Evolution of FastFlow: C++11 extensions, distributed systems

Massimo Torquati

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

PARAPHRASE Summer School 2014
Dublin, Ireland, June 9th – 12th
Outline

• FastFlow C++11 extensions
  • ff_pipe vs ff_pipeline
  • New constructors for the ff_farm
  • ParallelFor and ParallelForReduce pattern

• Targeting distributed systems in FastFlow
  • Concepts
  • Communication patterns
  • Simple examples
  • Marshalling/unmarshalling
FastFlow basic concepts

• To define an FastFlow (FF) node, it is mandatory to subclass from the `ff_node` class defining the `svc` method:

```c
struct myClass: ff_node {
    void *svc(void *intask) {
        // code to be executed in this node here
        return outtask
    }
};
```

• That means that it is in general required to move existing sequential code, typically with some minor changes adjustments
  • This may be considered costly or not practical but has some benefits
FastFlow basic concepts

- Defining each single node separately allows:
  - To have pipeline with potentially different input and output types
  - To implement MISD farms (each worker executing a different function)
  - To have pipeline and farm with multi-input and multi-output nodes to create more complex network graphs
“functional-like” pipeline and farm in FastFlow

- C++11 lambdas allow to create functions on the fly reducing code moving

  Lambda function example:

  ```cpp
  auto F = [x,&y] (const long z) → long { return x+(y++)+z; }
  ```

- We defined a new `ff_pipe` class (the old one is `ff_pipeline`) that accepts lambdas (and also `ff_node(s)`) as pipeline stages
  - In this case the input and output types of each node must be the same (i.e. the `ff_pipe` template parameter)

- Furthermore, we added more constructors to the `ff_farm<>` class that simplify the instantiation of simple farms using lambdas
C++11 interface for the \texttt{ff\_pipe} and \texttt{ff\_farm}

- 3-stage pipeline
  \[
  f, g, h: (\text{myTask}^*, \text{ff\_node}^*\text{const}) \rightarrow \text{myTask}^* \\
  \text{ff\_pipe<myTask> pipe1}(f, g, h); \quad // h( g( f( x) ) ) \\
  \text{pipe1.run\_and\_wait\_end}();
  \]

- pipeline of \texttt{ff\_pipe(s)}:
  \[
  m, n, o: (\text{myTask}^*, \text{ff\_node}^*\text{const}) \rightarrow \text{myTask}^* \\
  \text{ff\_pipe<myTask> pipe2}(m, \&\text{pipe1}, n, o); \quad //o(n(h(g(f(m(x))))))) \\
  \text{pipe2.run\_and\_wait\_end}();
  \]
Farms: simplified syntax

- **ff_farm**
  
g: (myTask*,ff_node*const) → myTask*  // function or lambda
ff_farm<> farm1(g,3); // farm with default Emitter and Collector
farm1.run_and_wait_end();

- pipeline with farm stages :
  
f,g,h: (myTask*,ff_node*const) → myTask*
ff_pipe<myTask> pipe1(f, &farm1, h);
pipe1.run_and_wait_end();
A simple example

Sequential code

int main(int argc, char *argv[]) {
  ..... 
  File_t* file = readFile(Dir);
  while(file) {
    do_work(file);
    file = readFile(Dir);
    writeFile(file,Dir); 
  }
  ..... 
}

FF parallel code

int main(int argc, char *argv[]) {
  auto Read = [&](File_t*,ff_node*const)-> File_t* { return readFile(DIR); } 
  auto Work = [&](File_t*file,ff_node*const)-> File_t* { do_work(file); return file; }
  auto Write = [&](File_t*file,ff_node*const)-> File_t* {
    writeFile(file,Dir); return (File_t*)GO_ON; }

  ff_pipe<File_t> pipe(Read, new ff_farm<>(Work,3), Write);
  pipe.run_and_wait_end();
}

