Parallel Query Processing in Databases on Multicore Architectures

Ralph Acker[1], Christian Roth[1], and Rudolf Bayer[2]

[1] Transaction Software, Willy-Brandt-Allee 2, D-81829 München, Germany
{Ralph.Acker, Christian.Roth}@transaction.de
[2] Institut für Informatik, TU-München, Boltzmannstr. 3, D-85747 Garching, Germany
rdlf.bayer@informatik.tu-muenchen.de
http://www.transaction.de

Abstract. In this paper we present a novel and complete approach on how to encapsulate parallelism for relational database query execution that strives for maximum resource utilization for both CPU and disk activities. Its simple and robust design is capable of modeling intra- and inter-operator parallelism for one or more parallel queries in a most natural way. In addition, encapsulation guarantees that the bulk of relational operators can remain unmodified, as long as their implementation is thread-safe. We will show, that with this approach, the problem of scheduling parallel tasks is generalized, so that it can be safely entrusted to the underlying operating system (OS) without suffering any performance penalties. On the contrary, relocation of all scheduling decisions from the DBMS to the OS guarantees a centralized and therefore near-optimal resource allocation (depending on the OS's abilities) for the complete system that is hosting the database server as one of its tasks. Moreover, with this proposal, query parallelization is fully transparent on the SQL interface of the database system. Configuration of the system for effective parallel query execution can be adjusted by the DB administrator by setting two descriptive tuning parameters. A prototype implementation has been integrated into the Transbase® relational DBMS engine.

Keywords: relational dbms, parallel query processing, encapsulation, intra-operator, inter-operator, scheduling, optimization.

1 Introduction and Related Work

Computer architecture is currently shifting, making concepts formerly restricted to supercomputers available on inexpensive server systems, desktop and laptop computers. Hardware-parallelism, in form of multicore computing and RAID-controlled access to secondary storage has apparently become the most promising cure for stagnation in the constant longing for more computing power.

Based on this trend, it has become tempting to revisit the concepts of database parallelism in the light of those emerging hardware architectures, and of modern operating system characteristics that support this hardware.

A. Bourgeois and S.Q. Zheng (Eds.): ICA3PP 2008, LNCS 5022, pp. 2–13, 2008.

Over the last two decades parallel query processing in database systems was the topic of considerable research. Its outcome is now undoubtedly in daily use as part of major commercial DBMSs. Most of the work was concentrated on shared nothing (SN) architectures, e.g. the research prototypes Gamma [1] and Bubba [2]. Now it is applied in modern grid and cluster computing. Other approaches focus on symmetric multiprocessing architectures (SMP), such as XPRS [3] and Volcano [4], [5]. Extensive additional efforts on scheduling parallel tasks have been made, e.g. [6], [7]. The most recent work published in this field focuses on the special requirements of simultaneous multithreading (SMT), e.g. [8], and especially on the well-known problem of stalls in the memory hierarchy [9].

However, the fundamental concepts of task identification and resource scheduling were not revised recently to honor emerging technologies in modern SMP systems. Our approach adopts the evident idea of encapsulating asynchronous relational query execution as an opaque relational operator. All implementation details are hidden within this operator while its usage poses minimal requirements to other relational operators, allowing them to remain unaffected. This idea of an asynchronous relational operator was originally proposed in [10] and [4], but it appears that it never reached maturity. Our parallel operator differs from the Volcano *exchange operator*, as it inherently supports intra-operator parallelism and also addresses the problem of order preservation. Tandem's *parallel operator* is a commercial solution and no details were published, but according to [4] it seems to be very similar to Volcano's exchange operator.

