FUJITSU

FUJITSU Software
Technical Computing Suite V4.0L20

ジョブ運用ソフトウェア
エンドユーザ向けガイド
マスタ・ワーカ型ジョブ編

J2UL-2536-01Z0(02)
2022年3月

まえがき

本書の目的

本書では、Technical Computing Suiteのジョブ運用ソフトウェアにおけるジョブモデルの1つのマスタ・ワーカ型ジョブについて説明します。

本書の読者

本書は、マスタ・ワーカ型ジョブを作成、実行するエンドユーザが対象です。

本書を読むためには、以下の知識が必要です。

- Linux に関する基本的な知識
- ジョブ運用ソフトウェアを利用してジョブを投入、実行する知識 (マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」を参照)

本書の構成

本書は、次の構成になっています。

第1章 マスタ・ワーカ型ジョブとは

マスタ・ワーカ型ジョブの概要を説明します。

第2章 マスタ・ワーカ型ジョブの作成

ジョブの操作方法について説明します。

第3章 システム異常時の影響

マスタ・ワーカ型ジョブ実行中にノードダウンなどの異常が起こった場合の影響について説明します。

付録A プログラム例

本書で説明するマスタ・ワーカ型ジョブのプログラム例です。

本書の表記について

機種名の表現

本書では富士通製CPU A64FXを搭載した計算機を「FXサーバ」、FUJITSU server PRIMERGYを「PRIMERGYサーバ」(または単に「PRIMERGY」)と表記します。
また、本書で説明する機能の一部には、対象機種によって仕様に差があります。このような機能の説明では、以下のように対象機種を略称で表記します。
[FX] : FXサーバを対象にした機能です。
[PG] : PRIMERGYサーバを対象にした機能です。

マニュアル内のアイコンについて

本書では、以下のアイコンを使用しています。

注意

特に注意が必要な事項を説明しています。必ずお読みください。

参照

詳細な情報が書かれている参照先を示しています。

参考

ジョブ運用ソフトウェアに関連した参考記事を説明しています。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

商標

- Linux®は米国及びその他の国におけるLinus Torvaldsの登録商標です。
- そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。

出版年月および版数

版数	マニュアルコード
2022年3月 第1.2版	J2UL-2536-01Z0(02)
2020年6月 第1.1版	J2UL-2536-01Z0(01)
2020年2月 初版	J2UL-2536-01Z0(00)

著作権表示

変更履歴

変更内容	変更箇所	版数
マスタ・ワーカ型ジョブは、pjsub コマンドの --mswk オプションを指定して投入することを記載しました。	第2章	第1.2版
ジョブ内での pjaexe コマンドの同時実行数は 128 個までであることを記載しました。	2.3.3	
ストレージ I/O ノードの説明と参照先を記載しました。	3.1	
ランク番号の求め方がわかるサンプルプログラム例を記載しました。	A.4 図A.3	
サンプルプログラムの誤記やわかりにくい部分を修正しました。	2.3.3 A.3 A.4	第1.1版
ストレージI/Oノードやジョブスレーブノードがダウンしたときのマスタ・ワーカ型ジョブへの影響を記載しました。	3.1	

目　次

第1章 マスタ・ワーカ型ジョブとは

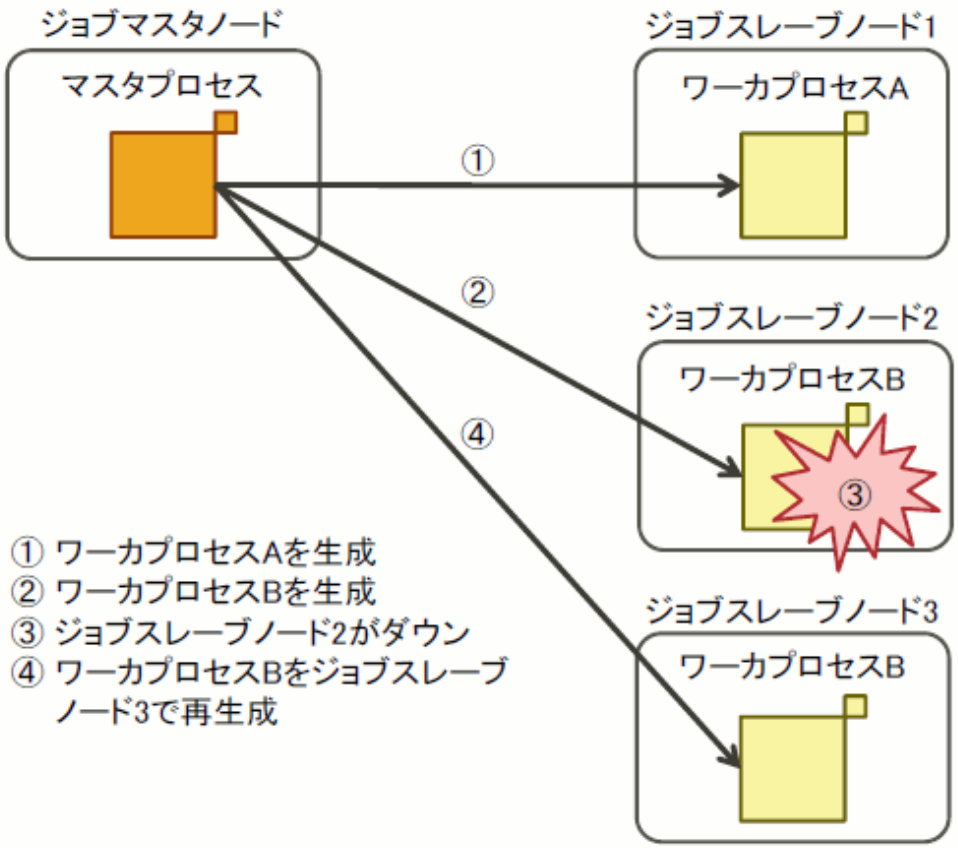
マスタ・ワーカ型ジョブとは、通常ジョブ、ステップジョブ、バルクジョブと並ぶジョブモデルの1つで、以下の特徴を持つジョブです。

- マスタ・ワーカ型ジョブは、ジョブスクリプトプロセス、マスタプロセスおよびワーカプロセスから構成されます。
  マスタプロセスとワーカプロセスが協調することで計算タスクを実行します (タスク:並列プログラムの処理単位)。
- マスタプロセスは、計算タスク全体を統括し、ワーカプロセスの生成や管理、計算結果の取りまとめをします。
  ワーカプロセスは、マスタプロセスから依頼された計算タスクを実行し、結果をマスタプロセスに返します。
- マスタ・ワーカ型ジョブに割り当てられた計算ノードのダウンやプロセスの異常終了が発生しても、ジョブスクリプトプロセスが動作している限り、マスタ・ワーカ型ジョブは継続します。

この特徴を利用して、計算ノードダウンやワーカプロセスの異常終了に対し、ワーカプロセスを別のノードで再実行する仕組みをユーザーが作ることで計算タスクを継続できます。

マスタ・ワーカ型ジョブで実行されるユーザープログラムの実行イメージを以下に示します。

図1.1 マスタ・ワーカ型ジョブの実行イメージ



参考

- ジョブスクリプトプロセスが動作するノードを「ジョブマスタノード」、それ以外のノードを「ジョブスレーブノード」と呼びます。マスタプロセスは、ジョブスレーブノードで動作するワーカプロセスを管理することが目的なので、ジョブマスタノードで動作させる必要があります。
- 複数のプロセスを生成するという点で、類似したジョブモデルに「バルクジョブ」があります。バルクジョブは、同一のジョブスクリプトを複数のサブジョブとして投入する方式で、それぞれのサブジョブは独立して動作します。バルクジョブでは、サブジョブの数はジョブ投入時に指定され、実行中に変化しません。
  一方、マスタ・ワーカ型ジョブでは、マスタプロセスと各ワーカプロセスが1つのジョブとしてプロセス間通信を行いながら動作します。また、ワーカプロセスはマスタプロセスによって動的に生成するため、ジョブの実行中に数が変化します。マスタ・ワーカ型ジョブ以外のジョブモデルについては、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」を参照してください。

ジョブ運用ソフトウェアは、マスタ・ワーカ型ジョブの実装方式として、ワーカプロセスの生成方法の観点で、以下の3つの方式をサポートします。

- MPI の動的プロセス生成によるワーカプロセス生成
  この方式は、マスタプロセス、ワーカプロセスが共に MPI プログラムの場合に利用します。すなわち、ワーカプロセスの生成や通信は MPI の仕組みを利用します。
  ワーカプロセスを生成するノードの選択はジョブ運用ソフトウェアが行います。

- Agent プロセスによるワーカプロセス生成
  この方式は、ワーカプロセスが MPI プログラムでない場合に利用します。
  プロセス間の通信はソケットなどの通信機構を利用し、ワーカプロセスの生成やそれを生成するノードの選択はユーザーが制御・管理します。

- pjaexe コマンドによるワーカプロセス生成
  前項と同様に、この方式もワーカプロセスが MPI プログラムでない場合に利用します。
  この方式でもプロセス間の通信はソケットなどの通信機構を利用し、ワーカプロセスを生成するノードの選択もユーザーが制御・管理します。ただし、ワーカプロセスの生成は、ジョブ運用ソフトウェアが提供するpjaexe コマンドを利用します。

