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

The RePhrase Extended Pattern Set for Data Intensive Parallel Computing

M. Danelutto⋆, T. De Matteis⋆, D. De Sensi⋆, G. Mencagli⋆, M. Torquati⋆, M. Aldinucci∗, P. Kilpatrick°

the date of receipt and acceptance should be inserted later

Abstract We discuss the extended parallel pattern set identified within the EU-funded project *RePhrase* as a candidate pattern set to support data intensive applications targeting heterogeneous architectures. The set has been designed to include three classes of pattern, namely i) core patterns, modelling common, not necessarily data intensive parallelism exploitation patterns, usually to be used in composition; ii) high level patterns, modelling common, complex and complete parallelism exploitation patterns; and iii) building block patterns, modelling the single components of data intensive applications, suitable for use–in composition–to implement patterns not covered by the core and high level patterns. We discuss the expressive power of the *RePhrase* extended pattern set and results illustrating the performances that may be achieved with the FastFlow implementation of the high level patterns.

Keywords Parallel design patterns, data intensive computing, stream computing, algorithmic skeletons

1 Introduction

Data intensive applications are becoming more and more important. On one side more and more data is available from different sources including mobile devices and distributed computing platforms. On the other side, the notable improvement in the hardware available for data processing favoured the development of new, highly demanding algorithms and applications.

However, the design, development and tuning of efficient data intensive applications still represents a very challenging task. By necessity, these applications must be designed and implemented as parallel applications. In addition to all the usual problems related to parallel computing, these applications also

Univ. of Pisa⋆, Queen's Univ. of Belfast°, Univ. of Torino∗
E-mail: {marcod, dematteis, desensi, mencagli, torquati}@di.unipi.it⋆, p.kilpatrick@qub.ac.uk°, aldinuc@di.unito.to∗

Fig. 1: *RePhrase* methodology workflow overview

face the programmers with the problem of efficiently managing considerable amounts of data, often available as streams dictating precise performance constrains.

Parallel design patterns have been identified as viable mechanisms to support parallel programmers in the hard task of designing and implementing efficient and portable parallel applications [11, 27]. Several existing and widely used programming frameworks provide the programmer of parallel applications with ready to use parallel patterns. Google mapreduce [16], Hadoop [35] and OpenMP basically provide a single pattern while Intel Thread Building Blocks [30] and Microsoft TPL [28] both provide a larger set of patterns. All the programming frameworks developed as algorithmic skeleton programming frameworks also include comprehensive sets of parallel patterns provided as ready to use programming abstractions: FastFlow [15], Muesli [22], SKEPU [21], SkeTo [20]. In some cases the programming frameworks may be exploited to target different kind of architectures. As an example, Muesli targets workstation clusters, shared memory multicores and GP-GPUs and FastFlow targets multicores, GP-GPUs and provides partial support to target clusters of workstations as well as FPGAs (through TPC [24]).

Although some of the pattern frameworks just mentioned have been explicitly designed to support data intensive applications, there is no clear idea about the set of patterns needed to support data intensive applications.

Within *RePhrase*, an EU H2020 funded project started in April 2015, we aim at defining a set of parallel patterns supporting the development of efficient data intensive applications on heterogeneous hardware platforms. In particular, we aim at providing a set of parallel design patterns as ready to use programming abstractions fully compliant with standard C++ (11 and following standard releases) paired with a set of tools suitable to support pattern introduction in existing or brand new C++ code via refactoring and to check and ensure different properties on the resulting parallel code. Fig. 1 summarizes the overall approach of the *RePhrase* project.

In this paper, we introduce the parallel pattern set identified within the *RePhrase* project to support data intensive applications. Our main contribution consists therefore in the formalization of a comprehensive parallel pattern set along with some preliminary results demonstrating the expressive power of the patterns and some performance results achieved with a FastFlow implementation of these patterns.

The rest of the paper is organized as follows: Sec. 2 introduces data intensive computing paradigm. Sec. 3 describes the full set of patterns included in the *RePhrase* extended pattern set. Sec. 4 discusses expressive power and usability of the pattern set. Eventually, Sec. 5 discusses some preliminary results relative to the pattern set implementation using our structured parallel programming framework FastFlow.

2 Data Intensive Processing

We live in a world driven by information: electronic devices, manufacturing equipment and information systems produce data driven by user interactions or in automatic way. In this scenario, *Data Intensive Computing* is gaining importance as a means of collecting, analysing and unveiling the knowledge that this data encapsulates. Clearly, this possibility constitutes a valuable opportunity for many businesses and scientific applications.

For these reasons, *Big Data* is one of the leading IT trending topics of today. It is characterized by the so-called 3Vs [25]: *variety*, *volume* and *velocity*. Variety refers to the nature and structure of the information. Volume refers to the magnitude of data produced. Finally, velocity refers to the frequency of data generation as well as to the dynamic aspects of the data. Different processing paradigms tackle different combinations of these aspects.

Pure *Data Parallel* systems tackle the volume and variety aspects: they process large masses of data, usually in an off-line fashion. Typically applications range across various scientific sectors: we can have the analysis of massive data coming from scientific experiments [37], studies of human digital traces (e.g. GPS traces) to discover and understand patterns in human mobility [29] or to support health care assistance [33]. Frameworks in this field take inspiration from the *Google Map Reduce* work [17]. Notable open source implementations include Apache Hadoop [8] and more recently Apache Spark [9], which is gaining attention due to its versatility and efficiency.

