Online Reconstruction API - CM45

Y. Karadzhov

UNIGE - DPNC

July 28, 2016

Processors

BaseProcessor is an interface class that defines the minimal implementation required from the user.
It is a kind of similar to the IModule class used in Maus.

```
class BaseProcessor {
public:
  BaseProcessor()=delete; // Never use this default constructor!!!
  BaseProcessor(std::string n);
  ...
  virtual void init(std::string)=0;
  virtual bool process()=0;
  virtual void close()=0;
  ...
protected:
  std::string name_;
  int processCount_;
  double time_spent_;
};
```

This interface class will be inherited by abstract processor classes defining the input and output data types.

Processors

The processors are meant to divide the job into small pieces. The first processor in the chain is the input and the last one is the output.

```
template <class objOut>
class InProcessor : public BaseProcessor {
public:
  InProcessor()=delete; // Never use this default constructor!!!
  InProcessor(std::string n) : BaseProcessor(n), output_(nullptr) {}
  ...
protected:
  objOut **output_;
};

template <class objIn>
class OutProcessor : public BaseProcessor {
public:
  OutProcessor()=delete; // Never use this default constructor!!!
  OutProcessor(std::string n) : BaseProcessor(n), input_(nullptr) {}
  ...
protected:
  objIn  **input_;
};
```

Processors

A more general definition is required for a processor, which will work in the middle of the chain. The input and the output of this processor are independent and may have different types.

```
template <class objIn, class objOut>
class InOutProcessor : public BaseProcessor {

public:
  InOutProcessor()=delete;  // Never use this default constructor!!!
  InOutProcessor(std::string n)
  : BaseProcessor(n), input_(nullptr), output_(nullptr) {}
  ...
protected:
  objIn  **input_;
  objOut **output_;
};
```

Again, the definition of this class is a kind of similar to the Maus Mappers.

Fifo

In order to implement Producer-Consumer pattern we need a bounded buffer.

```
template <typename T>
class Fifo {
 public:
  Fifo(std::size_t s);
  Fifo(const Fifo&)=delete; // disable copying
  Fifo(const Fifo&&)=delete;// disable moving
  Fifo& operator=(const Fifo&)=delete; // disable assignment
  Fifo& operator=(const Fifo&&)=delete;

  T pop();
  void push(const T& item);
  ...
 private:
  std::queue<T> queue_;
  std::atomic_size_t max_size_;
  std::atomic_int    n_producers_;
  std::atomic_int    n_consummers_;

  std::mutex mutex_;
  std::condition_variable cond_empty_;
  std::condition_variable cond_full_;
};
```

Fifo + Processor + Fifo = Worker.

Base Interface class:

```
class WInterface {
public:
  WInterface() = delete;  // Never use this default constructor!!!
  WInterface(std::string n, BaseProcessor *p, int id=0);

  void init(std::string s=);
  virtual void start(int n) =0;
  void close();
  bool process();
  ...
protected:
  BaseProcessor *processor_;

private:
  std::string name_;
  int id_;
}
```

Fifo + Processor + Fifo = Worker.

abstract input and output classes:

```
template <class outDataType>
class WOutput {
public:
  WOutput();
  virtual ~WOutput();
  ...
protected:
  void push();
  outDataType **output_;
  std::shared_ptr< Fifo<outDataType*> > fifo_out_;
};
template <class inDataType>
class WInput{
public:
  WInput();
  virtual ~WInput();
  ...
protected:
  bool pull();
  inDataType  **input_;
  std::shared_ptr< Fifo<inDataType*> > fifo_in_;
};
```

TransformWorker

The most general definition of a worker is the TransformWorker:

```
template <class inDataType, class outDataType>
class TransformWorker : public WInterface,
public WInput<inDataType>, public WOutput<outDataType> {
 public:
  TransformWorker() = delete;  // Never use this default constructor!
  TransformWorker(string n,
                  InOutProcessor<inDataType, outDataType> *p,
                  int id)
  : WInterface(n, p, id), WInput<inDataType>(),
    WOutput<outDataType>() {
    p->setInputObj( WInput<inDataType>::input_ );
    p->setOutputObj( WOutput<outDataType>::output_ );
  }

  virtual ~TransformWorker() {}

  void start(int n) final;

private:
  void stopWork();
};
```