ユーザーは、ジョブの用途に合った方式を選択し、マスタ・ワーカ型ジョブを作成する必要があります。

これらの方式の詳細は、"第2章 マスタ・ワーカ型ジョブの作成" を参照してください。

第2章 マスタ・ワーカ型ジョブの作成

本章では、マスタ・ワーカ型ジョブの3つの実装方式と作成における注意事項について説明します。

参照

マスタ・ワーカ型ジョブは、pjsub コマンドの --mswk オプションを指定して投入します。
詳細は、マニュアル「ジョブ運用ソフトウェア エンドユーザ向けガイド」の"第2章 ジョブの操作方法"およびpjsubコマンドのmanマニュアルを参照してください。

2.1 ワーカプロセスの動的生成

MPIプログラムでマスタプロセスからワーカプロセスを動的に生成する方式では、ユーザーは以下の機能を実装する必要があります。

a. マスタプログラム (マスタプロセス)
    1. ワーカプロセスの生成
    2. ワーカプロセスへの演算実行依頼
    3. ワーカプロセスの生存確認
b. ワーカプログラム (ワーカプロセス)
    1. マスタプロセスからの演算実行依頼受信と演算結果の送信
    2. マスタプロセスへの演算終了通知の送信

以降では、処理詳細を説明します。

参照

説明に登場するプログラムの構成やコーディング例は "A.1 ワーカプロセスを動的生成する例" を参照してください。

2.1.1 ノードの割り当て

ワーカプロセスを動的に生成する方式では、MPI プログラムの開始時に生成されるマスタプロセスがジョブマスタノードだけで起動するように、pjsub コマンドの --mpi "shape=1" オプションを指定する必要があります。

以下の例では、ジョブに対して96ノードを割り当て、MPI プログラム起動時に生成されるマスタプロセスに1ノードを割り当てます。残りの95ノードが、ワーカプロセスを動的に生成するためのノードになります。

```
[ジョブスクリプトの例: job_dynamic.sh]

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

2.1.2 マスタ・ワーカ型ジョブ用のMPI関数およびMPIサブルーチンについて

マスタ・ワーカ型ジョブで実行する MPI プログラムでは、一部の MPI 関数やMPIサブルーチンをマスタ・ワーカ型ジョブ向けのものに置き換える必要があります。これはジョブ運用ソフトウェアの内部でマスタ・ワーカ型ジョブ固有の処理をする必要があるためです。

以下に、これらのMPI関数名およびMPIサブルーチン名を示します。

表2.1 マスタ・ワーカ型ジョブ用のMPI関数 (C言語)

通常ジョブでの MPI 関数名	マスタ・ワーカ型ジョブ用MPI関数名
MPI_Comm_connect()	FJMPI_Mswk_connect()

通常ジョブでの MPI 関数名	マスタ・ワーカ型ジョブ用MPI関数名
MPI_Comm_disconnect()	FJMPI_Mswk_disconnect()
MPI_Comm_accept()	FJMPI_Mswk_accept()

注意

上記のマスタ・ワーカ型ジョブ用MPI関数は、ヘッダファイル mpi-ext.h で宣言されています。

表2.2 マスタ・ワーカ型ジョブ用のMPIサブルーチン (Fortran 言語)

通常ジョブでの MPI サブルーチン名	マスタ・ワーカ型ジョブ用MPIサブルーチン名
MPI_COMM_CONNECT()	FJMPI_MSWK_CONNECT()
MPI_COMM_DISCONNECT()	FJMPI_MSWK_DISCONNECT()
MPI_COMM_ACCEPT()	FJMPI_MSWK_ACCEPT()

注意

上記のマスタ・ワーカ型ジョブ用MPIサブルーチンは、モジュールmpi_f08_extとmpi_extで宣言されています。モジュールmpi_f08_extとmpi_extは、それぞれMPIのモジュールmpi_f08とmpiに対応しますので、どちらかをUSE文で引用できます。

マスタ・ワーカ型ジョブ用MPI関数およびMPIサブルーチンの仕様は、通常ジョブ用のものと同じですので、MPI の仕様やDevelopment Studioのマニュアル「MPI 使用手引書」を確認してください。

以降では、これらのマスタ・ワーカ型ジョブ用MPI関数を利用した例になっています。

2.1.3 マスタプロセスの生成

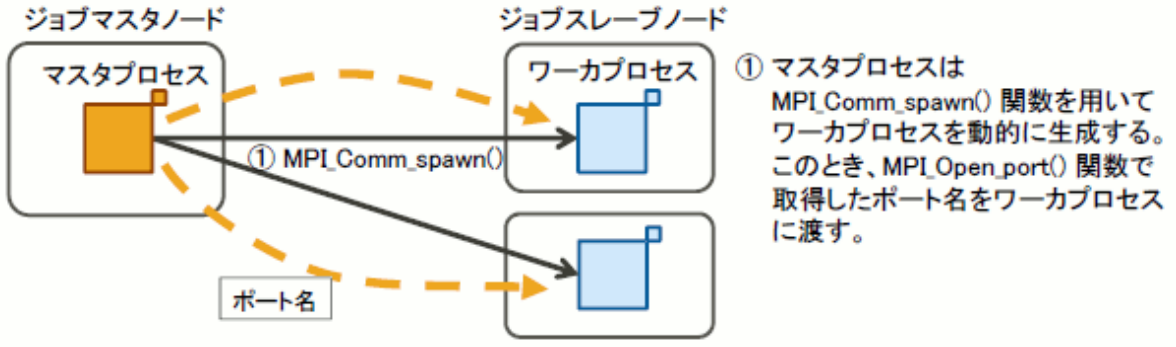
mpiexec コマンドを使用して MPI プログラムを実行し、最初に生成されるプロセスがマスタプロセスとなります。

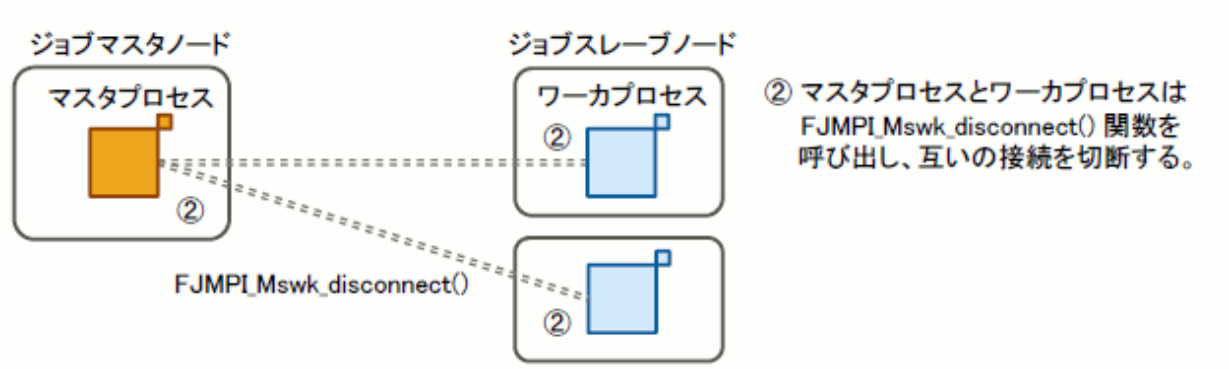
```
[ジョブスクリプトの例: job_dynamic.sh]
...
mpiexec ./master_spawn.out          ← MPIプログラム master_spawn.out を実行
...
```

2.1.4 ワーカプロセスの生成

MPIプログラム開始時に生成されたマスタプロセスは、MPI_Comm_spawn() (またはMPI_Comm_spawn_multiple()) 関数を使用してワーカプロセスを動的に生成します。

図2.1 MPI_Comm_spawn()関数によるワーカプロセスの生成






```
[マスタプロセスのプログラム例: master_spawn.c]

...
// ワーカプロセスとの通信用ポートをオープンする。
MPI_Open_port(MPI_INFO_NULL, master_port);
...

// ワーカプロセスを生成する。
MPI_Comm_spawn("./worker_spawn.out", ..., &worker_comm, ...);

// ワーカプロセスへポート名を送信する。
MPI_Send(master_port, ..., worker_comm);

