Noname manuscript No.
(will be inserted by the editor)

Data Stream Processing via Code Annotations

Marco Danelutto · Tiziano De Matteis · Gabriele Mencagli · Massimo Torquati

Received: date / Accepted: date

Abstract Time-to-solution is an important metric when parallelizing existing code. The REPARA approach provides a systematic way to instantiate stream and data parallel patterns by annotating the sequential source code with `C++11` attributes. Annotations are automatically transformed in a target parallel code that uses existing libraries for parallel programming (e.g., FastFlow). In this paper we apply this approach for the parallelization of a data stream processing application. The description shows the effectiveness of the approach in easily and quickly prototyping several parallel variants of the sequential code by obtaining good overall performance in terms of both throughput and latency.

Keywords Code Annotations · Parallel Patterns · Data Stream Processing.

1 Introduction

Data Stream Processing [2] (briefly, DaSP) is a paradigm that enables the computation on *unbounded* data streams coming from various sources (e.g., sensors, social media and network devices). Streaming data are not stored persistently in memory and then computed; rather they are processed "on-the-fly" by *continuous queries* whose *operators* run constantly over time in order to produce outputs in a continuous way. The goal is to extract greater knowledge from the data by maintaining a feasible history of the stream, and to detect recurrent patterns by generating complex events in real-time.

The parallelization of *stateful operators*, in which the stream history is maintained in succinct data structures (e.g., synopses, sketches and wavlets) or, more often, in *windows* containing the most recent data [9], has received

M. Danelutto, T. De Matteis, G. Mencagli and M. Torquati
Department of Computer Science, University of Pisa, Italy
Phone: +39-050-2213132
E-mail: {marcod, dematteis, mencagli, torquati}@di.unipi.it

increasing attention by the research community. Common parallelizations [16, 7] consist in having replicas of the same operator that receive inputs belonging to the same sub-stream identified by a partitioning data attribute. Examples of such operators are sorting, aggregation and one-way joins. Available parallel implementations are usually hand-coded using standard streaming frameworks like IBM InfoSphere [17], Spark Streaming [27] and Storm [28]. These frameworks do not offer to the programmer adequate high-level abstractions for parallel programming (e.g., bolts and spouts of Storm are too low-level to express high-level parallel patterns), and they require to re-write the application from scratch without the option to re-use the available sequential code.

Rewriting applications from scratch using parallel programming frameworks and libraries (e.g., Storm [28] TBB [18], FastFlow [15], TPL [21]) is typically too costly in terms of time-to-development. For this reason, a great effort has been made by the research community in developing high-level programming interfaces to introduce parallelism in existing sequential (legacy) codes. Examples are the pragma-based approach (OpenMP [8] and OpenACC [14] models), language extensions like Cilk [6] and data-flow models [20]. Unfortunately, these frameworks are not particularly suitable to deal with data streams and stateful operators.

Recently, the REPARA project [24] proposed a programmer friendly high-level parallel programming interface based on `C++11` program annotations (i.e *attributes* [19]). `C++11` generalized attributes allow to attach arbitrary annotations to regions of a program by integrating such annotations directly into the abstract syntax tree of the program itself. The advantages of this approach are that `C++11` attributes are portable, fully integrated in the same language used to define the sequential code, and most of all, they allow to introduce *parallel patterns* without explicit program refactoring thus preserving the entire original sequential code.

In the context of the REPARA project, `C++11` attributes are used to define parallel regions (called "kernels") in an application and their parallel behavior through parallel patterns. Those patterns have a well-know parallel semantics and relieve the programmer from the burden of dealing with the traditional low-level parallel programming issues such as thread synchronization and mapping, and job scheduling. By using the REPARA attributes, the programmer can express both stream computations (by using *pipeline* and *task-farm* attributes) and data parallelism (by using *map* and *reduce* attributes) [11].

The programmer using the REPARA model is involved in two phases: *i)* reshaping the sequential code in order to be compliant to the standard REPARA `C++` [25]; *ii)* identifying the regions of the code that need to be accelerated by using proper attributes to express the parallel patterns. At the time of writing, this second phase still requires a direct programmer intervention for introducing code annotations. In the future, the model will be enriched in order to automatically generate annotations with negligible user assistance.

The goal of this paper is to investigate how the REPARA methodology can be used to parallelize DaSP applications on multicores. To this end, we consider

a high-frequency trading (HFT) application that represents a paradigmatic example of the DaSP application domain.

Preliminary results have been reported in [10]. In this previous work a preliminary evaluation of the REPARA approach for the same HFT application has been proposed. Different parallelizations have been defined by using proper REPARA attributes. In the present paper, by building upon the attribute extension proposed in the previous work, we propose a new and more sophisticated parallel implementation of the application that is capable to offer good performance as well as good resilience to very dynamic arrival rates. The new parallelization provides a good tradeoff between the positive aspects of the other two parallelizations proposed in the previous work [10].

