FUJITSU Software
Technical Computing Suite V4.0L20

Job Operation Software End-user's Guide for Master-Worker Job

J2UL-2536-01ENZ0(02)
March 2022

Preface

Purpose of This Manual

This manual describes master-worker jobs, which are one of the job models of the job operation software included in Technical Computing Suite.

Intended Readers

This manual is intended for the end users who create and execute master-worker jobs.

The manual assumes that readers have the following knowledge:

- Basic knowledge of Linux
- Knowledge of using the job operation software to submit and execute jobs (Refer to "Job Operation Software End-user's Guide")

Organization of This Manual

This manual is organized as follows.

Notation Used in This Manual

Notation of model names

In this manual, the computer that based on Fujitsu A64FX CPU is abbreviated as "FX server", and FUJITSU server PRIMERGY as "PRIMERGY server" (or simply "PRIMERGY").
Also, specifications of some of the functions described in the manual are different depending on the target model. In the description of such a function, the target model is represented by its abbreviation as follows:
[FX]: The description applies to FX servers.
[PG]: The description applies to PRIMERGY servers.

Symbols in this manual

This manual uses the following symbols.

Note

The Note symbol indicates an item requiring special care. Be sure to read these items.

See

The See symbol indicates the written reference source of detailed information.

Information

The Information symbol indicates a reference note related to the job operation software.

Date of Publication and Version

Version	Manual code
March 2022, Version 1.2	J2UL-2536-01ENZ0(02)
June 2020, Version 1.1	J2UL-2536-01ENZ0(01)
February 2020, First Version	J2UL-2536-01ENZ0(00)

Update history

Changes	Location	Version
Added that a master-worker job is submitted with the --mswk option of the pjsub command.	Chapter 2	1.2
Added that the maximum number of concurrent executions of the pjaexe command in a job is 128.	2.3.3	
Added a description and reference to the storage I/O node.	3.1	
Added a sample program example that shows how to calculate rank numbers.	A.4 Figure A.3	
Corrected errors and confusing contents of the sample program.	2.3.3 A.3 A.4	1.1
Added the effect on a master-worker job when the storage I/O node or job slave node goes down.	3.1	

Contents

Chapter 1 Master-Worker Job

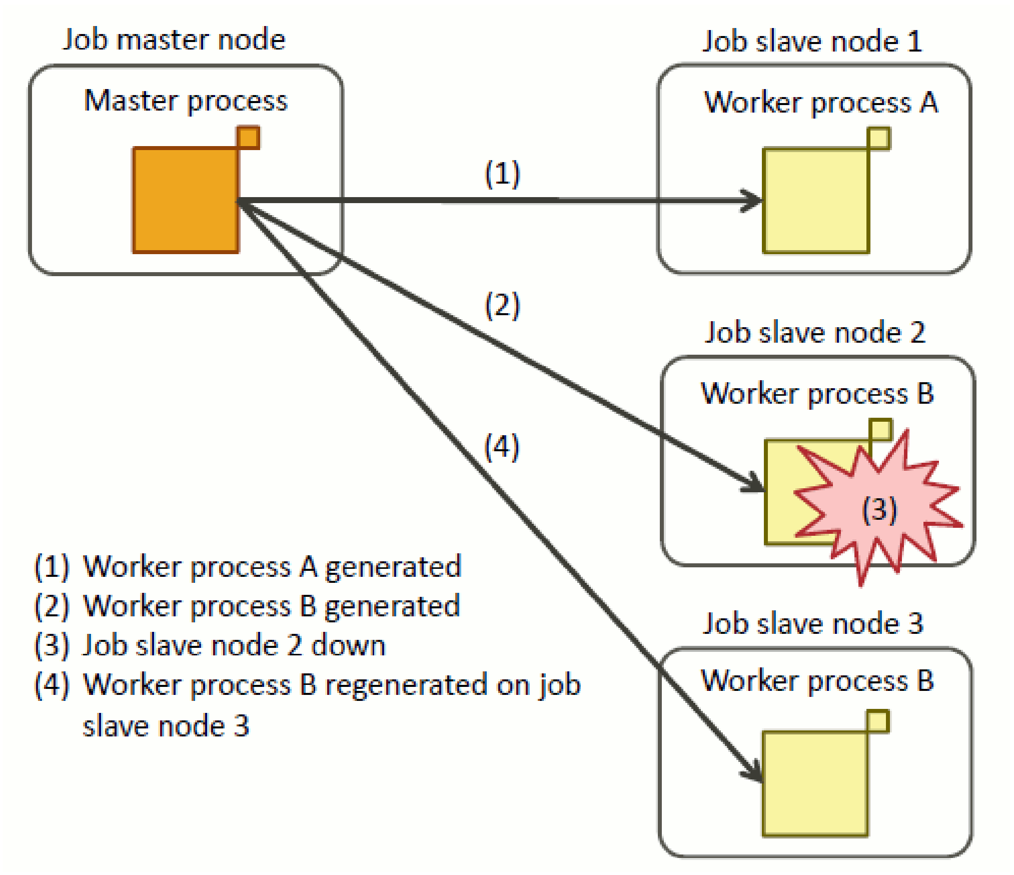
Master-worker jobs are one of the job models, like normal jobs, step jobs, and bulk jobs, and they have the following characteristics.

- A master-worker job consists of a job script process, master process, and worker process.
  The master and worker processes perform computing tasks in collaboration with each other. (Task: Unit of parallel program processing)
- The master process controls entire computing tasks, generates and manages worker processes, and integrates computed results.
  Worker processes perform computing tasks upon request from the master process and then return the results to the master process.
- The master-worker job continues as long as the job script process is running, even if the compute node allocated to the master-worker job fails or the process ends abnormally.

The user can use this characteristic to continue a computing task by creating a mechanism to re-execute a worker process on a different node when a compute node is down or the worker process has ended abnormally.

The following diagram shows a conceptual image of a master-worker job executing a user program.

Figure 1.1 Conceptual image of execution of a master-worker job


Job master node
Master process
Job slave node 1
Worker process A
(1)
(2)
Job slave node 2
Worker process B
(3)
(4)
Job slave node 3
Worker process B
(1) Worker process A generated
(2) Worker process B generated
(3) Job slave node 2 down
(4) Worker process B regenerated on job slave node 3

Information

- The job script process runs on a node called a "job master node," and the other nodes are called "job slave nodes." The master process must run on the job master node because the purpose of the master process is to manage the worker processes running on job slave nodes.
- In terms of generating multiple processes, "bulk jobs" are a similar job model. In that method, a bulk job submits the same job script as multiple sub jobs, and each of these sub jobs runs independently. The bulk job, when submitted, specifies the number of sub jobs. The number does not change during execution.
  Meanwhile, in a master-worker job, the master process runs together with each worker process as one job while the processes communicate with one another. The number of worker processes changes during execution of the job since they are dynamically

generated by the master process. For details on job models other than the master-worker jobs, see "Job Operation Software End-user's Guide."

The job operation software supports the following three methods as methods for implementing master-worker jobs, from the viewpoint of how to generate worker processes.

- Generating a worker process with MPI dynamic process generation

  This method is used when both the master process and worker process are MPI programs. That is, MPI mechanisms will be used for worker process generation and communication.
  The job operation software selects the node that will generate a worker process.

- Generating a worker process from an Agent process

  This method is used when the worker process is not an MPI program.

  Communication between processes uses a communication function such as a socket. The user controls and manages the generation of the worker process and the selection of the node generating the worker process.

- Generating a worker process with the pjaexe command

  Similar to the preceding paragraph, this method is also used when the worker process is not an MPI program.
  Likewise in this method, communication between processes uses a communication function such as a socket, and the user controls and manages the selection of the node that will generate the worker process. However, the pjaexe command provided by the job operation software is used to generate the worker process.

To create a master-worker job, the user needs to select a method matching the purpose of the job.

For details on these methods, see "Chapter 2 Creating a Master-Worker Job."

Chapter 2 Creating a Master-Worker Job

This chapter describes three methods of implementing master-worker jobs and provides notes on creating them.

See

A master-worker job is submitted with the --mswk option of the pjsub command.
For details on how to submit a master-worker job, see "Chapter 2 Job Operation Procedures" in "Job Operation Software End-user's Guide", and the man manual of the pjsub command.

2.1 Dynamically Generating a Worker Process

In the method of dynamically generating a worker process from the master process for an MPI program, the user needs to implement the following functions:

a. Master program (master process)

    1. Generate a worker process
    2. Request a worker process to perform a computation
    3. Perform an alive check of a worker process

b. Worker program (worker process)

    1. Receive a computation request from the master process, and send back the computational results
    2. Send notification of computation end to the master process

The following sections describe processing details.

See

For details on the program configurations and coding examples shown in the descriptions, see "A.1 Example of Dynamically Generating a Worker Process."

2.1.1 Node allocation

The method of dynamically generating a worker process needs to have the --mpi "shape=1" option of the pjsub command specified such that only the job master node starts the master process generated when the MPI program starts.

Of the 96 nodes allocated to a job in the following example, 1 node is allocated to the master process generated when the MPI program starts. The remaining 95 nodes are nodes for dynamically generating worker processes.

```
[Job script example: job_dynamic.sh]

#!/bin/bash
#PJM -L "node=96"
#PJM --mpi "shape=1"
...
```

2.1.2 MPI functions and MPI subroutines for master-worker jobs

Some of the MPI functions or MPI subroutines in the MPI programs to be executed in master-worker jobs must be replaced with the functions for the master-worker jobs. This is because the job operation software needs to internally execute processes specific to the master-worker jobs.

