This is a postprint version of the following published document:

Challenging the abstraction penalty in parallel patterns libraries

Adding FastFlow support to GrPPI

J. Daniel Garcia · D. del Rio · M. Aldinucci · F. Tordini · M. Danelutto · G. Mencagli · M. Torquati

Abstract [1] In the last years, pattern-based programming has been recognized as a good practice for efficiently exploiting parallel hardware resources. Following this approach, multiple libraries have been designed for providing such high-level abstractions to ease the parallel programming. However, those libraries do not share a common interface. To pave the way, GrPPI has been designed for providing an intermediate abstraction layer between application developers and existing parallel programming frameworks like OpenMP, Intel TBB or ISO C++ threads. On the other hand, FastFlow has been adopted as an efficient object-based programming framework that may benefit from being supported as an additional GrPPI backend. However, the object-based approach presents some major challenges to be incorporated under the GrPPI type safe functional programming style. In this paper, we present the integration of FastFlow as a new GrPPI backend to demonstrate that structured parallel programming frameworks perfectly fit the GrPPI design. Additionally, we also demonstrate that GrPPI does not incur in additional overheads for providing its abstraction layer, and we study the programmability in terms of lines of code and cyclomatic complexity. In general, the presented work acts as reciprocal validation of both FastFlow (as an efficient, native structured par-

J.D. Garcia, D. del Rio
University Carlos III of Madrid E-mail: {josedaniel.garcia,david.rio}@uc3m.es

M. Aldinucci, F. Tordini
University of Torino E-mail: {aldinuc,tordini}@di.unito.it

M. Danelutto, G. Mencagli, M. Torquati
University of Pisa E-mail: {marcod,mencagli,torquati}@di.unipi.it

allel programming framework) and GrPPI (as an efficient abstraction layer on top of existing parallel programming frameworks).

1 Introduction

Parallel design patterns have been identified since years as the tool to be leveraged to overcome the difficulties and the scarce productivity typical of parallel programming [4]. Leveraging also on the availability of mature algorithmic skeleton [12] based programming frameworks [6,7,11,5] this lead to an increasing interest in the possibility of widely using parallel patterns in the development of parallel applications, targeting different kind of architectures, both homogeneous and heterogeneous (e.g. exploiting different kind of accelerators). As an example, different EU funded research projects in the FP7 and H2020 frameworks adopted some kind of algorithmic skeleton/parallel design pattern approach, including FP7 ParaPhrase, Excess [8] and Repara [15] projects, and more recently H2020 RePhrase [16] project.

Existing "structured" parallel programming environments provide the application programmer with high level tools for the development of parallel applications. In particular, through proper libraries or language extensions they provide ready-to-use programming abstractions modeling common parallel patterns that can be used, alone or in composition, to express the full parallel behavior of a given application. These abstractions encapsulate the more challenging aspects related to parallelism exploitation (synchronization, communication, load balancing, scheduling, etc.) and mainly relieve the application programmers from the burden to manage the target hardware dependent aspects related to the efficient implementation of parallel applications. It is worth pointing out that part of the *state-of-the-art* parallel programming frameworks also include some patterns. OpenMP [13] provides since the very beginning a *map* pattern through its `parallel for` *pragma*, Intel TBB [20] provides pipeline and farm (master worker) patterns, and Microsoft Parallel Pattern Library [14] also provides different patterns.

However, the lack of a) a common terminology to denote the different kind of patterns and b) the lack of a common, recognized and assessed API to use these patterns prevented the diffusion of the concept and a general acknowledgment of the related advantages.

GrPPI (the Generic Parallel Pattern Interface [10]) has been designed as a fully modern C++ compliant interface providing to the application programmers an easy way to parallelize existing code using well known, efficient stream and data parallel patterns. In its original version, GrPPI targeted C++11 threads to guarantee portability to any ISO C++11 compliant platform. Then, support for OpenMP and TBB has been added. The idea back to GrPPI design was exactly to overcome the two problems just mentioned and, possibly, to provide something that might be eventually included in a future

revision of the C++ standard [22]. The inclusion in the language standard would favour the possibility that different parallel programming frameworks providers invest in the implementation of suitable back-ends with the perspective of getting applications ported for free on their frameworks through a common API.

FastFlow [9] is a structured parallel programming framework–built on top of standard C++ and of the `pthreads` library–providing efficient implementation of both stream and data parallel patterns on top of shared memory architectures. It has been demonstrated to be particularly efficient in the implementation of fine grain computations, due to its extremely optimized communication mechanisms [2]. While simple usage of high-level patterns in FastFlow does not require sensibly different expertise than the one required using the same patterns through GrPPI, the FastFlow API does not reach GrPPI 's level of abstraction.

Here we discuss the integration of FastFlow as an additional backend for GrPPI. The integration has been challenging being FastFlow a framework already providing parallel patterns as objects of a header only library that application programmers and system programmers may use alone or in composition to model a variety of different parallel patterns. The main goal of this paper is to show how the introduction of an extra layer on top of FastFlow, namely GrPPI, does not necessarily imply a performance penalty and at the same time may simplify the task of expressing parallel applications.

The main contributions of the paper can be summarized as follows:

- We describe the implementation of the FastFlow back-end for GrPPI to demonstrate that GrPPI can be used as interface for object-based programming frameworks.
- We analyze the potential performance penalties introduced by the GrPPI abstraction layer.
- We evaluate the performance of a set of synthetic benchmarks using FastFlow and the different supported GrPPI backends.
- We compare the programmability in terms of lines of code and cyclomatic complexity between GrPPI and FastFlow and analyze the benefits of using both parallel programming frameworks.