In turn, *Data Stream Processing* (DaSP) deals with the velocity and variety aspects of the "Big Data Challenge". According to the DaSP paradigm, applications receive a *continuous* flow of data that has to be processed on the fly, usually with performance requirements in terms of bandwidth and/or latency [12,6]. Examples in this field include financial applications that try to spot revenue opportunities by analyzing live market data [5], *Intrusion Detection Systems* that monitor network traffic in real-time to identify possible attacks [38], social media analytics that gather users' news feeds and try to detect notable events [34]. In the area of DaSP frameworks, we have, in recent years, assisted to a large number of proposals from both academia, open source and industry communities. Generally, applications are expressed as compositions of core functionalities in directed flow graphs, where vertices are *operators* (that encapsulate user defined logic) and arcs model streams, i.e. unbounded sequences of data items (*tuples*) sharing the same properties in terms of name and type of attributes. Examples of solutions in this sphere include Apache Storm [10], Apache Flink [7] and IBM InfoSphere Stream [23].

At times both aspects of data intensive processing can be present, allowing systems to serve a wider range of workloads and use cases. This approach is sometimes referred as *Lambda Architecture* [26].

3 The *RePhrase* pattern set

The set of parallel patterns developed within *RePhrase* has been incrementally designed. In a first phase, we considered classical parallel patterns already know to be effective in the support of classical parallel applications. This initial set includes two kind of pattern:

- a set of *core* patterns, that comprises classical primitive parallelism exploitation patterns and may be specialized by means of a set of parameters to implement various applications using the pattern in slightly different ways; and
- a set of *high level* patterns, representing common, complex and specialized parallel patterns.

The first class includes, for example, pipeline and parallel for/map patterns, while the second class includes examples such as divide&conquer and Google mapreduce patterns.

Subsequently, taking into account the industrial use cases employed to assess the project results, we extended the pattern set with some further high level patterns and with a collection of small "building block" patterns suitable for use, in composition, to model those data intensive patterns not captured by the *RePhrase* high level patterns.

In the remainder of this section we introduce the patterns included in the *RePhrase* pattern set. The patterns are divided into classes according to the kind of parallelism exploited (data, task, stream, etc.).

3.1 Stream Parallel "core" patterns

Stream parallel patterns exploit parallelism in the processing of different items belonging to one or more input data streams. An input data stream is characterized by having a type[1] and by being able to provide items (to be computed) one after the other with a given *inter-arrival time*. We will denote the type of a stream of data items of type α by α `stream`. A stream may be *finite*–in this case the last item of the stream will be the special item `eos`–or *infinite*. The infinite streams usually originate from some kind of input devices, e.g. a network card. Our "core" stream parallel patterns all process a single input stream to produce a single output stream.

Pipeline (`pipe`): the pattern computes in parallel several stages $f_1, \dots, f_n$ on a stream of items, where $f_i : \alpha_{i-1} \to \alpha_i$ and (`pipe` $f_1 \ \dots \ f_n$) : α_0 `stream` $\to$ α_n `stream`. Each stage processes data produced by the previous stage in the pipe and delivers results to the next stage. For each stream item x an item $f_n(f_{n-1}(\dots f_1(x)\dots))$ is eventually delivered in the pipeline output stream. Pipeline stages are executed in parallel.

[1] in the following we'll use greek letters to denote data types. The expression $x : \alpha$ will be used to denote an object x whose type is α while the expression $f : \alpha \to \beta$ will be used to denote a function f computing a result of type β out of an input data of type α.

Task-Farm (`farm`): the pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing in an input stream and therefore $(\texttt{farm}\ f) : \alpha\ \mathsf{stream} \rightarrow \beta\ \mathsf{stream}$. type α stream delivering the results on the output stream of type β stream. Computations relative to different stream items are independent.

Stream Filter (`filter`): the pattern computes in parallel a filter $p : \alpha \rightarrow \{true, false\}$ over an input stream of type α stream, that is passes to the output stream only those input data items x such that $p(x) = true$. p must be a pure function and $(\texttt{filter}\ p) : \alpha\ \mathsf{stream} \rightarrow \alpha\ \mathsf{stream}$.

Stream Accumulator (`accumulator`): The pattern "sums up" using a binary function $\oplus : \alpha \times \alpha \rightarrow \alpha$ all items from the input stream and delivers the result to the output. The function used to sum up values ($\oplus$) may be any kind of binary function of type $\oplus : \alpha \times \alpha \rightarrow \alpha$, although commutative and associative functions will provide much better and more scalable implementations. $(\texttt{accumulator}\ \oplus) : \alpha\ \mathsf{stream} \rightarrow \alpha$.

Stream Iteration (`iteration`): the pattern iterates the computation of another pattern over one or more items appearing onto the input stream, and delivers results on the output stream. The pattern has type $\texttt{iteration}\ (\alpha\ \mathsf{stream} \rightarrow \alpha\ \mathsf{stream}) \times (\alpha \rightarrow \texttt{bool}) \rightarrow (\alpha\ \mathsf{stream} \rightarrow \alpha\ \mathsf{stream})$. The first parameter is the nested pattern, the second one is the function used to redirect output item x to the input of the nested pattern (true) or to the output of the iteration pattern (false).

