Virtual Grid Engine

ユーザーマニュアル

2019 年 3 月

東京大学　医科学研究所

ヒトゲノム解析センター

更 新 履 歴

版	発行日	変更内容
第 1 版	2019 年 3 月	・VGE ユーザーマニュアル 初版

目次

１．VGE の概要

Virtual Grid Engine (VGE)とはグリッドエンジン環境をサポートしていない大規模スーパーコンピュータでパイプラインアプリケーションを実行するためにパイプライン処理を高速に分散並列処理するためのプログラムです。

VGE は Python を用いて開発されたプログラムです。MPI を使ったマスターワーカーモデルにより並列化を実現しています。VGE の処理機能には以下があります。

- マスターワーカー形式による MPI プロセス並列で分散処理する機能
- 対象アプリケーションとソケット通信を行ってジョブ内容を受け取り、内部処理する機能

これらの２つの処理機能は VGE 内部でのデータ連携を行います。

また、VGE にはコンピュータの時間制限などによりパイプラインが途中終了した場合に、パイプラインタスク単位でリスタートする機能を備えています。この機能を使用すると、既に実行されたパイプラインタスクを実行せずに、途中終了したパイプラインタスクから再開させることができるので、計算資源を効率よく使用できます。

図 1-1 はパイプラインジョブと VGE の連携動作概要の模式図です。

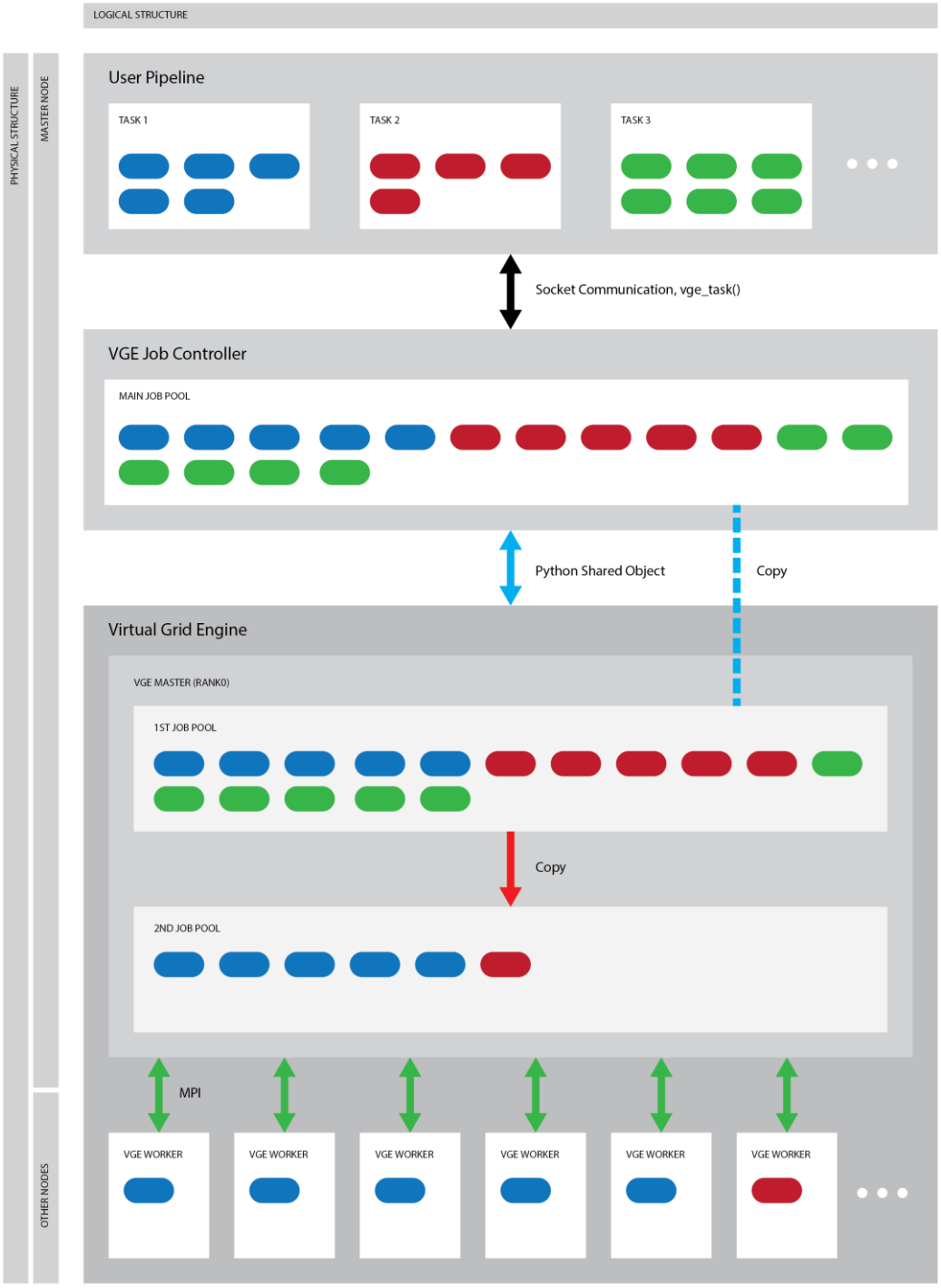

図 1-1 パイプラインジョブと VGE の連携動作概要の模式図

Pipeline と VGE は以下の手順で連携を行います。

- Pipeline の Task が作成した job の情報をジョブコントローラにプロセス間通信で送信
- ジョブコントローラから VGE 本体に Python 共有オブジェクトを用いてジョブ情報を共有
- MPI プロセス並列化された worker によって計算ノードにて job を実行

2. 動作環境

表 2-1は本ソフトウェアの動作環境です。

表 2-1　本ソフトウェアの動作環境

ソフトウェア	OS：Linux Python：Version 2.7.11 互換 MPI：Open MPI v2.0.0 または MPICH 2.0 以上 MPI4PY: Version 2.0.0 以上

3. インストール方法

インストール方法について説明します。
VGE のインストールには setup.py を用いてください。インストール例は以下のとおりです。

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

４. VGE の使い方

本章では VGE の使い方について説明します。

４.１. VGE の動作

本節では VGE の動作について説明します。

４.１.１. VGE の起動方法

VGE は、MPI 実行コマンド mpiexec により起動します。これは VGE が MPI プロセス並列により使用する計算ノードを確保するために行います。対象アプリケーションは VGE とソケット通信によりデータ通信を行います（データは Python 構造です）。このために、VGE の起動後に対象アプリケーションを VGE と同じ計算ノードで別途、独立に実行させる必要があります。

４.１.２. 処理の構成

処理の構成について説明します。VGE の処理は前処理、各ランクでの処理、終了処理に分かれます。各ランクはマスターランクとワーカーランクで処理が分かれます。マスターランクでは MPI ジョブコントロール、Pipeline ジョブコントロールに分かれます。VGE の処理フローチャートが図 ４-１です。図 ４-１の赤色矢印は、データ通信処理を示します。

