# Targeting distributed systems in FastFlow

**Authors of the work:**

**Marco Aldinucci**

Computer Science Dept. - University of Turin - Italy

**Sonia Campa, Marco Danelutto and Massimo Torquati**

Computer Science Dept. - University of Pisa - Italy

**Peter Kilpatrick**

Queen's University Belfast - UK

**Speaker:**

**Massimo Torquati** e-mail: torquati@di.unipi.it

# Talk outline

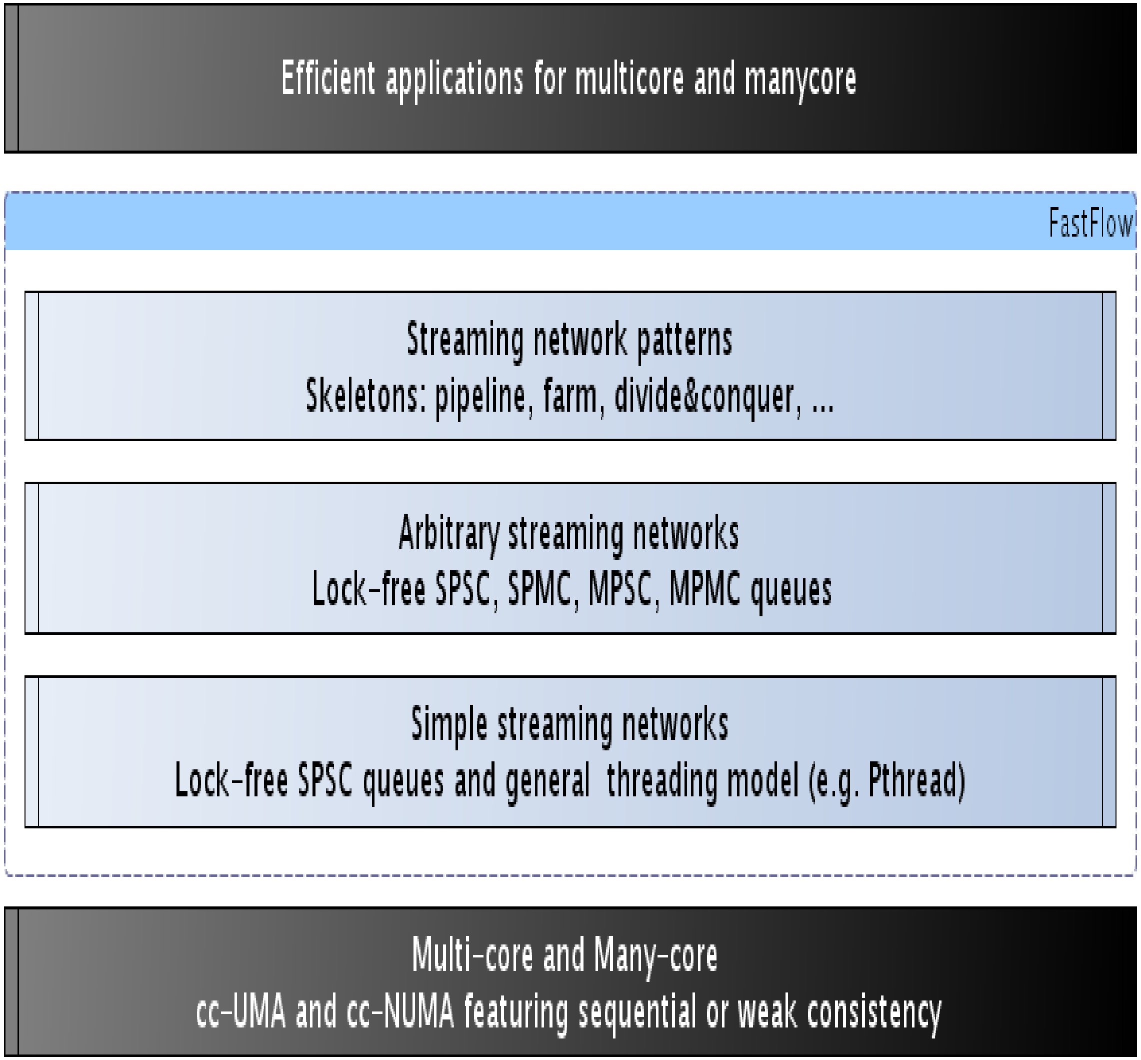
- The FastFlow framework: basic concepts
- From single to many multi-core workstations
    - Two-tier parallel model
    - Definition of the *dnode* concept in FastFlow
- Implementation of communication patterns
    - ZeroMQ as distributed transport layer
    - Marshalling/unmarshalling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# Talk outline

- The FastFlow framework: basic concepts
- From single to many multi-core workstations
  - Two-tier parallel model
  - Definition of the *dnode* concept in FastFlow
- Implementation of communication patterns
  - ZeroMQ as distributed transport layer
  - Marshalling/unmarshalling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# FastFlow parallel programming framework

- Originally designed for shared-cache multi-core
- Fine-grain parallel computations
- Skeleton-based parallel programming model



Efficient applications for multicore and manycore

FastFlow

Streaming network patterns
Skeletons: pipeline, farm, divide&conquer, ...

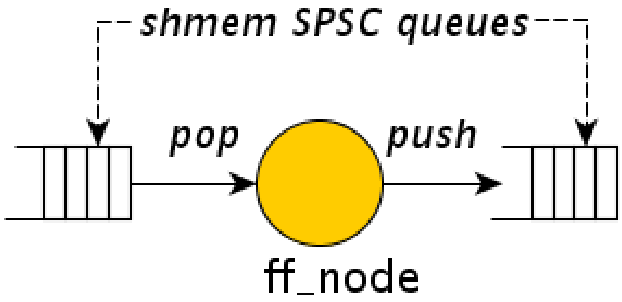
Arbitrary streaming networks
Lock-free SPSC, SPMC, MPSC, MPMC queues

Simple streaming networks
Lock-free SPSC queues and general threading model (e.g. Pthread)

Multi-core and Many-core
cc-UMA and cc-NUMA featuring sequential or weak consistency

# FastFlow basic concepts

- FastFlow implementation
  - based on the concept of *node* (ff_node class)



- A *node* is an abstraction with an input and an output SPSC queue.
- Queues can be bounded or unbounded.
- Nodes are connected one each other by queues.

