Zbyněk Falt

Towards Efficient Parallel Data Processing on Modern Hardware

Department of Software Engineering

Supervisor of the doctoral thesis: RNDr. Jakub Yaghob, Ph.D.
Study programme: 4I2 Software Systems

Prague 2014
I would like to thank everybody who supported me in my studies and in the work on this thesis. I appreciate the help of my supervisor Jakub Yaghob who gave me valuable and inspiring comments which helped me solve many problems. I would also like to thank my colleagues from our department, especially David Bednárek, Martin Kruliš, and Filip Zavoral, for their help, remarks, great cooperation and friendly atmosphere.

Last but not least, I thank my family and friends who supported me in the work and had endless patience with me while I was sitting at a computer.
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date ............
Title: Towards Efficient Parallel Data Processing on Modern Hardware

Author: Zbyněk Falt

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Yaghob, Ph.D., Department of Software Engineering

Abstract:

Parallel data processing is a very hot topic in current research, since the amount of data and the complexity of the operations performed on them has been increasing significantly in the past few years. In this thesis, we focus on a specific domain of this research – the design and implementation of parallel algorithms used mainly in database systems. First, we introduce important enhancements in the Bobox system, which is a framework for the development of parallel data processing applications. Then, we introduce a new domain specific language called Bobolang which makes the implementation of those applications easier. Next, we propose parallel and scalable algorithms used in the domain of databases, namely sort and merge join, and introduce their efficient implementation using the combination of Bobox and Bobolang. Finally, we introduce parallel runtime for SPARQL engine as an example of a parallel data processing application which demonstrates the main contributions of this thesis in complex and real-life situations.

Keywords: Bobox, Bobolang, parallel algorithms, SPARQL
4 Parallel Algorithms for Data Processing

4.1 Introduction ................................................. 53
4.2 Filter Operator ............................................. 53
4.3 Nested Loop Join Operator ................................. 55
  4.3.1 Results .............................................. 56
4.4 Sort Operator .............................................. 57
  4.4.1 Single-threaded Sorting Algorithm .................... 57
  4.4.2 Parallel Sorting Algorithm ........................... 58
  4.4.3 Parallel Merging of Streams ......................... 58
  4.4.4 Results .............................................. 61
4.5 Merge Join Operator ...................................... 64
  4.5.1 Join without Duplicates .............................. 64
  4.5.2 Join with Duplicates ................................ 66
  4.5.3 Results .............................................. 69
4.6 Related Work .............................................. 72
  4.6.1 Sort .................................................. 72
  4.6.2 Merge Join .......................................... 73
4.7 Chapter Summary and Future Work ....................... 74

5 In-memory Parallel SPARQL Engine ....................... 77

5.1 Introduction .............................................. 77
5.2 Implementation of the Engine ............................ 78
  5.2.1 Query Compilation and Optimization ................. 78
  5.2.2 Representation of RDF Terms ....................... 79
  5.2.3 Representation of RDF Database .................... 79
  5.2.4 Format of Envelopes ................................ 79
  5.2.5 Join Operator ...................................... 80
  5.2.6 Left Join Operator ................................ 80
  5.2.7 Scan Operator ..................................... 80
  5.2.8 Sort Operator ...................................... 81
  5.2.9 Distinct Operator .................................. 81
  5.2.10 Union Operator ................................. 81
  5.2.11 Slice Operator ................................. 82
5.3 Results .................................................... 82
  5.3.1 Inter-operator Parallelization ....................... 82
  5.3.2 Intra-operator Parallelization ....................... 83
  5.3.3 Comparison with other Engines ...................... 84
5.4 Related Work ............................................. 85
5.5 Chapter Summary and Future Work ....................... 86

6 Conclusion .................................................. 87

List of Figures ................................................ 99

List of Listings .............................................. 101

Appendix A ................................................... 105

Appendix B ................................................... 119
1. Introduction

In the past few years, multiprocessor systems have become a standard in the world of computers. These systems contain multiple execution units; therefore, they usually offer more computation performance than traditional uniprocessor systems. Unfortunately, executing a single-threaded application on multiprocessor computer does not benefit from its performance, since it utilizes only one of the available processing units.

The development of applications or algorithms which are able to run efficiently on modern computers is very actual topic in the current research. This research basically involves several distinct areas like the research on parallel algorithms, the development of frameworks, libraries, languages, or tools which simplify the development of parallel applications on various architectures.

This thesis partially covers all these areas. We introduce enhancements of a parallel framework and a new language suitable for the development of the specific class of parallel applications. We also propose several parallel algorithms and their efficient and scalable implementation which utilizes the enhanced framework as well as the language. Finally, we introduce an in-memory SPARQL [112] engine as an example of parallel data processing application, which was designed as a proof of concept and which demonstrates benefits of our innovations.

1.1 Motivation

The development of parallel applications is much more complex and error-prone than the development of single-threaded applications. The first challenging task is the design of parallel algorithms which the application uses. Although there exist many patterns for parallel programming [101, 103] which help with the design of the algorithm, none of them is universal enough to make this task simple.

The second challenging task is the efficient, scalable, and functional implementation of such algorithms. Basically, there are two approaches – the implementation in general purpose programming language (GPL) such as C/C++, Java, C# or in some domain specific language (DSL) for parallel programming.

In case of GPL, the developer can directly use API for creating, manipulating, and synchronization of threads provided by operating system or runtime library like Pthreads [37], Windows API [9], Java threads [109] etc.. However, this low-level approach is usually tedious and error-prone. In order to simplify the implementation of parallel programs in GPL, many libraries and frameworks exist. The most popular ones like Intel Threading Building Blocks [114], OpenMP [11], or MPI [67] help significantly since they provide a high-level API for easier implementation of common parallel patterns. Despite the fact that these libraries are really helpful, the developer still has to introduce the parallelism into the application explicitly.

In case of DSL, the developer does not have to take the parallelism into account, since the parallel evaluation of the application is implicitly ensured by the compiler, interpreter, or the runtime. On the other hand, the use of such languages is limited only for particular problem domains [127].

One representative of the group of domain specific languages are streaming
languages. The streaming languages are based on the idea that all data are represented as data streams, i.e., flows of tuples. These data streams are processed by operators (sometimes denoted as filters or kernels) which should transform input streams into the required output streams. These streams are typically considered to be infinite; however, in this thesis we focus on streams with a finite length.

Streaming languages typically depend on a runtime which is responsible for the evaluation of the operators and management of the streams. In the rest of this thesis, we use the term streaming system to denote that runtime. From this point of view, the streaming language is only a syntactic sugar which enables specification of applications for streaming systems.

Figure 1.1 shows an example of a streaming application. It reads data from two independent sources by the source operators. Their resulting streams are merged together by the merge operator, filtered by the filter operator, and printed by the print operator.

![Figure 1.1: Example of a streaming application.](image)

The advantage of streaming systems is that they naturally introduce inter-operator parallelism, i.e., the operators may be executed in parallel with each other. Therefore, the developer typically provides the implementation of the operators and their mutual connections (sometimes called an execution plan) and the streaming system ensures that the operators are processed concurrently if possible.

The main disadvantage of the streaming systems is that they are suitable only for some domains since not all data may be represented as data streams. However, data such as multimedia, samples from sensors, or relational data can be represented as data streams.

Notice that the combination of streaming systems and relations is natural, since query plans used for the processing of relations can be considered as streaming applications. Moreover, this combination is useful since the streaming systems introduce inter-operator parallelism automatically. In this thesis, we focus primarily on this area and on development of a parallel data processing engine on the top of a parallel streaming system.

### 1.2 Objectives and Contributions

The main objective of this thesis is to introduce new techniques, methods, approaches and algorithms for parallel data processing in streaming systems and to show their applicability by introducing real-life parallel processing application which utilizes them. In order to reach this objective, we performed several consecutive steps which also determine the structure of this work.

First of all, we focused on a streaming system called Bobox [28] which has been developed at Charles University [15]. It is a parallel framework which sim-
plifies the design and the implementation of parallel applications and serves as an experimental platform for the development of parallel algorithms [61]. In Chapter 2, we propose multiple enhancements and optimizations of Bobox which have significant positive impact on the efficiency and applicability of the system. Consequently in Chapter 3, we introduce a new domain specific language called Bobolang for the specification of execution plans.

This thesis focuses primarily on the design and implementation of parallel algorithms for data processing. We got inspiration in the domain of database systems and focused on the algorithms commonly used there. Chapter 4 presents our results in this area. Some of the algorithms (like filtering or nested loop join) are very easy to parallelize and they are intended mainly to illustrate the usage of Bobox and Bobolang in simple, but real applications. Therefore, the main contribution of that chapter is to introduce parallel sorting algorithm and parallel merge join algorithm.

In Chapter 5, we combine the innovations from the previous chapters together and use them to implement a parallel runtime for SPARQL engine. We briefly introduce the implementation of the operators which are required by the SPARQL algebra [47] and several rather technical, but notable details which influence the performance of the engine. Finally, we perform detailed experiments in order to show that the main objective of the thesis was fulfilled. Chapter 6 concludes and summarizes our work.

The summarize, the main contributions of this work are as follows:

- Design of a new task scheduler for the Bobox system.
- Implementation of a new user friendly API for the development of applications for Bobox.
- Introduction of a new domain specific language for streaming systems called Bobolang.
- Scalable parallel sort algorithm and its implementation in Bobox/Bobolang.
- Scalable parallel merge join algorithm and its implementation in Bobox/Bobolang.
- Implementation of a parallel runtime for the in-memory SPARQL engine.
2. Bobox

2.1 Introduction

Bobox [28] is a parallelization framework developed at the Department of Software Engineering of the Charles University. It was primarily intended as a runtime for evaluation queries over (not only) relational data and it was successfully used in the pilot implementation of XQuery [26] and TriQuery [27] engines.

Bobox can be considered as the implementation of a streaming system, since it assumes that the developer provides the implementation of operators (called boxes) and their mutual connections as an execution plan. However, the use of the original version of Bobox was very limited and challenging. Therefore, we continued in that work and introduced new features such as support for I/O operations, locality aware task scheduler and memory allocator. Additionally, we implemented a new and more applicable API for the development of operators.

2.2 Background

2.2.1 Architecture of Multiprocessor Systems

Before we focus on Bobox and its enhancements, we briefly describe the architecture of modern multiprocessor systems in order to establish the terminology used in the rest of this thesis.

For the sake of simplicity, we first focus on uniprocessor systems. These systems contain single processing unit, i.e., one single-core physical processor which has a uniform access to the main memory, i.e., the whole memory has the same latency and bandwidth from the view of the processing unit.\footnote{Naturally, there are some exceptions like memory mapped I/O, but it is not relevant for this thesis.}

In order to reduce the number of accesses to the (relatively slow) main memory, processors contain caches. The main objective of the caches is to temporarily store recently (and frequently) accessed data in a faster memory with a lower latency and a higher bandwidth. However, such memories are expensive and hard to produce; therefore, processors typically contain multiple levels of caches – the lower the level of the cache is, the faster but the smaller it is. Common computers typically contain two or three levels of caches (L1, L2 and L3, respectively). The LL cache (the last level cache) denotes the highest (the largest, but the slowest) level of the cache in a processor, i.e., either the L2 or L3 cache.

In opposite to the uniprocessor systems, multiprocessor systems contain more than one processing unit. There are several distinct methods (which might be arbitrarily combined) of adding the units to a system:

1. increasing the number of physical processors,
2. increasing the number of cores within the physical processor,
3. creation of virtual cores.
Increasing the number of physical processors is a straightforward solution. However, as the number of them within one system is increasing, shared resources are turning into bottlenecks. The most critical shared resource is the main memory. Therefore, Non-Uniform Memory Access (NUMA) systems were developed as a solution to this problem. The NUMA system basically consists of several NUMA nodes. Each node acts as a standalone system, i.e. it has its own local and uniform main memory. However, these nodes are connected together by a bus and the whole system shares the physical address space. Therefore, one node can access local memory of another node, but this access is slower than the access to its local memory. This slowdown is known as NUMA factor. The NUMA factor tells how many times slower is the access to the non-local memory in comparison to the access to the local memory. Notice that we may consider the Symmetric multiprocessing (SMP) as a NUMA system with a single NUMA node.

The creation of virtual cores is a method widely used in processors by Intel and is known as Hyper-Threading Technology [99]. In this case, two virtual cores per one physical core are created. These virtual cores share execution units as well as cache memories. This sharing of the execution units allows better utilization of them, since they are rarely fully loaded by a single processing unit. Therefore, the computation power of a processor with Hyper-Threading technology is slightly higher than the computation power of the same processor without that technology (up to 30% according to [99]). Notice that processors by AMD use similar technology in the Bulldozer microarchitecture. In this case, only the floating point unit is shared by a pair of virtual cores [82].

Notice that the structure of caches in multi-core processors is not as straightforward as in uniprocessors, since multi-core processors may contain several distinct cache memories for each cache level and these cache memories might be variously shared by processing units.

A simplified schema of a NUMA system is depicted in Figure 2.1. This system
has two NUMA nodes each with two dual-core physical processors and each core contains two processing units.

### 2.2.2 Bobox Architecture

In this section, we briefly describe the architecture of the Bobox system. Notice that this section is not the result of this thesis, since it describes the architecture and API of the initial version of Bobox [28].

The overall structure of Bobox is shown in Figure 2.2. The most important part of the system are operators (called boxes) which are responsible for the data processing. Each box may have arbitrary number of inputs and outputs. The main task of boxes is to transform their input streams into resulting output streams.

![Bobox Architecture Diagram](image)

Figure 2.2: Bobox architecture.

The data streams are implemented as flows of data units called *envelopes*. Each envelope corresponds to a sequence of tuples. Notice that the tuples are stored by columns, i.e., the envelope internally consists of several equal-sized columns which contain the data. This representation enables advanced manipulation with the data (see Section 2.6.2). The Figure 2.3 shows an example of an envelope. Notice that envelopes might also contain scalar data in addition to the tuples.

Besides envelopes containing data, we distinguish one special envelope called *poisoned pill*. The poisoned pill is sent after the last valid data envelope of a stream and denotes the end of that stream.

All envelopes are allocated using the memory allocator which is described more detailed in Section 2.5.

When an envelope is sent to a box, the envelope is inserted to the corresponding input buffer of the box and the box is scheduled. This tells the task scheduler
to select a thread from a thread pool which executes the box. The exact scheduling algorithm is quite complex since it influences the performance of the system significantly; therefore, its detailed description is in Section 2.4.

The request manager receives the execution plans (called requests) and is responsible for their evaluation. The requests are managed through two special boxes which must be contained in every execution plan:

- **init** box – this is the first box (in a topological order) of the plan and initiates the evaluation of the plan.

- **term** box – this is the last box (in a topological order) and its completion denotes that the execution plan was completely evaluated.

The evaluation of the given execution plan is started by the scheduling of the **init** box. The only thing the box does is sending the poisoned pill to its outputs. This causes that its successor is scheduled and executed. On the other hand, the only responsibility of the **term** box is to wait until it receives the poisoned pill from all its inputs and then tell manager that the execution plan (i.e., the request) was completely evaluated.

Notice that the execution plans for Bobox have no source and no sink in a classical sense. However, we may consider boxes which transform the empty input to a data stream as sources and boxes which transform a data stream into empty output as sinks.

### 2.2.3 Bobox API

As we already mentioned in previous section, the main objective of boxes is to transform their input streams into the output streams. Recall that streams are implemented as the flows of envelopes. Therefore, another point of view is that box is responsible for the processing of envelopes it receives and for the producing of output envelopes. In this section, we briefly describe the API for the development of boxes available in the initial version of the system.

This API was straightforward – Bobox provides an abstract base class with one pure virtual member, the **body** function. This function is called by the scheduler when the box receives an envelope. The purpose of the method **body** is obvious – it should process the received envelopes and send resulting envelopes to the
output respectively. Therefore, the developer had to derive this base class and implement this function.

For example, the simplified body of a box which receives streams of integers and removes all composite numbers from that stream is listed in Listing 2.1 (see the filter.box class in Appendix B for the real C++ code). Notice that the function try_pop_envelope returns an envelope from the given input buffer. If there is no envelope it simply returns null. However, if the box has exactly one input then it is safe to assume that this call always succeeds since the body function was called because an envelope was received.

Listing 2.1: Implementation of the filter box in the initial version of the Bobox API

```cpp
// an envelope for storing prime numbers
envelope envelope_to_send = create_data_envelope();

virtual void body() override
{
    // pop the envelope from the input 0
    envelope received_envelope = try_pop_envelope(0);

    if (received_envelope->is_poisoned_pill()) {
        if (!envelope_to_send->is_empty()) {
            // send the envelope to the output 0
            send(envelope_to_send, 0);
        }
        // send the poisoned pill to the output 0
        send(create_poisoned_pill(), 0);
    } else {
        for (int i=0; i<received_envelope->get_size(); i++) {
            // get the i-th integer of the envelope
            int number = received_envelope->get_integer(i);

            // if the number is prime, add it to the output 0
            if (is_prime(number)) {
                envelope_to_send->append_integer(number);

                // if the envelope is full, send it to the output 0
                if (envelope_to_send->is_full()) {
                    send(envelope_to_send, 0);
                    // and allocate a new one
                    envelope_to_send = create_data_envelope();
                }
            }
        }
    }
}
```
2.3 Enhancements in the Bobox API

2.3.1 API for the Stream Processing

The initial API described in Section 2.2.3 is suitable for the development of simple boxes. The main problems arise when the box is more complex or has more than one input. Consider a box which merges two streams of integers. Such box needs to read from its inputs on demand, i.e., when the box processes one envelope from one input completely, it needs to wait until another envelope arrives on that input.

The first problem is that the body function is called whenever an envelope is received on any input. Therefore, the body function might be called several times before an envelope on the desired input is received. This causes increased overhead of the task scheduler.

We solved this problem by the introduction of the prefetch_envelope(input) function. This method causes that the body function is not called until the box receives an envelope on the given input. Therefore, when the body function is called, the developer can rely on the fact that all required input envelopes are available for the processing.

The second and more significant problem is that the developer needs to save all local variables and the position within the body function (i.e., the inner state of the box) before the processing of another envelope. The reason is that the only way of waiting for another envelope is to call the prefetch_envelope function appropriately and to return from the function body.

Therefore, we introduce the wait function. The wait function tells the scheduler that the box has nothing to do until all prefetched envelopes are received, i.e., the current worker thread is available for execution of another box. This function is implemented using fibers [6, 3]. Each box is associated with its own context (i.e., stack, instruction pointer, etc.) and when the task scheduler decides to execute a box, it simply switches to the context of that box. The wait function (or returning from the body function respectively) switches back to the context of the scheduler.

The wait function together with the prefetch_envelope function is the most important enhancement introduced in the Bobox API. It allows the implementation of another important function, the pop_envelope function which is listed in Listing 2.2.

Listing 2.2: Implementation of the pop_envelope function

envelope pop_envelope(input)
{
    envelope result = try_pop_envelope(input);
    if (result) {
        return result;
    }
    prefetch_envelope(input);
    wait();
    return try_pop_envelope(input);
}

This function always guarantees that a valid envelope is returned. If there is
any in the input buffer, the first call of the `try_pop_envelope` function succeeds and the envelope is returned. Otherwise, the function prefetches an envelope on the given input, waits until it is received and then the second call of the `try_pop_envelope` succeeds.

Thanks to the `pop_envelope` function, we could encapsulate input and output streams as a special classes `input_stream` and `output_stream`. These classes logically transform the stream of envelopes into the stream of tuples. The simplified source code of the `input_stream` class is listed in Listing 2.3 and the source of the `output_stream` class in Listing 2.4.

Listing 2.3: Implementation of the `input_stream` class

```cpp
class input_stream {
    envelope current_envelope;
    int position = 0; // position in the current envelope
    int size = 0; // size of the current envelope

    // checks if there is next valid tuple in the input stream
    bool has_next() {
        if (position == size) {
            current_envelope = pop_envelope(input);
            if (current_envelope->is_poisoned_pill()) {
                return false;
            }
        }
        position = 0;
        size = current_envelope->get_size();
        return true;
    }

    // returns the next tuple of the input stream
    tuple next_tuple() {
        return current_envelope->get_tuple(position++);
    }
};
```

Thanks to these classes, we can implement the example from Listing 2.1 much easier as shown in Listing 2.5 (see the `merge_box` class in Appendix B for the real C++ code which uses these classes).

In fact, this version is slightly less efficient than the original one, since the encapsulation of streams introduce some additional overhead. But it is the price for the easier development of boxes. On the other hand, this slowdown is not significant according to the experiments and the introduced functions/classes do not prevent from the using of the original API.

### 2.3.2 API for Blocking Calls

The main objective of the Bobox framework is data processing, i.e., it is expected that boxes reading the source data or writing the output data perform many I/O
Listing 2.4: Implementation of the `output_stream` class

```cpp
class output_stream {

    envelope current_envelope = create_data_envelope();

    // appends the tuple to the output stream
    void append(tupel)
    {
        current_envelope->append_tuple(tupel);
        if (current_envelope->is_full()) {
            send(current_envelope, output);
            current_envelope = create_data_envelope();
        }
    }

    // closes the output stream
    ~output_stream()
    {
        if (!current_envelope->is_empty()) {
            send(current_envelope, output);
        }
        send(create_poisoned_pill(), output);
    }
};
```

Listing 2.5: Implementation of the `filter` box in the enhanced version of Bobox API

```cpp
void body()
{
    // create an input stream for the input 0
    input_stream input(0);

    // create an output stream for the output 0
    output_stream output(0);

    while (input.has_next()) {
        int number = input.next_integer();
        if (is_prime(number)) {
            output.append_integer(number);
        }
    }
}
```

operations. I/O operations are usually blocking, which means that the calling thread is blocked by the operating system until the requested data are read from or written to storage, network etc.

