Extending Database Task Schedulers for Multi-threaded Application Code

Florian Wolf
TU Ilmenau, SAP SE
florian.wolf@tu-ilmenau.de

Iraklis Psaroudakis
EPFL, SAP SE
iraklis.psaroudakis@epfl.ch

Norman May
SAP SE
norman.may@sap.com

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

Kai-Uwe Sattler
TU Ilmenau
kus@tu-ilmenau.de

ABSTRACT

Modern databases can run application logic defined in stored procedures inside the database server to improve application speed. The SQL standard specifies how to call external stored routines implemented in programming languages, such as C, C++, or JAVA, to complement declarative SQL-based application logic. This is beneficial for scientific and analytical algorithms because they are usually too complex to be implemented entirely in SQL. At the same time, database applications like matrix calculations or data mining algorithms benefit from multi-threading to parallelize compute-intensive operations. Multi-threaded application code, however, introduces a resource competition between the threads of applications and the threads of the database task scheduler. In this paper, we show that multi-threaded application code can render the database's workload scheduling ineffective and decrease the core throughput of the database by up to 50%. We present a general approach to address this issue by integrating shared memory programming solutions into the task schedulers of databases. In particular, we describe the integration of OpenMP into databases. We implement and evaluate our approach using SAP HANA. Our experiments show that our integration does not introduce overhead, and can improve the throughput of core database operations by up to 15%.

Keywords

Databases, OpenMP, Multi-threading, User Defined Functions, Stored Procedures, Workload Management

1. INTRODUCTION

Throughput and response time of data-intensive applications can be significantly improved by running application logic inside the database server. This can avoid the transportation of raw data between the database and the application, and potentially avoid multiple round trips. The most impressive recent use case for application logic in the database is the Proteomics DB [23] project, which has identified human proteome and made it available for the scientific community. It is based on the SAP HANA database and contains domain-specific algorithms that have immediate access to the large data set, which enables an efficient analysis. Another use case are statistical methods such as time series forecasting [9], which enable the forecasting of, e.g., sales, capacity of renewable energy sources, and traffic, directly in the database. A third use case are sparse matrix operations [14] that are used for database applications in, e.g., nuclear physics, genome analysis, and graph algorithms. All these scenarios have in common that they are based on database management system (DBMS) extensions for executing compute-intensive application logic. Furthermore, they all benefit from multi-threaded code. Proteomics DB for instance, uses the IMSL (International Mathematical and Statistical Library) as statistic package, and IMSL is highly parallelized using OpenMP [6]. Also, statistical methods and linear algebra are supported by IMSL and therefore profit from multi-threading.

DBMS offer proprietary or standardized extensions to evaluate application logic as close as possible to the data. They support *stored routines*, such as stored procedures and stored functions, as specified in the SQL Persistent Stored Modules (SQL/PSM) extension [7]. As many scientific and analytical algorithms are typically too complex to be implemented exclusively in declarative, set-orientated and script-style languages, SQL/PSM and its proprietary equivalents contain *external routines*. These are stored routines implemented in programming languages such as C, C++ or JAVA.

Apart from their complexity, scientific and analytical algorithms typically benefit from parallelism as already mentioned above. In modern DBMS, parallelism is not achieved through straight-forward thread creation. DBMS, such as SAP HANA and IBM DB2 BLU, employ their own *task scheduler* with task queues and worker threads [18, 20]. External routines, however, do not have access to the task scheduler of the DBMS. Directly creating threads or using *shared memory programming* solutions, such as OpenMP or Intel Thread Building Blocks (TBB) [15], in application code is possible. Application threads, however, are managed by the operating system (OS) and are independent from the task scheduler of the DBMS. Furthermore, *shared memory programming* solutions assume exclusive access to the resources of a server and attempt to utilize all available H/W contexts. Consequently, threads created either manually,

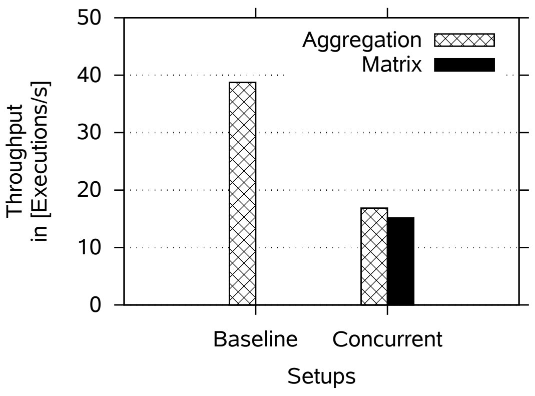


Figure 1: The concurrent execution of multi-threaded application workloads severely hurts the throughput of core database workloads.

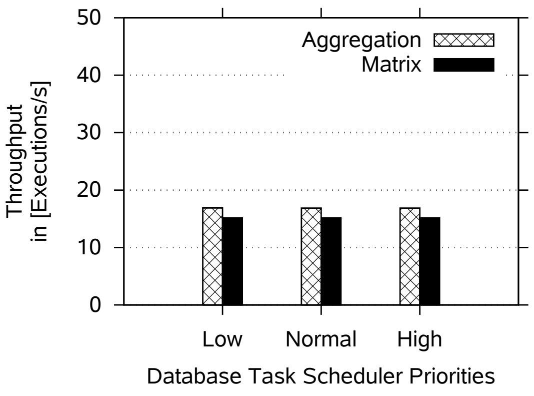


Figure 2: The workload management features of the DBMS task scheduler are ineffective for concurrent multi-threaded application workloads.

via OpenMP, or Intel TBB running concurrently with the DBMS threads may result in a significantly higher number of active threads than H/W contexts. In such a case, the OS schedules the usage of H/W contexts among the database and the application threads fairly, using time slices. Thus, the throughput of the core database operations is hurt, in an uncontrollable way, by the parallelism in application code.

In Figure 1, we illustrate the contention between concurrent database and multi-threaded application workloads (see Section 6 for the experimental configuration). We simulate the core database workload with hash-based `SUM` aggregations, executed with the task scheduler of the DBMS. In the same process as the DBMS, we execute a multi-threaded application workload consisting of matrix-vector multiplications implemented with OpenMP. The throughput of the two workloads reveals that the application workload can decrease the core database throughput by more than 50%, mainly due to the concurrent application threads consuming CPU resources. Furthermore, in Figure 2, we show that the workload management features of the DBMS task scheduler, consisting of different prioritization levels, have no effect on the throughput of the application workload. Since the threads created and used by the OpenMP library bypass the database task scheduler, the prioritization is ineffective.

In this paper we propose to extend database task schedulers with the interface of a shared memory programming solution, which is available in external routines. Although we focus on OpenMP, our conceptual approach is realizable for other shared memory programming solutions as well, such as Intel TBB. The core idea of our concept is to execute database and multi-threaded application workloads with a single task scheduler. External routines can be parallelized with, e.g., OpenMP and achieve maximum parallelism on the server. At the same time, the DBMS can fully control CPU resources and prioritize tasks effectively in case of a concurrent core database workload. Another benefit is that the DBMS can additionally use e.g. OpenMP to parallelize its operations, which can offer increased productivity and shorter learning curves for developers.

While we have implemented this approach in SAP HANA using OpenMP to express parallelism in external routines, the concept also applies to other databases. Our approach is based on the following assumptions for the DBMS task scheduler: (a) it supports shared memory parallelism, (b) it offers an interface for executing tasks with arbitrary code, e.g., by passing a function pointer or overriding a `run` method, and (c) it offers synchronization primitives, e.g., barriers, to avoid synchronization issues (see Section 5). Additionally, our approach supports workload management features, e.g. prioritization, if it is offered by the DBMS task scheduler.

Scope. Our work targets the scheduling of CPU resources among core database operations and multi-threaded application code, by supporting, as a first step, the features of the OpenMP standard inside the database task scheduler. Features beyond the standard, e.g., NUMA-awareness, or cross-application prioritization, or support for a distributed setup, are not considered in this work.

