

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 [2], GridDB [18], GridDB-lite [21]). In particular, query processors for Grid-enabled databases, such as [2] and [30], 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 care for parallelism. This is due to the fact that many complicated tasks 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.

Resource heterogeneity can have many faces. Systems like OGSA-DQP are capable of performing query processing over heterogeneous local database management systems, like mySQL and Oracle. Such systems may employ a number of machines to run the query processing tasks. These machines can be heterogeneous as well, in terms of their computational capacity and characteristics. For instance, some may have a larger amount of memory available or may be less loaded than others. This form of heterogeneity has motivated our work. The setting we assume is a set of machines with different characteristics of memory, CPU speed, disk I/O throughput etc. All such machines are candidates for running parts of the query execution plan independently of whether they hold any data initially. Furthermore, we assume that the database owners are willing to send data remotely if this is estimated to lead to performance improvements. It is significant to mention that Grid technologies have made both the creation and provision of heterogeneous pools of autonomous resources, and the access to them feasible.

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 [14], 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., [17, 27, 33]), and their Grid variants (e.g., [34]) tend to allocate a graph vertex to a single machine, which leads to no partitioned parallelism. More comprehensive proposals

(e.g., GrADS [7]) 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., [24]) and parallel database scheduling (e.g., [9, 10]), are restricted to homogeneous settings. We defer a more detailed discussion of such proposals to Section 5.

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.[1] 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. In summary, the contribution of this work includes a system-independent algorithm that does not restrict the degree of intra-operator to any extent and does take into account the fact that the resources available are heterogeneous, along with empirical evidence that the algorithm leads to performance improvement and robustness. Also, a comprehensive analysis of the limitations of existing parallel database techniques to solve the resource scheduling problem in Grid settings is presented.

The remainder of the paper is structured as follows: Firstly, we present the problem in Section 2, followed by our proposed solution in Section 3. This solution is then evaluated in Section 4. We compare our work with others in Section 5, and conclude in Section 6.

2. 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 [36]. 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.

The three dimensions (i.e., how many, which and for which part of the query) cannot be separated from each other to simplify the algorithm in a divide-and-conquer fashion. E.g., it is meaningless to determine the number of selected nodes from a heterogeneous pool without specifying these machines; this is in contrast to what can be done in homogeneous systems since in a homogeneous setting each machine may have different capabilities. The problems of resource selection and matching of plan fragments and resources have to be solved within a single algorithm. 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

[1] A short, less detailed description of the proposal has appeared in [13].

and efficient resource utilisation. As the problem is theoretically intractable [17], effective and efficient heuristics need to be employed.

3. A practical query scheduler for the grid

3.1. 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 receives a query plan which can be partitioned into subplans that can be evaluated on different machines. For the shape of the query plan we assume the existence of a query optimiser which first constructs a single-node plan, and then transforms the single-node plan into a multi-node one, in order to reduce the search space. This method has been widely adopted in parallel and distributed query processors [16], as it performs well in a wide range of cases [8]. The scheduler proposed in this paper deals with the second stage, where the single node plan is parallelised. As such it extends and does not replace optimisers that produce non-parallelised query plans. An example of a query plan is shown in Fig. 1. *Exchange* operators encapsulate the parallelism and involve communication [14], and they naturally define the boundaries of different subplans. *Operation-calls* are used to encapsulate user defined functions [30]. *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.

Initially, each of the operators of the query plan is scheduled on one machine. I.e., the algorithm proposed here starts from a query plan with minimum partitioned parallelism; such a query plan is unlikely to perform well for intensive computations. After this initial resource allocation, which is at the lowest possible degree of partitioned parallelism, it enters a loop (see Algorithm 1). 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 defines which criteria should be used for selecting machines for this operator. Following this step, the algorithm increases the operator's degree of parallelism by one degree if that increase improves the performance of the query plan above a certain

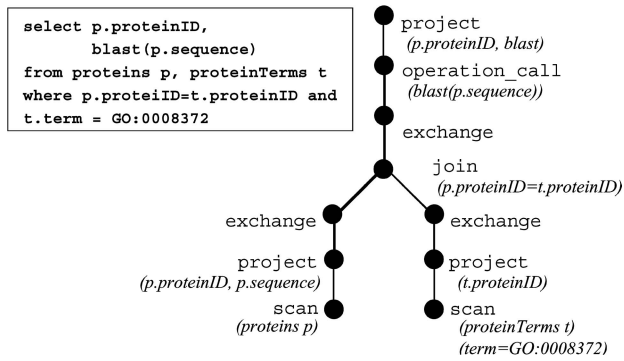

Fig. 1 An example query plan

Algorithm 1 High level description of the scheduling algorithm. Algorithms 2 and 3 present a more detailed view.

```
1: repeat
2:    get costliest parallelisable operator
3:    define the criteria for machine selection
4:    repeat
5:       get the set of available machines
6:       check if more parallelism is beneficial
7:    until no changes in the degree of parallelism of the costliest
      operator
8: until no changes in which operator is the costliest
```

threshold. Such an approach is in line with works like [23, 24]. 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. From a higher level point of view the algorithm transforms an existing plan to a more efficient one at each step, by modifying the set of resources allocated to a part of the query plan. Transformational approaches to query optimisation have already been employed for constructing query plans (e.g., simulated annealing and iterative improvement techniques [15]) but not for scheduling resources.

