Loom Documentation

Release 0.7

Loom Team

May 16, 2017

Contents

CHAPTER 1

User guide

Introduction

HyperLoom is a platform for defining and executing workflow pipelines in a distributed environment. HyperLoom aims to be a highly scalable framework that is able to efficiently execute millions of interconnected tasks on hundreds of computational nodes.

User defines and submits a plan - a computational graph (Directed Acyclic Graph) that captures dependencies between computational tasks. The HyperLoom infrastructure then automatically schedules the tasks on available nodes while managing all necessary data transfers.

Architecture

HyperLoom architecture is depicted in Fig. 1.1. HyperLoom consist of a server process that manages worker processes running on computational nodes and a client component that provides an user interface to HyperLoom.

The main components are:

- **client** – The Python gateway to HyperLoom – it allows users to programmatically chain computational tasks into a plan and submit the plan to the server. It also provides a functionality to gather results of the submitted tasks after the computation finishes.
- **server** – receives and decomposes a HyperLoom plan and reactively schedules tasks to run on available computational resources provided by workers.
- **worker** – executes and runs tasks as scheduled by the server and inform the server about the task states. HyperLoom provides options to extend worker functionality by defining custom task or data types. (Server and worker are written in C++.)

Basic terms

The basic elements of Loom's programming model are: **data object**, **task**, and **plan**. A **data object** is an arbitrary data structure that can be serialized/deserialized. A **task** represents a computational unit that produces data objects. A

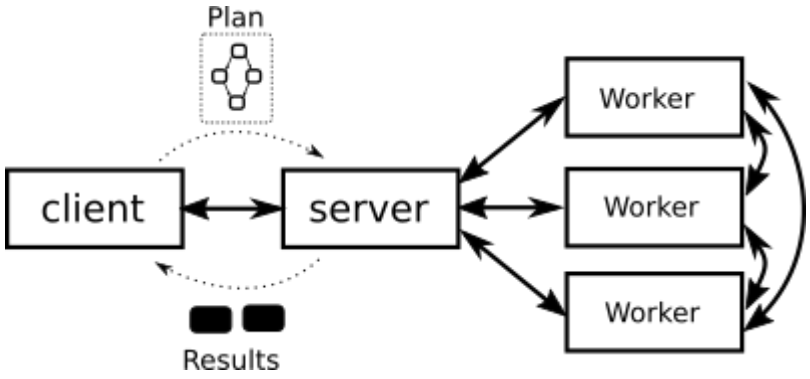

Fig. 1.1: Architecture of HyperLoom

plan is a set of interconnected tasks.

Tasks

A **task** is an object representing a computation together with its dependencies and a configuration. Each task has the following attributes:

- Task inputs – task's prerequisites (some other tasks)
- Task type – the specification of the procedure that should be executed
- Task policy – defines how should be the task scheduled
- Configuration – a sequence of bytes that is interpreted according the task type
- Resource constraints

By task *execution*, we mean executing a procedure according to *task type*, which takes data objects and configuration, and returns a new data object. The input data objects are obtained as a result of executing tasks defined in task inputs. Resource constraints serve to express that a task execution may need some specific hardware or number of processes.

Plan

Plan is a set of tasks. Plan has to form a finite asyclic directed multigraph where nodes are tasks and arcs express input dependencies between tasks. *Plan execution* is an execution of tasks according to the dependencies defined in the graph.

Note:

- We have formally restricted each task to return only a single data object as its result. However, a task can produce more results by returning an array of data objects.
- Input data objects are always results of a previous tasks. To create a specific constant data object, there is a standard task (`tasks.const` in Python API) that takes no input and only creates a data object from its configuration.

Symbols

Customization and extendability are important concepts of HyperLoom. HyperLoom is designed to enable creating customized workers that providies new task types, data objects and resources. HyperLoom uses the concept of name spaces to avoid potential name clashes between different workers. Each type of data object, task type and resource type is identified by a symbol. Symbols are hierarchically organized and the slash character / is used as the separator of each level (e.g. *loom/data/const*). All built-in task types, data object types, and resource types always start with *loom/* prefix. Other objects introduced in a a specialized worker should introduce its own prefix.

