Structured Parallel Programming with “core” FastFlow

Chapter · March 2015

DOI: 10.1007/978-3-319-15940-9_2

2 authors:

Marco Danelutto
Università di Pisa
289 PUBLICATIONS 2,711 CITATIONS
SEE PROFILE

Massimo Torquati
Università di Pisa
118 PUBLICATIONS 885 CITATIONS
SEE PROFILE

Structured Parallel Programming with "core" FastFlow

Marco Danelutto(✉) and Massimo Torquati

Department of Computer Science, University of Pisa, Pisa, Italy
marco.danelutto@unipi.it, torquati@di.unipi.it

Abstract. FastFlow is an open source, structured parallel programming framework originally conceived to support highly efficient stream parallel computation while targeting shared memory multi cores. Its efficiency mainly comes from the optimized implementation of the base communication mechanisms and from its layered design. FastFlow eventually provides the parallel applications programmers with a set of ready-to-use, parametric algorithmic skeletons modeling the most common parallelism exploitation patterns. The algorithmic skeleton provided by FastFlow may be freely nested to model more and more complex parallelism exploitation patterns. This tutorial describes the "core" FastFlow, that is the set of skeletons supported since version 1.0 in FastFlow, and outlines the recent advances aimed at (i) introducing new, higher level skeletons and (ii) targeting networked multi cores, possibly equipped with GPUs, in addition to single multi/many core processing elements.

1 Introduction

FastFlow is an algorithmic skeleton (see Fig. 1) programming environment developed and maintained by researchers at the Dept. of Computer Science of the Univ. of Pisa and Univ. of Torino [1]. A number of different papers and technical reports discuss the different features of this programming environment [3,11,16], the kind of results achieved while parallelizing different applications [4,12,14,15,22] and the usage of FastFlow as *software accelerator*, i.e. as a mechanisms suitable to exploit unused cores of a multi core architecture to speedup execution of sequential code [7,8]. This work represents instead a tutorial aimed at instructing programmers in the usage of the FastFlow skeletons and in the typical FastFlow programming techniques.

Therefore, after recalling the FastFlow design principles in Sect. 2, in Sect. 3 we describe the (trivial) installation procedure. Then, in Sects. 4 to 9 we introduce the main features of the FastFlow programming framework: how to implement a simple "hello world" program (Sect. 4), how to manage streams (Sect. 5), how to wrap sequential code (Sect. 6), how to use explicit sharing (Sect. 7) and how to use pipelines and farm (Sects. 8 and 9). Then Sect. 10 deals with FastFlow usage as software accelerator, Sect. 11 discusses how FastFlow skeletons

This work has been partially supported by FP7 STREP project "ParaPhrase" (www.paraphrase-ict.eu).

V. Zsók et al. (Eds.): CEFP 2013, LNCS 8606, pp. 29–75, 2015.
DOI: 10.1007/978-3-319-15940-9_2

Algorithmic skeletons have been introduced by M. Cole in late 88 [19]. According to this original work

> The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

Later on, in his algorithmic skeleton "manifesto" [20] this definition evolved as follows:

> many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic patterns of computation and interaction. For example, many diverse applications share the underlying control and data flow of the pipeline paradigm. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognize these to enhance performance.

Different research groups started working on the algorithmic skeleton concept and produced different programming frameworks providing the application programmers with algorithmic skeletons. The definition of algorithmic skeletons evolved and eventually a widely shared definition emerged stating that:

> An algorithmic skeleton is parametric, reusable and portable programming abstraction modeling a known, common and efficient parallelism exploitation pattern.

At the moment being, different frameworks exists that provide the application programmers with algorithmic skeletons. Usually, the frameworks provide stream parallel skeletons (pipeline, task farm), data parallel (map, reduce, scan, stencil, divide&conquer) and control parallel (loop, if-then-else) skeletons mostly as libraries to be linked with the application business code. Several programming frameworks are actively maintained, including Muesli `http://www.wi1.uni-muenster.de/pi/forschung/Skeletons/1.79/index.html`, Sketo `http://sketo.ipl-lab.org/`, OSL `http://traclifo.univ-orleans.fr/OSL/`, SKEPU `http://www.ida.liu.se/~chrke/skepu/`, FastFlow `http://calvados.di.unipi.it/fastflow`, Skandium `https://github.com/mleyton/Skandium`. A recent survey of algorithmic skeleton frameworks may be found in [28].

Fig. 1. Algorithmic skeletons

may be nested and Sect. 12 discusses how to use "cyclic" skeletons. Eventually Sect. 13 outlines the main FastFlow RTS accessory routines and Sect. 15 outlines the major improvements to the "core" FastFlow currently being designed and implemented (high level patterns, targeting heterogeneous and distributed architectures, refactoring parallel programs).

2 Design Principles

FastFlow[1] has been designed to provide programmers with efficient parallelism exploitation patterns suitable to implement (fine grain) stream parallel applications. In particular, FastFlow has been designed

- to promote high-level parallel programming, and in particular skeletal programming (i.e. pattern-based explicit parallel programming), and
- to promote efficient programming of applications for multi-core.

[1] See also the FastFlow home page at http://mc-fastflow.sourceforge.net.

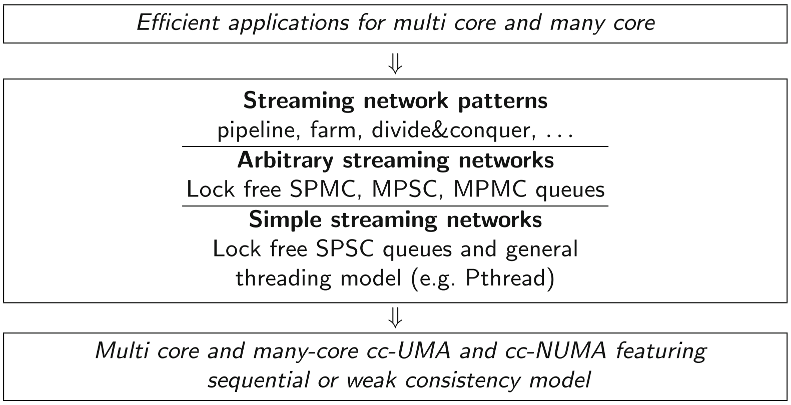

Efficient applications for multi core and many core
Streaming network patterns
pipeline, farm, divide&conquer, ...
Arbitrary streaming networks
Lock free SPMC, MPSC, MPMC queues
Simple streaming networks
Lock free SPSC queues and general threading model (e.g. Pthread)
Multi core and many-core cc-UMA and cc-NUMA featuring sequential or weak consistency model

Fig. 2. Layered FastFlow design

The whole programming framework has been incrementally developed according to a layered design on top of Pthread/C++ standard programming framework and targets shared memory multi core architectures (see Fig. 2).

A first layer, the **Simple streaming networks** layer, provides very efficient lock-free Single Producers Single Consumer (SPSC) queues on top of the Pthread standard threading model [9].

A second layer, the **Arbitrary streaming networks** layer, provides lock-free implementations for Single Producer Multiple Consumer (SPMC), Multiple Producer Single Consumer (MPSC) and Multiple Producer Multiple Consumer (MPMC) queues on top of the SPSC implemented in the first layer.

Eventually, the third layer, the **Streaming Networks Patterns** layer, provides common stream parallel patterns. The primitive patterns include pipeline and farms. Simple specialization of these patterns may be used to implement more complex patterns, such as divide and conquer, map and reduce patterns.

Parallel application programmers are assumed to use FastFlow directly exploiting the parallel patterns available in the Streaming Network Patterns level. In particular:

- defining sequential concurrent activities, by sub classing a proper FastFlow class, the `ff_node` class, and
- building complex stream parallel patterns by hierarchically composing sequential concurrent activities, pipeline patterns, farm patterns and their "specialized" versions implementing more complex parallel patterns.

The `ff_node` sequential concurrent activity abstraction provides suitable ways to define a sequential activity that (a) processes data items appearing on a single input channel and (b) delivers the related results onto a single output channel. Particular cases of `ff_nodes` may be simply implemented with no input channel or no output channel. The former is used to install a concurrent activity *generating* an output stream (e.g. from data items read from keyboard or from a disk file); the latter to install a concurrent activity *consuming* an input stream (e.g. to present results on a video or to store them on disk).

The pipeline pattern may be used to implement sequences of streaming networks $S_1 \rightarrow \ldots \rightarrow S_k$ with S_i receiving input from S_{i-1} and delivering outputs to S_{i+1}. The generic *stage* S_i may be either a sequential activity or another parallel pattern. When the pipeline is used standalone (i.e. not as component of another skeleton) S_1 must be a stream generator activity and S_k a stream consuming one.

The farm pattern models different embarrassingly (stream) parallel constructs. In its simplest form, it models a master/worker pattern with workers producing no stream data items. Rather the worker consolidate results directly in memory. More complex forms including either an emitter, or a collector of both an emitter and a collector implement more sophisticated patterns:

– by adding an emitter, the user may specify policies, different from the default round robin one, to schedule tasks from the farm input stream to the workers;
– by adding a collector, the user may use workers producing some output values, which are gathered and delivered to the farm output stream by the collector component. Different policies may be implemented on the collector to gather data from the worker and deliver them to the output stream.

In addition, a feedback channel may be added to a farm, moving output results back from the collector (or from the collection of workers in case no collector is specified) back to the emitter input channel.

Specialized versions of the farm may be used to implement more complex patterns, such as:

– divide and conquer, using a farm with feedback loop and proper stream items tagging (input tasks, subtask results, results)
– MISD (multiple instruction single data, that is something computing $f_1(x_i)$, $\ldots$, $f_k(x_i)$ out of each x_i appearing onto the input stream) pattern, using a farm with an emitter implementing a broadcast scheduling policy
– map, using an emitter partitioning an input collection and scheduling one partition per worker, and a collector gathering sub-partitions results from the workers and delivering a collection made out of all these results to the output stream.

Actually, in the new versions of FastFlow—built on top of the "core FastFlow" described here—the divide&conquer and the map skeletons have been implemented in proper classes and they are provided to the application programmers as *high level* skeletons. It is worth pointing out the different usage of the core FastFlow skeletons. On the one hand, when using plain pipeline and farms (with or without emitters and collectors) to model staged or embarrassingly parallel computations actually these programming abstractions may be classified as "skeletons" according to the traditional definition of algorithmic skeletons. When using specialized versions of the farm streaming network to implement different parallel patterns, instead, the core FastFlow farm should be considered a kind fo "pattern template", being used to build new patterns rather than to provided a primitive skeletons.

2.1 FastFlow Usage Models

Concerning the usage of FastFlow to support parallel application development on shared memory multi cores, the framework provides two abstractions of structured parallel computation:

– a *skeleton program abstraction* used to implement applications completely modeled according to the algorithmic skeleton concepts. When using this abstraction, the programmer writes a parallel application by providing the business logic code, wrapped into proper `ff_node` subclasses, a skeleton (composition) modeling the parallelism exploitation pattern of the application and a single command starting the skeleton computation and awaiting for its termination.
– an *accelerator abstraction* used to parallelize (and therefore accelerate) only some parts of an existing application. In this case, the programmer provides a skeleton (composition) which is run on the "spare" cores of the architecture and implements a parallel version of part of the business logic of the application, e.g. the one computing a given $f(x)$. The skeleton (composition) will have its own input and output channels. When an $f(x_j)$ has actually to be computed within the application, rather than writing proper code to call to the sequential f code, the programmer may insert code asynchronously "offloading" x_j to the accelerator skeleton. Later on, when the result of $f(x_j)$ is to be used, some code "reading" accelerator result may be used to retrieve the accelerator computed values.

This second abstraction fully implements the "minimal disruption" principle stated by Cole in his skeleton manifesto [20], as the programmer using the accelerator is only required to program a couple of `offload/get_result` primitives in place of the single $\ldots = f(x)$ function call statement (see Sect. 10).

