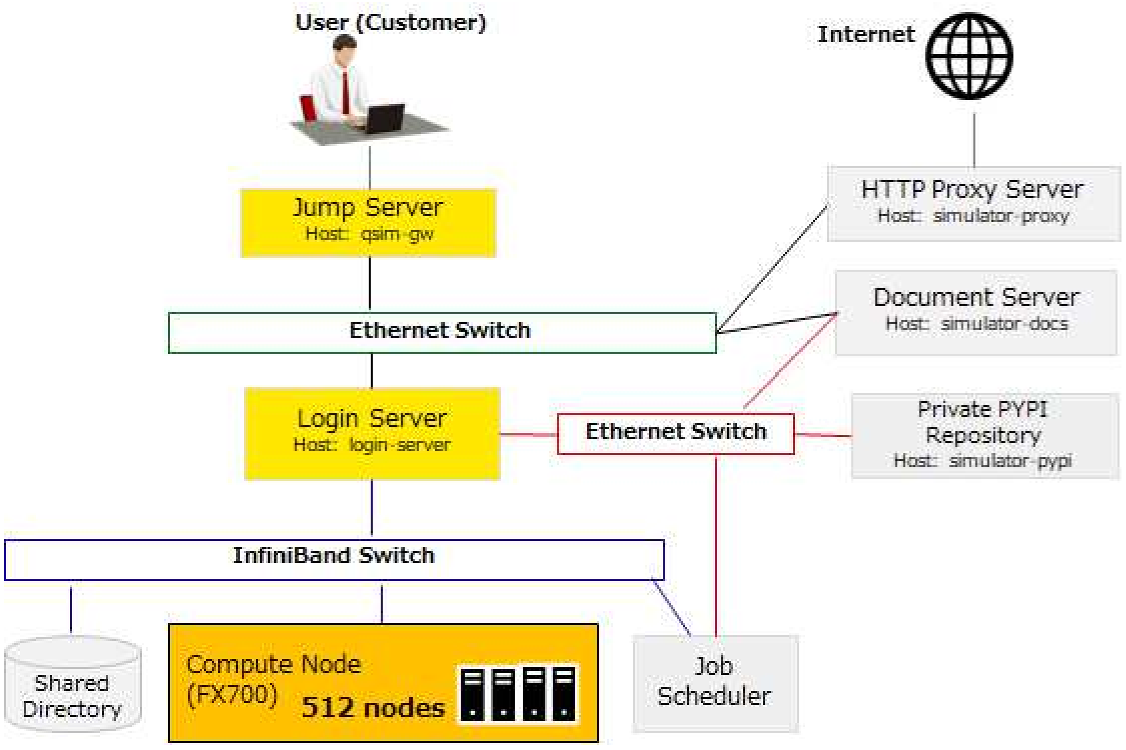
Fujitsu Quantum Simulator2

If you are using it for the first time, please see the System Usage section first.
If you want to make quantum programs using the Qulacs API, see the mpiQulacs section.
If you want to use the Qiskit API (mpiQulacs as a backend), see the section qiskit-qulacs.

1. System Overview

1.1. System Configutation

The Quantum Simulator System consists of compute nodes for simulation (consisting of 512 FX 700), a jump server for login, a login server for job submission, a shared directory, a job scheduler, a private PyPI repository, a document server, and a proxy server for Internet access.



The user logs in to the login server through the jump server. The compute nodes are available from the login server through the job scheduler Slurm. The compute nodes are connected by InfiniBand, and the quantum circuit is simulated while performing parallel calculation using MPI (Message Passing Interface).

The information for each server/node that makes up the quantum simulator system is as follows.

The role and IP address/hostname of each server

Name	IP / Host	Explanation
Jump Server	106.184.62.10 Port number: Notification from Fujitsu	Server in a DMZ
Login Server	login-server	User environment for utilizing computation Accessible from the jump server
Compute Node	N/A	Cluster system for executing quantum program
Shared Directory	N/A	Directory for placing files shared between compute nodes
Private PyPI Repository	simulator-pypi	Private Python package repository for Fujitsu software packages (e.g. mpiQulacs)
Document Server	simulator-docs	Server that contains updated information and API manuals
HTTP Proxy Server	simulator-proxy	Proxy server used to access Internet from the compute node

Specifications for each server/node

Server name	Item	
Login Server	OS	Rocky Linux 8.6
	CPU	Intel(R) Xeon(R) Gold 6348 (2.6GHz) x 1core
	Memory	32GB
Compute Node	OS	Rocky Linux 8.5
	CPU	Fujitsu A64FX (2.0GHz) x 48cores
	Memory	32GB
Shared Directory	File System	GlusterFS
	Storage Capacity	3TB/group

Note

Use the jump server only as a relay to access the login server. Do not create or execute programs on the jump server.

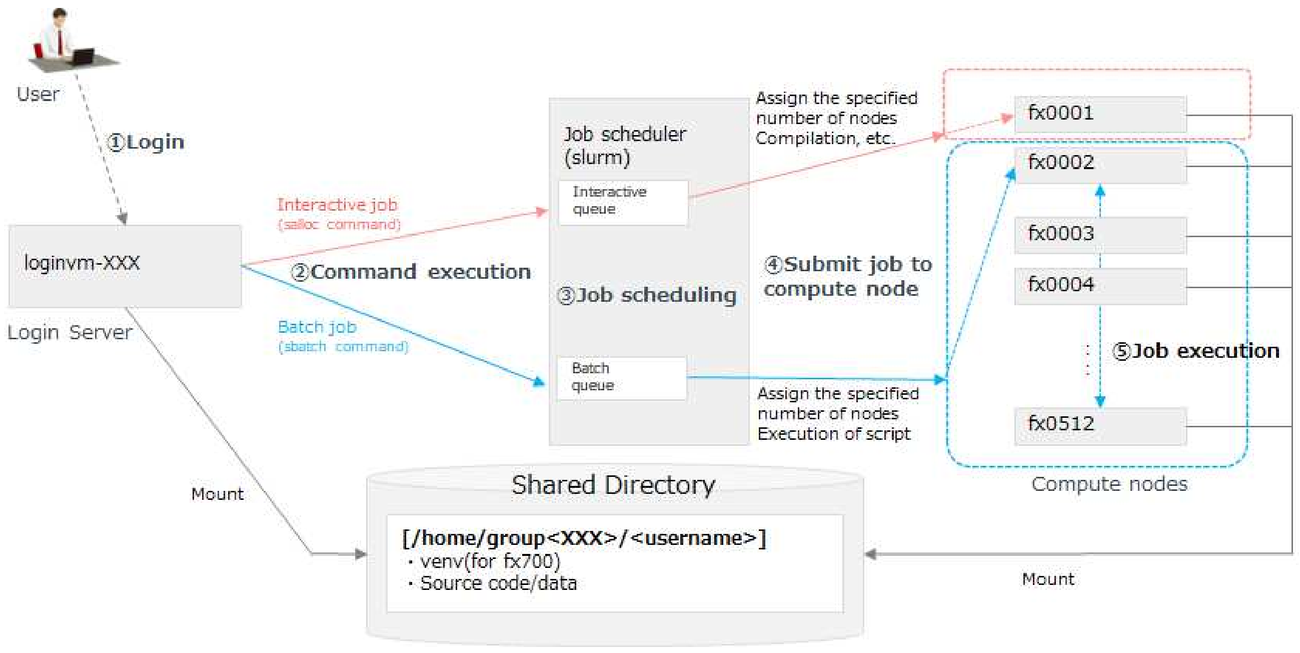
Note

The initial storage capacity of the shared directory is 3 TB, but if you still run out of space after deleting unnecessary files, contact your Fujitsu representative.

1.2. Job Scheduler

The job scheduler is a system that divides multiple shared computer resources so that multiple users can execute jobs and manages them so that each user can occupy the compute node for a certain period of time. A job is a set of shell processing units that, when a command or script is submitted as a job, is executed when the requested resources are ready. This system uses Slurm Workload Manger (Slurm) as the job scheduler.

The following figure shows the flow from the login server to the calculation node via the job scheduler Slurm, and the connection to the shared directory.

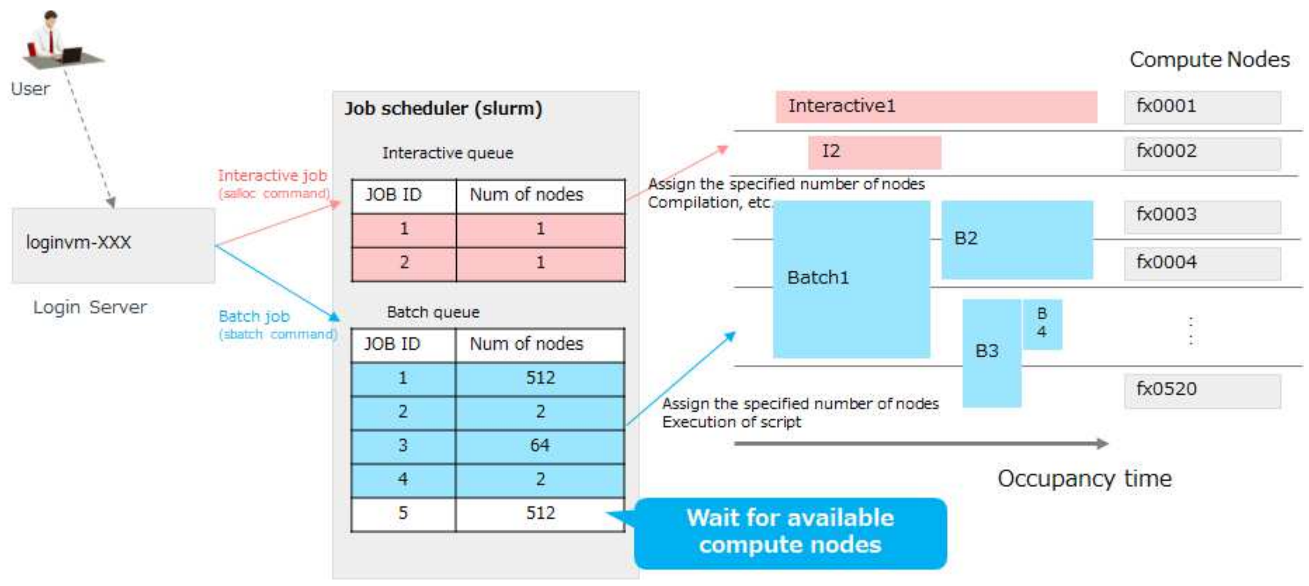


① Log into the login server (loginvm- < three digits above group ID >)
② Run the Slurm command on the login server
③ Job scheduler schedules jobs based on the command and number of nodes required
④ Job scheduler submits jobs to compute nodes
⑤ Job runs on compute nodes

Note

The home directory of each user’s login server mounts the shared directory. The same directory is mounted on all compute nodes.

The following is an image of ③ job scheduler scheduling.



The job scheduler uses queues to manage jobs submitted by users.

Interactive queue is a queue for logging in to compute nodes and executing commands interactively. It is mainly used to create an MPI execution environment on compute nodes, compile scripts, and check the normal operation before batch execution. It is assumed to allocate one or two nodes at a time in the Interactive queue.
Batch queue is a queue for submitting scripts from the login server to the compute nodes.

We will inform you separately about the total number of nodes available for Interactive and Batch queues.

Note

The number of nodes used and the maximum number of qubits are as follows, and quantum circuits up to 39 qubits can be executed when 512 nodes are used.

Number of nodes used	512	256	128	64	32	16	8	4	2	1
Maximum qubit number	39	38	37	36	35	34	33	32	31	30

2. Login

Here's how to log in to the quantum simulator system.
The operation example in Windows is described, but you can log in in the same way for other OS such as Linux.

2.1. Operation for First Login

2.1.1. Creating an ssh key

You open Windows PowerShell and create an ed25519 format key in `ssh-keygen` .
An example of the command is:

```
PS C:\Users\fujitsu> ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (C:\Users\fujitsu/.ssh/id_ed25519): (Leave blank and enter.
Can be changed if necessary)
Enter passphrase (empty for no passphrase): (any secure passphrase)
Enter same passphrase again: (any secure passphrase)
Your identification has been saved in C:\Users\fujitsu/.ssh/id_ed25519.
Your public key has been saved in C:\Users\fujitsu/.ssh/id_ed25519.pub.
The key fingerprint is:
SHA256:cg/n/kQ20baFH30QEcxno3+UrpNOmZLVKkyfvUDn5p0 fujitsu@pc
The key's randomart image is:
+--[ED25519 256]--+
|             o=+ |
|             .o++|
|            . =+*|
|             +.=+|
|      . S ..+o++.|
|       o =oo=.Oo.|
|          o=.Oo+.|
|         . .+++ +|
|          ..o..E.|
+----[SHA256]-----+
PS C:\Users\fujitsu>
```

In the following, the operation example is described on the assumption that the private/public key is created in `C:\Users\fujitsu/.ssh/id_ed25519(.pub)` .

2.1.2. Sending user account names and key information

Send the desired user account name and public key (id_ed25519.pub) to the Fujitsu representative. we will prefix the user account name with “gXXX -“(where XXX is a three-digit number) that identifies you. The representative will inform you of the user account name after adding the character string.

2.1.3. Configuring ssh config

`C:/Users/fujitsu/.ssh/config` shows the login settings for the quantum simulator system.

description example

```
# ssh settings to the jump server
Host qsim-gw
    HostName 106.184.62.10
    Port (Port number reported by Fujitsu)
    User (User account name reported by Fujitsu)
    StrictHostKeyChecking no
    IdentityFile ~/.ssh/id_ed25519

# ssh settings to the login server
Host qsim
    HostName login-server
    User (User account name reported by Fujitsu)
    IdentityFile ~/.ssh/id_ed25519
    StrictHostKeyChecking no
    ProxyCommand ssh -W %h:%p qsim-gw
```

2.2. Login Method

You can log in to the login server at `ssh qsim` .

```
PS C:\Users\fujitsu> ssh qsim
Enter passphrase for key 'C:\Users\fujitsu/.ssh/id_ed25519':(passphrase for ssh keys)
Enter passphrase for key 'C:\Users\fujitsu/.ssh/id_ed25519':(passphrase for ssh keys)
Last login: Wed Nov  9 13:41:31 2022 from 172.19.102.2
[(User account name reported by Fujitsu)@loginvm-XXX ~]$
```

If `[(User account name reported by Fujitsu)@loginvm-XXX ~]$` is displayed, you have logged in correctly.
If you don’t want to enter a passphrase, use ssh-agent or something else.

