Implementation of new parallel processing scheme in basf2

R.Itoh, KEK

Introduction

- Basf2 has been restructured to have two layers.

- kbasf2 (basf2 kernel) serves as the module driver of the framework which manages the registration, parameter manipulation and execution of modules.

- pbasf2 (basf2 parallel) is implemented as a “super-framework” for kbasf2 and manages parallel processing of kbasf2.

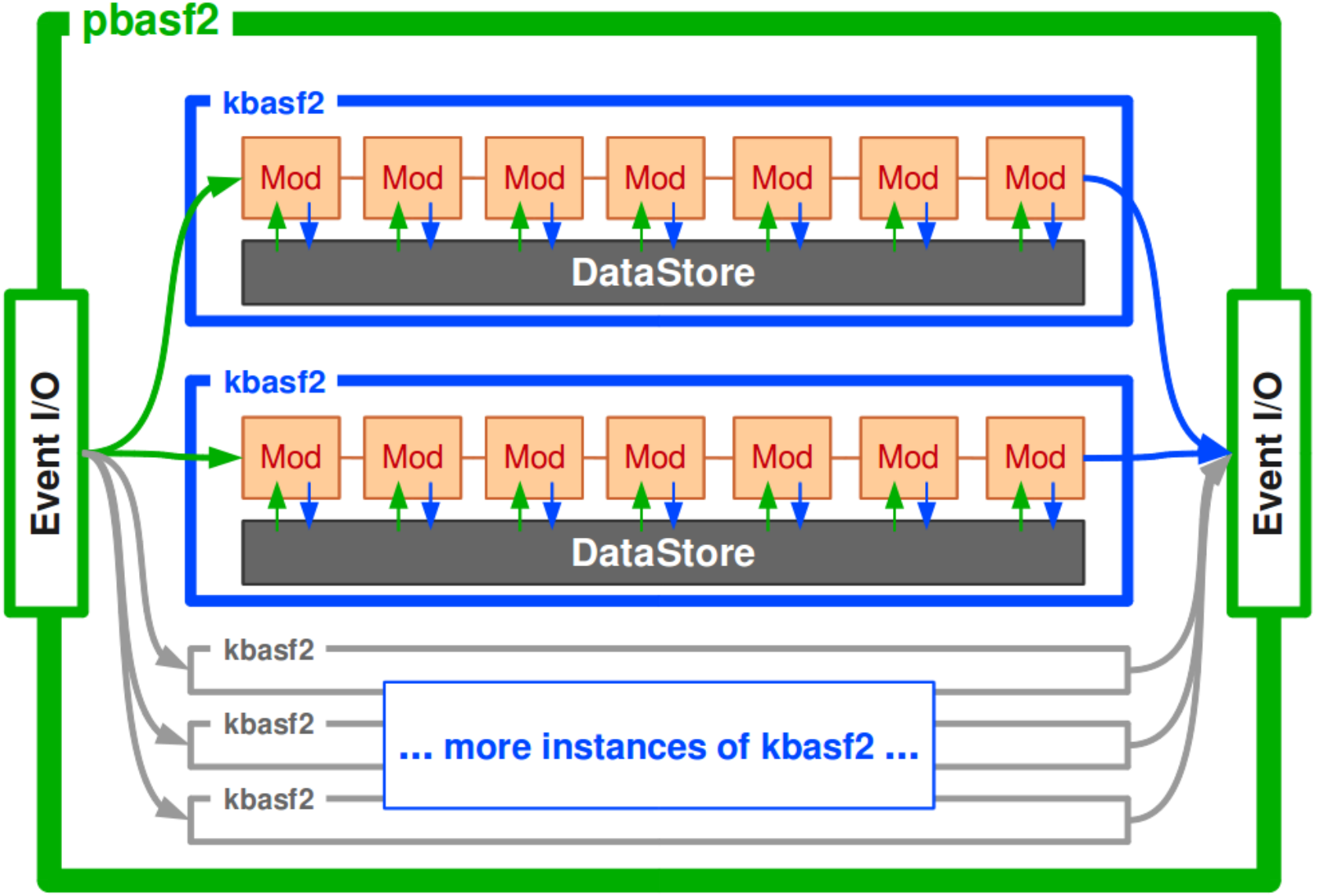
- “basf2” is the generic name of pbasf2+kbasf2 complex.

- The execution of kbasf2/pbasf2 is invoked through Python script.

- Input/output data streams are managed by “modules”.
  <- separated from Framework.