Data objects

Data objects are fundamental entities in HyperLoom. They represent values that serves as arguments and results of tasks. There are the following build-in basic types of data objects:

- **Plain object** – An anonymous sequence of bytes without any additional interpretation by HyperLoom.
- **File** – A handler to an external file on shared file system. From the user's perspective, it behaves like a plain object; except when a data transfer between nodes occurs, only a path to the file is transferred.
- **Array** – A sequence of arbitrary data objects
- **Index** – A logical view over a D-Object data object with a list of positions. It is used to slice data according some positions (e.g. positions of the new-line character to extract lines). It behaves like an array without explicit storing of each entry.
- **PyObj** – Contains an arbitrary Python object

We call objects that are able to provide a content as continous chunk of memory as **D-Objects**. Plain object and File object are D-Objects; Array, Index, and PyObj are *not* D-Objects.

Each data object

- **size** – the number of bytes needed to store the object
- **length** – the number of 'inner pieces'. Length is zero when an object has no inner structure. Plain objects and files have always zero length; an array has length equal to number of lements in the array.

Note: **size** is an approximation. For a plain object, it is the length of data itself without any metada. The size of an array is a sum of sizes of elements. The size of PyObj is obtained by `sys.getsizeof`.

Installation

HyperLoom has two components from the installation perspective:

- Runtime - the HyperLoom infrastructure (Server and Worker)
- Python client

Both components resides in the same Git repository, but their installations are independent.

The main repository is: https://code.it4i.cz/ADAS/loom

Runtime

The HyperLoom infrastructural components depend on the following libraries that are not included in the HyperLoom source code:

- **libuv** – Asychronous event notification
- **Protocol buffers** – Serialization library
- **Python >=3.4** (optional)
- **Clouldpickle** (optional)

(HyperLoom also depends on **spdlog** and **Catch** that are distributed together with HyperLoom)

In **Debian** based distributions, dependencies can be installed by the following commands:

```
apt install libuv-dev libprotobuf-dev
pip install cloudpickle
```

Note: If you are going to create plans with many tasks, you can obtain a significant speedup by using PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION="cpp" feature.

When dependencies are installed, HyperLoom itself can be installed by the following commands:

```
cd loom
mkdir _build
cd _build
cmake ..
make
make install
```

Python client

Python client depends on:

- **Protocol buffers**
- **Cloudpickle**

Python client can be installed by the following commands:

```
cd loom/python
sh generate.sh
python setup.py install
```

Python client

Basic usage

The following code contains a simple example of HyperLoom usage. It creates two constants and a task that merge them. Next, it creates a client and connect to the server and submits the plan and waits for the results. It assumes that the server is running at address *localhost* on TCP port 9010.

```
from loom.client import Client, tasks

task1 = tasks.const("Hello ")        # Create a plain object
task2 = tasks.const("world!")        # Create a plain object
task3 = tasks.merge((task1, task2))  # Merge two data objects together
```

```
client = Client("localhost", 9010)    # Create a client
result = client.submit_one(task3)     # Submit task
print(result.gather())                # prints b"Hello world!"
```

The full list of build-in tasks can be found in *Tasks*. Method `submit_one` is non-blocking and returns instance of `loom.client.Future` that represents a remote computation in HyperLoom infrastructure. There are basic four operations that is provided by `loom.client.Future`:

- `wait()` - The operation blocks the client until the task is not finished.
- `fetch()` - The operation waits until the task is not finished, then it downloads the content to the client (while the results also remains on workers).
- `release()` - It removes results from workers. This method is automatically called in __del__ method of the object, hence you do not have to called it manually. However, it is a good practice to explicitly call the method to release resources as soon as possible and do not depend on garbage collecting in the client.
- `gather()` - Basically, it is a short cut for `fetch()` + `release()`. It downloads data to the client and removes them from the workers. For a single future it is actually the same as calling `fetch()` followed by `release()` but when we work with more futures it allows some optimizations.

