rabit

Release 1.0

rabit developers

February 27, 2016

Contents

rabit is a light weight library that provides a fault tolerant interface of Allreduce and Broadcast. It is designed to support easy implementations of distributed machine learning programs, many of which fall naturally under the Allreduce abstraction. The goal of rabit is to support **portable** , **scalable** and **reliable** distributed machine learning programs.

CHAPTER 1

API Documents

1.1 Python API of Rabit

This page contains document of python API of rabit. Reliable Allreduce and Broadcast Library.

Author: Tianqi Chen

`rabit.`**`allreduce`**(*data*, *op*, *prepare_fun=None*)

Perform allreduce, return the result.

Parameters

- **`data`** (*`numpy array`*) – Input data.
- **`op`** (*`int`*) – Reduction operators, can be MIN, MAX, SUM, BITOR
- **`prepare_fun`** (*`function`*) – Lazy preprocessing function, if it is not None, prepare_fun(data) will be called by the function before performing allreduce, to intialize the data If the result of Allreduce can be recovered directly, then prepare_fun will NOT be called

Returns **result** – The result of allreduce, have same shape as data

Return type array_like

Notes

This function is not thread-safe.

`rabit.`**`broadcast`**(*data*, *root*)

Broadcast object from one node to all other nodes.

Parameters

- **`data`** (*`any type that can be pickled`*) – Input data, if current rank does not equal root, this can be None
- **`root`** (*`int`*) – Rank of the node to broadcast data from.

Returns **object** – the result of broadcast.

Return type int

`rabit.`**`checkpoint`**(*global_model*, *local_model=None*)

Checkpoint the model.

This means we finished a stage of execution. Every time we call check point, there is a version number which will increase by one.

Parameters

- **`global_model`** (*`anytype that can be pickled`*) – globally shared model/state when calling this function, the caller need to gauranttees that global_model is the same in all nodes
- **`local_model`** (*`anytype that can be pickled`*) – Local model, that is specific to current node/rank. This can be None when no local state is needed.

Notes

local_model requires explicit replication of the model for fault-tolerance. This will bring replication cost in checkpoint function. while global_model do not need explicit replication. It is recommended to use global_model if possible.

`rabit.`**`finalize`**`()`

Finalize the rabit engine.

Call this function after you finished all jobs.

`rabit.`**`get_processor_name`**`()`

Get the processor name.

Returns **name** – the name of processor(host)

Return type str

`rabit.`**`get_rank`**`()`

Get rank of current process.

Returns **rank** – Rank of current process.

Return type int

`rabit.`**`get_world_size`**`()`

Get total number workers.

Returns **n** – Total number of process.

Return type int

`rabit.`**`init`**`(`*args=None*, *lib='standard'*`)`

Intialize the rabit module, call this once before using anything.

Parameters

- **`args`** (*`list of str, optional`*) – The list of arguments used to initialized the rabit usually you need to pass in sys.argv. Defaults to sys.argv when it is None.
- **`lib`** (*`{'standard', 'mock', 'mpi'}`*) – Type of library we want to load

`rabit.`**`load_checkpoint`**`(`*with_local=False*`)`

Load latest check point.

Parameters **`with_local`** (*`bool, optional`*) – whether the checkpoint contains local model

Returns **tuple** – if with_local: return (version, gobal_model, local_model) else return (version, gobal_model) if returned version == 0, this means no model has been CheckPointed and global_model, local_model returned will be None

Return type tuple

`rabit.`**`tracker_print`**`(`*msg*`)`

Print message to the tracker.

This function can be used to communicate the information of the progress to the tracker

Parameters **`msg`** (*`str`*) – The message to be printed to tracker.

`rabit.`**`version_number`**`()`

Returns version number of current stored model.

This means how many calls to CheckPoint we made so far.