// ワーカプロセスとの接続を切断する。
FJMPI_Mswk_disconnect(&worker_comm);
...
```

参考

上記例では、ワーカプロセス生成後、FJMPI_Mswk_disconnect() 関数を呼び出し、ワーカプロセスとの接続を切断しています。これは、MPI の仕様に基づいた処理です。詳細は "2.4 マスタ・ワーカ型ジョブ作成における注意事項" を参照してください。

2.1.5 マスタプロセスとワーカプロセスの通信

マスタプロセスとワーカプロセスは以下のように通信します。

1. 通信開始

   マスタプロセスは FJMPI_Mswk_accept() 関数を、ワーカプロセスは FJMPI_Mswk_connect() 関数を呼び出し、接続を確立します。マスタプロセスが FJMPI_Mswk_connect() 関数を、ワーカプロセスが FJMPI_Mswk_accept() 関数を呼び出してもかまいません。この場合、ワーカプロセスが MPI_Open_port() 関数を呼び出し、得られたポート名をマスタプロセスに送信してください。

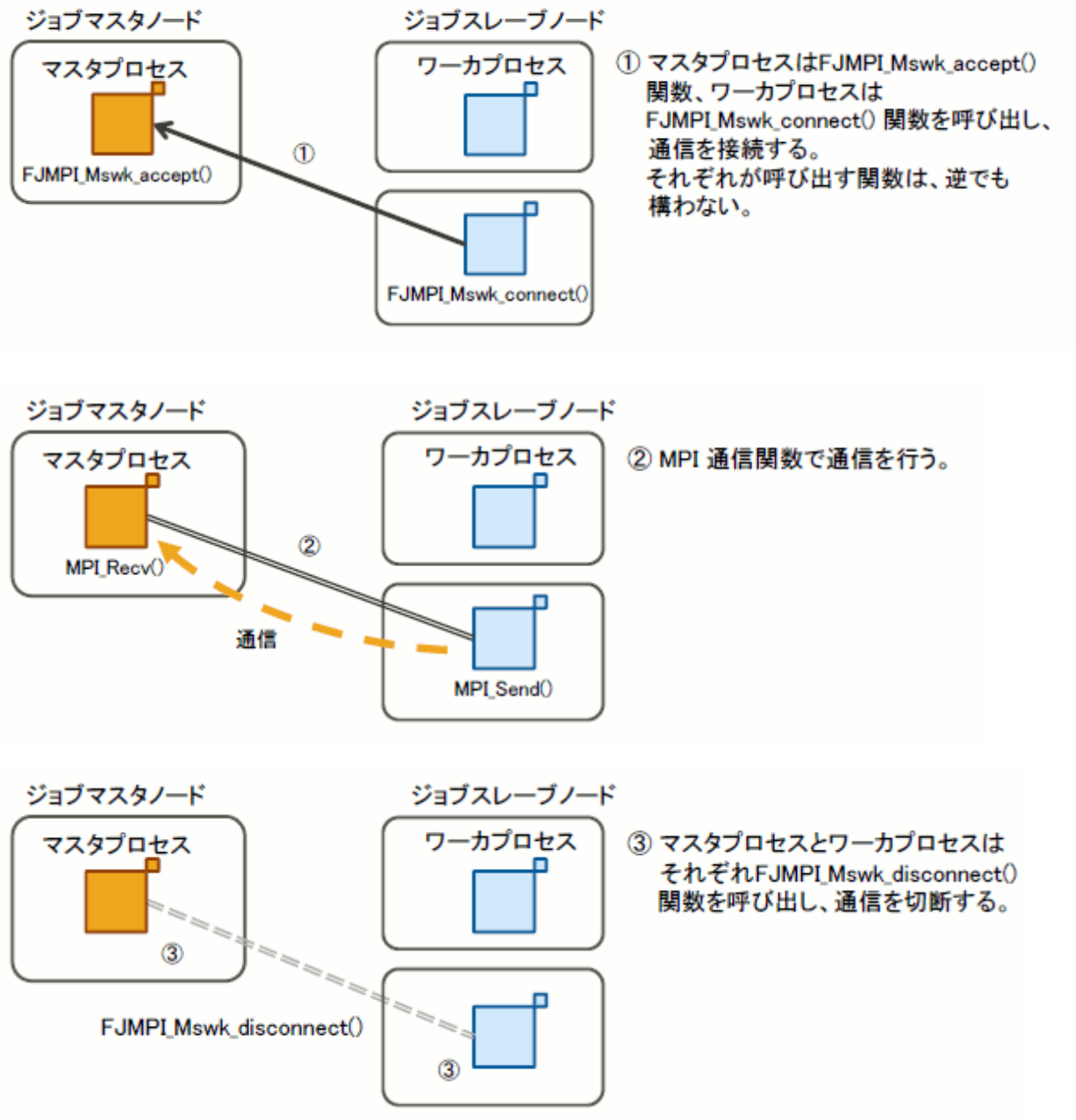
2. 通信処理

   MPI 通信関数 MPI_Send() や MPI_Recv() を使用して通信します。

3. 通信終了

マスタプロセス、ワーカプロセスそれぞれが FJMPI_Mswk_disconnect() 関数を呼び出し、接続を切断します。

図2.2 MPI 通信関数を使用したマスタプロセスとワーカプロセス間の通信



以下に、マスタプロセスとワーカプロセスのプログラム例を示します。

```
[マスタプロセスのプログラム例: master_spawn.c]
...
// ワーカプロセスからの接続を受け付ける。
FJMPI_Mswk_accept(master_port, ..., &worker_comm);
// ワーカプロセスからのデータを受信する。
MPI_Recv(..., worker_comm, ...);
// ワーカプロセスとの通信を切断する。
FJMPI_Mswk_disconnect(&worker_comm);
...
```

```
[ワーカプロセスのプログラム例: worker_spawn.c]
...
// マスタプロセスへ接続する。
FJMPI_Mswk_connect(master_port, ..., &master_comm);
// マスタプロセスへデータを送信する。
MPI_Send(..., master_comm);
// マスタプロセスとの通信を切断する。
FJMPI_Mswk_disconnect(&master_comm);
...
```

2.1.6 ジョブの終了

ジョブスクリプトから実行したmpiexecコマンドは、最初に生成したマスタプロセスが終了すると復帰します。このとき、ワーカプロセスの終了は待ち合わせません。

その後、ジョブスクリプトの終了と共に、マスタ・ワーカ型ジョブも終了します。残っているワーカプロセスはジョブ終了時にジョブ運用ソフトウェアによって終了させられます。このため、ユーザーは明示的に終了させる必要はありません。

2.2 Agentプロセスによるワーカプロセス生成

Agentプロセスからワーカプロセスを生成する方式(以降、Agentプロセス方式)は、ワーカプロセスが非MPIプログラムの場合に利用します。

この方式では、ユーザーはジョブに対し、以下の機能を実装する必要があります。

a. ジョブスクリプト
    1. マスタプロセスとなるマスタプログラムの実行
    2. Agent プロセスの生成
    3. マスタプロセスの終了待ち合わせ
b. マスタプログラム (マスタプロセス)
    1. Agent プロセスの生存確認
    2. Agent プロセスへのワーカプロセス生成依頼
c. Agent プログラム (Agent プロセス)
    1. マスタプロセスとの通信確立
    2. ワーカプロセスの生成
    3. ワーカプロセスの処理結果のマスタプロセスへの送信

以降では、Agent プロセス方式の処理詳細を説明します。

参照

説明に登場するプログラムの構成やコーディング例は "A.2 シェルスクリプト utility.sh" および "A.3 Agent プロセス方式の例" を参照してください。

2.2.1 ノードの割り当て

Agentプロセス方式では、各ノードでAgentプロセスが1つだけ生成されるようにするため、ジョブに割り当てるノード数とmpiexecコマンドで生成するプロセス数を同じにしてください。
以下の例では、ジョブスクリプト内の指示行(#PJMで始まる行)で、ジョブに割り当てるノード数(-L "node=96")とmpiexecコマンドで生成するプロセス数(--mpi "proc=96")を指定しています。

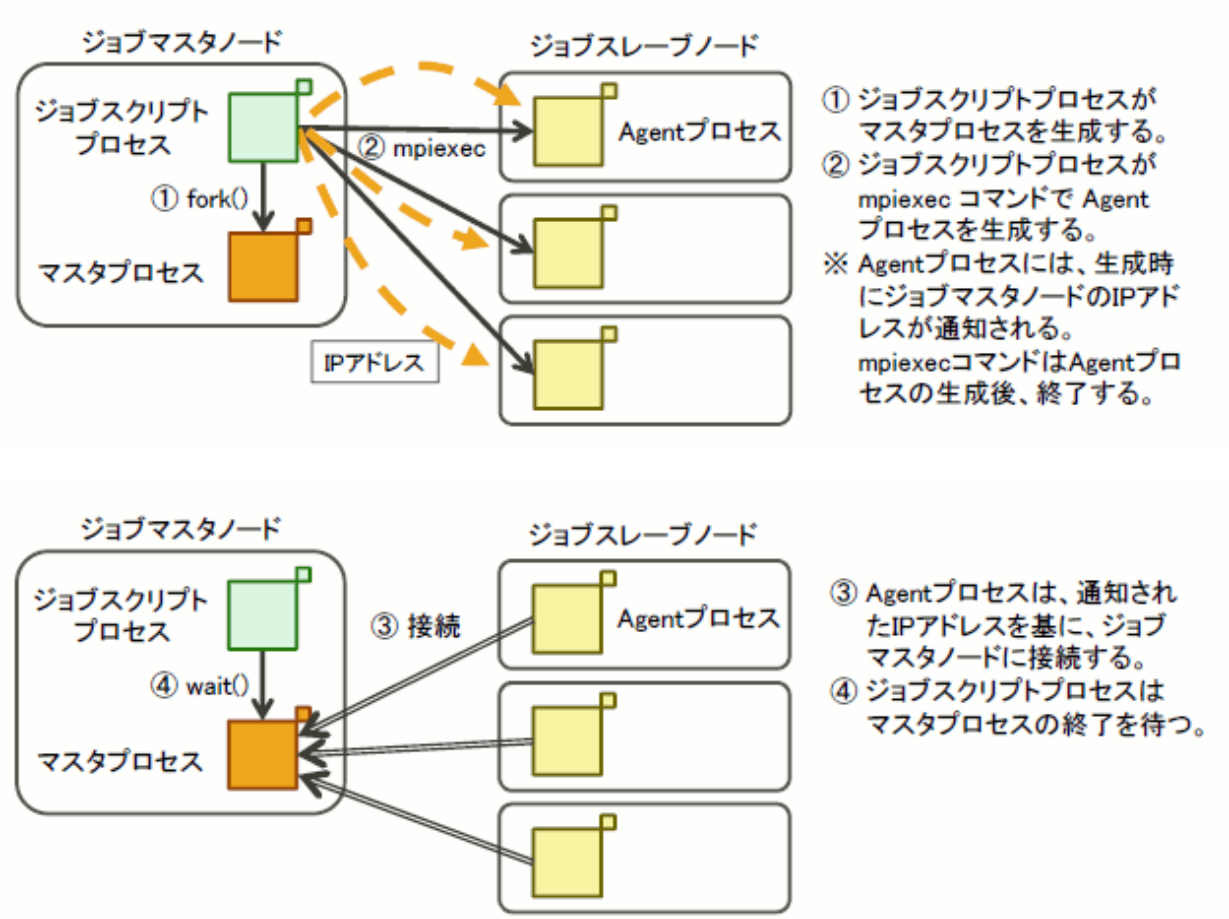
```
[ジョブスクリプトの例: job_agent.sh]
#!/bin/bash
#PJM -L "node=96"
#PJM --mpi "proc=96"
```