3 Installation

Before entering the details of how FastFlow may be used to implement efficient stream parallel (and not only) programs on shared memory multi core architectures, let's have a look at how FastFlow may be installed[2]. FastFlow is provided as a set of header files. Therefore the installation process is trivial, as it only requires to download the last version of the FastFlow source code from SourceForge (http://sourceforge.net/projects/mc-fastflow/) by using `svn`:

```
svn co https://svn.code.sf.net/p/mc-fastflow/code/fastflow
```

Once the code has been downloaded, the directory containing the `ff` subdirectory with the FastFlow header files should be named in the `-I` flag of `g++`, such that the header files may be correctly found.

[2] We only detail instructions needed to install FastFlow on Linux/Unix/BSD machines here. A Windows port of FastFlow exist, that requires slightly different steps for the installation.

Take into account that, being FastFlow provided as a set of `.hpp` source files, the `-O3` switch is fundamental to obtain good performances. Compiling with no `-O3` compiler flag will lead to poor performances because the run-time code will not be optimized by the compiler. Also, remember that the correct compilation of FastFlow programs requires to link the `pthread` library (`-lpthread` flag). Sample `makefiles` are provided both within the `$FF_ROOT/tests` and the `$FF_ROOT/examples` directories in the source distribution.

4 Hello World in FastFlow

As all programming frameworks tutorials, we start with a `Hello world` code, that is a program simply printing a string onto the screen. We first discuss how to implement it sequentially, then we introduce a pipeline skeleton and we show how two stages may be used to print the components of a string in parallel.

In order to implement our sequential `Hello world` program, we use the following code, that uses a single stage pipeline:

```
 1 #include <iostream>
 2 #include <ff/pipeline.hpp>
 3
 4 using namespace ff;
 5
 6 class Stage1: public ff_node {
 7 public:
 8
 9     void * svc(void * task) {
10         std::cout << "Hello world" << std::endl;
11         return NULL;
12     }
13 };
14
15 int main(int argc, char * argv[]) {
16
17     ff_pipeline pipe;
18     pipe.add_stage(new Stage1());
19
20     if (pipe.run_and_wait_end()<0) {
21         error("running pipeline\n");
22         return -1;
23     }
24
25     return 0;
26 }
```

Line 2 includes all what's needed to compile a FastFlow program just using a pipeline pattern and line 4 instructs the compiler to resolve names looking (also) in the `ff` namespace. Lines 6 to 13 host the application business logic code, wrapped into a class sub classing `ff_node`. The `void * svc(void *)` method[3] wraps the body of the concurrent activity. It is called every time the concurrent activity is given a new data item from its input stream. The input stream data item is passed through the `svc` input `void *` parameter. The result of the single invocation of the concurrent activity body is passed back to the FastFlow runtime returning the `void *` result. In case a `NULL` is returned, the concurrent activity

[3] We use the term `svc` as a shortcut for "service".

actually terminates itself. The application main only hosts the code needed to setup the FastFlow streaming network and to start the skeleton (composition) computation: lines 17 and 18 declare a pipeline pattern (line 17) and insert a single stage (line 18) in the pipeline. Line 20 starts the computation of the skeleton program and awaits for skeleton computation termination. In case of errors the `run_and_wait_end()` call returns a negative number (according to the Unix/Linux syscall conventions).

When the program is run, the FastFlow RTS accomplishes to start the pipeline. In turn the first stage is started. When the stage `svc` returns a `NULL`, the FastFlow RTS immediately terminates it and the whole program terminates.

If we compile and run the program, we get the following output:

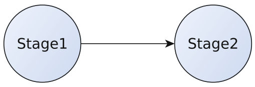
```
ffsrc$ g++ -I $FF_ROOT helloworldSimple.cpp -o hello -lpthread
ffsrc$ ./hello
Hello world
ffsrc$
```

There is nothing parallel here, however. The single pipeline stage is run just once and there is nothing else, from the programmer viewpoint, running in parallel.

A more interesting `Hello World` program may be implemented using a two stage pipeline where the first stage prints the “Hello” and the second one, after getting the results of the computation of the first one, prints “world”. In order to implement this behavior, we have to write two sequential concurrent activities and to use them as stages in a pipeline. Additionally, we have to send something out as a result from the first stage to the second stage. Let’s assume we just send the string with the word to be printed. The code may be written as follows:

```
#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

class Stage1: public ff_node {
public:

    void * svc(void * task) {
        std::cout << "Hello" << std::endl;
        char * p = new char[10];
        strcpy(p,"World");
        sleep(1);
        return ((void *)p);
    }
};

class Stage2: public ff_node {
public:

    void * svc(void * task) {
      std::cout << ((char *)task) << std::endl;
      delete [] (char*)task;
      return GO_ON;
    }
};

int main(int argc, char * argv[]) {

    ff_pipeline pipe;
    pipe.add_stage(new Stage1());
    pipe.add_stage(new Stage2());
33
34     if (pipe.run_and_wait_end()<0) {
35         error("running pipeline\n");
36         return -1;
37     }
38
39     return 0;
40 }
```

We define two sequential stages. The first one (lines 6–16) prints the “Hello” message on the screen, then allocates some memory buffer storing the “World” message in the buffer and sending the buffer pointer to the output stream (return command on line 14). The `sleep` on line 13 is here just for making more evident the FastFlow scheduling of concurrent activities. The second one (lines 18–26) just prints whatever it gets on the input stream (the data item stored after the `void * task` pointer of `svc` header on line 21), frees the allocated memory and then returns a `GO_ON` mark. This mark is interpreted by the FastFlow framework as something indicating: *I finished processing the current task, I give you no result to be delivered onto the output stream, but please keep me alive ready to receive another input task*. The `main` on lines 28–40 is almost identical to the one of the previous version but for the fact we add two stages to the pipeline pattern. Implicitly, this sets up a streaming network with `Stage1` connected by a stream to `Stage2`. Items delivered on the output stream by `Stage1` will be read on the input stream by `Stage2`. The concurrent activity graph is therefore:



If we compile and run the program, however, we get a kind of unexpected result:

```
ffsrc$ g++ -I $FF_ROOT hello2stages.cpp -o hello2stages -lpthread
ffsrc$ ./hello2stages
Hello
WorldHello

Hello World

Hello World

Hello World

^C
ffsrc$
```

First of all, the program keeps running printing an “Hello world” every second. We in fact terminate the execution through a CONTROL-C. Second, the initial sequence of strings is a little bit strange[4].

The “infinite run” is related to way FastFlow implements concurrent activities. Each `ff_node` is run as many times as the number of the input data items appearing onto the output stream, unless the `svc` method returns a `NULL`. Therefore, if the method returns either a task (pointer) to be delivered onto the concurrent activity output stream, or the `GO_ON` mark (no data output to the output

[4] And depending on the actual number of cores of your machine and on the kind of scheduler used in the operating system, the sequence may vary a little bit.

Value	Semantics
NULL	*Termination*: the svc method has been executed for the last time and will never be executed again
void * ptr	*Result*: the svc method is returning a result to be passed on onto the output stream. It will await to be called again with another input task
GO_ON	*Pass*: the svc method has been executed, no result has been generated (or in case it has been passed on through a ff_send_out but it expects to be called again with another input task)
FF_EOS	*End of stream*: this is the last value generated on the output stream. Other activities may follow, not leading to generation of more output data.

Fig. 3. Legal return values in ff_nodes

stream but continue execution), it is re-executed as soon as there is some input available (the different legal return values from an ff_node are summarized in Fig. 3). The first stage, which has no associated input stream, is re-executed up to the moment it terminates the svc with a NULL. In order to have the program terminating, we may use the following code for Stage1:

```
class Stage1: public ff_node {
public:
  void * svc(void * task) {
    if(task==NULL) {
      std::cout << "Hello " << std::endl;
      char * p = new char[10];
      strcpy(p,"World");
      sleep(1);
      first = false;
      return ((void *)p);
    }
    return NULL;
  }
};
```

If we compile and execute the program with this modified Stage1 stage, we'll get an output such as:

```
ffsrc$ g++ -I $FF_ROOT hello2terminate.cpp -o hello2terminate -lpthread
ffsrc$ ./hello2terminate
Hello
World
ffsrc$
```

that is the program terminates after a single run of the two stages. Now the question is: why the second stage terminated, although the svc method return value states that more work is to be done? The answer is in the stream semantics implemented by FastFlow. FastFlow streaming networks automatically manage end-of-streams. That is, as soon as an ff_node returns a NULL–implicitly declaring he wants to terminate its output stream, the information is propagated to the node consuming the output stream. This nodes will therefore also terminate execution–without actually executing its svc method–and the end of stream will be propagated onto its output stream, if any. This is why `Stage2` terminates immediately after the termination of `Stage1`.

The other problem, namely the appearing of the initial 2 “Hello” strings apparently related to just one “world” string is related to the fact that FastFlow does not guarantee any scheduling semantics of the `ff_node svc` executions. The first stage delivers a string to the second stage, then it is executed again and again. The `sleep` inserted in the first stage prevents to accumulate too much “hello” strings on the output stream delivered to the second stage. If we remove the `sleep` statement, in fact, the output is much more different: we will see on the input a large number of “hello” strings followed by another large number of “world” strings. This because the first stage is enabled to send on its output stream as much data items as of the capacity of the SPSC queue used to implement the stream between the two pipeline stages.

5 Generating a Stream

In order to achieve a better idea of how streams are managed within FastFlow, we slightly change our `HelloWorld` code in such a way the first stage in the pipeline produces on the output stream n integer data items and then terminates. The second stage prints a “world -i-” message upon receiving each i item onto the input stream.

Recalling the already discussed role of the return value of the `svc` method, a first version of this program may be implemented using the following code:

```
#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

class Stage1: public ff_node {
public:

  Stage1(int n):streamlen(n),current(0) {}

  void * svc(void * task) {
    if(current < streamlen) {
      current++;
      std::cout << "Hello number " << current << " " << std::endl;
      int * p = new int(current);
      sleep(1);
      return ((void *)p);
    }
    return NULL;
  }
private:
  int streamlen, current;
};

class Stage2: public ff_node {
public:

    void * svc(void * task) {
      int * i = (int *) task;
      std::cout << "World -" << *i << "- " << std::endl;
      delete task;
      return GO_ON;
    }
};

int main(int argc, char * argv[]) {

    ff_pipeline pipe;
    pipe.add_stage(new Stage1(atoi(argv[1])));
    pipe.add_stage(new Stage2());

    if (pipe.run_and_wait_end()<0) {
        error("running pipeline\n");
        return -1;
    }

    return 0;
}
```

The output we get is the following one:

```
ffsrc$ g++ -I$FF_ROOT helloStream.cpp -o helloStream -lpthread
ffsrc$ ./helloStream 5
Hello number 1
Hello number 2World - 1-

Hello number World -32 -

World -3- Hello number
4
Hello number 5World - 4-

World -5-
ffsrc$
```

However, there is another way we can use to generate the stream, which is a little bit more "programmatic". FastFlow makes available an `ff_send_out` method in the `ff_node` class, which can be used to direct a data item onto the concurrent activity output stream, without actually using the `svc return` way.

In this case, we could have written the `Stage1` code as follows:

```
class Stage1: public ff_node {
public:

  Stage1(int n):streamlen(n),current(0) {}

  void * svc(void * task) {
    while(current < streamlen) {
      current++;
      std::cout << "Hello number " << current << " " << std::endl;
      int * p = new int(current);
      sleep(1);
      ff_send_out(p);
    }
    return NULL;
  }
private:
  int streamlen, current;
};
```

In this case, the `Stage1` is run just once (as it immediately returns a `NULL`. However, during the single run the `svc while` loop delivers the intended data items on the output stream through the `ff_send_out` method. In case the sends fill up the SPSC queue used to implement the stream, the `ff_send_out` will block up to the moment `Stage2` consumes some items and consequently frees space in the SPSC buffers.

