Lab 7 of SP19-BL-ENGR-E599-30563

In this session, we will first recap the vectorization techniques used by Harp-DAAL. Secondly, we will introduce the python interface of invoking Harp-DAAL application, which enables Harp-DAAL to be integrated into the data science workflow.

Goal

- Understand the vectorization behind Harp-DAAL
- Hands on of launching Harp-DAAL K-means from Python Interface

Deliverables

- Running Harp-DAAL K-means from Python and submit the centroid values

Evaluation

Lab participation: credit for 1 point based upon a successful completion of the lab tasks

Vectorization inside Harp-DAAL Applications

Harp-DAAL leverages high-performance native kernels pre-built by Intel. Most of these kernels invoke Intel's MKL (Math Kernel Library) at the backend. MKL provides C/C++ interface to BLAS (Basic Linear Algebra Subprograms) optimized on Intel's hardware architecture. For instance, the matrix-vector multiplication we practiced in last session is such a BLAS operation named `gemv`.

$y = \alpha Ax + \beta y$

MKL has the following API for double precision `gemv`

```
void cblas_dgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const
MKL_INT m,
const MKL_INT n, const double alpha, const double *a, const MKL_INT lda,
const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT
incy);
```

Some important parameters are

- trans: whether to apply a transposition on matrix A
- m: the number of rows in matrix A
- n: the number of columns in matrix A

- a: the array of data stored in matrix A
- x: the array of x
- y: the array of y

The MKL invocation within Harp-DAAL K-means

In Lab session 5, we know that the Harp-DAAL K-means offloads the local computation to DAAL K-means kernel

```
...
// specify centroids data to daal kernel
kmeansLocal.input.set(InputId.inputCentroids, cenTable_daal);
// first step of local computation by using DAAL kernels to get partial result
PartialResult pres = kmeansLocal.compute();
...
```

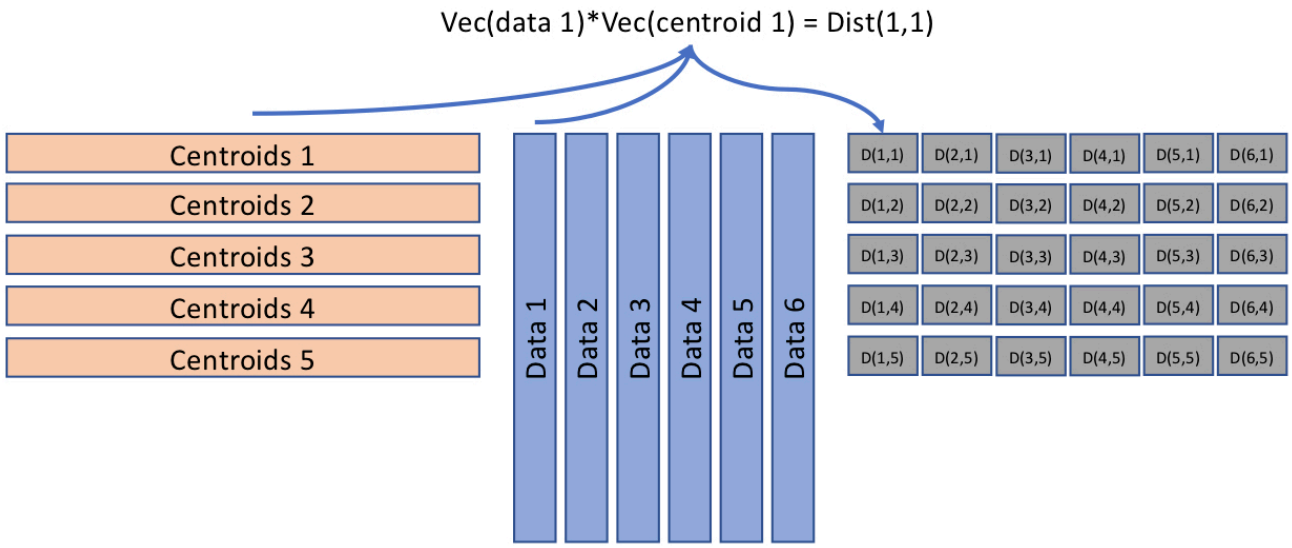
This API invokes the C++ kernels in DAAL implementation

```
template<typename algorithmFPType, Method method, CpuType cpu>
services::Status DistributedContainer<step1Local, algorithmFPType, method,
cpu>::compute()
{
    Input         *input = static_cast<Input *>(_in  );
    PartialResult *pres  = static_cast<PartialResult *>(_pres);
    Parameter     *par   = static_cast<Parameter *>(_par );

    const size_t na = 2;
    NumericTable *a[na];
    a[0] = static_cast<NumericTable *>(input->get(data          ).get());
    a[1] = static_cast<NumericTable *>(input->get(inputCentroids).get());

    ...

    __DAAL_CALL_KERNEL(env, internal::KMeansDistributedStep1Kernel,
                       __DAAL_KERNEL_ARGUMENTS(method, algorithmFPType), compute,
na, a, nr, r, par);
}
```



