Petuum Parameter Server Reference Manual

Jinliang Wei

Carnegie Mellon University, School of Computer Science

Revision 0.1
Last Update: January 5, 2015

Parameter Server Essentials
A Brief Introduction

1 What is Parameter Server

The Parameter Server is a key-value store that allows different processes to share access to a set of variables. In the context of data-parallel ML applications, data is partitioned across multiple machines which are connected via a common network, the model or sufficient statistics of the model is typically shared by the learning processes.

Since sharing parameters over network involves high overhead of network communication, it is encoraged for the application developers to construct their applications in a way that minimizes parameter sharing and only store the parameters that have to be shared across processes in the Parameter Server.

2 High-Level System Abstraction

The system consists of a **Server** and multiple **Workers**. The **Server** maintains the master copy of the parameters and propagates the **workers'** writes (updates) to other **workers**.

The **workers** access the parameters via a **client** library. The **client** library caches the previously accessed parameters to speedup future accesses. The **client** side cache is referred to as **process storage**.

Writes to the parameter are inserted into an **update table**, which is referred to as **OpLog** in the code.

3 Parameter Server Data Abstraction

In Parameter Server, the parameters (key-value pairs) stored are organized as **Table**s, and each table consists of multiple **Row**s. Each cell in a row is identified by a **Column ID**, and typically stores one parameter. In other words, parameters stored in the Parameter Server is identified by a tuple of **Row ID** and **Column ID**.

Table-Row also represents the underlying storage format. The Parameter Server allows the application to choose the most suitable data structure for organizing data stored in a row. We even allow user-defined **Row**s, which will be discussed in later sections.

Each **table** has its own **update table**. The **update table** also consists of rows, which are referred to as **row oplog**.

4 Creating Your First Parameter Server Application

In this section, we create a simple parameter server application that has only one single-threaded **client** and involves only one **Table**.

Step 0. Include the parameter server header files.

```
#include <petuum_ps_common/include/petuum_ps.hpp>
```

This is the only header file that needed for all Parameter Server APIs.

The first step is to initialize the parameter server environment. The thread that initializes the environment is referred to as the **init thread**.

In this example, we run only one worker process. In the case of multiple worker processes, the same initializaiton process should happen on all worker processes in cluding create all tables.

Step 1. Register row types.

The Parameter Server allows applications to use different **row types**. In order for the parameter server to create rows of the correct type, the rows have to be registered with the Parameter Server before the computation starts. Registration is done by the following API, which creates a mapping from **row ID** (a 32-bit integer) to the row type. Then the application may refer to the **row ID** for the corresponding type when creating tables.

In this example, we will a templatized vector row type provided by the Parameter Server . The template parameter is the element value type stored in the row. We register the row type `petuum::DenseRow<int>` (value type is `int`) with the Parameter Server as:

```
// register row type petuum::DenseRow<int> with ID 0.
petuum::PSTableGroup::RegisterRow<petuum::DenseRow<int> >(0);
```

Now we can refer to the row type with integer 0.

Step 2. Parameter Server Initialization.

For this simple application, the only configuration parameter needs to be set is the `host_map`. We need to add one entry for each worker process. Each entry has an ID (IDs are a contiguous range of integers, starting from 0); an IP address, such as "127.0.0.1"; and an open, unsed port number, such as "10000".

Other configuration parameters will become necessary for distributed applications and we will discuss them later.

The second parameter for `petuum::PSTableGroup::Init()` is a boolean flag indicating whether the init thread intends to access any table API – APIs defined in `petuum::Table` and `petuum::PSTableGroup::GetTableOrDie()`. Typically, this flag is set to `false`.

```
petuum::TableGroupConfig table_group_config;
table_group_config.host_map.insert(std::make_pair(0, HostInfo(0,
    "127.0.0.1", "10000")));
```

```
petuum::PSTableGroup::Init(table_group_config, false);
```

Step 3. Create Tables.

```
petuum::ClientTableConfig table_config;
table_config.table_info.row_type = 0;
table_config.table_info.row_capacity = 100;

table_config.process_cache_capacity = 1000;
table_config.oplog_capacity = 1000;

// here 0 is the table ID, which will be used later to get table.
bool suc = petuum::PSTableGroup::CreateTable(0, table_config);
```

The example above lists the minimal set of configuration parameters that an application program must set and we explain their meanings below.

Name	Default	Explanation
`table_info.row_type`	*N/A*	The row type to be used with this table.
`table_info.row_capacity`	0	For the dense row type, we must specify its capacity.
`process_cache_capacity`	0	The maximal number of rows in this table.
`process_cache_capacity`	0	The maximal number of rows that can be written to.

