Constraint Checking in a Parallel Object-Oriented Database System

N. Bassiliades and I. Vlahavas
Dept. of Informatics,
Aristotle University of Thessaloniki,
54006 Thessaloniki,
Greece.

Tel: +30-31-998145
Fax: +30-31-998419
E-mail: [bassiliades|vlahavas]@olymp.ccf.auth.gr

Abstract

This paper deals with parallel checking of passive constraints in object-oriented databases. It presents a parallel algorithm for constraint checking based on a master-slave technique and discusses its implementation on a parallel object-oriented database system. The system is named PRACTIC and is based on class concurrency. Passive constraints, unlike active database rules, are independent and can be executed using AND-parallelism. Simulation shows that the proposed algorithm offers considerable speed-up, which mainly depends on the number of constraints and the total constraint execution time, while it is only slightly affected from the distribution of constraints and the constraint scheduling policy. Finally, it is explained how the PRACTIC system enhances the algorithm's performance using features, like nested query parallelism and constraint overlapping.

Key words: passive constraints, parallel constraint checking, object-oriented databases, parallel database systems, independent-task parallelism

C.R. Categories: H.2.0, H.2.6, I.6.8

1. Introduction

Object-oriented databases (OODB) gained considerable amount of interest [8], because they can model real-world concepts for complex modern applications. These applications impose many complicated relations and restrictions among data, named semantic integrity constraints. Constraints ensure the validity of data held in the database, according both to the system and the user, and also ensure that subsequent updates on data will not violate these restrictions. There are many important research topics associated with constraints in databases [10], like constraint semantics, specification and enforcement.

Although, constraints are considered vital for database applications, there is a major preventive factor for using complex constraints in large databases, namely their enforcement performance cost [9]. The work described here is concerned with the improvement of constraint enforcement performance in OODBs, by checking constraints in parallel.

This paper proposes a parallel algorithm, based on the master-slave technique, for checking passive constraints, a violation prevention strategy for constraint enforcement. The algorithm executes independent passive constraints using the operational semantics of AND-parallelism. Furthermore, it is shown how this algorithm can be implemented in the parallel OODB system, named PRACTIC [2]. The algorithm is benefited from the parallel database system, since nested parallelism of queries and sharing of common results between dependent constraints lead to significant speed-up in constraint checking.

In the following, after an introduction to constraint checking in section 2, we present the parallel constraint checking algorithm in section 3. In section 4, we describe briefly the PRACTIC model and its abstract machine and then we propose an implementation of the constraint checking algorithm on PRACTIC. In section 5, we compare performance of parallel against sequential constraint checking, by simulation. Finally, section 6 concludes the paper with a summary of our work and a discussion of future work.

2. Constraint Checking in Object-Oriented Databases

The imperative need for integrity maintenance in database systems is a long recognized fact [10]. However, it is still an active research area, since many issues involving integrity constraint specification and enforcement need to be resolved. Object-oriented databases gave a new rise in the interest for integrity constraints because of the new structural and semantics capabilities they offer. Constraint maintenance in OODBs differs from that in relational databases [12] mainly due to the localized control and rich modeling constructs of the former. However, the following important issues must still be addressed to, in OODBs:

- When a constraint is checked, before or after the attempted transition?
- How to detect constraint violation? A constraint compilation approach has been widely used [3, 7, 12], where the declarative constraint specification is transformed into a database query or a set of inserted and deleted objects.
- How to recover the integrity of the database when a constraint is violated?

  Constraints are classified according to their integrity recovery actions:

  Passive constraints, which are checked for validity every time an update on the database is attempted and if their validity is violated, then the update is rejected [3, 12].

  Active constraints, maintain database integrity by performing the update and trying to recover integrity violation by asserting and/or deleting data from the database [6].

Active constraint checking is usually implemented by event-condition-action (ECA) rules [5], an extension of production rules with an extra pre-condition context of triggering, called event. A rule is considered only when its event has been detected in the database, after an update or an internal event, etc. The algorithms for executing database rules in parallel are quite complex [1, 11], to cater for interferences of rules and impose rule serializability in order to preserve the semantics of the corresponding sequential production systems.

On the other hand, parallel enforcement of passive constraints has not been adequately tackled in the literature, because they represent a less general (but not least important) aspect of database integrity. Passive constraint enforcement is simpler and more efficiently implemented by mechanisms based on method invocation [3, 7, 12]. Furthermore, passive constraints are independent from each other and they can be executed in parallel.

Parallel constraint checking has been studied for relational databases [9], where a single constraint is checked in parallel on a distributed relation, while multiple constraints are checked sequentially. In contrast, our work is concerned with parallel checking of multiple constraints on a single tuple update, since OODBs are more tailored to single-object transactions. Furthermore, we also consider the case for constraint violation.

3. The Parallel Constraint Checking Mechanism

Passive constraints are independent from each other, because they do not share variables and the execution of one does not affect the rest. Furthermore, the order of execution is not important, since no modifications are performed on the database. Thus, the operational semantics of passive constraints is AND-parallelism. All constraints must be satisfied to accept the update, while one violation suffices to reject it. Their evaluation on the current update can proceed in parallel.

The transaction model used throughout this paper constitutes of single update messages, sent to individual objects, i.e. tuple-at-a-time semantics according to the relational database terminology. Therefore, constraint checking is performed only at a single class, the one that the instance object belongs to. Constraints are of preventive nature, i.e. they are checked before the transition and if violated they reject it.

Constraints are mandatory; normally, every object should satisfy the constraints defined at its class and all its super-classes [3]. However, some constraints may not be applicable to a class, because they are overridden by more specific ones. Constraint overriding, exception and subsumption are factors that alternate the default constraint checking mechanism and reduce the total number of constraints that an update must satisfy. These features are really part of constraint analysis and design and have little to do with run-time constraint checking. In our discussion, we assume that a constraint analyser has already re-solved the above problems.