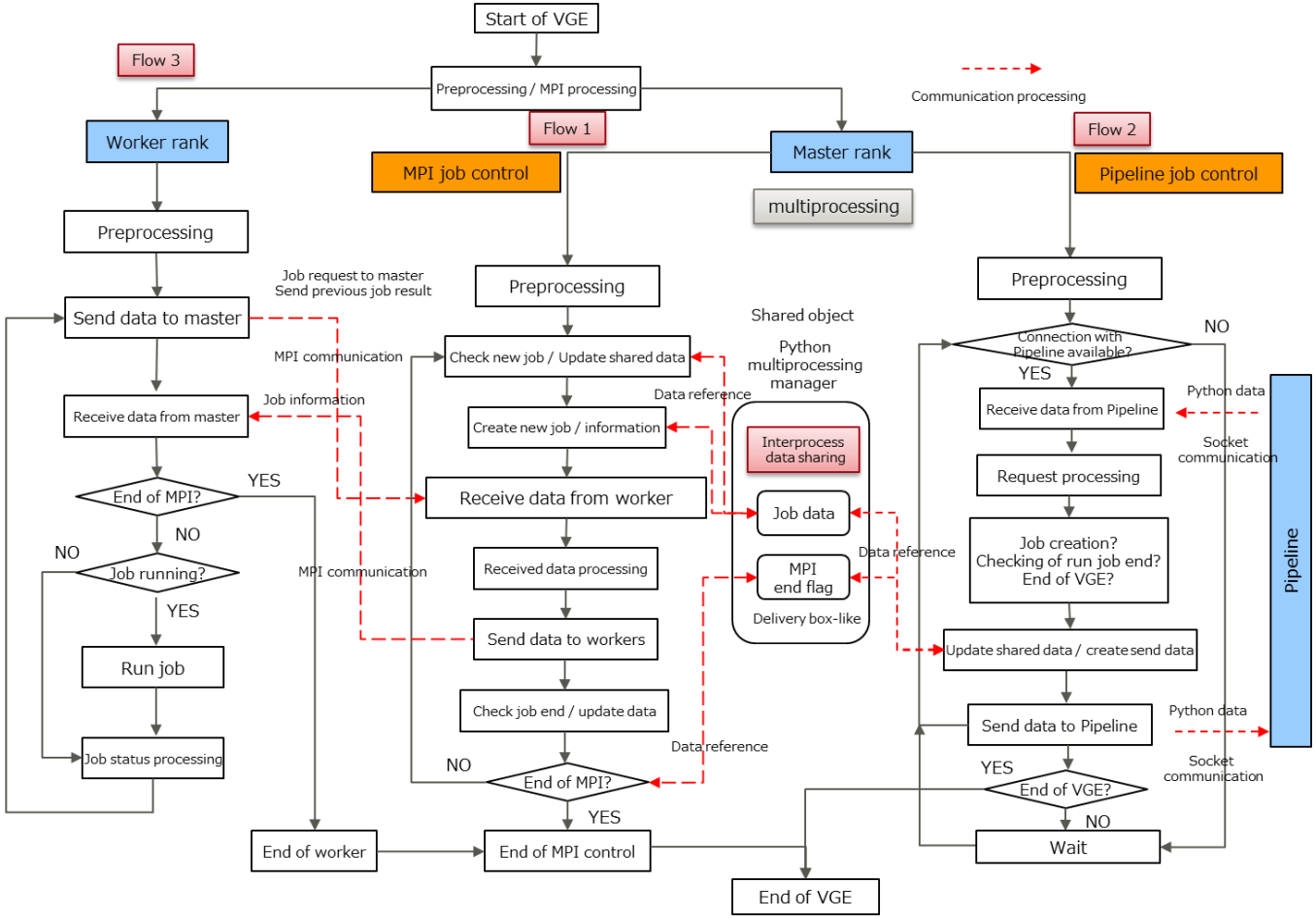

図 4-1 バーチャルグリッドエンジン(VGE)の処理フローチャート

- 前処理
  VGE 起動開始後に前処理として MPI 初期化・初期変数設定を行います。
- 各ランクでの処理
  MPI によるマスターワーカー処理のため、MPI プロセス並列で分散処理をコントロールするマスターランクと実際にジョブを実行するワーカーランクに処理が分かれます。

・マスターランク
  MPI のランク 0 のプロセスをマスターランクとします。マスターランクは MPI ジョブコントローラと Pipeline ジョブコントローラに処理が分かれ、ワーカーランクのジョブのコントロールを行います。

・ワーカーランク
  マスターランクから送られたジョブ情報に従ってジョブの実行を行うランクです。

ワーカーランクにアサインされた計算ノードの処理動作（図 4-1 VGE のフローチャートのフロー３）は、すべてのワーカーランクで同じです。MPI のマスターランクとワーカーランク間では MPI 通信によるデータの送受信が行われます。VGE のワーカーランク間の MPI 通信は発生しません。

- マスターランク
  マスターランクでの処理フローについて説明します。マスターランクでは MPI プロセス並列で分散処理をコントロールする MPI ジョブコントローラ（図 4-1 VGE のフローチャートのフロー１）と、対象アプリケーションからジョブデータを受信し、対象アプリケーションのジョブ処理を応答するためのコントローラ（Pipeline ジョブコントローラ、図 4-1 VGE のフローチャートのフロー２）の２つのコントローラに処理が分かれます。これら２つのコントローラは Python Multiprocessing モジュールにより並列処理されます。VGE の処理高速化のために、VGE マスターランクの MPI ジョブコントローラと Pipeline ジョブコントローラとの内部データのやりとりについてはお互いのコントローラの処理を干渉させないように、２つのコントローラが共有できるデータ構造を作成し、宅配ボックスのようにお互いのコントローラが独立にそれぞれ必要なタイミングにおいて、共有データ構造を更新・参照して、２つのコントローラ間でデータをやり取りします。これには Python Multiprocessing Manager モジュールの共有オブジェクトを使用しています。共有するデータは、ジョブ内容およびジョブ管理のためのデータと VGE 実行終了用フラグ（MPI マスターと MPI ワーカーを終了させるためのフラグ）です。以下で、これらの共有データを VGE 共有データと呼びます。

  - MPI ジョブコントローラ
    マスターランクの MPI ジョブコントローラの処理フロー（図 4-1 VGE のフローチャートのフロー１）を説明します。MPI ジョブコントローラ起動後、内部変数初期化などの前処理を行い、MPI ワーカーとのジョブコントロールループに入ります。VGE 共有データを参照して対象アプリケーションからの新しいジョブ情報を確認します。次に、

MPI ワーカーに送るジョブ情報を作成します。VGE 実行終了フラグがオンの場合は、このジョブ情報には MPI ワーカーを終了させる信号を入れます。ワーカーのジョブ量を均一化するロードバランサーがオンになっていれば、MPI ワーカーの作業時間またはジョブ回数により、ターゲットとなる MPI ワーカーを決める処理を行います。ワーカーからのデータを受信し、受信したデータ内容の処理を行います。受信したワーカーへ新しいジョブ情報を送信します。受信したデータには受信したワーカーが前に行ったジョブ処理の結果が含まれるので、ジョブデータを VGE 共有データに更新します。ただし、最初の受信ではジョブ状態が含まれないのでデータ更新はありません。VGE 共有データを参照し VGE 実行終了フラグがオンの場合は、MPI ワーカーのすべてに終了信号を送った後、MPI ジョブコントローラは終了します。実行終了フラグがオフの場合は、ジョブコントロールループを繰り返します。

✧ Pipeline ジョブコントローラ

マスターランクの Pipeline ジョブコントローラの処理フロー（図 4-1 VGE のフローチャートのフロー２）を説明します。Pipeline ジョブコントローラ起動後、内部変数やソケット通信の初期化を実行し、対象アプリケーションとのソケット通信のループに入ります。ループではソケット通信のサーバとして接続待ちを開始し、対象アプリケーションからの通信接続を待ちます（VGE を終了させる信号は対象アプリケーションからではなく vge_connect プログラムから送信されます）。このとき、対象アプリケーションからの通信がなく、タイムアウト設定時間を越えたら待機処理の後、再度ソケット通信のサーバを起動します。対象アプリケーションからソケット通信が接続されると、対象アプリケーションから直ちにデータを受信します。次にそのデータに含まれる要求事項の処理を行います。このとき対象アプリケーションから受信するデータ内容としては大きく２つあります。１つ目は対象アプリケーションからの新しいジョブです。２つ目は対象アプリケーションが出したジョブが終了したかどうかの問合せです。この２つの場合に対応してそれぞれジョブデータ構造の作成および VGE 共有データの更新処理を行います。受信したデータが新しいジョブデータの場合、後に対象アプリケーションからそのジョブが終了したかどうかの問合せに必要な ID を作成し、そのデータを含む送信情報を対象アプリケーションへ送信します。VGE を終了させる信号を受信した場合は、VGE 共有データの VGE 終了フラグをオンにして Pipeline ジョブコントローラの処理は終了します。VGE 終了信号がなければ待機処理の後、対象アプリケーションからのソケット通信ループ先頭に戻ります。

➢ ワーカーランク

VGE の MPI ワーカーランクの処理フロー（図 4-1 VGE のフローチャートのフロー３）を説明します。MPI ワーカーランクは MPI 初期化後に起動し、前処理として内部変数の初期化を行います。次にジョブの作業ループへ入ります。MPI マスターランクへ前に実行したジョブのステータスデータを送信します。この際、MPI ワーカーが何も前にジョブを実行していなければ、送信するジョブデータとしてはワーカーが何もしていないことを含む情報を入れます。ジョブステータスデータを MPI マスターランクに送信したのち、MPI マスターラン

クからデータを受信します。このデータ内容としては次の３つの内容があります。１つ目はMPI ワーカーを終了する内容、２つ目は新しいジョブ内容、３つ目はワーカーがやるべきことが何もないという内容です。１つ目の場合は、MPI ワーカーは終了する処理を実行します。２つ目の場合は、受信したジョブ内容を実行します。ジョブを実行したのちにジョブステータス処理を実行し、MPI マスターランクへデータを送信します。３つ目の場合はワーカーがやるべき処理がないので、直ちにジョブステータス処理を行い、MPI マスターランクへデータを送信します。受信したデータが２、３つ目の場合には MPI ワーカーはワーク処理のループを繰り返します。

- 終了処理

  VGE は起動後自発的に終了することはなく、別のプロセスから VGE を終了させるための信号を受信した場合に VGE を終了します。この終了信号は対象アプリケーションとは異なる別の外部プログラムから VGE 終了信号を送信します。終了信号を送るプログラムは vge_connect です。

