Virtual Grid Engine

User Manual

May 2019

THE INSTITUTE OF MEDICAL SCIENCE, THE

UNIVERSITY OF TOKYO

Human Genome Center

Revision History

Edition	Publication Date	Changed Contents
First Edition	May 2019	VGE User Manual First Edition

Table of Contents

1. VGE overview

Virtual Grid Engine (VGE) is a program for performing high-speed distributed parallel processing of pipeline processing to run pipeline applications on large-scale supercomputers that do not support grid engine environments.

VGE is developed using Python. It realizes parallelization through the master-worker model using MPI. The VGE processing functions include:

- Distributed processing through MPI parallel processing by master-worker type
- Socket communication with the target application to receive job contents
- Internal processing

These two processing functions perform data linkage in VGE.

VGE also has a function to restart with each pipeline task when a pipeline is terminated prematurely due to e.g. computer time restriction. This function allows efficient use of computational resources as it can be resumed from a prematurely terminated pipeline task without executing an already executed pipeline task.

Figure 1-1 is a schematic diagram of the pipeline job and VGE cooperative operation overview.

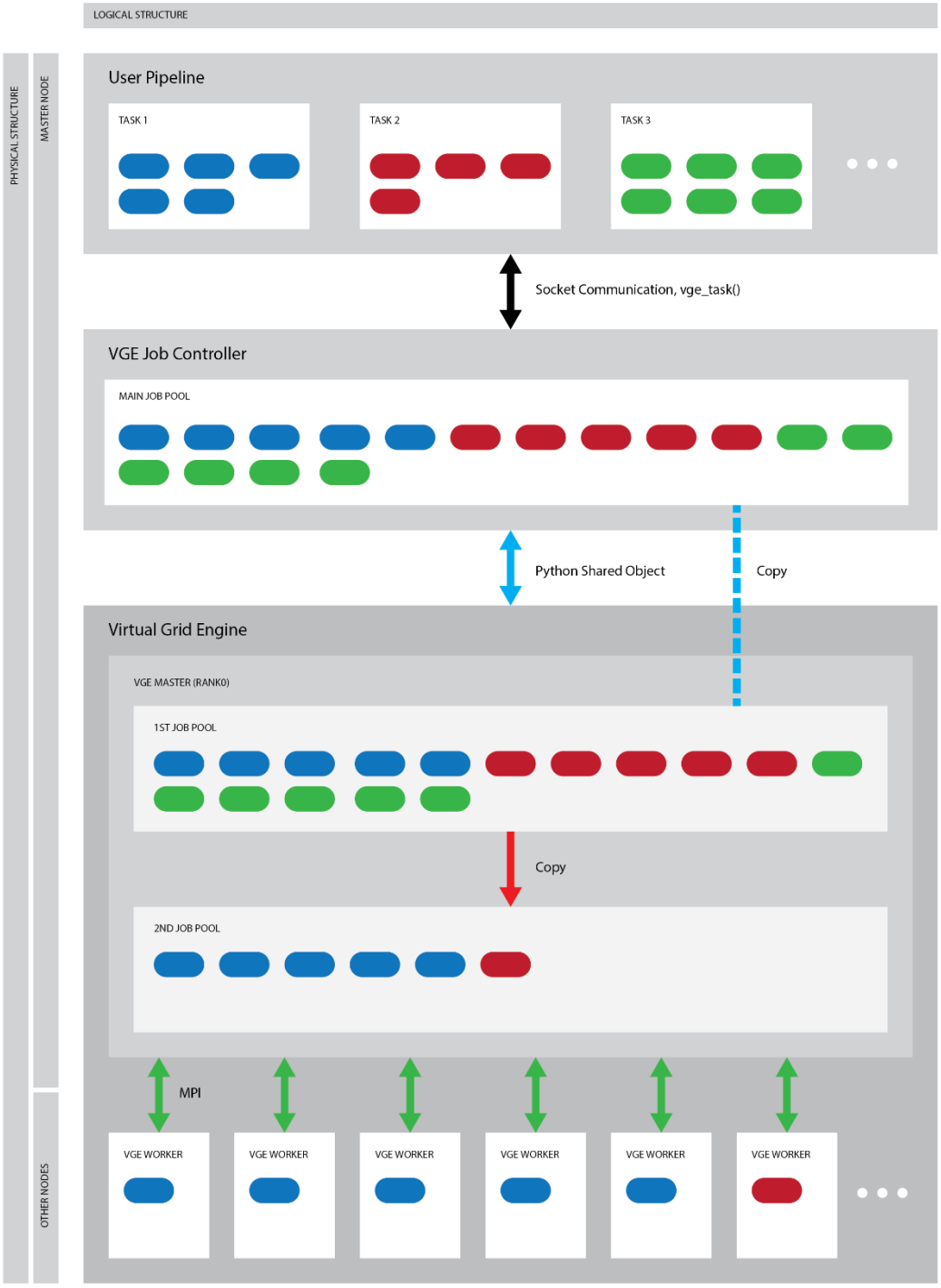

Figure 1-1: Schematic diagram of the pipeline job and VGE cooperative operation overview

The pipeline and VGE cooperate through the following procedures:

- Sending job information created by a pipeline task to the job controller via inter-process communication
- Sharing job information from the job controller to VGE through a Python shared object
- Executing a job on a compute node through an MPI-parallelized worker

2. Operating environment

Table 2- shows the operating environment of this software.

Table 2-1: Operating environment of this software

Software	OS: Linux Python: Version 2.7.11 compatible MPI: Open MPI v2.0.0 or MPICH 2.0 or later MPI4PY: Version 2.0.0 or later

3. How to install

This section describes how to install VGE.

To install VGE, use setup.py. The installation can be run as follows.

```
>cd virtual_grid_engine
>python ./setup.py install --user
```

4. How to use VGE

This chapter explains how to use VGE.

4.1. How VGE works

This section explains how VGE works.

4.1.1. How to start VGE

VGE starts through the MPI execution command (mpiexec). This is done to secure the compute node used in MPI parallel processing. The target application performs data communication with VGE by socket communication (the data is a Python structure). For this purpose, the target application needs to be run separately and independently on the same compute node as VGE after starting VGE.

4.1.2. Processing configuration

This section explains the configuration of processing. VGE processing is divided into preprocessing, processing in each rank, and end processing. Each rank is divided into the master rank and worker rank. The master rank is divided into MPI job control and pipeline job control. Figure 4- shows the VGE processing flowchart. The red arrow in Figure 4- shows data communication processing.