Contributions. The main contributions of our paper are:

- We show that naively executing multi-threaded application code can render the workload management of the DBMS ineffective and decrease the throughput of concurrent core database workloads by up to 50%.
- We describe an approach to integrate shared memory programming solutions in the DBMS. Specifically, we show how the OpenMP runtime can be integrated. Our experiments with SAP HANA show that our integration does not incur any overhead.
- We show how the workload management of the DBMS can be applied to multi-threaded application workloads within the database server process as well. Our experiments show that task prioritization can improve the throughput of the concurrent core database workload by up to 15%.

Paper outline. In Section 2, we review related work. In Section 3, we present our general approach for extending DBMS with shared memory parallelism solutions for applications. In Section 4, we provide an overview of OpenMP. In Section 5, we detail our implementation of integrating OpenMP in the DBMS. In Section 6, we show our experimental evaluation. Finally, we conclude in Section 7.

2. RELATED WORK

In this section, we review related work on shared memory programming, application logic inside the DBMS, and task scheduling in the execution layer of the DBMS.

Shared memory programming. One of the most important topics in parallel computing is the scalability of software on parallel hardware. For a long time message passing, as provided by MPI [21] and Erlang, seemed to be the only option for scalable software. When multi-core architectures with cache coherence protocols emerged, shared memory programming was introduced. It is based on multi-threading and communication via the process memory. Shared memory programming is less complex and more efficient compared to process-based parallelism such as MPI, due to shared resources, faster context switches, and less communication overhead. Overall, there are three different concepts for shared memory programming: (a) languages with thread support like Java or C#, (b) language extensions such as OpenMP [6] or Intel Cilk Plus, and (c) threading libraries. Such libraries include Intel Threading Building Blocks (TBB) [15], Intel Array Building Blocks (ArBB), Microsoft Parallel Patterns Library (PPL), PThreads, Win32-Threads, Boost C++ Libraries and C++ 11 Threads. Major obstacles for the success of shared memory programming were threading libraries with different APIs that suffered from missing portability and few options for data parallelism. This was addressed by OpenMP [6], a standardized API for scalable shared memory programming on multi-core architectures. About 2005, when the processor clock rate reached its technical limit, the "free lunch" for increasing the processor speed was over [22], and shared memory programming became the most promising option to further improve application speed on large multi-core systems.

Application logic in the DBMS. Running application logic inside the database system is realized by stored routines, e.g., stored procedures and stored functions. Historically they were introduced to extend database functionality [17] and relieve the client machines and network connections [12]. Proprietary solutions for stored routines such as PL/SQL by Oracle and Transact-SQL by Sybase motivated the ISO/IEC panel to publish a corresponding SQL extension called SQL/PSM (Persistent Stored Modules) [7]. Both SQL/PSM and the proprietary solutions support external routines which are written in third-generation languages (3GL) such as C, C++, or Java. While stored procedures focus on processing bulks of data using SQL statements, external routines also excel at CPU-intensive algorithms. Today, all major DBMS, e.g., SAP HANA, IBM DB2, MS SQL Server, and Oracle, support either SQL/PSM or proprietary variants to support external routines inside the database server. As an example, we consider SAP HANA [8] in this paper to integrate the scheduling of multi-threaded code of external routines with the task scheduler of the database. Besides standard SQL, SAP HANA offers a rich set of interfaces to realize database applications, e.g., the stored procedure dialect SqlScript [2], Application Function Libraries [1], or the statistical package R [11] to implement scientific or enterprise applications. Several application libraries use multi-threaded libraries like IMSL – which in turn use OpenMP internally, and thus create the challenges outlined in the introduction.

There are rather few publications that focus on the efficient parallel execution of user-defined code. According to Jaedicke and Mitschang [13], the parallel execution of user-defined code requires a static partition strategy to be defined for a user-defined table function so that the database can parallelize it. The map-reduce paradigm for large-scale data processing triggered a number of solutions to execute user-defined functions in the core database engine, e.g. [3, 5, 10]. For example SQL/MapReduce [10] supports user-defined table functions (UDFs) that are called from standard SQL. Parallelism is achieved by processing single map and reduce jobs in parallel, either in a multi-node database or in single-node database with multiple cores.

Task-based execution in the DBMS. An alternative to traditional thread-based execution in databases is task-based (state-based) execution as introduced in [4]. Instead of immediately using OS threads for every query, the DBMS wraps operations into tasks and stores them in task queues. A pool of worker threads is employed to process the tasks. Specialized database schedulers have numerous advantages compared to the multi-purpose OS schedulers. In addition to enabling a fine-grained workload prioritization, they are aware of workload characteristics and data dependencies. Task-based execution is used in numerous modern DBMS and research prototypes, including SAP HANA [18], IBM DB2 BLU [20], HyPer [16], etc. Task scheduling works for both OLTP and OLAP workloads, although we have shown that workload management, e.g., through task prioritization, is essential to avoid the OLAP workload dominating over a concurrent OLTP workload [19]. The main reason is that analytical workloads are typically more aggressively parallelized than transactional workloads. In a similar manner, we show in this paper that workload management is essential to be unified for the core database task scheduler and the multi-threaded external routines.

3. INTEGRATION APPROACHES

In order to unify the scheduling of the DBMS with the scheduling of multi-threaded external routines, one can potentially use OS facilities, such as thread binding or thread prioritization. These techniques, however, rely on the OS implementation, and can only support approximate workload management techniques. Ideally, we would like a full integration of the DBMS task scheduler with shared memory programming extensions used by multi-threaded external routines. A single set of worker threads should process both core database workloads and application code. To realize an integrated task scheduler, we can consider the following three alternative approaches.

The first option is a single task scheduler which implements the interface of both the proprietary DBMS task scheduler and the shared memory programming extension. Having a single task scheduler implies also a single thread pool which is shared by both the database workload and the application code in external routines. We do not pursue this option, however, because it requires changes to the proprietary and highly tuned task scheduler of the database.

The second option is to design the DBMS task scheduler using the shared memory programming extension. Consequently, the task scheduler employs the threads of the extension to execute database workloads. We do not pursue this option as well, since it also requires modifications to

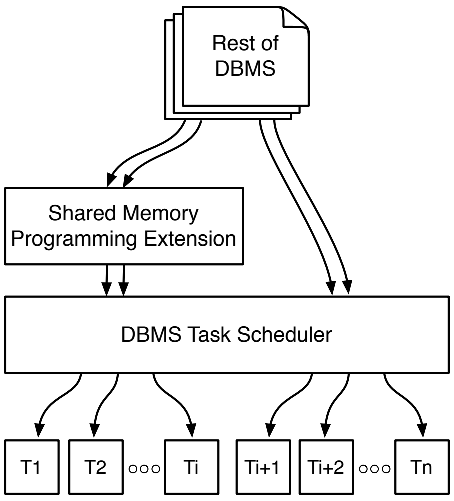


Figure 3: Chosen approach – integrate shared memory programming extensions into the DBMS task scheduler.

the database task scheduler. Furthermore, not all shared memory programming extensions support explicit workload management techniques, such as prioritization.

The last option, which we pursue in this work, is depicted in Figure 3. The shared memory programming extension uses the DBMS task scheduler as its runtime. A major advantage of this option is that the task scheduler remains intact. If the task scheduler obeys to the three basic assumptions mentioned in the introduction (see Section 1), it is expressive enough to evaluate the operations of the shared memory programming extension.

Next, we give an overview of OpenMP, the shared memory programming extension which we integrate into the DBMS task scheduler. We then detail how our chosen approach can be realized, and its implications for performance and workload management.

4. OPENMP

OpenMP is an interface specification for shared memory programming, and available as a language extension for C, C++, and Fortran on all major compilers and platforms. In this paper, we pick OpenMP as our exemplary shared memory parallelism interface for external routines, mainly due to the following properties of OpenMP.