The following table lists the names of these MPI functions and MPI subroutines.

Table 2.1 MPI functions for master-worker jobs (for C language)

MPI function name for normal job	MPI function name for master-worker job
MPI_Comm_connect()	FJMPI_Mswk_connect()
MPI_Comm_disconnect()	FJMPI_Mswk_disconnect()
MPI_Comm_accept()	FJMPI_Mswk_accept()

Note

The above-described MPI functions for master-worker jobs are declared in the header file mpi-ext.h.

Table 2.2 MPI subroutines for master-worker jobs (for Fortran language)

MPI subroutine name for normal job	MPI subroutine name for master-worker job
MPI_COMM_CONNECT()	FJMPI_MSWK_CONNECT()
MPI_COMM_DISCONNECT()	FJMPI_MSWK_DISCONNECT()
MPI_COMM_ACCEPT()	FJMPI_MSWK_ACCEPT()

Note

The above-described MPI subroutines for master-worker jobs are declared in the mpi_f08_ext and mpi_ext modules. Since the mpi_f08_ext and mpi_ext modules correspond to the MPI modules mpi_f08 and mpi, respectively, either can be the module enclosed in quotation marks in a USE statement.

The specifications of the MPI functions and MPI subroutines for master-worker jobs are the same as for the MPI functions for normal jobs. For the specifications, see the MPI specifications and "MPI User's Guide," which is a Development Studio manual.

The following sections provide examples of using these MPI functions for master-worker jobs.

2.1.3 Generating a master process

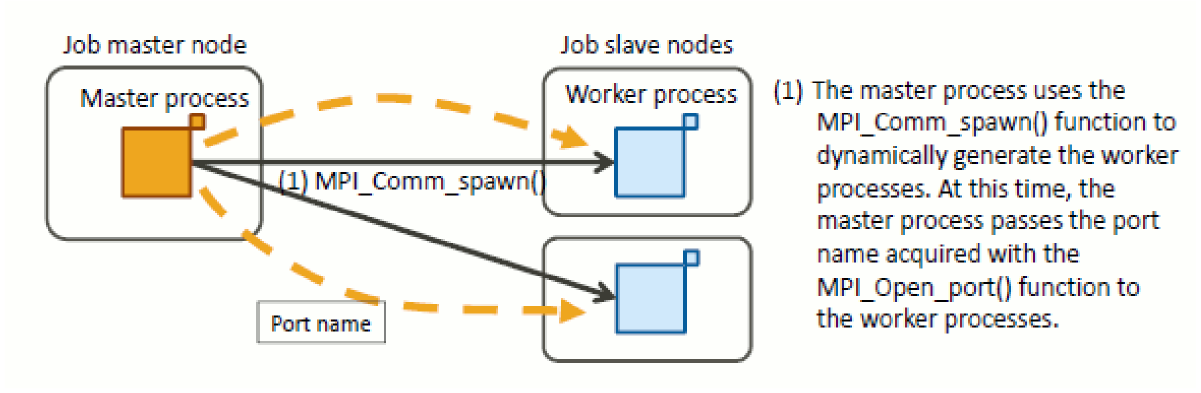
A master process is the process initially generated by the MPI program executed with the mpiexec command.

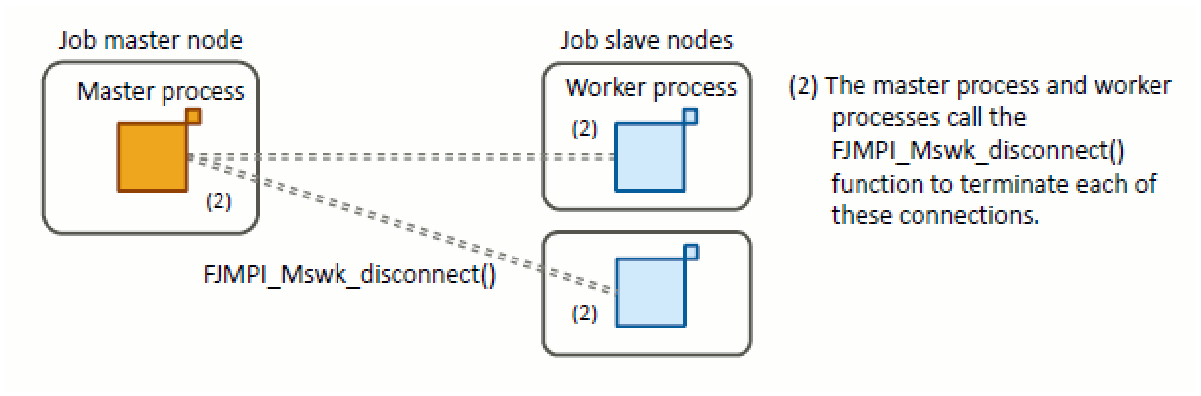
```
[Job script example: job_dynamic.sh]
...
mpiexec ./master_spawn.out          <- Execute the MPI program master_spawn.out.
...
```

2.1.4 Generating a worker process

The master process generated at the MPI program start time uses the MPI_Comm_spawn() (or MPI_Comm_spawn_multiple()) function to dynamically generate a worker process.

Figure 2.1 Generating worker processes with the MPI_Comm_spawn() function


Job master node
Job slave nodes
Master process
Worker process
(1) MPI_Comm_spawn()
Port name
(1) The master process uses the MPI_Comm_spawn() function to dynamically generate the worker processes. At this time, the master process passes the port name acquired with the MPI_Open_port() function to the worker processes.


Job master node
Job slave nodes
Master process
Worker process
(2)
(2)
(2)
FJMPI_Mswk_disconnect()
(2) The master process and worker processes call the FJMPI_Mswk_disconnect() function to terminate each of these connections.

```
[Master process program example: master_spawn.c]

...
// Open a communication port for a worker process.
MPI_Open_port(MPI_INFO_NULL, master_port);
...

// Generate a worker process.
MPI_Comm_spawn("./worker_spawn.out", ..., &worker_comm, ...);

// Send a port name to a worker process.
MPI_Send(master_port, ..., worker_comm);

// Disconnect the connection with a worker process.
FJMPI_Mswk_disconnect(&worker_comm);
...
```

Information

The above example generates a worker process and subsequently calls the FJMPI_Mswk_disconnect() function to disconnect the connection with the worker process. This processing is necessary based on the MPI specifications. For details, see "2.4 Notes on Creating a Master-Worker Job."

2.1.5 Communication of a master process and worker processes

The master process and a worker process communicate as follows.

1. Communication start

   To establish a connection, the master process calls the FJMPI_Mswk_connect() function, and the worker process calls the FJMPI_Mswk_accept() function. The master process can also call the FJMPI_Mswk_accept() function, and the worker process can also call the FJMPI_Mswk_connect() function. In this case, the worker process have to call the MPI_Open_port() function and send the port name to the master process.

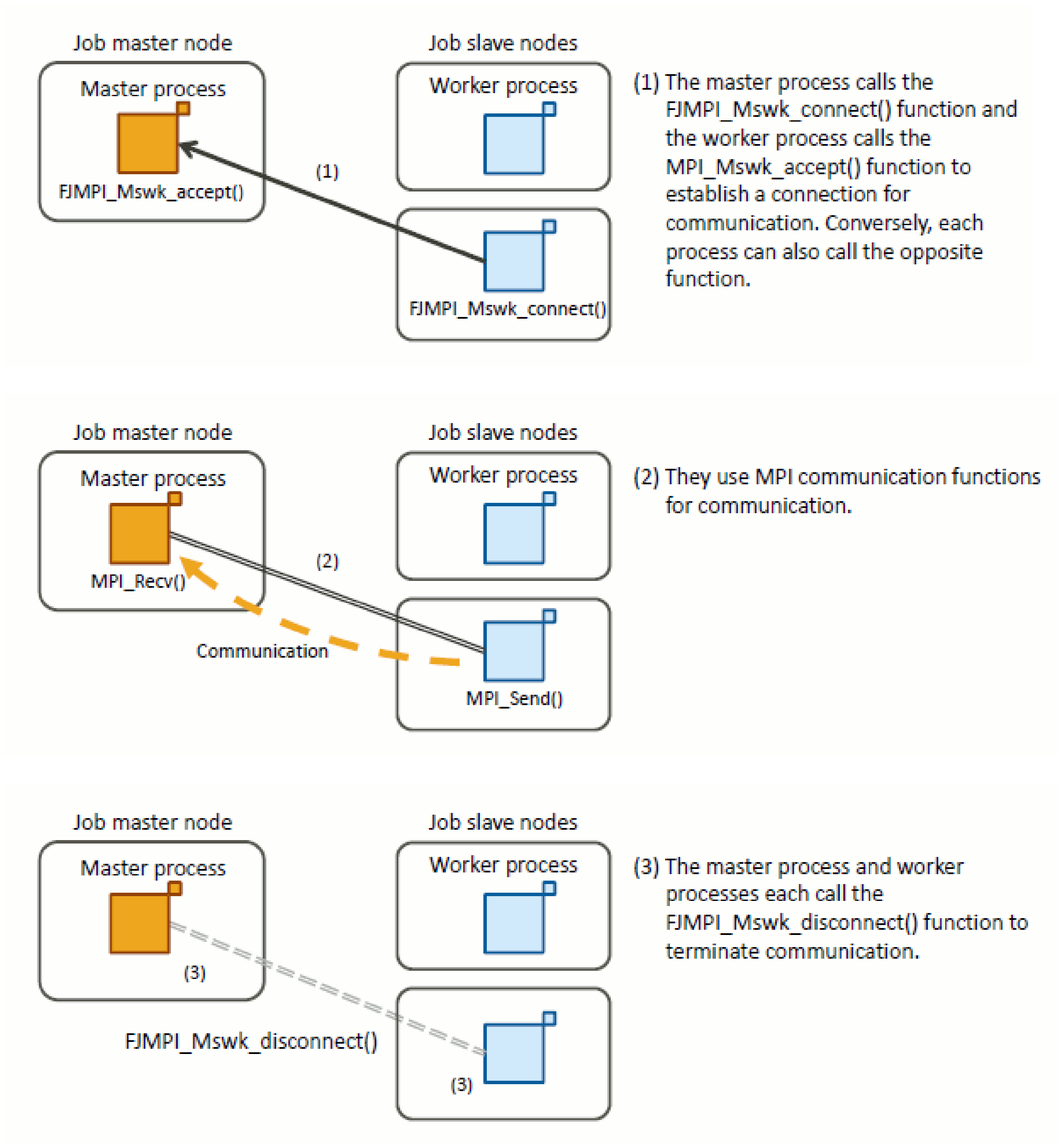
2. Communication processing

   They communicate using the MPI communication functions MPI_Send() and MPI_Recv().

