The Bobox Project
Parallelization Framework and Server for Data Processing

David Bednárek Jiří Dokulil Jakub Yaghob

Technical Report 2011/1
Department of Software Engineering
Charles University Prague

Abstract

Today, parallel processing is one of the most significant trends in data processing. Several parallelization libraries and frameworks have been created to provide more advanced programming environment that the low-level services provided by the operating system. The Bobox project is a parallel processing framework, designed specifically for processing of large amounts of data, and a server utilizing this framework. It sacrifices generality to provide easier to use interface and possibly better performance for a certain class of problems – data-intensive computations based on a non-linear pipeline.

1 Introduction

One of the main trends in software development today is parallel processing of data [13, 27]. Most CPUs sold today have more than one core and with multiple slot CPUs and HyperThreading technology [4], one computer node may easily reach 32 physical cores and 64 logical threads in the near future. We may expect, the number of cores and logical threads will further increase. On the other hand, the performance of the individual cores is increasing very slowly due to physical limits, most notably to maintain manageable levels of heat output. Moreover, systems with heterogeneous processors/application accelerators will be available [1]. One form of heterogeneous computing is already present in today's personal computers in the form of GPU (graphics processing unit) used as GPGPU (general-purpose computing on graphics processing units). Unified CPU and GPGPU computation is enabled by the recently published OpenCL standard [14].

Writing software that can execute in parallel in C++ using just the services provided by the operating system is a very complex and error prone task. Moreover, C++ is lacking any form of parallel facilities in the language standard for years, although it will change in the near future. To make it easier, several libraries and compiler extensions have been created. Although OpenMP [21] has a long history, it is best suited for mathematical computations. The MPI library [19] is aimed at distributed computing, but may also be used on just one node. Lately, the Intel's Threading Building Blocks (TBB) library [17] is gaining popularity as a library of choice for parallel processing, due to its modern design and ability to handle a diverse range of problems.

Unfortunately making parallel programs in TBB, OpenMP or MPI requires significant knowledge about synchronization, cache hierarchy and a lot of other technical details. The Bobox project has two primary goals: simplify writing parallel, data intensive programs and to serve as a testbed for developing generic parallel algorithms and data-oriented parallel algorithms. The simplicity comes from four sources:

1. All synchronization is hidden from the user (programmer), user code is only single-threaded
2. Most technical details (Non-Uniform Memory Access – NUMA, cache hierarchy, system and CPU architecture) are handled by the Bobox
3. The Bobox offers only high-performance messaging as the means of communication and synchronization; the user doesn't need to understand and learn large number of functions, data structures, etc.
4. The basic paradigms used in the Bobox are task parallelism and non-linear pipeline, both of which are easy to comprehend.

The Bobox project is realized as a standalone Bobox server and client libraries, where the server is implemented using a Bobox parallelization framework. The Bobox framework provides a run-time environment that is used to execute a non-linear pipeline in parallel. The pipeline consists of computational components provided by the user and connecting parts that are part of the Bobox framework. The structure of the pipeline is defined by the user but the communication and execution of individual parts is handled by the run-time based on several rules (in short, a component is executed when it has data waiting to be processed on its inputs; see the Section 3) and the pipeline's structure. This simplifies the design of the individual computational components, since communication, synchronization and scheduling are handled by the framework. Some of the design principles are similar to the TBB library – the Bobox framework also provides task level parallelism, but for a more specialized class of tasks. The basic design principle of task parallelism, fixed number of worker threads, and task stealing are present in both TBB and Bobox, but there are major differences in the organization of the task pool (completely different rules that determine when a task can be executed), in the way the whole program is put together (in TBB, the user's code invokes the library to perform sub-tasks which may in again invoke the library; in Bobox, the structure of whole computation is defined by the user before the start), and in the way data is handled (Bobox passes the data in envelopes with library-defined structure; TBB leaves data representation almost completely to the user). Integral part of the Bobox framework is a C++ library as an interface to the run-time environment, which allows use of the Bobox framework in a software written in C++.

But there is also a certain degree of data level parallelism – the data is sent through the pipeline in a way similar to column oriented databases [26]. This way, all values of a column are stored in a continuous block of memory which in turn enables the computation to use vector instructions like SSE available in modern CPUs.

The following text describes the Bobox parallel architecture in greater detail, but also provides brief information about other parts of the Bobox project. First, the main ideas behind the design of the parallel library are discussed in the Section 2. Then, a detailed description of the parallel run-time environment is given in the Section 3, followed by a closer look at the way the data are handled by the system in the Section 4. Some of the non-essential, detailed information about the Bobox library and a brief description of the overall architecture of the Bobox query processing server are given in the Section 5. The Section 6 shows two sets of experiments we have performed with the system: a test implementation of the SP^{2}Bench benchmark and synthetic benchmarks aimed at testing various parameters of the system. The SP^{2}Bench is used since it is an RDF query benchmark and RDF querying is one of the main scenarios the Bobox was originally designed to handle. The Section 7 shows several C++ code examples taken from an application that utilizes the Bobox system. It demonstrates the programmatic interface of Bobox and common patterns that a developer would use when working with the library. The Section 8 refers to related work. The last sections conclude the paper and discuss future work.

2 Main design concepts

The main idea behind Bobox is to connect a set of relatively simple computational components into a (non-linear) pipeline. The data then flow through the pipeline, which controls the execution of the individual computational components. The components are executed in parallel, when they have data to process. But each computational component is guaranteed by the system not to be executed in parallel. Furthermore, the interface between the components and the rest of the system is created to eliminate any possible race conditions when reading inputs and writing outputs. So if the components do not communicate by other means than by sending messages over the pipeline, they need not handle problems common to parallel programming.

2.1 Task level parallelism

The environment with many simple components and pipeline-based communication is very suitable for task level parallelization. In this paradigm, the program is not viewed as a process divided into several threads. Instead, it is seen as a set of small tasks [17]. A *task* is a piece of data together with the code that should be executed on the data. Their execution is handled by a *task scheduler*. The scheduler maintains a pool of tasks to be executed and a pool of execution threads and allocates the tasks to the threads. At any given time, a thread can either be executing a task or be *idle*. If it is idle, the task scheduler finds a suitable task in the task pool and starts the execution of the task on the idle thread.

There are several advantages to this approach. One of them may be the fact that in some cases such system may be easier to use for the software developers. But there are even more fundamental advantages. Task based parallelization may provide better utilization of system resources. First, the smaller granularity of tasks compared to threads means that the CPUs can be better utilized when there is a synchronization barrier in the execution – the finer granularity of work decreases idle time of the CPU cores. While in classical thread based parallelization a thread that finishes before the other threads has to idle (wait), in task based parallelization it can execute other tasks.

Second, a carefully designed scheduler can achieve a much better use of CPU caches. When a piece of data is transferred from one thread to another, the data usually has to be loaded into the cache of the processor core where the receiving thread is currently running. In task parallelism, this situation is usually handled by creation of a new task with the data. This new task can be executed on the same CPU core as the task that created the data (after it finishes). This usually means that the data is still *hot* (stored) in the cache and the loading is avoided. Naturally, this only works in situation, where the transferred piece of data is small enough to fit into the CPU cache. Most CPUs have several levels of cache that are progressively larger but slower. This cache levels have strong influence on the way in which the overall speed of the computation changes with the size of each data chunk. If the chunk can fit into the smallest (Level 1) cache, it takes the least time to process the chunk. But we have to consider not only the size of the processed data, but also the memory that is used while the data is being processed (temporary variables, state variables of the component, I/O buffers, etc.), since these memory operations also use the CPU cache. The size of the memory used this way may significantly exceed the size of the processed data and it is usually hard to predict, for example in the case of various system and library calls, where the code is outside user's control. But as a general rule, smaller size of data chunks increases the chance that the data can be processed in the cache without the (relatively slow) access to the main memory.

On the other hand, there is overhead associated with the work of the task scheduler, namely maintaining the task and thread pool and the execution of tasks – each task takes some time to set up before the actual code is executed and also some time to clean up after the execution. Although the scheduling algorithm complexity is $O(1)$, total overhead for the whole computation is proportional to the number of tasks, so its percentage in the total run-time decreases if the individual tasks take a long time to complete. But the smaller run time of tasks improves CPU utilization, since it reduces the time that the threads spend idling when there is not enough work available for all threads. So, some balance between these aspects has to be found when looking for the optimal task size. The optimal size greatly depends on the task that is being performed, but experiments that we have performed with an early version of Bobox suggest, that the sizes of various cache levels are very significant in this case, so a good starting point is to test the performance with data chunks that are half the size of the different levels of cache. Since today's CPUs usually feature three levels of cache, testing just three different chunk sizes should give the user a decent first idea.

3 Run-time architecture

One of the main differences between other parallelization frameworks and the Bobox architecture is the way the user's code interacts with Bobox. OpenMP and TBB are used to invoke parts of the code in parallel; MPI provides means for communication between processes. Bobox is more similar to the first two systems, but there are two key differences. First, it uses a declarative approach

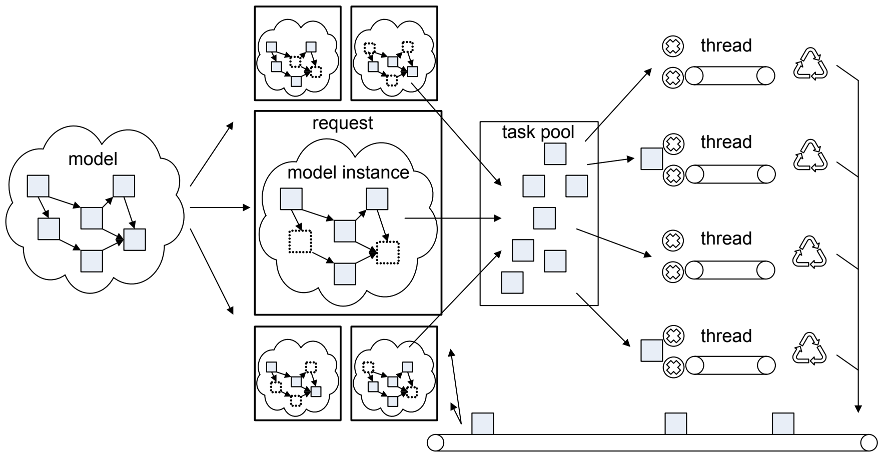


Figure 1: Run-time architecture

to describe the way in which elements of the computation are put together. Second, it provides more services to the user code (data transport, flow control etc.) but also imposes greater restrictions on the code (only pipeline, no recursive calls to Bobox to complete sub-tasks, etc.).

The parallel execution environment displayed in the Figure 1 is somewhat similar to that of TBB, since it contains a task pool and several threads that execute tasks from that pool (the right half of the figure). However, the way in which the tasks are created and added to the pool is completely different. In TBB, this is controlled either directly by the user's code or using a thin layer of parallel algorithms provided by the library.