OpenMP is an industry-wide accepted standard. It has a convenient API, and contains parallelization patterns such as parallel blocks, loops and tasks, supported by data sharing and synchronization concepts. Parallel algorithms can be expressed in simple and compact code, improving productivity. Additionally, OpenMP provides a more abstract API than most conventional DBMS task schedulers, and is well-suited for implementing analytical and scientific algorithms. There is already a large code base built using OpenMP, e.g., Intel's Math Kernel Library[1] (MKL), AMD's Core Math Library[2] (ACML), IBM's Parallel Engineering and Scientific Subroutine Library[3] (Parallel ESSL), RogueWave's International Mathematical and Statistical Library[4] (IMSL), the NAG Library[5], and the Automatically Tuned Linear Algebra Software[6] (ATLAS). These implementations are used in a large variety of applications, e.g., financial services, resource planning, seismic analysis, nuclear engineering, high performance computing, forecasting, risk assessment, quality control, and visualization. By integrating OpenMP in the DBMS task scheduler, OpenMP-based libraries and applications can be easily integrated in the database.

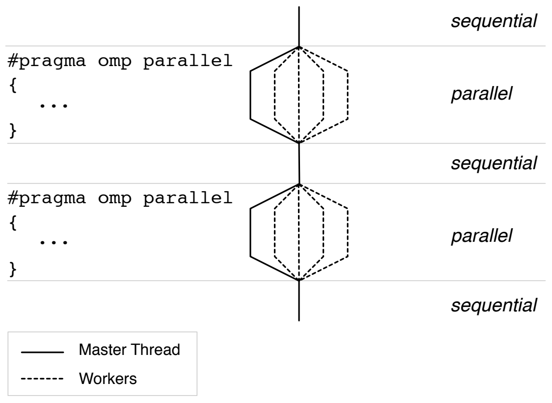


Figure 4: Basic OpenMP parallelization model.

This work focuses on OpenMP for C/C++, where OpenMP is based on pre-processor pragma annotations, runtime functions and environment variables. OpenMP's core concept is the *fork/join* model for parallelization, and the corresponding `#pragma omp parallel` directives for annotating `parallel blocks` as shown in Figure 4. A parallel block has a *master thread*, a few *worker threads*, and a specified *operation*. The master thread and the set of workers are called a *team*. A team has a certain team size which consists of the number of workers plus the master thread. At the beginning of each parallel block, there is a fork barrier at which the master thread and the worker threads are synchronized. After all team threads reach the fork barrier, each team member starts executing its assigned part of the team operation. The parts are assigned by the compiler at compile time. At the end of each parallel block, there is a join barrier to synchronize the master thread and workers again, to continue the execution sequentially on the master thread. It is possible to have nested parallelism, so that a parallel block can contain another parallel block. In this case there are several master threads, since all parent team threads are master threads for child-level teams. The top-level master thread is then called *root thread*. Moreover, several root threads can exist since also manually created threads can specify parallel blocks. OpenMP's implementation manages threads and barriers, which makes it easy to create a series of sequential and parallel blocks.

OpenMP requires strong compiler support. Figure 5 shows the compilation of an OpenMP C/C++ program using `gcc`. Translating C/C++ code into machine code starts in the C Preprocessor that typically handles all `#pragma` directives. Nevertheless, resolving the OpenMP directives requires a higher level of sophistication that is beyond the capabilities

[1]`https://software.intel.com/en-us/intel-mkl`
[2]`http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/`
[3]`http://www-03.ibm.com/systems/power/software/essl/`
[4]`http://www.roguewave.com/products-services/imsl-numerical-libraries`
[5]`http://www.nag.co.uk/numeric/numerical_libraries.asp`
[6]`http://math-atlas.sourceforge.net/`

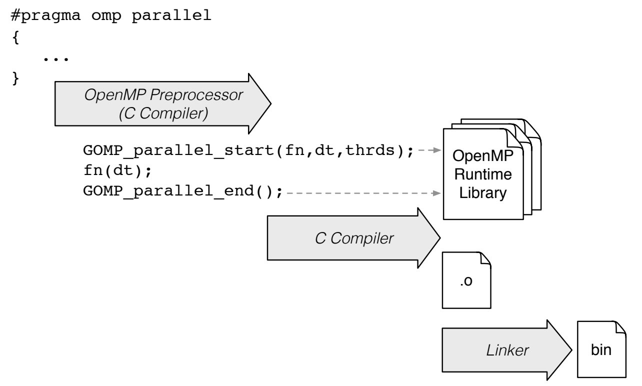


Figure 5: Compiling application code, which uses OpenMP, with the gcc compiler.

of the C Preprocessor. For that reason, the directives are then forwarded to the C Compiler. Compiling starts with the OpenMP preprocessing, so that the OpenMP directives are transformed into usual C code. The generated C code does not create threads or synchronization mechanisms, but instead calls functions of a `gcc`-compatible OpenMP runtime library. After the OpenMP preprocessing, compilation continues as usual. The C Compiler creates an object file, and the Linker creates the final program binary using all object files and the OpenMP runtime library.

To understand how a parallel block is conceptually executed within an OpenMP runtime library, we depict the master thread's processing as a flow chart in Figure 6. The processing is divided into the following three main steps.

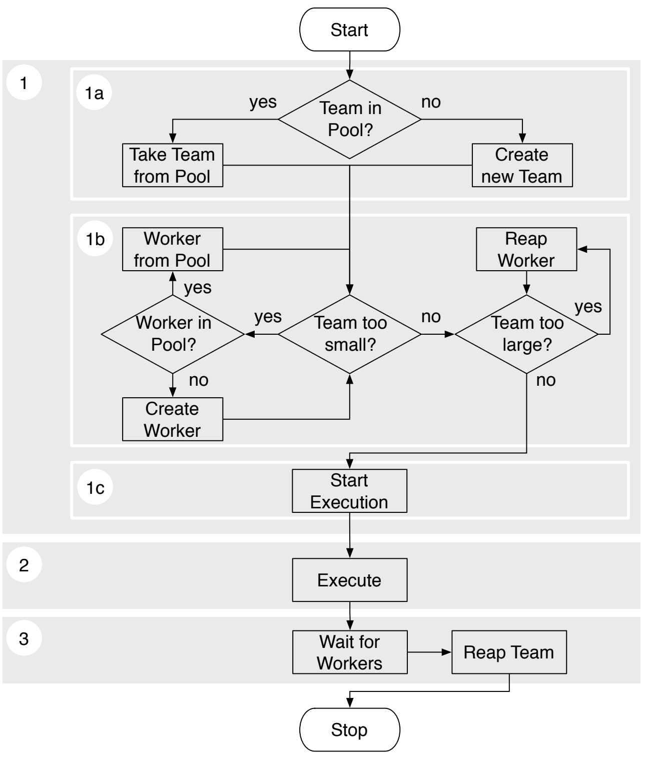


Figure 6: Flow chart showing the processing of a parallel block in an OpenMP runtime library.

(1) Fork. In this step, the master thread allocates a team (1a), adjusts the team size (1b), and starts the execution (1c). Allocating a team means creating a new team or taking one from the team pool. After that, the team size is adjusted to the specified size. New teams contain only the master thread and have a team size of 1, which usually requires adding additional workers. A worker is a software thread, and either created or taken from a worker pool. Finally, all team workers wait at a barrier. Recycled teams from the team pool can have more workers than necessary for the parallel block, which causes the "reaping" of excess workers. After the team is allocated and adjusted, the master thread enters the barrier to start the execution.

(2) Execute. In this step, the master thread processes its assigned part of the team operation, concurrently with the other worker threads. This includes invoking a function pointer that was created by the OpenMP compiler, together with a set of input parameters.