Paraphrase
const int K = 10; int k=0;
auto lambda=[K,&k]() -> void*{ if (k++ == K) return NULL; return new int(k); }
auto F=[](ff_task* t,ff_node*const node) -> ff_task*{ do_work(t); return t; }

struct Emitter:public ff_node {
  std::function<void*()> F;
  Emitter(std::function<void*()> F):F(F) {}  
  void *svc(void*) { return F(); }  
};

ff_farm<> farm2(F,3);
farm2.remove_collector();
farm2.add_emitter(new Emitter(lambda));
farm2.set_scheduling_ondemand();
farm2.run_and_wait_end();
ParallelFor

- High-level map pattern implemented with a task-farm-with-feedback skeleton

- Mimic the `#pragma-based parallel for` of OpenMP
- ParallelFor class defined in `<ff/parallel_for.hpp>`
- Several different options available:
  - static scheduling vs. dynamic scheduling
  - dynamic scheduling with or without a scheduler thread
  - spin barrier vs blocking barrier at the end of loop iterations
ParallelFor: how it works

Sequential loop:
long start = 0, stop = 100, step = 2;
for(long i=start; i<stop; i+=step)
  f(i);

FastFlow PARFOR transformation:
long start=0, stop=100, step=2;
long chunk=8, nworkers=3;
parallel_for(start, stop, step, chunk,[&](const long i) {
  f(i);
}, nworkers);

Initial tasks table owned by forSched:

<table>
<thead>
<tr>
<th>wid</th>
<th>#tasks</th>
<th>min-max</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>0-32</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>32-64</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>64-100</td>
</tr>
</tbody>
</table>

Dynamic scheduling of iteration chunks:

- **W₁**: 0-8, 8-16, 16-24, 24-32, 88-96
- **W₂**: 32-40, 40-48, 48-56, 56-64, 96-100
- **W₃**: 64-72, 78-80, 80-88

end of loop computation

farm-with-feedback skeleton
ParallelFor: basic interface

ParallelFor pf(maxnumthreads=-1, spinWait=false, skipwarmup=false);

- pf.parallel_for(start, stop, body);  // the simplest case
  body : [capture-list] (const long idx) { body-using-idx; }

pf.parallel_for(start, stop, step, grain, body, numthreads);
  body : [capture-list] (const long idx) { body-using-idx; }

pf.parallel_for_thid((start, stop, step, grain, body, numthreads);
  body : [capture-list] (const long idx, const int thid) { body-using-idx; }

pf.parallel_for_idx(start, stop, step, grain, body, numthreads);
  body : [capture-list] (const long start, const long stop, const int thid)  
  { for(idx=start;idx<stop;idx+=step) { body-using-idx;}}
ParallelFor: a simple example

Sequential loop with independent iterations

```c
for(long i=0;i<N;i+=2)  A[i]=f(i);
```

FastFlow ParallelFor parallelization:

```c
ParallelFor pf;  // maxhreads equal to the available num. cores
pf.parallel_for(0,N,2,[&](const long i){f(i);});
pf.parallel_for(0,N,2,[&](const long i){f(i);},16);
pf.parallel_for(0,N,2,8,[&](const long l){f(l);},16);
pf.parallel_for_static(0,N,2,8,[&](const long l){f(l);});
```

- Static scheduling using all worker threads
- Static scheduling using 16 worker threads
- Dynamic scheduling with grain 8
- Static scheduling with grain 8
• What does it mean default static scheduling?
  It means that ($\text{#iteration/numthreads}$) iterations are assigned to each thread.
  **Example**: 10 iterations, 3 threads: 
  
<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-3</td>
</tr>
<tr>
<td>1</td>
<td>4-6</td>
</tr>
<tr>
<td>2</td>
<td>7-9</td>
</tr>
</tbody>
</table>

• What does it mean grain $k$ in the dynamic scheduling?
  It means that no more than $k$ iterations at a time are assigned to one worker thread using an auto-scheduling policy.
  **Example**: 10 iterations, 3 threads, grain 2
  
<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2-3 8-9</td>
</tr>
<tr>
<td>1</td>
<td>4-5 6-7</td>
</tr>
<tr>
<td>2</td>
<td>0-1</td>
</tr>
</tbody>
</table>

