Parallel Programming Using FastFlow

(Version August 2014)

Massimo Torquati
Computer Science Department, University of Pisa
torquati@di.unipi.it

Summary

Chapter 1

Introduction

FastFlow is an open-source, structured parallel programming framework originally conceived to support highly efficient stream parallel computation while targeting shared memory multi-core. Its efficiency comes mainly from the optimised implementation of the base communication mechanisms and from its layered design. FastFlow provides the parallel applications programmer with a set of ready-to-use, parametric algorithmic skeletons modelling the most common parallelism exploitation patterns. The *algorithmic skeletons* provided by FastFlow may be freely nested to model more and more complex parallelism exploitation patterns.

FastFlow is an *algorithmic skeleton* programming framework developed and maintained by the parallel computing group at the Departments of Computer Science of the Universities of Pisa and Torino [9].

Fig. 1.1 presents a brief history of the *algorithmic skeleton* programming model. For a more in-depth description, please refer to [14].

A number of papers and technical reports discuss the different features of this programming environment [10, 5, 2], the results achieved when parallelizing different applications [18, 11, 15, 8, 1, 7, 6] and the use of FastFlow as *software accelerator*, i.e. as a mechanism suitable for exploiting unused cores of a multi-core architecture to speed up execution of sequential code [3, 4]. This work represents instead a tutorial aimed at describing the use of the main FastFlow skeletons and patterns and its programming techniques, providing a number of simple (and not so simple) usage examples.

This tutorial describes the basic FastFlow concepts and the main skeletons targeting **stream-parallelism**, **data-parallelism** and **data-flow parallelism**. It is still not fully complete: for example important arguments and FastFlow features such as the FastFlow memory allocator, the thread to core affinity mapping, GPGPU programming and distributed systems programming are not yet covered in this tutorial.

Algorithmic skeletons were introduced by M. Cole in late 1988 [12]. According to this original work

> The new system presents the user with a selection of independent "algorithmic skeletons", 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, in his algorithmic skeleton "manifesto" [13] 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 programmer with algorithmic skeletons. The definition of algorithmic skeletons evolved and eventually a widely shared definition emerged stating that:

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

Currently various frameworks exist that provide the application programmer 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://mc-fastflow.sourceforge.net/fastflow`, Skandium `https://github.com/mleyton/Skandium`. A recent survey of algorithmic skeleton frameworks may be found in [17].

Figure 1.1: Algorithmic skeletons [14]

Parallel patterns vs parallel skeletons:

Algorithmic skeletons and parallel design patterns have been developed in completely disjoint research frameworks but with almost the same objective: providing the programmer of parallel applications with an effective programming environment. The two approaches have many similarities addressed at different levels of abstraction. Algorithmic skeletons are aimed at directly providing pre-defned, efficient building blocks for parallel computations to the application programmer, whereas parallel design patterns are aimed at providing directions, suggestions, examples and concrete ways to program those building blocks in different contexts.
We want to emphasise the similarities of these two concepts and so, throughout this tutorial, we use the terms pattern and skeleton interchangeably. For an in-depth discussion on the similarities and the main differences of the two approaches please refer to [14].

This tutorial is organised as follow: Sec. 1.1 describes how to download the framework and compile programs, Sec. 2 recalls the FastFlow application design principles. Then, in Sec. 3 we introduce the main features of the stream-based parallel programming in FastFlow: how to wrap sequential code for handling a steam of data, how to generate streams and how to combine pipelines, farms and loop skeletons, how to set up farms and pipelines as software accelerator. In Sec. 4 we introduce data-parallel computations in FastFlow. ParallelFor, ParallelForReduce, ParallelForPipeReduce and Map are presented in this section. Finally, in Sec. 5, the macro data-flow programming model provided by the FastFlow framework is presented.

1.1 Installation and program compilation

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 the 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 (with the above `svn` command, a `fastflow` folder will be created in the current directory), the directory containing the `ff` sub-directory with the FastFlow header files should be named in the `-I` flag of `g++`, such that the header files may be correctly found.

For convenience may be useful to set the environment variable `FF_ROOT` to point to the FastFlow source directory. For example, if the FastFlow tarball (or the svn checkout) is extracted into your home directory with the name `fastflow`, you may set `FF_ROOT` as follows (bash syntax):

```
export FF_ROOT="$HOME/fastflow
```

Take into account that, since FastFlow is provided as a set of `.hpp` source files, the `-O3` switch is essential to obtain good performance. Compiling with no `-O3` compiler flag will lead to poor performance because the run-time code will not be optimised by the compiler. Also, remember that the correct compilation of FastFlow programs requires linking the `pthread` library (`-lpthread` flag).

```
g++ -std=c++11 -I$FF_ROOT -O3 test.cpp -o test -lpthread
```

1.1.1 Tests and examples

In this tutorial a set of simple usage examples and small applications are described. In almost all cases, the code can be directly copy-pasted into a text editor and then compiled as described above. For convenience all codes are provided in a separate tarball file `fftutorial_source_code.tgz` with a `Makefile`.

At the beginning of all tests and examples presented, there is included the file name containing the code ready to be compiled and executed, for example:

```
/* *************************** */
/* ******* hello_node.cpp ***** */

#include <iostream>
#include <ff/node.hpp>
using namespace ff;
struct myNode:ff_node {
....
```

means that the above code is in the file `hello_node.cpp`.

Chapter 2

Design principles

FastFlow has been originally 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.

More recently, within the activities of the EU FP7 STREP project "Para-Phrase"[1] the FastFlow framework has been extended in several ways. In particular, in the framework have been added:

- several new high-level patterns
- facilities to support coordinated execution of FastFlow program on distributed multi-core machines
- support for execution of new data parallel patterns on GPGPUs
- new low-level parallel building blocks allowing to build almost any kind of streaming graph and parallel patterns.

The whole programming framework has been incrementally developed according to a layered design on top of Pthread/C++ standard programming framework as sketched in Fig. 2.1).

The **Building blocks** layer provides the basics blocks to build (and generate via C++ header-only templates) the run-time support of core patterns. Typical objects at this level are queues (e.g. lock-free SPSC queues, bounded and unbounded), process and thread containers (as C++ classes) mediator threads/processes (extensible and configurable schedulers and gatherers). The shared-memory run-time support extensively uses non-blocking lock-free algorithms,

[1]http://paraphrase-ict.eu

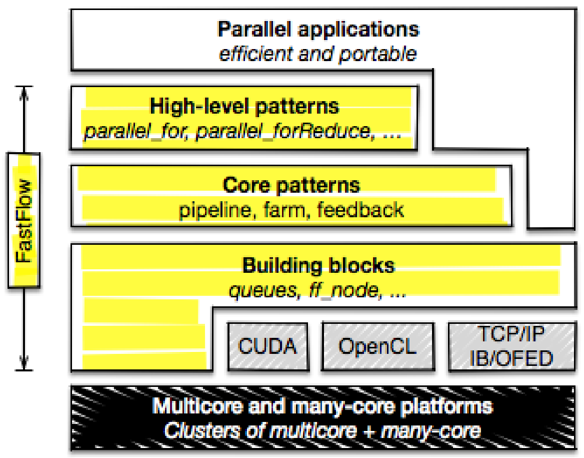


Figure 2.1: Layered FastFlow design

the distributed run-time support employs zero-copy messaging, the GPGPUs support exploits asynchrony and SIMT optimised algorithms.

The **Core patterns** layer provides a general data-centric parallel programming model with its run-time support, which is designed to be minimal and reduce to the minimum typical sources of overheads in parallel programming. At this level there are two patterns (*task-farm* and all its variants and *pipeline*) and one pattern-modifier (*feedback*). They make it possible to build very general (deadlock-free) cyclic process networks. They are not graphs of tasks, they are graphs of parallel executors (processes/threads). Tasks or data items flows across them according to the data-flow model. Overall, the programming model can be envisioned as a shared-memory streaming model, i.e. a shared-memory model equipped with message-passing synchronisations. They are implemented on top of building blocks.

The **High-level patterns** are clearly characterised in a specific usage context and are targeted to the parallelisation of sequential (legacy) code. Examples are exploitation of loop parallelism (*ParallelFor* and its variants), stream parallelism (*pipeline* and *task-farm*), data-parallel algorithms (*map*, *poolEvolution*, *stencil*, *StencilReduce*), execution of general workflows of tasks (*mdf* - Macro Data-Flow), etc. They are typically equipped with self-optimisation capabilities (e.g. load-balancing, grain auto-tuning, parallelism-degree auto-tuning) and exhibit limited nesting capability. Some of them targets specific devices (e.g. GPGPUs). They are implemented on top of core patterns.

Parallel application programmers are supposed to use FastFlow directly exploiting the parallel patterns available at the "High-level" or "Core" levels. In particular:

- defining sequential concurrent activities, by sub classing a proper FastFlow class, the `ff_node` (or `ff_minode` and `ff_monode`) class, and

- building complex stream parallel patterns by hierarchically composing sequential concurrent activities, *pipeline* patterns, *feedbacks*, *task-farm* patterns and their "specialised" versions implementing more complex parallel patterns.

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

- a *skeleton program abstraction* used to implement applications completely modelled 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` sub-classes, a skeleton (composition) modelling 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 to 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, if operating on stream (i.e. pipeline or task-farm based compositions), will have its own input and output channels. When an $f(x_j)$ has to be computed within the application, rather than writing code to call to the sequential f code, the programmer may insert asynchronous "offloading" calls for sending x_j to the accelerator skeleton. Later on, when the result of $f(x_j)$ has to be used, the code needed for "reading" accelerator results may be used to retrieve the computed values.

This second abstraction fully implements the "minimal disruption" principle stated by Cole in his skeleton manifesto [13], 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 Sec. 3.7).

Chapter 3

Stream parallelism

An application may operate on values organised as *streams*. A *stream* is a possibly infinite sequence of values, all of them having the same data type, e.g. a stream of images (not necessarily all having the same format), a stream of files, a stream of network packets, a stream of bits, etc.

A complex streaming application may be seen as a graph (or workflow) of computing modules (sequential or parallels) whose arcs connecting them bring streams of data of different types. The typical requirements of such a complex streaming application is to guarantee a given Quality of Service imposed by the application context. In a nutshell, that means that the modules of the workflow describing the application have to be able to sustain a given throughput.

There are many applications in which the input streams are primitive, because they are generated by external sources (e.g. HW sensors, networks, etc..) or I/O. However, there are cases in which streams are not primitive, but it is possible that they can be generated directly within the program. For example, sequential loops or iterators. In the following we will see how to generate a stream of data in FastFlow starting from sequential loops.

3.1 Stream parallel skeletons

Stream parallel skeletons are those natively operating on streams, notably *pipeline* and *task-farm* (or simply *farm*).

pipeline

The *pipeline* skeleton is typically used to model computations expressed in stages. In the general case, a pipeline may have more than two stages, and it can be built as a single pipeline with N stages or as pipeline of pipelines. Given a stream of input tasks

$$x_m, \ldots, x_1$$

the pipeline with stages

$$s_1, \ldots, s_p$$

computes the output stream

$$s_p(\ldots s_2(s_1(x_m))\ldots), \ldots, s_p(\ldots s_2(s_1(x_1))\ldots)$$

The parallel semantics of the pipeline skeleton ensures that all the stages will be execute in parallel. It is possible to demonstrate that the total time required to entirely compute a single task (latency) is close to the sum of the times required to compute the different stages. And, the time needed to output a new result (throughput) is close to time spent to compute a single task by the slowest stage in the pipeline.

task-farm

The *task-farm* (sometimes also called master-worker) is a stream parallel paradigm based on the replication of a purely functional computation. The *farm* skeleton is used to model embarrassingly parallel computations. The only functional parameter of a farm is the function f needed to compute the single task. The function f is stateless. Only under particular conditions, functions with internal state, may be used.

Given a stream of input tasks

$$x_m, \ldots, x_1$$

the farm with function f computes the output stream

$$f(x_m), \ldots, f_(x_1)$$

Its parallel semantics ensures that it will process tasks such that the single task latency is close to the time needed to compute the function f sequentially, while the throughput (only under certain conditions) is close to $\frac{f}{n}$ where n is the number of parallel agents used to execute the farm (called *workers*), i.e. its parallelism degree.

The concurrent scheme of a farm is composed of three distinct parts: the *Emitter*, the pool of workers and the *Collector*. The Emitter gets a farm's input tasks and distributes them to workers using a given scheduling strategy. The Collector collects tasks from workers and sends them to the farm's output stream.

3.2 FastFlow abstractions

In this section we describe the sequential concurrent activities (`ff_node`, `ff_minode`, `ff_monode` and `ff_dnode`), and the "core" skeletons *pipeline* and *task-farm* (see Fig 3.1) used as building blocks to model composition and parallel executions. The core skeletons are `ff_node` derived objects as well, so they can be nested and composed in almost any arbitrary way. The `feedback` pattern modifier can also be used in the *pipeline* and *task-farm* to build complex cyclic streaming networks.

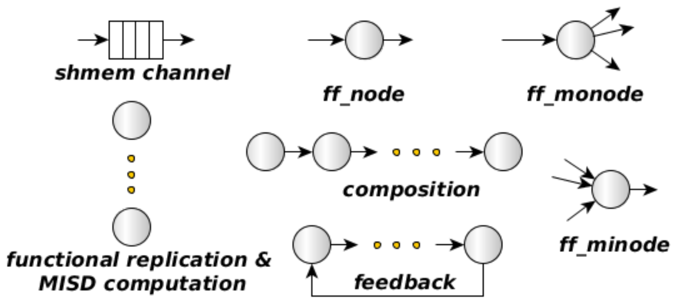


Figure 3.1: FastFlow basic abstractions: channel (implemented as a SPSC bounded or unbounded lock-free queue), single-input single-output node (*ff_node*), single-input multi-output (*ff_monode*) node, multi-input single-output node (*ff_minode*), composition (*pipeline*) and functional replication and MISD computation *task-farm*, loopback channel (*feedback*).

3.2.1 *ff_node*

The ff_node sequential concurrent activity abstraction provides a means to define a sequential activity (via its svc method) that a) processes data items appearing on a **single input channel** and b) delivers the related results onto a **single output channel**.

In a multi-core, a ff_node object is implemented as a **non-blocking thread** (or a set of non-blocking threads). This means that the number of ff_node(s) concurrently running should not exceed the number of logical cores of the platform at hand.

The ff_node class actually defines a number of methods, the following three virtual methods are of particular importance:

```
public:
 virtual void* svc(void *task) = 0; // encapsulates user's business code
 virtual int   svc_init() { return 0; }; // initialization code
 virtual void  svc_end()  {}  // finalization code
```

The first is the one defining the behaviour 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 sub-classes to implement initialisation code and finalisation code, respectively. Since the svc method is a pure virtual function, it *must* be overwritten.

A FastFlow ff_node can be defined as follow:

```
#include <ff/node.hpp>
using namespace ff;
struct myStage: ff_node {
   int svc_init() { // not mandatory
     // initialize the stage
     return 0; // returing non zero means error!
   }
   void *svc(void *task) {
     // business code here working on the input 'task'
     return task; // may return a task or EOS,GO_ON,GO_OUT,EOS_NOFREEZE
   }
   void svc_end() { // not mandatory
     // finalize the stage, free resources,...
   }
};
```

The `ff_node` behaves as a loop that gets an input task (coming from the input channel), i.e. the input parameter of the `svc` method, and produces one or more outputs, i.e. the return value of the `svc` method or the invocation of the `ff_send_out` method that can be called inside the `svc` method. The loop terminates either if the output provided or the input received is a special value: "End-Of-Stream" (EOS). The EOS is propagated across channels to the next `ff_node`.