Note

If “*WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!*” occurs during ssh login, follow the error message to remove the entry in .ssh/known_hosts, and then try ssh again.
you would run a command similar to the following:

ssh-keygen -R 106.184.62.10
ssh-keygen -R login-server
ssh qsim

3. Job Execution Method

Here is an example of how to run the quantum simulator software mpiQulacs on a quantum simulator system.
(For more information about mpiQulacs, see the mpiQulacs documentation in the sidebar.)
The flow of job execution is as follows:

1. Log in to the compute nodes from the login server by Interactive job and build the environment
2. Return to the login server and create a shell script
3. Submit a job into the Batch queue from the login server (running a shell script)
4. Check the execution results

3.1. Environmental Construction

3.1.1. Installing Python packages

You log in to the compute nodes (FX 700) from the login server using an Interactive job.
The following example sets aside one node, one hour, for interactive execution of a compute node belonging to the Interactive queue.
See Job assignment (salloc) for other option.

Note

Since the number of nodes is limited, please `exit` when you are finished.

```
[username@loginvm-XXX ~]$ salloc -N 1 -p Interactive --time=1:00:00
```

You create an execution environment on the compute node. The following example creates a Python virtual environment in the `example` directory and installs the packages needed to use mpiQulacs.

```
[username@fx-XX-XX-XX ~]$ mkdir example
[username@fx-XX-XX-XX ~]$ cd example
[username@fx-XX-XX-XX ~]$ python3.8 -m venv qenv
[username@fx-XX-XX-XX ~]$ source ./qenv/bin/activate
(qenv) [username@fx-XX-XX-XX ~]$ pip install --upgrade pip wheel
(qenv) [username@fx-XX-XX-XX ~]$ pip install mpi4py  # Required to use mpiQulacs
(qenv) [username@fx-XX-XX-XX ~]$ pip install mpiQulacs
(qenv)  [username@fx-XX-XX-XX ~]$ exit
```

3.2. Creating scripts

3.2.1. Creating a job script file for MPI execution

mpiQulacs performs parallel processing using MPI.
(See Overview of MPI Parallel Computing in Python for a description of MPI.)

This is where you create the necessary script file to run MPI.
Create a script file referring to the following `~/example/job.sh` .

~/example/job.sh

```
#!/usr/bin/env bash

# Usage:
#     mpirun -n <num_ranks> -npernode 1 job.sh <path/to/venv> <command> <command arguments>
#
# Example:
#     (python): mpirun -n 2 -npernode 1 job.sh ~/example/venv python ./sample.py
#     (pytest): mpirun -n 2 -npernode 1 job.sh ~/example/venv pytest

# Setting the Environment Variables required to use MPI in the quantum simulator system
export UCX_IB_MLX5_DEVX=no
export OMP_PROC_BIND=TRUE

# Number of threads used for OpenMP parallelization
# Due to the OpenMP multi-threading behavior, some libraries (such as scipy) can introduce
small calculation errors, which can lead to inconsistent calculation results between processes
during MPI execution. When using mpiQulacs, set it to 1.
export OMP_NUM_THREADS=1

# Number of threads used by mpiQulacs (if not set, the value of OMP_NUM_THREADS is used)
export QULACS_NUM_THREADS=48

#
source $1/bin/activate
shift

### Workaround for the glibc bug (https://bugzilla.redhat.com/show_bug.cgi?id=1722181)).
if [ -z "${LD_PRELOAD}" ]; then
    export LD_PRELOAD=/lib64/libgomp.so.1
else
    export LD_PRELOAD=/lib64/libgomp.so.1:$LD_PRELOAD
fi

#
LSIZE=${OMPI_COMM_WORLD_LOCAL_SIZE}
LRANK=${OMPI_COMM_WORLD_LOCAL_RANK}
COM=$1
shift

if [ $LSIZE -eq 1 ]; then
    numactl -m 0-3 -N 0-3 ${COM} "$@"
elif [ $LSIZE -eq 4 ]; then
    numactl -N ${LRANK} -m ${LRANK} ${COM} "$@"
else
    ${COM} "$@"
fi
```

After writing the code, execute the following command to grant execute permission.

```
$ chmod u+x ~/example/job.sh
```

3.2.2. Python script example of quantum program

Use mpiQulacs to write quantum programs in Python.

Basically, you can write a program in the same way as Qulacs . See Usage for details.

~/example/example.py

```
from qulacs import QuantumCircuit, QuantumState
from mpi4py import MPI  # import is required for MPI execution even if MPI classes are not used

seed=1234

# Get process rank
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# creation of quantum states
num_qubits = 10
state = QuantumState(num_qubits, use_multi_cpu=True)

# Define quantum circuits and update quantum states
circuit = QuantumCircuit(num_qubits)
circuit.add_H_gate(0)
for i in range(num_qubits - 1):
    circuit.add_CNOT_gate(i, i + 1)
circuit.update_quantum_state(state)

# Sampling values from quantum states
sampled_values = state.sampling(20, seed)

# Show results only for a process with rank 0 (node01)
if rank == 0:
    print(sampled_values)
```

Note

To run MPI, you must write an import statement “from mpi4py import MPI” before running the quantum circuit.

Note

The program (example.py) runs simultaneously on multiple compute nodes. Simply calling the print method will print the message to standard output multiple times for as many compute nodes. Therefore, the print statement is limited to node01. See Overview of MPI Parallel Computing in Python for details.

3.2.3. Creating a batch file

~/example/sim.job

```
#!/bin/bash

#SBATCH -p Batch # Specify Batch queue
#SBATCH -o test-%j # Output file name
#SBATCH -N 64 # Number of nodes allocated
#SBATCH -t 06:00:00 # Limit on the job execution time

mpirun -npernode 4 job.sh ~/example/qenv python example.py # The -n option is not required when executing with the number of nodes allocated above.
#   mpirun -n 1 -npernode 1 job.sh ~/example/qenv python example.py # To run on more than the allocated number of nodes, give the -n option; in this example, one node.
```

You create a file for job execution.
Fujitsu tells you how many nodes are available. If you have any questions, please contact the person in charge.
This example creates a Python program and runs it in parallel on two compute nodes.
-npernode specifies the number of processes per node.
Following *job.sh* is the path to the Python virtual environment, followed by the command and then the command arguments.
In the example above, *~/example/qenv* specifies the path to the Python virtual environment created in Environmental Construction .
It then specifies the command `python` followed by the command argument `example.py` .
This causes `python example.py` to run in parallel on 64 compute nodes via job.sh.

Note

The optimal number of parallel circuits depends on the size and configuration of the circuit. Please consult with Fujitsu Researcher and adjust the parallel number accordingly.

Warning

The default job execution time limit is 6 hours.
Although it is not necessary to specify the job execution time limit, setting the execution time limit may cause the job to execute earlier.
(By default, a job will not run if there is scheduled maintenance within 6 hours, but it may run if the execution time is short.
In addition, jobs with a small number of nodes and a short job execution time are executed first if there are any unused nodes before the job is executed, even if there are jobs waiting.)
However, note that if the job does not end within the specified execution time, it will end even if the job is still running.

3.3. Job Submission

You run the batch file you created.

```
[username@loginvm-XXX example]$ sbatch sim.job
```

3.4. Check Output Results

When the job runs, the test-<JOBID> file is created and the results are output. If the JOBID is 10000, the results are output to the file `test-10000` .

~/example/test-10000

```
[username@loginvm-XXX example]$ cat test-10000
[1023, 0, 1023, 1023, 0, 1023, 1023, 1023, 0, 1023, 1023, 0, 0, 1023, 0, 1023, 0, 0, 0, 1023]
```

4. Slurm Command

4.1. Confirmation of nodes and partions (sinfo)

View information about queues (partitions).

Command format

```
$ sinfo
```

Output

Item	Explanation
PARTITION	Name of a partition
AVAIL	Partition state
TIMELIMIT	Maximum time limit for any user job. *infinite* is showed to identify a partition without a job time limit.
NODES	Number of nodes allocated to the partition.
STATE	State of nodes. The suffix “*” identifies nodes that are presently not responding.
NODELIST	Names of nodes

Example of command execution

```
[UserY@loginvm-XXX ~]$ sinfo
PARTITION   AVAIL  TIMELIMIT  NODES  STATE NODELIST
Interactive    up   infinite      1  down* fx-01-12-06
Interactive    up   infinite      2  alloc fx-01-12-[00-01]
Batch*         up   infinite      5   idle fx-01-12-[02-05,07]
 .
 .
 .
```

4.2. Job assignment (salloc)

Get a set of nodes to which you want to assign the job and execute the command. Release the allocation after the command ends.

Command format

```
$ salloc <option> <command>
```

Options

Option	Explanation
-J <*job name*>	Specify a name for the job allocation
-p <*partition name*>	Submit a job to a specified queue (partition)
-N <*number of node*>	Specify number of nodes
-n <*number of process*>	Specify number of processes
–time=<*time*>	Set a limit on the total execution time of job assignments

4.3. Job execution (srun)

Run a parallel job on cluster managed by Slurm.

Command format

```
$ srun <option> <execute job>
```

Options

Option	Explanation
-J <*job name*>	Specify a name for the job allocation
-p <*partition name*>	Submit a job to a specified queue (partition)
-N <*number of node*>	Specify number of nodes
-n <*number of process*>	Specify number of processes
-o ./out_%j.log	Output standard output to a file called “out_*jobID*.log”
-e ./err_%j.log	Output standard error output to a file called “err_*jobID*.log”

Option	Explanation
–time=<*time*>	Set a limit on the total execution time of job assignments
–pty <*SHELL*>	Run interactively
–preserve-env	Pass the current values of environment variables SLURM_JOB_NODES and SLURM_NTSASKS to the executable

4.4. Job execution (sbatch)

Send the batch script to Slurm.

Command format

```
$ sbatch <option> <job script>
```

Options

Option	Explanation
-J <*job name*>	Specify a name for the job allocation
-p <*partition name*>	Submit a job to a specified queue (partition)
-N <*number of node*>	Specify number of nodes
-n <*number of process*>	Specify number of processes
-o ./out_%j.log	Output standard output to a file called “out_*jobID*.log”
-e ./err_%j.log	Output standard error output to a file called “err_*jobID*.log”
–time=<*time*>	Set a limit on the total execution time of job assignments

4.5. Check running jobs (squeue)

Display a list of currently running jobs and job information. Jobs executed by other users are not displayed.

Command format

```
$ squeue
```

Output

Item	Explanation
JOBID	Job ID assigned to the job

Item	Explanation
PARTITION	Name of the queue (partition) that submitted the job
NAME	Displays the job name. Displays the command string if unspecified.
USER	Displays the user who executes the job submission request
ST	Displays the status of the job. See table below for status list.
TIME	Job execution time
NODES	Number of nodes used for job execution
NODELIST(REASON)	List of host names on which jobs are executed

Job status description

State	Explanation
CA(CANCELLED)	State cancelled by user/administrator
CD(COMPLETED)	Terminate all processes on all nodes
CF(CONFIGUREING)	Wait for resources to become available after they are allocated
CG(COMPLETING)	Process of the termination procedure
F(FAILED)	Terminated with a non-zero exit code or other failure
NF(NODE_FAIL)	Terminated because one of the assigned nodes failed
PD(PENDING)	Pending for resource allocation
PR(PREEMPTED)	Job aborted and terminated
R(RUNNING)	Currently Running
S(SUSPENDED)	Suspend execution to wait for resource allocation
TO(TIMEOUT)	Terminated due to timeout

Example of command execution

```
[UserY@loginvm-XXX ~]$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
10114     Batch    sleep    UserY  R       1:41      1 fx-01-10-02
```

4.6. Check running jobs including other users (squeues)

Display a list of the node usage information for your and other users' currently running jobs on the login server. The information might be out of date because it is updated every 10 seconds. This is different from the slurm standard command.

Command format

```
$ squeues
```

Output

Item	Explanation
JOBID	Job ID assigned to the job
NODES	Number of nodes used for job execution
END_TIME	Job end time
TIME_LEFT	Time to end job
NODELIST	List of nodes used for job execution
ST	Displays the status of the job. See the squeue command for status list.
SCHEDNODES	The node that will be used if the job is pending. If running, show (null).

Example of command execution

```
[UserY@loginvm-XXX ~]$ squeues
JOBID NODES END_TIME TIME_LEFT NODELIST ST SCHEDNODES
10115 1 2023-03-01T0:30:00 2:00:00 fx-01-12-00 R (null)
```

4.7. Abort job (scancel)

Abort the currently running job by specifying the job ID. You can also cancel multiple jobs at once by including job IDs in a series separated by spaces.

Command format

```
scancel <JOBID> <JOBID>
```

4.8. Check the jobs that have completed execution (sacct)

Display a list of jobs that have completed execution. Jobs executed by other users are not displayed.

Command format

```
sacct <option>
```

Options

Option	Explanation
-j <job ID>	Specify job ID
-o <item, item, …>	Specify output items separated by commas. See the table below for output items.
-e	Show items that can be specified with the -o option
-S, –starttime	Display information after the specified date and time. If not specified, the current day's 0: 00 is set.
-E, –endtime	Display information before the specified date and time

Output items

Item	Explanation
User	Execution user of the job submission request (job)
JobID	Job ID assigned to the job
Partition	Name of the queue (partition) that submitted the job
NNodes	Number of nodes used for job execution
Submit	Date and time the job was submitted
Start	Date and time when job execution started
End	Date and time when job execution completed
Elapsed	Job execution time
State	Job status

Example of command execution

```
[UserY@loginvm-XXX ~]$ sacct
JobID           JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
10122              bash Interacti+                    48  COMPLETED      0:0
10122.0            bash                               48  CANCELLED     0:53
10124              bash Interacti+                    48  COMPLETED      0:0
10124.0            bash                               48  CANCELLED     0:53
10125           sim.job      Batch                  3072     FAILED      4:0
```

5. Document Server

Documentation on quantum simulator systems, mpiQulacs and other software packages is available on the documentation server.
You can browse by using ssh portforward.

For example, to set the port forward on `~/.ssh/config`, add a line for LocalForward as follows:

```
# ssh settings to the login server
Host qsim
    HostName login-server
    User (User account name reported by Fujitsu)
    IdentityFile ~/.ssh/id_ed25519
    StrictHostKeyChecking no
    LocalForward 10000 172.19.102.201:80
    ProxyCommand ssh -W %h:%p qsim-gw
```

After running ssh, you open http://localhost:10000/ in your browser to view the documentation.
The document server provides updated information on the quantum simulator system.
Please check when using the quantum simulator system.

