UNIXとプログラミングの基礎

大島 聡史 (東京大学 情報基盤センター 助教)
ohshima@cc.u-tokyo.ac.jp

東京大学
THE UNIVERSITY OF TOKYO

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

この講習の目的

- Oakleaf-FX/Reedbushを効率的に利用するための技術を習得する

- 内容
  1. シェルスクリプト
  2. ファイルシステム
  3. MPI-IO
  4. 少し高度なジョブの操作
  5. makeの活用

演習で使用するプログラムなどのありか

- Oakleaf-FX
  - /home/z30097/public2
- Reedbush-U
  - /lustre/gt00/z30097/public2

シェルスクリプトについて

少し便利なシェルの操作
簡単なテキスト処理

シェル・シェルスクリプト

- UNIX（Un*x、Linux）のコマンドラインユーザーインターフェイスと、それを扱うスクリプト
- Oakleaf-FX/Reedbushを含む多くのLinux環境ではbash（バッシュ）がデフォルトで使われている
- コマンドライン操作やバッチジョブスクリプト中でもシェルの様々な機能を使うことができる
  - 変数の利用、繰り返し処理、etc.
  - バッチジョブスクリプトもシェルスクリプトの一種

- できることが非常に多いので、ここでは最低限の使い方や例などを紹介する
  - 情報はWebにいくらでもあります

シェルスクリプトの構成と基本的な使い方

- 一行目に使うシェルを指定
  - シェルの種類によって記述方法なども異なる
  - ここではbashの例を挙げる
- 上から下へ順に実行
  - 関数（手続き）は呼ばれたら実行
- 空白文字（スペース）に注意
  - 無駄に入れてはいけない
- 変数が利用可能
  - 計算も可能

```
FOO=1
echo ${FOO}
BAR=`expr ${FOO} ¥* 2`
echo ${BAR}
```

```
#!/bin/bash
hoge (){
  echo $1
}
foo
bar
hoge 2
```

- 繰り返し処理が可能

```
for i in 1 2 3 4
do
  echo ${i}
done
```

簡単なテキスト処理

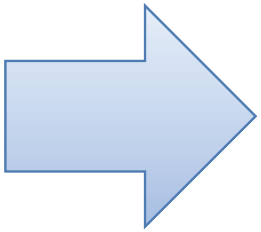
- 多くのLinux環境にはテキスト処理に便利な様々なプログラムが用意されている
  - シェル（シェルスクリプト）からも利用できる
  - 出力結果の加工など色々なことに利用できる

  - grep : 指定文字列を含む行を抽出する、など
    - 例：`grep time ./result.txt`
  - awk : 特定の区切り文字で分割し、部分文字列を抽出する、など
    - 例：`awk -F, '{print $3}' ./result.txt`
  - sed : 文字列の置換、など
    - 例：`sed -e "s/time/msec/" ./result.txt`
  - perl, python, ruby : スクリプト言語、その気になればなんでもできる
  - etc.

利用例１

- バッチジョブスクリプト内でパラメタ（実行時引数）の異なるプログラムを順番に実行する

```
for i in 1 2 3
do
  ./a.out ${i}
done
```



```
./a.out 1
./a.out 2
./a.out 3
と同様
```

- コマンド実行時に繰り返し処理を行う

```
$ for i in 1 2 3
>do
>cp template.dat input${i}.dat
>sed -i -e  “s/NUM/${i}/”  input${i}.dat
>done
```

template.datを
input1.dat input2.dat input3.dat
に複製、各ファイル内のNUMを
1 2 3の数字に置換

- sedの-iオプションはファイルの上書き
- 改行の代わりに;で区切れば一行で入力も可能

template.datの例

```
echo NUM
```

利用例２

- awkで「,」区切りの出力ファイルから2項目だけを抽出

```
awk -F ,  ‘{print $2}’  result.txt
```

- さらに足し算して合計値や平均値を求める

```
awk -F , 'BEGIN{SUM=0}{print $2;SUM+=$2}END{print SUM}' result.txt
awk -F , 'BEGIN{SUM=0}{print $2;SUM+=$2}END{print SUM/NR}' result.txt
```

  - 四則演算が使える、最初と最後の処理を設定できる、NRは行数
    - 変数の初期値は０なので実はBEGIN処理は不要
    - Excelを使うより便利な（手っ取り早い）ことも
    - grepしたファイルをawkで処理することが多い

result.txtの例
```
1,2,3
2,3,4
3,4,5
```

- 演習
  - 各コマンドを実際に実行してみよう
  - awkを使って最大値・最小値を求めてみよう
  - 各処理の実行前には「echoでコマンド列を出力するシェルスクリプト」を作成して動作を確認することを推奨

関係ファイル：scriptディレクトリ内

シェルスクリプト：まとめ

- シェルとシェルスクリプトについて少しだけ紹介した
- Oakleaf-FX/Reedbushのみならず Unix系環境（Linux、Cygwin、Mac OSなど）の色々な場面で利用可能
- 「この処理は面倒だな」と思ったらシェルや各種コマンドの活用を考えてみると良い
  - 考えすぎると一度だけの処理のために無駄に凝ったスクリプトを作成することになりかねないため注意

