Distributed Machine Learning Toolkit (DMTK)

Microsoft Research Asia
2015.11
Overview

- Introduction to DMTK
- Using Binary Tools in DMTK
- Developing Your Own Algorithms on DMTK
- Resource and Support
DMTK

• DMTK (current release) includes
  • Distributed machine learning framework (code name: Multiverso)
    • Parameter server + client SDK for developing distributed machine learning algorithms
  • Two distributed algorithms implemented on Multiverso
    • LightLDA
      • Distributed trainer for LightLDA[^1] topic model
    • Distribute Word Embedding (DWE)
      • Distributed trainer for word2vec[^2] (abbr. as DWE) and multi-sense word embedding[^3] (abbr. as DMWE).

Multiverso Framework

Multiverso Server

Hybrid model store

- Aggregation of model updates
- Send model to clients

Server processing threads

- Client updates
- Client requests
- Server responses

Multiverso Client

Parameter loading thread

- Pre-fetch parameter for next data block
- Automatic pipelining

Training threads

- Model Scheduling
- Aggregation of model updates
- Send model to clients

Intermediate Data store

- Training data block

Update cache

- Gradients $\Delta_i, \Delta_j, \ldots$

Local Model Store

- Update cache

Communicator Thread

- Update Parameter
- Get Parameter

Message queue
Software and Hardware Requirements

• Running platform - x64
  • Tested on Windows server 2012, and Ubuntu 12.04

• Developing environment and compilers
  • The codes were developed with Visual Studio 2013 on Windows; slight code conversion might be needed when using other versions of Visual Studio.
  • The codes were compiled with g++ 4.8 on Linux.

• Welcome to help us migrate DMTK to other environments
Using Binary Tools

Running LightLDA and Distributed Word Embedding (DWE/DMWE)
Prerequisites

• Install MPI/ZMQ
  • [https://github.com/Microsoft/multiverso/tree/master/windows](https://github.com/Microsoft/multiverso/tree/master/windows)

• Firewall exception
  • If your cluster has firewall enabled, please make sure you have firewall exception for the following network applications
    • All binaries for MPI, i.e., executables in the “bin” folder under the MPI path
    • The commands you are going to run on MPI/ZMQ, e.g., LightLDA.exe
    • Server applications (when using ZMQ as inter-process communication library)

• Make third-party dlls ready for applications
  • Mpi.dll, ZMQ-related dlls.
  • Make sure they are in the system path or in the application path
Running Multiverso

• Multiverso supports two inter-process communication modes
  • MPI:
    • The parameter server routines are bound to the worker processes running as server threads. MPI will help start all the applications.
    • Command line to start a MPI job: `mpiexec.exe –machinefile machineFile app.exe – yourAppArgs` ...
  • ZMQ:
    • The parameter server is a standalone process. You need to start servers and your client applications separately. For client applications, a server endpoint file is needed, which contains the `ip:port` information of each server line by line.
    • Go to every server and start the following commands:
      • To start server: `multiverso_server.exe server_id worker_number`
      • To start a particular client task: `app.exe –server_endpoint_file server_file -yourAppArgs` ...
More about MPI Jobs

• Follow the Prerequisites page to install MPI, and add firewall exceptions

• Before running the MPI job, please make sure you have started “SMPD.exe –p port_id –d” on each machine. The port_id is the port used to communicate among MPI nodes.

• Compose the machine.txt file to specify the machine names (or IPs) and node (or process) number on each machine.

• Run the following command to kickoff the distributed training on any machine within the cluster.

```
mpiexec -p port_id -machinefile machine.txt path\App.exe -arguments
```

• Remarks
  • Please make sure every machine has the working directory on their local disk.
  • Please make sure port_id is consistent with the value you start smpd.

In this example, machine1 and machine2 are used: 2 nodes (processes) on machine1, and 4 nodes (processes) on machine2.
Running LightLDA

• LightLDA only accepts binary input data; we have provided a tool to convert libsvm format text file to the desirable binary file.

• For more details, please refer to
  • [https://github.com/Microsoft/lightlda/tree/master/example](https://github.com/Microsoft/lightlda/tree/master/example)
Running DWE/DMWE

• DWE/DMWE only accepts raw text file as input. In addition, **Vocabulary file, stop words file** are also required. We have provided a tool to generate vocabulary file based on the raw text.

• For more details, please refer to
  • [https://github.com/Microsoft/distributed_word_embedding/examples](https://github.com/Microsoft/distributed_word_embedding/examples)
Developing Your Own Algorithm

Build more new distributed machine learning algorithms based on Multiverso
Setup Development Environment

• Download
  
  `git clone https://github.com/Microsoft/multiverso`

• Build
  
  • On Linux
    • Run `./third_party/install.sh`
    • Run `make all -j4`
  
  • On Windows
    • Install third-party libraries
      `https://github.com/Microsoft/multiverso/tree/master/windows`
    • Open `windows/multiverso.sln`, change configuration and platform as Release and x64, then build the solution.
Key Steps

• Define the main training routine
  1) Start the Multiverso environment
  2) Configure the Multiverso parameter server
  3) Define the overall training process based on user defined data block

• Implement you algorithm logic
  1) Define your data block
  2) Define parameter loader
  3) Define training logic of each data block
Start the Multiverso Environment

- Initialize the Multiverso environment.
  - pass the configuration to the environment by setting a `Config` object.
  - call the `Init` function to start Multiverso.
    - Parameters of `Init` function include `trainer` and `parameter loader`, which define the detailed training algorithm.
  - call `Close` to close Multiverso.