The rest of the paper is organized as follows. Sect. 2 describes the related work. Sect. 3 provides a brief introduction to the REPARA approach. Sect. 4 describes the high-frequency trading application. Sects. 5 and 6 present the parallel variants and the experiments on a commodity multi-core machine.

2 Related Works

The most challenging data stream processing computations are those characterized by stateful operators that maintain complex data structures to produce results [2,4]. Usually, in this domain the applications are written from scratch by using the programming API of stream processing frameworks such as IBM InfoSphere [17], Spark Streaming [27] and Apache Storm [28].

In this paper, we propose a different approach, more commonly used in the high-performance computing domain for parallelizing sequential applications on multi-core platforms: introducing parallelism in the sequential code by means of program annotations. Annotations are introduced by using the standard annotation mechanism of `C++11` attributes [19]. In the REPARA framework [24], `C++11` attributes can be used to parallelize portion of code called "kernels" by means of well-known parallel patterns [22,1].

Compared to pragma-based approaches (OpenMP [8] and OpenACC [14]), REPARA attributes provide additional flexibility as they can be directly attached to syntactic program elements. Moreover, high-level data streaming parallel patterns, as pipeline with unbounded input streams and task-farm with non-trivial task scheduling policies, are not suitable to be easily expressed through OpenMP and OpenACC pragma directives.

Other approaches are those based on explicit parallel programming languages such as StreamIt [29]. StreamIt provides primitive constructs for pipeline, split-join and feedback loop which are used to write the entire application graph through combination and nesting of these constructs. However, as for other streaming frameworks, it requires to rewrite the entire application from scratch by using the constructs provided by the language.

3 The REPARA Parallelization Methodology

The REPARA project [24] is aimed at providing a methodology to parallelize a program starting from a sequential code (in some cases reshaped if necessary) by using a set of C++ attributes to define parallel regions of code named *kernels*. The methodology (sketched in Fig. 1) is structured into steps, each one taking into account different aspects of the code parallelization according to a clear *separation of concerns* design principle. A *code annotation* phase, performed directly by the programmer, identifies the "kernels" subject to parallelization. Then a *source-to-source* transformation phase deals with the refactoring of the identified parallel kernels into suitable run-time calls. Eventually, a target-specific *compilation phase* generates the actual executable code.

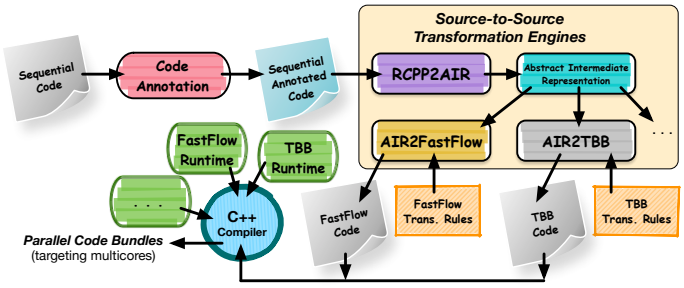


Fig. 1: Overview of the REPARA parallelization methodology.

The parallelization process of the REPARA methodology consists of three main phases. The first phase starts with a sequential program in which the user detects those parts of the code that can be annotated using the `rpr::kernel` attribute. Kernels are portions of sequential code subject to parallelization. In the second phase the annotated code is passed to the *Source-to-Source Transformation Engine*[1] whose internal workflow is sketched in Fig. 1. From the annotated source code, an *Abstract Intermediate Representation* (AIR) is generated [25,26]. Then, the engine uses the AIR and a set of rules, specific for each parallel programming model, for determining whether the corresponding code can be transformed into a *Parallel Programming Model Specific Code* (PPMSC)[2]. The PPMSC is the parallel generated code that is functionally equivalent to the original sequential code extended with parallel kernels execution accordingly to the attribute parameters and to the selected programming model (e.g., Intel TBB and FastFlow).The third phase includes the target compilation phase using a standard C++ compiler and all low-level dependencies needed to run the code. The runtime used provides coordination and all the mechanisms needed to support the deployment, scheduling and synchronization of kernels on the target platform(s).

[1] At the time of writing, this phase is hand-made and not fully automatized.
[2] REPARA imposes restrictions on the source code when targeting specific hardware.

4 A High-Frequency Trading Application

In this paper we apply the REPARA methodology to an application from the High Frequency Trading domain (HFT). HFT applications are characterized by stringent high-throughput and low-latency constraints. The goal is to discover fresh trading opportunities before the competitors by analyzing market feeds in near real-time. Many HFT applications can be described according to the so-called *split-compute-join* computational scheme [3] shown in Fig. 2.

