Moller Documentation

Release 1.1-dev

ISSP, University of Tokyo

Mar 06, 2024

Contents

Chapter 1

Introduction

1.1 What is moller?

In recent years, the use of machine learning for predicting material properties and designing substances (known as materials informatics) has gained considerable attention. The accuracy of machine learning depends heavily on the preparation of appropriate training data. Therefore, the development of tools and environments for the rapid generation of training data is expected to contribute significantly to the advancement of research in materials informatics.

moller is provided as part of the HTP-Tools package, designed to support high-throughput computations. It is a tool for generating batch job scripts for supercomputers and clusters, allowing parallel execution of programs under a series of computational conditions, such as parameter parallelism. Currently, it supports the supercomputers ohtaka (using the slurm job scheduler) and kugui (using the PBS job scheduler) provided by the Institute for Solid State Physics, University of Tokyo.

1.2 License

The distribution of the program package and the source codes for moller follow GNU General Public License version 3 (GPL v3) or later.

1.3 Contributors

This software was developed by the following contributors.

- ver.1.0.0 (Released on 2024/03/06)
- ver.1.0-beta (Released on 2023/12/28)
  - Developers
    - Kazuyoshi Yoshimi (The Instutite for Solid State Physics, The University of Tokyo)
    - Tatsumi Aoyama (The Instutite for Solid State Physics, The University of Tokyo)
    - Yuichi Motoyama (The Instutite for Solid State Physics, The University of Tokyo)
    - Masahiro Fukuda (The Instutite for Solid State Physics, The University of Tokyo)
    - Kota Ido (The Instutite for Solid State Physics, The University of Tokyo)

        * Tetsuya Fukushima (The National Institute of Advanced Industrial Science and Technology (AIST))
        * Shusuke Kasamatsu (Yamagata University)
        * Takashi Koretsune (Tohoku University)
    - Project Corrdinator
        * Taisuke Ozaki (The Instutite for Solid State Physics, The University of Tokyo)

1.4 Copyright

This software was developed with the support of " Project for advancement of software usability in materials science " of The Institute for Solid State Physics, The University of Tokyo.

1.5 Operating environment

moller was tested on the following platforms:

- Ubuntu Linux + python3

Chapter 2

Installation and basic usage

Prerequisite

Comprehensive calculation utility `moller` included in HTP-tools requires the following programs and libraries:

- Python 3.x
- ruamel.yaml module
- tabulate module
- GNU Parallel (It must be installed on servers or compute nodes on which the job script is executed.)

Official pages

- GitHub repository

Downloads

moller can be downloaded by the following command with git:

```
$ git clone https://github.com/issp-center-dev/Moller.git
```

Installation

Once the source files are obtained, you can install moller by running the following command. The required libraries will also be installed automatically at the same time.

```
$ cd ./Moller
$ python3 -m pip install .
```

The executable files `moller` and `moller_status` will be installed.

Directory structure

```
.
|-- LICENSE
|-- README.md
|-- pyproject.toml
|-- docs/
|   |-- ja/
|   |-- en/
|   |-- tutorial/
```

(continues on next page)

(continued from previous page)

```
|-- src/
|   |-- moller/
|       |-- __init__.py
|       |-- main.py
|       |-- platform/
|       |   |-- __init__.py
|       |   |-- base.py
|       |   |-- base_slurm.py
|       |   |-- base_pbs.py
|       |   |-- base_default.py
|       |   |-- ohtaka.py
|       |   |-- kugui.py
|       |   |-- pbs.py
|       |   |-- default.py
|       |   |-- function.py
|       |   |-- utils.py
|       |-- moller_status.py
|-- sample/
```

Basic usage

`moller` is a tool to generate batch job scripts for supercomputers in which programs are run in parallel for a set of execution conditions using concurrent execution features.

1. Prepare job description file

   First, you need to create a job description file in YAML format that describes the tasks to be executed on supercomputers. The details of the format will be given in File Format section of the manual.

2. Run command

   Run moller program with the job description file, and a batch job script will be generated.

   ```
   $ moller -o job.sh input.yaml
   ```

3. Run batch jobs

   Transfer the generated batch job scripts to the supercomputer. Prepare a directory for each parameter set, and create a list of the directory names in a file `list.dat`. Note that the list contains the relative paths to the directory where the batch job is executed, or the absolute paths.

   Once the list file is ready, you may submit a batch job. The actual command depends on the system.

   - In case of ISSP system B (ohtaka)

     In ohtaka, slurm is used for the job scheduling system. In order to submit a batch job, a command `sbatch` is invoked with the job script as an argument. Parameters can be passed to the script as additional arguments; the name of list file is specified as a parameter.

     ```
     $ sbatch job.sh list.dat
     ```

     If the list file is not specified, `list.dat` is used by default.

- In case of ISSP system C (kugui)

  In kugui, PBS is used for the job scheduling system. In order to submit a batch job, a command `qsub` is invoked with the job script. There is no way to pass parameters to the script, and thus the name of the list file is fixed to `list.dat`.

```
$ qsub job.sh
```

4. Check the status of the calculation

   After the job finishes, you may run the following command

```
$ moller_status input.yaml list.dat
```

   to obtain a report whether the calculation for each parameter set has been completed successfully.

5. Retry/resume job

   In case the job is terminated during the execution, the job may be resumed by submitting the batch job again with the same list file. The yet unexecuted jobs (as well as the unfinished jobs) will be run.