Submitting more tasks at once

All previously mentioned methods have alternatives for working with more tasks/futures at once:

```
from loom.client import Client, tasks

task1 = tasks.const("Hello ")                    # Create a plain object
task2 = tasks.const(" ")                         # Create a plain object
task3 = tasks.const("world!")                    # Merge two data objects together

client = Client("localhost", 9010)               # Create a client
results = client.submit((task1, task2, task3))   # Submit tasks; returns list of␣
↪futures
print(client.gather(results))                    # prints [b"Hello world!", b" ", b
↪"world!"]
```

In this case, we have replaced `submit_one` by method `submit` that takes a collection of tasks and we have called the method `gather` not on the future but directly on the client. Client also have methods `wait`, `relase`, and `fetch` for collective future processing.

When possible, it is recommdended to use collective processing futures, since it allows some optimizations in comparison of processing tasks/futures in a loop separately.

Reusing futures as tasks inputs

Futures can be also used as input for tasks. This allows to use a gradual submitting, i.e. HyperLoom may already computes some part of the computation while the remaining plan is still composed.

```
task1 = ...                         # create a task
f1 = client.submit_one(task1)       # submit task

task2 = ...                         # create a second task
taskA = tasks.merge((f1, tasks2)) # create task that uses f1 and taskA
fA = client.submit_one(f1)
```

It does not matter if `task1` is finished yet or not, as far it is not released it can be used as an input. In other words, you can call `wait` and `fetch` on futures and they can be still used in tasks; however `release` or `gather` releas tasks from the workers and it cannot be used anymore.

Important: The following code is usually a bad pattern:

```
task1 = ...
task2 = tasks.run("program1", stdin=task1)
f2 = client.submit_one(task2)
task3 = tasks.run("program1", stdin=task1)
f3 = client.submit_one(task3)
client.gather((f2, f3))
```

Task `task1` is computed twice! Task `task1` is requested in both submissions but we did not indicate that we want to reuse its result later.

The better code:

```
task1 = ...
f1 = client.submit_one(task1)
task2 = tasks.run("program1", stdin=f1)
f2 = client.submit_one(task2)
task3 = tasks.run("program1", stdin=f1)
f3 = client.submit_one(task3)
client.gather((f2, f3))
```

or (without gradual submmiting):

```
task1 = ...
task2 = tasks.run("program1", stdin=task1)
task3 = tasks.run("program1", stdin=task1)
f2, f3 = client.submit((task2, task3))
client.gather((f2, f3))
```

In both cases, `task1` is computed only once.

Running external programs

In this subsection, we demonstrate a running of external programs. The most basic scenario is execution of a program while mapping a data object on standard input and capturing the standard output. It can be achieved by the following code:

```
task1 = ...
task_run = tasks.run("/bin/grep Loom", stdin=task1)
```

If the `task_run` is executed, the standard unix program *grep* is executed. Result from `task` is mapped on its standard input and output is captured. Therefore, this example creates a new plain data object that contains only lines containing string *Loom*.

If the first argument is string, as in the above example, then Loom expects that arguments are separated by white spaces. But argument may be provided explicitly, e.g.

```
task_run = tasks.run(("/path/to/program", "--arg1", "argument with spaces"))
```

Mapping input files

If the executed program cannot read data from the standard input or we need to provide more inputs, `run` allows to map data objects to files.