(3) Join. In this step, the master thread waits for all workers to finish using a barrier. Then, the team is reaped, which means it is pushed back to the team pool or destroyed.

5. INTEGRATION OF OPENMP WITH A DATABASE TASK SCHEDULER

In this section, we describe the concept and concrete implementation for adapting runtime libraries of OpenMP to use a database task scheduler. We start with a summary of the main prerequisites a database task scheduler needs to fulfill to be used as part of an OpenMP runtime. Next, we summarize the main steps to integrate an OpenMP runtime with a database task scheduler, independent from a concrete implementation, as shown in Figure 3. Finally, we detail our concrete implementation where the task scheduler of SAP HANA serves as the runtime for Intel's OpenMP library.

5.1 Prerequisites for the Task Scheduler

While we implement an OpenMP runtime library that is based on the task scheduler of SAP HANA, we conjecture that other DBMS task schedulers can also be used in the same way. The task scheduler needs to fulfill the following prerequisites in order to be integrated with the OpenMP runtime. First, the task scheduler must be based on shared memory parallelism. This is a direct consequence of the shared memory programming model of OpenMP. Second, the task scheduler has to offer an interface for executing arbitrary code. This can be realized either by passing a function pointer or by overriding a virtual `run` method from an abstract task class. This feature is required because, as discussed in Section 4, during compilation of a program with OpenMP directives, the compiler generates arbitrary code blocks which are passed to the OpenMP runtime for execution. Third, the task scheduler needs to expose a set of synchronization primitives that can be used to synchronize the main thread with the other worker threads. Otherwise, it is not possible to implement the fork/join parallelization model offered by OpenMP.

Finally, if the task scheduler supports workload management, e.g. through task prioritization, the most important benefit of the integration can be attained: unified workload mangement across both database workloads and external routines. This allows the database task scheduler to deal with the different requirements regarding performance and robustness of the core database workload and the external routines using OpenMP. While the core database must work in a responsive and robust way at all times, the external routine can be served in a best-effort manner. These complementary performance requirements are the main motivation for extending the database task scheduler for multi-threaded application code, and they cannot be satisfied and fully controlled with the generic thread scheduling of the OS.

5.2 Generic Integration Approach

In this section, we describe the necessary steps to integrate a database task scheduler, which fulfills the prerequisites mentioned above, with an OpenMP runtime. There are two OpenMP concepts that need to be considered: The `#pragma omp parallel` directive for parallel blocks, which is the basis for most OpenMP features, and the `#pragma omp task` directive for supporting tasks. As depicted in Figure 6, a parallel block contains three main steps: (1) fork, (2) execute, and (3) join. Next, we describe the differences of these steps for a parallel block, and then we detail our concept for supporting OpenMP tasks.

(1) Fork. The core idea of our concept is to create a database task instead of a new thread in the fork step of a parallel block, and run the conventional execution logic of an OpenMP runtime library in the database task. The processing of a parallel block is delegated to database tasks.

This raises the challenge of dealing with the mismatch between threads and database tasks in the OpenMP runtime library design. Threads are typically expensive to create. For that reason, conventional OpenMP runtime library designs contain worker pools to keep the worker threads alive, as long as necessary, and recycle them. In contrast, database tasks are typically short running and finish immediately after their work is done. Endless tasks for the database task scheduler would significantly hurt its workload strategy, and prevent the threads of the database task scheduler from taking on new tasks. As a consequence, our OpenMP runtime library concept does not keep workers and the related database tasks alive, and finishes the worker and its task immediately after processing a parallel block. The fact that workers finish by themselves has to be considered in the design of an OpenMP runtime library. Our concept deals with the finished workers in the join step. Compared to conventional OpenMP runtime libraries, our concept does not need a worker pool anymore because this is now provided by the database task scheduler.

Starting a parallel block requires the synchronization of all workers of the parallel block with the master thread. It is essential to use the corresponding synchronization primitives provided by the database task scheduler. The task scheduler is then able to recognize which threads are waiting on a synchronization primitive. Bypassing the database task scheduler and attempting to directly use OS synchronization primitives is not an option. Otherwise, in cases where there are more OpenMP workers than threads in the database task scheduler, execution would get stuck. Furthermore, a few conventional OpenMP runtime libraries use a busy-waiting strategy for the barrier implementations to improve performance. In a heavily loaded system we can expect short waiting times until a thread is dispatched a new task, and hence busy waiting can be effective. In a database setup, however, this is problematic because busy waiting barriers consume CPU time that could be otherwise used beneficially. For this reason, it is recommended to only use the synchronization primitives offered by the database task scheduler for synchronizing the workers.

(2) Execution. This step is similar to conventional OpenMP runtime concepts. Each worker invokes the function provided in its worker data structure, and the master thread processes his assigned part of the parallel block operation. The only difference is that the workers use a database task instead of directly a thread.

(3) Join. As depicted in Figure 6, the join step starts with synchronizing the master thread and the workers using a barrier. Similar to the barrier in the fork step, it is important to also use a database task scheduler barrier, and not try to use OS synchronization primitive directly. In general, the synchronization primitives used to synchronize starting and joining the workers of a parallel block should be also used to map OpenMP's `#pragma omp barrier` feature. After the barrier of the join step, but before reaping the team, the mismatch between threads and database task has to be dealt with. At the end of a parallel block, the master thread is the only member in a team, because the workers and especially their execution contexts, the database tasks, are already finished. Before the team is potentially pushed back to the team pool, the finished workers have to be removed from the team, and the team size has to be set to 1. When such a team is recycled the team size of 1 is not a problem because the team is adjusted at the beginning of each parallel block. All in all, at the end of each parallel block, it has to be ensured that all records and references to finished workers are removed, and the library data structures are consistent. A few OpenMP runtime library designs use global data structures to maintain worker information, which have to be cleaned when applying our concept.

OpenMP tasks. Besides parallel blocks, OpenMP also features a task concept. In conventional OpenMP runtime libraries tasks are pushed into a queue for further scheduling and dispatching to workers. When the queue is full, the execution of a task is invoked immediately in the master thread. Clearly, our concept can easily support OpenMP tasks by directly mapping an OpenMP task to a task of the database task scheduler.

5.3 Implementation in SAP HANA

In this section, we detail an exemplary implementation of our approach. Instead of creating an OpenMP runtime library from scratch, we refactor `libiomp5`, which was initially created by Intel for Intel's `icc` compiler. It is available as open source, it supports `icc`, `gcc`, and `clang` as front-end compilers, and it has a BSD-style license. Furthermore, we pick SAP HANA for our integration because its task scheduler fulfills all prerequisites (see Section 5.1). The task scheduler of SAP HANA is called *JobExecutor*, and has an abstract *job* class (corresponding to a task), with a virtual `run` method that can be overridden to specify arbitrary code to run. The JobExecutor itself contains priority queues for jobs and a pool of worker threads [18]. Jobs are passed to the JobExecutor together with their execution priority, and pushed to the priority queues. Afterwards the JobExecutor schedules jobs from the priority queues to the worker