4.1.3. VGE とユーザープログラムの接続

ユーザープログラムはジョブ投入処理(vge_task)により VGE に処理を依頼します。図 4-2はジョブ投入処理のフローチャートです。ユーザープログラムでジョブスクリプトなどのジョブ情報を作成したのち、その情報を vge_task へ渡します。vge_task 内部では、内部変数およびソケット通信の初期化およびユーザープログラムから受け取ったジョブ情報処理などの前処理を行います。その後、vge_task では VGE へ送るジョブコマンドを作成し、VGE とソケット通信接続します。vge_task と VGE の接続が完了すると、VGE へジョブコマンドを送信します。その後、VGE では vge_task が送ったジョブコマンドに対応するジョブ番号が発行されるので、vge_task は VGE からそのジョブ番号を受信します。待機処理の後、vge_task は VGE へジョブ番号を送信して、ジョブが終了したかどうか問合せします（VGE へ問合せのためのデータを送信します）。VGE からの返答（データ受信）が、そのジョブが終了していない場合は待機処理後に再度問合せをします。このループはジョブが終了したと回答が VGE からあるまで繰り返されます。ジョブが終了した場合は、ジョブ終了情報をチェックし、vge_task が終了し、ユーザープログラムへ戻ります。

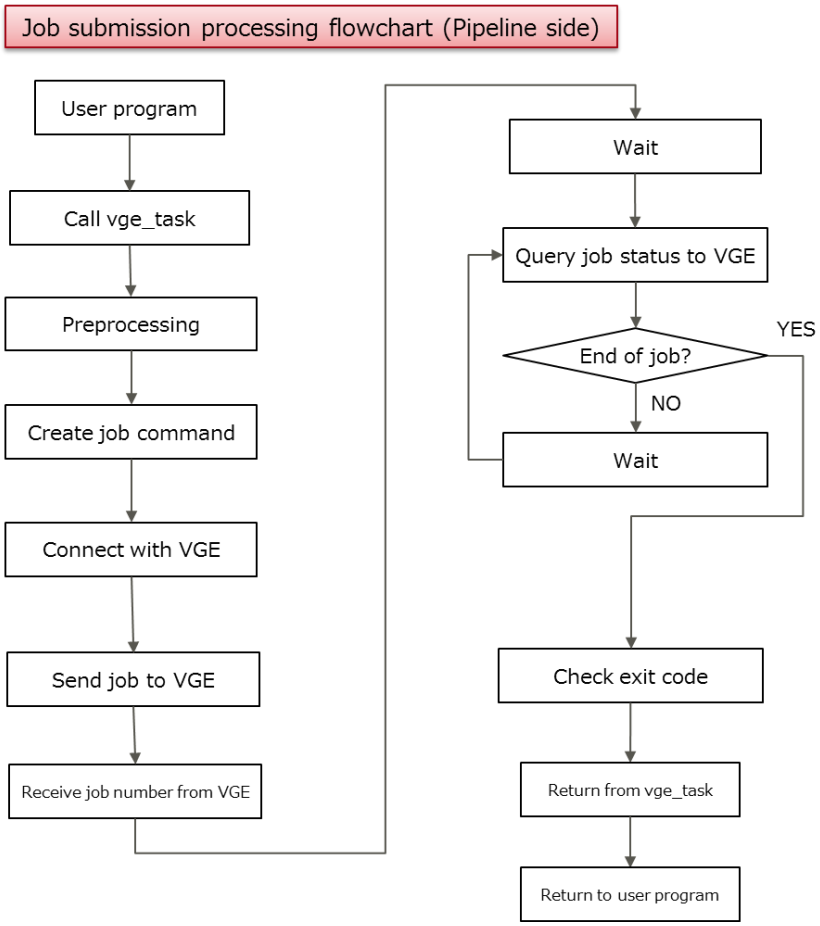

図 4-2 VGE を使用したジョブ投入処理のフローチャート

4.2. VGE の適用方法

VGE を使用するためには以下を作成する必要があります。

- VGE を使用したプログラム
- 実行スクリプト

4.2.1. VGE を使用したプログラム

ジョブを実行する Python プログラムを作成します。

VGE を使用するためには以下の関数を呼び出す必要があります。各関数の詳細は後述します（7章参照）。

1. vge_init 関数
   メイン処理の最初に呼び出す関数。VGE の初期化を行う関数。
2. vge_task 関数

ユーザープログラム内で VGE に実行させたい処理をジョブ情報として VGE に送る関数。

3. vge_finalize 関数

メイン処理の最後に呼び出す関数。VGE の終了処理を行う関数。

VGE を使うためには以下の修正が必要になります。

1. vge モジュールのインポート
2. プログラムの開始時に vge_init の呼び出し
3. vge_task に投入したいタスクの情報を指定
4. vge_finalize を呼び出してからプログラムを終了

図 4-3は VGE を使ってファイルに”TEST”の文字列を出力するのみのスクリプトを動作させるサンプルプログラムです。本サンプルでは、二つのジョブ（TEST1 の出力をするジョブ、TEST2 の出力をするジョブ）を並列に（同時に）VGE へ投入するため、Python Multiprocessing Manager モジュールを使用していますが、単一ジョブの投入または逐次処理（TEST1 の出力ジョブが完了してから TEST2 のジョブを投入する）場合には、vge_task を直接記述してジョブ投入をすることが出来ます。

```
#!/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

図 4-3 VGE を使用したサンプルプログラム

4.2.2. 実行スクリプト

以下の構成の実行スクリプトを作成します。

1. VGE をバックグラウンドで起動

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

mpiexec コマンドを用いて MPI プロセスで VGE をバックグラウンドで起動します。（例ではマスタープロセス 1、ワーカープロセス 2 で VGE を起動しています。）MPI による VGE 起動には各計算ノードでのプロセス起動や起動完了待ちを含みます。

2. VGE の通信準備

```
vge_connect --start
```

vge_connect --start コマンドにより、VGE が起動し通信準備が完了することを待ちます。このコマンドは VGE との正常通信確認処理のため、必須ではありません。

3. プログラムをバックグラウンドで実行

