

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/281147715>

# Parallelizing High-Frequency Trading Applications by Using C++11 Attributes

Conference Paper · August 2015

DOI: 10.1109/Trustcom.2015.623

CITATIONS

8

READS

555

4 authors:



**Marco Danelutto**

Università di Pisa

319 PUBLICATIONS 3,589 CITATIONS

[SEE PROFILE](#)



**Tiziano De Matteis**

ETH Zurich

35 PUBLICATIONS 406 CITATIONS

[SEE PROFILE](#)



**Gabriele Mencagli**

Università di Pisa

88 PUBLICATIONS 676 CITATIONS

[SEE PROFILE](#)



**Massimo Torquati**

Università di Pisa

138 PUBLICATIONS 1,400 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



GridCOMP [View project](#)



In.Sy.Eme. [View project](#)

# Parallelizing High-Frequency Trading Applications by using C++11 Attributes

Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli and Massimo Torquati

Department of Computer Science, University of Pisa

Largo B. Pontecorvo, 3, I-56127, Pisa, Italy

Email: {marcod, dematteis, mencagli, torquati}@di.unipi.it

**Abstract**—With the wide diffusion of parallel architectures parallelism has become an indispensable factor in the application design. However, the cost of the parallelization process of existing applications is still too high in terms of time-to-development, and often requires a large effort and expertise by the programmer. The REPARA methodology consists in a systematic way to express parallel patterns by annotating the source code using C++11 attributes transformed automatically in a target parallel code based on parallel programming libraries (e.g. FastFlow, Intel TBB). In this paper we apply this approach in the parallelization of a real high-frequency trading application. The description shows the effectiveness of the approach in easily prototyping several parallel variants of the same code. We also propose an extension of a REPARA attribute to express a user-defined scheduling strategy, which makes it possible to design a high-throughput and low-latency parallelization of our code outperforming the other parallel variants in most of the considered test-cases.

**Keywords**—High-Frequency Trading, Parallel Patterns, C++11 Attributes, REPARA, Data Stream Processing, FastFlow.

## I. INTRODUCTION

Nowadays, with the increasing pervasivity of parallel architectures like multi-/many-core CPUs and GPUs, parallel programming has become not an alternative but rather a need for increasing the software performance. Programmers are forced to exploit parallelism to accelerate the execution of their applications. The approach of rewriting applications from scratch using parallel programming frameworks and libraries (e.g. TBB [1], FastFlow [2], TPL [3]) is too costly in terms of development time and time-to-solution. For this reason, a great effort has been made by the parallel computing research community in studying high-level programming interfaces to easily introduce parallelism in existing sequential (often legacy) codes. Examples are the pragma-based OpenMP model [4] and language extensions like Cilk [5].

The approach developed in the REPARA project [6] represents a novel and more programmer friendly high-level parallel programming interface. It is based on the C++11 program annotation features. The new C++11 standard offers the attribute mechanism [7], [8] to attach annotations to regions of a program by integrating such annotations directly into the abstract syntax tree. REPARA attributes have a rigorous syntax and clear semantics, and provide enough information for the transformation process needed to succeed in generating efficient run-time code (e.g. based on existing parallel programming libraries such as TBB and FastFlow). The programmer, usually an application domain expert but not a parallel programming expert, is involved in two phases: *i*) reshaping

its sequential code in order to be compliant to the standard REPARA C++ [9]; *ii*) identifying the regions of code that need to be accelerated by using proper attributes to express the desired parallelism *pattern*. 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 finally, they allow to express parallel patterns having a well-known parallel semantic relieving the programmer from the burden of dealing with the traditional parallel programming issues (e.g. low-level optimizations, synchronizations, mapping).

In this paper we show how the REPARA methodology can be effectively used in the domain of *Data Stream Processing* [10] (briefly, DaSP) to easily prototype various parallel implementations of the same code, reaching good performance with a reduced development cost. In the DaSP paradigm data from the streams are not stored persistently and then computed, but are rather processed on-the-fly by *continuous queries* that run constantly over the time by producing output results in a continuous way. The goal is to extract greater knowledge from the data, by maintaining a feasible history of the stream in the form of limited *windows*, detect patterns and generate complex events in realtime. A notable instance of these concepts can be found in the high-frequency trading domain. Financial applications recognize chart patterns within stock prices in order to predict their future development. In this paper we study a real trading application and we parallelize it using the REPARA approach. The contribution of this work is twofold:

- we evaluate the REPARA methodology step-by-step in a novel use-case, different from the project use-cases. We show how different parallel implementations can be easily defined by using proper REPARA attributes;
- we extend the set of attributes to accept a user-defined hash function for the scheduling of tasks in the *farm* pattern. The proposed solution is a general result, not limited to the financial application domain studied in this work.

The paper is organized as follows. Sect. II describes the related works. Sect. III provides a brief introduction about DaSP and the REPARA approach. Sect. IV shows the details of the studied high-frequency trading application. Sects. V and VI present the parallel implementations and their experimental evaluation on an off-the-shelf multi-core platform.