- Histogram management is also provided by a module.

- kbasf is supposed to run as a standalone application
  in a separate process



- Parallel processing capability in basf2 is already implemented and is ready to use by users.
  (Tested with Jan.6 belle2 library used in HLT test.)
  -> See basf2 Twiki for the usage.

http://b2comp.kek.jp/~twiki/bin/view/Computing/Basf2manual

- You don't have to prepare any specials for parallel processing.
- The module on the path are automatically executed in parallel for different event data utilizing multi-core CPUs.
- The parallel processing can be turned on by inserting a single line in your steering script:

  fw.set_nprocess (N)

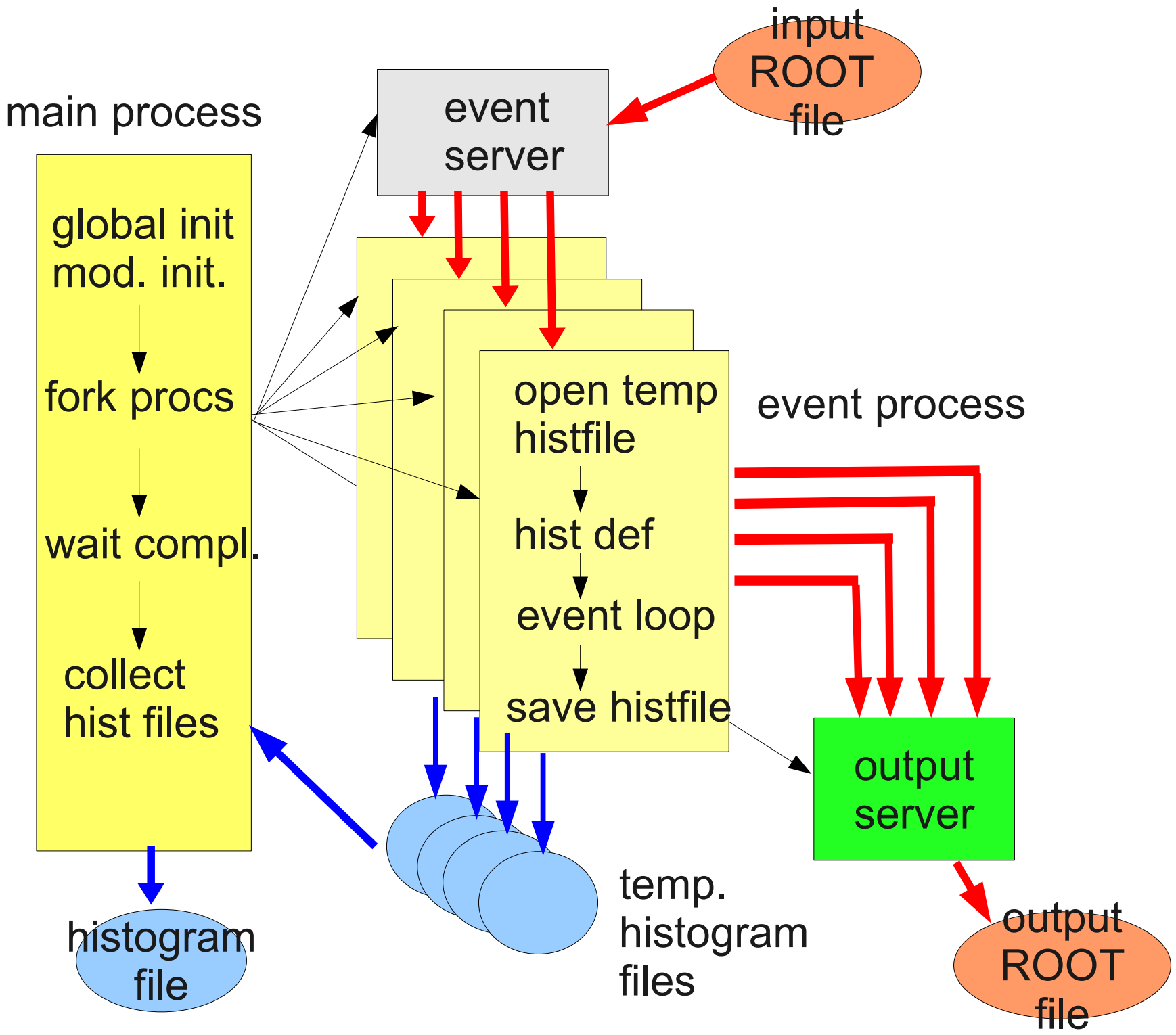
  where N is the number of processes for parallel processing. (could be number of cores in your PC). If set to 0 (default), it invokes kbasf in a single process.