6 More on `ff_node`

The `ff_node` class actually defines three distinct virtual methods:

```
public:
    virtual void* svc(void * task) = 0;
    virtual int   svc_init() { return 0; };
    virtual void  svc_end()  {}
```

The first one is the one defining the behavior of the node while processing the input stream data items. The other two methods are automatically invoked once and for all by the FastFlow RTS when the concurrent activity represented by the node is started (`svc_init`) and right before it is terminated (`svc_end`).

These virtual methods may be overwritten in the user supplied `ff_node` subclasses to implement initialization code and finalization code, respectively. The `svc` method *must* be overwritten as it is defined as a pure virtual method.

We illustrate the usage of the two methods with another program, computing the Sieve of Eratosthenes. The sieve uses a number of stages in a pipeline. Each stage stores the first integer it got on the input stream. Then it cycles passing onto the output stream only the input stream items which are not multiple of the stored integer. An initial stage injects in the pipeline the sequence of integers starting at 2, up to n. Upon completion, each stage has stored a prime number.

We can implement the Eratosthenes sieve in FastFlow as follows:

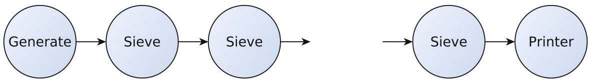
```
#include <iostream>
#include <ff/pipeline.hpp>

using namespace ff;

class Sieve: public ff_node {
public:

  Sieve():filter(0) {}

  void * svc(void * task) {
    unsigned int * t = (unsigned int *)task;

    if (filter == 0) {
      filter = *t;
      return GO_ON;
    } else {
      if(*t % filter == 0)
        return GO_ON;
      else
        return task;
    }
  }

  void svc_end() {
    std::cout << "Prime(" << filter << ")\n";
    return;
  }


private:
  int filter;
};

class Generate: public ff_node {
public:

    Generate(int n):streamlen(n),task(2) {
    std::cout << "Generate object created" << std::endl;
    return;
  }


  int svc_init() {
    std::cout << "Sieve started. Generating a stream of " << streamlen <<
      " elements, starting with " << task << std::endl;
    return 0;
  }

  void * svc(void * tt) {
    unsigned int * t = (unsigned int *)tt;

    if(task < streamlen) {
       int * xi = new int(task++);
       return xi;
    }
    return NULL;
  }
private:
  int streamlen;
  int task;
};

class Printer: public ff_node {

  int svc_init() {
    std::cout << "Printer started " << std::endl;
    first = 0;
  }

  void * svc(void *t) {
    int * xi = (int *) t;
    if (first == 0) {
      first = *xi;
    }
    return GO_ON;
  }

  void svc_end() {
    std::cout << "Sieve terminating, prime numbers found up to " << first
           << std::endl;
  }

private:
  int first;
};

int main(int argc, char * argv[]) {
  if (argc!=3) {
    std::cerr << "use: " << argv[0] << " nstages streamlen\n";
    return -1;
  }

  ff_pipeline pipe;
  int nstages = atoi(argv[1]);
  pipe.add_stage(new Generate(atoi(argv[2])));
  for(int j=0; j<nstages; j++)
    pipe.add_stage(new Sieve());
  pipe.add_stage(new Printer());

  ffTime(START_TIME);
  if (pipe.run_and_wait_end()<0) {
    error("running pipeline\n");
    return -1;
  }
```

```
106   ffTime(STOP_TIME);
107
108   std::cerr << "DONE, pipe  time= " << pipe.ffTime() << " (ms)\n";
109   std::cerr << "DONE, total time= " << ffTime(GET_TIME) << " (ms)\n";
110   pipe.ffStats(std::cerr);
111   return 0;
112 }
```

The `Generate` stage at line 35–62 generates the integer stream, from 2 up to a value taken from the command line parameters. It uses an `svc_init` just to point out when the concurrent activity is started. The creation of the object used to represent the concurrent activity is instead evidenced by the message printed in the constructor.

The `Sieve` stage (lines 6–33) defines the generic pipeline stage. This stores the initial value got from the input stream on lines 14–16 and then goes on passing the inputs not multiple of the stored values on lines 18–21. The `svc_end` method is executed right before terminating the concurrent activity and prints out the stored value, which happen to be the prime number found in that node.

The `Printer` stage is used as the last stage in the pipeline (the pipeline built at lines 94–99 in the program `main`) and just discards all the received values but the first one, which is kept to remember the point where we arrived storing prime numbers. It defines both an `svc_init` method (to print a message when the concurrent activity is started) and an `svc_end` method, which is used to print the first integer received, representing the upper bound (non included in) of the sequence of prime numbers discovered with the pipeline stages. The concurrent activity graph of the program is the following one:



The program output, when run with 7 `Sieve` stages on a stream from 2 to 30, is the following one:

```
ffsrc$ ./sieve 7 30
Generate object created
Printer started
Sieve started. Generating a stream of 30 elements, starting with 2
Prime(2)
Prime(3)
Prime(5)
Prime(7)
Prime(Prime(Sieve terminating, prime numbers found up to 1317)
)
19
Prime(11)
DONE, pipe  time= 0.275 (ms)
DONE, total time= 25.568 (ms)
FastFlow trace not enabled
ffsrc$
```

showing that the prime numbers up to 19 (excluded) has been found.

7 Managing Access to Shared Objects

Shared objects may be accessed within FastFlow programs using the classical pthread concurrency control mechanisms. The FastFlow program is actually a multithreaded code using the pthread library, in fact.

As an example, in order to avoid mixing parts of the strings output by different ff_nodes in a FastFlow program, we may wrap each output to the cout file descriptor with proper locks to a pthread_mutex variable. Therefore, after declaring the mutex

```
#include <pthread.h>
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

we may explicitly wrap each statement printing to the console in our source code with a lock and an unlock call to the mutex:

```
pthread_mutex_lock(&lock);
std::cout << ... << std::endl;
pthread_mutex_unlock(&lock);
```

In other cases, a viable alternative consists in wrapping the calls operating on the shared state in properly synchronized functions. As an example, if we need to update a shared state represented by an integer value by adding or subtracting values, we may use a function:

```
pthread_mutex_t state_lock = PTHREAD_MUTEX_INITIALIZER;
int state_var = 0;
int update(int amount) {
  pthread_mutex_lock(&state_lock);
  state_var += amount;
  pthread_mutex_lock(&state_lock);
  return(state_var);
}
```

or we may consider using new C++11 features such as the std::lock_guard supporting kind of more "automatic" locking, e.g.

```
int update(int amount) {
  std::lock_guard<std::mutex> lock(state_mutex);
  state_mutex++;
  state_var += amount;
  return(state_var);
}
```

This requires compiling with the proper -std=c++11 but the lock is automatically release at the end of the scope of the state_mutex variable.

It is worth pointing out that any additional synchronization mechanism inserted in a FastFlow program interacts with the primitive, optimized synchronizations of the framework. In particular, the FastFlow programmer must be aware that:

- FastFlow ensures correct access sequences to the shared object used to implement the streaming networks (the graph of concurrent activities), such as the SPSC queues used to implement the streams, for instance.
- FastFlow stream semantics guarantees correct sequencing of activation of the concurrent activities modeled through ff_nodes and connected through streams. The stream implementation actually ensures *pure data flow* semantics.

- passing data from concurrent activity A to concurrent activity B (e.g. when stage$_i$ `svc` method returns a `void *` pointer which is passed as input parameter to the `svc` method of stage$_{i+1}$ in a pipeline) conceptually transfers the *capability* to operate on the pointed data from one concurrent activity to the other one.
- any access to any user defined shared data structure must be protected with either the primitive mechanisms provided by FastFlow (see above) or the primitives provided within the `pthread` library.

but also that

- any synchronization mechanism added in the user code may impair the efficiency achieved by the FastFlow runtime in the orchestration of the parallel activities defined by the FastFlow skeletons used in the parallel application.

8 More Skeletons: The FastFlow Farm

In the previous sections, we used only pipeline skeletons in the sample code. Here we introduce the other primitive skeleton provided in FastFlow, namely the `farm` skeleton.

The simplest way to define a farm skeleton in FastFlow is by declaring a `farm` object and adding a vector of *worker* concurrent activities to the `farm`. An excerpt of the needed code is the following one:

```
#include <ff/farm.hpp>

using namespace ff;

int main(int argc, char * argv[]) {

  ...
  ff_farm<> myFarm;
  std::vector<ff_node *> w;
  for(int i=0;i<nworkers;++i)
    w.push_back(new Worker);
  myFarm.add_workers(w);
  ...
```

This code basically defines a farm with `nworkers` workers processing the data items appearing onto the farm input stream and delivering results onto the farm output stream. The default scheduling policy used to send input tasks to workers is the round robin one. Workers are implemented by the `ff_node` `Worker` objects. These objects may represent sequential concurrent activities as well as further skeletons, that is either pipeline or farm instances.

However, this farm may not be used alone. There is no way to provide an input stream to a FastFlow streaming network but having the first component in the network generating the stream. To this purpose, FastFlow supports two options:

- we can use the farm defined with a code similar to the one described above as the second stage of a pipeline whose first stage generates the input stream according to one of the techniques discussed in Sect. 5. This means we will use the farm writing a code such as:

```
 ...
 ff_pipeline myPipe;

 myPipe.add_stage(new GeneratorStage());
 myPipe.add_stage(myFarm);
```

– or we can provide an `emitter` and a `collector` to the farm, specialized in such a way they can be used to produce the input stream and consume the output stream of the farm, respectively, while inheriting the default scheduling and gathering policies.

The former case is simple. We only have to understand why adding the farm to the pipeline as a pipeline stage works. This will discussed in detail in Sect. 11. The latter case is simple as well, but we discuss it through some more code.

8.1 Farm with Emitter and Collector

First, let us see what kind of objects we have to build to provide the farm an `emitter` and a `collector`. Both `emitter` and `collector` must be supplied as `ff_node` subclass objects. If we implement the `emitter` just providing the `svc` method, the tasks delivered by the `svc` on the output stream—either using a `ff_send_out` or returning the proper pointer with the `svc return` statement—will be dispatched to the available workers according to the default round robin scheduling. An example of `emitter` node, generating the stream of integer tasks eventually processed by the farm `workers` is the following one:

```
 1 class Emitter: public ff_node {
 2 public:
 3     Emitter(int n):streamlen(n),task(0) {}
 4
 5     void * svc(void *) {
 6         sleep(1);
 7         task++;
 8         int * t = new int(task);
 9         if (task<streamlen)
10             return t;
11         else
12             return NULL;
13     }
14
15 private:
16     int streamlen;
17     int task;
18 };
```

In this case, the node `svc` actually does not take into account any input stream item (the input parameter name is omitted on line 8). Rather, each time the node is activated, it returns a task to be computed using the internal `tasks` value. The task is directed to the “next” worker by the FastFlow farm run time support.

In order to provide a `collector`, we can use again a `ff_node`. In case the results need further processing, they can be directed to the next node in the streaming network using the mechanisms detailed in Sect. 5. Otherwise, they can be processed within the `svc` method of the `ff_node` subclass used to instantiate the collector. As an example, a `collector` just printing the tasks/results it gets from the workers may be programmed as follows:

```
class Collector: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        std::cout << "Collector got " << *t << std::endl;
        return GO_ON;
    }
};
```

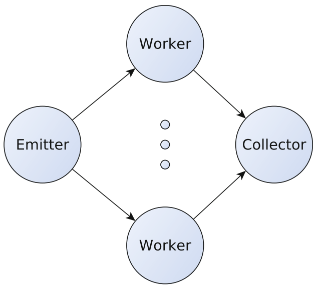
With these `Emitter` and `Collector` classes defined and assuming to have a worker defined by the class:

```
class Worker: public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        (*t)++;
        return task;
    }
};
```

we can define a program processing a stream of integers by increasing each one of them with a farm as follows:

```
int main(int argc, char * argv[]) {
    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);

    ff_farm<> farm;

    Emitter E(streamlen);
    farm.add_emitter(&E);

    std::vector<ff_node *> w;
    for(int i=0;i<nworkers;++i)
      w.push_back(new Worker);
    farm.add_workers(w);

    Collector C;
    farm.add_collector(&C);

    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
        return -1;
    }
    return 0;
}
```

The concurrent activity graph in this case is the following one:



When run with the first argument specifying the number of workers to be used and the second one specifying the length of the input stream generated in the collector node, we get the expected output:

```
ffsrc$ ./a.out 2 10
Collector got 2
Collector got 3
Collector got 4
Collector got 5
Collector got 6
Collector got 7
Collector got 8
Collector got 9
Collector got 10
ffsrc$
```

8.2 Farm with No Collector

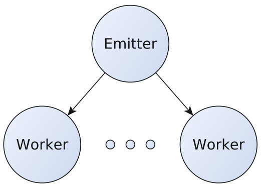
We move on considering a further case: a farm with emitter but no collector. Having no collector the workers may not deliver results: all the results computed by the workers must be consolidated in memory. The following code implements a farm where a stream of tasks of type `TASK` with an integer tag `i` and an integer value `t` are processed by the worker of the farm by:

- computing `t++` and
- storing the result in a global array at the position given by the tag `i`.

Writes to the global result array need not to be synchronized as each worker writes different positions in the array (the `TASK` tags are unique, the array is managed according a "single owner computes" rule).