3.1. Constraint Checking Algorithm

Here we discuss the parallel constraint checking (PCC) algorithm (listing in appendix A), which requires the following processes and processor resources:

- A master processor that schedules constraint execution and retains the top-level of control,
- One or more slave processors that execute the constraints.

The master processor runs the MAIN algorithm, which is responsible for initializing the system and then gives control to SCHEDULE. The scheduler process is responsible for:

- Allocating the applicable pieces of code that enforce the constraints to the slaves.
- Gathering and evaluating the results returned by the slaves.
- Co-ordinating and monitoring the state of the slaves.

The scheduler also constantly monitors (through the algorithms SEARCH1 and SEARCH2) the slaves' messages, in order to diagnose constraint violation (if a constraint is violated) as soon as possible and avoid unnecessary execution of constraints. The SEARCH1 algorithm is responsible for checking if any constraint has finished execution (either successfully or not), while the scheduler is still allocating constraints. On the other hand, the SEARCH2 algorithm does the same thing but only after every constraint has been allocated.

The SCHEDULE algorithm does not determine which strategy is being used to pick up a constraint and assign its execution to a slave processor. These two parameters can be adjusted to control system's performance and behaviour. Currently, since we have not outlined any features that differentiate constraints, a constraint is chosen at random. The allocation is based on the three scheduling policies described in the simulation section (ALLOCATE algorithms).

The slave processes (algorithm SLAVE) are all equivalent and their task is to receive and execute constraint bodies from the master processor. When a constraint finishes execution, the slave informs the master and waits for the next constraint, unless the previous one has been violated. Slaves also constantly monitor master's messages to detect early termination of the algorithm, due to constraint violation at another slave. Slaves do not directly communicate with each other, as this would complicate synchronization and jeopardize the abstraction of the algorithms and the extendibility of its implementation.

4. Algorithm Implementation on PRACTIC

The algorithm presented is quite abstract and could be implemented on any parallel database system with client-server features. In this section we describe how it has been implemented on a parallel OODB system, named PRACTIC [2].

4.1. Overview of PRACTIC

PRACTIC [2] is a parallel OODB system, based on the concurrency of the active database objects. The latter are objects that can have direct instances (e.g. classes). The PRACTIC model provides class-hierarchy parallelism, which offers a significant speed-up in query evaluation [14]. Queries received by a class are evaluated in parallel by that class and its sub-classes, since the extension of a class is a sub-set of all its super-classes (appendix B).

An abstract machine (APRAM) for executing PRACTIC model has also been proposed [2]. APRAM (fig. 1) consists of a hierarchical network of processing elements (PEs), of depth 2. The PEs of depth 1 are called Active Object Processors (AOP) and execute the active class processes. Each AOP has zero, one or more attached PEs, called Active Extension Managers (AEM) and manage the instances of the active class (class extension).

Some of the AEMs have a disk attached to store a partition of the class extension, while the rest are disk-less and are used for general purposes, like method execution, constraint checking, etc. The AEMs of one class can communicate with the rest of the system only through their AOP, while all AOPs can communicate directly.

The APRAM architecture also provides for intra-class parallelism:

- Method execution pipeline. When a method is specialized along the class hierarchy, then method execution on a set of selected objects can proceed in a stream-parallel fashion among the method specializations. Disk-less AEMs support method execution pipeline.
- Object partitioning. The class extension is partitioned among the AEMs, providing data parallelism.

Following the conventions of logic programming, class-hierarchy parallelism and data parallelism are equivalent to OR-parallelism and method execution stream-parallelism is a form of AND-parallelism with dependencies.

4.2. Mapping the Algorithm on APRAM

Both types of intra-class parallelism, mentioned above, do not suffice to perform an efficient constraint check on the updates attempted by the user. If constraints are regarded as pieces of code to be executed using the default method execution strategy, then APRAM will allow only stream parallelism. However, since passive constraints are independent, there is not a preferred order of execution and the constraint bodies can be executed AND-parallel.

The slave processes of our parallel constraint checking mechanism run on the disk-less AEMs, of the APRAM, while the scheduler process runs on the AOP itself. The constraint-scheduling process is a part of the more general scheduler of the AOP, responsible for allocating various pieces of code (i.e. methods, constraints, active rules, etc. [1, 2]) for execution to the AEMs (fig. 1).

The AEMs receive constraint bodies and execute them. When a constraint issues a query to the database, it is forwarded back to the AOP. The latter is responsible for evoking the appropriate query evaluation procedures and return the result(s) to the constraint. When AEMs receive termination messages from the scheduler, then the currently active constraint-processes and the spawned queries must terminate. This is achieved by keeping information about each constraint-process running on the AEMs, such as the currently active spawned queries. The termination messages will cause anti-messages [13] to be sent to the AOP, which will accordingly forward them to the target classes of the spawned queries.

4.3. Optimizing Constraint Execution

The implementation of the above constraint checking mechanism on APRAM leaves room for various optimizations, which are based on the parallel nature of the PRACTIC model. More specifically, these are: a) Nested Query Parallelism, and b) Constraint Overlapping.

Nested Query Parallelism

The actual constraint-checking code consists of database queries, in the form of messages (appendix B). Therefore, constraint checking can be benefited from a speed-up in query execution. The PRACTIC system was designed for this aim, through class-hierarchy and intra-class parallelism [2]. Thus, constraints are not only executed in parallel, but also queries issued by the constraints are evaluated in parallel. This nested query parallelism reduces constraint checking time for a single constraint.

Constraint Overlapping

Passive constraints are independent from each other, but they may overlap, i.e. their checking condition may include one or more similar queries. These constraints can still be executed independently but this leads to the existence of redundant queries and unnecessary checking overhead. In appendix B there is an example of two different constraints that apply to the same class and method and they overlap.