The following code maps the result of `task_a` to *file1* and result of `task_b` to *file2*.

```
task_a = ...
task_b = ...
task_run = tasks.run("/bin/cat file1 file2",
                     [(task_a, "file1"), (task_b, "file2")])
```

A new fresh directory is created for each execution of the program and the current working directory is set to this directory. Files created by mapping data objects are placed to this directory. Therefore, as far as only relative paths are used, no file conflict occurs. Therefore the following code is correct, even all three tasks may be executed on the same node simultaneously.

```
task_a = ...
task_b = ...
task_c = ...

task_1 = tasks.run("/bin/cat file1", [(task_a, "file1")])
task_2 = tasks.run("/bin/cat file1", [(task_b, "file1")])
task_3 = tasks.run("/bin/cat file1", [(task_c, "file1")])
```

Mapping output files

So far, the result of `run` tasks is created by gathering the standard output. There is also an option to create a result from files created by the program execution.

Let us assume that program */path/program1* creates *outputs.txt* as the output, then we can run the following program and capturing the file at the end (standard output of the program is ignored).

```
task = tasks.run("/path/program1", outputs=("output.txt",))
```

The user may define more files as the output. Let us consider the following code, that assumes that *program2* creates two files.

```
task = tasks.run("/path/program2", outputs=("output1.txt", "output2.txt"))
```

The result of this task is an array with two elements. This array contains with two plain data objects.

If `None` is used instead of a name of a file, than the standard output is captured. Therefore, the following task creates a three element array:

```
task = tasks.run("/path/program3",
                 outputs=("output1.txt",  # 1st element of array is got from 'output1.
↪txt'
                          None,           # 2nd element of array is stdout
                          "output2.txt")) # 3rd element of array is got from 'output2.
↪txt'
```

Variables

In previous examples, we have always used a constant arguments for programs; however, programs arguments can be also parametrized by data objects. When an input data object is mapped to a file name that starts with character *$* then

no file is mapped, but the variable with the same name can be used in arguments. HyperLoom expands the variable before the execution of the task.

The following example executes program *ls* where the first argument is obtained from data object.

```
path = tasks.const("/some/path")
task = tasks.run("/bin/ls $PATH", [(path, "$PATH")])
```

Note: See *Task redirection* for a more powerfull dynamic configuration of `run`.

Error handling

When an executed program exits with a non-zero exit code then the server reports an error that is propagated as `TaskFailed` exception in the client.

```
task = tasks.run("ls /non-existent-path")
try:
   result = client.submit_one(task)
   result.wait()
except TaskFailed as e:
   print("Error: " + str(e))
```

This program prints the following:

```
Error: Task id=2 failed: Program terminated with status 2
Stderr:
ls: cannot access '/non-existing-dictionary': No such file or directory
```

Python functions in plans

HyperLoom allows to execute directly python functions as tasks. The easiest way is to use decorator `py_task()`. This is demonstrated by the following code:

```
from loom.client import tasks

@tasks.py_task()
def hello(a):
    return b"Hello " + a.read()

task1 = tasks.cont("world")
task2 = hello(task1)

result = client.submit_one(task2)
result.gather()  # returns b"Hello world"
```

The `hello` function is seralized and sent to the server. The server executes the function on a worker that has necessary data.

- When `str` or `bytes` is returned from the function then a new plain data object is created.
- When `loom.client.Task` is returned then the the task redirection is used (see *Task redirection*).
- When something else is returned or exeption is thrown then the task fails.
- Input arguments are wrapped by objects that provide the following methods

  - `read()` - returns the content of the object as `bytes`, if data object is not D-Object than empty bytes are returned.
  - `size()` - returns the size of the data object
  - `length()` - returns the length of the data object

- `tasks.py_task` has optional `label` parameter to set a label of the task if it is not used, then the name of the function is used. See XXX for more information about labels

Decorator `py_task()` actually uses `loom.client.tasks.py_call()`, hence the code above can be written also as:

```
from loom.client import tasks

def hello(a):
    return b"Hello " + a.read()

task1 = tasks.cont("world")
task2 = tasks.py_call(tasks.py_value(hello), (task1,))
task2.label = "hello"

client.submit_one(task2)  # returns b"Hello world"
```

Task redirection

Python tasks (used via decorator `py_task` or directoly via `py_call`) may return `loom.client.Task` to achive a task redirection. It is useful for simple dynamic configuration of the plan.