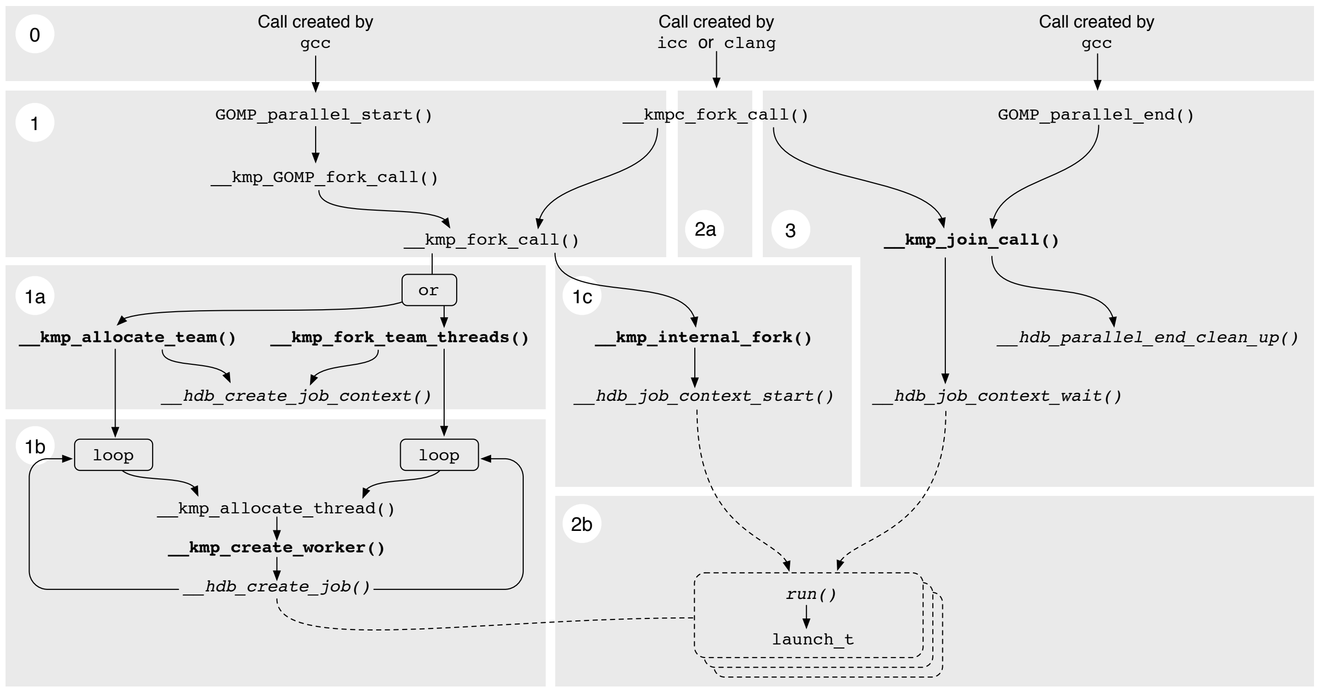


Figure 7: Main callstacks involved in the processing of a parallel block in `libhomp`.

threads by taking the job priority into account. Furthermore, jobs can be bundled into a *JobContext*, so that they can be started simultaneously, and the caller can wait for the whole bundle to finish. Furthermore, the JobExecutor provides synchronization primitives, such as barriers.

We call our OpenMP runtime library implementation `libhomp`. Figure 7 shows its main callstacks involved in the processing a parallel OpenMP block. Depicted in bold are `libiomp5` functions that we modified. New functions to invoke SAP HANA's JobExecutor are shown in italic. Default `libiomp5` functions that are not touched to apply our concept are depicted in normal text. We divided Figure 7 into eight sections that also correspond to the flow chart elements of Figure 6. Next, we detail these callstacks.

(0) Compiler Calls. When `gcc` compiles OpenMP directives it creates calls to the `GOMP` runtime library interface. `icc` and `clang`, in contrast, create calls to the `__kmpc` runtime library interface. The concepts for processing a parallel block are slightly different in both compiler interfaces. In the `__kmpc` interface, a parallel block is mapped to one runtime library invocation, and in the `GOMP` interface to two. The difference is the master thread. In the `GOMP` interface the master thread leaves the runtime library between fork and join to do its part of the parallel block. In the `__kmpc` interface, the master thread is not leaving the library during a parallel block, and executes its part of the team work inside the runtime library. Nevertheless, at some point the two different compiler interfaces invoke exactly the same internal functions for the fork and join steps.

(1) Fork. The master thread starts the fork operation with allocating a team (1a), and adjusting the team size (1b). After the team is allocated and adjusted, the master thread starts the parallel processing (1c).

(1a) Team allocation. There are two different functions for allocating a team depending on the nesting level of the parallel block. Additionally, a team can be allocated by either creating a new team or taking an existing team from the team pool. In both allocation functions, we invoke a new function to create a JobContext that is referenced in the corresponding team data structure.

(1b) Team size adjustment. At this point, the team size is always 1, because new teams and teams from the pool contain only the master thread. If a team larger than 1 is required, which is usually the case, the next step is adjusting the team size by adding new workers. The master thread enters a loop to allocate new workers. We do not keep workers alive and consequently do not employ a worker pool. As a result, we cannot allocate a worker from the worker pool and always create new workers. Instead of creating one software thread per worker, we create a new job. Each new worker job is added to the JobContext of its team.

(1c) Start execution. After the team is allocated, the master thread initiates the worker jobs. Instead of using a conventional fork barrier, we use the JobContext to start all the worker jobs (2b). The fork operation is finished at this point. In the `GOMP` interface, the master thread leaves the library at this point, and re-enters it in step (3). In the `__kmpc` interface, the master thread continues with step (2a) and then step (3).

(2a) Execution – master thread. This step is only relevant for the `__kmpc` interface case. At this point the workers are allocated and started, and the master thread is executing its part of the team operation. In the `GOMP` interface case, this step is done outside the runtime library.

(2b) Execution – worker jobs. In this step, the workers do their part of the parallel block operation. The worker jobs only call the function pointer referenced in their worker data structure, and finish immediately after doing their work.

(3) Join. In this step the master thread waits for all worker jobs to finish. We use again the JobContext instead of a conventional join barrier at this point. Since this step represents the end of the parallel block, we also introduce an additional function to clean the team data structures. Afterwards, the master thread invokes the conventional `libiomp5` procedure to reap the team, and leaves the runtime library, which finishes the parallel block.

6. EVALUATION

In this section we evaluate our implementation of integrating an OpenMP runtime library, `libiomp5`, with a database task scheduler, in SAP HANA. We start with an analysis of the functional coverage of OpenMP (specification version 3.1) and detail why two features are not supported in our implementation. Next, we analyze how the performance of our approach compares to other shared memory programming solutions. After that, we present how our integrated task scheduler solves the issues of prioritizing core database workloads and multi-threaded user-defined routines. Finally, we share insights regarding the programmer productivity based on the code complexity measured in lines of code required to implement two parallel algorithms.

For our experiments, we use a HP Z620 workstation with two six-core Intel Xeon E5-2643V2 processors at 3.50 GHz with Hyper Threading enabled (for a total of 24 hardware contexts). Each processor has 25 MB last-level cache, and the Z620 is equipped with 96 GB main memory. The OS is SuSE Linux Enterprise Server 11 SP3. It is a 64-bit SMP Linux, with a 3.0 kernel. We use the `gcc` 4.7.2 compiler, with support for OpenMP 3.1.

6.1 Functional Coverage

We analyze the functional coverage of our OpenMP runtime library `libhomp`. Since it is based on `libiomp5`, it also supports `gcc` as front-end compiler. The `gcc` source contains a set of test cases for its OpenMP compiler and runtime library. Out of these 129 tests `libiomp5` successfully executes 118 tests. Compared to `libiomp5` there are three additional tests failing with `libhomp` because our concept does not support the `#pragma omp threadprivate` directive. This directive is mapped to thread local storage (TLS). TLS is based on the POSIX thread interface which allows storing `void` pointers for each thread. In `gcc`, TLS variables are declared using the `__thread` prefix. The `#pragma omp threadprivate` directive is applied to declared variables, and causes `gcc` to rewrite the variable declaration by adding the `__thread` prefix. Using TLS with a database task scheduler, such as the JobExecutor of SAP HANA, is problematic because there is no control over the mapping of jobs to threads. Furthermore, different jobs can be scheduled to the same thread. Some internal concepts in `libhomp`, e.g., the global thread ID (gtid) implementation, rely on TLS because of particular preconditions. All jobs run uninterrupted on the same PThread until they are finished, and the TLS variables, e.g., for the gtid, are set at the beginning of the job and unset at the end of the job. Nevertheless, the `#pragma omp threadprivate` directive that forwards TLS functionality to the user level is not supported in `libhomp` for the following reasons.