To estimate the cost of the query plan and individual operators, the algorithm 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 Dail et al. [6, 7]. In other words, although the algorithm depends on the existence of a cost model, it is cost model-generic. 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. The importance of cost models is significant, given that heuristic-based optimisations can err substantially in distributed heterogeneous environments [26].

3.2. The algorithm

3.2.1. The input of the algorithm

The inputs to the algorithm are:

- 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. Such a query plan can be produced by traditional query optimisers that are present in any modern query engine.
- 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 [5] or NWS [37], 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.

Defining an optimal threshold, or developing a technique for its estimate, is both an open research problem and out of the scope of this work. However, there is a clear trade-off between fast execution time and good convergence to optimal scheduling decisions. The smaller the threshold, the closer the final point to the optimal point will be. Nevertheless, 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. Note that the convergence to the optimal point depends additionally (i) on the accuracy of cost estimates, and (ii) on the absence of local minima in the performance for different degrees of parallelism.

3.2.2. Notation

A query plan *QueryPlan* is represented as a directed acyclic graph $G = (V, E)$, where V is a set of n operators, and E is the set of edges. M is the set of the m available machines. Each machine M_{i} is described by the vector:

$$\{CPU_i, Mem_i, I/O_i, ConSpeed_i, tables_i, programs_i\},$$

where CPU_i, Mem_i, I/O_i and $ConSpeed_i$ are the available CPU power, available memory, disk bandwidth and average connection speed of the ith machine respectively (to keep the presentation simple, it is assumed here that all connections from the ith machine are of the same speed). $tables_i$ and $programs_i$ are the lists of database tables participating in the query and programs called externally by the query that reside on that machine, respectively. M_{table} is the set of machines that can evaluate a *scan* (*table*,. . .) operator, whereas $M_{program}$ is the set of machines that can evaluate an (*operation_call*(. . .,*program*,. . .).

Without loss of generality, an assumption is made that cost is measured in time units. *TimeCost*(G) (or *TimeCost*(*QueryPlan*)) is the time cost of the plan, while *TimeCost*(V_i) is the time cost of operator V_i. Note that typically, cost models estimate the cost of a query plan by estimating the cost model of individual operators and taking into account any parallelism if appropriate (e.g., [28]). Thus, the same cost model is responsible for the calculation of *TimeCost*(G) and *TimeCost*(V_i).

3.2.3. Detailed description of the algorithm's steps

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 [25]. As there already exists a significant number of proposals for resource scheduling without partitioned parallelism (e.g., [19]), this phase is not covered in detail.

In the second phase, which is the main contribution of this work in its own right, there are two basic steps (Algorithms 2 and 3), and a third one for exiting (Algorithm 5). In the

Algorithm 2 The first step of the scheduling algorithm after the initial resource allocation in detail.

```
1: repeat
2:    /* STEP 1: Estimate the cost of query and operators */
3:    n = NumberOfOperators
4:    m = NumberOfMachines
5:    a = threshold
6:    for i = 1 to n do
7:       estimate TimeCost(V_i)
8:    estimate TimeCost(QueryPlan)
9:    /*list the operators according to their cost in descending order*/
10:   list V_sorted = SORT(V) on TimeCost(V_i)
11:   /*if true, the costliest operator has been allocated more machines*/
12:   bool changes_made = false
13:   RUN STEP 2: Examine the costliest operator - see Algorithm 3
14: until !changes_made
```

first step (Algorithm 2), the cost model evaluates the cost of the query plan and its individual operators. The operators are sorted by their cost. The second step (Algorithm 3) is the step where the partitioned parallelism can be increased. 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, *SUC*, that permits performance improvement larger than the relevant thresholds (see the first `repeat` loop, lines 3–18 in Algorithm 3). For this operator, the *check_more_parallelism* function is repeatedly called (line 29) 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.

check_more_parallelism (Algorithm 4) is a basic function. It checks whether the addition of one machine for a specific operator in the query plan is profitable. Such addition affects all the operators between the closest *exchange* higher and the closest *exchange* lower in the plan. The list of choice criteria for machines, which is one of the function's parameters, defines which machine is checked first, with the conflict resolution being done in the *choose_according_to* function (line 9 in Algorithm 4). 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 criteria depend on the nature of each operator and are defined in the second step of the algorithm (lines 20—27 in Algorithm 3).

The next line (line 10 in Algorithm 4). involves the evaluation of the achieved improvement ratio with the help of the cost model. A prerequisite for the evaluation of the improvement ratios is to define a data distribution policy. Finding an optimal data distribution policy is NP-hard when the resources have different characteristics, and the cost to transfer data over the network is considered; however, a simple and efficient approach for heterogeneous settings is to allocate to each machine an amount of tuples that is proportional to its available CPU power, if the CPU cost is the cost that dominates, and proportional to its connection speed, if the communication cost is the higher cost. If the improvement ratios are above the

Algorithm 3 The detailed steps of the scheduling algorithm after the initial resource allocation.