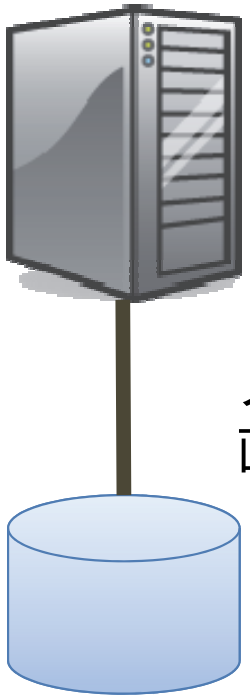
ファイルシステムについて

ファイルシステムの大まかな分類

- ローカルディスク
  - 他のノードから直接接続できない記憶域
    - Oakleaf-FXとReedbushは計算ノード・インタラクティブノードにローカルディスクを持たない
- NFS (Network File System)
  - ネットワーク経由で複数クライアントからアクセス可能なファイルシステム
  - 単一サーバ
- 分散ファイルシステム
  - ネットワーク経由で複数クライアントからアクセス可能なファイルシステム
  - 複数のファイルサーバにデータ・メタデータを分散配置
- （Ramdisk）
  - メモリ上に確保したファイルシステム

ローカルディスク

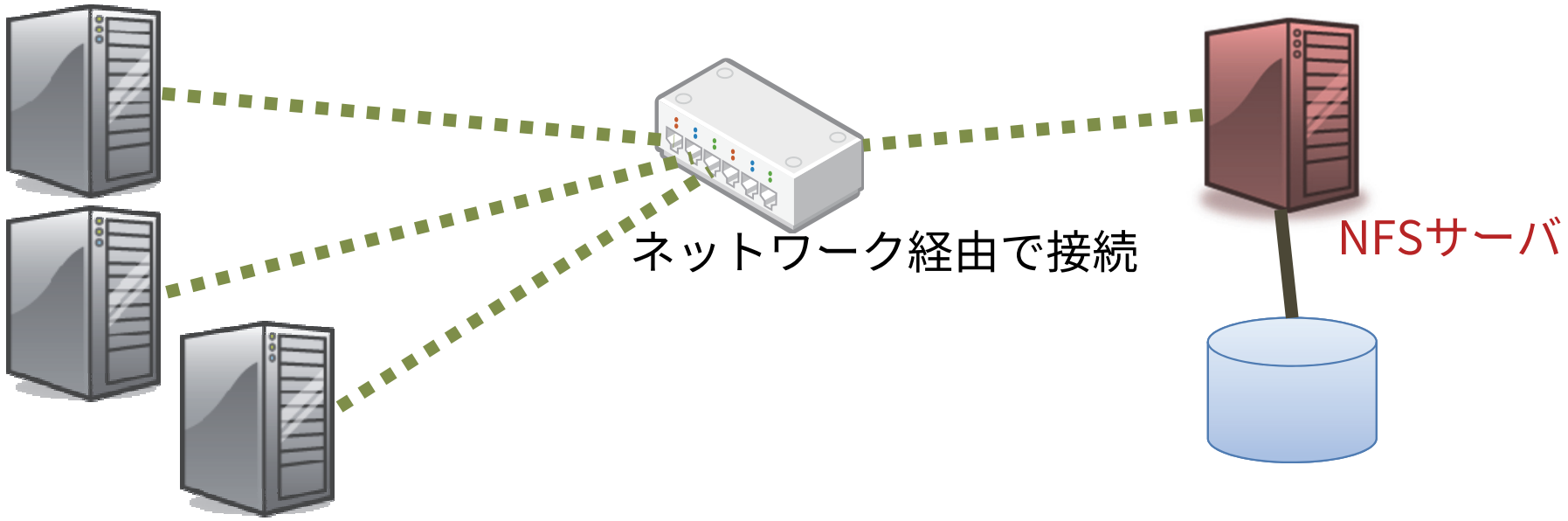
- 他のノードから直接アクセスできない記憶域
  - Oakleaf-FX、Reedbushは計算ノード・インタラクティブノードにローカルディスクを持たない