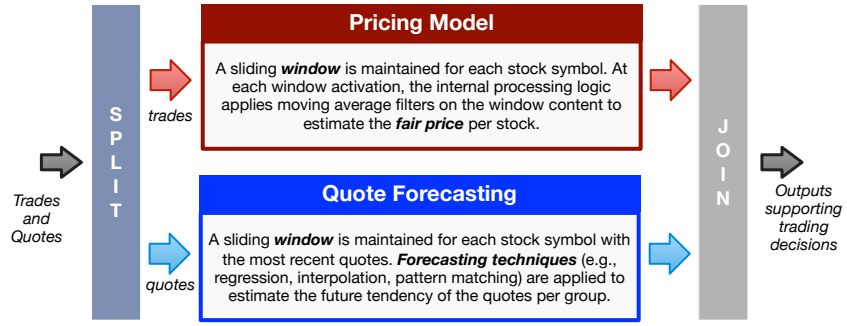


Fig. 2: The split-compute-join application scheme. Forecasting from existing quote data and correlation with the fair price for each stock symbol.

The application is fed by an unbounded stream of elementary elements from the market (financial ticks) having two types: *trades* represent closed transactions characterized by a price, a stock symbol and number of stocks (volume); *quotes* are buy or sell proposals (*bid* and *ask*) featuring a proposed price, a stock symbol and a volume.

Trades, grouped by stock symbol, are processed by the *Pricing Model* in order to estimate their fair price using the most recent trades (see [3] for further details). Usually, the model uses moving averages [3] (e.g., Volume-Weighted Average Price) by maintaining a *window* of the last trades per symbol.

Windows are the predominant abstraction used in data stream processing to deal with stateful operators that need to store the entire input to produce the outputs [2]. Due to the unbounded nature of the streams, the computation is applied on the most recent tuples buffered in a temporary window buffer. The window semantics is expressed by two parameters: *i)* the *window size* $|\mathcal{W}|$, in time units for *time-based windows* or in number of tuples for *count-based windows*; *ii)* the *sliding factor* δ (in tuples or time units), which expresses how frequently the window moves and its content gets processed by the operator internal algorithm. Commonly $\delta < |\mathcal{W}|$, and the model is called *sliding window*.

The *Quote Forecasting* phase processes bid and ask proposals grouped by the stock symbol. It represents the most compute-intensive part of the application for two main reasons: *i)* quotes are around ten times more common than trades [3]; *ii)* the prediction models (e.g., neural networks, regressions) are aimed at estimating the future volume and prices of the quotes based on historical data, and their execution is more compute-intensive than the moving

averages computed by the pricing model. Also in this case we need to maintain a sliding window of the most recent quotes per group. Quote forecasts and current fair prices are finally correlated (*join*) based on the stock symbol and the results processed by further decision-making phases, e.g., to update the quotes owned by the user (a market trader) by changing their volume and price attributes based on the results of the computation. The results of the quote forecasting are often visualized off-line by the human users to have a graphical feedback.

4.1 Starting Code of the Application

In this paper we focus on the Quote Forecasting processing chain. Our goal is to apply the REPARA methodology to speedup the execution of this phase with a reduced developing effort by the programmer, which is only involved in adding proper REPARA attributes to a sequential code (shown in Listing 1) by respecting the REPARA C++ directives [25].

Listing 1: REPARA C++ code of the Quote Forecasting processing.

```
1  int main(){
2   long w_size, w_slide;
3   quote_t quoteIn, q_Filtered;
4   HReturn_type result;
5   WinTask w_task;
6   Socket_Interface socket(PORT);
7   std::map<int, CBWindow*> map;
8   ...
9   while(socket.receive(quoteIn, sizeof(quote_t))){
10   filterQuotes(quoteIn, q_Filtered);
11   winManager(map, w_size, w_slide, q_Filtered, w_task);
12   computeWindow(w_task, result);
13   sendAndWrite(result);
14  }
15 }
16 void winManager(std::map<int, CBWindow*>& map, unsigned int w_size,
       unsigned int w_slide, const quote_t& quote, WinTask& w_task){
17  CBWindow* win=map[quote.stock_symbol];
18  if(win==nullptr){win=new CBWindow(w_size, w_slide);
19                   map[quote.stock_symbol]=win;}
20  bool isTriggered=win->insert(quote);
21  if(isTriggered) w_task.set(win, SUCCESS);
22  else w_task.set(nullptr, EMPTY);
23 }
24 void computeWindow(const WinTask& w_task, HReturn_type& res){
25  if(w_task.status==EMPTY) res.status=EMPTY;
26  else w_task.fitting_model(res);
27 }
```

The program consists of a while loop (lines 9-14) in which at each iteration we process a quote received from an input socket (interfaced using the object `socket`). Each quote is filtered by a function `filterQuotes`, which deletes unused fields and removes outliers. This function is very fine grained, and its definition is omitted for brevity. The core part of the code are the `winManager` and `computeWindow` functions. The first one receives a filtered quote and gets

the corresponding window from a hash table based on the stock symbol attribute. If the window does not exist it is created (lines 18, 19). The `insert` operation (line 20) on a window data structure adds a new quote to the window (by copying it) and removes the oldest quote (expiration).