/*STEP 2: Examine the costliest operator*/
$i = 0$
repeat
  $i++$
  if $V_{sorted}[i] = SCAN(database)$ **then**
    /*define the candidate nodes for each king of operator*/
    $M^{'} = M_{database}$
  else if $V_{sorted}[i] = OPERATION - CALL(program)$ **then**
    $M^{'} = M_{program}$
  else
    $M^{'} = M$
  /*remove already allocated nodes*/
  $M^{'} = M^{'} - (V_{sorted}[i] \rightarrow machines)$
  for $j = 1$ *to* $LengthOf(M^{'})$ **do**
    if $(SUC_j \geq (1-a) \cdot TimeCost(V_{sorted}[i]))$
    OR $(SUC_j \geq (1-a) \cdot TimeCost(QueryPlan))$ **then**
      /*do not consider machines with relatively high startup cost*/
      $M^{'} = M^{'} - M^{'}_j$
until $(M^{'}$ *not empty*$)$ OR $(i = n)$
/*Parallelise the most expensive parallelisable operator, which is $V_{sorted}[i]$*/
if $V_{sorted}[i] = Communication - Bounded$ **then**
  $L[1] = ConSpeed,\ L[2] = ConSpeed \times CPU,\ L[3] = CPU$
else if $V_{sorted}[i] = I/O - Mounded$ **then**
  $L[1] = I/O \times ConSpeed,\ L[2] = CPU,\ L[3] = Mem$
else if $V_{sorted}[i] = Memory - Bounded$ **then**
  $L[1] = Mem,\ L[2] = CPU$
else if $V_{sorted}[i] = CPU - Bounded$ **then**
  $L[1] = CPU,\ L[2] = ConSpeed \times CPU,\ L[3] = ConSpeed$
repeat
  $local\_changes\_made = check\_more\_parallelism(L, M^{'}, V_{sorted}[i])$
  $changes\_made = changes\_made\ OR\ local\_changes\_made$
  if $local\_changes\_made$ **then**
    /*update the set of candidate machines*/
    $M^{'} = M^{'} - (V_{sorted}[i] \rightarrow machines)$
until $!local\_changes\_made$

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, as presented, is independent of the physical implementation of the operators that can differ between database systems. As shown in Step 2, 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.

Algorithm 4 The basic auxiliary function, which is responsible for checking greater degrees of parallelism.

```
1: /*This function returns true if allocating a new machine
2: to operator v is beneficial.*/
3: bool check_more_parallelism(criteria_list L,
4:                             machine_set M', operator v){
5: bool result = false
6: /*the loop runs until all the criteria are checked
7: or an addition of a machine improves the query plan adequately*/
8: while L do
9:   machine i = choose_according_to(L, M')
10:  if (ImprovementRatio(QueryPlan) ≥ a) then
11:    v → machines = v → machines → add(i)
12:    update scheduling of other operators affected
13:    return true
14:  else
15:    /*the first choice criterion is removed*/
16:    L = L − L[1]
17: return result
18: }
```

Algorithm 5 The exiting step of the scheduling algorithm.

```
1: /*STEP 3: Exit*/
2: for i = 1 to n do
3:   if V_i = EXCHANGE && LengthOf(V_i → machines) = 1
     then
4:     parent(child(V_i)) = parent(V_i), child(parent(V_i)) = child(V_i)
5: EXIT
```

The final step evaluates the exit function. If Step 2 (Algorithm 3) has not resulted in any changes in the query plan, then the algorithm checks the scheduling of *exchanges*, deletes them if necessary, and exits (Algorithm 5). Otherwise, the algorithm goes back to Step 1 (Algorithm 2), to evaluate again the cost of the plan and their operators, and subsequently to try to parallelise the operator that has become the most costly after the changes. The scheduling of *exchanges* is checked in order to delete the ones that receive data from and send data to the same single node (lines 2–4 in Algorithm 5).

3.2.4. Complexity

The algorithm comprises two loops. Steps 1 and 2 are repeated until the exit condition is met. Also, Step 2 runs a second loop until a local exit condition is satisfied. 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.

4. 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. We are interested in both the performance in time units and in the parallelisation efficiency. 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 [2, 30]. Queries are executed according to the *iterator* model [14], 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 [28], 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 instance separately in time units. This cost is further decomposed in (i) computation cost, (ii) disk I/O cost, and (iii) communication cost.

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 1 Kbyte. 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 1 MB/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 600 KB/sec. We assume that the available memory on each machine is enough to hold the hash table in case this machine is assigned a *hash join*, and that the connection speed is a property of the data sender only and not of both the data sender and receiver. The machine start-up cost, which includes the initialisation of a pre-existing query evaluation engine and the submission of the query subplan, is 1 second. The parameters above form the input to the cost model adopted by the system, and are realistic, according to our experience with the OGSA-DQP Grid-enabled query processor [2].

For such configurations and datasets, the two queries are computationally intensive. When they are applied to the first dataset, and due to the size of the joins, the computation cost is greater than the communication cost and the disk I/O cost by an order of magnitude without partitioned parallelism. If the second dataset is used, the I/O cost is still smaller, but the communication cost, although smaller initially, contributes significantly to the total cost. In both queries, we assume that each machine chosen to evaluate a portion of a *hash join* is allocated a number of tuples that is inversely proportional to its *hash join* evaluation speed (in a few cases in the following where a different workload distribution is assumed, this is explicitly stated). As the computation cost is the dominant cost, this workload distribution policy ensures high performance, without adding much complexity. 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 because their results neither contradict nor contribute significantly more than these results.

