Adaptive query parallelization in multi-core column stores

Mrunal Gawade
CWI, Amsterdam
mrunal.gawade@cwi.nl

Martin Kersten
CWI, Amsterdam
martin.kersten@cwi.nl

ABSTRACT

With the rise of multi-core CPU platforms, their optimal utilization for in-memory OLAP workloads using column store databases has become one of the biggest challenges. Some of the inherent limitations in the achievable query parallelism are due to the degree of parallelism dependency on the data skew, the overheads incurred by thread coordination, and the hardware resource limits. Finding the right balance between the degree of parallelism and the multi-core utilization is even more trickier. It makes parallel plan generation using traditional query optimizers a complex task.

In this paper we introduce *adaptive parallelization*, which exploits execution feedback to gradually increase the level of parallelism until we reach a sweet-spot. After each query has been executed, we replace an expensive operator (or a sequence) by a faster parallel version, i.e. the query plan is morphed into a faster one. A convergence algorithm is designed to reach the optimum as quick as possible.

The approach is evaluated against a full-fledged column-store using micro-benchmarks and a subset of the TPC-H and TPC-DS queries. It confirms the feasibility of the design and proofs to be competitive against a statically optimized heuristic plan generator. Adaptively parallelized plans show optimal multi-core utilization and up to five times improvement compared to heuristically parallelized plans on the workload under evaluation.

1. INTRODUCTION

Column store databases are designed with a focus on analytical workloads. Almost all database vendors these days have a column store implementation. A recent study by Microsoft showed that a majority of real world analytic jobs process less than 100 GB of input [3]. This can be accommodated by an in-memory solution on a single high-end server. They come with an abundance of CPU power using tens of cores [27, 23]. Query parallelization is one of the ways to utilize multi-cores. This calls for a renewed look at the traditional query parallelization techniques, such as the *exchange operator* based parallelization [15], since the state of the art column store systems such as IBM BLU accelerator [24], HyPer [30] use work stealing based approach for multi-core scalability.

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

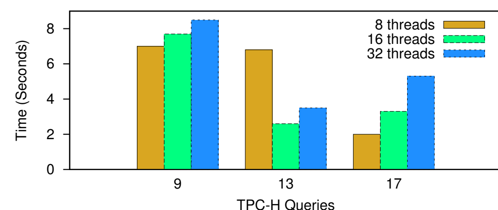


Figure 1: **Response time variations due to varying degree of parallelism under concurrent workload (32 hyper-threaded cores)**.

An important issue is the degree of parallelism **(DOP)** of a plan which reflects the maximum number of parallel operator executions. With tens of cores on CPUs, finding the optimal degree of parallelism of a query plan using heuristic and cost model based exchange operator approach is difficult [2]. Some of the prominent problems are a huge multi-core aware plan search space, parallelism aware accurate cost model estimations, and the optimal placement of exchange operators in the plan. The degree of parallelism problem becomes even more difficult under a concurrent workload due to competition for shared resources, such as CPU cores, memory, and memory controllers. This forces many systems to take a conservative approach towards plan parallelization decisions, as a sub-optimal parallel plan could often degrade performance. Often a serial plan is preferred as long as it ensures a robust performance [1].

For example, consider Figure 1, which shows execution of three TPC-H heuristically parallelized queries for different DOP under a heavy concurrent CPU bound workload, which ensures 0% CPU core idleness (Scale factor 10 on 256 GB RAM with 32 hyper-threaded cores). The queries show varying performance under different DOP. The traditional plan generation approaches based on heuristic and cost model [10] fall short, as the plans do not reflect runtime resource variations, making them suboptimal under a concurrent workload.

We introduce *adaptive parallelization*, a new mechanism to generate *range partitioned* parallel plans using query execution feedback, while taking into account the run-time resource contention. Adaptive parallelization generates a better plan (P1) from an old plan (P0) in a greedy manner, by parallelizing the most expensive operator from P0, under repeated query invocations. The inspiration is derived from the observation that in real world systems the same query templates get reused multiple times only changing some parameters. Starting with a serial plan, each successive query invocation results in a new parallel plan, until a near minimal execution time parallel plan is detected, which ensures a near optimal DOP. Adaptive parallelization under concurrent workload reflects resource contention, making adaptive parallelized plans resource

contention aware [11]. The success of adaptive parallelization depends on its ability to converge quickly, while ensuring a near minimal execution parallel plan.

Adaptive parallelization also allows to analyze the relation between DOP and multi-core utilization. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. Maximum multi-core utilization however need not improve performance, as it might lead to memory bandwidth pressure due to parallel operator executions [22]. Hence, finding the right balance between the DOP and multi-core utilization is important. Since adaptive parallelization generates new parallel plans incrementally, it enables us to analyze the relation between DOP and multi-core utilization. Adaptive parallelized plans have minimal multi-core utilization and a near optimal degree of parallelism, which helps in achieving better response time during concurrent workloads.
We summarize our main contributions as follows.

- We introduce *adaptive parallelization*, a new execution feedback based parallel plan generation technique, that ensures near optimal degree of parallelism.
- We introduce an adaptive parallelization convergence algorithm for different scenarios.
- We analyze the parameters affecting the speedup of the core relational algebra operators.
- A near optimal DOP allows adaptive parallelized plans to show up to five times response time improvement compared to heuristically parallelized plans.

Paper outline: The paper is structured as follows. In Section 2 we describe the architecture of adaptive parallelization. We also provide parallelization heuristics for operators and illustrate the dynamic partitioning scheme and discuss related problems. Section 3 describes the convergence algorithm to find the near minimal execution parallel plan along with various convergence scenarios. In Section 4 we provide a detailed experimental evaluation. Related work is described in Section 5. We conclude citing the major lessons learned in Section 6.

2. ARCHITECTURE

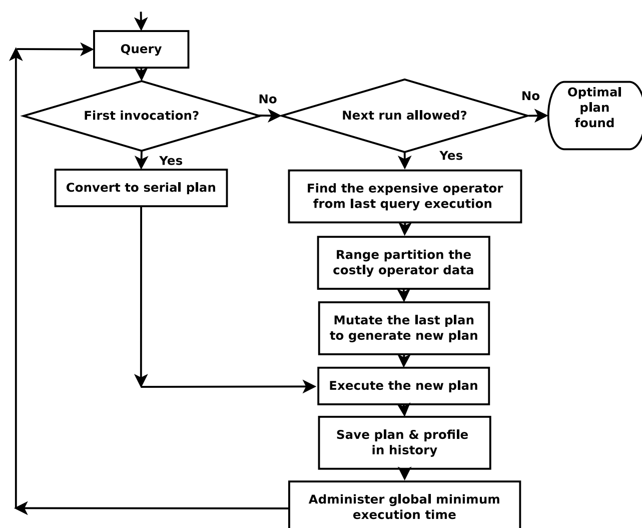


Figure 2: **Adaptive parallelization workflow.**

Adaptive parallelization could be used by any columnar database system as long as its plan representation allows identification of individual expensive operators.

Run-time environment: It consists of a scheduler, an interpreter, and a profiler. The scheduler uses a data-flow graph based scheduling policy, where an operator is scheduled for execution once all its input sources are available. While an interpreter per CPU core executes the scheduled operators, the profiler gathers performance data on an executed operator basis. The profiling overhead is minimal due to vectorized nature of execution. The profiled data consists of operator's execution time, memory claims, and thread affiliation id. Cost model based plan generation approaches often suffer from incorrect cardinality estimates. We use a heuristic plan generation approach where parallelization decisions are based on execution time feedback, without a need for operator's cardinality statistics.

The execution time is a good metric for parallelization decisions as it reflects the system state such as the memory bandwidth pressure and the processor usage. Though the presence of system noise might affect execution time, such disturbances level out during adaptation.

Infrastructure components: The Adaptive Parallelization **(AP)** infrastructure is implemented using the following components a) operator stubs to morph a plan based on past behavior, b) the plan administration policies to choose a suitable plan from the plan history, and c) the AP convergence algorithm, which we describe in Section 3.

Workflow: The adaptive parallelization work-flow is summarized in Figure 2. The first phase is similar to most systems [16] where an optimal serial plan (Figure 3 Plan 1) is generated. Our approach differs in the second phase where the the query is cached, plan is fed to the framework, executed, and the profiling information such as the query execution time, the operator execution time, the number of invocations, etc. are stored. On the next query invocation a new parallel plan (Figure 3 Plan 2) is derived from the immediate old plan (Plan 1) by parallelizing the most expensive operator (Select on input A). The AP process iterates by invoking the same query again and generating parallel plans in an incremental manner by parallelizing the most expensive operators in successive steps. The number of iterations to find the minimal execution time parallel plan is controlled by a convergence algorithm described in Section 3. As the book keeping and compilation time is minimal we only report the execution time.