## II. RELATED WORK

The REPARA approach introduces parallelism by using C++11 attribute annotations. Compared to OpenMP [4], which

is pragma based, REPARA attributes provide additional flexibility as they can be attached to syntactic program elements while pragmas need to appear in separate source code lines. Moreover, complex patterns, as pipeline of parallel stages, are not suitable to be easily expressed with OpenMP directives. Frameworks and libraries like TPL [3], TBB [1] and FastFlow [2] generally require substantial changes in the source code to introduce parallelism, and sequential equivalence may not be easily preserved. Cilk [5] is a language extension of C/C++ for multithreaded parallel computing targeting recursive algorithms. It provides constructs for parallel loops and the fork-join idioms. However, it is an extension of the language and it is not directly supported by the C/C++ standard.

A common approach for multicores is to introduce parallelism by using patterns having well-known parallel semantics [11], [12]. Unfortunately, this requires a significant effort by the programmer not only limited to identify possible sources of parallelism, but also to select the most suitable pattern among those available. Moreover, this way to express parallelism is generally perceived as tedious because it requires changing/moving lines of code while introducing patterns. The REPARA approach is aimed at being a departure from this vision, by expressing parallel patterns through attributes of the same language used to express the sequential code. It uses the standard annotation mechanism of C++11 which requires low development effort for the programmer compared with using directly parallel programming frameworks and libraries.

This paper describes the application of the REPARA approach in the domain of Data Stream Processing [10]. Particular attention in the literature has been given to the parallelization of *stateful* operators in which the stream history is maintained by using succinct data structures (synopses, sketches, wavlets) or, more often, windows of the most recent data [13]. Common parallelizations [14], [15] consist in having replicas of the same operator that receive input elements belonging to the same substream identified by a partitioning key attribute. Examples of such operators are sorting, aggregation and one-way joins. Parallel implementations discussed in the literature are usually hand-coded using standard streaming frameworks like IBM InfoSphere [16], Spark Streaming [17] and Storm [18]. These frameworks do not offer to the programmer any adequate high-level abstractions for parallel programming (bolts and spouts of Storm are not sufficient to express high-level parallel programming), which is a distinguishable difference with the approach described in this paper.

### III. DATA STREAM PROCESSING AND THE REPARA APPROACH

This section provides the background about Data Stream Processing and the REPARA parallelization methodology.

#### A. Data Stream Processing

Data Stream Processing [10], [19] is a computing paradigm enabling the support to real-time processing of continuous data streams (e.g. financial data analysis, social media, networking and anomaly detection). DaSP applications can be modeled as flow graphs [13] where vertices are *operators* and arcs represent streams. The internal processing logic of each operator is responsible for consuming input elements

(*tuples*) and applying proper transformations on them. The first DaSP frameworks (*Data Stream Management Systems*) provide relation algebra operators such as *map*, *filters*, *aggregates* (sum, count, max), *sort*, *joins*, *skyline* and many others. More recent frameworks like Apache Storm [18], IBM InfoSphere [16] and Apache Spark Streaming [17] include the possibility to explicitly program operators by using an imperative programming style. DaSP operators can be classified into two classes [19], [20]: *i) stateless operators* (e.g. selection, filtering) work on a item-by-item basis without maintaining data structures created as a result of the computation on earlier data; *ii) stateful operators* maintain and update a set of internal data structures while processing input data. The result of the internal processing logic is affected by the current value of the internal state.

A special case is represented by *partitioned-stateful* operators [19], in which the data structures supporting the internal state can be divided into partitions corresponding to distinct segments of the input data, e.g. the partition to update depends on the value of a specific attribute (*key*) of the input tuples.

In most of the applications the internal state is used to maintain the history of the stream, e.g. to apply statistics or commonly in the financial domain to detect the presence of specific chart patterns (e.g. head-and-shoulders, double-bottoms). However, due to the potential unlimited length of the stream it is infeasible to maintain the stream history as a whole. There are two possible solutions to this problem [19]: *i)* the state can be implemented by succinct data structures such as *synopses*, *sketches*, *histograms* and *wavelets* especially useful to maintain aggregate statistics of the tuples received so far; *ii)* the internal processing logic could inspect all the attributes of the tuples that need to be maintained as a whole in the internal state. Fortunately, in realistic applications the significance of each tuple is time-decaying, and it is often possible to buffer only the most recent tuples in a temporary *window* buffer.

Windows are the predominant abstraction to implement the internal state of operators. The window semantics is expressed by two parameters: *i)* the *window size*  $|\mathcal{W}|$  in time units (seconds, minutes, hours) for *time-based windows* or in number of tuples for *count-based windows*; *ii)* the *sliding factor*  $\delta$  (in tuples or time units) expresses how the window moves and its content gets processed by operator's algorithm. Different windowing models can be identified: *sliding windows* with overlapping regions ( $\delta < |\mathcal{W}|$ ), *tumbling* (disjoint) *windows* ( $\delta < |\mathcal{W}|$ ) and *hopping windows* ( $\delta > |\mathcal{W}|$ ).

#### B. The REPARA parallelization methodology

The REPARA project [6] 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*. In this subsection we provide a brief description of this approach.