We combine the encapsulation of parallelism with a two-phase query plan optimization. The first phase is common static optimization by the DMBS optimizer. The optimizer drafts a plan on how a query should be carried out in parallel. The additional complexity of parallelization adds to the complexity of the NP-hard problem of query plan optimization. We completely evade this complexity by using an approach to query plan parallelization that is based on a minimal set of boundary conditions. The second phase of optimization dynamically refines the query plan during the query execution phase, rebalancing the threads of execution into an equilibrium that guarantees maximum resource utilization. This in itself is also a common concept for maximizing parallelism on restricted resources. This form of two-phase optimization was first applied to the problem of parallelism in database systems by [11]. The original concept identifies (generates) tasks, and schedules them such that resources (I/O and CPU) are optimally utilized. Therefore the tasks are categorized based on criteria such as I/O or CPU boundedness, tuple size, tuples per page and estimated execution time. These estimates can be inaccurate or vary over time, e.g. due to data skew. So resource utilization is constantly monitored and whenever it is not optimal rescheduling is used. Our work diverges strongly from this approach in the aspect that we understand parallelization as a concurrency problem. In our approach all tasks are strictly data-driven, i.e. they are 'runnable' at any time, provided that input data is available. They all apply for limited resources at the same time, while they are collaboratively calculating the result of a relational query. There is no requirement to actively interfere with the scheduling or prioritization of tasks, or to act on assumptions on these tasks. Slow tasks are accelerated by assigning additional threads. Fast tasks will wait on empty input buffers, consuming no resources at all. So permanent, automatic and data-driven rebalancing of resources takes place, always striving to achieve optimal resource allocation through concurrency.

To the best of our knowledge, this is the only work that covers both aspects of general parallelization of relational queries, i.e. query plan optimization and load balancing. It refers to the only implementation ready for integration into a relational database system. Therefore this discussion of an asynchronous relational operator is unique and complete.

2 Encapsulation of Parallelism

We model parallelization as a new operator (ASYNC) in the query execution plan (QEP). Without loss of generality, we presume query plans consisting of operators based on the well-known iterator model, i.e. each operator in the operator tree exposes an open(), next(), and close() method to other operators, while all implementation details are hidden within the operator. The ASYNC operator is an abstraction of thread boundaries in the operator tree, i.e. data moving through an ASYNC node is passed from one thread to another. An ASYNC node may be placed between any pair of operators in a sequential QEP, i.e. on any edges of the operator tree, as long as it has no side-effect compromising the functional integrity of the QEP. As an example for such a side effect, imagine an ASYNC node allowing out-of-order execution below a relational grouping operation. The GROUP operator relies on an input order on the grouping field, which is arranged by a sequential QEP (SORT before GROUP in Figure 1a). But in a parallelized QEP the ASYNC operator might disrupt the order (Figure 1b).

Exchange of data (tuples) over thread boundaries is done via a buffer that is encapsulated and operated on by every ASYNC operator and thereby shared among the two adjacent threads. Access to this buffer is synchronized, resembling the classic consumer-producer problem. In order to minimize synchronization overhead, the buffer is not locked for every tuple insertion/retrieval, but it is divided equally into three partitions, which are used in a round-robin fashion. Consumer and producer need to be synchronized only when they acquire access to a new partition. We chose partitioning into three regions in order to allow one thread to switch partitions without necessarily having to wait for the other thread. Thereby the first thread has the opportunity to completely utilize its time slice rather than to give it up when its partition is exhausted. A higher number of partitions would further improve this behavior, but it would also induce additional synchronization operations for switching partitions. Experimental results have confirmed that partitioning into three regions is optimal for common database operations.

Calling the open() routine of the ASYNC operator assigns a new producer thread that evaluates the QEP sub-tree below this ASYNC node. The new thread propagates the open() call to its sons and afterwards *asynchronously* starts evaluating its sub-tree. It retrieves all input data by calling next() and copying results into its current buffer partition, until end-of-data is reached and the lock on the current buffer partition can be released. Finally a call to close() frees all resources and terminates the thread. The ASYNC nodes operate strictly in FIFO mode, so any ordering of the input data will be retained.