The waiting for the completion of I/O request typically does not consume computation resources, since the most time is spend by the waiting on a response from a data storage, from a remote server etc.. This causes that some processing units might be temporarily unused during I/O request despite the fact that there could be some boxes waiting for the execution.
To avoid this situation, Bobox provides functions `detach` and `attach` \[90\], which may be called from the body function of a box. The `detach` function tells the task scheduler that the box is about to perform a blocking call, while the `attach` function informs that the blocking operation has finished.

The `detach` function increases the number of worker threads in the thread pool, whereas the `attach` function decreases this number. Therefore, if every blocking operation is correctly enclosed by these functions, the number of active threads remains constant and the parallelism is not limited.

Notice that another possible solution would be to use asynchronous I/O interface provided by most operating systems. However, I/O operations are not the only blocking calls. For example, waiting until an additional parallel device such as GPU or Xeon Phi finishes its work is blocking as well. Additionally, the implementation of the `detach/attach` functions is very simple and much more portable as shown in Section 2.4.3.

### 2.4 Task Scheduler

The task scheduler is the critical part of the Bobox system which influences its performance significantly. In Section 2.2.1, we briefly described interaction between the boxes and the scheduler, i.e., that the task scheduler is responsible for the execution of boxes which have envelopes ready for processing. In this Section, we focus on this interaction more precisely and describe the scheduling algorithm in detail.

#### 2.4.1 Factors Influencing Efficiency

First of all, we analyse several factors which influence the efficiency of the system from the view of task scheduler and which we took into account when designing the final scheduling algorithm.

**Caches**

The effective utilization of caches can significantly increase the performance of any application. Bobox does not analyse the real implementation of the body function; therefore, it cannot predict which data and when the box accesses (i.e., which envelopes from which input, which columns of that envelopes, etc.). However, we can basically assume that a box accesses its incoming envelopes and that the envelopes sent by a box are hot in cache, since they were created recently by the box.

Therefore, one possible optimization is to try to run the consumer of an envelope on the same processing unit as its producer. However, even this simple rule is hard to enforce, since one producer may have multiple consumers or multiple producers may have one common consumer.

Additionally, the producer can produce data faster than the consumer can process, i.e., the envelopes are stored in an input buffer and eventually forced out of cache instead of being immediately processed. This problem is partially solved by the limitation of the size of the input buffers. When a buffer becomes full, the producer of the envelopes is suspended until at least one envelope from the
buffer is processed. This increases data locality and also leads to lower memory consumption, since there is only a limited number of unprocessed envelopes which occupy the memory at a time. On the other, limited input buffers may cause a deadlock during the evaluation of an execution plan \[130\].

The cache hierarchy is also important factor which influences the performance. Modern multi-core systems have various hierarchies, i.e., physical processors have the different number of cache levels, the different sizes of caches and the different ways of sharing caches among the processing units. For example, if a producer has several consumers, it is better to execute these consumers on processing units which share some cache memory with the processing unit on which the producer was executed.

Another factor which influences the effective utilization of the cache memories is the size of envelopes. The smaller the envelopes are, the higher the probability that they remain hot in cache until they are processed is. On the other hand, the smaller the envelopes are, the higher overhead with their transfers is since the task scheduler has to deal with more tasks. The detailed analysis of the optimal size of envelopes was researched in \[29\]; however, no particular solution was proposed in that paper.

**NUMA Factor**

In order to speed up the evaluation of execution plans on NUMA systems, processing units should access preferably their own local memory. In the opposite case, the evaluation is slowed down by the NUMA factor. Therefore, the efficient scheduler has to avoid the situation when data allocated on one node are processed on another node.

**Hyper-Threading**

Hyper-Threading increases the performance of a physical processor; however, task scheduler must be careful when choosing a processing unit for the execution of a task. Consider situation when we have two tasks and one dual-core processor with Hyper-Threading technology. Running these tasks on processing units that do not share execution units is much faster than running them on processing units which share them. This load balancing is typically performed well by the operating system; therefore, usually a developer does not have to take it into account.

However, if the task scheduler manages which worker thread run on which processing unit (by setting the affinities for that threads \[8, 10\]), it may restrict the operating system from doing that load balancing which can lead to performance loss \[13\].

### 2.4.2 Tasks Creation and Their Types

From the view of the task scheduler, each box can be in one of three states:

- **waiting** – The box is waiting for the prefetched envelopes.
- **scheduled** – All prefetched envelopes are ready in input buffers and the box is scheduled for the execution.
• running – The box is processing the envelopes.

When a box is switched to the scheduled state, a new task which will execute the box is created and passed to the task scheduler.

Notice that the reception of an envelope is not the only situation when a task is created. For example if the box finishes the processing and all prefetched envelopes are already ready in input buffers (i.e., they were received before or during the execution of the box), then the box is immediately scheduled again.

Additionally, if no envelope is prefetched during the function body (Section 2.3), the box is scheduled immediately when the body function is left (or the wait function called). This behaviour is the consequence of the definition of the prefetch_envelope function, i.e., that the box is scheduled when all (i.e., even zero) prefetched envelopes are available in the corresponding input buffers. This can be used to yield execution to another box after the sending of an envelope (typically to its consumer). A task can also be created when a box stops to be suspended because the input buffer of its consumer was full.

Therefore, Bobox distinguishes two types of the tasks in order to achieve better performance [29]:

• The immediate tasks are associated with the processing of input data, i.e., with the reception of an envelope. This type of tasks is expected to be executed as soon as possible and preferably by the same thread which spawned it or by a thread which is close (in the hierarchy of cache memories) to that thread since its input data are probably hot in cache.

• The deferred tasks are those which are not directly bound to any data; therefore, they can be basically executed by any thread. On the other hand, running them on processing unit close to the unit on which it was created may increase performance, since the inner state (e.g., data structures, call stack, instructions etc.) of a box which corresponds to that task might be hot in cache.

Notice that each task belongs to exactly one request (i.e., execution plan). This is very useful information since tasks from different requests do not share any data. Therefore, it is safe to keep tasks from one request running on one physical processor while tasks from another request on another physical processor without performance loss. Moreover, it is useful to do that, since data from one request do not force data from the other request out of cache since physical processors do not share caches.

2.4.3 Implementation

In this section, we describe the implementation of the task scheduler [60] based on the ideas described in Section 2.4.1.

Data Structures and Scheduling Algorithm

During the initialization of the task scheduler, the host system is analysed and configuration and properties of the system such as the hierarchy of the cache memories or NUMA factors detected.
Processing units which share at least one level of cache are grouped together in processing groups and one thread pool per group is created. Each thread pool has as many threads as there are processing units available in the corresponding group and the affinity of those threads is set to this group. Notice that the affinity of a thread is set to the whole group, not to a single processing unit; therefore, it does not prevent the operating system to perform load balancing on physical processors with Hyper-Threading technology.

Each processing group maintains one queue of immediate tasks per processing unit and one queue of deferred tasks shared by the whole group. The deferred queue is in fact more complex data structure than a simple queue, since it maintains tasks of each request separately. Moreover, these requests are kept sorted from the youngest to oldest one.

We define the distance between two processing units within one processing group and the distance between two groups. Distance of units within one group is equal to the lowest level of cache which these two units are sharing. Distance of two groups is equal to the distance of their corresponding NUMA nodes (i.e., equal to the NUMA factor) and zero for groups that share a NUMA node.

The main objective of our scheduling algorithm is to increase data locality as much as possible, i.e., when a thread is searching for a next task to execute, it tries to select a task which is as close as possible to the task executed previously. Therefore, the thread first detects on which processing unit it is currently running\(^2\) and than the first applicable rule of the following list is taken:

1. The youngest task from the queue of immediate tasks of the current processing unit.

2. Other processing units of the same group are scanned (in the increasing distance) and the first non-empty immediate queue is found. If such queue exists, its oldest task is taken.

3. The youngest deferred task of the oldest request from the shared queue of deferred tasks of the current group is taken. This rule ensures that all threads of one group work on the same (the oldest) request if possible.

4. Other processing groups are scanned (in increasing distance) and the first non-empty shared queue of deferred tasks is found. If such queue exists, the oldest deferred task of the second oldest request is taken. If the queue has tasks of only one request, its oldest deferred task is taken instead. This rule ensures that each core group works on its own request if possible. However, it does not prevent the situation that the whole system cooperates on one common request.

5. Immediate queues of the processing units from other groups are scanned. If non-empty queue is found, its oldest task is taken. The immediate queues are scanned in round robin manner and the thread remembers the last non-empty queue found. When this rule is applied again, the scan is resumed where it previously ended, i.e., when the thread steals a task from another

\(^2\)This is done by calling \texttt{sched\_getcpu} on Linux or \texttt{GetCurrentProcessorNumber} on Windows.
processing unit, it tries to steal a task from the same unit again since it may have the context of the tasks created on this unit partially hot in cache.

If all steps fail (i.e., there is no available task to execute), the thread is suspended, so that it will not consume system resources (see next section for more details).

When a thread creates a task, it first determines on which processing unit it runs. The immediate tasks are inserted to the immediate queue of that unit and the deferred tasks are inserted to the shared queue of deferred tasks of the corresponding group.

When a new request is created, the init box of the corresponding execution plan is scheduled as a deferred task and this task is inserted to the queue of deferred tasks of the least busy group. As the least busy group is considered that group which has most suspended threads.

**Suspending and Resuming Threads**

When a thread does not have another task to process, it tries to suspend itself on a synchronization primitive shared by its processing group; however, the thread is not allowed to suspend if there is at least one task of either type in its group. This ensures that all tasks in a group are always processed and that any thread is not suspended uselessly.

As the synchronization primitive, we use a standard semaphore accompanied by an atomic variable which holds information such as the number of running threads, the total number of tasks in all task queues of the core group etc. This solution scales well, since there is no global synchronization primitive which would limit the scalability of Bobox on highly parallel systems.

When a thread creates a task, it attempts to wake up one of the suspended threads. First, it tries to wake up a thread in the same core group. If all threads in the group are running, it scans all other groups and attempts to wake up a thread there. The groups are scanned in an increasing distance from the original group and the search finishes when a group with suspended thread is found or all groups are scanned, i.e., all threads are running. We do not take a possibility of a race condition into account, since they are compensated by the combination of the algorithm for task stealing and thread suspending.

**Support for Blocking Operations**

As mentioned in Section 2.3.2, Bobox supports calling of blocking operations within the function body through pair of functions detach and attach. The detach function adds one thread to the thread pool of the current core group and the attach function removes one thread from that pool.

The implementation of these functions makes use of the fact that all threads in one thread pool are identical, i.e., they have set the same affinity and they do not own any data structure, since the queues of immediate tasks belong to the processing units, not to the threads.

---

3We use one 64b atomic integer, which is partitioned to multiple sub-integers using bitwise and shift operators.
Therefore, we are allowed to create more threads per thread pool than the number of corresponding processing units and manage their count through the synchronization primitive described in the section above. The only modification of this primitive was that we enhanced the atomic variable by the limit of allowed threads. This limit is initially set to the number of processing units in the core group.

The `detach` function increases this limit and wakes up a thread eventually, while the `attach` function decreases it. The group is responsible that the number of running threads is not greater than the limit, i.e., it does not allow to resume a thread when this limit is reached.

Additionally, when a thread finishes a task, it first checks whether the number of running threads is not greater than this limit and if so, the thread suspends itself. This means that the `attach` function may suspend a different thread than the `detach` function woke up; however, it is not relevant since all threads are identical.\(^4\)

Furthermore, more threads than the allowed limit might be temporarily active at a time. This can be easily solved by the call of the `wait` function immediately after the `attach` function, i.e., by the yielding execution (Section 2.4.2); however, according to our experience it is not necessary.

Notice that the number of additional threads per thread pool is passed as a parameter to Bobox during its initialization.

### 2.4.4 Results

We performed multiple experiments to prove that the new scheduling algorithm significantly improved the performance of the system. For the testing of the scheduling algorithm from Section 2.4.3, we used our parallel implementation of the in-memory SPARQL engine (see Chapter 5) and the SP\(^2\)Bench benchmark\(^5\) with its 5m testing dataset. The implementation of the engine is able to generate parallel execution plans without significant serial bottlenecks, i.e., worker threads are heavily utilized during their evaluation. Additionally, the SP\(^2\)Bench benchmark contains several queries which generate various and really complex query execution plans (their description and execution plans can be found in Appendix A). This is useful since this variety shows the behaviour of the task scheduler under various circumstances.

We selected queries Q2, Q4, Q5a, Q5b, Q6, Q7, Q8, Q9, and Q11 from the benchmark, since they take reasonable time to evaluate and their query execution plans are complex enough. Other queries are evaluated so fast that the results would be negligible (see Section 5.3.3 which contains the results for all queries).

We used two hardware configurations for the experiments:

- A server (Dell PE1955) with two Intel Xeon E5310 processors, both running at 1.60Ghz. This type of processor has 4 cores and two shared 4MB L2 caches. First two cores share the first L2 cache, second two cores share

\(^4\)Except some data such as stack hot in cache, but according to our experiments this performance loss is negligible in comparison with the performance gain.

\(^5\)We used smaller dataset for some queries in order to avoid memory swapping or to ensure that the query is evaluated in a reasonable time. Such queries are always appropriately labelled in the results.
the other. Additionally, each core has its own L1 cache (32kB + 32kB). This configuration represents a non-trivial SMP system and our scheduling strategy creates 4 thread pools for this configuration. The size of the main memory is 8GB. In the rest of this thesis, we denote this server as SMP8.

- A NUMA server (Dell PEM910) with four Intel Xeon E7-4820 processors, all running at 2.0Ghz. This type of processor has physical 8 cores with Hyper-Threading Technology, i.e., each processor has 16 processing units in total. Each core has its own L1 cache (32kB + 32kB), L2 cache (256kB) and all cores share one L3 cache (18MB). This configuration represents non-trivial NUMA system and our scheduling strategy creates 4 thread pools as well. The size of the main memory is 128GB. We denote this server as NUMA64. Notice that because of Hyper-Threading technology, the maximal expected speed-up of parallel algorithms is according to [99] about \(32 \cdot 130\% = 41.6\times\).

Both systems are running Red Hat Enterprise Linux 6.2. We used gcc 4.8.2 with -O2 switch for the compilation.

We performed two different experiments for each hardware configuration:

- We run the selected query only once. This experiment demonstrates the situation when there is many various data dependencies among the tasks, since all tasks belong to one request.

- We run the selected query multiple times in parallel (16 times in all measurements). This experiment simulates a real database system which should be able to process multiple queries at a time and demonstrates the situation when there are many tasks which do not have any dependency on each other, i.e., each thread can process its own instance of the query without any cooperation with the others.

Finally, we used two different schedulers for each experiment:

- The scheduler which implements the scheduling strategy described in the Section 2.4.3. We denote this scheduler as Locality Aware Scheduler - LAS.

- The scheduler from the initial version of Bobox [29] which was based on the TBB scheduling strategy. Each thread keeps its local queue of immediate task and all threads share one queue of deferred tasks. A thread executes the first existing task in this order: the latest immediate task from its local queue, the oldest deferred task from the shared queue and the oldest immediate task from the local queue of another thread. In other words, immediate tasks are handled in the same manner as the spawned tasks in TBB [16] and deferred tasks are handled in the same manner as the enqueued tasks in TBB. The scheduler does not keep the tasks of distinct requests separated. We denote this scheduler as Reference Scheduler - REF.

We run each testing scenario 10 times and the result is the arithmetic mean of the last 5 runs in order to eliminate fluctuations caused by the cold start of the application and by the memory allocator which must stabilize its caches of super blocks first (see Section 2.5).
SMP System

The results for the SMP8 system are shown in Figure 2.4 for single query and in Figure 2.5 for multiple parallel queries.

Single query  As expected, the REF scheduling algorithm performs well on a SMP system. However, queries Q4 and Q6 benefit from LAS and especially query Q2 is almost \(2 \times\) faster. This query consists of one long pipeline; therefore, it is especially sensitive to the data locality. Other queries contain such pipelines as well; however, these pipelines are typically split to multiple independent parts because of sorting operators which break up the pipeline processing and cause that the execution plans are evaluated by parts, i.e., that all threads cooperate close to each other (see Appendix A).

Multiple queries  The second experiment shows that the LAS performs better than REF in more cases than in the first experiment. The main reason is that the LAS better separates individual requests, i.e., the requests do not force out each other from cache memory.

NUMA System

Both experiments on the NUMA system (see Figure 2.6 and Figure 2.7) proves that taking the NUMA factor into account is very important in modern systems. The main problem is that accessing memory of another node slows down both communicating nodes and the system bus.

LAS tries to keep one request on one NUMA node as much as it is possible. If it is not possible, it tries to keep different branches of execution plans on different nodes.
NUMA nodes which minimizes data interference between the NUMA nodes. REF does not distinguish among the NUMA nodes; therefore, the relationship between a thread and the memory being accessed is almost arbitrary. This is significant especially in the multiple queries case, when for example query Q2 is more than $3 \times$ faster and queries Q4, Q5a, Q5b, Q8, and Q9 more than $2 \times$.

All performed experiments proved that the locality awareness has a significant impact on the performance of the parallel streaming system running on either SMP or NUMA systems.

### Blocking Operations

Additionally, we performed an experiment which proves that the support of blocking operations is also very important. We prepared two types of requests for this experiment:

- **I/O requests**, which write an *infinite* stream of numbers to a file,
- **computation requests**, which simulate a processing of *finite* stream (1 million of 32b integers, each integer is 500$\times$ multiplied by a random number). Because the performance of the `rand()` function in the standard library turned out to be very poor in parallel environment, we used the linear congruent generator \[ \text{Knuth} \] to obtain random numbers.

The idea of the experiment is to run both types of requests in parallel and measure the influence of the I/O requests on the time needed for the evaluation of the computation requests. In each scenario, we first ran $N$ parallel I/O requests (with increasing $N$) and then 16 parallel computation requests. We measured the time needed for the evaluation of the computation requests. We used scheduler on
the SMP8 system with 8 additional threads for each core group (i.e., \(8 + 4 \times 8 = 40\) threads together) and ran each scenario in two settings:

- **IO-unaware**, when I/O requests do not use those functions,
- **IO-aware**, when I/O requests use functions `detach/attach` correctly.

The results are shown in Figure 2.8. For \(N = 0\), the result are same as expected, however, with increasing \(N\) the IO-unaware setting becomes slower in contrary to the IO-aware setting which is almost independent on the \(N\). For \(N = 8\), the evaluation of computation requests takes significant more time since all worker threads are utilized mainly by I/O requests (which started before computation request and therefore they have higher priority, see Section 2.4.3)). Recall that the I/O requests are infinite; therefore, also the computation requests take asymptotically infinite time. The same situation occurs also in the IO-aware setting but with significantly higher \(N\).

### 2.5 Memory Allocation

During the development of the Bobox system, we identified two problems with the default memory allocations in C/C++, i.e., the `malloc/free` functions (or the `new/delete` operators respectively).

The default allocation policy of virtual memory in Linux and Windows is the first touch/node local policy [24, 92], i.e., when a thread first accesses an unmapped page it is mapped to a physical frame that preferably belongs to the node on which the thread is running\(^6\). However, this does not solve a problem which typical allocators have – they do not keep the track of locality of the allocated heap blocks [81]. Therefore, if a heap block is first allocated on the node A (and first touched on that node), deallocated on the node B and reallocated on the node B, than this heap block is obviously not local to the node B. This causes that an application converges to the state when heap blocks are returned independently on their locality.

The second problem is that when a virtual memory is allocated, the operating system has to create the mapping between the virtual page and physical frame

---

\(^6\) Another commonly used policy is the interleave policy, when frames are mapped in a round-robin manner among the nodes. This policy is used when a memory is expected to be accessed from multiple nodes.
and clear that frame. According to our observations, this process does not scale well in parallel environment which is significant especially on Windows operating system.

Notice that deallocated blocks might be hot in cache, since their data were probably accessed before their deallocation. Therefore, the memory allocator should return preferably recently deallocated blocks in order to increase data locality and thus the utilization of caches.

Because of these problems, we decided to implement our own memory allocator in Bobox in order to achieve better performance of the system. This allocator is NUMA-aware and tries to reuse already allocated virtual memory as much as possible.

2.5.1 Implementation

The memory allocator used in Bobox for the allocation of envelopes is already described and analysed in detail in our previous work; therefore, we focus only on its main aspects in this thesis.

Blocks larger than 512kB are allocated/deallocated directly using system functions for allocating virtual memory. We suppose that blocks with such size are requested rarely, since the preferred size of an envelope is always smaller than 256kB (see Section 2.6.3).

For the blocks larger than 8kB but smaller than 512kB, the allocator has prepared several fixed-size blocks sub-allocators. The sizes of these blocks (and the number of the sub-allocators, respectively) are chosen so that internal fragmentation of the allocator is kept less than approximately 7 percent.