Calculate distance in a vectorized way

This kernel named `KMeansDistributedStep1Kernel` mainly has the following two steps of computation:

- A BLAS matrix-matrix multiplication (`xxgemm`) to compute the distance among data points to centroids.

```
// compute distance between each data points to each centroids
template<typename algorithmFPType, CpuType cpu, int assignFlag>
services::Status addNTToTaskThreadedDense(void *task_id, const NumericTable
*ntData, algorithmFPType *catCoef, NumericTable *ntAssign = 0)
{
    ...
    Blas<algorithmFPType, cpu>::xxgemm(&transa, &transb, &_m, &_n, &_k, &alpha,
data,
    &lda, inClusters, &ldy, &beta, x_clusters, &ldaty);
    ...
}
```

- A vectorized loop (by PRAGMA_ICC_OMP) to assign data points to new clusters.

```
PRAGMA_ICC_OMP(simd simdlen(16))
for (algIntType i = 0; i < (algIntType)blockSize; i++)
{
    algorithmFPType minGoalVal = x_clusters[i];
    algIntType minIdx = 0;

    for (algIntType j = 0; j < (algIntType)nClusters; j++)
    {
        algorithmFPType localGoalVal = x_clusters[i + j*blockSize];
        if( minGoalVal > localGoalVal )
```

```
        {
            minGoalVal = localGoalVal;
            minIdx = j;
        }
    }

    minGoalVal *= 2.0;
    *((algIntType*)&(x_clusters[i])) = minIdx;
    x_clusters[i+blockSize] = minGoalVal;
}
```

Python Interface of Harp-DAAL Invocation

You machine shall have already installed the following modules

1. Hadoop 2.6.0
2. Harp-DAAL
3. Python 2.7+
4. Numpy (install with your pip)

Items 1 and 2 shall have been installed at your home directory by previous lab sessions.

Python and Numpy

A python 2.7.5 is already available under `/usr/bin/python`

```
$ which python
$ /usr/bin/python
$ python --version
$ Python 2.7.5
```

Numpy is also installed with python

```
$ python
>>> import numpy
>>> print numpy.__version__
1.7.1
```

