APPENDIX A

FASTFLOW

A.1 DESIGN PRINCIPLES

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

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

The whole programming framework has been incrementally developed according to a layered design on top of Pthread/C++ standard programming framework and targets shared memory multicore architectures (see Fig. A.1).

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

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

Eventually, the third layer, the **Streaming Networks Patterns** layer, provides common stream parallel patterns. The primitive patterns include pipeline and farms. Simple

\(^{1}\)see also the FastFlow home page at http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about
specialization of these patterns may be used to implement more complex patterns, such as
divide and conquer, map and reduce patterns.

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

- defining sequential concurrent activities, by subclassing a proper FastFlow class, the
  ff_node class, and
- building complex stream parallel patterns by hierarchically composing sequential con-
current activities, pipeline patterns, farm patterns and their “specialized” versions
implementing more complex parallel patterns.

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

The pipeline pattern may be used to implement sequences of streaming networks $S_1 \rightarrow
\ldots \rightarrow S_k$ with $S_k$ receiving input from $S_{k-1}$ and delivering outputs to $S_{k+1}$. $S_i$
may be either a sequential activity or another parallel pattern. $S_1$ must be a stream generator
activity and $S_k$ a stream consuming one.

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

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

![Layered FastFlow design](image)

Figure A.1 Layered FastFlow design
In addition, a feedback channel may be added to a farm, moving output results back from
the collector (or from the collection of workers in case no collector is specified) back to the
emitter input channel. The feedback channel may only be added to the farm/pipe at the root of the skeleton tree.

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

- divide and conquer, using a farm with feedback loop and proper stream items tagging
  (input tasks, subtask results, results)

- MISD (multiple instruction single data, that is something computing $f_1(x_i), \ldots, f_k(x_i)$
  out of each $x_i$ appearing onto the input stream) pattern, using a farm with an emitter
  implementing a broadcast scheduling policy

- map, using an emitter partitioning an input collection and scheduling one partition
  per worker, and a collector gathering sub-partitions results from the workers and
  delivering a collection made out of all these results to the output stream.

It is worth pointing out that while using plain pipeline and farms (with or without emitters
and collectors) actually can be classified as “using skeletons” in a traditional skeleton
based programming framework, the usage of specialized versions of the farm streaming net-
work can be more easily classified as “using skeleton templates”, as the base features of the
FastFlow framework are used to build new patterns, not provided as primitive skeletons.

Concerning the usage of FastFlow to support parallel application development on shared
memory multicores, the framework provides two abstractions of structured parallel compu-
tation:

- a “skeleton program abstraction” which is used to implement applications completely
  modelled according to the algorithmic skeleton concepts. When using this abstraction,
  the programmer write a parallel application by providing the business logic code,
  wrapped into proper `ff_node` subclasses, 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” which is used to parallelize (and therefore accelerate) only
  some parts of an existing application. In this case, the programmer provides a skeleton
  (composition) which is run on the “spare” cores of the architecture and implements a
  parallel version of the business logic to be accelerated, that is the computing a given
  $f(x)$. The skeleton (composition) will have its own input and output channels. When
  an $f(x)$ has actually to be computed within the application, rather than writing proper
  code to call to the sequential $f$ code, the programmer may insert code asynchronously
  “offloading” $x$ to the accelerator skeleton. Later on, when the result of $f(x)$ is to be
  used, some code “reading” accelerator result may be used to retrieve the accelerator
  computed values.

This second abstraction fully implements the “minimal disruption” principle stated by Cole
in his skeleton manifesto [37], 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. A.7).

2 Although this may change in future FastFlow releases, this is the current situation as of FastFlow version 1.1
A.2 INSTALLATION

Before entering the details of how FastFlow may be used to implement efficient stream parallel (and not only) programs on shared memory multicore architectures, let’s have a look at how FastFlow may be installed.

The installation process is trivial, actually:

1. first, you have to download the source code from SourceForge (http://sourceforge.net/projects/mc-fastflow/)
2. then you have to extract the files using a tar xzvf fastflow-XX.tgz command, and
3. eventually, you should use the top level directory resulting from the tar xzvf command as the argument of the -I flag of g++.

As an example, the currently available version (1.1) is hosted in a fastflow-1.1.0.tar.gz file. If you download it and extract files to your home directory, you should compile FastFlow code using the flags g++ -I $HOME/fastflow-1.1.0 -lpthread in addition to any other flags needed to compile your specific code. Sample makefiles are provided both within the fastflow-1.1.0/tests and the fastflow-1.1.0/examples directories in the source distribution.

