1-1-1990

Automatic Parallelization of Database Queries

Myong H. Kang
Purdue University

Henry G. Dietz
Purdue University

Kang, Myong H. and Dietz, Henry G., "Automatic Parallelization of Database Queries" (1990). *Department of Electrical and Computer Engineering Technical Reports.* Paper 699.
https://docs.lib.purdue.edu/ecetr/699

Automatic Parallelization of Database Queries

Myong H. Kang
Henry G. Dietz

TR-EE 90-7
January, 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Automatic Parallelization of Database Queries

Myong H. Kang and Henry G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

Although automatic parallelization of conventional language programs is now widely accepted, relatively little emphasis has been placed on automatic parallelization of database query programs (sometimes referred to as ''multiple queries''). In this paper, we discuss the unique problems associated with automatic parallelization of database programs. From this discussion, we derive a complete approach to automatic parallelization of database programs. Beside integrating a number of existing techniques, our approach relies heavily on several new concepts, including the concepts of ''algorithm-level'' analysis and hybrid static/dynamic scheduling.

1. Introduction

High level relational database query languages such as SQL and QUEL allow users to express what information is desired, not how to obtain it. This enables applications to be independent from details of not only secondary storage management but also the target machine. The query optimizer is responsible for determining the strategy for efficiently evaluating the queries presented by the user.

Another way to increase performance is through the use of parallelism. Recent technological advances have made it cost-effective to build database machines which have multiple processors and multiple disk drives. However, parallel processing does not guarantee better performance; performance will only be improved if the parallelism of the programs can be made to match the parallelism of the hardware.

Parallel execution can be either explicitly stated in the program or mechanically derived from the implicit constraints on parallelism imposed by the algorithm's structure. For example, the XPRS database system [Sto88] allows the programmer to use the keyword **parallel** to specify that two POSTGRES commands are to be executed in parallel.

However, we believe automatic detection of parallelism is feasible and that the advantages far outweigh the difficulties in implementation. An obvious advantage of automatic parallelization is that it eases the burden on the programmer. Less apparent, but perhaps more important, is the fact that only command-level parallelism can be expressed directly in a command language, whereas automatic parallelism detection can detect parallelism at any other level (e.g., at "algorithm level"). Further, in automatic parallelization the parallel structure can be selected based on intimate knowledge of the target machine. For these reasons, automatic parallelization can be expected to find more useful parallelism than a typical user would specify.

In this paper, a framework for automatic parallelization of database queries which can be applied to not only individual query but also a sequence of queries is described. Throughout this paper, individual query is treated as a special case of multiple queries.

In section 2, a summary of relevant work is presented, discussing both conventional program parallelization and previous work on query parallelization. Terminology and symbols to be used for the rest of the paper are defined in section 3. Section 4 discusses issues relating to the level at which parallelism detection is performed. A brief overview of the automatic parallelizer, and our target machine model, is given in section 5. Section 6 gives a more detailed discussion of the proposed process packaging and scheduling methods. Finally, section 7 summarizes the paper and indicates the direction of future research.

2. Related Work

Automatic parallelization of database queries which is introduced in this paper appears to be entirely new work, however, automatic parallelization of conventional programs has been well-studied and some work has been done in manual parallelization of queries. These are discussed in sections 2.1 and 2.2, respectively.

2.1. Conventional Program Parallelization

For automatic parallelism detection, data-dependence tests are used to find concurrency and dependency. If a datum flows from one operation S_1 to another operation S_2, we say that S_2 depends on S_1 and S_1 must be executed before S_2. Any operations which do not have such dependencies can be executed concurrently (in parallel).

However, not all detected parallelism is useful for a given target machine. This is due to the overhead of some parallelism (e.g., communication and synchronization cost) being greater than the reduction in execution time achieved by parallel execution of that construct. By repackaging the fine-grained parallelism into larger serial ''chunks,'' the overhead can often be reduced so that some speedup can be gained through parallel execution of the chunks.

Many conventional-language parallelization techniques are known [PaW86] and can be applied. However, for the purpose of this discussion, it is useful to center on just ''loop concurrentization,'' since this technique applies to nearly all database operation algorithms. Loop concurrentization takes a parallelizable loop and partitions the iteration space to create a set of parallel-executable loops, each of which performs a subset of the original loop's iterations.

Aside from detecting that the loop is parallelizable, the key concerns in loop concurrentization are achieving good load balancing and minimizing synchronization overhead (synchronization introduced for load balancing).

For example, if there are N independent iterations to be executed on a parallel machine which has p processors, the compiler can *preschedule* the iterations of the loop onto the p

processors either in contiguous blocks of size *N/p* or by assigning every *p*th iteration to the same processor.

Alternatively, the processors can be self-scheduled [TaY86], meaning that each processor is given an initial set of iterations to perform and that as each processor completes its set it requests more iterations to process. In the simplest form, one can imagine assigning individual iterations and achieving excellent load balancing; however, the overhead in synchronizing to insure that the same iteration isn't given to two processors would probably negate the benefits of the load balancing. This simple self-scheduling works well only if the work for each iteration is relatively large, but may vary between different iterations, perhaps due to conditional statements within the loop body.