```
#include <vector>
#include <iostream>
#include <ff/farm.hpp>

static int * results;

struct task_t {
    task_t(int i,int t):i(i),t(t) {}
    int i;
    int t;
};

using namespace ff;

class Worker: public ff_node {
public:
    void * svc(void * task) {
        TASK * t = (TASK *) task;
        results[t->i] = ++(t->t);
        return GO_ON;
    }
};

class Emitter: public ff_node {
public:
    Emitter(int n):streamlen(n),task(0) {}

    void * svc(void *) {
        task++;
        task_t * t = new task_t(task,task*task);
31          if (task<streamlen) return t;
32          return NULL;
33      }
34  private:
35      int streamlen;
36      int task;
37  };
38
39  int main(int argc, char * argv[]) {
40
41      int nworkers=atoi(argv[1]);
42      int streamlen=atoi(argv[2]);
43      results = (int *) calloc(streamlen,sizeof(int));
44
45      ff_farm<> farm;
46
47      Emitter E(streamlen);
48      farm.add_emitter(&E);
49
50      std::vector<ff_node *> w;
51      for(int i=0;i<nworkers;++i)
52          w.push_back(new Worker);
53      farm.add_workers(w);
54
55      std::cout << "Before starting computation" << std::endl;
56      for(int i=0; i<streamlen; i++)
57          std::cout << i << " : " << results[i] << std::endl;
58      if (farm.run_and_wait_end()<0) {
59          error("running farm\n");
60          return -1;
61      }
62      std::cout << "After computation" << std::endl;
63      for(int i=0; i<streamlen; i++)
64          std::cout << i << " : " << results[i] << std::endl;
65      return 0;
66  }
```

The `Worker` code at lines 15–22 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non existing, see lines 45–53: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 19 are directly stored in the global array. In this case the concurrent activity graph is the following:



The main program prints the results vector before calling the FastFlow

`start_and_wait_end()`

and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc$ farmNoC 2 10
Before starting computation
0 : 0
1 : 0
2 : 0
```

```
3 : 0
4 : 0
5 : 0
6 : 0
7 : 0
8 : 0
9 : 0
After computation
0 : 0
1 : 2
2 : 5
3 : 10
4 : 17
5 : 26
6 : 37
7 : 50
8 : 65
9 : 82
ffsrc$
```

Besides demonstrating how a farm without collector may compute useful results, the program of the last listing also demonstrates how complex task data structures can be delivered and retrieved to and from the FastFlow streaming network streams via `svc void *` parameters.

8.3 Specializing the Scheduling Strategy in a Farm

In order to select the worker where an incoming input task has to be directed, the FastFlow farm uses an internal `ff_loadbalancer` that provides a method `int selectworker()` returning the index in the worker array corresponding to the worker where the next task has to be directed. This method cannot be overwritten, actually. But the programmer may subclass the `ff_loadbalancer` and provide his own `selectworker()` method and pass the new load balancer to the farm emitter, therefore implementing a farm with a user defined scheduling policy.

The steps to performed in this case are exemplified with the following, relevant portions of code. First, we subclass the `ff_loadbalancer` and we provide our own `selectworker()` method:

```cpp
class my_loadbalancer: public ff_loadbalancer {
protected:
    // implement your policy...
    inline int selectworker() { return victim; }

public:
    // the ff_loadbalancer requires the maximum number of workers
    my_loadbalancer(int max_num_workers):ff_loadbalancer(max_num_workers)
        {}

    void set_victim(int v) { victim=v;}
private:
    int victim;
};
```

Then we create a farm specifying the new load balancer class as a type parameter:

```cpp
ff_farm<my_loadbalancer> myFarm(...);
```

Please note that the class `ff_loadbalancer` needs to know the maximum number of worker threads it has to manage. Eventually, we create an emitter that within its `svc` method invokes the `set_victim` method right before outputting a task towards the worker string, either with a `ff_send_out(task)` or with a `return(task)`. The emitter is declared as:

```
class myEmitter: public ff_node {

  myEmitter(my_loadbalancer * lb):lb(lb) {}

  ...

  void * svc(void * task) {
    ...
    workerToBeUsed = somePolicy(...);
    lb->set_victim(workerToBeUsed);
    ...
    ff_send_out(task);
    return GO_ON;
  }

  ...
private:
  my_loadbancer * lb;
}
```

and it is inserted in the farm with the code

```
myEmitter emitter(myFarm.getlb());
myFarm.add_emitter(emitter);
```

Another simpler option for scheduling tasks directly in the `svc` method of the farm emitter is to use the `ff_send_out_to` method of the `ff_loadbalancer` class. In this case what is needed is to pass the default load balancer object to the emitter thread and to use the `ff_loadbalancer::ff_send_out_to` method instead of `ff_node::ff_send_out` method for sending out tasks.

What we get is a farm where the worker to be used to execute the task appearing onto the input stream is decided by the programmer through the proper implementation of `my_loadbalancer` rather than being decided by the current FastFlow implementation.

Two particular cases specializing the scheduling policy in different way by using FastFlow predefined code are illustrated in the following two subsections.

8.4 Broadcasting a Task to all Workers

FastFlow supports the possibility to direct a task to all the workers in a farm. It is particularly useful if we want to process the task by workers implementing different functions. The broadcasting is achieved through the declaration of a specialized load balancer, in a way very similar to what we illustrated in Sect. 8.3.

The following code implements a farm whose input tasks are broadcasted to all the workers, and whose workers compute different functions on the input tasks, and therefore deliver different results on the output stream.

```
#include <iostream>
#include <ff/farm.hpp>
#include <ff/node.hpp>
#include <cmath>

using namespace std;
using namespace ff;


// should be global to be accessible from workers
#define MAX 4
int x[MAX];

class WorkerPlus: public ff_node {
  int svc_init() {
    cout << "Worker initialized" << endl;
    return 0;
  }

  void * svc(void * in) {
    int *i = ((int *) in);
    int ii = *i;
    *i++;
    cout << "WorkerPLus got " << ii << " and computed " << *i << endl ;
    return in;
  }
};

class WorkerMinus: public ff_node {
  int svc_init() {
    cout << "Worker initialized" << endl;
    return 0;
  }

  void * svc(void * in) {
    int *i = ((int *) in);
    int ii = *i;
    *i--;
    cout << "WorkerMinus got " << ii << " and computed " << *i << endl ;
    return in;
  }
};

class my_loadbalancer: public ff_loadbalancer {
public:
    // this is necessary because ff_loadbalancer has non default parameters
      ....
    my_loadbalancer(int max_num_workers):ff_loadbalancer(max_num_workers)
      {}

    void broadcast(void * task) {
        ff_loadbalancer::broadcast_task(task);
    }
};

class Emitter: public ff_node {
public:
    Emitter(my_loadbalancer * const lb):lb(lb) {}
    void * svc(void * task) {
        lb->broadcast(task);
        return GO_ON;
    }
private:
    my_loadbalancer * lb;
};

class Collector: public ff_node {
public:
    Collector(int i) {}
    void * svc(void * task) {
        cout << "Got result " << * ((int *) task) << endl;
        return GO_ON;
    }

```

```
 73
 74 };
 75
 76
 77
 78 #define NW 2
 79
 80 int main(int argc, char * argv[])
 81 {
 82   ffTime(START_TIME);
 83
 84   cout << "init " << argc  << endl;
 85   int nw = (argc==1 ? NW : atoi(argv[1]));
 86
 87   cout << "using " << nw << " workers " << endl;
 88
 89   // init input (fake)
 90   for(int i=0; i<MAX; i++) {
 91     x[i] = (i*10);
 92   }
 93   cout << "Setting up farm" << endl;
 94     // create the farm object
 95     ff_farm<my_loadbalancer> farm(true,nw);
 96     // create and add emitter object to the farm
 97     Emitter E(farm.getlb());
 98     farm.add_emitter(&E);
 99   cout << "emitter ok "<< endl;
100
101
102   std::vector<ff_node *> w;   // prepare workers
103   w.push_back(new WorkerPlus);
104   w.push_back(new WorkerMinus);
105   farm.add_workers(w);        // add them to the farm
106   cout << "workers ok "<< endl;
107
108   Collector C(1);
109   farm.add_collector(&C);
110   cout << "collector ok "<< endl;
111
112   farm.run_then_freeze();     // run farm asynchronously
113
114   cout << "Sending tasks ..." << endl;
115   int tasks[MAX];
116   for(int i=0; i<MAX; i++) {
117      tasks[i]=i;
118      farm.offload((void *) &tasks[i]);
119   }
120   farm.offload((void *) FF_EOS);
121
122   cout << "Waiting termination" << endl;
123   farm.wait();
124
125   cout << "Farm terminated after computing for " << farm.ffTime() << endl;
126
127   ffTime(STOP_TIME);
128   cout << "Spent overall " << ffTime(GET_TIME) << endl;
129
130 }
```

At lines 44–52 a `ff_loadbalancer` is defined providing a `broadcast` method. The method is implemented in terms of an `ff_loadbalancer` internal method. This new loadbalancer class is used as in the case of other user defined schedulers (see Sect. 8.3) and the emitter eventually uses the load balancer `broadcast` method *instead* of delivering the task to the output stream (i.e. directly to the string of the workers). This is done through the `svc` code at lines 57–60. Lines

103 and 104 are used to add two different workers to the farm. The rest of the program is standard, but for the fact the resulting farm is used as an accelerator (lines 112–123, see Sect. 10).

8.5 Using Autoscheduling

FastFlow provides suitable tools to implement farms with "auto scheduling", that is farms where the workers "ask" for something to be computed rather than accepting tasks sent by the emitter (explicit or implicit) according to some scheduling policy. This scheduling behaviour may be simply implemented by using the `ff_farm` method `set_scheduling_ondemand()`, as follows:

```
ff_farm<> myFarm(...);
myFarm.set_scheduling_ondemand();
...
farm.add_emitter(...);
...
```

The scheduling policy implemented in this case is an approximation of the auto scheduling, indeed. The emitter simply checks the length of the SPSC queues connecting the emitter to the workers, and delivers the task to the first worker whose queue length is less or equal to 1. To be more precise, FastFlow should have implemented a request queue where the workers may write tasks requests tagged with the worker id and the emitter may read such request to choose the worker where the incoming tasks is to be directed. This is not possible as of FastFlow 1.1 because it still doesn't allow to read from multiple SPSC queues preserving the FIFO order.

8.6 Ordered Farm

Tasks passing through a task-farm can be subjected to reordering because of different execution times in the worker threads. To overcome the problem of sending packets in a different order with respect to input, tasks can be reordered after collection from the workers, although this solution might introduce extra latency mainly because reordering checks have to be executed even if the packets already arrive at the farm collector in the correct order.