Then, a `WinTask` object is prepared. It contains a reference to the quotes in the actual window and a *status* flag that can be *SUCCESS* or *EMPTY*. The first identifies a non-empty task and is generated if and only if the reception of the new quote triggers a window activation. Otherwise, if the window has not been triggered, the function returns an empty `WinTask`. The `computeWindow` function receives a `WinTask` and, based on the status value, executes the forecasting model if it is SUCCESS, or produces an empty result otherwise. In this application the forecasting model (`fitting_model`, line 26) is the Levenberg-Marquardt regression algorithm implemented by the `lmfit` library[3]. Finally, the result is stored in a local database and sent to the next stage of the application by the function `sendAndWrite`.

It is worth pointing out that the code described in Listing 1 is not the original one, but rather a reshaped version that adheres to the REPARA C++ constraints [25,23]. To be annotated by using the current version of the source-to-source transformation tools, the body of the while loop must contain only plain function call statements but not conditional or jump statements (`if`, `switch`, `break`). These limitations will be relaxed in future releases. Each iteration of the while loop corresponds to the execution of a quote, and must consists in a sequence of statements that can be eventually parallelized as it will be shown in Sect. 5. The function `filterQuotes` is always executed for each quote, while the `winManager` and `computeWindow` functions need to discriminate if a window has been triggered or not. This condition must be tested inside the body of the two functions. As a consequence, the output of the `winManager` function can be a valid or an empty task. In the last case the task is discarded by `computeWindow`, which in turn produces an empty result to the last function.

5 Prototyping Parallel Variants

In this section we will show how the REPARA methodology can be used for the fast prototyping of different parallel implementations.

The basic pattern that we use is the *pipeline* one. It combines multiple *stages* executed in a strict sequential order on the stream elements. The `rpr::pipeline` attribute identifies a loop whose body is composed of several functions (stages). Stages are annotated with the `rpr::kernel` attribute and the variables corresponding to the input and output elements at each iteration are identified by the `rpr::stream` attribute. The REPARA pipeline uses the loop termination condition to determine the end of the stream.

While the pipeline pattern allows different input elements to be processed in parallel on different stages, the attribute `rpr::farm` is used to replicate the

[3] `http://apps.jcns.fz-juelich.de/lmfit`

same kernel. Each *worker* thread of the runtime executes a replica of the kernel on a subset of the input elements. An *emitter* thread schedules input elements to the workers using standard policies like the *round-robin* or the *on-demand* ones, enabled through specific options in the farm attribute. The parallel execution of the same kernel may introduce ordering problems between results because of their relative running times. The *ordered* or *unordered* options can be added in the farm attribute to specify whether the results must be produced to the next stage with the exact same order of the corresponding input elements, or if any order can be tolerated. The ordered option can be used provided that the farm produces one output for each input element.

In our previous work [10], we have shown that the pipeline implementation is not able to achieve good speedup for this application, because the computational cost of the `computeWindow` function is about 95% of the overall execution time of a single iteration of the while loop. Therefore, combinations of pipeline and task-farm patterns are needed to improve the speedup.

5.1 Pipeline and Farm

Listing 2 shows the version in which the `computeWindow` kernel is executed in parallel by a set of workers. The number of workers is statically expressed by the integer variable `nw` whose value is chosen by the programmer (e.g., it is passed as a command line argument). With this parallelization, windows with the same stock symbol and with different symbols can be executed in parallel by distinct workers. Each window, once triggered and copied in the `winManager` kernel, is distributed to a worker according to a round-robin distribution policy. As typical of the farm pattern, if the computation time per task exhibits a high variance the results can be received to the next stage in a different order than the corresponding inputs. To avoid this, we use the option *ordered* of the farm attribute to enforce a total ordering between input and output elements. The attribute `rpr::target(CPU)` is used to specify the device on which the kernel is executed (alternatives are GPU and FPGA).