Let us assume that we want to run `tasks.run`, but configure it dynamically on the actual data. The following function takes two arguments, checks the size and then executes `tasks.run` with the bigger one:

```
from loom.client import tasks

@tasks.py_task()
def my_run(a, b):
    if a.size() > b.size():
        data = a
    else:
        data = b
return tasks.run("/some/program", stdin=data)
```

Task context

Python task can configured to obtain a `Context` object as the first argument. It provides interface for interacting with the HyperLoom worker. The following example demonstrates logging through context object:

```
from loom.client import tasks

@tasks.py_task(context=True)
def hello(ctx, a):
    ctx.log_info("Hello was called")
    return b"Hello " + a.read()
```

The function is has the same behavior as the `hello` function in *Python functions in plans*. But not it writes a message into the worker log. `Context` has five logging methods: `log_debug`, `log_info`, `log_warn`, `log_error`, and `log_critical`.

Moreover `Context` has attribute `task_id` that holds the indentification number of the task.

Direct arguments

Direct arguments serve for the Python task configuration without necessity to create HyperLoom tasks. From the user perspective it works in a similar way as context – they introduces extra parameters. The values for parameters are set when the task is called. They can be arbitrary serializable objects and they are passed to the function when the py_task is called. Direct arguments are always passed as the first n arguments of the function. They are specified only by a number, i.e. how many first n arguments are direct (the rest arguments are considered normal HyperLoom tasks).

Let us consider the following example:

```
from loom.client import tasks

@tasks.py_task(n_direct_args=1)
def repeat(n, a):
    return n * a.read()

c = tasks.const("ABC")
t1 = repeat(2, c)
t2 = repeat(3, c)

client.submit_one(t1).gather()  # returns: b"ABCABC"
client.submit_one(t2).gather()  # returns: b"ABCABCABC"
```

Note: When *context* and *direct arguments* are used together, then the context is the first argument and them follows the direct arguments.

For the completeness, the following code demonstrates the usage of direct arguments via `py_call`:

```
from loom.client import tasks

def repeat(n, a):
    return n * a.read()

c = tasks.const("ABC")
t1 = tasks.py_call(tasks.py_value(repeat), (c,), direct_args=(2,))
client.submit_one(t1).gather()  # returns: b"ABCABC"
```

Python objects

Data objects in HyperLoom can be directly a Python objects. A constant value can be created by `tasks.py_value`:

```
from loom.client import tasks

my_dict = tasks.py_value({"A": "B"})
```

It is similar to `tasks.const`, but it creates PyObj instead of Plain object.

PyObj can be used in `py_task`. It has to be unwrapped from the wrapping object first:

```
@py_task()
def f(a):
    d = a.unwrap()
```

```
    return "Value of 'A' is " + d["A"]

t = f(my_dict)
client.submit_one(t).gather()  # returns b"Value of 'A' is B"
```

If we want to return a PyObj from py_task we have wrap it to avoid implicit conversion to Data objects:

```
@py_task()
def example_1():
    return "Hello"

@py_task(context=True)
def example_2(ctx):
    return ctx.wrap("Hello")

@py_task(context=True)
def example_3(ctx):
    return [ctx.wrap({"A", (1,2,3)}), "Hello"]
```

The first example returns a plain object. The second example returns PyObj. The third one returns HyperLoom array with PyObj and plain object.

Important: HyperLoom always assumes that all data objects are immutable. Therefore, modyfing unwrapped objects from PyObj leads to highly undefined behavior. It is recommended to store only immutable objects (strings, tuples, frozensets, ...) in PyObj to prevent problems. If you store a mutable object in PyObj, be extra carefull to not modify it.

```
# THIS EXAMPLE CONTAINS ERROR
@py_task()
def modify_arg(a):
    my_obj = a.unwrap()
    my_obj[0] = 321  # HERE IS ERROR, we are modyfing unwrapped object

value = tasks.py_value([1,2,3,4])
modify_arg(value)
```

Note: Applying `wrap` on Data wrapper returns the argument without wrapping.