Returns **version** – Version number of currently stored model

Return type int

1.2 C++ Library API of Rabit

This page contains document of Library API of rabit.

namespace **`rabit`**

rabit namespace

Typedefs

typedef dmlc::Stream **`Stream`**

defines stream used in rabit see definition of Stream in dmlc/io.h

typedef dmlc::Serializable **`Serializable`**

defines serializable objects used in rabit see definition of Serializable in dmlc/io.h

Functions

void **`Init`**(int *argc*, char **argv*[])

initializes rabit, call this once at the beginning of your program

Parameters

- `argc` - number of arguments in argv
- `argv` - the array of input arguments

void **`Finalize`**(void)

finalizes the rabit engine, call this function after you finished with all the jobs

int **`GetRank`**(void)

gets rank of the current process

int **`GetWorldSize`**(void)

gets total number of processes

bool **`IsDistributed`**(void)

whether rabit env is in distributed mode

std::string **`GetProcessorName`**(void)

gets processor's name

void **`TrackerPrint`** (**const** std::string &*msg*)
prints the msg to the tracker, this function can be used to communicate progress information to the user who monitors the tracker

Parameters

- `msg` - the message to be printed

void **`TrackerPrintf`** (**const** char **fmt*, ...)
prints the msg to the tracker, this function may not be available in very strict c++98 compilers, though it usually is. this function can be used to communicate progress information to the user who monitors the tracker

Parameters

- `fmt` - the format string

void **`Broadcast`** (void **sendrecv_data*, size_t *size*, int *root*)
broadcasts a memory region to every node from the root

Example: int a = 1; Broadcast(&a, sizeof(a), root);

Parameters

- `sendrecv_data` - the pointer to the send/receive buffer,
- `size` - the data size
- `root` - the process root

template <**typename** DType>
void **`Broadcast`** (std::vector<DType> **sendrecv_data*, int *root*)
broadcasts an std::vector<DType> to every node from root

Parameters

- `sendrecv_data` - the pointer to send/receive vector, for the receiver, the vector does not need to be pre-allocated
- `root` - the process root

Template Parameters

- `DType` - the data type stored in the vector, has to be a simple data type that can be directly transmitted by sending the sizeof(DType)

void **`Broadcast`** (std::string **sendrecv_data*, int *root*)
broadcasts a std::string to every node from the root

Parameters

- `sendrecv_data` - the pointer to the send/receive buffer, for the receiver, the vector does not need to be pre-allocated
- `root` - the process root

template <**typename** OP, **typename** DType>
void **`Allreduce`** (DType **sendrecvbuf*, size_t *count*, void (**prepare_fun*)) void *
, void **prepare_arg* performs in-place Allreduce on sendrecvbuf this function is NOT thread-safe

Example Usage: the following code does an Allreduce and outputs the sum as the result

```
vector<int> data(10);
...
Allreduce<op::Sum>(&data[0], data.size());
...
```

Parameters

- `sendrecvbuf` - buffer for both sending and receiving data
- `count` - number of elements to be reduced
- `prepare_fun` - Lazy preprocessing function, if it is not NULL, prepare_fun(prepare_arg) will be called by the function before performing Allreduce in order to initialize the data in sendrecvbuf. If the result of Allreduce can be recovered directly, then prepare_func will NOT be called
- `prepare_arg` - argument used to pass into the lazy preprocessing function

Template Parameters

- `OP` - see namespace op, reduce operator
- `DType` - data type

template <typename OP, **typename** DType>
void **`Allreduce`**(DType **sendrecvbuf*, size_t *count*, std::function<void)
> *prepare_fun* performs in-place Allreduce, on sendrecvbuf with a prepare function specified by a lambda function

Example Usage:
```
// the following code does an Allreduce and outputs the sum as the result
vector<int> data(10);
...
Allreduce<op::Sum>(&data[0], data.size(), [&]() {
                    for (int i = 0; i < 10; ++i) {
                      data[i] = i;
                    }
                  });
    ...
```

Parameters

- `sendrecvbuf` - buffer for both sending and receiving data
- `count` - number of elements to be reduced
- `prepare_fun` - Lazy lambda preprocessing function, prepare_fun() will be invoked by the function before performing Allreduce in order to initialize the data in sendrecvbuf. If the result of Allreduce can be recovered directly, then prepare_func will NOT be called