   - In case of ISSP system B (ohtaka)

```
$ sbatch job.sh list.dat
```

   To retry the failed tasks, the batch job is submitted with `--retry` command line option.

```
$ sbatch job.sh --retry list.dat
```

   - In case of ISSP system C (kugui)

   For kugui, to retry the failed tasks, the batch job script should be edited so that `retry=0` is changed to be `retry=1`.

```
$ qsub job.sh
```

   Then, the batch job is submitted as above.

References

[1] O. Tange, GNU Parallel - The command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.

Chapter 3

Tutorial

3.1 Basic usage

The procedure to use the batch job script generator `moller` consists of the following steps: First, a job description file is prepared that defines the tasks to be executed. Next, the program `moller` is to be run with the job description file, and a batch job script is generated. The script is then transferred to the target supercomputer system. A batch job is submitted with the script to perform calculations. In this tutorial, we will explain the steps along a sample in `docs/tutorial/moller`.

3.1.1 Prepare job description file

A job description file describes the content of calculations that are carried out in a batch job. Here, a *batch job* is used for a set of instructions submitted to job schedulers running on supercomputer systems. On the other hand, for the concurrent execution of programs that `moller` handles, we call a series of program executions performed for one set of parameters by a *job*. A job may consist of several contents that we call *tasks*. `moller` organizes job execution so that each task is run in parallel, and the synchronization between the jobs is taken at every start and end of the tasks.

An example of job description file is presented in the following. A job description file is in text-based YAML format. It contains parameters concerning the platform and the batch job, task descriptions, and pre/post-processes.

```
name: testjob
description: Sample task file

platform:
  system:  ohtaka
  queue:   i8cpu
  node:    1
  elapsed: 00:10:00

prologue:
  code: |
    module purge
    module load oneapi_compiler/2023.0.0 openmpi/4.1.5-oneapi-2023.0.0-classic

    ulimit -s unlimited
```

(continues on next page)

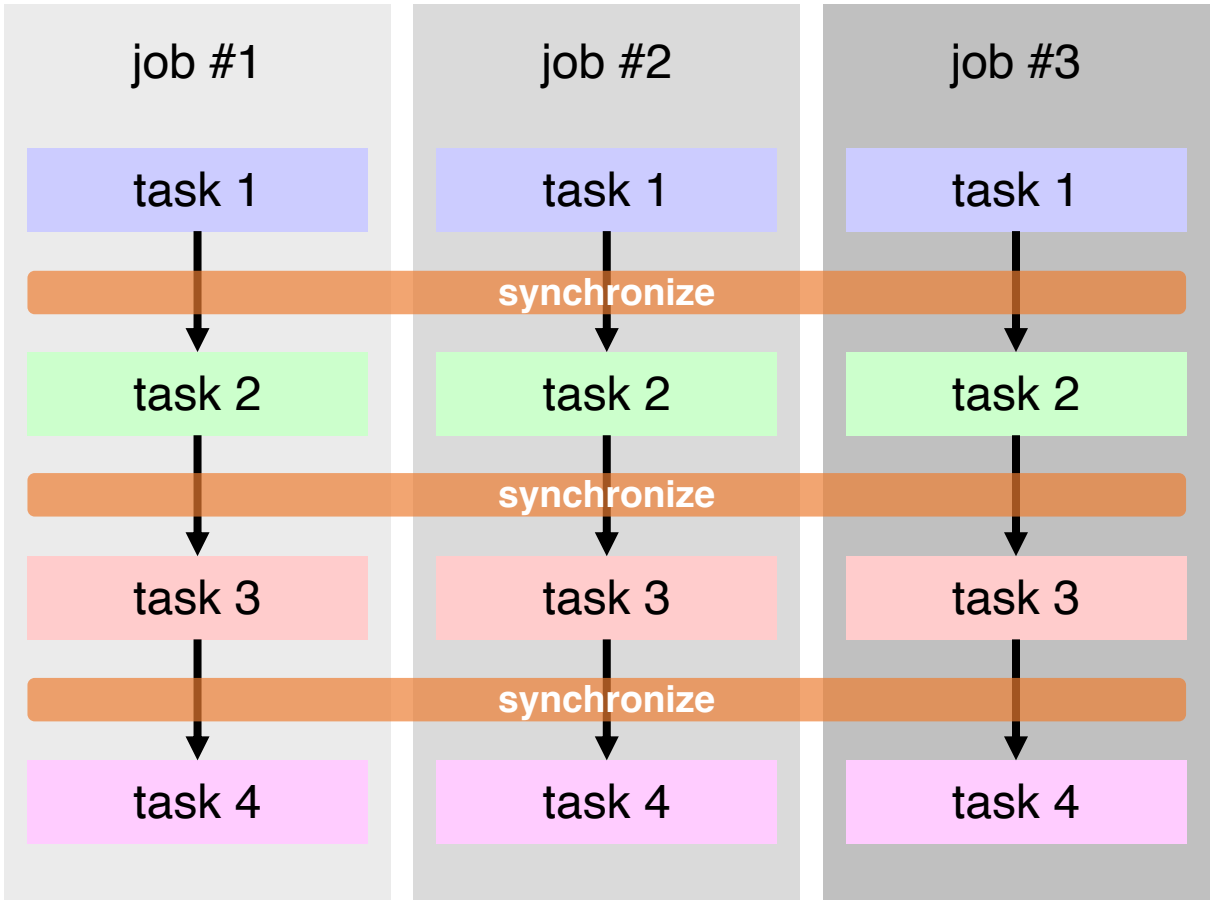


Fig. 3.1: An example of tasks and jobs: Three jobs #1 ⋯ #3 are carried out within a single batch job. Each job corresponds to different set of parameters. A job consists of 4 tasks. Each task is run in parallel among these three jobs.