A.3 TUTORIAL

As all programming frameworks tutorials, we start with a Hello world code. In order to implement our hello world program, we use the following code:

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

using namespace ff;

class Stage1: public ff_node {
public:

    void * svc(void * task) {
        std::cout << "Hello world" << std::endl;
        return NULL;
    }
};

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

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

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

    return 0;
}
```

3We only detail instructions needed to install FastFlow on Linux/Unix/BSD machines here. A Windows port of FastFlow exist, that requires slightly different steps for the installation.
Line 2 includes all what's needed to compile a FastFlow program just using a pipeline pattern and line 4 instruct compiler to resolve names looking (also) at ff namespace. Lines 6 to 13 host the application business logic code, wrapped into a class sub classing ff_node. The void * svc(void *) method wraps the body of the concurrent activity resulting from the wrapping. It is called every time the concurrent activity is given a new input stream data item. The input stream data item pointer is passed through the input void * parameter. The result of the single invocation of the concurrent activity body is passed back to the FastFlow runtime returning the void * result. In case a NULL is returned, the concurrent activity actually terminates itself. The application main only hosts code needed to setup the FastFlow streaming network and to start the skeleton (composition) computation: lines 17 and 18 declare a pipeline pattern (line 17) and insert a single stage (line 18) in the pipeline. Line 20 starts the computation of the skeleton program and awaits for skeleton computation termination. In case of errors the run_and_wait_end() call will return a negative number (according to the Unix/Linux syscall conventions).

When the program is started, the FastFlow RTS accomplishes to start the pipeline. In turn the first stage is started. As the first stage svc returns a NULL, the stage is terminated immediately after by the FastFlow RTS.

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

```
ffsrc $ g++ -lpthread -I /home/marcod/Documents/Research/CodeProgramming/fastflow -1.1.0 helloworldSimple.cpp -o hello
ffsrc $ ./hello
Hello world
ffsrc $
```

There is nothing parallel here, however. The single pipeline stage is run just once and there is nothing else, from the programmer viewpoint, running in parallel. The graph of concurrent activities in this case is the following, trivial one:

![Stage1](image)

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

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

using namespace ff;

class Stage1 : public ff_node {
public:

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

If we compile and run the program, however, we get a kind of unexpected result:
First of all, the program keeps running printing an "Hello world" every second. We in fact terminate the execution through a CONTROL-C. Second, the initial sequence of strings is a little bit strange.

The "infinite run" is related to way FastFlow implements concurrent activities. Each ff_node is run as many times as the number of the input data items appearing onto the output stream, unless the svc method returns a NULL. Therefore, if the method returns either a task (pointer) to be delivered onto the concurrent activity output stream, or the GO_ON mark (no data output to the output stream but continue execution), it is re-executed as soon as there is some input available. The first stage, which has no associated input stream, is re-executed up to the moment it terminates the svc with a NULL. In order to have the program terminating, we therefore may use the following code for Stage1:

```c
class Stage1 : public ff_node {
public:

    Stage1() { first = (1==1); }

    void * svc(void * task) {
        if (first) {
            std::cout << "Hello " << std::endl;
            char * p = (char *) calloc(sizeof(char),10);
            strcpy(p,"World");
            sleep(1);
            first = 0;
            return ((void *)p);
        } else {
            return NULL;
        }
    }

private:
    int first;
};
```

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

```
Hello World
Hello World
Hello World
Hello World
```

and depending on the actual number of cores of your machine and on the kind of scheduler used in the operating system, the sequence may vary a little bit.
that is the program terminates after a single run of the two stages. Now the question is: why the second stage terminated, although the \texttt{svc} method return value states that more work is to be done? The answer is in the stream semantics implemented by \texttt{FastFlow}. \texttt{FastFlow} streaming networks automatically manage end-of-streams. That is, as soon as an \texttt{ff\_node} returns a NULL–implicitly declaring he wants to terminate its output stream, the information is propagated to the node consuming the output stream. This nodes will therefore also terminate execution–without actually executing its \texttt{svc} method–and the end of stream will be propagated onto its output stream, if any. Therefore Stage\texttt{2} terminates after the termination of Stage\texttt{1}.

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

### A.3.1 Generating a stream

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

We already discussed the role of the return value of the \texttt{svc} method. Therefore a first version of this program may be implemented using as the \texttt{Stage1} class the following code:

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

using namespace ff;

class Stage1 : public ff\_node {
public:
  Stage1(int n) {
    streamlen = n;
    current = 0;
  }

  void * svc(void * task) {
    if (current < streamlen) {
      current++;
      std::cout << "Hello number " << current << " " << std::endl;
      int * p = (int *) calloc(sizeof(int),1);
    }
```
```cpp
private:
  int streamlen, current;
};

class Stage2: public ff_node {
public:
  void * svc(void * task) {
    int * i = (int *) task;
    std::cout << "World " " " " << std::endl;
    free(task);
    return GO_ON;
  }
};

int main(int argc, char * argv[]) {
  ff_pipeline pipe;
  pipe.add_stage(new Stage1(atoi(argv[1])));
  pipe.add_stage(new Stage2());
  if (pipe.run_and_wait_end()<0) {
    error("running pipeline\n");
    return -1;
  }
  return 0;
}
```

The output we get is the following one:

```
ffsrc$ g++ -lpthread -I
/home/marcod/Documents/Research/CodeProgramming/fastflow -1.1.0
  helloStream.cpp -o helloStream
ffsrc$ ./helloStream
Hello number 1
Hello number 2World - 1--
Hello number World -32 -
World -3- Hello number
4
Hello number 5World - 4--
World -5--
ffsrc$
```

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

In this case, we could have written the Stage as follows:

```java
class Stage1: public ff_node {
  public:
    Stage1(int n) {
      streamlen = n;
      current = 0;
    }

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

  private:
    int streamlen, current;
};
```

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

### A.4 MORE ON FF_NODE

The ff_node class actually defines three distinct virtual methods:

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

The first one is the one defining the 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 subclasses to implement initialization code and finalization code, respectively. Actually, the svc method must be overwritten as it is defined as a pure virtual method.

