Query Processing on Multi-Core Architectures

Frank Huber and Johann-Christoph Freytag
Department for Computer Science, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
{huber,freytag}@dbis.informatik.hu-berlin.de

Abstract

The upcoming generation of computer hardware poses several new challenges for database developers and engineers. Software in general and database management systems (DBMSs) in particular will no longer benefit from performance gains of future hardware due to increase clock speed, as it was the case for the last 35 years; instead, the number of cores per CPU will increase steadily. Today's approach is to run each query on a single core or only a few different cores using parallel query execution. This approach suffers from several problems (e.g. contention problem) and therefore leads to poor speed up and scale up behavior. These observations open several important research questions on how to use the new multi-core CPU architecture for improving the overall performance of DBMSs.

This paper outlines our approach for query processing on multi-core CPU architectures. We present an abstract architecture view for multi-core CPUs, meta operators to control and to interact with the hardware, and a new query operator model that makes use of the meta operators to control the parallel execution of a query over different cores. We illustrate how each of these parts fits in our framework for query processing on multi-core architectures.

1 Introduction

The next generation of computer hardware poses several new challenges for software developers and engineers. Due to physical limitation, next generation CPUs will no longer scale in clock rate. Instead, a scale in number of cores within a single CPU is most likely. The extended address scheme of 64bits allows us to use large main memory installations even for low budget systems. With almost infinite memory, we can assume that a majority of databases fit entirely into main memory. Hence, disk - main memory latency and bandwidth will become less important factors. Instead, main memory - CPU latency mismatch and limited bandwidth form the "memory wall". Hence, parallelization and cache efficiency are major objectives.

Parallelization of query execution can be viewed in three dimensions [1]. The first dimension, *interquery parallelism*, determines the number of queries that can be executed concurrently. The second dimension, *interoperator parallelism*, defines the number of operators in a single query executed in parallel. The last dimension, *intraoperator parallelism*, indicates to what extend a single operator is parallelized.

Most of today's DBMSs make use of the first dimension and a specialized version of the second - *horizontal interoperator parallelism*[1]. Furthermore, queries are currently optimized to minimize disk main memory transfers. These limitations and the increasing number of available cores on a single chip lead to serious problems. E.g. running hundreds of queries in parallel will result in different contention problems. The first problem relates *data contention*, i.e. different queries require access to the same object at the same time. Even without data contention, stagnancy in CPU clock rates will result in poor speed up performance. *Resource contention* is the second problem: different queries utilize the same hardware, especially CPU - main memory

[1] Exchange Operator[2]

bandwidth and CPU caches. If, for example, two queries are executed on two cores that share the same physical cache, both cores will try to prefetch necessary data thereby causing thrashing of the cache most likely. Such behavior results in additional data loads (i.e. CPU stalls, waste of bandwidth). In summary, today's approach may results in faster query execution on single core CPUs, but not on multi-core CPUs.

2 A Framework for Query Processing on Multi-Cores

The new generation of hardware technology raises two main research questions. First, how do we handle the memory wall? Second, how do we use the new opportunities that come with multi-core architectures?

Concepts like vertical fragmentation [3, 4] seem to be a good choice when it comes to attack the memory wall. In our opinion the open challenge is, how to relate these new concepts to query processing highly parallelized for CMPs[2]. For this reason, we have developed a framework that simplifies the development and the maintenance of DBMSs on multi-core architectures. Among others, our work covers the following issues:

- Integration of multi-core in the whole process of query processing;
- Finding parallelisms in query optimization;
- The influence of multi-core on cost estimation and plan selection;
- The choice of execution model that best fits query execution on CMPs;
- Ease of development and maintenance of DBMSs on different multi-core architectures;
- The design and implementation of a global optimizer for such systems;
- Is there an optimal CMP architecture with respect to DBMSs? Which functionality should be provided by such a chip?

Regarding the complete flow of query processing, we focus on the two phases (i) query optimization and (ii) query execution. Both phases are reflected in our architecture framework that has the following layout: The bottom layer consists of an abstract view of the hardware architecture and a physical data model. On top, the next level comprises of meta operators to control the hardware by using the abstract view. A model of query execution and query optimization form the upper level of our framework. Query optimization employs the abstract view to create optimized QEPs from a global point of view as well as from a local point of view. Query Execution utilizes the meta operators to describe and to implement its operators over the physical data model. In the following we briefly outline the physical data model and detail the abstract architecture view, meta operators, and query execution.