Why feedback based approach? Like parallel databases, multi-core CPUs make the parallel plan search difficult [19]. The main problem is finding the *optimal number of partitions* per operator for an *optimal input serial plan*. Finding an optimal input serial plan is out of the scope of this paper. During parallelization when an operator's data is partitioned, there are combinatorial possible choices for the partition size. For example, in the worst case, each operator's data could be partitioned in a single tuple, such that the total number of operators equal the number of tuples. In the best case a single operator could work on the entire non-partitioned input. The possible partition size choices for different operators represent multiple parallel plans with different execution times, making this a combinatorial plan search problem. The plan search space exploration is usually done using a combination of both the heuristic and the cost model based approach. It allows to prune the search space for an efficient search. Overall, finding an optimal multi-core aware parallel plan using traditional approaches is difficult. In comparison the feedback based approach we propose is relatively easy, as the assumption is the input serial plan we start with is an *optimal plan.* Since the approach explores the search space in a guided way

by parallelizing only the most expensive operator, we avoid a large space of uninterested plans.

2.1 Plan mutation

We refer to the process described in the workflow as *plan mutation*. Plan mutation could be guided by different policies. In this paper we use parallelization of the *expensive operator* in a plan as the guiding principle. An operator is considered *expensive* if its execution time is the highest amongst all operators. Based on the complexity, we categorize mutation in three types as Basic, Medium, and Advanced.

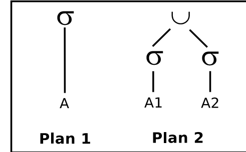


Figure 3: **Basic mutation.**

Basic: Basic mutation involves parallelization of an expensive operator by introducing two new operators of the same type, called expensive operator's **cloned operators**. The cloned operators work on the expensive operator's partitioned data. Partitioning is cheap when it involves no data copy, but introducing range partitioned sliced view of the columnar data. (Value / hash based partitioning needs the presence of a partition operator, about which we discuss in Section 5.) An *exchange union* operator (either a newly introduced or an existing one) combines the result of the cloned operators. In Figure 3 we see one such example for select operator parallelization.

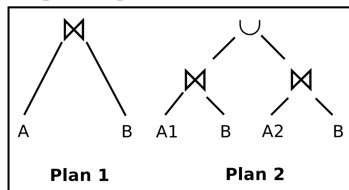


Figure 4: **Join parallelization.**

The two most popular algorithms for the join operator are the hash join and the sort merge join. We analyze the hash join implementation as it suits most workloads due to the omnipresence of non-sorted data. We consider adaptive parallelization of the join operator plan (Figure 4 Plan 1) when only the larger (outer) input is split into equi-range partitions on consecutive runs. Figure 4 Plan 2 shows the parallelized plan with the two new join cloned operators. An exchange union operator combines the output of the cloned operators.

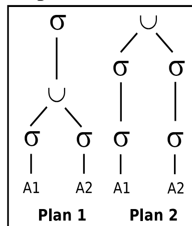


Figure 5: **Medium mutation.**

Medium: Medium mutation handles plan parallelization when the *exchange union* operator (U) itself turns out to be expensive, as a result of intermediate data copying due to low selectivity input. This mutation stage arises, when the exchange union operator is introduced as a result of the basic mutation.

Figure 5 shows one such example where Plan 1 with an expensive exchange union operator is mutated into Plan 2. The mutation process involves propagating the inputs to the exchange union operator, to its data flow dependent operators. The data flow dependent operators are cloned to match the exchange union operator's input. Finally a newly introduced exchange union operator combines the result of the cloned operator's output.

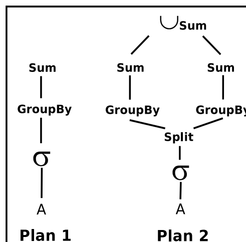


Figure 6: **Advanced mutation.**

Advanced: Advanced mutation involves parallelization of operators such as group-by and sort, that do not exhibit the filtering property (selectivity = 0).

Figure 6 shows parallelization of a group-by operator with the advanced mutation. The expensive operator (group-by) is parallelized by introducing two cloned operators on its equi-range partitioned data. Next the aggregation operators such as sum and average are parallelized by introducing two aggregation cloned operators. The cloned operators (group-by) result is propagated to the aggregation cloned operators (sum). Finally, an exchange union operator combines the parallelized aggregation operators result. Since the aggregation cloned operators always show very high filtering property, the exchange union operator combining their result is cheap.

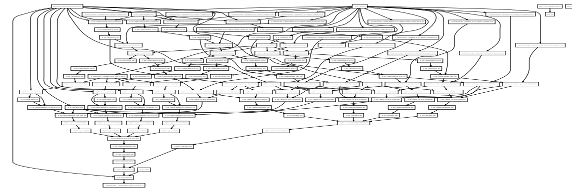


Figure 7: **Complex operator dependencies in TPC-H Q14 parallel plan. Rectangles represent operators, and edges between them represent the dependencies. The graph is only meant to give a high level perspective of the plan's complexity, abstracting individual operator details. [12] shows graphs where operators are visible.**

Summary: A relational operator gets parallelized in two cases. In the first case, the operator itself might be expensive and gets parallelized using either the basic or the advanced mutation. In the second case, operator parallelization occurs as a result of using the medium mutation, where the operator is in the data flow dependent path of the expensive exchange union operator. In both cases identifying and resolving the parallelizable operator's output propagation dependency across the entire plan is an essential step.

The three mutation schemes we described cover all possible mutations as an operator could either get parallelized due to its own expensiveness or as a result of its presence in the data flow path of another parallelizable operator.

2.2 Making plans simpler to mutate

Most columnar systems [1, 4, 7, 18] use a simple representation of plan with operators represented using physical algebra. The operators use standardized interfaces for individual columnar data and related argument passing. Column store specific functionality such as operations on multiple columns and tuple reconstruction are mostly hidden away as the internal logic in the execution engine framework. Some column stores like the open-source system MonetDB, however use an abstract language to represent plans [6], where column store specific functionality such as the tuple reconstruction and other columnar operations is exposed in the plan representation itself. Use of operators with different semantics and specialized operators such as the tuple reconstruction operators is common. Figure 7 shows one such plan with complex data flow dependencies.

Plan mutations using either the medium or advanced mutation involves resolving parallelized operator's propagation dependencies. Hence, care has to be taken to resolve parallelized operator's propagation dependencies. To make plan mutations simpler, modification of some of the operator's semantic representation is needed. We describe the related aspects in the rest of the section.

Adaptive parallelization and operator semantics: Operators can have different semantics depending on primitives being used. Adaptive parallelization could further add more information such as as the partition under use, total number of partitions, etc. Plan mutations thus generate combination of different operator semantics.

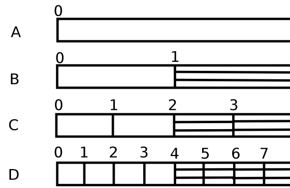


Figure 8: **Column A dynamically partitioned with iterations.**

For example, consider the case of the filter operator which can have two representations. One which accepts normal columnar input and the other which accepts column and also a bit vector from another selection operator's output. Hence, the filter operator could be represented using two primitives depending on the number and the type of inputs. Depending on the data flow dependency, a suitable filter operator gets parallelized during plan mutations.

Adaptive parallelization uses different parallelization rules catered to different operator semantics. Since any operator could be expensive, resulting in its parallelization, the challenge for different mutation schemes lies in how well they are able to resolve the data flow dependencies across different operator semantics.

Plan rewriting: One of the techniques to ease the mutation process is to modify the original input serial plan from the SQL compiler using a query rewriter. The rewriter substitutes original operators (for example, aggregation operators and tuple reconstruction operators) with new adaptively parallelization aware operators. These new operators use modified implementation of operators such as group-by, aggregation operators (sum, avg), and sort, by keeping their original semantics, but with changed arguments ordering, to resolve possible operator propagation dependencies. For systems with simple plan representations the operator propagation dependencies due to multiple columns could be handled in the execution engine framework logic.

2.3 Adaptive parallelization aware partitioning

In a column store the operators operate on an array or vector representation of the data. For readability, we consider the array representation with range partitioning. It involves creating read only slices on the base or the intermediate column. Creating slices involves marking the boundary ranges for the base or intermediate columnar data and is cheap, as there is no data copying involved. This technique could be also used during vectorized execution where the vectors are derived from the partitioned range of the base and intermediate input. We briefly describe a value based partitioning approach use case in Section 5.