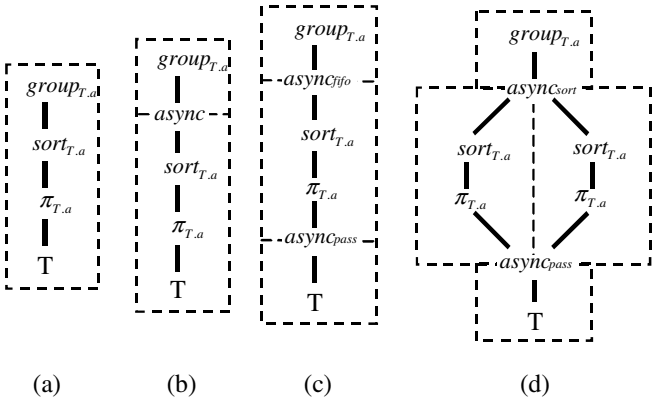

Fig. 1. Different parallelized QEPs for the query *SELECT T.a FROM T GROUP BY T.a*. Dotted lines represent thread boundaries. (a) Sequential plan (b) Example for ASYNC placement, (c) Inter-operator parallelism and order preservation, (d) Intra-operator parallelism and sorting.

Using an ASYNC operator yields three valuable advantages. First, it is entirely consistent with the recursive programming paradigm of the iterator model. Second, existing operators do not need any modification, provided that they are thread-safe. Third, ASYNC nodes can be placed (almost) freely in the QEP by the optimizer, allowing for arbitrary forms of parallelization. An algorithm for parallelization will be presented in Section 3.

Obviously, inter-operator parallelism can be easily modeled with this approach by simply inserting ASYNC nodes into any given operator tree. In order to reduce the costs for creating and terminating threads, the established concept of thread pooling is used. The system maintains a pool of worker threads. Threads from this pool can be assigned to all sorts of tasks. If the operation is completed, i.e. the ASYNC operator is closed, the thread will return into the pool where it can be reused for another task.

Intra-operator parallelism is modeled by identifying *pipeline fragments* in the operator tree that are suitable for parallel execution. These fragments are enclosed by two ASYNC nodes, i.e. execution is to be carried out by three threads (e.g. Figure 1c). Each of these fragments is logically replicated and its clone is executed in parallel as a data pipeline (Figure 1d). At each thread boundary a partitioned buffer is installed. The lower ASYNC node acts as data partitioner, i.e. it distributes its input data among the pipelines' input buffers, while the upper ASYNC merges the result from the different pipelines' output buffers. Here special attention has to be paid to data ordering. Partitioning and merging can be organized in two ways, reflected as two operating modes of the involved ASYNC nodes. In pass-through mode (PASS) the pipeline is adjusted for maximum throughput. Input data is written to any pipeline that currently accepts more input and the upper ASYNC passes processed data on as soon as it becomes available. So a pipeline in PASS mode is likely to process data in an out-of-order fashion.

In some cases however, it is preferable to retain a given input order of the data, e.g. if this order can be exploited by a consecutive operation. Therefore the pipeline may also operate in FIFO mode, at the expense of losing some throughput (Figure 1c). In

this mode, data may also be inserted into any pipeline. But, as a meta information, a partition is also marked with a sequential number that is also accessible for the upper end of the pipeline. The upper ASYNC will then arrange its output along this sequence, thereby preserving the original order. Throughput is lost compared to the PASS variant at the upper end of the pipeline, where the ASYNC potentially has to wait for data with the next sequence number to become available. This might cause the other pipelines to stall, because data is not retrieved fast enough. Another problem of FIFO pipelining is that the amount of data is typically not constant in a pipeline, as a pipeline might produce dramatically lesser (e.g. pipeline contains a restriction or projection) or greater (e.g. pipeline contains a JOIN operator) amounts of data than the input amount. For relational operators it is particularly hard to estimate how the data size will change. This presents a profound problem in buffer space allocation for FIFO pipelines. However, the problem is attenuated by roughly estimating changes in data size where this is possible, e.g. for a projection eliminating one column. And it is completely overcome by adding an additional flag of meta information into the pipeline buffers. This flag specifies whether all data from the source partition with the current sequential number was retrieved, i.e. whether more data from a source partition has to be processed from this particular pipeline, or if the upper ASYNC can move on to the next partition with the next sequence number. With this simple modification, one input partition of the pipeline may evolve to one, possibly empty, or more output partitions of the same pipeline.