3.2 Data Parallel "core" patterns

Data parallel patterns exploit parallelism in the processing of different items or (possibly overlapping) partitions of items belonging to a single "collection" data item. The key point in this case is the existence of two (logical) functions decomposing a single input data collection (of type α collection) into a collection of collections ($\texttt{decomp}: \alpha\ \mathsf{collection} \rightarrow (\alpha\ \mathsf{collection})\ \mathsf{collection}$) and building the result out of the collection of subresults ($\texttt{comp}: \beta\ \mathsf{collection} \rightarrow \gamma$). Data parallel patterns process a single collection at a time, but nothing prevents they are used to operate on a *stream* of collections to produce a *stream* of collections.

Map (`map`): this pattern computes a given function $f : \alpha \rightarrow \beta$ over all the data items of an input collection whose elements have type α ($\texttt{map}\ f : \alpha\ \mathsf{collection} \rightarrow \beta\ \mathsf{collection}$). Therefore the `decomp` function (logically) returns a set of α singletons out of the α collection input and the the `comp` rebuilds a β collection out of the collection of singleton results. Given the input collection $x_1, \ldots, x_N$, the output collection is $f(x_1), \ldots, f(x_N)$. Since each data item in the input collection is independent of the other items, all the elements can be computed in parallel.

Reduce (`reduce`): the pattern "sums up" all the data items of a collection of items of type α using a binary function $\oplus : \alpha \times \alpha \to \alpha$ which is usually associative and commutative (`reduce` $\oplus : \alpha$ $\mathsf{collection} \to \alpha$). Given the input collection $x_1, \ldots, x_N$, the `reduce` computes $x_1 \oplus \ldots \oplus x_n$.

Stencil (`stencil`): the pattern decomposes an input collection ($x : \alpha$ $\mathsf{collection}$) in a set of as many sub collections ($y : \alpha$ $\mathsf{collection}$) as the original collection component count. Each sub collection hosts a distinct item of the original collection along with a set of *neighbour* items. A function $f : \alpha$ $\mathsf{collection} \to \beta$ is used to compute in parallel the new values of the output $z : \beta$ $\mathsf{collection}$.

3.3 High Level Patterns

High level patterns model more complex parallel patterns. We only informally specify the intended parallel semantics. All are used to compute the result relative to a single input, although they may be used in composition with stream parallel patterns to compute stream of results out of stream of inputs.

Divide and Conquer (`dac`): the pattern computes a problem for which *a)* the solution for some base cases are known and *b)* non-base case problems may be divided into a collection of sub-problems and *c)* the solution of the non-base case problems may be computed out of the solutions of the sub-problems. The type of the pattern is `dac` : $divide \times conquer \times isBaseCase \times SolveBaseCase \times \alpha \to \beta$ with $divide : (\alpha \to \alpha$ $\mathsf{collection})$, $conquer : (\beta$ $\mathsf{collection} \to \beta)$, $isBaseCase : (\alpha \to \mathsf{bool})$ and $solveBaseCase : (\alpha \to \beta)$

Mapreduce (`mapreduce`): the pattern computes the Google mapreduce [17], using two functions $f : \alpha \to \beta \times \kappa$ and $\oplus : \beta \times \beta \to \beta$, where κ is the key type, and has type `mapreduce` $f \oplus : \alpha$ $\mathsf{collection} \to (\beta \times \kappa)$ $\mathsf{collection}$. The first function (f) is used to map all the items in the input collection to ⟨key, value⟩ pairs, while the second one ($\oplus$) is used to compute a unique value out of the *value* entries in ⟨key, value⟩ pairs with the same *key* value.

Pool pattern (`pool`): the pattern models the evolution of a population of individuals. Iteratively, selected individuals are subject to evolution steps. The resulting new individuals are inserted in the population or discarded according to their fitness score. The process is iterated up to a given number of iterations (or up to a given computation time) or up to the point an individual with a given fitness is inserted in the population. Low fitness individuals may be removed from the population to keep the population size constant at each iteration. The type of this pattern is therefore `pool` *sel evol fit merge term* : α $\mathsf{collection} \to \alpha$ $\mathsf{collection}$ where $sel : \alpha$ $\mathsf{collection} \to \alpha$ $\mathsf{collection}$, $evol : \alpha \to \alpha$, $fit : \alpha \to \beta$, $merge : \alpha$ $\mathsf{collection} \times \alpha$ $\mathsf{collection} \to \alpha$ $\mathsf{collection}$, $term : \alpha$ $\mathsf{collection} \to$ `bool`

Image convolution pattern (`convolve`): this pattern computes image convolution according to some input kernel parameter and has type `convolve` : α mat $\times$ int mat $\rightarrow$ α mat. A kernel parameter is an $N \times N$ matrix (usually 3×3 or 5×5) of integer values. The image convolution is obtained from the source image processing each pixel at position i, j by taking the $N \times N$ values centered at i, j, multiplying each of the values by the corresponding value of the kernel and summing up all the results to get the new i, j pixel of the resulting matrix. Image convolution may be used to obtain different effects with different kernels, ranging from image blurring to image enhancement, embossing, sharpening etc. The image convolution pattern may be obviously implemented using a stencil pattern, but it is provided as a first class pattern due to its wide usage.

Windowed stream farm (`windowedSF`): the pattern computes functions on windows of stream item values, and has type `windowedSF` : α stream $\times$ (α vec $\rightarrow$ β) $\rightarrow$ β stream. In particular, this pattern implements a computation that outputs items on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the input stream. The windows have a length (number of items to be listed in the window) and an overlap factor (number of items in window w_i also appearing in window w_{i+1}). The number of items in a window may be defined either as an actual number (count-based windows) or as a time interval, that is as the items appearing onto the input stream within the given interval of time (time-based windows).