• What does it mean grain $k$ in the static scheduling?
  It means that the iteration space is divided in chunks of no more than $k$ iterations and then they are assigned to the threads in a round-robin fashion.
  **Example**: 10 iterations, 3 threads, grain 2
  
<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-1 6-7</td>
</tr>
<tr>
<td>1</td>
<td>2-3 8-9</td>
</tr>
<tr>
<td>2</td>
<td>4-5</td>
</tr>
</tbody>
</table>
ParallelForReduce

- ParallelForReduce is a combination of a ParallelFor (map pattern) and a reduction operation (reduce pattern).
  - Implemented with a task-farm-with-feedback skeleton
- Mimic the OpenMP parallel for with reduction operation
- ParallelForReduce class defined in `<ff/parallel_for.hpp>`
- A ParallelForReduce can be used as a ParallelFor
- Several different options available:
  - multi reductions (more than 1 single variable)
  - static scheduling vs. dynamic scheduling selection
  - dynamic scheduling with or without the scheduler thread
  - spin barrier vs blocking barrier at the end of loop iterations
Sequential piece of code:

```c
for(long k=0;k<10;++k) {  // 10 iterations
    double s=0.0;
    for(long i=0;i<N;++i) s+=A[i]*A[i];  // map+reduction
    s=sqrt(s);
    for(long i=0;i<N;++i) A[i] -= s;  // map
}
```

Parallel code using ParForReduce

```c
ParallelForReduce<double> pfr(16, true); // maxnumthreads 16, spinwait enabled
Fsum= [](double &v, const double e) { v+=e;}
for(long k=0;k<10;++k) {  // 10 iterations
    double s=0.0;
    pfr.parallel_reduce(s,0.0,0,N, [&](const long i, long &s) { s+=A[i]*A[i]; }, Fsum);
    s=sqrt(s);
    pfr.parallel_for(0,N, [&](const long i) {A[i] -= s});
}
```
OpenMP vs FastFlow syntax

Comparing syntax: Dot-Product using static scheduling:

**OpenMP (compiler based):**

```c
sum = 0.0;
#pragma omp parallel for schedule(static) reduction(+:sum)
for(long i=0;i<N;++i) sum += A[i]*B[i];
```

**FastFlow (library):**

```c
sum = 0.0;
ParallelForReduce<double> pfr;
pfr.parallel_reduce_static(sum,0.0,0,N,[A,B](const long i, double &sum) {
    sum += A[i]*B[i];
},[](double &v, const double elem) { v+=elem});
```
2nd part

Targeting distributed memory platforms in FastFlow
To scale to hundreds/thousands of cores we need more than a single multi-core workstation.

The data-flow model of FastFlow can be easily extended to work in a cluster of workstations.

We introduced the ff_dnode, which subclass the ff_node adding an “external” communication channel.
ff_node and ff_dnode

- The external channel may be specialised in: input, output or both input and output.
- The basic idea is that only the edge nodes of the FastFlow network are ff_dnode(s).
ff_dnode class sketch

• Same interface of the ff_node class
• It encapsulates the external channel whose type is passed to the ff_node as template parameter
• The init method creates and initializes the communication end-point
Communication patterns

- Currently the following patterns are available:
  - **Unicast**: one-to-one unidirectional channel
  - **Broadcast**: one-to-many, i.e. the same task is sent to all connected peers (receivers)
  - **Scatter**: one-to-many, i.e. the task is split in partitions each one sent to a different connected peer (receiver)
  - **On-Demand**: one-to-many, i.e. the task is sent to one of the connected peers using an auto-scheduling policy
  - **Gather-All**: many-to-one, i.e. receives a partition of the task from all connected peers, and eventually the message is recomposed
  - **Collect-from-Any**: many-to-one, i.e. receives one task from one of the connected peers (senders)
A “complex” example
Communication patterns interface