3. Communication end

   The master process and worker processes each call the FJMPI_Mswk_disconnect() function to terminate the connection.

Figure 2.2 Communication between a master process and worker processes using MPI communication functions


Job master node
Job slave nodes
Master process
Worker process
FJMPI_Mswk_accept()
(1)
FJMPI_Mswk_connect()
(1) The master process calls the FJMPI_Mswk_connect() function and the worker process calls the MPI_Mswk_accept() function to establish a connection for communication. Conversely, each process can also call the opposite function.
Job master node
Job slave nodes
Master process
Worker process
MPI_Recv()
(2)
Communication
MPI_Send()
(2) They use MPI communication functions for communication.
Job master node
Job slave nodes
Master process
Worker process
(3)
FJMPI_Mswk_disconnect()
(3)
(3) The master process and worker processes each call the FJMPI_Mswk_disconnect() function to terminate communication.

Program examples for the master process and worker processes are shown below.

```
[Master process program example: master_spawn.c]
...
// Accept a connection from the worker process.
FJMPI_Mswk_accept(master_port, ..., &worker_comm);
```

```
// Receive data from the worker process.
MPI_Recv(..., worker_comm, ...);
// Terminate communication with a worker process.
FJMPI_Mswk_disconnect(&worker_comm);
...
```

```
[Worker process program example: worker_spawn.c]
...
// Connect to the master process.
FJMPI_Mswk_connect(master_port, ..., &master_comm);
// Send data to the master process.
MPI_Send(..., master_comm);
// Terminate communication with the master process.
FJMPI_Mswk_disconnect(&master_comm);
...
```

2.1.6 Job end

After being executed from a job script, the mpiexec command returns when the initially generated master process ends. There is no waiting for the end of a worker process at this time.

Then, the master-worker job ends when the job script ends. The job operation software terminates the remaining worker processes when the job ends. Therefore, the user does not need to explicitly terminate them.

2.2 Generating a Worker Process from an Agent Process

Use the method of generating a worker process from an Agent process (referred to below as the Agent process method) in cases where the worker process is not an MPI program.

In this method, the user needs to implement the following functions for the job:

a. Job script

    1. Execute the master program that will become the master process

    2. Generate an Agent process

    3. Wait for the master process to end

b. Master program (master process)

    1. Perform an alive check of an Agent process

    2. Request an Agent process to generate a worker process

c. Agent program (Agent process)

    1. Establish communication with the master process

    2. Generate a worker process

    3. Send the processing results of a worker process to the master process

The following sections describe processing details for the Agent process method.

See

For details on the program configurations and coding examples shown in the descriptions, see "A.2 Shell Script utility.sh" and "A.3 Agent Process Method Examples."

2.2.1 Node allocation

The Agent process method generates only one Agent process on each node. For this reason, allocate the same number of nodes to the job as the number of processes generated by the mpiexec command.

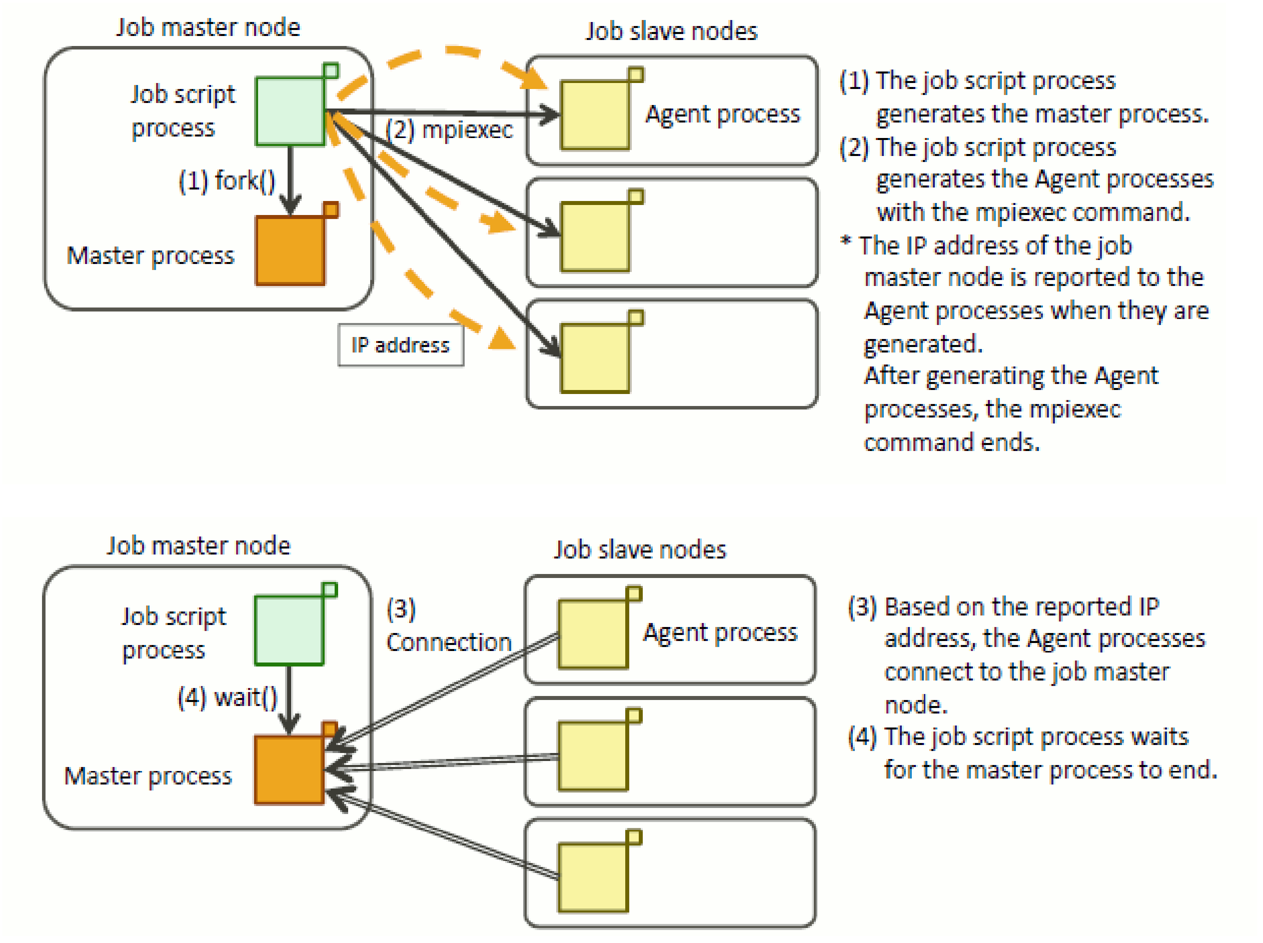
The following example specifies the number of nodes and the number of processes to be generated within the directive line (begin with "#PJM") in the job script. This example allocates 96 nodes to the job (-L "node=96") and generates 96 Agent processes (--mpi "proc=96").

```
[Job script example: job_agent.sh]
#!/bin/bash
#PJM -L "node=96"
#PJM --mpi "proc=96"
...
mpiexec start_agent.sh args ...
...
```

2.2.2 Generating a master process and Agent processes

The flow of master process and Agent process generation in the Agent process method is as follows.

Figure 2.3 Generating a master process and Agent processes


Job master node
Job slave nodes
Job script process
(1) fork()
Master process
(2) mpiexec
Agent process
IP address
(1) The job script process generates the master process.
(2) The job script process generates the Agent processes with the mpiexec command.
* The IP address of the job master node is reported to the Agent processes when they are generated. After generating the Agent processes, the mpiexec command ends.
Job master node
Job slave nodes
Job script process
(4) wait()
Master process
(3) Connection
Agent process
(3) Based on the reported IP address, the Agent processes connect to the job master node.
(4) The job script process waits for the master process to end.

The following describes an implementation example.

1. Generating a master process [job script]

   The job script process generates a master process as a child process of the job script.

   At this time, save the process ID of the master process in memory for the wait for the end of the master process. Also, retrieve the necessary information (port number) for the Agent process to connect to the master process. In the following example, the program master_agent.out outputs a port name to the file port_num.txt.

```
[Job script example: job_agent.sh]
...
. ./utility.sh

./master_agent.out port_num.txt &   # Generate a master process.
MASTER_PID=$!                           # Set the shell variable MASTER_PID as the process ID of
                                            # the master process.
PORT_NUM=$(cat port_num.txt)          # Set the shell variable PORT_NUM as the port number for
                                            # connecting the aster process.
...
```

The shell script utility.sh is read within the above job script.