Listing 2: Using *rpr::pipeline* and *rpr::farm* attributes.

```
1  [[rpr::pipeline,rpr::stream(w_task,result)]]
2   while(socket.receive(&quoteIn,sizeof(quote_t))){
3    [[rpr::kernel,rpr::out(w_task),rpr::target(CPU)]] {
4     filterQuotes(quoteIn,q_Filtered);
5     winManager(map,w_size,w_slide,q_Filtered,w_task);
6    }
7    [[rpr::kernel,rpr::farm(nw,ordered),
8      rpr::in(w_task),rpr::out(result),rpr::target(CPU)]]
9     computeWindow(w_task,result);
10   [[rpr::kernel,rpr::in(result),rpr::target(CPU)]]
11    sendAndWrite(result);
12 }
```

5.2 Providing a Customized Distribution Policy

A different parallelization can be designed by annotating with the `rpr::farm` attribute a block containing both the `winManager` and the `computeWindow` function calls. In this way, the input elements of the second stage are single quotes filtered by the first stage, and not entire windows as in the previous parallelization. Quotes must be properly transmitted to the workers according to a user-defined distribution policy in such a way that the parallel computation performs the very same set of windows for all the stock symbols of the sequential program, i.e. without altering the computation semantics.

For this reason, we propose to extend the `rpr::farm` attribute by passing as an extra optional parameter the name of a user-defined function that will be used by the runtime to determine to which workers the input elements have to be sent. The proposed extension can be formalized as shown in Listings 3.

Listing 3: Expressing a customized distribution function.

```
1 std::vector<size_t> routing(const size_t N,const input1_t& input1
      ,...,const inputK_t& inputK) {...}
2 [[rpr::kernel,rpr::farm(nw,routing,...),rpr::in(input1,...,inputK)
      ,...]]
3  kernel-region
```

The function `routing` gets in input the number of active workers $N \geq 1$, and the $K \geq 1$ input elements specified with the `rpr::in` attribute of the farm. It must return a non-empty vector of integers each one in the range $[0, N)$. The runtime will distribute a copy of the input arguments to each worker whose identifier is in the vector returned by the function. Based on the definition of the routing function we propose two possible parallel implementations.

5.2.1 Pipeline and farm with hashing

A first parallel version can be designed by using a hash function for the distribution of the quotes. The idea is to distribute all the quotes with the same stock symbol to the same worker. Therefore, each worker maintains a subset of all the windows. Listing 4 shows the annotated code of this parallel version. The distribution can be performed by any hash function that returns one worker identifier for each quote. Without any further information about the frequency distribution of the stock symbols, the hash function (`SchedByKey`) needs to map roughly the same number of symbols to each identifier, e.g., $q.s \bmod N$, where s is the unique number that identifies the stock symbol of the quote q and N is the number of worker threads.

This implementation has positive effects and also some shortcomings. A positive aspect is its simplicity: input quotes are not buffered and then the whole window bulk of data transmitted to the farm as in the previous implementation. Instead, each single quote is routed "on-the-fly" to a worker once it has been received from the input socket. Furthermore, all the windows of

Listing 4: Using *rpr::pipeline* and *rpr::farm* with *hashing*.

```
1  [[rpr::pipeline,rpr::stream(q_Filtered,result)]]
2   while(socket.receive(&quoteIn,sizeof(quote_t))){
3   [[rpr::kernel,rpr::out(q_Filtered),rpr::target(CPU)]]
4    filterQuotes(quoteIn,q_Filtered);
5   [[rpr::kernel,rpr::farm(nw,unordered,SchedByKey),
6     rpr::in(q_Filtered),rpr::out(result),rpr::target(CPU)]] {
7    winManager(map,w_size,w_slide,q_Filtered,w_task);
8    computeWindow(w_task,result);
9   }
10  [[rpr::kernel,rpr::in(result),rpr::target(CPU)]]
11   sendAndWrite(result);
12 }
```

the same stock symbol are processed by the same worker sequentially, thus the results arrive to the next stage already ordered within the same group. This partial ordering is often sufficient for the correct real-time analysis of outputs by the successive processing phases of the application. Therefore, the attribute `ordered` of the REPARA farm (which implies a total ordering of inputs with respect to outputs) is not needed in this implementation.

However, this version may suffer from possible load unbalancing. When a small subset of the stock symbols have much higher probability than the others, a subset of the workers might receive more quotes in the same time period thus hampering the speedup. If the frequency distribution of the stock symbols is statically fixed and known, this issue can be mitigated by using proper hash functions. However, with unknown or time-varying frequency distributions the problem cannot be solved statically, and an autonomic support for state migration between workers is needed [13].

5.2.2 Pipeline and on-the-fly farm

A new parallelization can be designed by trading-off the positive aspects of the two previous solutions. The idea is the following: all the quotes within the same group (i.e. with the same stock symbol) are assigned to a unique identifier starting from 1. Identifiers are also assigned to the windows within the same group. The new distribution function (`SchedByMulticast`) (inspired by the work in [5]) executed on an input quote q consists in the following steps:

- all the consecutive windows that contain the quote q will be identified. The first has identifier $\lceil (q.id + w\_size)/w\_slide \rceil + 1$, the last $\lceil q.id/w\_slide \rceil$;
- each window $\mathcal{W}$ with identifier $\mathcal{W}.id$ is assigned to the worker with identifier $j = (\mathcal{W}.id - 1) \bmod nw$;
- the result of `SchedByMulticast` is a vector containing all the identifiers of the workers receiving a copy of the quote q.