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

We can implement the Eratostheness sieve with the following FastFlow program.

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

using namespace ff;

class Sieve : public ff_node {

  public:

  Sieve() { filter = 0; }

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

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

private:
  int filter;
};

class Generate : public ff_node {

  public:

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

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

  void * svc(void * tt) {
    unsigned int * t = (unsigned int *) tt;
```
if (task < streamlen) {
    int * xi = (int *) calloc(1, sizeof(int));
    *xi = task++;
    return xi;
} else {
    return NULL;
}
}
private:
    int streamlen;
    int task;
};
class Printer: public ff_node {
    int svc_init() {
        std::cout << "Printer started " << std::endl;
        first = 0;
    }
    void * svc(void **t) {
        int * xi = (int *) t;
        if (first == 0) {
            first = *xi;
        }
        return GO_ON;
    }
    void svc_end() {
        std::cout << "Sieve terminating, prime numbers found up to " << first
            << std::endl;
    }
private:
    int first;
};
int main(int argc, char * argv[]) {
    if (argc!=3) {
        std::cerr << "use: " << argv[0] << " nstages streamlen\n";
        return -1;
    }
    ff_pipeline pipe;
    int nstages = atoi(argv[1]);
    pipe.add_stage(new Generate(atoi(argv[2])));
    for(int j=0; j<nstages; j++)
        pipe.add_stage(new Sieve());
    pipe.add_stage(new Printer());
    ffTime(START_TIME);
    if (pipe.run_and_wait_end() < 0) {
        error("running pipeline\n");
        return -1;
    }
    ffTime(STOP_TIME);
    std::cerr << "DONE, pipe time= " << pipe.ffTime() << " (ms)\n";
MANAGING ACCESS TO SHARED OBJECTS

Shared objects may be accessed within FastFlow programs using the classical pthread concurrency control mechanisms. The FastFlow program is actually a multithreaded code using the pthread library, in fact.
We demonstrate how access to shared objects may be ensured within a FastFlow program forcing mutual exclusion in the access to the std::cout file descriptor. This will be used to have much nicer strings output on the screen when running the Sieve program illustrated in the previous section.

In order to guarantee mutual exclusion on the shared std::cout descriptor we use a pthread_mutex_lock. The lock is declared and properly initialized as a static, global variable in the program (see code below, line 7). Then each one of the writes to the std::cout descriptor in the concurrent activities relative to the different stages of the pipeline are protected through a pthread_mutex_lock / pthread_mutex_unlock “brackets” (see line 29–31 in the code below, as an example).

```cpp
#include <iostream>
#include <ff/pipeline.hpp>
#include <pthread.h>

using namespace ff;

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

class Sieve: public ff_node {
public:

Sieve() { filter = 0; }

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

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

private:
    int filter;
};

class Generate: public ff_node {
public:

Generate(int n) {
    streamlen = n;
    task = 2;
    pthread_mutex_lock(&lock);
    std::cout << "Generate object created" << std::endl;
```
```cpp
int svc_init() {
    pthread_mutex_lock(&lock);
    std::cout << "Sieve started. Generating a stream of " << streamlen
              << " elements, starting with " << task << std::endl;
    pthread_mutex_unlock(&lock);
    return 0;
}

void * svc(void *tt) {
    unsigned int *t = (unsigned int*)tt;
    if (task < streamlen) {
        int *xi = (int*)calloc(1, sizeof(int));
        *xi = task++;
        return xi;
    } else {
        return NULL;
    }
}

private:
    int streamlen;
    int task;
};

class Printer: public ff_node {
    int svc_init() {
        pthread_mutex_lock(&lock);
        std::cout << "Printer started " << std::endl;
        pthread_mutex_unlock(&lock);
        first = 0;
    }

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

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

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

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

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

  std::cerr << "DONE, pipe time= " << pipe.ffTime() << " (ms)\n";
  std::cerr << "DONE, total time= " << ffTime(GET_TIME) << " (ms)\n";
  pipe.ffStats(std::cerr);
  return 0;
}
```

When running the program, we get a slightly different output than the one we obtained when the usage of `std::cout` was not properly regulated:

```
ffsrc/sievelock.cpp

ffsrc$ ./a.out 7 30
Sieve started. Generating a stream of 30 elements, starting with 2
  Prime(2)
  Prime(5)
  Prime(13)
  Prime(11)
  Prime(7)
Sieve terminating, prime numbers found up to 19
  Prime(3)
  Prime(17)
DONE, pipe time= 58.439 (ms)
DONE, total time= 64.473 (ms)
FastFlow trace not enabled
ffsrc$
```

The strings are printed in clearly separated lines, although some apparently unordered string sequence appears, which is due to the FastFlow scheduling of the concurrent activities and to the way locks are implemented and managed in the pthread library.

It is worth pointing out that

- FastFlow ensures correct access sequences to the shared object used to implement the streaming networks (the graph of concurrent activities), such as the SPSC queues used to implement the streams, as an example.
• FastFlow stream semantics guarantee correct sequencing of activation of the concurrent activities modelled through *ff_node* objects and connected through streams. The stream implementation actually ensures pure data flow semantics.

• any access to any user defined shared data structure must be protected with either the primitive mechanisms provided by FastFlow (see Sec. A.5) or the primitives provided within the pthread library.

### A.6 MORE SKELETONS: THE FastFlow FARM

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

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

```cpp
#include <ff/farm.hpp>
using namespace ff;

int main(int argc, char * argv[]) {
    ...
    ff_farm<> myFarm;
    std::vector<ff_node *> w;
    for (int i=0; i<nworkers;++i)
        w.push_back(new Worker);
    myFarm.add_workers(w);
    ...
```

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

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

• we can use the farm defined with a code similar to the one described above as the second stage of a pipeline whose first stage generates the input stream according to one of the techniques discussed in Sec. A.3.1. This means we will use the farm writing a code such as:

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

• or we can provide an emitter and a collector to the farm, specialized in such a way they can be used to produce the input stream and consume the output stream of the farm, respectively, while inheriting the default scheduling and gathering policies.
The former case is simple. We only have to understand why adding the farm to the pipeline as a pipeline stage works. This will discussed in detail in Sec. A.8. The latter case is simple as well, but we discuss it through some more code.

### A.6.1 Farm with emitter and collector

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

```cpp
class Emitter : public ff_node {
public:
    Emitter(int n) {
        streamlen = n;
        task = 0;
    }

    void * svc(void *) {
        sleep(1);
        task++;
        int * t = new int(task);
        if (task<streamlen)
            return t;
        else
            return NULL;
    }

private:
    int streamlen;
    int task;
};
```

In this case, the node svc actually does not take into account any input stream item (the input parameter name is omitted on line 5). Rather, each time the node is activated, it returns a task to be computed using the internal ntasks value. The task is directed to the “next” worker by the FastFlow farm run time support.

Concerning the collector, we can also use a ff_node: in case the results need further processing, they can be directed to the next node in the streaming network using the mechanisms detailed in Sec. A.3.1. Otherwise, they can be processed within the svc method of the ff_node subclass.

As an example, a collector just printing the tasks/results he gets from the workers may be programmed as follows:

```cpp
class Collector : public ff_node {
public:
    void * svc(void * task) {
        int * t = (int *)task;
        std::cout << "Collector got " << *t << std::endl;
        return GO_ON;
    }
};
```
With these classes defined and assuming to have a worker defined by the class:

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

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

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

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

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

The concurrent activity graph in this case is the following one:
When run with the first argument specifying the number of workers to be used and the second one specifying the length of the input stream generated in the collector node, we get the expected output:

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

A.6.2 Farm with no collector

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

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

Writes to the global result array need not to be synchronized as each worker writes different positions in the array (the `TASK` tags are unique).

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

static int *results;

typedef struct task_t {
  int i;
  int t;
};
```
} TASK;

using namespace ff;

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

class Emitter: public ff_node {
public:
    Emitter(int n) {
        streamlen = n;
        task = 0;
    }
    void * svc(void *) {
        task++;
        TASK * t = (TASK *) malloc(1, sizeof(TASK));
        t->i = task;
        t->t = task*task;
        if (task<streamlen)
            return t;
        else
            return NULL;
    }
private:
    int streamlen;
    int task;
};

int main(int argc, char * argv[]) {
    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);
    results = (int *) malloc(streamlen, sizeof(int));
    ff_farm <> farm;
    Emitter E(streamlen);
    farm.add_emitter(&E);
    std::vector<ff_node *> w;
    for (int i=0;i<nworkers;++i)
        w.push_back(new Worker);
    farm.add_workers(w);
    std::cout << "Before starting computation" << std::endl;
    for (int i=0; i<streamlen; i++)
        std::cout << i << " : " << results[i] << std::endl;
    if (farm.run_and_wait_end()<0) {
        error("running farm\n");
    }
The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

The Worker code at lines 14–21 defines an `svc` method that returns a `GO_ON`. Therefore no results are directed to the collector (non-existing, see lines 55-74: they define the farm but they do not contain any `add_collector` in the program `main`). Rather, the results computed by the worker code at line 18 are directly stored in the global array. In this case the concurrent activity graph is the following:

![Concurrent Activity Graph](image-url)

The main program prints the results vector before calling the `FastFlow start_and_wait_end()` and after the call, and you can easily verify the results are actually computed and stored in the correct place in the vector:

```
ffsrc/farmNoC.cpp

```
A.6.3 Specializing the scheduling strategy in a farm

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

The steps to performed in this case are exemplified with the following, relevant portions of code.