The methodology is structured into steps, each one taking into account different aspects of the parallelization of the code and implementing a clear *separation of concerns*. A *code annotation* phase, currently performed directly by the programmer, identifies the “kernels” subject to parallelization.

Then a *source-to-source* transformation phase<sup>1</sup> deals with the refactoring of the identified parallel kernels into suitable run-time calls, according to the Hardware&Software architecture(s) targeted (i.e. multi-cores equipped with GP-GPUs, FPGAs and DSPs). Eventually, a target-specific *compilation phase* generates the actual “object” (executable) code. The entire methodology steps are sketched in Fig. 1a.

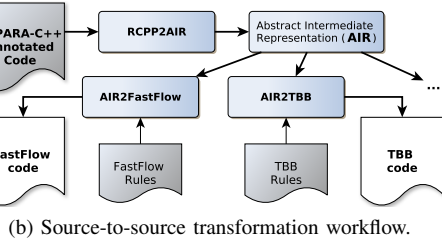
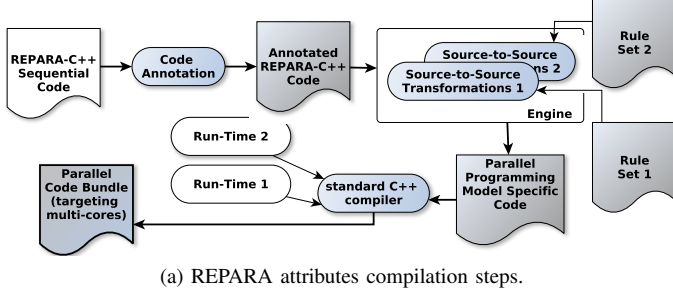


Fig. 1: Overview of the REPARA methodology.

The steps of the methodology are described in more detail in the following:

- 1) The first phase starts with a sequential program in which the user detects those parts of the code which can be annotated using C++ attributes. Tab. I shows a subset of the REPARA attributes, excluded the ones for the offloading on GPUs and FPGAs that are out of the scope of this paper.
- 2) In the second phase the annotated code is passed to the *Source-to-Source Transformation Engine* whose internal workflow is sketched in Fig. 1b. From the annotated source code, an *Abstract Intermediate Representation (AIR)* is generated. 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)*<sup>2</sup>. 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).
- 3) 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 should provide coordination and all the mechanisms

needed to support the deployment, scheduling and synchronization of kernels on the target platform(s).

REPARA C++ attr.	Description
rpr::kernel	Determines parallel region in a source code.
rpr::target	Specifies the target devices: CPU,FPGA,GPU,DSP,ANY.
rpr::pipeline	Defines pipeline pattern in a source code.
rpr::farm	Specifies the task-farm pattern as execution model.
rpr::in	Defines kernel input parameters.
rpr::out	Defines kernel output parameters.

TABLE I: Subset of the used REPARA C++ attributes.

In rest of this paper we will describe a financial trading application that uses the concepts of Data Stream Processing exposed in Sect. III-A. Then, we will apply the REPARA approach to parallelize it on a multi-core architecture by generating FastFlow code [21].

#### IV. FINANCIAL TRADING APPLICATION

High-Frequency Trading (HFT) is a representative application domain of Data Stream Processing characterized by tight 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 pattern [22] shown in Fig. 2.

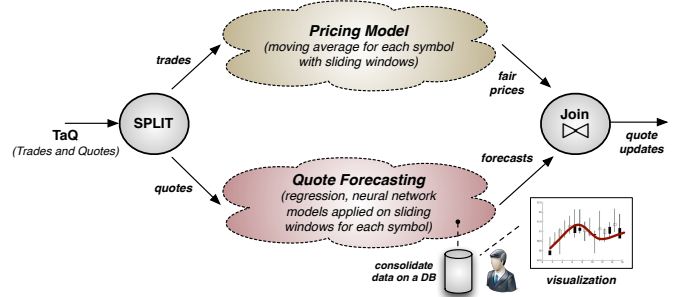


Fig. 2: High-Frequency Trading application: example of the split-compute-join pattern. Forecasting from existing quote data and correlation with the fair price for each stock symbol.

As sketched in the figure, the application is fed by a stream of elementary elements from the market. Input elements can be of two types: *trades* represent closed transactions (sell or buy market orders) 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. The raw input stream (*TaQ*) is typically split into two substreams - a trade stream and a quote stream processed independently by different processing chains.

Trades, grouped by the stock symbol attribute, are processed by a *Pricing Model* in order to estimate the fair price per group based on the most recent trades received from the market. The model usually employs simple moving average approaches [22] (e.g. Volume-Weighted Average Price) by maintaining a sliding window for each group. The *Quote Forecasting* chain processes bid and ask proposals grouped

<sup>1</sup>The tool is not fully implemented at present. For the experiments developed in this paper this phase is currently a manual step.

<sup>2</sup>REPARA imposes some restrictions on the parallelizable source code when targeting specific hardware.

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 [22], thus the quote stream pressure is more intensive than the trade stream; *ii*) prediction models (e.g. neural networks, regressions) are aimed at estimating the future volume and prices of bids and asks based on historical data, and are more computationally demanding than the moving averages computed by the trade chain. 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 earlier computation.