```
...
mpiexec start_agent.sh 引数 ...
...
```

2.2.2 マスタプロセスとAgentプロセスの生成

Agent プロセス方式での、マスタプロセスとAgentプロセスの生成の流れは以下のようになります。

図2.3 マスタプロセスとAgentプロセスの生成



以下では、実装例について説明します。

1. マスタプロセスの生成 [ジョブスクリプト]

   マスタプロセスは、ジョブスクリプトの子プロセスとして生成します。
   このとき、マスタプロセスの終了を待ち合わせるために、マスタプロセスのプロセスIDを記憶してください。また、Agentプロセスがマスタプロセスに接続するために必要な情報 (ポート番号) を取得してください。以下の例では、プログラム master_agent.out がファイル port_num.txt にポート名を出力します。

```
[ジョブスクリプトの例: job_agent.sh]
...
. ./utility.sh

./master_agent.out port_num.txt &     # マスタプロセスを生成する。
MASTER_PID=$!                         # マスタプロセスのプロセスIDをシェル変数 MASTER_PID に設定する。
PORT_NUM=$(cat port_num.txt)          # マスタプロセスに接続するためのポート番号をシェル変数 PORT_NUM に設定する。
...
```

上記のジョブスクリプト内では、シェルスクリプト utility.sh を読み込んでいます。
これは、次項で使用するシェル関数 print_ipaddr や timeout を定義しているシェルスクリプトです。utility.sh については "A.2 シェルスクリプト utility.sh" を参照してください。

マスタプログラム master_agent.out は、以下の例のように、Agent プロセスからの接続要求を待ち、接続が確立した Agent に対し、ワーカプロセスの実行を依頼します。

```
[マスタプロセスのプログラム例: master_agent.c]

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

    int sockfd = socket(...);          // ソケットを作成する。
    bind(sockfd, ...);                 // ソケットを特定のポートにバインドする。
    listen(sockfd, ...);               // ワーカプロセスからの接続を待つ。

    FILE *fp = fopen(port_file, "w");
    fprintf(fp, "%d", port_number);    // ポート番号をファイルに書き出す。
    fclose(fp);

    while (1) {
        accept(sockfd, ...);           // ワーカプロセスからの接続を受け付ける。
        ...                            // ワーカプロセスに対し、処理を要求する。
    }
}
```

2. Agent プロセスの生成 [ジョブスクリプト]

Agentプロセスを mpiexec コマンドを使用して生成します。
このとき、マスタプロセスが動作するノード (ジョブマスタノード) の IP アドレス、およびマスタプロセスとAgentプロセス間の通信に使うポート番号を Agent プロセスに渡してください。

以下の例は、IP アドレスを取得するシェル関数 print_ipaddr を定義し、その結果を start_agent.sh の引数として指定しています。

```
[ジョブスクリプトの例: job_agent.sh]
...
# ジョブマスタノード(このノード)のIPアドレスを取得する。
IP_ADDR=$(print_ipaddr "tofu1")

# mpiexec コマンド実行のための環境変数を設定する。
MPI_HOME=/opt/FJSVxxxx/<version>
export PATH=.:${MPI_HOME}/bin:${PATH}
export LD_LIBRARY_PATH=${MPI_HOME}/lib64:/lib64:${LD_LIBRARY_PATH}

# mpiexec コマンドで Agent プロセスを生成する。
mpiexec start_agent.sh "${IP_ADDR}" "${PORT_NUM}" &
...
```

参考

- MPI_HOMEには、Development Studioのmpiexecコマンドがインストールされているディレクトリの基準パスを指定します。このパスについては、Development Studioのマニュアル「MPI使用手引書」を参照するか、管理者にお問い合わせください。
- シェル関数 print_ipaddr は、引数に指定されたネットワークインターフェースの IP アドレスを返します。この例では、ジョブマスタノードの IP アドレスをエージェントプロセスに渡すために使っています。
  引数に指定している "tofu1" は、FXサーバにおける Tofu インターコネクトのネットワークインターフェース名です。FXサーバでワーカプロセスを実行する場合はこの名前を使用してください。

start_agent.sh はシェルスクリプトで、Agent プロセスとなるプログラム agent.out を実行します。

```
[start_agent.sh の例]
#!/bin/bash

./agent.out $@
```

3. マスタプロセスへの接続 [Agent プロセス]

Agentプロセスは、引数に指定されたジョブマスタノードの IP アドレスから、ソケット接続などでジョブマスタノードのマスタプロセスに接続します。

ここで確立した接続は、マスタプロセス・ワーカプロセス間の通信や、マスタプロセスがワーカプロセスの生存を確認するために使用します。

前述の手順1、2 のように Agent プロセスを生成すると、ジョブマスタノードを含むすべての割り当てノードでAgent プロセスが起動されます。ジョブマスタノードで Agent プロセスを実行したくない場合は、以下の例のように、ジョブマスタノードでは Agent プロセスを終了させてください。

```
[Agent プロセスのプログラム例: agent.c]

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

int main(int argc, char **argv)
{
    // コマンドライン引数で与えられたジョブマスタノードのIPアドレスとポート番号を代入する。
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);
    char *my_ip_addr;

    get_ipaddr("tofu1", &my_ip_addr);
    // 自ノードがジョブマスタノードの場合（ジョブマスタノードと同一IPアドレスを持つ場合)、
    // Agent プロセスを終了させる。
    if (strcmp(my_ip_addr, ip_addr) == 0) {
      exit(0);
    }

    // ジョブマスタノードに接続する。(ソケット接続などを使用する)
    connect_to(ip_addr, port_num);

    // ジョブマスタノードからの要求に応じて処理する。
    ...
}
```

参考

上記例における get_ipaddr() 関数は、自ノードのIPアドレスを返す関数です。

4. マスタプロセスの終了待ち合わせ [ジョブスクリプト]

ジョブスクリプトは、マスタプロセス起動時に取得したマスタプロセスのプロセスIDを使用して、マスタプロセスの終了を待ち合わせてから終了するように作成します。

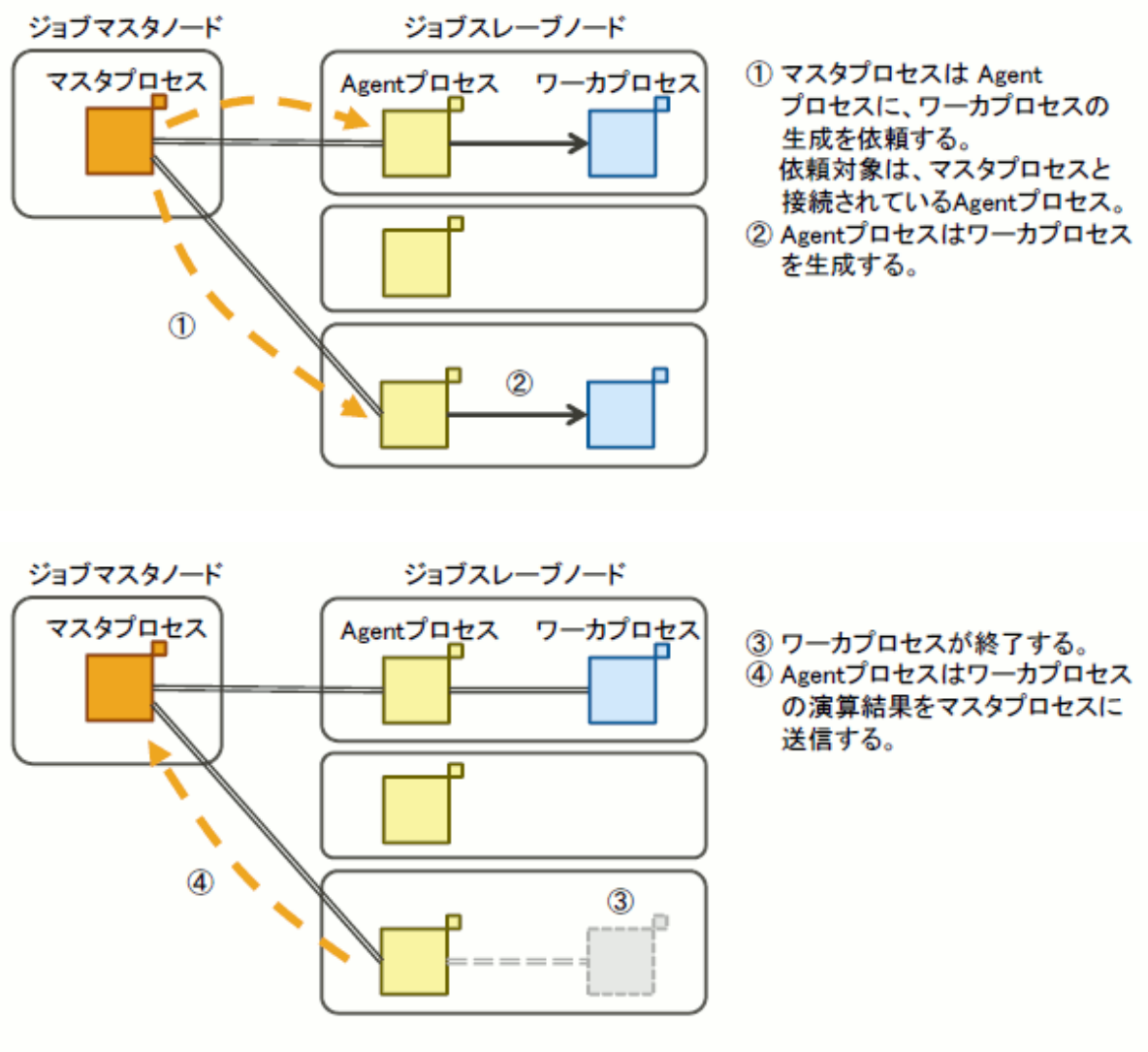
```
[ジョブスクリプトの例: job_agent.sh]
...
wait ${MASTER_PID}       # マスタプロセスの終了を待ち合わせる。
...
# ジョブスクリプト終了（マスタ・ワーカ型ジョブ終了）
```

2.2.3 ワーカプロセスの生成

前節のようにして生成された Agent プロセスは、マスタプロセスからの依頼に応じて、ワーカプロセスを生成します。Agent プロセスは、ワーカプロセスの演算結果をマスタプロセスに送信します。

ワーカプロセスの生成方法やプロセス間の通信方法は、プログラムの実装方法に依存しますので、本書では説明を省略します。

図2.4 ワーカプロセスの生成



2.2.4 ジョブの終了

ジョブスクリプトの終了によって、マスタ・ワーカ型ジョブは終了します。このため、ジョブスクリプトはマスタプロセスの終了を待ち合わせてから終了させます ("2.2.2 マスタプロセスとAgentプロセスの生成" の手順4)。

なお、Agent プロセス および Agent プロセスから起動されたワーカプロセスは、ジョブ終了時に、ジョブ運用ソフトウェアによって終了させられます。このため、これらのプロセスをユーザーが明示的に終了させる必要はありません。

2.3 pjaexe コマンドによるワーカプロセス生成

ワーカプロセスの生成は、Agent プロセス方式 ("2.2 Agentプロセスによるワーカプロセス生成") 以外に、ジョブ運用ソフトウェアが提供する pjaexe コマンドを利用する方式もあります。この方式は、非MPIプログラムの場合に利用します。

この方式では、ユーザーは以下の機能を実装する必要があります。

a. ジョブスクリプト

   pjaexe コマンドによるワーカプロセスの起動

b. マスタプログラム (マスタプロセス)

   1. ワーカプロセスの生存確認 (ワーカプロセスからのコネクションの有無で判断)

2. ワーカプロセスへのタスク実行依頼

c. ワーカプログラム (ワーカプロセス)

1. マスタプロセスとの接続確立
2. マスタプロセスへの計算結果の送信

以降では、処理詳細を説明します。

参照

説明に登場するプログラムの構成やコーディング例は "A.2 シェルスクリプト utility.sh" および "A.4 pjaexe コマンドを使用する例" を参照してください。

2.3.1 ノードの割り当て

pjaexe コマンドによるワーカプロセス生成では、ノード数の指定方法に特別な考慮は不要です。ただし、プログラムによっては、ノード形状については以下のような考慮が必要な場合があります。

- 通信は、マスタプロセス、ワーカプロセス間だけの場合

  この場合、ノードの形状を意識する必要性は低く、ノード数が適切であればよいです。

- ワーカプロセス間での通信がある場合

  通信処理によっては、通信距離が短くなるようなノード形状を考慮してください。

以下は、ジョブに対し、96 ノードを12x8の形状で割り当てる例です。