Keyed stream farm (`keyedSF`): the pattern computes functions on windows of stream item values, and has type `keyedSF` : α stream $\times$ (α vec $\rightarrow$ β) $\times$ (α $\rightarrow$ γ key) $\rightarrow$ β stream. Each input item belongs to a unique class called key (with type γ key); that is, the physical stream can be viewed as a multiplexing of several logical streams, each of which conveys items with the same key value. This pattern implements a computation that outputs items on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the same logical input stream. The windows have a length (number of items to be listed in the window) and an overlap factor (number of items in window w_i also appearing in window w_{i+1}). The number of items in a window may be defined either as an actual number (count-based windows) or as a time interval, that is as the items appearing onto the input stream within the given interval of time (time-based windows).

3.4 Data intensive building block patterns

The patterns in this class are further divided into patterns used to generate/collapse data streams and in patterns used to process existing streams.

3.4.1 Stream generate/collapse patterns

Stream generator pattern (`streamgen`): this pattern is used to generate a stream from an internal (e.g. a stateful function) or external (e.g. a disk file) data source and has type `streamgen` : $() \to \alpha$ stream[2].

Stream collapser pattern (`streamdrain`): this pattern is used to "consume" all the items appearing on its input stream and has type `streamdrain` : α $\mathsf{stream} \to ()$.

Data splitter pattern (`datasplitter`): the pattern is used to generate a stream of items out of the components of a data collection (possibly from the pattern input stream) according to a user-defined strategy and has type `datasplitter` : α $\mathsf{collection} \to \beta$ stream, where β is either α or α $\mathsf{collection}$.

Data merger pattern (`datamerger`): the pattern is used to gather items appearing onto an input stream in a data collection according to a user defined strategy and to deliver the data collection onto the pattern input stream and has type `datamerger` : α $\mathsf{stream} \to (\alpha$ $\mathsf{collection})$ stream.

3.4.2 Stream processing patterns

Stream filter pattern This is the very same `filter` pattern included in the "core" stream patterns (Sec. 3.1. It is listed here as logically it belongs to the stream processing subclass of the data intensive building block patterns.

Stream merger pattern (`streamMerger`): this pattern is used to merge two or more input streams into a single output stream according to a pre-defined or user-specified merge policy and has type `streamMerger` : $(\alpha$ $\mathsf{stream})$ $\mathsf{collection} \to$ α stream.

Stream tupler pattern (`streamTupler`): this pattern processes items from a set of input streams to produce a tuple onto a single output stream with exactly one item from each of the input streams and has type `streamTupler` : α_1 $\mathsf{stream} \times \ldots \times \alpha_m$ $\mathsf{stream} \to (\alpha_1 \times \ldots \times \alpha_m)$ stream.

Stream splitter pattern (`streamSplitter`): this pattern directs the items appearing onto a single input stream to one of the different output streams according to a pre-defined or user-defined split policy and has type `streamMerger` : α $\mathsf{stream} \to (\alpha$ $\mathsf{stream})$ $\mathsf{collection}$.

[2] being () the "no parameter (void) type

Stream detupler pattern (`streamDetupler`): the pattern processes tuples appearing onto an input stream. Each tuple is used to generate items on different output streams according to a parameter policy and has type `streamTupler` : $(\alpha_1 \times \ldots \times \alpha_m)$ $\mathsf{stream} \rightarrow \alpha_1$ $\mathsf{stream} \times \ldots \times \alpha_m$ stream. Default policies are provided including:

- scatter (tuple components to different output streams, in order)
- unicast (tuple components to the same output stream, one after the other, the stream is identified through a user supplied function)

4 Expressive power of the *RePhrase* pattern set

We discuss the expressive power of the *RePhrase* extended pattern set in terms of two different aspects:

- the class of data intensive applications supported; and
- the programming effort required to code a data intensive application using the patterns in comparison with the effort required to program the same applications using traditional, "non patterned" programming frameworks.

4.1 Applications supported

The three different kinds of pattern provided within the *RePhrase* extended pattern set all support partially overlapping, different classes of applications (see below). We are currently completing the port of the full Parsec benchmarks in FastFlow (see Fig. 5c), the library we use to provide the application programmers RePhrase patterns [13] and we have already checked the possibility to implement all of the Cowichan problems [36] using the RePhrase pattern set. In addition, and obviously, the pattern set cover all the parallel needs of the RePhrase use case set [31].

High level patterns Each of the high level patterns in the *RePhrase* extended pattern set supports a complex and complete set of well-know parallel pattern. In general each pattern may also be implemented using a (composition of) core pattern(s) although this alternative is not necessarily more efficient nor easier to implement. As an example, a divide and conquer pattern may be implemented using a task farm pattern where the workers are able to compute all of the specific phases (divide, test base case, solve base case, conquer) and the tasks produced while dividing are routed back from collector to emitter for further processing. The implementation of the divide and conquer pattern in FastFlow follows a similar strategy, but implements a number of optimizations such that a high level of efficiency is achieved in the implementation of a wide range of divide and conquer kernels and applications. The divide and conquer pattern therefore supports the implementation of a variety of parallel algorithms, ranging from non-data intensive algorithms to data intensive ones including sorting of large datasets or computation bound algorithms such as Strassen dense matrix multiplication. The pool pattern is particularly suited to evolutionary computing applications. It has been demonstrated to be useful in the exploration of complex space search algorithms, in the implementation of genetic algorithm based applications and in the implementation of iterative algorithms modelling approximation of complex solutions through progressive refinement. Finally, the key and windowed stream farm patterns have been shown to efficiently support financial applications processing data intensive stream of records and may naturally support those applications, e.g. from social networks, processing large sets of records available across single or multiple data streams to infer more structured information about the stream contents and behaviour.