Particular cases of `ff_node`s 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, to store them on disk or to send output packets into the network).

The simplified life cycle of an `ff_node` is informally described in the following pseudo-code:

```
do {
 if (svc_init() < 0) break;
 do {
   intask = input_stream.get();
   if (task == EOS) output_stream.put(EOS);
   else {
    outtask = svc(intask);
    output_stream.put(EOS);
   }
 } while(outtask != EOS);
 svc_end();
 termination = true;
 if (thread_has_to_be_frozen() == "yes") {
  freeze_the_thread_and_wait_for_thawing();
  termination = false;
 }
} while(!termination);
```

It is also possible to return from a `ff_node` the value *GO_ON*. This special value tells the run-time system (RTS) that there are no more tasks to send to the next stage and that the `ff_node` is ready to receive the next input task (if any). The *GO_ON* task is not propagated to the next stage. Other special values can be returned by an `ff_node`, such as *GO_OUT* and *EOS_NOFREEZE*, both of which are not propagated to the next stages and are used to exit the main node loop. The difference is that while *GO_OUT* allows the thread to be put to sleep (if it has been started with `run_then_freeze`), the second one instead allows to jump directly to the point where input channels are checked for receiving new tasks without having the possibility to stop the thread.

An `ff_node` cannot be started alone (unless the method `run()` and `wait()` are overwritten). Instead, it is assumed that `ff_node` derived objects are used as pipeline stages or task-farm workers. In order to show how to execute and wait for termination of `ff_node` derived objects, we provide here a simple wrapper class (in the next sections we will see that *pipeline* and *task-farm* are `ff_node` derived objects) :

```
 1 /* **************************** */
 2 /* ******* hello_node.cpp ***** */
 3
 4 #include <iostream>
 5 #include <ff/node.hpp>
 6 using namespace ff;
 7 struct myNode:ff_node {
 8   // called once at the beginning of node's life cycle
 9   int svc_init() {
10       std::cout << "Hello, I'm (re-)starting...\n";
11       counter = 0;
12       return 0; // 0 means success
13   }
14   // called for each input task of the stream, or until
15   // the EOS is returned if the node has no input stream
16   void *svc(void *task) {
17     if (++counter > 5) return EOS;
18     std::cout << "Hi! (" << counter << ")\n";
19     return GO_ON; // keep calling me again
20   }
21   // called once at the end of node's life cycle
22   void svc_end() { std::cout << "Goodbye!\n"; }
23
24
25   // starts the node and waits for its termination
26   int  run_and_wait_end(bool=false) {
27     if (ff_node::run() < 0) return -1;
28     return ff_node::wait();
29   }
30   // first sets the freeze flag then starts the node
31   int  run_then_freeze() { return ff_node::freeze_and_run(); }
32   // waits for node pause (i.e. until the node is put to sleep)
33   int  wait_freezing()   { return ff_node::wait_freezing();}
34   // waits for node termination
35   int  wait()            { return ff_node::wait();}
36
37   long counter;
38 };
39 int main() {
40   myNode mynode;
41   if (mynode.run_and_wait_end()<0)
42     error("running myNode");
43   std::cout << "first run done\n\n";
44   long i=0;
45   do {
46     if (mynode.run_then_freeze()<0)
47       error("running myNode");
48     if (mynode.wait_freezing())
49       error("waiting myNode");
50     std::cout << "run " << i << " done\n\n";
51   } while(++i<3);
52   if (mynode.wait())
53     error("waiting myNode");
54   return 0;
55 }
```

In this example, a `myNode` has been defined redefining all methods needed to execute the node, put the node to sleep (freeze the node), wake-up the node (thaw the node) and waiting for its termination.

Line 41 starts the computation of the node and waits for its termination syn-

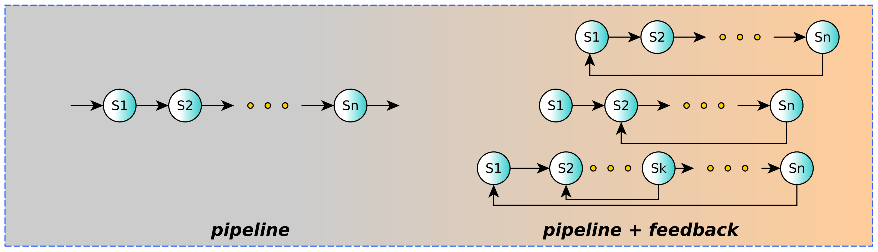


Figure 3.2: Possible *pipeline* and *pipeline+feedback* FastFlow skeleton versions.

chronously. At line 46 the node is started again but in this case the `run_then_freeze` method first sets the `ff_node::freezing` flag and then starts the node (i.e. creates the thread which execute the node). The `ff_node::freezing` flag tells the run-time that the node has to be put to sleep (*frozen* using our terminology) instead of terminating it when an `EOS` is received in input or it is returned by the node itself. The node runs asynchronously with respect to the main thread, so in order to synchronise the executions, the `wait_freezing` method is used (it waits until the node is frozen by the FastFlow run-time). When the `run_then_freeze` method is called again, since the node is frozen, instead of starting another thread, the run-time just "thaws" the thread.

3.2.2 *ff_pipe*

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

The standard way to create a pipeline of n different stages in FastFlow is to create n distinct `ff_node` objects and then pass them in the correct order to the `ff_pipe` constructor[1]. For example, the following code creates a 3-stage pipeline:

```
/* **************************** */
/* ******* hello_pipe.cpp ***** */

#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp> // defines ff_pipeline and ff_pipe
using namespace ff;
typedef long myTask;  // this is my input/output type
struct firstStage: ff_node {  // 1st stage
    void *svc(void*) {
        // sending 10 tasks to the next stage
  for(long i=0;i<10;++i)
      ff_send_out(new myTask(i));
  return EOS; // End-Of-Stream
    }
};
struct secondStage: ff_node { // 2nd stage
    void *svc(void *t) {
  return t; // pass the task to the next stage
    }
};
```

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

```
struct thirdStage: ff_node {  // 3rd stage
    void *svc(void *t) {
  myTask *task=reinterpret_cast<myTask*>(t);
  std::cout << "Hello I'm stage 3, I've received " << *task << "\n";
  return GO_ON;
    }
};
int main() {
  firstStage  _1;
  secondStage _2;
  thirdStage  _3;
  ff_pipe<myTask> pipe(&_1,&_2,&_3);
  if (pipe.run_and_wait_end()<0) error("running pipe");
  return 0;
}
```

To execute the pipeline it is possible to use the `run_and_wait_end()` method. This method starts the pipeline and synchronously waits for its termination [2].

For example, the following

The `ff_pipe` constructor also accepts as pipeline stage functions with signature `T*(*F)(T*,ff_node*const)`. This is because in many situations it is simpler to write a function than an `ff_node`. The FastFlow run-time wraps those functions in a `ff_node` object whose pointer is passed as second parameter to the function.

As an example consider the following 3-stage pipeline :

```
/* **************************** */
/* ******* hello_pipe2.cpp ***** */

#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp> // defines ff_pipeline and ff_pipe
using namespace ff;
typedef long myTask;  // this is my input/output type
myTask* F1(myTask *t,ff_node*const node) {     // 1st stage
  static long count = 10;
  std::cout << "Hello I'm stage F1, sending 1\n";
  return (--count > 0) ? new long(1) : (myTask*)EOS;
}
struct F2: ff_node {                           // 2nd stage
  void *svc(void *t) {
    myTask *task=reinterpret_cast<myTask*>(t);
    std::cout << "Hello I'm stage F2, I've received " << *task << "\n";
    return task;
  }
} F2;
myTask* F3(myTask *t,ff_node*const node) {     // 3rd stage
    myTask *task=reinterpret_cast<myTask*>(t);
    std::cout << "Hello I'm stage F3, I've received " << *task << "\n";
    return task;
}
int main() {
  ff_pipe<myTask> pipe(F1,&F2,F3);
  if (pipe.run_and_wait_end()<0) error("running pipe");
  return 0;
}
```

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 for the middle stage. Note that the first stage, generates 10 tasks and then an EOS.

Finally, it is also possible to add stages (of type `ff_node`) to a pipeline using the `add_stage` method as in the following sketch of code:

[2] It is also possible to run the pipeline asynchronously

```
int main() {
  ff_pipe<myTask> pipe(F1,&F2,F3);
  pipe.add_stage(new Stage4); // Stage4 is an ff_node derived object
  pipe.add_stage(new Stage5); // Stage5 is an ff_node derived object
  if (pipe.run_and_wait_end()<0) error("running pipe");
  return 0;
}
```

3.2.3 *ff_minode* and *ff_monode*

The *ff_minode* and the *ff_monode* are **multi-input** and **multi-output** FastFlow nodes, respectively. By using these kinds of node, it is possible to build more complex skeleton structures. For example, the following code implements a 4-stage pipeline where each stage sends some of the tasks received in input back to the first stage.

```
/* ******************************** */
/* ******* fancy_pipeline.cpp ***** */

/*
 *   Stage0 -----> Stage1 -----> Stage2 -----> Stage3
 *   ^  ^ ^             |             |              |
 *    \  \ \------------              |              |
 *     \  \-------------------------                 |
 *      \-------------------------------------------
 */
#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp> // defines ff_pipeline and ff_pipe
#include <ff/farm.hpp> // defines ff_minode and ff_monode
using namespace ff;
long const int NUMTASKS=20;
struct Stage0: ff_minode {
    int svc_init() { counter=0; return 0;}
    void *svc(void *task) {
        if (task==NULL) {
            for(long i=1;i<=NUMTASKS;++i)
                ff_send_out((void*)i);
            return GO_ON;
        }
  printf("Stage0 has got task %ld\n", (long)task);
        ++counter;
        if (counter == NUMTASKS) return EOS;
        return GO_ON;
    }
    long counter;
};
struct Stage1: ff_monode {
    void *svc(void *task) {
        if ((long)task & 0x1) // sends odd tasks back
            ff_send_out_to(task, 0);
        else ff_send_out_to(task, 1);
        return GO_ON;
    }
};
struct Stage2: ff_monode {
    void *svc(void *task) {
        // sends back even tasks less than ...
        if ((long)task <= (NUMTASKS/2))
            ff_send_out_to(task, 0);
        else ff_send_out_to(task, 1);
        return GO_ON;
    }
};
struct Stage3: ff_node {
    void *svc(void *task) {
```

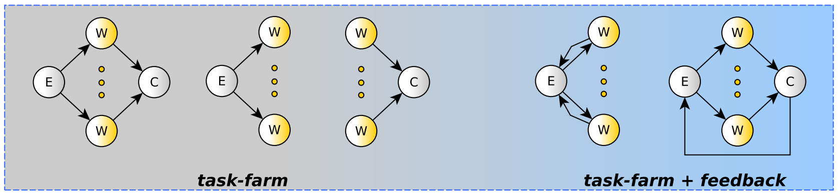


Figure 3.3: Possible *farms* and *farms+feedback* FastFlow skeleton versions.

```
        assert(((long)task & ~0x1) && (long)task>(NUMTASKS/2));
        return task;
    }
};
int main() {
    Stage0 s0; Stage1 s1; Stage2 s2; Stage3 s3;
    ff_pipe<long> pipe1(&s0,&s1);
    pipe1.wrap_around();
    ff_pipe<long> pipe2(&pipe1,&s2);
    pipe2.wrap_around();
    ff_pipe<long> pipe(&pipe2,&s3);
    pipe.wrap_around();
    if (pipe.run_and_wait_end()<0) error("running pipe");
    return 0;
}
```

To create a loopback channel we have used the `wrap_around` method available in both the `ff_pipe` and the `ff_farm` skeletons. More details on *feedback channels* in Sec. 3.5.

3.2.4 *ff_farm*

Here we introduce the other primitive skeleton provided in FastFlow, namely the `ff_farm` skeleton.

The standard way to create a *task-farm* skeleton with n workers is to create n distinct `ff_node` objects (workers node), pack them in a `std::vector` and then pass the vector to the `ff_farm` constructor. Let's have a look at the following "hello world"-like program:

```
/* ************************** */
/* ******* hello_farm.cpp ***** */

#include <vector>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
  void *svc(void *t) {
    std::cout << "Hello I'm the worker " << get_my_id() << "\n";
    return t;
  }
};
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_farm<> myFarm(Workers);
  if (myFarm.run_and_wait_end()<0) error("running myFarm");
  return 0;
}
```
This code basically defines a farm with `nworkers` workers processing 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 "pseudo round-robin one" (see Sec. 3.3). Workers are implemented by the `Worker` objects. These objects may represent sequential concurrent activities as well as further skeletons, that is either pipeline or farm instances. The above defined farm `myFarm` has the default Emitter (or scheduler) and the default Collector (or gatherer) implemented as separate concurrent activity. To execute the farm synchronously, the `run_and_wait_end()` method is used.

Compiling and running the above code we have:

```
ffsrc$ g++ -std=c++11 -I$FF_ROOT hello_farm.cpp -o hello_farm -pthread
ffsrc$ ./hello_farm 3
Hello I'm the worker 0
Hello I'm the worker 2
Hello I'm the worker 1
```
As you can see, the workers are activated only once because there is no input stream. The only way to provide an input stream to a FastFlow streaming network is to have the first component in the network generating a stream or by reading a "real" input stream. To this end, we may for example use the farm described above as a second stage of a pipeline skeleton whose first stage generates the stream and the last stage just writes the results on the screen:

```
/* ***************************** */
/* ******* hello_farm2.cpp ***** */

#include <vector>
#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
 int svc_init() {
   std::cout << "Hello I'm the worker " << get_my_id() << "\n";
   return 0;
 }
 void *svc(void *t) {
  return t;// just sends out tasks
 }
};
struct firstStage: ff_node {
 long size=10;
 void *svc(void *) {
  for(long i=0; i < size; ++i)
   ff_send_out(new long(i));// sends the task out
  return EOS;
 }
} streamGenerator;
struct lastStage: ff_node {
 void *svc(void *t) {
  const long &task=*reinterpret_cast<long*>(t);
  std::cout << "Last stage, received " << task << "\n";
  delete reinterpret_cast<long*>(t);
  return GO_ON; // this means ``no task to send out, let's go on...''
 }
} streamDrainer;
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_pipe<long> pipe(&streamGenerator,
          new ff_farm<>(Workers),
          &streamDrainer);
  if (pipe.run_and_wait_end()<0) error("running pipe");
  return 0;
}
```

In some cases, could be convinient to create a task-farm just from a single function (i.e. withouth defining the `ff_node`). Provided that the function has a proper signature, i.e. `T*(*F)(T*,ff_node*const)`, a very simple way to instanciate a farm is to pass the function and the number of workers you want to use (replication degree) in the `ff_farm` construct, as in the following sketch of code:

```
#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/farm.hpp>
using namespace ff;
struct myTask { .... }; // this is my input/output type

myTask* F(myTask *in,ff_node*const node) {...}
ff_farm<> farm(F, 3); // creates a farm executing 3 replicas of F
```

Defining Emitter and Collector in a farm

Both `emitter` and `collector` must be supplied as `ff_node` to the farm constructor. Considering the farm skeleton as a particular case of a a 3-stage pipeline (the first stage and the last stage are the Emitter and the Collector, respectively) we now want to re-write the previous example using only the FastFlow farm skeleton:

```
/* ***************************** */
/* ******* hello_farm3.cpp ***** */

#include <vector>
#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
  int svc_init() {
    std::cout << "Hello I'm the worker " << get_my_id() << "\n";
    return 0;
  }
  void *svc(void *t) {
   return t; // just sends out tasks
  }
};
struct firstStage: ff_node {
  long size=10;
  void *svc(void *) {
    for(long i=0; i < size; ++i)
      ff_send_out(new long(i));
    return EOS;
  }
} Emitter;
struct lastStage: ff_node {
  void *svc(void *t) {
      const long &task=*reinterpret_cast<long*>(t);
      std::cout << "Last stage, received " << task << "\n";
      delete reinterpret_cast<long*>(t);
      return GO_ON;
  }
} Collector;
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_farm<> farm(Workers, &Emitter, &Collector);
  if (farm.run_and_wait_end()<0) error("running farm");
  return 0;
}
```

The Emitter node encapsulates user's code provided in the `svc` method and the task scheduling policy which defines how tasks will be sent to workers. In the same way, the Collector node encapsulates the user's code and the task gathering policy defining how tasks have to be collected from the workers.