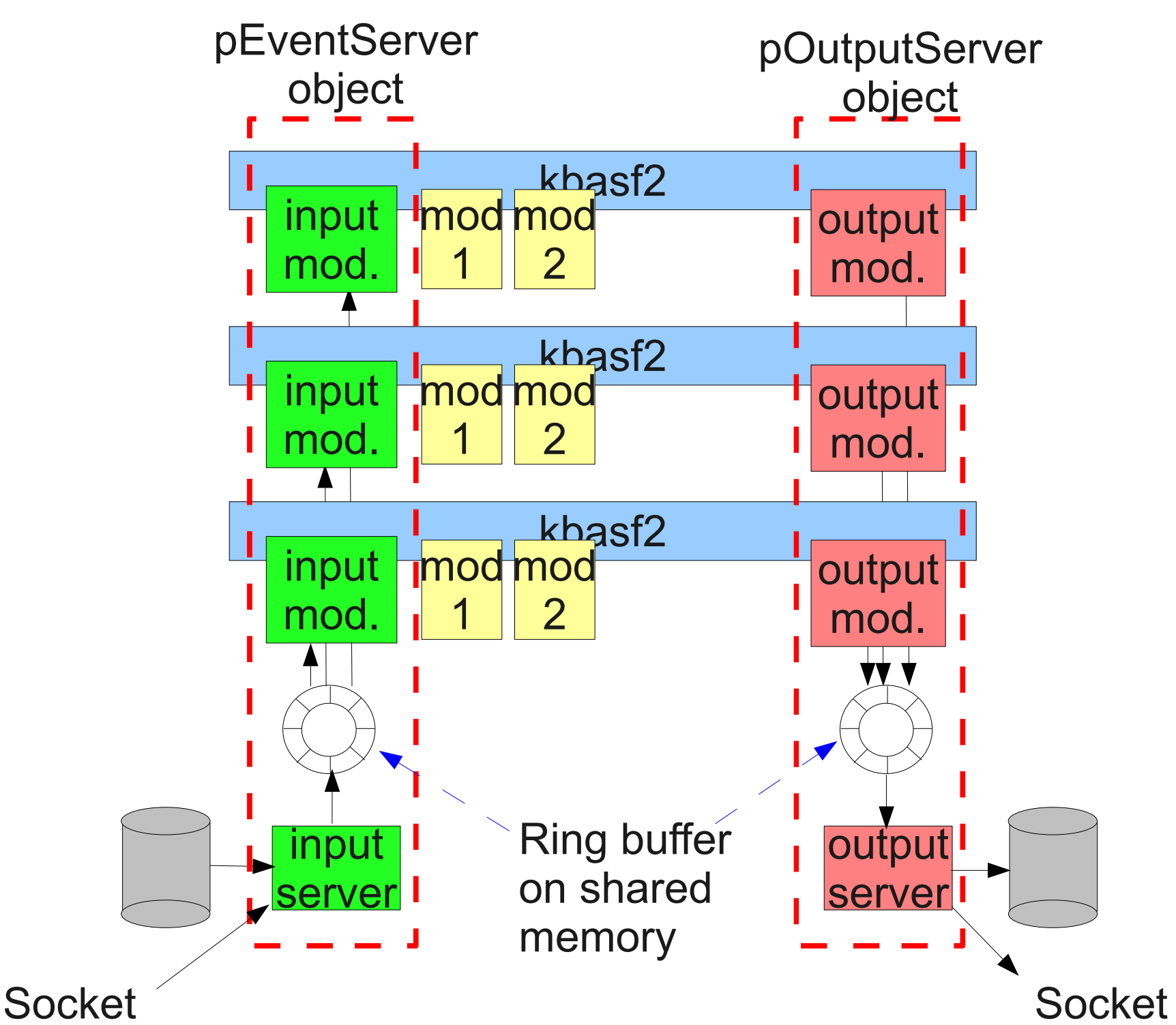
- One note: the input and output persistency modules have to be capable of parallel processing. You need to use

  pRootInput instead of SimpleImput as input module,
  pRootOutput instead of SimpleOutput as output module.