- **init** and **close**
- **get** and **put**
  (multipart messages – sender side)
- **done** (multipart messages – receiver side)

```cpp
// Communication pattern interface
template <typename Impl>
class commPattern {
    protected:
        Impl impl;
    public:
        typedef typename Impl::descriptor descriptor;
        typedef typename Impl::tosend_t tosend_t;
        typedef typename Impl::torecv_t torecv_t;
        typedef typename Impl::TransportImpl TransportImpl;
        // Creates a an empty communication pattern.
        commPattern():impl() {}
        // D is an implementation-based descriptor for the pattern and
        // contains all the low-level implementation details.
        commPattern(descriptor* D):impl(D) {} 
        // sets the descriptor
        inline void setDescriptor(descriptor* D) { impl.setDescriptor(D); }
        // returns the descriptor
        inline descriptor* getDescriptor() { return impl.getDescriptor(); }
        // initializes the communication pattern
        inline bool init(const std::string& address, const int nodeId=-1) { 
            impl.init(address, nodeId); }
        // the message being sent is just a part of the entire message.
        inline bool putmore(const tosend_t& msg) { return impl.putmore(msg); }
        // sends one message to target node
        inline bool put(const tosend_t& msg) { return impl.put(msg); }
        inline bool put(const tosend_t& msg, const int toNode) { 
            return impl.put(msg, toNode); }
        // gets the message header
        inline bool gethdr(torecv_t& msg, int& peer) { return impl.gethdr(msg, peer); }
        // receives one message part
        inline bool get(torecv_t& msg) { return impl.get(msg); }
        // all message has been received
        inline void done() { impl.done(); } 
        // closes pattern
        inline bool close() { return impl.close(); }
};
```
Communication transport interface

```cpp
// communication transport interface
template <typename Impl>
class commTransport {
protected:
    Impl impl;

public:
    typedef typename Impl::endpoint_t endpoint_t;
    typedef typename Impl::msg_t msg_t;

    // creates a transport end-point with a given id
    commTransport(const int procId): impl(procId) {}

    // initializes the transport
    int initTransport() { return impl.initTransport(); }

    // closes the transport
    int closeTransport() { return impl.closeTransport(); }

    // creates and returns a new communication end-point P=SENDER|RECEIVER
    endpoint_t* newEndPoint(const bool P) { return impl.newEndPoint(P); }

    // destroys an end-point
    int deleteEndPoint(endpoint_t* ep) { return impl.deleteEndPoint(ep); }

    // returns the communication transport id
    int getProcId() const { return impl.getProcId(); }
};
```
Communication patterns implementation

• TCP/IP networks
  • using the ZeroMQ library
  • ZeroMQ is an open-source communication library
    • www.zeromq.org
• Infiniband network
  • ZeroMQ does not provide any support for Infiniband so we decided to implement our own library for Infiniband
  • Library based on Linux verbs
• Future direction
  • Communication patterns implementation using MPI
typedef zmqTransportMsg_t msg_t;

zmqTransport transport(SENDER); // creates the network using 0mq as transport
if (transport.initTransport()<0) abort(); // init transport

UNICAST Unicast(new UNICAST_DESC(name,&transport, SENDER));
if (!Unicast.init(address)) abort();

msg_t msg;
for(int i=0;i<100;++i) {
    msg.init(new long(i),sizeof(long));
    Unicast.put(msg);
}
Unicast.close();
delete Unicast.getDescriptor();
transport.closeTransport();

// receiver code
msg_t msg;
for(int i=0;i<100;++i) {
    Unicast.get(msg);
    long data=*static_cast<long*>(msg.getData());
}