4.1. 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. 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 1.2 MB/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 300 KB/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 1.2 MB/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 300 KB/sec). We compare two configurations of our proposal, one with lower improvement ratio threshold and one with higher, against six other simpler approaches:

1. ***Use all machines with workload balancing:*** it uses all the available nodes in the way that parallel systems tend to do, taking into consideration their heterogeneous capabilities for workload balancing though (in the way described previously), and not considering them as being the same (i.e., using all the available nodes without applying workload balancing);
2. ***No intra-operator parallelism:*** it does not employ partitioned parallelism and places the *hash joins* on the site where the larger input resides in order to save communication cost;
3. ***Use only the machines that hold data:*** it employs limited partitioned parallelism and places the *hash joins* on the nodes that already hold data participating in the join, i.e., it does not employ nodes that have not been used for scanning data from store;
4. ***Use the two most powerful machines:*** it uses only the two most powerful from the extra machines to parallelise all the query operators;
5. ***Use all machines evenly:*** it uses 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 the *hash join* evaluation speed as in the first approach); and
6. ***Use all machines with workload balancing for the most expensive operator:*** it applies the first approach only to the most expensive operator and not to the complete query plan (this applies only to the multi-join query).

Figures 2–5 show the results when the two queries are applied to the two datasets (four combinations in total). 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:

– the proposed scheduler manages to reduce the parallelisable cost; compare, for example, the difference of the algorithm proposed (first two bars in each bar set) from the dashed

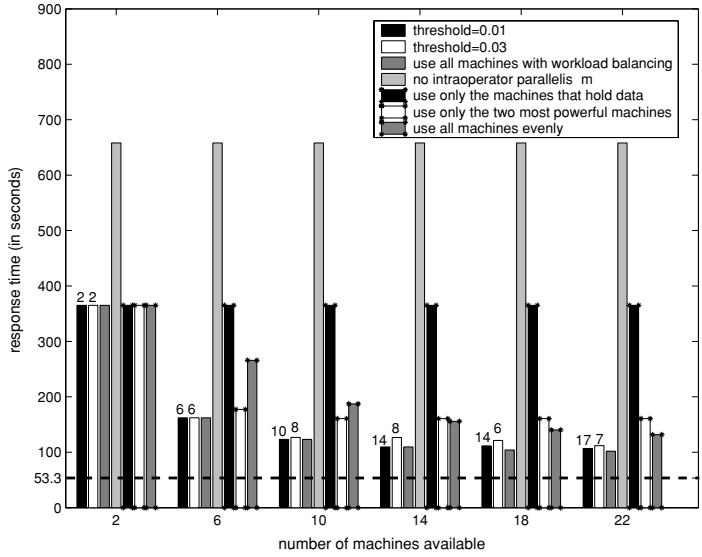

Fig. 2 Comparison of different schedulings for the 1-join query for setA

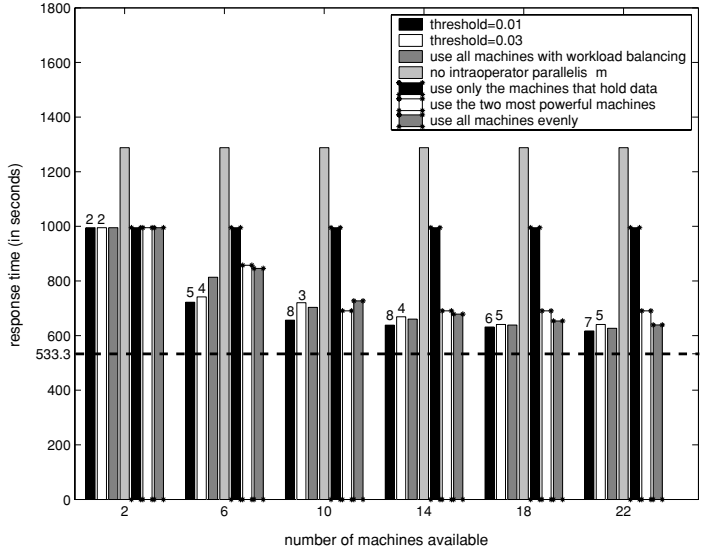

Fig. 3 Comparison of different schedulings for the 1-join query for setB

line, with the difference of the performance of the system with no intra-operator parallelism (2nd test approach, which corresponds to the 4th bar in each bar set) from this line;

– techniques with no, or limited, partitioned parallelism (2nd, 3rd and 6th test approaches, which correspond to the 4th, 5th and 8th, if exists, bar in each bar set, respectively) yield significantly worse performance than our proposal;