ノードとディスクが
直接接続

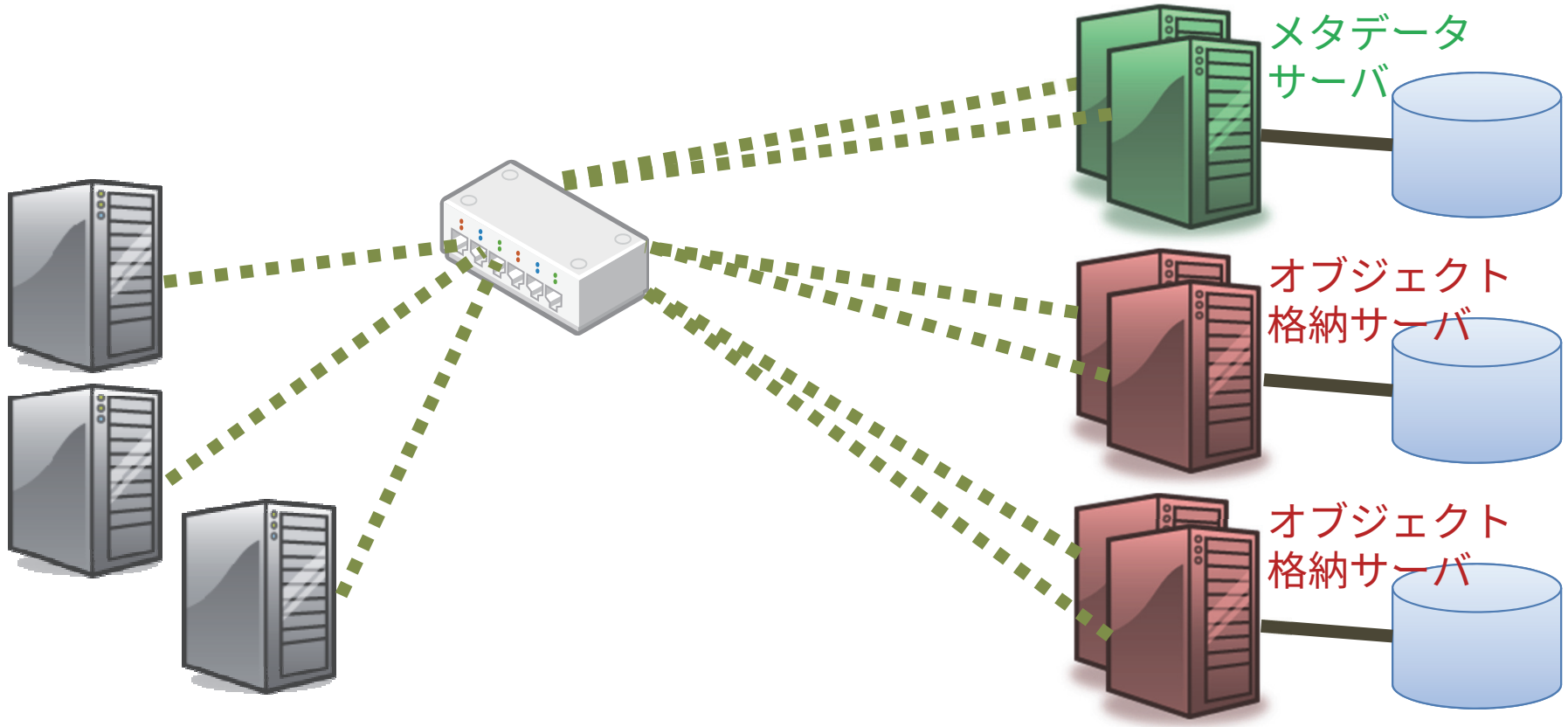
NFS (Network File System)

- ネットワーク経由で複数クライアントからアクセス可能
- 普通のLinux系OS等で安価かつ容易に構築可能
- サーバは1台のみ、動的な負荷分散機能がない
- Oakleaf-FXにおける設定の例
  - OS起動等のために、1ラック(96ノード)ごとに1台使用
  - /tmp として割り当てられているが、使用は非推奨



分散ファイルシステム

- 複数のファイルサーバにデータおよびメタデータを分散配置
  - 1ファイルのデータを複数台のサーバに分散可能
  - フェイルオーバーにより、サーバ故障に対応可能



分散ファイルシステムの特徴

- 複数のファイルサーバにデータを分散可能
  - 多くのクライアントからアクセスする場合に効率がよい

- 構成がNFSより複雑
  - NFSに比べると1クライアントからのアクセス性能は低い場合がある
  - ただし、1ファイルのデータを複数のサーバに分散させれば、1クライアントからのアクセス性能を上げることができる

Oakleaf-FXで利用可能なファイルシステム

PATH	種類	備考
/home/ログイン名	分散 (FEFS)	共有ファイルシステム
/group[1-3]/グループ名/ログイン名　※		
/mppx[bc]/ログイン名　※		外部ファイルシステム （データ保存期間は9ヶ月間）
/work		ローカルファイルシステム （ステージング用、ログインノードからは直接使えない）
/tmp	NFS	使用を推奨しない
/dev/shm	Ramdisk	メモリ上に確保された領域

※　負荷分散のため、グループ、ユーザ毎に/group[1-3]および/mppx[bc]のいずれかを使用

Reedbushで利用可能なファイルシステム

PATH	種類	備考
/home/グループ名/ログイン名	NFS	ログインノードからのみ利用可能 容量が小さい ログインに必要なもの・各種設定ファイルなど、最低限のものだけ置くこと
/lustre/グループ名/ログイン名	分散 (Lustre)	ログインノードからも計算ノードからも利用可能 一般的な用途に使える
/tmp	Ramdisk	メモリ上に確保された領域 容量が小さい 使用を推奨しない
/dev/shm		