First, each thread allocates memory for a TLS variable instance. As a result, each database task scheduler thread would have such instances no matter if it executes OpenMP jobs or not. Having many user-level TLS variables would result in wasting memory and increased thread creation time. Second, there is no control over the user-level TLS variables. In the gtid implementation for `libhomp`, the TLS variable is set at the beginning of the job and unset at the end of the job. For code, generated by `gcc`, such control over TLS variables is not possible without modifying the code generation in the compiler. Third, OpenMP specifies data sharing between parallel blocks based on TLS. It requires that corresponding workers in two different parallel blocks to run on the same threads. This cannot be guaranteed in our concept because workers run in jobs, and jobs may be scheduled to any thread in the pool of worker threads.

In summary, the `#pragma omp threadprivate` directive is not supported in our concept. In order to support it, the `gcc` has to be modified to not map it to TLS, and create different function code respectively. Not supporting the directive does not reduce the expressiveness of OpenMP programs. Instead of creating a TLS variable, it is always possible to create an array instead. The array size is equal to either `omp_get_max_threads()` or a manually chosen thread count, and each OpenMP thread can access its private variable instance in the array using its ID (`omp_get_thread_num()`).

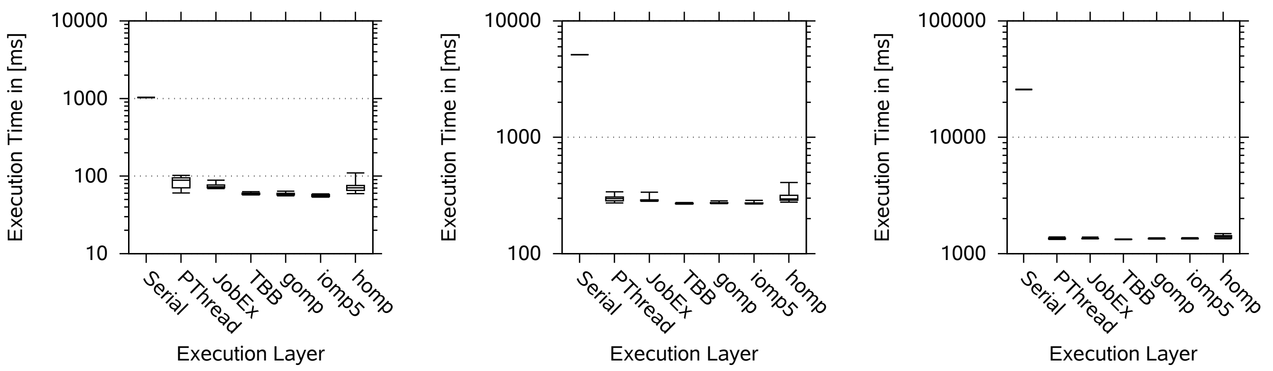
Another OpenMP feature that is not supported in our concept is the `OMP_STACKSIZE` environment variable. It can be used to set a default stack size for OpenMP threads and is read by the runtime library, which usually creates worker threads regarding this setting. In our concept, the work is forwarded to the database task scheduler that maintains its own set of threads. Only the database task scheduler would be able to create threads regarding this stack size hint, but it would affect all database task scheduler threads, since there is no distinction between threads that serve conventional jobs and threads that serve OpenMP jobs. A very small stack size could also be set to all threads, which would cause stack overflows in the DBMS. For that reason, it is more safe not to support the `OMP_STACKSIZE` variable in our concept and keep the default stack size strategy of the database.

Apart from the `#pragma omp threadprivate` directive and the `OMP_STACKSIZE` environment variable, our concept supports the full OpenMP 3.1 feature set. This contains directives such as `for`, `sections`, `single`, `task`, `master`, `critical`, `barrier`, and `ordered` including the corresponding data sharing clauses. Also all OpenMP runtime routines are supported, e.g., `omp_set_num_threads`, `omp_get_num_threads`, `omp_set_dynamic`, and `omp_set_nested`. Supported environment variables are for instance `OMP_SCHEDULE`, `OMP_DYNAMIC`, `OMP_NESTED`, and `OMP_NUM_THREADS`.

6.2 Performance of the OpenMP Runtime

We experimentally validate the competitiveness of our OpenMP runtime library, and compare it to other execution layers, i.e., shared memory programming solutions, or OpenMP runtime libraries. We pick a parallel matrix-vector multiplication as an example for a typical application-level scientific algorithm, and a parallel hash-based `SUM` aggregation as a typical core database algorithm.

As input, the matrix-vector multiplication takes a two-



(a) 80M Elements (b) 400M Elements (c) 2bn Elements

Figure 8: Execution layer comparison using matrix-vector multiplication.

dimensional integer array and a one-dimensional integer array. The result of the multiplication is written to the vector which is a one-dimensional `double` array. In order to prevent being memory bandwidth bound, we apply a sequence of square root calculations on each matrix element, before we multiply it with the corresponding vector element. Our implementation is basically a nested loop, in which each outer loop iteration calculates one element in the result vector. The outer loop is easy to parallelize because there are no concurrent writes to the output data structure. Furthermore, it does not matter in which order the outer loop iterations are processed. We run this algorithm with three different input sizes as shown in Table 1. We fill the input matrix and vector with random integer values.

Data Set	Matrix Size	Elements	Memory Size
Small	8944x8945	80 million	305 MB
Medium	20000x20000	400 million	1.5 GB
Large	44721x44722	2 billion	7.6 GB

Table 1: Data sets for matrix-vector multiplication.

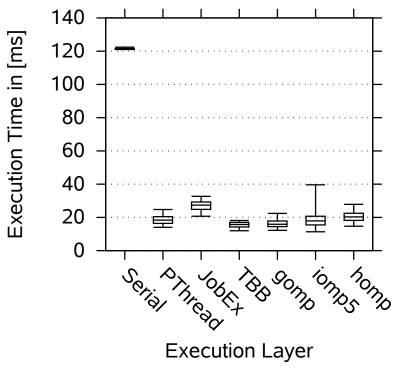
The `SUM` aggregation groups a set of tuples and calculates the sum for each group. The input of our implementation is a set of blocks. Each block contains a set of tuples, and a tuple contains a key for the grouping, and a value. In order to prevent being memory bandwidth bound, we apply a sequence of square root calculations on each value before we add it up. From the programming point of view, the input is a `std::vector< std::vector<tuple> >`. `tuple` is a struct that contains a key integer and a value integer. The input size is expressed in total number of tuples, determined by the number of blocks and the number of tuples per block. We run this algorithm with the three input sizes shown in Table 2. Initially, keys and values in the input data structures are filled with random integers between [0, 99], and [0, 19] respectively. The grouping is hash-based, using a C++11 `std::unordered_map`. It also contains the final result, which is the aggregated sum per key. In order to avoid write contention, the parallel implementations of this algorithm have local hash maps for each parallel execution context that are merged at the end. Merging the local hash maps is implemented as sequential operation. Since there are only 24 local hash maps and 100 groups, it is not expensive.