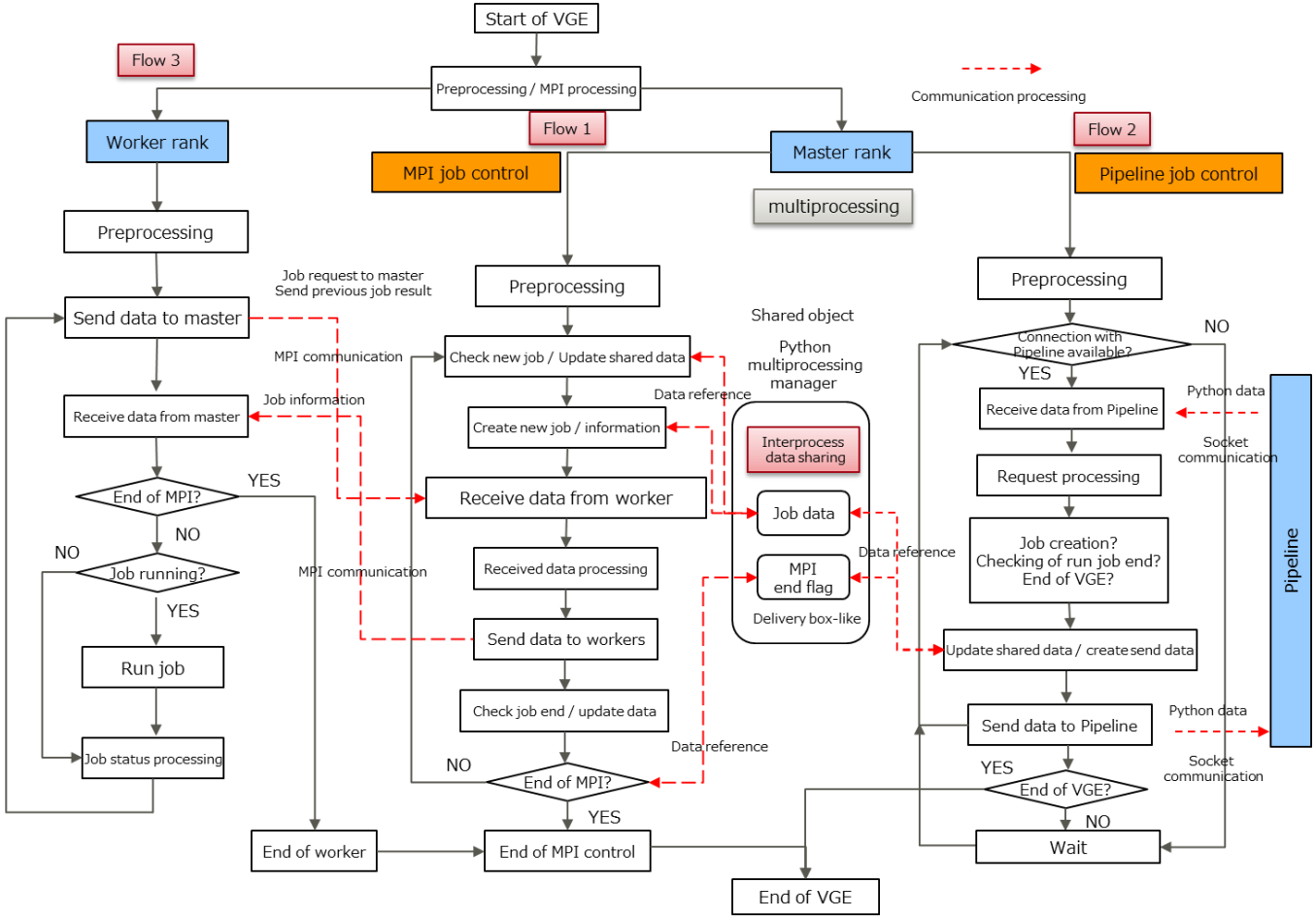

Figure 4-1: Processing flowchart of Virtual Grid Engine (VGE)

- Preprocessing

  After starting VGE, perform MPI initialization/initial variable setting as preprocessing.

- Processing in each rank

  For MPI master-worker processing, processing is divided into a master rank that controls distributed processing through MPI parallel processing and a worker rank that actually runs jobs.

  - Master rank

    The master rank shall be rank 0 of VGE MPI processes. The master rank is divided into MPI job control and pipeline job control, conducting job control of the worker rank.

  - Worker rank

    The worker rank is the rank that runs jobs according to the job information sent from the master rank.

The processing operation of the compute node assigned to the worker rank (flow 3 in Figure 4-: the VGE processing flowchart) is the same for all worker ranks. Data transmission/reception is performed by MPI communication between MPI master ranks and worker ranks. MPI communication does not occur between the VGE worker ranks.

➢ Master rank

The following explains the processing flow in the master rank. In the master rank, processing is divided into two controllers: the MPI job controller that controls distributed processing with MPI parallel processing (flow 1 in Figure 4-: the VGE processing flowchart) and the pipeline job controller that receives job data from the target application and responds to job processing of the target application (flow 2 in Figure 4-: the VGE processing flowchart). These two controllers are processed in parallel by the Python Multiprocessing module. To speed up VGE processing, exchange of internal data between the MPI job controller and the pipeline job controller in the VGE master rank is performed between the two controllers by creating a data structure that can be shared by two controllers, and updating/referring to the shared data structure at the required timing independently of each other, like a delivery box, so that the processing of the controllers does not interfere with each other. This uses the shared object of the Python Multiprocessing Manager module. The data to be shared are the data for job contents and job management and the ending flag for VGE execution (the flag for ending the MPI master and MPI worker). Hereafter, these shared data are described as “VGE shared data.”

✧ MPI job controller

The following explains the processing flow of the MPI job controller in the master rank (flow 1 in Figure 4-: the VGE processing flowchart). After starting the MPI job controller, perform preprocessing such as internal variable initialization and step into the job control

loop with MPI workers. Check the new job information from the target application by referring to the VGE shared data. Next, create job information to send to the MPI worker. If the ending flag for VGE execution is on, this job information should be signaled to end the MPI worker. If a load balancer that equalizes the number of jobs workers have is on, the system performs processing to determine the target MPI worker according to the work time or the number of jobs of the MPI workers. Receive data from the worker and perform the processing of the received data contents. Send new job information to the received worker. As the received data includes the result of the job processing previously performed by the received worker, the job data is updated in the VGE shared data. However, there is no data update since the first reception does not include job status. If the ending flag for VGE execution is on when referring to VGE shared data, the MPI job controller ends after sending an ending signal to all MPI workers. If the ending flag for execution is off, repeat the job control loop.

✧ Pipeline job controller

The following explains the processing flow of the pipeline job controller in the master rank (flow 2 in Figure 4-: the VGE processing flowchart). After starting the pipeline job controller, initialize internal variables and socket communication and step into the loop of socket communication with the target application. In the loop, wait for connection as a server of socket communication, and wait for communication connection from the target application (the signal to end VGE is sent not from the target application but from the vge_connect program). At this time, if there is no communication from the target application and the time set for timeout has been exceeded, start the server for socket communication again after the standby process. When socket communication is connected from the target application, data is immediately received from the target application. Next, perform processing of the requirements contained in the data. There are two main data contents received at this time from the target application: one is a new job from the target application, and the other is a query as to whether the job issued by the target application has ended. Creating job data structure and update processing of the VGE shared data are performed in response to these two cases. If the received data is new job data, create an ID necessary for the future query as to whether the job issued by the target application has ended, and send transmission information including the data to the target application. When a signal to end VGE is received, turn on the VGE ending flag of the VGE shared data and end processing of the pipeline job controller. When there is no VGE ending signal, return to the beginning of the socket communication loop from the target application after the standby process.

➢ Worker rank

The following explains the processing flow of the MPI worker rank of VGE (flow 3 in Figure 4-: the VGE processing flowchart). The MPI worker rank starts after MPI initialization and

initializes internal variables as preprocessing. Then step into the job working loop. Send status data of previously run jobs to the MPI master rank. If no job has been run by MPI workers before this time, the information that the worker has not done anything will be included as the job data to be sent. After sending the job status data to the MPI master rank, data is received from the MPI master rank. The contents of this data may be of three forms, as follows. The first is to finish the MPI worker, the second is the new job, and the third is that the worker has nothing to do. In the first case, the MPI worker performs an ending process. In the second case, the received job content is run. Perform job status processing after running the job, and send the data to the MPI master rank. In the third case, since there is no work to be done by the worker, immediately perform job status processing, and send the data to the MPI master rank. If the received data applies to the second or third case, the MPI worker repeats the work processing loop.

- Ending process

  VGE does not end spontaneously after starting but ends when it receives a signal from another process to end VGE. This end signal sends the VGE end signal from another external program that is different from the target application. The program that sends the end signal is vge_connect.

4.1.3. **Connection of VGE with the user program**

The user program sends a request to VGE to perform processing through the job submission process (vge_task). Figure 4- shows the flowchart of the job submission process. After creating job information such as the job script in the user program, pass the information to vge_task. In vge_task, preprocessing such as initialization of internal variables and socket communication and processing of job information received from the user program are performed. vge_task then creates a job command to be sent to VGE and connect to VGE with socket communication. When the connection between vge_task and VGE is completed, a job command is sent to VGE. After that, a job number corresponding to the job command sent by vge_task is issued in VGE, and vge_task receives the job number from VGE. After the standby process, vge_task sends the job number to VGE and queries whether the job has ended (it sends the data for the query to VGE). If the response from VGE (the data reception) indicates that the job has not ended, query again after the standby process. This loop repeats until VGE answers that the job has ended. If the job ends, check the job ending information, end vge_task, and return to the user program.