In Bobox, the user first specifies a *model*. The model defines the way in which the individual computational components are connected. The model is then *instantiated* to produce a *model instance*. There can be multiple instances of one model. The instance then forms a basis for a *request*. The request contains only little information besides the model instance, mainly a unique request identifier. When the user code sends the request to the parallel execution environment, it no longer has control over the execution of the model instance and can only wait for the request to finish – the execution is fully handled by the execution environment based on the Bobox rules and the structure defined by the model. The most important "Bobox rule" is that components should be executed when they have unprocessed data on their inputs. Further details about this rule are presented in the Section 3.2, other rules are discussed in the Section 3.5.

The elements of the model instance are used as tasks. When they are ready, they are *enqueued* – added to the task pool. Later, a thread takes a task from the pool, perform the action (*invokes* the task) and then the model instance element is returned and can again be used as a new task and added to the pool. This conforms to the requirement that no box may be invoked more than once at the same time.

The Bobox system cannot easily be used for an arbitrary parallel computation. It is suited for a certain class of problems, due to the way in which the system decides what computational components should be executed. This is controlled by the flow of the data through the pipeline. The data must be passed in a way defined by the system so that the system is aware of the fact that a component consumed or created (sent further down through the pipeline) some data.

This simplifies the design of the individual computational components. They do not have to be concerned with controlling the execution and data flow. Their code simply reads the input data, computes the results and writes them to the output.

3.1 Basic elements and terminology

As we have already stated, the overall computation is defined using a *model* which is then used to create the actual *model instance* used in the computation. There are several elements that form the model instance.

Box is the computational component mentioned in the previous text. It has zero or more inputs and zero or more outputs in the pipeline. *Via* represents one *link* in the pipeline. It connects one output of one box to one or more inputs of other boxes. The link only provides one-way data transfer. *Envelope* is the smallest unit of data transfered through the pipeline. When a box sends an envelope to one of its outputs it is sent to the appropriate via. The via then makes a copy of the envelope for each box on the receiving end of the link and sends the copy to the appropriate box.

The model is an almost exact image of the structure of the instance. It is composed of *box models* and *via models* that define the types of boxes and the links to be used in the model instance. The main difference between a box and a box model is on an implementation level – the box does contain the actual code, buffers and state information required to run the computation while the box model does not. The same holds for vias and via models.

An example of a model (or model instance) can be seen in the Figure 2. It displays some of the basic situations like a via with more outputs, box with multiple inputs or different length paths.

3.2 Execution of the model instance

All model instances have one initialization box (see the Figure 2). This box is executed by the system at the beginning of the evaluation and its only purpose is to send one envelope to its only output. All other boxes are executed “when needed” depending on the data flow in the pipeline. For instance, a box or via is executed when an envelope is received on one of its inputs. This is the most significant case, although there are several other due to the needs of parallel execution and buffer management.

In combination, this means that the initialization box sends one envelope which is then received by all boxes linked to its output and these boxes are then executed by the system. These boxes can then send envelopes further down the pipeline which in turn results in other boxes being executed. One box can produce multiple envelopes so the whole execution is not just one “wave”.

It is important to interpret the term “executed” in the context of the task parallelism. This means that instead of directly performing the action a new task is created and added to the task pool to be executed later. We say that the box or via is *enqueued* (added to the pool) and then *invoked* (the code is actually

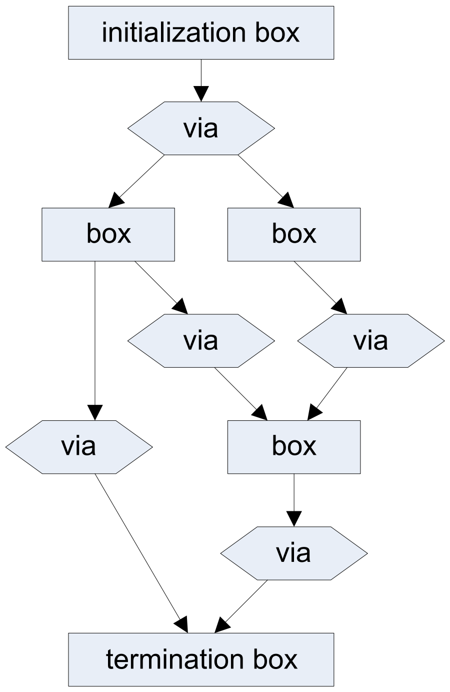


Figure 2: Pipeline example

executed by the CPU). This suits the run-time mechanism quite well, since one via can send an envelope to multiple boxes and all of the boxes should then be executed. Instead of sequential execution or launching a thread for each box a new task is created for each of the boxes (the boxes are enqueued). These tasks are then executed (invoked) on a fixed number of threads at an appropriate time determined by the task scheduler. This allows for a parallel execution without spawning too many threads, which would result in poor overall performance.

An important factor in this design is the fact that each box (or via) can be present in the task pool at most once. All attempts to enqueue a box that has already been enqueued but not yet executed have no effect – they are legal operations but do not change the state of the box or the task pool. This means, that we can enqueue a box each time it receives data on any of its inputs. Consider the case where the box has two inputs, both inputs receive an envelope, box is then invoked and processes both inputs. There were two attempts to enqueue the box but the box is invoked only once, which is the desired behavior in this case. Should the box process only one of its inputs (which is a bad design practice, but it is possible to do in Bobox), the box detects that there are still unprocessed envelopes on its input at the end of the invocation and enqueues itself, which will eventually result in the invocation of the box and processing of the second input. The same process handles the situation, where two envelopes arrive on one input of the box – the box is enqueued, invoked, processes the first envelope, it is enqueued again and then invoked to process the second envelope. The whole process of re-enqueueing the box is handled by the Bobox library as part of box invocation.

From the scheduling point of view, the boxes and vias behave the same way – they use the same enqueue-invoke mechanism, they are enqueued using the same task pool, etc. But there is a significant difference in the way they interact with the rest of the system. It is due to the fact that while boxes contain user-defined code, vias are an integral part of the framework and their behavior cannot be modified. The next section describes specific properties of boxes that provide support for the user-defined code and as such are not present in vias. A detailed description of vias will be presented later, since it requires deeper understanding of the internal structure of envelopes.

3.3 Boxes

The invocation of a box is performed in three steps: the prologue, the action and the epilogue. Only the middle part contains any user code – it performs the actual computation. The rest is provided as part of the Bobox framework and it handles communication and synchronization. The user code interacts with the rest of the system only by accessing selected member variables and methods of the box class. This creates a controlled environment for the action – it provides an abstract, stable interface and eliminates possible race conditions. For instance, the user code may use a method to send an envelope to one of the outputs. Since this action affects the rest of the system and may require synchronization, the envelope is only stored in an internal queue of the box and sent later, during the epilogue.

The prologue creates a snapshot of the inputs of the box and stores it in a member variable available to the user code. This requires synchronization with the rest of the system. The epilogue sends the enqueued output envelopes – this also requires synchronization – and may enqueue the box in the scheduler's task queue. This may occur for two reasons: the user code may have requested this explicitly (by setting a flag) or the box may have some unprocessed data on the inputs. The second condition is applied only if any data has been processed by the user code during this invocation. Otherwise, the box is considered to be waiting for further input data and is not enqueued until such data arrive (the arrival of data automatically enqueues the recipient).

The ability to again enqueue the box during its invocation has one advantage over repeatedly executing the code of the box right away – the task is placed in the task pool with other tasks and the thread can be used to perform other tasks. The subsequent execution of the original task is thus delayed. This is useful for boxes that can produce large number of envelopes without requiring more envelopes on its inputs. Such box can produce a reasonable number of envelopes and then delay further computation using this “trick”. Without it, the box would have to produce all of the envelopes thus filling the buffers unnecessarily and using a lot of memory for the envelopes. The long run-time of this box would also limit parallelism.

Another notable aspect is handling of multiple inputs – boxes can have more than one input link. Furthermore, since a box can produce several envelopes and send them to an output link, the box on the receiving end of the link needs to handle this situation as well. To address both issues, each box has a small buffer for each of its inputs. When a box is invoked (not scheduled) a snapshot of the head of each buffer is created and made available to the box. It is then up to the box whether it actually processes the envelopes on the input or leaves them in the input buffer. If the box processed any of the envelopes and the buffers are still not empty after the invocation ends, the box is enqueued again. This is the situation where the box has processed some of its inputs but there are more inputs available. On the other hand, if the box finishes the invocation without processing any of the inputs, it is not enqueued even if the buffers are not empty. This is usually the case of boxes that perform an operation similar to a database join where the data is available for just one of the inputs. Then the join can not do any computation and has to wait for the other input. Since a box is enqueued when a new input arrives, it does not have to be enqueued after the invocation.

The input buffers are also used to balance speed of different parts of the pipeline. They have a limited size and when the buffer is full, the invocation of boxes that send data to this input is suspended. The input buffers of the suspended boxes get eventually full as well. This way a large part of the pipeline can be suspended. This part contains the boxes that produced envelopes faster than the box whose buffer was first filled completely is able to process them. When this buffer is emptied, the invocation of the boxes that send data to this input is resumed. Note that the invocation is suspended or resumed not directly by suspending or resuming the execution of the thread but rather by controlling the process in which they are enqueued. This way the effect is delayed but in our case we do not need it to be immediate.

3.4 Parallel execution environment

Having one central task scheduler would create a bottleneck that would reduce parallelism and scalability (the speedup achieved when more CPUs are added). Instead, there is a separate scheduler for each thread in the thread pool. Each maintains two queues of tasks that have been enqueued as a result of an action performed on the thread that the manager is assigned to. One queue (*immediate queue*) is dedicated for tasks, which should be immediately scheduled on this thread while the data are hot in cache, the second one (*stealing queue*) holds tasks, which are not tightly bound with the thread. When a task finishes, the scheduler starts the execution of the first task in the immediate queue. If the queue is empty, the scheduler tries to schedule the head of the stealing queue. If the stealing queue is empty as well, the scheduler *steals* tasks from an another scheduler.

The process of task stealing is following: first, a *victim* is selected from the other schedulers. Current selection algorithm uses simple round-robin strategy excluding itself, future versions will be NUMA and hyper-threading aware. Then several of the tasks near the end of the victim's stealing queue are removed and placed in the stealing task's immediate queue. Then the execution continues normally. The scheduler steals more than one task to reduce the total number of task stealing, since it requires synchronization between the stealing task and the victim. The tasks are taken from the end of the stealing queue, since they have the smallest chance that their data would still be hot in the cache, so moving them to another CPU (which is likely to result in data not being in cache at all) has the lowest impact on the total performance. The round-robin selection of the victim may result in a suboptimal solution if some of the schedulers have empty queues, but this strategy eliminates the need for any central coordination thus providing better scalability. Although TBB uses similar stealing algorithm for scheduling, Bobox framework is more data aware thus keeping data hot in cache is very important factor.