Finally, parallelizing SORT operations (Figure 1d) in a pipeline is a particularly attractive feat. To achieve this, the lower end of the pipeline operates in PASS mode, as any input order becomes irrelevant at the upcoming SORT operation in the pipeline. The upper end operates in SORT mode, i.e. the ASYNC performs a heap-merge operation with all input pipeline tuples. Thereby a SORT operation sorting n tuples can be performed in m parallel pipelines with an estimated complexity of $n/m*log(n/m)$ for each pipeline, i.e. $n*log(n/m)$ for m pipelines. The final heap merge has a complexity of $n*log(m)$, so the final *linear* complexity of the pipelined SORT operation remains $n*log(n)$, just like for the sequential operation.

It is also important to emphasize, that the static parallelization, as well as dynamic load balancing discussed in the next sections, are not based on any assumptions or statistics such as data sizes or distributions, estimations on operators such as number of machine instructions per operation, number of I/Os or classification in CPU- or I/O-bound query plan fragments. Statistics regularly tend to be compromised by data skew and the constant activities of monitoring and refining their validity induces unwanted additional computational costs.

3 Optimization for Parallel Execution

As proposed in many other approaches, we also adopt a two phase optimization. The first phase performs static optimization and takes place during general query optimization. The DBMS SQL compiler and optimizer generate a QEP. This constitutes what the DBMS considers an optimal plan for sequential processing. Afterwards, this sequential plan is statically parallelized, i.e. the sequential execution plan is split up into tasks that can be executed in parallel.

In order to reduce the complexity of this famous NP-hard problem of query plan optimization, the query optimizer will generally consider only left-deep operator trees. It is well understood [12] that these plans are not always optimal, especially for parallel execution, where under certain circumstances (size of intermediate results, available memory) bushy join trees may perform better, because joins are performed in parallel. Considering all forms of operator trees is still an open research field and will not be addressed in this paper. Without loss of generality, we assume in the following, that the result of the optimizer for a sequential query evaluation is also near-optimal for parallel processing, while we emphasize the fact that our approach to encapsulation of parallelism applies to arbitrary operator trees without restrictions.

3.1 Static Parallelization

Two configuration parameters are of importance in static parallelization. The first is the total size of memory available per ASYNC operator (*async_max_buffer*). Every ASYNC operator requires a buffer for inter-thread communication. If this buffer is chosen too small, a lot of synchronization has to be performed when large amounts of data are passed through it. If the buffer is too large, memory might be wasted. The optimal size would allow a thread to process one partition of the buffer and release its locks on it. Given the versatile forms of relational operators and their numerous combinations, this goal is very difficult to achieve. However, a buffer partition should be at least big enough, that most threads cannot process it in a mere fraction of their time slices. Thus synchronization and scheduling overhead is limited reasonably.

The second parameter is the maximum number of parallel *pipeline fragments* that should be active at any time (*async_max_threads*). QEPs for complex statements tend also to be complex, involving several thousands of operators. If such a plan is evaluated sequentially, only one single operator is active and consuming CPU and/or I/O resources at any point in time, and typically only operators in the immediate vicinity are likely to demand massive memory allocation. Parallel query execution behaves contrarily. Here it is actually desired to have many fractions of the tree running in parallel, each fragment having one active operator consuming CPU and/or I/O and each with potentially heavy memory requirements. Obviously some precautions, like limiting the number of ASYNC operators, have to be taken in order to cope with this problem.

Yet there exists a special class of relational operators, the so-called *blocking* operators. These operators have to process all input data before any output can be generated, e.g. SORT/ AGGR. They represent a rupture in the data flow, i.e. the plan below and above such a blocking node will be executed mutually exclusively, even by a parallel QEP. By identifying these blocking operators, the parallelizer can apply maximum parallelism by inserting *async_max_threads* ASYNC nodes below and above such a blocking node.

With these two configuration parameters, we can devise an algorithm for static parallelization. Its task is to identify QEP fragments that can be split off for asynchronous execution. To limit memory consumption, synchronization and communication overhead, we choose to find QEP portions of maximum length, i.e. the number of initial ASYNCs is minimized. We can rely on dynamic load balancing, which is discussed below, to increase the number of ASYNCs for optimal parallelism.