The results of the forecasting chain are visualized off-line by the human user to have a graphical feedback. Typically, candlestick charts are plotted by the visualization process as exemplified in Fig. 3 for a single stock symbol. Each

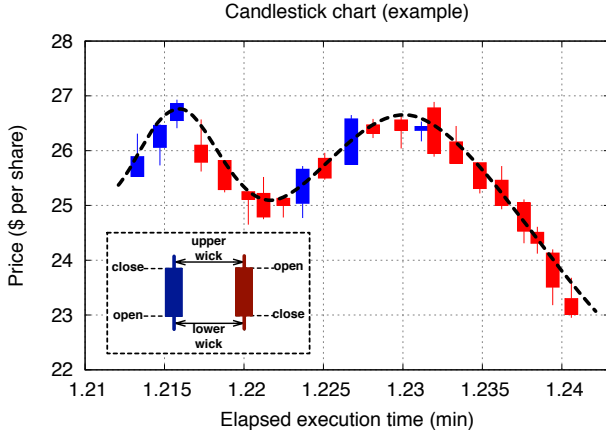


Fig. 3: Candlesticks for a stock symbol (bid quotes). The fitting line is a polynomial.

candlestick consists of a body (thickest part) and a upper and a lower wick (thinnest parts). The body is delimited by the opening and closing price of the stock symbol in a given period of time. The wicks are delimited by the maximum and the minimum price in the same timeslot. Fig. 3 shows the candlesticks, one for each slide of  $\delta = 100$  quotes composing the current window. In this example a window consists of  $10\delta$  quotes. Each time the computation is triggered (a new burst of  $\delta$  tuples is received), the oldest candlestick is removed and the new one added to the chart and the forecasting model updated. The figure shows the results with a regression curve that fits the data.

#### A. Sequential implementation

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) respecting the REPARA C++ directives [9].

The main consists of a while loop until the end of the stream. Each quote is filtered by a function `filterQuotes()`,

Listing 1: Quote Forecasting REPARA C++ code.

```
1 int main() {
2     long w_size, w_slide;
3     quote_t quoteIn, q_Filtered;
4     HReturn_type result;
5     WinTask w_task;
6     Receiver rcv(port);
7     std::map<int, CBWindow*> map;
8     ...
9     while(rcv.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, ...,
17                 const quote_t &quote, WinTask &w_task) {
18     CBWindow *win=map[quote.stock_symbol];
19     if(win==nullptr) {
20         win=new CBWindow(w_size, w_slide);
21         map[quote.stock_symbol]=win;
22     }
23     bool isTriggered=win->insert(quote);
24     if(isTriggered) w_task.set(win, SUCCESS);
25     else w_task.set(nullptr, EMPTY);
26 }
27 void computeWindow(const WinTask &w_task,
28                    HReturn_type &res) {
29     if(w_task.status==EMPTY) res.status=EMPTY;
30     else w_task.compute(res);
31 }
```

which deletes unused fields, updates statistics of the last received quotes and removes outliers, i.e. quotes that do not fit with the current trend of the last received quotes. This function is very fine grained, thus it is not central for the parallelization and its implementation details are 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. The quote is added to the window and a `WinTask` object is prepared with a reference to the window and a status attribute with two possible values: `SUCCESS` or `EMPTY`. The first identifies a non-empty task and is generated if and only if the reception of the new quote triggers the window activation (new  $w\_slide$  quotes of the same symbol have been received and the window slides forward). This is indicated by the boolean result of the `win->insert()` method. Otherwise, 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. Results are then stored in a local database and sent to the next stage of the application workflow by the function `sendAndWrite()`.

It is worth noting that the code described in Listing 1 is not the original one, but rather a reshaped version that adheres to the REPARA C++ constraints [9], [23]. In particular, the body of the while loop should contain only plain function calls and statements but not conditional or jump statements (`if`, `switch`, `break`, etc.). Each iteration of the while loop corresponds to the execution of a stream element, and should consists in a sequence of statements that can be eventually parallelized as it will be shown in Sect. V. In other words, the REPARA approach does not support control parallelism at present. In our application the function `filterQuotes()` is always



executed for each stream element, 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. The consequence is that the output of the `winManager()` function can be a significant task (if the window has been triggered by the last tuple reception) or an empty task which is simply discarded by `computeWindow()` which in turn produces an empty result. As it will be discussed in Sect. V, the presence of empty tasks and results may have impact on the performance, but it is an inevitable choice to produce parallel versions by simply annotating the sequential code with the REPARA attributes.

## V. PARALLEL IMPLEMENTATIONS

In this section we consider the REPARA parallelization of the sequential pseudo-code sketched in Listing 1. Since the application is a *stream-based* computation, we decided to use the REPARA *pipeline* pattern as a basic parallelization pattern.