It is possible to redefine both scheduling and gathering policies of the FastFlow farm skeleton, please refer to to [16].

Farm with no Collector

We consider now a further case: a farm with the Emitter but without the Collector. Having no collector the farm's workers may either consolidates the results in the main memory or send them to the next stage (in case the farm is a pipeline stage) provided that the next stage is defined as `ff_minode` (i.e. multi-input node).

It is possible to remove the collector from the `ff_farm` by calling the method `remove_collector`. Let's see a simple example implementing the above case:

```
/* **************************** */
/* ******* hello_farm4.cpp ***** */

/*                        ________
 *                  _____|        |
 *                 |     | Worker |
 *    ________     |      ________   ____       ___________
 *   | Emitter|___|          .            |     | LastStage |
 *   |        |    |         .        ___ |-->  |           |
 *    ________     |         .            |      ___________
 *                 |      ________   ___
 *                  _____| Worker |
 *                       |        |
 *                        ________
 */
#include <vector>
#define HAS_CXX11_VARIADIC_TEMPLATES 1  // needed to use ff_pipe
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
  int svc_init() {
    std::cout << "Hello I'm the worker " << get_my_id() << "\n";
    return 0;
  }
  void *svc(void *t) {  return t; } // it does nothing, just sends out
      tasks
};
struct firstStage: ff_node {
  long size=10;
  void *svc(void *) {
    for(long i=0; i < size; ++i)
      ff_send_out(new long(i)); // sends the task into the output
          channel
    return EOS;
  }
} Emitter;
struct lastStage: ff_minode { // NOTE: multi-input node
  void *svc(void *t) {
      const long &task=*reinterpret_cast<long*>(t);
      std::cout << "Last stage, received " << task
                << " from " << get_channel_id() << "\n";
      delete reinterpret_cast<long*>(t);
      return GO_ON;
  }
} LastStage;
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_farm<> farm(Workers, &Emitter);
  farm.remove_collector(); // this removes the default collector
  ff_pipe<long> pipe(&farm, &LastStage);
  if (pipe.run_and_wait_end()<0) error("running pipe");
  return 0;
}
```

3.3 Tasks scheduling

Sending tasks to specific farm workers

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. The programmer may subclass the `ff_loadbalancer` and provide his own `selectworker()` method and then pass the new load balancer to the farm emitter, therefore implementing a farm with a user defined scheduling policy. To understand how to do this, please refer to [16].

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.

Let's see a simple example showing how to send the first and last task to a specific workers (worker 0).

```
/* ******************************** */
/* ******* ff_send_out_to.cpp ***** */

#include <vector>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
  void *svc(void *t) {
    long task = *reinterpret_cast<long*>(t);
    std::cout << "Worker " << get_my_id()
      << " has got the task " << task << "\n";
    return GO_ON;
  }
};
```

```
struct Emitter: ff_node {
  Emitter(ff_loadbalancer *const lb):lb(lb) {}
  ff_loadbalancer *const lb;
  const long size=10;
  void *svc(void *) {
    for(long i=0; i <= size; ++i) {
      if (i==0 || i == (size-1))
      lb->ff_send_out_to(new long(i), 0);
      else
      ff_send_out(new long(i));
    }
    return EOS;
  }
};
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  ff_farm<> farm(Workers);
  Emitter E(farm.getlb());
  farm.add_emitter(&E);     // add the specialized emitter
  farm.remove_collector(); // this removes the default collector
  if (farm.run_and_wait_end()<0) error("running farm");
  return 0;
}
```

Broadcasting a task to all workers

FastFlow supports the possibility to direct a task to all the workers in the farm. It is particularly useful if we want to process the task by workers implementing different functions (so called MISD farm). The broadcasting is achieved calling the `broadcast_task` method of the farm `ff_loadbalancer` object, in a very similar way to what we have already seen for the `ff_send_out_to` method in the previous section.

In the following a simple example.

```
/* *************************** */
/* ******* farm_misd.cpp ***** */

#include <vector>
#include <ff/farm.hpp>
using namespace ff;
struct WorkerA: ff_node {
  void *svc(void *t) {
      long task = *reinterpret_cast<long*>(t);
      std::cout << "WorkerA has got the task " << task << "\n";
      return t;
  }
};
struct WorkerB: ff_node {
  void *svc(void *t) {
      long task = *reinterpret_cast<long*>(t);
      std::cout << "WorkerB has got the task " << task << "\n";
      return t;;
  }
};
struct Emitter: ff_node {
  Emitter(ff_loadbalancer *const lb):lb(lb) {}
  ff_loadbalancer *const lb;
  const long size=10;
  void *svc(void *) {
    for(long i=0; i <= size; ++i)
  lb->broadcast_task(new long(i));
```

```
    return EOS;
  }
};
struct lastStage: ff_node {
  void *svc(void *task) {
      delete reinterpret_cast<long*>(task);
      return GO_ON;
  }
} Collector;
int main(int argc, char *argv[]) {
  assert(argc>1);
  int nworkers = atoi(argv[1]);
  assert(nworkers>=2);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers/2;++i)          Workers.push_back(new WorkerA);
  for(int i=nworkers/2;i<nworkers;++i) Workers.push_back(new WorkerB);
  ff_farm<> farm(Workers,nullptr,&Collector);
  Emitter E(farm.getlb());
  farm.add_emitter(&E);     // add the specialized emitter
  farm.remove_collector(); // this removes the default collector
  if (farm.run_and_wait_end()<0) error("running farm");
  return 0;
}
```

Using auto-scheduling

The default *farm* scheduling policy is "loosely round-robin" (or pseudo round-robin) meaning that the Emitter try to send the task in a round-robin fashion, but in case one of the workers' input queue is full, the Emitter does not wait till it can insert the task in the queue, but jumps to the next worker until the task can be inserted in one of the queues. This is a very simple policy but it doesn't work well if the tasks have very different execution costs.

FastFlow provides a suitable way to define a task-farm skeleton with the "auto-scheduling" policy. When using such policy, the workers "ask" for a task to be computed rather than (passively) accepting tasks sent by the emitter (explicit or implicit) according to some scheduling policy. This scheduling behaviour may be simply implemented by using the method `set_scheduling_ondemand()` of the `ff_farm` class, as in the following:

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

It is worth to remark that, this policy is able to ensure quite good load balancing property when the tasks to be computed exhibit different computational costs and up to the point when the Emitter does not become a bottleneck.

It is possible to increase the asynchrony level of the "request-reply" protocol between workers and Emitter simply by passing an integer value grater than zero to the `set_scheduling_ondemand()` function. By default the asynchrony level is 1.

3.4 Tasks ordering

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 by the Collector. This solution might introduce extra latency mainly because reordering checks have to be executed even in the case 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 specialised 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.
TBD: put an example here

3.5 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 the divide&conquer pattern using the task-farm.

The feedback channel in a farm or pipeline may be introduced by the `wrap_around` method on the interested skeleton. As an example, in the following code it is implemented a task-farm with default Emitter and Collector and with a feedback channel between the Collector and the Emitter:

```
Emitter myEmitter;
Collector myCollector;
ff_farm<> myFarm(W,&myEmitter,&myCollector);
myFarm.wrap_aroud();
...
```

Starting with FastFlow version 2.0.0, it is possible to use feedback channels not only at the outermost skeleton level. As an example, in the following we provide the code needed to create a 2-stage pipeline where the second stage is a farm without Collector and a feedback channel between each worker and the farm Emitter:

```
/* ***************************** */
/* ******* feedback.cpp ***** */

/*
 *                 -------------
 *                |             |
 *                |     --> F --
 *                v    |     .
 *  Stage0 --> Sched   |     .
 *                ^    |     .
 *                |     --> F --
 *                |             |
 *                 -------------
 */
#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/farm.hpp>
#include <ff/pipeline.hpp>
using namespace ff;
const long streamlength=20;
typedef std::function<long*(long*,ff_node*const)> fffarm_f;
long *F(long *in,ff_node*const) {
    *in *= *in;
    return in;
}
```

```
struct Sched: ff_node {
    Sched(ff_loadbalancer *const lb):lb(lb) {}
    void* svc(void *t) {
  const long task=*reinterpret_cast<long*>(t);
  int channel=lb->get_channel_id();
  if (channel == -1) {
      std::cout << "Task " << task << " coming from Stage0\n";
      return t;
  }
  std::cout << "Task " << task << " coming from Worker" << channel << "\
      n";
  return GO_ON;
    }
    void eosnotify(ssize_t) {
  // received EOS from Stage0, broadcast EOS to all workers
        lb->broadcast_task(EOS);
    }
    ff_loadbalancer *lb;
};
long *Stage0(long*, ff_node*const node) {
    for(long i=0;i<streamlength;++i)
  node->ff_send_out(new long(i));
    return (long*)EOS;
}
int main() {
    ff_farm<>   farm((fffarm_f)F, 3);
    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();
    // creates the pipeline
    ff_pipe<long> pipe(Stage0, &farm);
    if (pipe.run_and_wait_end()<0) error("running pipe");
    return 0;
}
```

In this case the Emitter node of the farm receives tasks both from the first stage of the pipeline and from farm's workers. To discern different inputs it is used the `ff_get_channel_id` method of the `ff_loadbalancer` object: inputs coming from workers have channel id greater than 0 (the channel id correspond to the worker id). The Emitter non-deterministic way processes input tasks giving higher priority to the tasks coming back from workers.

It is important to note how EOS propagation works in presence of loopback channels. Normally, the EOS is automatically propagated onward by the FastFlow run-time in order to implement pipeline-like termination. When internal loopback channels are present in the skeleton tree (and in general when there is multi-input nodes), the EOS is propagated only if the EOS message has been received from all input channels. In this case is useful to be notified when an EOS is received so that the termination can be controlled by the programmer. In the proposed example above, we want to propagate the EOS as soon as we receive it from the `Stage0` and then to terminate the execution only after having received all EOS from all workers.

3.6 Mixing farms pipelines and feedbacks

FastFlow *pipeline*, *task-farm* skeletons and the *feedback* pattern modifier can be nested and combined in many different ways. Figure 3.4 sketches some of the

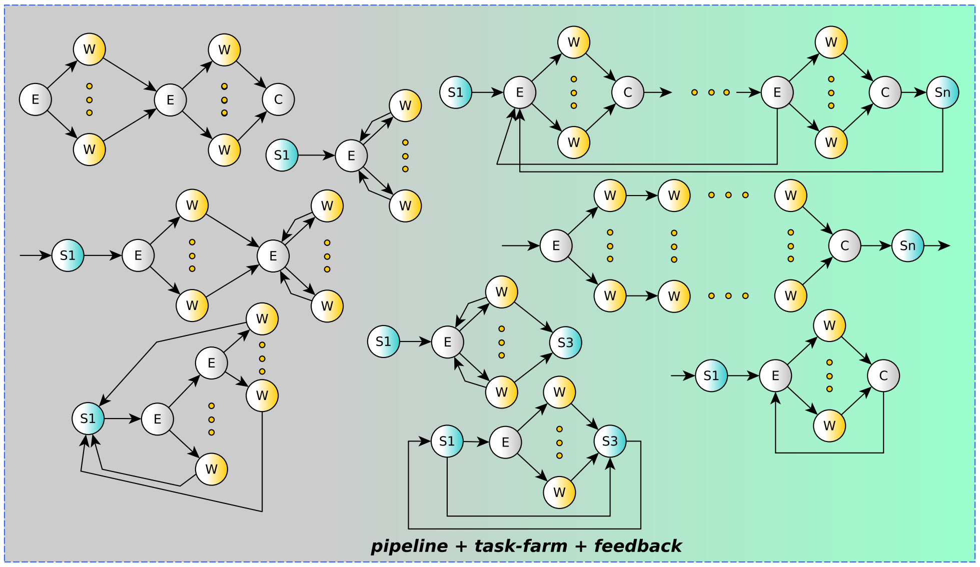


Figure 3.4: FastFlow's core patterns may be almost arbitrary composed

possible combinations that can be realised in a easy way.

3.7 Software accelerators

FastFlow can be used to accelerate existing sequential code without the need of completely restructuring the entire application using algorithmic skeletons. In a nutshell, programmers may identify potentially concurrent tasks within the sequential code and request their execution from an appropriate FastFlow pattern on the fly.

By analogy with what happens with GPGPUs and FPGAs used to support computations on the main processing unit, the cores used to run the user defined tasks through FastFlow define a software "accelerator" device. This device will run on the "spare" cores available. FastFlow accelerator is a "software device" that can be used to speedup portions of code using the cores left unused by the main application. From a more formal perspective, a FastFlow accelerator is defined by a skeletal composition augmented with an input and an output stream that can be, respectively, pushed and popped from outside the accelerator. Both the functional and extra-functional behaviour of the accelerator is fully determined by the chosen skeletal composition.