To reduce this synchronization overhead, guided self-scheduling (GSS) is proposed [PoK87]. Guided self-scheduling dynamically varies the size of the iteration set given when a processor request more work; larger sets are given first to minimize overhead and smaller sets are given later to achieve good load balancing. Another nice property of GSS is that all processors which will be used to execute a task *T* need not begin executing *T* at the same time, yet good load balance will be achieved.

2.2. Query Parallelization

Most of the research in parallel execution of database queries has focused on parallel execution *within* a single query or relational algebra operation. One popular scheme for parallelizing individual queries involves constructing a ''query tree'' representing how data must flow between relational algebra operations. Each node in the tree is allocated a set of processors.

Much work has been done to parallelize individual relational algebra operations. Since the `join` operation is the most costly commonly-used database operation, it has received the most attention. Nested-loop, sort-merge, and hash-based multiprocessor `join` algorithms are considered [DeG85,QaI88,ScD89]. Among these algorithms, hash-based `join` algorithms perform best over a wide range of parallel computers, mostly because very little (if any) synchronization is required.

Parallel execution of a sequence of queries has not been considered in depth. The closest work appears to be the XPRS database system [Sto88], which assumes a multi-user environment.

There are two kinds of parallelism that can be explored: inter-query parallelism and intra-query parallelism. They choose the following approach:

[1] Inter-query parallelism is user-specified by the `parallel` construct, which indicates parallel execution of sets of queries.

[2] A collection of good sequential plans is automatically constructed taking into account the limited size of main memory.

[3] Each plan is parallelized.

[4] At execution time, choose the plan which best matches the memory space available.

It appears that no previous work has attempted to *automatically detect concurrency* in a series of queries.

3. Terminology and Symbols

Throughout the examples in this document, queries are expressed in the relational-calculus language QUEL [Sto76]. Table 1 gives the definitions of symbols and notations used to represent costs or other significant properties of query execution.

I/O	time to read a block from disk into main memory (assumed to be equivalent to the time to write a block from main memory to disk)
$n \mid R \equiv \{R_1, R_2, \ldots, R_n\}$	number of hash partition elements (R_i) for relation R
$\lvert R \rvert$	number of pages (blocks) in relation R
$\lvert M \rvert$	number of pages (blocks) that can be stored in main memory

Table 1: Definition of Symbols and Notations

4. Parallelism at Different Levels

In this section, three levels at which concurrency of database queries can be detected (i.e., command-level, algorithm-level, and low-level), are introduced. Also it will be shown that what kind of parallelization can be achieved at different levels.

4.1. Command-Level

Not only *explicit* parallelism can be expressed at this level, but also some *inter-* and *intra-*query parallelism can be found automatically. Consider 2 relations:

```
EMP(name, salary, department)
SALES(department, item)
```

and two queries Q1 and Q2:

```
Q1: retrieve EMP
    where SALES.item = 'radio'
        and EMP.salary > 20000
        and EMP.department = SALES.department
Q2: retrieve EMP
    where SALES.item = 'toy'
        and EMP.salary < 15000
        and EMP.department = SALES.department
```

Since neither Q1 nor Q2 modify database, two queries can be executed in parallel. However, if the ''selection before join'' heuristic is employed, no intra-query parallelism can be found at this level.

4.2. Algorithm-Level

Once an algorithm for each database operation has been chosen (e.g., hash-based join algorithm is chosen for command-level *join* operation), queries can be represented in algorithm-level intermediate form. Throughout this paper, functional notation is used as algorithm-level intermediate form — `operation(relations, condition)`, where `relations` are either created or modified relation names if the `condition` is satisfied.

The first query Q1 can be represented as following (using the "selection before join" heuristic):

```
S1: select(Temp1, SALES.item = 'radio')
S2: select(Temp2, EMP.salary > 20000 )
S3: hash([$T1_1$, ... , $T1_n$], $h_1$(Temp1.department))
S4: hash([$T2_1$, ... , $T2_n$], $h_1$(Temp2.department))
S5[1..n]: DOALL i = 1 to n
            join(Result1, $T1_i$.department = $T2_i$.department)
```

Substantial parallelism is revealed at this level. The two select operations S1 and S2 can be executed in parallel, two hash functions can be applied in parallel and then the join[1] operations can be executed in parallel[2]. However, S1, S3, and S5[1..n] should be executed in sequence because S5[1..n] uses $T1_i$, which is defined in S3, and S3 uses Temp1 which is defined in S1. For the same reason, S2, S4, and S5[1..n] should be executed in serial. Well known dependency analysis [PaW86] will reveal these relationships; the resulting task graph is shown in figure 1.

[1] Further level can be defined depends on the algorithm of join operation at algorithm-level. But this is sufficient to demonstrate our idea. Furthermore we assume that the join operation at this level employs non-split hash-based join algorithm [Nak88] or nested-loop join algorithm.