The default round-robin and auto scheduling policies are not order preserving, for this reason a specialized version of the FastFlow farm has been introduced which enforce the ordering of the packets.

The ordered farm may be introduced by using the `ff_ofarm` skeleton. The following code sketches how to use the it as a middle stage of a 3-stage pipeline:

```
  ...
  ff_pipeline pipe;
  pipe.add_stage(new FirstStage(...));

  ff_ofarm ofarm;  // defines an order-preserving farm
  std::vector<ff_node *> w;
  for(int i=0;i<nworkers;++i) w.push_back(new Worker);
  ofarm.add_workers(w);

  pipe.add_stage(&ofarm); // adds the farm as 2nd stage
  pipe.add_stage(new LastStage(...));

  pipe.run_and_wait_end();
```

8.7 Simplified Ways to Create Farms Using C++11

In order to simplify the creation of farm objects—and especially for simple cases—some new constructors have been introduced in the latest release of FastFlow.

Using these new features, the simplest way to create a farm in FastFlow is to create a function F having the signature `T* F(T*, ff_node*const)` where T is a generic type, and then use this function to create the workers of the farm. As an example, in the following code excerpt a farm with 5 workers, each executing the function F, is declared and run:

```
ff_farm<> farm((std::function<T*(T*,ff_node*const)>)F, 5);
farm.run_and_wait_end();
```

With this code, the default emitter and collector nodes and the default round-robin scheduling and gathering policy are automatically instantiated. As different features of the new C++11 standard are used in the implementation of this farm, to compile the example above the correct flags to used with the g++ command are:

```
-std=c++11 -DHAS_CXX11_VARIADIC_TEMPLATES
```

Another way to create a task-farm skeleton in FastFlow is to pass the vector of `ff_nodes`, the Emitter and Collector nodes (if both exist) directly as parameters of the farm constructor as in the following simple examples:

```
#include <vector>
#include <ff/farm.hpp>

using namespace ff;
int main() {
    struct MyWorker: ff_node {
        void *svc(void *t) {
            printf("worker %d got one task\n", get_my_id());
            return t;
        }
    };
    // Emitter
    struct Emitter:public ff_node {
        std::function<void*()> F;
        Emitter(std::function<void*()> F):F(F) {}
        void *svc(void*) { return F(); }
    };
    const int K  = 20; // stream length
    const int nw = 7;  // n. of workers
    // function executed in the Emitter node
    auto F = [K]() -> void* {
        static int k = 0;
        if (k++ == K) return NULL;
        return new int(k);
    };
    std::vector<ff_node*> W;
    for(int i=0;i<nw;++i) W.push_back(new MyWorker);

    // farm with specialized Emitter and without collector
    ff_farm<> farm(W, new Emitter(F));
    farm.run_and_wait_end();

    return 0;
}
```

9 More on `ff_pipeline`

Pipeline skeletons can be easily instantiated using the new C++11-based constructors available in the latest release of FastFlow.

The simplest way to create a pipeline of n-stage in FastFlow is to create a number of functions with signature `T* (*F) (T*,ff_node*const)` and/or `ff_node` objects and then to pass them in the correct order to the `ff_pipe` constructor[5]. As an example consider the following 3-stage pipeline:

```
  struct myTask { .... }; // this is my input/output type

  myTask* F1(myTask *in,ff_node*const node) {...} // 1st function
  struct F2: ff_node {      // 2nd stage
    void *svc(void *t) {...}
  } F2;
  myTask* F3(myTask *in,ff_node*const node) {...} // 3rd function

  ff_pipe<myTask> pipe(F1,&F2,F3);
  pipe.run_and_wait_end();
};
```

Here 2 functions getting a `myTask` pointer as input and return type are used as first and third stage of the pipeline whereas an `ff_node` object is used as middle stage. To compile the above snippet of code, the command to use is:

```
ffsrc$ g++ -std=c++0x -DHAS_CXX11_VARIADIC_TEMPLATES -I $FF_ROOT test.
    cpp -o test -lpthread
```

that is the same switches needed to use the new farm syntax outlined in Sect. 8.7 are needed.

Since `ff_pipe` accepts both functions and `ff_node`, it is possible to easily build "complex" streaming networks using just few commands as in the following example where `ff_pipe` and `ff_farm` skeletons are mixed together:

```
  std::vector<ff_node*> W;
 W.push_back(new ff_pipe<myTask>(&F2,F3)); // 2-stage pipeline, 1st worker
 W.push_back(new ff_pipe<myTask>(&F2,F3)); // 2-stage pipeline, 2nd worker

  ff_pipe<myTask> pipe(F1, new ff_farm<>(W)); // 2-stage pipeline, seq+farm

  pipe.run_and_wait_end();
};
```

10 FastFlow as a Software Accelerator

Up to know we just showed how to use FastFlow to write a "complete skeleton application", that is an application whose complete flow of control is defined through skeletons. In this case the `main` of the C++ program written by the user is basically providing the structure of the parallel application by defining a

[5] The class *ff_pipe* is a wrapper of the class *ff_pipeline*.

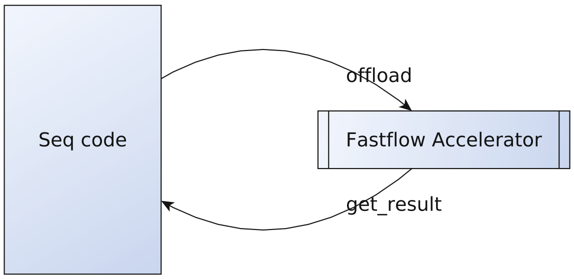


Fig. 4. FastFlow accelerator

proper FastFlow skeleton nesting and the commands to start the computation of the skeleton program and to wait its termination. All the business logic of the application is embedded in the skeleton parameters.

Now we want to discuss the second kind of usage which is supported by FastFlow, namely the FastFlow accelerator mode. The term "accelerator" is used the way it is used when dealing with hardware accelerators. An hardware accelerator—a GPU or an FPGA or even a more "general purpose" accelerator such as Tilera 64 core chips, Intel Many Core or IBM WireSpeed/PowerEN—is a device that can be used to compute particular kind of code faster that the CPU. FastFlow accelerator is a "software device" that can be used to speedup portions of code using the cores left unused by the main application. In other words, it's a way FastFlow supports to accelerate particular computation by using a skeleton program and offloading to the skeleton program tasks to be computed.

The FastFlow accelerator will use $n-1$ cores of the n core machine, assuming that the calling code is not parallel and will try to ensure a $n-1$ fold speedup is achieved in the computation of the tasks offloaded to the accelerator, provided a sufficient number of tasks to be computed are given.

Using FastFlow accelerator mode is not that different from using FastFlow to write an application only using skeletons (see Fig. 4). In particular, the following steps must be followed:

- A skeleton program has to be written, using the FastFlow skeletons computing the tasks that will be given to the accelerator. The skeleton program used to program the accelerator is supposed to have an input stream, used to offload the tasks to the accelerator.
- Then, the skeleton program must be run using a particular method, different from the `run_and_wait_end` we have already seen, that is a `run_then_freeze()` method. This method will start the accelerator skeleton program, consuming the input stream items to produce either output stream items or to consolidate (partial) results in memory. When we want to stop the accelerator, we will deliver and end-of-stream mark to the input stream.
- Eventually, we must wait the computation of the accelerator is terminated.

A simple program using FastFlow accelerator mode is shown below:

```
#include <vector>
#include <iostream>
#include <ctime>
#include <ff/farm.hpp>

using namespace ff;

int * x;
int nworkers = 0;

class Worker: public ff_node {
public:

  Worker(int i) {
    my_id = i;
  }

  void * svc(void * task) {
    int * t = (int *)task;
    x[my_id] = *t;
    return GO_ON;
  }
private:
  int my_id;
};

int main(int argc, char * argv[]) {

  if (argc<3) {
    std::cerr << "use: "
           << argv[0]
           << " nworkers streamlen\n";
    return -1;
  }

  nworkers=atoi(argv[1]);
  int streamlen=atoi(argv[2]);

  x = new int[nworkers];
  for(int i=0; i<nworkers; i++)
    x[i] = 0;

  ff_farm<> accelerator(true);

  std::vector<ff_node *> w;
  for(int i=0;i<nworkers;++i)
    w.push_back(new Worker(i));
  accelerator.add_workers(w);

  if (accelerator.run_then_freeze()<0) {
    error("running farm\n");
    return -1;
  }

  for(int i=0; i<=streamlen; i++) {
    int * task = new int(i);
    accelerator.offload(task);
  }
  accelerator.offload((void *) FF_EOS);
  accelerator.wait();

  for(int i=0; i<nworkers; i++)
    std::cout << i << ":" << x[i] << std::endl;

  return 0;
}
```

We use a farm accelerator. The accelerator is declared at line 43. The "true" parameter is the one telling FastFlow this farm has to be used as an accelerator. Workers are added at lines 45–48. Each worker is given its id as a constructor parameters. This is the very same code as the one used in plain FastFlow applications. Line 50 starts the skeleton code in accelerator mode. Lines 55 to 58 offload tasks to be computed to the accelerator. These lines could be part of any larger C++ program, indeed. The idea is that whenever we have a task ready to be submitted to the accelerator, we simply "offload" it to the accelerator. When we have no more tasks to offload, we send and end-of-stream (line 59) and eventually we wait for the completion of the computation of tasks in the accelerator (line 60).

This kind of interaction with an accelerator without output stream is intended to model those computations than consolidate results directly in memory. We can also assume that results are awaited from the accelerator through its output stream. In this case, we first have to write the skeleton code of the accelerator in such a way an output stream is supported. In the new version the accelerator sample program below, we add a collector to the accelerator farm (line 45). The collector simply merges the results from workers to the output stream (lines 18–24 in the code listing below). Once the tasks have been offloaded to the accelerator, rather than waiting for accelerator completion, we can ask computed results as delivered to the accelerator output stream through the `bool load_result(void **)` method (see lines 59–61).

```
 1 #include <vector>
 2 #include <iostream>
 3 #include <ctime.h>
 4 #include <ff/farm.hpp>
 5
 6 using namespace ff;
 7
 8 class Worker: public ff_node {
 9 public:
10
11   void * svc(void * task) {
12     int * t = (int *)task;
13     (*t)++;
14     return task;
15   }
16 };
17
18 class Collector: public ff_node {
19 public:
20     void * svc(void * task) {
21         int * t = (int *)task;
22         return task;
23     }
24 };
25
26
27 int main(int argc, char * argv[]) {
28
29   if (argc<3) {
30     std::cerr << "use: "
31            << argv[0]
32            << " nworkers streamlen\n";
33     return -1;
34   }
35
```

```
  int nworkers=atoi(argv[1]);
  int streamlen=atoi(argv[2]);

  ff_farm<> accelerator(true);

  std::vector<ff_node *> w;
  for(int i=0;i<nworkers;++i)
    w.push_back(new Worker());
  accelerator.add_workers(w);
  accelerator.add_collector(new Collector());

  if (accelerator.run_then_freeze()<0) {
    error("running farm\n");
    return -1;
  }

  for(int i=0; i<=streamlen; i++) {
    int * task = new int(i);
    accelerator.offload(task);
  }
  accelerator.offload((void *) FF_EOS);

  void * result;
  while(accelerator.load_result(&result)) {
    std::cout << "Got result :: "<< (*((int *)result)) << std::endl;
  }
  accelerator.wait();

  return 0;
}
```

The `bool load_result(void **)` methods synchronously await for one item being delivered on the accelerator output stream. If such item is available, the method returns "true" and stores the item pointer in the parameter. If no other items will be available, the method returns "false". An asynchronous method is also available with signature `bool load\_results\_nb(void **)` If no result is available at the moment the method is called, it returns a "false" value, and you should retry later on to see whether a result may be retrieved.