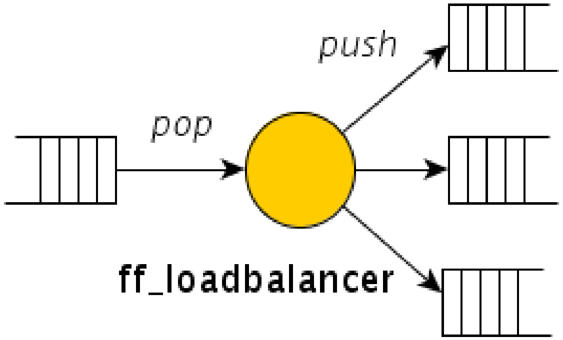
# FastFlow ff_node

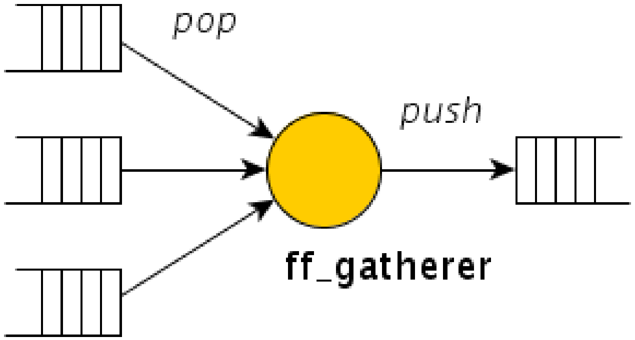
- At *lower level*, FastFlow offers a Process Network (-like) MoC where channels carry shared memory pointers
- Business-logic code encapsulated in the ***svc*** method
- ***svn_init*** and ***svc_end*** used for initialization and termination

```
class ff_node {   // class sketch
protected:
   virtuall bool push(void* data) {
         return qout->push(data);
   }
   virtual bool pop(void** data) {
         return qin->pop(data);
   }
public:
   virtual void* svc(void* task)=0;
   virual int svc_init() { return 0;}
   virtual void svc_end() {}
private:
   SPSC* qin;
   SPSC* qout;} ;
```

# FastFlow ff_node

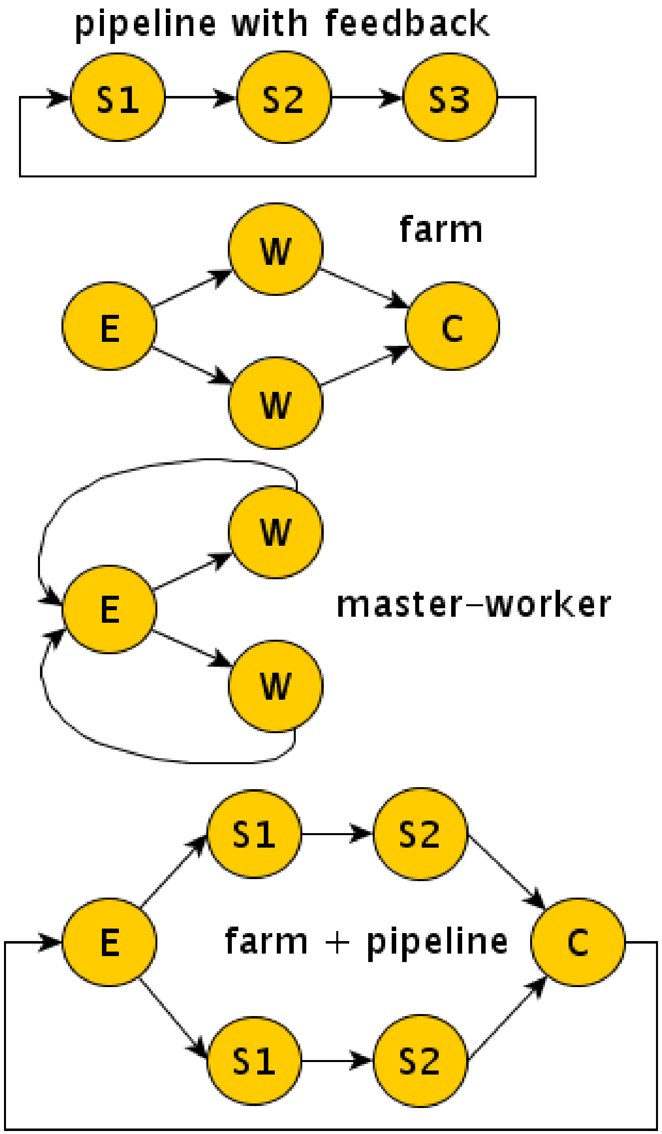
- A sequential node is eventually (at run-time) a POSIX thread
- There are 2 “special” nodes which provide SPMC and MCSP queues using arbiter threads for *scheduling* and *gathering* policy control





# Basic skeletons

- At *higher level*, FastFlow offers a *pipeline* and *farm* skeletons
- Basic skeletons can be composed
- There are some limitations on the possible nesting of nodes when cycles are present



# Talk outline

- The FastFlow framework: basic concepts
- From single to many multi-core workstations
  - Two-tier parallel model
  - Definition of the *dnode* concept in FastFlow
- Implementation of communication patterns
  - ZeroMQ as distributed transport layer
  - Marshalling/unmarshalling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# Extending FastFlow

- Currently, a FastFlow parallel application uses only one single multi-core workstation
    - We are extending FastFlow to target GPGPUs and general-purpose HW accelerators (Tile*Pro*64)
- We need to scale to hundreds/thousands of cores

→ we have to use many multi-core workstations

- The FastFlow streaming network model can be easily extended to work outside the single workstation

# Two tier parallel model

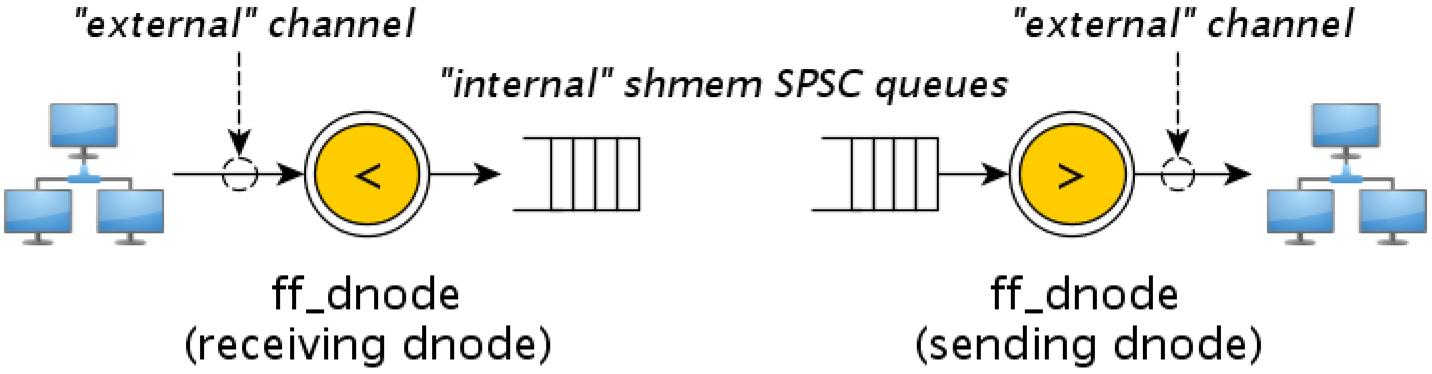
- We propose a two-tier model:
    - **Lower-layer**: supports file grain parallelism on a single multi/many-core workstation leveraging GPGPUs and HW accelerators

    - **Upper-layer**: supports structured coordination of multiple workstations for medium/coarse parallel activities

- The lower-layer is basically the FastFlow framework extended with suitable mechanisms