[2] Output dependencies among join operations can be ignored because the order of adding tuples to Result1 is not important — does not cause any anomalies or inefficiencies.

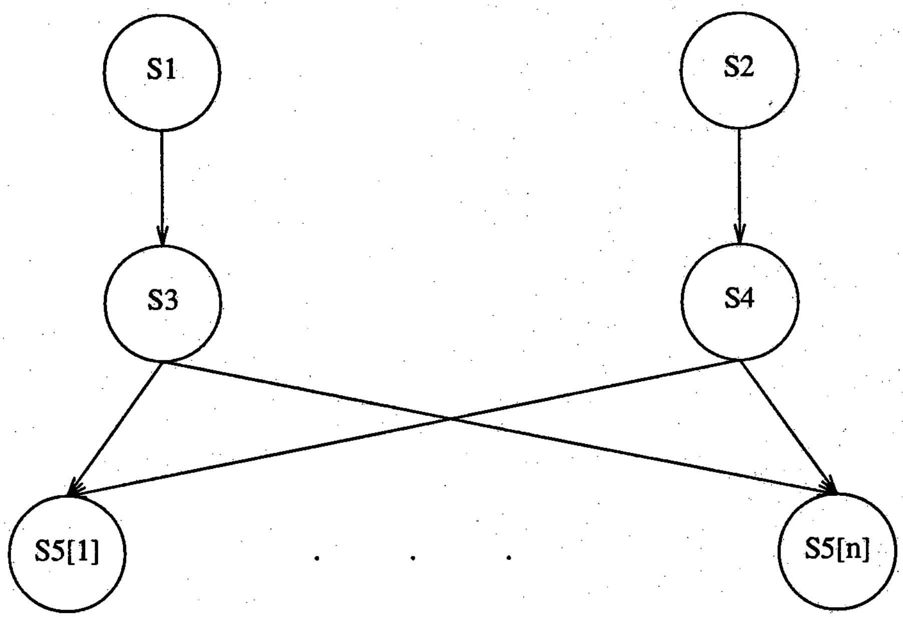

Figure 1: Task Graph of Query Q1

Even though the algorithm-level representation reveals much more parallelism than the command-level representation, not all of this parallelism can be used to reduce the total execution time. For example, consider two joins: join(Result1, $T1_1$.department = $T2_1$.department) and join(Result1, $T1_2$.department = $T2_2$.department). Since these two joins operate on different data, they can be run in parallel. Further, they neither synchronize nor communicate, hence parallel execution seems entirely beneficial. However, this is not necessarily so.

Assume that $|M| = 10$, $|T1_1| = |T1_2| = 8$, $|T2_1| = |T2_2| = 15$, and that the data are roughly uniformly distributed across all disks. If these joins are run in parallel using the non-split hash-based join algorithm [NaK88], the approximate I/O cost is $|T1_1| + 2 * |T2_1| + |T1_2| + 2 * |T2_2| = 76$. If the two joins are run in sequence, the approximate I/O cost is $|T1_1| + |T2_1| + |T1_2| + |T2_2| = 46$. Hence, given that the computation is I/O limited, the sequential form would execute faster than the parallelized version. Therefore, it is very important to decide which potentially parallel tasks should be executed in parallel and which should be executed in sequence. This topic will be discussed in section 6.3.

4.3. Low-Level

Algorithm-level representations of database queries can be translated into a lower-level corresponding to conventional language programs. Each select, hash, or join operation will be translated into a parallelizable loop. For example, the select operation S1 can be translated into:

```
for each tuple x in SALES {
    if ( x.item == 'radio' ) {
        add(Temp1,  x)
    }
}
```

Note that the parallelism-width of this example is directly related to the number of tuples in the relation SALES — which might not be known at compile time.

Parallelization and scheduling of the low-level representation closely resemble parallelization and scheduling of conventional programs and much work on this topic appears in the literature [PaW86, PoK87]. Hence, these techniques are not a major focus of this paper.

5. Overview of Automatic Parallelizer

Even though much work has been done in the area of multiple-query optimization, it has centered on high-level (i.e., command level) optimization, leaving low-level optimization for the compiler. By considering both high-level and low-level optimization as a single, coordinated, process, some additional optimizations[3] are made possible.

Aside from optimization, queries can be made to execute faster using parallelism. Despite recent advances in automatic parallelization of conventional language programs, very little work has been done toward parallelizing query programs. This is partly due to the fact that conventional language parallelization techniques need to be modified to operate on queries, and partly because other techniques must be developed to manage the relatively dynamic properties of queries (e.g., memory requirements based on relation size).

[3] The details of query optimization are beyond the scope of this paper. A good overview appears in [KaD89].

The unified query optimizer/parallelizer structure we propose is shown in figure 2.