TOADD: mapreduce

Core patterns Core patterns, alone or in composition, may be used to support those applications and kernels where embarrassingly parallel, staged (i.e. pipelined) or iterative parallel components are present. They have been included in the *RePhrase* pattern set but have already been present in several other programming frameworks (including [20,21,22,30,15,4]). The class of applications supported includes numerical applications, video processing applications, soft computing and AI applications up to learning and massive data processing applications.

Building block patterns The building block patterns included in the *RePhrase* pattern set must be used in composition to model the parallel patterns needed for data intensive applications that are not supported by either high level or core patterns. As such, they provide support to the implementation of any generic streaming network built out of an arbitrary number of data sources and data drains with an arbitrary number of processing nodes, transforming, filtering, merging, splitting and collapsing stream (portions). The set of patterns in the building block class have clearly been inspired by the kind of computations usually supported by programming frameworks such as Storm and Flink [10,7]. A data processing network such as that in Fig. 2 may be easily built by combining our building block and core patterns.

4.2 Programming effort

The programming effort required to implement data intensive parallel applications varies according to the application at hand and to the targeted parallel programming framework.

A mapreduce application programmed on top of Hadoop simply requires specification of the code for the map and the reduce "functions" along with some input data and the Hadoop framework turns these minimal inputs into an efficient, running application. However, if you wish to program yourself the mapreduce pattern using MPI, the amount of code required increases exponentially. On the other hand, if you wish to program a non-mapreduce application

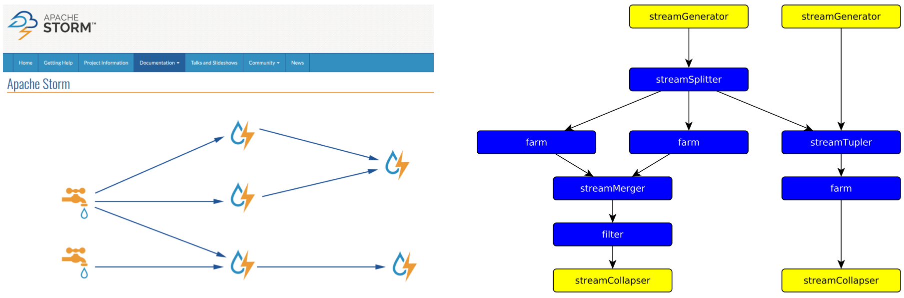


Fig. 2: Sample data streaming network with *RePhrase* building block and core patterns: Apache Storm website picture (left, from `http://storm.apache.org/`) and sample RePhrase building block outline (right).

on top of Hadoop you may end up concluding that either this is not possible at all or the effort needed to mutate the mapreduce pattern into the actual pattern to be implemented is too large.

The programming effort required of the programmer using the *RePhrase* patterns is similar to that required of the Hadoop programmer developing a mapreduce application. *RePhrase* patterns are provided using plain C++ programming abstractions (higher order functions or classes) that may be instantiated with suitable functional and non functional parameters to implement the particular instance of the pattern required by the application programmer.

We discuss two simple examples here, relative to the usage of FastFlow patterns[3] and of patterns wrapped in the *RePhrase* specific GrPPI, a C++11 specific pattern interface designed within *RePhrase* to provide a target framework-agnostic way of expressing patterns.

- In FastFlow a pipeline pattern with sequential stages may be expressed by declaring a `ff_pipeline` object and then adding stages (lambdas or `ff_node_t` objects wrapping a `function` object). Once the object has been declared its execution may be triggered by invoking the `run_and_wait_end()` method of the `ff_pipeline` object (see code snippet in Fig. 3 (top)).
- Using GrPPI a pipeline pattern may be declared and run as a function with parameters that denote the kind of target parallel programming framework and the stages to be used in the pipeline (this is a variable length list of `callable` object) (see code snippet in Fig. 3 (bottom)).

Overall, the extended pattern set provides significant support for the parallel applications programmer in the implementation process by making available the patterns as ready to use objects and functions that the programmer may freely and immediately use to program the parallel part(s) of his/her application. However, it is worth pointing out that the *RePhrase* methodology (as depicted in Fig. 1) aims at introducing patterns into applications by means of

[3] FastFlow is one of the target backends considered within *RePhrase*

```
//
// FASTFLOW two stage pipeline
//
auto f1 = [](T1 * x)->(T2*) { ... };
auto f2 = [](T2 * x) { ... };

struct Stage1 : ff_node_t<T1,T2> {
  T2 * svc(T1 * x) { return (f1(x)); }
};

struct Stage2 : ff_node_t<T2> {
   void * svc(T2 * x) { return(f2(x)); }
};

int main(int argc, char * argv[]) {
  ...
  ff_Pipe pipe(Stage1,Stage2);
  ...
  pipe.run_and_wait_end();
  ...
}
```

```
//
// GrPPI two stage pipeline
//
auto f1 = [](T1  x) { ... };
auto f2 = [](T2  x) { ... };

int main(int argc, char * argv[]) {
  ...
  parallel_execution_ff ff_mode{};
  pipeline(ff_mode, f1, f2);
  ...
}
```