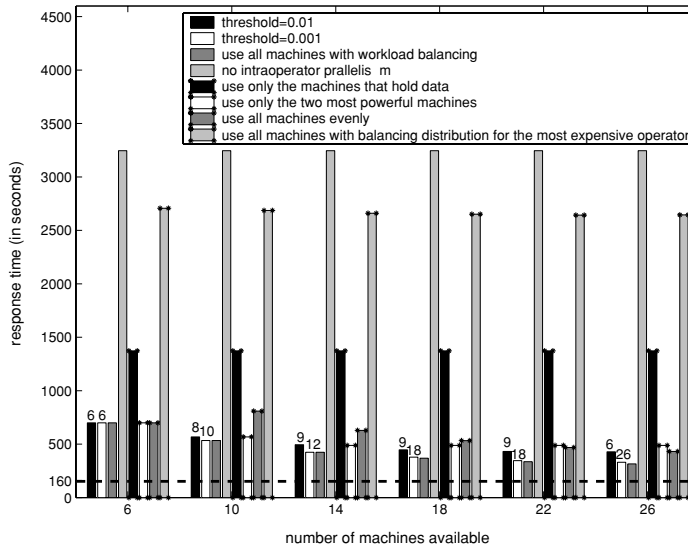

Fig. 4 Comparison of different schedulings for the 5-join query for setA

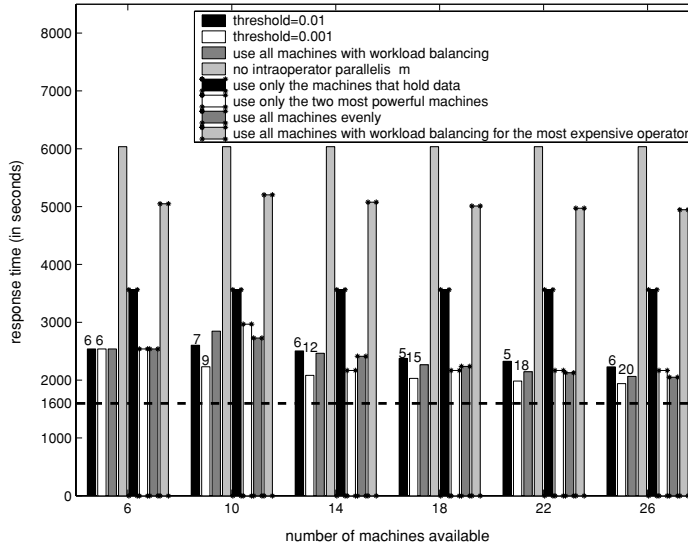

Fig. 5 Comparison of different schedulings for the 5-join query for setB

– policies that use only a small set of powerful nodes or do not try to perform workload balancing (4th and 5th test approaches, which correspond to the 6th and 7th bar in each bar set, respectively) are also outperformed by the scheduler proposed, provided that the threshold is not relatively high for complex queries and the number of machines available is relatively high;
– relatively high thresholds can operate well when the cost of the query is concentrated on a single point in the query plan (Figs. 2 and 3);

Table 1 The time cost of the scheduling algorithm for 20 extra nodes

Number of joins	Threshold	Scheduler cost
1	0.01	0.0106 Secs.
1	0.03	0.0055 Secs.
5	0.001	0.3138 Secs.
5	0.01	0.2083 Secs.

– as expected, using all nodes and applying workload balancing (1st test approach, which corresponds to the 3rd bar in each bar set) 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 in more detail in the following experiments.

Table 1 shows that the cost to execute the proposed scheduling algorithm for each of the two queries and threshold is reasonable and negligible compared to the query completion time cost.

4.2. Presence of slow connections

In Figs. 6 and 7 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. Even if the number of slow connections is large compared to the total number, our algorithm may show no performance degradation at all; conversely a single slow connection is enough to slow down the whole query if all machines are used. The scenario of this experiment is one of the many where naive extensions from parallel systems are inappropriate for Grid settings. Another argument that cannot be shown from this experiment is that the schedulers that use all the available nodes depend strongly on good workload distribution decisions. Such decisions usually lead to NP-hard problems. The approach we presented, mitigates this dependency.

4.3. Parallelisation efficiency

Using machines efficiently is an important issue, especially in a Grid setting where the resources usually belong to different organisations and there may be a charge for using them.

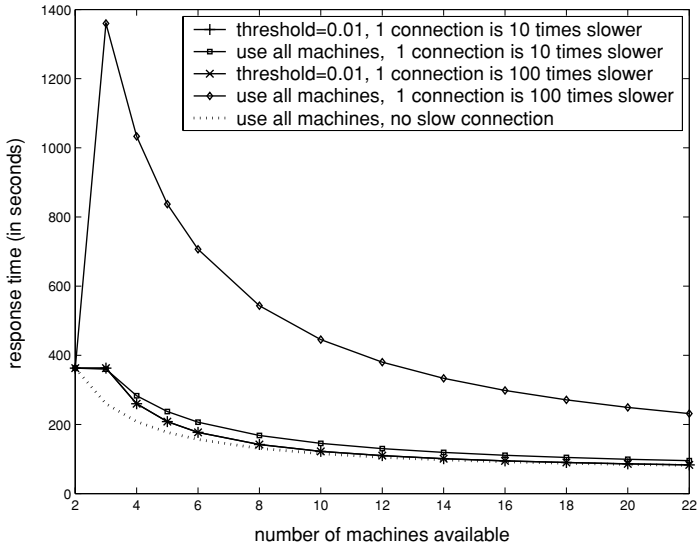

Fig. 6 Comparison of different scheduling policies in the presence of a slow connection for the single-join query