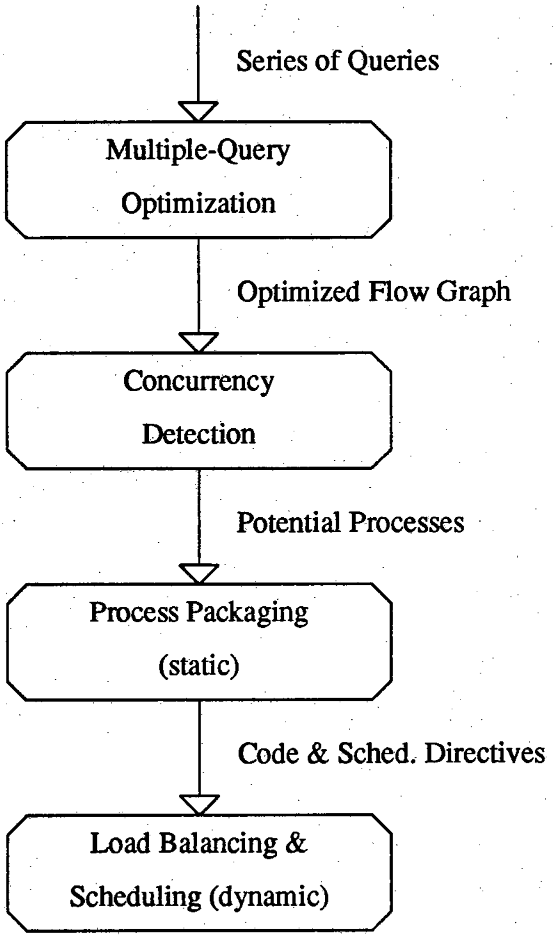

Figure 2: Structure of Query Optimizer/Parallelizer

In this section, we describe the target machine model and outline our approach to automatic parallelization of database programs.

5.1. Target Machine Model

The target machine described here serves primarily to simplify our discussion — it is not our intention to exclude other architectures, but to present an overview of our approach for a relatively straightforward machine.

Our target architecture will be a tightly-coupled, general-purpose, shared memory, multiprocessor (see figure 3). The machine is tightly-coupled in that, although each processor may operate independent of all others, it is also possible for multiple processors to synchronize quickly so that multiple processors can easily cooperate on a single task. All p processors of the machine are assumed to be identical and general-purpose in the sense that they do not have any special-purpose database hardware (e.g., sorter, bit filter). Main memory is assumed to be physically distributed across m memory modules, but access to all memory is shared by all processors — processors simply access memory addresses and hardware services these requests using some type of interconnection network.

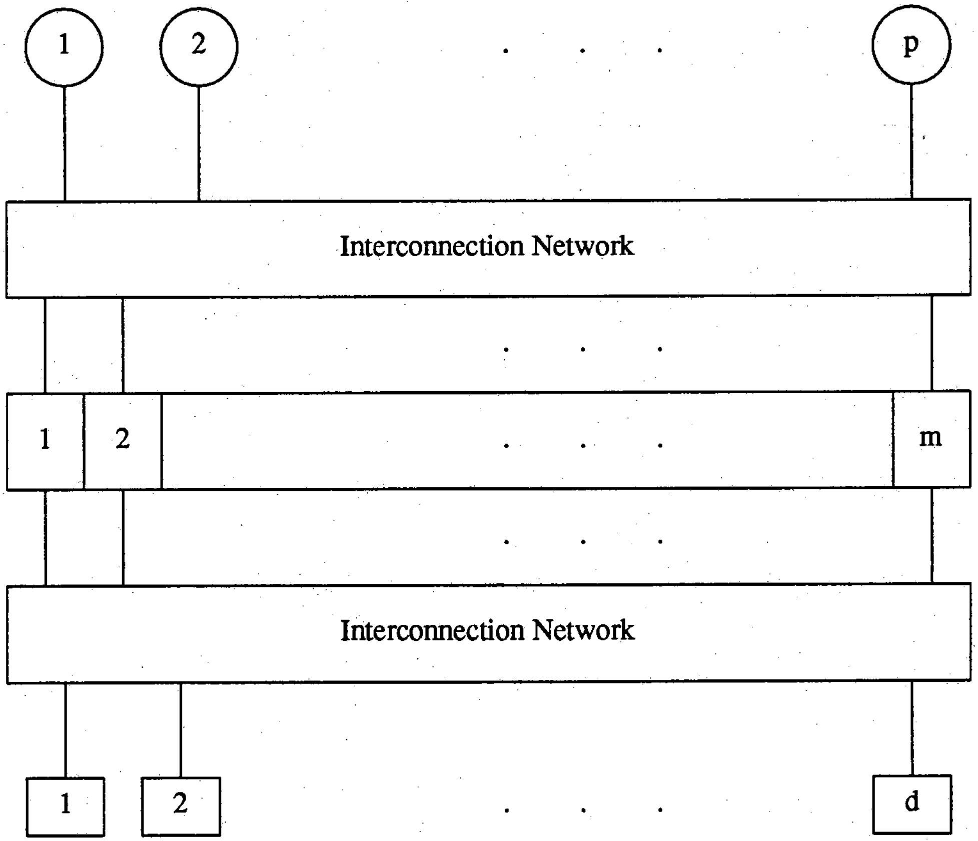

Figure 3: Target Architecture

It is further assumed that the target machine has the capability of accessing a very large amount of disk storage spread across d disk drives. Each of these disk drives is assumed to be capable of performing a disk-to-memory or memory-to-disk operation independent of the actions of the other disk drives. Relations are assumed to be horizontally partitioned across all disk drives.