It defines the shell functions print_ipaddr and timeout, which are used in the next section. For details on utility.sh, see "A.2 Shell Script utility.sh."

The master program master_agent.out waits for a connection request from the Agent process. After a connection is established to the Agent, the master program requests the Agent to execute a worker process, as shown in the following example.

```
[Master process program example: master_agent.c]

int main(int argc, char **argv)
{
    char *port_file = argv[1];

    int sockfd = socket(...);         // Create a socket.
    bind(sockfd, ...);                 // Bind the socket to a specific port.
    listen(sockfd, ...);               // Wait for a connection from a worker process.

    FILE *fp = fopen(port_file, "w");
    fprintf(fp, "%d", port_number);  // Write a port number to the file.
    fclose(fp);

    while (1) {
        accept(sockfd, ...);           // Accept a connection from a worker process.
        ...                               // Request processing by the worker process.
    }
}
```

2. Generating an Agent process [job script]

   The job script process uses the mpiexec command to generate an Agent process.

   At this time, pass both the IP address of the node (job master node) where the master process runs and the port number used in communication between the master process and Agent processes to the Agent process.

   The following example defines the shell function print_ipaddr for retrieving the IP address and specifies the result as an argument of start_agent.sh.

```
[Job script example: job_agent.sh]
...
# Retrieve the IP address of the job master node (this node).
IP_ADDR=$(print_ipaddr "tofu1")

# Set an environment variable to execute the mpiexec command.
MPI_HOME=/opt/FJSVxxxx/<version>
export PATH=.:${MPI_HOME}/bin:${PATH}
export LD_LIBRARY_PATH=${MPI_HOME}/lib64:/lib64:${LD_LIBRARY_PATH}

# Generate an Agent process with the mpiexec command.
mpiexec start_agent.sh "${IP_ADDR}" "${PORT_NUM}" &
...
```

Information

- In MPI_HOME, specify the standard path to the installation directory of the mpiexec command of Development Studio. For details on this path, see "MPI User's Guide," which is a Development Studio manual, or contact the administrator.
- The shell function print_ipaddr returns the IP address of the network interface specified in an argument. This example uses that function to pass the IP address of the job master node to the Agent process.
  "tofu1" is specified in an argument for the name of the network interface of the Tofu interconnect on the FX server. Use this name to execute the worker process on the FX server.

start_agent.sh is a shell script used to execute the program agent.out, which will become an Agent process.

```
[start_agent.sh example]
#!/bin/bash

./agent.out $@
```

Note

The shell script start_agent.sh executes the program mpi_init_finalize.out, which only calls the MPI_Init() and MPI_Finalize() functions.
The job operation software expects the program executed by the mpiexec command to be an MPI program, in which case the MPI_Init() and MPI_Finalize() functions are called. If the Agent process ends without these functions being called, an abnormal end is assumed and the job will be forcibly terminated.
To execute a non-MPI program in this example, the program mpi_init_finalize.out is executed in advance so that no abnormal end is assumed. The program only calls the MPI_Init() and MPI_Finalize() functions.

3. Connecting to the master process [Agent process]

   The Agent process connects to the master process of the job master node through a socket connection, etc. using the IP address of the job master node specified in an argument.

   The connection established here is used for communication between the master process and worker processes and for an alive check of the worker process by the master process.

   Once an Agent process is generated as described above in steps 1 and 2, the Agent process is started on all allocated nodes, including the job master node. If you do not want to execute the Agent process on the job master node, terminate the Agent process on the job master node as shown in the following example.

```
[Agent process program example: agent.c]

#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    // Assign the job master node IP address and port number given by command line arguments.
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);
    char *my_ip_addr;

    get_ipaddr("tofu1", &my_ip_addr);
    // Terminate the Agent process if the local node is the job master node
    // (if it has the same IP address as the job master node).
    if (strcmp(my_ip_addr, ip_addr) == 0) {
      exit(0);
    }

    // Connect to the job master node (through a socket connection, etc.).
    connect_to(ip_addr, port_num);
    // Process according to the request from the job master node.
    ...
}
```

Information

The get_ipaddr() function in the above example is a function for returning the IP address of the local node.

4. Waiting for the master process to end [job script]

   Create the job script such that it ends after waiting for the master process to end. When creating it, use the process ID of the master process. The process ID was retrieved when the master process started.

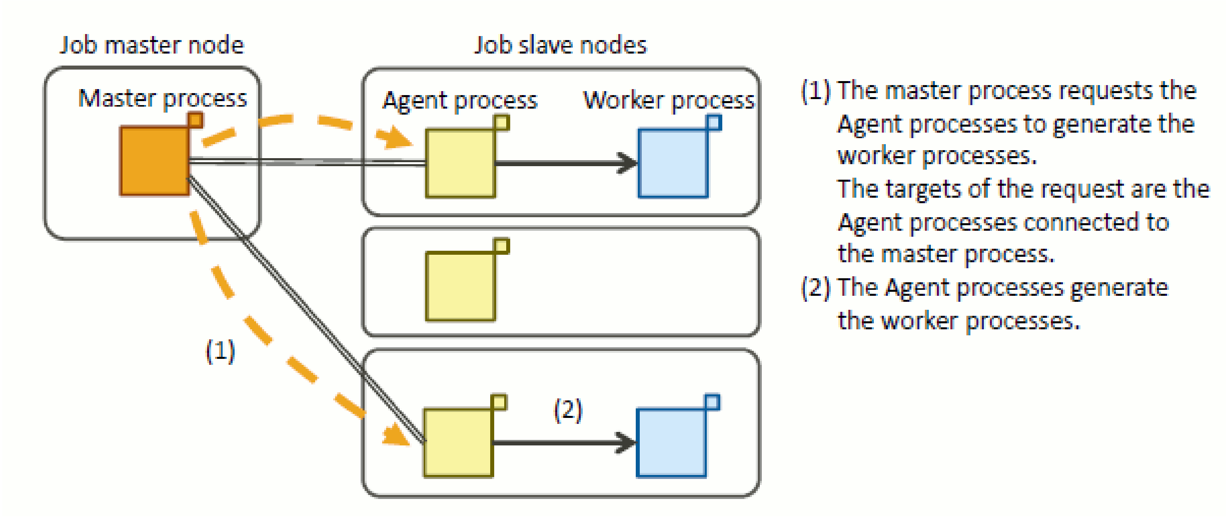
```
[Job script example: job_agent.sh]
...
wait ${MASTER_PID}      # Wait for the master process to end.
...
# Job script end (end of the master-worker job)
```

2.2.3 Generating a worker process

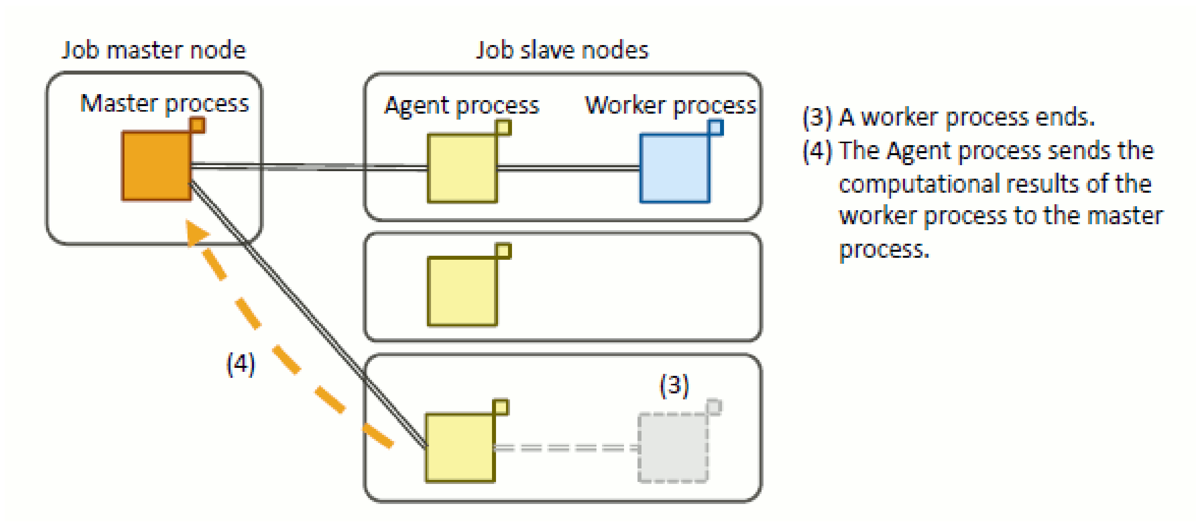
The Agent process generated as described in the previous section generates a worker process upon request from the master process. The Agent process sends the computational results of the worker process to the master process.

The worker process generation method and the method of communication between processes depend on how the program is implemented. For this reason, their descriptions are omitted from this manual.

Figure 2.4 Generating worker processes



Job master node
Job slave nodes
Master process
Agent process
Worker process
(1)
(2)
(1) The master process requests the Agent processes to generate the worker processes. The targets of the request are the Agent processes connected to the master process.
(2) The Agent processes generate the worker processes.


Job master node
Job slave nodes
Master process
Agent process
Worker process
(3) A worker process ends.
(4) The Agent process sends the computational results of the worker process to the master process.
(4)
(3)

2.2.4 Job end

A master-worker job ends with the end of the job script. Therefore, have the job script wait for the master process to end before it itself ends. (See step 4 in "2.2.2 Generating a master process and Agent processes.")