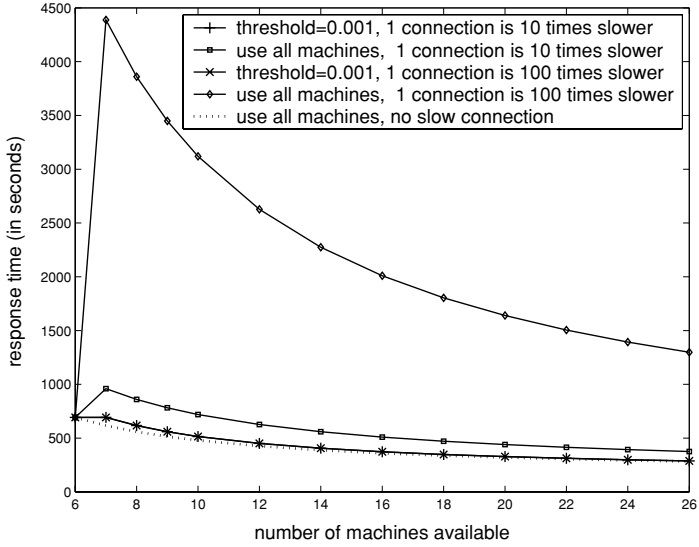

Fig. 7 Comparison of different scheduling policies in the presence of a slow connection for the 5-join query

We define efficiency as the inverse of the product of the response time and the number of machines used. According to this definition, sensible comparisons of the efficiency of different schedulers can be made only for the same query. Higher values of the efficiency indicate, in practice, better utilisation of the resources. Consider, for example, three cases: (i) a query executes in 2 time units and 2 machines are used; (ii) a query executes in 1 time unit and 4 machines are used; and (iii) a query executes in 0.9 time units and 10 machines

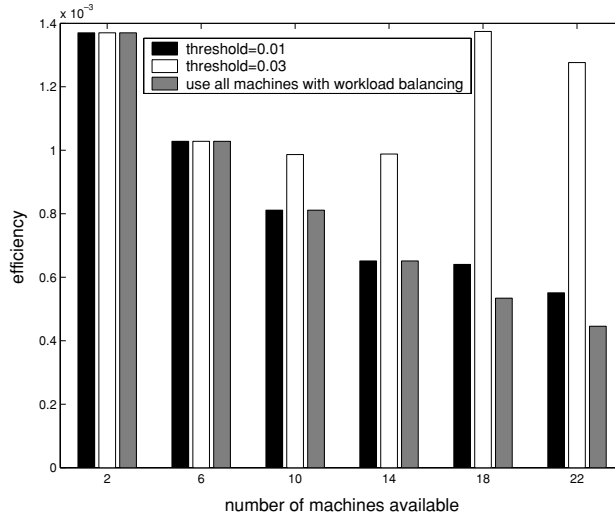

Fig. 8 Comparison of the efficiency of different scheduling policies for the single-join query

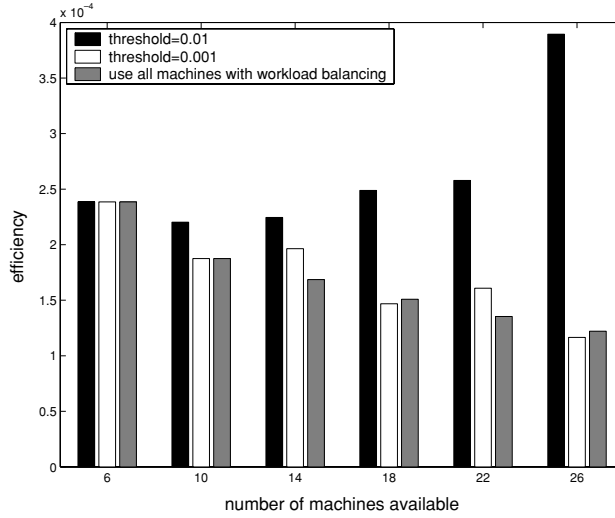

Fig. 9 Comparison of the efficiency of different scheduling policies for the 5-join query

are used. In the first two cases, the parallelisation efficiency is the same, and the machines are exploited to the same extent. However, in the third case, the efficiency is significantly lower, although the response time is decreased. This indicates that usage of more machines has not been traded for performance improvements as efficiently as in the first two cases.

Figures 8 and 9 show the efficiency of the scheduler proposed as compared against the policy of employing all the machines available, for the two queries applied to the first dataset, respectively (the performance of these queries is shown in Figs. 2 and 4). For lower degrees of parallelism, the efficiency can much higher without the performance being considerably lower. For instance, compare the difference between the 2nd and the 3rd bars in Figs. 2 and 8,

when there are 22 machines available. By tuning the threshold we can trade the efficiency for better response times. This makes the new proposal easily adaptable to cases where the nodes are not provided for free and the economic cost may be considered as significant as the time cost.

5. Related work

Although there is no previous work that deals with the scheduling of resources in heterogeneous environments to support arbitrary degrees of partitioned parallelism, there exist proposals that relate to ours in certain ways.