The rest of the paper is structured as follows: Sec. 2 and Sec. 3 introduce FastFlow and GrPPI, respectively. Then Sec. 4 discusses the implementation of the FastFlow back-end in GrPPI. Eventually Sec. 5 shows the experimental results validating the porting and Sec. 6 draws conclusions.

2 FastFlow

FastFlow is a structured parallel programming framework, which is provided as a header only C++ library. FastFlow has been designed to target shared memory multi/many core architectures and at the moment being exploits pthreads as the main concurrency mechanism, backed up by an original, lock and wait

free communication queue that provides ultra efficient inter-thread communications.

The design of FastFlow is layered:

- A first layer implements the basic communication mechanisms.
- A second layer implements the "core" parallel patterns. These patterns represent notable and efficient parallel patterns exposing a low level interface suitable to use them both as "application" patterns and as "building block" patterns to be used to provide different patterns to the application programmer.
- A third layer provides "high level" patterns, with a specific, high level, full C++11 compliant interface, ready to be used within application code.

In either the "core" or the "high level" pattern layer, FastFlow provides i) stream parallel patterns (pipeline and farm, possibly with feedback); ii) data parallel patterns (parallel for, map, reduce, stencil); and iii) higher level patterns (divide&conquer, evolution pool, windowed stream processing patterns).

All these patterns are provided as objects the application (or the system) programmer may instantiate and compose to set up a parallel program which is eventually executed by issuing a specific member function call to the top level pattern. In other words, the declaration of the pattern expression modeling the parallel behavior of the application and the actual execution of the program happen at different places in the program.

Fig. 1 shows an excerpt of the code needed to run a pipeline with a farm stage inside in FastFlow.

3 Generic and reusable parallel pattern interface

In this section we introduce GrPPI, a generic and reusable parallel pattern interface for C++ applications [19]. This interface takes full advantage of modern C++ features, generic programming, and metaprogramming concepts to act as common interface between different programming models hiding away the complexity behind the use of concurrency mechanisms. Furthermore, the modularity of GrPPI allows to easily integrate new patterns, as well as to combine them to arrange more complex constructions. Thanks to this property, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relative small efforts, having as a result portable codes that can be executed on multiple platforms. Figure 1 depicts the general view of the GrPPI library. As shown, GrPPI acts as a layer between the user and the different programming models.

Basically, this library provides four main components in order to provide a unified interface for the supported frameworks: *i)* type traits, *ii)* pattern classes, *iii)* pattern interfaces and *iv)* execution policies.

Firstly, in order to accomplish the overloading of functions and to allow composition of different patterns, GrPPI provides a set of type traits and `constexpr` functions that are evaluated at compile time. In this sense, these functions can be used along with the `enable_if` construct from the C++ standard library to conditionally remove functions from compiler resolution. This way, only version of functions meeting specific compile-time requirements

Listing 1: Sample farm+pipeline code in FastFlow

```cpp
class Stage1: public ff_node_t<long,long> {
public:
  long* svc(long* task) {
    // ... here compute res out of task
    return res;
  }
};

class Stage2: public ff_node_t<long,long> {...};
class Emitter: public ff_node_t<noinput_t, long> {...};
class Collector: public ff_node_t<long,nooutput_t> {...};

int main(int argc, char* argv[]) {
  // ...
  // build the pattern expression ...
  std::vector<std::unique_ptr<ff_node> > workers;
  for(int i=0;i<nworkers;++i) {
    // build worker pipeline
    workers.push_back(make_unique< ff_Pipe<long,long> >(
        make_unique<Stage1>(), make_unique<Stage2>()));
  }
  Emitter E;
  Collector C;
  // build the farm with emitter and collector
  ff_Farm<noinput_t,nooutput_t> farm(std::move(workers), E, C);
  // Launch farm execution
  if (farm.run_and_wait_end()<0) {
    error("running farm\n");
    return -1;
  }
  return 0;
}
```
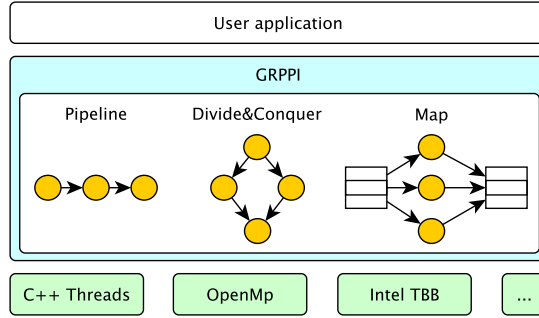

Fig. 1: GrPPI architecture.

on their parameters become available to the compiler. Consequently, this approach allows providing the same interface for different implementations.

On the other hand, GrPPI provides a set of classes that represent each of the supported patterns in order to allow compositions. These objects store references to the necessary functions and information (e.g. concurrency degree) related to the pattern configuration. Thus, they can be composed in order to express complex constructions that can not be represented by making use of a single pattern.

Focusing on the pattern interfaces, GrPPI provides a set of functions for each supported parallel pattern and offers two different alternatives for pattern

execution and composition. Both alternatives take the user functions that will be executed accordingly to the pattern and configuration parameters as function arguments, however, they differ in the first argument. The first version takes an execution policy in order to select the back-end which will be used for executing the pattern. On the other hand, the interface designed for pattern composition does not receive the execution policy, as it will use the one from its outer pattern. Instead of launching pattern execution, this version returns the pattern representation that will be used for the composition. Listing 2 shows both interface alternatives for the pipeline pattern.