In APRAM the redundancy is avoided, because queries are always directed to the local AOP, which is responsible for activating the appropriate (may be remote) procedures for answering them. Therefore, if more than one constraints issue the same query, the AOP receives several times the same request but evokes the answering mechanism only once. When the results to the query are obtained, they are broadcast to all the requesting constraints.

The constraint scheduler of the AOP detects the above constraint overlapping case before allocating these two constraints to AEMs and uses it as an extra heuristic in order to assign the execution of these two constraints to the same AEM, if possible. This reduces message traffic, since only a single copy of the query result is propagated in the local network.

5. Simulation

The proposed algorithm has been tested on a simulated model, on the same platform[1] that PRACTIC system is being implemented. The algorithms presented are closely simulated, using, however, a do-nothing time period instead of an actual constraint body execution. Nevertheless, the duration of this do-nothing period time is carefully chosen to provide a realistic measurement for the actual system.

5.1. Simulation Parameters

The main parameters used in the simulations are described in the following:

Constraint Execution Time

Since constraint execution is simulated, the major parameters that define constraint duration, namely: a) constraint body complexity, b) method execution time, and c) distribution of constraints in a given time range, must be justified.

Each high-level constraint expression is compiled into an OODB query that constitutes of one or more messages and its complexity depends on the complexity of the corresponding methods. We assume that there are constraints with bodies consisting of 1 to 5 messages, which is a realistic assumption for most common cases [3].

The actual constraint execution time depends on both the number of messages and the execution time of each corresponding method. Based on the simulations for method execution in [2], we used 0.01 to 1 sec. The actual range, within the constraint execution times lie, is between 1 and 5, multiplied by 0.01 to 1 sec.

The distribution of constraints within the above time range depends on the nature of the application. We used six different distributions: a) Gaussian, b) Random, c) Uniform, d) Dirac, e) increasing linear, and f) decreasing linear.

However, not all of the above distributions correspond to realistic situations. It can be said that the decreasing linear distribution is the most realistic, since it assumes that there are many "cheap" and less "expensive" constraints. This is usually the case in real world applications, where complex constraints are more rare than simple ones. This distribution though, is optimistic, since it assumes that not many "expensive" constraints exist. The random distribution is the most fair one, since it does not pre-suppose any preferences, but it may not be realistic. Despite these remarks, the "uniform" distribution has been chosen for all the other simulation cases, since it is both quite fair and not very optimistic.

Number of Constraints

This parameter represents the average number of constraints checked for each individual update. However, this does not denote the total number of constraints of the database schema, but the number of applicable constraints on the update. In the simulation 5 to 50 constraints were used, to cover even for the most extreme cases.

Constraint Scheduling Policy

The SCHEDULE algorithm assigns constraint execution tasks according to some allocation strategy (algorithm ALLOCATE). The allocation strategies used in the simulation are: a) Round-Robin, b) Minimum number of Tasks, and c) Free Processor.

The Round-Robin is a well known strategy: each new constraint is assigned to the next slave in a circular fashion. Tasks are assigned to processors without waiting for the previous one to finish, because our development platform allows asynchronous messaging. This buffered communication does not risk safe message delivery, since buffers cannot get full with only few constraints. Thus, this kind of buffered communication is suited to this kind of master-slave architecture and is indeed proved better than an un-buffered scheme.

The second strategy assigns tasks to processors according to the “Minimum number of Tasks” criterion, that is the processor with the minimum number of currently assigned tasks, is assigned the next one. Finally, the third strategy assigns a task to a processor only after the previous task on the processor has finished execution, i.e. synchronous communication.

Number of Slave Processors

The number of slaves tests the speed-up behaviour of the PCC algorithm. We used 1 to 4 slaves. Using only one slave turns constraint-checking into sequential, but still two processors are used: the master and a slave. The comparison of this case with the “pure” sequential algorithm, provides an estimate for the overheads caused by the parallelisation of constraint checking, i.e. process scheduling and inter-processor communication.

Constraint Satisfaction and Violation

Finally, simulations are divided into constraint satisfaction and violation cases. All simulations are performed twice, both for success of all constraints and for the failure of a single constraint. The choice of the constraint to fail is random. The case for more failed constraints is not tested, since only the first one suffices to reject the update.

The average time of constraint execution is out of 20 runs for constraint satisfaction and 50 runs for violation. The latter is justified since the violating constraint is chosen randomly. The corresponding number of runs for the sequential execution are 10 and 25, respectively.

5.2. Simulation Results

In this section we present and analyse five different simulation test cases. Table 1 summarizes the parameters for the simulation cases. The performance is measured in terms of speed-up, which is defined [4], as the fraction:

$$S = \frac{T_{seq}}{T_{par}},$$

where T_{seq} is the constraint checking time of the sequential algorithm and T_{par} is the equivalent time of the parallel algorithm.

Constraint Number Scale-up (PCC1A)

This case tests the scalability of the algorithm, i.e. its behaviour when the only parameter that changes is the number of applicable constraints. Simulation shows (fig. 2) that speed-up (in case of constraint satisfaction) is linear for a small to a medium number of constraints, while it saturates at higher numbers. Fortunately, the case for more constraints should not be considered, since 50 constraints is high enough.

When a single constraint is violated, the behaviour is similar. We notice that the curve for violation represents the average-case, but the worst- and best- cases differ considerably. This and the random choice of the failed constraint are the reasons for the un-stability of the curve, compared to the (more) regular curve of constraint satisfaction.

Total Constraint Time Scale-up (PCC1B)

This case considers time scale-up, i.e. the only parameter that changes is the total constraint execution time, while the number of constraints remains fixed. This can be achieved by changing the time scale of constraint execution times, which is a factor multiplied by the times produced by the constraint generator.