6. Python

6.1. Available Python Version

Python 3.8.X is installed on this system (`python3.8` command)。
If you want to use a different version of Python, please use pyenv.

6.2. Install of pyenv

The following is an example of installing and setting pyenv in Bash.

Installing pyenv

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
cd ~/.pyenv && src/configure && make -C src
```

Setting up the shell environment (If you are using Bash)

```
echo 'if [ -z $PYENV_ROOT ]; then
  export PYENV_ROOT="$HOME/.pyenv"
  export PATH="$PYENV_ROOT/bin:$PATH"
  eval "$(pyenv init --path)"
  eval "$(pyenv init -)"
fi' >> ~/.bashrc
```

Note

The above settings should be in .bashrc, not .bash_profile. The .bash_profile setting is not read by mpirun.

After doing the above, please login again to read the shell settings.
Then you install and configure the Python version you want to use.

Installing and configuring Python

```
# install will take a few minutes to do the source build
pyenv install 3.9.10
# Specifies the Python version to use under the current directory
pyenv local 3.9.10
python -V  # -> Python 3.9.10
```

6.3. Overview of MPI Parallel Computing in Python

When you run a program with mpirun, the program you want to run runs on multiple processes (multiple compute nodes) at the same time. Each process is assigned a sequential number, called a rank, starting with 0.
Without MPI control in the program, each rank (each process) will run the same program and behave the same way. By obtaining the rank number in the program and describing different processing depending on the rank number, it is possible to have different behavior for each rank.

mpiQulacs attempts to speed up quantum circuit simulation by distributing and calculating quantum state vectors and gate operation information required for quantum circuit evaluation to each rank.

6.3.1. Notes on programming

6.3.1.1. Determinism of program behavior

Basically, the program needs to act decisively, unless you intentionally want each rank (each compute node) to behave differently.

The same program code is executed for each rank unless parallel control using MPI is intentional. When a program performs a non-deterministic operation, for example, a different quantum circuit is generated in each rank, and the calculation processing of the quantum state vector starts in that state. If this happens, data cannot be synchronized and exchanged correctly between ranks, leading to program crashes and abnormal results.

For example, if the rotation angle of the RY gate is specified nondeterministically in the following code, the rotation angle in the quantum circuit existing on the rank 0 memory and the rotation angle in the quantum circuit on the rank 1 are shifted. Since the contents of quantum circuits assumed by each rank are different, inconsistencies occur in the calculation of state vectors arranged across ranks, leading to program crashes and abnormal results.

```
import numpy as np
import random
from qulacs import QuantumCircuit, QuantumState
from mpi4py import MPI  # required for using mpiQulacs

circ = QuantumCircuit(3)

# N.G. (non-deterministic)
# angle = random.random() * np.pi

# O.K. (deterministic)
random.seed(1234)
angle = random.random() * np.pi

circ.add_RY_gate(0, angle)

state = QuantumState(3, use_multi_cpu=True)
circ.update_quantum_state(state)
samples = state.sampling(1024, 2022)  # n_shots=1024 and seed=2022
```

If you use random numbers, you should explicitly set the seed value to make the program behave decisively. In addition, you should configure the various Python packages, if any, to make them behave decisively.

6.3.1.2. Data output

When a character string is output to the standard output by the print method or the like in a program, the output results of all ranks are displayed in the calculation node where mpirun is executed (rank 0). For example, if you run mpirun with 16 compute nodes, you will see the same message 16 times. Therefore, it is desirable to limit the rank of the message output to the standard output. (Please refer to the program example above. (~/example/example.py)) Alternatively, you can use option `--output-filename <directory name>` of the mpirun command to save the output to a separate file for each rank. (See *man mpirun* for details)

In the quantum simulator system, the home directory is shared between compute nodes. If you write to a file without limiting its rank, multiple processes will simultaneously write to files with the same path. When writing files to the home directory, be sure to limit the rank.

Warning

Special attention should be paid to the determinism of the program and the writing of files to the home directory as described above.

6.3.1.3. Speed up by parallel processing

There are 48 CPU cores and 32 GB of RAM available per node . Quantum circuit simulation processing by mpiQulacs is performed in parallel using multiple nodes and multiple CPUs.

In order to speed up the entire quantum application, it is important to parallelize the non-quantum circuit simulation part (the part of the user written code that calls the API of mpiQulacs) appropriately to the extent possible so that the computational resources (CPU core) can be utilized as much as possible. Keep in mind that a lot of single-threaded or single-process processing outside of quantum circuit simulation can slow down the overall quantum application.

7. Jupyter

This chapter describes how to operate Jupyter Notebook (or JupyterLab) on compute nodes, and how to run Python programs that run in MPI parallel on it. This example uses the environment `~/example` created in Job Execution Method . Please change */home/groupxxx/user1* depending on the name of the logged in user account.

Note

The path to the home directory of the logged-in user can be found with the following command: `[user1@loginvm-XXX ~]$ pwd`

/home/groupxxx/user1

7.1. Preparation

You use an Interactive job to enter the compute nodes. The number of compute nodes is limited, so please `exit` when you are finished.

```
[user1@loginvm-XXX ~]$  salloc -N 1 -p Interactive --time=1:00:00
[user1@fx-XX-XX-XX ~]$
```

7.1.1. Installing required packages

Install the following packages.

```
[user1@fx-XX-XX-XX ~]$ cd example
[user1@fx-XX-XX-XX example]$ source qenv/bin/activate
(qenv) [user1@fx-XX-XX-XX example]$ pip install mpi4py jupyter ipyparallel jupyter_server

# If you use JupyterLab, run `pip install jupyterlab`.
# Also, in the following discussion, replace the Jupyter Notebook with the JupyterLab.
```

7.1.2. IPython Parallel settings

You use IPython Parallel (ipyparallel) to run Python programs that run in MPI parallel on the Jupyter Notebook.

7.1.2.1. Preparing NUMA Control file

Create the following file `/home/groupxxx/user1/local/bin/numa-launcher`.
(Note: This file allows the IPython engine to perform the `numactl` related operations described in *job.sh* in Job Execution Method.)

/home/groupxxx/user1/local/bin/numa-launcher

```
#!/usr/bin/bash

LSIZE=${OMPI_COMM_WORLD_LOCAL_SIZE}
LRANK=${OMPI_COMM_WORLD_LOCAL_RANK}
COM=$1
shift

if [ $LSIZE -eq 1 ]; then
    numactl -m 0-3 -N 0-3 ${COM} "$@"
elif [ $LSIZE -eq 4 ]; then
    numactl -N ${LRANK} -m ${LRANK} ${COM} "$@"
else
    ${COM} "$@"
fi
```

You grant execute permission.

```
$ chmod +x /home/groupxxx/user1/local/bin/numa-launcher
```

7.1.2.2. Preparing IPython configuration files

Generate a template for the configuration profile with the following command. '*example-profile*' is the name of the profile to create. Please specify any name.

```
(qenv) [user1@fx-XX-XX-XX example]$ ipython profile create example-profile --parallel
```

After execution, open `~/.ipython/profile_example-profile/ipcluster_config.py` and specify the following.

```
c.MPILauncher.mpi_args = ["-x", "UCX_IB_MLX5_DEVX=no", "-x", "OMP_PROC_BIND=TRUE", "-x",
"OMP_NUM_THREADS=1", "-x", "QULACS_NUM_THREADS=48", "-x", "LD_PRELOAD=/lib64/libgomp.so.1"]
c.MPILauncher.mpi_cmd = ['mpirun']

import sys
c.MPIEngineSetLauncher.engine_cmd = ['/home/groupxxx/user1/local/bin/numa-launcher',
sys.executable, '-m', 'ipyparallel.engine']
c.MPIEngineSetLauncher.engine_args = ['--profile', 'example-profile']
```

In this example, the environment variables described in job.sh created in Job Execution Method are set. **Please change the path to numa-launcher and the profile name accordingly.**

> **Note**
>
> If you need to set environment variables to run your Python program, add them with the `-x` option to the `c.MPILauncher.mpi_args` entry above.

7.1.2.3. Configure port forwarding

You set port forwarding from your local environment to the compute node(fx-XX-XX-XX) used by the Interactive job. You open a new terminal on your local environment and use an ssh port forwarding command as follows. You can bring up the Jupyter Notebook screen in your local web browser.

```
ssh -N -L 8888:fx-XX-XX-XX:8888 qsim
```

7.2. Execution

You will run IPython on the backend and run it from the Jupyter Notebook. The following describes how to execute the command.

7.2.1. Starting IPython Parallel

You create the following batch file. Please refer to the batch file created in Creating a batch file. The -n option specifies the parallel number (number of compute nodes).

~/example/ipython.job

```
#!/usr/bin/bash

#SBATCH -p Batch
#SBATCH -o test-ipy-%j # Output file name
#SBATCH -N 2 # Number of nodes allocated

ipcluster start -n 2 --engines=MPI --ip='*' --profile=example-profile
```

You submit the job. Open the output file and verify that startup is complete.

```
(qenv) [user1@fx-XX-XX-XX example]$ sbatch ipython.job
(qenv) [user1@fx-XX-XX-XX example]$
(qenv) [user1@fx-XX-XX-XX example]$ tail test-ipy-XXX
2022-03-29 11:27:24.656 [IPClusterStart] Starting ipcluster with [daemonize=False]
2022-03-29 11:27:25.759 [IPClusterStart] Starting 2 engines with <class
'ipyparallel.cluster.launcher.MPIEngineSetLauncher'>
2022-03-29 11:27:55.803 [IPClusterStart] Engines appear to have started successfully
```

If *Engines appear to have started successfully* appears, the boot is successful. It takes some time to start up.
When you finish the work, finish the submitted job referring to the following.

```
(qenv) [user1@fx-XX-XX-XX example]$ squeue # Get a list of running jobs
           JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           10696     Batch ipython.     user1 R        0:18      2 fx-01-10-[04-05]
           10694 Interacti interact     user1 R        0:24      2 fx-01-12-[00-01]
(qenv) [user1@fx-XX-XX-XX example]$ scancel 10696 # ipython.jobのJOBID
```

7.2.2. Starting Jupyter Notebook

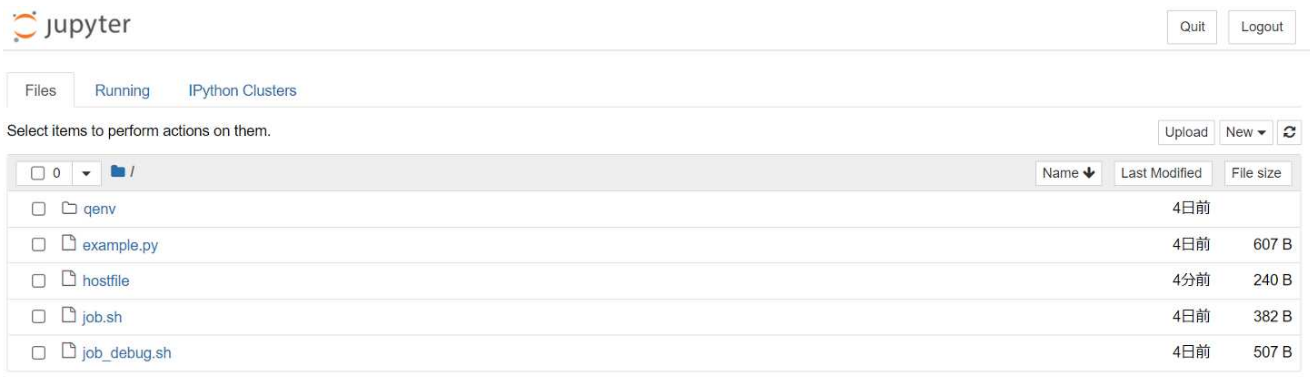
You start Jupyter Notebook after ipcluster has finished booting.

```
(qenv) [user1@fx-XX-XX-XX example]$
(qenv) [user1@fx-XX-XX-XX example]$ jupyter notebook --ip='*'
# If you want to use JupyterLab, run jupyter lab instead of jupyter notebook.
```

When Jupyter Notebook is started, the following message is displayed.