5.2. Multiple-Query Parallelization

Our goal is to minimize the total execution time of a given sequence of queries by optimization and parallelization. This is not equivalent to maximizing parallelism; parallelism is used *iff* the total execution time will be reduced through its use. In this sense, parallelization is really nothing more than optimization for a parallel target machine.

After multiple-query optimization is performed, dependency analysis will be applied to the optimized intermediate form of the query program. The resulting task graph directly provides both dependence and inherent concurrency information. However, the concurrency is typically at too fine a level of granularity — executing each potentially parallel operation as a separate process would result in excessive parallelism overhead (e.g., process creation, process termination, synchronization, and communication). Hence, the next step is to re-package the inherent parallelism so as to achieve the most efficient parallel structure possible, as suggested in [Die87]. However, unlike a conventional program, a program which represents a series of database queries embodies widely varying granularity and a relatively strong dependence on runtime information. Hence, new process-packaging and scheduling techniques are needed.

A scheduler will determine the order of execution and reassigning processes to balance load. Even though load balancing will be achieved by the static scheduler, it may be too coarse because the granularity of some processes may be large. Finer load balancing will be achieved at runtime using dynamic scheduling.

6. Process Packaging And Scheduling

Clearly, more parallelism is evident at the lower-level than at algorithm-level or command-level. Much work has been done in the area of low-level parallelization and scheduling in the context of conventional program parallelization. Because most database algorithms result in low-level code strongly resembling the parallelizable `DO` loops found in Fortran programs, work such as that discussed in [PaW86, PoK87] is particularly relevant. However, unlike Fortran loops, lower-level code structures representing database operations are derived from higher-level forms, hence some information may be lost and some analysis made more complex by ignoring the higher-level forms.

For this reason, we wish to consider higher-level forms. However, all the objects which existed at the command-level still exist at the algorithm-level (unless they were redundant or otherwise unnecessary) — no relevant information is lost in using algorithm-level instead of command-level for our analysis. Hence, the discussion in this section centers on the analysis of algorithm-level constructs.

6.1. Estimation of I/O Costs

Concurrency detection determines which operations *could* be executed in parallel; automatic parallelization, however, must decide which operations *should* be executed in parallel. The problem is to be able to detect when parallelism will result in a speedup and to select the form with the greatest speedup. In automatically parallelizing conventional programs, this is primarily a matter of estimating synchronization and communication. However, database systems typically operate on data structures which do not fit within main memory, hence it is important to estimate I/O costs and memory requirements.

We propose to use a cost function which accurately represents the costs derived using a ''smart'' page management technique.

One such technique is the ''query locality set model'' [ChD85], in which it was observed that for each type of database operation, one particular page management scheme (e.g., LRU, MRU, etc.) exhibited consistently better performance than the others, hence performance could be improved by altering page management technique depending on the database operation being performed. Each relation is given a local buffer pool to hold its locality set, which is the set of the buffered pages associated with the relation. The size of locality set is determined before the database operation is executed, and needs not be recalculated while the execution of the database operation.

More recently, [ChD88, Chi89] presented a compiler-driven technique for deriving the optimal set of register/cache management operations given the references extracted from an arbitrary program; this work can be directly applied to generate code implementing optimal page management. Unfortunately, the compile-time analysis is more complex than that for the locality set model.

Since [Chi89] insures optimality whereas [ChD85] does not, we prefer to think in terms of implementing Chi's technique, however, either technique generates a valid cost function which can be applied toward improving the quality of the parallelization — which is the primary goal of this paper.

For example, consider a non-split hash-based join on two relations, R1 (|R1| = 4) and R2 (|R2| = 8). Assume that the join is implemented as the usual pair of nested loops such that the elements of R1 are enumerated within the inner loop. Further, assume that the join result is simply displayed without taking memory space or causing I/O additional operations.

Intuitively, it is clear that the maximum number of input operations would be |R1|*|R2|, which is 32. However, if optimal control of paging is used, the actual number can be substantially lower. For example, if at least 5 page frames fit in memory, then only 12 page reads are required. If fewer than 5, but at least 3, page frames fit in memory, then only 20 reads (i.e., memory requirement is 3 pages) are required. If only 2 pages frames fit, then 32 reads are required.