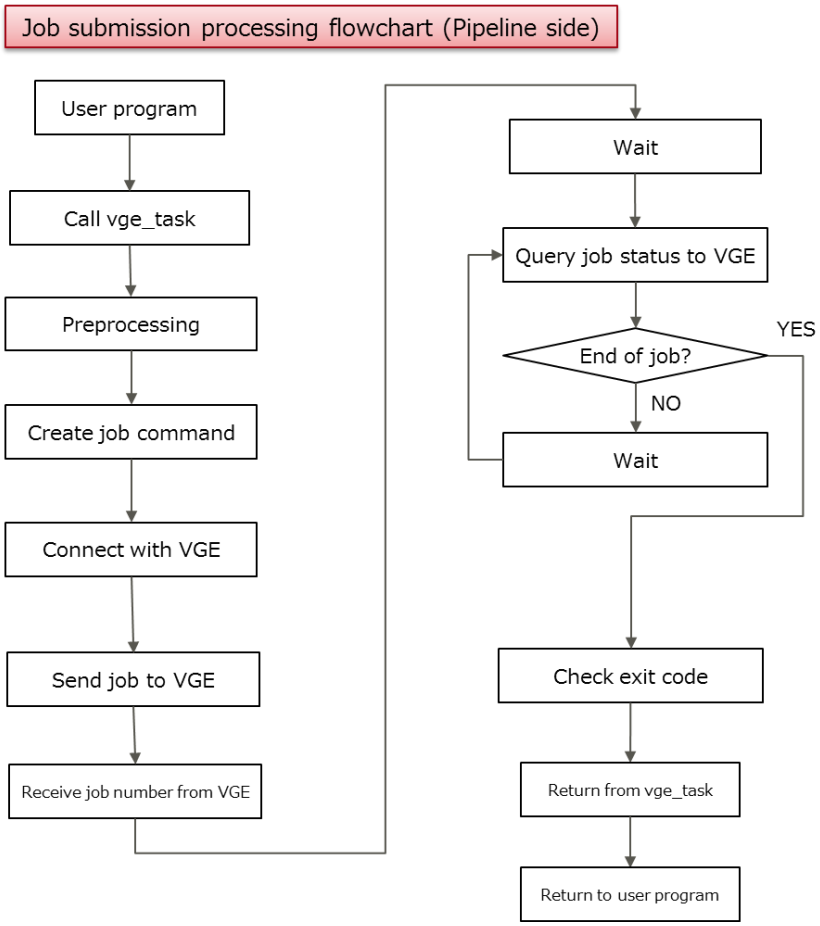

Figure 4-2: Flow chart of the job submission process using VGE

4.2. How to apply VGE

To use VGE, create the following:

- Program using VGE
- Running script

4.2.1. Program using VGE

Create a Python program to run the job.

To use VGE, invoke the following functions (details of each function will be described in Chapter 7):

1. vge_init function

   Invoked at the beginning of the main processing. Initialize VGE.

2. vge_task function

   Send the process that you want VGE to run in your program to VGE as job information.

3. vge_finalize function

Invoked at the end of the main processing. Perform VGE end processing.

The following modifications are required to use VGE.

1. Import vge module
2. Invoke vge_init at the beginning of the program
3. Specify the task information to be submitted to vge_task
4. Exit the program after invoking vge_finalize

Figure 4- is a sample program that operates a script that only outputs the string "TEST" to a file using VGE. In this sample, the Python Multiprocessing Manager module is used to submit two jobs (the job to output TEST1 and the job to output TEST2) to VGE in parallel (simultaneously). However, if performing single-job submission or serial processing (i.e. submitting the TEST2 job after the completion of the TEST1 job), the job submission can be performed by directly describing vge_task.

```
#!/bin/env python

import os
import subprocess, multiprocessing

from VGE.vge_init import *
from VGE.vge_finalize import *
from VGE.vge_task import vge_task

#
# create a job command, then submit it to VGE
#
def main(command,max_task):

    basefilename="test_"+str(os.getpid())
    vge_task(command, max_task, basefilename, "")

    return

#
# main
#
if __name__ == '__main__':

    # command
    command =""

    max_task=1

    # initialize VGE...
    vge_init()

    command1 =""
    command1 +="#!/bin/bash¥n"
    command1 += "echo TEST1 >& sample1.txt ¥n"

    command2 =""
    command2 +="#!/bin/bash¥n"
    command2 += "echo TEST2 >& sample2.txt ¥n"

    jobs=[]

    p=multiprocessing.Process(target=main, args=(command1, max_task))
    jobs.append(p)
    p.start()

    p=multiprocessing.Process(target=main, args=(command2, max_task))
    jobs.append(p)
    p.start()

    for p in jobs:
        p.join()

    # finalize VGE...
    vge_finalize()
```

Import vge modules

Specify task to be parallelized for vge_task.

Call vge_init at program start

call vge_finalize at end of program

Figure 4-3: Sample program using VGE

4.2.2. **Running script**

Create a running script with the following configuration.

1. Start VGE in the background

```
mpiexec -n 3 vge --monitor_workerjob &
```

Start VGE in the background with the MPI process using the mpiexec command. (In the example, VGE is started in master process 1 and worker process 2.) Starting VGE with MPI includes the process start on each compute node and waiting for the completion of starting.

2. Communication preparation of VGE

```
vge_connect --start
```

Wait for VGE to start and complete communication preparation with the vge_connect --start command. This command is not mandatory as it is the confirmation process of normal communication with VGE.

3. Run the program in the background

```
./vge_sample &
```

Start the target application (vge_sample in the sample program) in the background.

4. Wait for program termination

```
vge_connect --wait --monitor vge_sample
```

Run the vge_connect --wait command in the foreground. Wait for the target application to end with this command. When the target application ends, the vge_connect --wait command also ends.

5. VGE end processing

```
vge_connect --stop --target vge_sample
```

Perform the stopping process of VGE with the vge_connect --stop command. When VGE is stopped, the vge_connect --stop command also ends. At the end of the vge_connect --stop command, all VGE and vge_sample processes are gone.

Figure 4- shows an example of a running script with the above configuration. Details of each command used here will be described later (see Chapter **エラー! 参照元が見つかりません。**).

```
#!/bin/sh -x

····Preprocessing

#
# start VGE up
#
mpiexec -n 3 vge --monitor_workerjob &

#
# wait for VGE to run
#
vge_connect --start 2> start.log

./vge_sample > stdout1.log 2> stderr1.log &

#
# wait for vge_sample (s) to finish
#
vge_connect --wait --monitor vge_sample --wait_maxtime 60 --sleep 60 > wait_out.log 2>
wait_err.log

#
# stop VGE
#
vge_connect --stop --stop_maxtime 1200 > stop_out.log 2> stop_err.log

····Postprocessing
```

Start VGE as a background process

Preparation for communication with VGE

Start a target application

Wait for end of program

End process of VGE

Figure 4-4: Running script using VGE

4.3. How to see the output result of VGE

VGE processes jobs from the pipeline in parallel and creates three job information lists when VGE execution ends. The name of this file is vge_joblist.csv、vge_jobcommands.csv、vge_worker_result.csv. These files are created by default when VGE is executed, but not when executed with the --nowrite_vgecsv option. vge_joblist.csv contains the following information for each job (sub-job). The actual output will be described later (see Section 4.4).

I. Job number (in order of jobs received by VGE)
II. Job result status (exit with “done,” abort with “aborted”)
III. Time sent from pipeline to VGE
IV. Serial number in array job (bulk job)
V. Order of jobs actually run
VI. Job end time
VII. Job start time
VIII. Rank number of the worker responsible for the work

IX. Return code
X. File name
XI. Process ID of the main part of the pipeline that submitted the job
XII. Job running time (seconds)
XIII. Total number of array jobs
XIV. Job process ID (corresponding to the array job identification number)
XV. Command number (corresponding to the number of vge_jobcommands.csv)
XVI. Job number in VGE (job number on the pipeline controller)
XVII. Job transmission status to the worker (job transmission completed with “True”)

- About job number

vge_joblist.csv has four numbers: (1) Job number, (4) Serial number in array job, (15) Command number, and (16) Job number in VGE. (1) is a job serial number, (4) is a serial number closed in an array job, (15) is a number allocated for each array job, and they are all numbers used for identification by the MPI job controller; (16) is a number used for job management by the pipeline job control. If running the array job that performs job 4 running and the array job that performs job 6 running, the number of each is as Figure 4-.

	(1)	(4)	(15)	(16)
Array job 1-1	0	0	0	0
Array job 1-2	1	1	0	0
Array job 1-3	2	2	0	0
Array job 1-4	3	3	0	0
Array job 2-1	4	0	1	1
Array job 2-2	5	1	1	1
Array job 2-3	6	2	1	1
Array job 2-4	7	3	1	1
Array job 2-5	8	4	1	1
Array job 2-6	9	5	1	1

Figure 4-5: Example of numbers allocated for array jobs

vge_jobcommands.csv contains the following information for each job (sub-job).

I. Command number
II. Command contents

vge_worker_result.csv contains the following information for each job (sub-job).

I. Rank number of the worker
II. Number of jobs run
III. Total working time of workers (seconds)

These three job information list files contain VGE job processing contents and time information, which allow analysis of the pipeline job processing with this information.

4.4. Details of the output result of VGE

This section explains the details of the output result.

4.4.1. vge_joblist.csv

Figure 4- is an example of the output of the job information list (vge_joblist.csv).
The first row gives the name of each item. Item names and contents correspond as follows.