Listing 2: Pipeline interfaces.

```
// Interface for pattern execution
template <typename Execution, typename ... Transformers>
void pipeline(const Execution & ex, Transformers && ... transform_ops);

// Interface for pattern composition
template <typename ... Transformers>
pipeline_t<Transformers...> pipeline(Transformers && ... transform_ops);
```

Finally, a key point of GrPPI is the ability to easily switch between different programming framework implementations for a given pattern. This is achieved by providing a set classes that encapsulate the actual pattern implementations in a specific framework. This way, by only changing the execution policy provided as first argument to the pattern call, the framework used underneath is selected accordingly. The current GrPPI version provides support for sequential, C++ threads, OpenMP and Intel TBB frameworks.

In this paper we extend the set of GrPPI execution policies in order to support FastFlow as a new back-end.

4 FastFlow under GrPPI

In this section we discuss how the FastFlow framework has been integrated into the GrPPI interface.

FastFlow natively supports high-level stream and data parallel programming patterns built on top of a *lock-free* run-time designed to boost performance on multi-core and many-cores architectures. This makes it perfectly suitable for being used as a back-end in the GrPPI interface. FastFlow provides different high-level patterns as well as different lower level patterns (parallel applications building blocks) that may be both used to implement quite efficient parallel applications. In this section, we will go through the process of implementing GrPPI's interface by means of FastFlow patterns, and we will provide an evaluation of the performance of the integration of FastFlow into GrPPI by comparing the pure FastFlow implementation against the wrapped one, in order to identify possible overheads.

4.1 Design

GrPPI provides parallel patterns to support both stream processing and data-intensive applications. These parallel patterns can be nested and composed

together, in order to model more complex behaviors. GrPPI currently supports different patterns, including: *data-parallel* patterns (map, reduce, mapreduce, stencil), *task-parallel* patterns (divide&conquer), and *stream-parallel* patterns (pipeline, farm, stream filter, stream reduction, stream iteration).

The parallel semantics associated to these patterns will not be discussed here. The interested reader may refer to the *RePhrase* deliverables available at `rephrase-ict.eu`, in particular in D2.1 [18] and D2.5 [17].

FastFlow natively supports most of the aforementioned skeletons, which are implemented by properly combining instances of the `ff_node` class, which is the main building block upon which every pattern can be designed. In particular, the `ff_node` class implements the abstraction of an independent concurrent activity receiving input tasks from an input queue, processing them and delivering the related results to the output queue. For instance, `ff_pipeline` and `ff_farm` are FastFlow's core patterns built on top of `ff_node`, which represent the most basic and generic stream parallel patterns and *de facto* orchestrate different kind of concurrent activities: stages, in the case of the pipeline pattern, or scheduler, workers and result collector, in the case of the farm pattern.

In its most recent fully C++11 compliant API FastFlow introduced enhanced features supporting a better control of the programming patterns typing system. Starting from its building blocks, the `ff_node_t<TypeIn, TypeOut>` becomes a typed abstract container for parallel activities, useful to enforce better type checking on streaming patterns, while among the high-level patterns `ff_Farm<TypeIn, TypeOut>` and `ff_Pipe<TypeIn, TypeOut>` provide typed Farms and Pipelines.

GrPPI takes full advantage of modern C++ features, making an extensive use of template meta-programming and generic programming concepts, that surely help code minimization and leverage automatic compile-time optimizations. The transformations applied to input data are described using C++ lambda functions, where the callable entities are forwarded via template function parameters, and the body of the lambda may be completely inlined inside a specific specialization of the template function that accepts it. Here the compiler will optimize the whole body of the function: in performance-critical applications, this solution certainly brings remarkable benefits.

Pattern interfaces are described as function templates, making them more flexible and usable with any data type. The extensive use of variadic templates allows an arbitrary number of data sets to be used by a pattern, and also facilitates handling an arbitrary number of stages/components as typical in a Pipeline pattern, by taking a sequence of callable entities passed as arguments to the function: template overloading guarantees that the right case is matched for every combination of Pipeline stages.

Each pattern function contains an `Execution` type, which represents the back-end that will eventually power pattern execution (see listing 3). For instance, it can be set to operate sequentially, or in parallel by means of one of the supported parallel execution policies. The FastFlow back-end is fully compliant with these techniques: it has been injected among the supported

execution type, and its declaration updated among the type traits and meta-functions that GrPPI applies to assert correctness of execution.

Listing 3: GrPPI's Pipeline interface

```
1  template <typename Execution, typename Generator, typename ... Transformers,
2           requires_execution_supported<Execution> = 0>
3  void pipeline(const Execution & ex, Generator && gen_op, Transformers && ... transf_ops)
4  {
5    ex.pipeline(std::forward<Generator>(gen_op), std::forward<Transformers>(transf_ops)...);
6  }
```

The programming framework to be used to support the execution of a high level GrPPI pattern is described specifying the proper `parallel_execution_*` object as a pattern parameter. Each `parallel_execution_*` class provides member functions for interacting with the underlying parallel framework and contains the actual definition of patterns. Listing 4 reports an excerpt of the `parallel_execution_ff` class, which reflects the way the other back-ends have actually been designed: by default the concurrency degree is set to the number of available physical cores, but it could be changed via proper access member functions or by using the specific constructor taking the concurrency degree as a parameter.