```
[ジョブスクリプトの例: job_pjaexe.sh]

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

2.3.2 マスタプロセスの生成

マスタプロセスはジョブスクリプトが生成します。

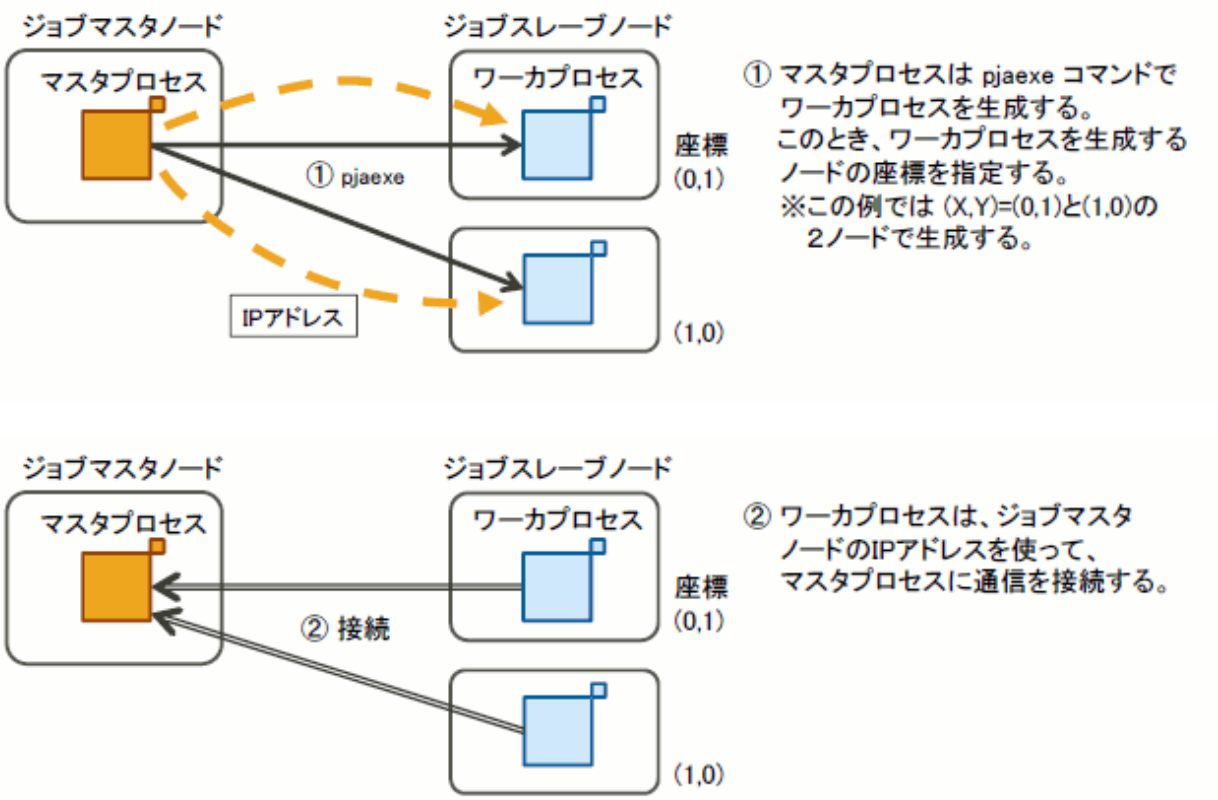
以下の例では、マスタプログラム master_pjaexe.sh はシェルスクリプトで、ワーカプロセスを生成します。

```
[ジョブスクリプトの例: job_pjaexe.sh]

./master_pjaexe.sh
```

2.3.3 ワーカプロセスの生成

マスタプロセスは、pjaexe コマンドを使用して以下のようにワーカプロセスを生成します。

図2.5 pjaexe コマンドによるワーカプロセス生成



1. pjaexe コマンドによるワーカプロセスの生成 [マスタプロセス]

   pjaexe コマンドを使用する場合、ワーカプロセスをどのノードで生成するかはマスタプロセスが管理する必要があります。
   ワーカプロセスを生成するノードの座標を pjaexe コマンドの --vcoord または --vcoordfile オプションで指定します。座標とは、ジョブに割り当てたノード形状内での座標です。pjaexe コマンドについてはマニュアル「ジョブ運用ソフトウェア コマンドリファレンス」を参照してください。
   マスタプロセスは座標の原点に位置するノードに生成されます。このため、ワーカプロセスはそれ以外の座標のノードを選択してください。
   また、ワーカプロセスがマスタプロセスと通信をするために、マスタプロセスが動作しているジョブマスタノードの IP アドレスを渡すインターフェースが必要です(例:プログラムの引数、環境変数など)。
   pjaexe コマンドはワーカプロセスを生成すると即座に終了し、ワーカプロセスは実行を継続します。

注意

- 1 ジョブで同時に実行可能な pjaexe コマンドの個数は 128 個までです。128 個を超えて pjaexe コマンドを実行しようとした場合は、次のメッセージを出力して pjaexe コマンドが異常終了します。

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

- pjaexe コマンドは --vcoordfile オプションで *vcoordfile* を指定することで 1 回の実行で複数のノードにプロセスを同時生成できるので、pjaexe コマンドの実行回数が削減できます。

  [*vcoordfile* の記述例]

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

  [ジョブスクリプトの記述例]

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

ー pjaexeコマンドが実行した*command*の標準入力は/dev/zeroに設定され、標準出力および標準エラー出力は/dev/nullに設定されます。
以下は、ジョブスクリプトからpjaexeコマンドを実行する場合に、a.outの標準出力と標準エラー出力をファイルに出力させるための記述例です。

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

ー pjaexeコマンドで指定したノードがダウンしている場合、pjaexeコマンドは復帰しません。このため、ユーザーは一定の時間内にpjaexeコマンドが復帰しない場合に、タイムアウトさせる仕組みを作る必要があります。
以降の例では、シェル関数timeoutを定義し、利用しています。

参考

ノードの座標の指定は、ノード形状と同様に、ワーカプロセス間の通信距離に関係します。
ワーカプロセス間で通信をする場合、ノード間の距離が短くなるように座標を指定すると、効率的に通信ができます。

2. マスタプロセスへの接続 [ワーカプロセス]