Using FastFlow accelerator mode is not that different from using FastFlow to write an application only using skeletons (see Fig. 3.5). The skeletons must be started as a software accelerator, and tasks have to be offloaded from the main program. A simple program using the FastFlow accelerator mode is shown below:

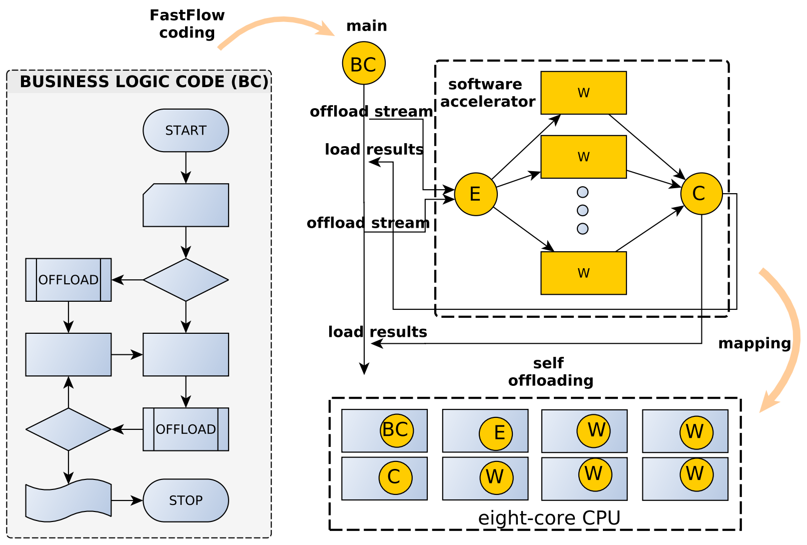


Figure 3.5: FastFlow software accelerator conceptual design

```
/* ***************************** */
/* ******* accelerator.cpp ***** */

#include <vector>
#include <ff/farm.hpp>
using namespace ff;
struct Worker: ff_node {
  void *svc(void *t) {
      long *task = reinterpret_cast<long*>(t);
      *task = pow(*task,3);
      return t;
  }
};
int main(int argc, char *argv[]) {
  assert(argc>2);
  int nworkers = atoi(argv[1]);
  int streamlen= atoi(argv[2]);
  std::vector<ff_node *> Workers;
  for(int i=0;i<nworkers;++i) Workers.push_back(new Worker);
  // task-farm with default Emitter and Collector
  ff_farm<> farm(Workers,nullptr,nullptr,true /* accelerator turned on*/
      );
  // now run the accelator asynchronously
  if (farm.run_then_freeze()<0) // farm.run() can also be used here
      error("running farm");
  long *result;
  for (long i=0;i<streamlen;i++) {
      long *task = new long(i);
      // the task is offloaded to the accelerator
      farm.offload(task);

      // do something smart here...
      for(volatile long k=0; k<10000; ++k);

      // try to get one result (in a non-blocking fashion)
      if (farm.load_result_nb((void**)&result)) {
          std::cerr << "[inside for loop] result= " << *result << "\n";
```

```
            delete result;
        }
    }
    farm.offload(EOS); // sending End-Of-Stream
    // get all remaining results syncronously, here is also
    // possible to use load_result_nb
    while(farm.load_result((void**)&result)) {
        std::cerr << "[outside for loop] result= " << *result << "\n";
        delete result;
    }
    farm.wait();  // wait for termination
    return 0;
}
```

The ”true” parameter in the farm constructor (the same is for the pipeline) is the one telling the run-time that the farm (or pipeline) has to be used as an accelerator. The idea is to start (or re-start) the accelerator and 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 the End-Of-Stream and eventually we `wait` for the completion of the computation of tasks in the accelerator or, we can `wait_freezing` to temporary stop the accelerator without terminating the threads in order to restart the accelerator afterwards.

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 **)` When the method is called, if no result is available, it returns ”false”, and might retry later on to see whether a result is ready.

3.8 Examples

In this sections we consider an images filtering application in which 2 image filters have to be applied to a stream of images. We prove different possible FastFlow implementation using both pipeline and task-farm skeletons. The different implementations described in the following are sketched in Fig. 3.6 (all but versions *img_farm2.cpp* and *img_farm3.cpp* are reported as source code in this tutorial): the starting sequential implementation (img.cpp), a 4-stage FastFlow pipeline implementation (img_pipe.cpp), the same version as before but the pipeline (3-stage) is implemented as a ”software accelerator” while the image reader stage is directly the main program (img_pipe2.cpp), a 4-stage FastFlow pipeline with the second and third stages implemented as task-farm skeletons (img_pipe+farm.cpp), a variant of the previous case, i.e. a 3-stage pipeline whose middle stage is a farm whose workers are 2-stage pipelines (img_farm+pipe.cpp), and finally the so called ”normal-form”, i.e. a single task-farm having the image reader stage collapsed with the farm Emitter and having the image writer stage collapsed with the farm collector (img_farm.cpp). The last 2 versions (not reported as source code here), i.e. *img_farm2.cpp* and *img_farm3.cpp*, are incremental optimizations of the base *img_farm.cpp* version

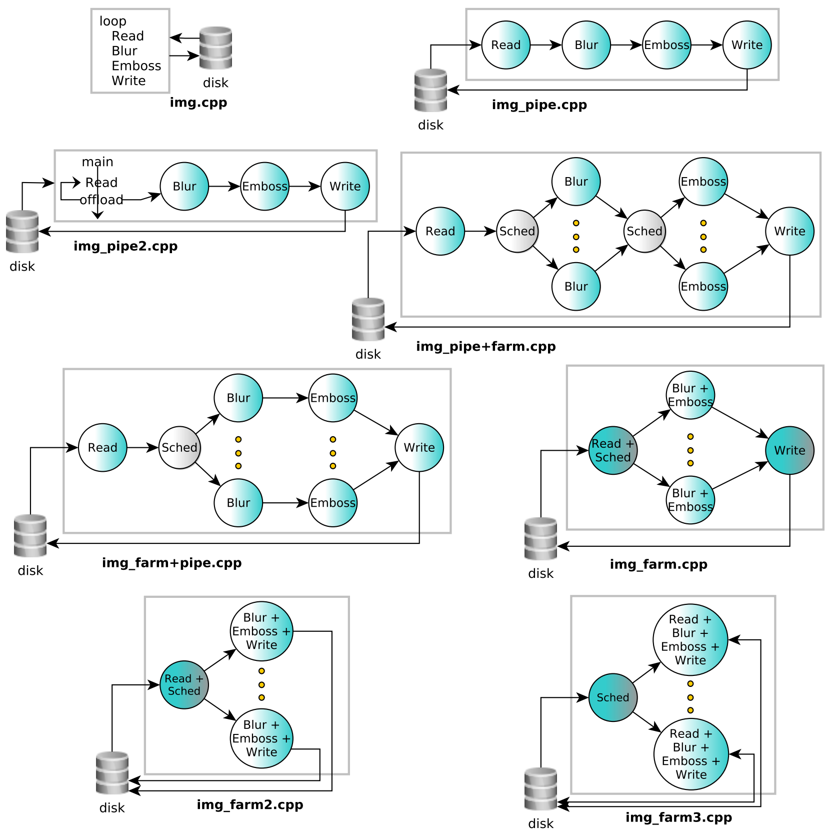


Figure 3.6: Different implementations of the image filtering example.

in which the input and output is performed in parallel in the farm's workers.

3.8.1 Images filtering

Let's consider a simple streaming application: two image filters (blur and emboss) have to be applied to a stream of input images. The stream can be of any length and images of any size (and of different format). For the sake of simplicity images' file are stored in a disk directory and the file names are passed as command line arguments of our simple application. The output images are stored with the same name in a separate directory.

The application uses the ImageMagick library[3] to manipulate the images and to apply the two filters. In case the ImageMagick library is not installed, please refer to the "Install from Source" instructions contained in the project web site. This is our starting sequential program:

```
/* ********************* */
```

[3]Project web site: http://www.imagemagick.org

```
/* ******* img.cpp ***** */

#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>
#include <Magick++.h>
using namespace Magick;

// helping functions: it parses the command line options
char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
int main(int argc, char *argv[]) {
    double radius = 1.0;
    double sigma  = 0.5;
    if (argc < 2) {
        std::cerr << "use: " << argv[0] << " [-r radius=1.0] [-s sigma
            =.5] image-files\n";
        return -1;
    }
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    if (r) { radius = atof(r); start+=2; argc-=2; }
    if (s) { sigma  = atof(s); start+=2; argc-=2; }

    InitializeMagick(*argv);

    long num_images = argc-1;
    assert(num_images >= 1);
    // for each image apply the 2 filter in sequence
    for(long i=0; i<num_images; ++i) {
        const std::string &filepath(argv[start+i]);
        std::string filename;

        // get only the filename
        int n=filepath.find_last_of("/");
        if (n>0) filename = filepath.substr(n+1);
        else     filename = filepath;

        Image img;
        img.read(filepath);

        img.blur(radius, sigma);
        img.emboss(radius, sigma);

        std::string outfile = "./out/" + filename;
        img.write(outfile);
        std::cout << "image " << filename
                  << " has been written to disk\n";
    }
    return 0;
}
```

Since the two filters may be applied in sequence to independent input images, we can compute the two filters in pipeline. So we define a 4-stage pipeline: the first stage read images from the disk, the second and third stages apply the two filters and the forth stage writes the resulting image into the disk (in a separate directory). The code for implementing the pipeline is in the following:

```
/* ************************* */
/* ******* img_pipe.cpp ***** */

/*
```

```
 *  Read --> Blur --> Emboss --> Write
 *
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>

#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/pipeline.hpp>
#include <Magick++.h>
using namespace Magick;
using namespace ff;
// this is the input/output task containing all information needed
struct Task {
 Task(Image *image, const std::string &name, double r=1.0,double s=0.5):
      image(image),name(name),radius(r),sigma(s) {};

 Image             *image;
 const std::string  name;
 const double       radius,sigma;
};

char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
// 1st stage
struct Read: ff_node {
    Read(char **images, const long num_images, double r, double s):
        images((const char**)images),num_images(num_images),radius(r),
            sigma(s) {}

    void *svc(void *) {
        for(long i=0; i<num_images; ++i) {
            const std::string &filepath(images[i]);
            std::string filename;

            // get only the filename
            int n=filepath.find_last_of("/");
            if (n>0) filename = filepath.substr(n+1);
            else     filename = filepath;

            Image *img = new Image;;
            img->read(filepath);
            Task *t = new Task(img, filename,radius,sigma);
            ff_send_out(t); // sends the task t to the next stage
        }
        return EOS; // computation completed
    }
    const char **images;
    const long num_images;
    const double radius,sigma;
};
// function executed by the 2nd stage
Task* BlurFilter(Task *in, ff_node*const) {
    in->image->blur(in->radius, in->sigma);
    return in;
}
// function executed by the 3rd stage
Task* EmbossFilter(Task *in, ff_node*const) {
    in->image->emboss(in->radius, in->sigma);
    return in;
}
// function executed by the 4th stage
Task* Write(Task* in, ff_node*const) {
    std::string outfile = "./out/" + in->name;
    in->image->write(outfile);
    std::cout << "image " << in->name << " has been written to disk\n";
    delete in->image;
    delete in;
    return (Task*)GO_ON;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "use: " << argv[0]
                  << " [-r radius=1.0] [-s sigma=.5] image-files\n";
        return -1;
    }
    double radius=1.0,sigma=0.5;
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    if (r) { radius = atof(r); start+=2; argc-=2; }
    if (s) { sigma  = atof(s); start+=2; argc-=2; }

    InitializeMagick(*argv);
    long num_images = argc-1;
    assert(num_images >= 1);

    ff_pipe<Task> pipe(
        new Read(&argv[start], num_images, radius, sigma), // 1st stage
        BlurFilter,                                          // 2nd stage
        EmbossFilter,                                        // 3rd stage
        Write);                                              // 4th stage
    if (pipe.run_and_wait_end()<0) { // executes the pipeline
        error("running pipeline\n");
        return -1;
    }
    return 0;
}
```

It is possible to instantiate the pipeline as a *software accelerator*. In the following we report only the code of the main function since it is the only part of the code that differs:

```
/* ************************** */
/* ******* img_pipe2.cpp ***** */

/*
 *      main
 *        |
 *      ->|
 *     |  Read
 *     |  offload --->  pipeline( BlurFilter, EmbossFilter, Write )
 *      - |
 *        |
 */
int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "use: " << argv[0]
                  << " [-r radius=1.0] [-s sigma=.5] image-files\n";
        return -1;
    }
    double radius=1.0,sigma=0.5;
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    if (r) { radius = atof(r); start+=2; argc-=2; }
    if (s) { sigma  = atof(s); start+=2; argc-=2; }

    InitializeMagick(*argv);

    long num_images = argc-1;
    assert(num_images >= 1);

    ff_pipe<Task> pipe(true,             // enable accelerator
                       BlurFilter,       // 2nd stage
                       EmbossFilter,     // 3rd stage
                       Write);           // 4th stage

    if (pipe.run_then_freeze()<0) { // start the pipeline
        error("running pipeline\n");
        return -1;
    }
    for(long i=0; i<num_images; ++i) {
        const std::string &filepath(argv[start+i]);
        std::string filename;

        // get only the filename
        int n=filepath.find_last_of("/");
        if (n>0) filename = filepath.substr(n+1);
        else     filename = filepath;

        Image *img = new Image;;
        img->read(filepath);
        Task *t = new Task(img, filename,radius,sigma);
        pipe.offload(t); // sends the task t to the pipeline
    }
    pipe.offload(EOS); // sends the End-Of-Stream

    if (pipe.wait()<0) { // wait for pipeline termination
        error("waiting pipeline\n");
        return -1;
    }
    return 0;
}
```

Now, since the same filter may be applied in parallel to independent input images, we can replace the second and third stage with two task-farm having the same previous stage as worker. This is safe because we know that we can replicate the function computing the filters: it is thread safe and has no internal shared state. The resulting code is:

```
/* ****************************** */
/* ******* img_pipe+farm.cpp ***** */

/*
 *                   --> Blur --             --> Emboss --
 *                  |           |           |             |
 *  Read --> Sched |--> Blur --|-- >Sched |--> Emboss --| --> Write
 *                  |           |           |             |
 *                   --> Blur --             --> Emboss --
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>
#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <Magick++.h>
using namespace Magick;
using namespace ff;
struct Task {
  Task(Image *image, const std::string &name, double r=1.0,double s=0.5)
      :
      image(image),name(name),radius(r),sigma(s) {};
  Image              *image;
  const std::string   name;
  const double        radius,sigma;
```

```
};
typedef std::function<Task*(Task*,ff_node*const)> fffarm_f;
char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
// 1st stage
struct Read: ff_node {
    Read(char **images, const long num_images, double r, double s):
        images((const char**)images),num_images(num_images),radius(r),
            sigma(s) {}

    void *svc(void *) {
        for(long i=0; i<num_images; ++i) {
            const std::string &filepath(images[i]);
            std::string filename;

            // get only the filename
            int n=filepath.find_last_of("/");
            if (n>0) filename = filepath.substr(n+1);
            else     filename = filepath;

            Image *img = new Image;;
            img->read(filepath);
            Task *t = new Task(img, filename,radius,sigma);
            std::cout << "sending out " << filename << "\n";
            ff_send_out(t); // sends the task t to the next stage
        }
        return EOS; // computation completed
    }
    const char **images;
    const long num_images;
    const double radius, sigma;
};
// function executed by the 2nd stage
Task* BlurFilter(Task *in, ff_node*const) {
    in->image->blur(in->radius, in->sigma);
    return in;
}
// function executed by the 3rd stage
Task* EmbossFilter(Task *in, ff_node*const) {
    in->image->emboss(in->radius, in->sigma);
    return in;
}
// function executed by the 4th stage
Task *Write(Task* in, ff_node*const) {
    std::string outfile = "./out/" + in->name;
    in->image->write(outfile);
    std::cout << "image " << in->name << " has been written to disk\n";
    delete in->image;
    delete in;
    return (Task*)GO_ON;
}
// 4th stage
struct Writer: ff_minode { // this is a multi-input node
    void *svc(void *task) {
        std::cout << "Writer received task\n";
        return Write(reinterpret_cast<Task*>(task), this);
    }
};

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "use: " << argv[0]
                  << " [-r radius=1.0]"
                  << " [-s sigma=.5]"
                  << " [ -n blurWrk=2]"
                  << " [ -m embossWrk=2] image-files\n";
        return -1;
    }
    double radius=1.0,sigma=0.5;
    int blurWrk = 2, embossWrk = 2;
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    char *n = getOption(argv, argv+argc, "-n");
    char *m = getOption(argv, argv+argc, "-m");
    if (r) { radius    = atof(r); start+=2; argc-=2; }
    if (s) { sigma     = atof(s); start+=2; argc-=2; }
    if (n) { blurWrk   = atoi(n); start+=2; argc-=2; }
    if (m) { embossWrk = atoi(m); start+=2; argc-=2; }

    InitializeMagick(*argv);

    long num_images = argc-1;
    assert(num_images >= 1);

    ff_farm<> blurFarm((fffarm_f)BlurFilter,blurWrk);
    blurFarm.remove_collector();
    blurFarm.set_scheduling_ondemand();   // set auto scheduling
    ff_farm<> embossFarm((fffarm_f)EmbossFilter,embossWrk);
    // this is needed because the previous farm does not has the
        Collector
    embossFarm.setMultiInput();
    embossFarm.remove_collector();
    embossFarm.set_scheduling_ondemand(); // set auto scheduling
    ff_pipe<Task> pipe(
        new Read(&argv[start], num_images, radius, sigma),  // 1st stage
        &blurFarm,                                           // 2nd stage
        &embossFarm,                                         // 3rd stage
        new Writer);                                         // 4th stage
    if (pipe.run_and_wait_end()<0) { // executes the pipeline
        error("running pipeline\n");
        return -1;
    }
    return 0;
}
```