```
To access the notebook, open this file in a browser:
    file:///home/user3/.local/share/jupyter/runtime/nbserver-40491-open.html
Or copy and paste one of these URLs:
    http://localhost:8888/?token=2c9932ac7b7b813b925e460035903e21ed48394a8614348a
 or http://127.0.0.1:8888/?token=2c9932ac7b7b813b925e460035903e21ed48394a8614348a
```

If you open the URL http://127.0.0.1:8888/?token shown here in a browser, the Jupyter Notebook will be displayed.

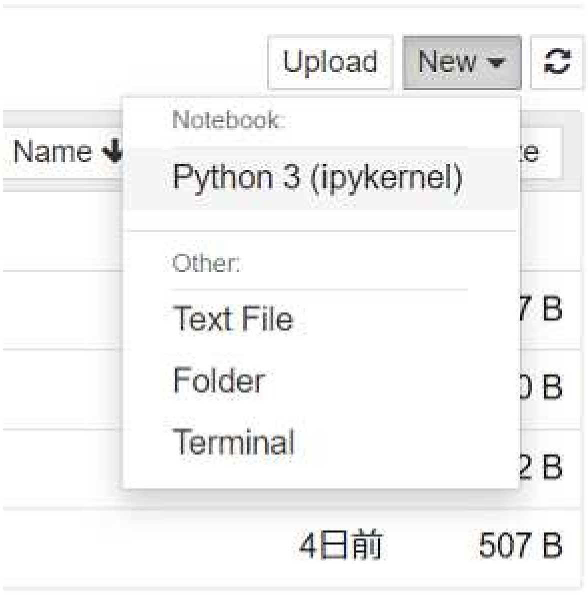


Select items to perform actions on them.

qenv	4日前	
example.py	4日前	607 B
hostfile	4分前	240 B
job.sh	4日前	382 B
job_debug.sh	4日前	507 B

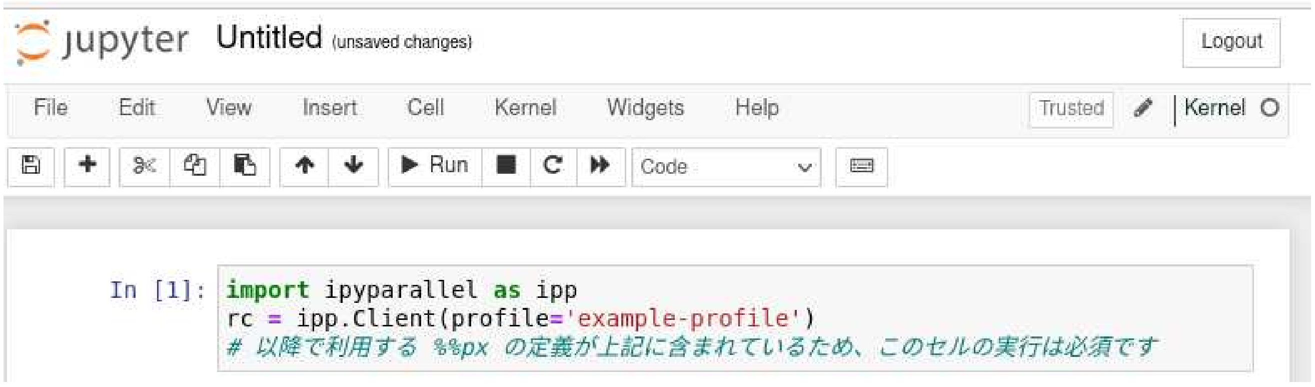
7.2.3. Working with Jupyter Notebook

Create a new Python3 file.



Create a cell in the open notebook and run it as follows.

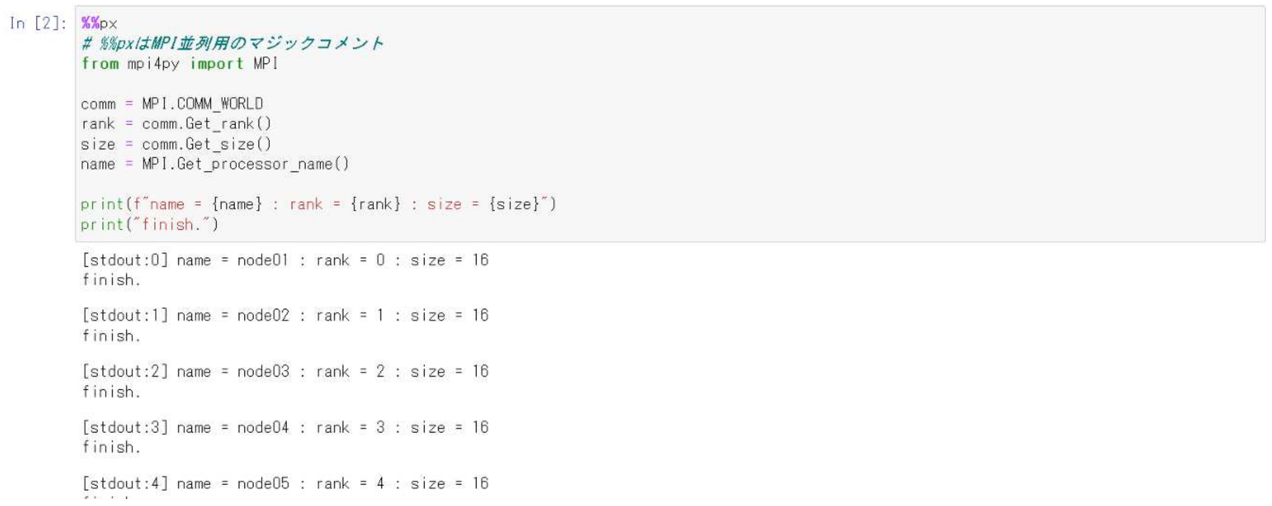
```
import ipyparallel as ipp
rc = ipp.Client(profile='example-profile') #Profile name set in "Preparing IPython
configuration files"
# Execution of this cell is mandatory because it contains the definition of %%px which will be
used after.
```



```
In [1]: import ipyparallel as ipp
        rc = ipp.Client(profile='example-profile')
        # 以降で利用する %%px の定義が上記に含まれているため、このセルの実行は必須です
```

Create a new cell and execute it as follows.
This confirms that MPI is running parallel processing.

```python
%%px
# %%px is a magic comment for MPI parallelism
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
name = MPI.Get_processor_name()

print(f"name = {name} : rank = {rank} : size = {size}")
print("finish.")
```

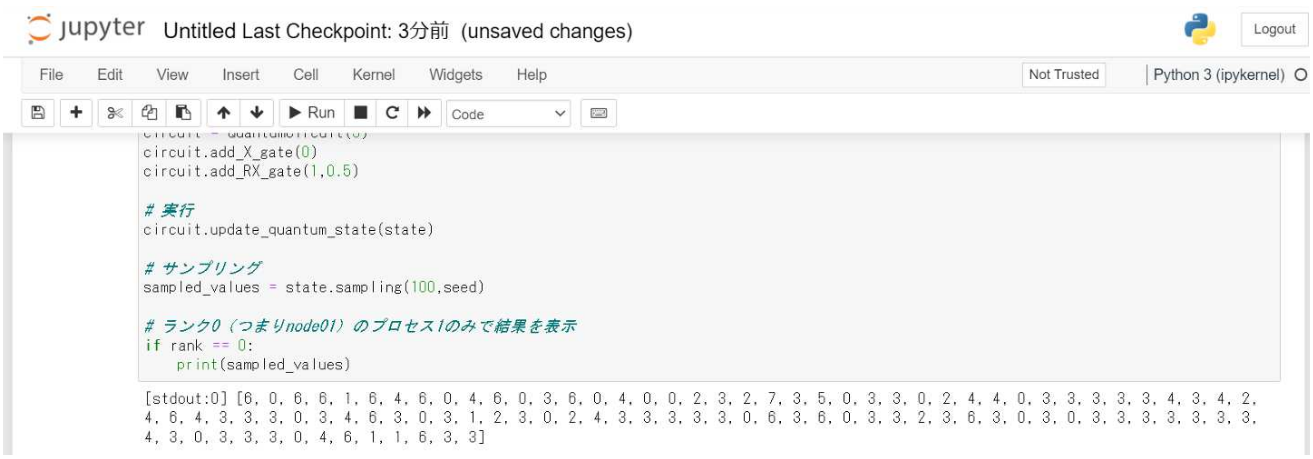


```
In [2]: %%px
        # %%pxはMPI並列用のマジックコメント
        from mpi4py import MPI

        comm = MPI.COMM_WORLD
        rank = comm.Get_rank()
        size = comm.Get_size()
        name = MPI.Get_processor_name()

        print(f"name = {name} : rank = {rank} : size = {size}")
        print("finish.")

        [stdout:0] name = node01 : rank = 0 : size = 16
        finish.

        [stdout:1] name = node02 : rank = 1 : size = 16
        finish.

        [stdout:2] name = node03 : rank = 2 : size = 16
        finish.

        [stdout:3] name = node04 : rank = 3 : size = 16
        finish.

        [stdout:4] name = node05 : rank = 4 : size = 16
```

Whenever you write a program that uses MPI for parallel processing, write %%px .
(Please refer to the ipyparallel documentation for more details on %%px.)

You can run a quantum program using mpiQulacs by writing the following in the cell.

```
%%px
from qulacs import QuantumCircuit, QuantumState
from mpi4py import MPI

# Seed of random number
seed=1234

# Get process rank
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Prepare quantum states
state = QuantumState(3, use_multi_cpu=True)
state.set_Haar_random_state(seed)

# Prepare uantum circuits
circuit = QuantumCircuit(3)
circuit.add_X_gate(0)
circuit.add_RX_gate(1, 0.5)

# Execution
circuit.update_quantum_state(state)

# Sampling
sampled_values = state.sampling(100, seed)

# Show results only at rank 0 (node01)
if rank == 0:
    print(sampled_values)
```


jupyter
Untitled Last Checkpoint: 3分前 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
circuit.add_X_gate(0)
circuit.add_RX_gate(1,0.5)

# 実行
circuit.update_quantum_state(state)

# サンプリング
sampled_values = state.sampling(100,seed)

# ランク0（つまりnode01）のプロセス1のみで結果を表示
if rank == 0:
    print(sampled_values)
```

```
[stdout:0] [6, 0, 6, 6, 1, 6, 4, 6, 0, 4, 6, 0, 3, 6, 0, 4, 0, 0, 2, 3, 2, 7, 3, 5, 0, 3, 3, 0, 2, 4, 4, 0, 3, 3, 3, 3, 3, 4, 3, 4, 2,
4, 6, 4, 3, 3, 3, 0, 3, 4, 6, 3, 0, 3, 1, 2, 3, 0, 2, 4, 3, 3, 3, 3, 3, 0, 6, 3, 6, 0, 3, 3, 2, 3, 6, 3, 0, 3, 0, 3, 3, 3, 3, 3, 3, 3,
4, 3, 0, 3, 3, 3, 0, 4, 6, 1, 1, 6, 3, 3]
```

Note

When you run mpirun from the command line, there is a wait time for the ssh connection to be established each time you run it, but when you run cells on the Jupyter Notebook, there is no wait time because the ssh connection between the nodes is already established when you run ipcluster.

8. Debugging (Visual Studio Code)

Here is an example of stepping through Python code executed by MPI using Visual Studio Code (hereinafter referred to as VS Code). Step execution can be performed by attaching a debugger to a rank 0 process among the processes running in MPI parallel.

Warning

In the debugging using debugpy described here, communication is performed between VS Code and the compute node executing the code to be debugged, but the communication port on the compute node side is an any connection, which is accessible even to the person who is not debugging. Please understand that before using it. Also, please do not access the compute node other than the one you have reserved.

8.1. Installing

In your local environment, follow the VS Code web page (https://code.visualstudio.com) to install VS Code. After installing VS Code, install a extension function “Remote - SSH”.

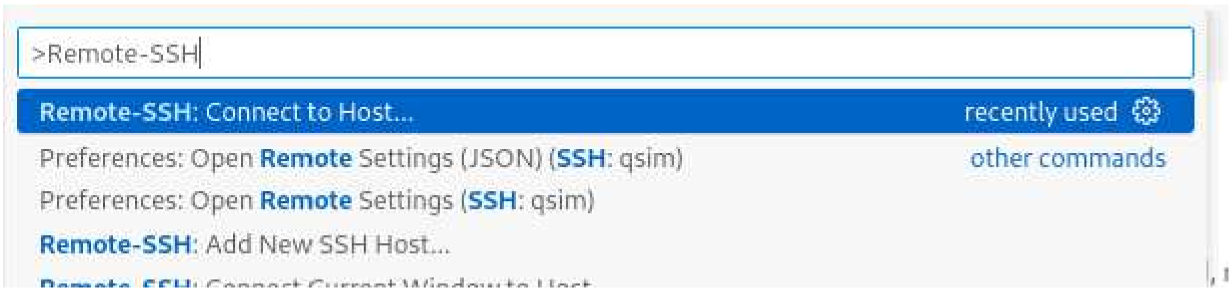
8.2. Running Interactive Job

Log in to the login server and run the Interactive job. This example uses two compute nodes. (Since the number of compute nodes is limited, please exit the Interactive job with the `exit` command when you are finished.)

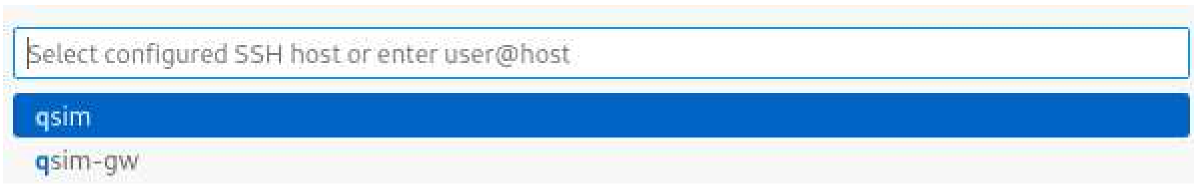
```
[user1@loginvm-XXX ~]$ salloc -N 2 -p Interactive --time=1:00:00
[user1@fx-XX-XX-XX ~]$
```

8.3. Connecting to the Login Server from VS Code

Connect to the compute node from VS Code using *Remote - SSH*. Press the F1 key and search for and select *Remote-SSH: Connect to Host*.


>Remote-SSH
Remote-SSH: Connect to Host...
recently used
Preferences: Open Remote Settings (JSON) (SSH: qsim)
other commands
Preferences: Open Remote Settings (SSH: qsim)
Remote-SSH: Add New SSH Host...

Enter *qsim* in the next dialog.



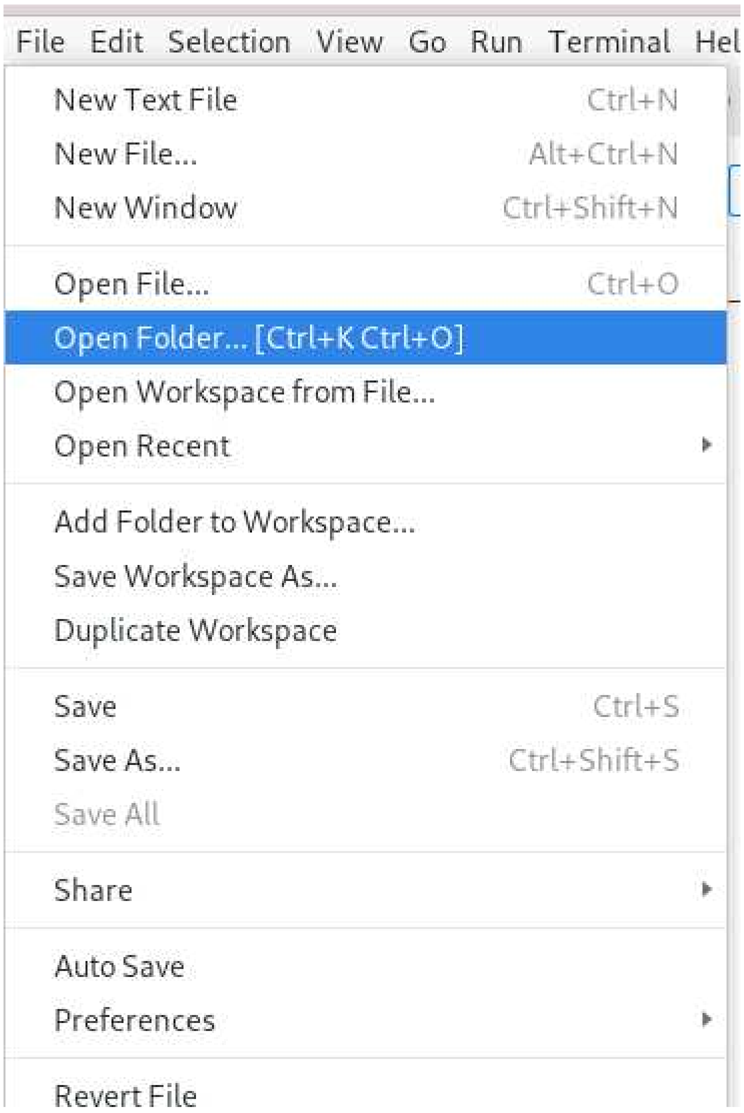
This completes the connection from VS Code to the login server.