# From *node* to *dnode*

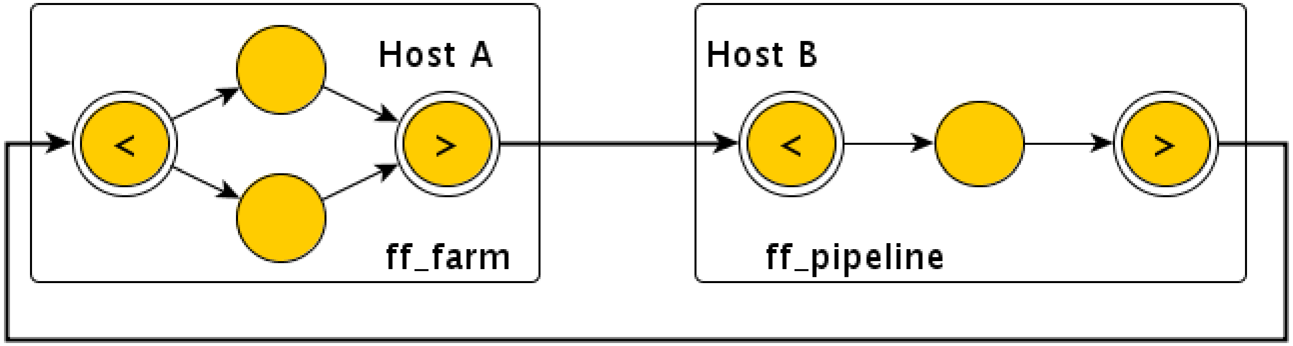
- A dnode (class ff_dnode) is a node (i.e. extends the ff_node class) with an external communication channel:



- The **external channels** are specialized to be input or output channels (not both)

# From *node* to *dnode* (2)

- Idea:only the **edge-node**s of the FastFlow skeleton network are able to “talk to” the outside word.



Above we have 2 FastFlow applications whose edge-node are connected using an unicast channel.

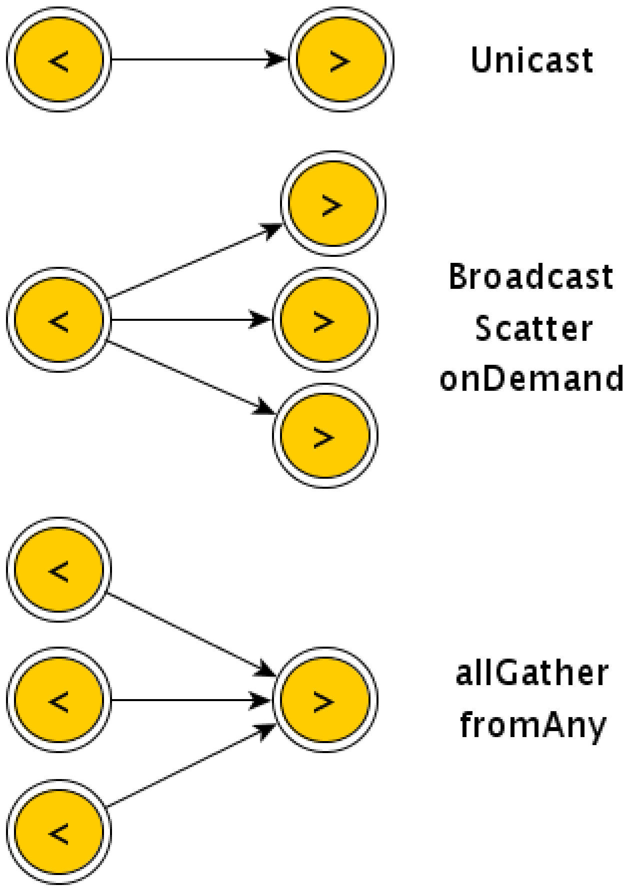
# FastFlow ff_dnode

- The ff_dnode offers the same interface as the ff_node
- In addition it encapsulates the “external channel” whose type is passed as template parameter
- The *init* method initializes the communication end-points

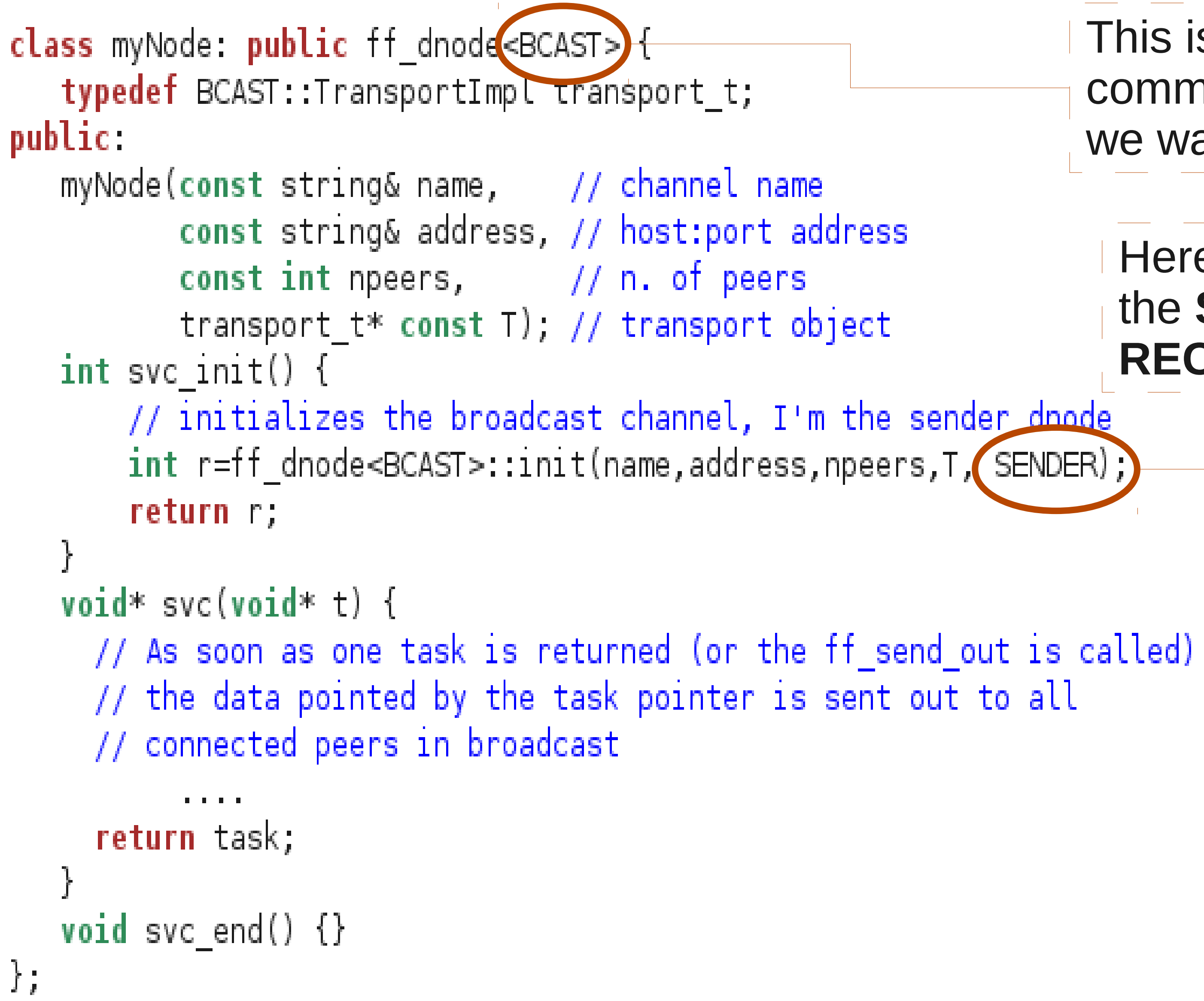
```
template <class CommImpl>
class ff_dnode: public ff_node {
protected:
  virtuall bool push(void* data) {
        …. com->push(data);
  }
  virtual bool pop(void** data) {
        …. com->pop(data);
  }
public:
  int init(...) { ... return com.init(...); }
  int run() { return ff_node::run(); }
  int wait() { return ff_node::wait();}
private:
  CommImpl com;};
```