Script used for “linearity test”

```
import os
from basf2 import *

#Register modules
input       = fw.register_module("pSeqRootInput")
evtmetagen  = fw.register_module("EvtMetaGen")
evtmetainfo = fw.register_module("EvtMetaInfo")
paramloader = fw.register_module("ParamLoaderXML")
geobuilder  = fw.register_module("GeoBuilder")
g4sim       = fw.register_module("SimModule")
cdcdigitizer = fw.register_module("CDCDigitizer")
output = fw.register_module("pSeqRootOutput")

#Set parameters
.....

input.param("inputFileName", "SeqSimDriveOutput.sroot" )
output.param("outputFileName", "DataFlow.sroot" )

#Create paths
main = fw.create_path()

#Add modules to paths
main.add_module(input)
main.add_module(evtmetagen)
main.add_module(paramloader)
main.add_module(geobuilder)
main.add_module(g4sim)
main.add_module(cdcdigitizer)
main.add_module(output)

#Process events
fw.set_nprocess(15)
fw.process(main)
```


main process
global init
mod. init.
fork procs
wait compl.
collect
hist files
histogram
file
event
server
input
ROOT
file
open temp
histfile
hist def
event loop
save histfile
event process
output
server
temp.
histogram
files
output
ROOT
file


pEventServer object
pOutputServer object
kbasf2
input mod.
mod 1
mod 2
output mod.
kbasf2
input mod.
mod 1
mod 2
output mod.
kbasf2
input mod.
mod 1
mod 2
output mod.
input server
Ring buffer on shared memory
output server
Socket
Socket

- Current implementation needs special input/output module for parallel processing.

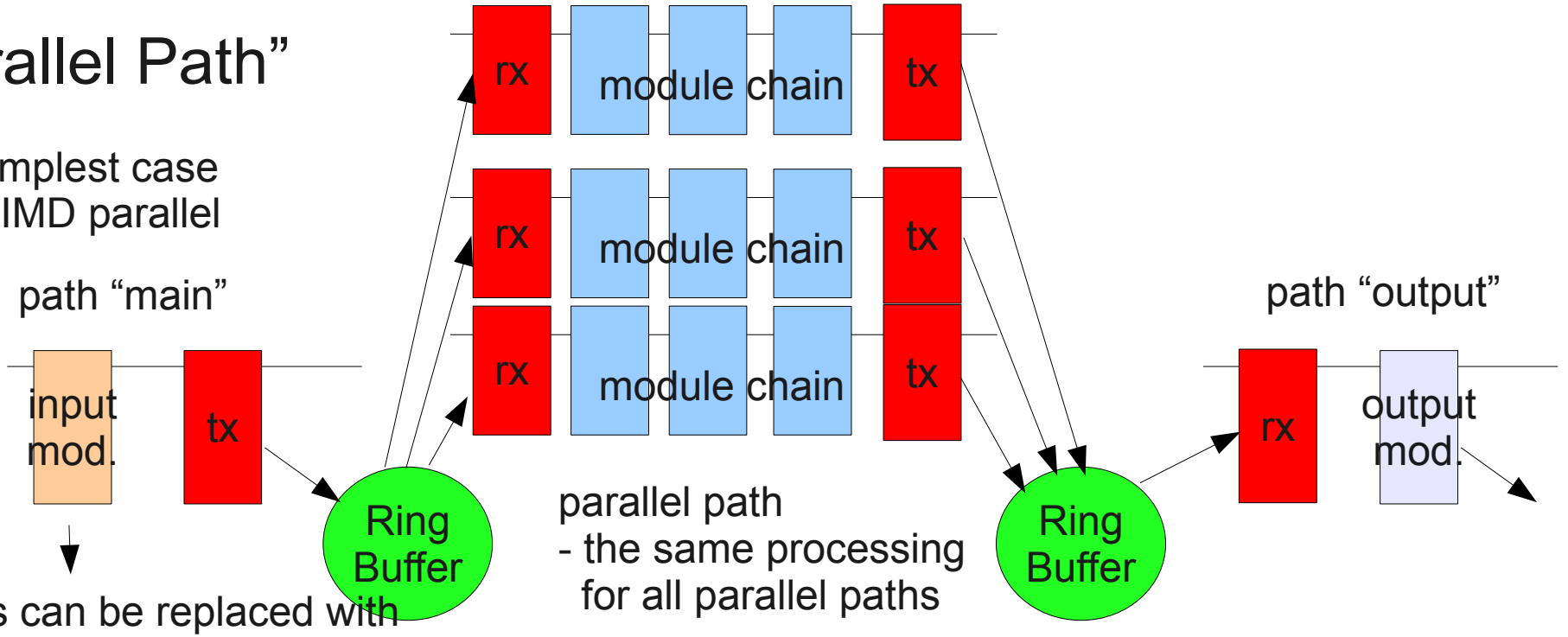
- When using event generators as an input module, a consistent management of random number seed in different processes could be a problem.