Additionally an ASYNC is always placed at the root of the QEP (not shown in Figure 1) to ensure, that the server will work ahead on a bulky query result set, while the database client is processing the last portion of the result. More ASYNCs are always placed above any leaves of the QEP, i.e. data sources (relation or index accesses) that are likely to perform I/O, so asynchronous I/O is maximized. Pipelines (intra-operator parallelism) contain only unary operators. The subtrees of n-ary operators are split into several threads by inserting ASYNCs below this operator, so n input streams are calculated independently in n threads. The only exception to this rule is NL-JOIN (nested-loop) operator, because of its strong functional dependency in looking up join partners; it may reside in one pipeline as a whole.

The static parallelization is essentially a depth-first traversal of the initially sequential QEP. While moving down the QEP, we keep an account if the current operator (or a parent operator) relies on the current data order, so we can later apply the optimal mode to an ASYNC. When we reach a leaf of the QEP we insert the first ASYNC to encourage asynchronous I/O. Then we retrace our steps upwards over any unary operators. When we reach an n-ary operator (not NL-JOIN), we insert another ASYNC node and thereby build a pipeline. Finally for these two associated ASYNCs the operation mode is set to PASS/ FIFO or SORT (if the pipeline contains a sort operation). A pipeline may be executed *async_max_threads* times in parallel and each ASYNC may allocate as much as *async_max_buffer* memory for its buffer.

These are the basic steps of our algorithm to parallelize a sequential QEP. Obviously its complexity is determined by the complexity of the tree traversal O(n), and is therefore linear with the number of operators in the QEP.

3.2 Dynamic Load Balancing

The second phase of optimization is carried out during query run-time. While one, or possibly several, queries are executed in parallel, the system resources must be constantly reallocated to ensure optimal utilization of CPU, memory and disk resources. We distinguish three phases of query execution, each with its own special requirements to load balancing.

3.2.1 Phases of Query Execution

During the first phase, execution startup, all portions of operator trees (separated by ASYNC nodes) are initiated and start computation one by one. As calculation is data-driven, in this phase only the ‘leaf’-threads will run. All other threads are currently waiting for input. This fact exhibits two important problems. The leaves of the operator trees are the data sources and typically involve mostly I/O, while the inner operator nodes of a tree are more biased towards CPU utilization. This brings about a heavy load imbalance in the earliest phase of query execution.

In the second phase all parts of the operator tree are active. In this phase, if we temporarily assume uniform data distribution, the tasks will automatically be rebalanced for maximum data throughput and optimal resource utilization. Data skew will constantly disturb this balance, making permanent rebalancing necessary. In this phase however, it is most likely to achieve the target of full resource utilization.

In the third stage, the leaf nodes are already exhausted. Now, all input data is available and stored in various buffers across all involved operators, and no more I/O is

necessary, balance shifts again for a massive CPU shortness. Thus, the second phase has most potential to compensate for the incurred overhead of parallelization by maximized resource utilization. Moreover, if this phase is too short, parallelization is likely not to pay off. Particularly in the presence of simple and short-running queries we will show in our experimental results, that this fact alone can already make parallelization disadvantageous over sequential query execution without further precautions.

We therefore propose a robust approach of gradually increasing parallelism using *retarded pipeline activation* and *dynamic buffer size*. As in the first phase of query execution only the leaf threads are running, there is no point in activating all pipelines, as they would immediately block on their empty input buffers. In addition we artificially limit the memory in the input buffers on our ASYNC buffers. Immediately after startup only a small fraction of the buffer region is used. Therefore, a buffer partition fills up relatively fast and a producer is forced to switch to the next partition. The consumer is signaled that input data has become available and begins work much earlier, i.e. the startup phase is shortened dramatically at the expense of some additional synchronization on the input buffers. When the producer has to wait on a full input buffer (imbalanced consumer-producer relationship), it increases the fraction of its buffer that it may use next time. This is repeated until either a balance between producer and consumer is reached or the buffer is used completely. In the latter case, if the buffer in question is at the lower end of a pipeline, one additional pipeline is activated. In combination, these two simple techniques guarantee a very agile and short query startup. They make sure that asynchronous execution of short-running queries has only a negligible overhead compared to sequential query execution while optimal balance between consumers and producers is established quickly.