First, we subclass the `ff_loadmanager` and provide our own `setworker()` method:

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

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

    void set_victim(int v) { victim=v; }

private:
    int victim;
};
```

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

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

Eventually, we create an emitter that within its `svc` method invokes the `set_victim` method right before outputting a task towards the worker string, either with a `ff_send_out(task)` or with a `return(task)`. The emitter is declared as:

```cpp
class myEmitter: public ff_node {
    myEmitter(ff_loadbalancer * ldb) {
        lb = ldb;
    }

    ... 

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

    ... 

private:
```
and inserted in the farm with the code

```cpp
myEmitter emitter = myFarm.loadbalance();
myFarm.add_emitter(emitter);
```

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

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

A.6.3.1 Broadcasting a task to all workers  FastFlow supports the possibility to direct a
task to all the workers in a farm. It is particularly useful if we want to process the task
by workers implementing different functions. The broadcasting is achieved through the
declaration of a specialized load balancer, in a way very similar to what we illustrated in
Sec. A.6.3.

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

```cpp
#include <iostream>
#include <ff/farm.hpp>
#include <ff/node.hpp>
#include <math.h>

using namespace std;
using namespace ff;

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

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

  void *svc(void *in) {
    int *i = ((int *) in);
    int ii = *i;
    *i += 1;
    cout << "WorkerPLus got " << ii << " and computed " << *i << endl;
    return in;
  }
};

class WorkerMinus: public ff_node {
  int svc_init() {
    cout << "Worker initialized" << endl;
    return 0;
  }
};
```
```cpp
void * svc(void * in) {
    int *i = ((int *) in);
    int ii = *i;
    *i--;
    cout << "WorkerMinus got " << ii << " and computed " << *i << endl;
    return in;
}

class my_loadbalancer: public ff_loadbalancer {
public:
    // this is necessary because ff_loadbalancer has non default parameters....
    my_loadbalancer(int max_num_workers): ff_loadbalancer(max_num_workers) {}
    void broadcast(void * task) {
        ff_loadbalancer::broadcast_task(task);
    }
};

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

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

#define NW 2

int main(int argc, char * argv[]) {
    ffTime(START_TIME);
    cout << "init " << argc << endl;
    int nw = (argc==1 ? NW : atoi(argv[1]));
    cout << "using " << nw << " workers " << endl;
    // init input (fake)
```
for(int i=0; i<MAX; i++) {
    x[i] = (i*10);
}

for(int i=0; i<MAX; i++) {
    cout << "Setting up farm" << endl;
    // create the farm object
    ff_farm<my_loadbalancer> farm(true, nw);
    // create and add emitter object to the farm
    Emitter E(farm.getlb());
    farm.add_emitter(&E);
    cout << "emitter ok" << endl;

    std::vector<ff_node*> w; // prepare workers
    w.push_back(new WorkerPlus);
    w.push_back(new WorkerMinus);
    farm.add_workers(w); // add them to the farm
    cout << "workers ok" << endl;

    Collector C(1);
    farm.add_collector(&C);
    cout << "collector ok" << endl;

    farm.run_then_freeze(); // run farm asynchronously
    cout << "Sending tasks ..." << endl;
    int tasks[MAX];
    for(int i=0; i<MAX; i++) {
        tasks[i]=i;
        farm.offload((void*) &tasks[i]);
    }
    farm.offload((void*) FF_EOS);
    cout << "Waiting termination" << endl;
    farm.wait();
    cout << "Farm terminated after computing for " << farm.ffTime() << endl;

    ffTime(STOP_TIME);
    cout << "Spent overall " << ffTime(GET_TIME) << endl;
}

ffsrc/ff_misd.cpp

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

Lines 103 and 104 are used to add two different workers to the farm.

The rest of the program is standard, but for the fact the resulting farm is used as an accelerator (lines 112–123, see Sec. A.7).

A.6.3.2 Using autoscheduling FastFlow provides suitable tools to implement farms with “auto scheduling”, that is farms where the workers “ask” for something to be computed rather than accepting tasks sent by the emitter (explicit or implicit) according to some
scheduling policy. This scheduling behaviour may be simply implemented by using the `ff_farm` method `set_scheduling_ondemand()`, as follows:

```c
ff_farm myFarm(...);
myFarm.set_scheduling_ondemand();
... 
farm.add_emitter(...);
```

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

A.7 FastFlow AS A SOFTWARE ACCELERATOR

Up to know we just showed how to use FastFlow to write a “complete skeleton application”, that is an application whose complete flow of control is defined through skeletons. In this case the `main` of the C/C++ program written by the user is basically providing the structure of the parallel application by defining a proper FastFlow skeleton nesting and the commands to start the computation of the skeleton program and to wait its termination. All the business logic of the application is embedded in the skeleton parameters.

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

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

Using FastFlow accelerator mode is not too much different from using FastFlow to write an application only using skeletons (see Fig. A.2). In particular, the following steps must be followed:

- A skeleton program has to be written, using the FastFlow skeletons (or their customized versions), computing the tasks that will be given to the accelerator. The skeleton program used to program the accelerator is supposed to have an input stream, used to offload the tasks to the accelerator.

- Then, the skeleton program must be run using a particular method, different from the `run_and_wait_end` we have already seen, that is a `run_then_freeze()` method. This method will start the accelerator skeleton program, consuming the input stream items to produce either output stream items or to consolidate (partial) results in
memory. When we want to stop the accelerator, we will deliver an end-of-stream
mark to the input stream.

- Eventually, we must wait for the computation of the accelerator to be terminated.

A simple program using FastFlow accelerator mode is shown below:

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

using namespace ff;

int *x;
int nworkers = 0;

class Worker: public ff_node {
public:
    Worker(int i) {
        my_id = i;
    }

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

private:
    int my_id;
};

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

    nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);
```
We use a farm accelerator. The accelerator is declared at line 43. The “true” parameter is the one telling FastFlow this has to be used as an accelerator. Workers are added at lines 45–48. Each worker is given its id as a constructor parameter. This is the same as the code in plain FastFlow applications. Line 50 starts the skeleton code in accelerator mode. Lines 55 to 58 offload tasks to be computed to the accelerator. These lines could be part of any larger C++ program, indeed. The idea is that whenever we have a task ready to be submitted to the accelerator, we simply “offload” it to the accelerator. When we have no more tasks to offload, we send an end-of-stream (line 59) and eventually we wait for the completion of the computation of tasks in the accelerator (line 60).

This kind of interaction with an accelerator not having an output stream is intended to model those computations than consolidate results directly in memory. In fact, the Worker code actually writes results into specific position of the vector x. Each worker writes the task it receives in the i-th position of the vector, being i the index of the worker in the farm worker string. As each worker writes a distinct position in the vector, no specific synchronization is needed to access vector positions. Eventually the last task received by worker i will be stored at position i in the vector.

We can also assume that results are waited from the accelerator through its output stream. In this case, we first have to write the skeleton code of the accelerator in such a way an output stream is supported. In the new version the accelerator sample program below, we add a collector to the accelerator farm (line 45). The collector is defined as just collecting results from workers and delivering the results to the output stream (lines 18–24). Once the tasks have been offloaded to the accelerator, rather waiting for accelerator completion, we can ask computed results as delivered to the accelerator output stream through the bool load_result(void **) method (see lines 59–61).
```cpp
#include <vector>
#include <iostream>
#include <ff/farm.hpp>
#include <time.h>

using namespace ff;

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

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

int main(int argc, char * argv[]) {
    if (argc < 3) {
        std::cerr << "use: 
        " << argv[0]
        " nworkers streamlen\n"
        return -1;
    }
    int nworkers = atoi(argv[1]);
    int streamlen = atoi(argv[2]);
    ff_farm<> accelerator(true);
    std::vector<ff_node *> w;
    for (int i = 0; i < nworkers; ++i)
        w.push_back(new Worker());
    accelerator.add_workers(w);
    accelerator.add_collector(new Collector());
    if (accelerator.run_then_freeze() < 0) {
        error("running farm\n");
        return -1;
    }
    for (int i = 0; i <= streamlen; i++) {
        int * task = new int(i);
        accelerator.offload(task);
    }
    accelerator.offload((void *) FF_EOS);
    void * result;
    while (accelerator.load_result(&result)) {
```

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 `bool load_results_nb(void **)`. In this case, if no result is available at the moment, the method returns a “false” value, and you should retry later on to see whether a result may be retrieved.

### A.8 SKELETON NESTING

In **FastFlow** skeletons may be arbitrarily nested. As the current version only supports farm and pipeline skeletons, this means that:

- farms may be used as pipeline stages, and
- pipelines may be used as farm workers.

There are no limitations to nesting, but the following one:

- skeletons using the `wrap_around` facility (see also Sec. A.9) cannot be used as parameters of other skeletons.

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

```cpp
ff::farm <> myFarm;

std::vector<ff::node *> w;
for(int i=0; i<NW; i++)
    ff::pipeline * p = new ff::pipeline;
    p->add_stage(new S1());
    p->add_stage(new S2());
    w.push_back(p);
myFarm.addWorkers(w);
```

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

```cpp
ff::pipeline * p = new ff::pipeline;
ff::farm <> f = new ff::farm;
...;
f.addWorkers(w);
...;
p->add_stage(new SeqWorkerA());
p->add_stage(f);
p->add_stage(new SeqWorkerB());
```
The concurrent activity graph in this case will be the following one:

![Diagram of concurrent activity graph]

while in the former case it will be such as

![Diagram of concurrent activity graph]

### A.9 FEEDBACK CHANNELS

In some cases, it will be useful to have the possibility to route back some results to the streaming network input stream. As an example, this allows to implement divide and conquer using farms. Task injected in the farm are split by the workers and the resulting splitted tasks are routed back to the input stream for further processing. Tasks that can be computed using the base case code, are computed instead and their results are used for the conquer phase, usually performed in memory.