I.	jobid	Job number
II.	status	Job result status (exit with “done,” abort with “aborted”)
III.	sendvgetime	Time sent from the pipeline to VGE
IV.	bulkjob_id	Serial number in array job (bulk job)
V.	execjobid	Order of jobs actually run
VI.	finish_tiem	Job end time
VII.	start_time	Job start time
VIII.	worker	Rank number of the worker responsible for the work
IX.	returncode	Return code
X.	filename	File name
XI.	pipeline_parent_pid	Process ID of the pipeline that submitted the job
XII.	elapsed_time	Job running time (seconds)
XIII.	max_task	Total number of array jobs
XIV.	pipeline_pid	Process ID of the pipeline that submitted the job
XV.	command_id	Command number
XVI.	unique_jobid	Job number in VGE
XVII.	sendtoworker	Job transmission status to the worker

```
jobid,status,sendvgetime,bulkjob_id,execjobid,finish_time,start_time,worker,return_code,filename
,pipeline_parent_pid,elapsed_time,max_task,pipeline_pid,command_id,unique_jobid,sendtowork
er
0,done,2019-03-18 16:01:42.331488,0,0,2019-03-18 16:01:52.345544,2019-03-18
16:01:42.333507,1,0,test_31640.sh.0,31622,10.012037,1,31640,0,0,True
1,done,2019-03-18 16:01:42.331880,0,1,2019-03-18 16:01:52.346111,2019-03-18
16:01:42.334294,2,0,test_31639.sh.0,31622,10.011817,1,31639,1,1,True
```

Figure 4-6: Example of the output of the job information list (vge_joblist.csv)

4.4.2. **vge_jobcommands.csv**

Figure 4- is an example of the output of the job information list (vge_jobcommands.csv).

The first row gives the name of each item. Item names and contents correspond as follows.

The command number and the command contents are described.

I.	command_id	Command number
II.	command	Command contents

```
command_id,command
0, "#!/bin/bash¥necho TEST2 >& sample2.txt ¥n"
1, "#!/bin/bash¥necho TEST1 >& sample1.txt ¥n"
```

Figure 4-7: Example of the output of the job information list (vge_jobcommands.csv)

4.4.3. **vge_worker_result.csv**

Figure 4- is an example of the output of the job information list (vge_worker_result.csv).

The first row describes the name of each item. Item names and contents correspond as follows.

I.	worker_rank	Rank number of the worker
II.	job_count	Number of jobs run
III.	work_time	Total working time of workers (seconds)

```
worker_rank,job_count,work_time
1,1, 1.00120e+01
2,1, 1.00118e+01
```

Figure 4-8: Example of the output of the job information list (vge_worker_result.csv)

4.5. **How to write the VGE configuration file**

The following explains about the VGE configuration file.

1. Configuration file name

   vge.cfg

2. Location of the configuration file

   The priority of the location available for the configuration file is as follows.

   1. Directory for VGE execution
   2. Directory indicated by environment variable of VGE_CONF

3. Item

   The items that can be specified in the configuration file are as follows.

   - [pipeline]

     This is the configuration of the vge_task module used in the pipeline and vge_connect.

Table 4-1: Configuration of the [pipeline] item in the VGE parameter configuration file (vge.cfg)

[pipeline]	Initial value	Contents (for control of the vge_task of the pipeline)
socket_interval_after	20.0 seconds	Standby time after sending a job to VGE
socket_interval_request	20.0 seconds	Standby time after receiving the job status from VGE
socket_interval_error	1.0 seconds	Standby time after an error in socket communication
socket_interval_send	0.0 seconds	Standby time after sending a message to VGE through socket communication
socket_interval_update	0.0 seconds	Standby time after updating the shared job data through socket communication
socket_interval_close	1.0 seconds	Standby time after socket communication ends due to timeout
socket_timeout1	600.0 seconds	Timeout upper limit of socket communication when sending a job to VGE
socket_timeout2	600.0 seconds	Timeout upper limit of socket communication when querying job status
verbose	0	Information output level of vge_task.py (0: None except fatal error, 1: Information level, 3: Debug level)

   - [vge]

     This is job control configuration of VGE

Table 4-2: Configuration of the [vge] item in the VGE parameter configuration file (vge.cfg)

[vge]	Initial value	Contents (for control of VGE)
mpi_interval_update	0.0 seconds	Standby time after updating the shared job data (on the MPI job controller side)
mpi_command_size	131071 bytes	Upper limit of job script size for MPI workers to run job scripts directly in the subprocess module If the upper limit is exceeded, create and run a job script
socket_timeout	600.0 seconds	Timeout upper limit of socket communication
socket_interval_send	0.0 seconds	Standby time after sending a message to the pipeline through socket communication
socket_interval_update	0.0 seconds	Standby time after updating the shared job data (on the pipeline job controller side)
socket_interval_close	1.0 seconds	Standby time after socket communication ends due to timeout
socket_interval_error	1.0 seconds	Standby time after an error in socket communication
mpi_interval_probe	0.0001 seconds	Standby time after the result of MPI_Probe processing for MPI workers becomes FALSE
mpi_num_probe	10 times	Maximum number of iterations of MPI_Probe processing for MPI workers
worker_interval_irecv_test	30.0 seconds	Time interval at which MPI workers check the forced outage signal from the MPI master while running the job
worker_interval_runjobcheck	1.0 seconds	Time interval at which MPI workers check the job end while running a job
verbose	0	Information output level of VGE (0: None except fatal error, 1: Information level, 2: Information level only for MPI master, 3: Debug level)

- [vge_connect]

  This is for configuring the vge_connect command.

Table 4-3: Configuration of the [vge_connect] item in the VGE parameter configuration file (vge.cfg)

[vge_connect]	Initial value	Contents
connect_interval_processcheck	10.0 seconds	Time interval at which to check the target process to be monitored (valid when using standby mode, --wait option)
connect_interval_vgerequest	30.0 seconds	Time interval at which to acquire a pipeline running status list that VGE keeps (valid when using standby mode, --wait option)
connect_interval_checkvgefinish	30.0 seconds	Time interval at which to check the VGE running process (valid when using stop mode, --stop option)
verbose	0	Information output level of vge_connect (0: None except fatal error, 1: Information level, 3: Debug level)

- [socket]

  This is configuration related to socket communication.

Table 4-4: Configuration of the [socket] item in the VGE parameter configuration file (vge.cfg)

[socket]	Initial value	Contents
port	8000	Port number of socket communication
bufsize	16384 bytes	Upper limit of socket communication reception size for one time

- [restart]

  This is configuration related to the restart function.

Table 4-5: Configuration of the [restart] item in the VGE parameter configuration file (vge.cfg)

[restart]	Initial value	Contents
chk_file	2	Consistency determination mode (0: Do not perform consistency determination, 1: Perform consistency determination with file name, 2: Perform consistency determination with file size)
exclude_checkpoint_list	["", ""]	List of tasks to exclude from restart (a list whose elements are comma separated strings)

5. How to use the restart function

This chapter explains how to use the restart function. The restart function is developed to restart pipelined applications. The following describes how to use the function with a pipelined application written in Ruffus (http://www.ruffus.org.uk/).

5.1. Description of the pipeline corresponding to the restart function

This section explains how to construct a pipeline to support the restart function with the diamond-shaped pipeline shown in Figure 5-1 as an example. This pipeline has three different routes: after Task1A and Task2A of Route A are processed, Task3A of Route A and Task3B of Route B are processed. After that, they merge in Task4C of Route C in this pipeline configuration.

In this context, “task” refers to pipeline functions such as Task1A and Task2A. The length of the route will increase if the tasks branch or merge. The naming rules for each route are as follows:

- The route name is not changed unless it is branched or merged
- In case of branching, give one root the same root name as before branching and give new unique root names to the other roots
- When merging, give new routes unique names
- The route name of a route that has ended once can never be used again

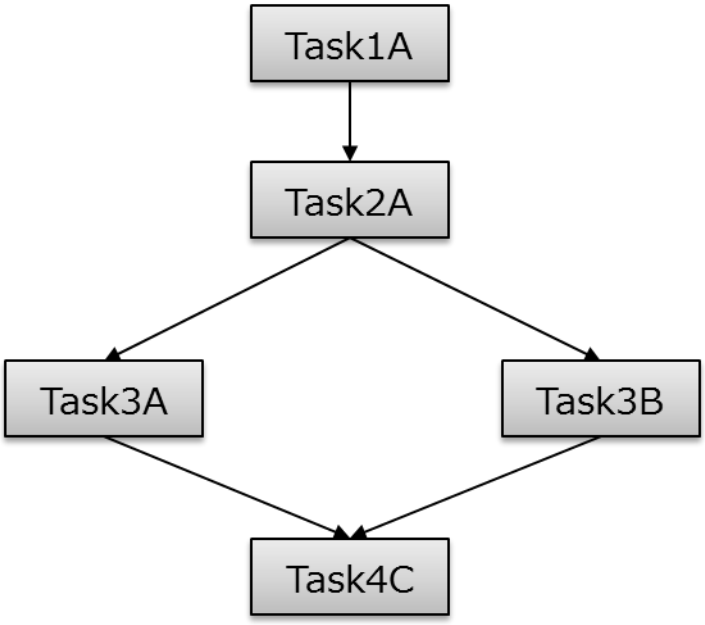

Figure 5-1 The diamond-shaped pipeline

To use the restart function, invoke the following functions:

- write_checkpoint function
  Write out the information needed to restart.