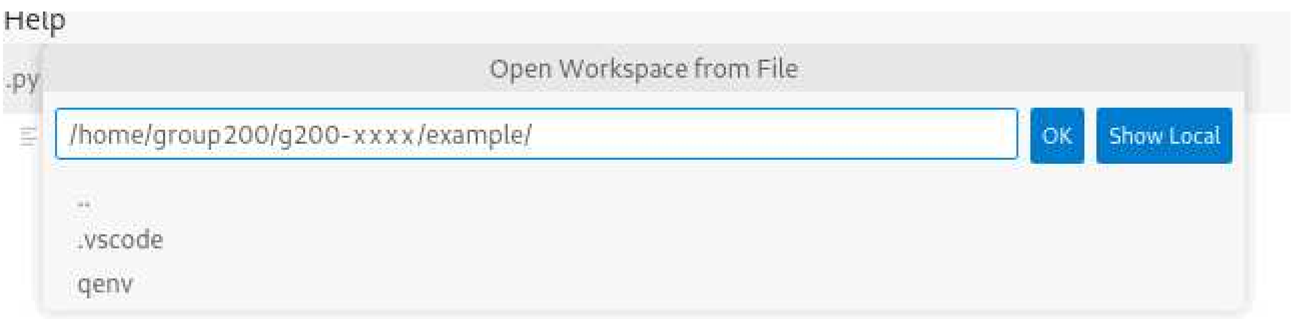
Note

The initial connection takes a long time to complete because the VS Code server is installed on the remote host.

8.4. Stepping with VS Code

Step execution is performed using `~/example` created in Job Execution Method as an example.
With VS Code connected to the login server, choose File menu - Open Folder, and then open the `~/example` .

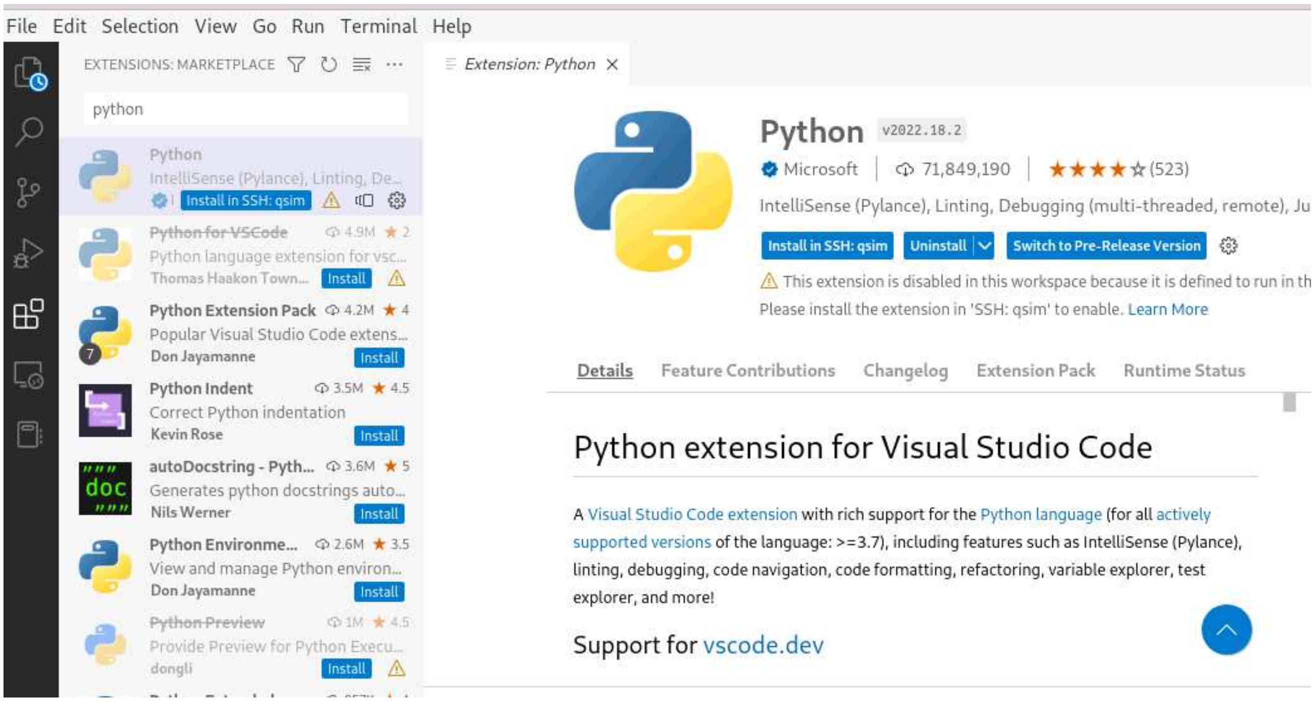




8.4.1. Preparation

8.4.1.1. Installing Python extensions

Install the following Python extension in VS Code.



Python v2022.18.2
Microsoft | 71,849,190 | ★★★★☆ (523)
IntelliSense (Pylance), Linting, Debugging (multi-threaded, remote), Ju

This extension is disabled in this workspace because it is defined to run in th
Please install the extension in 'SSH: qsim' to enable. Learn More

Python extension for Visual Studio Code

A Visual Studio Code extension with rich support for the Python language (for all actively supported versions of the language: >=3.7), including features such as IntelliSense (Pylance), linting, debugging, code navigation, code formatting, refactoring, variable explorer, test explorer, and more!

Support for vscode.dev

8.4.1.2. Creating a VS Code configuration file

Create `~/example/.vscode/launch.json` with the following content. Note that host should be the host name (fx-XX-XX-XX) of the compute node assigned in the Interactive job.

Note

From now on, each time you run an Interactive job, you must edit the host in the configuration file to match.

~/example/.vscode/launch.json

```
{
  "version": "0.2.0",
  "configurations": [
      {
          "name": "Python: remote connection",
          "type": "python",
          "request": "attach",
          "connect": {
              "host": "fx-XX-XX-XX",
              "port": 5678
          },
          "pathMappings": [
              {
                  "localRoot": "${workspaceFolder}",
                  "remoteRoot": "${workspaceFolder}"
              }
          ],
          "justMyCode": true
      }
  ]
}
```

8.4.1.3. Creating a job file

Create `~/example/job_debug.sh` with the following content.

~/example/job_debug.sh

```
#!/usr/bin/env bash

# Usage:
#     srun -n 2  job_debug.sh <path/to/venv> <python file>
#
# Example:
#     srun -n 2  job_debug.sh ~/example/qenv example.py

# Environment variable settings required for using MPI in a quantum simulator system
export UCX_IB_MLX5_DEVX=no
export OMP_PROC_BIND=TRUE

# Number of threads used for OpenMP parallelization
# Due to the OpenMP multi-threading behavior, some libraries (such as scipy) can introduce
small calculation errors,
# which can lead to inconsistent calculation results between processes during MPI execution.
When using mpiQulacs, set it to 1.
export OMP_NUM_THREADS=1

# Number of threads used by mpiQulacs (if not set, the value of OMP_NUM_THREADS is used)
export QULACS_NUM_THREADS=48

# Activating Python virtual environment
source $1/bin/activate
shift

### Workaround for the glibc bug (https://bugzilla.redhat.com/show_bug.cgi?id=1722181)).
if [ -z "${LD_PRELOAD}" ]; then
    export LD_PRELOAD=/lib64/libgomp.so.1
else
    export LD_PRELOAD=/lib64/libgomp.so.1:$LD_PRELOAD
fi

# Executing a Python command
RANK=${PMIX_RANK}

if [ ${RANK} -eq "0" ]; then
    COM='python -m debugpy --listen 0.0.0.0:5678 --wait-for-client'
else
    COM='python'
fi

${COM} "$@"
```

Open a terminal in VS Code and grant execute permission to *~/example/job_debug.sh* .

```
chmod u+x ~/example/job_debug.sh
```

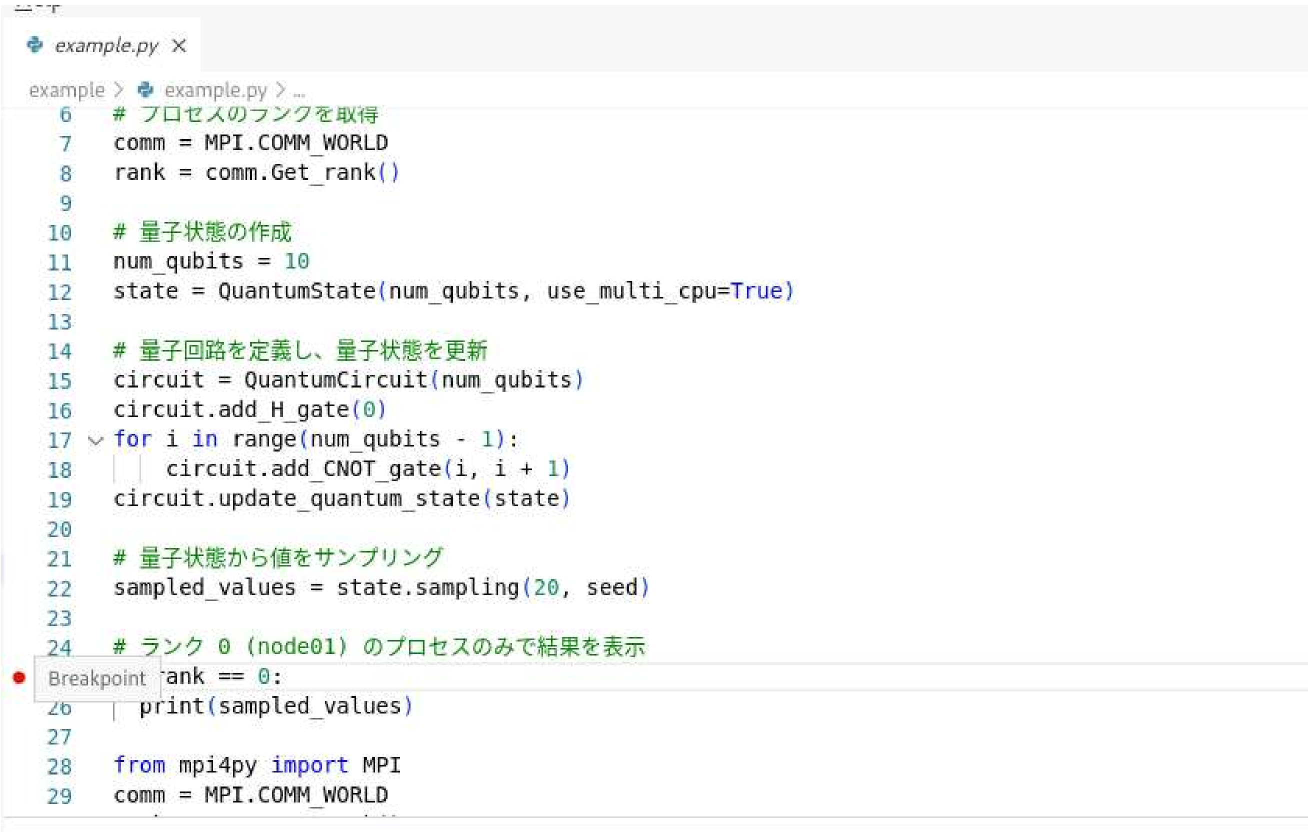
8.4.1.4. Installing debugpy

Install debugpy in a virtual environment for Interactive jobs.

```
[user1@fx-XX-XX-XX ~]$ cd example
[user1@fx-XX-XX-XX example]$ source ./qenv/bin/activate
(qenv) [user1@fx-XX-XX-XX example]$ pip install debugpy
```

8.4.2. Step execution

Open `~/example/example.py` in VS Code.
In this example, a breakpoint is set in the following locations.



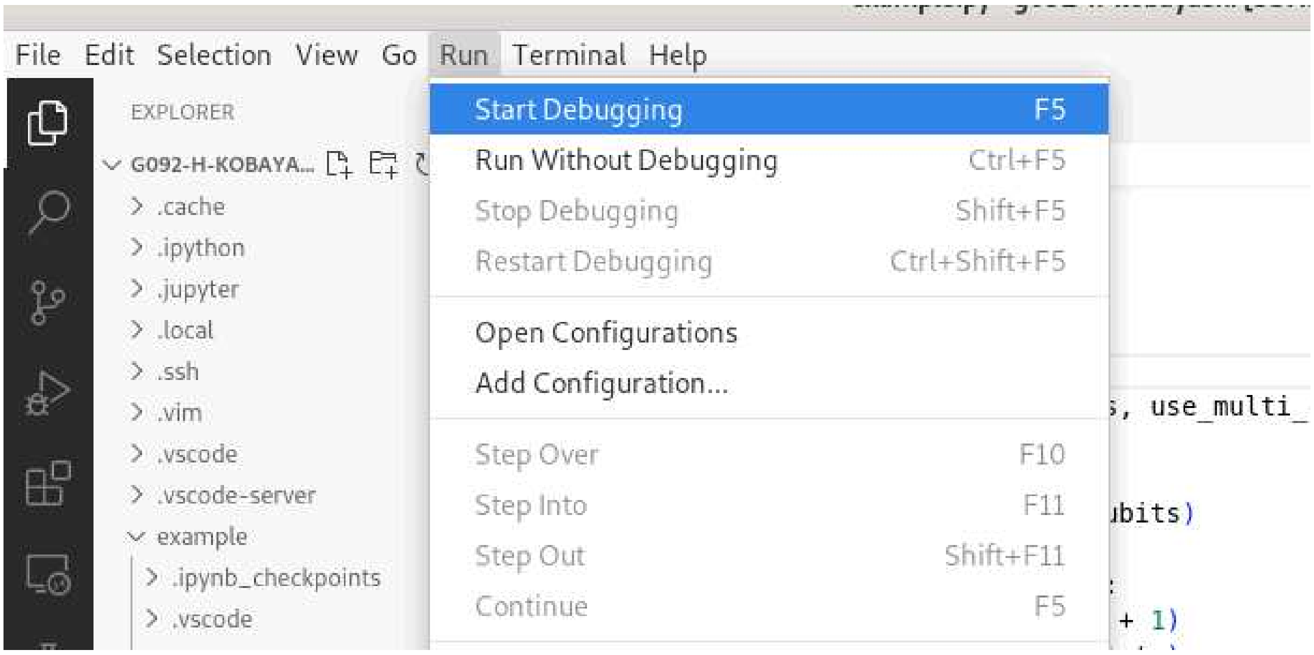
```
# プロセスのランクを取得
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# 量子状態の作成
num_qubits = 10
state = QuantumState(num_qubits, use_multi_cpu=True)

# 量子回路を定義し、量子状態を更新
circuit = QuantumCircuit(num_qubits)
circuit.add_H_gate(0)
for i in range(num_qubits - 1):
    circuit.add_CNOT_gate(i, i + 1)
circuit.update_quantum_state(state)

# 量子状態から値をサンプリング
sampled_values = state.sampling(20, seed)

# ランク 0 (node01) のプロセスのみで結果を表示
      ank == 0:
    print(sampled_values)

from mpi4py import MPI
comm = MPI.COMM_WORLD
```

Then, in the Interactive job virtual environment, do the following.