The *pipeline* pattern combines multiple stages executed in a particular order. Data flows through these stages producing a *stream* of elements. Each stage consumes tasks present on the input stream and provides the output to the next stage (if present). The `rpr::pipeline` attribute identifies a block of code with several stages (functions) as a pipeline pattern. Stages are annotated with `rpr::kernel` attribute and the data flowing through the stages by the `rpr::stream` attribute. Usually, the data stream is created at the beginning of the first stage of the pipeline (i.e. by reading from a file or from a network socket). Produced data are defined as output parameter in the first kernel of the pipeline, and the data stream will be deleted at the end of the last stage of the pipeline. The semantics of streaming input and output attributes ensures that inputs are buffered until the next activation of the producer kernel, and outputs are buffered until the next activation of the consumer kernel. The REPARA pipeline requires a loop statement as a block of code, and uses the loop termination condition to determine the end of the stream. In case of nested loops, the pipeline will use the combination of the loop conditions defined below the pipeline attribute definition.

The REPARA attribute `rpr::farm` represents the execution of different stream elements (tasks) by the same kernel, which is replicated a number of times (each replicas called Worker) executed in parallel. The parallel execution of the same kernel may introduce tasks ordering problems due to their relative running times. The farm attribute makes it possible to specify whether the tasks must be produced in output by the kernel with the same order of input or not (i.e. *ordered*, *unordered* attributes). Ordering tasks when it is not necessary may produce lower overall performance because of higher main memory pressure and not optimal task load-balancing, therefore the default REPARA attribute is *unordered*. The attribute `rpr::farm` may be used in a pipeline kernel.

**Pipeline (Pipe):** Listing 2 shows the portion of the sequential code (lines 10–15 in Listing 1) that has been annotated using the REPARA attributes to implement a 3-stages pipeline. Annotations are highlighted in red in the code.


The while loop block defines three pipeline stages, whereas the loop condition defines the termination condition of the pipeline (*end-of-stream*). The first kernel is identified by the

Listing 2: Using `rpr::pipeline` pattern.

```

1  [[ rpr::pipeline, rpr::stream(w_task, result) ]]
2  while (rcv.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
8      [[ rpr::kernel, rpr::in(w_task), rpr::out(result)
9         rpr::target(CPU) ]]
10     computeWindow(w_task, result);
11
12     [[ rpr::kernel, rpr::in(result), rpr::target(CPU) ]]
13     sendAndWrite(result);
14 }

```



block statement containing the while loop and the two function calls `filterQuotes()` and `winManager()`. For each quote received from the network, the output produced is either a valid task containing a window of filtered quotes, all having the same stock symbol attribute, or an empty `w_task` if the input task has not triggered the window activation. The second stage executes the `computeWindow()` function on each valid input tasks producing in output either a valid or an empty result depending on the validity of the task received. Finally, the third stage executes the kernel `sendAndWrite()` on each valid (not empty) input results.

This parallelization is straightforward but has two major issues: *i)* the `computeWindow()` kernel has a higher computational cost than the other two kernels, thus representing a stream bottleneck in the pipeline; *ii)* there are many empty tasks flowing through the entire pipeline structure that cannot be discarded in the intermediate stages due to the REPARA C++ constraints. The management of such empty tasks will hamper the performance of this parallel implementation, as it will be proved experimentally in Sect. VI.

**Pipeline and Farm (Pipe&Farm):** to overcome the shortcomings of the pure pipeline implementation, the REPARA `rpr::farm` attribute can be used to annotate the `computeWindow()` kernel. Listing 3 shows the version that uses the farm pattern to execute in parallel  $nw$  replicas<sup>3</sup> of the `computeWindow()` kernel on different windows in order to (ideally) decrease the service time of that stage by a factor of  $nw$ . We use the attribute *ordered*, thus enforcing a strict ordering of tasks between farm’s input and output channels (total task ordering). Such strong condition is not really required by the application considered, which instead requires that output tasks are ordered by stock symbol attributes only (i.e. partial task ordering). To comply with this partial ordering condition, we have to enforce a stronger condition that may result a limiting performance factor. A different possibility, not conforming to the REPARA approach guidelines, would have been to not use the *ordered* attribute and to add some extra user code in the last kernel of the pipeline in order to enforce proper ordering “by hand”. Since this approach modifies the original sequential code, we decide to not consider this possibility in our implementations space.

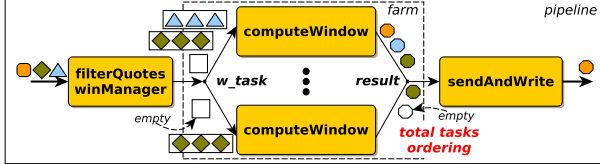
<sup>3</sup> $nw$  is the number of farm’s Workers. It is an application argument.

Listing 3: Using *rpr::pipeline* and *rpr::farm* patterns.

```

1 [[ rpr::pipeline , rpr::stream(w_task, result) ]]
2 while(rcv.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
8   [[ rpr::kernel, rpr::farm(nw, ordered),
9     rpr::in(w_task), rpr::out(result),
10    rpr::target(CPU) ]]
11   computeWindow(w_task, result);
12
13   [[ rpr::kernel, rpr::in(result), rpr::target(CPU) ]]
14   sendAndWrite(result);
15 }

```



With respect to the number of empty tasks, the pure pipeline implementation and the farm implementation have the same number of empty tasks flowing through.