Template Parameters

- `OP` - see namespace op, reduce operator
- `DType` - data type

int **`LoadCheckPoint`**(*Serializable* **global_model*, *Serializable* **local_model*)
loads the latest check point

```
// Example usage code of LoadCheckPoint
int iter = rabit::LoadCheckPoint(&model);
if (iter == 0) model.InitParameters();
```

```
for (i = iter; i < max_iter; ++i) {
  // do many things, include allreduce
  rabit::CheckPoint(model);
}
```

Return

the version number of the check point loaded if returned version == 0, this means no model has been CheckPointed the p_model is not touched, users should do the necessary initialization by themselves

See

CheckPoint, *VersionNumber*

Parameters

- `global_model` - pointer to the globally shared model/state when calling this function, the caller needs to guarantee that the global_model is the same in every node
- `local_model` - pointer to the local model that is specific to the current node/rank this can be NULL when no local model is needed

void **`CheckPoint`** (**const** *Serializable* **global_model*, **const** *Serializable* **local_model*)

checkpoints the model, meaning a stage of execution has finished. every time we call check point, a version number will be increased by one

See

LoadCheckPoint, *VersionNumber*

Parameters

- `global_model` - pointer to the globally shared model/state when calling this function, the caller needs to guarantee that the global_model is the same in every node
- `local_model` - pointer to the local model that is specific to the current node/rank this can be NULL when no local state is needed NOTE: local_model requires explicit replication of the model for fault-tolerance, which will bring replication cost in the CheckPoint function. global_model does not need explicit replication. So, only CheckPoint with the global_model if possible

void **`LazyCheckPoint`** (**const** *Serializable* **global_model*)

This function can be used to replace CheckPoint for global_model only, when certain condition is met (see detailed explanation).

This is a “lazy” checkpoint such that only the pointer to the global_model is remembered and no memory copy is taken. To use this function, the user MUST ensure that: The global_model must remain unchanged until the last call of Allreduce/Broadcast in the current version finishes. In other words, the global_model model can be changed only between the last call of Allreduce/Broadcast and LazyCheckPoint, both in the same version

For example, suppose the calling sequence is: LazyCheckPoint, code1, Allreduce, code2, Broadcast, code3, LazyCheckPoint/(or can be CheckPoint)

Then the user MUST only change the global_model in code3.

The use of LazyCheckPoint instead of CheckPoint will improve the efficiency of the program.

See

LoadCheckPoint, *CheckPoint*, *VersionNumber*

Parameters

- `global_model` - pointer to the globally shared model/state when calling this function, the caller needs to guarantee that the global_model is the same in every node

int **`VersionNumber`**(void) **Return**

version number of the current stored model, which means how many calls to CheckPoint we made so far

See

LoadCheckPoint, *CheckPoint*

template <typename DType, void(*)(DType &dst, const DType &src) *freduce*>
class `Reducer`

#include <rabit.h> template class to make customized reduce and all reduce easy Do not use reducer directly in the function you call Finalize, because the destructor can execute after Finalize

Template Parameters

- `DType` - data type that to be reduced
- `freduce` - the customized reduction function DType must be a struct, with no pointer

template <typename DType>
class `SerializeReducer`

#include <rabit.h> template class to make customized reduce, this class defines complex reducer handles all the data structure that can be serialized/deserialized into fixed size buffer Do not use reducer directly in the function you call Finalize, because the destructor can execute after Finalize

Template Parameters

- `DType` - data type that to be reduced, DType must contain the following functions:
- `freduce` - the customized reduction function (1) Save(IStream &fs) (2) Load(IStream &fs) (3) Reduce(const DType &src, size_t max_nbyte)

namespace `op`

reduction operators namespace

class `Max`

#include <rabit.h> maximum reduction operator

class `Min`

#include <rabit.h> minimum reduction operator

class `Sum`

#include <rabit.h> sum reduction operator

class `BitOR`

#include <rabit.h> bitwise OR reduction operator

1.3 Parameters

This section list all the parameters that can be passed to rabit::Init function as argv. All the parameters are passed in as string in format of `parameter-name=parameter-value`. In most setting these parameters have default value or will be automatically detected, and do not need to be manually configured.