Listing 4: Parallel execution with FastFlow

```
1  class parallel_execution_ff {
2  public:
3    parallel_execution_ff() noexcept :
4      parallel_execution_ff{static_cast<int>(std::thread::hardware_concurrency()}
5    {}
6    parallel_execution_ff(int degree) noexcept :
7      concurrency_degree_{degree}
8    {}
9
10   // Members for access and modification
11
12   template <typename InputIterator, typename Identity, typename Combiner>
13   auto reduce(InputIterator first, std::size_t sequence_size,
14              Identity && identity, Combiner && combine_op) const;
15
16   // other patterns...
17
18   template <typename Generator, typename ... Transformers>
19   void pipeline(Generator && generate_op, Transformers && ... transform_op) const;
20 }
```

Stream parallel and data parallel patterns are declared and implemented within the execution type: adding FastFlow to GrPPI's data parallel patterns is a rather straightforward job, as they can all be built on top FastFlow's `ParallelFor` skeleton. Stream parallel patterns require a slightly bigger effort, as the existing Pipeline pattern has to be used to support the "pattern-matching" mechanism that drives template overloading, leading to the definition of the streaming patterns as composition of Pipelines with a minimum of 3 stages.

Concerning the Divide-and-Conquer (DAC) pattern, FastFlow' DAC pattern does not fully correspond to GrPPI's one, thus the porting requires some adjustments, due to differences in their programming interface. Further details will be given in the following section.

4.2 Implementation

Data Parallel Patterns

FastFlow provides the `ParallelFor` and `ParallelForReduce` skeletons, designed to efficiently exploit loop parallelism over input data collections. The FastFlow `ParallelFor` has obtained comparable, or even better performance results with respect to those achieved with well-known frameworks, such as OpenMP or Intel TBB [1,3], which were already part of GrPPI's back-end. As previously mentioned, a FastFlow back-end to data parallel patterns is rather straightforward, as they can all be built on top of FastFlow's `ParallelFor`, even though it requires some caution on pointers arithmetic.

Listing 5 shows the Map pattern built on top of the `ParallelFor`: the object is constructed with the proper concurrency degree, enabling a non-blocking run-time. The grain size also enables a dynamic scheduling of tasks to the threads, and chunks of no more than *grain* iterations at a time are computed by each thread, while the chunk of data is assigned to worker threads dynamically.

Listing 5: Map pattern for FastFlow back-end

```
template <typename ... InputIterators, typename OutputIterator, typename Transformer>
void parallel_execution_ff::map(std::tuple<InputIterators...> firsts,
        OutputIterator first_out, std::size_t sequence_size,
        Transformer transform_op) const
{
  ff::ParallelFor pf{concurrency_degree_, true};
  long step = 1;
  long grain = sequence_size/concurrency_degree_;

  pf.parallel_for_idx(0, sequence_size, step, grain,
      [&](const long start_, const long stop_, const int thid) {
        for (size_t it_ = start_; it_ < stop_; ++it_)
          *(first_out+it_) = apply_iterators_indexed(transform_op, firsts, it_);
      },
      concurrency_degree_);
}
```

The `ParallelFor` body traverses each chunk with indexes ranges, which are controlled using pointer arithmetic: the `apply_ iterators_indexed` is a meta-function provided by GrPPI that applies a callable entity (`transform_op`) to the values obtained from the iterators in a tuple by indexing. Results of the callable are written directly into the output data structure.

Similarly, the Reduce pattern is built on top of the `ParallelFor Reduce`: the reduction is executed in two phases: a partial reduction phase first runs in parallel, while the second phase reduces partial results in series, returning the final output. Iterations are scheduled to worker threads in blocks of at least *grain* iterations.

The Map/Reduce pattern computes a Map-like operation on the input data set, and then applies a Reduce operation on intermediate results[2]. It exploits

---

[2] This pattern is not the titled Google's MapReduce, which exploits *key-value* pairs to compute problems that can be parallelized by mapping a function over a given data set or stream of data, and then combining the results.

the intrinsic parallelism provided by the Map and the Reduce patterns, and the implementation is thus a combination of two phases: the Map phase stores its result in a temporary collection, which then undergoes the Reduce phase, and the final result is returned (see listing 6).

Listing 6: Map/Reduce pattern for FastFlow back-end

```
template <typename ... InputIterators, typename Identity, typename Transformer,
          typename Combiner>
auto parallel_execution_ff::map_reduce(std::tuple<InputIterators...> firsts,
    std::size_t sequence_size, Identity && identity,
    Transformer && transform_op, Combiner && combine_op) const
{
  std::vector<Identity> partials(sequence_size);
  map(firsts, partials.begin(), sequence_size, std::forward<Transformer>(transform_op));

  return reduce(partials.begin(), sequence_size,
      std::forward<Identity>(identity), std::forward<Combiner>(combine_op));
}
```

The Stencil pattern (see Listing 7) is a generalization of the Map pattern, except that it also performs transformations on a set of neighbors in a given coordinate of the input data set. Just like the Map pattern, it is implemented on top of the `ParallelFor`, with the difference that for each element, the result of the transformation is combined with the result of a function applied to the neighboring elements of the current one. The pattern can deal with multiple input data sets, which can be specified by means of variadic parameters in its declaration.