(continued from previous page)

```
    source /home/issp/materiapps/intel/parallel/parallelvars-20210622-1.sh

jobs:
  start:
    parallel: false
    run: |
      echo "start..."

  hello:
    description: hello world
    node: [1,1]
    run: |
      echo "hello world." > result.txt
      sleep 2

  hello_again:
    description: hello world again
    node: [1,1]
    run: |
      echo "hello world again." >> result.txt
      sleep 2

epilogue:
  code: |
    echo "done."
    date
```

In the platform section, you can specify the type of platform on which to execute. In this case, settings for the System B (ohtaka) are being made.

The prologue section describes the preprocessing of the batch job. It details the common command line to be executed before running the task.

In the jobs section, the content of the task processing is described. The series of tasks to be executed in the job are described in a table format, with the task name as the key and the processing content as the value.

In this example, a task that first outputs " start⋯ " is defined with the task name " start ". Here, it is set to `parallel = false`. In this case, the content of `run` parameter is executed sequentially.

Next, a task that outputs " hello world. " is defined with the task name " hello_world " . Here, since " parallel " is not set, it is treated as `parallel = true`. In this case, parallel processing is performed on a per-job basis. Similarly, next, a task that outputs " hello world again. " is defined with the task name " hello_again ".

Finally, in the epilogue section, the post-processing of the batch job is described. It details the common command line to be executed after running the task.

For more details on the specifications, please refer to the chapter *File Format*.

3.1.2 Generate batch job script

`moller` is to be run with the job description file (`input.yaml`) as an input as follows:

```
$ moller -o job.sh input.yaml
```

A batch job script is generated and written to a file specified by the parameter in the job description file, or the command line option `-o` or `--output`. If both specified, the command line option is used. If neither specified, the result is written to the standard output.

The obtained batch job script is to be transferred to the target system as required. It is noted that the batch job script is prepared for `bash`; users may need to set the shell for job execution to `bash`. (A care should be needed if the login shell is set to csh-type.)

3.1.3 Create list file

A list of jobs is to be created. `moller` is designed so that each job is executed within a directory prepared for the job with the job name. The job list can be created, for example, by the following command:

```
$ /usr/bin/ls -1d * > list.dat
```

In this tutorial, an utility script `make_inputs.sh` is enclosed which generates datasets and a list file.

```
$ bash ./make_inputs.sh
```

By running the above command, a directory `output` and a set of subdirectories `dataset-0001` ⋯ `dataset-0020` that correspond to datasets, and a list file `list.dat` are created.

3.1.4 Run batch job

The batch job is to be submitted to the job scheduler with the batch job script. In this example, the job script and the input parameter files are copied into the `output` directory, and the current directory is changed to ``output` as follows:

```
$ cp job.sh input.yaml output/
$ cd output
```

In ohtaka, slurm is used for the job scheduling system. In order to submit a batch job, a command `sbatch` is invoked with the job script as an argument. Parameters can be passed to the script as additional arguments; the name of list file is specified as a parameter.

```
$ sbatch job.sh list.dat
```

Files named ‘ result.txt ’ will be generated in each directory listed on the list.dat. You can confirm that the ‘ result.txt ’ contains the strings ‘ hello world. ’ and ‘ hello world again. ’ as the job results.

3.1.5 Check status

The status of execution of the tasks are written to log files. A tool named `moller_status` is provided to generate a summary of the status of each job from the log files. It is invoked by the following command in the directory where the batch job is executed:

```
$ moller_status input.yaml list.dat
```

The command takes the job description file `input.yaml` and the list file `list.dat` as arguments. The list file may be omitted; in this case, the information of the jobs are extracted from the log files.

An example of the output is shown below:

```
| job          | hello   | hello_again   |
|--------------|---------|---------------|
| dataset-0001 | o       | o             |
| dataset-0002 | o       | o             |
| dataset-0003 | o       | o             |
| dataset-0004 | o       | o             |
| dataset-0005 | o       | o             |
| dataset-0006 | o       | o             |
| dataset-0007 | o       | o             |
| dataset-0008 | o       | o             |
| dataset-0009 | o       | o             |
| dataset-0010 | o       | o             |
| dataset-0011 | o       | o             |
| dataset-0012 | o       | o             |
| dataset-0013 | o       | o             |
| dataset-0014 | o       | o             |
| dataset-0015 | o       | o             |
| dataset-0016 | o       | o             |
| dataset-0017 | o       | o             |
| dataset-0018 | o       | o             |
| dataset-0019 | o       | o             |
| dataset-0020 | o       | o             |
```

where " o " corresponds to a task that has been completed successfully, " x " corresponds to a failed task, " - " corresponds to a skipped task because the previous task has been terminated with errors, and " . " corresponds to a task yet unexecuted. In the above example, the all tasks have been completed successfully.