When a new block is requested, appropriate sub-allocator is chosen (that with the smallest size of blocks larger or equal to the requested size) and a free block from this sub-allocator is returned. If there is no such block, it allocates super block and uses it as an additional memory pool. The free blocks are kept in a linked list; therefore, allocation and deallocation can be done quickly. Additionally, the linked list ensures that recently deallocated blocks are returned in LIFO manner, i.e., the allocator returns recently deallocated blocks preferably which increases the data locality.

Each allocated block has a small header with the information about the sub-allocator and the super block to which it belongs. When all blocks from a super block are deallocated, then the super block is also deallocated. Notice that the size of super blocks is approximately 10MB.

The allocations/deallocations of super blocks are served by the super block allocator. Each NUMA node has one such allocator and allocates memory preferably from that node. The allocator is also responsible for the reusing of the super blocks.

Blocks smaller than 8kB are allocated in the similar way as the previous blocks; however, we implemented several optimizations in order to reduce the internal fragmentation and the memory overhead associated with the management of small blocks.

\footnote{I.e., using functions \textit{mmap/munmap} or \textit{VirtualAlloc/VirtualFree}}

\footnote{Last in, first out}
The implementation of the allocator eliminates the need of one global lock through the replication of memory pools; therefore, the allocation/deallocation requests are not serialized as for example in Visual Studio C Run-Time Libraries (CRT) \[5\].

Notice that the memory allocator does not handle a situation when there is not enough memory to fulfil an allocation request. In that case, evaluation of the execution plan simply fails with an appropriate error code. This behaviour might be unacceptable for systems such as databases when the complete evaluation of each valid request should be guaranteed. If such behaviour is unacceptable, the evaluation plan of the application must be modified in order to use operators which are able to work correctly with the limited memory so that all parallel requests fit into the main memory. For example sorting should use some external sort algorithm etc. However, this is the responsibility of the query compiler/optimizer and thus beyond the scope of this thesis.

### 2.5.2 Results

In order to show benefits of the Bobox allocator, we performed an experiment which compares its performance with the performance of default system allocator (i.e., \texttt{malloc}/\texttt{free}). In this experiment, we reused the scenario for NUMA64 and SMP8 system, with multiple queries and LAS scheduler (see Section 2.4.4) and ran it two times – once with default allocator and once with Bobox allocator.

The results for NUMA64 are shown in Figure 2.9. Query Q4 benefits from the new allocator most (about 35%). This query is very sensitive to data locality (see the results in Figure 2.7 for comparison). Additionally, it produces very large result set (approximately 18 millions tuples); therefore it makes use of super block reusing.

Query Q2 also benefit significantly, since it is about 15% faster. This query is sensitive to data locality as well, but does not consume as much memory; therefore, the speed up is caused mainly by the NUMA-awareness of the allocator.

The results for other queries are comparable (but the new allocator is never worse). This is caused by the fact that NUMA factor is slightly eliminated as a result of locality aware scheduler which reduces accesses to the main memory.

On SMP8 (see Figure 2.10), no result is significant; however, new allocator leads generally to slightly better results and never to worse.

We also performed the same experiment on Windows operating system. We used computer with Intel Core i7 2600, running at 3.4GHz. The processor has
In this section we focus on other features (or consequences of the Bobox system). In the previous sections, we described the main features and enhancements of the Bobox system, which can be used for speeding up of applications.

Therefore, we represent envelopes only as shared pointers to their data.

The current version of Bobox is intended to be used on shared-memory systems.

2.6.1 Efficient Manipulation with Envelopes

The results are shown in Figure 2.11. The results for queries Q6 and Q9 are

Windows 7 (logarithmic scale).

efficient manipulation with envelopes (which can be used for speeding up of applications).
transfer of one envelope is then equivalent to the transfer of one shared pointer, i.e., the time complexity of the transfer is constant.

For this reason, it is possible to implement following three operators very efficiently:

- The dispatch operator – this operator has one input and multiple outputs. Its task is to forward each incoming envelope to one of its outputs according to some rule, i.e., the dispatch operator splits the input stream into multiple sub-streams. The simplest rule is to use a round-robin manner, i.e., the envelopes are cyclically forwarded to the outputs. This rule ensures that the resulting sub-streams have probably same size. However, more sophisticated algorithms might be used (see Section 2.6.2).

- The consolidate operator – this operator has multiple inputs and one output. Its task is to forward envelopes from its inputs to its output, i.e., it consolidates multiple input streams to one output stream. The simplest forwarding algorithm is also a round-robin manner.

- The broadcast operator – this operator forwards the input envelopes from its only input to all its outputs, i.e., it broadcasts the shallow copies of the input stream to all its outputs.

The main advantage of these operators is that we can use them in execution plans without a risk that they become a serial bottleneck. The main reason is that each envelope typically contains much more tuples than the number of processing units in the system. If $L$ is the number of tuples in one envelope, $N$ is the number of processing units in the system and the stream consists of $E$ envelopes, then the time complexity of the dispatch operator is $O(E)$, while the time complexity of the processing of all tuples in parallel is $O(E\frac{L}{N})$. Since $N \ll L$, it holds that $O(E) \ll O(E\frac{L}{N})$, i.e., the dispatch box is able to process its input envelopes faster than the parallel processing of individual tuples in the envelopes. The same rule holds for the consolidate box.

The time complexity of the broadcast operator with $O$ outputs is $O(E \cdot O)$. Therefore, it must hold that $N \cdot O \ll L$ in order to be faster than $O(E\frac{L}{N})$. But this is true in all reasonable cases, since usually $O \leq N$ (see Chapter 3 for the typical use of the broadcast operator) and $N^2$ is still much less than $L$ (see Section 2.6.3 which discusses how large the envelopes are).

Notice that these three operators are very important for the intra-operator parallelization (see Section 3.2).

It is also worth mentioning that tuples within one envelope might be accessed randomly. Therefore, it is possible to implement several algorithms in a sub-linear time complexity. See for example algorithms in Chapter 4 which make use of that fact.

### 2.6.2 Efficient Manipulation with Columns

Each column in an envelope is also represented by the shared pointer to its data. Therefore, we can permute columns of an envelope or multiple envelopes may share one or several columns.
Additionally, we are allowed to make a column which shares a subsequence of another column. This allowed the implementation of the fast \texttt{split} function which splits one envelope into two envelopes at the given index so that the first resulting envelope contains tuples below that index and the second contains tuples above that index.

This can be very useful. For example \texttt{dispatch} operator with $O$ outputs can split each incoming envelope to $O$ equally sized envelopes and send them in a round robin manner to the outputs. This ensures that the resulting sub-streams have the same size regardless the number and size of input envelopes.

### 2.6.3 Optimal Size of Envelopes

Choosing the optimal size of an envelope, i.e., the optimal number of its tuples, was already researched in [29]. This paper showed that the size of envelopes significantly influences the performance of the system, since smaller values increase overhead with the task scheduling, whereas larger values decrease utilization of the cache. Additionally, larger values decreases parallelization since the tasks could be too coarse grained. However, no particular solution was proposed in that paper.

We also performed multiple experiments on various systems and discovered that the optimal size is almost unpredictable. It depends on the size of caches, on the type of the application on its level of parallelism etc. Therefore, we chose the following heuristic:

$$\text{preferred size of envelope} = \frac{1}{2} \cdot \frac{\text{size of LL cache in bytes}}{\text{processing units per NUMA node}} \cdot \frac{\text{size of one tuple in bytes}}{\text{size of one tuple in bytes}}$$

The idea of this equation is as follows – it is possible to have $N$ envelopes hot in one half of the LL cache simultaneously ($N$ is the number of processing units per NUMA node), while the other half of the LL cache is available for stack, auxiliary data structures etc. Therefore, the processing units of one NUMA node may either process their own envelopes without forcing other envelopes out of the LL cache, or may cooperate on the processing of $N$ envelopes hot in the LL cache.

In order to avoid extremely large or extremely small envelopes, we keep the size of envelopes within the range 64kB and 256kB.

Notice that this heuristic does not take the L1 cache (or even the L2 cache if L3 is present) into account. According to our experiments, decreasing the preferred size of envelopes so that they fit into the L1 (L2) cache may increase the performance of the system. However, this performance is very sensitive to the structure of an application and to the exact size of envelopes, i.e., some applications were faster but some significantly slower. Therefore, we used the LL cache which leads to stable and reasonable results.

Worth mentioning is that the preferred size of envelopes was set to 256kB in all systems used for the experiments in this thesis.
2.7 Related Work

2.7.1 Streaming Systems

Stream processing has no formal definition and this term is used to describe a variety of systems [122].

Special category of the streaming systems are the data stream management systems (DSMS). Systems such as Borealis [19], STREAM [21], Nile [70] or NiagaraST [97] can be considered as representatives of such systems and are used to query continuous, real-time and typically infinite data streams. For example network traffic or data from sensors are such data streams. These systems typically provide a high-level API, a set of predefined operators, a set of predefined data types, their own query language or tools to deploy the system on distributed systems, etc.

In contrary to DSMS, Bobox is a lightweight framework for the development of parallel applications. Bobox supposes that the streams are finite (and all active envelopes fit into main memory), data are created or read into the main memory on demand (i.e., they are not real-time or continuous) and that applications are separated from each other and might be evaluated independently.

Besides DSMS, there is a large number of streaming systems which focus on similar domain as Bobox. The StreamIt [125] is one of those. This system provides a domain specific language as well as the compiler of the language and the runtime. The system enables the development of efficient streaming applications and is able to utilize various hardware architectures such as multi-core and distributed systems. The main difference between StreamIt and Bobox is that StreamIt follows Synchronous Data Flow paradigm [94], i.e., each operator must explicitly state how many tuples it pops from its inputs and pushes into its outputs. This restriction allows static scheduling [95]; however, disallows the implementation of operators which have unpredictable number of output tuples (e.g., database join or filter) or the efficient implementation of operators like dispatch, consolidate or broadcast.

Another streaming system/language is Brook [35] which enhances the ANSI C language by a stream data type. Besides this new data type it allows to declare kernels and reductions, i.e., special functions which can operate on the streams. However, these functions must be able to be applied to each tuple of a stream independently, i.e., it can use only its parameters and local variables and cannot access global variables and functions [35]. In other words, each operator must be strictly stateless. This allows better optimization of the applications, but restricts the applicability of the language.

There are also many hardware specific streaming languages/systems, like StreamC/KernelC [48] designed for Imagine processor [100] or Cg [98], sH [104] and BrookGPU [36] designed for GPUs.

The Auto-Pipe [40] is another streaming system. It is similar to Bobox; however, it is mainly intended to make the development of complex streaming applications on various architectures easier. Its basic idea is that the developer specifies operators (denoted as tasks), computation resources, communication topology of these resources and the system finds an optimal mapping between the tasks and these resources. This system allows the utilization of various hardware (not only shared-memory systems like Bobox); however, in case of multiprocessor
systems each operator is statically mapped to a particular processing unit. This may limit utilization of caches in case of pipe-line parallelism (envelope produced on one processing unit might be always consumed on another processing unit) or cause significant imbalance if the operators do not work uniformly during the evaluation of execution plan.

Intel Threading Building Blocks (TBB) [114] is not the true streaming system. However, it provides similar functionality through its Flow Graph component [4]. If we consider the messages of Flow Graph as envelopes then this component is comparable to the initial version of Bobox. On the other hand, the current version of TBB does not provide any equivalent to the wait function, does not support I/O operations (this feature is available through an extension [52]) and its scheduling algorithm is outperformed by the Bobox scheduler (see Section 2.4.4). Notice that Flow Graph is only one of its components and in contrary to Bobox, it provides much more functionalities.

Another well-known and widely used framework is OpenMP [41] which has a relatively new extension for data stream processing [111]. The extension enables an intuitive description of the pipeline stages of an algorithm; however, more complex constructions (such as examples from Section 3.3.2) are tricky and difficult to implement. Furthermore, it is almost impossible to develop a parallel operator with an output of unpredictable length. For instance, the size of the output of a natural join may have from zero to $L \times R$ tuples, where $L$ and $R$ are the numbers of tuples in the left and right input respectively.

Worth mentioning is also the MapReduce programming model [49]. This programming model is similar to the fork/join model [93], but much more coarse-grain and intended for distributed environments [123]. Despite the fact that it is considered a step back [124] in parallel database engines, this model gained significant attention. The MapReduce was also studied on shared-memory systems [113, 89], but its main disadvantage remains, i.e., that this model does not support more complex execution plans like non-linear pipelines. This is partially solved by the Apache Pig [11] environment together with the Pig latin language [110] which is able to transform non-linear pipeline into a sequence of MapReduce programs, but this platform is optimized mainly for the analysis of big data.

Very close to the Bobox system is also the Dryad system [77]. It models the application in the same way as Bobox, i.e., applications consist of operators connected together with communication channels; therefore, it is more general than the MapReduce model. However, Dryad is intended for the development of coarse-grain data-parallel applications as well as MapReduce.

2.7.2 Task Scheduling on CPUs

As we already mentioned in Section 2.4.1 modern architectures became complex and complicated. Therefore, the optimization of the performance of applications through elaborate task scheduling strategies is a challenging task and a very hot topic in current research. The fact that finding an optimal scheduling plan is NP-hard problem [121] causes that scheduling strategies generally try to find a sub-optimal solution using heuristics and approximation techniques [121].

In streaming systems, there are several aspects of the scheduling optimization,
such as memory usage [23], cache-efficiency [45], response time, throughput, etc. or their combinations [80, 116].

In this work, we relaxed many of such aspects and focused mainly on the maximization of data locality in order to increase the performance of the system. This allowed us to adopt techniques used in non-streaming systems, e.g., in popular parallel frameworks such as Cilk [31], OpenMP [53, 34] or Intel Threading Building Blocks [91, 16].

The works [29, 62] were the first step and showed that data flow awareness (i.e., using immediate and deferred tasks) in streaming systems increases data locality. The division of the tasks to immediate and deferred tasks ensures that threads work with data hot in cache if possible. However, it showed up that the bottleneck of the system is the task stealing since the tasks were stolen from a randomly chosen thread which is an issue of the papers cited in the previous paragraph as well.

The task stealing optimization is researched thoroughly in [43, 42]. In fact, the algorithm CATS/CAB from this work is similar to our scheduling algorithm described in Section 2.4.3; however, there are several differences between these two algorithms. We partition physical processors more precisely according to the structure of shared caches, whilst CATS/CAB creates always one group per physical processor (socket). Furthermore, our algorithm for the task stealing within a group also considers the cache hierarchy, which is advantageous when the processing units in one group share more than the last level of cache. Additionally, we set the affinity of threads for the whole group. This has two advantages – first, we can freely add and remove threads to the thread pool which enables easier support of IO operations (see Section 2.4.3), second, this strategy copes better with Hyper-Threading Technology, since it does not restrict the operating system from its own load balancing strategy [13]. Finally, we optimize the situation when the system processes multiple independent requests in parallel.

2.8 Chapter Summary and Future Work

In this chapter, we introduced multiple enhancements of the Bobox framework which made it more applicable and efficient on SMP and NUMA systems. The new API for the development of boxes made this process much simpler and more user friendly. The new task scheduler speeds up the system up to 10% on SMP and up to 60% on NUMA systems. The new version of system supports blocking I/O operations, effective manipulation with envelopes and their columns. This manipulation enables the implementation of several non-trivial algorithms (see Chapter 4).

On the other hand, many challenges still remain. One of them is the more detailed analysis of the optimal size of envelopes. This size depends on the structure of the application as well as on the behaviour of boxes used in the application. Therefore, we want to analyse runtime characteristics of the particular application and choose the optimal size according to them.

Additionally, it showed up that despite the user friendliness of the Bobox API, the debugging of applications is very tedious, since the source code of one box might be called from different contexts (the box might be replicated or be used multiple times in one execution plan) or multiple threads. Therefore, the
standard debugging methods available in integrated development environments such as call-stack, breakpoints and tracing are difficult to use. Also profiling of such applications is non-trivial because of the same reasons. Therefore, we want to develop a set of tools which would make the debugging and profiling easier.

Another challenging task is the porting of Bobox to other architectures such as Intel Many Integrated Cores Architecture or distributed systems.
3. Bobolang Language

3.1 Introduction

Streaming systems like Bobox consider the application to be an oriented graph. The nodes of this graph are operators and the edges are data streams. This implies two problems:

1. how to specify the behaviour of operations,
2. how to specify their mutual connections, i.e., the execution plan.

The languages for streaming systems are (among others) designed to help with these problems. Some of them focus on the first problem, some on the second problem and several languages on both simultaneously (see discussion in Section 3.4)

The specification of operations does not require a dedicated language, since it might be straightforwardly developed in a common programming language. See for example Section 2.3 which shows the implementation of a box in the C++ language.

The execution plan might be also specified in a common programming language; however, corresponding source codes are hard to read and maintain. Consider for instance the example of the construction of a flow-graph in TBB listed in Listing 3.1. This code creates the execution plan depicted in Figure 3.1.

Listing 3.1: Example of the construction of a flow-graph in TBB.

graph g;
broadcast_node<int> input(g);
function_node<int,int> squarer(g, unlimited, square());
function_node<int,int> cuber(g, unlimited, cube());
join_node<tuple<int,int>, queueing> join(g);
function_node<tuple<int,int>,int> summer(g, serial, sum(result));

make_edge(input, squarer);
make_edge(input, cuber);
make_edge(squarer, get<0>(join.input_ports()));
make_edge(cuber, get<1>(join.input_ports()));
make_edge(join, summer);

Figure 3.1: Execution plan of the graph g.
For this simple schema, this method is suitable. However, the specification of execution plans like in Figure 3.8 is tricky and requires statements like nested for-cycles etc.

Languages like StreamIt are able to specify the operators as well as the execution plan simultaneously. See for example the source code of a band-pass filter [12] listed in Listing 3.2. This code produce the execution plan depicted in Figure 3.2.

Listing 3.2: Source code of a band-pass filter in StreamIt.

```c
float -> float pipeline BandPassFilter
(float rate, float low, float high, int taps) {
    add BPFCore(rate, low, high, taps);
    add Subtracter();
}
float -> float splitjoin BPFCore
(float rate, float low, float high, int taps) {
    split duplicate;
    add LowPass(rate, low, taps, 0);
    add LowPass(rate, high, taps, 0);
    join roundrobin;
}
float -> float filter Subtracter {}{
    work pop 2 push 1 {
        push(peek(1) - peek(0));
        pop(); pop();
    }
}
```

Figure 3.2: Execution plan of the **BandPassFilter** operator.

As we already mentioned, Bobox does not provide any dedicated language for the specification of boxes. It makes use of the C++ language and provides powerful and easy to use API for their development. However, during the development of real applications in Bobox, we found out that the specification of execution plans is also important part of the development and that the system should provide a powerful and easy to use way of their specifications as well. Therefore, we decided to design the Bobolang [56] language, which solves common problems elegantly and contains many other features which are suitable for the development of parallel data processing applications.

The easier specification of execution plans was not the only reason why we decided to design a completely new language. As we mentioned in Section 1.1,
Figure 3.3: Decomposition of an operator into a non-linear chain of sub-operators.

streaming systems naturally introduce the inter-operator parallelism, i.e., the execution of operators in parallel. This increases the performance of the application, but its scalability is very limited (see Section 5.3.1 for some experiments), since the scalability does not depend on the number of processing units in the system, but on the structure of the execution plan.

The solution is to introduce the intra-operator parallelism along with the inter-operator parallelism, i.e., to enable parallel evaluation of a single operator. However this type of parallelism is not common in streaming systems since some of them expect that the implementation of operators is single-threaded (Bobox is one of them). The main reason is that the threads created in the implementation of operators would interfere with the threads managed by the streaming system. This interference may cause performance loss.

Fortunately, there are several methods of transforming the intra-operator parallelism into the inter-operator parallelism (we introduce some of them in the following section). Making this transformation easier is the second main reason why we designed Bobolang.

### 3.2 Intra-operator Parallelization

The intra-operator parallelism is difficult to achieve automatically in streaming systems since operators are usually designed as black boxes. One solution is to express complex operators as the composites of simpler sub-operators, i.e., to convert the intra-operator parallelism into inter-operator parallelism. In this section we focus on several methods which can be helpful with this conversion.

One of the methods is the decomposition of an operator to a (not necessary linear) chain of sub-operators (see Figure 3.3 for an example). However, this kind of parallelization can be used only under certain circumstances and usually does not scale well, since the number of sub-operators is limited and determined by the nature of the original operator. This type of decomposition is an example of task and pipeline parallelism [66].

Another method is the replication of an operator which corresponds to data parallelism [66]. This method is much more scalable, since the number of replicas is not theoretically limited. However, in order to explore this method further, we require additional information about the operators. Especially, whether they maintain an inner state which is modified by the incoming tuples. From this
point of view, we recognize two groups of operators:

1. **Stateless operator** is an operator which do not maintain an inner state. The result of its function for a particular tuple does not depend on the previous tuples. For example filter operator is stateless, since the filter condition is evaluated for a tuple independently on any other.

2. **Stateful operator** is an operator that maintains an inner state and this state may be updated by every processed tuple. A merge join is an example of stateful operator.

In the following section, we focus on both operator categories from the view of automatic and semi-automatic parallelization.