- rabit_tracker_uri [passed in automatically by tracker]

  - The uri/ip of rabit tracker
- rabit_tracker_port [passed in automatically by tracker]
  - The port of rabit tracker
- rabit_task_id [automatically detected]
  - The unique identifier of computing process
  - When running on hadoop, this is automatically extracted from enviroment variable
- rabit_reduce_buffer [default = 256MB]
  - The memory buffer used to store intermediate result of reduction
  - Format "digits + unit", can be 128M, 1G
- rabit_global_replica [default = 5]
  - Number of replication copies of result kept for each Allreduce/Broadcast call
- rabit_local_replica [default = 2]
  - Number of replication of local model in check point

1.4 Tutorial

This is rabit's tutorial, a ***Reliable Allreduce and Broadcast Interface***. All the example codes are in the guide folder of the project. To run the examples locally, you will need to build them with `make`.

List of Topics

1.4.1 What is Allreduce

The main methods provided by rabit are Allreduce and Broadcast. Allreduce performs reduction across different computation nodes, and returns the result to every node. To understand the behavior of the function, consider the following example in basic.cc (there is a python example right after this if you are more familiar with python).

```
#include <rabit.h>
using namespace rabit;
const int N = 3;
int main(int argc, char *argv[]) {
  int a[N];
  rabit::Init(argc, argv);
  for (int i = 0; i < N; ++i) {
    a[i] = rabit::GetRank() + i;
  }
  printf("@node[%d] before-allreduce: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  // allreduce take max of each elements in all processes
  Allreduce<op::Max>(&a[0], N);
  printf("@node[%d] after-allreduce-max: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  // second allreduce that sums everything up
  Allreduce<op::Sum>(&a[0], N);
  printf("@node[%d] after-allreduce-sum: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  rabit::Finalize();
  return 0;
}
```

You can run the example using the rabit_demo.py script. The following command starts the rabit program with two worker processes.

```
../tracker/rabit_demo.py -n 2 basic.rabit
```

This will start two processes, one process with rank 0 and the other with rank 1, both processes run the same code. The `rabit::GetRank()` function returns the rank of current process.

Before the call to Allreduce, process 0 contains the array `a = {0, 1, 2}`, while process 1 has the array `a = {1, 2, 3}`. After the call to Allreduce, the array contents in all processes are replaced by the reduction result (in this case, the maximum value in each position across all the processes). So, after the Allreduce call, the result will become `a = {1, 2, 3}`. Rabit provides different reduction operators, for example, if you change `op::Max` to `op::Sum`, the reduction operation will be a summation, and the result will become `a = {1, 3, 5}`. You can also run the example with different processes by setting -n to different values.

If you are more familiar with python, you can also use rabit in python. The same example as before can be found in basic.py:

```python
import numpy as np
import rabit

rabit.init()
n = 3
rank = rabit.get_rank()
a = np.zeros(n)
for i in xrange(n):
    a[i] = rank + i

print '@node[%d] before-allreduce: a=%s' % (rank, str(a))
a = rabit.allreduce(a, rabit.MAX)
print '@node[%d] after-allreduce-max: a=%s' % (rank, str(a))
a = rabit.allreduce(a, rabit.SUM)
print '@node[%d] after-allreduce-sum: a=%s' % (rank, str(a))
rabit.finalize()
```

You can run the program using the following command

```
../tracker/rabit_demo.py -n 2 basic.py
```

Broadcast is another method provided by rabit besides Allreduce. This function allows one node to broadcast its local data to all other nodes. The following code in broadcast.cc broadcasts a string from node 0 to all other nodes.

```
#include <rabit.h>
using namespace rabit;
const int N = 3;
int main(int argc, char *argv[]) {
  rabit::Init(argc, argv);
  std::string s;
  if (rabit::GetRank() == 0) s = "hello world";
  printf("@node[%d] before-broadcast: s=\"%s\"\n",
         rabit::GetRank(), s.c_str());
  // broadcast s from node 0 to all other nodes
  rabit::Broadcast(&s, 0);
  printf("@node[%d] after-broadcast: s=\"%s\"\n",
         rabit::GetRank(), s.c_str());
  rabit::Finalize();
  return 0;
}
```

The following command starts the program with three worker processes.

```
../tracker/rabit_demo.py -n 3 broadcast.rabit
```

Besides strings, rabit also allows to broadcast constant size array and vectors.