3.1.6 Rerun failed tasks

If a task fails, the subsequent tasks within the job will not be executed. The following is an example of job status in which each task fails by 10% change.

```
| job          | task1   | task2   | task3   |
|--------------|---------|---------|---------|
| dataset_0001 | o       | o       | o       |
| dataset_0002 | o       | x       | -       |
| dataset_0003 | x       | -       | -       |
| dataset_0004 | x       | -       | -       |
| dataset_0005 | o       | o       | o       |
| dataset_0006 | o       | o       | o       |
```

(continues on next page)

(continued from previous page)

```
| dataset_0007 | o       | x       | -       |
| dataset_0008 | o       | o       | o       |
| dataset_0009 | o       | o       | x       |
| dataset_0010 | o       | o       | o       |
| dataset_0011 | o       | o       | o       |
| dataset_0012 | o       | o       | o       |
| dataset_0013 | o       | x       | -       |
| dataset_0014 | o       | o       | o       |
| dataset_0015 | o       | o       | o       |
| dataset_0016 | o       | o       | o       |
| dataset_0017 | o       | o       | o       |
| dataset_0018 | o       | o       | o       |
| dataset_0019 | o       | o       | o       |
| dataset_0020 | o       | o       | o       |
```

There, the jobs of dataset_0003 and dataset_0004 failed at task1, and the subsequent task2 and task3 were not executed. The other jobs were successful at task1, and proceeded to task2. In this way, each job is executed independently of other jobs.

Users can rerun the failed tasks by submitting the batch job with the retry option. For SLURM job scheduler (e.g. used in ISSP system B), resubmit the job as follows:

```
$ sbatch job.sh --retry list.dat
```

For PBS job scheduler (e.g. used in ISSP system C), edit the job script so that the line `retry=0` is replaced by `retry=1`, and resubmit the job.

```
| job          | task1   | task2   | task3   |
|--------------|---------|---------|---------|
| dataset_0001 | o       | o       | o       |
| dataset_0002 | o       | o       | x       |
| dataset_0003 | o       | x       | -       |
| dataset_0004 | o       | o       | o       |
| dataset_0005 | o       | o       | o       |
| dataset_0006 | o       | o       | o       |
| dataset_0007 | o       | o       | o       |
| dataset_0008 | o       | o       | o       |
| dataset_0009 | o       | o       | o       |
| dataset_0010 | o       | o       | o       |
| dataset_0011 | o       | o       | o       |
| dataset_0012 | o       | o       | o       |
| dataset_0013 | o       | o       | o       |
| dataset_0014 | o       | o       | o       |
| dataset_0015 | o       | o       | o       |
| dataset_0016 | o       | o       | o       |
| dataset_0017 | o       | o       | o       |
| dataset_0018 | o       | o       | o       |
| dataset_0019 | o       | o       | o       |
| dataset_0020 | o       | o       | o       |
```

The tasks that have failed will be executed in the second run. In the above example, the task1 for dataset_0003 was successful, but the task2 failed. For dataset_0004, task1, task2, and task3 were successfully executed. For the jobs of datasets whose tasks have already finished successfully, the second run will not do anything.

N.B. the list file must not be modified on the rerun. The jobs are managed according to the order of entries in the list file, and therefore, if the order is changed, the jobs will not be executed properly.

3.2 Example for *moller* calculation with HPhi

3.2.1 What 's this sample?

This is an example of `moller` with HPhi, which is an open-source software package for performing the exact diagonalization method for quantum many-body problems. In this example, we will calculate the system size dependence of the excitation gap Δ of the $S = 1/2$ (`2S_1` directory) and $S = 1$ (`2S_2`) antiferromagnetic Heisenberg chain under the periodic boundary condition. By using `moller`, calculations with different system sizes are performed in parallel. This is corresponding to section 1.4 of HPhi 's official tutorial.

3.2.2 Preparation

Make sure that `moller` (HTP-tools) package and `HPhi` are installed. In this tutorial, the calculation will be performed using the supercomputer system `ohtaka` at ISSP.

3.2.3 How to run

1. Prepare dataset

   Run the script `make_inputs.sh` enclosed within this package.

   ```
   $ bash ./make_inputs.sh
   ```

   Working directories `L_8`, `L_10`, ⋯, `L_24` (up to `L_18` for `2S_2`)) will be generated. A list of the directories is written to a file `list.dat`. Additionally, a shell script, `extract_gap.sh`, to gather energy gaps from working directories is generated.

2. Generate job script using `moller`

   Generate a job script from the job description file using `moller`, and store the script as a file named `job.sh`.

   ```
   $ moller -o job.sh input.yaml
   ```

3. Run batch job

   Submit a batch job with the job list as an argument.

   ```
   $ sbatch job.sh list.dat
   ```

4. Check status

   The status of task execution will be summarized by `moller_status` program.

   ```
   $ moller_status input.yaml list.dat
   ```

5. Gather results

   Once the calculation finishes, gather energy gaps from jobs as

   ```
   $ bash extract_gap.sh
   ```

This script writes pairs of the length L and the gap Δ into a text file, `gap.dat`.