Listing 7: Stencil pattern for FastFlow back-end

```
template <typename ... InputIterators, typename OutputIterator,
          typename StencilTransformer, typename Neighbourhood>
void parallel_execution_ff::stencil(std::tuple<InputIterators...> firsts,
        OutputIterator first_out, std::size_t sequence_size,
        StencilTransformer&& transform_op, Neighbourhood&& neighbour_op) const
{
  ff::ParallelFor pf(concurrency_degree_, true);
  long step = 1;
  long grain = sequence_size/concurrency_degree_;

  pf.parallel_for_idx(0, sequence_size, step, grain,
    [&](long start, long stop, int thid) {
      const auto first_it = std::get<0>(firsts);
      for (size_t i = start; i < stop; ++i) {
        auto next_chunks = iterators_next(firsts, i);
        *std::next(first_out,i) = transform_op(std::next(first_it,ot),
            apply_increment(neighbour_op, next_chunks));
      }
    },
      concurrency_degree_);
}
```

Stream Parallel Patterns

In streaming parallelism, the *end-of-stream* is a special event used to manage termination that has to be orderly notified to the involved concurrent activities, so that they know the computation is terminated and they could start

the appropriate closing actions. For this reason, a tag or a marker that identifies the end of data stream is required. GrPPI uses the `optional<T>` data type, officially introduced in C++17 but already available in C++14 under the *experimental* namespace. This data type is basically a class template that manages an optional contained value, i.e. a value that may or may not be present. The empty `optional` is used to represent the EOS.

On the other hand, FastFlow propagates special tags through its actors to notify special events, such as the termination of the data stream, or items to be discarded. For instance, the end-of-stream is identified with the tag `(OutType *) EOS`. This divergence among both interfaces has been handled at the lower level, by wrapping the `ff_node_t` class at the base of patterns' implementation.

A key characteristic of FastFlow is the possibility of using their its own memory management mechanism. To allow this we provide customized versions of operators `new` and `delete` using a special tag `ff_arena`. The use of this internal allocator is highly optimized for *producer/consumer* scenarios, and helps maintaining a low memory footprint during program execution.

By exploiting template overloading, a `node_impl` has been defined for three basic situations in a Pipeline pattern: the first stage generating items, intermediate stages transforming items, and the las stage consuming items. Template `node_impl` is parametrized by input and output data types as well as the callable entity that transforms items.

An intermediate stage is modeled upon the `ff_node_t` class, and models a sequential stage that receives data from the input channel, executes its transformer on the input data and returns its result on the output channel. Both input and output data type need to be specified (see Listing 8).

Listing 8: `node_impl` for intermediate stage

```
1  template <typename Input, typename Output, typename Transformer>
2  class node_impl : public ff::ff_node_t<Input,Output> {
3  public:
4    node_impl(Transformer && transform_op) :
5        transform_op_{transform_op}
6    {}
7
8    Output * svc(Input * p_item) {
9      return new (ff_arena) Output{transform_op_(*p_item)};
10   }
11 private:
12   Transformer transform_op_;
13 };
```

Listing 9 reports the corresponding node for the first stage as an specialization of `node_impl` which does not have an input data type and send items into the data stream as long as its generator returns a value.

The end-of-stream case is adapted by returning the `EOS` marker when the generator produces and empty object which is GrPPI end-of-stream. Note that this wrapper extends the generic `ff_node` instead of the typed one, because FastFlow prohibits to have a node object typed with a `void` input type and a specialized output type: in this case, it falls back to the generic version.

Listing 9: `node_impl` for first stage

```cpp
template <typename Output, typename Generator>
class node_impl<void,Output,Generator> : public ff::ff_node {
public:
  node_impl(Generator && generate_op) :
      generate_op_{generate_op}
  {}

  void * svc(void *) {
    std::experimental::optional<Output> result{generate_op_()};
    if (result) { return new (ff_arena) Output{*result}; }
    else { return EOS; }
  }
private:
  Generator generate_op_;
};
```

A final stage does not return any data. It is a consumer stage, where input data is stored or printed, and nothing is further placed in the data stream. It is modeled upon the `ff_node_t` class, and no output data type needs to be specified (see listing 10): if an input value exists, it is consumed and then destroyed (such that memory leaks are avoided), additionally FastFlow's GO‑ON marker is returned.

Listing 10: `node_impl` for final stage

```cpp
template <typename Input, typename Consumer>
class node_impl<Input,void,Consumer> : public ff::ff_node_t<Input,void> {
public:
  node_impl(Consumer && consume_op) :
      consume_op_{consume_op}
  {}

  void * svc(Input * p_item) {
    consume_op_(*p_item);
    operator delete(p_item, ff_arena);
    return GO_ON;
  }
private:
  Consumer consume_op_;
};
```

We built the back-end for stream parallel patterns using these building blocks. By exploiting template overloading, a Pipeline is constructed out of the stages passed to the variadic parameter in the Pipeline interface (see listing 3). Except for the very first stage, which must be the stream generator[3], other stages can be any intermediate sequential stage, or another streaming pattern that embeds a parallel logic (e.g., a Farm pattern). For instance, an intermediate stage can be a nested Pipeline as well.

At the base of this mechanism there is FastFlow's Pipeline pattern. By wrapping the `ff_pipeline` class, pattern matching on function templates allows to add stages to the Pipeline in the same order as they have been passed to the variadic parameter in the interface. The only requirement is that each stage must subclass one of the `ff_node` or `ff_node_t` classes, just like the `node_impl` does, as well as FastFlow's Farm pattern. Once the pipeline is constructed, the execution is triggered calling the `run_and_wait_end()` function,

---

[3] GrPPI assumes all stream parallel comutations are pipelines, whose first stage acts as stream generators and the last stage acts as stream absorber. In both cases, the stages may anyway implement some kind of computation on the generated/absorbed stream items.

which starts the computation and awaits its termination in a cooperative way (see listing 11).