- checkpoint_restart function
  Read the information written by the write_checkpoint function and return the information needed to restart.

This paragraph explains an example of using the checkpoint_restart function and its return value, restart_files_list_dict. The checkpoint_restart function is called at the beginning of preprocessing for pipeline processing; if there is checkpoint data that was generated at the previous execution, information for restarting pipeline processing is acquired from that point as a return value. At this time, the following information is stored: a skip flag that sets a task not to be executed for the return value of flag_skiptask, a restart flag that sets a start task for the return value of flag_starttask, and an output file that requires restarting each task for the return value of restart_files_list_dict. If there are multiple processing routes, the information is stored in the list prepared for each route. Details of the checkpoint_restart function will be described later (see Section 7.5). Figure 5- shows this example.

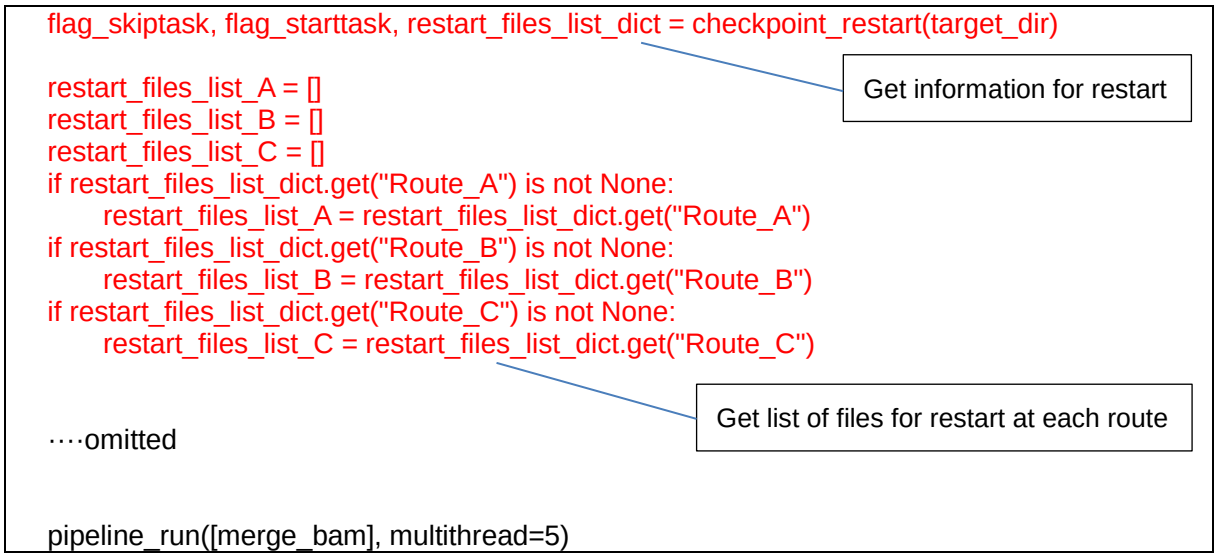

Figure 5-2: Example of using the checkpoint_restart function and its return value, restart_files_list_dict

The following paragraph explains how to describe each task in the pipeline, an example of using the write_checkpoint function, and the Ruffus decorator settings to restart. The Ruffus decorator is different depending on the type of task processing. Please refer to the Ruffus documentation (http://www.ruffus.org.uk/) to read more about the Ruffus decorator.

The main steps of describing tasks to properly construct the pipeline when restarting are as follows. Merge tasks are not considered here.

- Prepare an if statement that determines whether to execute the task according to the flag.
  There are three possible cases for each task execution:
  Case 1: Execute as the first task when restarting
  Case 2: Skip without executing the task

Case 3: Execute in the same way as in normal pipeline execution when restarting

Processing is determined by the if statement because whether or not the description of the Ruffus decorator or the task are executed changes depending on which of these cases it is.

- Change the decorator's first argument at restart for each route.

  Prepare a list of input files for each route and specify it as the first argument of the Ruffus decorator (transform, merge, etc.).
- Add functions to write out checkpoint data between existing tasks. In order to write out the checkpoint data used when restarting, add functions to write out checkpoint data after each task is completed.
- Add decorator @follows (checkpoint_task***) to the function at the normal execution.

  Since it is necessary to wait for the end of the checkpoint writing-out task, Task2A, for example, sets up the decorator to be processed after the checkpoint writing-out task (checkpoint_task1A), which is done after the Task1A.

The following shows an example of rewriting the description of Task 2A from one not supporting the restart function to one that supports the restart function according to the above rules. Figure 5- is the description before rewriting and Figure 5- is the description after rewriting. Each argument of the write_checkpoint function is shown as follows: the first argument is the task name to write out the checkpoint, the second argument is the list of task names, the third argument is the list of root names of the next task, and the fourth argument is the list of absolute paths for each output file. Details of the write_checkpoint function will be described later (see Section 7.4).

```
@transform(run_pre , suffix(".fastq"),".bam1")
def run_bwa1(input_file, output_file1):
    print "test input %s to output 1 -> %s " % (input_file, output_file1)
    fw=open(output_file1,"w")
    fi=open(input_file,"r")
    for row in fi:
        fw.write(row.strip())
        fw.write('¥n')
        print('test 1')
    fw.write('run bwa1¥n')
    print('test 1-2')
    fw.close
    fi.close
```