   ワーカプロセスは、マスタプロセスから渡されたIPアドレスに基づいて、マスタプロセスとの通信路を確立します。
   具体的な手順は、プログラムの通信方法に依存しますので、ここでは説明を省略します。

以下に、マスタプロセスとなるプログラム(マスタプログラム)をシェルスクリプトmaster_pjaexe.shで作成する例を示します。

```
[マスタプログラムの例: master_pjaexe.sh]
...
. utility.sh

# ジョブマスタノード(このノード)のIPアドレスを取得する。
IP_ADDR=$(print_ipaddr "tofu1")

# ワーカプロセスからの接続を受け付けられるようにソケットを初期化する。
PORT_NUM=port番号
...

# すべてのジョブスレーブノードでワーカプロセス worker.out を起動する。
for X in $(seq 0 11); do
    for Y in $(seq 0 7); do
        VCOORD="(${X},${Y})"
        # 60 秒以内に pjaexe コマンドが復帰しない場合は、タイムアウトとする。
        timeout 60 pjaexe --vcoord ¥"${VCOORD}¥" ./worker.out "${IP_ADDR}" "${PORT_NUM}"
        RC=$?
        if [ "${RC}" -eq 1 ]; then
            # ユーザーの指定ミスの場合、マスタプロセスを異常終了させる。
            exit 1
        fi
        if [ "${RC}" -ne 0 ]; then
            # pjaexe コマンドが異常終了した場合、ノード故障したと判断し、故障ノードリストに追加する。
            echo "${VCOORD}" >> broken_node_list.txt
        fi
    done
done
...
```

参考

- 通常、マスタプロセスとなるプログラムはC言語やFortran言語などのプログラミング言語で記述しますが、ここでは処理論理を説明するために、マスタプログラムをシェルスクリプトで実装した例を示しています。

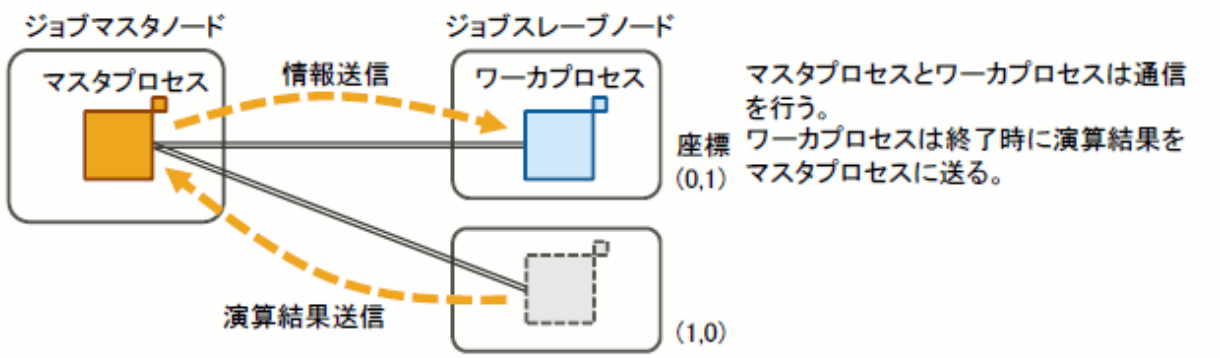
シェルスクリプトでは実装が難しい処理(ソケットの初期化処理やワーカプロセスとの通信処理)は省略しています。これらの処理については一般的なプロセス間通信の手段を参考にしてください。

- 上記例では、ワーカプロセスを生成するノードの座標は、X座標が 0 から 11、Y座標が 0 から 7 の範囲で指定しています。
- 最初にスクリプト utility.sh を読み込んでいるのは、IP アドレスを取得するユーティリティ関数 print_ipaddr や timeout を定義するためです。
- シェル関数 timeout は、指定したコマンドをタイムアウト付きで実行します。指定した時間(秒)以内にコマンドが復帰しない場合は、そのプロセスに対し、シグナル SIGKILL を送信します。
  この例では、pjaexe コマンドが 60 秒以内に復帰しなかった場合、pjaexeコマンドのプロセスにシグナル SIGKILL を送信し、強制終了させます。

2.3.4 マスタプロセスとワーカプロセスの通信

マスタプロセスとワーカプロセスの通信は、前節の手順2で確立した通信路を使用します。
具体的な手順は、プログラムの通信方法に依存しますので、ここでは説明を省略します。

図2.6 マスタプロセスとワーカプロセスの通信



2.3.5 ジョブの終了

ジョブスクリプトの終了によって、マスタ・ワーカ型ジョブは終了します。このため、ジョブスクリプトはマスタプロセスの終了を待ち合わせてから終了させます。
なお、ワーカプロセスは、ジョブ終了時にジョブ運用ソフトウェアによって終了させられます。このため、ユーザーは明示的にワーカプロセスを終了させる必要はありません。

2.4 マスタ・ワーカ型ジョブ作成における注意事項

マスタ・ワーカ型ジョブを作成する際は、以下に注意してください。

- マスタ・ワーカ型ジョブでは、ワーカプロセスの生成や異常の検出、およびその対処はジョブの作成者が考える必要があります。
  具体的な実装方法は、ジョブやプログラムの処理内容や処理論理に依存しますので、本章で説明する実装例では説明を省略しているところがあります。
- ワーカプロセスを動的に生成する方式では、ワーカプロセスが実行されているノードがダウンした場合、そのノード上のワーカプロセスと同じ MPI_COMM_WORLD に属するすべてのワーカプロセスは動作ができなくなります。
  これらのワーカプロセスは、ユーザーが終了させるか、またはジョブが終了するまで残ります。
  また、これらのワーカプロセスが動作していたノードは mpiexec コマンドが終了するまでは、ワーカプロセスの再生成先として選択されません。
  ただし、mpiexec コマンドを再実行すると、ダウンしたノードが再びワーカプロセスの生成先として選択される可能性があることに注意してください。
- "2.3 pjaexe コマンドによるワーカプロセス生成" で説明した実装方式では、プロセスを生成するノードはユーザーが指示するため、--mpi rank-map-hostfile オプションは意味を持たず、指定しても無視されます。

- マスタ・ワーカ型ジョブはバッチジョブとしてのみ実行でき、会話型ジョブとしては実行できません。
  また、マスタ・ワーカ型ジョブは、ステップジョブおよびバルクジョブとは排他のジョブモデルです。

- マスタ・ワーカ型ジョブ内で、MPI 通信関数を利用する場合は、以下に注意してください。

  MPI規格によると、MPI 通信処理に失敗した場合、デフォルトでは、通信関数の呼び出し元プロセスも異常終了します。例えば、処理中に通信先プロセスが異常終了した場合や通信先ノードがダウンした場合です。

  この通信処理は、ユーザーが明示的に MPI_Send()、MPI_Recv()、MPI_Bcast() などの MPI 通信関数を呼び出した場合だけではなく、MPIライブラリの内部処理で実行される場合もあります。このような場合、プログラムにとっては、マスタプロセスが通信とは関係ない処理を実行している最中に、突然、異常終了してしまうように見えます。

  マスタ・ワーカ型ジョブで MPI 通信関数を使用する場合は、ワーカプロセスの異常終了に伴ってマスタプロセスが異常終了しないように、以下に示す対処をしてください。これにより、マスタプロセスが異常終了する確率が低くなります。

  1. MPI_Comm_spawn() 関数の呼び出し後、FJMPI_Mswk_disconnect() 関数を呼び出します。

     MPI_Comm_spawn() 関数によって動的にプロセスを生成した場合、生成されたプロセスと MPI_Comm_spawn() 関数の呼び出し元プロセスの間は、通信が接続した状態になります。
     MPIの内部通信処理は、この通信が接続状態で発生し、切断された状態だと発生しません。このため、"2.1.4 ワーカプロセスの生成" の例のように、MPI_Comm_spawn() 呼び出し後に、FJMPI_Mswk_disconnect() を呼び出す必要があります。

  2. MPI通信関数の呼び出し前に FJMPI_Mswk_connect() または FJMPI_Mswk_accept() を呼び出し、MPI通信関数呼び出し後に FJMPI_Mswk_disconnect() を呼び出します。

     接続先プロセスが異常終了していた場合、FJMPI_Mswk_connect() 関数、FJMPI_Mswk_accept() 関数、および FJMPI_Mswk_disconnect() 関数は、異常復帰するだけで、呼び出し元プロセスが異常終了することはありません。
     このため、以下の例のように MPI通信関数を呼び出す前に毎回 FJMPI_Mswk_connect() 関数を呼び出す必要があります。これにより、ワーカプロセスの異常がマスタプロセスに及ぼす影響を低減できます。

     [例] マスタプロセスからワーカプロセスに接続する場合

     ```
     // マスタプロセス
     FJMPI_Mswk_connect(worker_port, …, &worker_comm);
     MPI_Send(…, worker_comm, …);
     FJMPI_Mswk_disconnect(&worker_comm);

     // ワーカプロセス
     FJMPI_Mswk_accept(worker_port, …, &master_comm);
     MPI_Recv(…, master_comm, …);
     FJMPI_Mswk_disconnect(&master_comm);
     ```

     上記手順に従わなかった場合、ワーカプロセスの異常終了後、またはワーカプロセスが動作しているノードのダウン後、任意のタイミングでマスタプロセスが異常終了する可能性があります。マスタプロセスが異常終了すると、mpiexec コマンドも異常終了します。ただし、ジョブスクリプトは実行を継続します。