Dynamic partitioning: Adaptive parallelization generates dynamically sized partitions on the base or intermediate column, as any operator could be parallelized during successive iterations. In contrast a heuristically parallelized plan often uses a fixed number of partitions based on the available CPU cores. To explain dynamic partitioning of a column using a select operator, we use Figure 8.

When the select operator on the column in 8A turns expensive, the column is sliced in two partitions represented by 8B. When the select operator on partition 1 in 8B turns expensive, two new partitions are introduced, represented by 2nd and 3rd in 8C. Now there are three select operators, one on 0th partition of 8B and two on 2nd and 3rd partition of 8C. When the select operator on 2nd partition in 8C becomes expensive, it is divided further and two new partitions 4th and 5th in 8D are introduced. So now there are total 4 select operators working on 0th partition of 8B, 3rd partition of 8C and 4th,5th partition of 8D. Please note that the partitions are of different sizes and their boundaries are aligned on the base column in 8A. Maintaining the alignment during dynamic partitioning is important, as misalignment could lead to problems such as a) repetition of data b) omission of the data across different operator partitions. Thus, dynamic partitioning allows the operators to work on different sized partitions of the same column in parallel.

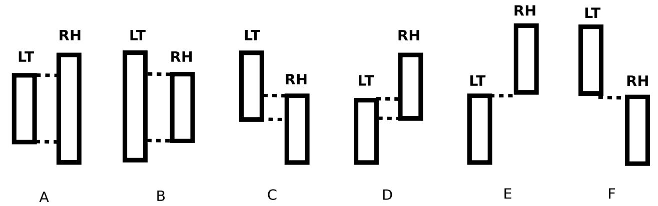


Figure 9: **Different alignment scenarios between columns during tuple reconstruction due to dynamic partitioning.**

Dynamic partitioning and tuple reconstruction: Tuple reconstruction is a well known problem in column stores [17] and is implemented as join lookup. Column stores use either early or late materialization strategies for column projection using tuple reconstruction, which involves using row-ids to fetch values from the column that needs projection. For example, consider the columnar representations in Figure 10, which shows head (H) and tail (T) columns grouped as Left (L) and Right (R). The head column (LH / RH) contains row-ids, whereas the tail column contains either row-ids (LT) or actual values (RT). The @ indicates a row-id. When the head column (LH / RH) contains consecutive row-ids, it is not materialized and used as a virtual column. During tuple reconstruction, the row-ids in the left tail (LT) are used as an index in (RH) to fetch the corresponding values from the right tail (RT). For example, row-ids 2, 4, 5, 7 from LT are probed in the RH, whose corresponding values in RT are 12, 11, 20, and 13.

One important aspect is the effect dynamic partitioning has on the tuple reconstruction due to possible misalignment between LT and RH. When row-ids from LT are used as an index to fetch values from RT, the row-ids in LT should be a subset of row-ids in RH. If not, then a lookup using row-id in LT, for the row-id index in RH does not exist, resulting in an invalid access.

LH	LT
1@	2@
2@	4@
3@	5@
4@	7@
5@	8@

RH	RT
1@	15
2@	12
3@	44
4@	11
5@	20
6@	16
7@	13

Figure 10: **Tuple reconstruction between two columns.**

Since adaptive parallelization generates variable sized partitions, it gives rise to different alignment scenarios as shown in Figure 9 (B,C,...F). Consider the mis-alignment example in Figure 10. Here LT start row-id=2, which is greater than RH start row-id=1, and LT end row-id=8, which is greater than RH end row-id=7. Hence, LT's upper boundary starts after RH's upper boundary, whereas LT's lower boundary extends beyond RH's lower boundary as represented in Figure 9D. In Figure 9 the lengths of columns provide just a logical representation of over and undershooting of boundaries, and do not represent the actual content. To maintain the alignment the lower boundary of LT is adjusted by removing row-id=8, to match the lower boundary of RH. The correct boundary alignment is represented by dashed lines in Figure 9D. Adaptive parallelization depending on the operator semantics uses one of the alignment scenarios, to make sure that the partitions align correctly. Fixed size partitions always lead to correct alignment (See Figure 9A), resulting in a valid access.

Another important aspect arises when the output of operators working on the dynamically partitioned data is packed together. Here the exchange union operator must maintain the correct ordering to avoid the incorrect results. The correct ordering is maintained, as the operators whose results are packed follow the mutation sequence order, hence the results being packed together fol-

low the same order. Adaptive parallelized plans could become very large due to successive partitioning and operator propagation, which could make partition misalignment related problems, if any, hard to identify and resolve.

Plan explosion: As adaptive parallelization involves propagating the parallelized operator's output on its dependent operators, the plans could quickly grow large. Plan explosion results as a side effect of the exchange union operator removal during the *medium* mutation. For example, when a descendant of the same type of operator stays expensive during successive invocations, it gets parallelized and a single exchange union operator combines the output of all such parallelized operators. As a result the number of input parameters to the exchange union operator could become very large. Eventually if the exchange union operator itself turns expensive, it is removed using the *medium* mutation. This leads to a plan explosion, as the medium mutation propagates inputs of the exchange union operator on its data flow dependent operators. For each operator in the data flow path, new instructions (operators) which equals the number of the exchange union operator inputs are added in the plan. Hence, if the number of input parameters to the exchange union operator is large, the plans could grow very large.

The growth of large plans is suppressed by not removing the exchange union operator if its input parameters cross a certain threshold. The threshold in the current implementation is 15 parameters, chosen on the basis of empirical observations from different parallelization cases. Suppressing the exchange union operator removal however stops further plan parallelization, as the exchange union operator stays the most expensive operator in all further query invocations.

We have described adaptive parallelization aware infrastructure changes so far. Obtaining a minimal execution parallel plan however depends on how fast the adaptive parallelization process converges. In the next section we describe a new algorithm that ensures convergence in different scenarios.

3. ALGORITHM

In this section we introduce the heuristics for the global minimum execution identification from the set of available plans and the corresponding convergence algorithm. The algorithm is loosely inspired by the hill climbing approach [26]. Figure 11 shows different cases of the presence of minima, plateaus, and up-hills in the execution times, during adaptive parallelized runs of a join operator plan. We refer to the minimal execution time amongst them as the *global minimum execution (GME)*. Like most systems that generate parallel plans, our base assumption is an optimal input serial plan. Hence, the focus of parallelization is to identify the optimal number of partitions for operator's data in the input plan. Problems such as sub-optimal parallel plans due to poor join ordering which might require backtracking are not considered, as the input is an optimal serial plan.

The convergence algorithm should be able to find the GME in all cases of minima, plateaus, and up-hills in the execution time, and converge in minimal number of runs. Next we formally define the GME first, the convergence algorithm next, and then illustrate different convergence scenarios.

3.1 Global minimum execution (GME)

As the runs progress, the GME is the minimal execution time amongst so far observed runs, and keeps on changing, during an active adaptive parallelization instance.

We denote the current run's execution time as $CurExec$. The *execution time improvement (CurExecImprv)* at the current run is calculated with respect to 0th run's (SerialExec) execution time.

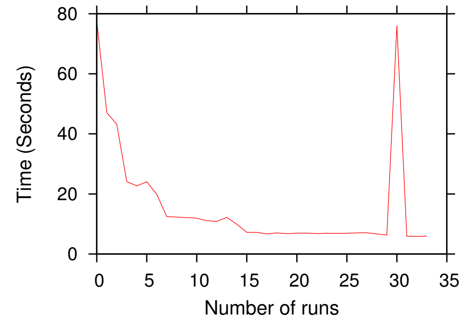


Figure 11: **Adaptive parallelization convergence algorithm scenarios for join operator parallelization.**

$CurExecImprv = |(SerialExec - CurExec)|/SerialExec.$

To calculate the first GME improvement, we initialize GME to the first run's execution time after the serial execution (0th run).

$GMEimprv = |(SerialExec - GME)|/SerialExec.$

As the runs progress, new GME needs to be identified. A new run's exuection time becomes the new GME, if the run's execution time improvement is better than the current GME's execution time improvement by a certain threshold.

$GME = CurExec$
$\{if(CurExecImprv - GMEimprv) > threshold\}.$

As the runs progress, the new execution times could be slightly lower than the existing GME, hence, selecting the correct threshold is important in discarding such new execution times, which otherwise could become the new GME. For example, consider a hypothetical adaptive parallelization instance. Let CurExecImprv at the 8th run be 96%, GMEimprv at the 3rd run = 90%, and threshold = 5%, then *(CurExecImprv - GMEimprv) >5*. Hence, the Current run Execution time at the 8th run is considered the new Global Minimum Execution (GME). Correct tuning of the *threshold* parameter is thus crucial as it helps to discard multiple possible GMEs and to chose the optimal execution time amongst them, as the new GME.

Finding GME could be also difficult due to the presence of many local minima, about which we illustrate next.