The counterpart in python can be found in broadcast.py. Here is a snippet so that you can get a better sense of how simple is to use the python library:

```
import rabit
rabit.init()
n = 3
rank = rabit.get_rank()
s = None
if rank == 0:
    s = {'hello world':100, 2:3}
print '@node[%d] before-broadcast: s=\"%s\"' % (rank, str(s))
s = rabit.broadcast(s, 0)
print '@node[%d] after-broadcast: s=\"%s\"' % (rank, str(s))
rabit.finalize()
```

1.4.2 Common Use Case

Many distributed machine learning algorithms involve splitting the data into different nodes, computing statistics locally, and finally aggregating them. Such workflow is usually done repetitively through many iterations before the algorithm converges. Allreduce naturally meets the structure of such programs, common use cases include:

- Aggregation of gradient values, which can be used in optimization methods such as L-BFGS.
- Aggregation of other statistics, which can be used in KMeans and Gaussian Mixture Models.
- Find the best split candidate and aggregation of split statistics, used for tree based models.

Rabit is a reliable and portable library for distributed machine learning programs, that allow programs to run reliably on different platforms.

1.4.3 Use Rabit API

This section introduces topics about how to use rabit API. You can always refer to API Documentation for definition of each functions. This section trys to gives examples of different aspectes of rabit API.

Structure of a Rabit Program

The following code illustrates the common structure of a rabit program. This is an abstract example, you can also refer to wormhole for an example implementation of kmeans algorithm.

```
#include <rabit.h>
int main(int argc, char *argv[]) {
  ...
  rabit::Init(argc, argv);
  // load the latest checked model
  int version = rabit::LoadCheckPoint(&model);
  // initialize the model if it is the first version
  if (version == 0) model.InitModel();
  // the version number marks the iteration to resume
  for (int iter = version; iter < max_iter; ++iter) {
    // at this point, the model object should allow us to recover the program state
    ...
    // each iteration can contain multiple calls of allreduce/broadcast
    rabit::Allreduce<rabit::op::Max>(&data[0], n);
    ...
    // checkpoint model after one iteration finishes
    rabit::CheckPoint(&model);
  }
  rabit::Finalize();
  return 0;
}
```

Besides the common Allreduce and Broadcast functions, there are two additional functions: `LoadCheckPoint` and `CheckPoint`. These two functions are used for fault-tolerance purposes. As mentioned before, traditional machine learning programs involve several iterations. In each iteration, we start with a model, make some calls to Allreduce or Broadcast and update the model. The calling sequence in each iteration does not need to be the same.

- When the nodes start from the beginning (i.e. iteration 0), `LoadCheckPoint` returns 0, so we can initialize the model.
- `CheckPoint` saves the model after each iteration.
  - Efficiency Note: the model is only kept in local memory and no save to disk is performed when calling Checkpoint
- When a node goes down and restarts, `LoadCheckPoint` will recover the latest saved model, and
- When a node goes down, the rest of the nodes will block in the call of Allreduce/Broadcast and wait for the recovery of the failed node until it catches up.

Please see the *Fault Tolerance* section to understand the recovery procedure executed by rabit.

Allreduce and Lazy Preparation

Allreduce is one of the most important function provided by rabit. You can call allreduce by specifying the reduction operator, pointer to the data and size of the buffer, as follows

```
Allreduce<operator>(pointer_of_data, size_of_data);
```