**Pipeline and Farm with hashing (Pipe&Farm-wH):** a different parallel version can be designed by enforcing the nature of the computation which is, according to the description of Sect. III-A, a partitioned-stateful operator. A different window buffer is maintained for each stock symbol. Correctness of the computation is preserved if workers simultaneously update and compute windows corresponding to different stock symbols, i.e. if stock symbols are partitioned among workers. To do that, each incoming quote must be routed to a proper replica according to the result of a hashing function, i.e. all the tuples with the same stock symbol are always distributed to the same replica, which is in charge of executing both the `winManager()` and the `computeWindow()` functions such that each replica keeps a disjoint partition of the window hash-table (`std::map<int, CBWindow*> map`).

This parallel version cannot be directly implemented using the current REPARA attributes. Basically, it requires to define a tasks scheduling policy based on some hashing function. For this reason, we propose to extend the `rpr::farm` attribute by adding an extra optional parameter (other than the number of replicas and the ordering behavior) that is the name of a user-defined function that will be used by the runtime to determine to which farm's Workers the task has to be sent. The proposed extension is formalized as follow:

```

std::vector<size_t>
Func(const size_t N,
     const Tin1 &task1, ..., const TinK &taskK) {...}

[[ rpr::kernel, rpr::farm(nw, ordering, Func),
  rpr::in(task1, ... taskK), ... ]] kernel-region

```

the user function *Func* gets in input the current number of active farm's replicas  $N \in [0, nw[$  and the input *tasks* just received by the farm, whose types are defined by the user as  $Tin_i \forall i \in [1, K[$ . It returns a vector of indexes each one in the range  $[0, N[$ . The semantics is that the runtime will schedule the input tasks to all workers whose index is stored in the vector returned by *Func*. Listing 4 shows this parallel

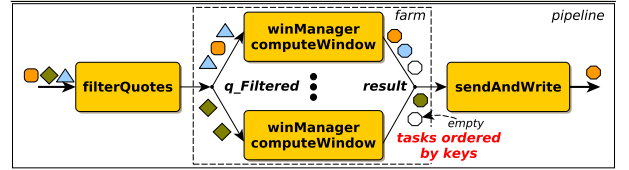
version that uses the proposed extension for task scheduling. It is worth noting that the extension proposed does not impair the sequential equivalence of the REPARA code.

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

```

1 std::vector<size_t>
2 SchedByKey(const size_t activeWorkers,
3            const q_Filtered &quote) {
4   std::vector<size_t> V(1);
5   V[0]=(quote.stock_symbol % activeWorkers);
6   return V;
7 }
8
9 [[ rpr::pipeline, rpr::stream(q_Filtered, result) ]]
10 while(rcv.receive(&quoteIn, sizeof(quote_t))) {
11   [[ rpr::kernel, rpr::out(w_task), rpr::target(CPU) ]]
12   filterQuotes(quoteIn, q_Filtered);
13
14   [[ rpr::kernel, rpr::farm(nw, unordered, SchedByKey),
15     rpr::in(q_Filtered), rpr::out(result),
16     rpr::target(CPU) ]] {
17     winManager(map, w_size, w_slide, q_Filtered, w_task);
18     computeWindow(w_task, result);
19   }
20
21   [[ rpr::kernel, rpr::in(result), rpr::target(CPU) ]]
22   sendAndWrite(result);
23 }

```



In this version, we use as hashing key the `stock_symbol` field of the quote received in input from the first pipeline stage. The function `SchedByKey()` returns the same replicas index for all quotes having the same symbol. The runtime guarantees to route each input task to the correct replicas.

As a result of this approach the number of empty tasks is reduced since they can be produced only between the second and the third stage. On the other hand, this version may suffer of possible load-balancing problems. When a small subset of all quote symbols have much higher probability than all the others, it may happen that a subset of Workers will have assigned much more tasks than the others. If we can predict the distribution of the quote symbol arrivals, this issue could be mitigated by using a smarter hashing function. Furthermore, since all the results with the same stock symbol are produced in output by the same replica, the attribute *ordered* in the `rpr::farm` annotation can be removed, thus enforcing the partial ordering of results without additional overhead.

## VI. EXPERIMENTS

In this section we evaluate and compare the parallel versions described before. We point out that we are not interested in the absolute performance. Rather, our goal is to show that our parallel implementations, easily developed using REPARA attributes on the sequential code, represent a good tradeoff between ease-of-development and performance. The target code produced by the source-to-source compilation phase (Sect. III-B) uses the FastFlow [21] runtime. The target architecture for the experiments is a dual-socket NUMA Intel multi-core Xeon E5-2695 Ivy Bridge running at 2.40GHz

featuring 24 cores (12+12) each with 2-way Hyperthreading. Each core has 32KB private L1, 256KB private L2 and 30MB shared L3. The operating system is Linux 2.6.32 x86\_64 shipped with CentOS 6.5. We compiled our tests using GNU gcc 4.8.2 with optimization flag `-O3`.

We distinguish two types of experiments: stress tests with fast streams and fixed probability distributions of the stock symbols, and experiments using a real dataset.