Results show (fig. 3) that the speed-up saturates at high constraint execution times[2]. There is no difference between the curves for satisfaction and violation. The violation curve is more stable, since all measurements involve a fixed number of constraints, thus random selection of the failed constraint has almost the same average for all cases.

Constraint Number Speed-up (PCC2A)

This case tests the behaviour of the algorithm when the number of processors increases, while the size and number of tasks remains fixed. We performed this test is four times, to detect possible differences in the behaviour of the speed-up curves for different number of constraints. As the graphs (fig. 4, 5) show, the speed-up curves exhibit the same behaviour for all tests from 5 to 50 constraints. The speed-up is slightly under-linear for both constraint satisfaction and violation. For the violation case, the more constraints, the closer the speed-up curve is to become linear. This is an outcome of the fact that the algorithm offers considerable gains when tasks get large.

We use the measurements for one processor to calculate the parallelisation overheads, in table 2, where the time and the percentage of the overhead are presented. It is obvious that the percentage is almost constant and the absolute parallelisation overhead is linear to the number of constraints. The “violation” case is un-stable because of the extra communication between the master and the slaves, when a constraint is violated.

Total Constraint Time Speed-up (PCC2B)

This case tests how the algorithm behaves when the number of processors increases, while the size of the task remains fixed. Different tests are performed by keeping the total number of constraints fixed, for different total constraint execution times.

From the graphs (fig. 6, 7) we can derive the following:

- Speed-up curves for large time scales (1 sec) exhibit very good, linear, behaviour for both satisfaction and violation. The corresponding tasks are suited for coarse-grain parallelism.
- Speed-up curves for medium time scales (0.1 sec) exhibit under-linear behaviour for both satisfaction and violation.
- Speed-up for small time scales (0.01 sec), a) decreases when the number of processors increases and b) is constantly under speed-up 1, thus sequential execution is better.

Communication and scheduling overheads are calculated in table 2. It is clear that the overheads are almost constant in absolute value, since the number of constraints (and messages) is constant. The percentage is very large (> 50%) when tasks are fine (scale 0.01 sec), while it gets negligible (< 1%) when constraints are complex (scale 1 sec). For very large tasks, the communication overhead is zero, because the master processor assigns all the tasks before the first task finishes and the slave processor does not wait. On the other hand the sequential algorithm has little, but important, overhead from the loop management (recursion or backtracking in Prolog). The latter explains the negative sign in the “violation” case.

Comparison of Scheduling Policies for Different Constraint Distributions (PCC3)

A final set of simulation cases includes the test of the behaviour of the algorithm for different distributions of constraints and different scheduling policies. As graphs show (fig. 8, 9) no single scheduling policy can be considered the best, since for different constraint distributions, the relative speed-ups are different. However, it can be noticed that all three are very close to each other and the relative differences are less than 10% in most cases.

The results for the three most important distributions are:

- The “Minimum Tasks” policy is slightly better than the rest for both satisfaction and violation, at “decreasing-linear” distribution.
- The “Minimum Tasks” policy is slightly better for satisfaction, while the “Free Processor” policy is considerably better for violation, at “random” distribution.
- At “uniform” distribution, the “Round-Robin” policy is better for satisfaction, while the “Minimum Tasks” policy is better for violation.

6. Conclusions and Future Work

In this paper we have proposed a parallel constraint checking algorithm for OODBs. The algorithm executes, in parallel, passive constraints defined on a class and/or inherited from its super-classes. The algorithm is based on the master-slave technique of parallel programming and depends on the pure independent parallelism of constraints, contrasted to the pipe-lined parallelism of method execution ([2]).

The implementation of the proposed algorithm on the PRACTIC machine [2] offers additional opportunities for parallelism and speed-up, as: a) nested query parallelism, and b) sharing of intermediate results between dependent constraints.

Our algorithm differs from trivial master-slave algorithms, in the following aspects:

- Tasks are not fine-grained and are usually only few tens,
- Tasks can fail and this means that there is no need to execute the rest of pending tasks,
- Constraint violation imposes additional synchronization between the processors.

The algorithm offers considerable speed-up that is dependable upon the number of constraints and/or the total constraint execution time, as the simulation showed. However, there are some cases that the parallel algorithm suffers from the inter-processor communication overheads, that need to be addressed by the constraint scheduler. Other factors like distribution and scheduling policies affect speed-up only weakly.

Future work on parallel constraint checking includes the classification of constraints according to computational complexity. This will provide for a rough estimation of the expected execution time and guide the constraint scheduler to: a) choose between the sequential or the parallel algorithm, and b) assign priorities to constraints for execution. Priorities among constraints can also be derived from statistical information collected during the database life-time, based on the “fail as soon as possible” heuristic rule of logic programming.

Acknowledgments

The first author is supported by a scholarship from the Greek State Scholarships Foundation (I.K.Y.).

References

[1] N. Bassiliades and I. Vlahavas, A multi-processor machine for the parallel management of an active object-oriented database (in Greek), *Proc. 4th Hellenic Conf. on Informatics*, Patras, Greece, Dec. 1993, pp. 501-512.

[2] N. Bassiliades and I. Vlahavas, PRACTIC: A concurrent object data model for a parallel object-oriented database system, submitted for publication, 1993.

[3] N. Bassiliades and P.M.D. Gray, CoLan: A functional constraint language and its implementation, forthcoming in *Data & Knowledge Eng*.

[4] D. DeWitt and J. Gray, Parallel database systems: The future of high performance database systems, *CACM* **35** (1992), 85-98.

[5] O.G. Diaz, N.W. Paton and P.M.D. Gray, Rule management in object oriented databases: A uniform approach, *Proc. 17th VLDB*, 1991, pp. 317-326.