第3章 システム異常時の影響

ここでは、マスタ・ワーカ型ジョブの実行中にノードダウンなどシステムに起因する異常が起こった場合の影響について説明します。

3.1 ジョブの動作への影響

異常の内容によって、マスタ・ワーカ型ジョブは終了する場合と継続する場合があります。

- マスタ・ワーカ型ジョブが終了するケース

  以下の場合は、ほかのジョブモデルと同様に、マスタ・ワーカ型ジョブは終了し、再度キューイングされます。

  - ジョブマスタノードがダウンした場合
  - ストレージI/Oノードがダウンした場合
  - ジョブスレーブノードがダウンした状態で、当該ジョブをサスペンドまたはリジュームしようとした場合
  - マスタ・ワーカ型ジョブに割り当てた計算ノードで ICC または Port 故障が発生した場合
  - 管理者(クラスタ管理者) がマスタ・ワーカ型ジョブに割り当てたノードを運用から切り離す際に、ジョブを即時に終了させるように指定した場合

- マスタ・ワーカ型ジョブが継続するケース

  以下の場合は、マスタ・ワーカ型ジョブは継続します。ただし、これらに起因してワーカプロセスが異常になった場合は、ほかのノードでワーカプロセスを実行するなどの対処を、ユーザープログラム内で考慮する必要があります。

  - ジョブスレーブノードのジョブ運用ソフトウェアのサービス異常
  - ジョブスレーブノードのダウン
  - ジョブスレーブノードのハードウェア (CPU またはメモリ) の異常

参考

- ジョブの終了原因がユーザー側にある場合(例:CPU時間などの資源制限値超過)、ジョブが再キューイングされるかどうかは、ほかのジョブモデルと同様に、ジョブ投入時の指定やジョブACL機能の設定によります。
- ストレージI/Oノードとは第1階層ストレージに対する入出力を担うI/Oノードで、16ノード中に1ノード存在します。詳細はマニュアル「ジョブ運用ソフトウェア 概説書」を参照してください。

3.2 ジョブ統計情報への影響

マスタ・ワーカ型ジョブの実行中にノードダウンが発生した場合、pjsub -s/-S や pjstat -v で出力されるジョブ統計情報は以下のようになります。

項目	値
PC、PJM CODE (ジョブ終了コード)	ジョブマスタノードのダウンや、FXサーバの故障(ICC 異常)によってマスタ・ワーカ型ジョブが異常終了した場合、ジョブ統計情報のジョブ終了コードは通常ジョブの場合と同様になります。 ジョブスレーブノードだけのダウンのように、マスタ・ワーカ型ジョブの継続に影響がない異常の場合は、マスタ・ワーカ型ジョブは最後まで実行されます。正常に終了した場合は、ジョブ終了コードは 0 になります。
REASON (終了原因)	前項のジョブ終了コードと同様ですが、ジョブが正常終了した場合は "-" になります。
name (REQUIRE) (要求資源量)	"NODE NUM (REQUIRE)" などの "(REQUIRE)" が付く項目は、ノードのダウンに関係なく、ジョブ投入時にユーザーが指定した値になります。
name (ALLOC) (割り当て資源量)	"NODE NUM (ALLOC)" などの "(ALLOC)" が付く項目は、ノードのダウンに関係なく、ジョブ投入時に決定した値になります。

項目	値
name (USE) (使用資源量)	"NODE NUM (USE)" などの "(USE)" が付く項目は、故障したノードの分は除外された値になります。

3.3 コマンドの表示への影響

ジョブ運用ソフトウェアが提供する pjshowrsc コマンドは、計算機資源としてのノード数を表示できます。
マスタ・ワーカ型ジョブに割り当てられたノードがジョブ実行中にダウンした場合、ダウンしたノードは利用可能資源から除外されます。

a. 引数なしの場合

TOTAL と ALLOC の値が、ダウンしたノードの数だけ減ります。

```
[ジョブスレーブノードがダウンする前]
$ pjshowrsc
  CLUSTER          NODE
                   TOTAL   FREE  ALLOC
  mswk-comp          194    182     12

[ジョブスレーブノードのうち、1ノードがダウンした後]
$ pjshowrsc
  CLUSTER          NODE
                   TOTAL   FREE  ALLOC
  mswk-comp          193    182     11
```

b. -l オプションを指定した場合

各資源の TOTAL と ALLOC の値が、ダウンしたノードの分だけ減ります。

```
[ジョブスレーブノードがダウンする前]
$ pjshowrsc -l
     RSC    TOTAL    FREE   ALLOC
    node      384     372      12
     cpu     3072    2976      96
     mem    5.3Ti   5.1Ti   180Gi

[ジョブスレーブノードのうち、1ノードがダウンした後]
$ pjshowrsc -l
     RSC    TOTAL    FREE   ALLOC
    node      383     372      11
     cpu     3064    2976      88
     mem    5.3Ti   5.1Ti   170Gi
```

c. -v オプションを指定した場合

ダウンしたノードの情報は出力されなくなります。

```
[ジョブスレーブノードがダウンする前]
$ 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

[ジョブスレーブノードのうち、1ノード(0x01010011) がダウンした後]
$ 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
  ※ダウンしたノード 0x01010011 の情報が表示されなくなる。
```

付録A　プログラム例

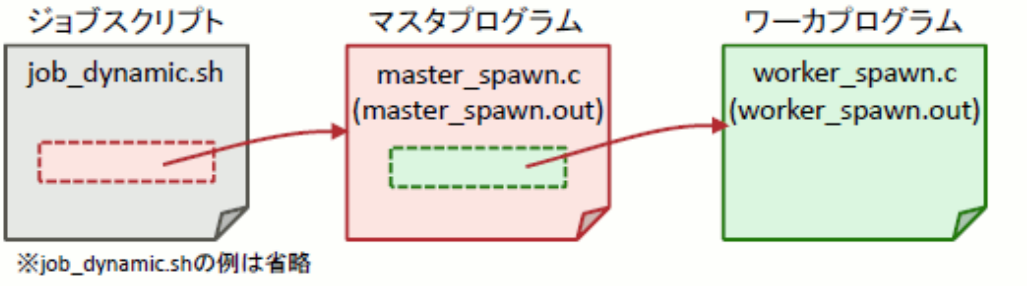
ここでは、マスタ・ワーカ型ジョブのジョブスクリプトやプログラムの例を示します。

A.1　ワーカプロセスを動的生成する例

ここでは、"2.1 ワーカプロセスの動的生成" で説明したプログラムの例を示します。

MPIプログラムの作成に関する詳細は、MPIの仕様やDevelopment Studioのマニュアル「MPI 使用手引書」を参照してください。

図A.1 プログラムの構成



[マスタプログラム 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] = "";

    // 各コミュニケーターのルートランク
    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";

    // 初期化処理
    MPI_Init(&argc, &argv);

    // world_size,  universe_size を取得する。
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    if (world_size != 1) {
        // マスタプロセスが複数個存在する場合
        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) {
        // universe_size の取得に失敗した場合
```

```
        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) {
        // universe_size が 1 の場合
        fprintf(stderr, "Error! universe_size=%d (expected > 1)", universe_size);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // ワーカプロセスとの通信用ポートをオープンする。
    MPI_Open_port(MPI_INFO_NULL, master_port);
    printf("master_port=%s¥n", master_port);

    // ワーカプロセスを生成する。
    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);

    // ワーカプロセスへポート名を送信する。
    MPI_Send(master_port, MPI_MAX_PORT_NAME, MPI_CHAR, worker_root, tag, worker_comm);

    // ワーカプロセスとの接続を切断する。
    FJMPI_Mswk_disconnect(&worker_comm);

    // ワーカプロセスからのデータを受信する。(ワーカプロセスのポート名が送信されてくる)
    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);

    // ワーカプロセスへデータを送信する。
    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);

    // 終了処理
    MPI_Close_port(master_port);
    MPI_Finalize();
}
```

[ワーカプログラム 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] = "";

    // 各コミュニケーターのルートランク
    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] = "";
```

```
    // 初期化処理
    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) {
        // マスタプロセスからのデータを受信する。
        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);
    }

    //  マスタプロセスとの通信を切断する。
    FJMPI_Mswk_disconnect(&master_comm);

    // マスタプロセスへポート名を送信する。
    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);
    }

    // マスタプロセスからのデータを受信する。
    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);
    }

    // 終了処理
    if (rank == worker_root) {
        MPI_Close_port(worker_port);
    }
    MPI_Finalize();
}
```

A.2 シェルスクリプト utility.sh

ここでは、"2.2 Agentプロセスによるワーカプロセス生成" や "2.3 pjaexe コマンドによるワーカプロセス生成" で使用したシェルスクリプト utility.sh を示します。
ジョブスクリプト内で utility.sh を読み込んで使います。

注意

- utility.sh は、ユーザーが作成します。
- utility.sh には実行権が設定され、ジョブスクリプト内から utility.sh にアクセスできる必要があります。