The global minimum detection problem: The problem could be formally stated as finding the global minimum execution from many local minima that occur as the runs progress. When the execution time of a run is more than its previous run, a local minimum results at the previous run. For example, a local minimum occurs at the 4th run in Figure 11. The convergence algorithm has to overcome many such local minima during its exploration of the global minimum. We use the rate of improvement in the execution time of the runs as a heuristic, to avoid the local minimas.

The execution time of consecutive runs could improve or worsen depending on the run-time conditions (execution skew, operating system interference), giving rise to positive or negative *rate of execution time improvement* **(ROI)**. The ROI of a run is defined with respect to its previous run's execution time *(PrevExec)*. We define ROI as follows.

$ROI = (PrevExec{-}CurExec)/MAX(CurExec, PrevExec).$

In Section 3.2 while describing the core convergence algorithm, we illustrate how to use ROI to avoid local minimas. Finding GME is difficult, however, another equally difficult task is to find it in the minimal convergence runs, about which we illustrate next.

The minimal run convergence problem: The problem could be formally stated as finding GME in a minimal number of runs, dur-

ing consecutive query invocations. Too few runs have a risk of non-occurrence of the global minimum and the algorithm converging on a local minimum. Too many runs might ensure a global minimum at the cost of a slow convergence. Hence, finding the right balance between the minimum convergence runs and the GME is of prime importance for the convergence algorithm.

3.2 Convergence algorithm

We describe the convergence algorithm using the context described so far in Section 3. The aim is to find the GME in minimal number of convergence runs. We model the number of convergence runs *(Convergence_Runs)* using the parameters ***credit*** and ***debit***. A credit reflects the number of runs accumulated at each run due to a positive ROI. A debit reflects the number of runs accumulated at each run due to a negative ROI.

$Credit = Credit + (ROI * Number_Of_Cores).$
$Debit = Debit + (|(ROI)| * Number_Of_Cores).$

The value of *(credit - debit)* at each run reflects the balance (*Convergence_Runs*) available for the system to converge. Hence, the next run is allowed only if the balance is positive i.e. *((credit - debit) >0)*.

$Convergence_Runs = Credit - Debit = f(ROI).$

The algorithm starts with the value of credit = 1 and debit = 0. When parallelism reduces the execution time, the ROI of the first run is positive and very high (Figure 11 - The algorithm starts with the 0th run). With an increase in runs, the ROI decreases. During the initial few runs the algorithm should ensure availability of sufficient runs as a balance, to avoid premature convergence. During the later runs, as the ROI slows down, the algorithm should ensure as few balance runs as possible, to ensure fast convergence. From the formula above, as both credit and debit are dependent on ROI, they are a function of ROI, which makes the Convergence_Runs also a function of ROI. The algorithm convergence is hence guaranteed, since the heuristic ***Credit - Debit >0***, which decides the available *Convergence_Runs* becomes invalid eventually. Next we describe various convergence scenarios and how the heuristic *Credit - Debit >0* becomes invalid, which guarantees the algorithm's convergence.

3.3 Convergence scenarios

We identify three scenarios during which the algorithm should ensure the convergence, 1. No premature convergence in a local minimum before identifying a global minimum. 2. No extended convergence, and 3. The convergence in a noisy environment. We expect these scenarios to cover the entire spectrum as the aim is to find the global minimum, and the possible problems for the convergence algorithms could be its early termination, late termination, and termination during noisy environment. We describe these scenarios next.

3.3.1 No premature convergence

When parallelism improves the execution time, the first run always has a very high ROI (Figure 11 - The algorithm starts with the 0th run). Hence, the credit accumulated after the first run is very high with an upper limit of *(Number_Of_Cores + 1)*. This ensures that there are sufficient runs available as a balance in the system during the initial stages to overcome plateaus and up-hills. Each run after the first run contributes more credit, ensuring more runs. This is also analogous to the concept of accumulation of the *potential energy* by a body when it falls from great heights. The greater the height, the higher the potential energy. The energy allows the body to keep moving in plateaus and climb high hills, as long as there is a balance energy.

3.3.2 No extended convergence

Accumulation of high credit in the few initial runs on a stable system could result in a state where the algorithm *never* converges. In a stable system the execution time variations are minimal, leading to fewer *debits* being made. In such a system, the proportion of accumulated *credit* will always be much higher than the accumulated *debit* after a few initial runs. For example, consider Figure 11. After 15 runs the ROI is minimal, ensuring that no new significant credit or debit is introduced. However, the accumulated credit till 7 runs is very high, as the ROI till 7 runs is very high. This situation leads to non-convergence as there are always balance runs available i.e. *(credit - debit<= 0)* is never true.

Leaking debit: To ensure the algorithm converges in a finite number of runs we introduce the concept of *leaking debit*. In this scheme after a *threshold* on the number of runs is crossed, a constant debit gets deducted from the available credit at each run. *It ensures the available credit is drained to 0, so that the algorithm converges in a finite number of runs.* Hence, *leaking debit* is a function of the available *credit* at the threshold run. The threshold run value is calibrated to be the *Number_Of_Cores* on the CPU. It ensures at least those many runs are used to find the optimal execution time. The *Leaking_Debit* is calculated by dividing the available credit at the threshold run amongst the possible remaining number of runs during the global minimum search.

$Remaining_Runs = Extra_Runs * Number_Of_Cores.$
$Leaking_Debit = Credit/Remaining_Runs.$

Based on plan complexity, some queries converge early, while some take longer after crossing the *threshold run* reference. To avoid premature convergence, the system specific tunable parameter *Extra_Runs* is used, which ensures that the *remaining number of runs* to search the global minimum are sufficient. Note that *Remaining_Runs* is just an approximate bound. Plan representations vary considerably across systems. Hence, based on empirical observations from different parallelization cases, and multiple experimental runs (five), for the current platform, Extra_Runs=eight is considered a safe boundary value to avoid the premature convergence. Higher values result in an extended convergence.

3.3.3 Convergence in a noisy environment

Depending on the stability of the run-time environment (operating system process interference, memory flushes, etc.) the execution time of an individual run could vary considerably. The execution time of some of the runs in a noisy environment is often greater than the serial plan execution time. One such peak is visible in Figure 11 at the 30th run. Most peak executions are followed and preceded by a normal execution. If care is not taken such peaks will make the algorithm halt *immediately* as the debit due to peak ascent will be higher than the accumulated credit. Hence, the algorithm should converge gracefully in such a noisy environment.

Our solution is to mark all such unique peaks as outliers, and ignore their presence. The algorithm incorporates this by allowing the immediate next run to execute. This ensures the balance runs stay unaffected, as the *debit* made during the peak *ascent* is compensated by an equivalent *credit* during the peak *descent*, during the next run. Concurrent workload could also affect the convergence, however, tuning the *Extra_Runs* parameter to find the leaking debit should take care of it.

3.3.4 Global minimum plan identification proof

The convergence algorithm should ensure a global minimum plan while converging in a reasonable number of runs. The lower bound

Table 1: System configuration

CPU	Sockets	Threads	L1 cache	L2 cache	Shared L3 cache	Memory	OS
Intel Xeon E5-2650@ 2.00GHz	2	32	32KB	256 KB	20MB	256GB	Fedora 20
Intel Xeon E5-4657Lv2@ 2.40GHz	4	96	32KB	256KB	30MB	1TB	Fedora 20

on the convergence runs is *Number_Of_Cores + 1*, while the upper bound approximates between *(Number_Of_Cores + 1 + Remaining_Runs)* and extra runs added to the previous upper bound, if any, due to a large credit accumulation. The convergence runs are directly influenced by the *Leaking_Debit*, and credit / debit accumulation.

The global minimum plan's existence beyond the upper bound of the convergence runs is not possible. We provide a proof by contradiction. If such a plan exists then its execution should be significantly better. In that case the corresponding expensive operator should have been identified much earlier, even before the first upper bound on the convergence runs is reached. If multiple such plans exists, then that indicates improved execution with each run. Such improvement should then add extra runs (more credit) to the first upper bound on the convergence runs, which would prolong the global minimum search further, to find a more optimal plan. Hence, no matter the situation, a near global minimum plan is identified in the available convergence runs. In all the convergence scenarios when the heuristic ***Credit - Debit >0***, that decides available *Convergence_Runs*, turns invalid, the algorithm converges.

4. EXPERIMENTS

Adaptive parallelization is implemented in MonetDB, being the only full fledged open-source columnar system, with memory mapped columnar representation for the base and the intermediate data. The operators are represented in an intermediate language called MonetDB Assembly Language (MAL) [6], with their implementation in C. The operators have variable number of arguments depending on their semantics, and form complex data flow patterns in MAL plans, as shown in Figure 7.