A possible variant of the previous implementation, which uses only one scheduler is the following one:

```
/* ****************************** */
/* ******* img_farm+pipe.cpp ***** */

/*
 *                       --> Blur --> Emboss --
 *                      |                      |
 *    Read --> Sched -|--> Blur --> Emboss --| -->Write
 *                      |                      |
 *                       --> Blur --> Emboss --
 *
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>

#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <Magick++.h>
using namespace Magick;
using namespace ff;
struct Task {
    Task(Image *image, const std::string &name, double r=1.0,double s
        =0.5):
```

```
        image(image),name(name),radius(r),sigma(s) {};

    Image              *image;
    const std::string  name;
    const double        radius;
    const double        sigma;
};
char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
// 1st stage
struct Read: ff_node {
    Read(char **images, const long num_images, double r, double s):
        images((const char**)images),num_images(num_images),radius(r),
            sigma(s) {}

    void *svc(void *) {
        for(long i=0; i<num_images; ++i) {
            const std::string &filepath(images[i]);
            std::string filename;

            // get only the filename
            int n=filepath.find_last_of("/");
            if (n>0) filename = filepath.substr(n+1);
            else     filename = filepath;

            Image *img = new Image;;
            img->read(filepath);
            Task *t = new Task(img, filename,radius,sigma);
            std::cout << "sending out " << filename << "\n";
            ff_send_out(t); // sends the task t to the next stage
        }
        return EOS; // computation completed
    }

    const char **images;
    const long num_images;
    const double radius, sigma;
};
// function executed by the 2nd stage
Task* BlurFilter(Task *in, ff_node*const) {
    in->image->blur(in->radius, in->sigma);
    return in;

}
// function executed by the 3rd stage
Task* EmbossFilter(Task *in, ff_node*const) {
    in->image->emboss(in->radius, in->sigma);
    return in;
}
// function executed by the 4th stage
Task *Write(Task* in, ff_node*const) {
    std::string outfile = "./out/" + in->name;
    in->image->write(outfile);
    std::cout << "image " << in->name << " has been written to disk\n";
    delete in->image;
    delete in;
    return (Task*)GO_ON;
}
// 4th stage
struct Writer: ff_minode { // this is a multi-input node
    void *svc(void *task) {
        return Write(reinterpret_cast<Task*>(task), this);
    }
};

```

```cpp
int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "use: " << argv[0]
                  << " [-r radius=1.0]"
                  << " [-s sigma=.5]"
                  << " [ -n Wrk=2] image-files\n";
        return -1;
    }
    double radius=1.0,sigma=0.5;
    int Wrks = 2;
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    char *n = getOption(argv, argv+argc, "-n");
    if (r) { radius    = atof(r); start+=2; argc-=2; }
    if (s) { sigma     = atof(s); start+=2; argc-=2; }
    if (n) { Wrks      = atoi(n); start+=2; argc-=2; }

    InitializeMagick(*argv);

    long num_images = argc-1;
    assert(num_images >= 1);

    std::vector<ff_node*> W;
    for(int i=0;i<Wrks;++i) W.push_back(new ff_pipe<Task>(BlurFilter,
        EmbossFilter));
    ff_farm<> farm(W);
    farm.remove_collector();
    farm.set_scheduling_ondemand(); // set auto scheduling
    ff_pipe<Task> pipe(
       new Read(&argv[start], num_images, radius, sigma),  // 1st stage
       &farm,                                               // 2nd stage
       new Writer);                                         // 4th stage
    if (pipe.run_and_wait_end()<0) { // executes the pipeline
        error("running pipeline\n");
        return -1;
    }
    return 0;
}
```

The next step is to reduce the number of resources used. For example the farm Emitter can be used to read files from the disk, whereas the farm Collector for writing files to the disk. Furthermore, the blur and emboss filters may be computed sequentially using a single workers. This is the so called "normal form" obtained optimising the resource usage. The code implementing the normal form is the following:

```cpp
/* ************************ */
/* ****** img_farm.cpp ***** */

/*
 *
 *                  --> Blur+Emboss --
 *                 |                  |
 *    Read+Sched --|--> Blur+Emboss --|-->Collector+Write
 *                 |                  |
 *                  --> Blur+Emboss --
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>

#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <Magick++.h>
```

```
using namespace Magick;
using namespace ff;
struct Task {
    Task(Image *image, const std::string &name, double r=1.0,double s
        =0.5):
        image(image),name(name),radius(r),sigma(s) {};

    Image             *image;
    const std::string  name;
    const double       radius, sigma;
};
char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
// 1st stage
struct Read: ff_node {
    Read(char **images, const long num_images, double r, double s):
        images((const char**)images),num_images(num_images),radius(r),
            sigma(s) {}

    void *svc(void *) {
        for(long i=0; i<num_images; ++i) {
            const std::string &filepath(images[i]);
            std::string filename;

            // get only the filename
            int n=filepath.find_last_of("/");
            if (n>0) filename = filepath.substr(n+1);
            else     filename = filepath;

            Image *img = new Image;;
            img->read(filepath);
            Task *t = new Task(img, filename,radius,sigma);
            std::cout << "sending out " << filename << "\n";
            ff_send_out(t); // sends the task t to the next stage
        }
        return EOS; // computation completed
    }

    const char **images;
    const long num_images;
    const double radius,sigma;
};
// function executed by the 2nd stage
Task* BlurFilter(Task *in, ff_node*const) {
    in->image->blur(in->radius, in->sigma);
    return in;

}
// function executed by the 3rd stage
Task* EmbossFilter(Task *in, ff_node*const) {
    in->image->emboss(in->radius, in->sigma);
    return in;
}
struct BlurEmbossWrapper: ff_node {
    void *svc(void *t) {
        Task *task = reinterpret_cast<Task*>(t);
        return EmbossFilter(BlurFilter(task,this),this);
    }
};
// function executed by the 4th stage
Task *Write(Task* in, ff_node*const) {
    std::string outfile = "./out/" + in->name;
    in->image->write(outfile);
    std::cout << "image " << in->name << " has been written to disk\n";
    delete in->image;
    delete in;
    return (Task*)GO_ON;
}
// 4th stage
struct Writer: ff_minode { // this is a multi-input node
    void *svc(void *task) {
        return Write(reinterpret_cast<Task*>(task), this);
    }
};

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "use: " << argv[0]
        << " [-r radius=1.0]"
        << " [-s sigma=.5]"
        << " [ -n Wrks=2]"
        << " [ -m Wrk=2] image-files\n";
        return -1;
    }
    double radius=1.0,sigma=0.5;
    int Wrks = 2;
    int start = 1;
    char *r = getOption(argv, argv+argc, "-r");
    char *s = getOption(argv, argv+argc, "-s");
    char *n = getOption(argv, argv+argc, "-n");
    if (r) { radius    = atof(r); start+=2; argc-=2; }
    if (s) { sigma     = atof(s); start+=2; argc-=2; }
    if (n) { Wrks      = atoi(n); start+=2; argc-=2; }

    InitializeMagick(*argv);
    long num_images = argc-1;
    assert(num_images >= 1);
    std::vector<ff_node*> W;
    for(int i=0;i<Wrks;++i) W.push_back(new BlurEmbossWrapper);
    ff_farm<> farm(W,new Read(&argv[start], num_images, radius, sigma),
        new Writer);
    if (farm.run_and_wait_end()<0) { // executes the task-farm
        error("running pipeline\n");
        return -1;
    }
    return 0;
}
```

Chapter 4

Data parallelism

In data parallel computation, data structures (typically large) are partitioned among the number of concurrent resources each of which computes the same function on the assigned partition. In a nutshell, the input task, possibly but not necessarily coming from an input stream, is split into multiple sub-task each one computed in parallel and then collected together in one single output task. The computation on the sub-tasks may be completely independent (i.e. the sub-task computation uses data only coming from the the current sub-task) or dependent on previously computed data (non necessarily in the corresponding sub-task). The main goal of data parallel computation is to reduce the *completion time* of the single task to compute. It is important to note that, data decomposition using large sub-tasks, together with static assignment of such partitions to workers, may introduce *load imbalance* during the computation mainly because of the variable calculation time associated to distinct partitions. In general, it is possible to state that load balancing is a feature of anonymous task assignment, i.e. tasks to be computed are dynamically assigned to available computing resources, without a-priori correspondence between tasks and available computing resources. The task-farm paradigm is naturally based on this concept (where tasks are stream items), and for this reason quite often data-parallel computations are implemented on top of the task-farm skeleton by using the *auto-scheduling* policy (see Sec. 3.3). Many other scheduling strategies have been devised for balancing data-parallel computations, among these, the *work-stealing* scheduling strategy is of particular importance. It is worth to note here that, when the task-farm is used to implement an unbalanced data parallel computation, it is possible to "customise" it to use a *work-stealing* scheduling strategy. In other words, the task-farm pattern just models functional replication, while data partitioning and task-scheduling depends on the way the Emitter entity is implemented.

Well known and widely used data parallel skeletons are: *map reduce* and *stencil*.

4.1 Data parallel skeletons

In this section we describe *map*-like and *reduce*-like patterns, whereas the *stencil* pattern is not covered.

map

The simplest data parallel computation is the *map*, in which the concurrent workers operating on a partition of the input data structure (typically an N-dimensional array) are fully independent, i.e. each of them operates on its own local data only, without any cooperation with other computing resources. As an example consider the following piece of code:

```
const size_t N = 1<<15;
long A[N],B[N];
for(size_t i=1; i<N; ++i)
  A[i-1] = F(B[i]);
A[N-1] = F(B[0]);
```

If the function F has no internal state, each loop iteration may be computed independently since there are no true dependency among iterations. In this case we may apply the map pattern, splitting the two arrays A, B in $n = \frac{N}{nworkers}$ parts and assign each part to one worker. Each worker executes the same loop as above on a restricted index range. In principle, the execution time can be reduced by a factor of n if we consider a run-time with zero overhead, and as a particular case, if $nworkers = N$, the execution time is reduced from $O(N)$ to $O(1)$. It is worth to note that, the *map* skeleton may be simply implemented using a *task-farm* skeleton operating on a single input data. The Emitter split the input arrays and distributes the partitions to each worker. The workers apply the function F on the input partition, finally the Collector collects the workers' partitions in one single output task.

Loop-parallelism

A sequential iterative kernel with independent iterations is also known as a *parallel loop*. Parallel loops may be clearly parallelized by using the *map* or *farm* skeletons, but this typically requires a substantial re-factoring of the original loop code with the possibility to introduce bugs and not preserving *sequential equivalence*. Furthermore, the selection of the appropriate implementation skeleton together with a correct implementation of the sequential wrapper code is of foremost importance for obtaining the best performance.

For these reasons, in the FastFlow framework there are a set of data parallel patterns implemented on top of the basic FastFlow skeletons to ease the implementation of parallel loops.

4.2 FastFlow abstractions

In this section we describe the different parallel-for abstractions that are used to implement almost all data-parallel skeletons currently available in the FastFlow framework for multi-core systems.

4.2.1 *ParallelFor*

Here we introduce the FastFlow *ParallelFor* pattern that can be used to parallelize loops having independent iterations:

```
for(long idx=first; idx < last; idx += step) bodyF;
```

The class interface of the *ParallelFor* pattern is described in the `parallel_for.hpp` file. The constructor accepts two parameters:

```
ParallelFor(const long maxnworkers=FF_AUTO,bool spinWait=false);
```

the first parameter sets the maximum number of worker threads that can be used in the ParallelFor (that is the maximum concurrency degree), the second argument sets non-blocking run-time for parallel computations. At the beginning you may just leave the default parameters for these two arguments.

The `ParallelFor` object, encapsulates a number of `parallel_for` methods, which differentiate each other for the number of arguments they get and for the signatures of the function body. A single ParallelFor object can be used as many times as needed to run different parallel-for instances (different loop bodies). Nested invocations of ParallelFor methods are not supported.

The loop body may be a standard function or a C++11 lambda-function. A C++11 lambda-function is a new feature of the C++11 standard already supported by many compilers. They are unnamed closures (i.e. function objects that can be constructed and managed like data) that allow functions to be syntactically defined where and when needed. When lambda functions are built, they can capture the state of non-local variables named in the wrapped code either by value or by reference.

The following list presents the most useful parallel-for methods provided by the `ParallelFor` class:

```
parallel_for(first,last, bodyF, nworkers);// step=1,grain=FF_AUTO
parallel_for(first,last,step, bodyF, nworkers);// grain=FF_AUTO
parallel_for(first,last,step,grain, bodyF, nworekrs);
bodyF = F(const long idx);

parallel_for_thid(first,last,step,grain, bodyF, nworkers);
bodyF = F(const long idx,const int thid);
// thid is the id of the thread executing the body function

parallel_for_idx(first,last,step,grain, bodyF, nworkers);
bodyF = F(const long begin,const long end,const int thid);
```

Now, given the following sequential loop:

```
auto F = [](const long i) { return i*i;}
for(long i=1; i < 100; i +=2) A[i] = F(i);
```

we can write the following parallel-for:

```
ParallelFor pf;
auto F = [](const long i) { return i*i;}
pf.parallel_for(1,100,2,[&A](const long i) { A[i]=F(i);});
```

or by using the `parallel_for_idx` we have:

```
ParallelFor pf;
auto F = [](const long i) { return i*i;}
pf.parallel_for_idx(1,100,2,[&A](const long begin,const long end,
                                 const int thid){
 std::cout << "Hello I'm thread " << thid
  << " executing iterations (" << start <<"," << end <<")\n";
 for(long i=begin; i<end; i += 2) A[i]=F(i);
});
```

the `parallel_for_idx` is just a ”low-level” version of the `parallel_for`, where the internal loop, iterating over all iterations assigned to the worker, has to be written directly by the user. This may be useful when it is needed to execute a pre-computation (executed in parallel) before starting the execution of the loop iterations, or in general for debugging purposes.

It is important to remark that, when `spinWait` is set to `true` (see Sec. 4.2.1 for details), in some particular cases, the body function is called the first time with `begin==end==0` so it would be safe to test this condition at the beginning of the `parallel_for_idx` (i.e. `if (begin==end) return;`).

Let's now see a very simple usage example of the parallel-for:

```
/* ************************* */
/* ******* parfor1.cpp ***** */

#include <ff/parallel_for.hpp>
using namespace ff;

int main(int argc, char *argv[]) {
  assert(argc>1);
  long N = atol(argv[1]);
  long *A = new long[N];
  ParallelFor pf;
  // initialize the array A
  pf.parallel_for(0,N,[&A](const long i) { A[i]=i;});
  // do something on each even element of the array A
  pf.parallel_for(0,N,2,[&A](const long i) { A[i]=i*i;});
  // print the result
  for(long i=0;i<N;++i) std::cout << A[i] << " ";
  std::cout << "\n";
  return 0;
}
```

in this case, first the array `A` has been initialised using a parallel-for and then the square value of the even entries is computed in parallel over $N/2$ iterations.