**Results of the stress tests:** the results are depicted in Figs. 4a and 4b. The plots show for each implementation (Seq, Pipe, Pipe&Farm and Pipe&Farm-wH) the maximum rate of the input stream (throughput) that the parallel implementations are able to sustain without being bottleneck over the entire execution. Fig. 4a shows the results with count-based windows (per stock symbol) of  $|\mathcal{W}| = 1000$  quotes with slide of  $\delta = 25$  quotes. We can observe that our code can be also adapted to time-based windows. Fig. 4b shows the results of the same experiment by doubling the window size ( $|\mathcal{W}| = 2000$  quotes) with the same slide parameter. This second case represents a more coarse grained execution: windows move with the same frequency but are larger than in the first case. The results are presented for three possible probability distributions of the stock symbols: 1) *real* represents a real distribution obtained during a trading day (daily TaQ of 30 Oct 2014) from the NASDAQ market with 2,836 traded symbols<sup>4</sup>; 2) *uniform* represents the uniform probability distribution among the stock symbols; 3) *skewed* represents a pessimistic scenario in which the more frequently traded stock symbol has probability 0.20.

The results show that the Pipe implementation provides results comparable (or slightly lower) than the original sequential code (Seq). The reason for the slightly lower results in some tests is due to the large number of empty tasks and results generated in the pipeline, and because the `computeWindow()` kernel, executed serially by the second stage, represents about 95% of the overall running time of the code. Better throughput is achieved by Pipe&Farm and Pipe&Farm-wH. In the uniform and real cases the version with hashing scheduling outperforms Pipe&Farm. This is due to two main reasons. Firstly, the farm version requires to buffer entire windows and distribute them to the workers, whereas the Pipe&Farm-wH version distributes single tuples on-the-fly to the replicas according to their symbol as soon as they are received from the stream. Secondly, the farm version produces extra empty tasks that are not present in the Pipe&Farm-wH version.

The skewed scenario shows a different behavior. In this case the Pipe&Farm version is the best one. The reason is that the version with the hashing scheduling is hampered by serious load balancing issues, as the scheduling policy distributes all the tuples with the same stock symbol to the same worker. Since in this case the most frequent symbol has a very high probability, the assigned worker receives more tasks than the others, preventing to balance the workload properly. The speedup results (ratio between the highest sustainable throughput of the parallel implementation and the one of the sequential code) are summarized in Tab. II showing also the corresponding number of workers between parenthesis.

A similar trend is depicted in Fig. 4b with larger windows of 2,000 tuples. As obvious, in terms of absolute values the

	Pipe&Farm		Pipe&Farm-wH	
	$ \mathcal{W}  = 1000$	$ \mathcal{W}  = 2000$	$ \mathcal{W}  = 1000$	$ \mathcal{W}  = 2000$
Real	8.12(12)	9.87(14)	12.47(19)	11.57(19)
Uniform	8.00(10)	11.10(12)	16.65(20)	18.53(20)
Skewed	8.66(16)	9.24(16)	4.46(20)	4.46(20)

TABLE II: Speedup of the best parallel implementations in different scenarios:  $\text{speedup}(\text{parDegree})$ .

parallel implementations are capable of sustaining lower input rates, since the computation is more coarse grained.

**Results with the real dataset:** we consider a real dataset of quotes generated in the NASDAQ market during one trading day. The peak rate observed in our dataset is near to 60,000 quotes per second, with an average rate well below this figure. Recent estimates for market data rates [22] are near to 1 million of transactions per seconds (especially if we consider options data and not stocks quotes), highlighting the need of high-performance implementations of trading strategies on today's parallel machines. To adhere to this trend, we study three cases in which we accelerate our dataset in order to reproduce throttled input rates with a realistic distribution of quotes. We consider three scenarios of  $50\times$ ,  $100\times$  and  $200\times$  throttled input rates. Fig. 5a shows for the three scenarios the minimum number of cores needed by the parallel implementations to sustain the throttled input rate in the three scenarios. As expected, with very fast rates ( $200\times$ ) only the Pipe&Farm-wH implementation is capable of sustaining the peak rate, since it exhibits better speedup than the Pipe&Farm version. Fig. 5b concludes the experimental section by showing the latency throughout the execution. We refer to latency as the average time between the reception a tuple triggering the window activation, and the production of the corresponding output. As we can observe, lower latency is achieved with the parallelization exploiting the hashing distribution, since single quotes are distributed on-the-fly to the workers without needing to buffer entire windows and in the Pipe&Farm case.

## VII. CONCLUSIONS

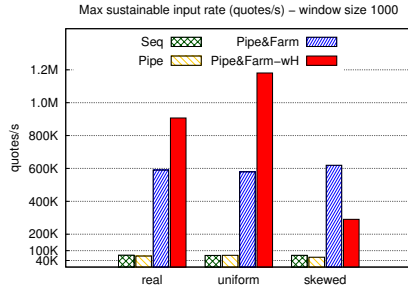
In this paper we apply the REPORA methodology to the parallelization of a high-frequency trading application. We show that various parallel implementations can be easily prototyped using this methodology, simply using proper C++11 attributes in the needed regions of the sequential code. We propose an extension of the REPORA attribute set to express a user-defined distribution policy. The results show that our parallel implementations are capable of achieving good performance on synthetic benchmarks as well as on a real-world experiment using a real dataset. In the future we plan to compare the REPORA methodology with similar existing approaches like the most recent OpenMP 4.0 standard and the previous work developed in the ParaPhrase project [24].