How communications work in FastFlow

- The user does not have to use explicitly put/get/putmore/ etc....
- Sending and receiving data is in charge of the FF run-time
- The user has to:
  - define and connect the application parts running on different hosts
    - typically using an SPMD model
  - prepare the data that has to be sent by the run-time
  - provide the run-time with the message buffer in which the data has to be received
Simple examples

hostA:> test11_pipe A 1 hostB:port
hostB:> test11_pipe A 0 hostB:port

hostA:> test11_torus A B 1 hostB:port hostA:port
hostB:> test11_torus A B 0 hostB:port
How to define a dnode

```cpp
// just an edge-node of my application
class myNode: ff_dnode<zmqBcast> {
public:
    myNode(const std::string& name,       // unique identifier for the channel
            const std::string& address,   // peer address [host:port]
            const int npeers,            // n. of peers
            zmqBcast::TransportImp* const t); // transport object

    int svc_init() {
        // I am the producer, i.e. the one that sends data out.
        // The consumer has to call init with the last param set to false.
        return ff_dnode<zmqBcast>::init(name, address, npeers, t, SENDER);
    }

    void* svc(void* task) {
        // As soon as a task is returned or the ff_send_out method is called,
        // the message is sent out in broadcast (because of zmqBcast pattern)
        // to all connected peers.
    }

    void svc_end() { .... }
...};
```
ffrun: an ssh-based launcher

- A launcher is needed to simplify the deployment of complex application
- ffrun is an ssh-based launcher developed for FastFlow at UNITO
  - It is based on a JSON file containing the description of the parts of the application to launch
  - It uses ssh to execute remote commands
    - The executable files have to be transferred before launching on remote machines (or you need a distributed FS)
- ffrun is still under development
Marshalling/Unmarshalling

• Suppose we need to send 2 or more objects in one single message
• If the 2 objects are contiguous in memory, then no problem because it can be seen as a single message
• If the 2 objects are non contiguous in memory, then we have to allocate a buffer and copy both (or in the best case only one) of the two to create a single contiguous memory object.
  • The copies may be quite costly in term of performance
One solution to avoid copying is to use multi-part messages via `readv/writev` system calls.

```c
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

struct iovec {
    void  *iov_base; // starting address
    size_t  iov_len;   // number of bytes to transfer
};

std::vector<iovec> V;
iovec v1={ptr1,size1}, v2={ptr2,size2};
V.push_back(v1); V.push_back(v2);
writev(fd,V.data(), 2);
```
The ff_dnode class provides 3 methods that can be overloaded to manage marshalling/unmarshalling:

- **2 prepare** methods (1 to be called by the sender and 1 to be called by the receiver)
- **1 unmarshall** method used only by the receiver

**sender-side:** the prepare method is called by the run-time before sending data into the channel

**receiver-side:** the prepare method is called by the run-time to get the message buffers where the input data will be stored; the unmarshall method is called before passing the received data to the svc() method, thus overall adjustments on the entire input data may be applied
Marshalling/Unmarshall in FastFlow

Object definition:

```c
struct mystring_t {
    long     length;
    char  *str;
}; mystring_t* ptr;
```

Memory layout:

```
ptr
12
Hello world!
str
```

```c
// Called by the PRODUCER before sending data
// Used to prepare (non contiguous) output messages
virtual void prepare(svector<iovec>& v, void* ptr, const int sender=-1) {
    mystring_t* p = static_cast<mysting_t*>(ptr);
    struct iovec iov={ptr,sizeof(mystring_t)};
    v.push_back(iov);
    iov.iov_base = p->str;
    iov.iov_len   = p->length+1;
    v.push_back(iov);
}
```

```c
// Called by the CONSUMER before receiving data
// Gives a pool of messages on which input data can be received
virtual void prepare(svector/msg_t*>* v, size_t len, const int sender=-1) {
    svector<msg_t>* v2 = new svector<msg_t*>(len);
    assert(v2);
    for(size_t i=0;i<len;++i) {
        msg_t * m = new msg_t;
        assert(m);
        v2->push_back(m);
    }
    v = v2;
```

```c
// Called by the CONSUMER before calling the svc() method
virtual void unmarshalling(svector<msg_t*>* const v[],const int vlen, void *task) {
    assert(vlen==1 && v[0]->size()==2);
    mystring_t* p = static_cast<mysting_t*>(v[0]->operator[](0)->getData());
    p->str = static_cast<char*>(v[0]->operator[](1)->getData());
    assert(strlen(p->str)== p->length());
    task=p;
}
THANK YOU!

http://www.paraphrase-ict.eu

@paraphrase_fp7