Fig. 3: Pipeline sample code snippets in FastFlow e GrPPI

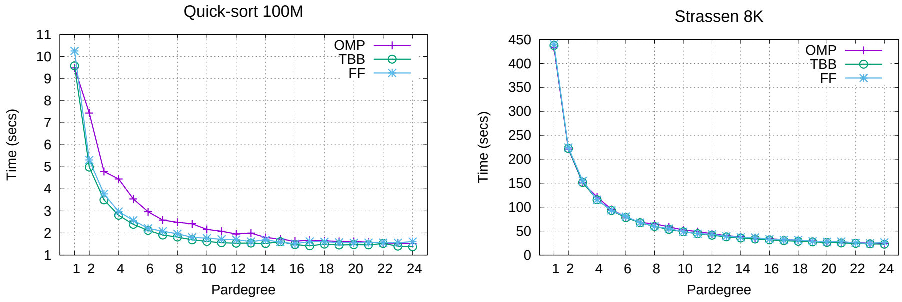
the *RePhrase* refactoring tools. Places where patterns may be introduced are identified by using the *pattern discovery* tool which spots those locations and those portions of code that may be turned into parallel pattern instances.

Added

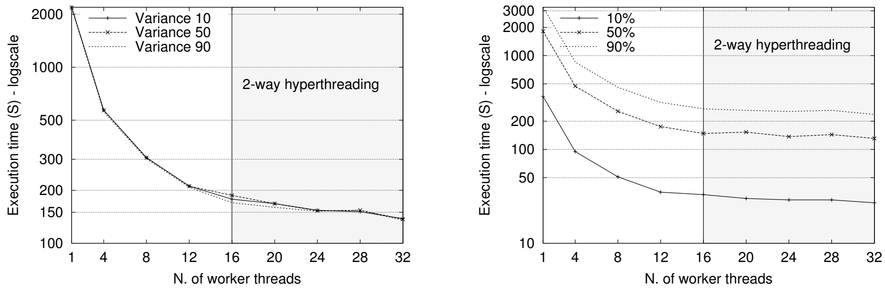
In terms of LOC (lines of code), the programming effort required to use the native FastFlow pattern interface is comparable to the one required by similar programming frameworks (e.g. Intel TBB [30]) but definetely FastFlow requires a larger number of lines of code with respect to pragma based programming frameoworks such as OpenMP. However, we must point out that the FastFlow programming interface provides much more patterns than OpenMP. For those natively supported in OpenMP–e.g. parallel for/map–LOC is definetely better (lower) in OpenMP, but those that are not natively supported in OpenMP require a comparable or even larger LOC in OpenMP compared to FastFlow (see Fig. 5c). A completely different perspective comes from the usage of GrPPI [18]. In this case the LOC count is balanced even when comparing RePhrase framework with pragma based frameworks such as OpenMP, due to the fact the GrPPI profitably leverages all those new features recently added to the C++ standard that *de facto* support functional style abstractions. It is worth pointing out that, while pragma based patterns require some intervention on the compiler toolchain, the RePhrase wrapper approach implemented in GrPPI works with (pre-compiled or source header only) libraries.

5 Experimental results

In this section we outline some experimental results obtained with the high level patterns in the *RePhrase* extended pattern set. We limit discussion to



(a) Performance of different recursive computations (quick-sort and Strassen's algorithm) implemented through the DAC pattern via different back-end runtimes. The plots show the completion time with different parallelism degrees.



(b) Random noise (*salt-and-pepper*) (c) Gaussian noise

(d) Completion time in stencil based denoiser application

Fig. 4: Experimental results related to the RePhrase pattern set (1)

high level patterns as we have not yet developed, within the project, applications which employ the core patterns.

5.1 Window-based Streaming Patterns

Data stream processing applications process unbounded data streams coming from a plurality of sensor devices. Input items received at high speed are usually accumulated by updating an internal state of the pattern (e.g., a sliding window containing the most recent data) and by applying a user-defined function periodically, e.g., at each window triggered according to the activation semantics (time-based, count-based or hybrid). When the input stream conveys data items belonging to different logical sub-streams, a natural parallelism can be exploited among the computations on windows of different sub-streams.

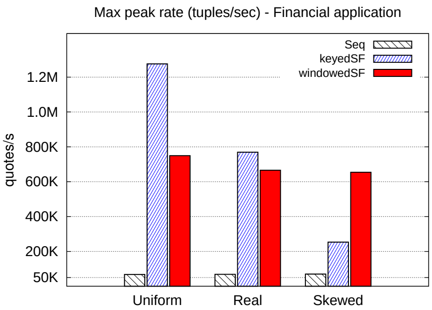
The `keyedSF` pattern has been adopted in our previous work [14] in order to parallelize a high-frequency trading application. The application is fed by a continuous stream of financial ticks that can be *trades*, i.e. closed transactions with a price, a stock symbol and a volume (number of stocks), and *quotes*, that is buy or sell proposals with a proposed price, a stock symbol and a volume. The goal of the application is to automatically discover trading opportunities by analyzing the market feeds in near real-time. The computation maintains a sliding window of the most recent data items of each stock symbol (sub-stream) and executes a continuous query at each new window activation. We used count-based windows of $1,000$ tuples with a refresh slide of 25 new data items. The query computes a least squares curve fitting using the well-known Levenberg-Marquardt algorithm.