### 3.2.1 Stateless Operators

Stateless operations are very easy to parallelize, since we may split the input stream into several sub-streams and process each sub-stream independently in parallel. The resulting sub-streams are then joined together into a single output stream, so that the functionality of the original operator is preserved. Figure 3.4 shows the general idea of the operator replication. This operation can be done automatically if the system knows that the operator is stateless. Recall that the Dispatch operator and the Consolidate operator are fast enough, therefore, they do not become bottlenecks (see Section 2.6.1).

![Figure 3.4: Parallelization schema for stateless operators.](image)

### 3.2.2 Stateful Operators

The parallelization of stateful operators is more complicated and it requires minor modification of the internal function of the operator. Furthermore, we need to notice that some operators cannot be modified this way effectively, since the modification would cause serious performance hit. However, we will show that there are many situations where a stateful operator may be parallelized at acceptable cost. Let us consider a typical schema of a general stateful operator with an internal state $S$ show in Algorithm 3.1.

If the processing of the tuples and the update of the state $S$ can be effectively separated from each other and the updating of state $S$ takes significantly less time than the tuples processing, then the stateful operator can be effectively parallelized (we denote such operators as parallelizable). The concurrency is achieved by the replication of the operator while each replica keeps and updates its own copy of the state. Each replica has a unique index from 0 to $N - 1$
Algorithm 3.1 General schema of stateful operators.

\[
S \leftarrow \text{initial state} \\
\textbf{while} \not\text{finished} \textbf{do} \\
\quad \text{envelope} \leftarrow \text{next envelope} \\
\quad \text{process envelope using state } S \text{ and update state } S \text{ simultaneously} \\
\textbf{end while}
\]

Algorithm 3.2 Algorithm performed by the replicas of a parallelizable operator.

\[
S \leftarrow \text{initial state} \\
\text{phase} \leftarrow 0 \\
\textbf{while} \not\text{finished} \textbf{do} \\
\quad \text{envelope} \leftarrow \text{next envelope} \\
\quad \text{if phase mod } N = \text{RID then} \\
\quad \quad \text{produce output for envelope using state } S \text{ and update state } S \\
\quad \text{else} \\
\quad \quad \text{update state } S \text{ using envelope} \\
\quad \textbf{end if} \\
\quad \text{phase} \leftarrow \text{phase} + 1 \\
\textbf{end while}
\]

(for \( N \) replicas) called \( \text{RID} \) (Replica ID) and performs the algorithm listed in Algorithm 3.2.

The schema for the parallelization of the parallelizable operators is depicted in Figure 3.5. The replicas receive a shallow copy of the original stream and alternate in the processing of the envelopes and in the production of the output stream. The resulting envelopes are then gathered by the \textbf{Consolidate} operator as in the case of parallelization of the stateless operators.

In the rest of this section, we analyse conditions which must be fulfilled in order to achieve effective parallelization. If the processing of one envelope takes time \( T_p \) and update of the state \( S \) takes time \( T_u \) for one envelope, then \( N \) phases performed by one replica takes time \( T_p + (N-1)T_u \). When all replicas are working in parallel, then the processing of \( N \) envelopes in parallel takes the same time, i.e., \( T_p + (N-1)T_u \), while the processing of \( N \) envelopes by non-parallelized operator takes \( NT_p \). Therefore, if it holds that \( NT_p > T_p + (N-1)T_u \), which is equivalent to \( T_p > T_u \), we achieve speed-up by using the proposed parallelization schema.

On the other hand, the parallelization schema may introduce significant overhead. The total amount of work performed by all replicas of the operator is \( N(T_p + (N-1)T_u) \), i.e., \( NT_p + N(N-1)T_u \), while the total amount of work performed by the non-parallelized operator is only \( NT_p \). Therefore, if less than \( N \) processing units are available for the evaluation\(^1\) then the parallelization of parallelizable operators may cause significant slowdown. Notice that this situation occurs commonly when multiple distinct operators or requests run in parallel.

We identified two possible solutions of this problem which might be used under particular circumstances. The first is based on the idea that each replica works in one phase and updates the state in \( N-1 \) phases. In some operators, it might be significantly faster to update the state once for \( N-1 \) envelopes at a time than

\(^1\)I.e., the work assigned to \( N \) replicas is evaluated by less than \( N \) processing units in parallel.
Figure 3.5: Parallelization schema for parallelizable operators (RIDs are in brackets).

$N - 1$ updates for each envelope separately. This idea leads to the schema listed in Algorithm 3.3. We use this schema in our parallel multi-way merge algorithm (see Section 4.4.3).

**Algorithm 3.3** Algorithm performed by the replicas of a parallelizable operator with the aggregated state updating.

\[
S \leftarrow \text{initial state} \\
\text{update state } S \text{ using the next } RID \text{ envelopes} \\
\text{while not finished do} \\
\quad \text{produce output for the next envelope using state } S \text{ and update state } S \\
\quad \text{update state } S \text{ using the next } N - 1 \text{ envelopes} \\
\text{end while}
\]

The second idea is based on the fact that one envelope is processed by one replica, while $N - 1$ replicas use that envelope only for the updating of the state, i.e., $N - 1$ replicas perform exactly the same operation for a particular envelope. This redundancy might be eliminated by a dedicated `Preprocess` operator, which applies the algorithm for the state updating for each envelope and stores the resulting state as scalar data to the envelopes (see Section 2.2.1) which are broadcasted to the replicas. In this method, the `Preprocess` operator performs the algorithm listed in Algorithm 3.4, while replicas perform the algorithm listed in Algorithm 3.5. We use this schema in parallel merge join algorithm (see Section 4.5).

**Algorithm 3.4** Algorithm performed by the `Preprocess` operator.

\[
S \leftarrow \text{initial state} \\
\text{while not finished do} \\
\quad \text{envelope} \leftarrow \text{next envelope} \\
\quad \text{store } S \text{ to } \text{envelope} \\
\quad \text{send } \text{envelope} \\
\quad \text{update } S \text{ using } \text{envelope} \\
\text{end while}
\]

**Example of a Parallelizable Operator**

A simple example of a parallelizable operation is the defragmentation of envelopes. Some operators generate envelopes which are smaller than the preferred size (see Section 4.2 for the example). This can decrease the efficiency of the system since
Algorithm 3.5 Algorithm performed by the replicas prepended by the Preprocess operator.

```
while not finished do
    envelope ← next envelope
    produce output for envelope using the state stored in envelope
    skip $N - 1$ envelopes
end while
```

the smaller envelopes increase the overhead with their management and with the scheduling of boxes.

If the preferred size of an envelope is $L$ tuples then the basic schema of the Defragment operator is shown in Algorithm 3.6. In order to make the operator parallelizable, this schema may be modified as shown in Algorithm 3.7 or as shown in Algorithm 3.8 in order to reduce the number of state updates.

Algorithm 3.6 Non-parallelized version of the Defragment operator.

```
while not finished do
    allocate a new envelope with $L$ tuples
    read $L$ input tuples and copy them to the allocated envelope
    send the envelope to the output
end while
```

Algorithm 3.7 Parallelized version of the Defragment operator.

```
phase ← 0
while not finished do
    if phase mod $N = RID$ then
        allocate a new envelope with $L$ tuples
        read $L$ input tuples and copy them to the allocated envelope
        send the envelope to the output
    else
        skip $L$ tuples
    end if
    phase ← phase + 1
end while
```

The skipping of tuples is very fast operation since we have random access to tuples within an envelope (see Section 2.6.2). Therefore, this algorithm is efficient.

More complex examples of parallelizable operators can be found in Chapter 4.

3.3 Bobolang Syntax and Constructs

Bobolang is a language designed to make the description of execution plans easy and intuitive. It takes the implementations of operators such as boxes in Bobox and a developer specifies which operators are used in the plan and how they are mutually connected. Besides the specification of complete execution plans,
Algorithm 3.8 Parallelized version of the Defragment operator with the aggregated state updating.

\begin{algorithm}
\caption{Parallelized Defragment operator}
\begin{algorithmic}
\State Skip $L \cdot RID$ tuples
\While{not finished}
\State Allocate a new envelope with $L$ tuples
\State Read $L$ input tuples and copy them to the allocated
\State Send the envelope to the output
\State Skip $(N-1) \cdot L$ tuples
\EndWhile
\end{algorithmic}
\end{algorithm}

Bobolang also enables creation of new operators from existing operators (i.e., from operators already defined in Bobolang). Notice that in the rest of this thesis, we use the following notation – the names in the CamelCase style denote the operator types, while the names in the underscore style denote the instances of operators.

The definition of a new operator in Bobolang is listed in Listing 3.3.

\begin{lstlisting}[language=Bobolang]
operator Process (int) -> (int, int, int)
{
\hspace{1em} Preprocess (int) -> (int, int) preprocess;
\hspace{1em} Postprocess (int, int) -> (int, int, int) postprocess;

\hspace{1em} input -> preprocess -> postprocess -> output;
}
\end{lstlisting}

The first line tells that we want to create a new operator called \texttt{Process}. The operator has one input (a stream of integers) and one output (a stream of triples of integers). The body of the operator has two parts. The first part contains the definitions of the sub-operators from which the operator is composed. This part has similar syntax as the definition of variables in C/C++. The second part specifies connections between operators and its syntax is rather intuitive.

In addition to the explicitly defined sub-operators, each body implicitly contains two special sub-operators – the \texttt{input} sub-operator and the \texttt{output} sub-operator. These sub-operators represent the input and the output of the \texttt{Process} operator. Therefore, the command \texttt{input -> preprocess}; says that the input of the \texttt{Process} operator is forwarded to the input of the \texttt{preprocess} sub-operator. The \texttt{output} sub-operator works analogically.

Because of the \texttt{init} and \texttt{term} operators in each execution plan (see Section 2.2.1), we may consider the execution plan as a single operator which has one input and one output. Therefore, we can define the whole execution plan in the same way as any other operator. The only difference is that the execution plan must be called \texttt{main}. Before the evaluation of the plan, Bobolang automatically connects the \texttt{init} operator to its input and the \texttt{term} operator to its output.

An example of a complete execution plan is listed in Listing 3.4 and the instantiated plan is depicted in Figure 3.6. The datatype \texttt{()} (i.e. empty tuple) is used for the transfer of empty streams (i.e., for the transfer of poisoned pills in
Listing 3.4: Specification of a complete execution plan in Bobolang.

```cpp
operator main() ->()
{
    Source() ->(int) source;
    Process(int) ->(int,int,int) process;
    Sink(int,int,int) ->() sink;

    input -> source -> process -> sink -> output;
}
```

3.3.1 Arity of Operators

Operators are allowed to have an arbitrary number of inputs and outputs. The inputs/outputs are numbered by consecutive numbers starting with zero. If an operator has only one input/output, its number might be omitted, otherwise the number must be specified as shown in Listing 3.5. The instantiated plan is depicted in Figure 3.7.

Listing 3.5: Arity of operators in Bobolang.

```cpp
operator main() ->()
{
    Broadcast() ->(),() broadcast;
    Source() ->(int) source1, source2;
    Merge(int),(int) ->(int) merge;
    Sink(int) ->() sink;

    input -> broadcast;
    broadcast[0] -> source1 -> [0]merge;
    merge -> sink -> output;
}
```

3.3.2 Multiplication of Inputs and Outputs

To specify schemas like the one in Figure 3.4, we need to define 4 instances of a stateless operator and connect their inputs manually to the dispatch operator and their outputs to the consolidate operator. First, this would be very tedious
and error-prone since the source code would be very long and many lines would be very similar. Moreover, it is not very flexible since if we want to add or remove several stateless operators, we have to modify the source code on multiple places. Additionally, it does not allow to modify the structure of the execution plan automatically according to criteria like the number of available processing units.

To solve these problems, Bobolang introduces the multiplication of inputs and outputs. By default, each input/output is single. If we want to specify that some input/output is multiple we have to do it explicitly as follows:

\[
\text{Broadcast}() \rightarrow ()* \text{broadcast} ;
\]

This line declares that the output of the Broadcast operator is multiplicated. Bobolang allows to connect a single or multiplicated output to a single or multiplicated input which leads to four possible combinations:

1. **single output to single input** – the target sub-operator is automatically replicated according to the number of replicas of the source sub-operator and the corresponding replicas get connected. This is demonstrated in the following example on operators \( \text{op2} \) and \( \text{op3} \).

2. **multipicated output to single input** – the target sub-operator is automatically replicated. Each output is connected to the input of the corresponding replica of the target. This situation usually fits the data dispatching/broadcasting scenario (see the operators \( \text{op1} \) and \( \text{op2} \)).

3. **single output to multiplicated input** – only one replica of the target sub-operator is created and its each input is connected to the output of the corresponding source. This situation is usually described as data consolidation and illustrated on the operators \( \text{op4} \) and \( \text{op5} \).

4. **multiplicated output to multiplicated input** – the target sub-operator is automatically replicated according to the number of replicas of the source sub-operator and each side has as many inputs/outputs as there are replicas. The connections are made so that each source replica has a connection to all target replicas, thus each target replica is connected with all source replicas. This situation is depicted in the following example as the connection between the operators \( \text{op3} \) and \( \text{op4} \).
Notice that when a sub-operator is automatically replicated due to multiplied input/output, every replica is assigned its own unique RID which is essential for the implementation of parallelizable operators.

The source code listed in Listing 3.6 demonstrates the behaviour of multiplied inputs/outputs. The instantiated execution plan is depicted in Figure 3.8.

Listing 3.6: Multiplicated inputs/outputs in Bobolang.

```
operator main () -> ()
{
    SomeOperator () -> ()* op1;
    SomeOperator () -> () op2;
    SomeOperator () -> ()* op3;
    SomeOperator ()* -> () op4;
    SomeOperator ()* -> () op5;

    input -> op1 -> op2 -> op3 -> op4 -> op5 -> output;
}
```

Choosing the Right Number of Replicas

Choosing the right number of replicas is very important in order to achieve optimal performance. On SMP systems, Bobolang tries to keep the number of replicas same as the number of processing units in the system. This ensures that a parallelized operator (either stateless or stateful) is able to utilize the whole system.

On NUMA systems, there are generally two possible choices for the number of replicas:

1. the number of processing units per node,
2. the number of all available processing units.

The first choice ensures that all replicas of one operator are preferably evaluated on the same node; however, it limits the scalability of the algorithm. The second choice ensures that the parallelized operator is able to utilize the whole system; however, it increases the overhead caused by the large number of replicas and the overhead caused by the NUMA factor.
In order to analyse these choices further, we performed the following experiment. We run selected queries from SP²Bench benchmark on NUMA64 system once with the number of replicas set to 16 (the first choice) and once set to 64 (the second choice). The results were very variable (see Figure 3.9 when we evaluated single query at a time and Figure 3.10 when we evaluated 16 multiple queries in parallel, the NUMA64/16 columns corresponds to 16 replicas and the NUMA64/64 corresponds to 64 replicas). In some queries the first choice was significantly better (Q6, Q7) and in some significantly worse (Q2, Q9 etc.). We identified that the first choice is better for queries when all tasks are uniform and relatively small, while the the second choice is better for queries which generate computation intensive tasks.

In case of computation intensive tasks, the overhead with their scheduling and the NUMA factor is almost eliminated by the significantly larger number of worker threads which processes these tasks in parallel. On the other hand, relatively small tasks benefit from the data locality; therefore, it is better to keep their processing on one NUMA node.

Bobolang interpreter is unable to make the right decision, since it does not know anything about the behaviour of the particular operators. We decided that Bobolang uses the first choice by default. But in order to enable the developer to specify it at least manually, we enhanced Bobolang by the symbol **. This symbol is equivalent to * on SMP systems, but on NUMA systems, it tells the interpreter to use the second choice, i.e., the number of all processing units available in the system.
Finally, we performed one additional experiment, when we modified the nested loop operator, which is the most computation intensive operator, to use the symbol **, while the rest of operators uses the symbol * (the NUMA64/** columns). The results (see Figure 3.9 and Figure 3.10) show that the performance is better or at least the same as the best result of NUMA64/16 and NUMA64/64, i.e., the symbol ** allowed us to increase the efficiency of the SPARQL engine on NUMA systems.

Notice that the developer is also allowed to directly use numerical constant instead of the symbols */**. However, this feature is intended primarily for scenarios when a Bobolang is generated automatically by a front-end which may perform better analysis and optimizations based on the context which the Bobolang interpreter does not have. For example, the SPARQL compiler and optimizer from Section 5.2.1 might better estimate number of replicas of parallel merge joins since it is able to estimate the distribution and the amount of input data (see the discussion in Section 4.5.3).

3.3.3 Operator Templates

In all previous examples, we had to explicitly define the datatypes of all inputs/outputs. This is not very flexible, since for example the parallelization schema of a sort operator would be probably the same for the stream of integers as for the stream of real numbers. If the datatypes must be explicitly stated, two very similar schemas would be required to solve such situation. The only difference between those schemas would be in the datatypes.

To avoid code replication in such situations, Bobolang contains keyword typename. Its semantics is inspired by the language C++. We may use this keyword in the declaration of the new operator in the following way:

```c
operator Sort(typename T)->(T)
{
    SomeOperator(T)->(T) op;

    // ...
}
```

The keyword typename before the T identifier denotes a type that is specified when the operator is instantiated. In the body of the operator we may use the datatype T as any other datatype. If we instantiate the Sort operator for example in the following way,

`Sort(int,int)->(int,int)`

then the datatype T will be equivalent to (int,int). If the input and output datatype of the Sort operator is different, then an error during the compilation is signalized.

3.3.4 Intra-operator Parallelization

We described two types of operators in Section 3.2 and methods of parallelizing them. To make the parallelization easier, we may specify the type of a sub-operator and let Bobolang parallelize it automatically. There are two keywords
for specification of the type of sub-operator — stateless and parallel (the abbreviation for parallelizable). They can be used in the definition of a sub-operator as follows:

```c
stateless StatelessOperator() ->() op1;
parallel ParallelizableOperator() ->() op2;
```

If we mark the operator as stateless then it is automatically replaced by the schema listed in Listing 3.7 (with the respect to the arity of the operator), i.e., it will be decomposed as in Figure 3.4 automatically.

Listing 3.7: Schema for the parallelization of a stateless operator.

```c
operator ParallelStatelessOperator(typename T)->(typename U)
{
    Dispatch(T)->(T)* dispatch;
    StatelessOperator(T)->(U) op;
    Consolidate(U)*->(U) consolidate;

    input -> dispatch -> op -> consolidate -> output;
}
```

The parallel keyword has almost the same effect. The only difference is that the Broadcast operator is used instead of the Dispatch operator.

### 3.3.5 Other Language Constructs

In previous sections, we have presented the fundamental principles of the Bobolang language. Besides these properties, there are several features of lower importance such as support of parameter passing to sub-operators, the typedef keyword, naming inputs/outputs etc. These features make the Bobolang language more user friendly; however, they are rather technical and we do not include their detailed description in this text. However, more details can be found in Appendix B and in Appendix C.

### 3.4 Related Work

Contemporary streaming languages basically differ in their focus. Each one is designed with a particular intent which significantly influences their syntax and semantics. In this section, we provide a comparison of the Bobolang language to other languages from the domain of streaming systems.

The languages Brook [35, 36], StreaMIT [125], and StreamC [48] are intended for the development of efficient streaming applications. They introduce a language based on the C/C++ syntax, which allows the developer to implement the operators as well as to specify their mutual connections. The compiler exploits the streaming nature of the application to perform specific analyses and optimizations designed to increase the performance with a particular emphasis on
concurrent execution. The compiler also creates a static mapping of the operators to the computation resources such as CPUs, GPUs, or FPGAs\textsuperscript{2}.

Another language designed for the development of streaming applications is Lucid \cite{22}. The language was not supposed to be used for parallel applications, so an extension called Granular Lucid (GLU) \cite{78} was developed. The developer basically implements operators in the C language and describe their structure in Lucid. This enables parallel evaluation of operators in the same way as in the Bobolang/Bobox framework.

The X Language \cite{65} is another example of a modern streaming language. This language uses a syntax which clearly and explicitly describes the connections between operators, which is especially useful for the design of complex algorithms. The X Language is used in the Auto-Pipe \cite{40} system (see Section \ref{subsec:autopipe}).

The main difference between these languages and Bobolang is that they lack the support for constructs such as the multiplication of inputs/outputs (see Section \ref{subsec:parallelism}). The absence of this feature makes the specification of some parallelization schemas tedious. On the other hand, Bobolang is not designed to be used in systems, which employ static scheduling, such as StreamIt. Moreover, modifying Bobolang in order to support communication between heterogeneous hardware platforms such as CPUs, GPUs, Xeon Phi cards, FPGAs or distributed systems will be a challenging task, since Bobolang makes heavy use of the fact that the underlying streaming system enables the efficient implementation of operators like \textbf{Broadcast}, \textbf{Dispatch} and \textbf{Consolidate} which typically requires shared-memory architecture.

Dryad \cite{77} system also contains a simple language for constructing execution plans. This language is embedded in C++ as a library using a mixture of method calls and operator overloading. This language contains support for the replications of operators as well as Bobolang. However, its syntax is not as intuitive and expressive as Bobolang since it requires control flow statements like for-cycles in situations when Bobolang does not. On the other hand, this language allows to specify the number of replicas as an runtime expression, whereas Bobolang allows only numerical constants or the symbols */**.