Iterations scheduling

Three distinct iteration schedulings are currently possible in parallel-for computations:

1. **default static scheduling**: the iteration space is (almost) evenly partitioned in large contiguous chunks, and then they are statically assigned to workers, one chunk for each worker.

2. **static scheduling with interleaving** k: the iteration space is statically divided among all active workers in a round-robin fashion using

a stride of k. For example, to execute 10 iterations (from 0 to 9) using a concurrency degree of 2 and a stride $k = 3$, then the first thread executes iterations $0, 1, 2, 6, 7, 8$ and the second thread executes iterations $3, 4, 5, 9$. The default static scheduling is obtained setting a stride $k = iterationspace/nworkers$.

3. **dynamic scheduling with chunk** k: in this case no more than k contiguous iterations at a time are dynamically assigned to computing workers. As soon as a worker completes computation of one chunk of iterations, a new chunk (if available) is selected and assigned to the worker. The run-time tries to select as many as possible contiguous chunks in order to better exploit spatial locality. This allows to have a good trade-off between iterations affinity and load-balancing.

By default the *default static scheduling* is used. In general, the scheduling policy is selected by specifying the `grain` parameter of the `parallel_for` method. If the `grain` parameter is not specified or if its value is `0`, then the *default static scheduling* is selected. If `grain` is greater than zero, then the *dynamic scheduling* is selected with $k = grain$. Finally, to use the *static scheduling with interleaving* k the `parallel_for_static` method must be used with $k = grain$. Note that, if in the `parallel_for_static` the `grain` parameter is zero, than the *default static scheduling* policy is selected.

Summarising, in the following different scheduling strategies are selected according to the `grain` parameter:

```
pf.parallel_for(1,100,2,   bodyF); // default static sched.
pf.parallel_for(1,100,2, 0,bodyF); // default static sched.
pf.parallel_for(1,100,2,10,bodyF); // dynamic sched. with k=10
pf.parallel_for_static(1,100,2,10,bodyF);// static sched.
                                       // interleaving k=10
pf.parallel_for_static(1,100,2,0,bodyF); // default static sched.
```

threadPause **and** ***disableScheduler***

The `spinWait` parameter in the `ParallelFor` constructor enables non-blocking computation between successive calls of `parallel_for` methods. This means that if the parallel-for is not destructed and not used for a while, the worker threads will be active in a busy waiting loop. In this case it may be useful to "pause" the threads until the parallel-for pattern is used again. To attain this, the `threadPause` method can be used. The next time the parallel-for is used, the non-blocking run-time will be still active.

Another interesting option of the `ParallelFor` object, is the possibility to switch between two distinct run-time support for the scheduling of loop iterations:

1. *active scheduling* where an active non-blocking scheduler thread (the farm's Emitter) is used to implement the *auto-scheduling* policy;

2. *passive scheduling* where workers cooperatively schedule tasks via non-blocking synchronisations in memory.

Which one is better ? It is difficult to say, as it depends on many factors: parallelism degree, task granularity and underlying platform, among others. As a rule of thumb, on large multi-core and with fine-grain tasks active scheduling typically runs faster; on the other hand, when there are more threads than available cores the passive scheduler allows reduction of CPU conflicts obtaining better performance.

By default, if the number of active workers is less than the number of available cores, than the active scheduling is used. To dynamically switch between active (false) and passive (true) scheduling strategies the `disableScheduler` method can be used.

4.2.2 *ParallelForReduce*

The FastFlow `ParallelForReduce` is used to perform a parallel-for computation plus a reduction operation (by using a combiner function named *reduceF* in the following) over a sequence of elements of type *T*. In order to deterministically compute the result, the reduction function needs to be associative and commutative.

The constructor interface is:

```
ParallelForReduce<T>(const long max=FF_AUTO,bool spinwait=false);
```

where the template type `T` is the type of the reduction variable, the first parameter sets the maximum number of worker threads that can be used in the `ParallelForReduce`, the second argument sets non-blocking run-time.

The `ParallelForReduce` class provides all the parallel-for methods already provided by the `ParallelFor` class and a set of additional `parallel_reduce` methods:

```
parallel_reduce(var,identity, first,last,
                bodyF, reduceF, nworkers);// step=1,grain=FF_AUTO
parallel_reduce(var,identity, first,last,step,
                bodyF, reduceF, nworkers);// grain=FF_AUTO
parallel_reduce(var,identity, first,last,step,grain,
                bodyF, reduceF, nworekrs);
bodyF   =F(const long idx,T& var);
reduceF =R(T& var,const T& elem);

parallel_reduce_thid(var,identity, first,last,step,grain,
                     bodyF, reduceF, nworkers);
bodyF   =F(const long idx,T& var,const int thid);
reduceF =R(T& var,const T& elem);
// thid is the id of the thread executing the body function

parallel_reduce_idx(var,identity, first,last,step,grain,
                    bodyF, reduceF, nworkers);
bodyF   =F(T& var,const long begin,const long end,const int thid);
reduceF =R(T& var,const T& elem);
```

The `reduceF` function specified in all `parallel_reduce` methods, executes the reduction operation.

As an example, let's consider the simple case of the sum of an array's elements:

```
/* ************************ */
/* ****** arraysum.cpp ***** */

#include <iostream>
#include <ff/parallel_for.hpp>
using namespace ff;
const size_t SIZE= 1<<20;
int main(int argc, char * argv[]) {
  assert(argc > 1);
  int  nworkers  = atoi(argv[1]);
  // creates the array
  double *A = new double[SIZE];

  ParallelForReduce<double> pfr(nworkers);
  // fill out the array A using the parallel-for
  pfr.parallel_for(0,SIZE,1, 0, [&](const long j) { A[j]=j*1.0;});

  auto reduceF = [](double& sum, const double elem) { sum += elem; };
  auto bodyF   = [&A](const long j, double& sum) { sum += A[j]; };
  {
    double sum = 0.0;
    std::cout << "\nComputing sum with " << std::max(1,nworkers/2)
      << " workers, default static scheduling\n";
    pfr.parallel_reduce(sum, 0.0, 0L, SIZE,
                        bodyF, reduceF, std::max(1,nworkers/2));
    std::cout << "Sum = " << sum << "\n\n";
  }
  {
    double sum = 0.0;
    std::cout << "Computing sum with " << nworkers
      << " workers, static scheduling with interleaving 1000\n";
    pfr.parallel_reduce_static(sum, 0.0, 0, SIZE, 1, 1000,
                               bodyF, reduceF, nworkers);
    std::cout << "Sum = " << sum << "\n\n";
  }
  {
    double sum = 0.0;
    std::cout << "Computing sum with " << nworkers-1
      << " workers, dynamic scheduling chunk=1000\n";
    pfr.parallel_reduce(sum, 0.0, 0, SIZE, 1, 1000,
                        bodyF, reduceF, nworkers);
    std::cout << "Sum = " << sum << "\n\n";
  }
  delete [] A;
  return 0;
}
```

in this simple test, we used a parallel-for for initialising the array `A`, and 3 `parallel_reduce` calls for computing the final `sum` (the reduction variable) using the default static scheduling, the static scheduling with interleaving and the dynamic scheduling, respectively.

4.2.3 *ParallelForPipeReduce*

The `ParallelForPipeReduce` uses a different skeleton implementation of the `ParallelForReduce` pattern. The `ParallelForPipeReduce` computes a *map* function and a sequential *reduce* function in a *pipeline* fashion.

This pattern is useful in cases in which the reduce function has to be computed sequentially, for example because there are concurrent write accesses in some memory locations (so they have to be serialised using a lock), or because the reduction operator is not fully commutative. In these cases, the typical solution is to execute the map part (for example using a parallel-for) and then when the map is completed, execute the reduce part sequentially and this may be expensive because a full barrier (between the map and the reduce) is required. The `ParallelForPipeReduce` pattern allows execution of the *map* and *reduce* part in pipeline without any barriers.

The `ParallelForPipeReduce` pattern is more complex to use than the `ParallelForReduce` pattern because it requires to explicitly send the tasks to the reduce stage inside the body of the *map* function. The `ParallelForPipeReduce` class defined in the `parallel_for.hpp` file provides only 2 parallel-for methods:

```
parallel_for_idx(first,last,step,grain,
                 mapF, nworkers);
mapF    =F(const long begin,const long end,const int thid,
           ff_buffernode& node);

parallel_reduce_idx(first,last,step,grain,
                    mapF, reduceF, nworkers);
mapF    =F(const long begin,const long end,const int thid,
           ff_buffernode& node);
reduceF =R(T& var);
```

As an example, let's consider the same simple case implemented in the previous section, i.e. the computation of the sum of array's elements:

```
/* *************************** */
/* ******* arraysum2.cpp ***** */

#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/parallel_for.hpp>
using namespace ff;
const size_t SIZE= 1<<20;
int main(int argc, char * argv[]) {
  assert(argc > 1);
  int  nworkers  = atoi(argv[1]);
  // creates the array
  double *A = new double[SIZE];
  ParallelForPipeReduce<double*> pfpipe(nworkers,true);
  // init
  pfpipe.parallel_for_idx(0,SIZE,1,0,
                          [&A](const long start,const long stop,
            const int thid,ff_buffernode &) {
          for(long j=start;j<stop;++j) A[j] = j*1.0;
      });
  double sum = 0.0;
  auto mapF = [&A](const long start,const long stop,
                   const int thid,ff_buffernode &node) {
     // needed to avoid sending spurious lsum values to the reduce stage
     if (start==stop) return;
     double *lsum=new double(0.0); // allocate a task to send
     for(long i=start;i<stop;++i) *lsum += A[i];
     node.put(lsum); // sending the partial sum to the next reduce stage
  };
  auto reduceF = [&sum](double *lsum) { sum += *lsum; delete lsum; };
  std::cout << "Computing sum with " << nworkers
    << " workers, using the ParallelForPipeReduce and default sched.\n";
  pfpipe.parallel_reduce_idx(0, SIZE, 1, 0, mapF, reduceF);
  std::cout << "Sum = " << sum << "\n\n";
  delete [] A;
  return 0;
}
```

4.2.4 *ff_map*

The FastFlow *map* parallel pattern is implemented as an `ff_node` abstraction and a `ParallelForReduce` pattern. The idea is that, the *map* pattern is just an interface to a parallel-for on multi-core platforms, while it provides a simple abstraction for targeting multi-GPGPUs (both OpenCL and CUDA) and, in the future versions, FPGAs on many-core platforms.

Since the `ff_map` pattern is an `ff_node`, it may be used as a pipeline stage and as a farm worker.

```
/* *************************** */
/* ******* hello_map.cpp ***** */

#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/parallel_for.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <ff/map.hpp>
using namespace ff;
const long SIZE = 100;
// task type
typedef std::pair<std::vector<long>,std::vector<long> > task_t;
// this is the farm worker
struct mapWorker: ff_Map<> {
  void *svc(void*) {
      task_t *task = new task_t;
      task->first.resize(SIZE);
      task->second.resize(SIZE);
      const int myid = get_my_id();
      ff_Map<>::parallel_for(0,SIZE,[myid,&task](const long i) {
          task->first.operator[](i)  = i + myid;
          task->second.operator[](i) = SIZE-i;
        },2); // just starts 2 worker threads
      ff_send_out(task);
      return EOS;
  }
};
// this is the second stage
struct mapStage: ff_Map<> {
  mapStage():ff_Map<>(ff_realNumCores()) {} // it uses all real cores
  void *svc(void *t) {
     task_t *task = reinterpret_cast<task_t*>(t);
     // this is the parallel_for provided by the ff_Map class
     ff_Map<>::parallel_for(0,SIZE,[&task](const long i) {
          task->first.operator[](i) += task->second.operator[](i);
       });
     for(size_t i=0;i<SIZE;++i)
       std::cout << task->first.operator[](i) << " ";
     std::cout << "\n";
     return GO_ON;
  }
};
int main() {
  std::vector<ff_node*> W;
  W.push_back(new mapWorker);
  W.push_back(new mapWorker);
  ff_farm<> farm(W);
  farm.cleanup_workers(); // memory is freed at farm termination
  mapStage stage;
  ff_pipe<task_t> pipe(&farm,&stage);
  if (pipe.run_and_wait_end()<0)
    error("running pipe");
  return 0;
}
```

in the above test, we have a *pipeline* of two stage, the first stage is a *task-farm* having two workers defined as *map*, the second stage is a *map*. The farm's workers initialise and generate in the output stream pairs of vector, the single pair of vector is summed in the second stage.

Why use the `ff_Map` instead of using directly a `ParallelFor` in a sequential `ff_node`? The big difference is only in the fact that in case of `ff_Map` the run-time knows that the `ff_node` is internally parallel whereas if the parallel-for is used inside the `svc` method of a sequential `ff_node` this information is completely transparent to the FastFlow run-time thus not allowing any kind of optimisation (as for example, removing the scheduler or applying a better thread-to-core mapping).

4.3 Examples

4.3.1 Matrix multiplication

Let's consider the standard ijk matrix multiplication algorithm. We have three nested loops that can be potentially parallelised (note that the internal k-loop is not a plain parallel for loop). In the following code we apply the *ParallelFor* pattern to the outermost loop (i-loop).

```
/* ************************ */
/* ******* matmul.cpp ***** */

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#include <ff/parallel_for.hpp>
using namespace ff;
int  PFWORKERS =1;       // parallel_for parallelism degree
int  PFGRAIN   =0;       // default static scheduling of iterations

void random_init (long M, long N, long P, double *A, double *B) { }

// triple nested loop (ijk) implementation
void seqMatMult(long m, long n, long p,
                const double* A, const long AN,
                const double* B, const long BN,
                double* C, const long CN)  {

    for (long i = 0; i < m; i++)
        for (long j = 0; j < n; j++) {
            C[i*CN+j] = 0.0;
            for (long k = 0; k < p; k++)
                C[i*CN+j] += A[i*AN+k]*B[k*BN+j];
        }
}
void PFMatMultI(long m, long n, long p,
                 const double* A, const long AN,
                 const double* B, const long BN,
                 double* C, const long CN)  {

    ParallelFor pf(PFWORKERS);
    pf.parallel_for(0,m,[A,B,CN,AN,BN,p,n,&C](const long i) {
        for (long j = 0; j < n; j++) {
            C[i*CN+j] = 0.0;
            for (long k = 0; k < p; k++) {
                C[i*CN+j] += A[i*AN+k]*B[k*BN+j];
            }
        }
        }); // it uses all PFWORKERS
}

int main(int argc, char* argv[]) {
    if (argc < 4) {
        printf("\n\tuse: %s M N P pfworkers:chunksize [check=0]\n", argv
            [0]);
        printf("\t       A is M by P\n");
        printf("\t       B is P by N\n");
        printf("\t check!=0 executes also the sequential ijk loops for
            checking the result\n\n");
        return -1;
    }

    long M = parse_arg(argv[1]);
    long N = parse_arg(argv[2]);
    long P = parse_arg(argv[3]);
    if (argc >= 5) {
        std::string pfarg(argv[4]);
        int n = pfarg.find_first_of(":");
        if (n>0) {
            PFWORKERS = atoi(pfarg.substr(0,n).c_str());
            PFGRAIN   = atoi(pfarg.substr(n+1).c_str());
        } else PFWORKERS = atoi(argv[5]);
    }
    if (argc >= 6) check = (atoi(argv[5])?true:false);

    const double *A = (double*)malloc(M*P*sizeof(double));
    const double *B = (double*)malloc(P*N*sizeof(double));
    assert(A); assert(B);

    random_init(M, N, P, const_cast<double*>(A), const_cast<double*>(B))
        ;
    double *C      = (double*)malloc(M*N*sizeof(double));

    PFMatMultI(M, N, P, A, P, B, N, C);

    free((void*)A); free((void*)B); free(C);
    return 0;
}
```

4.3.2 Dot product

The "dot" product is a simple example of a map pattern combined with a reduction operation. The map is the initial pair wise multiplication of vector elements, and the reduction is the summation of the results of that multiplication.