Task can be enqueued for execution either as an *immediate task* or as a *relaxed task*. An immediate task is placed at the head of the immediate queue of the scheduler, whereas a relaxed task is placed at the tail of the stealing queue. The selection of immediate or relaxed enqueueing is hard-wired into box and via framework.

One task can be in at most one queue at any time, so if there is a request to enqueue a task that is already in a queue, the request is ignored. However, if the request arrives while the task is being invoked, it is placed in the queue. This allows us to enqueue a box or via any time it receives data. Such event either enqueues the box (or via) or leaves them in the queue. If new data arrives during the invocation of the task (box or via), it is enqueued. The rationale for this is that each task is supposed to create a snapshot of the data (the input to be processed) at the start of the invocation and then process the snapshot. If new data arrive during the invocation, they are not "visible" to the task and become available the next time the task is invoked.

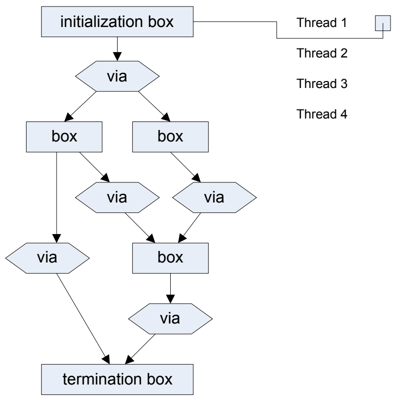
Another feature of the task scheduler is cancellation of requests. This means that request correspond to queries sent from the clients. Each task (its representation in the memory of the computer) contains the ID of the request it belongs to. If a task is canceled, all schedulers receive the ID of the canceled request and eventually (after they finish the current task invocation) remove all tasks that belong to the canceled request from the queues.

An example of the scheduling is displayed is displayed in the Figure 3 where the pipeline is shown along with the task queues of the four threads used in this case. First, in 3(a), the initial box is enqueued. When it is invoked, it produces the poisoned pill, sends it to the first via and enqueues the via (shown in 3(b)). The via duplicates the poisoned pill, sends it to the boxes and enqueues them (shown in 3(c)). The thread no. 2 steals one of the tasks (shown in 3(d)) and both tasks are invoked, the boxes produce the results and send them to the appropriate vias. Note that one of the boxes has two outputs and sends a result along both of them. Also note that the newly created tasks are enqueued with the same thread that the creator was invoked on. This is shown in 3(e). In the last step 3(f) shown in this example, one of the task gets stolen from thread no. 1 by the thread no. 3.

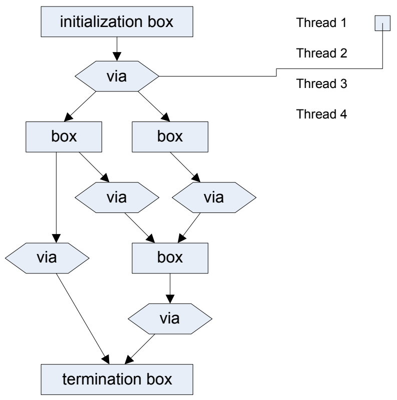
3.5 Flow control

Consider a part of a pipeline that consist of just two boxes. The first box P quickly produces data and sends it to the second box C that performs a complex computation. At some time after the start of the computation, P receives the poisoned pill, generates first envelope with data, sends the envelope to P and enqueues itself. The via V that connects P to C receives the first envelope ans is also enqueued. After invocation of P ends, V is immediately invoked by the scheduler (this is due to the way the tasks are placed into the scheduler's queue). V forwards the envelope to C, enqueues C and ends the invocation. The box C is then invoked and starts its long computation. At some point, the task corresponding to P gets stolen by another scheduler and invoked. Another envelope is produced and sent to C over V.

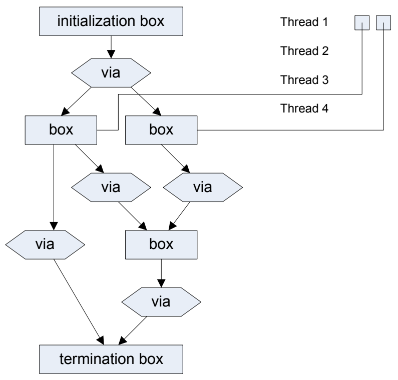
If this went on long enough, a great number of envelopes would be created by



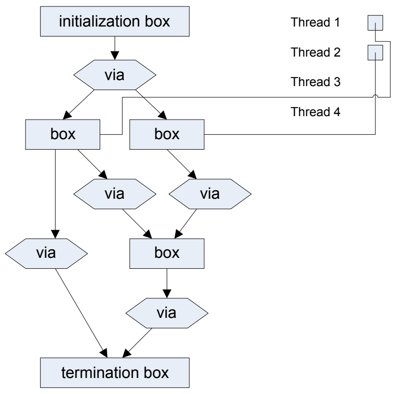
(a) Start



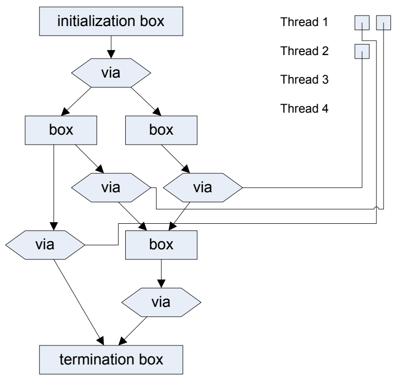
(b) Init finished



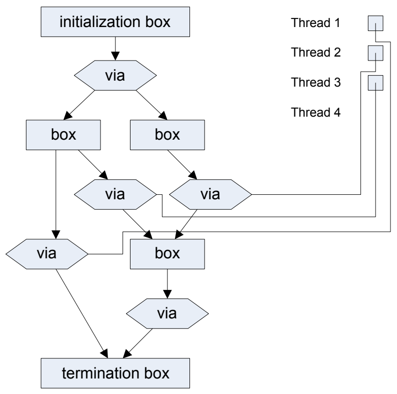
(c) Boxes scheduled



(d) Task stolen



(e) Boxes finished



(f) Another task stolen

Figure 3: Scheduling example

P and remain waiting on the input of C. This could consume a lot of memory – up to the total size of data produced by P. If the data were produced and consumed at a constant rate, it could cope with (small) constant number of envelopes.

To prevent wasting the memory, the size of the input buffer of each box is limited. If the buffer is full (it is *congested*), further data is rejected. When – at a later time – data is removed from the buffer, the via that sends data to this input is notified of this fact and resends the data. This is implemented by enqueuing the via – trying to resend rejected envelopes is default behavior for vias.

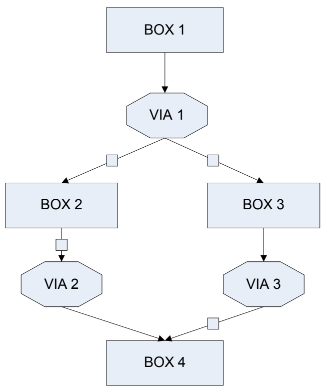
The same mechanism is also implemented in vias – they have a limited input buffer, reject inputs and then "wake up" (enqueue) the source box if the buffer is no longer full.

This way, the congestion can propagate in the direction opposite to the flow of data. As a result, even if the slow box is near the end of the pipeline, it can stall the whole pipeline and limit the number of memory used by the pipeline.

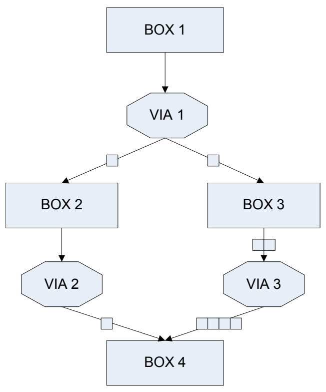
However, there is a downside to this mechanism. Consider a situation with one stream of numbers where the intention is to filter out those numbers that are above the average. The pipeline could split the data into two branches – this is done by a via V with one input and two outputs. One branch does nothing with the data (a "pass-through" branch) and the other computes the average (implemented by a box A). These branches are then combined by a box C with two inputs – the data stream and the average value. This box cannot process the data stream until it receives the average. The box A must wait for the end of the data stream (the poisoned pill) before producing the result. But, if the number of envelopes in the data stream is larger than the total size of buffers on the pass-through branch, the box C would cause congestion on the pass-through branch which will propagate up to the via V and the via V will also start rejecting any further inputs. This is – in effect – a deadlock situation. The box A is waiting for data to be sent by the via V, V is waiting for C to end the congestion, and C is waiting for the average value computed by A.

A similar example for the buffer size of four is shown in the Figure 4 – the small squares represent envelopes stored in a buffer that corresponds to the input that the arrow on which it is drawn leads to. First, the data flows without waiting in the buffers (the Figure 4(a)). Then box 4 starts rejecting inputs, which first start to queue at the input from via 3 to box 4 (the Figure 4(b)) but later fill all buffers on one branch of the pipeline, thus stopping it completely (the Figure 4(c)) until the box 4 starts once again consuming envelopes on its input.

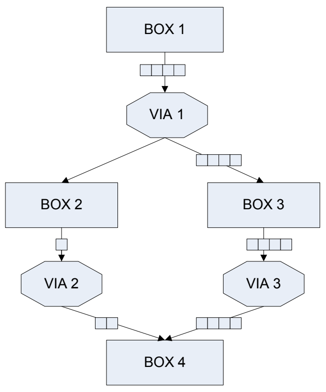
The solution is left to the language module that creates the model. It has to identify such situations and (in the above case) solve it by adding an extra box to the pass-through branch. The new box should accept all incoming envelopes and store them until the poisoned pill is received. This way, the C box does not get congested and the A box can get the complete data and compute the average value. It means that all of the data is kept in the memory, but this cannot be avoided in this scenario. However, there are alternative solutions. For instance, if the stream can be computed twice with reasonable cost, then both branches can have their own source and even if the pass-through branch gets congested, it does not create the deadlock cycle.



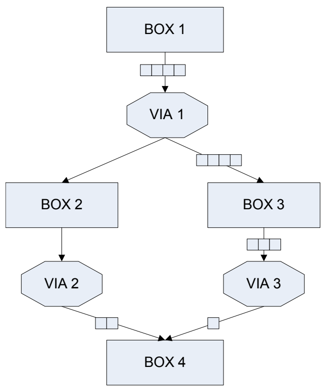
(a) Initial state



(b) Congestion starts



(c) Congested



(d) Congestion disperses

Figure 4: Congestion example

ID	FirstName	LastName	Age
1	John	Doe	32
2	Jane	Doe	28
3	Tom	Smith	51
4	Agnes	Jones	48

(a) Traditional organization

ID
1
2
3
4

FirstName
John
Jane
Tom
Agnes

LastName
Doe
Doe
Smith
Jones

Age
32
28
51
48

(b) Column oriented organization

Figure 5: Different table representations

4 Data level parallelism

The specific way in which the parallel execution is performed by Bobox makes it more useful for data processing scenarios, like database query evaluation or stream processing. This is further supported by the way in which the envelopes and vias are designed.