Table 1 summarizes our experimental hardware platform, which consists of two types of machines, with two and four socket CPUs each. All experiments, unless mentioned, use in-memory data (without disk IO) on the two socket machine. Heuristic parallelization unless mentioned uses 32 threads. Each graph plots an average of four runs of the same experiment. We use the four socket machine to test one of the workload's scalability from NUMA perspective.

The experimental section is divided into two broad categories. In the first we analyze how parallelization gets affected by various operator level parameters. In the second we analyze it at the SQL query level. We use a mix of micro-benchmarks, simple, and complex SQL queries to gain parallelization behavior insights.

We use TPC-H and TPC-DS workload for SQL query level performance comparison. We observe the TPC-H isolated execution of both the adaptive and the heuristic parallelization shows similar performance. However, adaptive plans are better as they use fewer number of cores, which helps during concurrent workload. Adaptive plans show better performance than the heuristic plans for the TPC-DS workload isolated execution, due to optimal number of partitions, and the presence of the skewed data. In the rest of the section we describe the experimental details.

4.1 Operator level analysis

Adaptive parallelization helps to analyze the role of individual operators in influencing parallelized execution, as it uses expensive operator parallelization as a heuristic. Getting insights into the issues such as the execution skew becomes easier. An operator's execution time varies on the basis of type of computation, data distribution, amount of data being read / written, type of data access (serial / random), and memory hierarchy of the access (cache / main memory / disk IO). We analyze some of these factors next.

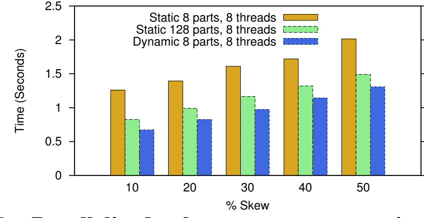


Figure 12: **Parallelized select operator execution on skewed data using static and dynamic (adaptive) sized partitioning. The second bar indicates a work stealing based approach.**

4.1.1 Data skew

This experiment highlights the role of *dynamic* sized partitions to avoid execution skew during parallelized execution, when the data distribution is *non-uniform* (skewed). The execution skew occurs when at least one of the parallelized operators takes longer to execute than the rest.

Static partitioning (equi-range partitioning) of skewed data leads to execution skew as some partitions have more matches than the rest. Adaptive parallelization performs well in skewed data scenario as the operator with the skewed partition turns expensive, and gets parallelized until expensiveness balances out.

Figure 12 shows the execution time when parallel select operators work on statically or dynamically (adaptively) partitioned skewed column of type *long (8 bytes)*. The number of tuples in the input column are 1000 million **(M)** (size = 8GB). Figure 13 shows the column's data distribution with 500 million random tuples in the first half. The second half contains skewed data with 5 sequential clusters of 100 million identical tuples. We vary the select operator's condition to generate the execution skew.

500M Random Numbers	100M Same	100M Same	100M Same	100M Same	100M Same

Figure 13: **Data distribution for a skewed column.**

Figure 12 shows execution with 8 threads on 8 dynamically sized partitions (blue) is up to 60% better than the execution with 8 threads on 8 static partitions (khaki). One may argue that the work stealing approach [5] could solve the problem of execution skew due to the static partitions. We analyze it by creating a large number of smaller partitions (128) operated upon by 8 threads. Large number of smaller partitions allows those threads that finish work early to operate on remaining partitions, while threads on skewed partitions stay busy. Identifying the optimal combination of static partitions and threads is however non trivial, as in some cases more partitions might lead to plan blow-up resulting in scheduling overheads. In contrast we observe that the dynamic sized (adaptive) partitioning approach with 8 threads and just 8 partitions fares competitively with static 8 threads, 128 partitioned approach.

Summary: Skew handling is a natural property of adaptive parallelization. It is a result of dynamically sized range partition creation, and a side effect of the expensive operator parallelization heuristic.

4.1.2 Selectivity, Input size and Exchange union operation

This experiment analyzes the effect of selectivity, input size, and the exchange union operation on the parallelized execution of select and join operators, in terms of their speed-up. *Speedup* is defined as the ratio of serial to parallel plan execution time. The experiment also allows to analyze the speed-up effect when the number of threads varies from 1 to 32. This is possible since with each it-

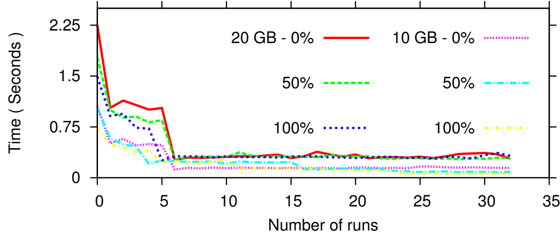


Figure 14: **Effect of variations of data size and selectivity on the speedup of the adaptively parallelized *Select* operator plan.**

Table 2: *Select* operator plan speedup (compared to serial execution) using adaptive and heuristic parallelization.

	AP = Adaptive, HP = Heuristic parallelization					
	Selectivity					
	0%		**50%**		**100%**	
Size (GB)	**AP**	**HP**	**AP**	**HP**	**AP**	**HP**
100	10	10	8.5	10	7	9
20	10.5	12	8.5	12	8	12
10	16	11	14.5	11	12	9.5

eration one more partition gets added and is available for one more execution thread (from a pool of 32 threads) to operate in parallel.

Exchange union operation: Many systems use the *exchange operator* based parallelism [15], where one of the concerns is to identify the correct placement of the exchange operator in a plan to minimize its overhead [2]. Most systems use a cost model based approach for this decision. A good example is [30], where Vectorwise is shown to have a limited speed-up due to the exchange operator overheads.

As the exchange union operator combines parallelized operator's result, its expensiveness varies depending on the size of the data being packed. Low selectivity reflects more matching data, hence more data to be packed. The packing overhead is minimized by pushing the exchange union operator as high as possible. It ensures the final data to be packed is relatively small, as it gets filtered by the intermediate operators.

Adaptive parallelization **(AP)** enables to analyze the exchange union operator's placement with successive iterations of parallelized plans, as it directly affects the speed-up. In the next two experiments we observe that the AP plan's speed-up is comparable to the speed-up of the heuristically parallelized **(HP)** plans. The speed-up gets hindered due to operator dependencies that form critical paths, which can not be parallelized. However, the AP plans benefit by their optimal multi-core utilization due to less partitions, which ensures improved *concurrent* execution performance, as described in Section 4.2.5.

Select operator adaptive parallelization: We use query 6 from the TPC-H benchmark to analyze the speedup of the select operator (See Table 2). Query 6 is a simple query with only selection predicates on the *Lineitem* table. We vary the selectivity by varying the parameter *l_quantity* from the selection predicate. Figure 14 plots the execution time of AP plans on the Y axis with respect to iterations (X axis), when selectivity is varied from 0% (all output) to 100% (no output), and scale factor is varied from 10 GB to 20 GB. We do not plot the graph for 100GB for readability purpose, and only list its speedup in Table 2.

From Table 2 as the selectivity increases the speedup decreases. During low selectivity a single select operator in a serial plan writes a large number of output tuples, as compared to its parallel plan counterparts. This results in the large speedup as serial execution time is much higher, whereas parallel execution time is much lower. During highest selectivity (100%) since there is no output the serial execution is less expensive as compared to 0% selectivity serial execution. This results in lower speedup. The speedup increases with a decrease in the input size. This is a result of lower minimum time

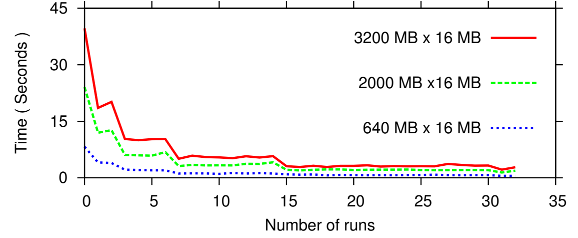


Figure 15: **Effect of variations of data size on the speedup of the adaptively parallelized *Join* operator plan. Outer input partitioned and inner input used to build hash table.**

during parallel execution, due to less input data. With increased selectivity the speedup for AP is less compared to HP. This is due to the presence of less expensive exchange union operators, which do not get pushed higher in the plan.

Table 3: *Join* operator plan speedup (compared to serial execution) using adaptive and heuristic parallelization.