## ACKNOWLEDGMENT

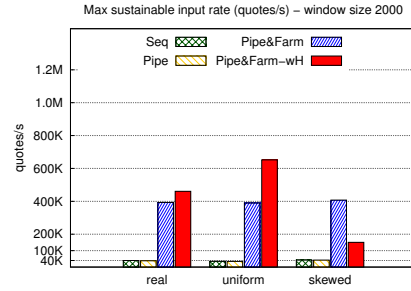
This work has been partially supported by the EU FP7 project REPORA (ICT-609666).

<sup>4</sup>The used dataset is freely available at the website: <http://www.nyxdta.com>.



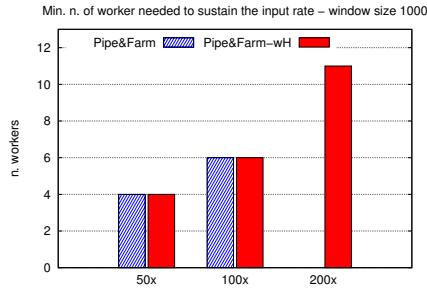


(a) Sustained throughput, window size 1000.

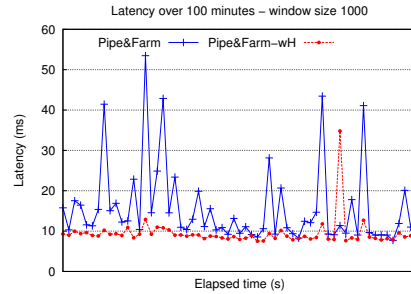


(b) Sustained throughput, window size 2000.

Fig. 4: Stress tests: highest sustained throughput with different quotes probability distributions (real, uniform and skewed distributions).



(a) Workers needed to sustain the input rate.



(b) Average latency (ms). 100x throttled input rate.

Fig. 5: Results using a real dataset of TaQ. The real input dataset has been throttled by 50 $\times$ , 100 $\times$  and 200 $\times$  factor.

## REFERENCES

- [1] Intel® TBB website, 2015, <http://threadingbuildingblocks.org>.
- [2] FastFlow website, 2015, <http://mc-fastflow.sourceforge.net/>.
- [3] D. Leijen, W. Schulte, and S. Burckhardt, “The design of a task parallel library,” in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’09. New York, NY, USA: ACM, 2009, pp. 227–242.
- [4] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
- [5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: an efficient multithreaded runtime system,” *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, Aug. 1995.
- [6] REPARA website, 2015, <http://repara-project.eu/>.
- [7] J. Maurer and M. Wong, “Towards support for attributes in C++ (revision 6),” in *JTC1/SC22/WG21 - The C++ Standards Committee*, 2008, N2761=08-0271.
- [8] ISO/IEC, “Information technology – Programming languages – C++,” ISO/IEC, Geneva, Switzerland, International Standard ISO/IEC 14882:20111, Aug. 2011.
- [9] REPARA Project Deliverable, “D2.1: REPARA C++ Open Specification document”, 2014, available at: <http://repara-project.eu/>.
- [10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’02. New York, NY, USA: ACM, 2002, pp. 1–16.
- [11] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison-Wesley Professional, 2004.
- [12] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, “Design patterns percolating to parallel programming framework implementation,” *International Journal of Parallel Programming*, vol. 42, no. 6, pp. 1012–1031, 2014.
- [13] G. Cugola and A. Margara, “Processing flows of information: From data stream to complex event processing,” *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.
- [14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, “Streamcloud: An elastic and scalable data streaming system,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.
- [15] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13. New York, NY, USA: ACM, 2013, pp. 725–736.
- [16] “Ibm infosphere streams,” <http://www-03.ibm.com/software/products/en/infosphere-streams>.
- [17] “Apache spark streaming,” <https://spark.apache.org/streaming>.
- [18] “Apache storm,” <https://storm.apache.org>.
- [19] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing*. Cambridge University Press, 2014, cambridge Books Online.
- [20] D. Buono, T. De Matteis, and G. Mencagli, “A high-throughput and low-latency parallelization of window-based stream joins on multicore,” in *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, Aug 2014, pp. 117–126.
- [21] M. Danelutto and M. Torquati, “Structured parallel programming with “core” fastflow,” in *Central European Functional Programming School*, ser. LNCS, V. Zsóka, Z. Horváth, and L. Csátó, Eds. Springer, 2015, vol. 8606, pp. 29–75.
- [22] H. Andrade, B. Gedik, K. L. Wu, and P. S. Yu, “Processing high data rate streams in system s,” *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 145–156, Feb. 2011.
- [23] REPARA Project Deliverable, “D3.3: Static partitioning tool”, 2014.
- [24] ParaPhrase website, 2015, <http://http://www.paraphrase-ict.eu/>.