When the job ends, the job operation software terminates the Agent process as well as the worker process started from the Agent process. Therefore, the user does not need to explicitly terminate these processes.

2.3 Generating a Worker Process with the pjaexe Command

Aside from the Agent process method ("2.2 Generating a Worker Process from an Agent Process"), another method of generating a worker process is to use the pjaexe command provided by the job operation software. Use the latter method for non-MPI programs.

In this method, the user needs to implement the following functions:

a. Job script

   Start a worker process with the pjaexe command

b. Master program (master process)

   1. Perform an alive check of a worker process (judging whether there is a connection from the worker process)
   2. Request a worker process to perform a task

c. Worker program (worker process)

   1. Establish a connection with the master process
   2. Send computed results to the master process

The following sections describe processing details.

See

For details on the program configurations and coding examples shown in the descriptions, see "A.2 Shell Script utility.sh" and "A.4 Example of Using the pjaexe Command."

2.3.1 Node allocation

The generation of a worker process with the pjaexe command does not require special consideration of how to specify the number of nodes. However, the following consideration of the node shape may be necessary, depending on the program.

- Cases with communication only between the master process and a worker process

  You do not need to give much consideration to the node shape in this case, provided that the number of nodes is appropriate.

- Cases with communication between worker processes

  Depending on the communication processing, consider a node shape that can shorten the communication distance.

  The following example allocates 96 nodes with the dimensions of 12 x 8 to a job.

```
[Job script example: job_pjaexe.sh]

#!/bin/bash
#PJM -L "node=12x8"
```

2.3.2 Generating a master process

A job script generates a master process.

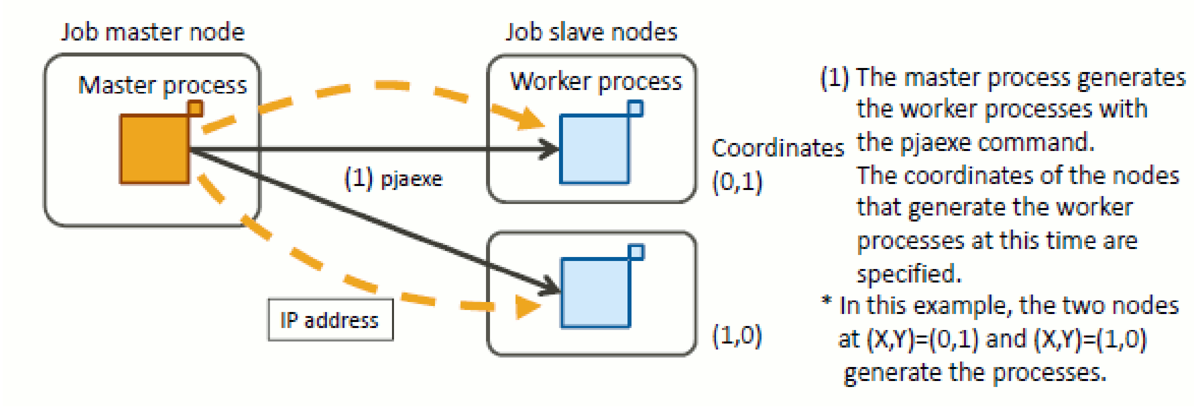
In the following example, the master program master_pjaexe.sh generates a worker process with a shell script.

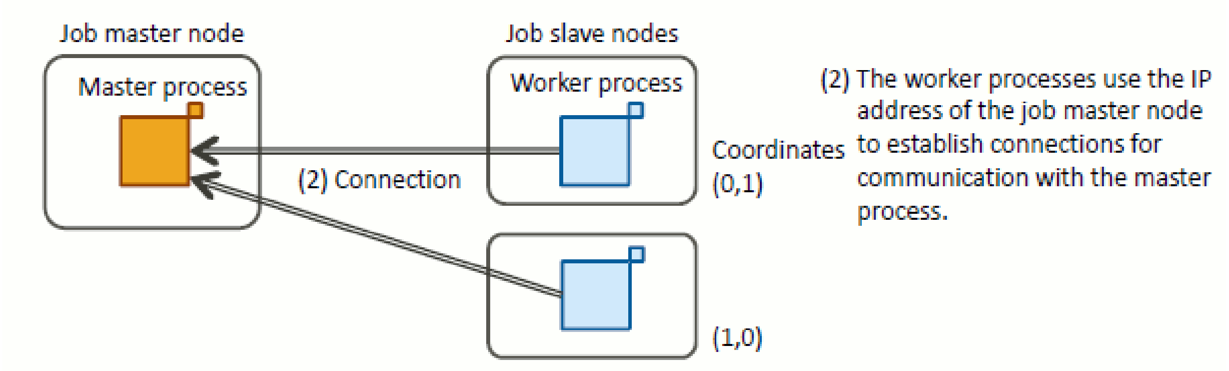
```
[Job script example: job_pjaexe.sh]

./master_pjaexe.sh
```

2.3.3 Generating a worker process

The master process uses the pjaexe command to generate a worker process as follows.

Figure 2.5 Generating worker processes with the pjaexe command


Job master node
Master process
Job slave nodes
Worker process
(1) pjaexe
IP address
Coordinates
(0,1)
(1,0)
(1) The master process generates the worker processes with the pjaexe command.
The coordinates of the nodes that generate the worker processes at this time are specified.
* In this example, the two nodes at (X,Y)=(0,1) and (X,Y)=(1,0) generate the processes.


Job master node
Master process
Job slave nodes
Worker process
(2) Connection
Coordinates
(0,1)
(1,0)
(2) The worker processes use the IP address of the job master node to establish connections for communication with the master process.

1. Generating a worker process with the pjaexe command [master process]

   To use the pjaexe command, the master process needs to manage which node generates a worker process.

With the --vcoord or --vcoordfile option of the pjaexe command, specify the coordinates of the node that will generate the worker process. Those coordinates are coordinates in the node shape allocated to the job. For details on the pjaexe command, see the manual "Job Operation Software Command Reference."

The master process is generated at the node located at the coordinate origin. Therefore, for the worker process, select a node at other coordinates.

Also, the worker process will communicate with the master process through an interface, so the interface needs to pass the IP address of the job master node where the master process runs. (Example: Program argument, environment variable, etc.)

The pjaexe command ends immediately when the worker process is generated. The worker process continues being executed.

Note

- Up to 128 pjaexe commands can be executed simultaneously in one job. If the pjaexe command is executed in excess of 128, the following message is output and the pjaexe command terminates abnormally.

```
[ERR.] PLE 0050 plexec cannot be executed any further.
```

- By specifying *vcoordfile* with the --vcoordfile option, the pjaexe command can generate processes on multiple nodes at the same time in one execution, reducing the number of times the pjaexe command is executed.

  [*vcoordfile* example]

```
(0)
(1)
(2)
(3)
(4)
```

  [Job script description example]

```
pjaexe --vcoordfile vcoordfile command "${IP_ADDR}" "${PORT_NUM}"
```

- The standard input of the *command* executed by the pjaexe command is set to /dev/zero, and the standard output and standard error output are set to /dev/null.
  The following is a description example for directing the standard output and standard error output of a.out to a file when executing the pjaexe command from a job script.

```
mkdir ./stdout ./stderr
pjaexe --vcoord "(0)" './a.out >> ./stdout/0.txt 2>> ./stderr/0.txt'
```

- If the node specified by the pjaexe command is down, the command does not return. Therefore, the user needs to create a mechanism to time out the pjaexe command in the event that it does not return within a specific period.
  The following example defines and uses the shell function timeout.

Information

As with the node shape, the specification of node coordinates is associated with the communication distance between worker processes.
To enable efficient communication between worker processes, specify the coordinates so as to shorten the distance between nodes.

2. Connecting to the master process [worker process]

   The worker process establishes a communication path to the master process, based on the IP address passed from the master process.

   The specific procedure depends on the communication method of the program, so its descriptions are omitted here.

The following example shows the shell script master_pjaexe.sh for creating a program that will become a master process (master program).

```
[Master program example: master_pjaexe.sh]
...
```

```
. utility.sh

# Retrieve the IP address of the job master node (this node).
IP_ADDR=$(print_ipaddr "tofu1")

# Initialize the socket for accepting a connection from a worker process.
PORT_NUM=port_number
...

# Start the worker process worker.out on all job slave nodes.
for X in $(seq 0 11); do
    for Y in $(seq 0 7); do
        VCOORD="(${X},${Y})"
        # Time out the pjaexe command if it does not return within 60 seconds.
        timeout 60 pjaexe --vcoord \"${VCOORD}\" ./worker.out "${IP_ADDR}" "${PORT_NUM}"
        RC=$?
        if [ "${RC}" -eq 1 ]; then
            # Abnormally terminate the master process if there is a user specification error.
            exit 1
        fi
        if [ "${RC}" -ne 0 ]; then
            # Assume the node failed and add it to the faulty node list if the pjaexe command
            # ends abnormally.
            echo "${VCOORD}" >> broken_node_list.txt
        fi
    done
done
...
```

Information

- Generally, the program that will become a master process is written in a programming language such as C or Fortran. However, to describe the processing logic, this section shows an example where the master program is implemented with a shell script.
  The processes that are difficult to implement with a shell script (a socket initialization process and a process for communication with a worker process) are omitted. For these processes, use the general procedures for communication between processes as a reference.

- The ranges for the specified coordinates of the node generating a worker process in the above example are 0 to 11 for the X-coordinate and 0 to 7 for the Y-coordinate.