	AP = Adaptive, HP = Heuristic parallelization			
Size (MB)	**64**		**16 (Smaller Input)**	
(Larger Input)	**AP**	**HP**	**AP**	**HP**
3200	15.75	14	18.5	18
2000	15	13.5	17.75	17.75
640	13.75	13	17	15

Join operator adaptive parallelization: For the join operator (hash) plan parallelization analysis, we partition the outer input and build up the hash table on the inner input. We use a micro-benchmark for a fine grained control, where the outer input has 400 M, 250 M, and 80 M (M = Millions) random tuples of type *long (8 bytes)*, and the inner input has 8 M and 2 M tuples. The outer inputs stay larger than the inner input of size 16 MB (2 M tuples) even after 32 partitions (threads on CPU). The 16 MB input fits in the shared L3 cache (20 MB).

Figure 15 shows the join operator plan speedup and Table 3 quantifies it. The speedup of 16 MB input join is more than the 64 MB input join, as the 16 MB input join's hash table fits partially in the L3 cache (20 MB), which improves the probe phase, due to reduced cache thrashing. Speedup also decreases as the outer table size decreases, as the serial execution time is directly proportional to the outer table size. For all sizes the best speedup is obtained when the number of partitions are 32, with 32 threads (hyper-threading enabled). Maximum speed-up observed is around the number of physical cores (16). Both AP and HP show a similar performance unlike the previous select operator plan analysis case, as the join plan contains only join and union operators.

Summary: Adaptive parallelization works for both the select and the join operator and these operators scale linearly with the number of physical cores. Input size, selectivity, and properties such as cache consciousness affects the speedup.

Having analyzed how individual operators affect parallel execution, in the next section we focus on holistic SQL query level analysis, from execution performance and convergence perspective.

4.2 SQL query level analysis

4.2.1 TPC-H queries

Since the TPC-H benchmark is considered the de-facto workload for performance comparison, in this section we use a subset of queries (see Table 4) from TPC-H (scale factor 10). TPC-H has uniformly distributed data. The adaptively parallelized group-by operator implementation at present supports single attribute group-by queries. Hence, we modify some queries so that they have a single attribute group-by representation. Since we use the same set of queries to evaluate multiple parallelization approaches, the

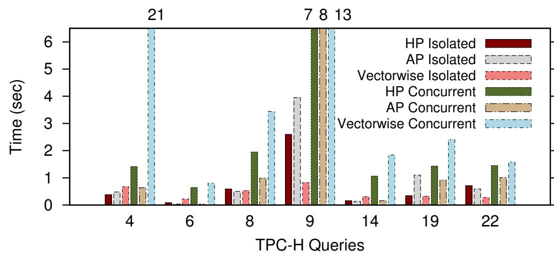


Figure 16: **Heuristic vs adaptive parallelization performance in isolated and concurrent environment for MonetDB and Vectorwise.**

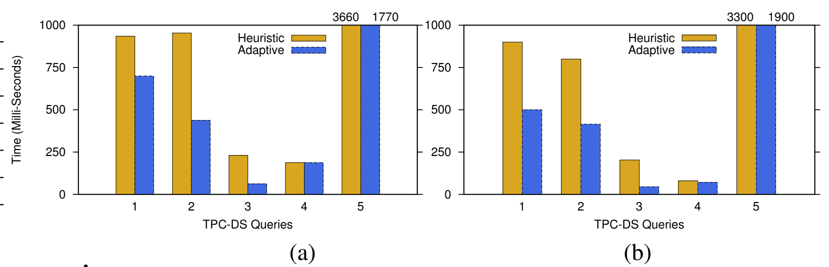


Figure 17: **Isolated execution performance of TPS-DS queries on a) 2 socket machine with 2.00 GHz CPU b) 4 socket machine with 2.40 GHz CPU, on 100GB data.**

comparison is fair. We plot an average of four executions using the experimental set-up described towards beginning of Section 4.

Table 4: TPC-H queries.

Simple	Q6	Q14			
Complex	Q4	Q8	Q9	Q19	Q22

We compare adaptive parallelization **(AP)** with heuristic parallelization **(HP)**, the default parallelization technique in MonetDB, under isolated execution setting. HP uses parameters such as the number of threads, physical memory size, and the largest table size to identify the number of partitions for the largest table in the serial plan. A plan re-writer generates a parallel plan from a serial plan by propagating the partitions to data flow dependent operators. Though both HP and AP start with the same serial plan, the final parallel plan is different for both techniques as in AP only the most expensive operator gets parallelized unlike in HP, where all possible parallelizable operators are parallelized. In Figure 16 the first two bars show HP vs AP performance when queries execute in isolation. All AP queries except Q9 and Q19 show similar performance as HP. Q9 and Q19 show a degraded performance due to the presence of some non-parallelizable operators, which prolong the query execution. The robustness of individual query execution could be observed in Figure 18C, where queries show minimal execution time variations. Though the execution performance of adaptive plans is similar to the heuristic plans, the adaptive plans are better as they use much fewer number of partitions (See Table 5). It helps during concurrent workload execution, where adaptive plans exhibit better execution performance due to better resource utilization.

Table 5: AP and HP Q14 plan statistics.

	AP	HP
# Select operators	10	65
# Join operators	16	32
% Multi-core Utilization	35	75

4.2.2 TPC-DS queries

TPC-DS benchmark has 25 tables, out of which 6 tables are relatively large (above 1GB in size), in a scale factor 100 dataset. The benchmark supports 99 query templates. We use a few modified queries. These queries are a subset of the original TPC-DS queries and are chosen such that they contain the large tables and a few smaller dimension tables. Since we compare both the adaptive and the heuristic parallelization technique with the same queries, the comparison gives a perspective of their respective performance.

We experiment on both the two socket and the four socket machine (See Table 1 for configuration) with 100GB dataset, to get a perspective of the NUMA effects. Graphs in Figures 17a and 17b show the comparison. Adaptive plans exhibit a maximum of 5 times better performance compared to heuristic plans, which could be attributed to *correct partitioning by adaptive parallelization compared to heuristic parallelization and the skewed data distribution.* The execution time for both two and four socket machine shows similar time, which indicates minimal NUMA effects. As authors in [14] observe, since MonetDB uses a memory mapped representation for the buffer data, as the number of partitions increase, we expect them to get assigned to the memory modules of the sockets on which operator execution gets scheduled. We also observe a limit on the execution improvement, even though a higher number of cores are used, which indicates increased parallelism need not improve performance beyond a threshold.

4.2.3 Concurrent workload execution

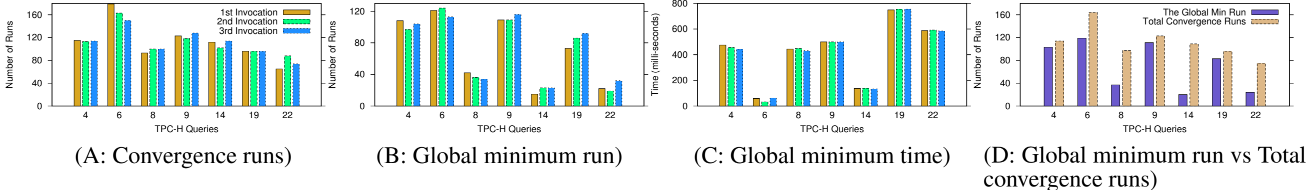
This experiment highlights the effectiveness of adaptively parallelized plans compared to heuristically parallelized plans in a concurrent workload setting. Concurrent query executions in batch workload leads to resource contention, which in turn affects the degree of parallelism of individual queries under execution. Resource contention varies with random workload, however for the current set-up we consider a homogeneous concurrent workload. In Figure 16 the 4th and 5th bar shows HP vs AP execution under a concurrent workload. The workload consists of random simple and complex queries from the TPC-H benchmark (10GB), where 32 clients invoke queries repeatedly. AP Q8 shows 50% improved execution compared to HP Q8. Simple queries such as Q6 and Q14 show around 90% execution improvement in AP. HP plans have too many partitions compared to the AP plans as shown in Table 5. AP plans also reflect the resource contention through execution feedback. Hence, AP plans are more robust and better performing under a concurrent workload, compared to statically generated HP plans. In [11] authors discuss HP vs AP plans comparison under different concurrent workload resource contention scenarios in a detailed manner.

4.2.4 Comparison with Vectorwise

We compare the concurrent workload performance of Vectorwise (version 3.5.1 with histogram build feature enabled to generate optimized plans), a leading analytical columnar database using pipelined vectorized execution [7], with adaptive parallelization in MonetDB. Vectorwise uses cost model based exchange operator dependent parallel plans. The resources are allocated based on the number of connected clients and the system load. During a heavy concurrent workload (32 clients invoking random TPC-H queries repeatedly on 10GB data), the first client's query gets all the resources, while the queries from the remaining clients get less resources based on an admission control scheme. Figure 16 shows MonetDB adaptive parallelized query execution performance is better than the Vectorwise execution performance, during the concurrent workload. MonetDB does not have explicit resource control based plan generation scheme, which helps in the current case. We hypothesize that as workload queries are invoked repeatedly, Vectorwise queries under analysis execute serially due to lack of resources.