All what’s needed to implement the feedback channel is to invoke the `wrap_around` method on the interested skeleton. In case our applications uses a farm pattern as the outermost skeleton, we may therefore add the method call after instantiating the farm object:

```java
f_farm <> myFarm;
...
myFarm.add_emitter(&e);
myFarm.add_collector(&c);
myFarm.add_workers(w);
myFarm.wrap_around();
...
```

and this will lead to the concurrent activity graph
The same if parallelism is expressed by using a pipeline as the outermost skeleton:

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

leading to the concurrent activity graph:

As of FastFlow 1.1, the only possibility to use the feedback channel provided by the `wrap_around` method is relative to the outermost skeleton, that is the one with no input stream. This because at the moment FastFlow does not support merging of input streams. In future versions this constrain will be possibly eliminated.

A.10 INTRODUCING NEW SKELETONS

Current version of FastFlow (1.1) only supports stream parallel pipeline and farm skeletons. However, the skeletons themselves may be used/customized to serve as “implementation templates”\(^6\) for different kinds of skeletons. The FastFlow distribution already includes sample applications where the farm with feedback is used to support divide\&conquer applications. Here we want to discuss how a data parallel map skeleton may be used in FastFlow, exploiting the programmability of farm skeleton emitter and collector.

\(^6\) according to the terminology used in the algorithmic skeleton community
A.10.1 Implementing a Map skeleton with a Farm “template”

In a pure map pattern all the items in a collection are processed by means of a function \( f \). If the collection was

\[
x = (x_1, \ldots, x_m)
\]

then the computation

\[
map f x
\]

will produce as a result

\[
(f(x_1), \ldots, f(x_m))
\]

In more elaborated map skeletons, the user is allowed to define a set of (possibly overlapping) partitions of the input collection, a function to be applied on each one of the partitions, and a strategy to rebuild–from the partial results computed on the partitions–the result of the map.

As an example, a matrix multiplication may be programmed as a map such that:

- the input matrixes A and B are considered as collections of rows and columns, respectively
- a set of items \( \langle A_{i,*}, B_{*,j} \rangle \)–the \( i-th \) row of \( A \) and the \( j-th \) column of \( B \)–are used to build the set of partitions
- an inner product is computed on each \( \langle A_{i,*}, B_{*,j} \rangle \): this is \( c_{i,j} \) actually
- the \( C \) matrix \((C = A \times B)\) is computed out of the different \( c_{i,j} \).

If we adopt this second, more general approach, a map may be build implementing a set of concurrent activities such as:

```
// a task requires to compute the matrix multiply C = A x B
// we assume square matrixes, for the sake of simplicity
typedef struct {
    int n;
    float **a;
    float **b;
```
we define the emitter to be used in the farm as follows:

```c
void *svc(void *t) {
    TASK *task = (TASK *) t; // tasks come in already allocated
    for (int i=0; i<task->n; i++) {
        for (int j=0; j<task->n; j++) {
            // SUBTASKe are allocated in the splitter and destroyed in the worker
            SUBTASK *st = (SUBTASK *) calloc(1, sizeof(SUBTASK));
            st->i = i;
            st->j = j;
            st->t = task;
            ff_send_out((void *)st);
        }
    }
    return GO_ON;
}
```

Basically, the first time the emitter is called, we generate all the tasks relative to the different \((A_{i,*}, B_{*j})\). These tasks are directed to the workers, that will compute the different \(c_{i,j}\) and direct the PART_RESULT to the collector. The worker `ff_node` will therefore be programmed as:

```c
class Worker: public ff_node {
    public:

    void *svc(void *task) {
        SUBTASK *t = (SUBTASK *) task;
        float *x = new float(0.0);
        for (int k=0; k<t->n; k++) {
            *x = *x + (t->a[t->i][k] * (t->b)[k][t->j]);
        }
        // prepare the partial result to be delivered
        PART_RESULT *pr = (PART_RESULT *) calloc(1, sizeof(PART_RESULT));
        pr->i = t->i;
    }
}
```
The collector will be defined in such a way the different partial results computed by the workers are eventually consolidated in memory. Therefore each $c_{i,j}$ received is stored at the correct entry of the $C$ matrix. The pointer of the result matrix is in fact a field in the TASK data structure and $c_{i,j}, i$ and $j$ are fields of the PART_RESULT data structure. The code for the collector is therefore:

```cpp
class Compose: public ff_node {
public:
    Compose() {
        n = 0;
        for (int i = 0; i < MAXDIFF; i++)
            tags[i] = 0;
    }

    void * svc(void * t) {
        PART_RESULT * r = (PART_RESULT *) t;
        TASK * tt = r->t;
        // consolidate result in memory
        ((r->t)->c)[r->i][r->j] = r->x;
        tags[((r->t)->tag)%MAXDIFF]++;
        if (tags[((r->t)->tag)%MAXDIFF] == ((r->t)->n) * ((r->t)->n)) {
            tags[((r->t)->tag)%MAXDIFF] = 0;
            free(t);
            return tt;
        } else {
            free(t);
            return GO_ON;
        }
    }
private:
    int n;
    int tags[MAXDIFF];
};
```