[6] O.G. Diaz and S.M. Embury, Generating active rules from high-level specifications, *Advanced Database Systems - Proc. 10th British National Conf. on Databases*, Aberdeen, Scotland, Springer-Verlag, 1992, pp. 227-243.

[7] S.M. Embury, P.M.D. Gray and N.D. Bassiliades, Constraint maintenance using generated methods in the P/FDM object-oriented database, *Proc. Internat. Workshop on Rules in Database Systems,* Edinburgh, Scotland, 1993.

[8] P. M. D. Gray, K. G. Kulkarni and N. W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach*, Prentice Hall, 1992.

[9] P.W.P.J. Grefen, J. Flokstra and P.M.G. Apers, Performance evaluation of constraint enforcement in a parallel main-memory database system, *Proc. 3rd Internat. Conf. on Database and Expert System Applications,* Valencia, Spain, 1992, pp. 96-101.

[10] P.W.P.J. Grefen and P.M.G. Apers, Integrity control in relational database systems - An overview, *Data & Knowledge Eng.* **10** (1993), 187-223.

[11] I.-M. Hsu, M. Singhal and M.T. Liu, Distributed rule processing in active databases, *Proc. IEEE Internat. Conf. on Data Engineering* , 1992, pp. 106-113.

[12] H.V. Jagadish and X. Qian, Integrity maintenance in an object-oriented database, *Proc. 18th Internat. Conf. on Very Large Data Bases,* Vancouver, Canada, 1992, pp. 469-480.

[13] P. Kacsuk and I. Futo, Multi-transputer implementation of CS-Prolog, *Proc. of AI and Communication Process Architecture*, Wiley and Sons, 1989, pp. 131-148.

[14] K.C. Kim, Parallelism in object-oriented query processing, *Proc. 6th Internat. IEEE Conf. Data Engineering*, 1990, pp. 209-217.

Appendix A - The Constraint Checking Algorithm

We will now present the listing of the algorithm described in section 3.1. The listing is in some cases simplified for illustrative purposes. Some of the symbols used are:

- `class`, is the class of the recipient object,
- `MaxSlaves`, is the total number of slaves used in the system,
- `executing`, is the counter of the currently executing constraints, and
- `tasks(Slave)`, is the number of the currently assigned tasks to `Slave`.

The constraint-objects have the following slots:

- `head`, defines the message that the constraint is relevant to,
- `code`, stores the compiled database program that checks the constraint,
- `slave`, stores which processor the constraint is executed on,
- `applicable`, is set to `true` when the constraint is relevant to the current update.

<u>Algorithm MAIN</u>

1. Wait until a message `message` arrives.
2. Set `LIST` to `nil`.
3. Until there are no more constraints in `class`, do:
    a. Get the next `ConstOID` in `class`.
    b. If `head(ConstOID)` matches `message`, then put `ConstOID` in LIST.
4. Set `executing` to 0.
5. For each `ConstOID` in `LIST`, set `applicable(ConstOID)` to `true`.
6. For each slave processor `i`, create a SLAVE(`i`) process on it.
7. Call SCHEDULE.
8. Go to step 1.

<u>Algorithm SCHEDULE</u>

1. While `LIST` is not empty do steps 2-8.
2. Call SEARCH1
3. If SEARCH1 returns `false`, then return (to MAIN).
4. Pick-up (in random) a `ConstOID` from `LIST`.
5. Call ALLOCATE.
6. If `Slave=0` then go to step 2.
7. Send the message `constraint(ConstOID,code(ConstOID))` to SLAVE(Slave).
8. Set `slave(ConstOID)` to `Slave` and increase `executing` by 1.
9. Call SEARCH2 and return (to MAIN).

<u>Algorithm SEARCH1</u>

1. Check if any slave processor has sent a `fail(ConstOID)` message to the scheduler.
2. If yes, then call FAIL and return `false` (to SCHEDULE).
3. Check if any slave has sent a `success(ConstOID)` message to the scheduler.
4. If yes, then call SUCCESS.
5. Return `true` (to SCHEDULE).

<u>Algorithm SEARCH2</u>

1. Check if any slave processor has sent a `fail(ConstOID)` message to the scheduler.
2. If yes, then call FAIL and return `false` (to SCHEDULE).
3. Check if `executing` is 0.
4. If yes, then constraint checking terminates successfully. Do the following:
    a. Send an `end(true)` message to all slaves.

b. Accept the update and return (to SCHEDULE).
5. Check if any slave has sent a `success(ConstOID)` message to the scheduler.
6. If yes, then call SUCCESS.
7. Go to step 1.

Algorithm FAIL

1. Set `Slave` to `slave(ConstOID)`.
2. Send an `end(fail)` message to all (but `Slave`) slaves.
3. Reject the update and inform the sender object about the violated constraint.
4. For each `ConstOID` such that `applicable(ConstOID)=true`, do:
   a. Re-set `applicable(ConstOID)` to `false`.
   b. Delete the value of `slave(ConstOID)`.
5. Return (to SEARCHx).

Algorithm SUCCESS

1. Delete the value of `slave(ConstOID)`.
2. Set `applicable(ConstOID)` to `false`.
3. Decrease `executing` by 1.
4. Return (to SEARCHx).

Algorithm SLAVE(i)

1. Wait until a message `M` arrives.
2. If `M` matches `end(_)` then end.
3. If `M` matches `constraint(ConstOID,Body)` then execute `Body`.
4. If `Body` succeeds, then send a `success(ConstOID)` message to the scheduler and go to step 1.
5. Else, send a `fail(ConstOID)` message to the scheduler and end.

There are 3 alternative algorithms for task allocation:

Algorithm ALLOCATE (Round-Robin)

1. When the algorithm runs for the first time, set `CurrentSlave` to `0`.
2. Set `CurrentSlave` to `CurrentSlave+1`.
3. If `CurrentSlave>MaxSlaves` then set `CurrentSlave` to `1`.
4. Return `CurrentSlave` (to SCHEDULE).