Listing 11: Construction of the Pipeline

```
template <typename Generator, typename ... Transformers>
void parallel_execution_ff::pipeline(Generator && generate_op,
    Transformers && ... transform_ops) const
{
  detail_ff::pipeline_impl pipe{concurrency_degree_, ordering_,
      std::forward<Generator>(generate_op),
      std::forward<Transformers>(transform_ops)...};
  pipe.setFixedSize(false);
  pipe.run_and_wait_end();
}

class pipeline_impl : public ff::ff_pipeline {
public:
  template <typename Generator, typename ... Transformers>
  pipeline_impl(int nworkers, bool ordered, Generator && gen,
      Transformers && ... transform_ops)
  :
    nworkers_{nworkers}, ordered_{ordered}, nodes_{}
  {
    // Type deduction...
    auto first_stage = std::make_unique<node_type>(std::forward<Generator>(gen_op));
    add_node(std::move(first_stage));
    add_stages<generator_value_type>(std::forward<Transformers>(transform_ops)...);
  }
```

In order to match the nesting of a stream parallel pattern into the Pipeline, first proper type checking is performed at compile time via template type matching, and then the pattern is constructed and added to the Pipeline. In listing 11 there are two functions that insert stages into the underlying pipeline object: `add_node` actually adds the stage, while `add_stages` forwards additional stages passed via the variadic parameter, and recurs over the template functions until it finds the right match.

Listing 12 shows the creation of a Farm stage, which is added to the Pipeline before subsequent stages: after type deduction is performed, the actual workers of the Farm are constructed. The workers will be responsible for actually applying Farm's transformation over the input data stream, whose callable is captured at compile time by the `FarmTransformer` parameter. The workers are then added to an ordered[4] Farm (`ff_OFarm`), which also adds default emitter and collector to the Farm. The new pattern is added to the Pipeline, while subsequent stages will be recursively added by matching the proper overloading.

Stream reduce and Stream filter are basically Farm patterns, and can easily be nested as Pipeline stages. Due to their semantics, they cannot be implemented as generic Farms, where workers carry on their job and return results, but need to be properly tuned.

The Stream reduce pattern aims at collapsing items appearing on the input stream, and then delivers these results to the output stream. This computation

---

[4] `FastFlow` provides two different Farm patterns: the *ordered* farm pattern preserves the input/output data stream ordering; the normal farm does not preserve it, for all those computations that, as an example, directly store results in shared memory and the order of the write/updates doesn't matter.

Listing 12: Adding a Farm stage to the Pipeline

```
1   template <typename Input, typename FarmTransformer, template <typename> class Farm,
2            typename ... OtherTransformers,
3            requires_farm<Farm<FarmTransformer>> = 0>
4   auto add_stages( Farm<FarmTransformer> && farm_obj,
5          OtherTransformers && ... other_transform_ops)
6   {
7     std::vector<std::unique_ptr<ff::ff_node>> workers;
8     for(int i=0; i<nworkers_; ++i) {
9       workers.push_back(std::make_unique<worker_type>(
10          std::forward<Farm<FarmTransformer>>(farm_obj)));
11    }
12
13    using node_type = ff::ff_OFarm<Input,output_type>;
14    auto p_farm = std::make_unique<node_type>(std::move(workers));
15    add_node(std::move(p_farm));
16    add_stages<output_type>(std::forward<OtherTransformers>(other_transform_ops)...);
17  }
```

is applied to a subset of the input stream called *window*, where each window is processed independently. In order to accommodate such behavior, a simple Windowed Farm is created, where the emitter is responsible for buffering items coming from the stream, and dispatching them in batch to the workers, as soon as the required window size is reached.

A Windowed Farm may have an *offset* parameter, which may create overlapping regions among two windows. This aspect is also managed by the emitter, that drops items or keeps duplicates, according to what scenario the offset size generates (depending on whether the offset is smaller, equal or greater than the window size). Again, in constructing the Windowed Farm, care has been taken for properly dealing with optional values, which are used to denote the end-of-stream event.

The Stream filter pattern applies a filter over the items in the input stream, such that only those items satisfying the filter pass to the output stream. The filter is computed in parallel using a FastFlow Farm. Despite rather straightforward, an *ad hoc* Farm has been created to reproduce this behavior in FastFlow, to properly deal with presence/absence of output item delivery by farm workers depending on the evaluation of the filter predicate on the corresponding input items.

The Stream iteration pattern applies a transformation to a data item in a loop fashion, until a given condition is satisfied. When the condition is met, the result is sent to the output stream. This pattern has been implemented again using an specialization of `ff_node_t`.

Task Parallel Patterns

GrPPI provides a Divide-and-Conquer (DAC) pattern, which applies a multi-branched recursion to break down a problem into several sub-problems, until the base case is reached: at this point distinct sub-problems can be solved in parallel and their solutions merged to form the definitive one. GrPPI comes with two interfaces for calling the DAC pattern, where one of them allows to

explicitly pass the Predicate function that leads the divide phase, rather than implicitly use it in the Divide function body.

FastFlow provides a DAC pattern, whose interface requires the Predicate condition for the base case to be explicitly passed as a pure function: this perfectly matches GrPPI, thus the implementation is rather easy and only requires proper deduction of the output type.

5 Experimental evaluation

In this section, we perform an experimental evaluation of the Map, Reduce, Stencil and Farm in order to compare the patterns' scalability among the different GrPPI backends, the new FastFlow backend and using directly the FastFlow programming framework. Additionally, we have evaluated the programmability in terms of lines of code (LOC) and cyclomatic complexity.