Like in the implementation of Sect. 5.2.1, single quotes are routed to the workers on-the-fly. Furthermore, as in the farm of Sect. 5.1, we are able to execute in parallel windows belonging to the same group. The code is shown in Listings 5.

Listing 5: Using *rpr::pipeline* and *rpr::farm* with multicast distribution.

```
1  [[rpr::pipeline,rpr::stream(q_Filtered,result)]]
2   while(socket.receive(&quoteIn,sizeof(quote_t))){
3   [[rpr::kernel,rpr::out(q_Filtered),rpr::target(CPU)]]
4    filterQuotes(quoteIn,q_Filtered);
5   [[rpr::kernel,rpr::farm(nw,unordered,SchedByMulticast),
6     rpr::in(q_Filtered),rpr::out(result),rpr::target(CPU)]] {
7    winManager(map,w_size,w_slide*nw,q_Filtered,w_task);
8    computeWindow(w_task,result);
9   }
10  [[rpr::kernel,rpr::in(result),rpr::target(CPU)]] {
11   std::vector<HReturn_type> &res_v=Ordering.newresults(result);
12   for (HReturn_type r : res_v) sendAndWrite(r);
13  }
14 }
```

At row 7 the `winManager` function is called with a proper slide parameter. Since quotes of the same stock symbol are assigned to different workers, the computation is activated each time a worker receives new `w_slide`×`nw` quotes. Furthermore, this implementation requires a change in the last stage of the pipeline. In fact, as for the basic farm of Sect. 5.1, the results of the same stock symbol can come to the last stage out of order. In this case it is not possible to use the *ordered* option of the farm attribute, because the window management is performed in the workers and there is no longer a one-to-one correspondence between inputs and outputs. To solve this problem the last stage consists in a block of two statements. The first is the call of a method of the `Ordering` object, which keeps the results of the same stock symbol ordered by their identifier and produces a vector of results once a new result has been received. Each result in this vector is finally passed to the `sendAndWrite`.

6 Experiments

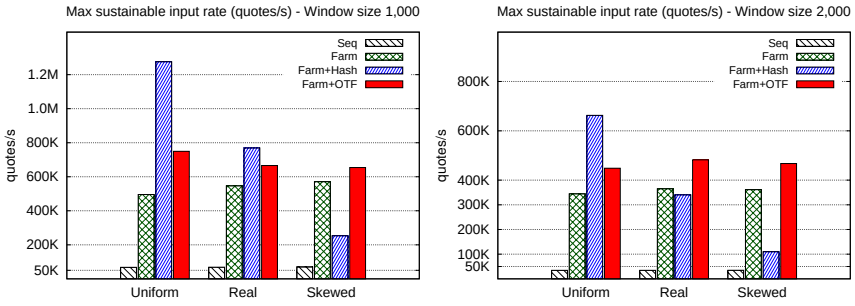
In this section we compare the parallel variants. The goal is to show the efficiency of our parallel implementations, designed through a high-level approach based on C++11 annotations and code refactoring techniques.

For the experiments, the target code produced by the source-to-source compilation (see Sect. 3) uses the FastFlow [12] library for pattern-based parallel programming. The target architecture is a dual-socket NUMA Intel multicore Xeon E5-2695 Ivy Bridge running at 2.40GHz featuring 24 cores (12 per socket). Each core has 32KB private L1, 256KB private L2 and 30MB shared L3. The operating system is Linux 3.14.49 x86_64 shipped with CentOS 7.1. We compiled our tests using GNU gcc 4.8.3 with the optimization flag `-O3`.

In the experiments we use the following acronyms to identify the implementations: `Farm` is the implementation described in Sect. 5.1 based on the REPARA farm, `Farm+Hash` is the implementation in which quotes are distributed according to a hash function (Sect. 5.2.1), and `Farm+OTF` is the implementation in Sect. 5.2.2, in which the standard farm is applied "On-The-Fly".

6.1 Synthetic Benchmarks

These experiments are aimed at understanding the maximum input rate that the computation sustains without becoming a bottleneck. The results take into account a slide of 25 quotes and windows of $1,000$ and $2,000$ quotes. In this second case the computation is coarser grained, as greater windows are triggered with the same frequency. The timestamps of the quotes are set to reproduce a stream with a constant input speed. We consider $2,836$ possible stock symbols with three distributions: *i) uniform*, in which all the symbols have the same frequency; *ii) real* is a distribution extracted from a random time instant of a trading day of NASDAQ[4], *iii) skewed* is a distribution in which the most frequent symbol has a probability of 0.20. The results showed in Figs. 3a and 3b are relative to implementations with 20 worker threads (two threads are used for the first and the last stage of the pipeline and other two for the emitter and collector of the farm).