- The script utility.sh is read first to define the utility functions print_ipaddr and timeout to retrieve an IP address.

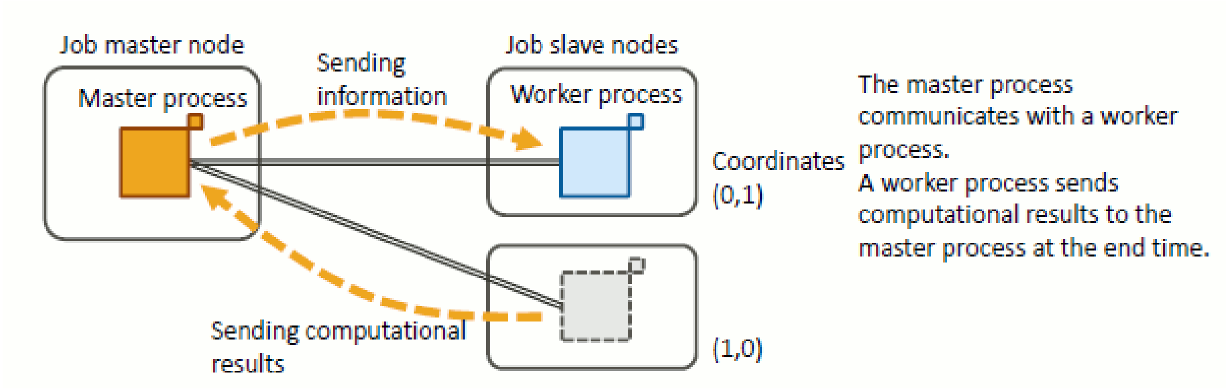
- The shell function timeout executes the specified command with a timeout. If the command does not return within the specified time (seconds), the signal SIGKILL is sent to its process.
  In this example, if the pjaexe command does not return within 60 seconds, the signal SIGKILL is sent to the process of the pjaexe command to forcibly terminate it.

2.3.4 Communication of a master process and worker processes

The master process and a worker process communicate with each other using the communication path established as described in step 2 of the previous section.

The specific procedure depends on the communication method of the program, so its descriptions are omitted here.

Figure 2.6 Communication of a master process and worker processes


Job master node
Master process
Sending information
Job slave nodes
Worker process
Coordinates (0,1)
(1,0)
Sending computational results
The master process communicates with a worker process.
A worker process sends computational results to the master process at the end time.

2.3.5 Job end

A master-worker job ends with the end of the job script. Therefore, have the job script wait for the master process to end before it itself ends.

When the job ends, the job operation software terminates its worker processes. Therefore, the user does not need to explicitly terminate the worker processes.

2.4 Notes on Creating a Master-Worker Job

Note the following about creating master-worker jobs.

- For a master-worker job, the job creator needs to plan for the generation of worker processes, the detection of their errors, and corrective actions for the errors.
  The specific method of implementation depends on the job and program processing contents and processing logic, so some descriptions are omitted from the implementation examples described in this chapter.

- On the dynamic generation of the worker process, when the node on which the worker process is running is down, the worker processes on the all nodes which belong same MPI_COMM_WORLD cannot be worked.
  The worker processes on these nodes remains until the user terminates them or the job is terminated.
  The node on which the worker process run does not choose again to re-generate the worker process until the termination of the mpiexec command.
  Note that the downed node might be chosen again as a node to generate the worker process when the mpiexec command is re-executed.

- In the implementation method described in "2.3 Generating a Worker Process with the pjaexe Command," a node generates a process because of a user's instruction. Therefore, the --mpi rank-map-hostfile option has no meaning and is ignored even if specified.

- Master-worker jobs cannot be executed as interactive jobs but only as batch jobs.
  Also, master-worker jobs are a job model used exclusively with step jobs and bulk jobs.

- Note the following about using MPI communication functions in master-worker jobs.

  If MPI communication processing fails, the calling process of the communication function also ends abnormally by default in accordance with the MPI standard. An example of such failure is a case where the process at the communication destination ends abnormally during processing or the node at the communication destination is down during processing.

  This communication processing is executed not only when the user explicitly calls an MPI communication function such as MPI_Send(), MPI_Recv(), or MPI_Bcast() but also in internal processing of the MPI library. As for the program in such cases, the master process will seem to end suddenly and abnormally while executing a process unrelated to communication.

  When using an MPI communication function in a master-worker job, take the following actions to prevent an abnormal end of a worker process from abnormally terminating the master process. They will reduce the possibility of the abnormal end of the master process.

  1. Call the MPI_Comm_spawn() function, and then call the FJMPI_Mswk_disconnect() function.

     When you use the MPI_Comm_spawn() function to dynamically generate a process, a connection is established for communication between the generated process and the calling process of the MPI_Comm_spawn() function.

Internal communication processing of MPI occurs while processes are connected for this communication and does not occur when they are disconnected. Therefore, after calling MPI_Comm_spawn(), you will need to call FJMPI_Mswk_disconnect() as shown in the example in "2.1.4 Generating a worker process."

2. Call FJMPI_Mswk_connect() or FJMPI_Mswk_accept() before calling an MPI communication function. Also, after calling the MPI communication function, call FJMPI_Mswk_disconnect().

   If the process at the connection destination ends abnormally, the FJMPI_Mswk_connect(), FJMPI_Mswk_accept(), and FJMPI_Mswk_disconnect() functions only return abnormally, and the calling process does not end abnormally.

   Therefore, before calling an MPI communication function, you need to call the FJMPI_Mswk_connect() function every time, as shown in the following example. This reduces the impact of a worker process error on the master process.

```
[Example] Connecting from the master process to a worker process

// Master process
FJMPI_Mswk_connect(worker_port, ..., &worker_comm);
MPI_Send(..., worker_comm, ...);
FJMPI_Mswk_disconnect(&worker_comm);

// Worker process
FJMPI_Mswk_accept(worker_port, ..., &master_comm);
MPI_Recv(..., master_comm, ...);
FJMPI_Mswk_disconnect(&master_comm);
```

   If you do not follow the above procedure, the master process may end abnormally at any given time after a worker process ends abnormally or after the node where a worker process is running goes down. The abnormal end of the master process would also cause the abnormal end of the mpiexec command. However, the job script would continue being executed.

Chapter 3 Effects of System Abnormalities

This chapter describes the effects from system errors that may happen, such as a node going down during execution of a master-worker job.

3.1 Effect on Job Operation

An error may cause a master-worker job to end or continue, depending on the error contents.

- Cases where the master-worker job ends

  As with other job models, the master-worker job ends and is queued again in the following cases.

  - The job master node is down.
  - The storage I/O node is down.
  - The job attempted to be suspended or resumed when the job slave node was down.
  - An ICC or port failure occurred on the compute node allocated to the master-worker job.
  - The administrator (cluster administrator) has specified that the master-worker job end immediately at this time to separate the nodes allocated to the job from operation.

- Cases where the master-worker job continues

  The master-worker job continues in the following cases. However, it is necessary to consider measures in the user program in case this situation causes an error in a worker process. Such measures include executing the worker process on another node.

  - The job operation software on a job slave node has a service error.
  - A job slave node is down.
  - A job slave node has a hardware (CPU or memory) error.

Information

- When the job is terminated due to the user factor (ex: excess of the resource limit such as CPU time), whether the job is queued again depends on the indication at the job submitting or the setting of the job ACL function as well as other job models.
- The storage I/O node is I/O node that is responsible for input/output to first-layer storage, and there is one node in 16 nodes. For details, see the "Job Operation Software Overview" manual.

3.2 Effect on Job Statistical Information

The following table shows the job statistical information output by pjsub -s/-S and pjstat -v in cases where a node failure occurs during execution of a master-worker job.

Item	Value
PC, PJM CODE (Job end code)	The job end code of job statistical information is the same as for a normal job in cases where the job master node is down or the master-worker job ends abnormally because of an FX server failure (ICC error). Like when only a job slave node is down, the error may not affect continuation of the master-worker job. If so, the master-worker job is executed until it ends. When the job ends normally, the job end code is 0.
REASON (End cause)	The end cause is the same as the above-mentioned job end code except when the job ends normally. When that happens, the end cause is "-".
name (REQUIRE) (Required resource amount)	Regardless of whether the node is down, an item with "(REQUIRE)" such as "NODE NUM (REQUIRE)" has the value specified by the user at the job submission time.

Item	Value
name (ALLOC) (Allocated resource amount)	Regardless of whether the node is down, an item with "(ALLOC)" such as "NODE NUM (ALLOC)" has the value determined at the job submission time.
name (USE) (Resource amount used)	An item with "(USE)" such as "NODE NUM (USE)" has a value that excludes the amount for failed nodes.

3.3 Effect on Command Display

The pjshowrsc command provided by the job operation software can display the number of nodes as compute resources.