# Communication patterns

- Possible communication patterns among dnode(s) can be:
  - Unicast
  - Broadcast
  - Scatter
  - OnDemand
  - fromAll (all-Gather)
  - fromAny



# How to define a dnode

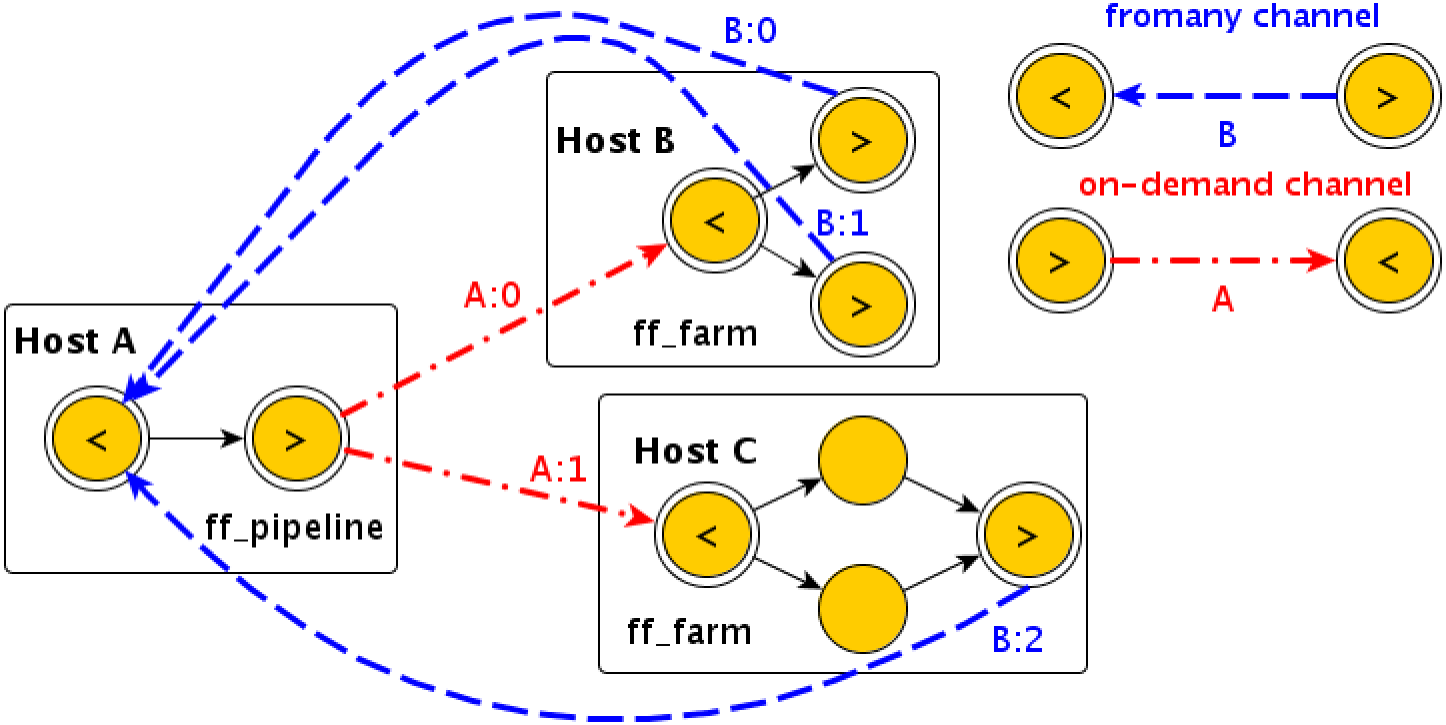


```
class myNode: public ff_dnode<BCAST> {
   typedef BCAST::TransportImpl transport_t;
public:
   myNode(const string& name,    // channel name
          const string& address, // host:port address
          const int npeers,      // n. of peers
          transport_t* const T); // transport object
   int svc_init() {
       // initializes the broadcast channel, I'm the sender dnode
       int r=ff_dnode<BCAST>::init(name,address,npeers,T, SENDER);
       return r;
   }
   void* svc(void* t) {
     // As soon as one task is returned (or the ff_send_out is called)
     // the data pointed by the task pointer is sent out to all
     // connected peers in broadcast
          ....
     return task;
   }
   void svc_end() {}
};
```

This is the communication pattern we want to use

Here we specify if we are the **SENDER** or the **RECEIVER** dnode.

# A possible application scenario



- Both SPMD and MPMD programming models supported.

# Talk outline

- The FastFlow framework: basic concepts
- From single to many multi-core workstations
    - Two-tier parallel model
    - Definition of the *dnode* concept in FastFlow
- Implementation of communication patterns
    - ZeroMQ as distributed transport layer
    - Marshalling/unmarshalling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# Communication pattern implementation

- The current version uses ZeroMQ to implement external channes
- ZeroMQ uses TCP/IP
- Why ZeroMQ?
  - It is easy to use.
  - Runs on most OSs and supports many languages
  - It is efficient enough
  - Offers an asynchronous communication model
  - Allows implementation zero-copy multi-part sends

# Marshalling/Unmarshalling of messages

- Consider the case when 2 or more objects have to be sent as a single message
- If the 2 objects are non-contiguous in memory we have to memcpy one of the two
    - It can be costly in term of performance
- A classical solution to avoid coping is to use POSIX readv/writev (scatter/gather) primitives, i.e. multi-part messages

# Marshalling/Unmarshalling of messages

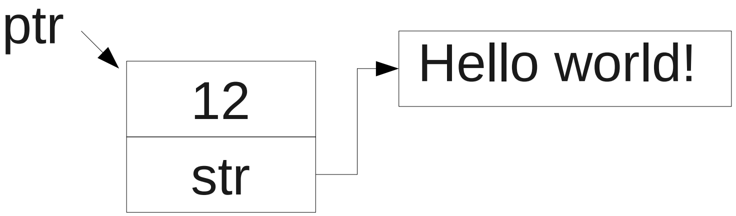
- All communication patterns implemented supports zero-copy multi-part messages
- The *dnode* provides the programmer with specific methods for managing multi-part messages:
  - Sender side: 1 method (prepare) called before data is being sent.
  - Receiver side: 2 methods (prepare and unmarshalling)
    - the 1st called before receiving data, used to give to the run-time the receiving buffers
    - the 2nd one called after all data have been received, used to reorganise data frames.