(a) Window size $1,000$. (b) Window size $2,000$.

Fig. 3: Maximum sustainable input rate with the *uniform*, *real* and *skewed* distributions.

In the case of uniformly distributed stock symbols, the `Farm+Hash` version is the winner, with a maximum rate of 1.35M quotes per second. In this case we use a simple hash function in which the quotes with symbol s are assigned to the i-th worker such that $i = s \bmod nw$. With the other distributions this hash function is not able to assign the same number of quotes to the workers, and workload results unbalanced. In particular, with the skewed distribution the maximum theoretical scalability is quite low (it is the inverse of probability of the most frequent symbol, i.e. $1/p^{max}$ where $p^{max} = 0.2$). Therefore, `Farm+Hash` sustains a maximum rate of only 250K quotes per second. This problem can be approached by using different hash functions, e.g., if the distribution frequency is known, it is possible to find a static assignment of symbols to workers that minimizes the load difference.

The other two parallel implementations are independent from the frequency of the stock symbols (although we measured some small variations especially

[4] The trades and quotes NASDAQ tracefile of the 30 Oct 2014, downloadable at http://www.nyxdata.com.

with $|\mathcal{W}| = 1,000$). In fact, they allow different windows of the same stock symbol to be processed in parallel. The `Farm` version provides results worse than `Farm+OTF`. This is because in the `Farm` solution the first stage performs a plain copy of each window (the default behavior of the REPARA pipeline is to copy the output value of a kernel before passing it to the next stage). By merging the `winManager` and the `computeWindow` calls into the same kernel, the `Farm+Hash` and `Farm+OTF` versions avoid this copy. The relative improvement of the `Farm+OTF` implementation with respect to `Farm` one is of 15%. However, the distribution is a major concern in this version, because the multicast requires several copies of each input quote by preventing `Farm+OTF` to achieve the same performance of `Farm+Hash` with high parallelism degrees and the uniform distribution.

A similar behavior is depicted in Fig. 3b with larger windows. As expected, in terms of absolute values, the parallel implementations are capable of sustaining lower input rates, since the computation is coarser grained. From the qualitative viewpoint the previous discussion remains valid also in this case. The speedup results with 20 worker threads are summarized in Tab. 1.

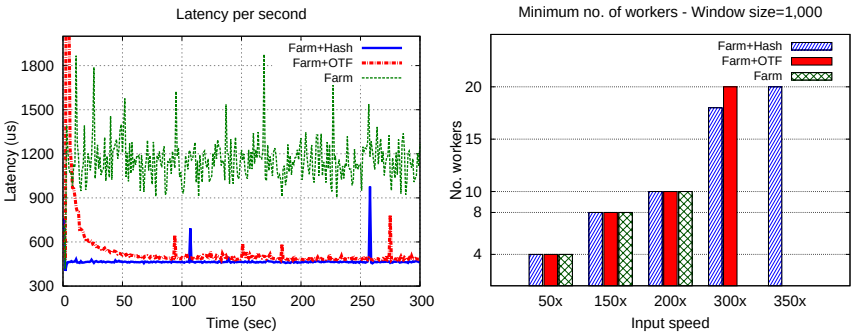
	`Farm+Hash`		`Farm+OTF`		`Farm`	
	$\|\mathcal{W}\|$=1,000	$\|\mathcal{W}\|$=2,000	$\|\mathcal{W}\|$=1,000	$\|\mathcal{W}\|$=2,000	$\|\mathcal{W}\|$=1,000	$\|\mathcal{W}\|$=2,000
`Uniform`	18.82	19.15	11.09	13.04	8.72	10.61
`Real`	11.25	9.87	9.75	14.05	8.00	11.03
`Skewed`	3.62	3.15	9.45	13.43	10.81	11.12

Table 1: Speedup results with 20 workers.

Furthermore, we study the provided *latency*. For each window we measure the time interval between the reception of the last quote of that window (measured by the first stage) to when the last stage receives the corresponding result. Fig. 4a shows an experiment consisting in 200 seconds in which we plot the average latency per second. We choose an input rate of 200K uniformly distributed quotes per second with windows of 1,000 tuples, and 10 workers. Under this configuration all the parallel variants do not act as bottleneck (see Fig. 3a). The latency provided by `Farm+Hash` is smaller (on average 577 us) compared to the standard `Farm` version, which provides an average latency of 1,460 us mainly due to the window copy. Instead, the `Farm+OTF` provides a latency close to the one of `Farm+Hash` (769 us). In conclusion, `Farm+OTF` represents a good compromise in scenarios in which we do not have (or it does not exist) a hash function that balances the workload among workers.