バーストバッファについては現在準備中

Oakleaf-FX/Reedbushの分散ファイルシステム

- Lustre
  - 大規模ファイル入出力、メタデータ操作の両方で高性能なファイルシステム
  - データの分散方法をファイルごとに指定可能(後述)

- FEFS (Fujitsu Exabyte File System)
  - Lustre ファイルシステムをベースに富士通が開発
    - Lustre との高い互換性（同じ機能が利用できる）
  - 数万規模のクライアントによるファイル利用を想定
    - 最大ファイルサイズ、最大ファイル数等を拡張

利用可能な容量の制限

- 共有ファイルシステムは、個人、またはグループに対して利用可能容量の制限(quota)がある
- show_quotaコマンドで構成を確認可能
  - Oakleaf-FXでの実行例

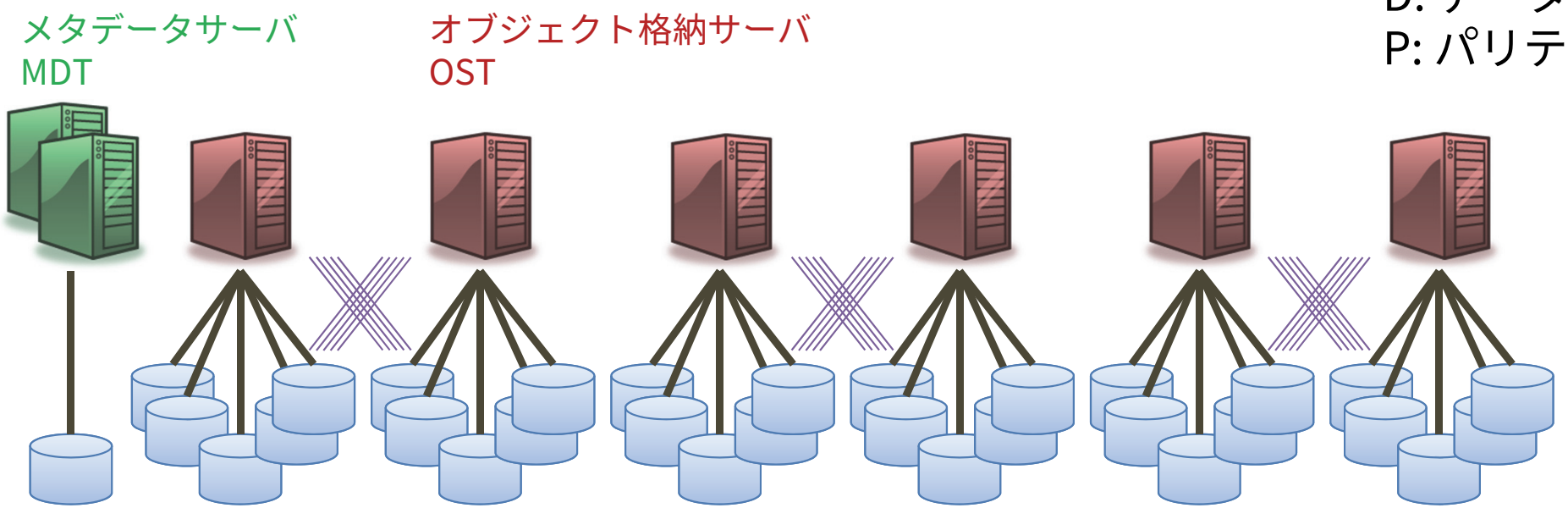
```
$ show_quota
Disk quotas for user c26002
    Directory                used(MB)    limit(MB)       nfiles
    /home/c26002              102,868      204,800       75,296
    /group/gc26/c26002         39,453            -          416
    /group/gv31/c26002              0            -            0
    /group/gv32/c26002              0            -            0
    /group/gv35/c26002              0            -            0
    /mppxb/c26002                   0            -            4

Disk quotas for group gc26 gv31 gv32 gv35
    Directory                used(MB)    limit(MB)       nfiles
    /group/gc26            39,697,121   40,960,000    8,160,598
    /group/gv31                     0    4,096,000            5
    /group/gv32                     0    8,192,000           13
    /group/gv35                     0    4,096,000            9
```

Lustre/FEFSの物理構成とデータ配置

- メタデータを格納するMDT（Metadata Target）と
  データを格納するOST（Object Storage Target）から構成
  - MDTにはOST上のデータ配置についての情報が格納されている
- Oakleaf-FXの例
  - MDT：RAID1
  - OST：共有: RAID6 (9D+2P) x 480、ローカル: RAID5 (4D+1P) x 600、外部: RAID6 (8D+2P) x 236