6.2. Process Packaging

After detecting the parallelism inherent in a sequence of queries by applying dependency analysis (i.e., after the initial task graph, as discussed in section 4.2, is determined), the scheduler must package the tasks so that execution time will be minimized. This consists of determining which potential tasks should be merged (packaged) to become a larger task. Potentially parallel tasks are merged when the parallel structure would have executed slower than the serial structure — when extra I/O due to sharing of main memory among processes, synchronization, and communication costs are greater than the time savings by parallel execution. Serial tasks are merged if the merged task achieves better load balancing and better utilization of main memory.

The conditions permitting initial tasks to be merged are:

[1] Each task should be translatable to a parallelizable loop (e.g., `select`, `hash`, and non-split hash-based or nested-loop `join`), and

[2] the memory requirement of the combined tasks does not exceed the size of the main memory

and additionally either:

[3] tasks which satisfy condition [1] and [2] and operate on the same relation (i.e., correspond to low-level loop fusion), or

[4] tasks, say S1 and S2, which satisfy conditions [1] and [2] and S1 depends on S2 and S2 operates on the subset of the relation on which S1 operates (i.e., correspond to software pipelining)

For example, consider the two queries `Q1` and `Q2` from section 4.1. The algorithm-level representation of `Q1` and `Q2` is:

```
S1: select(Temp1, SALES.item = 'radio')
S2: select(Temp2, EMP.salary > 20000 )
S3: hash([T1_1, ... , T1_n], h_1(Temp1.department))
S4: hash([T2_1, ... , T2_n], h_1(Temp2.department))
S5[1..n]: DOALL i = 1 to n
                join(Result1, T1_i.department = T2_i.department)
S6: select(Temp3, SALES.item = 'toy')
S7: select(Temp4, EMP.salary < 15000 )
S8: hash([T3_1, ... , T3_m], h_2(Temp3.department))
S9: hash([T4_1, ... , T4_m], h_2(Temp4.department))
S10[1..m]: DOALL i = 1 to m
                join(Result2, T3_i.department = T4_i.department)
```

Assuming that $|M| > (n + m)$ pages, `select` requires 2 pages of memory (i.e., one for input and the other for output) and `hash` requires $(n + 1)$ pages of memory. Since `S1` and `S6` operate on the same relation `SALES`, these `select` operations can be translated into a single parallelizable loop (by loop merging/fusion). Because the memory requirement for this parallelizable loop (i.e., 3 pages of memory) does not exceed $|M|$, it is profitable to combine `S1` and `S6` in this way.

In other words, `S1` and `S6` satisfy conditions [1], [2], and [3] given above. The same is true of `S2` and `S7`. Tasks `S1` and `S3` satisfy conditions [1], [2], and [4], hence, these tasks can also be combined, as can tasks `S6` and `S8`. The modified task graph is shown in figure 4.

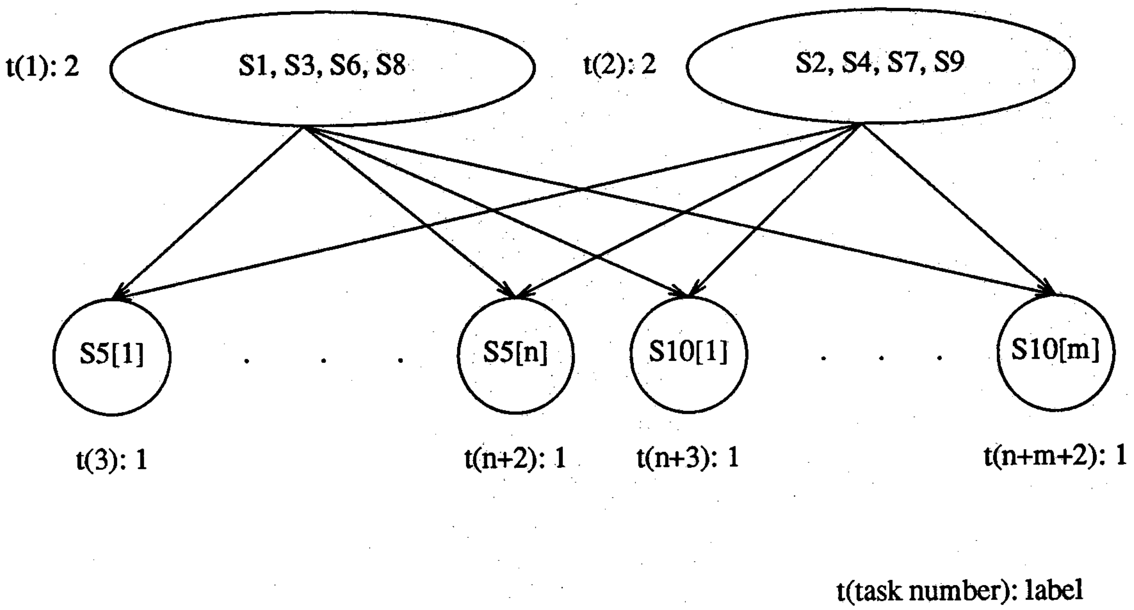

Figure 4: Task Graph of Queries Q1 and Q2

Performing these task merges has the beneficial effect of:

- reducing the start-up cost of each parallelizable loop,
- generating larger-grain processes, which provide better opportunities for use of fine-grain scheduling to hide pipeline delays, and
- simplifying algorithm-level scheduling.