6.2 Results with the Real Dataset

We analyze the parallelizations proposed by using a real dataset of quotes generated by the NASDAQ market during a trading day. The dataset has a peak rate near to 60,000 quotes per second, with an average rate well below this figure. The goal is to find the minimum number of workers needed by the



(a) Average latency (microseconds). (b) Workers to sustain the input rate.

Fig. 4: Average latency and no. of Workers to sustain the accelerated real dataset.

parallel versions to avoid being a bottleneck. In the dataset both the frequency distribution of the stock symbols and the arrival rate change significantly over the execution. Since the original peak rate does not stress our implementations, we accelerate it several times. Fig. 4b shows the results. As we can observe, the three parallel implementations are able to sustain an accelerated input rate up to a factor of 200×. As expected, a higher input rate needs more workers. With 300× only the `Farm+Hash` and `Farm+OTF` implementations sustain the input stream pressure. With a factor of 350× only `Farm+Hash` successes.

7 Conclusions

In this paper we used the REPARA methodology to parallelize a data stream processing application. We showed that various parallel implementations can be easily prototyped, simply using proper REPARA attribute annotations in the sequential code. The results showed that the approach is effective in providing good performance results both on synthetic benchmarks as well as on a real-world use-case scenario. As a future extension of this work, we plan to make use of hardware accelerators such as GPUs and FPGAs that are possible targets in the REPARA framework.

Acknowledgements This work has been partially supported by the EU FP7 project REPARA (ICT-609666).

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. International Journal of Parallel Programming **42**(6), 1012–1031 (2014)
2. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing. Cambridge University Press (2014). Cambridge Books Online

3. Andrade, H., Gedik, B., Wu, K.L., Yu, P.S.: Processing high data rate streams in system s. J. Parallel Distrib. Comput. **71**(2), 145–156 (2011)
4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems, PODS '02, pp. 1–16. ACM, New York, NY, USA (2002)
5. Balkesen, C., Tatbul, N.: Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In: VLDB Inter. Workshop on Data Management for Sensor Networks (DMSN'11). Seattle, WA, USA (2011)
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (1995)
7. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. of the 2013 ACM SIGMOD Inter.national Conference on Management of Data, SIGMOD '13, pp. 725–736. ACM, New York, NY, USA (2013)
8. Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (2007)
9. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012)
10. Danelutto, M., De Matteis, T., Mencagli, G., Torquati, M.: Parallelizing high-frequency trading applications by using c++11 attributes. In: Proc. of the 1st IEEE Inter. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (2015)
11. Danelutto, M., Garcia, J.D., Sanchez, L.M., Sotomayor, R., Torquati, M.: Introducing parallelism by using repara c++11 attributes. In: Proc. of the 17th Euromicro PDP 2016: Parallel Distributed and network-based Processing. IEEE, Crete, Greece (2016)
12. Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. In: V. Zsók, Z. Horváth, L. Csató (eds.) Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)
13. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016. ACM, New York, NY, USA (2016)
14. Enterprise, C., Inc., C., NVIDIA, the Portland Group: The OpenACC Application Programming Interface, v1.0a (November 2011)
15. FastFlow website (2015). `http://mc-fastflow.sourceforge.net/`
16. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: An elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2351–2365 (2012)
17. IBM Infosphere Streams website (2015). `http://www-03.ibm.com/software/products/en/ibm-streams`
18. Intel® TBB website (2015). `http://threadingbuildingblocks.org`
19. ISO/IEC: Information technology – Programming languages – C++. International Standard ISO/IEC 14882:20111, ISO/IEC, Geneva, Switzerland (2011)
20. Kramer, P., Egloff, D., Blaser, L.: The alea reactive dataflow system for gpu parallelization. In: Proc. of the HLGPU 2016 Workshop, HiPEAC 2016, Prague (2016)
21. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Proc. of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, pp. 227–242. ACM, New York, NY, USA (2009)
22. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, first edn. Addison-Wesley Professional (2004)
23. REPARA Project Deliverable, "D3.3: Static partitioning tool"
24. REPARA website (2016). `http://repara-project.eu/`
25. REPARA Project Deliverable, "D2.1: REPARA C++ Open Specification document"
26. REPARA Project Deliverable, "D2.2: Static analysis techniques for AIR generation". Available at: `http://repara-project.eu/`
27. Apache Spark Streaming website (2015). `https://spark.apache.org/streaming`
28. Apache Storm website (2015). `https://storm.apache.org`
29. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Proc. of the 11th Inter. Conference on Compiler Construction, CC '02, pp. 179–196. Springer-Verlag, London, UK, UK (2002)