To visualize the results, a Gnuplot file `gap.plt` is available. In this file, the obtained gap data are fitted by the expected curves,

$$\Delta(L; S = 1/2) = \Delta_\infty + A/L \tag{3.1}$$

and

$$\Delta(L; S = 1) = \Delta_\infty + B \exp(-CL). \tag{3.2}$$

The result is plotted as follows:

```
$ gnuplot --persist gap.plt
```

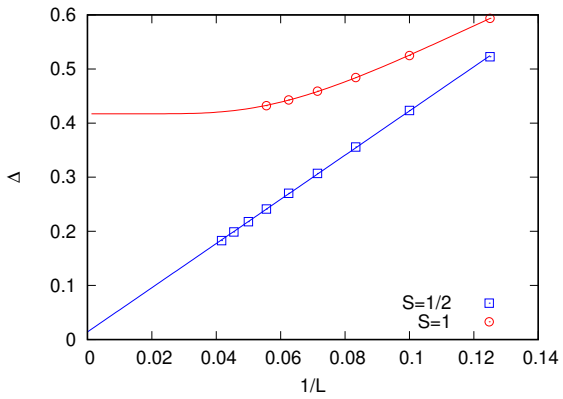


Fig. 3.2: Finite size effect of spin gap

Note that the logarithmic correction causes the spin gap for $S = 1/2$ to remain finite. On the other hand, for $S = 1$, the extrapolated value $\Delta_\infty = 0.417(1)$ is consistent with the previous results, e.g., $\Delta_\infty = 0.41048(6)$ by QMC (Todo and Kato, PRL **87**, 047203 (2001)).

3.3 Example for *moller* calculation with DSQSS

3.3.1 What 's this sample?

This is an example of `moller` with DSQSS, which is an open-source software package for performing the path-integral Monte Calro method for quantum many-body problem. In this example, we will calculate the temperature dependence of the magnetic susceptibilities χ of the $S = 1/2$ ($M = 1$ in the terms of DSQSS) and $S = 1$ ($M = 2$) antiferromagnetic Heisenberg chain under the periodic boundary condition with several length. By using `moller`, calculations with different parameters (M, L, T) are performed in parallel.

This example is corresponding to one of the official tutorials.

3.3.2 Preparation

Make sure that `moller` (HTP-tools) package and `DSQSS` are installed. In this tutorial, the calculation will be performed using the supercomputer system `ohtaka` at ISSP.

3.3.3 How to run

1. Prepare dataset

   Run the script `make_inputs.sh` enclosed within this package.

   ```
   $ bash ./make_inputs.sh
   ```

   This make an `output` directory (if already exists, first removed then make again). Under `output`, working directories for each parameter like `L_8__M_1__T_1.0` will be generated. A list of the directories is written to a file `list.dat`.

2. Generate job script using `moller`

   Generate a job script from the job description file using `moller`, and store the script as a file named `job.sh`.

   ```
   $ moller -o job.sh input.yaml
   ```

   Then, copy `job.sh` in the `output` directory, and change directory to `output`.

3. Run batch job

   Submit a batch job with the job list as an argument.

   ```
   $ sbatch job.sh list.dat
   ```

4. Check status

   The status of task execution will be summarized by `moller_status` program.

   ```
   $ moller_status input.yaml list.dat
   ```

5. Gather results

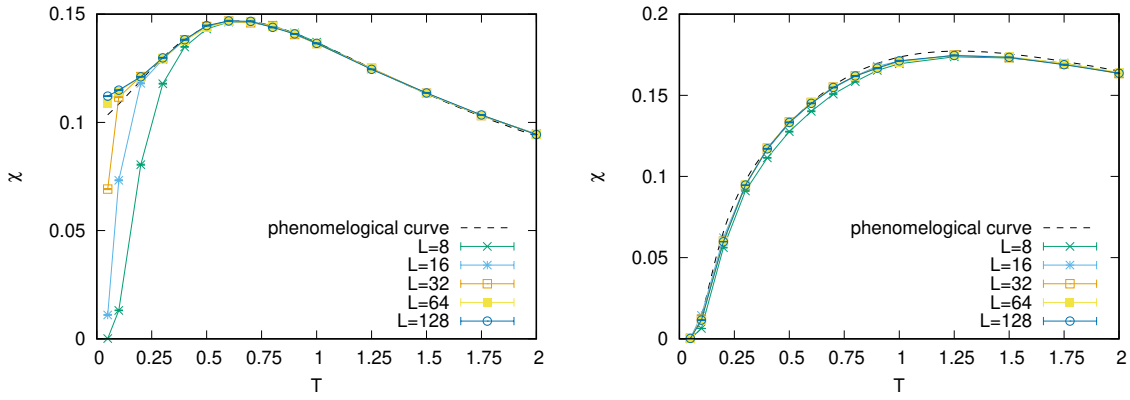
   After calculation finishes, gather result by

   ```
   $ python3 ../extract_result.py list.dat
   ```

   This script writes results into a text file `result.dat` which has 5 columns, M, L, T, mean of χ, and stderr of χ.

   To visualize the results, GNUPLOT files `plot_M1.plt` and `plot_M2.plt` are available.

   ```
   $ gnuplot --persist plot_M1.plt
   $ gnuplot --persist plot_M2.plt
   ```



The main different between $S = 1/2$ and $S = 1$ AFH chains is whether the excitation gap vanishes ($S = 1/2$) or remains ($S = 1$). Reflecting this, the magnetic susceptibility in the very low temperature region remains finite ($S = 1/2$) or vanishes ($S = 1$). Note that for the $S = 1/2$ case, the finite size effect opens the spin gap and therefore the magnetic susceptibility of small chains drops.