- Better implementation was suggested by Thomas.

- Andreas and I tried to generalize his idea and are working on the new implementation.

“Parallel Path”

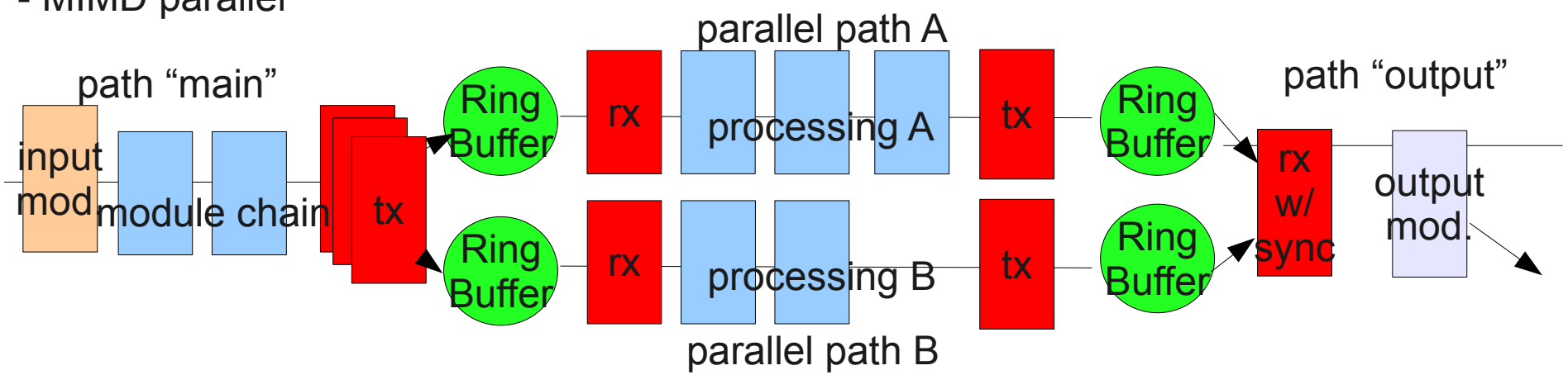
a) Simplest case
- SIMD parallel



This can be replaced with “EvtGen” for example
-> No worry for random number seed

b) Generalized parallel path
- MIMD parallel

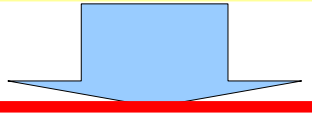
Different “parallel” processing for each path



- Module-by-module pipeline parallel processing becomes possible by repeating this.
  (i.e. different module on different CPU core!)

* tx->RingBuffer->rx
  - “tx” streams all (or selected) objects in DataStore in a record of RingBuffer.
  - “rx” picks up a record from RingBuffer, destream it, and restore DataStore.
  - These procedure is generic in any cases and can be unified.
  - Could be implemented in path processing function so that it becomes transparent to users.
  - “Synchronization” of records at “rx” in output path is required for MIMD parallel case.

* Management of parallel path
  - Should be integrated in the framework as a part of path management function
  - How?



- RingBuffer handling will be implemented inside framework in the path management so that it becomes transparent to users.

- “Parallel path” is implemented as some “internal path” which is not seen from users.

- “Parallel” specification in a path, example:
  main.set_path (“evtgen,[g4sim,recon1,recon2],evsel,output”)
  parallel path
  * How to manage jump to other path in parallel path?
  * Many things to consider before the actual implementation.

Idea for parallel path

- “DataStoreTransport” object :
  * Basic class to manage object passing between paths in different processes.
  - DataStoreTransport object first creates a RingBuffer.
    DataStoreTransport::DataStoreTransport (name)
    If RingBuffer with the name already exists, just attach to it.
  - DataStoreTransport::send(EDurability)
    Stream objects specified by EDurability in DataStore into RingBuffer
  - DataStoreTransport::receive(EDUrability& durability)
    Pick up a record from RingBuffer, destream it, and restores objects in DataStore.

- For now, we stick to the simplest case = a).
  * One input path, one parallel path, and one output path.
    <- We don't think about having “multiple” or “complicated” parallel paths.

  * When a parallel path is defined, two DataStoreTransport objects are created for input and output.