So far, we considered the envelope to be the smallest piece of data. But it does have an internal structure to store the actual values. Each envelope contains a small table with data in a column oriented way [26]. That is, instead of storing data for one row together, each column is stored separately as a sequence of fields of that column. For an example, see the Figure 5. Each column contains the same number of rows as the others, but each column has its own data type. The list of column data types (*type descriptors*) is a *schema* of the envelope. We require all envelopes that pass through one via to have the same schema. This schema is shared among all envelopes of the same type to conserve space.

The data for one column are stored in one continuous block of memory where each field has a fixed size. This allows for easier data alignment and the use of SIMD (single instruction – multiple data) instructions like SSE. The columns are stored as a set of shared pointers. This allows new columns to be added easily and even allow one column to be used in several envelopes and ensures that it is deallocated when all envelopes that reference it cease to exist. This can be useful in situations where one envelope is sent along two branches of the pipeline and some of the columns are used in both branches but none of them modifies it. If the data is not copied but only shared among the branches, we do not only conserve space and the time required to copy the data, but we also increase the chance that the data is present in the CPU cache.

4.1 Poisoned pill

There is a special type of envelope called a *poisoned pill*. It contains no data (the number of rows is zero). The purpose of this envelope is to inform the boxes that no more data will be sent over the link the pill was received from.

This is the envelope sent by the initialization box to start the pipeline. Other boxes have to be programmed to handle and pass the pill correctly, although they can postpone passing it for a long time.

The poisoned pill also terminates the computation. All pipelines are required to contain one termination box and all paths in the pipeline are required to end with this box. When this box receives the poisoned pill on all of the inputs, the computation ends and the pipeline is deallocated.

The poisoned pill is (from the programmers point of view) the same as an envelope with data, except that it contains no columns, no schema and a flag marks it as a poisoned pill. This is useful for the Bobox architecture, since it requires no special handling (no added complexity) of poisoned pills. Although the poisoned pill is used to mark that no more data will pass through a certain path, this is not used by the execution environment. It is not necessary, since the proper clean-up and correct interpretation of poisoned pills is left to the boxes. The little resources that could be saved by deallocating the boxes as soon as possible (after the poisoned pill was sent to all outputs of the box) would not justify the added complexity and limitation imposed on the design of the boxes.

For instance, a box with one input and output can receive the poisoned pill and immediately pass it to the output, but enqueue itself in the process. The poisoned pill is sent further down the pipeline and at a later time, the box is invoked again and it can do the clean-up at this time without delaying the rest of the pipeline until the clean-up is finished.

4.2 Variable size columns

Some of the columns store values whose binary representation has variable size. For instance, string literals with varying length and unlimited (or large) maximal length. Such strings cannot be stored in an array of same-sized fields. A similar problem exists in row oriented systems, but the whole row (with all the strings stored in the row) is stored in one block of memory. This may seem like a small difference, but when low-level data-intensive computations are concerned, such details become significant. The following sections discuss several approaches to storing variable size columns.

Limited size If the maximal size of the representation of all values is limited by a constant, the data can be stored by extending each value to the maximal size. The problem is that such maximum may not exist or that the average size is much lower and this layout would waste too much space.

Indirect storage The actual value is not stored in the column. Instead, only a fixed size *handle* for the value is stored in the column and the actual value is stored in a dedicated storage. The handle may reflect some properties of the value or be totally independent. For instance, the storage may be set up to detect duplicate entries and create handles in such a way that handle(X)=handle(Y) iff X=Y.

Overflow A modification of the previous technique. The data is split into two columns. The first contains first N bytes of each value and the second stores the rest (if any) in an indirect storage.

Direct storage Usually, each column is stored as a continuous block of memory with a well known size equal to (field size) * (number of rows). However, if we relax this requirement, the variable sized columns could be stored sequentially and directly in the column data block. There are two possible storage strategies: the data can be stored in one large continuous block of memory or it can be split into several smaller blocks. The first option uses less resources, but the other is more flexible, especially when the total required memory is not known in advance or when the data can be modified.

Except for the first option, all storage strategies can handle any data, but each have their individual strengths and weaknesses. At such low level, effects of caches and even virtual memory paging have to be considered.

4.3 Vias

Unlike boxes, whole vias are integral part of the Bobox framework – they contain no user code. They do not interpret the data in any way and pass it from their input to the outputs.

We require all envelopes that pass through a certain via to have the same schema. This implies that all envelopes that a box sends to one of its outputs must have the same schema and also all envelopes that a box receives on an input have the same schema. Different inputs/outputs of the same box may handle different schemas. This is quite natural requirement. If – for instance – a box performs a table join, it receives parts of the two tables on its two inputs and sends parts of the joined result onto its (one) output.

The via is also the only component of the system that can create new envelopes. The new envelope automatically inherits the schema of the via that created it. If a box needs to create an envelope, it must specify the output that the envelope will be sent to. This allows the request to be forwarded to the appropriate via that creates the actual envelope. This mechanism is used to allow the system to use the same schema definition for all envelopes that share the same schema not only from the logical point of view, but also in terms of using the same area of memory instead of each envelope having its own copy of the schema.

If a via has more than one output, a copy of any enveloped received by the via must be created for each of the outputs. However, as we mentioned earlier, making a copy of each column may not be necessary. Instead, the model specifies a vector of boolean values $priv^o$ for each output o. The vector $priv^o$ has the same dimension as the number of columns in the schema of the envelopes handled by the via. If a value on the N-th position $priv^o_N$ is true, the envelope sent to the output o gets a new copy of the N-th column data. Otherwise, the column is shared with the outputs o' where $priv^{o'}_N = false$.

5 Advanced concepts

The following sections describe some of the more detailed information related to the Bobox system that is not essential to the understanding of how the computation is performed.

5.1 Dynamic pipeline

To facilitate execution of recursive XQuery [5], the model instance is allowed to grow (expand) during the execution. The exact way in which the instance can grow is defined by the model, but the actual decision whether a certain expansion should be performed or not is made during the execution of an instance of the model. In the current version, the expansion is triggered when first envelope reaches a specified box. In that case, the box is replace by several other (interconnected) boxes, some of which may be also expandable.

In general, the number of boxes created this way could grow indefinitely. However, since each box requires some resources from a limited pool (e.g. memory space), the actual number of boxes that can be supported by the system is limited. If the number of boxes grows beyond a limit specified by the run-time, the whole request is canceled and all boxes (and vias) released.

In the case of the XQuery processor mentioned earlier, it has been proven, that if the original XQuery program halts, the Bobox version will only required a finite number of boxes.

5.2 Memory allocator

Creation of envelopes involves memory allocation – the most important part being the memory for the data, but other, smaller block of memory are also needed by the system. Traditional memory allocators used by C++ runtime environment allocate memory from one global heap. In a multi-threaded environment, synchronization is required – usually in the form of one global lock that is acquired at the start of the allocation (or deallocation) and freed after the operation has been completed. This lock is a bottleneck for parallel execution and limits scalability – with more threads the chance that two threads try to allocate memory at the same time increases. If they do, one of them is forced to wait and cannot perform any useful work.

To avoid this bottleneck, each thread is assigned one memory allocator that allocates memory from a private heap. This way, no locking is required and memory allocation is fast and easy. On the other hand, memory deallocation is more complex, since there is a significant chance that a block of memory b is deallocated by a different thread T_2 than the thread T_1 that allocated it. In that case, the allocator of T_2 must pass the information about b to the allocator of T_1. This requires some synchronization between T_1 and T_2. But the situation is different than in the case of one centralized allocator – we only have to synchronize a simple information transfer, not the whole allocation and deallocation process. Such transfer can be performed even with a lock-free structure that uses atomic instructions of the CPU. Another advantage is the fact that we only have to consider multi-threaded environment during memory deallocation, not allocation. And while the caller need the allocation to complete before it can continue, deallocation can be easily delayed. So when deallocation of b is requested on T_2, the call can return immediately after passing the information about b to T_1 and the computation performed at that time by T_1 is unaffected. The actual deallocation is performed by T_1 at a later time,

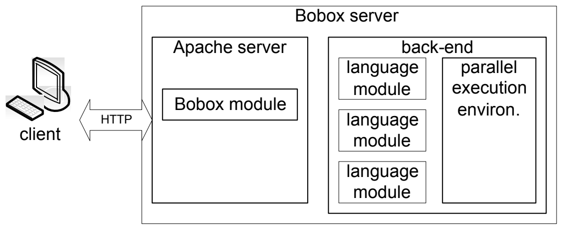


Figure 6: Bobox server

when the thread T_1 makes a call (an allocation/deallocation request unrelated to b) to its allocator.

Another issue with the memory allocator is the allocation strategy. Traditional allocators are general purpose and cannot make any assumptions about the size and number of memory blocks that will be requested. In Bobox, the most significant part of the memory is used for column data. Usually, the size of such blocks (see sections 4 and 4.2) is the size of one field times the number of fields (number of rows carried by the envelope). The field size is often a power of two – either due to the architecture (e.g., a 32bit integer) or design (e.g., a 64byte long string prefix). The same holds for the number of rows. With the boxes we have implemented so far, each envelope produced by a box (with the possible exception of the last envelope before the poisoned pill) has the same number of rows. Furthermore, the number of rows is usually relatively large to reduce the total number of envelopes used for the computation, since each envelope produces some overhead – memory is needed to store the envelope and CPU time is used to pass it around. As a result, the number of distinct block sizes is limited.

This observation can be used by the memory allocator speed up allocation and reduce memory fragmentation. On the other hand, the effects of CPU caches has to be considered even when memory allocation is concerned. When a block of memory is deallocated, it is usually still hot (stored in the CPU cache) and it would be useful to use it as the next block to be allocated. This effect is noticeable especially in the case of small computation and memory intensive models that are able to fit most of their data into the CPU cache. In other cases, the bad cache effects of the normal data access (reads and writes to the data) dwarf effects of the cache-aware allocation.

5.3 Bobox server

So far, we only considered Bobox to be just a parallelization framework for data processing. But there is also a server part of the project, which aims at creating a standalone, multi-purpose database server. It has a modular architecture and can support different data models and query languages. The following parts of the text describe only a general architecture of such server independent on any data model, query language or data storage.

The Bobox server consists of several parts (see the Figure 6). The front-end handles communication with clients and is completely independent on the query language and data format. As we have already mentioned, it is a module for the Apache HTTP server. The back-end consists of the Bobox parallel execution environment and one or more language modules. The language modules translate queries received from the client to the corresponding Bobox model, which is then executed. One module can handle one or more query languages, for example one module may handle SQL language while another module handles both SPARQL and SeRQL.