Chapter 4

Command reference

4.1 moller

Generate a batch job script for comprehensive calculation

SYNOPSIS:

```
moller [-o job_script] input_yaml
```

DESCRIPTION:

This program reads a job description file specified by input_yaml, and generates a batch job script. It takes the following command line options.

- `-o`, `--output job_script`

  specifies output file name. This option supersedes the output_file parameter in the job description file. If no output file is specified, the result is written to the standard output.

- `-h`

  displays help and exits.

4.2 moller_status

Reports the status of comprehensive calculation jobs

SYNOPSIS:

```
moller_status [-h] [--text|--csv|--html] [--ok|--failed|--skipped|--collapsed|--
↪yet] [-o output_file] input_yaml [list_file]
```

DESCRIPTION:

This program summarizes the status of tasks in jobs that are executed through the job scripts generated by moller, and outputs a report. The tasks are obtained from the job description file specified by `input_yaml`. The list of jobs is read from the file specified by `list_file`. If it is not provided, the job list is extracted from the log files. The format of the output is specified by a command line option. The default is the text format. The output file is specified by the `-o` or `--output` option. If it is not specified, the output is written to the standard output.

- output formats

  specifies the format of the output by one of the following options. If more than one option are specified, the program terminates with error. The default is the text format.

  - `--text` displays in text format.
  - `--csv` displays in CSV (comma-separated values) format.
  - `--html` displays in HTML format.

- `input_yaml`

  specifies the job description file for `moller`.

- `list_file`

  specifies the file that contains list of job directories. If this file is not specified, the list will be obtained from the logfile of the batch job `stat_{task}.dat`.

- `-o`, `--output output_file`

  specifies the output file name. If it is omitted, the result is written to the standard output.

- filter options

  specifies the status of jobs to be displayed by one of the following options. All jobs are displayed by default.

  - `--ok` displays only jobs whose tasks are all completed successfully.
  - `--failed` displays jobs, any of whose tasks are failed with errors, skipped, or not performed.
  - `--skipped` displays jobs, any of whose tasks are skipped.
  - `--yet` displays jobs, any of whose tasks are not yet performed.
  - `--collapsed` displays jobs, any of whose tasks are failed with errors.
  - `--all` displays all jobs. (default)

- `-h`

  displays help and exits.

FILES:

When the programs are executed concurrently using the job script generated by `moller`, the status of the tasks are written in log files `stat_{task}.dat`. `moller_status` reads these log files and makes a summary.

Chapter 5

File format

5.1 Job description file

A job description file contains configurations to generate a batch job script by `moller`. It is prepared in text-based YAML format. This file consists of the following parts:

1. General settings: specifies job names and output files.
2. platform section: specifies the system on which batch jobs are executed, and the settings for the batch jobs.
3. prologue and epilogue sections: specifies initial settings and finalization within the batch job.
4. jobs section: specifies tasks to be carried out in the betch job script.

5.1.1 General settings

`name`

: specifies the name of the batch job. If it is not given, the job name is left unspecified. (Usually the name of the job script is used as the job name.)

`description`

: provides the description of the batch job. It is regarded as comments.

`output_file`

: specifies the output file name. When the output file is given by a command-line option, the command-line parameter is used. When none of them is specified, the result is written to the standard output.

5.1.2 platform

`system`

specifies the target system. At present, either `ohtaka` or `kugui` is accepted.

`queue`

specifies the name of batch queue. The actual value depends on the target system.

`node`

specifies the number of nodes to be used. It is given by an integer specifying the number of nodes, or a list of integers specifying `[` number of nodes, number of cores per node `]`. The accepted range of parameters depends on the system and queue settings. (The number of cores is accepted for kugui and default systems; otherwise it is ignored.)

`core`

specifies the number of cores per node be used. The accepted range of parameters depends on the system and queue settings. If both the second parameter of `node` and `core` are specified, the value in `core` is used. (This parameter is accepted for kugui and default target systems.)

`elapsed`

specifies the elapsed time of the batch job in HH:MM:SS format.

`options`

specifies other batch job options. It is given as a list of options or as a multiple-line string with options in each line. The heading directives (e.g. `#PBS` or `#SBATCH`) are not included. The examples are given as follows.

- an example of SLURM job script in the string format:

```
options: |
  --mail-type=BEGIN,END,FAIL
  --mail-user=user@sample.com
  --requeue
```

- an example of PBS job script in the list format:

```
options:
  - -m bea
  - -M user@sample.com
  - -r y
```

5.1.3 prologue, epilogue

`prologue` section specifies the commands to be run prior to executing the tasks. It is used, for example, to set environment variables of libraries and paths. `epilogue` section specifies the commands to be run after all tasks have been completed.

`code`

specifies the content of the commands in the form of shell script. It is embedded in the batch job script, and executed within the batch job.

5.1.4 jobs

`jobs` section specifies a sequence of tasks in a table format, with the task names as keys and the contents as values.

key

name of task

value

a table that consists of the following items:

`description`

provides the description of the task. It is regarded as comments.

`node`

specifies the degree of parallelism in one of the following formats.

- `[` number of processes, number of threads per process `]`
- `[` number of nodes, number of processes, number of threads per process `]`
- number of nodes

When the number of nodes is specified, the specified number of nodes are exclusively assigned to a job. Otherwise, if the required number of cores for a job is smaller than the number of cores in a node, more than one job may be allocated in a single node. If a job uses more than one node, the required number of nodes are exclusively assigned.