### 3.5 Chapter Summary and Future Work

In this chapter, we have introduced the Bobolang language designed to make the specification of execution plans for streaming systems simple, developer friendly, and human readable. The language was designed with strong emphasis on concurrent execution and scalability, therefore, it naturally integrates some functionality that the regular streaming languages and frameworks do not possess. It contains syntactic and semantic features which make the intra-operator parallelization semi-automatic and more intuitive for the developer.

In our future work, we would like to enhance the language with a support for heterogeneous systems, so that the developer would be able to specify some important details like the distribution of the sub-operators among the nodes of the system. In fact, the symbol */** is the first step of this process.

\textsuperscript{2}Field-programmable gate array
4. Parallel Algorithms for Data Processing

4.1 Introduction

The Bobox system and the Bobolang language described in the previous chapters was implemented in order to simplify the implementation of scalable parallel algorithms. In this chapter, we introduce several such algorithms and their implementation.

We start with very simple ones, i.e., the filter operator and the nested loop join operator which are intended mainly to introduce the usage of Bobox and Bobolang in a real but simple scenario. The main contribution of this chapter is the parallel sort algorithm [58] and the parallel merge join [59] algorithm.

Notice that we used following notation in all results of the experiments in this section:

1. $ST$ shows the time needed for the evaluation of the experiment when we used one worker thread and non-parallelized version of the algorithm.
2. $MT1$ shows the time when we used one worker thread but parallelized version of the operator. The difference between $ST$ and $MT1$ illustrates the overhead introduced by the parallelization of the algorithm.
3. $MT$ shows the time when we used parallelized operator and all available worker threads. The ratio between $MT1$ and $MT$ illustrates the scalability of the algorithm.
4. */$MT1$, */$MT$, **/$MT1$ and **/$MT$ on NUMA64 are analogical to $MT1$ and $MT$ but show the behaviour of the operator when we used the symbol * or ** respectively for choosing the number of replicas (see Section 3.3.2).

4.2 Filter Operator

The only task of the filter operator is to remove all tuples which do not meet the given condition from the input stream. This operator is obviously stateless since the condition is evaluated for each tuple independently on the others. Therefore, its parallelization can be done almost automatically in the Bobolang language. It is enough to mark the filter operator as stateless in the execution plan.

Unfortunately, there is one drawback of this algorithm. The number of tuples produced by the filter operator is less or equal to the number of tuples received. If the filter operator is expected to preserve the ordering of the tuples, then it must produce one output envelope for one input envelope so that the consolidation of these envelopes produces a valid output stream. If the selectivity of the filter condition is high enough, then the output stream may consist of envelopes with significantly less tuples than the preferred number. Recall that small envelopes increase overhead with the task scheduling and may slow down the system (see Section 2.6.3).
One possible solution is to use the parallelizable **Defragment** operator from Section 3.2. However, the **Defragment** operator introduces additional overhead as well. The optimal solution would be to estimate the selectivity of the filter condition and use either **Defragment** operator or not. However, this is beyond the scope of this thesis.

Notice that this problem does not arise if the filter operator does not need to preserve the ordering of tuples. In this case each replica of the **filter** operator can safely produce envelopes with the optimal number of tuples.

The source code of the parallel filter operator in Bobolang is listed in Listing 4.1 and the schema of the operator is depicted in Figure 4.1. Since the **Filter** operator is stateless and the **Defragment** operator is parallelizable, it is enough to use the **stateless** and the **parallel** keyword (see Section 3.3.4) appropriately and Bobolang parallelizes them automatically.

**Listing 4.1**: Parallel filter operator in Bobolang.

```cpp
operator ParallelFilter<typename T>(T)->(T) {
    stateless Filter(T)->(T) filter;
    input -> filter -> output;
}

operator ParallelFilterWithDefragmentation<typename T>(T)->(T) {
    stateless Filter(T)->(T) filter;
    parallel Defragment(T)->(T) defragment;
    input -> filter -> defragment -> output;
}
```

![Diagram of execution plan](image)

(a) Without the defragmentation.

(b) With the defragmentation.

**Figure 4.1**: Execution plan of the parallel filter operators.
4.3 Nested Loop Join Operator

Join is a very important operator used in data processing which combines two relations into one. The most simple join algorithm is the nested loop join. It generates all combinations of the tuples from the first relation with the tuples from the second relation and selects those combinations which meet the join condition. This algorithm is quite inefficient since its time complexity is $O(|N| \times |M|)$, where $N$ and $M$ are the sizes of the relations; however, some joins have so complex join condition that the nested loop join is the only applicable algorithm.

Fortunately, the nested loop join is very easy to parallelize in Bobox. We can split the first input stream (i.e., the first relation) into multiple sub-streams, join these sub-streams with the whole second input stream (i.e., the second relation) in parallel and consolidate the output sub-streams into the result. The schema of this algorithm is shown in Figure 4.2 and the corresponding source code in Bobolang is listed in Listing 4.2.

Listing 4.2: Parallel nested loop join operator in Bobolang.

```bobolang
operator ParallelNestedLoopJoin
    (typename L),(typename R)->(typename T)
{
    Dispatch (L)->(L)* dispatch;
    Broadcast (R)->(R)* broadcast;
    NestedLoopJoin (L),(R)->(T) nested_loop_join;
    Consolidate (T)*->(T) consolidate;

    input[0] -> dispatch -> [0]nested_loop_join;
    nested_loop_join -> consolidate -> output;
}
```

On the other hand, one problem still remains. If the first input stream is short enough and fits into several envelopes, then the replicas of the NestedLoopJoin operator are not loaded uniformly; eventually, if the first input stream fits into a single envelope, then the algorithm becomes single-threaded. Therefore, we slightly modified the Dispatch operator – it does not dispatches individual envelopes to its outputs in a round-robin manner. It splits each incoming envelope into several equal-sized sub-envelopes (one sub-envelope for one output) and sends these sub-envelopes to its outputs. This operator is semantically equivalent to the original Dispatch operator, but ensures that the replicas are loaded uniformly. Recall that the splitting of an envelope is a fast operation (see Section 2.6.2).
4.3.1 Results

In order to show the scalability of the algorithm, we used 1m dataset from SP$^2$Bench benchmark and the following SPARQL query E1:

SELECT ?article1 ?article2
WHERE {
  ?article1 ?type bench:Inproceedings .
  ?article2 ?type bench:Inproceedings
  FILTER (?article1 = ?article2)
}

The execution plan of this query is depicted in Figure 4.3. This query generates all pairs of articles which were published in proceedings and selects those pairs where ?article1 = ?article2, i.e., it simply selects all articles which were published in proceedings. The main idea of this query is to have a computationally intensive join operator but with a high selectivity so that it does not produce many tuples and the memory does not become a bottleneck.

![Figure 4.3: Query plan of the query E1 (with the nested loop join operator).](image)

The results are shown in Figure 4.4 and prove that the evaluation of the query scales well (parallel version is $7.5 \times$ faster on SMP8 and $20.5 \times$ faster on NUMA64 than the serial version).

![Figure 4.4: Results for query E1 (1m, nested loop join).](image)

---

1We forced the compiler to use nested loop join operator only and disabled all optimizations of the query plan.
4.4 Sort Operator

The sorting is fundamental algorithm and is used by many applications; therefore, the parallel sort algorithms are subjected to an intensive research. In this section, we propose highly scalable parallel algorithm for the sorting of data streams.

First, we introduce an algorithm implemented by a single operator, i.e., the algorithm is single-threaded; however, it increases the pipeline parallelism when the operator is used in an execution plan. Then, we use the single-threaded algorithm to implement a completely parallel algorithm.

4.4.1 Single-threaded Sorting Algorithm

Sorting of data stream is a blocking operation, i.e., the first output envelope cannot be created until all input envelopes are received. Therefore, we can divide the workflow of each sorting operator can into three consecutive phases:

1. processing of incoming envelopes,
2. preparation for the third phase,
3. creation and sending of outgoing envelopes.

A straightforward implementation of the operator receives envelopes in the first phase, sorts all tuples in the second phase and sends the sorted data in the third phase. However, it has following drawbacks:

1. The time interval between the reception of the last incoming envelope and the sending of the first outgoing envelope is $O(N \log N)$, where $N$ is the number of tuples. This decreases the performance of the system, since the rest of the execution plan has nothing to do during the second phase of the sort operator.

2. The sort operator does not perform any computation during the first and the third phase. Therefore, neither the producer of the input, nor the consumer of the output runs in parallel with the sorting. This restricts the pipeline parallelism.

Therefore, our objective is to reduce the second phase and to spread the work of the sorting algorithm into the first and the third phase. We implemented an algorithm inspired by the external merge sort \[87\] as follows. In the first phase, each incoming envelope is immediately sorted using traditional sorting algorithm (\texttt{std::sort} in our implementation) and stored to a buffer. When the last input envelope is sorted and stored to the buffer, the operator starts to merge all envelopes from the buffer by a multi-way merge algorithm and sends the output envelopes immediately to the consumer. This algorithm completely eliminates the second phase since the real work is performed only in the first and in the third phase.

Notice that we use the cache oblivious multi-way merge algorithm \[32\] and its implementation from The Funnelsort Project \[132\] in order to increase the efficiency during the third phase.
4.4.2 Parallel Sorting Algorithm

In order to parallelize the sorting operator, we used a traditional approach – we split the input stream into multiple sub-streams, sort them in parallel with the single-threaded algorithm (see the previous section) and consolidate the sorted sub-streams into the resulting stream. Generally, there are two possible approaches to the implementation:

1. Split the stream to multiple (preferably equally sized) sub-streams so that all tuples in the \((i-1)\)-th sub-stream are less than all tuples in the \(i\)-th sub-stream, sort these sub-streams in parallel by the sort operators and then simply concatenate the sorted sub-streams.

2. Split the stream randomly to multiple equally sized sub-streams, sort the sub-streams in parallel and then merge the sorted sub-streams by a multi-way merge operator.

The first approach is easier to implement and has no obvious bottleneck, since the splitting may be performed in parallel, the sorting of independent sub-streams can be performed in parallel as well and the concatenation of streams is fast since it only forwards the incoming envelopes. However, we need to know the distribution of the input stream in advance in order to ensure that the sub-streams have at least approximately the same size. In the opposite case, the algorithm leads to a load imbalance and thus performance loss. Additionally, the distribution of data might be skewed which may cause the load imbalance as well.

Another problem, which is less obvious, is caused by the concatenation of the sub-streams. This operation is fast, since the corresponding operator only forwards its input envelopes to the output; however, it first forwards all envelopes from the first sub-stream, then all envelopes from the second sub-stream etc. Therefore, only one operator is responsible for the creation of the output stream in the beginning which slows down the consumer of the resulting stream temporarily.

Moreover, output envelopes of the sort operators are hot in cache after their creation; however, these envelopes are stored in the input buffers of the consolidate operator (and eventually forced out of the caches) instead of being immediately forwarded to their consumer. We can reduce the size of the input buffers in order to eliminate this effect; however, it prevents the sort operators from working in parallel (see Section 2.4.1).

Because of these reasons, we chose the second approach in our algorithm, i.e., split the input stream by the fast Dispatch operator into equally sized sub-streams, sort the sub-streams in parallel and then merge them together. The schema of this algorithm is depicted in Figure 4.5. The scalability of the whole algorithm is obviously influenced by the scalability of the merge operator; therefore, in the rest of this section, we focus on this operator.

4.4.3 Parallel Merging of Streams

The usual approach to the parallel multi-way merging of sorted sequences/streams used in similar works is based on a tree of two-way merges (see Figure 4.6 which shows the implementation of this algorithm for Bobox). The main problem of this
Figure 4.5: Parallelized sort operator with the single-threaded multi-way merge operator.

approach is the last merge operator which is a serial bottleneck of the whole algorithm. There are methods of overcoming this (see Section 4.6.1 which discusses these methods); however, we decided not to use an algorithm based on two-way merges and implemented a parallelizable operator which is able to merge multiple sub-streams directly.

Figure 4.6: Parallelized sort operator with the merge tree.

Let us focus on the classical algorithm for multi-way merging of $K$ sorted sequences. It keeps one pointer for each sequence. Each pointer is initially set to the beginning of its corresponding sequence. During one step of the algorithm it selects a pointer which points to the smallest value, puts this value to the output and increments the pointer. When the corresponding sequence becomes completely processed, its pointer is omitted from further steps. The algorithm performs this step repeatedly until all sequences are completely processed.

This operation is obviously stateful (see Section 3.2.2) since the positions of the pointers represent its inner state. Therefore, we can try to use the parallelization schema described in Section 3.2.2. We only need to find an algorithm which is able to update the inner state faster than by the simple processing of the tuples. In other words, we need an algorithm which gives us the positions of pointers after the processing of $L$ tuples, where $L$ is the input of the algorithm.

The main idea of the algorithm is to move the pointer by $B$ ($B > 1$) instead of 1 in each step of the algorithm. Let $P_j^B$ be the position of the pointers after $\lceil L/B \rceil$ steps. First notice that for $B = 1$, $P_j^B$ denotes the position in the streams, where would the original algorithm finish after processing $L$ tuples. Then it is easy to see that for arbitrary $B$ and each $j$ it holds that $|P_j^B - P_j^1| \leq B - 1$.

Consider that we performed the algorithm for some $B$ and we got the positions of the pointers $P_j^B$. Now we want to increase the precision (i.e., find the pointers which are closer to $P_j^1$), therefore, we perform the algorithm for some $\hat{B} < B$ to get $P_j^{\hat{B}}$. If $B$ is divisible by $\hat{B}$, the algorithm does not have to start from the
beginning. It is enough to start from the positions which are equal to \( \max(0, \hat{P}_j - B) \) for each \( j \), since these positions are certainly less or equal to \( \hat{P}_j \). In other words, we need to undo the last step of the algorithm for \( B \) for each sequence, since this step might exceed the correct solution for finer \( \hat{B} \).

The number of steps which the algorithm performs to get \( \hat{P}_j \) from \( \hat{P}_j \) is \( O\left(\frac{KB}{B} \right) \), since it is equal to

\[
\left\lfloor \frac{i \cdot L}{\hat{B}} \right\rfloor - \left( \left\lfloor \frac{i \cdot L}{B} \right\rfloor - O(K) \right) \cdot \frac{B}{\hat{B}}
\]

where \( \frac{i \cdot L}{\hat{B}} \) is the number of steps which would have had to be done if the algorithm for \( \hat{B} \) had started from the beginning, \( \frac{i \cdot L}{B} \) is the number of steps which were already performed by the algorithm for \( B \) (and which we want to reuse), \( O(K) \) is the number of steps for \( B \) which were undone and \( \frac{B}{\hat{B}} \) is the number of steps for \( \hat{B} \) which corresponds to one step of the algorithm for \( B \). This is equal to:

\[
\left\lfloor \frac{i \cdot L}{B} \right\rfloor - \left\lfloor \frac{i \cdot L}{\hat{B}} \right\rfloor - O\left(\frac{B}{\hat{B}}\right) + O\left(\frac{B}{\hat{B}}\right) = O\left(\frac{KB}{B} \right)
\]

The main idea of the algorithm is to choose initial \( B \) so that it is a power of two and to find initial \( \hat{P}_j \). Then it repeatedly divides \( B \) by 2 and refines \( \hat{P}_j \) until \( B \) is equal to 1. The whole algorithm for getting positions of the pointers after processing \( L \) tuples is shown in Algorithm 4.1.

**Algorithm 4.1** Algorithm for getting the positions of the pointers after processing \( L \) tuples.

```plaintext
B ← 2^{\lfloor \log_2 L \rfloor} // largest power of two less than L
pos ← 0 // position in the output stream
for j = 1 to K do
    P_j ← 0
end for
while B ≥ 1 do
    for j = 1 to K do // undoing of the last step
        if \( P_j > 0 \) then
            \( P_j ← P_j - B \)
            pos ← pos - B
        end if
    end for
    while pos + B ≤ L do // improving of \( P_j \)
        choose \( j \) so that \( S_j[P_j] \) is minimal
        \( P_j ← P_j + B \)
        pos ← pos + B
    end while
B ← \lfloor B/2 \rfloor
end while
```

The time complexity of this algorithm is \( O(K \log L \log K) \) since there are \( O(\log L) \) phases and each phase takes \( O(K \log K) \) – during each phase, \( O\left(\frac{K \cdot B}{B/2} \right) =
$O(K)$ steps for improving the pointers positions are performed and each step of this algorithm takes $O(\log K)$ since it must find a minimum of $K$ values. This can be done easily for example with a heap data structure.

The implementation of the parallelizable merge operator is now a straightforward application of the schema for parallelizable operators (with the aggregated state updating) from Section 3.2.2. The final algorithm is listed in Algorithm 4.2 ($N$ is the number of all replicas).

**Algorithm 4.2** Algorithm of one replica of the parallelizable merge operator.

```plaintext
size ← optimal size of an envelope
use Algorithm 4.1 to get $P_j$ after processing $RID \cdot size$ tuples
while not finished do
    merge $size$ tuples starting at $P_j$, update $P_j$ and send the envelope
    //skip tuples processed by other replicas
    use Algorithm 4.1 to update $P_j$ after processing $(N - 1) \cdot size$ tuples
end while
```

Only one minor problem remains – multiple tuples with the same values might occur in the input streams. In that case, the problem of finding $P_j$ might have multiple solutions which can lead to the situation when some tuples are twice in the output stream and some are omitted because the replica $i$ may finish its envelope in a different position than replica $i + 1$ starts its consecutive envelope. We solved this by the modification of the comparison function. If it compares two same values from two different streams, it additionally compares the indices of their corresponding sub-streams.

The final schema of the parallel sort operator is shown in Figure 4.7 and the corresponding source code in Bobolang is listed in Listing 4.3.

Listing 4.3: Parallel sort operator in Bobolang.

```plaintext
operator ParallelSort (typename T)->(T)
{
    Dispatch (T)->(T)* dispatch;
    Sort (T)->(T) sort;
    parallel ParallelizableMerge (T)*->(T) merge;
    input -> dispatch -> sort -> merge -> output;
}
```

### 4.4.4 Results

In this Section, we introduce several experiments which show the behaviour of the parallel sorting operator in two scenarios. In the first scenario, we use the sorting operator to sort an array of integers. In the second scenario, we use the sorting operator in a pipeline.

**Sorting of Integers**

In this experiment, we generate an array with 100 million 32b random integers and sort it. We used the `std::sort` function from the STL library [105] shipped with
figure 4.7: execution plan of the parallel sort operator.

gcc v4.8.2 and the tbb::parallel_sort function from the TBB library v4.2 [16] for the comparison with other algorithms. Because our algorithm is not designed to be used for sorting of arrays, we used the execution plan listed in Listing 4.4, where the source operator is a parallelizable operator which reads data from the input array.

Listing 4.4: Execution plan for the sorting of integers.

```cpp
operator main() -> ()
{
    parallel Source() -> (int) source;
    ParallelSort(int) -> (int) sort;
    Sink(int) -> (int) sink;

    input -> source -> sort -> sink -> output;
}
```

The results are shown in Figure 4.8 (the time needed for the initialization of the array is not included). They show that our algorithm have significant overhead in contrary to std::sort or tbb::sort/MT1. This is caused mainly because our algorithm accesses at least three times more memory than these in-place algorithms – the input array, the memory for the envelopes generated by the source operator and the memory for the envelopes generated by the ParallelSort operator. On the other hand, the scalability of our algorithm is better than the scalability of tbb::parallel_sort, (5.5× vs. 6.25× on SMP8 and 6× vs. 12.3× on NUMA64). Unfortunately, the overhead is greater than the benefit on SMP8, therefore, tbb::parallel_sort outperforms our algorithm, whereas our algorithm outperforms tbb::parallel_sort on NUMA64.

Sorting in a Pipeline

In order to show behaviour of our algorithm in the context for which it is intended, we used the execution plan listed in Listing 4.5.

The source operator generates sequence of 10 million 32b random integers and work multiplies each integer by a random number (we used the linear congruent generator [88] to obtain random numbers).

\[2\] In order to measure tbb::sort/MT1, we instantiated tbb::task_scheduler_init class with parameter max_threads = 1.
Listing 4.5: Execution plan for the sorting in a pipeline.

```c++
operator main() ->()
{
    parallel Source() -> (int) src;
    stateless Work(int) -> (int) work1, work2, 
        work3, work4, work5;
    ParallelSort(int) -> (int) sort1, sort2, sort3, sort4;
    Sink(int) -> () sink;

    input -> src -> work1 -> sort1;
    sort1 -> work2 -> sort2;
    sort2 -> work3 -> sort3;
    sort3 -> work4 -> sort4;
    sort4 -> work5 -> sink -> output;
}
```

For the comparison with the TBB library, we implemented the C++ code listed in Listing 4.6 which is equivalent to the execution plan from Listing 4.5.

Listing 4.6: Source code in C++ equivalent to Listing 4.5

```c++
std::vector<int32_t> v(10000000);
tbb::parallel_for(...) , initialize(v), ...);
tbb::parallel_for(...) , work(v), ...);

for (int i = 0; i < 4; i++) {
    tbb::parallel_sort(v.begin(), v.end());
    tbb::parallel_for(...) , worker(v), ...);
}
```

The results are shown in Figure 4.9. Our algorithm is still outperformed by TBB on SMP8; however, the scalability of our algorithm further increased,
whereas the scalability of TBB slightly decreased (5.3× vs. 6.7×). On the other hand, Bobox outperforms 2.7× TBB on NUMA64 and the scalability also further increased, whereas the scalability of TBB slightly decreased (5.4× vs. 20.6×).