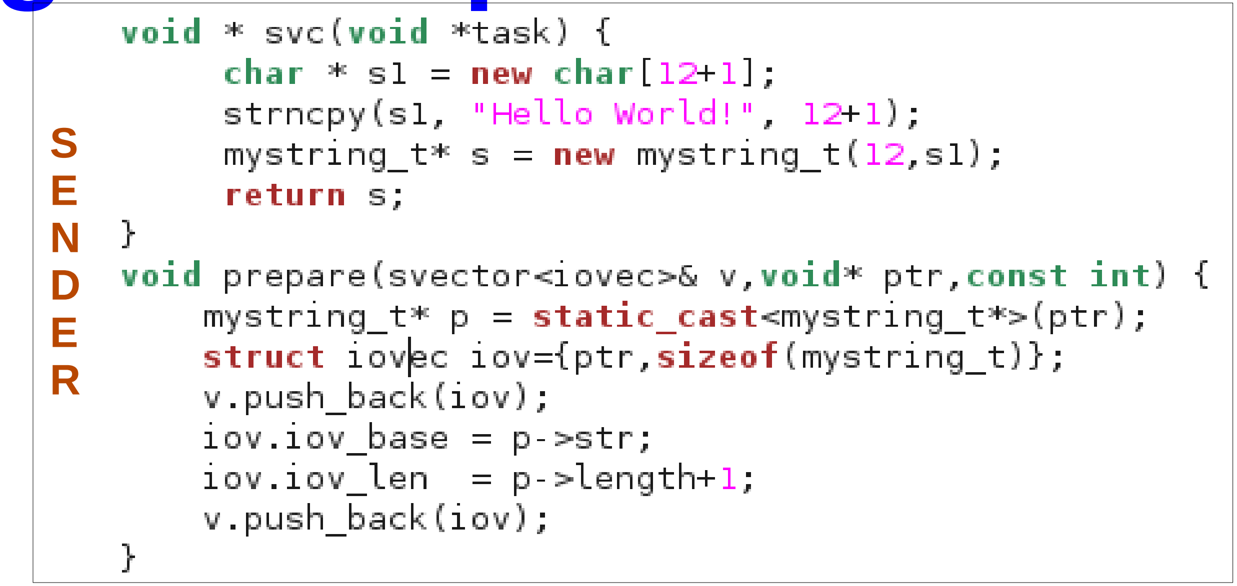
# Marshalling/Unmarshalling: usage example

**Object definition:**

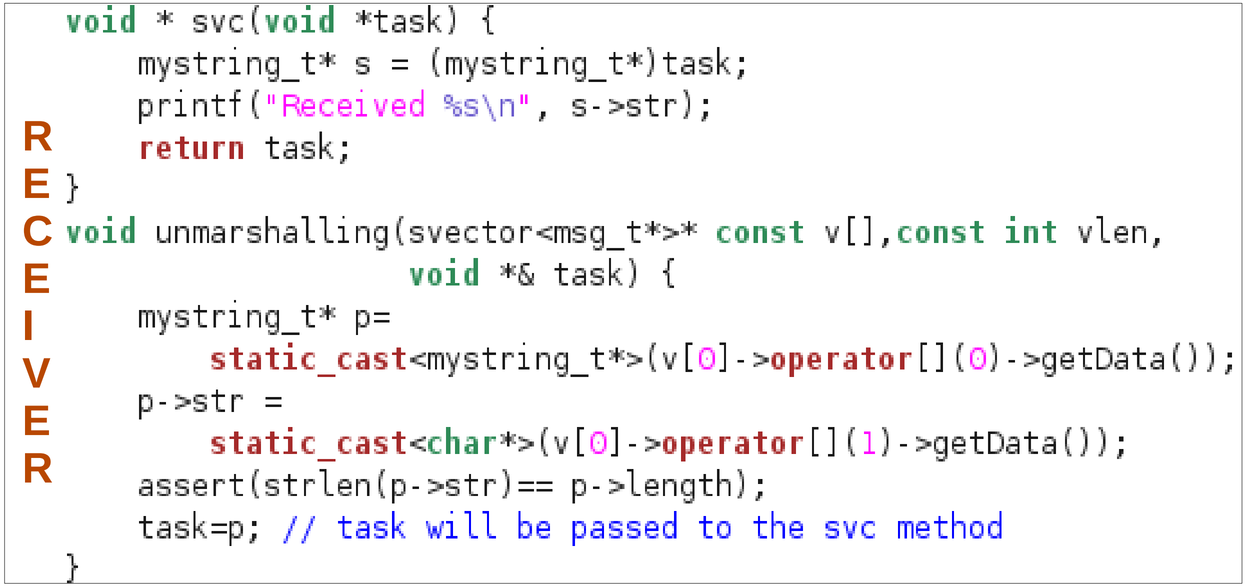
struct mystring_t {
int length;
char* str;
}; mystring_t* ptr;

**Memory layout:**




**SENDER**

```
void * svc(void *task) {
     char * s1 = new char[12+1];
     strncpy(s1, "Hello World!", 12+1);
     mystring_t* s = new mystring_t(12,s1);
     return s;
}
void prepare(svector<iovec>& v,void* ptr,const int) {
    mystring_t* p = static_cast<mystring_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);
}
```


**RECEIVER**

```
void * svc(void *task) {
    mystring_t* s = (mystring_t*)task;
    printf("Received %s\n", s->str);
    return task;
}
void unmarshalling(svector<msg_t*>* const v[],const int vlen,
                   void *& task) {
    mystring_t* p=
        static_cast<mystring_t*>(v[0]->operator[](0)->getData());
    p->str =
        static_cast<char*>(v[0]->operator[](1)->getData());
    assert(strlen(p->str)== p->length);
    task=p; // task will be passed to the svc method
}
```