More formally, given two arrays A and B each with n elements, the dot product $A \times B$ is the resulting scalar given by $\sum_{i=0}^{n-1} A[i] \times B[i]$.

```
/* ************************ */
/* ******* dotprod.cpp ***** */

```

```
#include <iostream>
#include <iomanip>
#include <ff/parallel_for.hpp>
using namespace ff;
int main(int argc, char * argv[]) {
  if (argc < 5) {
      std::cerr << "use: " << argv[0]
                << " ntimes size pfworkers G\n";
      std::cerr << " example:\n  " << argv[0] << " 2 100000 8 0\n\n";
      return -1;
  }
  const double INITIAL_VALUE = 1.0;
  int  NTIMES    = atoi(argv[1]);
  long arraySize = atol(argv[2]);
  int  nworkers  = atoi(argv[3]);
  long chunk     = atol(argv[4]);

  assert(nworkers>0); assert(arraySize>0);

  // creates the array
  double *A = new double[arraySize];
  double *B = new double[arraySize];

  ParallelForReduce<double> pfr(nworkers,true);

  // initialize the arrays A and B
  pfr.parallel_for(0,arraySize,1, chunk, [&A,&B](const long j){
    A[j]=j*3.14; B[j]=2.1*j;
  });
  // final reduce function
  auto Fsum = [](double& v, const double& elem) { v += elem; };
  double sum; // reduction variable
  for(int z=0;z<NTIMES;++z) { // repeats the computation NTIMES
    sum = INITIAL_VALUE;
    pfr.parallel_reduce(sum, 0.0, 0, arraySize,1, chunk,
       [&A,&B](const long i, double& sum) { sum += A[i]*B[i]; },
       Fsum);
  }
  std::cout << "Result = " << std::setprecision(10) << sum << "\n";
  delete [] A;  delete [] B;
  return 0;
}
```

4.3.3 Mandelbrot fractal

A Mandelbrot set M, is defined by the set of complex numbers c such that:

$$M = \{c : |M_k(c)| < \infty, \forall k \in N\}$$

where

$$\begin{cases} M_0(c) = 0 \\ M_{k+1}(c) = M_k(c)^2 + c \end{cases}$$

A point c belongs to the Mandelbrot set if and only if:

$$|M_k(c)| <= 2, \forall k \in N$$

The following program, which uses the *ParallelForPipeReduce* pattern for both computing the set line by line (*Map* part) and for displaying the line just computed (*Reduce* part), displays the Mandelbrot set in a X window. The result is shown in Fig 4.1.

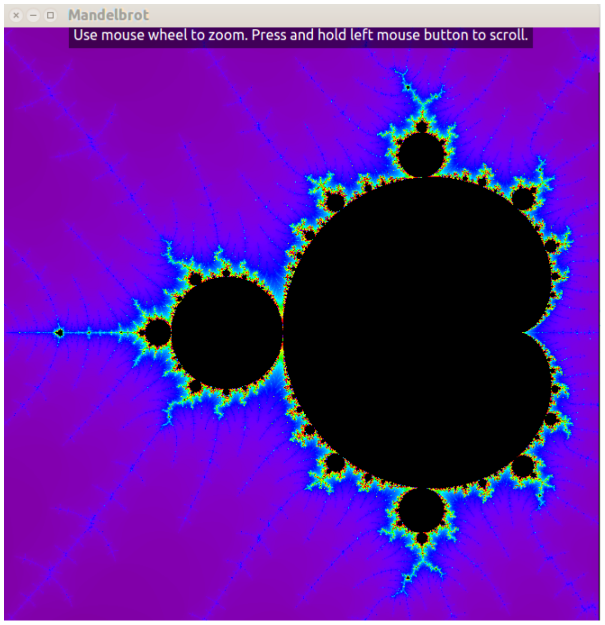

Mandelbrot
Use mouse wheel to zoom. Press and hold left mouse button to scroll.

Figure 4.1: Mandelbrot set: size=800, niterations=400

```
/* *********************** */
/* ******* mandel.cpp ***** */

#include <iostream>
#include <cstdio>
#include <cmath>
#include <cassert>
#include <ff/utils.hpp>
#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/parallel_for.hpp>
#include "lib/marX2.h"

const int DIM=800; // default dimension
const int ITERATION=1024; // default iteration
const double init_a=-2.125, init_b=-1.5, range=3.0;
using namespace ff;

// task type
struct ostream_t {
    ostream_t(const int dim):M(dim),line(-1) {}
    std::vector<unsigned char> M;
    int line;
};
int main(int argc, char **argv) {
    int dim = DIM, niter = ITERATION, retries=1, nworkers, chunk=0;
    bool scheduler = true;
    if (argc<3) {
      printf("\nUsage: %s <nworkers> <size> [niterations=1024]"
             " [retries=1] [chunk=0] [0|1]\n\n\n", argv[0]);
      return -1;
    }
    nworkers = atoi(argv[1]);
    dim = atoi(argv[2]);
    if (argc>=4) niter  = atoi(argv[3]);
    if (argc>=5) retries = atoi(argv[4]);
    if (argc>=6) chunk   = atoi(argv[5]);
    if (argc>=7)  scheduler = atoi(argv[6]);
    double step = range/((double) dim);
    double runs[retries];

    SetupXWindows(dim,dim,1,NULL,"ParallelForPipeReduce Mandelbrot");

    ParallelForPipeReduce<ostream_t*> pfr(nworkers,true);

    if (scheduler) pfr.disableScheduler(false);
    // map lambda
    auto Map = [&](const long start, const long stop,
                   const int thid, ff_buffernode &node) {
  for(int i=start;i<stop;i++) {
      double im=init_b+(step*i);
      ostream_t *task=new ostream_t(dim);
      for (int j=0;j<dim;j++) {
    double a=init_a+step*j;
    double b=im;
    const double cr = a;
    int k=0;
    for ( ; k<niter;k++) {
        const double a2=a*a;
        const double b2=b*b;
        if ((a2+b2)>4.0) break;
        b=2*a*b+im;
        a=a2-b2+cr;
    }
    task->M[j]= (unsigned char) 255-((k*255/niter));
      }
      task->line = i;
      node.put(task);
  }
    };
    // reduce lambda
    auto Reduce = [&](ostream_t *task) {
      ShowLine(task->M.data(),task->M.size(),task->line);
      delete task;
    };
    double avg=0.0, var=0.0;
    for (int r=0;r<retries;r++) {
  ffTime(START_TIME);
  pfr.parallel_reduce_idx(0,dim,1,chunk,Map,Reduce);
  ffTime(STOP_TIME);

  avg += runs[r] = ffTime(GET_TIME);
  std::cout << "Run [" << r << "] done, time = " << runs[r] << "\n";
    }
    avg = avg / (double) retries;
    for (int r=0;r<retries;r++)
  var += (runs[r] - avg) * (runs[r] - avg);
    var /= retries;
    std::cout << "The average time (ms) on " << retries
              << " experiments is " << avg
        << " Std. Dev. is " << sqrt(var) << "\n";

    std::cout << "Press a button to close the windows\n";
    getchar();
    CloseXWindows();
    return 0;
}
```

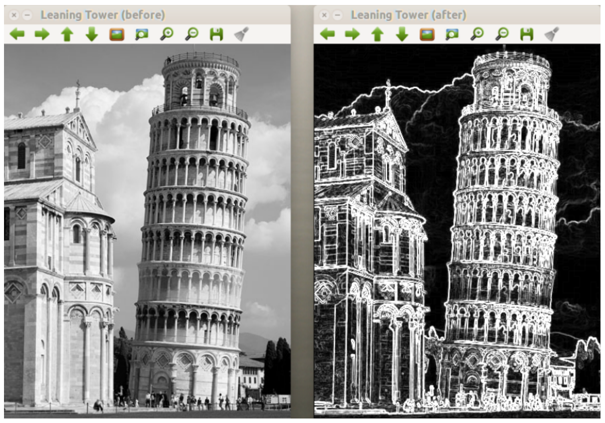


Figure 4.2: Sobel filter

4.3.4 Sobel filter

Let's consider a very simple usage example in which the `ff_map` is used as pipeline stage for computing the Sobel filter (see Fig. 4.2) on a stream of input images. The first stage reads image from disk, allocates memory and then sends a task to the second stage (the Map one) which computes the Sobel filter in parallel using a parallel-for. Finally the image is written into the local disk. In the following code we used OpenCV[1] for reading and writing images and for converting them to grayscale.

```
/* ************************ */
/* ******* ffsobel.cpp ***** */
#include<iostream>
#include<cmath>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
#define HAS_CXX11_VARIADIC_TEMPLATES 1
#include <ff/pipeline.hpp>
#include <ff/map.hpp>
using namespace ff;
using namespace cv;
const int MAPWORKERS=2;
/* my task */
struct Task {
    Task(uchar *src, uchar *dst, long rows, long cols, const std::string
         &name):
        src(src),dst(dst),rows(rows), cols(cols), name(name) {};
    uchar       *src, *dst;
    long rows, cols;
    const std::string  name;
};
/* ----- utility function ------- */
template<typename T>
T *Mat2uchar(cv::Mat &in) {
    T *out = new T[in.rows * in.cols];
    for (int i = 0; i < in.rows; ++i)
        for (int j = 0; j < in.cols; ++j)
            out[i * (in.cols) + j] = in.at<T>(i, j);
```

[1]Project web site: http://www.opencv.org

```
    return out;
}
char* getOption(char **begin, char **end, const std::string &option) {
    char **itr = std::find(begin, end, option);
    if (itr != end && ++itr != end) return *itr;
    return nullptr;
}
#define XY2I(Y,X,COLS) (((Y) * (COLS)) + (X))
/* ------------------------------------ */
// returns the gradient in the x direction
static inline long xGradient(uchar * image, long cols, long x, long y) {
    return image[XY2I(y-1, x-1, cols)] +
        2*image[XY2I(y, x-1, cols)] +
        image[XY2I(y+1, x-1, cols)] -
        image[XY2I(y-1, x+1, cols)] -
        2*image[XY2I(y, x+1, cols)] -
        image[XY2I(y+1, x+1, cols)];
}
// returns the gradient in the y direction
static inline long yGradient(uchar * image, long cols, long x, long y) {
    return image[XY2I(y-1, x-1, cols)] +
        2*image[XY2I(y-1, x, cols)] +
        image[XY2I(y-1, x+1, cols)] -
        image[XY2I(y+1, x-1, cols)] -
        2*image[XY2I(y+1, x, cols)] -
        image[XY2I(y+1, x+1, cols)];
}
// 1st stage
struct Read: ff_node {
  Read(char **images, const long num_images):
     images((const char**)images),num_images(num_images) {}

  void *svc(void *) {
    for(long i=0; i<num_images; ++i) {
      const std::string &filepath(images[i]);
      std::string filename;

      // get only the filename
      int n=filepath.find_last_of("/");
      if (n>0) filename = filepath.substr(n+1);
      else     filename = filepath;

      Mat *src = new Mat;
      *src = imread(filepath, CV_LOAD_IMAGE_GRAYSCALE);
      if ( !src->data ) {
        error("reading image file %s, going on....\n",
        filepath.c_str());
        delete src;
        continue;
      }
      uchar * dst = new uchar[src->rows * src->cols];
      for(long y = 0; y < src->rows; y++)
      for(long x = 0; x < src->cols; x++) {
        dst[y * src->cols + x] = 0;
      }
      Task *t=new Task(Mat2uchar<uchar>(*src),dst,src->rows,src->cols,
          filename);
      ff_send_out(t); // sends the task t to the next stage
    }
    return EOS; // computation completed
  }
  const char **images;
  const long num_images;
};
// 2nd stage
struct SobelStage: ff_Map<> {
  // sets the maximum n. of worker for the Map
  // spinWait is set to true
  // the scheduler is disabled by default
  int nw;
  SobelStage(int mapworkers)
    :ff_Map<>(mapworkers,true) {nw=mapworkers;}

  void *svc(void *t) {
    Task *task = reinterpret_cast<Task*>(t);
    uchar * src = task->src, * dst = task->dst;
    long cols =  task->cols;
    ff_Map<>::parallel_for(1,task->rows-1,[src,cols,&dst](const long y)
        {
      for(long x = 1; x < cols - 1; x++){
        const long gx = xGradient(src, cols, x, y);
        const long gy = yGradient(src, cols, x, y);
        // approximation of sqrt(gx*gx+gy*gy)
        long sum = abs(gx) + abs(gy);
        if (sum > 255) sum = 255;
        else if (sum < 0) sum = 0;
        dst[y*cols+x] = sum;
      }
    });
    const std::string &outfile = "./out/" + task->name;
    imwrite(outfile, cv::Mat(task->rows, task->cols, CV_8U, dst, cv::Mat
        ::AUTO_STEP));
    delete task->src; delete [] task->dst; delete task;
    return GO_ON;
  };
};

int main(int argc, char *argv[]) {
  if (argc < 2) {
    std::cerr << "\nuse: " << argv[0]
    << "  [-m <Wrks=2>] <image-file> [image-file]\n";
    std::cerr << "    Wrks2 is the n. of map's workers\n\n";
    return -1;
  }
  int start = 1;
  int Wrks  = MAPWORKERS;
  char *m = getOption(argv, argv+argc, "-m");
  if (m) { Wrks  = atoi(m); start+=2; argc-=2; }
  long num_images = argc-1;
  assert(num_images >= 1);

  Read        reader(&argv[start], num_images);
  SobelStage sobel(Wrks);
  ff_pipe<Task> pipe(&reader, &sobel);
  if (pipe.run_and_wait_end()<0) {
    error("running pipeline\n");
    return -1;
  }
  return 0;
}
```

Chapter 5

Data-flow parallelism

The *data-flow* programming model is a general approach to parallelization based upon data dependencies among a program's operations. Its theoretical and methodological value is quite fundamental in parallel processing.

The computations is expressed by the data-flow graph, i.e. a DAG whose nodes are instructions and arcs are pure data dependencies.

If instead of simple instructions, portions of code (sets of instructions or functions) are used as graph's nodes, then it is called the macro data-flow model (MDF). The resulting MDF program is therefore represented as a graph whose nodes are computational kernels and arcs *read-after-write* dependencies. It is worth noting that, the data-flow programming model is able to work both on stream of values and on a single value. For this reason, it is considered somehow a primitive model of computation.

5.1 The *ff_mdf* data-flow skeleton

The *mdf* skeleton in FastFlow calles `ff_mdf` implements the *macro data-flow* parallel pattern. The run-time of the FastFlow mdf pattern is responsible for scheduling *fireable* instructions (i.e. those with all input data dependencies ready) and managing data dependencies.

Currently, the internal implementation of the FastFlow mdf pattern is a 2-stage pipeline, the first stage is responsible for dynamically generating the graph instructions by executing the users' code, and the second stage (a task-farm with feedback channel) is responsible for managing task dependencies and task scheduling. In the next version of FastFlow, other different implementations using less resources will be provided to the users.