4.5 Merge Join Operator

Merge join is a very efficient join algorithm when both relations are sorted by the join attribute. The merge join algorithm works as follows. It reads the inputs (denoted as left and right) and finds sequences of the same values of the join attributes in the left and right input. Then it performs the cross product of these sequences. The pseudocode of the traditional implementation of merge join is shown in Algorithm 4.3.

```
Obviously, if we take any value \( V \) of the join attribute, than all tuples less than \( V \) from both inputs can be processed independently on the tuples which are greater or equal to \( V \). A common approach to the merge join parallelization is splitting the inputs into multiple parts by \( N - 1 \) values \( V_i \) and process them in parallel with \( N \) worker threads.

However, there are two problems with the selection of appropriate values \( V_i \):

1. The inputs of the join are data streams; therefore, we do not know how many input tuples there are until we receive all of them. Because of the same reason, we do not know the distribution of the input data in advance. Therefore, we cannot easily select \( V_i \), so that the resulting parts have approximately the same size.

2. The distribution of data could be skewed; therefore, it might be impossible to utilize processing units uniformly.

For the sake of simplicity, we first describe a simplified algorithm for the joining without duplicated join attribute values in Section 4.5.1. Then we extend the algorithm to take the duplicates into account in Section 4.5.2.

4.5.1 Join without Duplicates

In this section, we describe the algorithm which assumes that the input streams do not contain duplicated values of join attributes.

---

3If the inputs are not sorted, the merge join operator is prepended by the sort operator(s) and the algorithm is called sort-merge join.
Algorithm 4.3 Traditional merge join algorithm.

\[
\textbf{while } \text{left.has.next} \land \text{right.has.next do}
\]
\[
\begin{align*}
\text{left Tuple} & \leftarrow \text{left.current} \\
\text{right Tuple} & \leftarrow \text{right.current} \\
\text{if } \text{left Tuple} = \text{right Tuple then} & \\
\text{left seq.append}(\text{left Tuple}) \\
\text{left.move.next}() \\
\text{while } \text{left has.next} \land \text{left.current} = \text{left Tuple do} & \\
\text{left seq.append}(\text{left Tuple}) \\
\text{left.move.next}() \\
\text{end while} \\
\text{right seq.append}(\text{right Tuple}) \\
\text{right.move.next}() \\
\text{while } \text{right has.next} \land \text{right.current} = \text{right Tuple do} & \\
\text{right seq.append}(\text{right Tuple}) \\
\text{right.move.next}() \\
\text{end while} \\
\text{output cross product(left seq, right seq)} \\
\text{left seq.clear} \\
\text{right seq.clear} \\
\text{else if } \text{left Tuple} < \text{right Tuple then} & \\
\text{left.move.next}() \\
\text{else} & \\
\text{right.move.next}() \\
\text{end if} \\
\text{end while}
\]

The main idea of this algorithm is to transform the input streams (i.e., the flows of envelopes) into the flow of pairs of envelopes so that the tuples in these pairs can be joined independently on the other pairs. In other words, this algorithm transforms the stateful merge join operator into a stateless one. The source code of the ParallelMergeJoinWithoutDuplicates operator is listed in Listing 4.7 and its execution plan is depicted in Figure 4.10.

![Figure 4.10: Execution plan of the ParallelMergeJoinWithoutDuplicates operator.](image)

Now, we describe the idea and the algorithm of the Preprocess operator. Consider the first envelope \texttt{left.env} from the left input and the first envelope \texttt{right.env} from the right input. We denote the last tuple (the highest value) in
operator ParallelMergeJoinWithoutDuplicates
<typename L>, (typename R) -> (typename T)
{
Preprocess(L), (R) -> (L), (R) preprocess;
stateless MergeJoin(L), (R) -> (T) merge_join;

input[0] -> [0] preprocess[0] -> [0] merge_join;
merge_join -> output;
}

left_env by last_left and the last tuple in right_env by last_right. Then, one of these three cases occurs:

1. last_left > last_right – in this case, we can split left_env into two parts.
   The first part contains tuples which are less or equal to last_right and the second part contains the rest. The first part of left_env and the whole right_env is the first pair of envelopes.

2. last_left < last_right – in this case, we split right_env analogically as in the former case.

3. last_left = last_right – in this case, the whole left_env and the whole right_env might be joined together, i.e., the whole left_env and the whole right_env is the first pair of envelopes.

This leads to the algorithm of the Preprocess operator listed in Algorithm 4.4.

The Preprocess operator might seem to be a bottleneck of the algorithm. However, the only non-trivial task performed by this operator is the finding of the positions where to split the envelopes. This can be done by a binary search which has time complexity $O(\log L)$ where $L$ is the number of tuples in the envelope. Therefore, it access only $O(\log L)$ of the envelope and is still much faster than the consecutive replicas of the MergeJoin operator which process all the tuples even though in parallel. Recall that the splitting of an envelope is also a fast operation (see Section 2.6.2).

4.5.2 Join with Duplicates

Without the duplicates, the Preprocess operator is able to generate pairs of envelopes which can be processed independently. However, the possibility of their existence complicates the algorithm. Consider a situation depicted in Figure 4.11.

The first pair of envelopes might be processed independently; however, in order to process the second pair, the operator needs the access to the tuples in the first pair as well. Therefore, the MergeJoin cannot be stateless in this case and has to be implemented as a parallelizable one (see Section 3.2.2) in order to receive all pairs of envelopes for the case when there are sequences of the same tuples across multiple envelopes.
Algorithm 4.4 Algorithm of the Preprocess operator without duplicates.

Algorithm:

\[
\begin{align*}
    & \text{left}_\text{env} \leftarrow \text{next envelope from left input} \\
    & \text{right}_\text{env} \leftarrow \text{next envelope from right input} \\
    \textbf{while} & \ \text{left}_\text{env} \neq \text{poisoned\_pill} \land \text{right}_\text{env} \neq \text{poisoned\_pill} \ \textbf{do} \\
    & \text{last}_\text{left} \leftarrow \text{left}_\text{env}[\text{left}_\text{env}.\text{size} - 1] \\
    & \text{last}_\text{right} \leftarrow \text{right}_\text{env}[\text{right}_\text{env}.\text{size} - 1] \\
    \textbf{if} & \ \text{left}_\text{last} > \text{right}_\text{last} \ \textbf{then} \\
    & \text{split} \ \text{left}_\text{env} \ \text{to} \ \text{left}_\text{first} \ \text{and} \ \text{left}_\text{second} \\
    & \text{send} \ \text{right}_\text{env} \ \text{to} \ \text{the} \ \text{right} \ \text{output} \\
    & \text{send} \ \text{left}_\text{first} \ \text{to} \ \text{the} \ \text{left} \ \text{output} \\
    & \text{left}_\text{env} \leftarrow \text{left}_\text{second} \\
    & \text{right}_\text{env} \leftarrow \text{next envelope from right input} \\
    \textbf{else if} & \ \text{left}_\text{last} < \text{right}_\text{last} \ \textbf{then} \\
    & \text{split} \ \text{right}_\text{env} \ \text{to} \ \text{right}_\text{first} \ \text{and} \ \text{right}_\text{second} \\
    & \text{send} \ \text{right}_\text{first} \ \text{to} \ \text{the} \ \text{right} \ \text{output} \\
    & \text{send} \ \text{left}_\text{env} \ \text{to} \ \text{the} \ \text{left} \ \text{output} \\
    & \text{left}_\text{env} \leftarrow \text{next envelope from left input} \\
    & \text{right}_\text{env} \leftarrow \text{right}_\text{second} \\
    \textbf{else} \\
    & \text{send} \ \text{right}_\text{env} \ \text{to} \ \text{the} \ \text{right} \ \text{output} \\
    & \text{send} \ \text{left}_\text{env} \ \text{to} \ \text{the} \ \text{left} \ \text{output} \\
    & \text{left}_\text{env} \leftarrow \text{next envelope from left input} \\
    & \text{right}_\text{env} \leftarrow \text{next envelope from right input} \\
\end{align*}
\]

end while

Figure 4.11: Cross product across multiple envelopes during the processing of the second pair of envelopes.

Now, we describe the idea of the ParallelizableMergeJoin operator. The replicas of this operator receive all input pairs of envelopes, join these pairs alternately (using the previous envelopes eventually), keep envelopes which may be needed in future and discards the others. The decision whether to keep or discard an envelope \( E \) can be done easily by the comparison of the last tuple in the envelope \( E \) with the last tuple of the following envelope (separately for each input). If these tuples are same, then the envelope \( E \) contains the part of the sequence which continues in the following envelope and we need to keep it. Otherwise, we can discard the envelope \( E \) (and all envelopes prior to envelope \( E \)).
Algorithm 4.5 Algorithm of the ParallelizableMergeJoin operator.

```
left_env ← next envelope from left input
right_env ← next envelope from right input
phase ← 0
current_left_sequence ← empty
current_right_sequence ← empty

while left_env ≠ poisoned pill ∧ right_env ≠ poisoned pill do
    if phase mod P = RID then
        do the join of left_env and right_env using stored envelopes eventually
    end if
    phase ← phase + 1
    last_left ← left_env[left_env.size − 1]
    if last_left ≠ current_left_seq then
        current_left_seq ← last_left
        discard all stored left envelopes
    end if
    store left_env
    last_right ← right_env[right_env.size − 1]
    if last_right ≠ current_right_seq then
        current_right_seq ← last_right
        discard all stored right envelopes
    end if
    store right_env
    left_env ← next envelope from left input
    right_env ← next envelope from right input
end while
```

The complete algorithm of the ParallelizableMergeJoin operator is shown in Algorithm 4.5. This algorithm ensures that the operator works correctly even in the case of duplicated values of the join attribute. On the other hand, cross product is very time consuming and long sequences of same values may lead to the significant load imbalance of the replicas which may decrease the scalability of the operator when the input data are skewed.

Our solution is to artificially decrease the size of envelopes, so that such sequences are spread across more envelopes which increases the granularity of the tasks and decreases the probability of the load imbalance. Therefore, we slightly modified also the Preprocess operator. This operator additionally selects N samples from each envelope where N is the number of the replicas of the ParallelizableMergeJoin operator. If it detects that there is a sequence of the same value in both inputs, it splits both sequences to multiple parts. The longer the sequences are, the smaller the parts are.

The final schema of the complete parallel merge join is shown in Figure 4.12 and the corresponding source code in Bobolang is listed in Listing 4.8.
Listing 4.8: Parallel merge join operator.

```cpp
operator ParallelMergeJoin
    (typename L),(typename R)->(typename T)
{
    Preprocess(L),(R)->(L),(R) preprocess;
    parallel ParallelizableMergeJoin(L),(R)->(T) merge_join;

    input[0] -> [0]preprocess[0] -> [0]merge_join;
    merge_join -> output;
}
```

Figure 4.12: Execution plan of the ParallelMergeJoin operator with duplicates

4.5.3 Results

As in Section 4.3.1, we used dataset of SP²Bench benchmark for all experiments and several synthetic queries which illustrate the behaviour of the algorithm under various circumstances.

Scalability of the Merge Join

For the first experiment, we used the following query E2 on 5m dataset:

```
SELECT ?article1 ?article2
WHERE {
    ?article1 swrc:journal ?journal .
    ?article2 swrc:journal ?journal
    FILTER (?article1 = ?article2)
}
```

Figure 4.13: Query plan of the query E2.

The query generates all pairs of articles which were published in the same journal and then selects those pairs where \( ?\text{article1} = ?\text{article2} \), i.e., it simply
selects all articles which were published in a journal. The execution plan of the
query is depicted in Figure 4.13.

This experiment shows the scalability of the merge join algorithm when its
inputs contain non-trivial sequences of tuples with the same join attribute (i.e.,
the join produces high number of tuples) and with the join condition with very
high selectivity (i.e., the number of resulting tuples is relatively low). Since both
inputs of the join are sorted by join attribute, this algorithm shows only the
scalability of merge join and does not include eventual sorting. Notice that the
lengths of these sequences (i.e., the number of articles per journal) are uniform
and none is longer than 84 tuples.

The results shown in Figure 4.14 prove that the algorithm scales well on SMP8
(the parallelized version is 5.6× faster than the serial version). On NUMA64,
the algorithm is 5× faster with 16 replicas of the ParallelizableMergeJoin
operator and only 3.3× faster with 64 replicas. It confirms our observation from
Section 3.3.2 that for relatively small and uniform tasks it is better to keep their
evaluation on one NUMA node.

Very important result is that the parallelized version of merge join introduces
only negligible overhead.

**Skewness Resistance**

Experiments in this section prove that the merge join performs well even in cases
when the data are skewed.

For the first measurement, we used the following query E3 on 5m dataset:

```sql
SELECT ?article1 ?article2
WHERE {
    ?article1 dcterms:issued ?year .
    ?article2 dcterms:issued ?year
    FILTER (?article1 = ?article2)
}
```

This query is similar to the query E2, but it generates all pairs of articles
which were published in the same year. In this case, the lengths of the sequences
are very skewed (see Figure 4.15 which shows how many articles were published in
particular years). Despite this fact, the results shown in Figure 4.16 prove that
the algorithm scales well (7× on SMP8 and 15.7× on NUMA64). In contrary
to the previous experiment, the higher number of replicas further increases the
scalability of the algorithm. The main reason is that the computation of cross products is very time consuming for later years, i.e., it benefits from the higher number of processing units. The decision how many replicas is optimal should be made by the query optimizer (see discussion in Section 3.3.2).

![Figure 4.15: Number of articles per year in the 5m SP²Bench dataset.](image)

(a) Results on SMP8. (b) Results on NUMA64.

Figure 4.16: Results for query E3.

The last experiment reuses query E1 from Section 4.3.1, but we forced the compiler to use the merge join operator. Recall that the query E1 is as follows:

```sql
SELECT ?article1 ?article2
WHERE {
  ?article1 ?type bench:Inproceedings .
  ?article2 ?type bench:Inproceedings
  FILTER (?article1 = ?article2)
}
```

In this case, both inputs of the merge join consist of exactly one sequence of the same value of the join attribute (`rdf:type`). Despite the fact that this kind of join is impossible to parallelize by partitioning, our algorithm scales very well according to the results shown in Figure 4.17 (6.8× on SMP8 and 20.3× on NUMA64).

Notice that despite the fact that merge join and nested loop join has the same time complexity on this query, the merge join is faster than nested loop join (compare these results with the results in Figure 4.4). The reason is that in case of nested loop join, the real join condition is `?type from left input = ?type from right input AND ?article1 = ?article2`, while in case of merge join, the join condition is only `?article1 = ?article2`. Therefore, the evaluation of this condition is faster in case of merge join.
4.6 Related Work

4.6.1 Sort

Sorting is a very intensively researched area and many algorithms, methods and optimizations are already known \[87\]. The research on parallel sorting is also very popular and many algorithms had been published before the parallel computers became common \[83\]. However, these algorithms typically use some kind of theoretical computation model. A typical example is parallel merge sort \[46\] for CREW/EREW PRAM which has the optimal time complexity \(O(\log(N))\) on \(N\) processing units. However, an implementation of such algorithms for the limited number of processing units would be unusable.

Algorithms for modern systems (multi-cores or many-cores) typically follow the same schema as we do, i.e., they split the input into several blocks, sort them in parallel and then merge all the blocks into the result. Since the parallel merging of the streams is the main contribution of our approach, we focus mainly on this phase.

Generally, two approaches for the merging are used – binary merge tree or multi-way merge. In the case of binary merge tree, the merging of the last two sequences is obviously bottleneck and the algorithms deal with that differently. The algorithm in \[44\] ignores this problem and relies on the fact that there is enough nodes in the tree which may work in parallel.

Slightly better solution is used in \[74\]. The algorithm is able to perform the merging of the last two sequences in parallel; however, it is able to utilize only two threads – the first performs the merging from the beginning of the sequences and the second from the end.

The algorithm in \[117\] partitions both input sequences for merging into multiple sub-sequences which can be merged in parallel. This follows the idea used in \[69\]. The partitioning is done using samples and the samples are selected from both input sequences in order to limit the maximal size of each sub-sequence. However, the resulting sequences created by merging of these sub-sequences may have variable size. This is unusable in our algorithm, since we want to ensure that each output envelope has optimal size. On the other, it is more skew resistant than the algorithm used in AA-sort \[76\]. This algorithm selects the samples only from the first input which may lead to situation when the second input is partitioned to one non-empty sub-sequence only.

Algorithms which ensures that the input sequences are partitioned evenly are known for a long time \[64, 128, 129\]. They are able to split each input sequence
to $N$ parts which can be merged independently so that each processing unit merge the same number of items. Such method is used for example in modern algorithm [118].

The parallel multi-way merge algorithm was introduced in [63]. The algorithm works as follows – it first estimates where the processing unit number $i$ should start (it assumes uniform distribution of values in the inputs) and then refines the estimation until the final solution is obtained. On the other hand, our algorithm basically starts from the beginning of the sequences and advances to the correct positions (sometime a step back is needed); therefore, our algorithm fits better to the model of data stream processing. The resulting time complexity is the same.

To the best of our knowledge, no modern algorithm uses parallel multi-way merge algorithm. Only [118] mentions it as an alternative to the merge tree. Notice that parallel multi-way merge algorithm exists for PRAM model [73].

Another difference between cited sorting algorithms and our solution is that they assume that their input is an array with the items and the output is also an array (either the same as the input in case of in-place algorithm, or different) with the sorted items, i.e., all items are available and might be accessed randomly and the result might be produced in any order. This allows parallelization schema when a processing unit number $i$ creates $i$-th part of the output. This is in contrast to our algorithm, which expects that the input is given sequentially and that the output should be also produced sequentially, i.e., it does not have random access to all items and the processing units should alternate on the creation of the resulting stream with a finer granularity.

Very close to our work is [39] which focuses on sorting in data stream environment Auto-Pipe [65], which is similar to Bobox. However, it focuses only on the sorting as whole and does not analyse the influence of the algorithm on the rest of the execution plan through pipeline parallelism. Additionally, the algorithm for the parallel merge of streams is the straightforward implementation of the merge tree (see Figure 4.6), i.e., the last merge operator is a serial bottleneck.

Our single-threaded sorting algorithm does not use any new algorithm. Its main contribution is to show that approaches used in external sorting [87], which are considered to be impractical for sorting in internal memory [32], are effective when used in data stream systems, even though the streams are stored in the memory.

### 4.6.2 Merge Join

Many works on parallel merge join focus mainly on the sorting phase, since it is the most expensive part of the algorithm (up to 98% according to [84]). Therefore, this section discusses not only the merge join algorithm proposed in Section 4.5 but also its combination with the sorting algorithm introduced in Section 4.4. For instance, the algorithm used in [84] adopts the sorting algorithm [44] (discussed above) and enriches it for the possibility of sorting key-value pairs and the description of the merge join algorithm is almost omitted.

The work [20] introduces NUMA-aware Massive Parallel Sort-Merge join algorithm (MPSM). The algorithm uses range partitioning of one of the relations, so that the sort and merge phase can be done independently for each partition. The second relation is distributed (not range partitioned) uniformly across NU-
MA nodes and each partition is sorted locally on that nodes. Therefore, despite the fact that the merge join phase of one partition can be done in parallel with others, this phase needs to access the memory of all other nodes. The work also proposes how to handle data skew – it computes histograms representing distribution of the relations and then partitions them so that each worker thread is assigned balanced workload.

The work [25] revises parallel in memory hash and sort-merge joins. In case of sort-merge join, it continues the previous work and introduces several optimizations to the MPSM. Additionally, it introduces the m-way and m-pass algorithms which are other NUMA aware implementations of sort-merge join. Both algorithms basically sort and partition both relations (they differ in the sorting algorithm) among NUMA nodes. If the distributions of both relations are similar, then the merge join phase is performed almost locally on the NUMA nodes. This and several other optimizations cause that m-way algorithm outperforms MPSM algorithm.

To the best of our knowledge, all previous algorithms lack the support for join with duplicate values of the join attribute, which makes the problem easier, since it does not have to calculate cross product of such tuples. Additionally, handling data skew using histograms does not fit the model of data streams since we do not know the distribution of values in advance (tuples are received sequentially) and the used parallelization schema does not fit the model as well because of the same reason as mentioned in previous Section. On the other, our algorithm does not have any explicit support of NUMA, it relies only on the NUMA awareness of the underlying streaming system. This might lead to suboptimal results.

Algorithm proposed in [50, 51] called progressive-merge join tries to utilize pipeline parallelism of the merge join. It runs the sort phase in parallel with the merge phase. Additionally, it tries to avoid the blocking effect of the sort-merge join since the first output tuple of the sort-merge join is created traditionally after the last input tuple is received by the sort phase. This algorithm is able to produce tuples already during the creation of runs in the sorting phase. Our algorithm lacks this feature; however, our parallel sort and merge join algorithm is able to utilize all available processing units so the blocking effect is negligible.

Since the implementation of the merge join is used in our parallel SPARQL engine (see Section 5), worth mentioning is the work [68] which proposes parallel algorithms for join computations of SPARQL queries. This algorithm is based on partitioning; therefore, it is unable to parallelize query E3 (see Section 4.5.3).

4.7 Chapter Summary and Future Work

In this chapter, we introduced parallel algorithms which are heavily used in the worlds of databases. The filter operator and the nested loop join operator was intended mainly as simple examples which demonstrate basic features of Bobox and Bobolang.

In Section 4.4 we introduced parallel sort algorithm. This algorithm is optimized for the environment of streaming systems; therefore, it introduces significant overhead in comparison with in-place algorithms. On the other hand, it scales very well; therefore, on highly parallel systems it significantly outperforms code in C++ parallelized by the TBB library.
The Section 4.5 proved that Bobox and Bobolang are powerful enough to
design and implement highly scalable merge join operator. This algorithm scales
well also on skewed data and even in extreme cases when cross product of both
complete inputs must be performed.

In future, we want to focus on our parallel sort algorithm and try to reduce
the overhead caused by the parallelization. We would also like to introduce a cost
model which would estimate the overhead and possible speed up of the algorithms
according to the number of replicas of boxes used in the execution plan. This
would allow the query optimizer to generate more optimal query plans.
5. In-memory Parallel SPARQL Engine

In this chapter, we put our innovations together and introduce the implementation of the runtime for parallel in-memory SPARQL engine for querying RDF data as an example of parallel data processing application. This application makes use of the Bobox framework, the Bobolang language as well as parallel algorithms introduced in Chapter 4.

5.1 Introduction

The Resource Description Framework (RDF) is a framework for representing information in the Web [86]. The main idea of its data model is to allow making any statement about any resource. The statements are represented as triples. Each RDF triple has three parts (called RDF terms):

1. subject (S) – identifies the subject of the statement, i.e., the resource
2. predicate (P) – denotes a relationship between the subject and the object
3. object (O) – identifies the object of the statement

For example, statements in the simplified Notation3 format [30] about this thesis would look like as follows:

<http://localhost/theses/1> type PhDThesis.
<http://localhost/theses/1> title "Parallel Processing of Data".
<http://localhost/persons/13> type Person.
<http://localhost/persons/13> name "Zbynek Falt".

It states that the resource <http://localhost/theses/1> is a PhD thesis, the title of the thesis is "Parallel Processing of Data" and its author is the resource <http://localhost/persons/13>. The resource <http://localhost/persons/13> is a person and its name is "Zbynek Falt".

As the amount of RDF data and the complexity of queries grow, the need of systems which are able to process them efficiently increases. Therefore, we focused on development of one such system and implemented the parallel SPARQL [112] engine.

SPARQL is a popular language for querying RDF data. Typical SPARQL query basically contains a set of triple patterns. These triple patterns look like ordinary RDF triples with the difference that each of the subject, predicate and object may be a variable instead of a RDF term (variables start with the ’?’ character). The task of the SPARQL engine is to find all bindings for the variables so that the triple patterns match some triples in the queried RDF data.

For example a SPARQL query which finds titles of all PhD thesis and the names of their authors is as follows:

1. I.e., the whole database is loaded into the main memory during the startup of the engine.
SELECT ?title ?name
WHERE {
  ?thesis type PhDThesis.
  ?author name ?name.
}

Except triple patterns, queries may contain other patterns like optional, union, graph or minus pattern. However, their detailed description is beyond the scope of this thesis and is not necessary for the understanding of the rest of this chapter.

5.2 Implementation of the Engine

In this section, we focus only on the important implementation details which influence the performance of the engine most and are relevant to this work. All other details can be found on the attached DVD.

5.2.1 Query Compilation and Optimization

The query compilation and optimization is performed in a few steps by separate modules of the application as shown in Figure 5.1. After the query is parsed and validated, the SPARQL front-end prepare optimal query plan according to several heuristics, cost models and statistics of the RDF data. The query compilation and optimization is not the contribution of this thesis; therefore, the detailed description of this process is beyond the scope of the thesis and can be found in [61].

Figure 5.1: Architecture of the query compiler and optimizer.

Notice that query plans generated by the query compiler consist of common operators used in database processing such as scans, joins, filters, sorts or unions.
However, they need to be modified according to the SPARQL algebra \[47\]. Several examples of these plans can be found in Appendix A.

5.2.2 Representation of RDF Terms

RDF data are typically very redundant, since they contain many duplicates. Many triples typically share the same subjects, predicates or objects (see for instance the example from Section 5.1). To reduce the size of memory needed for storing the database, we keep only one instance of every distinct term in a memory. Besides the fact that this representation saves the memory, we may represent each term unambiguously by its address in the memory. Therefore, we can test equality of two terms only by the comparison of their addresses, which speeds up that test significantly.

Additionally, if we need to access the content of a term (e.g. for evaluation of a non-trivial filter condition) the address can be easily dereferenced to get the real value of the term.

5.2.3 Representation of RDF Database

There are three common approaches to storing RDF data \[119\]:

- **Triple table** – all triples are stored in one big table.

- **Property tables** – one table for each type of resource is created and its properties are stored in that table. For the example from Section 5.1 two tables would be created – one for theses (with three columns \textit{thesis}, \textit{title}, and \textit{author}) and one for persons (with two columns \textit{person} and \textit{name}). This approach basically transforms the RDF database in a traditional relation database.

- **Vertically partitioned table** – one two-column table for each predicate is created. The objects of that predicate are stored in the first column and its subjects in the second.

Since we focused mainly on the optimization of the query processing, we choose the simplest approach, i.e., the triple table. In order to speed up the scan operator (see Section 5.2.7) and eventual consecutive merge join operators, we keep six replicas of the triple table sorted in all possible orders (SPO, SOP, OPS, OSP, POS, and PSO). This approach is called Hexastore \[134\].

Notice that the triples in a triple table are stored in the same way as tuples in envelopes, i.e., by columns. Therefore, we may consider each triple table as one large envelope with three columns.

5.2.4 Format of Envelopes

The columns of an envelope correspond to a subset of the variables in the query and each tuple in the envelope corresponds to one possible binding of those variables.

If the variable is unbound in a particular tuple, then the value of the corresponding item in the corresponding tuple is set to the empty value (represented as a special RDF term).
5.2.5 Join Operator

Merge join and nested loop join are implemented in the same way as described in Chapter 4. Unfortunately, join of two relations when at least one is not sorted by the join attribute(s) is very common. In this case, merge join is unusable and nested loop join is too inefficient. Typical solution is to use either hash join or sort-merge join algorithm. Sort-merge join is in fact merge join prepended by sort operator(s) which sorts the relation by the join attribute so that the following merge join can work correctly.

On the other hand, hash join does not need its input sorted so it might look like better choice. Therefore, we tried to implement this algorithm in the Bobox framework; however, we identified two problems in its preliminary implementation:

- Hash join must read the whole one input before starting to read the second one. This causes that the part of the plan which produces data for the second input is blocked because of the limited input buffers of the operator until the first input is completely processed. This restricts the parallel evaluation of the plan.

- The construction of the hash table during the processing of the first input and the searching in the table during the processing of second input caused many cache misses and totally broke all optimization techniques of Bobox which try to increase data locality.

Despite the fact that the implementation was unoptimized and there was space for optimization, we decided to use sort-merge join, since it was significantly faster in all cases.

5.2.6 Left Join Operator

Left join in SPARQL is basically similar to the left join in SQL, i.e., the tuples from the left input which do not meet the join condition (i.e., they are not joined with any tuple from the right input) are also passed to the output and the variables which should come from the right input are set as empty value [47] (see Section 5.2.4).

In the implementation of this operator, we slightly modified the code of nested loop join and merge join operator. Each envelope from the left input is extended by one column of booleans initially set to false and every time when a tuple from the left is joined with a tuple from the right, we set the corresponding boolean value to true. Therefore, we know which tuples were not joined and should be passed to the output.

This modification has to be implemented carefully, since some tuples might be processed concurrently by several replicas. In the real implementation, we made use of the fact that replicas can share envelopes and that the writing/reading of boolean values is atomic on common shared-memory architectures.

5.2.7 Scan Operator

The objective of the scan operator is to fetch all triples that match the given triple pattern from RDF database. The scan operator first selects table sorted
appropriately (see Section 5.2.3) and then it uses binary search to find a range within the triple table (see Section 5.2.3) where the required triples are. If there is multiple choices (e.g., if only the predicate is given and the subject and the object are variables, it can choose either PSO or POS table), it chooses that one which leads to the lower number of sort operators in the eventual consecutive sort-merge join operators.

In the implementation, we made use of the fact that a triple table is technically one large envelope, i.e., it might be split by the split function (see Section 2.6.2). Therefore, when the scan operator finds the range of triples, it splits this range into multiple envelopes so that they have preferred size and sends them out. Since this operation is very fast, we did not parallelize the scan operator.

5.2.8 Sort Operator

The sort operator uses the algorithm described in Section 4.4. There is only one minor change in the implementation of the single-threaded sort operator – it does not sort the tuples in incoming envelopes directly. It creates the array of pointers for each incoming envelope which initially points to the tuples within that envelope and sorts these pointers instead of the tuples within the envelope. It has two consequences:

- It avoids unnecessary copying of tuples during their sorting (only pointers are copied) and thus unnecessary accesses to the columns which do not determine the resulting order.
- It does not modify the content of incoming envelopes. This behaviour is used in the implementation of the scan and the union operator.

5.2.9 Distinct Operator

The distinct operator should output only unique tuples. We implemented this operator by the modification of a sort operator. The first sorting phase is completely the same; however, during the merging phase, the duplicated tuples are omitted from the output. It is simple operation since they are next to each other after the sorting.

We decided to not implement the hash version of the distinct operator because of the same reasons as hash join. On the other hand, if there is a large number of duplicates then an algorithm based on hashing might outperform the algorithm based on the sorting since it is able to drop out the duplicate tuples immediately. We plan to enhance the query optimizer so that it uses the data statistics in order to select the appropriate algorithm.

5.2.10 Union Operator

The union operator unites its input streams to one output stream. The only problem is that one input may contain bindings of different variables than the second. In this case, the missing variables should considered be as unbound, i.e., incoming envelopes should be extended by columns of empty values on appropriate positions. In the implementation we made use of the fact that all operators
access their input envelopes as read-only (the small exceptions are left joins but it does not play a role in this case). Therefore, we can prepare one column of empty values in advance and use it for the extending of all envelopes. This operation is fast enough (it does not access the tuples in envelopes) so that there was no need to parallelize it.

5.2.11 Slice Operator

The task of the slice operator is to select a sub-stream of incoming tuples of a given length starting at a given index, i.e., drop out all tuples from the input stream with the index less than the given index and greater or equal to the given index plus the given length.

In the implementation of this operator, we utilize the split function which is able to split envelopes efficiently, i.e., the operator splits the envelope in which the required sub-stream begins and the envelope in which the sub-stream ends. Then it may simply forward the envelopes which contain the resulting sub-stream and drop out the others. Since this operation is fast enough we did not implement a parallelized version.

Notice that when the last valid envelope is sent to the output, the slice operator stops reading the input and sends the poisoned pill immediately. This can speed up the query processing since it terminates the evaluation of the query prematurely and avoids the unnecessary creation of tuples which would be dropped out anyway.

5.3 Results

5.3.1 Inter-operator Parallelization

In the first experiment, we measured the speed up caused by the inter-operator parallelization exploited by Bobox automatically. In this experiment, we used execution plans which are not parallelized, i.e., each operator of the query plan is implemented by exactly one box in Bobox. We evaluated these execution plans on SMP8 in two settings – once with only one worker thread (the ST columns) and once with all 8 worker threads (the MT columns).

The results are shown in Figure 5.2 and proved that the automatic inter-operator parallelization causes significant speed-up. This speed-up highly depends on the structure of the execution plan. For example Q11 has only one computational intensive operator (the sort operator); therefore, its speed up is negligible. The same holds for Q6, where the most time consuming operator is nested loop join. On the other hand, other queries are complex enough and use more computation intensive operators than one, so that they benefit from inter-operator parallelization, in several queries up to 50%.

The scalability is also limited by the sort operators. This operator is optimized to utilize pipeline parallelism; however, it still prevents the situation when the part of the execution plan before the sort operator runs in parallel with the other part (see Section 4.3) of that plan.

Since no query was able to fully utilize all 8 processing units on SMP8, we did not perform this experiment on NUMA64.
5.3.2 Intra-operator Parallelization

In the second experiment, we focused on the speed up caused by the intra-operator parallelization of all operators. We used the same method as in Chapter 4. The ST columns show the time spent by the single-threaded evaluation of non-parallelized version of the plan, the MT1 columns show the time spent by the single-threaded evaluation of parallelized version of the plan (it illustrates the overhead caused by the parallelization of the plan) and the MT columns shows the time spent by the parallel evaluation of the parallelized plan, i.e., the final results.

The results are shown in Figure 5.3 and in Figure 5.4. According to our expectations, intra-operator parallelism increases the scalability and causes a significant speed up on multiprocessor systems in most cases. On queries which are fast enough in serial version, the speed up is not so significant (query Q2); however, the speed up is up to 7.5× on queries such as Q6, Q7, or Q9 on SMP8 and up to 28× on query Q6 on NUMA64. In average over all measured queries, the speed up is 5.6× on SMP8 and 10.5× on NUMA64.

Notice that in several cases (e.g., query Q4 on SMP8, or query Q9 on NUMA64), MT1 is paradoxically faster than ST. This is caused by the fact that operators like merge join or nested loop join decreases the size of envelopes in their preprocessing phase (see Sections 4.5 and 4.3) which may cause better utilization of caches (see Section 2.6.3). However, generally, MT1 is slower than ST as expected.
5.3.3 Comparison with other Engines

The last set of experiments compares the Bobox SPARQL engine with the other mainstream SPARQL engines, such as Sesame v2.0 \cite{33}, Jena v2.7.4 with TDB v0.9.4 \cite{79} and Virtuoso v6.1.6.3127 (multithreaded) \cite{133}. They follow client-server architecture and we provide sum of the times of client and server processes. The Bobox engine was compiled as a single application; we applied timers in the way that document loading times were excluded to be comparable with a server that has data already prepared. The queries which were unable to evaluate within a time limit (which was set to 30min) are marked as N/A.

The results shown in Figure 5.5 indicate very good potential of our solution. In parallel version it significantly outperforms other engines in all queries (some queries are orders of magnitude faster) except query Q6 and Q8. These queries can be speeded up by the decomposition of the filter conditions, which our optimizer does not perform. Therefore, our execution plan is far from being optimal and is outperformed by other engines.
5.4 Related Work

Currently, a large number of SPARQL engines is available [17]. Well-known engines are Jena/ARQ [102, 14], Sesame [33], 3store [71], AllegroGraph [18], OWLIM [85] or Virtuoso [54]. These engines are complex database management systems which are ready to be used in real applications. They are optimized to be scalable with the respect to the size of the database, to the number of physical storages, or to fully implement related standards. However, they are typically not optimized to the computational-intensive queries (see the results of SP²Bench in Section 5.3.3 where some of them were unable to evaluate several queries within the time limit).

Besides these systems, there are projects which might be considered as scientific prototypes. For example RDF-3X [106, 108] is one of them and is currently considered to be one of the fastest RDF engines [75, 136] nowadays. This engine uses B+-tree to store the triples, compressed and aggregated indexes, effective query optimizer. However, it lacks support of intra-query parallelization. On the other hand, it uses techniques like sideways information passing [107] which typically cannot be used in parallel version of the operators.

The ideas used in RDF-3X were taken over in work [68] (already discussed in Section 4.6.2) which tries to parallelize join computations; however, the scalability of that solution is according to the experiments poor.

Another efficient solution is TripleBit [136]. This storage outperforms the RDF-3X; however, it still lacks support of intra-query parallelization (although the storage is designed to support that).

Worth mentioning is also the work [75], which introduces distributed SPARQL engine. Its main idea is to decompose SPARQL queries to several fragments which might be evaluated locally on the nodes of the distributed system. It also proposes technique called n-hop guarantee for partitioning and replication the data across the nodes in order to minimize the communication needed to join the intermediate results. This is in contrast to other distributed solutions like SHARD [115], YARS2 [72] which use hash partitioning. The n-hop guarantee enables to build a distributed solution on the top of the single-node RDF engines (the work uses RDF-3X engine and the distribution is done through Hadoop [135]), i.e., the speeding up of non-distributed engines speeds up distributed engines.

To the best of our knowledge, the closest work to our solution is [38] which presents parallel, in-memory SPARQL engine for shared-memory systems. However, their approach is different than ours. The engine compiles the SPARQL queries and generates source code in the C language instrumented by the OpenMP pragmas [41] which ensures its parallel run. The generator of the code uses Ullmann’s algorithm for subgraph isomorphism [126] enhanced by features like filtering or the optional matching. This technique searches the RDF graph according to the rules derived from the given query in parallel and writes out the found solutions immediately. It is in contrast to relation-algebra techniques used in our engine which creates many intermediate relations which are processed by operators in order to get the final relation (variable bindings).
5.5 Chapter Summary and Future Work

In this chapter, we presented the parallel SPARQL processing engine that was built using the Bobox and Bobolang framework with a focus on efficient parallel query processing. We performed multiple experiments to show the performance and the scalability of our solution. Additionally, we compare the engine with established frameworks for RDF data processing.

The results are very promising. Using SP²Bench queries, we identified that our solution is able to process many queries significantly faster than other engines and to obtain results on larger datasets. Therefore, our parallel approach to RDF data processing has a potential to provide better performance than current engines.

On the other hand, we also identified several issues with the query optimizer which sometimes returns non-optimal query plans. Therefore, we plan to improve data statistics, cost model and the analysis of filter conditions.

Finally, we plan to replace the in-memory storage by a persistent one in order to make the SPARQL engine more applicable.
6. Conclusion

Parallel data processing is an extensive research topic. In this thesis, we focused primarily on the domain of streaming systems and tried to develop a highly scalable parallel data processing application using those systems.

In order to reach that objective, we decided to continue in the work on the Bobox system which might be considered an implementation of a streaming system. We identified several disadvantages of that system, e.g., the system was not very user friendly, Bobox runtime did not support modern hardware like NUMA systems, and the framework had space for further optimizations. Therefore, we redesigned the API of the system in order to be much more applicable and attractive for the development of parallel applications. One of the most important contributions of this work is the new locality aware task scheduler which significantly increases the performance of the system on modern hardware, especially when used along with the NUMA-aware memory allocator.

After that, we introduced completely new domain specific language – Bobolang. This language makes the development of streaming applications much simpler since it enables intuitive specification of their execution plan. Moreover, it contains features which increase scalability of such applications. Especially the multiplication of inputs/outputs is quite useful since it allows automatic replication of operators with the respect to the hardware configuration of the host system. It also reflects two general approaches to the parallelization of operators (i.e., the parallelization of stateless and stateful operators) and contains syntactic features which make this process trivial for the developer.

The combination of Bobox and Bobolang allowed us easier development of parallel algorithms which are used intensively in database systems. Parallel sort and parallel merge join algorithms are the most important contributions in this part of the thesis. Despite the fact that the sort algorithm introduces significant overhead caused by the increased memory consumption and complex algorithm for parallel multi-way merging, it scales very well and it significantly outperforms algorithms parallelized with the TBB library on highly parallel systems. Parallel merge join scales well almost independently on the distribution of the data and also in cases when traditional approach to parallelization based on partitioning is not efficient.

Finally, we introduced an implementation of the in-memory SPARQL engine based on these algorithms. It demonstrates their behaviour as well as the behaviour of the Bobox/Bobolang framework in a real and rather complex data processing application. According to our experiments, application achieves significant speed up on SMP and NUMA systems and is able to significantly outperform other mainstream SPARQL engines by orders of magnitudes in some cases.

Despite our contributions, many challenges still remain. Bobox currently supports only shared-memory systems. We plan to port the Bobox to hardware platforms which fit this model, such as XeonPhi, which should be attainable without a major redesign of the system. On the other hand, porting Bobox to hybrid platforms, i.e., allowing communication between heterogeneous parts of that system will require further research. However, the first step was already done
thanks to the pilot implementation of the distributed Bobox [131]. Additionally, we plan to analyse the preferred size of envelopes much more thoroughly.

In case of Bobolang, we plan to focus on an algorithm which chooses the number of replicas more dynamically. In current implementation, the symbols * and ** are replaced by a number which depends only on the hardware configuration. However, it follows from experiments that the more replica is used, the more overhead the execution plan introduces. On the other hand, if the system is heavily loaded by a large number of parallel requests which are able to fully utilize the system, then the number of replicas can be decreased in order to reduce the overhead and increase the throughput of the system. Therefore, we want to modify the Bobolang interpreter so that it analyses the current load of the system and chooses the number of replicas more optimally.

Additionally, we want to introduce specialization of operators for certain level of parallelism. For example, for systems with lower number of processors, the merge tree can be a better choice than the complex parallel multi-way merge operator (see Section 4.4.2).

In the area of parallel algorithms, we want to focus on parallel join algorithm based on hashing. Hashing does not naturally fit to the streaming computation model; however, in cases when one relation is significantly larger than the other and the stream is not sorted according to the join attribute, then hash join algorithm might be a better choice than sort-merge join algorithm. We also want to optimize the implementation of the parallel sort algorithm so that it is faster in single-threaded evaluation while it keeps its scalability.

Finally, we want to replace our naive in-memory storage used in SPARQL engine by a persistent one. This would turn out our solution to powerful and full-fledged RDF storage capable to work efficiently with big data. Moreover, we want to expand our system to other areas besides the RDF storages and use our algorithms in other applications such as parallel SQL engine.
Bibliography


List of Figures

1.1 Example of a streaming application........................................... 6
2.1 Simplified schema of a NUMA system.......................................... 10
2.2 Bobox architecture............................................................. 11
2.3 Structure of an envelope...................................................... 12
2.4 Results for single query on SMP8 (logarithmic scale).................. 24
2.5 Results for multiple queries on SMP8 (logarithmic scale)............. 24
2.6 Results for single query on NUMA64 (logarithmic scale)............. 25
2.7 Results for multiple queries on NUMA64 (logarithmic scale)........ 25
2.8 Comparison of IO-unaware and IO-aware approach....................... 26
2.9 Comparison of the default allocator with the Bobox allocator on NUMA64 (logarithmic scale)......................................................... 28
2.10 Comparison of the default allocator with the Bobox allocator on SMP8 (logarithmic scale).......................................................... 29
2.11 Comparison of the default allocator with the Bobox allocator on Windows 7 (logarithmic scale)......................................................... 29
3.1 Execution plan of the graph $g$................................................... 37
3.2 Execution plan of the BandPassFilter operator............................. 38
3.3 Decomposition of an operator into a non-linear chain of sub-operators. 39
3.4 Parallelization schema for stateless operators................................ 40
3.5 Parallelization schema for parallelizable operators (RIDs are in brackets):................................................................. 42
3.6 Execution plan of Listing 3.4.................................................... 45
3.7 Execution plan of Listing 3.5.................................................... 46
3.8 Execution plan of Listing 3.6.................................................... 47
3.9 Results for different choices of the number of replicas on NUMA64, single query (logarithmic scale)......................................................... 48
3.10 Results for different choices of the number of replicas on NUMA64, multiple queries (logarithmic scale)..................................................... 48
4.1 Execution plan of the parallel filter operators.................................. 54
4.2 Execution plan of the ParallelNestedLoopJoin operator.................. 55
4.3 Query plan of the query E1 (with the nested loop join operator)........ 56
4.4 Results for query E1 (1m, nested loop join).................................. 56
4.5 Parallelized sort operator with the single-threaded multi-way merge operator................................................................. 59
4.6 Parallelized sort operator with the merge tree................................ 59
4.7 Execution plan of the ParallelSort operator.................................. 62
4.8 Sorting of 32b integers.......................................................... 63
4.9 Sorting in pipeline.............................................................. 64
4.10 Execution plan of the ParallelMergeJoinWithoutDuplicates operator................................................................. 65
4.11 Cross product across multiple envelopes during the processing of the second pair of envelopes......................................................... 67
4.12 Execution plan of the ParallelMergeJoin operator with duplicates..... 69
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13</td>
<td>Query plan of the query E2</td>
<td>69</td>
</tr>
<tr>
<td>4.14</td>
<td>Results for query E2</td>
<td>70</td>
</tr>
<tr>
<td>4.15</td>
<td>Number of articles per year in the 5m SP²Bench dataset</td>
<td>71</td>
</tr>
<tr>
<td>4.16</td>
<td>Results for query E3</td>
<td>71</td>
</tr>
<tr>
<td>4.17</td>
<td>Results for query E1 (1m, merge join)</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture of the query compiler and optimizer</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Speed up obtained by the inter-operator parallelization (logarithmic scale)</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>Speed up caused by the intra-operator parallelization on SMP8 (logarithmic scale)</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Speed up caused by the intra-operator parallelization on NUMA64 (logarithmic scale)</td>
<td>84</td>
</tr>
<tr>
<td>5.5</td>
<td>Results of SP²Bench benchmark for different SPARQL engines on SMP8 (logarithmic scale)</td>
<td>84</td>
</tr>
</tbody>
</table>


## List of Listings

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Implementation of the filter box in the initial version of the Bobox API</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Implementation of the pop_envelope function</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Implementation of the input_stream class</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Implementation of the output_stream class</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Implementation of the filter box in the enhanced version of the Bobox API</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of the construction of a flow-graph in TBB</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Source code of a band-pass filter in StreamIt</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Definition of a new operator in Bobolang</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Specification of a complete execution plan in Bobolang</td>
<td>45</td>
</tr>
<tr>
<td>3.5</td>
<td>Arity of operators in Bobolang</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Multiplicated inputs/outputs in Bobolang</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Schema for the parallelization of a stateless operator</td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Parallel filter operator in Bobolang</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallel nested loop join operator in Bobolang</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Parallel sort operator in Bobolang</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution plan for the sorting of integers</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution plan for the sorting in a pipeline</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>Source code in C++ equivalent to Listing 4.5</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>Parallel merge join operator without duplicates</td>
<td>66</td>
</tr>
<tr>
<td>4.8</td>
<td>Parallel merge join operator</td>
<td>69</td>
</tr>
</tbody>
</table>
## List of algorithms

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>General schema of stateful operators.</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Algorithm performed by the replicas of a parallelizable operator.</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm performed by the replicas of a parallelizable operator with the aggregated state updating.</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>Algorithm performed by the Preprocess operator.</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Algorithm performed by the replicas prepended by the Preprocess operator.</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Non-parallelized version of the Defragment operator.</td>
<td>43</td>
</tr>
<tr>
<td>3.7</td>
<td>Parallelized version of the Defragment operator.</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>Parallelized version of the Defragment operator with the aggregated state updating.</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Algorithm for getting the positions of the pointers after processing $L$ tuples.</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Algorithm of one replica of the parallelizable merge operator.</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Traditional merge join algorithm.</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>Algorithm of the Preprocess operator without duplicates.</td>
<td>67</td>
</tr>
<tr>
<td>4.5</td>
<td>Algorithm of the ParallelizableMergeJoin operator.</td>
<td>68</td>
</tr>
</tbody>
</table>
Appendix A

Queries of SP²Bench

This appendix contains the list of SPARQL queries of SP²Bench [120] benchmark which were used in this thesis. We focus mainly on the structure of the query plans, since it determines their behaviour during the evaluation and influences the results of experiments. The operators with red border are blocking, i.e., they emit the first output tuple after they receive last input tuple. Therefore, they break the pipeline processing.

Notice that we did not use queries Q12a, Q12b, and Q12c intentionally. Queries Q12a and Q12b are equivalent to queries Q5a and Q8 respectively, the only difference is that they use ASK query form instead of SELECT query form [112] on which we did not focus in this thesis. Query Q12c is intended to test the abilities of the underlying RDF storage by asking for non-existent triple. This query is basically equivalent to the query Q10 of which results are negligible.

Query Q1