This is the basic use case of Allreduce function. It is common that user writes the code to prepare the data needed into the data buffer, pass the data to Allreduce function, and get the reduced result. However, when a node restarts from failure, we can directly recover the result from other nodes(see also *Fault Tolerance*) and the data preparation procedure no longer necessary. Rabit Allreduce add an optional parameter preparation function to support such scenario. User can pass in a function that corresponds to the data preparation procedure to Allreduce calls, and the data preparation function will only be called when necessary. We use lazy_allreduce.cc as an example to demonstrate this feature. It is modified from basic.cc, and you can compare the two codes.

```
#include <rabit.h>
using namespace rabit;
const int N = 3;
int main(int argc, char *argv[]) {
  int a[N] = {0};
  rabit::Init(argc, argv);
  // lazy preparation function
  auto prepare = [&]() {
    printf("@node[%d] run prepare function\n", rabit::GetRank());
    for (int i = 0; i < N; ++i) {
      a[i] = rabit::GetRank() + i;
    }
  };
  printf("@node[%d] before-allreduce: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  // allreduce take max of each elements in all processes
  Allreduce<op::Max>(&a[0], N, prepare);
  printf("@node[%d] after-allreduce-sum: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  // rum second allreduce
  Allreduce<op::Sum>(&a[0], N);
  printf("@node[%d] after-allreduce-max: a={%d, %d, %d}\n",
         rabit::GetRank(), a[0], a[1], a[2]);
  rabit::Finalize();
  return 0;
}
```

Here we use features of C++11 because the lambda function makes things much shorter. There is also C++ compatible callback interface provided in the API. You can compile the program by typing `make lazy_allreduce.mock`. We link against the mock library so that we can see the effect when a process goes down. You can run the program using the following command

```
../tracker/rabit_demo.py -n 2 lazy_allreduce.mock mock=0,0,1,0
```

The additional arguments `mock=0,0,1,0` will cause node 0 to kill itself before second call of Allreduce (see also *mock test*). You will find that the prepare function's print is only executed once and node 0 will no longer execute the preparation function when it restarts from failure.

You can also find python version of the example in lazy_allreduce.py, and run it using the followin command

```
../tracker/rabit_demo.py -n 2 lazy_allreduce.py mock=0,0,1,0
```

Since lazy preparation function may not be called during execution. User should be careful when using this feature. For example, a possible mistake could be putting some memory allocation code in the lazy preparation function, and the computing memory was not allocated when lazy preparation function is not called. The example in lazy_allreduce.cc provides a simple way to migrate normal prepration code(basic.cc) to lazy version: wrap the preparation code with a lambda function, and pass it to allreduce.

Checkpoint and LazyCheckpoint

Common machine learning algorithms usually involves iterative computation. As mentioned in the section (*Structure of a Rabit Program*), user can and should use Checkpoint to `save` the progress so far, so that when a node fails, the latest checkpointed model can be loaded.

There are two model arguments you can pass to Checkpoint and LoadCheckpoint: `global_model` and `local_model`:

- `global_model` refers to the model that is commonly shared across all the nodes
  - For example, the centriods of clusters in kmeans is shared across all nodes
- `local_model` refers to the model that is specifically tied to the current node
  - For example, in topic modeling, the topic assignments of subset of documents in current node is local model

Because the different nature of the two types of models, different strategy will be used for them. `global_model` is simply saved in local memory of each node, while `local_model` will replicated to some other nodes (selected using a ring replication strategy). The checkpoint is only saved in the memory without touching the disk which makes rabit programs more efficient. User is encouraged to use `global_model` only when is sufficient for better efficiency.

To enable a model class to be checked pointed, user can implement a serialization interface. The serialization interface already provide serialization functions of STL vector and string. For python API, user can checkpoint any python object that can be pickled.

There is a special Checkpoint function called LazyCheckpoint, which can be used for `global_model` only cases under certain condition. When LazyCheckpoint is called, no action is taken and the rabit engine only remembers the pointer to the model. The serialization will only happen when another node fails and the recovery starts. So user basically pays no extra cost calling LazyCheckpoint. To use this function, the user need to ensure the model remain unchanged until the last call of Allreduce/Broadcast in the current version finishes. So that when recovery procedure happens in these function calls, the serialized model will be the same.

For example, consider the following calling sequence

```
LazyCheckPoint, code1, Allreduce, code2, Broadcast, code3, LazyCheckPoint
```

The user must only change the model in code3. Such condition can usually be satiesfied in many scenarios, and user can use LazyCheckpoint to further improve the efficiency of the program.

1.4.4 Compile Programs with Rabit

Rabit is a portable library, to use it, you only need to include the rabit header file.