In conclusion, data-driven load balancing as proposed here, is fully adaptive to all forms of hardware configurations, i.e. it is not limited to any specific number of CPU cores, particular memory hierarchies or disk configurations. On the contrary, it will automatically scale to the system parameters it is confronted with. This is solely achieved by its ability of finding the equilibrium of optimal data-flow and thereby optimal resource allocation. This is done completely independent from the particular hardware situation.

3.2.2 Asynchronous I/O System

In our approach, I/Os are issued against the centralized asynchronous I/O system of the operating system. They are issued one at a time as the threads in the QEP move along. Unlike other approaches where I/O is classified into sequential and random I/O and the DBMS decides on I/O serialization, the decision which I/O request to serve first is postponed to the operating system's I/O system. This is where all information for optimal global scheduling (physical I/O system layout, current position of head(s), complete system wide I/O request queue, physical addresses of all I/Os in queue) is available. This is particularly true, when several processes are competing for disk resources. Our only requirement to the I/O system is fairness in scheduling, i.e. no starvation and active I/O reordering. If the operating system would simply serve I/O request in FIFO order, this would severely undermine all efforts for efficiency. These requests are satisfied by all modern operating systems. As for task scheduling, prioritization of I/O requests is not a requirement. However, the DMBS could exploit a

second, lower priority. By constantly interspersing low priority write requests, it utilizes an otherwise idle I/O system for writing ahead modified pages from the database cache to disk. So these pages become replaceable at low costs [13].

Our results prove that this concept works well in most cases. But strict fairness in scheduling I/O requests affects the performance of sequential I/O. If one thread, performing sequential I/O, issues only one request at a time and another thread issues additional, possibly random I/Os, the former sequential I/O is permanently disturbed and becomes in fact random. This effect can be lessened, if both threads issue not only one request at a time, but a set of read-ahead requests against the I/O system. Then sequential requests are completed as batches without the overhead of positioning, resulting in near sequential I/O performance for sequential I/O while concurrent random I/O requests are still reasonably served.

4 Experimental Results

In the following, we will present the experimental findings of our parallelization approach. All measurements were made on a 2 CPU Intel Xeon server running both Linux 2.6.18 and Windows 2003 Server as operating systems. The results are equivalent for both operating systems.

Overall performance gain is examined using a query mix of partially complex ad-hoc queries that were extracted from a productive real-world data warehouse application.

This query suite is consisting of a total of 387 retrieval queries. For this presentation the queries were sorted along ascending elapsed times for sequential execution. In addition to the sequential time, the estimated optimal parallel query performance for a two-CPU system (Sequential/2) is shown as a theoretical lower boundary for parallel execution. A direct comparison to other RDBMS parallelization approaches is too extensive to fit in the given space constraints.

The following two diagrams of Figure 2 intend to clarify the interaction of the various components of our prototype. They do not show the full capabilities of our approach, but exhibit the outcome if critical features are disabled.

On the left side of Figure 2 only inter-operator parallelism is used, i.e. the QEP is parallelized as discussed but no pipelines are built. The first 219 are very short running queries (below 10 ms), with little or no potential for parallelization. Most of the measured elapsed time here is actually spent for client-server communication, SQL query compilation, and optimization. Query evaluation, although multithreaded, induces only a minor fraction to overall calculation time. These queries represent over 50% of this query suite. Note, that the cumulated difference graph ($CumDiff_n = \sum_{i=1}^{n} Par_i - Seq_i$, referring to the secondary axis) stays close to zero, meaning that parallelization has neither a positive, nor a negative impact for these queries.

Approximately 100 (queries 220-318) are medium-runners (10 to 600 ms), where parallelization is attempted. Most of them incur some minor overhead. The cumulated difference graph is rising slowly to its peak at 10.772 seconds. This indicates that this restricted parallelization is causing a performance penalty for medium-runners. Only