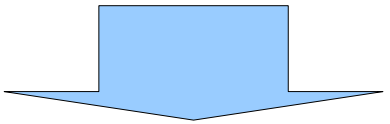
  * “Input path” is equivalent to path “main” in basf2.

Current implementation parallel processing in basf2

- In parallel path, it is necessary to pass DataStore objects between paths running on different processes.

- What is done in current p(seq)rootinput module:
  * fork out event_server process at the initialization.
  * event_processes are forked out at start.
  * event_server reads an event record into ROOT from a data file, stream it in a RingBuffer.
  * event_process picks up a record from the RingBuffer, destream it, and place objects in DataStore.

- p(seq)rootoutput module does the same in a reverse way.



generalization of DataStore passing between processes over RingBuffer

Streaming “DataStore” object

- Objects stored in “DataStore” have to be streamed to pass between processes/nodes.

- DataStore has 3 “durabilities” each of which has two stores = StoreObjects and StoreArrays

* DataStore objects are streamed in “EvtMessage” class object which can be directly placed in RingBuffer.

```
// initialize()
  for (int ii = 0; ii < c_NDurabilityTypes; ii++) {
    m_obj_iter[ii]   =
DataStore::Instance().getObjectIterator(static_cast<EDurability>(ii));
    m_array_iter[ii]
DataStore::Instance().getArrayIterator(static_cast<EDurability>(ii));
  }
```

```
// Collect objects and place them in msghandler
// 1. Stored Objects
m_obj_iter[durability]->first();
int nobjs = 0;
while (!m_obj_iter[durability]->isDone()) {
  m_msghandler->add ( m_obj_iter[durability]->value(), m_obj_iter[durability]->key() );
  nobjs++;
  m_obj_iter[durability]->next();
}
// 2. Stored Arrays
m_array_iter[durability]->first();
int narrays = 0;
while (!m_array_iter[durability]->isDone()) {
  m_msghandler->add ( m_array_iter[durability]->value(), m_array_iter[durability]->key());
  narrays++;
  m_array_iter[durability]->next();
}
// Encode EvtMessage
EvtMessage* msg = m_msghandler->encode_msg ( MSG_EVENT );
```

```
// Decode message
vector<TObject*> objlist;
vector<string> namelist;
int status = m_msghandler->decode_msg ( evtmsg, objlist, namelist );

// Restore objects in DataStore
// 1. Objects
for ( int i=0; i<nobjs; i++ ) {
  DataStore::Instance().storeObject (
                      objlist.at(i),namelist.at(i), durability );
}
for ( int i=0; i<narrays; i++ ) {
  DataStore::Instance().storeArray (
                  (TClonesArray*)objlist.at(i+nobjs),
                  namelist.at(i+nobjs), durability );
}
```

RingBuffer class

```
class RingBuffer {
public:
  /*! Constructor by creating a new shared memory */
  RingBuffer(char* name, int size);    // Create / Attach Ring buffer
  /*! Constructor by attaching to an existing shared memory */
  RingBuffer(int shmid);               // Attach Ring Buffer
  /*! Destructor */
  ~RingBuffer();
  /*! Function to detach and remove shared memory*/
  void cleanup(void);

  /*! Append a buffer in the RingBuffer */
  int insq(int* buf, int size);
  /*! Pick up a buffer from the RingBuffer */
  int remq(int* buf);
  /*! Returns number of buffers in the RingBuffer */
  int numq(void);
  /*! Clear the RingBuffer */
  int clear(void);
```

- DataStoreTransport class can be easily implemented by combining the streaming technique of DataStore and RingBuffer class.

How to describe parallel path in the script

Just like this.....

```
input = fw.create_path()
process = fw.create_path()
output = fw.create_path()

process.set_parallel ( input, output )
```

* How to manage/describe conditional branch in parallel path??

Currently I have no smart idea on the script description.....

Schedule

- I will try to implement the new parallel processing scheme next month aiming at a “show-case” release in next b2gm.

Backup Slides

EvtMessage : serialized object passed between cores/PC nodes

Header (16words)

word 0 : Number of bytes in this record
word 1 : Type RECORD_TYPE
word 2,3 : Time stamp (gettimeofday() format)
word 4 : source of this message
word 5 : dest. of this message
word 6-15 : Reserved (used to store durability, nobjs and narrays for DataStore objects)

List of serialized objects (word 16-)

Streamed object :

word 1 : nbytes of object name
bytes : object name
word n : nbytes of streamed object
bytes : streamed object