```
./vge_sample &
```

対象アプリケーション(サンプルプログラムでは vge_sample)をバックグラウンドで起動します。

4. プログラムの終了待ち合わせ

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

vge_connect --wait コマンドはフォアグラウンドで実行してください。このコマンドで対象アプリケーション が終了するまで待機します。対象アプリケーション が終了すると vge_connect --wait コマンドが実行終了となります。

5. VGE の終了処理

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

vge_connect --stop コマンドにより VGE の停止処理を実施します。VGE が停止すると vge_connect --stop コマンドが終了します。vge_connect --stop コマンドが終了した時点で、VGE および vge_sample のプロセスはすべてなくなります。

上記構成の実行スクリプトの例が図 4-4になります。ここで使用している各コマンドの詳細は後述します（6章参照）。

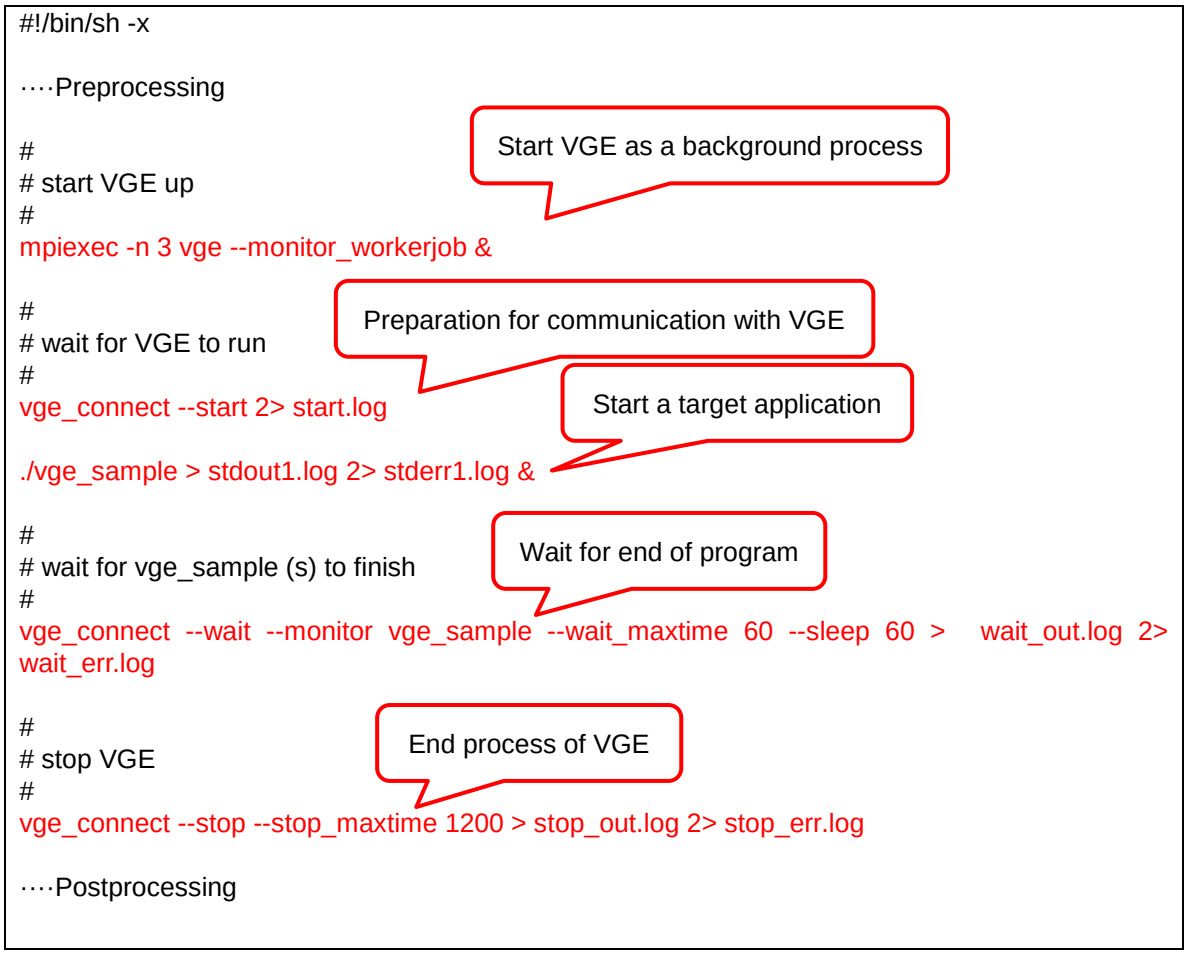

図 4-4VGE を使用した実行スクリプト

4.3. VGE の出力結果の見方

VGE は Pipeline からのジョブを並列処理し、VGE 実行終了時に３つのジョブ情報リストを作成します。このファイル名は、vge_joblist.csv、vge_jobcommands.csv、vge_worker_result.csv です。これらのファイルは VGE 実行時にデフォルトで作成されますが、--nowrite_vgecsv オプションを付加して実行した場合には作成されません。vge_joblist.csv は、各ジョブ（サブジョブ）について下記の情報が含まれます。実際の出力は後述します(4.4節参照)。

① ジョブ番号（VGE で受信されたジョブ順）
② ジョブ結果状態（done で終了、aborted で強制終了）
③ Pipeline から VGE へ送られた時刻
④ アレイジョブ（バルクジョブ）内での通し番号
⑤ 実際に実行されたジョブの順番
⑥ ジョブ終了時刻
⑦ ジョブ開始時刻

⑧　作業を担当したワーカーのランク番号
⑨　リターンコード
⑩　ファイル名
⑪　ジョブを投入した Pipeline の本体のプロセス ID
⑫　ジョブの実行時間（秒）
⑬　アレイジョブの総数
⑭　ジョブのプロセス ID（アレイジョブ識別番号に対応）
⑮　コマンド番号(vge_jobcommands.csv の番号に対応）
⑯　VGE でのジョブ番号(Pipeline コントローラ上のジョブ番号)
⑰　ワーカーへジョブ送信状態（True でジョブ送信が完了）

・ジョブの番号について

vge_joblist.csv には①ジョブ番号、④アレイジョブ内での通し番号、⑮コマンド番号、⑯VGE でのジョブ番号の４つの番号が存在します。①はジョブの通し番号、④はアレイジョブ内に閉じた通し番号、⑮はアレイジョブ単位で割り振られた番号であり、MPI ジョブコントローラ上で識別に用いられる番号、⑯は Pipeline ジョブコントロール上でのジョブ管理に用いられる番号です。４ジョブ実行するアレイジョブと６ジョブ実行するアレイジョブをそれぞれ実行しようとした場合のそれぞれの番号は図 4-5のようになります。

	①	④	⑮	⑯
アレイジョブ 1-1	0	0	0	0
アレイジョブ 1-2	1	1	0	0
アレイジョブ 1-3	2	2	0	0
アレイジョブ 1-4	3	3	0	0
アレイジョブ 2-1	4	0	1	1
アレイジョブ 2-2	5	1	1	1
アレイジョブ 2-3	6	2	1	1
アレイジョブ 2-4	7	3	1	1
アレイジョブ 2-5	8	4	1	1
アレイジョブ 2-6	9	5	1	1

図 4-5 アレイジョブ時の番号の割り振り例

vge_jobcommands.csv は、各ジョブ（サブジョブ）について下記の情報が含まれます。

①　コマンド番号
②　コマンド内容

vge_worker_result.csv は、各ジョブ（サブジョブ）について下記の情報が含まれます。

① ワーカーのランク番号
② 実行したジョブ回数
③ ワーカーの総作業時間（秒）

これら３つのジョブ情報リストファイルには、VGE のジョブ処理内容や時間の情報が含まれており、これらの情報から Pipeline のジョブ処理の分析をすることが可能です。

４.４. VGE の出力結果の詳細

本節では出力結果の詳細について説明します。

４.４.１. vge_joblist.csv

図 4-6はジョブ情報リスト（vge_joblist.csv）の出力例です。
1 行目に記載されているのは各項目名です。項目名と内容は以下のような対応となります

①	jobid	ジョブ番号
②	status	ジョブ結果状態（done で終了、aborted で強制終了）
③	sendvgetime	Pipeline から VGE へ送られた時刻
④	bulkjob_id	アレイジョブ（バルクジョブ）内での通し番号
⑤	execjobid	実際に実行されたジョブの順番
⑥	finish_tiem	ジョブ終了時刻
⑦	start_time	ジョブ開始時刻
⑧	worker	作業を担当したワーカーのランク番号
⑨	returncode	リターンコード
⑩	filename	ファイル名
⑪	pipeline_parent_pid	ジョブを投入した Pipeline の本体のプロセス ID
⑫	elapsed_time	ジョブの実行時間（秒）
⑬	max_task	ジョブが対応するアレイジョブの総数
⑭	pipeline_pid	ジョブを投入した Pipeline のプロセス ID
⑮	command_id	コマンド番号
⑯	unique_jobid	VGE でのジョブ番号
⑰	sendtoworker	ワーカーへジョブ送信状態