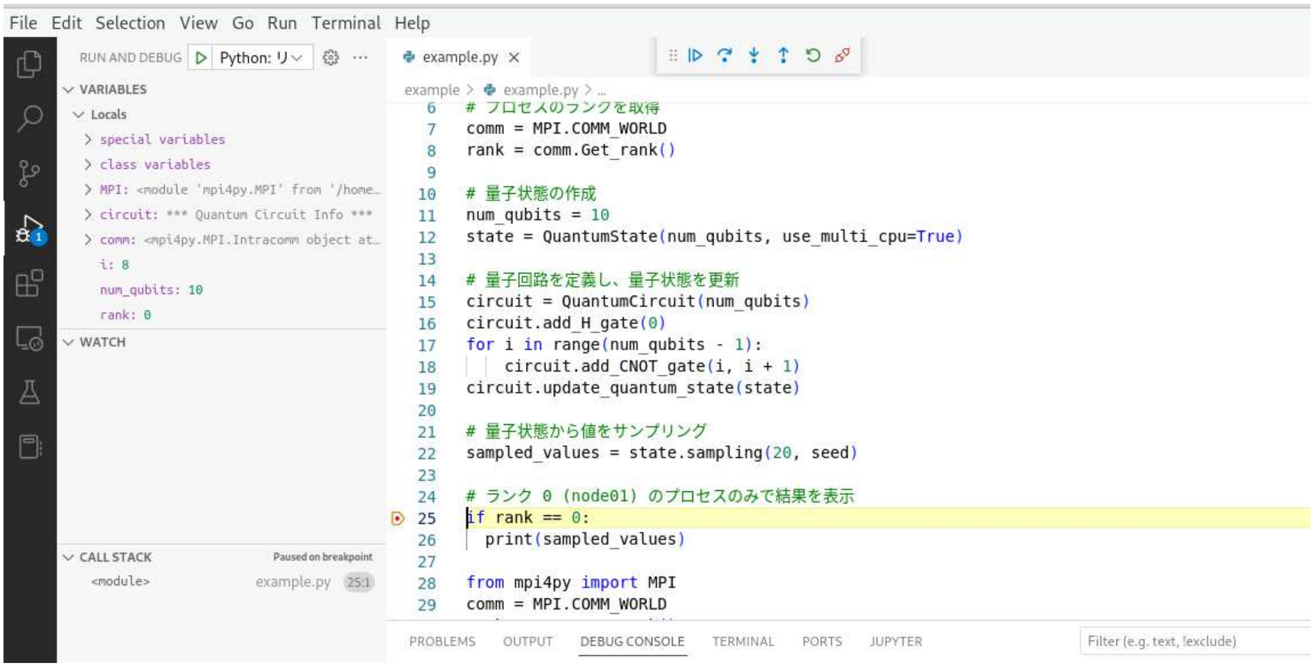
```
(qenv) [user1@fx-XX-XX-XX example]$ srun -n 2 job_debug.sh ~/example/qenv example.py
```

> **Note**
>
> Note that the arguments are different from job.sh created in Job Execution Method . The job_debug.sh takes a path to a Python virtual environment (~/example/qenv) and a Python file to debug. (The command `python` is listed in job_debug.sh, so you do not need to specify it as an argument.)

When executed, the process remains in a wait state until the debugger is attached.
After waiting about 10 seconds for the process to fully start, select *Run(R)* → *Start Debugging* at the top of the screen.



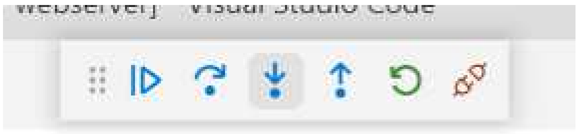
After a pause, the execution of the rank 0 process pauses when it reaches the breakpoint.



```
VARIABLES
  Locals
    > special variables
    > class variables
    > MPI: <module 'mpi4py.MPI' from '/home...
    > circuit: *** Quantum Circuit Info ***
    > comm: <mpi4py.MPI.Intracomm object at...
      i: 8
      num_qubits: 10
      rank: 0
WATCH
CALL STACK                      Paused on breakpoint
  <module>                      example.py 25:1
```

```python
# プロセスのランクを取得
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# 量子状態の作成
num_qubits = 10
state = QuantumState(num_qubits, use_multi_cpu=True)

# 量子回路を定義し、量子状態を更新
circuit = QuantumCircuit(num_qubits)
circuit.add_H_gate(0)
for i in range(num_qubits - 1):
    circuit.add_CNOT_gate(i, i + 1)
circuit.update_quantum_state(state)

# 量子状態から値をサンプリング
sampled_values = state.sampling(20, seed)

# ランク 0 (node01) のプロセスのみで結果を表示
if rank == 0:
    print(sampled_values)

from mpi4py import MPI
comm = MPI.COMM_WORLD
```

In this state, steps can be performed from the toolbar shown below on VS Code.



9. Python Package

9.1. PySCF

Version 1.7.6 and version 2.1.1 are currently available.

The following describes the installation procedure for version 1.7.6.

```
# If h5py>=3.3 is installed, uninstall it first.
# pip uninstall h5py

pip install "h5py<3.3" pyscf==1.7.6
# As h5py is a source installation, the first installation takes about 10 minutes.

# If you use dftd3, install pyscf-dftd3 additionally.
# (Install it after pyscf has been installed.)
pip install pyscf-dftd3
```

For version2.1.1, use Python3.9～3.11.

10. Troubleshooting

10.1. Error on Running Programs

10.1.1. Segmentation fault

Make sure `from mpi4py import MPI` is written int the program code.
If you run a program that uses MPI without MPI import, a segmentation fault occurs.
(However, using qiskit-qulacs requires no MPI import statements.)
If segmentation faults occur despite the MPI import statements, contact your Fujitsu representative.

10.1.2. Cannot allocate memory in static TLS block

This error is due to a bug in the GNU C libraries installed on the system.
If the error target is /lib64/libgomp.so.1, the workaround is to set the environment variable LD_PRELOAD=/lib64/libgomp.so.1 in job.sh (ipcluster_config.py for Jupyter).

If the error target is a file other than /lib64/libgomp.so.1, contact your Fujitsu representative.
(Reference) To fix the error, you need to re-create the wheel package by source-building the Python package on the quantum simulator system.

Warning

Searching the Internet for error messages will find a workaround to specify the path to the appropriate libgomp.so in the environment variable LD_PRELOAD, but this should not be done if LD_PRELOAD specifies a path other than /lib64/libgomp.so.1, which is installed on the system by default, as this may conflict with mpiQulacs behavior.

10.2. Error on Job Submission

When using the command squeue, the **(REASON)** of a running job may look like this.

- (Resources): Other user information is not visible in this system, so PD may occur even if the running queue is not displayed. Check the availability with the command sinfo.
- (PartitionNodeLimit): This is displayed when you run the job with more than the number of nodes each queue has. Check with your Fujitsu representative to determine the maximum number of nodes in each queue.

```
$ squeue
         JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            82     Batch test.job  USER PD       0:00      4 (PartitionNodeLimit)
            85     Batch test.job  USER PD       0:00      2 (Resources)
```

When using commands srun, sacct, sinfo, etc., the following error may occur.
If you encounter such an error or if any of the commands do not work correctly, contact your Fujitsu representative.

Error contents

- srun: error: Application launch failed: Communication connection failure
- slurm_load_partitions: Unable to contact slurm controller (connect failure)
- sacct: error: Problem talking to the database: Connection refused
- slurm_receive_msg: Zero Bytes were transmitted or received

About mpiQulacs

mpiQulacs is a quantum simulator based on Qulacs that enables parallel processing on multi-processes and multi-nodes, enabling high-speed execution of large-scale quantum circuits.
mpiQulacs is based on Qulacs.

mpiQulacs	Underlying Qulacs
~v1.2.2	v0.3.0
v1.3.0~	v0.3.1

Key Differences with Qulacs

Some features specific to mpiQulacs include:.
(See Usage for more information)

- You can specify a flag *use_multi_cpu* when you create a QuantumState instance.
  - QuantumState(qubits, use_multi_cpu)
    - “use_multi_cpu = True” indicates that the state vector will be distributed among multiple compute nodes (multiple ranks) as appropriate.
      Basically, always specify True.

Note that even if True is specified, the state vector is not distributed if the number of qubits is small ((N - k) ≦ $\log_2(S)$).
(S is the number of MPI ranks, N is the number of qubits, and k is the minimum number of qubits per process (constant k = 2))

- You can see whether the state vectors are distributed in state.get_device_name() .
    - Return value of state.get_device_name()

Return Value	Explanation
“cpu”	State vector created in 1 node
“multi-cpu”	State vector distributed across nodes (ranks)
(“gpu”)	Qulacs means that the state vector is placed on the GPU, but mpiQulacs does not support placement on the GPU.

- Special behavior when state vectors are distributed
    - The following methods display only the state vector information (part of the whole state vector) that each node has.
        - state.get_vector()
        - state.to_string()
    - The method state.set_Haar_random_state(seed) generates different state vectors if the number of state vector divisions is different, even if seed is specified.
- Random seed can now be specified in update_quantum_state().
    - Added API
        - QuantumGateBase::update_quantum_state(state, seed)
        - QuantumCircuit::update_quantum_state(state, seed)
- New adding of FusedSWAP gate
    - The higher the qubits, the more the upper qubits are distributed across the ranks.
      Qubits placed across ranks are called global qubits, and qubits completed within the same rank are called local qubits.
      Quantum gate operations on global qubits are slower than operations on local qubits because they involve communication between compute nodes.
      By using a FusedSWAP gate, the global and local qubits can be repositioned to reduce the amount of communication during gate operation.
- Optimization using FusedSWAP gate is added to QuantumCircuitOptimizer
- It is optimized for 512 bit-SVE instructions on the A64FX.

Installation

Requirements

mpiQulacs works in the following environments:.

- Python： >=3.8

- mpi4py： >=3.1

mpiQulacs uses the same package name as `qulacs` .
Avoid installing Qulacs and mpiQulacs in the same Python environment.

Installing mpiQulacs

mpiQulacs can be installed via pip. If you use the Python virtual environment, you can install it, for example:.

```
# Create virtual environment and activate
python3.8 -m venv qenv
source qenv/bin/activate
# Install the latest version of pip and wheel
pip install -U pip wheel
# Install a required module
pip install mpi4py numpy
# Install mpiQulacs
pip install mpiQulacs
```

mpiQulacs provides the same “qulacs” module as Qulacs. Do not install mpiQulacs and Qulacs in one python environment

Usage

mpiQulacs has the same API interface as Qulacs with some exceptions.
You can write quantum programs just like Qulacs.
See also Qulacs documents for descriptions and code examples for each API.

Example

```python
from qulacs import QuantumCircuit, QuantumState
from qulacs.circuit import QuantumCircuitOptimizer
from qulacs.gate import Y,CNOT,merge
from mpi4py import MPI  # import is required for MPI execution even if MPI classes are not used

comm = MPI.COMM_WORLD
mpirank = comm.Get_rank()

# Define quantum states and initialize them with random values
state = QuantumState(3, use_multi_cpu=True)
state.set_Haar_random_state()

# Define quantum circuits and update quantum states
circuit = QuantumCircuit(3)
circuit.add_X_gate(0)
merged_gate = merge(CNOT(0, 1), Y(1))
circuit.add_gate(merged_gate)
circuit.add_RX_gate(1, 0.5)

# quantum circuit optimization
Opt_type = "Heavy"
if Opt_type == "Light":
  # Optimize using SWAP (reduces inter-node communication in update processing)
  swap_level = 1
  # Quantum gates of a quantum circuit is merged by a greedy method
  # until the target quantum bit becomes the specified size.
  if state.get_device_name() == "multi-cpu":
    QuantumCircuitOptimizer().optimize_light(circuit, swap_level)
  else:
    QuantumCircuitOptimizer().optimize_light(circuit)
elif Opt_type == "Heavy":
  # Maximum quantum gate size allowed to be merged
  max_block_size = 2
  # More powerful SWAP/FusedSWAP optimizations than swap_level = 1
  swap_level = 2
  # The optimisation also takes into account the exchange of exchangeable quantum gates.
  # If the quantum circuit size is large, the optimisation process time may increase
significantly.
  if state.get_device_name() == "multi-cpu":
    QuantumCircuitOptimizer().optimize(circuit, max_block_size, swap_level)
  else:
    QuantumCircuitOptimizer().optimize(circuit, max_block_size)

# Update states using circuit
circuit.update_quantum_state(state)

# Sampling from quantum states
samples = state.sampling(20, 2022)  # with random seed = 2022
if mpirank == 0:
  print(samples)
```

The second argument to QuantumState must be `use_multi_cpu=True` . (This distributes state vectors across multiple compute nodes, enabling parallel processing by MPI.)

List of APIs with functional differences from Qulacs

- QuantumState class
    - QuantumState(qubits, use_multi_cpu)
        - use_multi_cpu = False
            - Positions the entire state vector within one node (same behavior as Qulacs)
        - use_multi_cpu = True
            - Distributes the state vector across multiple nodes.
              However, (N - k) ≦ $\log_2(S)$ does not distribute. (S is the number of MPI ranks, N is the number of qubits, and k is the minimum number of qubits per process (constant k = 2).)
            - The qubit is classified as local qubit and global qubit and is arranged as follows:
                - local qubit: Placed in same node
                - global qubit: Distributed across multiple nodes
                    - The top $\log_2$(number of ranks) qubits are classified as global qubits
    - state.get_device_name()
        - Returns the device where the state vector is located.

Return value	Explanation
“cpu”	State vector created in 1 node
“multi-cpu”	State vector distributed across nodes (ranks)
(“gpu”)	Qulacs means that the state vector is placed on the GPU, but mpiQulacs does not support placement on the GPU.