It is worth pointing out that the usage of FastFlow to build accelerators applies to both C++ and pure C sequential programs. In the latter case, the resulting program must be compiled using the C++ tool chain, of course, but eventually the code gets accelerated exactly as if the original sequential code was written in C++. This is because all the C++ part is confined in the FastFlow accelerator part and because the accelerator "interface" methods (the offloading and result retrieval methods) perfectly work with pure C parameters.

11 Skeleton Nesting

In FastFlow skeletons may be arbitrarily nested. Taking into account just farms and pipelines this means that farms may be used as pipeline stages, and pipelines may be used as farm workers.

As an example, you can define a farm with pipeline workers as follows:

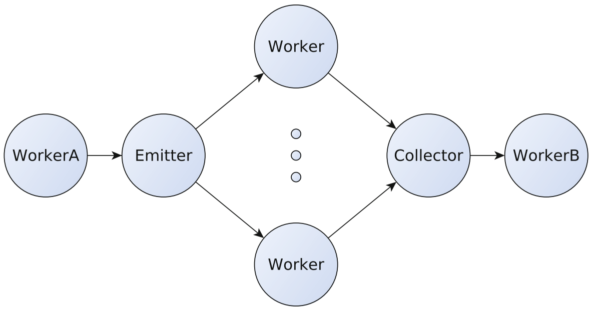
```
    ff_farm<> myFarm;

    std::vector<ff_node *> w;
    for(int i=0; i<NW; i++)
```

```
      ff_pipeline * p = new ff_pipeline;
      p->add_stage(new S1());
      p->add_stage(new S2());
      w.push_back(p);
    }
    myFarm.addWorkers(w);
```

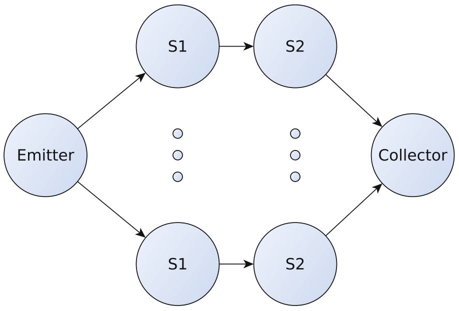
or we can use a farm as a pipeline stage by using a code such as:

```
    ff_pipeline * p = new ff_pipeline;
    ff_farm <>    f = new ff_farm;

    ...

    f.addWorkers(w);

    ...

    p->add_stage(new SeqWorkerA());
    p->add_stage(f);
    p->add_stage(new SeqWorkerB());
```

The concurrent activity graph in this case will be the following one:



while in the former case it will be such as



As a general rule, any skeleton may be used in any place where a `ff_node` is required. As more and more skeletons are added in FastFlow, more and more complex composite skeleton structures may be used.

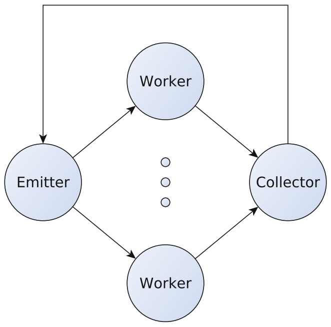
12 Feedback Channels

There are cases where it is useful to have the possibility to route back some results to the streaming network input stream for further computation. For example, this possibility may be exploited to implement divide&conquer pattern using the task-farm: tasks injected in the farm are split by the scheduler to the workers and the resulting tasks are routed back to the scheduler to evaluate if further splitting is needed. Tasks that can be computed using the base case code, are computed by the workers and their results are used for the conquer phase, usually performed in memory.

The feedback channel in a farm or pipeline may be introduced by the `wrap_around()` method on the interested skeleton. In case our applications uses a farm pattern as the outermost skeleton, we may therefore add the method call after instantiating the farm object:

```
ff_farm<> myFarm;
...
myFarm.add_emitter(&e);
myFarm.add_collector(&c);
myFarm.add_workers(w);

myFarm.wrap_aroud();
...
```

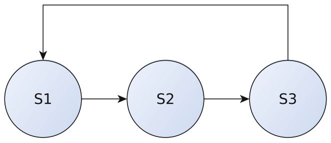
and this will lead to the concurrent activity graph



The same if parallelism is expressed by using a pipeline as the outermost skeleton:

```
ff_pipeline myPipe;

myPipe.add_stage(s1);
myPipe.add_stage(s2);
myPipe.add_stage(s3);
...
myPipe.wrap_around();
...
```

leading to the concurrent activity graph:



As of FastFlow 1.1, the only possibility to use the feedback channel provided by the `wrap_around` method is relative to the outermost skeleton, that is the one with no input stream. Starting with version 2.0.0, this limitation has been relaxed such that it is possible to use feedback channels (for some particular skeleton cases) where the `wrap_around` method can be called not only in the outmost skeleton but also to the last stage of a pipeline composing the application. Some of these possible cases are sketched in Fig. 5.

As an example, in the following we provide the code needed to create a 2-stage pipeline where the second stage is a farm with feedback channels between each worker and the farm emitter:

```
    ff_farm<>   farm(F, 8);  // farm of the function F using 8 workers
    farm.remove_collector(); // removes the default collector
    // the scheduler gets in input the internal load-balancer
    farm.add_emitter(new Sched(farm.getlb()));
    // adds feedback channels between each worker and the scheduler
    farm.wrap_around();

    ff_pipe<myTask> pipe(seq, &farm); // creates the pipeline
    pipe.run_and_wait_end();
```

In this case the emitter node of the farm receives tasks both from the first stage of the pipeline and from farm's workers. The emitter non-deterministically processes input tasks giving priority to the tasks coming back from workers.

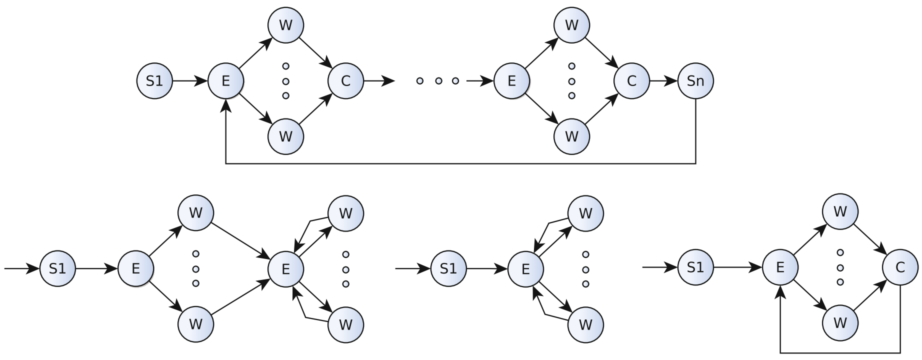


Fig. 5. Some possible FastFlow schemas with feedback channels.

13 Run Time Routines

Several utility routines are defined in the FastFlow runtime. We recall here the main ones.

- `virtual int get_my_id()`
  returns a virtual id of the node where the concurrent activity (its `svc` method) is being computed.
- `const int ff_numCores()3`
  returns the number of cores in the target architecture.
- `int ff_mapThreadToCpu(int cpu_id, int priority_level=0)`
  pins the current thread to `cpu_id`. A priority may be set as well, but you need root rights in general, and therefore this should non be specified by normal users.

– `void error(const char * str, ...)`
  is used to print error messages.
– `virtual bool ff_send_out(void * task,`
  `unsigned intretry=((unsigned int)-1),`
  `unsigned int ticks=(TICKS2WAIT))`
  delivers an item onto the output stream, possibly retrying upon failure a given number of times, after waiting a given number of clock ticks.
– `double ffTime()`
  returns the time spent in the computation of a farm or of pipeline, including the `svc_init` and `svc_end` time. This is a method of both pipeline and farm classes.
– `double ffwTime()`
  returns the time spent in the computation of a farm or of pipeline, in the `svc` method only.
– `double ffTime(int tag)`
  is used to measure time in portions of code. The `tag` may be: `START_TIME`, `STOP_TIME` or `GET_TIME`. A `ff_ffTime(GET_TIME)` returns the time elapsed in between two consecutive calls to `ffTime()` with parameter `START _TIME` and `STOP_TIME`, respectively.
– `void ffStats(std::ostream & out)`
  prints the statistics collected while using FastFlow. The program must be compiled with `TRACE_FASTFLOW` defined, however.

14 Threads Mapping

FastFlow performance can significantly depend on the mapping of concurrent activities to existing cores. Which is the best mapping depends on many factors of the underlying architecture at hand. Here we describe the low-level mechanisms provided by FastFlow that can be used to devise suitable mapping policies.

In FastFlow there are basically two main ways to pin a `ff_node` thread to a core:

– at thread creation time. This feature is supported via a gcc intrinsic operations that make it possible to create the thread implementing a FastFlow `ff_node` directly on a specific core;
– at any time during the run, by using the `ff_mapThreadToCpu` and `ff_getMyCpu` functions.

By default, all threads implementing sequential and parallel nodes of the skeleton tree are pinned to available cores according to a linear static mapping. For example, considering a 3-stage pipeline where the first and last stages are sequential `ff_nodes` whereas the middle stage of the pipeline is a farm node having 2 workers that are both pipeline of 2 sequential stages. In this case, we have in total 8 threads (with ids in the range (0–7)), including the farm emitter and collector threads. Such threads will be automatically pinned on the first 8 cores (supposing there are at least 8 cores available) of the underlying architecture. In particular the first pipeline stage will be pinned to core 0, the emitter of the farm to core 1, the two stages of the first worker to cores 2 and 3, the two stages of the second worker to cores 4 and 5, the collector to core 6 and the last stage to core 7.

To control the initial placement of the threads on the cores the FastFlow programmer may use the class `threadMapper`. The `threadMapper` gets in input a string (lets say mapstring) of comma-separated *core-id*s. At thread creation time, the FastFlow run-time will try to pin the thread with id `tid` in the core id specified in the corresponding position of the mapping string (modulo the number of core-ids in the string). Considering the 3-stage example described above and the following mapping string setup:

```
  const char worker_mapping[] = "0, 2, 4, 6, 8";
  threadMapper::instance()->setMappingList(worker_mapping);
  ....
  myprogram.run_and_wait_end();
```

The emitter of the farm is placed on core 2, the first sequential stages of the two pipelines in the farm on core 4 and 8, respectively.

In order to re-map threads after they have been created or during the computation, the function `ff_mapThreadToCpu` may be used. In the following there is an example of the usage of this function in the `svc_init` method of the `ff_node` class:

```
class myNode: public ff_node {
    ....

    int svc_init() {
        printf("Thread currently running on CPU %d\n",ff_getMyCpu());
        if (ff_mapThreadToCpu(mapThreads(get_my_id)) != 0)
          printf("Cannot map thread %d (local id %d) on CPU %d\n",
              getTid(), get_my_id(),
                   mapThreads(getTid()));
        return 0;
    }
    ...
```

The `mapThreads` function is a user defined function which gets in input the thread id and returns the core id in which the thread has to be pinned.

FastFlow also provides the possibility of not sticking concurrent activities to a particular core, and leaving instead to the operating system the full responsibility of the thread (dynamic) mapping. This possibility may be exploited by compiling the FastFlow program with the `NO_DEFAULT_MAPPING` symbol defined:

```
ffsrc$ g++ -DNO_DEFAULT_MAPPING -I $FF_ROOT ...
```

Under Linux, this allows threads to be moved between cores by the Linux scheduler. In most cases, this means threads are migrated with a substantial cache migration overhead, however.

15 Advanced Features

The skeletons and features discussed in the previous sections—apart from the C++11 related mechanisms of Sects. 8.7 and 9—are relative to the "core" FastFlow implementation, that is to the framework provided since version 1.0.