`parallel`

This parameter is set to `true` if the tasks of different jobs are executed in parallel. It is set to `false` if they are executed sequentially. The default value is `true`.

`run`

The content of the task is described in the form of shell script. The executions of MPI parallel programs or MPI/OpenMPI hybrid parallel programs are specified by

```
srun prog [arg1, ...]
```

where, in addition to the keyword `srun`, `mpirun` or `mpiexec` is accepted. In the resulting job script, they are replaced by the command (e.g. `srun` or `mpirun`) and the degree of parallelism specified by `node` parameter.

5.2 List file

This file contains a list of jobs. It is a text file with a job name in a line (The name of the directory is associated with the name of the job).

`moller` assumes that a directory is assigned to each job, and the tasks of the job are executed within the directory. These directories are supposed to be located in the directory where the batch job is submitted.

Chapter 6

Extension guide

N.B. The content of this section may vary depending on the version of *moller*.

6.1 Bulk job execution by *moller*

A bulk job execution means that a set of small tasks are executed in parallel within a single batch job submitted to a large batch queue. It is schematically shown as follows, in which N tasks are launched as background processes and executed in parallel, and a `wait` statement is invoked to wait for all tasks to be completed.

```
task param_1 &
task param_2 &
     ...
task param_N &
wait
```

To manage the bulk job, it is required to distribute nodes and cores allocated to the batch job over the tasks param_1 ⋯ param_N so that they are executed on distinct nodes and cores. It is also needed to arrange task execution where at most N tasks are run simultaneously according to the allocated resources.

Hereafter a job script generated by *moller* will be denoted as a moller script. In a moller script, the concurrent execution and control of tasks are managed by GNU parallel [1]. It takes a list holding the items param_1 ⋯ param_N and runs commands for each items in parallel. An example is given as follows, where list.dat contains param_1 ⋯ param_N in each line.

```
cat list.dat | parallel -j N task
```

The number of concurrent tasks is determined at runtime from the number of nodes and cores obtained from the execution environment and the degree of parallelism (number of nodes, processes, and threads specified by node parameter).

The way to assign tasks to nodes and cores varies according to the job scheduler. For SLURM job scheduler variants, the concurrent calls of `srun` command within the batch job are appropriately assigned to the nodes and cores by exploiting the option of exclusive resource usage. The explicit option may depend on the platform.

On the other hand, for PBS job scheduler variants that do not have such features, the distribution of nodes and cores to tasks has to be handled within the moller script. The nodes and cores allocated to a batch job are divided into *slots*, and the slots are assigned to the concurrent tasks. The division is determined from the allocated nodes and cores and the degree of parallelism of the task, and kept in a form of table variables. Within a task, the programs are executed on the assigned hosts and cores (optionally pinned to the program) through the options to mpirun (or mpiexec) and the environment variables. This feature depends on the MPI implementation.

Reference

[1] O. Tange, GNU Parallel - The command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.

6.2 How *moller* works

6.2.1 Structure of moller script

moller reads the input YAML file and generates a job script for bulk execution. The structure of the generated script is described as follows.

1. Header

   This part contains the options to the job scheduler. The content of the platform section is formatted according to the type of job scheduler. This feature depends on platforms.

2. Prologue

   This part corresponds to the prologue section of the input file. The content of the `code` block is written as-is.

3. Function definitions

   This part contains the definitions of functions and variables used within the moller script. The description of the functions will be given in the next section. This feature depends on platforms.

4. Processing Command-line options

   The SLURM variants accept additional arguments to the job submission command (sbatch) that are passed to the job script as a command-line options. The name of the list file and/or the options such as the retry feature can be processed.

   For the PBS variants, these command-line arguments are ignored, and therefore the name of the list file is fixed to `list.dat` by default, and the retry feature may be enabled by modifying the script with `retry` set to 1.

5. Description of tasks

   This part contains the description of tasks specified in the jobs section of the input file. When more than one task is given, the following procedure is applied to each task.

   When parallel = false, the content of the `run` block is written as-is.

   When parallel = true (default), a function is created by the name task_{task name} that contains the pre-processing for concurrent execution and the content of the `run` block. The keywords for the parallel execution (`srun`, `mpiexec`, or `mpirun`) are substituted by the platform-dependent command. The definition of the task function is followed by the concurrent execution command.

6. Epilogue

   This part corresponds to the epilogue section of the input file. The content of the `code` block is written as-is.

6.2.2 Brief description of moller script functions

The main functions of the moller script is briefly described below.

- `run_parallel`

  This function performs concurrent execution of task functions. It takes the degree of parallelism, the task function, and the status file as arguments. Within the function, it calls `_find_multiplicity` to find the number of tasks that can be run simultaneously, and invokes GNU parallel to run tasks concurrently. The task function is actually wrapped by the `_run_parallel_task` function to deal with the nested call of GNU parallel.

  The platform-dependence is separated out by the functions `_find_multiplicity` and `_setup_run_parallel`.

- `_find_multiplicity`

  This function determines the number of tasks that can be simultaneously executed on the allocated resources (nodes and cores) taking account of the degree of parallelism of the task. For the PBS variants, the compute nodes and the cores are divided into slots, and the slots are kept as table variables. The information obtained at the batch job execution is summarized as follows.