TransformWorker

```
template <class inDataType, class outDataType>
void TransformWorker<inDataType, outDataType>::start(int n) {

  int i=0;
  while(i<n || n==0) {
    if( !this->pull()) {
      this->stopWork();
      return;
    }

    *(WOutput<outDataType>::output_) = new outDataType;
    if( !this->process() )
      this->stopWork();

    delete *(WInput<inDataType>::input_);
    this->push();
    ++i;
  }

  this->stopWork();
}
```

Some other type of workers are:

```
template <class outDataType>
class InputWorker
: public WInterface, public WOutput<outDataType> {
...
};

template <class inDataType>
class OutputWorker
: public WInterface, public WInput<inDataType> {
...
};

template <class dataType>
class UpdateWorker : public WInterface,
public WInput<dataType>, public WOutput<dataType> {
...
};
```

Example

Lets define 3 dummy processors:

This processor class is a producer. The output data type is int.

```
class t_input : public InProcessor<int> {
 public:
  t_input(): InProcessor("t_input") {}

  void init(string s) {}
  bool process();
  void close() {}
};

bool t_input::process() {
  /// Simulate some work here.
  **output_ = processCount_ + 220;
  std::this_thread::sleep_for( milliseconds(5) );
}
```

Example

This processor class is a consumer and producer. The input data type is int and the output type is string.

```
class t_proc : public InOutProcessor<int, std::string> {
 public:
  t_proc() : InOutProcessor("t_proc") {}

  void init(string s) {}
  bool process();
  void close() {}
};

bool t_proc::process() {
  /// Do some work here.
  ...
}
```

Example

This processor class is a consumer. The input data type is string

```
class t_out : public OutProcessor<std::string> {
public:
  t_out() : OutProcessor("t_out") {}

  void init(string s) {}
  bool process();
  void close() {}
};

bool t_out::process() {
  /// Do some work here.
  ...
}
```

Example

For each processor, define a worker.

```
IMPLEMENT_INPUT(t_input_worker,   // worker name
                int,              // data type
                t_input)          // proce name

IMPLEMENT_TWORKER(t_proc_worker, // worker name,
                  int,           // input data type
                  std::string,   // output data type
                  t_proc)        // processor name

IMPLEMENT_OUTPUT(t_output_worker, // worker name
                 std::string,     // data type
                 t_out)           // processor name
```

Example

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

  /// Hire 4 workers.
  t_input_worker  w0(0);
  t_proc_worker   w1(1), w2(2);
  t_output_worker w3(3);

  /**Position the 4 workers.
  *
  *        |--> w1 -->|
  *        |          |
  *   w0-->|          |--> w3
  *        |          |
  *        |--> w2 -->|
  */

  w0.getOutput() >> w1 >> w2;

  w1 << w3.getInput();
  w2 << w3.getInput();
```

Example

```
//    // Start the work. Perform 1000 iterations.
  vector<thread> threads;
  threads.push_back( thread( JOB_STAR_N(w0, 100) ));
  threads.push_back( thread( JOB_STAR(w1) ));
  threads.push_back( thread( JOB_STAR(w2) ));
  threads.push_back( thread( JOB_STAR(w3) ));

  for (auto &t:threads)
    t.join();

  return 0;
}
```

Conclusion

- An API for parallel computation has been developed for the purposes of the MICE Online reconstruction.
- The API is very simple, flexible and scalable.
- The most up-to date version of the API, including user examples, is available at https://github.com/yordan-karadzhov/mic11api
- Mic11 currently uses a bit older version of this API but this divergence will disappear in the next release of the code.