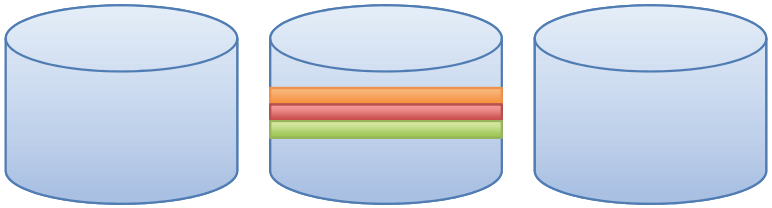
D: データ
P: パリティ



Lustreのデータ配置の指定

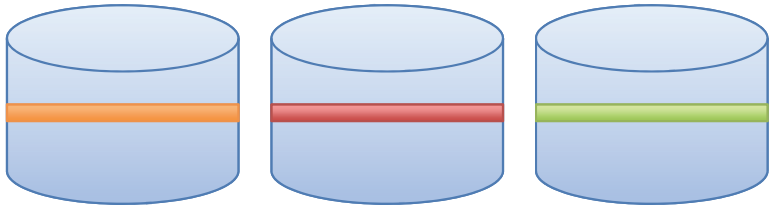
- データ配置の指定
  - １ファイルのデータをひとつのOSTに配置するか、複数のOSTに分散して配置するかはユーザが指定できる
    - デフォルトでは１ファイルあたりひとつのOSTに配置、ファイル単位で使用するOSTが決められる
    - lfs getstripe / lfs setstripeコマンドで参照・変更可能
  - 複数プロセスから単一のファイルに対する処理を高速化したい場合には指定が必要

ひとつのOSTに配置



どんなにがんばっても最大で1OST分の読み書き性能しか得られない

複数のOSTに配置



最大で複数（この場合は3）OST分の読み書き性能が得られる

Lustreのデータ配置の指定の方法

- lfs setstripe [OPTIONS] <ディレクトリ名|ファイル名>
  - 主なオプション：-s size -c count
    - ストライプサイズ size 毎にcount 個のOSTに渡ってデータを分散配置する、という設定にした空のファイルを作成する
    - 既存ディレクトリに対して行うとその後に作るファイルに適用される
    - countに-1を指定すると全OSTを使用
  - 使用例

```
$ rm /mppxc/t00001/data.dat
$ lfs setstripe -s 1M -c 50 /mppxc/t00001/data.dat
$ dd if=/dev/zero of=/mppxc/t00001/data.dat bs=1M count=4096
```

- lfs getstripe <ディレクトリ名|ファイル名>
  - 設定情報を確認する
- lfs df
  - MDT/OST構成情報を確認する

Lustreのデータ配置の指定の例（Oakleaf-FX）

- 共有ファイルシステム上の１ファイルに対する書き込み性能を比較

```
dd if=/dev/zero of=${OUTFILE} bs=1M count=4000
```

1プロセスで1ファイルに4GB書き込み：約270MB/s（約15秒）

```
dd if=/dev/zero of=${OUTFILE} bs=1M count=4000 &
dd if=/dev/zero of=${OUTFILE} bs=1M count=4000 seek=4000 &
……
```

seekでファイル内の位置をずらしながら、同じファイルに同時に10プロセスで書き込み：１プロセスあたり約19MB/s（合計約230秒）
10プロセスの処理が衝突した状態になり性能が低下したと考えられる

```
lfs setstripe -s 1M -c -1 ${OUTFILE}
dd if=/dev/zero of=${OUTFILE} bs=1M count=4000 &
dd if=/dev/zero of=${OUTFILE} bs=1M count=4000 seek=4000 &
……
```

setstripe指定後に実行：１プロセスあたり約200MB/s（合計約20秒）
1プロセスと比べるとまだ遅いが、性能が大幅に改善した（並列に実行した意味があった）

関係ファイル：stripeディレクトリ内

ファイルシステムについて：まとめ

- Oakleaf-FX/Reedbushは複数のファイルシステムを備える
- Oakleaf-FX/Reedbushに限らず、多くのスーパーコンピュータシステムは分散ファイルシステムを持つ
  - 分散ファイルシステムを適切に活用することで、大規模なデータの入出力を高速に行える可能性がある
  - Lustreはlfs setstripeでストライプ設定が可能、単一ファイルに対する並列アクセスをする場合には活用すべし
- 複数のファイルシステムを使い分けられる場合には、用途に合わせて適切なものを選択して利用する必要がある

