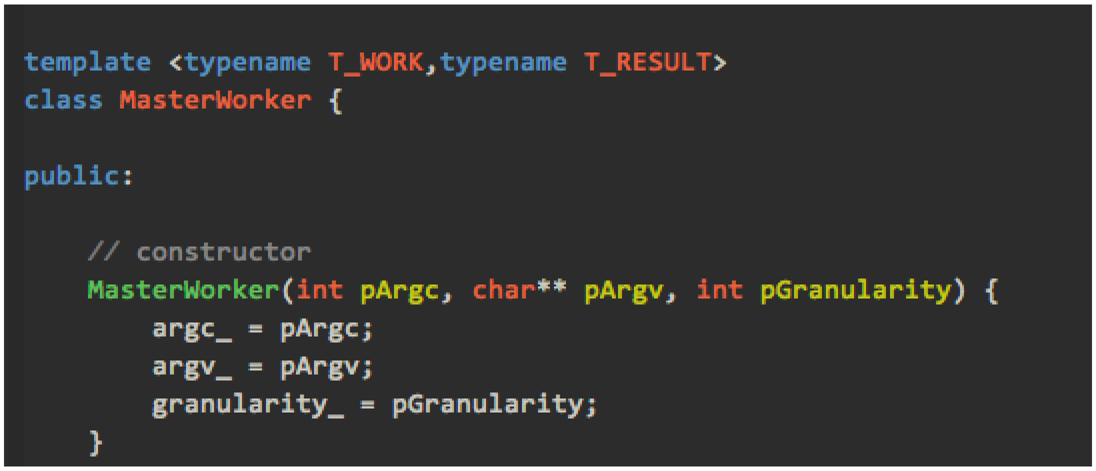
# Lab 2: Master-worker Framework

*Zeyue Chen(zc296) & Jiheng Lu(jl3474)*

## Part 1: Template for master-worker computation

We created a C++ base class **MasterWorker** which has two templates that are used as work struct and result struct.

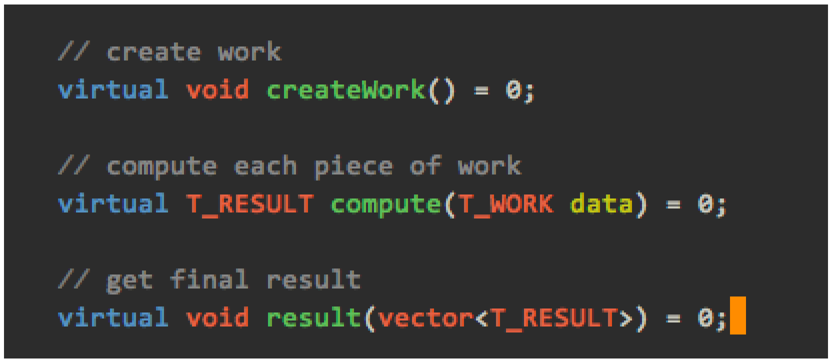


```
template <typename T_WORK,typename T_RESULT>
class MasterWorker {

public:

    // constructor
    MasterWorker(int pArgc, char** pArgv, int pGranularity) {
        argc_ = pArgc;
        argv_ = pArgv;
        granularity_ = pGranularity;
    }
```

The user should also implement these three pure virtual functions to create the initial pool of work, turns a piece of work into a result and merge all results to a final output.



```
  // create work
  virtual void createWork() = 0;

  // compute each piece of work
  virtual T_RESULT compute(T_WORK data) = 0;

  // get final result
  virtual void result(vector<T_RESULT>) = 0;
```

The workflow is as follows:

1. The user should create a bunch of T_WORKs and push them into a work pool.
2. In the initial round, the master will send T_WORKs to all workers. Then it begins to wait for the results.
3. Before sending the T_WORK, the master will also send a stop sign to the work, which indicate if the work should stop work.

4. If the worker receive a “NO” stop sign, it will call the computation function to turn the T_WORK to a T_RESULT and send this result back to master.
5. When the master receive a result from a worker, it puts the result to a vector and then check if the work pool is empty, if not, it will send another new T_WORK to this worker.
6. The step 5 will keep looping until all result are received.
7. Finally, the master will call the result function to merge all T_RESULTs to a final output.