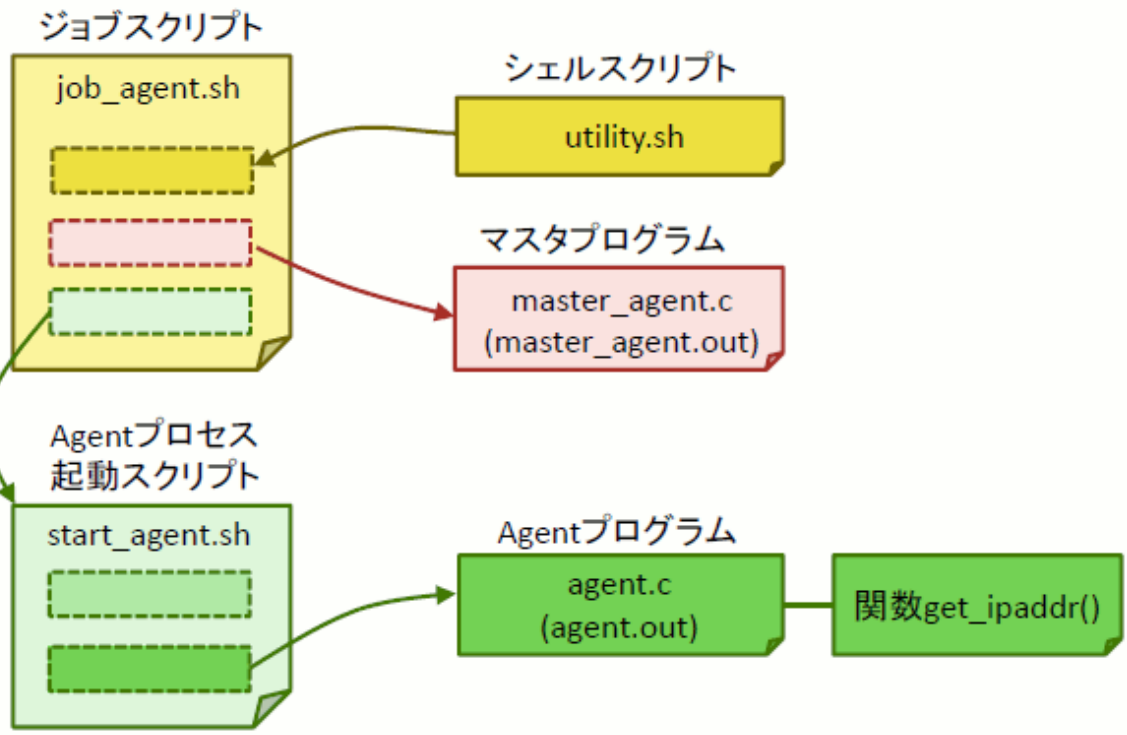
```
# 指定したネットワークインターフェースのIPアドレスを出力する。
# Usage: print_ipaddr <interface>
print_ipaddr() {
    local INTERFACE=$1
    LANG=C ip addr show dev "${INTERFACE}" | sed -n '/.*inet ¥([0-9.]*¥).*/{s//¥1/;p}'
}
```

```
# 指定したコマンドをタイムアウト付きで実行する。
# Usage: timeout <timeout_sec> <command> <arg1> <arg2> ...
timeout() {
    local TIMEOUT=$1
    shift 1

    # コマンド実行およびプロセスID を記録する。
    eval "$@" &
    local PID=$!

    echo ${PID}
    while true; do
        # コマンドプロセスの生存をチェックする。
        if ! ps -p "${PID}" >/dev/null 2>&1; then
            # プロセスが終了したため、ループを抜ける。
            break
        fi

        if [ "${TIMEOUT}" -le 0 ]; then
            # タイムアウトした場合、プロセスを異常終了させる。
            kill -KILL "${PID}"
            break
        fi

        # 1秒間スリープした後、ループの先頭に戻る。
        sleep 1
        TIMEOUT=$((TIMEOUT - 1))
    done

    # プロセスの終了コードを返す。
    wait "${PID}"
    return $?
}
```

A.3 Agent プロセス方式の例

ここでは、"2.2 Agentプロセスによるワーカプロセス生成" で説明したスクリプトやプログラムの例を示します。

図A.2 プログラムの構成



[ジョブスクリプト job_agent.sh]

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

. utility.sh

# マスタプロセスを生成する。
./master_agent.out port_num.txt &
MASTER_PID=$!
PORT_NUM=$(cat port_num.txt)

# ジョブマスタノード(このノード)のIPアドレスを取得する。
IP_ADDR=$(print_ipaddr "tofu1")

# mpiexec コマンド実行のための環境変数を設定する。
MPI_HOME=/opt/FJSVxxxx/<version>
export PATH=.:${MPI_HOME}/bin:${PATH}
export LD_LIBRARY_PATH=${MPI_HOME}/lib64:/lib64:${LD_LIBRARY_PATH}

# mpiexec コマンドで Agent プロセスを生成する。
mpiexec start_agent.sh "${IP_ADDR}" "${PORT_NUM}" &

wait ${MASTER_PID}
```

[マスタプログラム 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(...);          // ソケットを作成する。
    bind(sockfd, ...);                 // ソケットを特定のポートにバインドする。
    listen(sockfd, ...);               // ワーカプロセスからの接続を待つ。

    FILE *fp = fopen(port_file, "w");
    fprintf(fp, "%d", port_number);    // ポート番号をファイルに書き出す。
    fclose(fp);

    while (1) {
        accept(sockfd, ...);           // ワーカプロセスからの接続を受け付ける。
        ...                            // ワーカプロセスに対し、処理を要求する。
    }
}
```

[Agent プロセス起動スクリプト start_agent.sh]

```
#!/bin/bash

./agent.out $@
```

[Agent プログラム agent.c]

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

int main(int argc, char **argv)
{
    // コマンドライン引数で与えられたジョブマスタノードのIPアドレスとポート番号を代入する。
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);
    char *my_ip_addr;

    get_ipaddr("tofu1", &my_ip_addr);
    // 自ノードがジョブマスタノードの場合（ジョブマスタノードと同一IPアドレスを持つ場合)、
    // Agent プロセスを終了させる。
    if (strcmp(my_ip_addr, ip_addr) == 0) {
      exit(0);
    }

    // ジョブマスタノードに接続する。
    int sockfd = socket(...);
    connect(sockfd, ...);

    // ジョブマスタノードからの要求に応じて処理する。
    ...
}
```

[get_ipaddr()関数]

get_ipaddr()関数は、自ノードのIPアドレスを文字列として返す関数です。

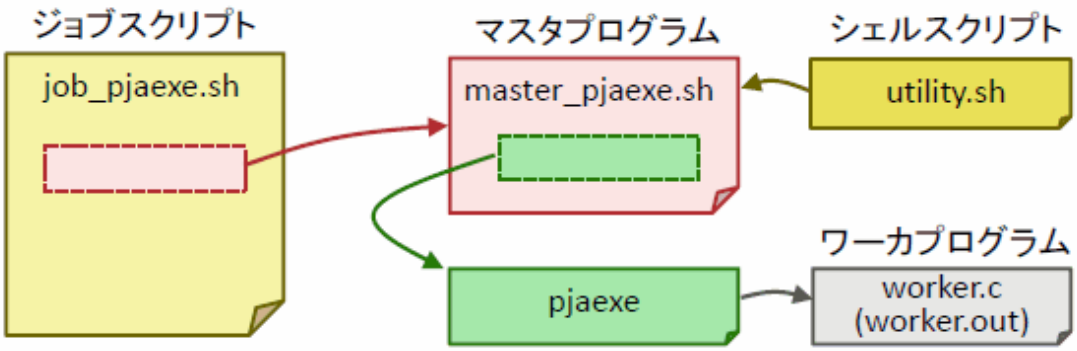
```
#include <string.h>
#include <unistd.h>
```

```
#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 pjaexe コマンドを使用する例

ここでは、"2.3 pjaexe コマンドによるワーカプロセス生成" で説明したスクリプトの例を示します。

図A.3 プログラムの構成



[ジョブスクリプト job_pjaexe.sh]

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

# マスタプロセスを実行
./master_pjaexe.sh
```

[マスタプログラム master_pjaexe.sh]

マスタプログラムは通常、C 言語、Fortran 言語などのプログラミング言語で記述しますが、ここでは処理論理の説明のために、シェルスクリプトで作成する例を示します。

```
#!/bin/bash

. utility.sh
```

```
# ジョブマスタノード(このノード)のIPアドレスを取得する。
IP_ADDR=$(print_ipaddr "tofu1")

# ワーカプロセスからの接続を受け付けられるようにソケットを初期化する。(*)
PORT_NUM=port番号
...

# すべてのジョブスレーブノードでワーカプロセスを起動する。
for X in $(seq 0 11); do
    for Y in $(seq 0 7); do
        VCOORD="(${X},${Y})"
        # 60 秒以内に pjaexe コマンドが復帰しない場合は、タイムアウトとする。
        timeout 60 pjaexe --vcoord ¥"${VCOORD}¥" ./worker.out "${IP_ADDR}" "${PORT_NUM}"
        RC=$?
        if [ "${RC}" -eq 1 ]; then
            # ユーザーの指定ミスの場合、マスタプロセスを異常終了させる。
            exit 1
        fi
        if [ "${RC}" -ne 0 ]; then
            # pjaexe コマンドが異常終了した場合、ノード故障したと判断し、故障ノードリストに追加する。
            echo "${VCOORD}" >> broken_node_list.txt
        fi
    done
done

# ワーカプロセス worker.out との通信や計算結果の取りまとめ処理
...〈省略〉...

# マスタプロセスの終了
exit
```

(*) 通常、マスタプロセスとなるプログラムはC言語やFortran言語などのプログラミング言語で記述しますが、ここでは処理論理を説明するために、マスタプログラムをシェルスクリプトで実装した例を示しています。
シェルスクリプトでは実装が難しい処理(ソケットの初期化処理やワーカプロセスとの通信処理)は省略しています。これらの処理については一般的なプロセス間通信の手段を参考にしてください。

[ワーカプログラム worker.c]

環境変数 PLE_VPID でランク番号を取得する例を示します。

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

int main(int argc, char **argv)
{
    // コマンドライン引数で与えられたジョブマスタノードのIPアドレスとポート番号を代入する。
    char *ip_addr = argv[1];
    int port_num = atoi(argv[2]);

    // ランク番号を取得する。
    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);

    // ジョブマスタノードからの要求に応じて処理する。
```

```
    . . .
}
```