In the database field, distributed query processing has mostly been influenced by some pioneering systems. The most influential, R* [19], simplified the problem of resource scheduling by neglecting the benefits of partitioned parallelism. The data are retrieved from a single site only, and are joined on a single site, which is either the site of one of the inputs or the site that asked for the data. SDD-1 [3] focused on semijoins and also did not employ partitioned parallelism. Distributed Ingres [10] took a step forward, and provided for partitioned parallelism, but only for machines that store data. Their scheduling algorithm assumed that all the participating machines have the same computational capabilities (a property that, in the general case, cannot be expected to hold on the Grid). It also forces a choice either to use all nodes available or only one. Rahm and Marek [25] discusses an approach for load balancing employing partitioned parallelism, and although it refers to completely homogeneous environments, it does not force the system to employ all the available nodes when these are not needed, extending the work of Wilschut et al. [36]. Other existing techniques for parallel and distributed databases do not consider partitioned parallelism or completely skip the resource selection phase by assuming a fixed set of resources and then trying to schedule tasks over these resources (e.g., [11, 20]). Moreover, they usually assume homogeneous and stable environments (e.g., [36]). E.g., Gamma [9] assumes that all the available machines are similar in terms of capabilities, connection speed and ownership.

In early commercial distributed database systems, the reason for not developing schedulers supporting partitioned parallelism was that the communication cost was the predominant cost. The biggest effort was put into minimising this cost [35], and each operator was executed at a single site. In modern applications, and due to advances in network technologies, the computation cost is often the dominating cost (e.g., [31]). However, advanced distributed query processing proposals still employ only pipelined and independent (e.g.,[26, 32]). A question may arise as to whether existing techniques that do not employ partitioned parallelism (like, for example, the query/data/hybrid shipping in client-server architectures [16]) are efficient in most cases. For a large range of queries, the answer is negative. The parallelisation saturation point is lower when the start-up cost increases. The start-up cost depends strongly on the machine and the evaluation method, but in most cases it occurs once for all the operators processed on a single machine. Therefore, the contribution of the startup cost becomes less significant for intensive queries. When the ratio between workload and machines used increases, the saturation point increases as well. These properties are quite appealing for tasks in computational grids insofar as they are envisaged to be of significant size and complexity. Consequently, especially for intensive queries, the optimal degree of parallelism is expected to be large, even in heterogeneous settings with high start-up and communication costs. In the recent years, several interesting proposals on Grid query processing have appeared (e.g., [12]), but these do not deal with the problem of resource allocation in parallel execution.

In the area of DAG scheduling to support mixed parallelism, there have been many interesting proposals. Our algorithm may be regarded as an adaptation of the task allocation in [23, 24] for heterogeneous environments, tailored to the needs of query processing. With respect to heterogeneity, Boudet et al. [4] takes one step further, by considering heterogeneous clusters of homogeneous machines, but has significantly higher complexity and is not compatible with the iterator model of query execution [14]. Ordinary DAG schedulers for heterogeneous and Grid environments, such as [27, 29, 34], do not consider partitioned parallelism. Not considering partitioned parallelism is the usual case even for homogeneous systems [17].

Our proposal is, to a certain extent, compatible with the GrADS project [7], as it can play the role of the GrADS mapper for database query applications. The cost model used by our scheduler has the same functionality as the GrADS performance model, and is similarly decoupled [6]. The main differences lie in the fact that, in query processing, the initial pruning of the resource sets cannot be done in a completely application-independent way, as in GrADS, because database locality is very important for efficient query plan construction. As a consequence, our proposal runs the main algorithm once, and not many times for the several candidate machine groups, resulting in lower algorithm execution times. In the context of Dail et al. [7], Yarkhan and Dongarra [38] has presented a heuristic algorithm that allocates nodes in an incremental way similarly to our proposal. One difference between the two techniques, apart from the higher complexity of query plans, is that in [38] the machines are first chosen and then it is decided how to use them, whereas in our approach, a bottleneck is first identified and then an attempt is made to increase the parallelism. Other schedulers developed in GrADS (e.g., [22]) do not provide for different degrees of parallelism in different parts of the query plan, and may require the user to prune first, explicitly, an extended set of resources. In our proposal, this is a responsibility of the scheduler.

6. 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. By being able to choose only the nodes that contribute significantly to the performance, it uses the machines more efficiently, and thus can be easily adapted to cases where the resources are not provided for free.

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.

Initial experiments with the OGSA-DQP system show that, in real scenarios, parallelising expensive query operators in a Grid setting is both beneficial and feasible [1]. The intention of the authors is to use this scheduling algorithm in such a system, provided that the metadata that is required as its input becomes available in a practical way.