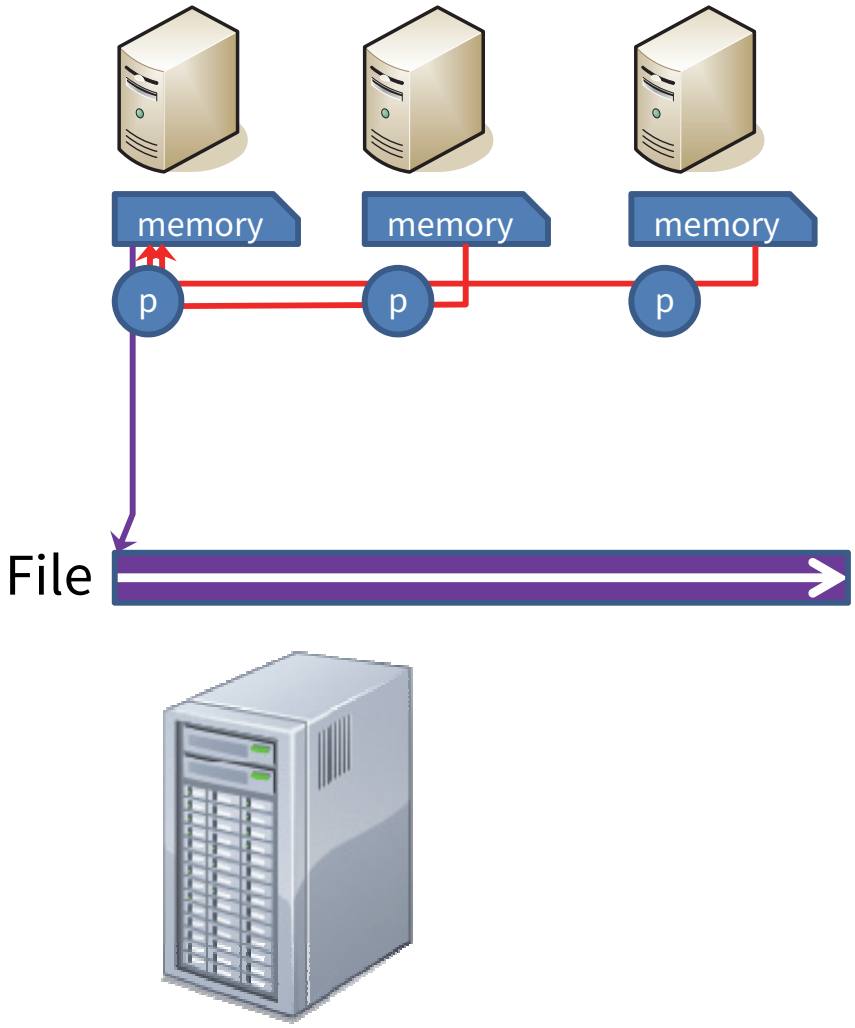
MPI-IOについて

MPI-IO とは

- 並列ファイルシステムをMPIの枠組みで効率的に利用するための仕組み
  - ある程度抽象化を持たせた記述をすることで、（利用者が気にすることなく）最適な実装が利用可能になる（ことが期待される）
    - まとめて読み込んでMPI通信で分配、など
- 以後 API は C言語での宣言や利用例を説明するが、Fortranでも同名の関数が利用できる
  - 具体的な引数の違いなどはリファレンスを参照のこと
  - C++宣言はもう使われていない、C宣言を利用する