```
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
```

図 4-6 ジョブ情報リスト（vge_joblist.csv）の出力例

4.4.2. **vge_jobcommands.csv**

図 4-7はジョブ情報リスト（vge_jobcommands.csv）の出力例です。

1 行目に項目名が記載されています。項目名と内容は以下のような対応となります。

コマンド番号およびコマンドの内容が記載されます。

① command_id コマンド番号

② command コマンド内容

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

図 4-7 ジョブ情報リスト（vge_jobcommands.csv）の出力例

4.4.3. **vge_worker_result.csv**

図 4-8はジョブ情報リスト（vge_worker_result.csv）の出力例です。

1 行目に記載されているのは各項目名です。項目名と内容は以下のような対応となります

① worker_rank ワーカーのランク番号

② job_count 実行したジョブ回数

③ work_time ワーカーの総作業時間（秒）

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

図 4-8 ジョブ情報リスト（vge_worker_result.csv）の出力例

4.5. **VGE 設定ファイルの書き方**

VGE の設定ファイルについて説明します。

1. 設定ファイル名

   vge.cfg

2. 設定ファイルの配置場所

   設定ファイルの配置可能な場所の優先順位は以下のとおりです。

   1. VGE 実行時のディレクトリ
   2. 環境変数 VGE_CONF で示すディレクトリ

3. 項目

   設定ファイルで指定可能な項目は以下のとおりです。

   - [pipeline]

     Pipeline および vge_connect で使う vge_task モジュールの設定です。

表 4-1　VGE パラメタ設定ファイル（vge.cfg）の[pipeline]項目の設定

[pipeline]	初期値	内容（Pipeline の vge_task 制御用）
socket_interval_after	20.0 秒	ジョブを VGE へ送信した後の待機時間
socket_interval_request	20.0 秒	VGE からジョブ状態を受信した後の待機時間
socket_interval_error	1.0 秒	ソケット通信でエラーが生じた後の待機時間
socket_interval_send	0.0 秒	ソケット通信で VGE へメッセージ送信した後の待機時間
socket_interval_update	0.0 秒	ソケット通信で共有ジョブデータを更新した後の待機時間
socket_interval_close	1.0 秒	ソケット通信がタイムアウトにより終了した後の待機時間
socket_timeout1	600.0 秒	ジョブを VGE へ送信するときのソケット通信タイムアウト上限値
socket_timeout2	600.0 秒	ジョブ状態を問い合わせるときのソケット通信タイムアウト上限値
verbose	0	vge_task.py の情報出力レベル（0:致命的エラー以外なし、1:情報レベル、3:デバックレベル）

   - [vge]

     VGE のジョブ制御設定です。

表 4-2　VGE パラメタ設定ファイル（vge.cfg）の[vge]項目の設定

[vge]	初期値	内容（VGE 制御用）
mpi_interval_update	0.0 秒	共有ジョブデータを更新した後の待機時間（MPI ジョブコントローラ側）
mpi_command_size	131071 バイト	MPI ワーカーが subprocess モジュールで直接ジョブスクリプトを実行するためのジョブスクリプトのサイズ上限 上限を超える場合、ジョブスクリプトを作成して実行する
socket_timeout	600.0 秒	ソケット通信のタイムアウト上限値
socket_interval_send	0.0 秒	ソケット通信で Pipeline へメッセージ送信した後の待機時間
socket_interval_update	0.0 秒	共有ジョブデータを更新した後の待機時間（Pipeline ジョブコントローラ側）
socket_interval_close	1.0 秒	ソケット通信がタイムアウトにより終了した後の待機時間
socket_interval_error	1.0 秒	ソケット通信でエラーが生じた後の待機時間
mpi_interval_probe	0.0001 秒	MPI ワーカーへの MPI_Probe 処理の結果が FALSE になった後の待機時間
mpi_num_probe	10 回	MPI ワーカーへの MPI_Probe 処理の繰り返し最大回数
worker_interval_irecv_test	30.0 秒	MPI ワーカーがジョブを実行中に、MPI マスターからの強制停止信号を確認する時間間隔
worker_interval_runjobcheck	1.0 秒	MPI ワーカーがジョブを実行中に、ジョブ終了をチェックする時間間隔
verbose	0	VGE の情報出力レベル(0:致命的エラー以外なし、1:情報レベル、2:MPI マスターのみ情報レベル、3:デバックレベル)

- [vge_connect]
  vge_connect コマンドの設定用です。

表 4-3　VGE パラメタ設定ファイル（vge.cfg）の [vge_connect]項目の設定

[vge_connect]	初期値	内容
connect_interval_processcheck	10.0 秒	監視するターゲットプロセスをチェックする時間間隔(待機モード--wait オプション使用時有効)
connect_interval_vgerequest	30.0 秒	VGE が把握している Pipeline 実行状態リストを取得する間隔(待機モード--wait オプション使用時有効)
connect_interval_checkvgefinish	30.0 秒	VGE 稼働プロセスをチェックする時間間隔（停止モード--stop オプション使用時有効）
verbose	0	vge_connect の情報出力レベル(0:致命的エラー以外なし、1:情報レベル、3:デバックレベル)

- [socket]
  ソケット通信関連の設定です。

表 4-4　VGE パラメタ設定ファイル（vge.cfg）の[socket]項目の設定

[socket]	初期値	内容
port	8000	ソケット通信のポート番号
bufsize	16384 バイト	1 回のソケット通信受信サイズの上限

- [restart]
  リスタート機能関連の設定です。

表 4-5　VGE パラメタ設定ファイル（vge.cfg）の [restart]項目の設定

[restart]	初期値	内容
chk_file	2	整合性判定のモード（0:整合性判定を行わない、1:ファイル名で整合性判定を行う、2:ファイルサイズで整合性判定を行う）
exclude_checkpoint_list	["", ""]	リスタートから除外するタスクのリスト（カンマ区切りの文字列を要素とするリスト）

5. リスタート機能の使い方

本章ではリスタート機能の使い方について説明します。リスタート機能はパイプライン化されたアプリケーションをリスタートさせるために開発された機能です。ここではruffus(http://www.ruffus.org.uk/)によるパイプライン化をされているアプリケーションを用いた使い方を記載しています。

5.1. リスタートに対応したパイプラインの記述

本節では、リスタート機能に対応するためのパイプラインの構築方法について、図 5-1 に示すダイアモンド型のパイプラインを例として説明します。このパイプラインにはルートが３つ存在し、ルート A の Task1A、Task2A が処理された後、ルート A の Task3A とルート B の Task3B が処理されます。その後、ルート C の Task4C で合流するパイプライン構成になっています。

ここで、タスクとはパイプラインの関数のことであり、Task1A、Task2A などを指します。分岐や合流するとルートが増えます。なお、各ルートの命名規則は以下のとおりです。

- 分岐、合流しない限りは、同一のルート名とします。
- 分岐した場合は、１つのルートを分岐前のルート名と同一にし、それ以外のルートは重複しない新規のルート名とします。
- 合流した場合は、重複しない新規のルート名とします。
- １度終了したルートのルート名を再度使用することはできません。

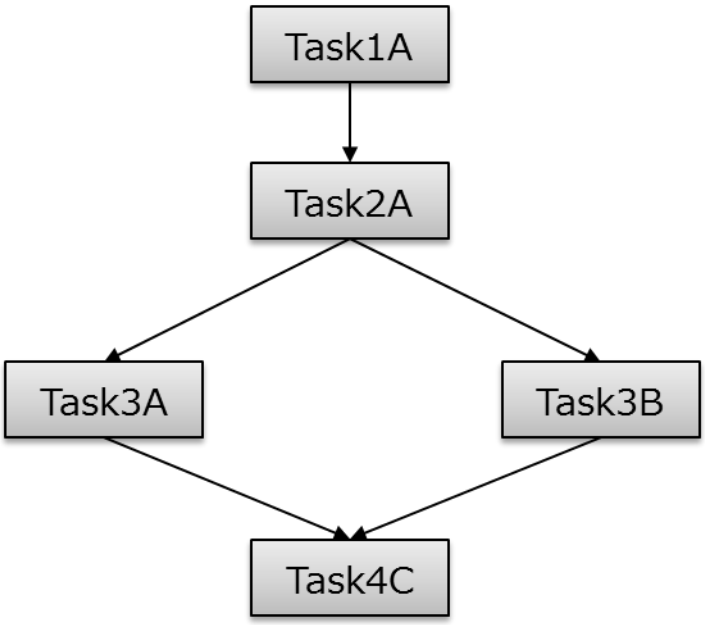

図 5-1 ダイアモンド型のパイプライン

リスタート機能を使用するためには以下の関数を呼び出す必要があります。

- write_checkpoint 関数
  リスタートするために必要な情報を書き出すための関数。
- checkpoint_restart 関数
  write_checkpoint 関数で書き出した情報を読み込み、リスタートに必要な情報を戻す関数。

まず、checkpoint_restart 関数と、その戻り値 restart_files_list_dict の使用例を説明します。checkpoint_restart 関数は、パイプライン処理に対する前処理の最初に呼ばれ、前回実行時に生成されたチェックポイントデータがあれば、そこからパイプライン処理をリスタートするための情報を戻り値として取得します。この時、戻り値 flag_skiptask には実行しないタスクを設定したスキップフラグ、戻り値 flag_starttask には開始タスクを設定したリスタートフラグ、戻り値 restart_files_list_dict には、各タスクのリスタートに必要なアウトプットファイルの情報が格納されており、複数の処理ルートがある場合には、そのルートごとに用意したリストに情報を格納します。checkpoint_restart 関数の詳細は後述します(7.5節参照)。使用例は図 5-2のとおりです。

```
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")

….omitted

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

Get information for restart

Get list of files for restart at each route

図 5-2 checkpoint_restart 関数と戻り値 restart_files_list_dict の使用例

パイプラインの各タスクの記述方法と、それに伴う write_checkpoint 関数の使用例、リスタートするための ruffus デコレータ設定について説明します。ruffus デコレータは、タスクの処理の種類によって異なるものが用いられます。ruffus デコレータについては ruffus(http://www.ruffus.org.uk/)のドキュメントを参照ください。

リスタートする際に正しくパイプラインを構築するための、タスク記述の要点は以下のとおりです。ただし、ここでは合流タスクについては考慮しません。

- フラグによってそのタスクを実行するか判定する if 文を用意します。
  １つのタスクの実行に対して、考えられるケースは以下の３つが挙げられます：
  ケース１：リスタート時の最初のタスクとして実行します
  ケース２：そのタスクを実行しないでスキップします
  ケース３：リスタート時は、通常のパイプライン実行と同じように実行します

これらのうちどのケースであるかによって、ruffus デコレータの記述、またはタスクを実行するか否かが変わるため、if 文により処理を決定します。

- リスタート時のデコレータの第一引数をルートごとに変更します。
  入力ファイルのリストをルートごとに用意し、ruffus デコレータ(transform、merge など)の第１引数に指定します。
- 既存のタスクの間に、チェックポイントデータ書き出しのための関数を追加します。リスタート時に用いるチェックポイントデータを書き出すため、各タスクの終了後にチェックポイントデータ書き出しのための関数を追加します。
- 通常実行時の関数に対して、デコレータ@follows(checkpoint_task〇〇)を追加します。
  チェックポイント書き出しタスクの終了を待つ必要があるため、例えば Task2A は、Task1A 終了後のチェックポイントデータ書き出しタスク(checkpoint_task1A)の後に処理されるようにデコレータを設定します。

例として、上記の規則に従って Task2A の記述をリスタート機能未対応のものからリスタート機能対応のものに書き換えます。図 5-3が変更前、図 5-4が変更後です。なお、write_checkpoint 関数の第１引数はチェックポイントを書き出すタスク名、第２引数は次のタスク名のリスト、第３引数は次のタスクのルート名のリスト、第４引数は各アウトプットファイルの絶対パスのリストを示します。write_checkpoint 関数の詳細は後述します(７.４節参照)。

```
@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
```