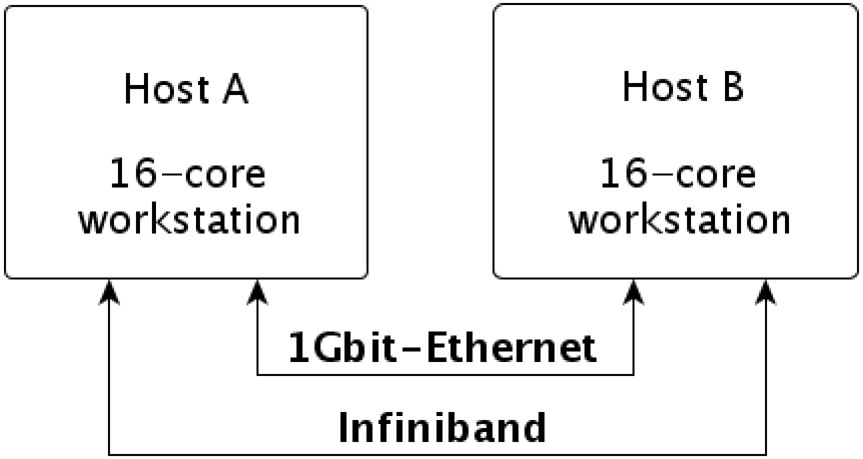
- *prepare* creates 2 iovec for the 2 parts of memory pointed by ptr and str. Two msgs are sent.
- unmarshalling (re-)arranges the received msgs to have a single pointer to the mysting_t object

# Talk outline

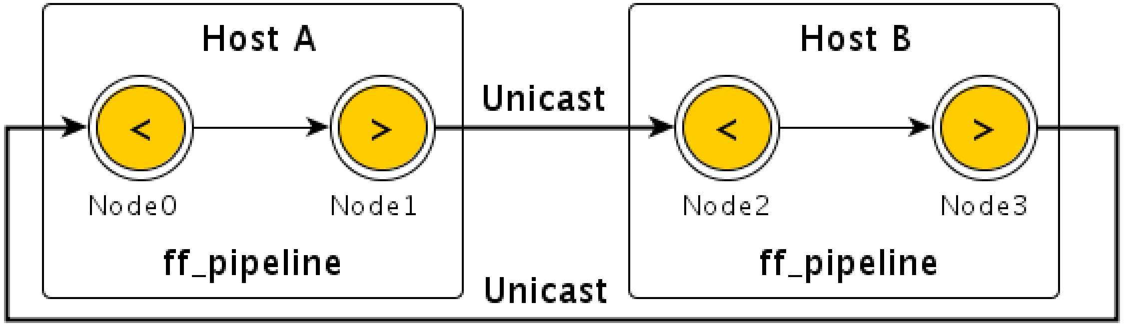
- The FastFlow framework: basic concepts
- From single to many multi-core workstations
    - Two-tier parallel model
    - Definition of the *dnode* concept in FastFlow
- ZeroMQ as distributed transport layer
    - Implementation of communication patterns
    - Marshaling/unmarshaling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# Experiments configuration

- 2 workstations each with 2CPUs Sandy-Bridge E5-2650 @2.0GHz, running Linux x86_64
- 16-cores per Host, 20MB L3 shared cache, 32GB RAM
- 1Gbit-Ethernet and Infiniband Connectx-3 card (40Gbit/s) - no network switch between



# Experiments: Unicast Latency



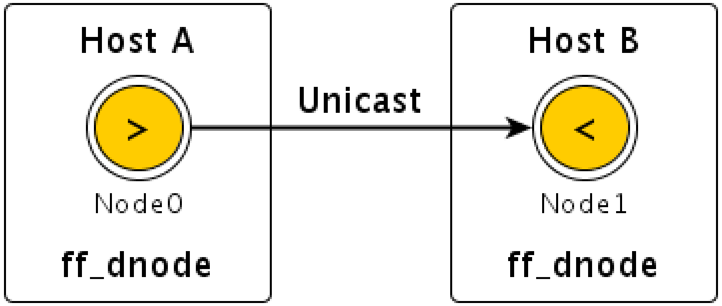
**Minimum Latency**

| msg size | 1Gbit Ethernet | Infiniband IPoIB |
|---|---|---|
| 8-Bytes | 69 us | 27 us |

**Latency test:**

- Node0 generates 8-bytes msgs, one at a time.
- Node1 sends the msg to Node2, Node2 to Node3 and Node3 back to Node0
- As soon as Node0 receives one input msg, it generates another one up to N msgs
- Min.Latency=
  Node0 Time / (2*N)

# Experiments: Unicast Bandwidth



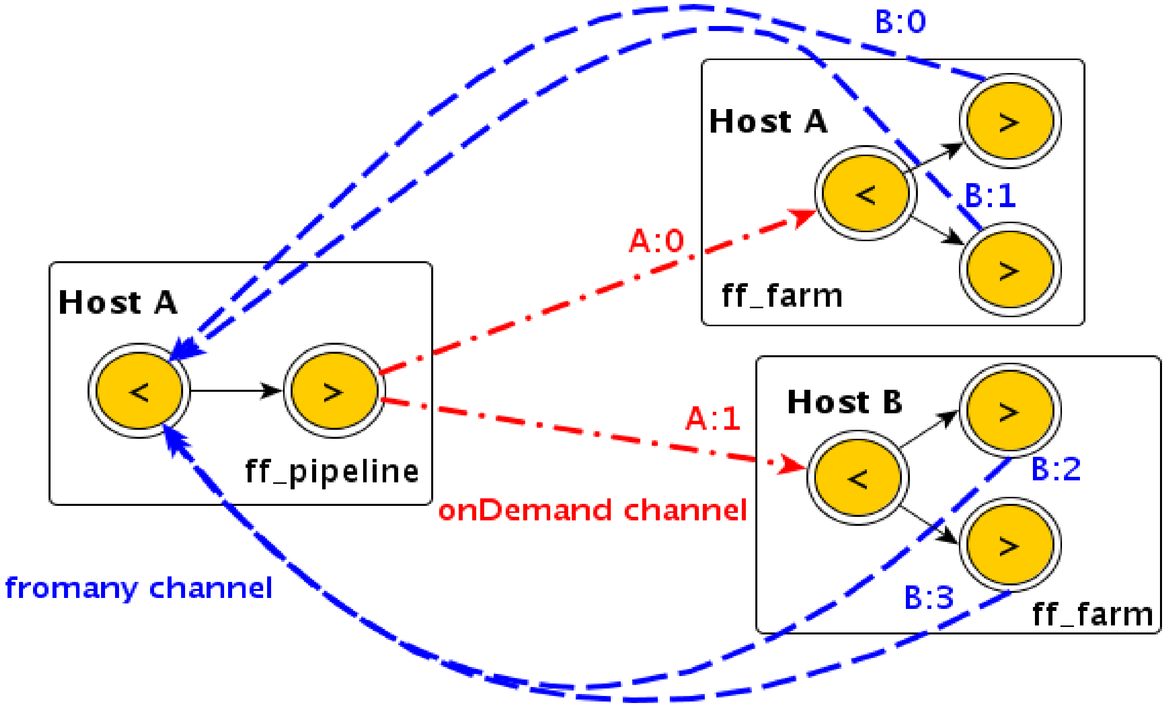
**Bandwidth test:**

- Node0 sends the same msg of size bytes N times.
- Node1 gets one msg at a time and free memory space
- Max.Bwd (Gb/s)=
  N / (Time Node1(s) * size * 8M)