-
    - state.load(vector)
        - When state vectors are distributed, only the vector elements of the entire state vector that are placed at each rank must be specified as arguments. (For calls to state.load with rank N, only vector elements that are placed in rank N should be specified as arguments to the load method.) If you want to load the entire state vector as an argument, see Method for loading/getting entire distributed state vector.
    - state.get_vector()
        - If the state vectors are distributed, returns only the state vector of each rank (that is, a portion of the entire state vector). If you want to get the entire state vector as a return value, see Method for loading/getting entire distributed state vector .
    - state.to_string()

        - For a state vector with a large number of elements, the first 256 elements are output.
        - If the state vectors are distributed, the first part of each rank's state vector (part of the whole state vector) is output.
        - The number of local qubit and global qubit is also output.

        Sample output

        ```
        -- rank 0 --------------------------------------
        *** Quantum State ***
        * MPI rank / size : 0 / 2
        * Qubit Count : 20 (local / global : 19 / 1 )
        * Dimension   : 262144
        * state vector is too long, so the (128 x 2) elements are output.
        * State vector (rank 0):
        (1,0)
        ...
        -- rank 1 --------------------------------------
        * State vector (rank 1):
        (0,0)
        ...
        ```

    - state.set_Haar_random_state([seed])
        - Initializes each element of the state vector with a random number.
        - If the state vector is distributed,
            - If you do not specify seed, the random seed value generated by rank 0 is first shared among all ranks.
            - Based on the specified or shared seed, each rank uses (seed + rank number) as the final seed value. Therefore, if the state vectors are distributed, **the state vectors generated will be different if the number of divisions is different**, even if the same seed value is specified for the argument.
    - state.sample(number_sampling [, seed])
        - If seed is not specified, a random seed value generated at rank 0 is shared among all ranks and used as the seed value.
- FusedSWAP(qubit_idx1, qubit_idx2, block_size)
    - Swaps the placement of block_size qubits from the index specified in qubit_idx1 with the placement of block_size qubits from the index specified in qubit_idx2.
        - The effect of the gate is equivalent to continuously applying a SWAP gate to the range of qubit index being swapped.
          When a FusedSWAP gate is used, its replacement process is completed with one gate.
    - The primary use of FusedSWAP is to speed up gate operations when state vectors are distributed.
    - As the qubit count increases, the information in the top $\log_2$(number of ranks) qubits is distributed across the ranks. Quantum gate operations on qubits (global qubits) that

    are placed across ranks are slower than operations on qubits (local qubits) that are placed in the same rank because they require communication between compute nodes.
    If you have a lot of gate operations on the global qubit, you can reduce the gate operations on qubits across the ranks by using a FusedSWAP gate to swap the placement of the global qubit and the local qubit. This reduces the amount of communication during gate operation and speeds up the operation.
  - Please also refer to Example of use of FusedSWAP .
- Specifying seed in update_quantum_state
  - QuantumGate.update_quantum_state(state, seed)
  - QuantumCircuit.update_quantum_state(state, seed)

    Random numbers used in gates using random numbers such as Measurement can be fixed. Once seed is specified, the random number generator initialized by seed is inherited.
- Automatic FusedSWAP gate insertion of QuantumCircuitOptimizer
  - Add the swap_level argument to the optimize method of QuantumCircuitOptimizer. By specifying swap_level = 1, SWAP/FusedSWAP gates are automatically inserted to reduce gate operations on the global qubit. This reduces the amount of communication during gate operation and increases the speed.
  - optimize(circuit, block_size, swap_level=0)
  - optimize_light(circuit, swap_level=0)
    - swap_level = 0
      - Do not insert SWAP/FusedSWAP gate.
    - swap_level = 1
      - Inserting SWAP/FusedSWAP gate reduces and speeds up inter-node communication in update operations.
    - swap_level = 2
      - Insert SWAP/FusedSWAP gates while changing the gate order. You might be able to reduce traffic even further than swap_level = 1.

    In mpiQulacs v1.3.0 and later, it is possible to perform optimization on a gate-by-gate basis and optimization using FusedSWAP at the same time.

Note

Basically, every method has to be run the same way in every rank.
(Do not perform a gate operation or call the sampling method in a program block that behaves differently depending on its rank, such as *if rank == 0* .)
If you specify seed, you must specify a value that is common to all ranks.
Please also refer to item Notes on programming in System Usage.

Note

The FusedSWAP gate and QuantumCircuitOptimizer’s automatic FusedSWAP gate insertion (swap_level > 1) have different effects depending on the structure of the quantum circuit and the number of MPI parallelism.
We recommend that you compare and consider the performance with and without them.

Environment variable settings

- QULACS_NUM_THREADS (1 - 1024)
  - Specifies the maximum number of threads to use with mpiQulacs. Takes precedence over OMP_NUM_THREADS.

Example of use of FusedSWAP

In some cases, you can use FusedSWAP to speed up the simulation by reducing gate operations on qubits across ranks.
The following code inserts FusedSWAP before performing any operations on the global qubit to prevent gate operations on qubits across ranks.

```python
import numpy as np
from qulacs import QuantumCircuit, QuantumState
from mpi4py import MPI

comm = MPI.COMM_WORLD
mpirank = comm.Get_rank()
mpisize = comm.Get_size()
np.random.seed(seed=32)

def build_circuit(depth, nqubits, mpisize):
    global_qc = int(np.log2(mpisize))
    local_qc = nqubits - global_qc
    swapped = False

    circuit = QuantumCircuit(nqubits)
    for _ in range(depth):
        # operation on local qubit
        for i in range(local_qc):
            circuit.add_H_gate(i)

        # Swap the placement of global qubit and local qubit before operating on global qubit
        # (Place global qubits in the same rank)
        if global_qc != 0:
            circuit.add_FusedSWAP_gate(local_qc - global_qc, local_qc, global_qc)
            swapped = not swapped

        # operation on global qubit
        for i in range(global_qc):
            circuit.add_H_gate(local_qc - global_qc + i)

    # In the for statement above, we added FusedSWAP depth times,
    # we added FusedSWAP as needed to return to the original qubit sequence.
    if (global_qc != 0) and swapped:
        circuit.add_FusedSWAP_gate(local_qc - global_qc, local_qc, global_qc)

    return circuit

if __name__ == '__main__':
    nqubits = 36
    st = QuantumState(nqubits, use_multi_cpu=True)
    circuit = build_circuit(10, nqubits, mpisize)
    circuit.update_quantum_state(st)
```

Method for loading/getting entire distributed state vector

As described in List of APIs with functional differences from Qulacs , QuantumState.load(vector), QuantumState.get_vector() processes only those vector elements that are placed at each rank in the entire state vector.

If you want to load/get_vector the entire state vector as an argument/return value, define a method like this and use it instead of load, get_vector:.

However, note the following:.
As the qubit number increases, the entire state vector becomes too large to fit in one node of memory. For this reason, the state vector is split into multiple nodes, and the load/get_vector method only processes the vector elements that each node is responsible for.

Note that if you define a method such as the one below and try to process the entire state vector within one node, a large number of qubits (close to 30 qubits) can cause a program crash due to lack of memory.

```
import numpy as np
from mpi4py import MPI

def load_state_vector(state, vector):
    """
    Loads the given entire state vector into the given state.

    Args:
        state (qulacs.QuantumState): a quantum state
        vector: a state vector to load
    """
    if state.get_device_name() == 'multi-cpu':
        mpicomm = MPI.COMM_WORLD
        mpirank = mpicomm.Get_rank()
        mpisize = mpicomm.Get_size()
        vector_len = len(vector)
        idx_start = vector_len // mpisize * mpirank
        idx_end = vector_len // mpisize * (mpirank + 1)
        state.load(vector[idx_start:idx_end])
    else:
        state.load(vector)

def get_state_vector(state):
    """
    Gets the entire state vector from the given state.

    Args:
        state (qulacs.QuantumState): a quantum state
    Return:
        vector: a state vector
    """
    if state.get_device_name() == 'multi-cpu':
        mpicomm = MPI.COMM_WORLD
        mpisize = mpicomm.Get_size()
        vec_part = state.get_vector()
        len_part = len(vec_part)
        vector_len = len_part * mpisize
        vector = np.zeros(vector_len, dtype=np.complex128)
        mpicomm.Allgather([vec_part, MPI.DOUBLE_COMPLEX],
                          [vector, MPI.DOUBLE_COMPLEX])
        return vector
    else:
        return state.get_vector()
```

Important Notes

- The rank value (specified by the -n option of the mpirun command) during MPI execution must be a power of 2.
- The current version supports the following class methods:.
  Other functions may cause segmentation faults or abnormal results.(Non-supported features are listed in Limitations)
  - QuantumCircuit

- QuantumCircuitOptimizer
  - optimize(circuit, block_size, swap_level=0)
    - Only (block_size=1, swap_level=0) or (block_size=0, swap_level=1) combinations are allowed
  - optimize_light(circuit, swap_level=0)
- ParametricQuantumCircuit
- QuantumState
  - copy
  - load
  - get_device_name
  - get_entropy
  - get_vector
  - normalize
  - sampling
  - set_computational_basis
  - set_Haar_random_state
  - to_string
- gate
  - X / Y / Z
  - CNOT / CZ / SWAP
  - Identity / H
  - P0 / P1
  - RX / RY / RZ
  - S / Sdag / T / Tdag
  - SqrtX / SqrtXdag / SqrtY / SqrtYdag
  - U1 / U2 / U3
  - DenseMatrix
    - Only supported when the target qubit number is less than or equal to 2.
    - Adding control qubits is supported only if the control qubit number is 1 and the target qubit number is 1.
  - DiagonalMatrix
    - Only supported when target qubit number is 1
  - merge
    - Only supported if qubit number is less than or equal to 2
  - Measurement
  - CPTP
  - Instrument
  - Adaptive
  - Pauli
  - PauliRotation
  - RandomUnitary
  - to_matrix_gate
- GeneralQuantumOperator (※1)
- Observable (※1)
- PauliOperator (※1)

(※1) However, get_transition_amplitude is not supported.

Limitations

The following class methods are not supported in the current version:.

- class methods to be added in future releases
  - gate
    - TOFFOLI
    - FREDKIN
    - DenseMatrix (When using any number of target qubit and control qubit)
    - DiagonalMatrix (When the target qubit number is 2 or more)
    - merge (hen the target qubit number is 3 or more)
  - QuantumCircuitSimulator
  - state
    - inner_product
    - tensor_product
    - permutate_qubit
    - drop_qubit
    - partial_trace
    - get_marginal_probability
    - get_zero_probability
  - QuantumGateBase
  - QuantumGateMatrix
  - GeneralQuantumOperator.get_transition_amplitude( )
  - Observable.get_transition_amplitude( )
  - PauliOperator.get_transition_amplitude( )
- class method for which the timing is unknown
  - gate
    - SparseMatrix
    - ReversibleBoolean
    - StateReflection
    - BitFlipNoise
    - DephasingNoise
    - IndependentXZNoise
    - DepolarizingNoise
    - TwoQubitDepolarizingNoise
    - AmplitudeDampingNoise
    - add
    - Probabilistic
    - ProbabilisticInstrument
    - CP
  - DensityMatrix
  - QuantumGate_SingleParameter

About qiskit-qulacs

qiskit-qulacs is a package for using mpiQulacs as a backend for Qiskit. Using the API of the Qiskit SDK, you can write quantum programs and let mpiQulacs execute quantum circuits.

Installation

Requirements

qiskit-qulacs works in the following environments.

- Python： >=3.8, <3.10
- Qiskit： >=0.32.1, <0.33.0
  - If you want to use Qiskit applications library, please use the following version.
    - Qiskit Nature： 0.2.2
    - Qiskit Machine Learning： >=0.2.1, <0.3.0
    - Qiskit Finance： >=0.3.1, <0.4.0
    - Qiskit Optimization： 0.2.3
- mpiQulacs： >=1.0.0, <2.0.0

Qiskit and mpiQulacs are automatically installed when you pip install qiskit-qulacs. If you have installed other versions of Qiskit and mpiQulacs, uninstall them first.

If you want to use a version of Python that is not installed on your system by default, see Install of pyenv .

Installing qiskit-qulacs

qiskit-qulacs can be installed via pip.
If you use the Python virtual environment, you can install it, for example:.

```
# Interactive from login server
[username@loginvm-XXX ~]$ salloc -N 1 -p Interactive --time=1:00:00
[username@fx-XX-XX-XX ~]

# Create virtual environment and activate
[username@fx-XX-XX-XX ~]$ mkdir example
[username@fx-XX-XX-XX ~]$ cd example
[username@fx-XX-XX-XX example]$ python3.8 -m venv qenv
[username@fx-XX-XX-XX example]$ source ./qenv/bin/activate
(qenv) [username@fx-XX-XX-XX example]$

# Install the latest version of pip and wheel
(qenv) [username@fx-XX-XX-XX example]$ pip install --upgrade pip wheel
# Install the qiskit-qulacs (For the first installation, it takes about 15 min.)
(qenv) [username@fx-XX-XX-XX example]$ pip install qiskit-qulacs

# If you want to use the Qiskit applications library, please install the following version.
# (For example, if you only need qiskit-nature, you do not need to install anything other than
qiskit-nature.)
(qenv) [username@fx-XX-XX-XX example]$ pip install qiskit-nature==0.2.2
(qenv) [username@fx-XX-XX-XX example]$ pip install qiskit-machine-learning==0.2.1
(qenv) [username@fx-XX-XX-XX example]$ pip install qiskit-optimization==0.2.3
# If you want to install qiskit-finance, install qiskit-optimization first.
(qenv) [username@fx-XX-XX-XX example]$ pip install qiskit-finance==0.3.1
(qenv) [username@fx-XX-XX-XX example]$
(qenv) [username@fx-XX-XX-XX example]$ exit
```

Note

Due to installation on aarch64, wheel files are often missing from pypi.org. As a result, pip install can run the source build and take a relatively long time to complete the installation. For subsequent installations, the wheel files are cached under your home directory, reducing installation time.