- Remark
  - More details could be found on our website
  - We also have comprehensive comments in the source codes to aid your programming
Multiverso stores parameters as Table. Before training, you need to configure and initialize the server tables by adding your logic between these two methods: BeginConfig, EndConfig.

Configure tables
- Call AddTable to create tables in both local cache and parameter server
- Call SetRow method to set row property, like size, format, type, etc.

Initialize tables
- Call AddToServer, if you want to initialize the server tables with non-trivial values.
Define Training Process

• The training logic should be placed between these two methods: `BeginTrain` and `EndTrain`. Then Multiverso will schedule the training process based on data blocks.

• In distributed training, the concept of clock is needed, which refers to a local training period. For each clock we do a sync-up with parameter server. To define a clock, you need to call `BeginClock` and `EndClock` to tell Multiverso that the data fed are within the clock.

• In a clock period, you may feed one or more data blocks into Multiverso by calling `PushDataBlock`. 

```cpp
Multiverso::Init(trainers, loader, config, argc, argv);

// Step 2: Config the table in Parameter Server
Multiverso::BeginConfig();

const int kTableId = 0, kNumRows = 1, kNumCols = 10;
// Create Table in Parameter Server
Multiverso::AddTable(kTableId, kNumRows, kNumCols, Type::Int, Format::Dense);
for (int k = 0; k < kNumCols; ++k)
    // Init the value in Parameter Server
    Multiverso::AddToServer<int>(kTableId, kNumRows, kNumCols, 1);

// Finish Configuration
Multiverso::EndConfig();

// Step 3: Train
const int kNumIteration = 100;
Multiverso::BeginTrain();

for (int i = 0; i < kNumIteration; ++i)
    { 
        Multiverso::BeginClock();

        // DataBlock will be defined later
        DataBlock* data = GetDataBlock();
        Multiverso::PushDataBlock(data);

        Multiverso::EndClock();
    }

Multiverso::EndTrain();

// End of Multiverso: Close the Multiverso Environment
Multiverso::Close();
return 0;
```
Define Your Data Block

- Multiverso schedules the training process based on data blocks. To define your data block, you just need to inherit the `DataBlockBase` class and implement your own data block.

- In `DataBlockBase`, you might need to define the functions for
  - composing data block from input data stream (in the form of file or memory buffer).
  - getting training examples from data block.

```cpp
class DataBlock : public DataBlockBase
{
  // Defines what's your training data
};

class Trainer : public TrainerBase
{
  public:

    // Defines your training logic for a data block 'data'
    // When calling this function, you can assume that all data
    // and parameter needed is local. Parameters have been prefetched
    // from parameter server by multiverso. Then you can write your
    // training logic as same as writing a single machine program.
    void trainIteration(DataBlock* data) override
    {
      DataBlock* data_block = reinterpret_cast<DataBlock*>(data);

      // The API GetRow<T> and Add<T> are member function inherited
      // from the TrainerBase

      // Access model, it is local memory access, no network happened here
      Row<int>& row = GetRow<int>(kTableId, kRowId);
      std::vector<int> updates;

      // Your training logic to produce updates
      // TODO
      // ...
      Add<int>(kTableId, kRowId, updates);
    }
};

class ParameterLoader : public ParameterLoaderBase
{
};
```
You need to implement parameter preparation and training for each data block separately, so that Multiverso can pipeline the process to enhance system throughput.

**ParameterLoader** is used to prepare parameters needed by training. To implement your loader, you need to inherit the **ParameterLoaderBase** class and override the **ParseAndRequest** method. In the method, you need to:

- Implement logics to parse the data block and identify a set of parameters needed during the upcoming training process.
- Use **RequestTable**, **RequestRow**, or **RequestElement** to pull these parameters from parameter servers.
Define the Training Logic

- Multiverso will create training threads based on user-defined data blocks.
- These training threads will call Trainer to perform the training. To implement your trainer, please inherit the TrainerBase class and override the TrainIteration method.
- In the method, you can
  - implement your training logic to process the training samples in the data block
  - Use the GetRow method to access parameters in local cache
  - Use the Add method to update the parameters.
Resources

• DMTK Website
  https://www.dmtk.io

• DMTK Source Codes
  https://github.com/Microsoft/DMTK

• DMTK Documents
  http://www.dmtk.io/document.html

• Multiverso API Documents
  http://www.dmtk.io/multiverso/annotated.html
Support

• Please send email to dmtk@microsoft.com for technical support or bug reporting.

• We will continue to enrich the tutorial to aid your development on top of DMTK; please check our website and GitHub site in a regular basis for new information.