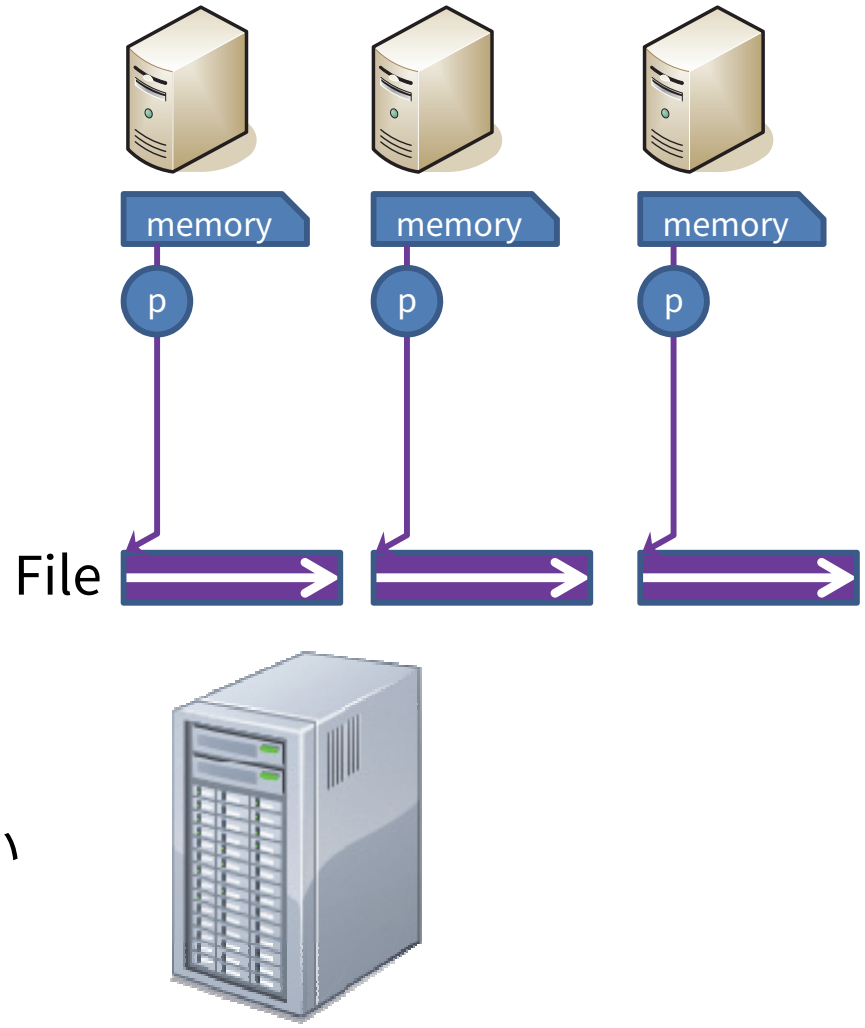
非MPI-IO：よくある並列アプリ上ファイルI/Oの例

- 逐次入出力
  - 1プロセスのみでI/Oを行い、(MPI)通信によりデータを分散・集約する
    - MPI_Gather + fwrite
    - fread + MPI_Scatter
  - 利点
    - 単一ファイルによる優秀な取り回し
    - 読み書き操作回数の削減
  - 欠点
    - スケーラビリティの制限
    - 並列入出力をサポートしたファイルシステムの性能を活かせない



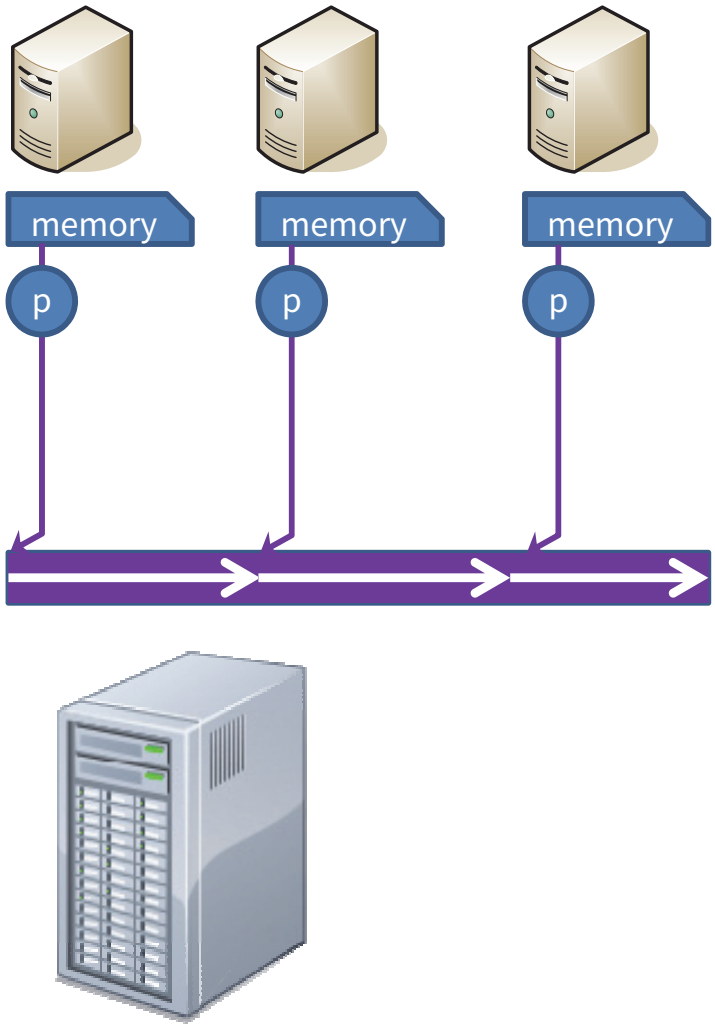
非MPI-IO：よくある並列アプリ上ファイルI/Oの例

- 並列入出力
  - 各プロセスが個別にI/Oを行う
  - 利点
    - ファイルシステムの並列入出力サポートを生かせる
    - スケーラビリティ向上
  - 欠点
    - 入出力回数の増大
      - 多数の小ファイルアクセス
    - 複数ファイルによる劣悪な取り回し
      - プログラム（ソース）も読み/書きにくい



MPI-IOによる並列アプリ上ファイルI/Oの一例

- 単一ファイルに対する並列入出力
  - スケーラビリティ向上
  - プログラム（ソース）もわかりやすい



想定するシナリオ

- MPI-IO関数を使うことで、複数プロセスによる単一ファイルへの並列入出力を簡単に行うことを考える
  - 単一ファイルに対する操作のため、入出力の性能も上げたい場合にはlfs setstripeの設定も必要
  - 今回は利用方法の習得と利便性の理解を目的とし、入出力性能についてはこだわらない

MPI-IOによる並列ファイル入出力

- 基本的な処理の流れ

  1. MPI_File_open(…);
  2. 読み書きの処理を実行
  3. MPI_File_close(…);

- ファイルのオープンとクローズ

```
int MPI_File_open(
  MPI_Comm comm,    // コミュニケータ
  char *filename,   // 操作ファイル名
  int amode,        // アクセスモード（読み書き、作成など）
  MPI_Info info,    // 実装へのユーザからのヒント
  MPI_File *fh      // ファイルハンドラ
)

int MPI_File_close(
  MPI_File *fh      // ファイルハンドラ
)
```