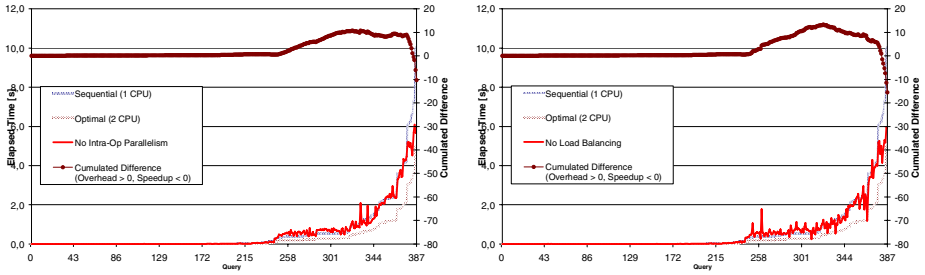

Fig. 2. Performance of *limited* parallel query execution. Left side: Intra-Operator Parallelism disabled. Right side: No load balancing. Queries are sorted by ascending sequential elapsed times. Cumulated Difference always refers to the secondary Y-axis.

the long-runner queries (elapsed times over 600 ms) show potential for parallelization. Here the cumulated difference falls monotonously below 0, i.e. this represents the total speedup of 4.5%. Clearly inter-operator parallelism is not sufficient for speeding up relational queries, because it offers no mechanism to eliminate performance bottlenecks. Only complex queries with extensive independent tasks can benefit.

The second measurement was carried out without load balancing. Data pipelines are established and immediately activated. This measurement shows more distinct peaks, depending on how near the fully parallel QEP happens to be to the optimally balanced plan. Some queries are close to the optimal performance but in total the cumulated difference peaks at 13,124 ms for medium runners. Again, this is compensated by the long-runners leading to a total speedup of 6.9%.

Both results are not very satisfactory as the medium-runners in both cases account for a perceptible overhead. In the next step we will examine the combination of all discussed proposals. The left diagram in Figure 3 shows elapsed times of queries 1 to 318. The right diagram shows the remaining queries in a different scale. Once again the short running queries are almost unaffected by parallelization. However, in this scenario the medium-runners sometimes pay off and sometimes incur some minor overhead. In total both effects almost eliminate each other, as the cumulated difference graph is rising only very slightly above zero, i.e. parallelization is causing a negligible performance penalty for medium-runners. The cumulated difference rises as high as 1.651 seconds, and is falling later on. It is adding up to an average penalty of 10 ms per query for the medium runners. However, an integral examination shows that the average overhead per medium-runner is only about 3%, which we consider acceptable. Moreover, a total of 1.651 seconds of overhead seems negligible compared to a cumulated runtime of 34.852 s for the short and medium-runners and a total elapsed time of over 226.5 seconds for the compete query suite. The long-runner queries (Figure 3, right hand side, elapsed times over 600 ms) consistently show good potential for parallelization. In this phase the cumulated difference falls monotonously far below 0. Clearly the performance of the parallel query execution is close to the estimated optimum. Only few long-runners stay close to the sequential performance. The reason for this would-be poor parallelization is that the involved relational operators in these queries are inherently barely parallelizable. Still, it is noteworthy that

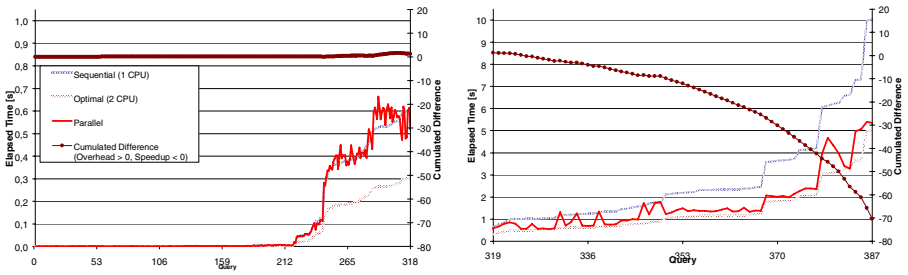

Fig. 3. Overall performance of parallel query execution. Left side: Short-runners (queries 1-219) and medium-runners (220-318). Right side: Long-Runners (319-387). Queries are sorted by ascending sequential elapsed times. Note the different scale on the primary Y-axis. Cumulated Difference always refers to the secondary Y-axis.

parallel query execution never surpasses sequential performance and that those that come close to sequential behaviour are very few.