Algorithm ALLOCATE (Min. no. of Tasks)

1. When the algorithm runs for the first time:
   a. For each `Slave` from `1` to `MaxSlaves` set `tasks(Slave)` to `0`.
2. Find the minimum `tasks(Slave)`.
3. Set `tasks(Slave)` to `tasks(Slave)+1`.
4. Return `Slave` (to SCHEDULE).

<u>Algorithm ALLOCATE (Free Processor)</u>

1. When the algorithm runs for the first time:
   a. For each `Slave` from `1` to `MaxSlaves` set `tasks(Slave)` to `0`.
2. Search for a `Slave` such that `tasks(Slave)=0`.
3. If such a `Slave` does not exist, return `0` (to SCHEDULE).
4. Otherwise, set `tasks(Slave)` to `1`.
5. Return `Slave` (to SCHEDULE).

Appendix B - Examples of Constraint Translation and Optimization

Constraints are described using the declarative language CoLan [3] and are translated into the database programming language of PRACTIC[3]. For example, the following constraint `C1`, restricts classes `lecturer` and `post_graduate` of the university schema (fig. 10):

```
forall l in lecturer
   exist at most 5 p in post_graduate
      such that l = supervisor of p
```

Constraint `C1` is compiled into the following database program, for the method `put_supervisor` of class `post_graduate`:

```
put_supervisor(...,[Lecturer]) :-
     message_recipient(P),
     findall(P1,get_by_supervisor([Lecturer],P1)=>post_graduate,List),
     length(List,Length),
     ifthenelse(Length>=5,fail,true).
```

Suppose there exists another constraint `C2`, that applies to the same classes and methods:

```
forall l in lecturer
   exist at most 2 p in post_graduate
      such that l = supervisor of p
      and subject of l \= subject of p
```

Constraint `C2` is translated into the following program:

```
put_supervisor(...,[Lecturer]) :-
    message_recipient(P),
    get_subject(SubjPost) => P,
    get_subject(SubjLect) => Lecturer,
    ifthenelse(SubjPost\=SubjLect,
         (findall(P1,(get_by_supervisor([Lecturer],P1) => post_graduate,
                   get_subject(SubjPost1) => P1,
                   SubjPost1\=SubjLect),
              List),
         length(List,Length),
         ifthenelse(Length>=2,fail,true))
         true).
```

Constraints `C1` and `C2` are dependent, since they both contain the same query (in italics):

```
get_by_supervisor([Lecturer],P1) => post_graduate,
```

which retrieves all post-graduate students with the same supervisor. If it was not for the constraint overlapping optimization, the query would be issued twice. Furthermore, the query is evaluated in parallel among classes `post_graduate` and `pgra`, due to class-hierarchy parallelism. Thus, queries nested in constraints are also evaluated in parallel, speeding-up constraint execution.

Figure/Table captions

	PCC1A	**PCC1B**	**PCC2A**	**PCC2B**	**PCC3**
Time Range (sec)	0.1-0.5	0.01-5	0.1-0.5	0.01-5	0.1-0.5
Distribution	uniform	uniform	uniform	uniform	*all*
No. of Constraints	5-50	25	5-50	25	25
Scheduling Policy	min	min	min	min	*all*
Satisfaction	T/F	T/F	T/F	T/F	T/F
No. of Slaves	4	4	1-4	1-4	4

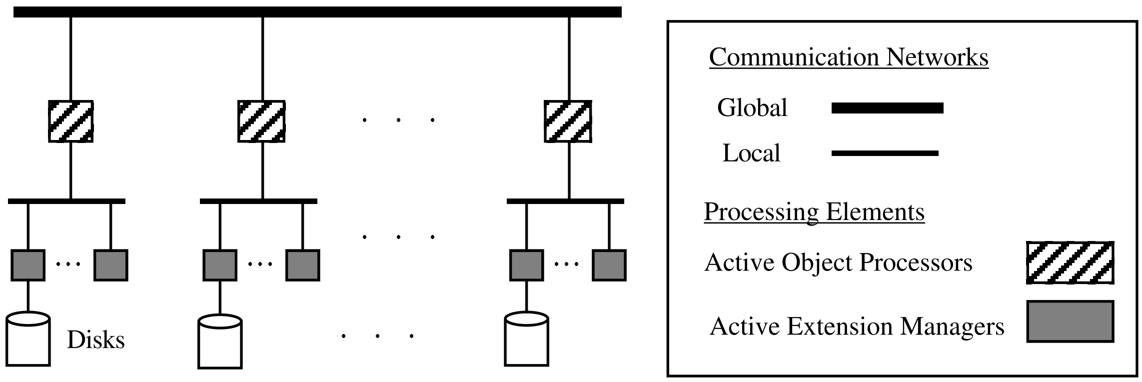
N. Bassiliades and I. Vlahavas

Table 1

	No. of Constraints - PCC2A				Time Scale (sec) - PCC2B		
	5	10	25	50	0.01	0.1	1
TRUE	0.31 (19%)	0.53 (16%)	1.40 (17%)	2.86 (18%)	1.01 (64%)	1.40 (17%)	0.73 (1%)
FAIL	0.29 (29%)	0.84 (51%)	0.75 (18%)	0.79 (9%)	0.73 (91%)	0.75 (18%)	-2.72 (-7%)