```sparql
SELECT ?yr
WHERE {
  ?journal dcterms:issued ?yr
}
```

This query returns the year when the journal with title ”Journal 1 (1940)” were issued. Therefore, the amount of data which it processes is very low and the evaluation is very fast.

Figure A.1: Query plan of query Q1.
Query Q2

```
WHERE {
  ?inproc rdfs:seeAlso ?ee .
  ?inproc dcterms:issued ?yr
OPTIONAL {
  ?inproc bench:abstract ?abstract
}
}
ORDER BY ?yr
```

This query contains linear chain of merge joins without any blocking operator. Therefore, the evaluation of the plan benefits heavily from inter-operator parallelism thanks to their pipeline processing.

![Query plan of query Q2.](query_plan.png)

Figure A.2: Query plan of query Q2.
Query Q3a

```sql
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:pages)
}
```

This query and following queries Q3b and Q3c are very simple ones. They do not process large amount of data and are not computational intensive as well as query Q1; therefore, their evaluation is very fast.

![Query plan of query Q3a.](image)

Figure A.3: Query plan of query Q3a.

Query Q3b

```sql
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:month)
}
```

![Query plan of query Q3b.](image)

Figure A.4: Query plan of query Q3b.
Query Q3c

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:isbn)
}
```

Figure A.5: Query plan of query Q3c.

Query Q4

```
SELECT DISTINCT ?name1 ?name2
WHERE {
  ?article1 rdf:type bench:Article .
  ?article2 rdf:type bench:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
  FILTER (?name1<?name2)
}
```

This query contains linear chain of merge joins as well as query Q2. However, this chain is interleaved with blocking operators such as sort operators and distinct operators. Therefore, its does not benefit from pipeline processing as much as query Q2. Notice that the number of results is very large and basically quadratic in the number and size of journals, therefore, it is very memory consuming.
Figure A.6: Query plan of query Q4.
Query Q5a

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

This query contains similar chain of sort-merge joins as query Q4, however, the number of results is linear to the size of database. This query is neither computation intensive nor memory consuming, therefore, its scalability perfectly reflects the scalability of our sort-merge join algorithm under normal circumstances.

![Query Plan of Q5a](image)

Figure A.7: Query plan of query Q5a.
Query Q5b

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

This query is semantically and structurally same as the query Q5a, the only difference is that the chain of sort-merge joins is slightly shorter.

![Query plan of query Q5b.](Figure A.8: Query plan of query Q5b.)
Query Q6

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

The most time consuming part of this query is the last nested loop join operator which basically performs cross product of all authors in the database. This operator overshadows all other operators, therefore, the scalability of this query reflects the scalability of our nested loop join algorithm.

Query Q7

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document .
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document .
      ?bag4 ?member4 ?doc3
    }
    FILTER (!bound(?doc4))
  }
  FILTER (!bound(?doc3))
}
```

The query plan of Q7 is very bushy, therefore, it benefits from inter-operator parallelism significantly.
Figure A.9: Query plan of query Q6.
Figure A.10: Query plan of query Q7.
Query Q8

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string .
  {
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
         ?author2!=?erdoes &&
         ?author2!=?author)
  } UNION {
    ?author foaf:name ?name
    FILTER (?author!=?erdoes)
  }
}
```

Query Q8 has also very bushy query plan which enables inter-operator parallelism. However, this plan is far from optimal since the query optimizer does not decompose the filter expression in order to decrease the size of intermediate results.

Query Q9

```
SELECT DISTINCT ?predicate
WHERE {
  {
    ?subject ?predicate ?person
  } UNION {
    ?person ?predicate ?object
  }
}
```

Dominant operator of this operator is the last distinct operator. Its input is very large while the output very small (only 4 tuples). In this case, the distinct operator based on hashing would be more appropriate.
Figure A.11: Query plan of query Q8.
Figure A.12: Query plan of query Q9.

**Query Q10**

```
SELECT ?subject ?predicate
WHERE {
  ?subject ?predicate <http://localhost/persons/Paul_Erdoes>
}
```

This query is intended to test the abilities of RDF storages and their indexes. Since we used all possible indexes in our in-memory storage, this query is evaluated almost instantly.

Figure A.13: Query plan of query Q10.
Query Q11

```
SELECT ?ee
WHERE {
  ?publication rdfs:seeAlso ?ee
}
ORDER BY ?ee
LIMIT 10
OFFSET 50
```

This query is also very simple one and is intended to test the slice operator primarily.

![Query plan of query Q11.](image)

Figure A.14: Query plan of query Q11.
Appendix B

Example of the Bobox/Bobolang Application

In this appendix, we include the complete listing of an application in C++ which uses Bobox and Bobolang. This application corresponds to the sample application from Chapter 1, i.e., its execution plan is same as in Figure B.1. The source box generates stream of even/odd integers, the merge box merges both input streams, the filter box selects only prime numbers and the print box prints the input stream to the standard output.

The source code is intended to clarify simplified source codes from Section 2.3 as well as the interaction between Bobox and the Bobolang language, i.e., passing parameters to boxes, registering of boxes, naming their inputs/outputs etc. Additional details might be found in the documentation on the attached DVD.

![Figure B.1: The execution plan of the application.](image)

```cpp
#include <iostream>
#include "bobox_manager.hpp"
#include "bobox_results.hpp"
#include "bobox_bobolang.hpp"
#include "bobox_runtime.hpp"
#include "bobox_request.hpp"
#include "bobox_results.hpp"
#include "bobox_basic_object_factory.hpp"
#include "bobox_basic_box.hpp"
#include "bobox_basic_box_model.hpp"
#include "bobox_basic_box_utils.hpp"

using namespace bobox;

// The source_box class generates the sequence of odd/even integers according to the boolean parameter "odd",
// the length of the sequence is given in the integer parameter "length".
// The implementation uses the output_stream class, which logically transforms the stream of envelopes into the
// stream of tuples.
class source_box : public basic_box {
    public:
```
// The model class is responsible for the instantiation of the box.
typedef generic_model<source_box> model;

// These macros enable to give names to inputs/outputs which might be used in Bobolang.
BOBOX_BOX_INPUTS_LIST(main, 0);
BOBOX_BOX_OUTPUTS_LIST(main, 0);

// The constructor receives the parameters passed to the operator during its instantiation in Bobolang.
source_box(const box::parameters_pack &box_params,
            const parameters_ptr_type &params)
  : basic_box(box_params)
  {
    params->get_parameter("odd", odd_);
    params->get_parameter("length", length_);
  }

// The body function
virtual void sync_body() override
  {
    output_stream<std::tuple<int>> stream(this, outputs::main());
    for (int i = 0; i < length_; i++) {
      stream.next() = i * 2 + (odd_ ? 1 : 0);
    }
  }

private:
  bool odd_;  // 
  int length_;  //
};

// The merge_box class merges two sorted streams into one.
// The implementation uses input_stream and output_stream classes, which makes the implementation of this class straightforward.
class merge_box : public basic_box {
public:
  typedef generic_model<merge_box> model;

  BOBOX_BOX_INPUTS_LIST(left, 0, right, 1);
  BOBOX_BOX_OUTPUTS_LIST(main, 0);

  merge_box(const box::parameters_pack &box_params)
    : basic_box(box_params)
  {
  }

  // The merge_box class merges two sorted streams into one.
  // The implementation uses input_stream and output_stream classes, which makes the implementation of this class straightforward.
  virtual void sync_body() override
    {
      output_stream<std::tuple<int>> stream(this, outputs::main());
      for (int i = 0; i < length_; i++) {
        stream.next() = i * 2 + (odd_ ? 1 : 0);
      }
    }

private:
  bool odd_;  // 
  int length_;  //
};
virtual void sync_body() override
{
  typedef std::tuple<int> io_type;

  input_stream<input_stream<io_type> left(this, inputs::left());
  input_stream<input_stream<io_type> right(this, inputs::right());
  output_stream<output_stream<io_type> output(this, outputs::main());

  while (!left.eof() && !right.eof()) {
    int l = left.current()->get<0>();
    int r = right.current()->get<0>();

    if (l <= r) {
      output.next()->get<0>() = l;
      left.move_next();
    } else {
      output.next()->get<0>() = r;
      right.move_next();
    }
  }

  while (!left.eof()) {
    output.next()->get<0>() = left.current()->get<0>();
    left.move_next();
  }

  while (!right.eof()) {
    output.next()->get<0>() = right.current()->get<0>();
    right.move_next();
  }
};

// The filter_box class filters out all composite numbers.
// The implementation corresponds to the initial version
// of the Bobox API.
class filter_box : public basic_box {
public:
  typedef generic_model<filter_box> model;

  filter_box(const box_parameters_pack &box_params)
    : basic_box(box_params)
  {}

  BOBOX_BOX_INPUTS_LIST(main, 0);
  BOBOX_BOX_OUTPUTS_LIST(main, 0);
static bool is_prime(int n) {
    if (n < 2 || (n > 2 && n % 2 == 0)) {
        return false;
    }
    for (int i = 3; i*i <= n; i += 2) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

virtual void init_impl() override {
    envelope_to_send_ = allocate(outputs::main());
}

virtual void sync_body() override {
    column_index_type column(0);

    auto envelope = pop_envelope(inputs::main());
    if (envelope->is_poisoned()) {
        if (!envelope_to_send_->is_empty()) {
            send_envelope(outputs::main(),
                          std::move(envelope_to_send_));
        }
    }
    send_poisoned(outputs::main());
    else {
        for (int i = 0; i < envelope->get_size(); i++) {
            int number = envelope->get_data<int>(column)[i];
            if (is_prime(number)) {
                size_t next = envelope_to_send_->get_size();
                envelope_to_send_->set_size(next + 1);
                envelope_to_send_->get_data<int>(column)[next] = number;
                if (envelope_to_send_->is_full()) {
                    send_envelope(outputs::main(),
                                  std::move(envelope_to_send_));
                    envelope_to_send_ = allocate(outputs::main());
                }
            }
        }
    }
}

private:
    mutable envelope_ptr_type envelope_to_send_;
// The print_box prints the input stream to the standard output.
// The implementation uses the pop_envelope function.

class print_box : public basic_box {
public:
    typedef generic_model<print_box> model;

    print_box(const box_parameters_pack &box_params)
        : basic_box(box_params)
    {}

    BOBOX_BOX_INPUTS_LIST(main, 0);
    BOBOX_BOX_OUTPUTS_LIST(main, 0);

    virtual void sync_body() override
    {
        column_index_type column(0);

        for (auto env = pop_envelope(inputs::main());
            env->is_data();
            env = pop_envelope(inputs::main())) {
            for (size_t i = 0; i < env->get_size(); i++) {
                std::cout << env->get_data<int>(column)[i] << std::endl;
            }
        }
        send_poisoned(outputs::main());
    }
};

// The runtime class is responsible for the registration of the boxes and datatypes so that they can be used from Bobolang.
// Additionally, it may contain global data for applications; however, we do not use it in this example.
class application_runtime : public runtime,
    public basic_object_factory {
public:
    virtual void init_impl() override
    {
        register_box<source_box::model>(box_model_tid_type("Source"));
        register_box<print_box::model>(box_model_tid_type("Print"));
        register_box<merge_box::model>(box_model_tid_type("Merge"));
        register_box<filter_box::model>(box_model_tid_type("Filter"));

        register_type<int>(type_tid_type("int"));
    }

    virtual runtime *get_runtime() override
    {
        return this;
    }
};
```cpp
int main(int argc, char *argv[]) {
    manager mng;

    application_runtime rt;
    rt.init();

    // Source code of the execution plan in Bobolang.
    std::stringstream bobolang(R"(
        operator main() -> () {
            bobox::broadcast() -> ()[to_odd], ()[to_even] broadcast;
            Source() -> (int) odd(odd=true, length=100);
            Source() -> (int) even(odd=false, length=100);
            Merge(int) -> (int) merge;
            Filter(int) -> (int) filter;
            Print(int) -> () print;

            input -> broadcast;
            broadcast[to_odd] -> odd;
            broadcast[to_even] -> even;
            odd -> [left] merge;
            even -> [right] merge;
            merge -> filter -> print -> output;
        }
    )");

    auto rqid = mng.create_request(bobolang::compile(bobolang, &rt));
    mng.run_request(rqid);
    mng.wait_on_request(rqid);
    mng.destroy_request(rqid);

    return 0;
}
```
Appendix C

Grammar of Bobolang

Program ::= Declaration*
Declaration ::= 'operator' OpIdentifier DeclIOs '->' DeclIOs '{' Body '}'
OpIdentifier ::= ( Namespace ':' ':' )* Identifier
Namespace ::= Identifier
Identifier ::= [a-zA-Z] [a-zA-Z0-9._]*
DeclIOs ::= DeclIO ( ',' DeclIO )*
DeclIO ::= '(' DeclTypes ')' IOName?
DeclTypes ::= ( ( Type ',' Type )* Tytypename ( ',' Type )* ) | ( Type ( ',' Type )* )?
Type ::= Identifier
Tytypename ::= 'typename' Identifier
IOName ::= '[' Identifier ']
Body ::= ( Command ';' )* Command ::= Instantiation | Connection | Typedef
Instantiation ::= InstModifier? OpIdentifier InstIOs '->' InstIOs Instances
InstModifier ::= 'stateless' | 'parallel'
InstIOs ::= InstIO ( ',' InstIO )* InstIO ::= '(' InstTypes ')' IOName? IOMultiplier?
InstTypes ::= ( Type ( ',' Type )* )?
IOMultiplier ::= '*' | '**' | Positive
Positive ::= [1-9] [0-9]*
Instances ::= Instance ( ',' Instance )* Instance ::= InstIdentifier Parameters?
InstIdentifier ::= Identifier
Parameters ::= '(' ( Parameter ( ',' Parameter )* )? ')' Parameter ::= Identifier '=' ( String | Integer | Double | Bool )
String ::= '"' EscChar* '"'
EscChar ::= ['"\b\n\r\0' | '\]' ['tbnrf0]
Integer ::= Sign? Unsigned
Sign ::= '+' | '-' Unsigned ::= [0-9]+
Bool ::= 'true' | 'false'
Connection ::= SrcSpec ( '->' DestSrcSec )* '->' DestSpec
SrcSpec ::= InstIdentifier IOSpec?
IOSpec ::= IOName? | ( '[' Unsigned ']' )
DestSrcSec ::= IOSpec? InstIdentifier IOSpec?
DestSpec ::= IOSpec? InstIdentifier
Typedef ::= 'typedef' '(' ( Type ( ',' Type )* )? ')' Type
Content of the Attached DVD

The attached DVD contains the following directories:

- **src** – Contains the source code of Bobox/Bobolang, the source code of the runtime parallel of the SPARQL engine and the source code of the application from Appendix B.

- **benchmark** – Contains SP²Bench queries compiled to the Bobolang language and SP²Bench datasets.

- **doc** – Contains the oxygen documentation of the Bobox system.

- **thesis** – Contains the PDF version of this thesis.