Physical Data Model

On multi-core architectures, we must consider the physical (memory) layout of the data very carefully for efficient query execution. In order to provide high inter- and intraoperator parallelism the physical data model provides data structures that are cache efficient and optimal for partitioning. Since vertical fragmentation of the data has been shown to be a good choice for cache efficiency, we will use this representation in the following. Like C-Store [4], we use bitmap indices to represent sets of ids.

[2] chip multiprocessor

Abstract Multi-Core Architecture View

Highly parallelized query execution depends very much on the actual hardware architecture (i.e. what kinds of functions are supported? How are cores interconnected? etc.). To be independent from specific architectures and for simplicity, we use an abstract multi-core architecture view. This view focuses on the common and important aspects of the different architectures, thus hiding some of the complexities of modern CPU architectures. Our foci are caches and cores. Therefore, our view contains these two hardware components. In addition, we add a logical (non-existent) component to group different cores and caches. We call this logical component a **workgroup**. During query execution an operator might be distributed over one or more workgroups. Different setups are conceivable. First, each core in a workgroup executes the same operation on different data. Second, a complex operation is decomposed into N smaller and simpler, operational fragments. Each of these N fragments is executed by one core of a workgroup of size N, thus forming a worker queue that feeds each other. Lastly, a combination of the previous two alternatives is possible, thus forming a hybrid combination. Furthermore, components - in particular cores - might be shared among different workgroups. In particular on SMT[3] systems, cores might be logically shared among different workgroups. We emphasize that workgroups do not have to be equally equipped, e.g. it is possible to create workgroups with a single core and without any assignment of cache.

Meta Operators

We define a number of meta operators for the following reasons. First, most modern programming languages do not provide any support for expressing cache or multi-core related issues. For example, in *C* or *C++* developers have to use intrinsic or inline assembler statements for this reason. But none of those statements is hardware independent, thereby leading to a high degree of CPU vendor / CPU architecture dependency and huge efforts in software development. Second, our goal is to separate the actual hardware implementation and optimization of operators from their development. Third, with our meta operators, we would like to give developers the ability to express all their knowledge (e.g. data access pattern) in a simple and comprehensive manner.

For example, consider a developer of a DBMS writing a new join operator that might be executed in parallel on two cores and needs some data exchange during its processing. The developer might use the meta operator *TransferData* to initiate a data transfer from one core to another one. The actual implementation of this operator is very system specific: A hardware expert will implement and optimize those meta operators for each individual hardware architecture.

Our abstract (hardware) architecture consists of meta operators controlling the behavior of cores, caches, and workgroups. We designed operators for resource management like allocation and deallocation of workgroups, for communication and data transfers between workgroups and cores, as well as for work scheduling. Our meta operators and abstract view consider cores and caches as regular system resources that might be allocated and deallocated. Allocated resources may not be shared by different system processes. This restriction assures that it is always known in which state each resource is at any give execution point.

To provide the reader with a flavor of the semantics and possible implementation of meta operators, we depict three of them in the following:

- $AllocWorkGroup(C, S)$ allocates a workgroup of C cores and a cache size of S and returns some kind of handle. This handle can be used to manage and to interact with the workgroup.
- $AssignTaskToWorkGroup(WG, T)$ assigns a task T to the workgroup with handle WG. Tasks are code pieces specific for the workgroup setup.

[3] simultaneous multithreading

- $TransferData(WG_1, WG_2)$ transfers data from workgroup WG_1 to workgroup WG_2. One implementation could be to use an interconnection bus if available. Alternatively, one could write the data back into main memory and the load the data into a different cache, or just do not perform any operation if WG_1 and WG_2 sharing the same physical cache.

Query Execution and Operator Model

With the abstract view and our meta operators in hand, we are able to describe the behavior of different multi-core CPU architectures in a simple manner. One of our questions was to identify an execution model that fits best for query execution on CMPs. It is our belief the iterator model as the operator model for query execution is inappropriate for query execution on multi-core architectures. The main reason for this opinion is that by its nature the iterator model is highly synchronized i.e. only one operator is executed at any time[4]. This behavior results in no vertical parallelism. In particular, vertical parallelism is very promising for CMP systems as it provides an opportunity to create query operators that run in parallel and that feed each other with data (asynchronously). This kind of parallelism might greatly utilize the sharing of caches (i.e. the output of one operator is the input another one) thus reducing CPU main memory bandwidth utilization. This reduction in bandwidth might be used to increase the degree of horizontal parallelism, as the degree of horizontal parallelism is bounded by the available bandwidth.