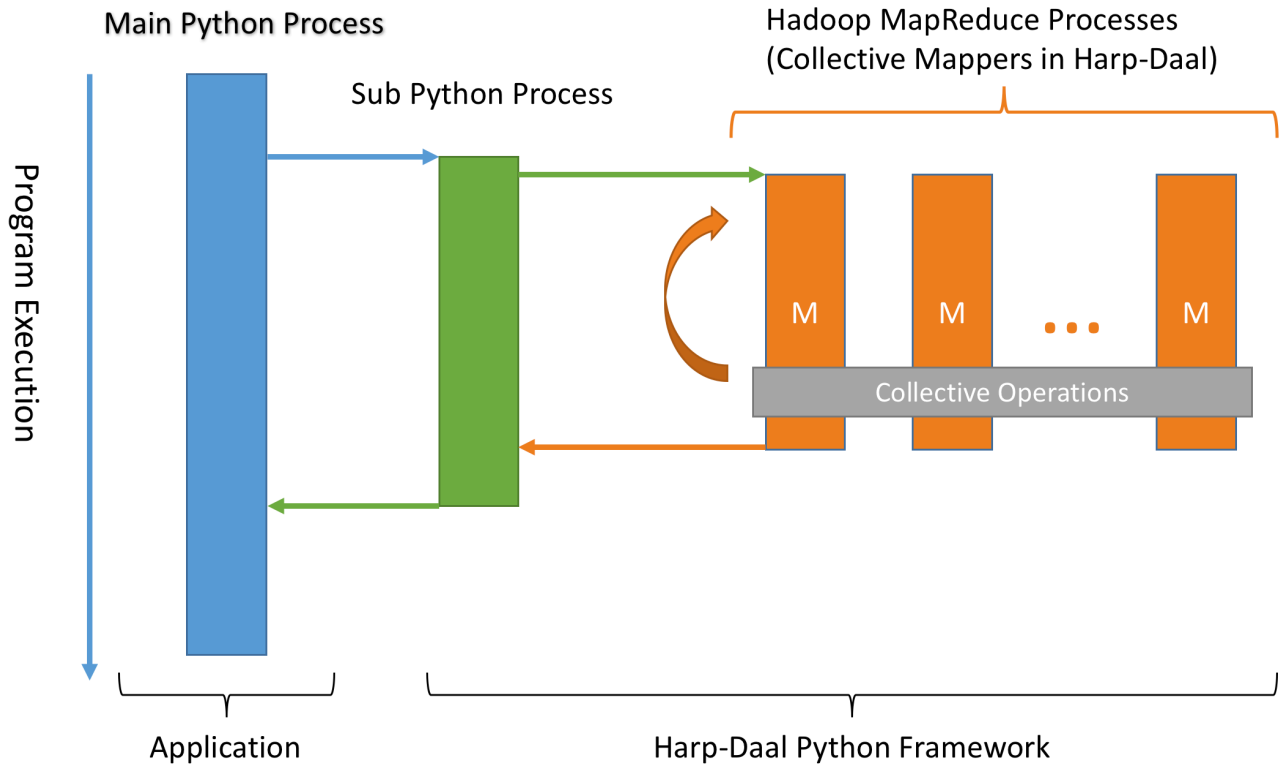
Get Harp-DAAL python Codes

Copy the labsession codes of Harp-DAAL python codes to your home directory

```
cp -r /share/jproject/lc37/harp-daal-python-tutorial ~/
cd ~/harp-daal-python-tutorial
```

The Interface of Harp-DAAL Python

The program starts as a python process (main python process), and it interacts with the Hadoop/Harp services through a sub python process. Once the control flow is transferred to the Hadoop/Harp process, it waits until the Hadoop/Harp process finished. The control is given back to the sub python process, and finally the sub python process merges back to the main python process.



Control flow of Harp-DAAL Python Invocation

The python interface codes are located at `harp/applications.py` and `harp/daal/applications.py`. `harp/applications.py` defines most of the interfacing functions

- Initialize and load hadoop/harp environment variables

```
def __init__(self, name):
        self.name = name
        self.hadoop_path = ''
        self.hadoop_cmd = ''
        self.harp_jar = ''
        self.class_name = ''
        self.cli = ''
        self.__load_hadoop_env()
        self.__load_harp_env()
```

- Start application and launch sub process

```
def run(self):
    self.log("{0} is running".format(self.__class__.__name__))
    self.sub_call(self.cli)
    self.log("{0} is finished".format(self.__class__.__name__))

def sub_call(self, command):
    try:
        return_code = subprocess.call(command, shell=True)
        if return_code < 0:
            print >> sys.stderr, "Child was terminated by signal", -return_code
    except OSError as e:
        print >> sys.stderr, "Execution failed:", e
```

- Print and retrieve the results from HDFS

```
def print_result(self, file_path):
    fs_cmd = self.hadoop_path + ' fs -cat ' + file_path
    self.log("Command: " + fs_cmd)
    self.sub_call(fs_cmd)

def result_to_array(self, file_path):
    cat = subprocess.Popen([self.hadoop_path, "fs", "-cat", file_path],
stdout=subprocess.PIPE)
    return numpy.loadtxt(cat.stdout)
```