## Part 2: Using the template

There are some interesting things during the implementation of this factor computing program.

1. “yum install gmp-devel” should not be used to install the GMP. Since this package is too old(only 4.0). We downloaded the source directly and compiled by ourselves. While compiling the source, we should also add cxx enable in the making
2. There should not be any class in the user defined T_WORK and T_RESULT. Since it will cause the memory address not consequent. We can only use basic types like int, char.

In the program, we are using char array to represent a large number when sending messages. When the worker received a message, it will turn this char array to mpz_class to do large number computation.

Our program also supports two different modes: pre-assign and dynamical-sending. When it is running on pre-assign mode. The worker will create work locally and only compute once.

We tested different combination of parameters.
In the pre-assign mode, we tested two number 1e14 and 1e16 on 2, 4, 8, 16 cores.
In the dynamical-sending mode, we tested two numbers 1e14 and 1e16 with 1000, 10000, 100000 granularities on 2, 4, 8, 16 cores. The results are shown below:

| **Pre-Assign** | **Core Count** | **Num** | **Time(s)** |
|---|---|---|---|
| YES | 2 | 1.00E+14 | 1.61 |
| YES | 4 | 1.00E+14 | 1.51 |
| YES | 8 | 1.00E+14 | 1.48 |
| YES | 16 | 1.00E+14 | 1.58 |
| | | | |
| YES | 2 | 1.00E+16 | 14.09 |
| YES | 4 | 1.00E+16 | 5.83 |

| | | | |
|---|---|---|---|
| YES | 8 | 1.00E+16 | 5.51 |
| YES | 16 | 1.00E+16 | 5.61 |

| Pre-Assign | Core Count | Granularity | Num | Time(s) |
|---|---|---|---|---|
| NO | 2 | 1000 | 1.00E+14 | 1.95 |
| NO | 4 | 1000 | 1.00E+14 | 1.91 |
| NO | 8 | 1000 | 1.00E+14 | 1.89 |
| NO | 16 | 1000 | 1.00E+14 | 2.41 |
| | | | | |
| NO | 2 | 10000 | 1.00E+14 | 1.83 |
| NO | 4 | 10000 | 1.00E+14 | 1.79 |
| NO | 8 | 10000 | 1.00E+14 | 1.77 |
| NO | 16 | 10000 | 1.00E+14 | 2.55 |
| | | | | |
| NO | 2 | 100000 | 1.00E+14 | 1.91 |
| NO | 4 | 100000 | 1.00E+14 | 1.83 |
| NO | 8 | 100000 | 1.00E+14 | 1.83 |
| NO | 16 | 100000 | 1.00E+14 | 2.32 |
| | | | | |
| NO | 2 | 1000 | 1.00E+16 | 19.03 |
| NO | 4 | 1000 | 1.00E+16 | 9.29 |
| NO | 8 | 1000 | 1.00E+16 | 9.11 |
| NO | 16 | 1000 | 1.00E+16 | 12.81 |
| | | | | |
| NO | 2 | 10000 | 1.00E+16 | 18.82 |
| NO | 4 | 10000 | 1.00E+16 | 9.34 |
| NO | 8 | 10000 | 1.00E+16 | 9.22 |
| NO | 16 | 10000 | 1.00E+16 | 15.24 |
| | | | | |
| NO | 2 | 100000 | 1.00E+16 | 18.62 |
| NO | 4 | 100000 | 1.00E+16 | 9.04 |
| NO | 8 | 100000 | 1.00E+16 | 8.69 |

|  |  |  |  |  |
|---|---|---|---|---|
| NO | 16 | 100000 | 1.00E+16 | 18.08 |

After analysing these results, we come into these conclusions:

1. Pre-assign mode are much faster than dynamical-sending mode(nearly 50%). We believe that there are a lot of time using in the data transferring part.
2. 8-core always has the best computation time in the same group. Since it matches the CPU counts physically. When the number of cores is 16, the time increase rapidly.
3. For the 1e14 number, the best result shows up in the granularity of 10000, but for the 1e16 number, the best result shows up in the granularity of 100000, we believe it depends on the number, the larger the number is, the larger granularity is needed to get the best running time.