4.2.5 Multi-core utilization

This experiment highlights that an AP plan is better than a HP plan from the multi-core utilization perspective. Multi-core utilization represents the fraction of actual CPU cores used versus the available cores during query processing. AP ensures minimal multi-core utilization as each operator is parallelized with a different degree of parallelism unlike HP. Figures 19 and 20 visualize AP vs HP plan execution of TPC-H Q14, in an isolated execution



(A: Convergence runs) (B: Global minimum run) (C: Global minimum time) (D: Global minimum run vs Total convergence runs)

Figure 18: **Convergence algorithm shows minimal variations across multiple invocations in the graphs A, B, and C for adaptively parallelized query execution. Graph D shows most queries converge quickly after detection of the global minimum.**

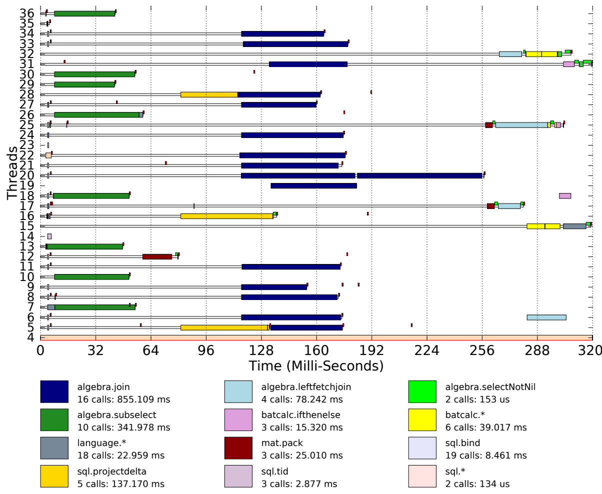


Figure 19: **Adaptive parallelization multi-core utilization (35%) during isolated execution of TPC-H Q14. Green- Select, Blue- Join, Brown- Exchange union operator.**

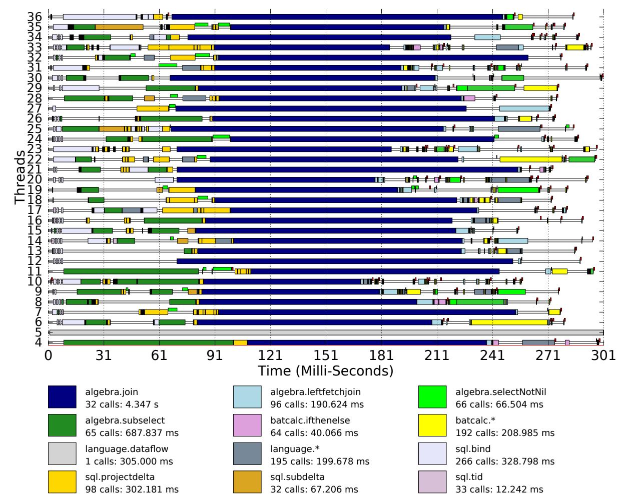


Figure 20: **Heuristic parallelization multi-core utilization (75%) during isolated execution of TPC-H Q14. Green- Select, Blue- Join, Brown- Exchange union operator.**

setting. The length of a colored box represents an operator's execution interval (In an operator-at-a-time execution model an operator executes completely. Blue- *join*, Green-*select*, Brown- *exchange union operator.*) A whitespace indicates no execution. The amount of whitespace in Figure 19 is much more than in Figure 20, indicating lower multi-core utilization for AP. In the HP execution (Figure 20) the length of the join operators is much longer than the corresponding operators in the AP execution (Figure 19), which hints at the memory bandwidth pressure. In [13] authors analyze parallel plans in detail using this visual scheme.

The degree of parallelism per operator thus influences the overall multi-core utilization. For example, while only ten select operators execute in AP, many more execute in HP (See Table 5). Since AP shows lower multi-core utilization (35%) during isolated execution, the spare resources ensure better response time during concurrent workload, as further elaborated in [11].

4.3 Convergence algorithm robustness

Adaptive parallelization not only should converge in minimal number of runs, but also should exhibit robustness. The robustness implies during *multiple* adaptive parallelization invocations of a query a) the total number of convergence runs b) the run at which the global minimum occurs, and c) the global minimum execution time should not show much variations. In this experiment we test the robustness of the convergence algorithm in an isolated execution setting. Graphs in Figure 18 (A,B,C) show these three cases.

Graph 18(A) shows the number of convergence runs to find the optimal execution time for three invocations. Except for Q6 and Q22 all other queries show minimal variations for convergence runs. Q6 is the most simple query in the given set of queries. It shows the most speed-up amongst all queries, but that also makes it vulnerable to external factors such as operating system noise interference, etc. Since the global minimum time is very low, even small interference affects its performance. Q22 is a complex query where join operator is always the most expensive operator.

Graph 18(B) shows the run where the global minimum time occurs during three invocations of the query set under evaluation. Depending on the resource contention and the run-time interference from the operating system we get small variations across different runs for all queries. The highest difference is observed between the first and the third run for Q19. However, overall the number of runs do not show much deviations.

Graph 18(C) shows the global minimum time for adaptively parallelized queries for three invocations. The global minimum time for all queries is almost stable across multiple invocations. This indicates the robustness of the generated plans.

Graph 18(D) shows that the queries 4, 6, 9, 19 converge quickly after detection of the global minimum. Different types of queries show different convergence properties and the algorithm gets tuned to converge in the least possible number of runs. For example, Q8, Q14, and Q22 show the global minimum with fewer than 40 runs, while the total convergence runs are around 100. The slow convergence is a result of the *Leaking_Debit* being too low, which leads to the *credit* getting drained slowly. The convergence runs are close to 60 for the same global minimum, when the *Leaking_Debit* is high.

How to lower number of convergence runs? At present plan parallelization introduces only a single new operator per invocation, which results into a higher number of convergence runs, as the execution skew introduced by a single new operator needs to level out. The number of runs could be made much lower if more and even number of operators are introduced per invocation. We avoid it at present to analyze the parallel plan evolution with each new operator addition.

5. RELATED WORK AND APPLICABILITY TO OTHER SYSTEMS

The basic optimizer approach of "optimize once and execute many" as proposed by System R has reached its limits [28]. Hence, adaptive query processing techniques are being proposed to address query optimization problems due to unreliable cardinality estimates, data skew, parameterized query execution, changing workload, complex queries with many tables, etc. [9]. In this section we describe some state of the art adaptive techniques.

Adaptive aggregation is used by the authors in [8] to handle different group-by based parallelization cases. The operator performs a lightweight sampling of the input to choose the best aggregation strategy with high accuracy, at runtime. Algorithmic approaches are based on using independent and shared hash tables with locking and atomic primitives to minimize hash table access contention. Three cases are identified that affect performance based on the average run-length of identical group-by values, locality of references to the aggregation hash table, and frequency of repeated access to the same hash table location. This work targets adaptivity from a single operator's perspective, whereas our work targets it at the plan level. The approach used here could be combined with our adaptive approach to improve per operator performance.

Vectorwise uses micro-adaptivity to improve query execution time by using run-time execution feedback [25]. Micro-adaptivity is defined as the ability to choose the most promising execution primitive at run-time, based on real time statistics. Most methods, like adaptive parallelization, use plan level modification, whereas micro-adaptivity uses the available execution primitives at run time. It chooses primitives based on the platform, instance, and call adaptivity using parameters such as compiler, branch prediction, selectivity, loop unrolling, etc.

The Learning Optimizer (LEO) in **DB2** uses query execution feedback for cardinality estimation corrections [29]. It uses learning and feedback based infrastructure to monitor query execution and generates feedback for correction in the cardinality estimation and related statistics. More learning helps in better cost model predictions. LEO has improved query execution performance by orders of magnitude. MonetDB does not use cardinality related statistics, however if used with the statistics correction methods, the selection of operator's to parallelize could be improved further.

In [4] authors illustrate adaptive parallel execution in Oracle for big data analytics. In Oracle problems such as reliance on query optimizer estimates are handled by changing the data distribution decisions adaptively, during query execution.

Column store architectures differ in various aspects such as plan representation, partitioning strategy, scale out support, etc. Encompassing all the requirements in a single architecture is not possible due to their architectural confinements. While describing the related state of the art column stores, we also describe the possibility of adaptive parallelization's application to them.

Vertica uses a value based partitioning approach [18]. It uses a Read Optimized Store (ROS), where the data is stored in multiple ROS containers on a standard file system. Two files per column within a ROS container are stored, one with the actual column data and the other with position index. This representation is very similar to the representation in Figure 10, where RH is the index, while RT is the actual value. Vertica also supports grouping multiple columns together in a file, however this hybrid row-column storage is rarely used in practice because of the performance and compression penalty it incurs.