The language modules are responsible for both translation of the queries and their optimization. For instance, if they perform cost-based optimization and require some statistics, they have to maintain those statistics themselves. They may also cache execution plans (models) for previously translated queries.

All queries are executed by the same execution environment, even if they are initiated by a different client or using a different query language. This means that (among other things) they share the same thread pool. This is a common technique in task parallelism and it prevents the system from executing more logical threads than there are physical cores in the system, which would result in task switching and reduce performance.

To allow interoperability between the server and clients that use different languages and platforms, we decided to use the HTTP as an interface between the server and clients. It is a well defined protocol and it can easily pass various NAT or proxy servers. However, creating a complete and efficient implementation of the HTTP is not an easy task.

Instead of writing the front-end from scratch, we decided to use the Apache HTTP Server [31] and create our front-end as an Apache module. This way, the HTTP communication is handled outside our code and we only need to transfer requests and responses between the Apache module and the Bobox server.

5.4 Implementation

The C++ programming language was chosen for the Bobox system. Some of the main reasons for this choice were:

- generic programming (templates) – for instance the ability to handle different data types with one code without resorting to virtual functions,
- ability to precisely control memory management and even use custom memory allocation methods tailored for specific tasks,
- portability – with carefully written code, the Bobox can be run on different operating system and hardware platforms.

Since the interface between the Bobox server and clients is built on top of the standard HTTP protocol, the programming language and environment used for the server does not limit their choice on the client side.

5.5 The ulibpp library

While the C++ language itself is portable across different platforms and operating systems, different properties of the platforms (for instance the pointer size), systems and APIs can result in different behavior of the same code in different environments. We have designed the ulibpp as part of the Bobox development process to hide these difference and provide a universal interface for all systems and platforms.

For example, it defines integral data types that have the same size on all platforms and compilers. The C++ standard only defines some constraints on these sizes, but in order to create a working database system, exact values are required. Another example is an interface for multithreaded programming – while the pthread library and the Windows API provide very similar services, the programming interface is different. Again, the ulibpp library provides one interface that works the same way on both systems.

5.6 Run analysis

Debugging complex, multithreaded programs is a very complex task. To provide at least some information of what operations the system performed, debugging messages are collected at different points in the system. To minimize the impact of logging on the overall system performance, the logs have a very simple binary format and they are collected and stored at many different places in the system rather than creating a centralized log that would require synchronization. At the end of the computation the logs have to be aggregated from all storages.

The logs are then saved to a file to be viewed offline later. A special tool – the Bobox log viewer – was created to do this. Since it is a general Bobox tool, it mostly logs engine related events, especially related to the scheduler.

Some examples of logged events include:

- a box/via was enqueued,
- the invocation of a box has just begun,
- the invocation of a user specified box code is about to begin,
- an envelope was received by a box,
- a new envelope was created.

The number of logged events is very high. Since they are collected at different places in the system in different threads they must be marked with some kind of a time stamp so that they can be ordered into one linear stream of events. However, the frequency of these events may be higher than the frequency of the clock provided by the system. The system does provide high frequency clock but they can either work incorrectly on multiprocessor machines or they may require too much time to complete.

For these reasons, we decided not to use wall clock as the timestamp but rather a logical clock. It is a global 64 bit integer which is used as a timestamp for the event and incremented. This is done using atomic CPU instructions. Although it requires the variable to be shared among different threads, the effect on the overall performance is minimal.

This way, we can only measure time required by different operations only in number of logged events that occurred during their execution. To create a mapping of events to real time, we have introduced new event – from time to time, we log current “wall clock” time. This way, we get mapping between the logical clock and time in seconds. But we only have mapping for some of the events and have to interpolate the rest.

The tool is intended to be used for debugging and can provide some overall idea of what is going on during the computation. The interpolated time is

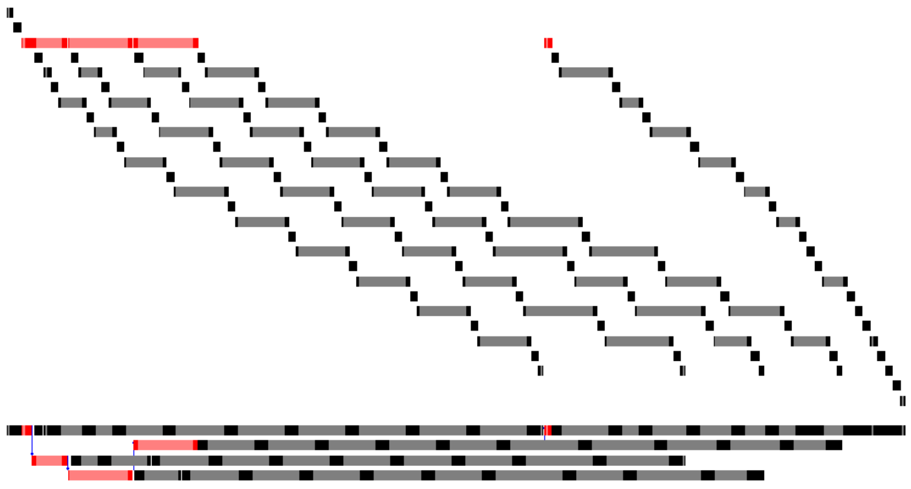


Figure 7: Time line of execution

not accurate enough for performance measurements of individual components, because the pace at which the timestamp increases is far from linear.

The logged events only contain limited information. Besides the timestamp, originator and event type, only two data fields sized 32 and 64 bits are available. This places some requirements on the log viewer since the events do not contain all the data that may be required to display enough usable information. The viewer must simulate the computation of the Bobox system and display information based also on the events that occurred before the displayed event.

For instance, when an envelope is received by a box, the system must check the type of the event that created the envelope. Only by checking this information can it determine whether it was a normal envelope (with data) or a poisoned pill. And since the event that the box received a poisoned pill is very significant, we need to perform this check.

The user interface of the log viewer is rather simple. It can list the events and filter the list by the event originator and event type. It can also display a time-line of the computation. For an example of the time-line see the Figure 7. The four rows at the bottom represent utilization of the four CPU cores used in the execution. Black parts represent the overhead, the gray parts the actual computation. Since we used very small data blocks (1024 rows per envelope), the overhead is large. The top part displays the time periods at which the individual boxes and vias were active – one box or via is represented by one row. They are sorted according to the time of their first execution, which creates the stairs effect. It is easy to see that there were 5 envelopes sent through the system, with the fifth being only a poisoned pill. The box that produces data is highlighted by red color, which shows it is the third row in the top part and that it was executed by all of the four cores (the bottom part), each time producing one envelope with data. The fifth execution at later time means it forwarded the poisoned pill, but only when the first of the CPU cores finished processing the data. Also note that all steps related to one envelope were performed by one CPU core, which is desirable for better cache utilization.

6 Experiments

Even though the implementation process is in its early stages, we have prototype implementations of the key Bobox components, most notably the parallel execution environment and some of the boxes. This allows us to conduct experiments to partially asses performance of the system, even though it cannot yet work as a stand alone server.

6.1 SP^{2}Bench

The system is now able to execute some of the queries of the SP^{2}Bench [24] benchmark in a scenario that closely simulates an in-memory RDF database. The reason for this choice is the fact, that it works with RDF data, which are still "young" compared to other popular database formats like relational databases or XML. This way, we can compare our new system to another one (in our case the Sesame server [8]) that is considered to be among the best in the field yet it is still more of a scientific prototype than an enterprise product with huge financial backing and many years of intense optimizations and tuning. On the other hand, the task is complex enough to demonstrate the performance of Bobox in a scenario that it was intended for – as a data processing server.

These tests were performed on an Intel Core i7 860 processor (2.8GHz, 4 cores, hyper-threading, 4x256KB L1 cache, 8MB L2 cache) workstation with 4GB of memory (DDR3, dual channel, 667MHz) running the 64bit version of Windows 7 operating system. The HDD performance is not significant for this (in-memory) test setup. We also do not need to "warm up" the system (run the query a few times to get data from HDD into memory cache) since no caches of this type are used in a in-memory database. The Bobox parallel execution environment was configured to use only four threads, to eliminate effects of hyper-threading since all Bobox threads do intensive computations. The Sesame benchmarks were performed on a default instalation of Sesame 2.3.1 with a fully in-memory database.

The SP^{2}Bench can be configured to produce data of different sizes. In the following text, we identify the data by the number of triples, e.g. 1M data contain one million triples.

The data is stored in memory and indexed with indexes PSO, POS, OPS, and SPO, where PSO denotes an index on predicate, subject and object is this order. All join operations were performed by merge joins.

Q1 This query is a simple selection of a pattern from a database.

```
SELECT ?yr
WHERE {
  ?journal rdf:type bench:Journal .
  ?journal dc:title "Journal 1 (1940)"^^xsd:string .
  ?journal dcterms:issued ?yr
}
```

The only question in making a query plan in this case is the ordering of the join operations implied by the graph pattern, but even if the triples are located and joined in the order in which they appear in the query, the plan works very well.

On 1M data and five runs, the average time was 23.2 milliseconds (standard deviation 4.38). This result only demonstrates that the total overhead of Bobox is small for simple queries. It would not be reasonable to directly compare it to other systems, since the overhead of e.g. network communication and execution plan generation would affect the other systems' results too significantly. Just to demonstrate, the Sesame can perform the query in 53 milliseconds (std. dev. 8). However, the architecture of the benchmark is much more complicated (it involves Tomcat server and HTTP communication).

Q2 A simple query, but with a large graph pattern.

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle .
  ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc .
  ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page .
  ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL {
    ?inproc bench:abstract ?abstract
  }
}
ORDER BY ?yr
```

Again, the most straightforward evaluation plan works very well, but the amount of data processed by the query is much larger. This results in a much longer execution time. The average time for five runs and 1M data was 1101 milliseconds (standard deviation 164). Note the relatively high std. dev. of the times. This is caused by the complexity of task-parallel execution – the different timing of subsequent runs may result in a different order of task execution and different assignment of task to threads. This does not affect the computed result, but it does affect performance. The Sesame performs the query in 550 milliseconds (std. dev. 16) when measuring the time from the start of the execution to the time first row arrives. The result is large, so waiting for the last row would be too affected by the performance of the transfer.

The effect of scheduling on the (in)stability of execution time is increased by the fact that the test was performed with small chunks of data – the envelopes only contained 1024 rows each. By increasing the number of rows, we reduce the overhead and impact created by scheduling. For instance, by increasing the chunk size four times, the time is 562 milliseconds (std. dev. 12.1).

Q3 and Q4 These queries are aimed at testing the sophistication of the query optimizer. Even though we did try the queries, the results are not given here, since they are not very relevant for our setup.

Q5 Two variants of one query, important is the use of the `DISTINCT` keyword. This is the Q5a version:

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person2 .
  ?person foaf:name ?name .
  ?person2 foaf:name ?name2
  FILTER (?name=?name2) }
```

The average time is 2284 milliseconds (std. dev. 75). The streamed data and merge joins performed quite well in this case. An example of the Bobox model that corresponds to the query is shown in the Figure 8. The Sesame server was unable to complete the query. However, the Q5b version of the query can be evaluated by the Sesame. The query looks like this:

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type bench:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name }
```

The Sesame took 683 milliseconds (std. dev. 13) while the Bobox version takes 486 milliseconds (std. dev. 17).

Q6 This query is in fact a simple "not exists" query.