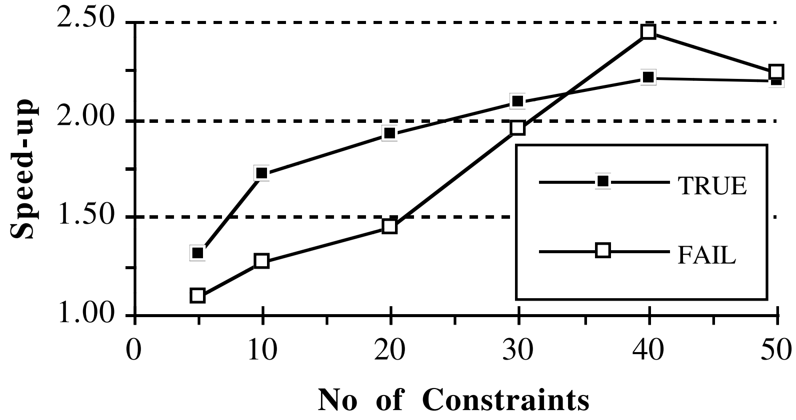
N. Bassiliades and I. Vlahavas

Table 2



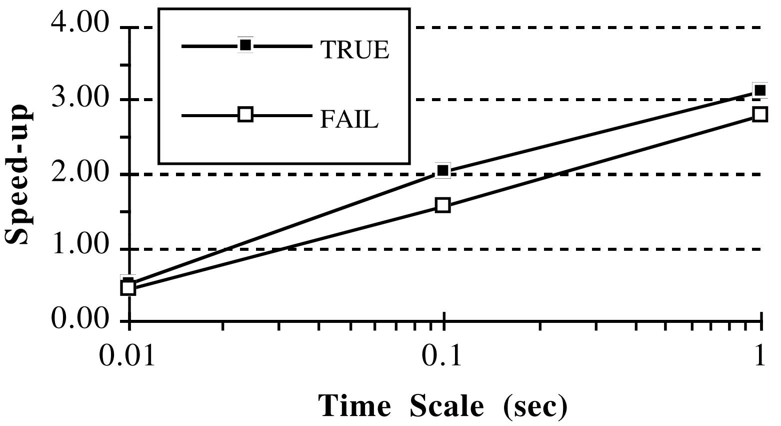
N. Bassiliades and I. Vlahavas

Figure 1



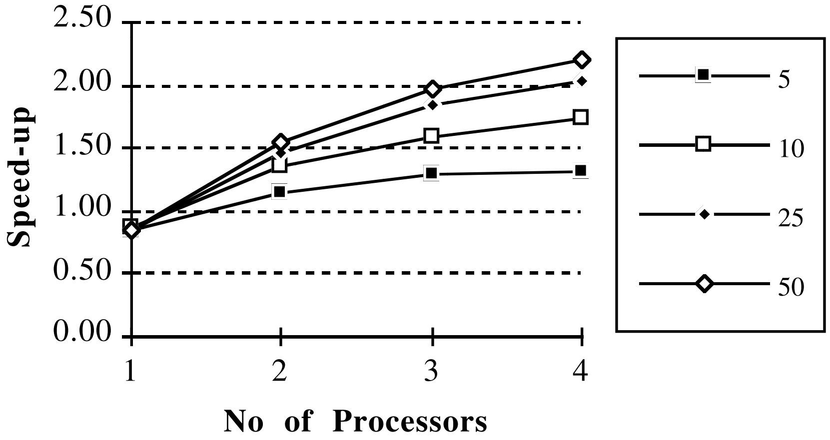
N. Bassiliades and I. Vlahavas

Figure 2



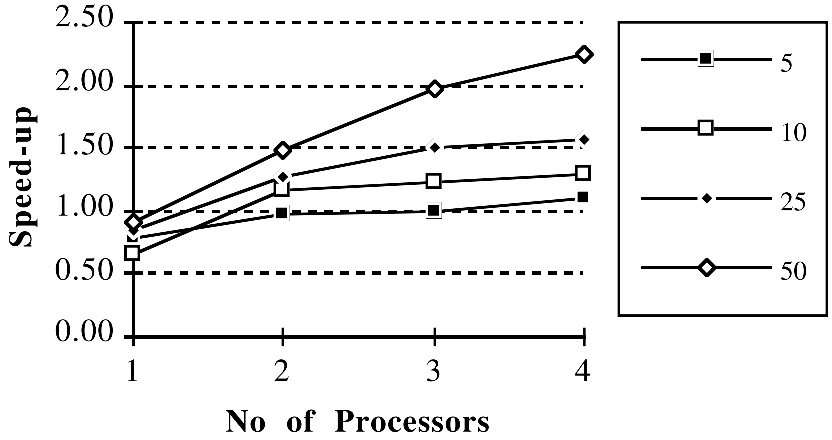
N. Bassiliades and I. Vlahavas

Figure 3



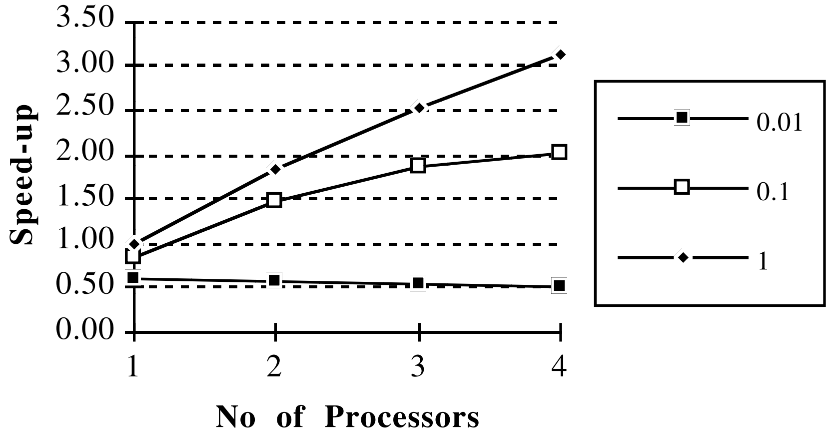
N. Bassiliades and I. Vlahavas

Figure 4



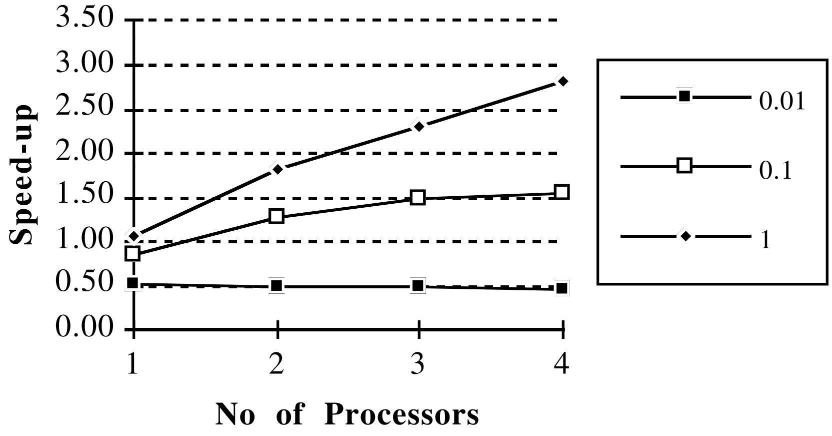
N. Bassiliades and I. Vlahavas

Figure 5



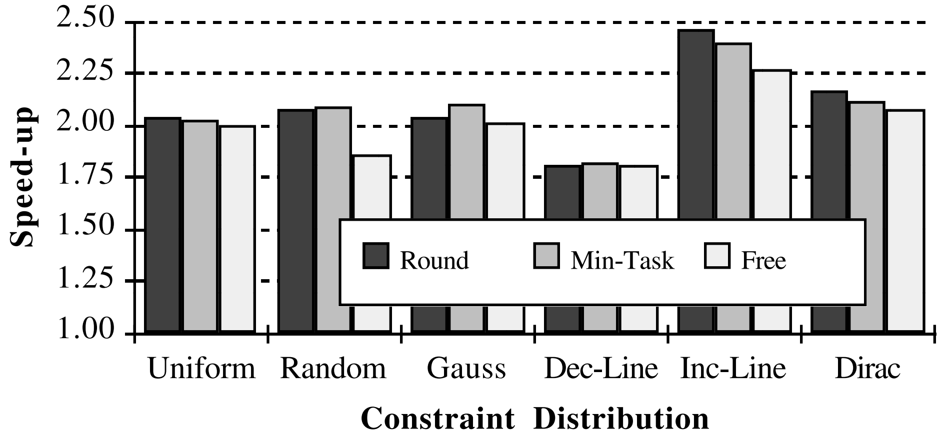