In the following, we outline some ideas for designing a new **asynchronous** operator model that consists of two kinds of operators. The first kind is a set of already known operators like join, selection, aggregation, projection, and grouping. Each of these operators is implemented according to our physical data model and uses different meta operators. The operator model uses pipes and buffers for data exchange i.e. the producing operator is responsible for transferring data to all its consumers. The second set currently consists of one special operator, named **watchdog**. This operator is responsible for managing the execution of a query by processing the QEP bottom-up. It is the only instance that knows about the current execution state. The watchdog uses the abstract architecture view to distribute the QEP and its operators among the different existing workgroups. The workgroups are managed and controlled by the meta operators. A workgroup regularly sends signals to the watchdog to give information about its current status. To make the query execution and the interaction of operators more concrete we give the following example.

Example

The example is the simplified Query 1 of the TPC-H benchmark [5]. Figure 1 depicts the query

```
SELECT l_returnflag ,
    l_linestatus ,
   SUM( l_quantity )
 FROM lineitem
 GROUP BY
     l_returnflag ,
     l_linestatus
```

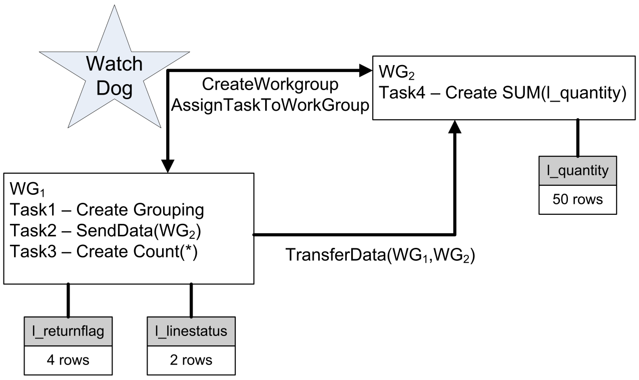

Figure 1: Query Execution of the example

[4] beside horizontal parallelism

execution of the sample query. In our case the watchdog creates Workgroup WG_1 and assigns Task 1 *CreateGrouping*. The *CreateGrouping* task creates all possible groupings using the columns $l\_returnflag$ and $l\_linestatus$. A possible implementation could be to create the workgroup WG_1 with 4 cores. Then, each core loads one attribute value[5] from attribute $l\_returnflag$ and intersects it with each attribute value of $l\_linestatus$. This execution ensures that all bitmap indices of $l\_returnflag$ and $l\_linestatus$ must only be loaded once. The result of this task are 4 different non empty groupings on $l\_returnflag$ and $l\_linestatus$. Meanwhile the watchdog allocates Workgroup WG_2 with an appropriate number of cores and an appropriate cache of a predetermined size. Furthermore, the watchdog assigns the second (*SendData*(WG_2)) and third Task (*CreateCount*(*)) to WG_1 and the fourth Task *CreateSum*(*l_quantity*) to WG_2. The task *SendData* uses the $TransferData$ meta operator to send all previously calculated results to WG_2. Finally, each core of WG_1 counts the bitmap index of one group to calculate the aggregation *Count*(*) as requested by Task 3. After WG_1 has send its data to WG_2 and notified WG_2 that now all data is available, WG_2 starts executing Task 4, by scanning through column *l_quantity* and calculating the intersection according to each group. At the end, the results of both workgroups must be joined to form the final result. This latter part is not included in Figure 1.

3 Conclusion and Outlook

In this paper, we presented our ideas for a new query optimization and query execution model that takes advantage of multi-core CPU architectures. We outlined our underlying physical data model and introduced an abstract view of multi-core architecture together with meta operators. Both, the abstract view and meta operators will hide some heterogeneity and complexity of different hardware architectures. Furthermore, we presented a new operator model for query execution. We believe that our model supports intraoperator parallelism as well as interoperator parallelism (both horizontal and vertical). Lastly, we illustrated our framework on an examples, to provide a basic understanding of, how queries are executed using our framework and our query execution model.

The next steps of our work are a more complete design and implementation of our concepts. The implementation should provide useful feedback about our operator model. Although our current work focuses on query execution new approaches for query optimization in a multi-core execution environment are equally important; they will be part of our future work.

References

[1] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[2] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[3] M. L. Kersten, S. Manegold, P. A. Boncz, and N. Nes. Macro- and micro-parallelism in a dbms. In *Euro-Par*, pages 6–15, 2001.

[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[5] Transaction Processing Performance Council. TPC Benchmark^{TM} H (decission support), 1999.

[5] value and bitmap index