- You will need to add the path to ../include to the header search path of the compiler
  - Solution 1: add `-I/path/to/rabit/include` to the compiler flag in gcc or clang
  - Solution 2: add the path to the environment variable CPLUS_INCLUDE_PATH
- You will need to add the path to ../lib to the library search path of the compiler
  - Solution 1: add `-L/path/to/rabit/lib` to the linker flag
  - Solution 2: add the path to environment variable LIBRARY_PATH AND LD_LIBRARY_PATH
- Link against lib/rabit.a
  - Add `-lrabit` to the linker flag

The procedure above allows you to compile a program with rabit. The following two sections contain additional options you can use to link against different backends other than the normal one.

Link against MPI Allreduce

You can link against `rabit_mpi.a` instead of using MPI Allreduce, however, the resulting program is backed by MPI and is not fault tolerant anymore.

- Simply change the linker flag from `-lrabit` to `-lrabit_mpi`
- The final linking needs to be done by mpi wrapper compiler `mpicxx`

Link against Mock Test Rabit Library

If you want to use a mock to test the program in order to see the behavior of the code when some nodes go down, you can link against `rabit_mock.a` .

- Simply change the linker flag from `-lrabit` to `-lrabit_mock`

The resulting rabit mock program can take in additional arguments in the following format

```
mock=rank,version,seq,ndeath
```

The four integers specify an event that will cause the program to `commit suicide`(exit with -2)

- rank specifies the rank of the node to kill
- version specifies the version (iteration) of the model where you want the process to die
- seq specifies the sequence number of the Allreduce/Broadcast call since last checkpoint, where the process will be killed
- ndeath specifies how many times this node died already

For example, consider the following script in the test case

```
../tracker/rabit_demo.py -n 10 test_model_recover 10000\
                         mock=0,0,1,0 mock=1,1,1,0 mock=1,1,1,1
```

- The first mock will cause node 0 to exit when calling the second Allreduce/Broadcast (seq = 1) in iteration 0
- The second mock will cause node 1 to exit when calling the second Allreduce/Broadcast (seq = 1) in iteration 1
- The third mock will cause node 1 to exit again when calling second Allreduce/Broadcast (seq = 1) in iteration 1
  - Note that ndeath = 1 means this will happen only if node 1 died once, which is our case

1.4.5 Running Rabit Jobs

Rabit is a portable library that can run on multiple platforms. All the rabit jobs can be submitted using dmlc-tracker

1.4.6 Fault Tolerance

This section introduces how fault tolerance works in rabit. The following figure shows how rabit deals with failures.

The scenario is as follows:

- Node 1 fails between the first and second call of Allreduce after the second checkpoint
- The other nodes wait in the call of the second Allreduce in order to help node 1 to recover.
- When node 1 restarts, it will call `LoadCheckPoint`, and get the latest checkpoint from one of the existing nodes.
- Then node 1 can start from the latest checkpoint and continue running.
- When node 1 calls the first Allreduce again, as the other nodes already know the result, node 1 can get it from one of them.

- When node 1 reaches the second Allreduce, the other nodes find out that node 1 has catched up and they can continue the program normally.

This fault tolerance model is based on a key property of Allreduce and Broadcast: All the nodes get the same result after calling Allreduce/Broadcast. Because of this property, any node can record the results of history Allreduce/Broadcast calls. When a node is recovered, it can fetch the lost results from some alive nodes and rebuild its model.

The checkpoint is introduced so that we can discard the history results of Allreduce/Broadcast calls before the latest checkpoint. This saves memory consumption used for backup. The checkpoint of each node is a model defined by users and can be split into 2 parts: a global model and a local model. The global model is shared by all nodes and can be backed up by any nodes. The local model of a node is replicated to some other nodes (selected using a ring replication strategy). The checkpoint is only saved in the memory without touching the disk which makes rabit programs more efficient. The strategy of rabit is different from the fail-restart strategy where all the nodes restart from the same checkpoint when any of them fail. In rabit, all the alive nodes will block in the Allreduce call and help the recovery. To catch up, the recovered node fetches its latest checkpoint and the results of Allreduce/Broadcast calls after the checkpoint from some alive nodes.

This is just a conceptual introduction to rabit's fault tolerance model. The actual implementation is more sophisticated, and can deal with more complicated cases such as multiple nodes failure and node failure during recovery phase.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search