N. Bassiliades and I. Vlahavas

Figure 6



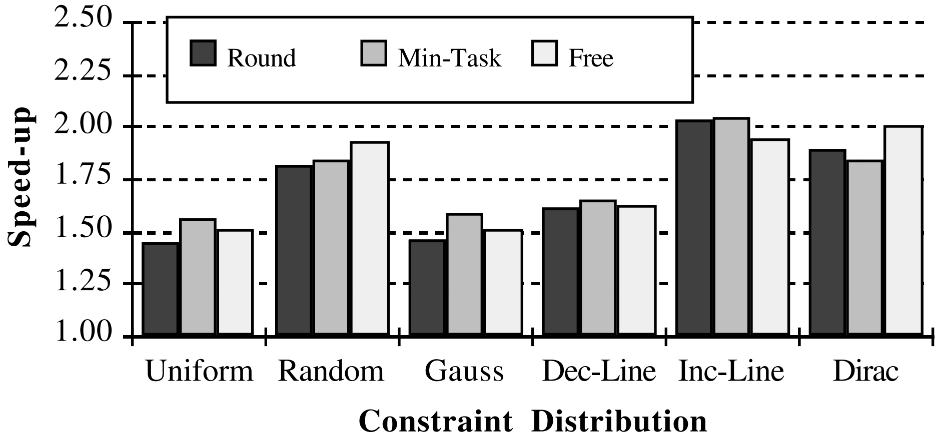
N. Bassiliades and I. Vlahavas

Figure 7



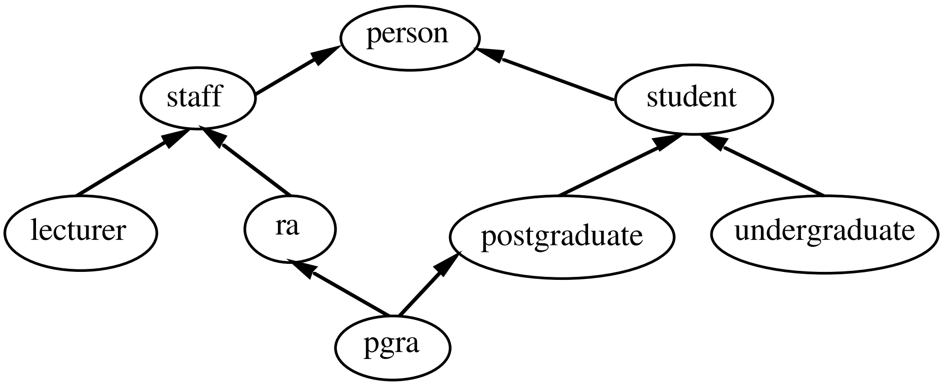
N. Bassiliades and I. Vlahavas

Figure 8



N. Bassiliades and I. Vlahavas

Figure 9



N. Bassiliades and I. Vlahavas

Figure 10

Footnotes

[1] A transputer board with CS-Prolog [13] as the implementation language.

[2] Speed-up is linear in a semi-log diagram, therefore it saturates in a linear diagram.

[3] The programming language of PRACTIC is an extension of the language of ADAM [8].