Vertica execution engine uses a multi-threaded pipelined vectorized execution where the execution plan consists of standard relational algebra operators. Operators such as StorageUnion are used for partitioning data across operators. Hence, StorageUnion is equivalent to a partition operator. Operators such as ParallelUnion are used for directing execution to multiple threads and to combine the parallelized operator's results. Hence, ParallelUnion is equivalent to an exchange union operator.

To understand the feasibility of applying adaptive parallelization in Vertica, let's assume that the execution starts with a serial plan and incrementally introduces value based partitions to partition the expensive operator's data. For example, when a select operator becomes expensive and needs to be parallelized, a partition operator is introduced which creates two value based partitions, which would be consumed by two new select operators. The two new select operator's output is combined using an exchange union operator. This is similar to the basic mutation scheme with the addition of a partition operator that feeds two newly introduced select operators.

When one of the newly introduced select operators itself becomes expensive, we further partition that operator's data into two more value based partitions, by introducing a new partition operator in the data flow path after the previous partition operator, and before the input of the expensive select operator. Thus in a hypothetical case when one of the select operators stays expensive during consecutive invocations, new partition operators would keep on getting added to the existing plan. We expect the cost of the partitioning operator to be small considering its presence in the existing Vertica execution plans. Quantifying the exact cost is difficult due to lack of sufficient references. Similar logic could be applied for other operator's parallelization.

Apollo creates column store indexes in a traditional row store database like SQL server [20]. It is the first database which uses the existing row store to create new column store indexes. The method involves creation of batches of rows to create segments from which individual columns are stored in individual column representations. The column segments information is stored in the directory structure, with a catalog.

The columns are compressed and encoded using different types of encoding. New operators called batch operators are introduced which get called if there is bulk data to be processed. The valid rows to process are noted down in bitvector formats.

Apollo uses range partitioning of data. Since traditional SQL server uses cost model based exchange operator induced parallelization, Apollo leverages the existing SQL server parallelization technique using the exchange operator based parallelization.

To understand how adaptive parallelization might be applied in Apollo type of column store, we need to find similarities between adaptive parallelization and Apollo architecture. Both do range partitioning of data, hence the fundamental assumption of range partitioned access stays the same, and could change in the way individual operators are implemented. For example, the operations like the join operator consists of separate build and probe operators, where build uses a shared hash table, where all threads build a hash table, and then probe operator probes it in parallel. As Apollo extends the exchange operator based parallelization as used in SQL server, we expect adaptive parallelization to be useful, due to its dependence on the exchange operator based parallelization.

Hyper uses LLVM [21] generated Just In Time (JIT) compiled plans. The longest pipeline in a plan is identified, by looking for a pipeline breaker operator. The operators in the longest pipeline are fused using JIT compilation such that their highly efficient machine language code represents a single task. The fusing allows tuples to be kept in registers to process them without generating intermedi-

ate results. Hyper's morsel driven parallelism uses work stealing based approach to assign the fused pipeline tasks to a fixed number of pre-created threads. The task allocation based approach allows controlling the number of tasks executing in parallel dynamically, at run-time, and allows better control over resource allocation during concurrent execution of queries.

Adaptive parallelization technique is based on the fundamental assumption that an expensive operator is always identifiable in an execution plan. This is a basic requirement since plan parallelization is a result of incrementally parallelizing the expensive operator during successive query invocations, until a global minimum plan is identified.

Identification of a single expensive operator is not feasible in Hyper's execution plans due to the JIT compiled fused nature of operator's pipeline, which prevents a direct application of adaptive parallelization. However, in a broader sense if the entire task is considered to be expensive and treated as an expensive operator, application of the adaptive parallelization logic could be possible. Hence, the feasibility of adaptive parallelization technique in Hyper depends on how to categorize the expensiveness metric.

DB2 BLU accelerator [24] uses evaluator chains, which comprises DB2 BLU operators working on columnar data. The data is accessed in strides. It uses novel data structures that minimize latching allowing seamless scaling with multi-cores. Parallelism involves cloning of evaluator chains once per thread, where the number of threads is decided by cardinality estimates, system resources and system load. Each thread requests strides for its evaluator chains until no more strides are available. DB2 BLU also uses work stealing based approach where worker threads operate on tasks comprising of evaluator chain based work.

6. CONCLUSION

Adaptive parallelization uses query execution feedback to generate resource contention and skew aware range partitioned multi-core parallel plans. It helps in finding the right balance between the multi-core utilization and the degree of parallelism for the exchange operator based parallel plans. We observe a near linear speedup with the number of cores while analyzing the parallel plan evolution using parameters such as the input size and selectivity. During TPC-DS isolated workload execution, the adaptively parallelized plans show up to five times better performance compared to heuristically parallelized plans. During TPC-H concurrent workload, they show minimal multi-core utilization, allowing better resource utilization. They also fare competitively with work stealing based scheduling approach.

Using different convergence scenarios we show that the adaptive parallelization convergence algorithm behaves robustly, and converges in a reasonable number of runs.

7. ACKNOWLEDGEMENT

We thank the MonetDB team for their support. This project is funded through COMMIT grant WP-02.

8. REFERENCES

[1] Sql server parallelization. "`http://www.simple-talk.com/sql/learn-sql-server/understanding-and-using-parallelism-in-sql-server/`".

[2] K. Anikiej. Multi-core parallelization of vectorized query execution. http://homepages.cwi.nl/ boncz/msc/2010-KamilAnikijej.pdf.

[3] R. Appuswamy, C. Gkantsidis, et al. Nobody ever got fired for buying a cluster. Technical report, Microsoft Technical Report MSR-TR-2013, 2013.

[4] S. Bellamkonda et al. Adaptive and big data scale parallel execution in oracle. *Proc of VLDB*, 6(11):1102–1113, 2013.

[5] R. D. Blumofe et al. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[6] P. Boncz and M. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[8] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *Proc of VLDB*, pages 339–350. VLDB Endowment, 2007.

[9] A. Deshpande et al. Adaptive query processing: why, how, when, what next? In *Proc of VLDB*, pages 1426–1427, 2007.

[10] S. Ganguly et al. Query optimization for parallel execution. In *ACM SIGMOD Record*, volume 21, pages 9–18, 1992.

[11] M. Gawade and M. Kersten. Multi-core query parallelism under resource contention - under review vldb 2016. https://sites.google.com/site/confproceed/AdParConcur.pdf.

[12] M. Gawade and M. Kersten. Stethoscope: a platform for interactive visual analysis of query execution plans. *Proc of VLDB*, 5:1926–1929, 2012.

[13] M. Gawade and M. Kersten. Tomograph: Highlighting query parallelism in a multi-core system. In *Proc of DBTest*, page 3. ACM, 2013.

[14] M. Gawade and M. Kersten. Numa obliviousness through memory mapping. In *Proc of DAMON*, page 7, 2015.

[15] G. Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. ACM, 1990.

[16] W. Hong. *Parallel query processing using shared memory multiprocessors*. PhD thesis, UC, Berkeley, 1992.

[17] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proc of SIGMOD*, pages 297–308. ACM, 2009.

[18] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc of VLDB*, 5(12):1790–1801, 2012.

[19] R. S. Lanzelotte et al. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, volume 93, pages 493–504, 1993.

[20] P.-Å. Larson et al. Sql server column store indexes. In *Proc of Sigmod*, pages 1177–1184, 2011.

[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[22] Z. Majo et al. Memory system performance in a numa multicore multiprocessor. In *Proc of ICSS*, page 12, 2011.

[23] C. Mohan. Impact of recent hardware and software trends on high performance transaction processing and analytics. In *Proc of PEMCCS*, pages 85–92. Springer, 2011.

[24] V. Raman et al. Db2 with blu acceleration: So much more than just a column store. *Proc of VLDB*, pages 1080–1091, 2013.

[25] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proc of SIGMOD*, pages 1231–1242, 2013.

[26] R. A. Rutenbar. Simulated annealing algorithms: An overview. *Circuits and Devices, IEEE*, 5(1):19–26, 1989.

[27] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *Business Intelligence*, pages 125–149. Springer, 2013.

[28] P. G. Selinger, M. M. Astrahan, Chamberlin, et al. Access path selection in a relational database management system. In *Proc of SIGMOD*, pages 23–34, 1979.

[29] M. Stillger, G. M. Lohman, V. Markl, et al. Leo-db2's learning optimizer. In *Proc of VLDB*, pages 19–28, 2001.

[30] L. Viktor et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proc of SIGMOD*, 2014.