The evaluation has been carried on a machine equipped with two Intel Xeon Ivy Bridge E-2595 with a total of 24 cores running at 2.40 Ghz, 03 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LST with the kernel 3.13.0-57. As for the C++ compiler we have used the GCC 6.3.0.

5.1 Performance evaluation

To evaluate the performance and programmability, we have designed a set of benchmarks to measure some of the patterns previously described: *1)* Map, *2)* Reduce, *3)* Stencil and *4)* Farm. In this section, we present the performance evaluation for the different supported GrPPI backends as well as for direct use of FastFlow. The evaluation allows to analyze the potential overheads introduced by the extra abstraction layer on top of FastFlow. Additionally, we show how GrPPI can be used to compare performance among different backends.

Map benchmark: This benchmark has been used to compute a *daxpy* operation. That is, it computes the operation: $y = \alpha \cdot x + y$, where x and y are vectors of floating point numbers, and α is a random floating point scalar. The result overwrites the initial values of vector y. Figure 2 compares execution times using GrPPI against a direct FastFlow implementation. As observed, the execution times scale similarly independently of the framework used underneath.

Reduce benchmark: This benchmark stresses the pattern by computing the addition of a sequence of natural numbers. Figure 3 shows the performance achieved by the different backends with respect to the FastFlow implementation. As can be observed, similarly to the Map pattern benchmark, no significant difference among different backends can be identified. However, in this case, we notice a slight performance degradation when using more than 13 cores. This is due to the inherent overhead related to the data sharing between the two NUMA nodes in which a thread that accesses memory on a remote NUMA node has increased transfer costs.
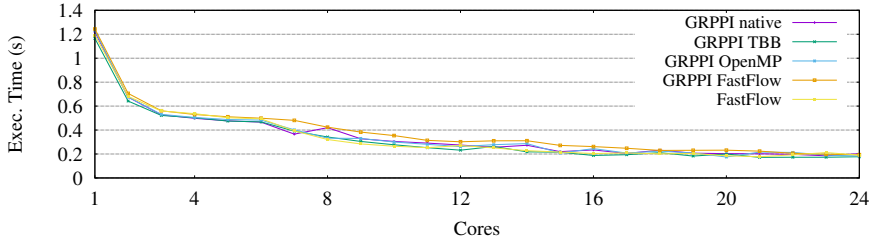
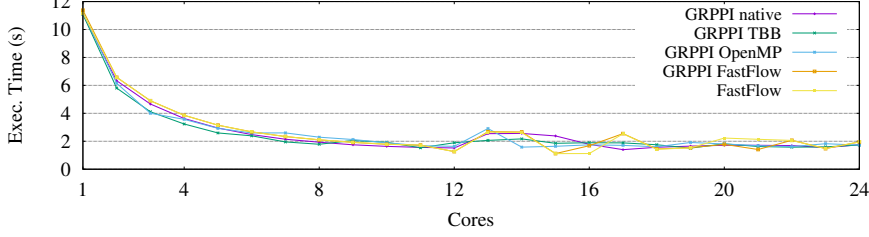Fig. 2: Execution time of the Map benchmark with respect to the number of cores.

Fig. 3: Execution time of the Reduce benchmark with respect to the number of cores.

Stencil benchmark: This benchmark performs a stencil computation over two one-dimensional vectors of the same size (v_1, v_2). The stencil function takes into account the two preceding and the two following neighbors of each item in each vector, and add their values to each item of the first vector. As observed in Figure 4, the performance is almost the same on all supported backends. Similar to the previous experiment, we notice the NUMA effect when using more than 12 cores in all backends except for the FastFlow implementation. This shows a better handling by FastFlow of NUMA platforms.
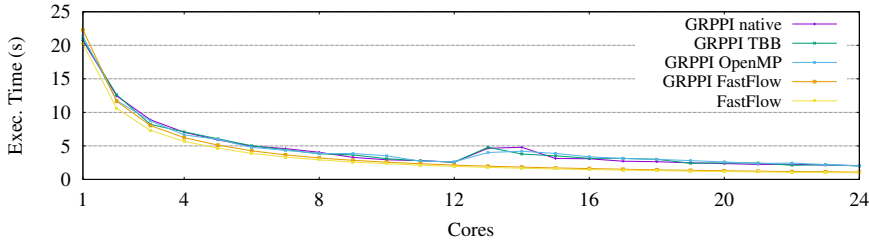

Fig. 4: Execution time of the Stencil benchmark with respect to the number of cores.

Farm benchmark: This benchmark executes some calculation on a stream of natural numbers: for each item in the input stream, workers perform repeated math operations on the input item, and return the result of their computation. As explained above, GrPPI builds all stream parallel patterns as a composition

of the Pipeline pattern: for example, a Farm is a three-stages pipeline where the middle stage is a Farm pattern that exploits as many worker threads as the concurrency degree defines. In this comparison, the Farm pattern used with the pure FastFlow benchmark is actually a Pipeline+Farm composition, as reported in listing 13.