Figure 5-3: Task2A before changing the construction method

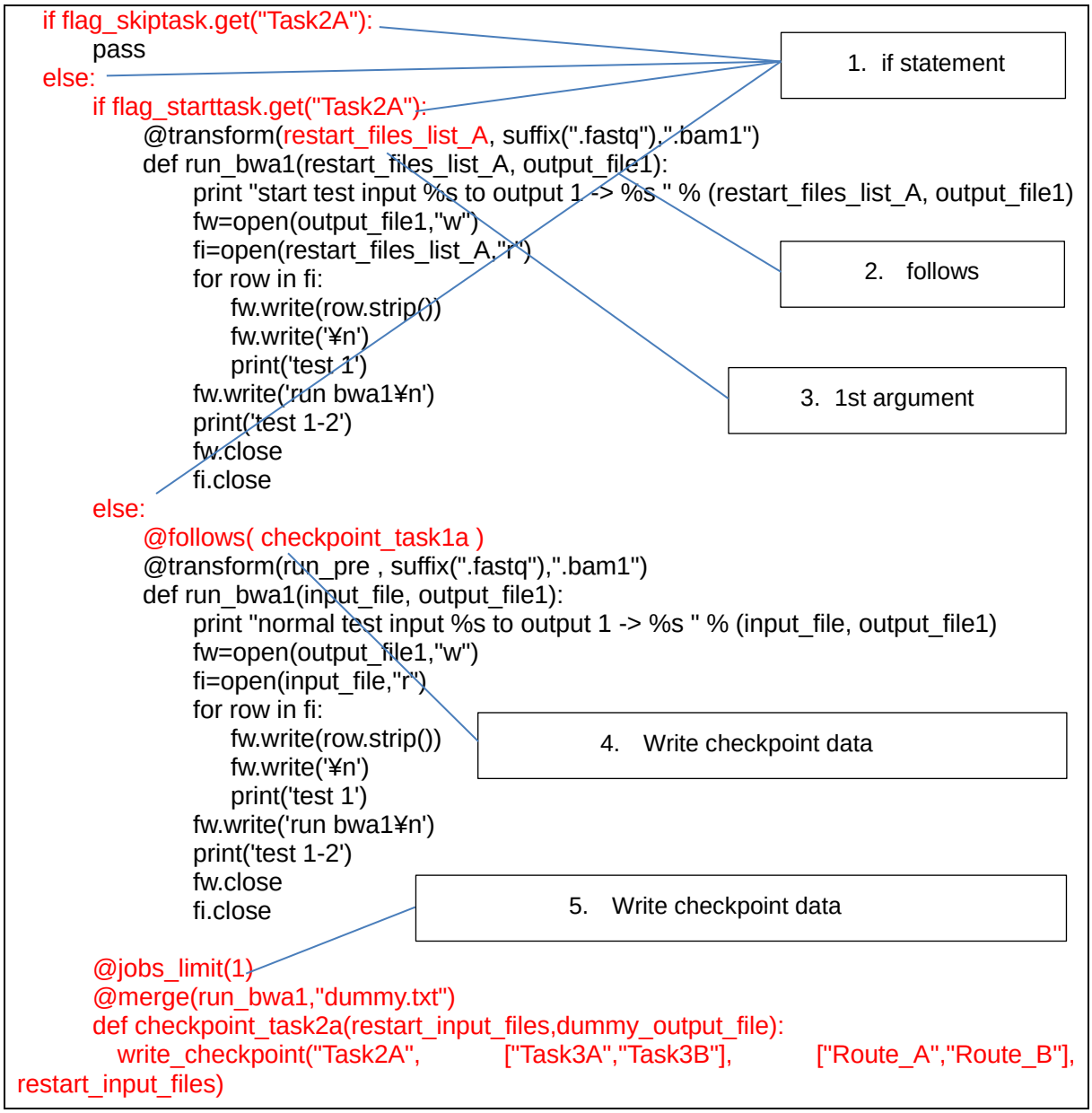

Figure 5-4: Task2A after changing the construction method

Next, rewrite the description of Task4C, which is a merge task. Figure 5- shows Task4C before rewriting and Figure 5- shows Task4C after rewriting. The merge decorator is used for confluence. For non-normal execution, specify restart_files_list_C, the list of absolute paths of the files to be given as input to the merge task, as the first argument of the merge decorator. In restart_files_list_C, a list of absolute paths of files to be given as input to Task4C is stored in advance before calling the pipeline_run function. As shown in Table 5-, the decorator to be set in Task4C changes depending on the task status directly before the merging task. Therefore, the higher the number of tasks directly before the merging task, the higher the number of branches of if statements after rewriting.

Table 5-1: Pattern of if statements in Task4C

Pattern	Status of Task3A	Status of Task3B	Task4C decorator
1	End	End	@merge(restart_files_list_C, “merge.bam”)
2	End	Incomplete	@follows(checkpoint_task3b) @merge([restart_files_list_C], “merge.bam”)
3	Incomplete	End	@follows(checkpoint_task3a) @merge([restart_files_list_C], “merge.bam”)
4	Incomplete	Incomplete	@follows(checkpoint_task3a) @follows(checkpoint_task3b) @merge([run_bwa2, run_bwa3], “merge.bam”)

```
@merge([run_bwa2,run_bwa3], "merge.bam")
def merge_bam(inputfiles, output_file4):
    fw=open(output_file4,"w")
    for input_file_name in inputfiles:
        fi=open(input_file_name,"r")
        for row in fi:
            fw.write(row.strip())
            fw.write('¥n')
        print "merge input %s " % ( input_file_name)
        fi.close
    print "merge output %s " % ( output_file4)
  fw.close
```

Figure 5-5: Task4C before changing the construction method

```
flag_skiptask, flag_starttask, restart_files_list_dict = checkpoint_restart(target_dir)
restart_files_list_A = []
restart_files_list_B = []
restart_files_list_C = []
if restart_files_list_dict.get("Route_A") is not None:
    restart_files_list_A = restart_files_list_dict.get("Route_A")
if restart_files_list_dict.get("Route_B") is not None:
    restart_files_list_B = restart_files_list_dict.get("Route_B")
if restart_files_list_dict.get("Route_C") is not None:
    restart_files_list_C = restart_files_list_dict.get("Route_C")
# A junction_filelist variable consits of output files of Task3A and Task3B.
# The output files are used in input files of Task4C at the junction of Task3A and Task3B.
# If the absolute path to the file does not exist, it should be added to restart_files_list_C.
for filepath in junction_filelist:
    filepath = os.path.abspath(filepath):
        continue
    restart_files_list_C.append(filepath)

····omitted

if flag_starttask.get("Task4C") and not flag_skiptask.get("Task3B"):
    @follows(checkpoint_task3b)
    @merge(restart_files_list_C, "outputdir/test/merge.bam")
    def merge_bam(inputfiles, output_file4):
        ····omitted

elif flag_starttask.get("Task4C") and not flag_skiptask.get("Task3A"):
    @follows(checkpoint_task3a)
    @merge(restart_files_list_C, "outputdir/test/merge.bam")
    def merge_bam(inputfiles, output_file4):
        ····omitted

elif flag_starttask.get("Task4C"):
    @merge(restart_files_list_C, "outputdir/test/merge.bam")
    def merge_bam(inputfiles, output_file4):
        ····omitted

else:
    @follows(checkpoint_task3a)
    @follows(checkpoint_task3b)
    @merge([run_bwa2,run_bwa3], "outputdir/test/merge.bam")
    def merge_bam(inputfiles, output_file4):
        ····omitted

pipeline_run([merge_bam], multithread=5)
```

Absolute path already stored

pattern 2

pattern 3

pattern 1

pattern 4

Figure 5-6: Task4C after changing the construction method

Finally, this paragraph explains the example of using the write_checkpoint function when the task is an end task. The task circled in red in Figure 5- is the end task, the task right before the merge task and the last task that is branched off the pipeline. Figure 5-８ and Figure 5- are examples of using the write_checkpoint function in each case. If there is only one route, the last task is the end task. The second argument of the write_checkpoint function of the end task shall be [“final”], and the third argument shall be a list that takes the task root name as an element.

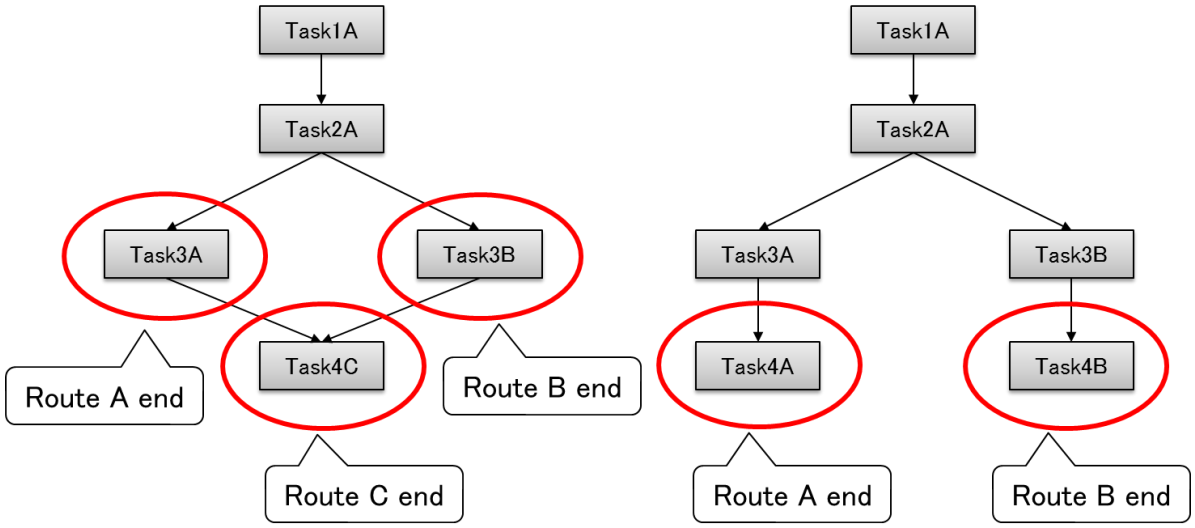

Figure 5-7: Examples of end tasks

```
@jobs_limit(1)
@merge( merge_sv, "dummy.txt")
def checkpoint_task3b(restart_input_files, dur

    command = "echo check Task3B checkpoint.... [%s]" %restart_input_files
    os.system(command)
    if len(restart_input_files) >0:
       write_checkpoint("Task3B", ["Task4C"], ["Route_C"], restart_input_files)
       write_checkpoint("Task3B", ["final"], ["Route_B"], [])
```

Call write_checkpoint twice

Figure 5-８ Example of using the write_checkpoint function directly before the merge task

```
@jobs_limit(1)
@merge( filt_sv, "dummy.txt")
def checkpoint_task4b(restart_input_files, dummy_output_file):

    command = "echo check Task4B checkpoint.... [%s]" %restart_input_files
    os.system(command)
    if len(restart_input_files) >0:
       write_checkpoint("Task4B", ["final"], ["Route_B"], restart_input_files)
```

Figure 5-9: Example of using the write_checkpoint function directly before the last branched task

5.2. Description of the output data of the restart function

The following explains checkpointrestart.py, a module related to pipeline restarting. checkpointrestart.py is a module that aims to write out data for restarting, read the written data, and restart.