```
SELECT ?yr ?name ?document
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dcterms:issued ?yr .
  ?document dc:creator ?author .
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document .
    ?document2 rdf:type ?class2 .
    ?document2 dcterms:issued ?yr2 .
    ?document2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr)
  } FILTER (!bound(?author2))
}
```

The average time is 7734 milliseconds (std. dev. 75). Unfortunately, the Sesame server was unable to evaluate this query.

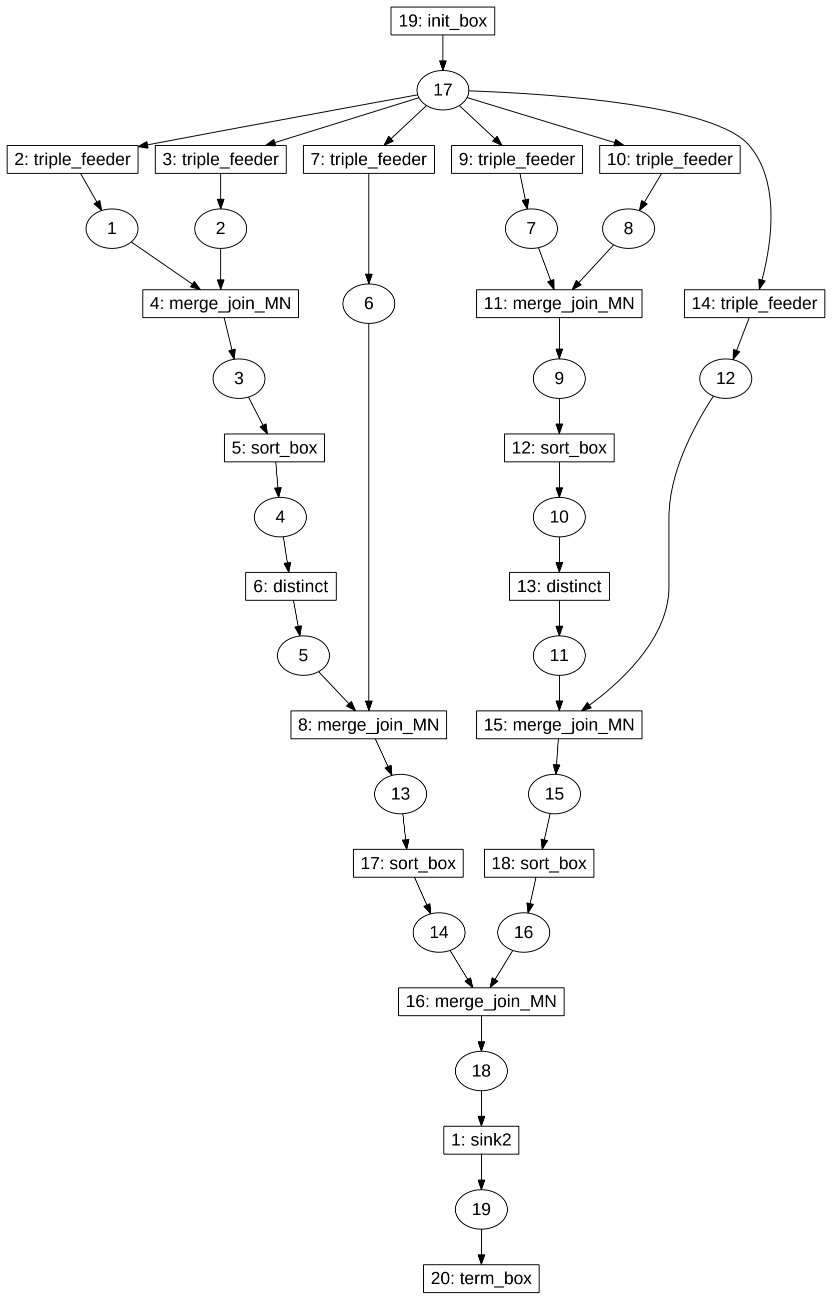


Figure 8: Model for query Q5a

Q7 Q7 is only a slightly more complex variant of the previous query.

Q8 Q8 combines graph patters, filters and union. Unfortunately, we do not have a usable implementation of some of the operation at this time.

Q9 The Q9 query is aimed at testing the query optimizer and the way it handles the `DISTINCT` modifier.

Q10 The query Q10 is extremely simple and is aimed to test selection on objects of triples.

```
SELECT ?subject ?predicate
WHERE {
  ?subject ?predicate person:Paul_Erdoes
}
```

Since we have an index (two of them, in fact) that has the predicate as the first indexed "column", the implementation is trivial and evaluation takes very little time. In fact, most of the runs were completed in times shorter than the resolution of the system timer. Surprisingly, the Sesame took 75 milliseconds to evaluate the query (std. dev. 7), which is slower, even considering the overhead caused by the more complicated test setup.

other queries The remaining queries of the SP^{2}Bench benchmark test variants of the basic `SELECT` query, for instance limit on the number of results or the `ASK` query that tests the non-emptiness of the result. They can either be evaluated in the traditional way and then interpreted differently or handled by the optimizer in a completely different way. But both cases are not relevant to our scenario.

These results of the Bobox implementation have to be interpreted carefully. First, they only represent time used by the query evaluation. In real-world deployment, the query has to be sent to the server, optimized, then it is executed and the results have to be sent back to the client. Second, we have to remember that the queries were translated and optimized by hand, although only using basic operations (merge join, sort, selection, projection, and indexed access) for evaluation.

The most important test in our case is the Q2 query of the benchmark. The reason is that the query takes long enough to evaluate in both Bobox and Sesame to make the results relevant even considering the limited precision of the system clock. Furthermore, it is very simple so the quality of the query optimizer is much less significant and even simple evaluation techniques currently implemented in Bobox are sufficient to handle the query with decent performance. With the initial setup, the Bobox took almost exactly twice the time of the Sesame. The times achieved by the Bobox implementation were less stable than in the case of the Sesame. With some correction of the Bobox parameters, the time gets very close to Sesame. The stability of times is also similar.

We are still working on a SPARQL language module with comprehensive support of the language and a cost-based optimizer. Only when it is completed,

size	time (ms)	std. dev.
1K	1101	164.3
2K	536	21.7
4K	562	12.1
8K	606	6
16K	714	16.8

Table 1: Query performance of Q2 with different envelope sizes

size	time (ms)	std. dev.
1K	2284	80.7
2K	1892	62.1
4K	1794	59.7
8K	2152	141.2
16K	3943	160.4

Table 2: Query performance of Q5a with different envelope sizes

tested and tuned to provide efficient plans, will we be able to do a fair performance test of the complete SP^{2}Bench benchmark.

If we take all these considerations into account, the parallel evaluation engine of the Bobox system provides competitive results compared to the Sesame. RDF databases provide an interesting comparison, since they are relatively young compared to XML or relational databases. This way, the performance of the system is still very significant and is not mitigated by the quality of the query optimizer.

6.2 Envelope size

As we have already noted in the analysis of the Q2 query performance in the previous section, the envelope size (the number of rows stored in an envelope) may significantly affect the overall query performance. At this moment, we only consider situations where the envelope size is the same for all envelopes, although the architecture of the system does not require it.

Tables 1 and 2 show the times that were achieved by the system for queries Q2 and Q5a with different envelope sizes. Both tables show the optimum to be around 2K and 4K sizes. The optimum is affected by two main aspects – scheduling granularity and cache size.

The scheduling granularity works in two opposite directions. Larger envelope size means that tasks take longer to complete and their number is reduced. This reduces the scheduling overhead. On the other hand, longer tasks reduce the amount of time the system is able to run in parallel. For instance, if there is just one box connected to the output of the starting box and it produces a lot of data, no other box can be enqueued and invoked until the first box completes, since no other box has any input data.

The main effect of the cache is that if the data become larger than the available cache, the performance decreases. In simple setups, a single point where the data no longer fit into a cache can usually be easily identified as a sudden drop in performance. Since most CPUs have several levels of caches of increasing sizes (and decreasing speed), several drops are usually present.

# of threads	time (ms)	std. dev.	speedup
1	13416	8.4	1
2	6741	11.0	1.99
4	3708	188.3	3.62
8	3358	607.1	3.99
16	9391	2519.6	1.43
32	17104	3044.1	0.78

Table 3: Query performance for different number of threads

In our scenario, the computations are not that memory access intensive, so the cache effect is insignificant when compared to the effect of scheduling. A lot of processing is performed between reads of consecutive row values, which allows the CPU to use its pre-fetch and speculative evaluation to mitigate the effect of cache misses.

We have also performed several tests that were focused on testing the impact of the CPU caches. These tests showed that the effects of L1 cache are negligible, but the highest level (L2 or L3) cache does affect the optimal size of the envelope. The problem is that this size depends significantly on the task that is being executed, since the ideal situation is that all the data that are actively being processed fit into the (highest level) cache. But the number of such envelopes may differ significantly from task to task and even the memory allocation strategy. At this moment, the cache effect is lower than the effect of the scheduler, so the size of the envelopes is chosen to create a reasonably sized work chunk without regard to the cache size.

6.3 Parallelism

The SP^{2}Bench tests do not work well as tests for parallelization. The problem is that some of the operations (mainly sorts) take significant portion of time and in their current implementations cannot be parallelized. The tests demonstrate the performance of data flow through the pipeline or they could be used to demonstrate the performance of multiple concurrently executed queries.

To test speedup of a single query and limit the influence of other factors, we created an "ideal conditions" test. In this test, we use a linear pipeline with 12 boxes (besides the initialization and termination box). First box produces a stream of data, then there are ten boxes that perform computation intensive operation on those data. The 12th box only receives the data and does no more processing. This is the setup that was used in the time-line example shown in the Figure 7 (in the Section 5.6). The Table 3 shows the results of the test performed on 256 envelopes (with 4096 rows each) for different number of threads. All tests were performed under the same conditions (except the thread number) as SP^{2}Bench tests.

The results are close to what was expected – for two and four threads, there is a significant speedup over one thread. When eight threads are used, the speedup is much lower, due to the fact that the CPU has only four physical cores and the eight logical cores are provided by HyperThreading function. The actions performed by the individual threads are computation-intensive, thus reducing the effect of HyperThreading. Higher number of threads only increases overhead

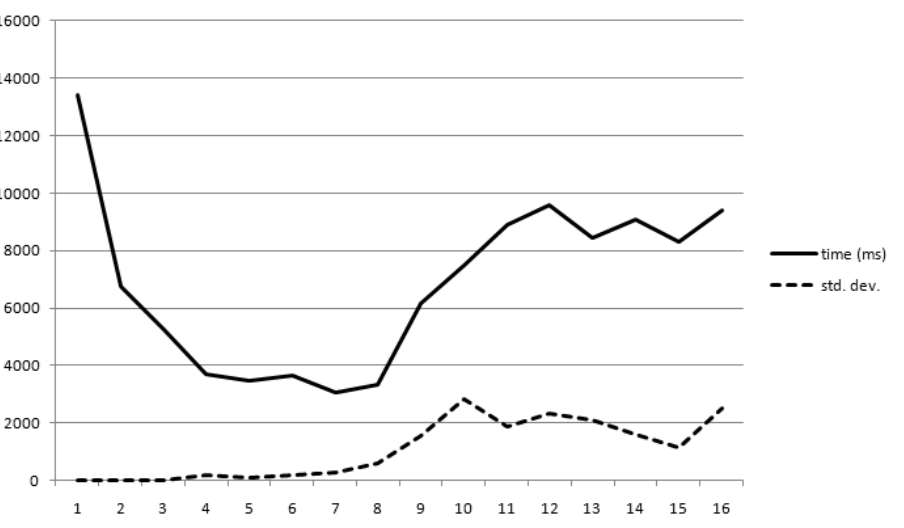


Figure 9: Query performance for different number of threads

while providing almost zero increase in available computational power, since all physical cores are already completely utilized.

The times for number of threads from one to sixteen is shown in the Figure 9. A reasonable number of threads appears to be from 4 to 8. In this test, the best performance was achieved by 7 threads. However, the difference in the 4 to 8 range is too small to make any definitive conclusions, until a more representative set of test models is available. The optimal number of threads can also be affected by using different scheduling strategies. Still, the range from 4 (the number of physical CPU cores) to 8 (number of logical threads of the CPU) appears to be a good option for task-based parallelism.

Also note the decreasing stability of the times with increased number of threads. With one thread, all five runs provided nearly identical times, but with 32 threads, times ranged from 14939 to 22481 milliseconds. This is also quite natural. With just one thread, the system always executes all tasks in the same order. For two or more threads, the order cannot be determined in advance due to, for instance, task stealing, which itself contains a random process (the "victim" is picked at random). Of course, exact timing of the concurrent actions starts to have an effect, as well as process scheduling performed by the operating system.

7 Example

This section demonstrates the way the Bobox framework may be used to perform simple operations on numbers. The following code demonstrates definition of a class of a box that generates a sequence of integers and sends them out in chunks (envelopes) of a specified size. Largest part of the code is concerned with serialization and deserialization of parameters, which is required for serialization of whole models. Model serialization is used for debugging, model storage and transfer.