Listing 13: Farm patterns

```
1  // Pure FastFlow version
2  void farm_bench_ff(int cores) {
3    ff::ff_Pipe<> pipe{make_unique<generator>(stream_len),
4      make_unique<ff::ff_OFarm<long>>(worker_func, num_workers), make_unique<Collector>()};
5    pipe.run_and_wait_end();
6  }
7
8  // GRPPI version
9  void farm_bench_grppi(const dynamic_execution & e, int cores) {
10   grppi::pipeline(e,
11     [&]() -> experimental::optional<long> {
12       if (idx_in < v.size()) {
13         idx_in++;
14         return v[idx_in-1];
15       }
16       else return {};
17     },
18     grppi::farm(cores, [&](long x) { return do_work(x); }),
19     [&](long x) { output += x; }
20   );
21 }
```

Figure 5 shows the execution time of the Farm benchmark with respect to the number of cores using the different backends. As observed, the TBB backend outperforms the other backends since TBB employs a completely different approach for implementing the pipeline. Intel TBB follows a task-based paradigm with work-stealing. In contrast, the rest of GrPPI backends and FastFlow use dedicated threads as for the generator and collector tasks.
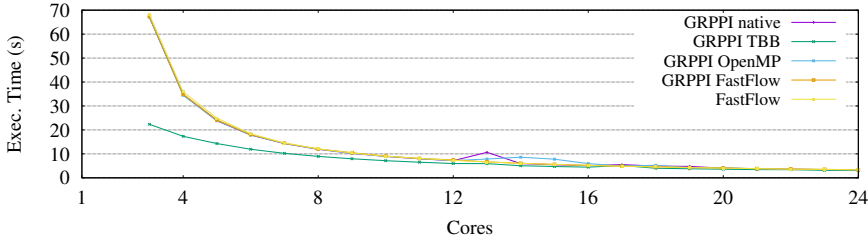

Fig. 5: Execution time of the Farm benchmark with respect to the number of cores.

In general, the GrPPI interface does not introduce additional overheads and allows to easily compare the performance by the different backends. This way, GrPPI eases the evaluation and selection of the most adequate backend with a single codebase at the application level and without needing to perform modifications when migrating from one backend to a different one.

5.2 Programmability evaluation

In this section we study the number of lines of code (LOC) and the average cyclomatic complexity number (CCN) of the GrPPI and FastFlow benchmark implementations. Table 1 shows the number of LOCs and average CCNs as programmability metrics calculated by the `lizard` tool [21]. As observed, since both FastFlow and GrPPI provide high-level interfaces, the number of required LOC and CCN are quite similar. The main difference among both interfaces is that FastFlow is targeted to expert parallel programmers providing more tuning options, while GrPPI provides a more simplified and user-friendly interface aimed to a wider range of application developers, while providing less fine-tuning options. Additionally, the GrPPI implementation supports multiple backends and allows to easily switch among them.

Benchmark	GrPPI		FastFlow	
	LOC	CCN	LOC	CCN
Map	63	1.4	65	1.4
Reduce	54	1.3	65	1.4
Stencil	76	2.2	73	2.2
Farm	66	1.4	80	1.3

Table 1: Lines of code and cyclomatic complexity of the benchmarks.

6 Conclusions

In this work, we have presented the design and implementation of the new FastFlow back-end for the GrPPI API supporting full pattern-based parallel programming in C++. The experimental evaluation demonstrates that the abstraction layer introduced by the GrPPI interface does not introduce additional overheads and performs equally to using FastFlow directly.

Additionally, we have analyzed the needed programming efforts for implementing a GrPPI application with respect to using an already existing framework. As we have seen, both approaches provide high-level interfaces that simplify expressing parallel programs. However, the FastFlow interface has been designed targeting expert parallel programmers and provide a way for fine-tuning the application, while GrPPI provides a more functional programming style interface aimed for a wider range of application developers.

References

1. Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Peretti Pezzi, G., Torquati, M.: The loop-of-stencil-reduce paradigm. In: Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara), pp. 172–177. IEEE, Helsinki, Finland (2015)

2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings, pp. 662–673. Springer (2012)
3. Aldinucci, M., Peretti Pezzi, G., Drocco, M., Spampinato, C., Torquati, M.: Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. International Journal of High Performance Computing Application (2015)
4. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
5. Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. In: V. Zsók, Z. Horváth, L. Csató (eds.) Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)
6. Ernsting, S., Kuchen, H.: A scalable farm skeleton for hybrid parallel and distributed programming. Int. J. Parallel Program. **42**(6), 968–987 (2014)
7. Ernstsson, A., Li, L., Kessler, C.: Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming (2017)
8. Excess home page (2017). `http://www.excess-project.eu/`
9. FastFlow home page (2017). `http://calvados.di.unipi.it/`
10. GrPPI github (2017). `https://github.com/arcosuc3m/grppi`
11. Haidi, M., Gorlatch, S.: High-Level Programming for Many-Cores using C++14 and the STL. International Journal of Parallel Programming **46**, 23–41 (2018)
12. Kessler, C., Gorlatch, S., Enmyren, J., Dastgeer, U., Steuwer, M., Kegel, P.: Skeleton Programming for Portable Many-Core Computing. In: Programming multi-core and many-core computing systems. Wiley (2017)
13. OpenMP home page (2017). `http://www.openmp.org/`
14. Microsoft Parallel Pattern Library home page (2017). Https://msdn.microsoft.com/en-us/library/dd492418.aspx
15. Repara home page (2017). `http://repara-project.eu/`
16. Rephrase home page (2017). `rephrase-ict.eu`
17. RePhrase, W.: Advanced patterns (2015). `https://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d2-5.pdf`
18. RePhrase, W.: Report on the initial pattern set (2015). `https://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d2-1.pdf`
19. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience pp. n/a–n/a (2017)
20. TBB home page (2017). `https://www.threadingbuildingblocks.org/`
21. Terry Yin: Lizard: an Cyclomatic Complexity Analyzer Tool Online; accesed 10 November 2018
22. Wong, M., Garcia, J.D., Keryell, R.: Supporting Pipelines in C++. Working paper P1261R0, ISO/IEC JTC1/SC22/WG21 (2018)