The write_checkpoint function included in checkpointrestart.py creates a checkpoint directory under the output directory, writes out checkpoint data and checkpoint master files, and creates a backup directory. Table 5- provides descriptions of each directory and file. In addition, the checkpoint_restart function creates information for restarting pipeline processing from the checkpoint data generated at the previous execution.

Table 5-2: Description of terms

Term	Description
Output directory	A directory for storing the output files of the pipeline. It is generated before pipeline execution.
Output file	A file output of pipeline processing or preprocessing. Files generated by the restart function (such as checkpoint data) are excluded.
Checkpoint directory	The checkpoint directory (checkpoint) is a directory that stores data necessary for a restart. Checkpoint data, the checkpoint master file, and the backup directory are included in this directory.
Checkpoint data	Checkpoint data (“checkpoint_” + task name + “.chk”) is a file that contains information output at the end of each task and the output file information of the output directory at that time.
Checkpoint master file	The checkpoint master file (“checkpoint_” + root name + “.mst”) is a file that has the absolute path of the checkpoint data and the task name information to be restarted using that checkpoint data. Here, "root name" is the root of the task to be restarted. It includes as many routes as exist.
Backup directory	The backup directory (“backupdata_” + root name) saves the backup of output files. Here, "root name" is the root of the task to be restarted using the backup. It includes as many routes as exist.

The configuration of the output directory is as follows.

```
Output directory/
  ├ Output file
  ├ checkpoint/ (Checkpoint directory)
    ├ checkpoint_***.chk (Checkpoint data)
    ├ checkpoint_***.mst (Checkpoint master file)
    ├ backup_***/ (Backup directory)
      ├ ・・・
```

• • •

Figure 5- shows an example of the output of the checkpoint data (checkpoint_Task3A.chk).


```
S''(dp0¥nS'targetdir'¥np1¥nS'/home/user/example/output1'¥np2¥nsS'outputfiles'¥np3¥n(lp4¥nS'/
home/user/example/output1/fastq/sample_normal/1_0000.fastq_split'¥np5¥naS'/home/user/exam
ple/output1/fastq/sample_normal/2_0000.fastq_split'¥np6¥naS'/home/user/example/output1/fastq
/sample_tumor/1_0000.fastq_split'¥np7¥naS'/home/user/example/output1/fastq/sample_tumor/2_
0000.fastq_split'¥np8¥nasS'targetdirdict'¥np9¥n(dp10¥nS'filename_list'¥np11¥n(lp12¥nS'script/b
am2fastq_20190109_1126_347580.sh'¥np13¥naS'script/bam2fastq_20190109_1126_347579.sh'
¥np14¥naS'script/fastq_splitter_20190109_1126_114856.sh'¥np15¥naS'script/fastq_splitter_201
90109_1126_194861.sh'¥np16¥naS'fastq/sample_tumor/unmatched_first_output.txt'¥np17¥naS'f
astq/sample_tumor/unmatched_second_output.txt'¥np18¥naS'fastq/sample_tumor/single_end_o
utput.txt'¥np19¥naS'fastq/sample_tumor/1_0000.fastq_split'¥np20¥naS'fastq/sample_tumor/2_00
00.fastq_split'¥np21¥naS'fastq/sample_tumor/fastq_line_num.txt'¥np22¥naS'fastq/sample_norm
al/unmatched_first_output.txt'¥np23¥naS'fastq/sample_normal/unmatched_second_output.txt'¥n
p24¥naS'fastq/sample_normal/single_end_output.txt'¥np25¥naS'fastq/sample_normal/1_0000.fa
stq_split'¥np26¥naS'fastq/sample_normal/2_0000.fastq_split'¥np27¥naS'fastq/sample_normal/fa
stq_line_num.txt'¥np28¥naS'bam/reference5/reference5.markdup.bam'¥np29¥naS'bam/referenc
e5/reference5.markdup.bam.bai'¥np30¥naS'bam/reference10/reference10.markdup.bam'¥np31¥
naS'bam/reference10/reference10.markdup.bam.bai'¥np32¥naS'bam/reference7/reference7.mar
kdup.bam'¥np33¥naS'bam/reference7/reference7.markdup.bam.bai'¥np34¥naS'bam/reference6/r
eference6.markdup.bam'¥np35¥naS'bam/reference6/reference6.markdup.bam.bai'¥np36¥naS'ba
m/reference4/reference4.markdup.bam'¥np37¥naS'bam/reference4/reference4.markdup.bam.bai
'¥np38¥naS'bam/reference3/reference3.markdup.bam'¥np39¥naS'bam/reference3/reference3.m
arkdup.bam.bai'¥np40¥naS'bam/reference2/reference2.markdup.bam'¥np41¥naS'bam/reference
2/reference2.markdup.bam.bai'¥np42¥naS'bam/reference1/reference1.markdup.bam'¥np43¥naS'
bam/reference1/reference1.markdup.bam.bai'¥np44¥naS'bam/reference9/reference9.markdup.b
am'¥np45¥naS'bam/reference9/reference9.markdup.bam.bai'¥np46¥naS'bam/reference8/referen
ce8.markdup.bam'¥np47¥naS'bam/reference8/reference8.markdup.bam.bai'¥np48¥naS'mutation
/control_panel/panel1.control_panel.txt'¥np49¥naS'sv/config/panel1.control.yaml'¥np50¥nasS'siz
e_list'¥np51¥n(lp52¥nI1362¥naI1369¥naI2012¥naI2006¥naI0¥naI0¥naI0¥naI21236823¥naI2123
6823¥naI6¥naI0¥naI0¥naI0¥naI21251939¥naI21251939¥naI6¥naI4677658¥naI8280¥naI461516
3¥naI53192¥naI4617939¥naI8280¥naI4639312¥naI8280¥naI4694499¥naI8280¥naI4679155¥na
I53192¥naI4563105¥naI53192¥naI4644582¥naI8280¥naI4646589¥naI53192¥naI4679086¥naI5
3192¥naI902¥naI1173¥nass."
p0
```


Figure 5-10: Example of the output of the checkpoint data (checkpoint_Task3A.chk)

Figure 5- shows an example of the output of the checkpoint master file (checkpoint_Route_A.mst).


```
Task3A,/home/user/example/output1/checkpoint/checkpoint_Task1A.chk
Task4A,/home/user/example/output1/checkpoint/checkpoint_Task3A.chk
Task5A,/home/user/example/output1/checkpoint/checkpoint_Task4A.chk
Task6A,/home/user/example/output1/checkpoint/checkpoint_Task5A.chk
Task7A,/home/user/example/output1/checkpoint/checkpoint_Task6A.chk
final,/home/user/example/output1/checkpoint/checkpoint_Task7A.chk
```


Figure 5-11: Example of the output of the checkpoint master file (checkpoint_Route_A.mst)

6. Command details

This chapter explains the details of the commands used in VGE.

6.1. vge

The vge command is a program that starts VGE. Table 6- shows the options that can be specified when executing the vge command.

Table 6-1: Options that can be specified when executing the vge command

Option name	Parameter	Contents
--loadbalancer	none	If only –loadbalancer is specified, the same operation as --loadbalancer time is performed.
	time	Turn on the load balancer function and balance job processing by the working time of the MPI workers.
	count	Turn on the load balancer function and balance job processing by the number of jobs of the MPI workers.
	off	Explicitly turn off the load balancer function (the load balancer function is off by default if “--loadbalancer” is not specified).
--schedule	none	If only –schedule is specified, the same operation as --schedule first is performed.
	first	Sequentially execute jobs listed in the job waiting list in the order of arrival to the VGE (the array of jobs is all deployed jobs).
	sample	Process jobs in the job waiting list in the order of the sample number.
	arrayjob	Process sequential jobs in the order of the array job number.
	mix	Process jobs in a mixed order of sample number and array job number.
--nowrite_jobscript	none	When this argument is specified, the job script for VGE processing and normal output result/error output result are not output to the file.
--nowrite_vgecsv	none	When this argument is specified, the VGE job processing information list in CSV file format is not output.
-o	directory name	Specify the directory for outputting the job script and job information list. The default is vge_output under the VGE execution directory.
--check_mpi_probe	none	Perform MPI IPROBE test to the MPI workers.
--monitor_workerjob	none	VGE workers execute a mode to monitor running jobs. This is an option for the forced termination function.

6.2. vge_connect

The vge_connect command is a program that controls VGE standby/stop. Table 6- shows the options that can be specified when executing the vge_connect command.