Reports

Reporting system serves for debugging and profiling the HyperLoom programs. Reports can be enabled by `set_trace` method as follows:

```
task = ...
client.set_trace("/path/to/mytrace")
result = client.submit_one(task)
...
```

The path provided to `set_trace` has to be placed on a network filesystem that is visible to server and all workers. It creates a directory `/path/to/mytrace` where server and workers writes its traces.

The trace can be explored by `loom.lore`.

```
$ python3 -m loom.lore /path/to/mytrace
```

It creates file *output.html* that contains the final report. The full list of commands can be obtained by

```
$ python3 -m loom.rview --help
```

Labels

Each task may optinally define a **label**. It serves for debugging purpose – it changes how is the task shown in *rview*. Label has no influence on the program execution. The label is defined as follows:

```
task = tasks.const("Hello")
task.label = "Initial data"
```

rview assigns colors of graph nodes or lines in a trace according the labels. The two labels have the same color if they have the same prefix upto the first occurence of character `:`. In the following example, three colors will be used. Tasks `task1` and `task2` will share the same color and `task3` and `task4` will also share the same color.

```
task1.label = "Init"
task2.label = "Init"
task3.label = "Compute: 1"
task4.label = "Compute: 2"
task5.label = "End"
```

Resource requests

Resource requests serves to specify some hardware limitations or inner paralelism of tasks. The current version supports only requests for a number of cores. It can be express as follows:

```
from loom.client import tasks

t1 = tasks.run("/a/parallel/program")
t1.resource_request = tasks.cpus(4)
```

In this example, `t1` is a task that reserves 4 cpu cores. It means that if a worker has 8 cores, that at most two of such tasks is executed simultaneously. Note that if a worker has 3 or less cores, than `t1` is never scheduled on such a worker.

When a task has no `resource_request` than scheduler assumes that the task is a light weight one and it is executed very fast without resource demands (e.g. picking an element from array). The scheduler is allows to schedule simultenously more light weight tasks than cores available for the worker.

Important: Basic tasks defined module `loom.tasks` do not define any resource request; except `loom.tasks.run`, `loom.tasks.py_call`, `loom.tasks.py_value`, and `loom.tasks.py_task` by default defines resource request for 1 cpu core.

Dynamic slice & get

HyperLoom scheduler recognizes two special tasks that dynamically modify the plan – **dynamic slice** and **dynamic get**. They dynamically create new tasks according the length of a data object and the current number of workers and their resources. The goal is to obtain an optimal number of tasks to utilize the cluster resources.

The following example:

```
t1 = tasks.dslice(x)
t2 = tasks.XXX(..., t1, ...)
result = tasks.array_make((t2,))
```

is roughly equivalent to the following code:

```
t1 = tasks.slice(x, 0, N1)
s1 = tasks.XXX(..., t1, ...)
t2 = tasks.slice(x, N1, N2)
s2 = tasks.XXX(..., t2, ...)
...
tk = tasks.slice(x, Nk-1, Nk)
sk = tasks.XXX(..., tk, ...)
result = tasks.array_make((s1, ..., sk))
```

where 0 < N1 < N2 ... Nk where Nk is the length of the data object produced by `x`.

Analogously, the following code:

t1 = tasks.dget(x) t2 = tasks.XXX(..., t2, ...) result = tasks.make_array((t2,))

is roughly equivalent to the following code (where is *N* is the length of the the data object produced by `x`:

```
t1 = tasks.get(x, 0)
s1 = tasks.XXX(..., t1, ...)
t2 = tasks.get(x, 1)
s2 = tasks.XXX(..., t2, ...)
...
tN = tasks.get(x, N)
sN = tasks.XXX(..., tk, ...)
result = tasks.array_make((s1, ..., sN))
```

Own tasks

Module `tasks` contains tasks provided by the worker distributed with HyperLoom. If we extend a worker by our own special tasks, we also need a way how to call them from the client.