From the performance viewpoint the scalability of this parallel pattern greatly depends on the frequency distribution of the sub-streams, because all the windows of the same sub-stream are computed sequentially. Several experiments were performed in order to evaluate the performance of this pattern under various conditions. Fig. 5a shows the result of an experiment performed on an Intel Ivy Bridge dual-socket multicore workstation featuring 24 cores. The figure depicts the maximum stream speed that the pattern is able to sustain without being a bottleneck by running the application with as many threads as the number of available cores. We also report the peak performance achieved with a single-threaded implementation of the whole application. While the scalability is almost ideal with a uniform probability distribution among stock symbols, in more realistic scenarios with a realistic skewness (*real*) and a heavy skewness (heavy), the scalability of the `keyedSF` pattern drops significantly due to load imbalance.

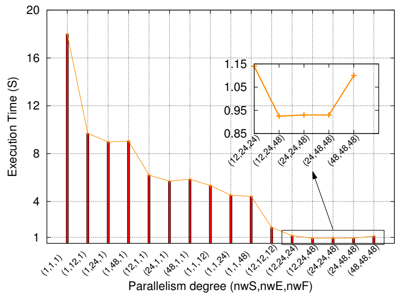
The figure also shows the performance achieved under the same execution conditions by an alternative implementation based on the `windowedSF` pattern. In this case the pattern exploits parallelism among windows within the same logical sub-stream by suitably scheduling data items to worker threads in such a way as to execute in parallel consecutive windows with the same stock symbol. The result is a pattern more sophisticated in its data distribution, i.e. an emitter thread is in charge of multicasting each data item to a subset of the workers. However, the performance and load balancing is not affected by the frequency of the sub-streams (the pattern works well also with one stock symbol in the extreme case). This behavior is evident in the figure, where the peak rate with this second solution is the best under a heavy skewness, while with the uniform distribution and in the real skewness case the `keyedSF` pattern is the winner owing to the more efficient point-to-point distribution of data items to the worker threads.

5.2 DAC

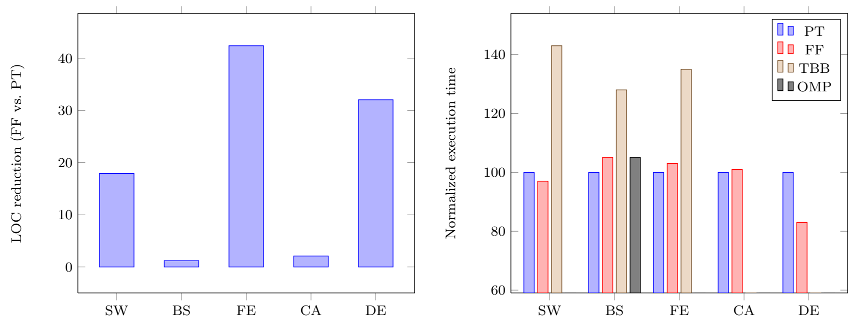
We implemented the Divide&Conquer parallel pattern in various backend environments, such that, while maintaining the same source code, the programmer can exploit the potential of different frameworks and target architectures. We proposed three different implementations for multicore architectures based on



(a) Maximum input speed sustained by the financial application with various patterns and a different skewness among substreams.



(b) FastFlow Pool pattern performances on Intel Sandy Bridge multicore with 24 cores 2-way Hyperthreading.



(c) FastFlow RePhrase pattern vs. standard programming frameworks: results on Parsec benchmarks. Line of code size reduction and cyclomatic count for Parsec benchmarks (left) and performance comparison on Sandy Bridge architectures (right). (SW = Swaptions, BS = Balcksholes, FE = Ferret, DE = Dedup, CA = Canneals) (FF = FastFlow, PT = Pthreads, TBB = Intel Thread Building Blocks, OMP = OpenMP)

Fig. 5: Experimental results related to the RePhrase pattern set (2)

OpenMP compiler annotations, Intel TBB and FastFlow parallel programming libraries. The experimental analysis, performed on a 24-core Intel server, showed that the reduced effort in programming does not come at the expense of significant performance penalties. The experimental study has been done by comparing the pattern-based solution with hand-made parallelizations using the same backend runtime. These results pave the way to further development of this work. First, the set of backend implementations can be further extended, including an MPI implementation for targeting distributed systems, and a CUDA/OpenCL-based implementation for GPUs. Second, we recognize that an important role in achieving good level of performance is played by the cutoff value, i.e. the point at which we stop the recursion and solve the problem sequentially to better exploit the cache hierarchy and/or limit the runtime support overhead. This value depends on the structure of the specific parallelized application and on the kind of platform used. As proposed in [19], using information from the application collected at runtime (without relying on any user hints), it is possible to automatically derive the cutoff technique that is best suited for the application.

5.3 Pool

In [1] we designed and implemented implementations of the different variants of the pool pattern in C++/FastFlow, as well as in Erlang/skel [32]. Both implementations have been used to run experiments on top of state-of-the-art shared memory multicore servers. A full set of experiments has been discussed assessing the features of the pool pattern as well as the efficiency and scalability of the pattern when used to implement various parallel applications. In particular, we have demonstrated that reasonable performances may be achieved with modest programming effort while noting that, in certain cases, manual, ad-hoc optimization of the parallel code taking into account the specific target architecture features may lead to further minor performance improvement. The typical performance figures achieved are exemplified in Fig. 5b. In this case we plot the completion times achieved in the execution of a synthetic benchmark when the number of processing elements (threads) used in the different phases of the pool pattern implementation vary. In particular, the triples (x, y, z) on the x-axis represent the number of threads used in the selection, evolution and filtering phases where the individuals submitted to evolution are selected from the whole population, their evolution is computed and the evolved individuals to be included back in the populations are selected, respectively.