Table 6-2: Options that can be specified when executing the vge_connect command

Option name	Parameter	Contents
--start	none	Send an order (message) to stand by for VGE set-up to VGE.
--wait	none	Specify the mode (standby mode) to monitor/stand by for the process until the pipeline executed in the background terminates.
--sleep	0 seconds	An option valid only for standby mode. Specify the standby time (unit: seconds) before monitoring the target process. The default setting is 0 seconds, which is no standby time.
--wait_maxtime	0 seconds	Maximum standby time (unit: seconds) for the standby mode. The default setting is 0 seconds, which is unlimited. Stand by for the target process running on the same node as vge_connect to end.
--monitor	'pipeline'	Specify the target process name to monitor the process to end.
--stop	none	Send an order (message) to stop VGE to VGE.
--force	none	Specify the forced termination of VGE.
--target	'pipeline'	An option valid only when using the VGE stop option. Execute processing to delete the process with the process name specified by this option before sending a VGE stop order to VGE.
--stop_maxtime	600 seconds	An option valid only in VGE stop mode. The maximum time to stand by for the VGE process of the VGE master node to end after sending a VGE stop signal to VGE.

6.3. **cleanvge**

The cleanvge command is a program that removes VGE-related processes. Table 6- shows the options that can be specified when executing the cleanvge command.

Table 6-3: Options that can be specified when executing the cleanvge command

Option name	Parameter	Contents
--verbose	1	Information output level of cleanvge (0: None except fatal error, 1: Information level, 3: Debug level).

7. Details of the function set on the user side

This chapter explains the details of the function set on the user side.

7.1. vge_init

The vge_init function is a function that reads the VGE configuration file and sends a hello message to VGE. The hello message is a message to notify VGE that the pipeline job process has started. The interface of the vge_init function is shown below.

Table 7-1: The interface of the vge_init function

I/O	Argument name/ Return value name	Type	Value
Argument	none	-	-
Returned value	-	int	0: Normal end, other than 0: Abnormal end

7.2. vge_finalize

The vge_finalize function is a function that sends a goodbye message to VGE. The goodbye message is a message to notify VGE that the pipeline job process has ended. The interface of the vge_finalize function is shown below.

Table 7-2: The interface of the vge_finalize function

I/O	Argument name/ Return value name	Type	Value
Argument	none	-	-
Return value	-	int	0: Normal end, other than 0: Abnormal end

7.3. vge_task

The vge_task function is a function that performs inter-process communication with VGE and sends the process that you want VGE to run in your program to VGE as job information. The interface of the vge_task function is shown below.

Table 7-3: The interface of the vge_task function

I/O	Argument name/ Return value name	Type	Value
Argument	arg1	String	Command to be executed
	arg2	int	Maximum number of tasks
	arg3	String	Basic file name
	arg4	Dictionary	Dictionary for changing parameters of the vge.conf [pipeline].
Return value	-	int	0: Normal end, other than 0: Abnormal end

7.4. write_checkpoint

The write_checkpoint function is a function that writes out checkpoint data used for the restart function and the checkpoint master file and creates a backup. checkpoint_tag and output_files are used to create checkpoint data. checkpoint_tag_next_list is used to create the checkpoint master file. checkpoint_route_tag_next_list is used to create the backup directory and checkpoint master file. The interface of the write_checkpoint function is shown below.

Table 7-4: The interface of the write_checkpoint function

I/O	Argument name/ Return value name	Type	Value
Argument	checkpoint_tag	String	Task name
	checkpoint_tag_next_list	list	List of subsequent task names
	checkpoint_route_tag_next_list	list	List of the route names of subsequent tasks
	output_files	list	A list of the absolute paths of each output directory
Return value	-	int	0: Normal end, 1: Abnormal end

7.5. checkpoint_restart

The checkpoint_restart function is a function that sets global variables used when writing out checkpoint for the restart function, creates a flag to skip or not skip tasks and a flag to indicate the start task when restarting, and creates a variable for input to the task to be restarted. target_dir is used to create a global variable in preparation and relocate the backup directory. flag_skiptask is used as a flag to skip when restarting a task that has already ended at the previous execution. flag_starttask is used as a flag to indicate the task at the restart point. restart_files_list_dict is used as input file information

necessary for restarting each task for each route. The interface of the checkpoint_restart function is shown below.

Table 7-5: The interface of the checkpoint_restart function

I/O	Argument name/ Return value name	Type	Value
Argument	target_dir	String	Absolute path of the output directory
Return value	flag_skiptask	Dictionary	Key: task name Element: bool value
	flag_starttask	Dictionary	Key: task name Element: bool value
	restart_files_list_dict	Dictionary	Key: route name Element: a list of absolute paths of files given as input for restarting of the task corresponding root

8. FAQ

Q1. Can an application that runs on VGE be added while a VGE job is running?

A1. It can be added if the application that calls vge_task can be run on the compute node that started VGE. If the same compute node cannot be used due to e.g. the job system, it cannot be added.

Q2. If one application runs multiple applications, is it possible to stop specific applications?

A2. Individual applications cannot be stopped. In such a case, stop all applications by stopping VGE.

Q3. Is it possible to deter the restart function (try not to output the file) when running the application for which the restart function call is set?

A3. The restart function cannot be deterred. Run the application after modifying it to not call the restart function call.

Q4. Is it possible to use the restart function for applications not using Ruffus?

A4. Yes. It can be used.

Q5. What happens if multiple users start VGE with the same compute node?

A5. If different users start VGE using the same port number, a connection with an unintended user process or an error may occur. In an environment in which multiple users use VGE, set the port numbers to be unique before executing VGE.

Q6. Why is mpiexec run in the background?

A6. mpiexec is run in the background to start the target application while VGE is started.

Q7. Why are there two locations for VGE configuration files?

A7. There are two locations to improve the convenience for users.
For example, the following options can be considered: use a specific file depending on the environment variable when multiple users use the same setting; use the runtime directory when using personal settings (especially port numbers).

Q8. Regarding the verbose setting, why is 2 sometimes missing?

A8. It is common that the verbose setting has 1 = information level and 3 = debug level. 2 is a setting only for VGE. This is for distinguishing between the output of only the master only and the output of both the master and the worker, since VGE operates multiple processes as a worker.

Q9. Is a checkpoint directory created each time the restart function is used?

A9. A directory is only created at the first checkpoint. Within the same job, the same directory is updated each time the checkpoint is performed. Checkpoint updates the checkpoint directory on restarting and does not create a new directory.

Q10. What happens if the target checkpoint directory does not exist at restarting?

A10. It is determined that there is no data to restart in the checkpoint_restart function, and normal execution is performed.

9. Operating precautions

This chapter explains operating precautions.

9.1. MPI-related limitations

In environments in which OpenFabrics is used as the interconnect driver used by MPI, a fork may not be supported. Some processes are executed in parallel by a fork using a Multiprocessing Manager module in VGE. For this reason, in an environment in which a fork is not supported as described above, it may not work properly.

9.2. Limitations of the restart function

The restart function may not work properly due to the nature of task processing, system failure, or user error. Therefore, the following points need to be noted.

- If the pipeline process is interrupted, there is a possibility that the file will be left in a halfway state due to the progress of the task. As the restart is done from the beginning of the interrupted task, this file can have an impact on the results. In this case, the file needs to be deleted by the user.
- If the status of the checkpoint master file changes, there may be variance between the contents, the status of the output directory, and the actual status of checkpoint creation. Therefore, if the checkpoint master file is deleted or edited, or if the checkpoint master file is corrupted, unexpected behavior may occur.

9.3. VGE process operation node

The process that processes the pipeline and the VGE master process run on the same compute node. If processing with a heavy CPU load is executed on the pipeline side, it may have an impact on processing performance on the VGE side. It is recommended that processing with a heavy CPU load is executed as a pipeline job.

9.4. Use of the cleanvge command

Processes may remain after an abnormal end of VGE. If a process remains, it can be deleted by using the cleanvge command. However, the node assigned by MPI execution processing may share the same node with a different batch job script. Therefore, in the case in which another pipeline process is executed when running the cleanvge command, the computational resources of operating the cleanvge command are secured to the node where the pipeline processing is running, which may delete the pipeline process that you do not want to delete. For that reason, use the cleanvge command in situations in which the user is not executing VGE-related processing.

Figure 9- shows the running script using cleanvge. In the running script, start cleanvge in the foreground through an MPI process using the mpiexec command.

```
#!/bin/sh -x

····omitted

#
# start cleanvge up
#
mpiexec -n 3 cleanvge --verbose 1

····omitted
```

Figure 9-1: Running script using cleanvge