If the node allocated to a master-worker job goes down during job execution, that node is excluded from the usable resources.

a. No argument specified

   The TOTAL and ALLOC values decrease by the number of nodes that are down.

```
[Before any job slave node goes down]
$ pjshowrsc
  CLUSTER          NODE
                   TOTAL   FREE  ALLOC
  mswk-comp          194    182     12

[After one of the job slave nodes goes down]
$ pjshowrsc
  CLUSTER          NODE
                   TOTAL   FREE  ALLOC
  mswk-comp          193    182     11
```

b. -l option specified

   The TOTAL and ALLOC values of each resource decrease by amounts corresponding to the number of nodes that are down.

```
[Before any job slave node goes down]
$ pjshowrsc -l
     RSC    TOTAL    FREE   ALLOC
    node      384     372      12
     cpu     3072    2976      96
     mem    5.3Ti   5.1Ti   180Gi

[After one of the job slave nodes goes down]
$ pjshowrsc -l
     RSC    TOTAL    FREE   ALLOC
    node      383     372      11
     cpu     3064    2976      88
     mem    5.3Ti   5.1Ti   170Gi
```

c. -v option specified

   No information on the nodes that are down is displayed.

```
[Before any job slave node goes down]
$ pjshowrsc -c mswk-comp -v 1
  [ CLST: mswk-comp ]
  [ NODEGRP: 0x01 ]
  [ NODE: 0x01010010 ]
       RSC    TOTAL    FREE   ALLOC
       cpu       16       0      16
       mem      57Gi       0    57Gi

  RUNNING_JOBS:1731987

  [ NODE: 0x01010011 ]
       RSC    TOTAL    FREE   ALLOC
       cpu       16       0      16
       mem      57Gi       0    57Gi

  RUNNING_JOBS:1731987

[After one of the job slave nodes goes down (0x01010011)]
$ pjshowrsc -c mswk-comp -v 1
  [ CLST: mswk-comp ]
  [ NODEGRP: 0x01 ]
  [ NODE: 0x01010010 ]
       RSC    TOTAL    FREE   ALLOC
       cpu       16       0      16
       mem      57Gi       0    57Gi

  RUNNING_JOBS:1731987
  * No information on node 0x01010011, which is down, is displayed.
```

Appendix A Program Examples

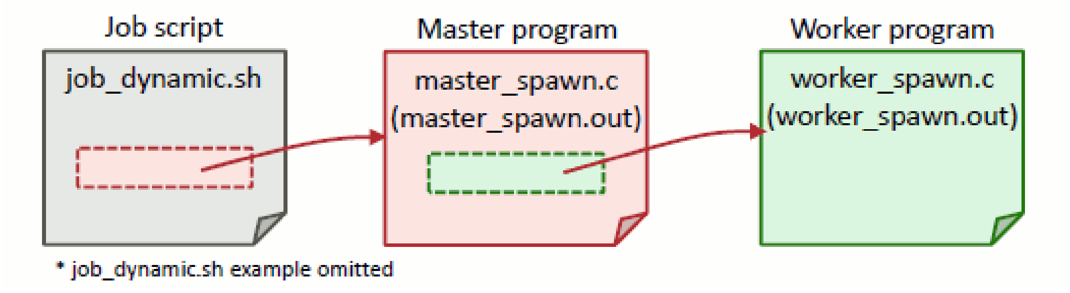
This appendix shows examples of job scripts and programs of master-worker jobs.

A.1 Example of Dynamically Generating a Worker Process

This section shows examples of the programs described in "2.1 Dynamically Generating a Worker Process."

For details on creating MPI programs, see the MPI specifications or "MPI User's Guide," which is a Development Studio manual.

Figure A.1 Program configuration



[Master program master_spawn.c]

```
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    int world_size, universe_size;
    int *universe_size_p;
    int flag;
    MPI_Status status;

    MPI_Comm worker_comm;
    char master_port[MPI_MAX_PORT_NAME] = "";
    char worker_port[MPI_MAX_PORT_NAME] = "";

    // Root rank of each communicator
    const int self_root   = 0; // SELF   (MPI_COMM_SELF)
    const int master_root = 0; // MASTER (MPI_COMM_WORLD)
    const int worker_root = 0; // WORKER (worker_comm)

    const int tag = 0;
    char *message = "Hello";

    // Initialization processing
    MPI_Init(&argc, &argv);

    // Retrieve world_size and universe_size.
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    if (world_size != 1) {
        // If multiple master processes exist
        fprintf(stderr, "Error! world_size=%d (expected 1)", world_size);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &universe_size_p, &flag);
    if (flag == 0) {
        // If retrieval of universe_size fails
        fprintf(stderr, "Error! cannot get universe_size");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    universe_size = *universe_size_p;
    printf("universe_size=%d\n", universe_size);
    if (universe_size == 1) {
        // If universe_size is 1
        fprintf(stderr, "Error! universe_size=%d (expected > 1)", universe_size);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // Open the port for communication with a worker process.
    MPI_Open_port(MPI_INFO_NULL, master_port);
    printf("master_port=%s\n", master_port);

    // Generate a worker process.
    MPI_Comm_spawn("./worker_spawn.out", MPI_ARGV_NULL, universe_size - 1,
                   MPI_INFO_NULL, self_root, MPI_COMM_SELF, &worker_comm, MPI_ERRCODES_IGNORE);

    // Send a port name to a worker process.
    MPI_Send(master_port, MPI_MAX_PORT_NAME, MPI_CHAR, worker_root, tag, worker_comm);

    // Terminate the connection with a worker process.
    FJMPI_Mswk_disconnect(&worker_comm);

    // Receive data from a worker process. (The port name of the worker process has been sent.)
    FJMPI_Mswk_accept(master_port, MPI_INFO_NULL, self_root, MPI_COMM_SELF, &worker_comm);
    MPI_Recv(worker_port, MPI_MAX_PORT_NAME, MPI_CHAR, worker_root, tag, worker_comm, &status);
    printf("worker_port=%s\n", worker_port);
    FJMPI_Mswk_disconnect(&worker_comm);

    // Send data to a worker process.
    FJMPI_Mswk_connect(worker_port, MPI_INFO_NULL, self_root, MPI_COMM_SELF, &worker_comm);
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, worker_root, tag, worker_comm);
    FJMPI_Mswk_disconnect(&worker_comm);

    // Exit processing
    MPI_Close_port(master_port);
    MPI_Finalize();
}
```

[Worker program worker_spawn.c]

```
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int rank;
    MPI_Status status;

    MPI_Comm master_comm;
    char master_port[MPI_MAX_PORT_NAME] = "";
    char worker_port[MPI_MAX_PORT_NAME] = "";

    // Root rank of each communicator
    const int self_root   = 0; // SELF   (MPI_COMM_SELF)
    const int master_root = 0; // MASTER (master_comm)
    const int worker_root = 0; // WORKER (MPI_COMM_WORLD)

    const int tag = 0;
    char message[100] = "";
    // Initialization processing
    MPI_Init(&argc, &argv);
    MPI_Comm_get_parent(&master_comm);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello! rank=%d\n", rank);

    if (rank == worker_root) {
        MPI_Open_port(MPI_INFO_NULL, worker_port);
        printf("worker_port=%s\n", worker_port); fflush(stdout);
    }

    if (rank == worker_root) {
        // Receive data from the master process.
        MPI_Recv(master_port, MPI_MAX_PORT_NAME, MPI_CHAR, master_root, tag, master_comm, &status);
        printf("master_port=%s\n", master_port); fflush(stdout);
    }

    //  Terminate communication with the master process.
    FJMPI_Mswk_disconnect(&master_comm);

    // Send a port name to the master process.
    if (rank == worker_root) {
        FJMPI_Mswk_connect(master_port, MPI_INFO_NULL, self_root, MPI_COMM_SELF, &master_comm);
        MPI_Send(worker_port, MPI_MAX_PORT_NAME, MPI_CHAR, master_root, tag, master_comm);
        FJMPI_Mswk_disconnect(&master_comm);
    }

    // Receive data from the master process.
    if (rank == worker_root) {
        FJMPI_Mswk_accept(worker_port, MPI_INFO_NULL, self_root, MPI_COMM_SELF, &master_comm);
        MPI_Recv(message, MPI_MAX_PORT_NAME, MPI_CHAR, master_root, tag, master_comm, &status);
        printf("message=%s\n", message);
        FJMPI_Mswk_disconnect(&master_comm);
    }

    // Exit processing
    if (rank == worker_root) {
        MPI_Close_port(worker_port);
    }
    MPI_Finalize();
}
```

A.2 Shell Script utility.sh

This section shows the shell script utility.sh used in "2.2 Generating a Worker Process from an Agent Process" and "2.3 Generating a Worker Process with the pjaexe Command."

The shell script utility.sh is read within the job script.