Usage

When using execute(...) or a QuantumInstance of Qiskit, you can force mpiQulacs to run quantum circuits by specifying a QulacsBackend instance as the backend parameter.
You can get a QulacsBackend instance as follows.

```python
from qiskit_qulacs import QulacsProvider
backend = QulacsProvider().get_backend()
# type(backend) --> qiskit_qulacs.QulacsBackend
```

Examples

Example 1 : implementation of quantum circuits by execute(...)

example1.py

```python
from qiskit import QuantumCircuit, execute
from qiskit_qulacs import QulacsProvider
from mpi4py import MPI  # If you do not use MPI classes, you do not need to write this (this is
done internally by qiskit_qulacs import).

# Get rank in MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Create quantum circuits
num_qubits = 10
circ = QuantumCircuit(num_qubits)
circ.h(0)
for i in range(num_qubits - 1):
    circ.cx(i, i + 1)
circ.measure_all()

# Run quantum circuits
if rank == 0:
   print('Execute a circuit of 10 qubits.')
backend = QulacsProvider().get_backend()  # Get QulacsBackend instance
job = execute(circ, backend=backend, shots=1024, memory=True, seed_transpiler=50,
seed_simulator=80)
result = job.result()

# Display results
if rank == 0:
   counts = result.get_counts()
   print(f'{counts=}')
   # memory = result.get_memory()
   # print(f'{memory=}')
```

Example job script sim_example1.job

```
#!/bin/bash

#SBATCH -p Batch # Specify Batch queue
#SBATCH -o test  # Output file name
#SBATCH -N 2     # Number of nodes allocated
#SBATCH -t 06:00:00  # max execution time

mpirun -npernode 1 job.sh ~/example/qenv python example1.py
```

Run with sbatch from the command line

```
$ sbatch sim_example1.job
```

Execution result

```
$ cat ./test
counts={'0000000000': 511, '1111111111': 513}
# memory=['0000000000', '1111111111', ...]  # Results per shot (available in execute(...) with
memory = True)
```

Example 2 : use of QuantumInstance

example2.py

```python
from qiskit import QuantumCircuit
from qiskit.algorithms import VQE
from qiskit.algorithms.optimizers import SPSA
from qiskit.circuit.library import TwoLocal
from qiskit.opflow import I, X, Z
from qiskit.utils import QuantumInstance, algorithm_globals
from qiskit_qulacs import QulacsProvider
from mpi4py import MPI  # If you do not use MPI classes, you do not need to write this (this is
done internally by qiskit_qulacs import).

# Get rank in MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

algorithm_globals.random_seed = 50

# Hamiltonian for an H2 molecule
H2_op = (-1.052373245772859 * I ^ I) + \
        (0.39793742484318045 * I ^ Z) + \
        (-0.39793742484318045 * Z ^ I) + \
        (-0.01128010425623538 * Z ^ Z) + \
        (0.18093119978423156 * X ^ X)

backend = QulacsProvider().get_backend()  # Get QulacsBackend instance
qi = QuantumInstance(backend=backend, seed_transpiler=1234, seed_simulator=20)
ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
spsa = SPSA(maxiter=125)
vqe = VQE(ansatz, optimizer=spsa, quantum_instance=qi)

# Run VQE
if rank == 0:
   print('Run VQE with Qiskit and mpiQulacs')
result = vqe.compute_minimum_eigenvalue(operator=H2_op)

# Display results
if rank == 0:
   print(f'VQE result is {result.eigenvalue.real:.5f}')
```

Example job script sim_example2.job

```
#!/bin/bash

#SBATCH -p Batch # Specify Batch queue
#SBATCH -o test  # Output file name
#SBATCH -N 2     # Number of nodes allocated
#SBATCH -t 06:00:00  # max execution time

$ mpirun -npernode 1 job.sh ~/example/qenv python example2.py
```

Run with sbatch from the command line

```
$ sbatch sim_example2.job
```

Execution result

```
$ cat ./test
VQE result is -1.85719
```

Specify Random Seed

As described in Notes on programming in System Usage, the program needs to behave deterministically, unless it is intended to behave differently in each rank.
Therefore, when using Qiskit, you must explicitly set a random number seed value for the parts of Qiskit where random numbers are used.
For example, if you use an algorithm package or a transpiler, you need to set the algorithm_globals.random_seed and seed_transpiler values respectively.

```python
from qiskit.utils import QuantumInstance, algorithm_globals

algorithm_globals.random_seed = integer value
execute(circuit, seed_transpiler=integer value, seed_simulator=integer value)
# If QuantumInstance is used instead of execute(...):
# qi = QuantumInstance(backend=backend, seed_transpiler=integer value, seed_simulator=integer
value)
```

As in the above example, always specify a seed value if you can set a random seed. (example: qiskit.quantum_info.random_unitary(dims, seed))

Note

quantum_instance parameters, such as qiskit.algorithms.VQE(...), accept a Backend object as well as a QuantumInstance object, but remember to specify a QuantumInstance object because a Backend object prevents you from specifying values of seed_transpiler and seed_simulator.

```
backend = QulacsProvider()
# vqe = VQE(..., quantum_instance=backend) # N.G.
qi = QuantumInstance(backend, seed_transpiler=integer value, seed_simulator=integer value)
vqe = VQE(..., quantum_instance=qi) # O.K.
```

Basis Gates

QulacsBackend supports the following as basis gates:.

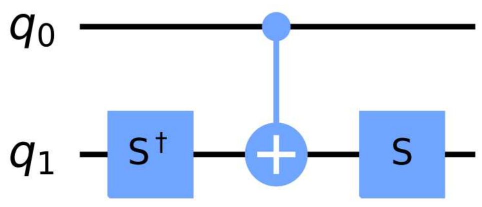
- Unitary operations
  - x, y, z, h, s, sdg, t, tdg, rx, ry, rz, u1, u2, u3, u, p, id, sx, sxdg, cx, cz, swap
- Non-unitary operations
  - measure, reset, barrier

If other gates are included in the quantum circuit, the quantum circuit is executed after they are decomposed into combinations of the basis gates by the transpile(...) method of Qiskit. (When the quantum circuit is executed via execute(...) and QuantumInstance, the transpile(...) method is automatically executed inside the library.)

> (reference) You can see how circuits are transpiled as follows.

```python
from qiskit import QuantumCircuit, transpile
from qiskit_qulacs import QulacsProvider

circ = QuantumCircuit(2)
circ.cy(0, 1)
backend = QulacsProvider().get_backend()
transpiled_circ = transpile(circ, backend=backend)
print(transpiled_circ.draw(fold=-1))
```

transpile result



Conditional Gates

QulacsBackend supports conditional gate operations based on classical register values (Qiskit c_if(...) method).

```
from qiskit import ClassicalRegister, QuantumCircuit, QuantumRegister, execute
from qiskit_qulacs import QulacsProvider

qreg = QuantumRegister(2)
creg = ClassicalRegister(2)
circ = QuantumCircuit(qreg, creg)
circ.h(0)
circ.measure(qreg, creg)
circ.x(1).c_if(creg, 0)
circ.measure(qreg, creg)
backend = QulacsProvider().get_backend()
counts = execute(circ, backend, shots=1024, seed_transpiler=50,
seed_simulator=80).result().get_counts()
print(counts)
```

Execution result

```
{'10': 503, '01': 521}
```

Optimization

When using QulacsBackend, there are two types of circuit optimization:.

- Optimization by Qiskit transpile(...) method
    - When the quantum circuit is executed via execute(...) and QuantumInstance, the transpile(...) method is called internally for optimization. The optimization level can be controlled by the value of the optimization_level parameter for execute(...) and QuantumInstance(...). See documents of Qiskit transpile(...) for details.
- Optimization by Qulacs QuantumCircuitOptimizer
    - After optimization with transpile(...) above, further optimization is done with Qulacs's QuantumCircuitOptimizer. (Reference: documents of Qulacs QuantumCircuitOptimizer) You can control the optimization level with a set_qulacs_optimization (mode, block_size, swap_level) method of the QulacsBackend instance. You can also get the current settings with the get_qulacs_optimization() method. A description of each parameter follows:.

Parameter	Explanation
mode	Ether None, 'normal', 'light' can be set. (Default is 'light'.) If None is specified, no optimization is performed by QuantumCircuitOptimizer. Specify 'normal' or 'light' to perform optimizations by QuantumCircuitOptimizer optimize or optimize_light method, respectively.

Parameter	Explanation
block_size	If mode is' normal ', QuantumCircuitOptimizer().optimize(circuit, block_size, swap_level) is executed using the value specified for block_size. See API specifications of mpiQulacs for possible block_size values. If mode is None or 'light', the value of the block_size argument is ignored.
swap_level	If mode is' normal 'or' light ', the optimize(circuit, block_size, swap_level) or optimize_light(circuit, swap_level) of QuantumCircuitOptimizer is executed using the value specified in swap_level. See API specification of mpiQulacs for the meaning of swap_level. The default is swap_level = 0 . If mode is None, the value of the swap_level argument is ignored.

Note

If a quantum circuit includes operations that involve measurements in the middle (excluding the measurement operation performed at the end of the circuit), the QuantumCircuitOptimizer does not optimize, regardless of the value set in mode. (QuantumCircuitOptimizer does not support optimization for such circuits.)

Warning

Note that the QuantumCircuitOptimizer optimizes by ignoring barrier gates.
(Qiskit's transpile(...) method optimization does not optimize across barriers.)

Limitation

Functions dedicated to Qiskit Aer Simulator

QulacsBackend does not support functions specific to the Aer simulator (Reference: Aer API reference)
For example, if you use QulacsBackend to evaluate a circuit that executes the Aer custom instruction circuit.save_statevector(), an error is output.

Some classes that are not listed in Aer API reference are for Aer only.
For example, qiskit.opflow.AerPauliExpectation is for Aer only. If you want to get the expected value with QulacsBackend, use qiskit.opflow.PauliExpectation instead.
(A class name that includes the string Aer may be Aer-specific.)

Noise simulation

QulacsBackend does not support noise simulation.
The simulation using NoiseModel of Aer and the simulation of the circuit including quantum error instruction (qiskit.quantum_info.Kraus, etc.) are not possible.

Number of qubits available when including intermediate measurements

If a quantum circuit includes operations that involve measurements in the middle (excluding the measurement operation performed at the end of the circuit), the maximum number of available qubits on the system is reduced by one qubit. (mpiQulacs functional limitation.)

Release Notes

v1.3.1 (2023/04/17)

- mpiQulacs v1.3.1 Release
  - Added error handling when insufficient memory is encountered during sampling process
  - Supported QuantumState.get_entropy function with multi-cpu
  - Improved performance of Swap Gate processing

v1.3.0 (2022/12/13)

- Update of system usage with system configuration change
  - Number of compute nodes changed from 64 to 512
  - Introduction of the Job Scheduler Slurm
  - Change the login destination to the login server instead of the representative compute node
  - Some IP addresses/host names of various servers, such as a jump server, were changed.
  - Add Job Scheduler Slurm Description
  - User account name at login, HostName on jump server, port number changed
  - Job execution method, Jupyter, and debugging (Visual Studio Code) steps changed
  - Adding v2.1.1 to PySCF in Python Package
- mpiQulacs v1.3.0 Release
  - Capture Correction in Qulacs v1.3.1
    - Corrects memory leak failures
  - The QuantumCircuitOptimizer function now supports optimization on a gate-by-gate basis and optimization with FusedSWAP at once.
  - Improved state vector update processing performance for some gates.

v1.2.3 (2022/07/20)

- Changed the contents of job.sh、job_debug.sh、ipcluster_config.py. Update the contents of each file you are using.
  - `job.sh`: We added the LD_PRELOAD environment variable only when using qiskit-qulacs, but changed it to always set the LD_PRELOAD environment variable whether or not you use qiskit-qulacs.

    - `job_debug.sh`: This file is used for debugging with VSCode. Added setting of environment variables (OMP_NUM_THREADS, QULACS_NUM_THREADS) to control the number of threads and setting of the LD_PRELOAD environment variable.
    - `ipcluster_config.py`: Configuration file for Jupyter. Added the LD_PRELOAD environment variable setting to reflect the changes made to job.sh.
- Added version numbers available in the Qiskit applications library when using qiskit-qulacs.
    - If you use Qiskit Nature, Machine Learning, Optimization, Finance, install the version described on the Installation page.
- Changed the way you log in to the document server to use ssh config.

v1.2.2 (2022/06/10)

- mpiQulacs v1.2.2 Release
    - Fixed a bug that could cause Observable.get_expectation_value(state) to return an incorrect value.
        - Occurrence condition: Can occur if an operator contains one or more pauli-X/Y and acts only on pauli-Z for global_qubit.

v1.2.1 (2022/06/08)

- mpiQulacs v1.2.1 Release
    - Corrects a bug where a QuantumState.load(vector) on a distributed vector does not load correctly for processes with rank numbers greater than 1.
- Documentation Updates
    - Added a description of the QuantumState load method's behavior during state vector distribution.
    - Added a way to load/get an entire distributed state vector (example code).

v1.2.0 (2022/05/25)

- mpiQulacs v1.2.0 Release
    - Environment variable QULACS_NUM_THREADS is provided to specify the maximum number of threads to be used by mpiQulacs (overrides OMP_NUM_THREADS).
    - In accordance with the above change of thread number control method, the contents of various configuration files for MPI execution have been changed.
        - Change the description of job.sh
        - Change the setting method of IPython Parallel for use of Jupyter
    - New FusedSWAP insertion method in QuantumCircuitOptimizer (with swap_level=2 argument). Optimizes inter-node communication by inserting a FusedSWAP gate while making possible gate processing changes.
    - Improved performance in expected value calculations

v1.1.1 (2022/05/23)

- Fix class method support on mpiQulacs API documentation
  - RandomUnitary supported, get_transition_amplitude not supported
- Added supplementary information about mpiexec execution parameters and how to determine the value of the hostfile slot.
- Add the following description to how Jupyter is run
  - How to start Jupyter outside node01
  - How to use JupyterLab
  - How to execute ssh without ssh config
  - How to set environment variables
- The name of the configuration file when using Jupyter is added to the installation method of PySCF.
- Change how to deal with cannot allocate memory in static TLS block in Troubleshooting

v1.1.0 (2022/05/16)

- mpiQulacs v1.1.0 Release
  - Support Observable, PauliOperator
  - Change update_quantum_state(…) argument to allow random seed
    - In probabilistic gates such as Measurement, random numbers are used to achieve probabilistic effects. You can specify the seed of the random number.
- qiskit-qulacs v1.1.0 Release
  - Update version of mpiQulacs used from v1.0.0 to v1.1.0
- Add instructions for installing PySCF
- Add OpenMP threading control method to job.sh sample code

v1.0.0 (2022/04/01)

Initial release

Support

Questions about how to use, bug reports, function requests, etc. are accepted at the following email address.

General contacts: fj-quantum-contact@dl.jp.fujitsu.com
Technical contacts: fj-quantum-tech-help@dl.jp.fujitsu.com