  - For SLURM variants,

    The number of allocated nodes (`_nnodes`)

    `SLURM_NNODES`

    The number of allocated cores (`_ncores`)

    `SLURM_CPUS_ON_NODE`

  - For PBS variants,

    The allocated nodes (`_nodes[]`)

    The list of unique compute nodes is obtained from the file given by `PBS_NODEFILE`.

    The number of allocated nodes (`_nnodes`)

    The number of entries of `_nodes[]`.

    The number of allocated cores

    Searched from below (in order of examination)

    * `NCPUS` (for PBS Professional)
    * `OMP_NUM_THREADS`
    * `core` parameter of platform section (written in the script as a variable `moller_core`.)
    * `ncpus` or `ppn` parameter in the header.

- `_setup_run_parallel`

  This function is called from the `run_parallel` function to supplement some procedures before running GNU parallel. For PBS variants, the slot tables are exported so that the task functions can refer to. For SLURM variants, there is nothing to do.

The structure of the task function is described as follows.

- A task function is created by a name `task_{task name}`.
- The arguments of the task function are 1) the degree of parallelism (the number of nodes, processes, and threads), 2) the execution directory (that corresponds to the entry of list file), 3) the slot ID assigned by GNU parallel.

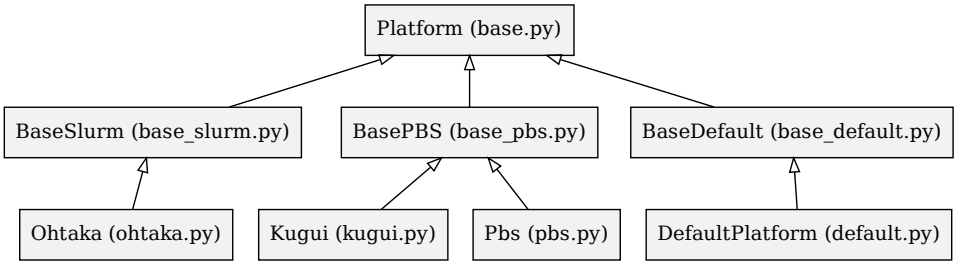
- The platform-dependent `_setup_taskenv` function is called to set up execution environment. For PBS variants, the compute node and the cores are obtained from the slot table based on the slot ID. For SLURM variants, there is nothing to do.
- The `_is_ready` function is called to check if the preceding task has been completed successfully. If it is true, the remaining part of the function is executed. Otherwise, the task is terminated with the status -1.
- The content of the `code` block is written. The keywords for parallel calculation (`srun`, `mpiexec`, or `mpirun`) are substituted by the command provided for the platform.

6.3 How to extend *moller* for other systems

The latest version of *moller* provides profiles for ISSP supercomputer systems, ohtaka and kugui. An extension guide to use *moller* in other systems is described in the following.

6.3.1 Class structure

The platform-dependent parts of *moller* are placed in the directory `platform/`. Their class structure is depicted below.



A factory is provided to select a system in the input file. A class is imported in `platform/__init__.py` and registered to the factory by `register_platform(system_name, class_name)`, and then it becomes available in the system parameter of the platform section in the input YAML file.

6.3.2 SLURM job scheduler variants

For the SLURM job scheduler variants, the system-specific settings should be applied to a derived class of BaseSlurm class. The string that substitute the keywords for the parallel execution of programs is given by the return value of `parallel_command()` method. It corresponds to the `srun` command with the options for the exclusive use of resources. See ohtaka.py for an example.

6.3.3 PBS job scheduler variants

For the PBS job scheduler variants (PBS Professional, OpenPBS, Torque, and others), the system-specific settings should be applied to a derived class of BasePBS class.

There are two ways of specifying the number of nodes for a batch job in the PBS variants. PBS Professional takes the form of select=N:ncpus=n, while Torque and others take the form of node=N:ppn=n. The BasePBS class has a parameter `self.pbs_use_old_format` that is set to `True` for the latter type.

The number of cores per compute node can be specified by node parameter of the input file, while the default value may be set for a known system. In kugui.py, the number of cores per node is set to 128 by default.

6.3.4 Customizing features

When further customization is required, the methods of the base class may be overridden in the derived classes. The list of relevant methods is given below.

- `setup`

  This method extracts parameters of the platform section.

- `parallel_command`

  This method returns a string that is used to substitute the keywords for parallel execution of programs (`srun`, `mpiexec`, `mpirun`).

- `generate_header`

  This method generates the header part of the job script that contains options to the job scheduler.

- `generate_function`

  This method generates functions that are used within the moller script. It calls the following methods to generate function body and variable definitions.

  - `generate_variable`
  - `generate_function_body`

  The definitions of the functions are provided as embedded strings within the class.

6.3.5 Porting to new type of job scheduler

The platform-dependent parts of the moller scripts are the calculation of task multiplicity, the resource distribution over tasks, and the command string of parallel calculation. The internal functions need to be developed with the following information on the platform:

- how to acquire the allocated nodes and cores from the environment at the execution of batch jobs.
- how to launch parallel calculation (e.g. mpiexec command) and how to assign the nodes and cores to the command.

To find which environment variables are set within the batch jobs, it may be useful to call `printenv` command in the job script.

6.3.6 Trouble shooting

When the variable `_debug` in the moller script is set to 1, the debug outputs are printed during the execution of the batch jobs. If the job does not work well, it is recommended that the debug option is turned on and the output is examined to check if the internal parameters are appropriately defined.