Skeleton	Modelled parallel pattern
`ff_farm`	computation of the same function (or of different functions) over all the items of the input stream
`ff_pipeline`	computation of functions in stages
`ff_map`	computation of the same function over all the items of a collection (array), reduction of a collection (array) by means of a binary, associative and commutative operator, combination of map and reduce on the same collection (array)
`FF_PARFOR`	map (possibly with an additional reduce phase) defined through a parallel for
`stencil2D`	computation of the same function over all the items of a collection (array). The function is computed on the value of the item and of the neighboring items, defined through a proper "stencil"
`poolevolution`	iteratively evolves a population of individuals (genetic like evolution)
`dc`	divide and conquer
`ff_graphsearch`	parallel graph search
`ff_mdf`	macro data flow (workflow) graph interpreter. The graph may be static or generated dynamically

Fig. 6. Skeletons available in FastFlow (as of December 2013)

More recently, within the activities of the EU FP7 STREP project "ParaPhrase" the FastFlow framework has been extended in several ways to accomplish different necessities from the application programmers. In particular:

- different high level patterns have been added,
- facilities to support coordinated execution of FastFlow program on internetworked multi core machines have been added,
- support for execution of new data parallel patterns on GPUs have been added as well,
- refactoring tools suitable to automatically introduce and refactor FastFlow skeletons in application code have been developed, and
- last but not least, a theoretical framework has been developed showing how the "core" FastFlow customizable components may be used to implement basically any kind of parallel pattern.

In this Section, we briefly outline these features.

15.1 High Level Patterns

FastFlow has been extended with new skeletons modeling general purpose and high level patterns. The table in Fig. 6 summarizes the skeletons provided so far, including the "core" ones. In particular, the following skeletons have been added to the "core" ones:

- The **map** skeleton applies the same computation to all the elements of an input collection (array) producing an output collection. The skeleton may be

specialized to target GPU instead of CPU, if present, using either OpenCL or CUDA. The only difference between the instantiation of a map skeleton targeting CPU cores and the same skeleton targeting GPU is in the way used to provide the "worker" code, as in the latter case, OpenCL or CUDA kernels are needed.
- The **reduce** skeleton "sums up" using a commutative and associative binary operator all the elements in a collection (array) producing a single scalar value. As for the map skeleton, two different implementations of the reduce pattern exist targeting CPU or GPU.
- The **divide and conquer** skeleton implements the classical *divide et impera* computations, only targeting CPU, at the moment.
- The **stencil** skeleton applies the same computation to all the elements in an input collection (array) to produce an output collection, as in the map case. However, in case of the stencil, the computation of the single item depends not only on the value of that item in the input collection (as in map skeleton) but also on the values of a set of "neighboring" elements in the input collection. Moreover, the computation is iterative, that is more iterations are performed up to a given "termination condition".
- The **workflow interpreter** skeleton computes a statically or dynamically generated macro data flow graph over input data. It is usually instantiated to compute numerical code—possibly including calls to optimized numerical libraries—with maximum efficiency.
- The **pool evolution** skeleton iteratively computes the evolution of a set of "individuals" in a population by repeatedly (i) selecting a subset of individuals, (ii) "evolving" the selected individuals in new individuals, (iii) selecting a subset of the new individuals to be included in the original population and (iv) checking if the resulting population satisfies a given termination condition.

All these skeletons are provided to application programmers as FastFlow classes that may be instantiated and used in their programs. As an example, a stencil computation expressed by the following sequential code:

```
while(k<=maxit && error > tot) {
  /* copy new solution into old matrix */
  for(int j=0;j<m;j++)
    for(int i=0;i<n;++i) uold[i+m*j]=u[i+m*j];
  /* computes the stencil and residual */
  for(int j=1;j<(m-1);++j)
     for(int i=1;i<(n-1);++i) {
         resid = compute_resid(f,i,j, uold,ax,ay);
         /* updates solution */
         u[i + m*j] = uold[i + m*j] - omega * resid;
         /* accumulates residual error */
         error =error + resid*resid;
     }
  error = sqrt(error) / (n*m);  k++;
}
```

may be implemented using a simple stencil skeleton instantiation such as:

```
  // instantiate stencil pattern
  stencil2D<double> stencil(u,uold,m,n,n,NUMTHREADS,1,1,false);

  stencil->initInFunc(initU);
```

```
stencil->initOutFunc(initUold);

stencil->computeFunc(stencilF, 1,m-1,1, 1,n-1,1);
stencil->reduceFunc(condF, maxit, reduceOp, 0.0);

stencil->run_and_wait_end();
```

where:

– *initU* initializes the single cell of the u matrix. The function is called in a parallel loop in order to execute the initialization phase in parallel;
– *initUold* initializes the single cell of the *uold* matrix as in the previous case;
– *computeFunc* executes the stencil for each pair (i, j) updating the u matrix and reading values from the *uold* matrix;
– *reduceFunc* reduction function used to evaluate the error and for terminating the computation;
– *run_and_wait_end* starts the stencil computation and wait for termination.

More details on the new skeletons provided may be found in [32,34].

15.2 Targeting Distributed Machines

`ff_nodes` model the concurrent activities in FastFlow. They are assumed to be independent threads, orchestrated by FastFlow in such a way they have a input channel (stream) providing input tasks to be computed and an output channel (stream) where the results computed out of the input tasks have to be delivered. The whole management of the input and output channels is in charge of the FastFlow run time support, rather than of the application programmer, as evident from the previous sections. These channel are implemented in "core" FastFlow by means of shared memory lock free queues.

Recently, a `ff_dnode` class has been added supporting the possibility to have one of the input and output channels implemented by means of *transport* layer on top of TCP/IP. The `d_node` class therefore supports communications between FastFlow threads running on different machines.

`ff_dnodes` may be used in any place where an `ff_node` may be used, that is as pipeline stages, farm workers, etc. They still define proper `svc` methods modeling the local computation of the thread. However, they have additional methods that allow to bind the input or output channel defined as distributed rather than local to an $\langle IP, port\rangle$ pair and to provide proper serialization (marshalling and un-marshalling) procedures for the tasks exchanged with the remote FastFlow node. Default serialization methods are provided in case tasks/results to be transmitted are represented within a single, contiguous memory region. Additional programming effort is required to accomplish non contiguous data serialization. In particular, in order to serialize non contiguous data, the programmer must implement a `prepare` method such as:

```
void prepare(svector<iovec>& v, void* ptr, const int sender=-1) {
    struct iovec iov={ptr,taskSize*taskSize*sizeof(double)};
    v.push_back(iov);
}
```

where a `svector<iovec>` is filled with `struct iovec` pairs hosting the pointer and the length of the different memory areas used to represent the complex data structure to be serialized.

The distributed channels implemented on top of the TCP/IP stack[6] support the implementation of different channels, including one-to-one channels as well as one-to-n, n-to-n channels implementing different distribution (scatter, multicast, broadcast) and collection (gather, gatherall) policies. By using `d_nodes` and the associate channels FastFlow currently supports the implementation of farms with remote workers, pipeline with remote stages targeting both clusters of Linux workstations and cloud nodes [18].

15.3 Targeting GPUs

In the last versions of FastFlow, the execution of FastFlow data parallel patterns such as the `ff_map` may be directed to GPU cores rather than to CPU cores. A `ff_mapCUDA` (`ff_mapOCL`) map skeleton may be used in place of a simple `ff_map` to use (possibly existing) CUDA (OpenCL) kernels to map the same application onto all the elements of a collection (array). Apart from the specific syntax details needed to pass the CUDA (OpenCL) kernel to the constructor of the `ff_mapXXX` skeleton, the two skeletons implement all the necessary steps to execute the kernel on the GPU, including all the data transfers needed to move input data to the device memory and results back to the CPU main memory. Additional macros has been defined that simplify the writing of kernel code as well as of the full map skeleton. In particular, the kernel code may be provided through a proper macro taking as arguments the name of the kernel, the type of the input parameter and of the result, the name of the input (formal) parameter and the body of the kernel function. As an example, the code:

```
FFMAPFUNC(mapf, float, elem, return (elem+1.0) );
```

defines a kernel named `mapf` taking a `float elem` parameter and returning the `float elem+1.0` value.

Using this kernel a farm with map workers targeting GPU may be defined as follows:

```
1      ff_farm<> farm;
2      Emitter   E(streamlen,inputsize);
3      Collector C(inputsize);
4      farm.add_emitter(&E);
5      farm.add_collector(&C);
6
7      std::vector<ff_node *> w;
8      for(int i=0;i<nworkers;++i)
9          w.push_back(NEWMAPONSTREAM(oclTask<float>, mapf));
10     farm.add_workers(w);
11     farm.run_and_wait_end();
```

The `NEWMAPONSTREAM` macro on line 9 defines a map targeting GPU through CUDA or OpenCL—depending on the macro defined in the preamble (`ff_CUDA` or `ff_OCL`)—and computing the `mapf` kernel on `float` arrays provided through

[6] We currently use the `zeroMQ` library to support the distributed channels ().

a stream[7]. More details on the FastFlow skeletons targeting GPUs may be found in [33]. Reference [38] discusses how a FastFlow module may be implemented automatically splitting map tasks among CPU and GPU cores, such that the overall execution time is optimized.

15.4 Refactoring

Programs using skeletons may be rewritten in such a way the functional semantics is preserved while the non-functional semantics (the aspects relative to performance or power consumption, as an example) is possibly changed.

Different rewriting rules hold, including[8]:

$farm(\Delta, nw) \equiv \Delta$	a farm with nw workers is equivalent to the computation of the worker alone
$pipeline(\Delta_1, \Delta_1) \equiv comp(\Delta_1, \Delta_2)$	a pipeline of two stages is equivalent to the sequential composition of the two stages
$pipe(map(\Delta_1), map(\Delta_2)) \equiv map(pipe(\Delta_1, \Delta_2))$	map promotion: two maps in a pipeline are equivalent to a map with a pipeline worker

These rules have obvious impact on performances. In [6] we have shown that any stream parallel skeleton composition involving farms and pipelines eventually may be substituted by a single farm of sequential workers derived applying the rules listed above and sporting a better or similar—with respect to the original composition—service time. As a further example, a pipeline with two unbalances stages $pipe(Stage1, Stage2)$ may be rewritten with much better performance applying the first rewriting rule in the table above "right-to-left" to obtain a $pipe(Stage1, farm(Stage2, nw))$ in case the second stage has a latency nw times the latency of the first stage.

Therefore, if our original pipeline is implemented through the code:

```
...
ff_pipeline pipe;
pipe.add_stage(new Stage1());
pipe.add_stage(new Stage2());
...
pipe.run_and_wait_end();
```

we can easily apply the rewriting rule introducing the farm obtaining a refactored program such as:

```
...
ff_pipeline pipe;
pipe.add_stage(new Stage1());

ff_farm<> farm;
```

[7] The map itself works on a single input task to produce a single output result, by default.

[8] Δ represents any skeleton composition, in this case.

```
std::vector<ff_node*> w;
for(int i=0;i<nw;i++) w.push_back(new Stage2());
farm.add_workers(w);

pipe.add_stage(&farm);
...
pipe.run_and_wait_end();
```

It is worth pointing out the negligible programming effort required to refactor the program according to the "logical" rewrite rule listed in the table, especially compared to the effort required when refactoring in the same way programs written with parallel programming frameworks not supporting skeleton/parallel patterns (e.g. MPI). Assuming the latencies of `Stage1` and `Stage2` were respectively of 100 and 300 ms (per task), and that the program computes 1000 tasks, the completion time of the first version would have been around 300 s, while the one relative to the second version could have been around 100 s, with a 3x speedup.

Within ParaPhrase a refactorer tool supporting rewriting of FastFlow skeletons according to a set of rules including the ones listed above is being developed on top of Eclipse [35].

15.5 RISC-pbb

Last but not least, the FastFlow components used to implement the "core" skeletons, namely the emitter, collector, string of workers, etc. may be considered a sort of "RISC" set of parallel building blocks suitable to be used to build a number of different skeletons implementing a variety of parallel patterns: simple general purpose parallel patterns, domain specific parallel patterns and high level parallel patterns implementing parallel programming models as well. In [23] we define the `RISC-pbb` set of parallel building blocks corresponding to the base components of FastFlow and in [2] we show:

- how these building blocks may be used to implement high level parallel models (e.g. BSP or Google mapreduce)
- that they are suitable to implement domain specific parallel patterns (e.g. from soft computing/genetic algorithms or from numerical applications)
- that a set of rewriting rules stating equivalences among building block expressions exist, such that they may be used to refactor building block expressions to (i) improve performances and (ii) to target different parallel architectures.