In total, parallelization reduces the total elapsed time for the whole query suite from 226.5 to 156 seconds, i.e. below 70 %. If only the long-runners are accounted for, the ratio sinks even below 63%. Similar results where produced on several other multicore machines without changing any parameters. This emphasizes our claim that the ASYNC operator is adaptive and universally applicable, independently from any particular hardware configuration.

5 Conclusion

We presented a complete evaluation of an approach to parallel relational query execution that is based on encapsulation of parallelism into the relational ASYNC operator. This work includes an algorithm that is capable of efficiently parallelizing sequential execution plans. Parallelization provided by this algorithm is sufficient to generate query execution plans that reach maximum resource utilization during query execution. Continuously high resource utilization in the presence of data skew and varying machine workloads is guaranteed by the robust and powerful dynamic load balancing capabilities of the ASYNC operator. Measurements in a productive environment have proven the capabilities and maturity of this concept and its implementation. They confirm near-linear speedup for queries that are well-suited for parallelization and a respectable average run-time reduction by over 30% for an extensive ad-hoc query suite. On the other hand, our approach incurs no noteworthy performance loss for queries that are adverse to parallelization, because they run too shortly or their relational calculus offers no possibility for parallelization.

Future work on the ASYNC operator will concentrate on further improving interaction with an asynchronous I/O system. Another field is the extension of parallelization of selected relational operators, such as GROUP and AGGR. Finally some fine-tuning is planned to improve the cooperation of the ASYNC operator and the main memory cache hierarchy, in order to reduce memory stalls in SMT environments by implementing aggressive data prefetch into the ASYNC node.

References

1. DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsaio, H.I., Rasmussen, R.: The gamma database machine project. IEEE Transactions on Knowledge and Data Engineering 2(1), 44–62 (1990)
2. Copeland, G., Alexander, W., Boughter, E., Keller, T.: Data Placement in Bubba. In: Proceedings of the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, USA, June 01-03, pp. 99–108 (1988)
3. Stonebraker, M., Katz, R.H., Patterson, D.A., Ousterhout, J.K.: The Design of XPRS. In: Proceedings of the 14th International Conference on Very Large Data Bases, August 29-September 01, pp. 318–330 (1988)
4. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: SIGMOD 1990: Proceedings of the 1990 ACM SIGMOD international conference on Management of data, pp. 102–111. ACM Press, New York (1990)
5. Graefe, G., Cole, R.L., Davison, D.L., McKenna, W.J., Wolniewicz, R.H.: Extensible Query Optimization and Parallel Execution in Volcano. Morgan-Kaufman, San Mateo (1994)
6. Lu, H., Tan, K.: Dynamic and load-balanced task-oriented database query processing in parallel systems. In: Pirotte, A., Delobel, C., Gottlob, G. (eds.) EDBT 1992. LNCS, vol. 580, pp. 357–372. Springer, Heidelberg (1992)
7. Mehta, M., DeWitt, D.J.: Managing intra-operator parallelism in parallel database systems. In: VLDB 1995: Proceedings of the 21th International Conference on Very Large Data Bases, pp. 382–394. Morgan Kaufmann Publishers Inc., San Francisco (1995)
8. Zhou, J., Cieslewicz, J., Ross, K.A., Shah, M.: Improving Database Performance on Simultaneous Multithreading Processors. In: Proc. VLDB Conference, pp. 49–60 (2005)
9. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs on a modern processor: Where does time go? The VLDB Journal, 266–277 (1999)
10. Englert, S., Gray, J., Kocher, R., Shah, P.: A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases, Tandem Computer Systems Report (1989)
11. Hong, W., Stonebraker, M.: Optimization of parallel query execution plans in XPRS. In: PDIS 1991: Proceedings of the first international conference on Parallel and distributed information systems, pp. 218–225. IEEE Computer Society Press, Los Alamitos (1991)
12. Hong, W.: Exploiting inter-operation parallelism in XPRS. In: SIGMOD 1992: Proceedings of the 1992 ACM SIGMOD international conference on Management of data, pp. 19–28. ACM Press, New York (1992)
13. Hall, C., Bonnet, P.: Getting Priorities Straight: Improving Linux Support for Database I/O. In: Proc. VLDB Conference (2005)