図 5-3 Task2A の構築方法の変更前

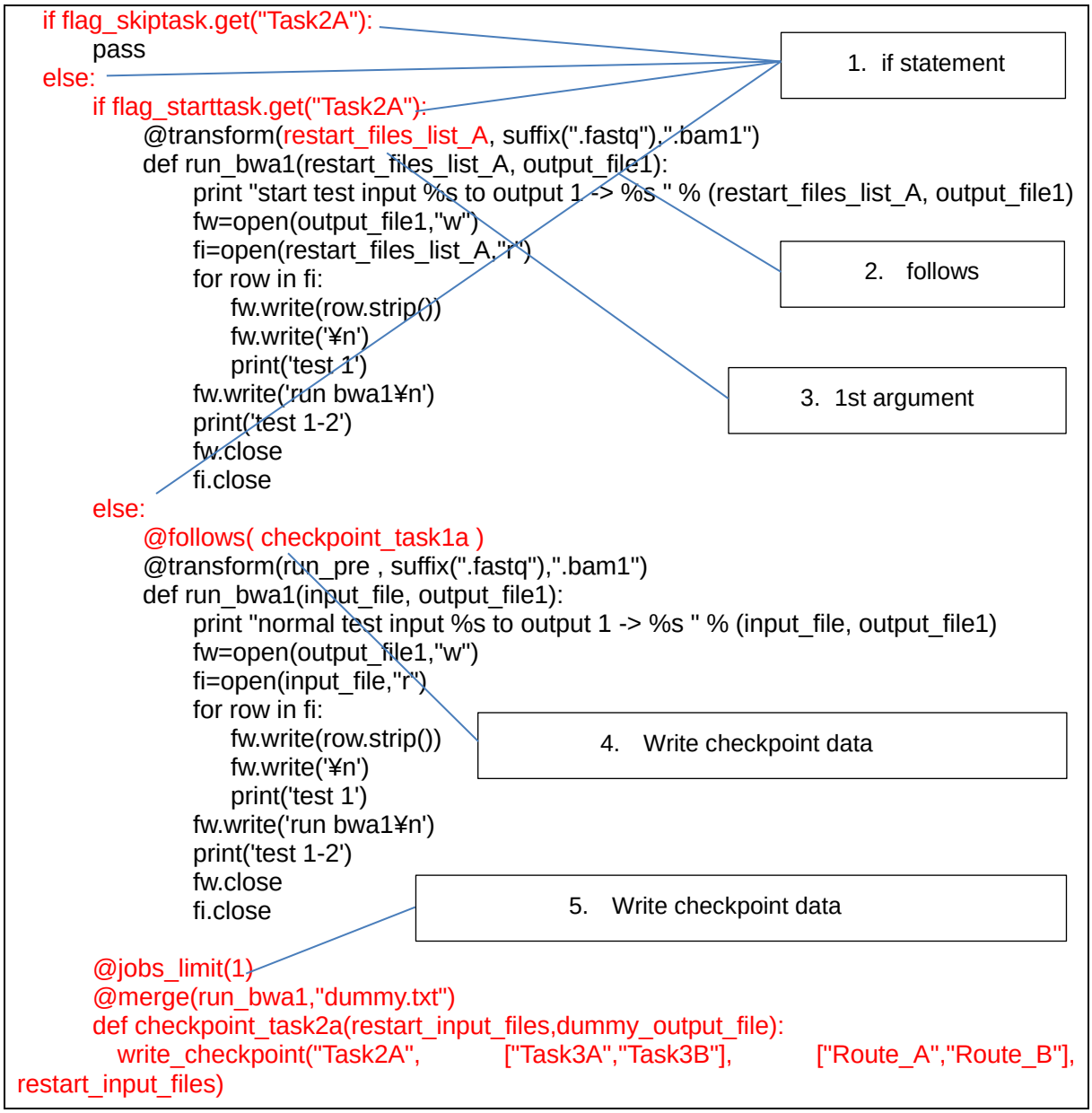

図 5-4 Task2A の構築方法の変更後

続けて、合流タスクである Task4C の記述を書き換えます。図 5-5が変更前、図 5-6が変更後です。合流には、merge デコレータを使用します。通常実行時以外は、merge デコレータの第１引数には、合流タスクの入力となるファイルの絶対パスのリスト restart_files_list_C を指定します。restart_files_list_C には、pipeline_run 関数を呼ぶ前に、Task4C への入力として与えるファイルの絶対パスのリストを予め格納します。また、表 5-1 に示すとおり、合流タスク直前のタスクの状態によって、Task4C に設定するデコレータが変わってきます。そのため、合流タスク直前のタスクの数が増えると、書き換え後の if 文の分岐数も増えます。

表 5-1 Task4C の if 文のパターン

パターン	Task3A の状態	Task3B の状態	Task4C のデコレータ
1	終了	終了	@merge(restart_files_list_C, “merge.bam”)
2	終了	未完了	@follows(checkpoint_task3b) @merge([restart_files_list_C], “merge.bam”)
3	未完了	終了	@follows(checkpoint_task3a) @merge([restart_files_list_C], “merge.bam”)
4	未完了	未完了	@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
```

図 5-5 Task4C の構築方法の変更前

```
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

図 5-6 Task4C の構築方法の変更後

最後に、タスクが終点タスクだった時の write_checkpoint 関数の使用例を説明します。図 5-7の赤丸で囲われたタスクは終点タスクであり、合流タスクの直前のタスクおよび、パイプラインの分岐先の最後のタスクです。図 5-8と図 5-9が、それぞれの場合の write_checkpoint 関数の使用例です。また、ルートが１つしか存在しなかった場合は、最後のタスクが終点タスクです。なお、終点タスクの write_checkpoint 関数の第２引数は[“final”]、第３引数はそのタスクのルート名を要素とするリストとします。

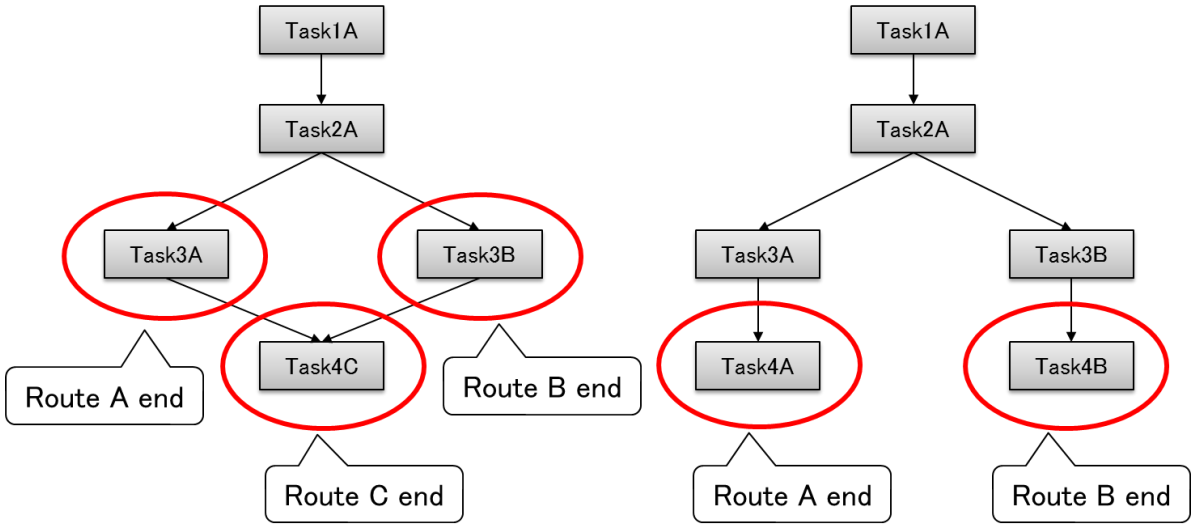

図 5-7 終点タスクの例

```
@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

図 5-8 合流タスクの直前の write_checkpoint 関数の使用例