Data Set	# Tuples	Memory Size
Small	8 million	61 MB
Medium	40 million	305 MB
Large	200 million	1.5 GB

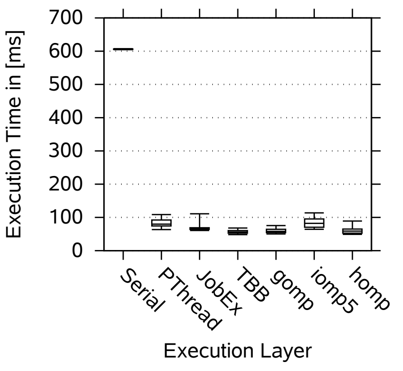
Table 2: Data sets for aggregation

Both algorithms are implemented in five different execution layers: (a) serial, (b) PThread-based, (c) SAP HANA JobExecutor-based, (d) Intel TBB-based, and (e) OpenMP-based. In addition to that, we run the OpenMP implementation with three different OpenMP runtime libraries: `libgomp` from `gcc`, `libiomp5` from Intel, and our implementation `libhomp`. We measure the execution time of each version 100 times in order to report the variance in the results. Each measuring run is a new process, and some execution layers initialize themselves when they are called for the first time. We consider this in our experiments by running each implementation twice and regard only the better result. To be fair, we exclude the thread creation time in the PThread experiments, since all other parallel execution layers employ thread pools, that are potentially initialized in the first execution. Furthermore, we do not apply any specific optimizations, and use each execution layer straight forward to create a basic and solid implementation. We strive to utilize all 24 available hardware contexts by creating either 24 PThreads, 24 SAP HANA jobs, or initializing TBB and OpenMP with 24 threads. All implementations are compiled with `gcc` using the `-O3` optimization flag.

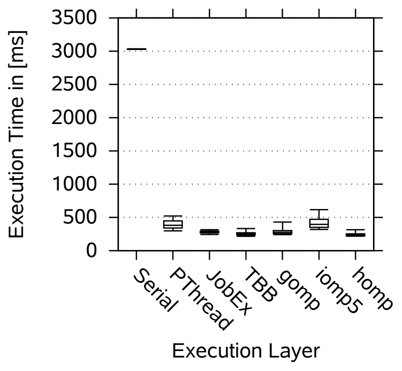
Before running the actual comparison, we experimentally determined the ideal setup for TBB and OpenMP. The TBB implementations of both algorithms rely on parallel loops, and TBB offers different partitioners for dividing loops into chunks. These calibration experiments showed no difference between the partitioners for the matrix-vector multiplication. For the `SUM` aggregation, the `auto` and `affinity` partitioner were slightly faster. Also, both OpenMP implementations rely on parallel loops, and OpenMP offers different loop scheduling strategies. We determined that `static` scheduling is the best for the matrix-vector multiplication, and `dynamic` scheduling for the `SUM` aggregation. `libiomp5`



(a) 8M Tuples



(b) 40M Tuples



(c) 200M Tuples

Figure 9: Execution layer comparison using `SUM` aggregation.

and `libhomp` support `gcc`, and also `icc`. We ran experiments on these two libraries using `icc` as compiler, but figured out that there is no performance improvement with `icc` compared to `gcc`, so we continue with `gcc`.

All in all, we have compared 7 different execution layers on two example problems with three different data sizes each. These are 42 single experiments. Figure 8 presents the results for the matrix-vector multiplication, and Figure 9 for the `SUM` aggregation. Subfigure (a) in each figure contains the smallest, subfigure (b) the medium, and subfigure (c) the largest input size. Each box plot depicts five quantiles of the variance of 100 iterations: 0% (lower whisker), 15% (lower end of box), 50% (middle of the box), 85% (upper end of the box), and 100% (upper whisker).

The results for both algorithms reveal that all parallel execution layers are in the same order of magnitude. Nevertheless, there are design and implementation differences between the parallel execution layers, resulting in slightly different runtimes for different algorithms and input sizes. For experiments with short runtimes (see Figure 9(a)) we face a slight overhead of around 10ms for the JobExecutor compared to PThread, which is mainly due to internal maintenance operations, e.g., for statistics. In the majority of cases the TBB and OpenMP implementations are slightly faster than the PThread and JobExecutor implementations. The difference between our implementation based on PThread and JobExecutor compared to the implementations based on TBB and OpenMP is the parallelization pattern. For TBB and OpenMP we use loop parallelization, which is not available in PThread and JobExecutor. Loop parallelization enables TBB and OpenMP to schedule each single iteration if necessary regarding their strategy. For PThread and JobExecutor, we manually partition the input and statically assign it to one of the 24 PThreads or jobs respectively, which is less efficient than the loop paralellization. Apart from that, all multi-threaded execution layers in this comparison are based on PThreads, and are mostly faster than our PThread implementations. Additional reasons are that the JobExecutor, TBB, and the OpenMP runtimes utilize optimizations such as binding threads to cores, busy waiting, or cache optimization strategies.

Binding threads to cores, for instance, ensures that each thread is scheduled to one H/W context, and avoids the rescheduling of threads. For our straight-forward PThread implementation, we have seen that multiple threads can be executed on the same H/W context, and that the threads can be rescheduled. Busy waiting avoids the overhead of scheduling and contexts switches, and results in a slightly better performance compared to conventional waiting. Checking the CPU times showed, that busy waiting is utilized by TBB, `libgomp`, and `libiomp5`. We did not further investigate the composition of the execution time for each execution layer, since this is beyond the objective of this section. Our intention is to show that `libhomp` is competitive, and that our concept does not introduce a severe performance overhead.

6.3 Unified Workload Management

A motivation of our work is to be able to prioritize the execution of both core database workloads and external routines. In this section, we validate that indeed our approach enables such a prioritization. We create a micro-benchmark which simulates a SAP HANA workload and at the same time executes an OpenMP-based application. We compare two different OpenMP runtime libraries for the application part: `libiomp5` which has its own set of threads, and our `libhomp` prototype which uses the database task scheduler.

SAP HANA is simulated by its JobExecutor and the parallel `SUM` aggregation (used in the previous section). The parallel aggregation simulates a typical core database operation. As an OpenMP-based application, we use the matrix-vector multiplication (used in the previous section). The input sizes of both algorithms are the medium dataset from Tables 1 and 2. The micro-benchmark's main thread creates three additional threads. One thread continuously triggers jobs to calculate an aggregation. Since creating and asynchronously starting the jobs and JobContexts is fast, the thread that triggers the JobExecutor workload sleeps for a few milliseconds when the number of incomplete jobs exceeds 100. This ensures that the aggregation is not over-represented in the workload. For the OpenMP part, we have two threads that continuously call the matrix-vector multiplication. In addition to creating the three workload threads, the main thread also sets the experiment time, determines the processor utilization and creates a summary file.

We have already presented the first part of the results, using `libiomp5` as runtime, in Figure 1 and 2 (see Section 1). The y-axis in both figures depicts the throughput of the

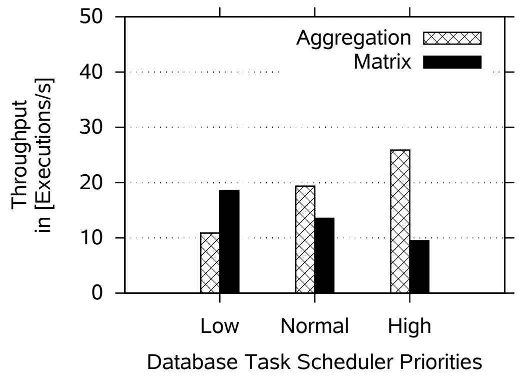


Figure 10: Workload management with `lihomp`.

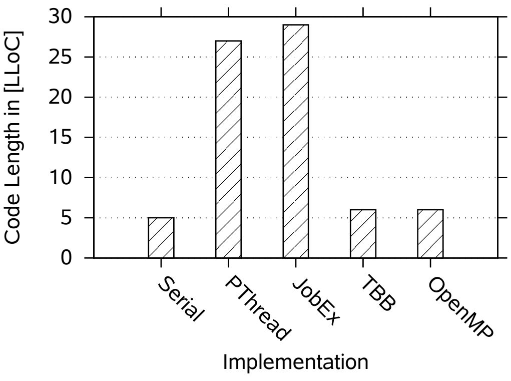


Figure 11: Code length of different execution layers for the matrix-vector multiplication.

core database and application algorithm in executions per second. It is the average of five benchmark runs, and each benchmark run is five minutes. In Figure 1, we compare the baseline, in which we only run the core database workload, with the concurrent setup, in which we run both workloads simultaneously. As we expect, this experiment shows that the application workload drops the core database throughput by more than 50%. Afterwards, we present the actual issue in Figure 2. On the x-axis, we depict three different task scheduler priorities for the core database workload. As we expect, the task priorities are not affecting the throughput behavior, and are not capable to improve the core database throughput. The OpenMP workload bypasses the database task scheduler, and the task scheduler has no control over the OpenMP runtime library threads.