A simple method for translating merged tasks into a lower-level form is presented in the appendix.

6.3. Scheduling

There are two typical task scheduling approaches: static scheduling and dynamic scheduling.

Static scheduling is performed before the execution of any task and makes use of *global* information about tasks. Most of the static scheduling schemes developed for conventional programs are complex, time consuming, and require some estimation of the execution time of tasks. These time estimates often depend critically on relation size, which, unfortunately, may vary widely at runtime. Further, database programs or multiple queries are not expected to be executed as many times as conventional programs might be executed, hence, long static analysis times might be unacceptable.

Dynamic scheduling is typically based on *local* information. Scheduling decisions are made at runtime, hence, time spent scheduling can be a significant overhead; further, the lack of global task information inherently limits the quality of the schedules.

For these reasons, database code is typically scheduled using very simple dynamic policies — which are easy to implement, but perform poorly. Instead, we propose to use a scheduling method which combines both static and dynamic scheduling techniques. This method is outlined in the hybrid scheduler algorithm (below). It begins with the static scheduling scheme proposed in [ChL74], but adds dynamic scheduling implemented by steps [3], [4], and [5]:

Algorithm: Hybrid Scheduler

[1] Produce the task precedence graph as described in sections 4.2 and 6.2.

[2] Label the task according to the following rules:

  [2a] The label of a task which has no successor is 1 (e.g., tasks t(3) through t(n+m+2) in figure 4).

  [2b] The label of a task that has more than one successor is equal to one plus the maximal label value of the successor of the task (e.g., task t(1) and t(2) in figure 4 have label 2).

[3] Classify each task as either (completely) parallelizable or non-parallelizable tasks (e.g. selection, hash, and non-split hash-based join algorithm are parallelizable but `append` a tuple to a relation or perform an associative reduction are classified as non-parallelizable tasks).

[4] At runtime (dynamically), use "level-order" priority scheduling among the *ready* tasks (i.e., a task is ready if all its predecessors have been executed). Highest priority is given to the non-parallelizable tasks which has the highest label (level), then to parallelizable tasks with the highest label, and so forth.

[5] By delaying the execution of ready parallelizable tasks, bottlenecks which may be created by the non-parallelizable tasks should be avoided in step [4], but load imbalances may result. Load-balancing is achieved by dynamic low-level scheduling *within* parallelizable tasks using guided self-scheduling [PoK87].

The hybrid scheduling algorithm uses static scheduling, at the very large grain level, in order to minimize the dynamic scheduling overhead. However, there is a second effect: due to steps [3] and [4], the tasks which are scheduled last generally will be tasks with large task parallelism widths. At first, this seems counterproductive, because these massively-parallel tasks will initially obtain only a small fraction of the machine... hence, some parallelism within these tasks is discarded. The paradox is that this is a *desirable* effect which meshes perfectly with the guided self-scheduling of step [5], which, as described in section 2.1, assumes that a large number of very fine-grain potentially-parallel processes are available and schedules them in groups of decreasing size.

The only potential problem with using guided self-scheduling in this way is that, in some cases, the assumption that a large-number of very fine-grain potentially-parallel processes are available may be invalid. For example, a `select` would normally be expected to contain such parallelism, but not if the relation it selects from only has 2 tuples. It is also possible that the assumption would be wrong because the system does not have enough memory to support parallel execution of a particular task while other tasks are executing. We attempt to avoid these conditions by runtime task clustering.

As demonstrated in section 4.2, database operation can produce many *compatible* parallelizable tasks (i.e., tasks are compatible if tasks can be executed in parallel) which have the same predecessors and the same successors (e.g., join of the same indexed buckets such as t(3) through t(n+m+2) in figure 4). When such tasks are scheduled by the scheduler described above, no new *ready* task will be generated until all of them are completed. Therefore, those tasks can be artificially combined and the clustering scheduler can schedule them based upon runtime information.

The following is the runtime task clustering algorithm:

Algorithm: Task Clustering

[1] Form a list of compatible tasks, sorted in order of increasing task memory requirement.

[2] If the list has fewer than 2 elements, ignore the rest of this algorithm — there is nothing to cluster (combine).

[3] Take out the last task and form task *T* (i.e., *T* = { last task })

[4] Try to schedule in parallel with the first task in the list with *T* — compare the memory requirements of potentially parallel tasks with the size of the available main memory. If task *T* and the first task in the list cannot be scheduled in parallel, the task *T* is scheduled by

itself and go to step [2]. Otherwise, proceed with step [5].

[5] In sequence, search for a task in the list such that this *i*th task cannot be scheduled in parallel with task *T*. Schedule the *(i-i)*th task with *T* in parallel and go to step [4].

Hence, this clustering algorithm eliminates the useless overhead imposed by guided self-scheduling technique in the aforementioned worst-case scenarios.