```
class sequence_box : public box
{
public:
  //required by the generic_box_model (see below)
  static const char* tid(){ return "int_sequence_box"; }
  //parameters of the box model and the box
  struct params
  {
    int begin;       //first number in the sequence
    int end;         //last number in the sequence plus 1
    std::size_t block_size; //size of each chunk

    //standard constructor
    params(int begin, int end,std::size_t block_size)
      : begin(begin), end(end), block_size(block_size){}

    //serialization - converts the parameters into a set
    //of textual key-value pairs
    void accept(box_model_serializer& s) const
    {
      s.add_parameter("begin",begin);
      s.add_parameter("end",end);
      s.add_parameter("block_size",block_size);
    }

    //deserialization - creates parameters from a set
    //of textual key-value pairs
    params(const box_model_serializer& s)
    {
      s.get_parameter("begin",begin);
      s.get_parameter("end",end);
      s.get_parameter("block_size",block_size);
    }
  };

  //standard constructor -- this exact form is required
  //by the generic_box_model
  sequence_box(const box_id_pack_type& id,
    const params& pars)
      : box(id), pars_(pars), current(pars.begin)
  {
  }

private:
  void mach_etwas();  //override of virtual method
  params pars_;       //the parameters
  int current;        //current value of the sequence
};

//convenient way to create a box model is to define
```

```
//the params structure and tid method and use
//generic_box_model like this:
typedef generic_box_model<sequence_box>
  sequence_box_model;
```

The only remaining code required by the box is the method `mach_etwas` that performs the actual computation. The method is defined in the parent class `box` and it is in fact the only code that a box must provide to work withing the Bobox framework. The other requirement is that an appropriate Bobox model must be provided – most methods and fields of the model are defined in the class `box_model_type` that serves as a common parent class for all box models. However, the developer is responsible for code that stores box parameters in the model and then passes them to the box when the model is instantiated. This code is usually very simple and repetitive, so we have provided a generic class that implements all required behavior for a properly constructed box.

In the following listing, note that the envelope that stores the results contains two columns. The actual data (the integral value) is stored as `int` in the second column. The first column stores a flag that determines, whether the current row of the envelope is valid. This is not required by Bobox, but it is a common technique used by boxes and allows the developer to handle deletion of some of the rows without having to move all subsequent items in the memory. Another advantage is, that more such columns can be present, which may be useful if the same envelope is sent through two different branches of the pipeline – the data of each row do not need to be copied, yet each branch can use independent validity flags and treat it as a completely separate envelope in respect to row deletion. On the other hand, the validity flag has to be tested by all boxes that use such envelopes.

```
void sequence_box::mach_etwas()
{
  //consume the poisoned pill that caused the box to be
  //executed the first time
  if (input_data_[0]) consume_input(0);

  //no action if all data has already been sent
  if (current>=pars_.end) return;

  //create envelope for the generated data
  envelope_ptr_type res(
    create_envelope(0,pars_.block_size));

  //the number of the current row of the envelope
  std::size_t rownum(0);

  //loop which generates the data and saves them
  while (rownum<pars_.block_size && current<pars_.end)
  {
    //set the row validity flag to "valid"
    *res->template cell<rowtype_column_policy>(0,rownum)
      = envelope::RT_VALID;
    //store the data
    *res->template cell<int_column_policy>(1,rownum)
      = current;

    ++current;  //move to next number in the sequence
    ++rownum;   //move to next row
  }

  //if the envelope hasn't been filled completely
  //mark the remaining rows as invalid
  while (rownum<pars_.block_size)
  {
    *res->template cell<rowtype_column_policy>(0,rownum)
      =envelope::RT_INVALID;
    ++rownum;
  }

  //send the envelope to the via that is bound to the
  //first output of the box
  forward_envelope(outarc_index_type(0),res);

  //if the end of the sequence has been reached,
  //send a poisoned pill.
  //otherwise, the box requests to be scheduled
  //again at some time in the future
  if (current>=pars_.end)
  {
    forward_envelope(outarc_index_type(0),
      create_poisoned_pill());
  }
  else run_again_=true;
}
```

Most of the code deals with managing the numeric sequence and storing it into the envelope. A noteworthy piece of code is the last `if`-condition in the algorithm. If the end of the sequence has been reached, a poisoned pill is sent to the output to notify subsequent boxes that no more data will be generated. If the end has not yet been reached, the code sets the `run_again_` flag that notifies the Bobox scheduler, that the box has still some work to do, even if no further data is received on any of its inputs (which is what normally causes a box to be enqueued). The scheduler enqueues such box immediately, which means that it will be executed some time in the future.

The `sequence_box` is an example of one common type of boxes – data generators. Another significant type are boxes that receive an envelope on their input, modify the data stored in the envelope and then send the envelope to their output. The following piece of code shows the `mach_etwas` method of a box that adds a constant to integers stored in a specified column.

```
void inc_box::mach_etwas()
{
  //read the input envelope
```

```
    envelope_ptr_type env = input_data_[0];

    //if it is not a poisoned pill, perform the addition
    if (env->type_==envelope::ET_DATA)
    {
      //iterate the rows
      for(std::size_t rownum=0;
         rownum<input_data_[0]->requested_row_count_;
         ++rownum)
      {
        //perform the addition
        *env->cell<int_column_policy>(
           column_index_type(pars_.col),rownum
          ) += pars_.delta;
      }
    }

    //send the envelope to the output
    forward_envelope(0,env);