Once all tables are created, we need to finalize the table creation phase by calling

```
petuum::PSTableGroup::CreateTableDone();
```

Step 4. Create and Run Worker Threads

Create a worker thread that accesses the parameters via the Table interface. Note that in this example, you are allowed to use only one worker thread.

A worker thread that accesses table APIs is referred to as **table thread**.

Before a **table thread** can access the table APIs, it has to register itself with the Parameter Server via

```
int thread_id = petuum::PSTableGroup::RegisterThread();
```

Then it can get a table via

```
petuum::Table<int> table = petuum::PSTableGroup::GetTableOrDie(0);
```

The `petuum::Table` type provides a set of APIs for you to access your parameters. And we will explain them in later sections.

Once the worker thread is done with the computation, it must deregister itself before exiting:

```
petuum::PSTableGroup::DeregisterThread();
```

In case your init thread needs to access table interface, it has to set the boolean flag `table_access` in `petuum::PSTableGroup::Init()` to `true`, but it should not register nor deregister (it is registered automatically). However, it should wait for other threads to register via

```
petuum::PSTableGroup::WaitThreadRegister();
```

Step 5. Terminate the Parameter Server

After all worker threads have successfully exited, we can shut down the parameter server:

```
petuum::PSTableGroup::ShutDown();
```

5 Compilation

5.1 Compile the Parameter Server

In the project root directory,

```
make third_party_core
make ps_lib -j8
```

The parameter server library relies on a set of 3rd party softwares and they are automatically installed by `make third_party_core`.

5.2 Compile Your Application

We suggest you include `defns.mk` in your Makefile, so the petuum macros can become visible you.

Parameter Server Table Interface

Functions listed in this chapter compose the table interface. A thread that accesses the table interface is refered to a **table thread** and must register with the Parameter Server as explained earlier. The only exception is the **init thread**, which must not register but have to assert `true` for the table access flag when intializing the Parameter Server.

6 Gain Table Access

In order to read or update any parameter stored in the Parameter Server , the **table thread** firsts gets access to the corresponding table, then it can read or update parameters via the table interface.

```
// Gain access to table.
template<typename UPDATE>
petuum::Table<UPDATE> petuum::PSTableGroup::GetTableOrDie(int table_id);
```

7 Read Parameters

```
template<typename ROW>
const ROW &petuum::Table::Get(int32_t row_id, RowAccessor *row_accessor
    = 0);
```

This is the default read API and it works for both evictable and non-evictable client cache types. By default, the client-side cache is non-evictable and we will explain the difference later.

This function returns a read-only reference to the row stored in the client-side cache.

If the client-side cache is non-evictable, `row_accessor` may be disgarded. Even if a valid pointer is provided, it is simply ignored.

However, if the client-side cache is evictable, the `row_accessor` pointer must point to a valid `RowAccessor` object. And it will be set to refer to the row being read.

```
void petuum::Table::Get(int32_t row_id, RowAccessor *row_accessor);
```

This is a legacy interface that works only for evictable cache types.

8 Update Parameters

We provide three functions for updating parameters stored in the Parameter Server , for different needs.

If you are updating a single parameter, you can use:

```
void petuum::Table<UPDATE>::Inc(int32_t row_id, int32_t column_id,
    UPDATE update);
```

However, it is a lot more efficient to update a set of parameters in batch, using

```
void petuum::Table<UPDATE>::BatchInc(int32_t row_id, const
    UpdateBatch<UPDATE>& update_batch);
```

If you are updating a contiguous range of parameters in a row, you may want to use `DenseBatchInc()`:

```
void petuum::Table<UPDATE>::DenseBatchInc(int32_t row_id,
const DenseUpdateBatch<UPDATE> &update_batch);
```

9 Completion of A Clock Tick

```
static void petuum::PSTableGroup::Clock();
```

Parameter Server System-wise Configuration

`petuum::TableGroupConfig` includes system-wise configuration parameters.

10 Set Up A Distributed Application

Extending the single-node application to a distributed cluster requires configuring a few more parameters.

First of all, your `host_map` need to have one entry per process. To make the configuration easier, we allow the user to use the **optional** server file, which can be processed by Parameter Server utilities:

```
void petuum::GetHostInfos(std::string server_file,
std::map<int32_t, HostInfo> *host_map);
```

This functions takes in the path to the server file and automatically set up the `host_map` object for you.

The server file contains one process per line. Each line has three white-space separated entries: process ID, IP address, and port number.