```
@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)
```

図 5-9 分岐先の終点タスクの write_checkpoint 関数の使用例

5.2. リスタート機能の出力データの説明

パイプラインのリスタートに関するモジュールである checkpointrestart.py について説明します。checkpointrestart.py はリスタートするためのデータの書き出しと、その書き出したデータを読み込みリスタートすることを目的としたモジュールです。

checkpointrestart.py に含まれる write_checkpoint 関数により、アウトプットディレクトリ配下にチェックポイントディレクトリを生成し、チェックポイントデータとチェックポイントマスタファイルの書き出し、バックアップディレクトリの作成を行います。ここで、各ディレクトリ、ファイルの説明は表 5-2のとおりです。また、checkpoint_restart 関数は、前回実行時に生成されたチェックポイントデータから、パイプライン処理をリスタートするための情報を作成します。

表 5-2　用語の説明

用語	説明
アウトプットディレクトリ	パイプラインの出力ファイル(アウトプットファイル)を格納するディレクトリです。パイプライン実行前に生成されます。
アウトプットファイル	パイプラインによる処理、またはその前処理によって出力されるファイルです。なお、リスタート機能により生成されるファイル(チェックポイントデータなど)は除きます。
チェックポイントディレクトリ	チェックポイントディレクトリ checkpoint は、リスタートのために必要なデータを格納するディレクトリです。配下にチェックポイントデータ、チェックポイントマスタファイル、バックアップディレクトリを含みます。
チェックポイントデータ	チェックポイントデータ「“checkpoint_” + タスク名 + “.chk”」は、タスク終了時の出力情報と、その時のアウトプットディレクトリのアウトプットファイル情報を持つファイルです。終了したタスクの数だけ存在します。
チェックポイントマスタファイル	チェックポイントマスタファイル「“checkpoint_” + ルート名 + “.mst”」は、チェックポイントデータの絶対パスと、そのチェックポイントデータを用いてリスタートするタスク名の情報を持つファイルです。ここで、「ルート名」は、「リスタートするタスク」のルートです。ルートの数だけ存在します。
バックアップディレクトリ	バックアップディレクトリ「“backupdata_” + ルート名」は、アウトプットファイルのバックアップを保存します。ここで、「ルート名」は、そのバックアップを用いてリスタートするタスクのルートです。ルートの数だけ存在します。

アウトプットディレクトリ配下の構成は下記のようです。

アウトプットディレクトリ/
  ├ アウトプットファイル
  ├ checkpoint/ (チェックポイントディレクトリ)
    ├ checkpoint_〇〇.chk (チェックポイントデータ)
    ├ checkpoint_××.mst (チェックポイントマスタファイル)
    ├ backup_××/ (バックアップディレクトリ)
      ├ ・・・
    ・・・

図 5-１０はチェックポイントデータ（checkpoint_Task3A.chk）の出力例です。


```
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/example/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/bam2fastq_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_20190109_1126_194861.sh'¥np16¥naS'fastq/sample_tumor/unmatched_first_output.txt'¥np17¥naS'fastq/sample_tumor/unmatched_second_output.txt'¥np18¥naS'fastq/sample_tumor/single_end_output.txt'¥np19¥naS'fastq/sample_tumor/1_0000.fastq_split'¥np20¥naS'fastq/sample_tumor/2_0000.fastq_split'¥np21¥naS'fastq/sample_tumor/fastq_line_num.txt'¥np22¥naS'fastq/sample_normal/unmatched_first_output.txt'¥np23¥naS'fastq/sample_normal/unmatched_second_output.txt'¥np24¥naS'fastq/sample_normal/single_end_output.txt'¥np25¥naS'fastq/sample_normal/1_0000.fastq_split'¥np26¥naS'fastq/sample_normal/2_0000.fastq_split'¥np27¥naS'fastq/sample_normal/fastq_line_num.txt'¥np28¥naS'bam/reference5/reference5.markdup.bam'¥np29¥naS'bam/reference5/reference5.markdup.bam.bai'¥np30¥naS'bam/reference10/reference10.markdup.bam'¥np31¥naS'bam/reference10/reference10.markdup.bam.bai'¥np32¥naS'bam/reference7/reference7.markdup.bam'¥np33¥naS'bam/reference7/reference7.markdup.bam.bai'¥np34¥naS'bam/reference6/reference6.markdup.bam'¥np35¥naS'bam/reference6/reference6.markdup.bam.bai'¥np36¥naS'bam/reference4/reference4.markdup.bam'¥np37¥naS'bam/reference4/reference4.markdup.bam.bai'¥np38¥naS'bam/reference3/reference3.markdup.bam'¥np39¥naS'bam/reference3/reference3.markdup.bam.bai'¥np40¥naS'bam/reference2/reference2.markdup.bam'¥np41¥naS'bam/reference2/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.bam'¥np45¥naS'bam/reference9/reference9.markdup.bam.bai'¥np46¥naS'bam/reference8/reference8.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'size_list'¥np51¥n(lp52¥nI1362¥naI1369¥naI2012¥naI2006¥naI0¥naI0¥naI0¥naI21236823¥naI21236823¥naI6¥naI0¥naI0¥naI0¥naI21251939¥naI21251939¥naI6¥naI4677658¥naI8280¥naI4615163¥naI53192¥naI4617939¥naI8280¥naI4639312¥naI8280¥naI4694499¥naI8280¥naI4679155¥naI53192¥naI4563105¥naI53192¥naI4644582¥naI8280¥naI4646589¥naI53192¥naI4679086¥naI53192¥naI902¥naI1173¥nass."
p0
```


図 5-１０ チェックポイントデータ（checkpoint_Task3A.chk）の出力例

図 5-１１はチェックポイントマスタファイル（checkpoint_Route_A.mst）の出力例です。