The tags here are used to deliver a result on the farm output stream (i.e. the output stream of the collector) when exactly $n^2$ results relative to the same input task have been received by the collector. A MAXDIFF value is used assuming that no more than MAXDIFF different matrix multiplication tasks may be circulating at the same time in the farm, due to variable time spent in the computation of the single $c_{i,j}$.

With these classes, our map may be programmed as follows:

```cpp
ff_farm <> farm(true);
farm.add_emitter(new Split()); // add the splitter emitter
farm.add_collector(new Compose()); // add the composer collector
std::vector<ff_node *> w; // add the convenient # of workers
```
It is worth pointing out that:

- the kind of knowledge required to write the Split and Compose nodes to the application programmer is very application specific and not too much related to the implementation of the map

- this implementation of the map transforms a data parallel pattern into a stream parallel one. Some overhead is paid to move the data parallel sub-tasks along the streams used to implement the farm. This overhead may be not completely negligible

- a much coarser grain implementation could have been designed assuming that the Split node outputs tasks representing the computation of a whole $C_i$, row and modifying accordingly the Worker.

- usually, the implementation of a map data parallel pattern generates as many subtasks as the amount of available workers. In our implementation, we could have left to the Split node this task, using the FastFlow primitive mechanisms to retrieve the number of workers actually allocated to the farm$^7$ and modifying accordingly both the Worker and the Compose code.

Also, the proposed implementation for the map may be easily encapsulated in a proper `ff_map` class:

```cpp
class ff_map {
    public:

    // map constructor
    // takes as parameters: the splitter, the string of workers and the result rebuilder
    ff_map(ff_node * splt, std::vector<ff_node *> wrks, ff_node * cmps) {
        exec.add_emitter(splt); // add the splitter emitter
        exec.add_collector(cmps); // add the composer collector
        exec.add_workers(wrks); // add workers
    }

    operator ff_node*() {
        return (ff_node*)&exec;
    }

    private:
    ff_farm <> exec;
};
```

With this definition, the user could have defined the map (and added the map stage to a pipeline) using the following code:

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

$^7$this is the `getnworkers` method of the farm loadbalancer.
A.11 PERFORMANCE

Up to now we only discussed how to use FastFlow to build parallel programs, either applications completely coded with skeletons, or FastFlow software accelerators. We want to shortly discuss here the typical performances improvements got through FastFlow.

In skeleton application or in software accelerator, using a FastFlow farm would in general lead to a performance increase proportional to the number of workers used (that is to the parallelism degree of the farm). This unless:

- we introduce serial code fragments—in this case the speedup will be limited according to the Amdahl law—or
- we use more workers than the available tasks
- or eventually the time spent to deliver a task to be computed to the worker and retrieving a result from the worker are higher than the computation time of the task.

This means that if the time spent to compute $m$ tasks serially is $T_{seq}$, we can expect the time spent computing the same $m$ tasks with an $nw$ worker farm will be more or less $T_{seq} \times \frac{nw}{m}$.

It is worth pointing out here that the latency relative to the computation of the single task does not decrease w.r.t. the sequential case.

In case a $k$ stage FastFlow pipeline is used to implement a parallel computation, we may expect the overall service time of the pipeline is $T_S = \max\{T_{S_1}, \ldots, T_{S_k}\}$. As a consequence, the time spent computing $m$ tasks is approximately $m \times T_S$ and the relative speedup may be quantified as $m \times \frac{\sum_{i=1}^{k} T_{S_i}}{\max\{T_{S_1}, \ldots, T_{S_k}\}}$. In case of balanced stages, that is pipeline stages all taking the same time to compute a task, this speedup may be approximated as $k$, being $\sum_{i=1}^{k} T_{S_i} = k \times T_{S_1}$ and $\max\{T_{S_1}, \ldots, T_{S_k}\} = T_{S_1}$.

A.12 RUN TIME ROUTINES

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

- **virtual int get_my_id()** returns a virtual id of the node where the concurrent activity (its svc method) is being computed
- **virtual bool ff_send_out(void * task,**
  unsigned int retry=((unsigned int)-1),
  unsigned int ticks=(TICKS2WAIT))** delivers an item onto the output stream, possibly retrying upon failure a given number of times, after waiting a given number of clock ticks.
- **double ffTime()** returns the time spent in the computation of a farm or of pipeline. This is method of both classes pipeline and farm.