16 Related Work

FastFlow aims at providing programmers of parallel applications with suitable tools enhancing their productivity while designing, developing and tuning parallel applications. In particular, it aims at supporting both the parallelization of existing applications–e.g. exploiting the accelerator facilities of FastFlow–and the development *ex novo* of parallel applications.

In this respect, FastFlow naturally competes with more famous and widely used parallel programming frameworks targeting the same kind of shared memory architectures such as OpenMP or Intel TBB.

OpenMP (http://openmp.org/wp/) naturally and seamlessly supports data parallel patterns, especially those expressed as parallelization of loops. It also supports the implementation of more complex, task parallel patterns, but this requires much more programming effort. In fact, different extensions (e.g. [36,37]) have been developed to increase the possibilities offered to parallelize applications with more complex task parallel patterns in OpenMP. Overall, even if the programming effort may by slightly larger in terms of lines of code, FastFlow still offers more–primitive or composite–patterns than OpenMP, with a comparable efficiency level and with better composition features.

Intel TBB (https://www.threadingbuildingblocks.org/) provides more primitive mechanisms with respect to both OpenMP and FastFlow. These mechanisms may be used to implement different patterns with different amounts of programming effort. However, the lower–with respect to FastFlow–level of abstraction provided to the parallel programmer by Intel TBB has a notable impact on both development and tuning time of parallel applications.

A number of different parallel programming environments from the algorithmic skeleton community have been developed and are currently maintained that directly compete with FastFlow. Muesli [25], Sketo [31], SKEPU [24], OSL [29], Skandium [30], just to mention some of them, all provide a set of parallel patterns as *algorithmic skeletons* suitable to be instantiated–alone or in composition–with business code parameters to implement parallel applications. Some of these frameworks only provide data parallel skeletons (e.g. SKEPU and Sketo). Other use different "host languages" (e.g. Skandium, which is provided as a Java library). In the former case, FastFlow offers a more structured and comprehensive parallel pattern set. In the latter case, the efficiency related to the usage of C++ rather than Java may represent a sensible advantage for FastFlow. Some of these programming frameworks also target clusters/networks of workstations (e.g. Muesli) or GPUs (e.g. Meusli and SKEPU). As explained in Sects. 15.2 and 15.3 FastFlow extensions may be used to orchestrate parallel computations on clusters of multi cores [5] and to direct data parallel computation to GPUs (using either CUDA or OpenCL kernels) or to a mix of GPU and CPU cores (using OpenCL) [26,38].

Different research teams participated to the development of FastFlow in different ways. Szũgyi and Pataki [39] developed a version of FastFlow fully exploiting the template mechanisms provided by C++ that eventually demonstrates good performances and much better type checking than the original FastFlow. Collins designed advanced tools suitable to automatically configure implementation parameters of FastFlow such that optimal performances may be automatically achieved [21]. Boob et al. [17] developed tools to distribute FastFlow computations on cloud resource exploiting typical virtual machine technologies. Goli and Gonzalèz-Velez demonstrated combined exploitation of CPU and GPU cores with FastFlow [26,27].

Last but not least, different groups are using FastFlow to implement different kinds of parallel applications and frameworks, including business transaction frameworks (http://fix8.org/), deep packet inspection tools (https://github.com/DanieleDeSensi/Peafowl) and rewriting-based calculus for the representation and simulation of biological systems (http://sourceforge.net/projects/cwcsimulator/).

17 Conclusions

We have introduced the basic features of the structured parallel programming framework FastFlow. We discussed this features by illustrating simple sample code and pointing at relevant papers and manuals hosting more detailed description of syntax and usage best practices for FastFlow.

FastFlow has been demonstrated to be very efficient in the execution of structured parallel applications, especially when fine grain parallelism has to be exploited. Different published papers show that the performances achieved with FastFlow applications/kernels on state-of-the-art multi core and distributed architectures are comparable or even better than the performances achieved executing the same applications/kernels using different, more traditional programming frameworks such as OpenMP or the Intel TBB library [10,13,15].

FastFlow is currently being adopted within two FP7 projects, ParaPhrase[9] and Repara[10]. In ParaPhrase, a methodology supporting the development of parallel applications where FastFlow parallel patterns are introduced through semi automatic refactoring of existing code is being developed. Preliminary results on industrial use cases show that the approach is feasible, FastFlow achieves comparable or slightly better performances that OpenMP and, last but not least, that the combined usage of refactoring and FastFlow technology greatly simplifies the development of efficient parallel applications, thus improving the time-to-market of parallel applications. Within Repara (just stared at the moment being), FastFlow is being adopted as the run time system orchestrating the execution of parallel applications on heterogeneous architectures including multicore CPUs, GPUs, FPGAs and DSPs.

FastFlow is an open source framework licensed under LGPL and available at SourceForge[11]. At the moment being the downloads from SourceForge are in the range of thousands. Different researchers and programmers are using it especially in those cases where fine grain parallelism needs to be exploited with the maximum efficiency.

References

1. FastFlow home page (2012). http://mc-fastflow.sourceforge.net
2. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. Int. J. Parallel Program. (2013). doi:10.1007/s10766-013-0273-6

[9] http://www.paraphrase-ict.eu.
[10] http://www.repara-project.eu/#!.
[11] http://sourceforge.net/projects/mc-fastflow/.

3. Aldinucci, M., Anardu, L., Danelutto, M., Torquati, M., Kilpatrick, P.: Parallel patterns + macro data flow for multi-core programming. In: Proceedings of International Euromicro PDP 2012: Parallel Distributed and network-based Processing, Garching, Germany. IEEE, February 2012
4. Aldinucci, M., Bracciali, A., Liò, P., Sorathiya, A., Torquati, M.: StochKit-FF: efficient systems biology on multicore architectures. In: Guarracino, M.R., et al. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 167–175. Springer, Heidelberg (2011)
5. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in FastFlow. In: Caragiannis, I., et al. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 47–56. Springer, Heidelberg (2013)
6. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: Proceedings of PDCS: International Conference on Parallel and Distributed Computing and Systems, pp. 955–962. IASTED, ACTA Press, Cambridge, Massachusetts (1999)
7. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating sequential programs using FastFlow and self-offloading. Technical report TR-10-03, Universitá di Pisa, Dipartimento di Informatica, Italy, February 2010
8. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 170–181. Springer, Heidelberg (2011)
9. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 662–673. Springer, Heidelberg (2012)
10. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting heterogeneous architectures via macro data flow. Parallel Process. Lett. **22**(2) (2012)
11. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, chap. 13. Wiley (2014)
12. Aldinucci, M., Danelutto, M., Meneghin, M., Kilpatrick, P., Torquati, M.: Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In: Chapman, B., Desprez, F., Joubert, G.R., Lichnewsky, A., Priol, T., Peters, F.J. (eds.) Parallel Computing: From Multicores and GPU's to Petascale (Proceedings of PARCO 2009, Lyon, France). Advances in Parallel Computing, vol. 19, pp. 273–280, Lyon, France. IOS Press, September 2009
13. Aldinucci, M., Drocco, M., Tordini, F., Coppo, M., Torquati, M.: Parallel stochastic simulators in system biology: the evolution of the species. In: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), pp. 410–419 (2013)
14. Aldinucci, M., Meneghin, M., Torquati, M.: Efficient Smith-Waterman on multi-core with FastFlow. In Danelutto, M., Gross, T., Bourgeois, J. (eds.) Proceedings of International Euromicro PDP 2010: Parallel Distributed and network-based Processing, Pisa, Italy. IEEE, February 2010
15. Aldinucci, M., Ruggieri, S., Torquati, M.: Decision tree building on multi-core using fastflow. Concurr. Comput. Practi. Experien. **26**(3), 800–820 (2014)
16. Aldinucci, M., Torquati, M., Meneghin, M.: FastFlow: efficient parallel streaming applications on multi-core. Technical report TR-09-12, Universitá di Pisa, Dipartimento di Informatica, Italy, September 2009

17. Boob, S., González-Vélez, H., Popescu, A.M.: Automated instantiation of heterogeneous FastFlow CPU/GPU parallel pattern applications in clouds. In: Proceedings of International Euromicro PDP 2014: Parallel Distributed and network-based Processing. IEEE Press (2014)
18. Campa, S., Danelutto, M., Torquati, M., González-Vélez, H., Popescu, A.: Towards the deployment of fastflow on distributed virtual architectures. In: Rekdalsbakken, W., Bye, R.T., Zhang, H. (eds.) ECMS, pp. 518–524. European Council for Modeling and Simulation (2013)
19. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
20. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
21. Collins, A., Fensch, C., Leather, H.: Optimization space exploration of the FastFlow parallel skeleton framework (2012). HLPGPU 2012, http://homepages.inf.ed.ac.uk/s1050857/collins-hlpgpu12.pdf
22. M. Danelutto, L. Deri, and D. De Sensi.: Network Monitoring on Multicores with Algorithmic Skeletons. In Volume 22: Applications, Tools and Techniques on the Road to Exascale Computing, Advances in Parallel Computing, pages 519–526. IOS Press, 2012. 2011, DOI: 10.3233/978-1-61499-041-3-519, Proc. of Intl. Parallel Computing (PARCO)
23. Danelutto, M., Torquati, M.: A RISC building block set for structured parallel programming. In: 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, 27 February–1 March, pp. 46–50. IEEE Computer Society (2013)
24. Dastgeer, U., Li, L., Kessler, C.: Adaptive implementation selection in the skepu skeleton programming library. In: Wu, C., Cohen, A. (eds.) APPT 2013. LNCS, vol. 8299, pp. 170–183. Springer, Heidelberg (2013)
25. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-gpu systems and clusters. IJHPCN **7**(2), 129–138 (2012)
26. Goli, M., González-Vélez, H.: Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures. In: 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, 27 February–1 March, pp. 148–156. IEEE Computer Society (2013)
27. Goli, M., González-Vélez, H.: N-body computations using skeletal frameworks on multicore cpu/graphics processing unit architectures: an empirical performance evaluation. Concurr. Comput. Practi. Experien. **26**(4), 972–986 (2014)
28. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exper. **40**(12), 1135–1160 (2010)
29. Legaux, J., Loulergue, F., Jubertie, S.: OSL: an algorithmic skeleton library with exceptions. In: Alexandrov, V.N., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J., Sloot, P.M.A. (eds.) ICCS. Procedia Computer Science, vol. 18, pp. 260–269. Elsevier (2013)
30. Leyton, M., Piquer, J.M.: Skandium: multi-core programming with algorithmic skeletons. In: Danelutto, M., Bourgeois, J., Gross, T. (eds.) PDP, pp. 289–296. IEEE Computer Society (2010)
31. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 644–653. Springer, Heidelberg (2004)

32. ParaPhrase. Final Pattern Definition Report (2013). http://www.paraphrase-ict.eu/Deliverables
33. ParaPhrase. Heterogeneous Implementation of Initial Generic Patterns (2013). http://www.paraphrase-ict.eu/Deliverables
34. ParaPhrase. Initial Implementation of Application-Specific Patterns (2013). http://www.paraphrase-ict.eu/Deliverables
35. ParaPhrase. Refactoring User Interfaces (2013). http://www.paraphrase-ict.eu/Deliverables
36. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with starss. IJHPCA **23**(3), 284–299 (2009)
37. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Self-adaptive ompss tasks in heterogeneous environments. In: IPDPS, pp. 138–149. IEEE Computer Society (2013)
38. Serban, T., Danelutto, M., Kilpatrick, P.: Autonomic scheduling of tasks from data parallel patterns to cpu/gpu core mixes. In: HPCS, pp. 72–79. IEEE (2013)
39. Szűgyi, Z., Pataki, N.: Generative version of the fastflow multicore library. Electr. Notes Theor. Comput. Sci. **279**(3), 73–84 (2011)