```
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
```

図 5-１１ チェックポイントマスタファイル（checkpoint_Route_A.mst）の出力例

6. コマンドの詳細

本章ではコマンドの詳細について説明します。

6.1. vge

vge コマンドは VGE 本体を起動するプログラムです。表 6-1 は vge コマンドの実行時に指定可能なオプションです。

表 6-1　vge コマンドの実行時に指定可能なオプション

オプション名	パラメタ	内容
--loadbalancer	なし	--loadbalancer だけを指定した場合は、--loadbalancer time と同等の動作をします。
	time	ロードバランサー機能をオンにし、ジョブ処理は MPI ワーカーの作業時間でバランスをとります。
	count	ロードバランサー機能をオンにし、ジョブ処理は MPI ワーカーのジョブ回数でバランスをとります。
	off	ロードバランサー機能を明示的にオフにする（--loadbalancer を付けなければロードバランサー機能はオフです）。
--schedule	なし	--schedule だけを指定した場合は、--schedule first と同等の動作をします。
	first	ジョブ待ちリストで並んでいるジョブを、ジョブが VGE へ来た順（アレイジョブはすべて展開されたジョブ）に逐次実行します。
	sample	検体番号順でジョブ待ちリストのジョブが処理されます。
	arrayjob	アレイジョブ単位での順番で逐次ジョブが処理されます。
	mix	検体番号とアレイジョブ番号順の混合型です。
--nowrite_jobscript	なし	この引数が指定されると、VGE 処理するジョブスクリプトとその標準出力結果・エラー出力結果をファイルに出力しません。
--nowrite_vgecsv	なし	この引数が指定されると、CSV ファイル形式の VGE ジョブ処理情報リストを出力しません。
-o	ディレクトリ名	ジョブスクリプトおよびジョブ情報リストを出力するディレクトリを指定します。デフォルトは、VGE の実行ディレクトリ下の vge_output です。
--check_mpi_probe	なし	MPI ワーカーへの MPI IPROBE テストを実行します。
--monitor_workerjob	なし	VGE ワーカーは実行中のジョブを監視するモードを実行します。これは強制終了機能用のオプションです。

6.2. vge_connect

vge_connect コマンドは VGE の待機・停止を制御するプログラムです。表 6-2 は vge_connect コマンドの実行時に指定可能なオプションです。

表 6-2　vge_connect コマンドの実行時に指定可能なオプション

オプション名	パラメタ	内容
--start	なし	VGE の起動を待つ命令（メッセージ）を VGE へ送信します。
--wait	なし	バックグラウンドで実行した Pipeline が終了するまでプロセスを監視・待機するモード（待機モード）を指定します。
--sleep	0 秒	待機モードのみ有効なオプションです。ターゲットプロセスを監視する前の待機時間（単位：秒）を指定します。デフォルト設定は 0 秒で、待ち時間がなしの設定です。
--wait_maxtime	0 秒	待機モードの最大待機時間（単位：秒）です。デフォルト設定は 0 秒で、無制限の設定です。vge_connect と同じノードで実行されているターゲットプロセスが終了するまで待機します。
--monitor	'pipeline'	プロセスが終了するのを監視するターゲットプロセス名を指定します。
--stop	なし	VGE を停止させる命令（メッセージ）を VGE へ送信します。
--force	なし	VGE の強制終了を指定します。
--target	'pipeline'	VGE 停止オプション使用時のみ有効なオプションです。VGE 停止命令を VGE へ送信する前に、このオプションで指定したプロセス名のプロセスを削除する処理を実行します。
--stop_maxtime	600 秒	VGE 停止モードでのみ有効なオプションです。VGE 停止信号を VGE へ送信後に、VGE マスターノードの VGE プロセスが終了するまで待機する最大時間です。

6.3. **cleanvge**

cleanvge コマンドは VGE 関連のプロセスを削除するプログラムです。表 6-3は cleanvge コマンドの実行時に指定可能なオプションです。

表 6-3　cleanvge コマンドの実行時に指定可能なオプション

オプション名	パラメタ	内容
--verbose	1	cleanvge の情報出力レベル（0:致命的エラー以外なし、1:情報レベル、3:デバックレベル）

7. ユーザー側で設定する関数の詳細

本章ではユーザー側で設定する関数の詳細について説明します。

7.1. vge_init

vge_init 関数は VGE 設定ファイルの読込および VGE へハローメッセージの送信を行う関数です。ハローメッセージは VGE にパイプラインジョブのプロセスが開始したことを通知するためのメッセージです。以下に vge_init 関数のインタフェースを示します。

表 7-1　vge_init 関数のインタフェース

I / O	引数名、戻り値名	型	値
引数	なし	-	-
戻り値	-	int	0：通常終了、0 以外：異常終了

7.2. vge_finalize

vge_finalize 関数は VGE へグッバイメッセージの送信を行う関数です。グッバイメッセージは VGE にパイプラインジョブのプロセスが終了することを通知するためのメッセージです。以下に vge_finalize 関数のインタフェースを示します。

表 7-2　vge_finalize 関数のインタフェース

I / O	引数名、戻り値名	型	値
引数	なし	-	-
戻り値	-	int	0：通常終了、0 以外：異常終了

7.3. vge_task

vge_task 関数は VGE とプロセス間通信を行い、VGE で実行させたい処理をジョブ情報として VGE に送る関数です。以下に vge_task 関数のインタフェースを示します。

表 7-3　vge_task 関数のインタフェース

I / O	引数名、戻り値名	型	値
引数	arg1	文字列	実行するコマンド
	arg2	int	最大タスク数
	arg3	文字列	基本ファイル名
	arg4	辞書	vge.conf の [pipeline]のパラメタを変更するための辞書
戻り値	-	int	0：通常終了、0 以外：異常終了

7.4. write_checkpoint

write_checkpoint 関数はリスタート機能で使用するチェックポイントデータ、チェックポイントマスタファイルの書き出し、バックアップの作成を行う関数です。checkpoint_tag と output_files は、チェックポイントデータ作成に使用します。checkpoint_tag_next_list は、チェックポイントマスタファイル作成に使用します。checkpoint_route_tag_next_list は、バックアップディレクトリ作成と、チェックポイントマスタファイル作成に使用します。以下に write_checkpoint 関数のインタフェースを示します。

表 7-4　write_checkpoint 関数のインタフェース

I / O	引数名、戻り値名	型	値
引数	checkpoint_tag	文字列	タスク名
	checkpoint_tag_next_list	リスト	次のタスク名のリスト
	checkpoint_route_tag_next_list	リスト	次のタスクのルート名のリスト
	output_files	リスト	各アウトプットファイルの絶対パスのリスト
戻り値	-	int	0：通常終了、1：異常終了

7.5. checkpoint_restart

checkpoint_restart 関数はリスタート機能のチェックポイントの書き出し時に用いるグローバル変数の設定、タスクをスキップする／しないを示すフラグ、リスタート時の開始タスクを示すフラグの作成、リスタートするタスクへの入力用の変数の作成を行う関数です。target_dir は、前準備のグローバル変数作成と、バックアップディレクトリの再配置で使用します。flag_skiptask は、前回実行時にすでに終了しているタスクをリスタート時にスキップするフラグとして使用します。flag_starttask は、リスタート地点のタスクを示すフラグとして使用します。restart_files_list_dict は、ルートごとの各タスクのリスタートに必要な入力ファイルの情報として使用します。以下に checkpoint_restart 関数のインタフェースを示します。

表 7-5　checkpoint_restart 関数のインタフェース

I / O	引数名、戻り値名	型	値
引数	target_dir	文字列	アウトプットディレクトリの絶対パス
戻り値	flag_skiptask	辞書	キー：タスク名 要素：bool 値
	flag_starttask	辞書	キー：タスク名 要素：bool 値
	restart_files_list_dict	辞書	キー：ルート名 要素：対応するルートのタスクのリスタート用入力として与えるファイルの絶対パスのリスト

8. Q&A

Q1. VGE のジョブ実行中に、VGE 上で動作するアプリケーションを追加できますか？

A1. VGE を起動した計算ノード上で vge_task を呼び出すアプリケーションを実行できる場合、追加できます。ジョブシステムなどの都合により同一の計算ノードを使用できない場合、追加できません。

Q2. １つの VGE により複数のアプリケーションを実行した場合、特定のアプリケーションを停止できますか？

A2. 特定のアプリケーションのみを停止させることはできません。そのような場合、VGE の停止によりすべてのアプリケーションを停止してください。

Q3. リスタート用関数の呼び出しが設定されているアプリケーションの実行時に、リスタート機能を抑止(ファイル出力をしないように)することは可能でしょうか？

A3. リスタート機能を抑止できません。リスタート用関数を呼び出さないように修正してから実行してください。

Q4. ruffus を用いていないアプリケーションでもリスタート機能は使用できますか？

A4. 使用できます。

Q5. 複数のユーザーが同じ計算ノードにて VGE を起動した場合の動作は？

A5. 別々のユーザーが同一のポート番号を使って VGE を起動した場合、意図しないユーザープロセスと接続、もしくはエラーとなる可能性があります。複数のユーザーが利用するような環境の場合、ポート番号を一意になるよう設定し実行してください。

Q6. mpiexec をバックグラウンドで実行するのは何故ですか？

A6. VGE が起動した状態で対象アプリケーションを起動させるためです。

Q7. VGE の設定ファイルの配置場所が２通りあるのは何故ですか？

A7. ユーザーの利便性向上のためです。
例えば、複数のユーザーが同じ設定を利用する場合に環境変数により特定のファイルを利用するようにし、個人で設定したもの(特に port 番号)を使う場合は実行時のディレクトリを使うなどの使い分けが考えられます。

Q8. verbose の設定について 2 がない場合があるのは何故ですか？

A8. verbose の設定は 1.情報レベル、3.デバッグレベルであることは共通です。2. は vge のみ設定があります。これは VGE が複数プロセスを worker として動作するため、マスターのみの出力とマスターとワーカー両方の出力を区別するためにあります。

Ｑ９．リスタート機能使用時にチェックポイントのディレクトリは毎回作成されるのでしょうか？

Ａ９．ディレクトリは初回のチェックポイント時のみ作成されます。同一ジョブ内ではチェックポイント実施毎に同じディレクトリが更新されます。また、リスタート後のチェックポイントはリスタート時のチェックポイントディレクトリを更新するためリスタート時に新規に作成することはありません。

Ｑ１０．リスタート時に対象のチェックポイントディレクトリが存在しない場合の動作はどうなりますか？

Ａ１０．checkpoint_restart 関数内でリスタートするデータがないと判断され通常の実行となります。

９．使用上の注意

本章では使用上の注意について説明します。

９.１．**MPI** 関連の制限事項

MPI が使用するインターコネクトのドライバに OpenFablics が使用されている環境において、fork がサポートされていないことがあります。VGE では Multiprocessing Manager モジュールを用いて fork により並列実行している処理があります。このため、前述した fork がサポートされていない環境の場合、正常に動作しない可能性があります。

９.２．リスタート機能の制限事項

タスクによる処理の特性や、システムの不具合、ユーザーによる誤操作によって、リスタート機能が正しく機能しない可能性があります。そのため、以下の点に注意する必要があります。

- パイプラインの処理が中断された際、タスクが途中まで進むことによって、中途半端な状態のファイルが存在する可能性が考えられます。リスタートは中断したタスクの最初から行われるため、このファイルが結果に影響を与える可能性があります。その場合、該当のファイルはユーザーが削除する必要があります。
- チェックポイントマスタファイルの状態が変化した場合、その内容と、アウトプットディレクトリの状態、実際のチェックポイントの作成状況などに齟齬が生じる可能性があります。そのため、チェックポイントマスタファイルを削除／編集した場合や、チェックポイントマスタファイルが破損していた場合には、想定外の動作をする可能性があります。

９.３．**VGE** プロセスの動作ノードについて

パイプラインを処理するプロセスと VGE のマスタープロセスは同じ計算ノードで実行されます。このため、パイプライン側で CPU 負荷が大きい処理を実行していた場合、VGE 側の処理性能に影響が出る可能性があります。CPU 負荷の大きい処理はパイプラインジョブとして実行することを推奨します。

９.４．**cleanvge** コマンドの使用について

VGE の異常終了時にプロセスが残ることがあります。プロセスが残っていた場合、cleanvge コマンドを使用してプロセスを削除することができます。ただし、MPI 実行処理でアサインされるノードは、異なったバッチジョブスクリプトで同ノードを共有する可能性があります。そのため、cleanvge コマンドを動かすときに、別のパイプライン処理が実行されている場合に、そのパイプライン処理が動いているノードに cleanvge コマンドを動かしたときの計算リソースが確保されてしまうと、消したくないパイプライン処理のプロセスを消してしまいます。そのため、cleanvge コマンドは、ユーザーが VGE 関連の処理を実行していない状況で使用してください。

図 9-1 は cleanvge を使用した実行スクリプトです。実行スクリプトでは mpiexec コマンドを用いて MPI プロセスで cleanvge をフォアグラウンドで起動します。

```
#!/bin/sh -x

····omitted

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

····omitted
```

図 9-1 cleanvge を使用した実行スクリプト