Note

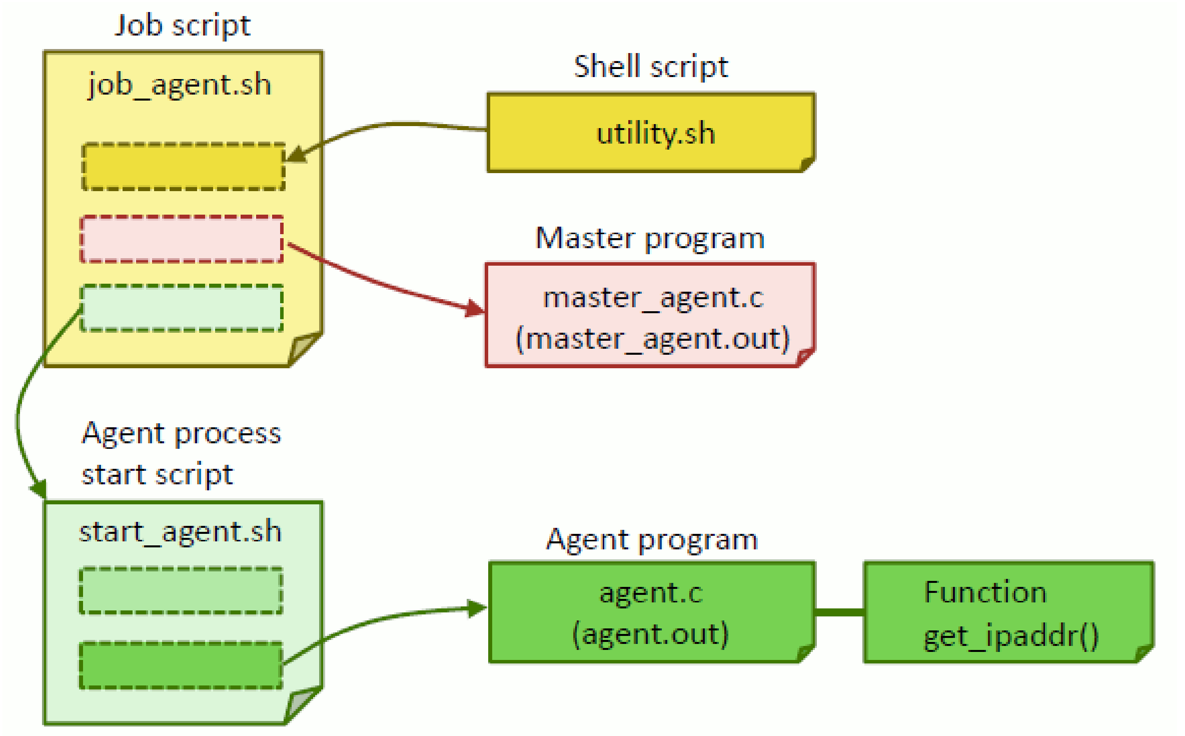
- utility.sh is created by the user.
- Execution privileges must be set for utility.sh, and utility.sh must be accessible from within the job script. utility.sh also applies to stage-in, when using the staging function.

```
# Output the IP address of the specified network interface.
# Usage: print_ipaddr <interface>
print_ipaddr() {
    local INTERFACE=$1
    LANG=C ip addr show dev "${INTERFACE}" | sed -n '/.*inet \([0-9.]*\).*/{s//\1/;p}'
}

# Execute the specified command with a timeout.
# Usage: timeout <timeout_sec> <command> <arg1> <arg2> ...
timeout() {
    local TIMEOUT=$1
    shift 1

    # Record the command execution and process ID.
    eval "$@" &
    local PID=$!

    echo ${PID}
    while true; do
        # Perform an alive check of the command process.
        if ! ps -p "${PID}" >/dev/null 2>&1; then
            # Break out of the loop since the process has been completed.
            break
        fi

        if [ "${TIMEOUT}" -le 0 ]; then
            # Abnormally terminate the process if there was a timeout.
            kill -KILL "${PID}"
            break
        fi

        # Return to the beginning of the loop after sleeping for 1 second.
        sleep 1
        TIMEOUT=$((TIMEOUT - 1))
    done

    # Return the process end code.
    wait "${PID}"
    return $?
}
```

A.3 Agent Process Method Examples

This section shows examples of the scripts and programs described in "2.2 Generating a Worker Process from an Agent Process."

Figure A.2 Program configuration



[Job script job_agent.sh]

```
#!/bin/bash
#PJM -L "node=96"
#PJM --mpi "proc=96"

. utility.sh

# Generate a master process.
./master_agent.out port_num.txt &
MASTER_PID=$!
PORT_NUM=$(cat port_num.txt)

# Retrieve the IP address of the job master node (this node).
IP_ADDR=$(print_ipaddr "tofu1")

# Set the environment variables for executing the mpiexec command.
MPI_HOME=/opt/FJSVxxxx/<version>
export PATH=.:${MPI_HOME}/bin:${PATH}
export LD_LIBRARY_PATH=${MPI_HOME}/lib64:/lib64:${LD_LIBRARY_PATH}

# Generate an Agent process with the mpiexec command.
mpiexec start_agent.sh "${IP_ADDR}" "${PORT_NUM}" &

wait ${MASTER_PID}
```

[Master program master_agent.c]

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>

int main(int argc, char **argv)
{
    char *port_file = argv[1];
    int port_number = 378000;

    int sockfd = socket(...);         // Create a socket.
    bind(sockfd, ...);                // Bind the socket to a specific port.
    listen(sockfd, ...);              // Wait for a connection from a worker process.

    FILE *fp = fopen(port_file, "w");
    fprintf(fp, "%d", port_number);   // Write a port number to the file.
    fclose(fp);

    while (1) {
        accept(sockfd, ...);          // Accept a connection from a worker process.
        ...                           // Request processing by the worker process.
    }
}
```

[Agent process start script start_agent.sh]

```
#!/bin/bash

./agent.out $@
```

[mpi_init_finalize.c]

```
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    MPI_Finalize();
    return 0;
}
```

[Agent program agent.c]

```
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(int argc, char **argv)
{
    // Assign the job master node IP address and port number given by command line arguments.
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);
    char *my_ip_addr;

    get_ipaddr("tofu1", &my_ip_addr);
    // Terminate the Agent process if the local node is the job master node
    // (if it has the same IP address as the job master node).
    if (strcmp(my_ip_addr, ip_addr) == 0) {
      exit(0);
    }

    // Connect to the job master node.
    int sockfd = socket(...);
    connect(sockfd, ...);
    // Process according to the request from the job master node.
    ...
}
```

[get_ipaddr() function]

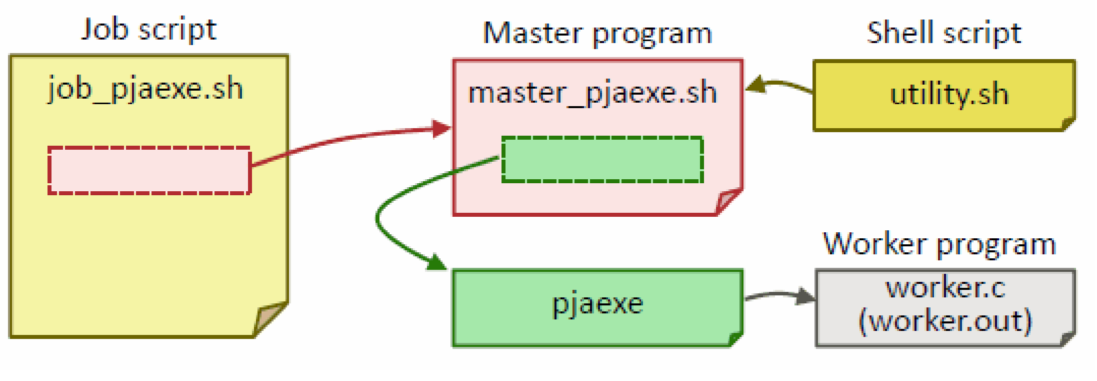
The get_ipaddr() function returns the IP address of the local node as a character string.

```
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <net/if.h>
#include <arpa/inet.h>

int get_ipaddr(const char *device_name, const char **ip_addr)
{
    int fd;
    struct ifreq ifr;

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    ifr.ifr_addr.sa_family = AF_INET;

    strncpy(ifr.ifr_name, device_name, IFNAMSIZ - 1);
    int rc = ioctl(fd, SIOCGIFADDR, &ifr);
    if (rc == 0) {
        *ip_addr = inet_ntoa(((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr);
    }
    close(fd);

    return rc;
}
```

A.4 Example of Using the pjaexe Command

This section shows an example of the script described in "2.3 Generating a Worker Process with the pjaexe Command."

Figure A.3 Program configuration



[Job script job_pjaexe.sh]

```
#!/bin/bash
#PJM -L "node=12x8"
```

```
# Execute the master process.
./master_pjaexe.sh
```

[Master program master_pjaexe.sh]

Generally, the master program is written in a programming language such as C or Fortran. However, to describe the processing logic, an example of creating it with a shell script is shown here.

```
#!/bin/bash

. utility.sh

# Retrieve the IP address of the job master node (this node).
IP_ADDR=$(print_ipaddr "tofu1")

# Initialize the socket for accepting a connection from a worker process. (*)
PORT_NUM=port number
...

# Start the worker processes on all job slave nodes.
for X in $(seq 0 11); do
    for Y in $(seq 0 7); do
        VCOORD="(${X},${Y})"
        # Time out the pjaexe command if it does not return within 60 seconds.
        timeout 60 pjaexe --vcoord \"${VCOORD}\" ./worker.out "${IP_ADDR}" "${PORT_NUM}"
        RC=$?
        if [ "${RC}" -eq 1 ]; then
            # Abnormally terminate the master process if there is a user specification error.
            exit 1
        fi
        if [ "${RC}" -ne 0 ]; then
            # Assume the node failed and add it to the faulty node list
            # if the pjaexe command ends abnormally.
            echo "${VCOORD}" >> broken_node_list.txt
        fi
    done
done

# Summary processing of communication with the worker process worker.out and computed results
...<Omitted> ...

# Master process end
exit
```

(*) Generally, the program that will become a master process is written in a programming language such as C or Fortran. However, to describe the processing logic, this section shows an example where the master program is implemented with a shell script.
The processes that are difficult to implement with a shell script (a socket initialization process and a process for communication with a worker process) are omitted. For these processes, use the general procedures for communication between processes as a reference.

[Worker program worker.c]

An example of getting the rank number with the environment variable PLE_VPID is shown here.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    // Substitute the IP address and port number of the job master node given by the command line
argument.
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);

    // Get the rank number.
    char *env_p;
    int  rank = 0;
    env_p = getenv("PLE_VPID");
    if (env_p != NULL) {
        rank = atoi(env_p);
    } else {
        fprintf(stderr, "client: getenv error\n");
        exit(1);
    }
    printf("Hello! rank=%d\n", rank);

    // Process according to the request from the job master node.
    ...
}
```