7. Summary

In this paper, we have presented an argument in favor of automatic parallelization for database programs (multiple queries) as well as the basic approach to constructing such a parallelizer.

In constructing a multiple-query parallelizer, conventional (e.g., Fortran code — what we refer to as "low-level" representations) vectorization and parallelization techniques can and should be applied, but these techniques alone are unlikely to produce good results. The primary differences between conventional and multiple-query parallelization derive from the fact that, unlike conventional programming languages, database queries have properties which are very dependent on dynamic (runtime) information. Further, queries differ in that higher-level abstractions of the database operations are readily available, hence some information is lost when only lower-level forms are considered.

In our approach, we have explicitly dealt with the dynamic character of database operations by suggesting a variety of mechanisms to efficiently integrate static (compile-time) and dynamic (runtime) control of parallel process structure.

Further, rather than ignoring the existence of higher-level forms, we have defined a new intermediate level — algorithm level — to be used to bridge the gap between the high-level information available in the original query representation and the increased parallelism exposed by the lower-level form.

Hence, the work presented in this paper forms a foundation upon which a complete multiple-query parallelizer will be built. Ongoing research involves the details of the algorithm-level analysis and parallelization, integration of conventional parallelization techniques, and finally the creation and performance evaluation of a prototype system.

References

[ChD85]

Chou, H. T., and Dewitt, D. J., An evaluation of buffer management strategies for relational database systems. Proceedings of the Conference on Very Large Data Bases (1985)

[ChD88]

Chi, C-H., and Dietz, H. G., Register Allocation for GaAs Computer Systems. Proceedings of the 1988 Hawaii International Conference on Systems Sciences, January 1988.

[Chi89]

Chi, C-H., Compiler-Driven Cache Management Using A State-Level Transition Model. PhD Dissertation, School of Electrical Engineering, Purdue University, May 1989.

[ChL74]

Chen, N. F., and Liu, C. L. On a class of scheduling algorithms for multiprocessor computing systems. Proceedings of Sagamore Computer Conference on Parallel Processing (1974)

[DeG85]

Dewitt, D., and Gerber, R. Multiprocessor hash based join algorithms. Proceedings of the Conference on Very Large Data Bases (1985)

[Die87]

Dietz, H. G. The refined-language approach to compiling for parallel supercomputers. Ph.D. dissertation, Dept. of Comp. Sci., Polytechnic Univ. 1987

[KaD89]

Kang, M. H., and Dietz, H. G. Optimization of temporary relation use in multiple queries. Submitted for publication (1989).

[Nak88]

Nakayama, M., et al. Hash-partitioned join method using dynamic destaging strategy. Proceedings of the Conference on Very Large Data Bases (1988)

[PaW86]

Padua, D. A., and Wolfe, M. J. Advanced compiler optimizations for supercomputers. Communications of the ACM, 29, 12 (1986)

[PoK87]

Polychronopoulos, C. D., and Kuck, D. J. Guided self-scheduling: A practical scheduling scheme for parallel supercomputer. IEEE Transactions on computers, C-36, 12 (1987)

[QaI88]

Qadah, G. Z., and Irani, K. B. The join algorithms on a shared-memory multiprocessor database machine. IEEE Transaction on Software Engineering, 14, 11 (1988)

[ScD89]

Schneider, D. A., and Dewitt, D. J. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. Proceedings of the ACM-SIGMOD International Conference on Management of Data, (1989)

[Sto76]

Stonebraker, M., et al. The design and implementation of INGRES. ACM Transactions on Database Systems, 1, 3 (1976)

[Sto88]

Stonebraker, M., et al. The design of XPRS. Proceedings of the Conference on Very Large Data Bases (1988)

[TaY86]

Tang, P., and Yew, P. Processor self-scheduling for multiple-nested parallel loops. Proceedings of the International Conference on Parallel Processing, (1986)

Appendix: Translation of Merged Tasks into Low-Level Form

In section 6.2, the technique of merging tasks is discussed, but no example is given of the resulting code structures. In this appendix, we outline the translation of merged tasks into a conventional low-level representation. This translation is guided by the following rules:

[1] If two tasks, S1 and S2, are merged and S1 and S2 are compatible (i.e., can be executed in parallel), but the conditions of S1 and S2 are mutually exclusive, then put S1 and S2 within an `if ... else` construct.

[2] If two tasks, S1 and S2, are merged and S2 depends on S1 or the condition of S2 is a subset of the condition of S1 then put S2 inside an `if` construct.

[3] If two merged tasks, S1 and S2, do not meet either condition [1] or [2], then put S1 and S2 in sequence.

For example, the low-level representation of the merged task `[S1, S3, S6, S8]` from the example in section 6.2 is:

```
for each tuple x in SALES {
    if ( x.item == 'radio' ) {
        i = $h_1$(x.department)
        add($T1_i$, x)
    }
    else if ( x.item == 'toy' ) {
        i = $h_2$(x.department)
        add($T2_i$, x)
    }
}
```