We run the same setup again, but link our `libhomp` as OpenMP runtime library which uses the database task scheduler. The results are presented in Figure 10. Compared to the results of `libiomp5` in Figure 2, `libhomp` enables the prioritization between core database workloads and application workloads as needed. With our concept it is not possible anymore that multi-threaded applications cause an uncontrollable decrease in core database throughput. As shown in Figure 10, increasing the priority for the core database workload results in a higher core database throughput. Also, decreasing the priority for the core database workload is possible, which results is a higher application throughput. All in all, achieving this behavior is our main objective for integrating the task scheduling of core database and application workload. Furthermore, core database or multi-threaded applications can still utilize the entire machine, when the resource consumption of the other concurrent workload is low.

6.4 Productivity Study

The lines of source code needed to implement some functionality is a simple yet popular metric to measure the effort required to develop a program, and thus it is used to estimate the programmer's productivity. OpenMP promises to improve the productivity for implementing parallel algorithms via shorter code. With this final study we compare the code lengths of different shared memory programming solutions. In the performance evaluation in Section 6.2, we have implemented a matrix-vector multiplication and a `SUM` aggregation in five different versions: (a) serial, (b) PThread-based, (c) SAP HANA JobExecutor-based, (d) Intel TBB-based, and (e) OpenMP-based. For both algorithms, we now analyze the code lengths of their five different implementations. We manually count the logical lines of code (LLoC) required to express the core algorithm. Furthermore, we do not consider pre-processor includes, time measuring and initialization code, as well as common input and output data structures. The JobExecutor and TBB have a C++-style API, PThread and OpenMP have a C-style API. We present the results for the matrix-vector multiplication in Figure 11, and for the `SUM` aggregation in Figure 12. The different implementations are depicted on the x-axis, and the code length in LLoC on the y-axis.

First of all, the serial implementation has the shortest code length. Using PThread or the SAP HANA JobExecutor require more lines than the other alternatives for thread creation, management and synchronization. Due to the JobExecutor's object oriented design, the corresponding implementations have the largest code length. Especially the job inheritance and the visibility accessors in the class definition increase the code length compared to the PThread version. The code lengths of the high-level execution layers TBB and OpenMP are in between the serial implementation and the PThread and OpenMP implementation. For algorithms that are easy to parallelize such as the matrix-

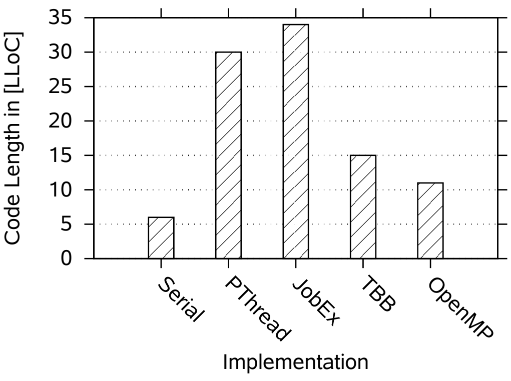


Figure 12: Code length of different execution layers for the `SUM` aggregation.

vector multiplication, OpenMP and TBB need just an additional line compared to the serial implementation. The SUM aggregation for instance is harder to parallelize and needs additional code for the local hashmaps and the merge operation. In this case, the code length advantage of TBB and OpenMP in contrast to PThread and JobExecutor is lower.

In summary, this analysis shows that OpenMP enables much shorter code compared to the low-level shared memory programming solutions that require manual thread management and synchronization, e.g., the PThread library. In addition to the advantages listed in Section 4, at least for SAP HANA, OpenMP is a better shared memory programming extension for multi-threaded application code than the default database task scheduler, since OpenMP can match its performance but also enable shorter code.

7. CONCLUSION

In this paper we argue that one should integrate the scheduling of core database workloads and parallelized external stored routines. This allows the database to use all available execution contexts while still being able to protect the core database logic from an overwhelming number of threads being created in external stored routines, outside the control of the database. Our implementation exposes the scheduling capabilities of SAP HANA to external stored routines via the OpenMP API, and the SAP HANA job scheduler serves as runtime library for OpenMP. We compare our runtime implementation to other approaches of shared memory programming and find that the performance is competitive. The main advantage however is a more robust and predictable behavior of the database system and tightly integrated and highly parallelizable application logic.

With our OpenMP integration concept, it is possible to create multi-threaded platform-independent application code based on a widely accepted standard. Furthermore, our work enables to leverage the OpenMP API for implementing core database logic where in the past this was not easily possible due to legacy interfaces of the proprietary database task scheduler. It can allow developers to use more high-level and well-understood programming idioms.

Finally, our paper enables the combination of data management features with strong analytical capabilities. Database systems typically offer advanced data management features and weak analytical capabilities, while mathematical and statistical packages offer advanced analytical capabilities and weak data management features. Numerous of these packages are OpenMP-based, e.g., IMSL, MKL and CML. Our work is the foundation to efficiently support this large OpenMP code base for analytical and scientific algorithms inside the database system.

8. REFERENCES

[1] *SAP HANA Predictive Analysis Library (PAL)*, 2015. http://help.sap.com/hana/SAP_HANA_Predictive_Analysis_Library_PAL_en.pdf.

[2] *SAP HANA SQLScript Reference*, 2015. http://help.sap.com/hana/sap_hana_sql_script_reference_en.pdf.

[3] D. Battré et al. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *Proc. ACM SoCC*, pages 119–130, 2010.

[4] D. Carney et al. Operator scheduling in a data stream manager. In *Proc. VLDB*, pages 838–849, 2003.

[5] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[6] L. Dagum et al. OpenMP: an industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng*, 5(1):46–55, 1998.

[7] A. Eisenberg. New standard for stored procedures in SQL. *SIGMOD Rec.*, 25(4):81–88, 1996.

[8] F. Färber et al. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[9] U. Fischer et al. Towards integrated data analytics: Time series forecasting in DBMS. *Datenbank-Spektrum*, 13(1):45–53, 2013.

[10] E. Friedman et al. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2(2):1402–1413, 2009.

[11] P. Große et al. Bridging two worlds with RICE integrating R into the SAP in-memory computing engine. *PVLDB*, 4(12):1307–1317, 2011.

[12] G. Harrison. MySQL stored procedures: Next big thing or relic of the past? *Linux Journal*, 2007(164).

[13] M. Jaedicke et al. On Parallel Processing of Aggregate and Scalar Functions in Object-relational DBMS. In *Proc. SIGMOD*, pages 379–389, 1998.

[14] D. Kernert et al. Slacid - sparse linear algebra in a column-oriented in-memory database system. In *Proc. SSDBM*, pages 11:1–11:12, 2014.

[15] W. Kim et al. Multicore desktop programming with intel threading building blocks. *IEEE Softw.*, 28(1):23–31, 2011.

[16] V. Leis et al. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proc. SIGMOD*, pages 743–754, 2014.

[17] V. Linnemann et al. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proc. VLDB*, pages 294–305, 1988.

[18] I. Psaroudakis et al. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *ADMS*, pages 36–45, 2013.

[19] I. Psaroudakis et al. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Proc. TPCTC*, 2014.

[20] V. Raman et al. DB2 with BLU Acceleration: So much more than just a column store. volume 6, pages 1080–1091, 2013.

[21] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[22] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[23] M. Wilhelm et al. Mass-spectrometry-based draft of the human proteome. *Nature*, 509(7502):582–587, 2014.