    //notify the system, that the input has been processed
    consume_input(0);
}
```

Note that when programmed this way, the validity flag for the row is ignored. This can be done, since performing the addition on the deleted row will not cause an exception to be thrown – it is still a valid piece of memory and the addition has no reason to fail. It means that unnecessary operations are performed, on the other hand, such loop may be vectorized by the compiler, so the time is actually wasted only in cases where several successive rows are invalid. Alternatively, the programmer could write the code to explicitly use some vector instructions.

8 Related work

The most similar project to the Bobox run-time is the TBB library. The TBB was one of the first libraries that extensively focused on task level parallelism and it provides a powerful set of structures, algorithms and run-time environment. Compared to the Bobox library, it is low-level solution – it provides basic algorithms like parallel for cycle or linear pipeline and very efficient task scheduler. The developers are even able to directly create task for the scheduler and create their own parallel algorithms. But the tasks are designed in a way that makes it very hard to create a non-linear pipeline similar to the one Bobox provides. Such pipeline may be necessary for complex data processing [5]. Bobox also provides more services for data passing and flow control.

The latest version of OpenMP also provides a way to execute tasks in parallel, but it provides less features and less control than TBB. The OpenMP library is mainly focused on mathematical computations – it can execute simple loops in parallel really fast, it can also run blocks of code in parallel, but it is not well suited for parallel execution of a complex structure of blocks. Unlike TBB or Bobox, it is a language extension and not just a library. This means that the compiler is well aware of the parallelization and optimize the code better, but it also enables OpenMP to provide features that cannot be done with just a library, like defining the way variables are shared or private among threads with a simple declaration. In TBB such variable has to be explicitly passed to an appropriate algorithm by the programmer. In Bobox, it must either be explicitly passed to the model or sent using an envelope at run-time.

But there are also completely different approaches to parallelization. One way would be to create a thread for each box and via in the model instance. This would also ensure that each box or via is running at most once at any given time. However, this is considered a bad practice [23]. There are two main reasons for not using this architecture. First, it creates a large number of threads, usually much larger than the number of CPU cores. Although it forces the operating system to switch the threads running on a core, it may not impact the overall performance that badly, since it can be arranged that the idling threads (those assigned to a box or via that is not processing any data at the moment) are suspended and do not consume any CPU time. The second problem is that when data (envelopes) are transfered from one box to another, there is very little chance that it would still be hot in the cache, since the thread that corresponds to the second box is likely to be scheduled to a different CPU, that does not share its cache with the original one. The concept of tasks used by TBB and Bobox avoid these problems and the use of thread pool, fixed number of threads and explicit scheduling gives developers of the libraries better control of parallel execution.

8.1 Parallel Databases

Within the context of databases, three kinds of parallel environment are usually defined under the following traditional names:

- *Shared-memory* database systems utilize *symmetric multi-processing (SMP)* architectures where shared memory is the fastest means of interprocess communication. Due to the prevalence of multi-core CPU's, this architecture is currently the default withing a wide range of applications.
- *Shared-disk* environment is usually a cluster of computers connected to shared disk space using a *storage area network (SAN)*. In an orthodox shared-disk environment, nodes communicate and synchronize solely using disk I/O. In reality, network communication or even shared memory may be used; the main difference between shared-memory and shared-disk architecture is the cost of passing data between nodes. In this sense, *non-uniform memory access (NUMA)* systems may be treated similarly to shared disk environment, whenever the cost of sharing memory by distant nodes is comparable to disk access.
- *Shared-nothing* environment consists of a set of nodes equipped with local disk storage where partitioning of the data set is inevitable. A shared-nothing database system usually includes a mechanism of failure isolation and recovery.

The Bobox project is currently aimed at the shared-memory environment.

In a relational database management system, parallelism may be employed at various levels of its architecture:

- *Inter-transaction parallelism.* Running different transactions in parallel has been a standard practice for decades. Besides dealing with disk latency, it is also the easiest way to achieve a degree of parallelism in shared-memory or shared-disk environment. Although it is not considered a specific feature of parallel databases, it must be carefully considered in the design of parallel databases since parallel transactions compete for memory, cache, and bandwidth resources [20, 30].
- *Intra-transaction parallelism.* Queries of a transaction may be executed in parallel, provided they do not interfere among themselves and they do not interact with external world. Since these conditions are met rather rarely, this kind of parallelism is seldom exploited except for experiments [10].
- *Inter-operator parallelism.* Since individual operators of a physical query plan have well-defined interfaces and mostly independent behavior, they may be arranged to run in parallel relatively easily. On the other hand, the effect of such parallelism is limited because most of the cost of a query plan is often concentrated in one or a few of the operators [29].
- *Intra-operator parallelism.* Parallelizing the operation of a single physical operator is the central idea of parallel databases. From the architectural point of view, there are two different approaches:
  - *Partitioning* [11]. This technique essentially distributes the workload using the fact that many physical algebra operators are distributive with respect to union (or may be rewritten using such operators). Although it may be realized inside an operator independently of the others, in most scenarios, the same partitioning scheme is applied to a portion of query plan to the extent which preserves the selected partitioning key. In shared-nothing environments, partitioning applied to a query is dictated by the partitioning of the persistent data. Abstractly speaking, partitioning is a transformation of query plan based on multiplication of operators and inserting partitioning/merging operators where necessary. After this transformation, the parallelism is achieved by the means of inter-operator parallelism (note that partitioning is considered a method of intra-operator parallelism because of its behavior with respect to the original operators). Skew in the distribution of computation complexity is the major problem of partitioning and different approaches to skew-resistance create the wide variety of partitioning methods [7, 2, 18].
  - *Parallel algorithms.* Implementing the operator using a parallel algorithm usually offers the freedom of control over the time and resource sharing and machine-specific means like atomic operations or SIMD instructions. However, designing, implementing, and tuning a parallel algorithm is an extremely complex task, often producing errors or varying performance results [3]. Moreover, the evolution of hardware may soon make a parallel implementation obsolete [15]. For these reasons, parallelizing frameworks are developed [9].

Besides these traditional techniques of parallel databases, the *MapReduce* approach gained significant attention. While it is often considered a step back

[25], compared to the partitioning mechanisms of parallel databases, there are application areas where MapReduce may outperform a parallel database [6]. On the distant end, MapReduce is being used in connection with the nebulous paradigm of cloud computing [28]; on the near end, MapReduce is being inserted into parallel SQL-based systems as a means of extensibility [12]. Although originally targeted at shared-nothing or shared-disk environments, MapReduce was already implemented and studied in shared-memory settings [22, 16].

The central principle of Bobox allows parallelism among boxes but prohibits (thread-based) parallelism inside a box. This is similar to inter-transaction and inter-operator parallelism; however, a box does not necessarily correspond to a relational operator. In particular, Bobox allows the same approach to partitioning as in parallel databases, using transformation of the query plan.

Bobox does not allow parallel algorithms to be implemented inside a box (except of the use of SIMD instructions). Therefore, individual single-threaded parts of a parallel algorithm must be enclosed in their boxes and the complete algorithm must be built as a network of these boxes. This is certainly a limitation in the expressive power of the system; on the other hand, the communication and synchronization tasks are handled automatically by the Bobox system.

9 Future work

At this moment, we have an interface between the boxes and the rest of the system and there is an experimental implementation of the run-time environment. Although the benchmarks show that it provides promising performance, there is still much space for an improvement. There is no fully working language module for any language, but we are actively working on SPARQL and XQuery modules. The latter is already capable of transforming some of the queries into a working Bobox model.

Besides parallel processing on one node, the system is designed to allow even distributed processing. Parts of the model instance may be distributed over several nodes with vias serving as the boundaries between the nodes, providing communication. However, the situation is more complicated than just transferring envelopes between nodes. The Section 4.2 dealt with variable size columns and some of the solutions involved storing part of the data outside the envelope which means that transferring just the envelopes may not be enough. This is part of the problem caused by not having one shared memory – we have to carefully consider what information is available on certain nodes and make sure that the boxes have all the information they need. The possible solutions range from always transferring all data to a distributed hash table.

10 Conclusions

We have designed a framework for parallel execution of data-intensive computations. It is designed to leverage both task level and data level parallelism, thus providing easy to use interface for developers, good cache utilization and the use of vector instructions. The Bobox system is well suited for database query evaluation but it may be used in other applications, for instance stream processing or data transformation.

The main difference from other parallelization libraries is the fact that the parallel execution is controlled by the flow of the data through the pipeline. While it limits the set of the problems where Bobox is applicable, it allows us to move lot of the work related to flow control, data passing and scheduling from the user's code to the Bobox framework.

Even an early version of the run-time environment combined with a naïve implementation of the boxes can perform queries from the SP^{2}Bench with performance comparable to the Sesame server. The experiments have also shown that on a machine with four physical cores the system scales well when the number of worker threads rises from one to four – the speedup for four cores is 3.62.

References

[1] Advanced Micro Devices, Inc.: The Industry-Changing Impact of Accelerated Computing (2008). http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf

[2] Afrati, F.N., Kyritsis, V., Lekeas, P.V., Souliou, D.: A new framework for join product skew. CoRR **abs/1005.5732** (2010)

[3] Aguilar-Saborit, J., Muntes-Mulero, V., Zuzarte, C., Zubiri, A., Larriba-Pey, J.L.: Dynamic out of core join processing in symmetric multiprocessors. In: PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 28—35. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/PDP.2006.31

[4] Akhter, S., Roberts, J.: Multi-Core Programming. Intel press (2006)

[5] Bednárek, D.: Bulk evaluation of user-defined functions in XQuery. Ph.D. thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague (2009)

[6] Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In: SIGMOD '10: Proceedings of the 2010 international conference on Management of data, pp. 975–986. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1807167.1807273

[7] Bouganim, L., Florescu, D., Valduriez, P.: Dynamic load balancing in hierarchical parallel database systems. In: VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases, pp. 436–447. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)

[8] Broekstra, J., Kampman, A., Harmelen, F.v.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, pp. 54–68. Springer-Verlag, London, UK (2002)

[9] Cieslewicz, J., Ross, K.A., Satsumi, K., Ye, Y.: Automatic contention detection and amelioration for data-intensive operations. In: SIGMOD '10: Proceedings of the 2010 international conference on Management of data, pp. 483–494. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1807167.1807221

[10] Colohan, C.B., Ailamaki, A., Steffan, J.G., Mowry, T.C.: Optimistic intra-transaction parallelism on chip multiprocessors. In: VLDB '05: Proceedings of the 31st international conference on Very large data bases, pp. 73–84. VLDB Endowment (2005)

[11] DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM **35**(6), 85–98 (1992). DOI http://doi.acm.org/10.1145/129888.129894

[12] Friedman, E., Pawlowski, P., Cieslewicz, J.: Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. Proc. VLDB Endow. **2**(2), 1402–1413 (2009)

[13] Khan, M.F., Paul, R., Ahmed, I., Ghafoor, A.: Intensive data management in parallel systems: A survey. Distributed and Parallel Databases **7**(4), 383–414 (1999)

[14] Khronos Group: The OpenCL Specification (2009). http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[15] Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core cpus. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)

[16] Kovoor, G.: MR-J: A MapReduce framework for multi-core architectures. Ph.D. thesis, University of Manchester (2009)

[17] Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in Intel Threading Building Blocks. Intel Technology Journal **11**(04), 309–322 (2007)

[18] Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In: SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing, pp. 75–86. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1807128.1807140

[19] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2 (2009). http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[20] Morvan, F., Hameurlain, A.: Dynamic memory allocation strategies for parallel query execution. In: SAC '02: Proceedings of the 2002 ACM symposium on Applied computing, pp. 897–901. ACM, New York, NY, USA (2002). DOI http://doi.acm.org/10.1145/508791.508964

[21] OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0 (2008). http://www.openmp.org/mp-documents/spec30.pdf

[22] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp. 13–24. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/HPCA.2007.346181

[23] Reinders, J.: Intel Threading Building Blocks. O'Reilly (2007)

[24] Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. CoRR **abs/0806.4627** (2008)

[25] Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: Mapreduce and parallel dbmss: friends or foes? Commun. ACM **53**(1), 64–71 (2010). DOI http://doi.acm.org/10.1145/1629175.1629197

[26] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented DBMS. In: VLDB '05: Proceedings of the 31st international conference on Very large data bases, pp. 553–564. VLDB Endowment (2005)

[27] Valduriez, P.: Parallel database systems: open problems and new issues. Distributed and Parallel Databases **1**(2), 137–165 (1993)

[28] Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. SIGCOMM Comput. Commun. Rev. **39**(1), 50–55 (2009). DOI http://doi.acm.org/10.1145/1496091.1496100

[29] Wilschut, A.N., Flokstra, J., Apers, P.M.G.: Parallel evaluation of multi-join queries. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pp. 115–126. ACM, New York, NY, USA (1995). DOI http://doi.acm.org/10.1145/223784.223803

[30] Zhang, Z., Trancoso, P., Torrellas, J.: Memory system performance of a database in a shared-memory multiprocessor (2007). URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.1924

[31] The Apache Software Foundation. http://www.apache.org