References

1. N. Alpdemir, A. Gounaris, A. Mukherjee, D. Fitzgerald, N.W. Paton, P. Watson, R. Sakellariou, A.A. Fernandes, and J. Smith, "Experience on performance evaluation with OGSA-DQP," in Proc. of Fourth UK e-Science All Hands Meeting, 2005.
2. 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, 2003, pp. 467–482.
3. P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie Jr., "Query processing in a system for distributed databases (SDD-1)," ACM TODS, vol. 6, no. 4, pp. 602–625, 1981.
4. V. Boudet, F. Desprez, and F. Suter, "One-step algorithm for mixed data and task parallel scheduling without data replication," in 17th International Parallel and Distributed Processing Symposium (IPDPS-2003). Los Alamitos, CA, IEEE Computer Society, 2003.
5. 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.
6. H. Dail, F. Berman, and H. Casanova, "A decoupled scheduling approach for Grid application development environments," Journal of Parallel and Distributed Computing, vol. 63, no. 5, pp. 505–524, 2003.
7. 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 (eds.), Grid Resource Management: State of the Art and Future Trends. Kluwer Academic Publishers Group, 2003.
8. A. Deshpande and J.M. Hellerstein, "Decoupled query optimization for federated database systems," in Proc. of ICDE 2002, pp. 716–732.
9. D.J. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "GAMMA—A high performance dataflow database machine," in Proc. of the 12th VLDB Conf., 1986, pp. 228–237.
10. R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in a relational data base system," in Proc. of the 1978 ACM SIGMOD Conf., 1978, pp. 169–180.
11. M. Garofalakis and Y. Ioannidis, "Parallel query scheduling and optimization with time- and space-shared resources," in Proc. of VLDB, 1997, pp. 296–305.
12. S. Goel, H. Sharda, and D. Taniar, "Atomic commitment and resilience in grid database systems," International Journal of Grid and Utility Computing, vol. 1, no. 1, pp. 46–60, 2005.
13. A. Gounaris, R. Sakellariou, N.W. Paton, and A.A.A. Fernandes, "Resource scheduling for parallel query processing on computational grids," in Proc. of 5th IEEE/ACM International Workshop on Grid Computing GRID, 2004, pp. 396–401.
14. G. Graefe, "Query evaluation techniques for large databases," ACM Computing Surveys, vol. 25, no. 2, pp. 73–170, 1993.
15. Y. Ioannidis, "Query optimization," ACM Computing Surveys, vol. 28, no. 1, 1996.
16. D. Kossmann, "The State of the art in distributed query processing," ACM Computing Surveys, vol. 32, no. 4, pp. 422–469, 2000.
17. Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," ACM Comput. Surveys, vol. 31, no. 4, pp. 406–471, 1999.
18. D.T. Liu, M. Franklin, and D. Parekh, "GridDB: A relational interface for the grid," in ACM SIGMOD, ACM Press, 2003, pp. 660–660.
19. L.F. Mackert and G.M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in Proc. of the 12th VLDB Conf., 1986, pp. 149–159.
20. T. Mayr, P. Bonnet, J. Gehrke, and P. Seshadri, "Leveraging non-uniform resources for parallel query processing," in 3rd IEEE CCGrid, 2003.
21. S. Narayanan, U. Catalyurek, T. Kurc, X. Zhang, and J. Saltz, "Applying database support for large scale data driven science in distributed environemnts," in Proc. of GRID, 2003.
22. A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar, "Numerical libraries and the grid," International Journal of High Performance Computing Applications, vol. 15, no. 4, pp. 359–374, 2001.
23. A. Radulescu, C. Nicolescu, A. van Gemund, and P. Jonker, "CPR: Mixed task and data parallel scheduling for distributed systems," in Proc. of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), IEEE Computer Society, 2001.

24. 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.
25. E. Rahm and R. Marek, "Dynamic multi-resource load balancing in parallel database systems," in 21th VLDB, Conf., 1995, pp. 395–406.
26. M. Roth, F. Ozcan, and L. Haas, "Cost models DO matter: Providing cost information for diverse data sources in a federated system," in The VLDB Journal, 1999, pp. 599–610.
27. R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems," in Proc. of 13th HCW Workshop, IEEE Computer Society, 2004.
28. S. Sampaio, N.W. Paton, J. Smith, and P. Watson, "Validated cost models for parallel OQL query processing," in Proc. of OOIS, 2002, pp. 60–75.
29. S. Shivle, R. Castain, H.J. Siegel, A.A. Maciejewski, T. Banka, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, P. Sugavanam, and J. Velazco, "Static mapping of subtasks in a heterogeneous ad hoc grid environment," in Proc. of 13th HCW Workshop, IEEE Computer Society, 2004.
30. J. Smith, A. Gounaris, P. Watson, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou, "Distributed query processing on the grid," International Journal of High Performance Computing Applications, vol. 17, no. 4, pp. 353–367, 2003.
31. E. Stolte and G. Alonso, "Optimizing scientific databases for client side data processing," in Proc. of EDBT, 2002, pp. 390–408.
32. M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," VLDB Journal, vol. 5, no. 1, pp. 48–63, 1996.
33. T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor—A distributed job scheduler," in T. Sterling (ed.), Beowulf Cluster Computing with Linux, MIT Press, 2002.
34. D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in F. Berman, G. Fox, and T. Hey (eds.), Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons Inc., 2003.
35. G. Thomas, G. Thompson, C. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman, "Heterogeneous distributed database systems for production use," ACM Computing Surveys, vol. 22, no. 3, pp. 237–266, 1990.
36. A.N. Wilschut, J. Flokstra, and P. Apers, "Parallelism in a main-memory DBMS: The performance of PRISMA/DB," in Proceedings of the 18th VLDB Conf., 1992.
37. R. Wolski, N.T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," Future Generation Computer Systems, vol. 15, nos. 5–6, pp. 757–768, 1999.
38. A. YarKhan and J. Dongarra, "Experiments with scheduling using simulated annealing in a Grid environment," in Proc. of 3rd International Workshop on Grid Computing GRID, 2002, pp. 232–242.