5.4 Stencil

In [2] we discussed the FastFlow implementation of a loop-of-stencil-reduce pattern, targeting iterative data parallel computations on heterogeneous multicores. We showed that various iterative kernels can be easily and effectively parallelized by using the Loop-of-stencil-reduce on the available GPUs by exploiting the OpenCL capabilities of the FastFlow parallel framework. We focused on capturing stencil iteration as a pattern, and on its integration in the established FastFlow pattern framework. The pattern demonstrated to be quite efficient on modern multicore architectures. Fig. 4d shows the completion times achieved on a 24 core Sandy Bridge machine while executing a video denoiser application using the stencil pattern [3].

6 Conclusions

We discussed an extended parallel pattern set designed to support data intensive applications on heterogeneous architectures build of state-of-the-art shared memory multicores and GP-GPUs. We outlined the expressive power of the set, in terms of the range of applications that may be programmed using the patterns and in terms of the programming effort required to implement these applications as compared to the effort required when using more traditional parallel programming frameworks. Finally, we presented some existing experimental results relative to high level patterns in the pattern set demonstrating the effectiveness of the our approach.

Acknowledgements This work has been partially funded by the EU H2020-ICT-2014-1 project No. 644235 *RePhrase* "Refactoring Parallel Heterogeneous Resource-Aware Applications" (`http://www.reprhase-ict.eu`).

References

1. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Pool evolution: A parallel pattern for evolutionary and symbolic computing. *Int. J. Parallel Program.*, 44(3):531–551, June 2016.
2. M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016.
3. M. Aldinucci, G. Peretti Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Application*, 2015.
4. M. Aldinucci and M. Torquati. FastFlow home page, 2016. `http://calvados.di.unipi.it/fastflow`.
5. H. Andrade, B. Gedik, K.-L. Wu, and P. Yu. Scale-up strategies for processing high-rate data streams in system s. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1375–1378, March 2009.
6. H. C. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics.* Cambridge University Press, 2014.
7. Apache Flink. `http://flink.apache.org/`, 2016.
8. Apache Hadoop. `http://hadoop.apache.org/`, 2016.
9. Apache Spark. `http://spark.apache.org/`, 2016.
10. Apache Storm. `http://storm.apache.org/`, 2016.
11. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.
12. 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 symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.
13. M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati. P^3arsec: Towards parallel patterns benchmarking. In *Proceedings of the 32nd Annual ACM Symposium on Applied Computing - To appear*, SAC '17, New York, NY, USA, 2017. ACM.
14. M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. Data stream processing via code annotations. *The Journal of Supercomputing*, pages 1–15, 2016.
15. M. Danelutto and M. Torquati. Structured parallel programming with "core" fastflow. In V. Zsók, Z. Horváth, and L. Csató, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.
16. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
17. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
18. D. del Rio Astorga, M. F. Dolz, L. M. Sánchez, J. G. Blas, and J. D. García. A C++ generic parallel pattern interface for stream processing. In *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*, pages 74–87, 2016.

19. A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.
20. K. Emoto and K. Matsuzaki. An automatic fusion mechanism for variable-length list skeletons in sketo. *International Journal of Parallel Programming*, 42(4):546–563, 2014.
21. J. Enmyren and C. W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.
22. S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *IJHPCN*, 7(2):129–138, 2012.
23. IBM. Ibm infosphere stream. `http://www-03.ibm.com/software/products/en/ibm-streams`, 2016.
24. J. Korinth, D. de la Chevallerie, and A. Koch. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 195–198. IEEE Computer Society, 2015.
25. D. Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.
26. N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.
27. T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.
28. Microsoft. Task Parallel Library (TPL), 2016. `https://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx`.
29. L. Pappalardo, F. Simini, S. Rinzivillo, D. Pedreschi, F. Giannotti, and A.-L. Barabási. Returners and explorers dichotomy in human mobility. *Nature Communications*, 6:8166+, Sept. 2015.
30. J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
31. RePhrase. Report on defined use cases, 2015. RePhrase report D6.3.
32. A Streaming Process-based Skeleton Library for Erlang, 2015. `https://github.com/ParaPhrase/skel`.
33. M. TB and D. AS. The inevitable application of big data to health care. *JAMA*, 309(13):1351–1352, 2013.
34. A. Weiler, M. Grossniklaus, and M. H. Scholl. An evaluation of the run-time and task-based performance of event detection techniques for twitter. *Information Systems*, 2016.
35. T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
36. G. Wilson and R. Irvin. Assessing and comparing the usability of parallel programming systems, 1995. University of Toronto.
37. A. Wright. Big data meets big science. *Commun. ACM*, 57(7):13–15, July 2014.
38. S. Zhao, M. Chandrashekar, Y. Lee, and D. Medhi. Real-time network anomaly detection system using machine learning. In *11th International Conference on the Design of Reliable Communication Networks, DRCN 2015, Kansas City, MO, USA, March 24-27, 2015*, pages 267–270, 2015.