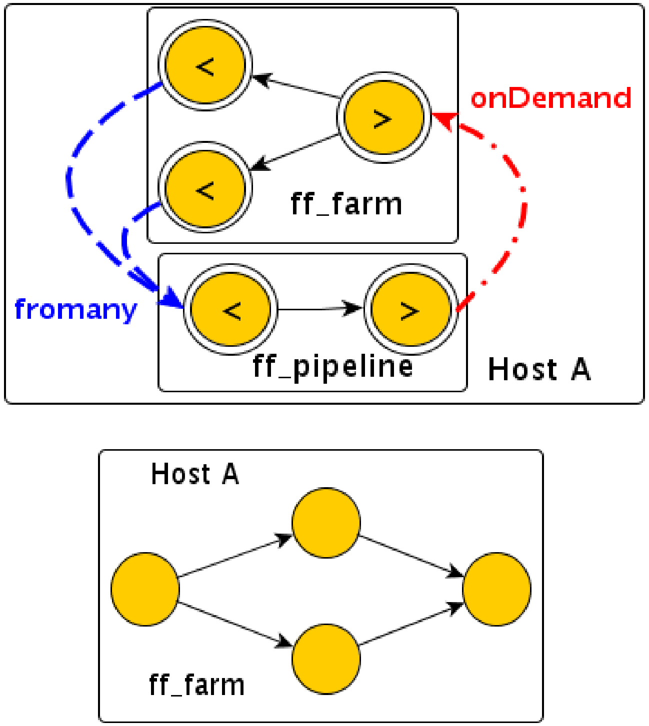
**Maximum Bandwidth**

| msg size | **1Gbit Ethernet** | **Infiniband IPoIB** | |
|---|---|---|---|
| | FastFlow | | iperf 2.0.5 |
| 1K | 0.50 Gb/s | 5.0 Gb/s | 0.6 Gb/s |
| 4K | 0.93 Gb/s | 5.1 Gb/s | 4.8 Gb/s |
| 1M | 0.95 Gb/s | 14.7 Gb/s | 17.6 Gb/s |

# Experiments: Benchmark



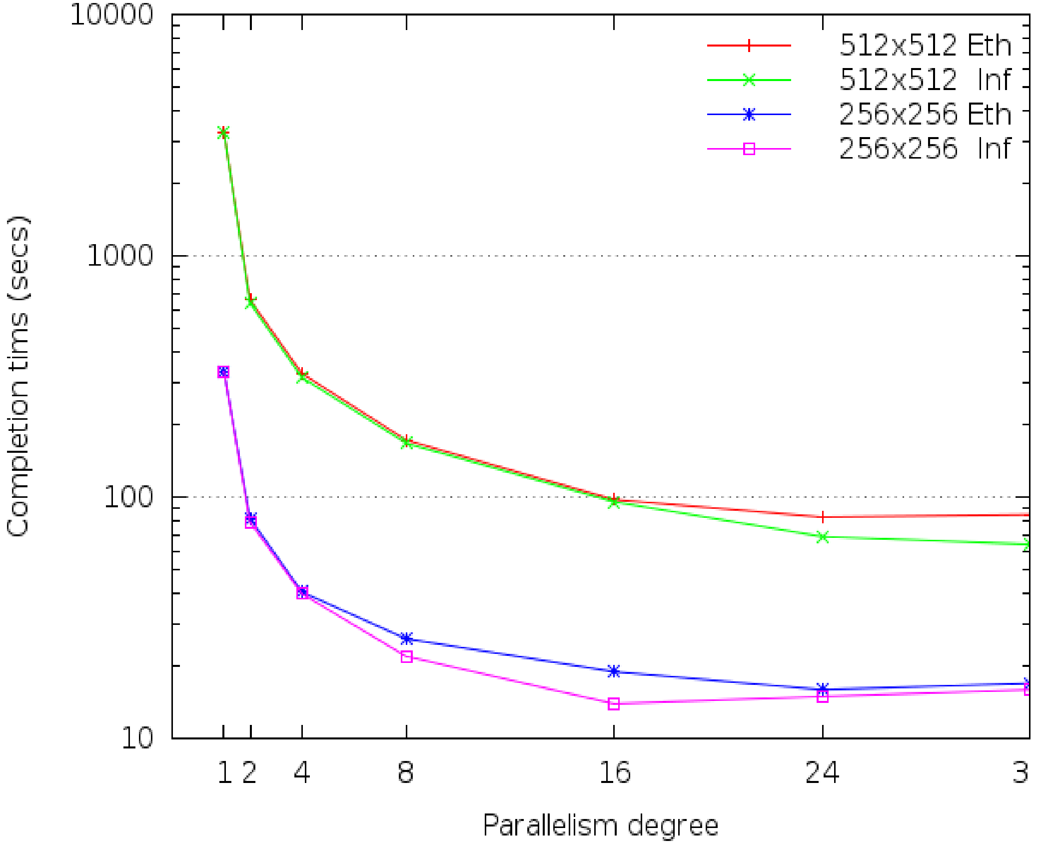
Two host schema

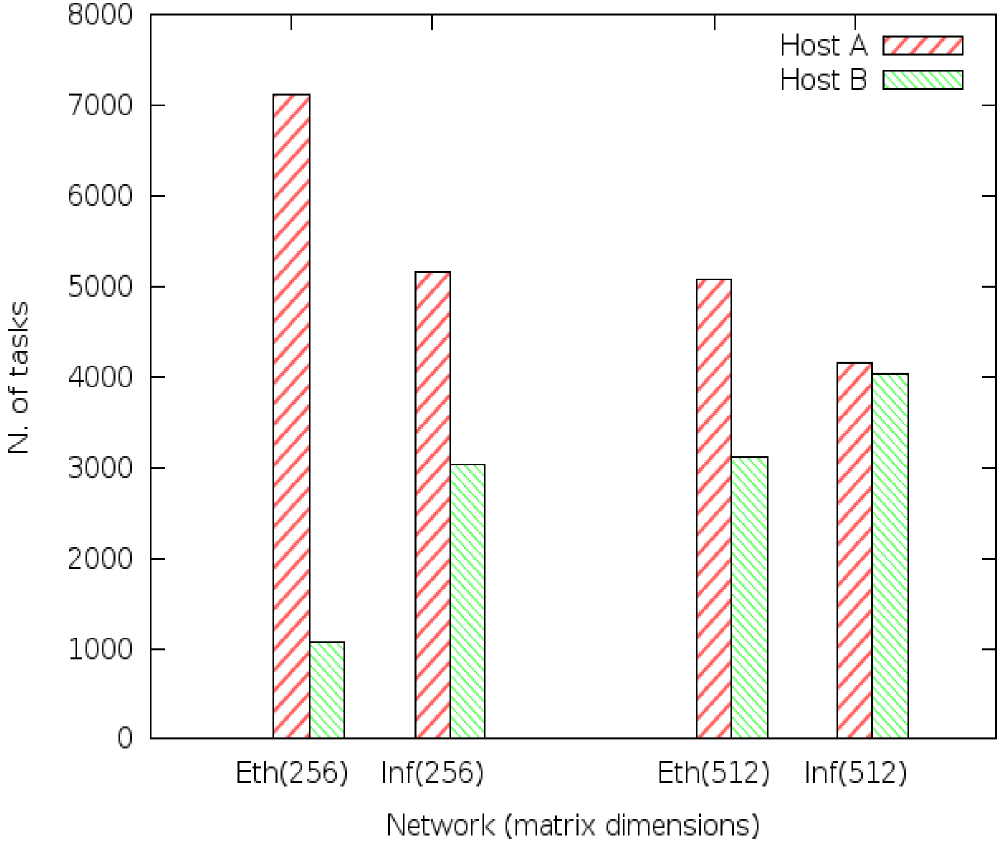


Single host schemas

- Square matrix computation. Input stream of 8192 matrices.
- Two cases tested: 256x256 and 512x512 matrix sizes.
- Parallel schema as in the figures. On the left using 2 hosts, on the right using just 1 hosts.