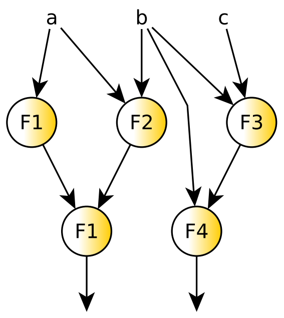


Figure 5.1: Data-flow dependency graph

_ff_mdf_ run-time components:

At the general level, it is possible to identify the following main parts of the FastFlow mdf run-time:

- Dynamic construction of the task graph (DAG): in complex program the task graph could be very large; its generation time can affect significantly the computation time, and the memory required to store the entire graph may be extremely large. To overcome these issues, a solution is to generate the graph during the computation of tasks (and possibly overlap task generation and task computation), such that only a window of the graph is maintained in memory.
- Handling inter-task dependencies: Handling task dependencies, i.e. update dependencies after the completion of previously scheduled tasks and determining ready (fireable) tasks to be executed.
- Scheduling of fireable tasks: a task having all input dependencies ready may be selected by the interpreter for execution. This selection needs to be performed in a smart way in order to optimise cache locality, particular important on shared-cache multi-core, and at the same time trying to maintain a good load balancing among worker threads.

Creating graph tasks

In order to show how to use the *mdf* pattern, let's take a look at the following simple case. Suppose you have the following sequential code:

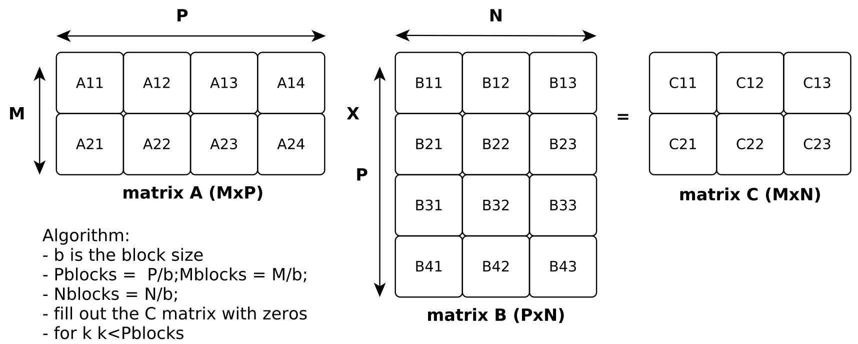
```
const size_t SIZE = 1<<20;
long *A,*B,*C,*D;
allocate(A,B,C,D, size);
sum2(A,B,SIZE);   // executes A = A + B;
sum2(C,B,SIZE);   // executes C = C + B;
sum2(D,B,SIZE);   // executes D = D + B;
sum3(D,A,C,SIZE);// executes D = A + C + D;
```

here we can identify 4 macro instructions that can be executed concurrently with the FastFlow mdf pattern by specifying, for each instruction, which are the input or output parameters of the function that has to be executed (in this case `sum2` and `sum3`). For example, the first instruction in the code above has two input parameters (A and B) and one output parameter (C) that have to be passed to the function `sum2`. A parameter is provided to the run-time as `param_info` type, which has two fields, one is the pointer of the input or output of the "real" parameter and the second is the "*direction*" which can be `INPUT`,`OUTPUT` or `VALUE` (this last one is used mainly for those parameters that can be evaluated as a single value, do not constitute a real dependency). The function and the parameters are passed to the run-time using the `AddTask` method of the `mdf` class. The resulting code is the following one:

```cpp
/* ************************** */
/* ******* hello_mdf.cpp ***** */

/*                                   --------
 *                                  /        \
 *                                  |         \
 * A = A + B;     // sum2     A     B     C   |     D
 * C = C + B;     // sum2     |     |     |   |     |
 * D = D + B;     // sum2      \   / \   /     \   /
 * D = A + C + D; // sum3        +     +         +
 *                               A     C         D
 *                               |     |         |
 *                                \     \       /
 *                                 ----- + -----
 *                                       D
 */
#include <ff/mdf.hpp>
const size_t SIZE = 1<<20;
long *A,*B,*C,*D;

void taskGen() {
  long *A = P->A,*B = P->B;
  long *C = P->C,*D = P->D;
  auto mdf = P->mdf;
  std::vector<param_info> Param;
  // A = A + B;
  {
    const param_info _1={(uintptr_t)A,INPUT};  // 1st param
    const param_info _2={(uintptr_t)B,INPUT};  // 2nd param
    const param_info _3={(uintptr_t)A,OUTPUT}; // 3rd param
    // pack the parameters in one single vector
    Param.push_back(_1); Param.push_back(_2);
    Param.push_back(_3);
    mdf->AddTask(Param, sum2, A,B,SIZE); // create on task
  }
  // C = C + B;
  {
    Param.clear();
    const param_info _1={(uintptr_t)C,INPUT};
    const param_info _2={(uintptr_t)B,INPUT};
    const param_info _3={(uintptr_t)C,OUTPUT};
    Param.push_back(_1); Param.push_back(_2); Param.push_back(_3);
    mdf->AddTask(Param, sum2, C,B,SIZE);
  }
  // D = D + B;
  {
    Param.clear();
    const param_info _1={(uintptr_t)D,INPUT};
    const param_info _2={(uintptr_t)B,INPUT};
    const param_info _3={(uintptr_t)D,OUTPUT};
    Param.push_back(_1); Param.push_back(_2); Param.push_back(_3);
    mdf->AddTask(Param, sum2, D,B,SIZE);
  }
  // D = A + C + D;
  {
    Param.clear();
    const param_info _1={(uintptr_t)A,INPUT};
    const param_info _2={(uintptr_t)C,INPUT};
    const param_info _3={(uintptr_t)D,INPUT};
```



```
Algorithm:
- b is the block size
- Pblocks =  P/b;Mblocks = M/b;
- Nblocks = N/b;
- fill out the C matrix with zeros
- for k k<Pblocks
    for j j<Mblocks
      for i i<Nblocks
        Cij = Cij + Aij * Bkj
```

Figure 5.2: Block-based matmul algorithm

```
    const param_info _4={(uintptr_t)D,OUTPUT};
    Param.push_back(_1); Param.push_back(_2);
    Param.push_back(_3); Param.push_back(_4);
    mdf->AddTask(Param, sum3, D,A,C,SIZE);
  }
}
int main() {
  allocate(A,B,C,D, SIZE);
  // creates the mdf object passing the task generetor function
  ff_mdf dag(taskGen);

  if (dag.run_and_wait_end() <0) error("running mdf");
  return 0;
}
```

5.2 Examples

In this section we propose two numerical application: the block-based matrix multiplication and the block-based Cholesky factorisation.

5.2.1 Block-based matrix multiplication

As a more complex usage example of the FastFlow mdf pattern we consider here the block-based matrix multiplication algorithm. The algorithm is sketched in Fig. 5.2.

This version computes in parallel the C_{ij} blocks doing p-passes. A more cache-friendly implementation can be implemented.

```
/* ******************************** */
/* ******* blk_matmul_mdf.cpp ***** */

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <sys/time.h>

#include <ff/mdf.hpp>
#include <ff/parallel_for.hpp>
using namespace ff;

const double THRESHOLD = 0.001;
bool PFSPINWAIT=false;       // enable/disable spinWait
int  PFWORKERS=1;            // parallel_for parallelism degree
int  PFGRAIN   =0;           // default static scheduling of iterations

void random_init (long M, long N, long P, double *A, double *B);
long parse_arg(const char *string);
void printarray(const double *A, long m, long n, long N);
// executes the standard ijk algorithm on a single block
// using the ParallelFor pattern. This is the macro instruction.
void blockMM(long b, const double *A, const long AN,
             const double *B, const long BN,
             double *C, const long CN) {

    ParallelFor pf(PFWORKERS,PFSPINWAIT);
    pf.disableScheduler(true); // removes the scheduler thread
    pf.parallel_for(0,b,1,PFGRAIN,[A,B,AN,BN,CN,b,&C](const long i) {
            for (long j = 0; j < b; j++) {
                for (long k = 0; k < b; k++)
                    C[i*CN+j] += A[i*AN+k]*B[k*BN+j];
            }
        });
}
// put the block pointed by C to zero
void zerosBlock(long b, double *C, const long CN) {
    for (long i = 0; i < b; i++)
        for (long j = 0; j < b; j++)
            C[i*CN+j] = 0.0;
}
// taskGen's parameters data type
template<typename T>
struct Parameters {
    Parameters(const double *A, const double *B, double *C):A(A),B(B),C(
        C) {}
    const double *A,*B;
    double       *C;
    long    AN,BN,CN;     // stride
    long    b, m, n, p;   // size
    T* mdf;
};

// Block-based algorithm:
void taskGen(Parameters<ff_mdf > *const P){
    const double *A = P->A, *B = P->B;
    double       *C = P->C;
    long m=P->m,n=P->n,p=P->p,b=P->b;
    long AN=P->AN,BN=P->BN,CN=P->CN;
  long mblocks = m/b, nblocks = n/b, pblocks = p/b;
    auto mdf = P->mdf;

    // set C blocks to zero
    for(long i=0; i<mblocks; ++i) {
        for(long j=0; j<nblocks; ++j) {
            double *Cij       = &C[b*(i*CN+j)];
            std::vector<param_info> Param;
            const param_info _1={(uintptr_t)Cij, INPUT};
            const param_info _2={(uintptr_t)Cij, OUTPUT};
            Param.push_back(_1); Param.push_back(_2);
            // create a task
            mdf->AddTask(Param, zerosBlock, b, Cij, CN);
        }
    }
    for(long k=0; k<pblocks; ++k) {
        for(long i=0; i<mblocks; ++i) {
            for(long j=0; j<nblocks; ++j) {
                const double *Aik = &A[b*(i*AN + k)];
                const double *Bkj = &B[b*(k*BN + j)];
                double *Cij       = &C[b*(i*CN+j)];
                std::vector<param_info> Param;
                const param_info _1={(uintptr_t)Aik, INPUT};
                const param_info _2={(uintptr_t)Bkj, INPUT};
                const param_info _3={(uintptr_t)Cij, OUTPUT};
                Param.push_back(_1); Param.push_back(_2); Param.push_back(_3);
                // create a task
                mdf->AddTask(Param, blockMM, b, Aik, AN, Bkj, BN, Cij, CN);
            }
        }
    }
}

int main(int argc, char* argv[]) {
    if (argc < 5) {
        printf("\n\tuse: %s <M> <N> <P> <blocksize>''
        printf'' [nworkers] [pfworkers:pfgrain]\n", argv[0]);
        printf("\t <-> required argument, [-] optional argument\n");
        printf("\t A is M by P\n");
        printf("\t B is P by N\n");
        printf("\t nworkers is the n. of workers of the mdf pattern\n");
        printf("\t pfworkers is the  n. of workers of the ParallelFor pattern\n");
        printf("\t pfgrain is the ParallelFor grain size\n");
        printf("\t NOTE: the blocksize must evenly divide M, N and P.\n");
        return -1;
    }
    int nw = -1;
    long M = parse_arg(argv[1]);
    long N = parse_arg(argv[2]);
    long P = parse_arg(argv[3]);
    long b = parse_arg(argv[4]);
    if (argc >= 6) nw = atoi(argv[5]);
    if (argc >= 7) {
        std::string pfarg(argv[6]);
        int n = pfarg.find_first_of(":");
        if (n>0) {
            PFWORKERS = atoi(pfarg.substr(0,n).c_str());
            PFGRAIN   = atoi(pfarg.substr(n+1).c_str());
        } else PFWORKERS = atoi(argv[6]);
    }
    const double *A = (double*)malloc(M*P*sizeof(double));
    const double *B = (double*)malloc(P*N*sizeof(double));
    assert(A); assert(B);

    random_init(M, N, P, const_cast<double*>(A), const_cast<double*>(B));
    double *C       = (double*)malloc(M*N*sizeof(double));

    Parameters<ff_mdf > Param(A,B,C);
    ff_mdf mdf(taskGen, &Param,32,(nw<=0?ff_realNumCores():nw));
    Param.m=M,Param.n=N,Param.p=P,Param.b=b;
    Param.AN=P,Param.BN=N,Param.CN=N;
    Param.mdf=&mdf;
    printf("Executing %-40s", "Block-based MM");
    ffTime(START_TIME);
    mdf.run_and_wait_end();
    ffTime(STOP_TIME);
    printf("  Done in %11.6f secs.\n", (ffTime(GET_TIME)/1000.0));

    free((void*)A); free((void*)B); free(C);
    return 0;
}
```

5.2.2 Block-based Cholesky factorisation

To be added

Bibliography

[1] Marco Aldinucci, Andrea Bracciali, Pietro Liò, Anil Sorathiya, and Massimo Torquati. StochKit-FF: Efficient systems biology on multicore architectures. In M. R. Guarracino, F. Vivien, J. L. Träff, M. Cannataro, M. Danelutto, A. Hast, F. Perla, A. Knüpfer, B. Di Martino, and M. Alexander, editors, *Euro-Par 2010 Workshops, Proc. of the 1st Workshop on High Performance Bioinformatics and Biomedicine (HiBB)*, volume 6586 of *LNCS*, pages 167–175, Ischia, Italy, aug 2011. Springer.

[2] Marco Aldinucci, Marco Danelutto, Lorenzo Anardu, Massimo Torquati, and Peter Kilpatrick. Parallel patterns + macro data flow for multi-core programming. In *Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing*, pages 27–36, Garching, Germany, feb 2012. IEEE.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating sequential programs using FastFlow and self-offloading. Technical Report TR-10-03, Università di Pisa, Dipartimento di Informatica, Italy, feb 2010.

[4] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with fastflow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, August 2011. Springer.

[5] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, March 2014.

[6] Marco Aldinucci, Marco Danelutto, Massimiliano Meneghin, Peter Kilpatrick, and Massimo Torquati. Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans Peters, and Thierry Priol, editors, *Parallel Computing: From Multicores and*

GPU's to Petascale (Proc. of PARCO 2009, Lyon, France), volume 19 of *Advances in Parallel Computing*, pages 273–280, Lyon, France, 2010. IOS press.

[7] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with fastflow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, feb 2010. IEEE.

[8] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. Decision tree building on multi-core using fastflow. *Concurrency and Computation: Practice and Experience*, 26(3):800–820, March 2014.

[9] Marco Aldinucci and Massimo Torquati. *FastFlow website*, 2009. `http://mc-fastflow.sourceforge.net/`.

[10] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, sep 2009.

[11] Marco Aldinucci, Massimo Torquati, Concetto Spampinato, Maurizio Drocco, Claudia Misale, Cristina Calcagno, and Mario Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, June 2013.

[12] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[13] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[14] Marco Danelutto. *Distributed Systems: Paradigms and Models*. SPM cource note, Computer Science Department University of Pisa, 2013.

[15] Marco Danelutto, Luca Deri, Daniele De Sensi, and Massimo Torquati. Deep packet inspection on commodity hardware using fastflow. In *Proc. of PARCO 2013 Conference, Munich, Germany*, Munich, Germany, 2013.

[16] Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" fastflow. In *Formal Methods for Components and Objects: 7th Intl. Symposium, FMCO 2008, Sophia-Antipolis, France, October 20 - 24, 2008, Revised Lectures*, volume 8606 of *LNCS*, pages 28–74. Springer, 2014.

[17] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw., Pract. Exper.*, 40(12):1135–1160, 2010.

[18] Claudia Misale, Giulio Ferrero, Massimo Torquati, and Marco Aldinucci. Sequence alignment tools: one parallel pattern to rule them all? *BioMed Research International*, 2014.