An example server file is as below:

```
0 192.168.1.1 10000
1 192.168.1.2 10000
```

Inform the Parameter Server about the number of processes that you intend to run and also the processes' IDs:

Name	Default	Explanation
`num_total_clients`	1	Number of processes to run.
`client_id`	0	This process's ID.

11 Run More Worker Threads Per Node

Set the number of applications threads, including the init thread.

Name	Default	Explanation
`num_local_app_threads`	2	Number of local application threads, including the init thread.

12 Create More Than One Tables

There could be several reasons that one may want to create multiple tables including, but not limited to:

1. Different row types, including different row capacities.
2. Different staleness constraints.
3. Exceeding number of rows that cannot fit into one table.

Name	Default	Explanation
`num_tables`	`1`	Number of tables the system has.

13 Taking SnapShots and Resume From SnapShots

Name	Default	Explanation
`snapshot_clock`	`-1`	Freqnency of taking snapshots.
`snapshot_dir`	`""`	Directories the snapshot to be written to.
`resume_clock`	`-1`	Clock number to resume from.
`resume_dir`	`""`	Directories to resume from.

14 Numa Optimization

Name	Default	Explanation
`numa_opt`	`false`	Enable NUMA optimization?

Advanced feature. Use with caution.

14.1 Runtime Statistics

To collect runtime statistics, uncomment this line in `defns.mk`

```
PETUUM_CXXFLAGS += -DPETUUM_STATS
```

Name	Default	Explanation
`stats_path`	`""`	Path to store stats.

Parameter Server Table-wise Configuration

15 Choose Client Cache Types

Name	Default	Explanation
`process_storage_type`	`BoundedDense`	Type for client parameter storage.

`BoundedDense` is a contiguous chunk of memory. It works well if your whole model fits into client machine's memory. The row IDs that you may access is restrited to `[0, C - 1]`, where `C` is the cache capacity.

If you have a large model that does not fit into the memory. You may use `BoundedSparse`, which supports eviction. However, it comes with considerable overhead.

16 Choose Update Table Type

Name	Default	Explanation
`oplog_type`	`Dense`	Type for client update table type.

Use `Dense` if you use `BoundedDense` for cache type. Otherwise use `Sparse`.

17 Staleness Threshold

Name	Default	Explanation
`table_info.table_staleness`	`0`	SSP staleness threshold.

18 Row Capacity

Some (for example, dense) row types require this to be set.

19 Row OpLog Type

`1` is sparse. `0` is dense. Use dense if your updates are dense.

Name	Default	Explanation
`table_info.row_capacity`	0	Row capacity.

Name	Default	Explanation
`table_info.row_oplog_type`	1	Row opLog type.
`table_info.oplog_dense_serialized`	`false`	Dense serialize oplog.
`table_info.dense_row_oplog_capacity`	0	Row oplog capacity.

20 Use Thread-Level Cache

Thead cache allows each of your *table_thread* to cache a small number of frequently accessed rows in thread-private memory to avoid lock contention. To configure the thread cache size:

Name	Default	Explanation
`thread_cache_capacity`	1	Thread cache capacity.

The following functions allows you to read and write to thread cache.

```
void ThreadGet(int32_t row_id, ThreadRowAccessor* row_accessor);

void ThreadInc(int32_t row_id, int32_t column_id, UPDATE update);

void ThreadBatchInc(int32_t row_id, const UpdateBatch<UPDATE>&
    update_batch);

void ThreadDenseBatchInc(int32_t row_id, const DenseUpdateBatch<UPDATE>
    &update_batch);
```

Frequently Asked Questions

1. **How many tables should I create?**
   Consider spliting your parameters into multiple tables if
   - It is necessary or benefitiable to use different data structures for rows;
   - The namespace of one table is not large enough.

   Some people like to directly map some algorithmic concepts into Parameter Server tables. Although this is very intuitive, it might not render the optimal performance or might cause you trouble. The most frequently encountered issue is that the Parameter Server does not allow applications to dynamically create tables (tables are all created beforehand). However, you may add as many rows into an existing table as you need. In this case, you may consider use one table to represent multiple algorithmic objects by manipulatign the row IDs and thus you don't need dynamically create tables.
2. **How many rows should I create?**
   First of all, you should try to put the parameters that are accessed together into the same row so the cost of calling `Get()` can be amortized.
   On the other hand, row is the unit of data partitioning. What this means is that each server node in the Parameter Server is responsible for the rows that fall under its range. So you should at least have one row for each server node.