# Experiments: Benchmark



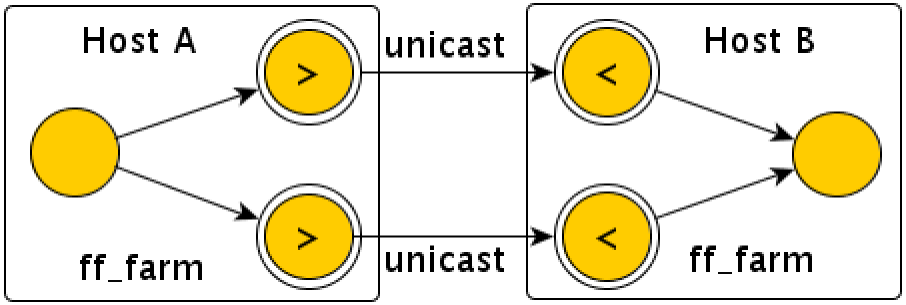


**Max Speedup**

| Mat size | FF | dFF-1 | dFF-2-Eth | dFF-2-Inf |
|---|---|---|---|---|
| 256x256 | 13.6X | 17.6X | 20.8X | 23.8X |
| 512x512 | 16X | 20.6X | 39.2X | 50.9X |

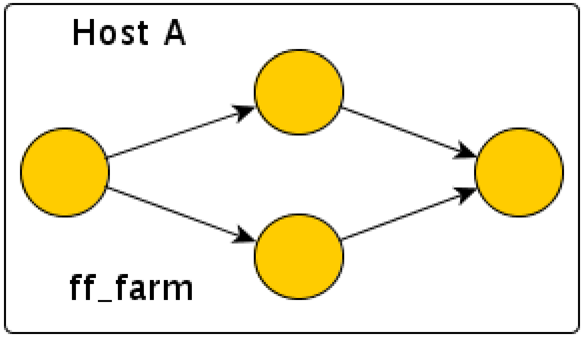
# Experiments: Image application

- Stream of 256 GIF images. We have to apply 2 image filters to each image (blur and emboss).
- Two cases tested: small size images ~ 256KB and coarser size images ~1.7MB.
- Parallel schema as in the figures below. On the left using 2 hosts, on the right using just 1 hosts.



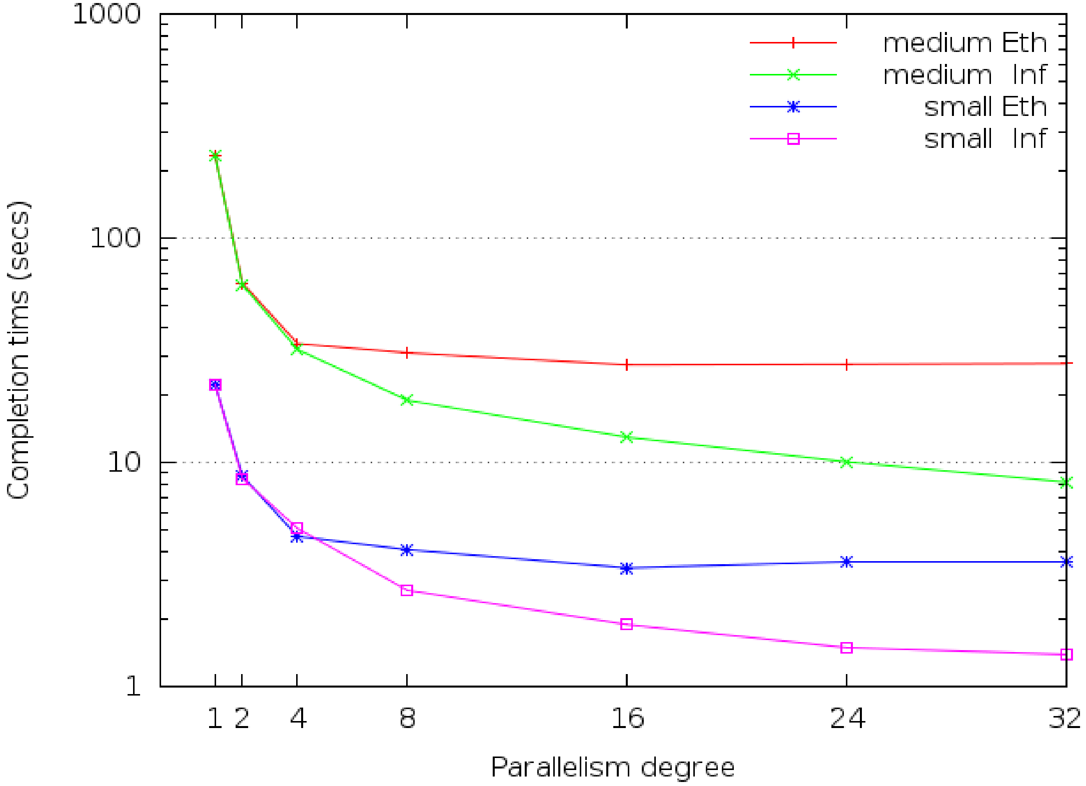
blur filter

emboss filter



blur & emboss filters

# Experiments: Image application



**Max Speedup**

| Image size | FF | dFF-2-Eth | dFF-2-Inf |
|---|---|---|---|
| small | 11.5X | 8X | 19.6X |
| medium | 12X | 8.5X | 28.3X |

**NOTE: Disk transfer time is not considered.**

# Talk outline

- The FastFlow framework: basic concepts
- From single to many multi-core workstations
  - Two-tier parallel model
  - Definition of the *dnode* concept in FastFlow
- ZeroMQ as distributed transport layer
  - Implementation of communication patterns
  - Marshaling/unmarshaling of messages
- Benchmarks and simple application results
- Conclusions and Future Work

# Conclusions & Future Works

- We extended the existing FastFlow programming framerork for targeting distributed systems
    - It is easy enough to add multiple distributed nodes in a FastFlow application
- Preliminar results are fairly good
    - We have to test it on bigger clusters !
- We are currently working at the **higher layer** of our two-tier model in order to provide algorithm skeletons implemented on top of the FastFlow framework.

# Thanks !

## Any questions?

Source code available within the SourceForge svn

FastFlow web-site:

http://mc-fastflow.sourceforge.net/