Let us assume that we have extended the worker by task *my/count* as is shown in *New tasks*. We can create the following code to utilize this new task type:

```
from loom.client import Task, tasks

def my_count(input, character):
    task = Task()
    task.task_type = "my/count"
    task.inputs = (input,)
    task.config = character
    return task

t1 = tasks.open("/my/file")
t2 = my_count(t1)

...

result = client.submit_one(t2)
result.gather()
```

Extending worker

Warning: The API in the following section is not yet fully stable. It may be changed in the near future.

HyperLoom infrastructure offers by default a set of operations for basic manipulation with data objects and running and external programs. One of this task is also task *loom/py_call* (it can be used via `tasks.py_call` or `tasks.py_task` in Python client). This task allows to executed arbitrary Python codes and the user may define new tasks.

The another way is to directly extend a worker itself. The primary purpose is efficiency, since worker extensions can be written in C++. Moreover, this approach is more powerfull than py_call, since not only tasks but also new data objects may be introduced.

On the implementation level, HyperLoom contains a C++ library **libloom** that implements the worker in an extensible way.

New tasks

Let us assume that we want to implement a task that returns a number of a specified characters in a D-object. First, we define the code of the task itself:

```
#include "libloom/threadjob.h"

class CountJob : public loom::ThreadJob
{
public:
   using ThreadJob::ThreadJob;

   std::shared_ptr<loom::Data> run() {
       // Verify inputs and configuration
       if (inputs.size() != 1 || task.config.size() != 1) {
            set_error("Invalid use of the task");
            return nullptr;
       }
       char c = task.config[0]; // Get first character of config

       if (!inputs[0].has_raw_data()) {
            set_error("Input object does not contain raw data");
            return nullptr;
       }

       // Get pointer to raw data
       const char *mem = inputs[0].get_raw_data();

       // Perform the computation
       size_t size = inputs[0].get_size();
       uint64_t count = 0;
       for (size_t i = 0; i < size;i ++) {
            if (mem[i] == c) {
                 count += 1;
            }
       }

       // Create result
       auto output = std::make_shared<RawData>();
       output->init_from_mem(work_dir, &count, sizeof(count));
```

```
        return std::static_pointer_cast<Data>(output);
    }
};
```

`loom::ThreadJob` serves for defining a tasks that are executed in its own thread. The subclass has to implement `run()` method that is executed when the task is fired. It should return data object or `nullptr` when an error occurs.

The following code defines `main` function for the modified worker. It is actually the same code as for the worker distributed with HyperLoom except the registartion of our new task. Each task has to be registered under a symbol. Symbols for buildin tasks, data objects and resource requests starts with prefix *loom/*. To avoid name clashes, it is good practice to introduce new prefix, in our example, it is prefix *my/*.

```
#include "libloom/worker.h"
#include "libloom/log.h"
#include "libloom/config.h"

#include <memory>

using namespace loom;

int main(int argc, char **argv)
{
    /* Create a configuration and parse args */
    Config config;
    config.parse_args(argc, argv);

    /* Init libuv */
    uv_loop_t loop;
    uv_loop_init(&loop);

    /* Create worker */
    loom::Worker worker(&loop, config);
    worker.register_basic_tasks();

    /* --> Registration of our task <-- */
    worker.add_task_factory<ThreadTaskInstance<CountJob>>("my/count");

    /* Start loop */
    uv_run(&loop, UV_RUN_DEFAULT);
    uv_loop_close(&loop);
    return 0;
}
```

New data objects

TODO

CHAPTER 2

API

API: Python Client

Client

Future

Tasks