`HarpDaalApplication` class is inherited from `HarpApplication` and overrides some functions as below.

```
class HarpDaalApplication(HarpApplication):
    def __init__(self, name):
        super(HarpDaalApplication, self).__init__(name)
        self.daal_flag = False  # By default, Intel Daal library is not used.
        self.__load_daal_env()
        self.__load_harp_daal_env()

    def __load_daal_env(self):
        try:
            daal_root = os.environ['DAALROOT']
            self.config_daal_root(daal_root)
        except KeyError:
            pass

    def __load_harp_daal_env(self):
        try:
```

```
            jar_path = os.environ['HARP_DAAL_JAR']
            self.config_harp_daal_jar(jar_path)
        except KeyError:
            pass

    def config_daal_root(self, daal_root):
        self.daal_root = daal_root
        self.libjars = "-libjars {0}/lib/daal.jar".format(self.daal_root)

    def config_harp_daal_jar(self, jar_path):
        self.harp_daal_jar = jar_path
        self.daal_flag = True

    def args(self, *args, **kwargs):
        self.cmd = ''
        for arg in args:
            self.cmd = self.cmd + str(arg) + ' '
        self.cli = "{0} {1} {2} {3} {4}".format(self.hadoop_cmd,
self.harp_daal_jar, self.class_name, self.libjars, self.cmd)
        self.log("Command: " + self.cli)
```

It also contains application class, such as the `KMeansDaalApplication`

```
class KMeansDaalApplication(HarpDaalApplication):
    def __init__(self, name):
        super(KMeansDaalApplication, self).__init__(name)
        self.class_name =
'edu.iu.daal_kmeans.regroupallgather.KMeansDaalLauncher'

    def init_centroids(self, data):
        file_path = self.get_workdir() + '/centroids/init_centroids'
        with tempfile.NamedTemporaryFile(delete=False) as tf:
            numpy.savetxt(tf, data, fmt="%f")
        self.put_file(tf.name, file_path)
```

The `__init__` function defines the invoked Harp-DAAL application JAR name as `edu.iu.daal_kmeans.regroupallgather.KMeansDaalLauncher`, and the function `init_centroids` links the centroids data in HDFS to data stored in Numpy.

The Example Codes of Harp-DAAL K-means

The example codes of Harp-DAAL K-means is located at `examples/daal/run_harp_daal_KMeansDaal.py`

```

```

```
## import harp-daal python interface and Numpy
from harp.daal.applications import KMeansDaalApplication
import numpy

## create the python k-means example
my_app = KMeansDaalApplication('My KMeansDaal with Harp')

## feed the command ine arguments
my_app.args('1 4 110000 100 /daal-kmeans-data /daal-kmeans-work 100 100 10 true
100000 5 /tmp/kmeans')

## launch the sub process
my_app.run()

## prints the results and fetch results from HDFS to python
my_app.print_result('/daal-kmeans-work/evaluation')
arr = my_app.result_to_array('/daal-kmeans-work/centroids/out/output')
print(arr)
sorted_arr = numpy.sort(arr)
print(sorted_arr)
```

Run the Harp-DAAL K-means Python Example

Use the `run.sh` script to run applications. First make sure that the Hadoop daemons are already launched.

```
$ jps
$ yarn node -list
```

The contents of scripts are

```
#!/bin/bash

# root directory of the lab session harp installation
export MyLabSRoot=/N/u/lc37/Project/LabSession/harp

## set up env vars
export HARP_JAR=$MyLabSRoot/ml/java/target/harp-java-0.1.0.jar
export HARP_DAAL_JAR=$MyLabSRoot/ml/daal/target/harp-daal-0.1.0.jar
export DAALROOT=$MyLabSRoot/third_party/daal-2018
export PYTHONPATH=$MyLabSRoot/harp-python

## load daal so files into hdfs
hdfs dfs -mkdir -p /Hadoop
```

```
hdfs dfs -mkdir -p /Hadoop/Libraries
hdfs dfs -rm /Hadoop/Libraries/*
hdfs dfs -put ${DAALROOT}/lib/intel64_lin/libJavaAPI.so /Hadoop/Libraries/
hdfs dfs -put ${DAALROOT}/../tbb/lib/intel64_lin/gcc4.4/libtbb*
/Hadoop/Libraries/

# bash copy-data.sh
python examples/daal/run_harp_daal_KMeansDaal.py
```

- Modify the `MyLabSRoot` by your path to the harp root directory of the labsession branch
- Modify the `PYTHONPATH` by the path to your `harp-daal-python-tutorial`
- Run the script and wait for the results

```
./run.sh
```

Now, you can copy the centroid values and submit it on Canvas.