読み書きする場所・パターンの指定

- いくつかの指定方法がある
  - _at 系のread/write関数でその都度指定する
  - MPI_File_seek であらかじめ指定しておく

```
int MPI_File_seek(
  MPI_File fh,            // ファイルハンドラ
  MPI_Offset offset,      // オフセット（バイト数）
  int whence              // 指定方法のバリエーション ※
)
```

※即値、現在位置からのオフセット、ファイル末尾からのオフセット

  - MPI_File_set_view であらかじめ指定しておく

```
int MPI_File_set_view(
  MPI_File fh,              // ファイルハンドラ
  MPI_Offset disp,          // オフセット（バイト数）
  MPI_Datatype dtype,       // 要素データ型
  MPI_Datatype filetype,    // ファイル型 ※
  char* datarep,            // データ表現形式
  MPI_Info info             // 実装へのユーザからのヒント
)
```

※要素データ型と同じ基本型または要素データ型で構成される派生型

読み書き方法のバリエーション

- 「view」を用いた書き込みだけでも様々なバリエーションが存在
  - ブロッキング・非集団出力
    - MPI_File_write
  - 非ブロッキング・非集団出力
    - MPI_File_iwrite / MPI_Wait
  - ブロッキング・集団出力
    - MPI_File_write_all
  - 非ブロッキング・集団出力
    - MPI_File_write_all_begin / MPI_File_write_all_end
- 用途に合わせて使い分ける
  - 非ブロッキング：読み書きが終わらなくても次の処理を実行できる
  - 集団出力：同一コミュニケータの全プロセスが行わねばならない、まとめて行われるため読み書き処理の時間自体は高速

並列出力の例：MPI_File_set_view/writeの場合

- プロセス番号（MPIランク）順に1つずつ整数を書き出すだけの単純な例

```
MPI_File mfh;
MPI_Status st;
int disp;
int data;
const char filename[] = "data1.dat";

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

disp = sizeof(int)*rank; // 各プロセスの書く場所を設定
data = 1+rank; // 適当なデータを設定

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDWR|MPI_MODE_CREATE, MPI_INFO_NULL, &mfh);
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write(mfh, &data, 1, MPI_INTEGER, &st);
MPI_File_close(&mfh);
MPI_Finalize();
```

出力結果の例（Reedbush16プロセス）

```
hexdump data1.dat
0000000 0001 0000 0002 0000 0003 0000 0004 0000
0000010 0005 0000 0006 0000 0007 0000 0008 0000
0000020 0009 0000 000a 0000 000b 0000 000c 0000
0000030 000d 0000 000e 0000 000f 0000 0010 0000
0000040
```

利点

- データの集約処理は不要
- 書き込み結果が１ファイルにまとまる

並列出力の例：その他の関数の場合

- MPI_File_iwrite / MPI_Wait

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_iwrite(mfh, &data, 1, MPI_INTEGER, &req);
MPI_Wait(&req, &st);
```

- MPI_File_write_all

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write_all(mfh, &data, 1, MPI_INTEGER, &st);
```

- MPI_File_write_all_begin / MPI_File_write_all_end

```
MPI_File_set_view(mfh, disp, MPI_INTEGER, MPI_INTEGER, "native", MPI_INFO_NULL);
MPI_File_write_all_begin(mfh, &data, 1, MPI_INTEGER);
MPI_File_write_all_end(mfh, &data, &st);
```

- MPI_File_write_at

```
MPI_File_write_at(mfh, disp, &data, 1, MPI_INTEGER, &st);
```

演習

- 配列を出力するMPI-IOプログラムを作成せよ
  - いままでの例は全て変数１つを出力していた、個数を制御しているパラメタは……
- 出力したデータを読み込むMPI-IOプログラムを作成せよ
  - 基本的にwriteをreadにするだけ
  - writeとreadのバリエーションが異なっていても良い
    - 非ブロッキング書き込み結果をブロッキング読み込みしても良い
  - writeとreadのプロセス割り当てが同じである必要も無い
    - 例：16プロセスで4要素ずつwrite→8プロセスで8要素ずつread
- Oakleaf-FXとReedbushによる結果を比較せよ
  - 完全に一致するのだろうか？
    - CPUのバイトオーダー（エンディアン）が違う、MPIの実装者が違う

関係ファイル：mpiioディレクトリ内

互換性・可搬性について

- MPI_File_set_viewのdatarep引数によるデータ表現形式によって可搬性を向上させることができる
  - “native”: メモリ上と同じ姿での表現（何も変換しない）
  - “internal”: 同じMPI実装を利用するとき齟齬がない程度の変換
  - “external32”: MPIを利用する限り齟齬がないように変換
  - その他のオプションはMPI仕様書を参照
- 「ビュー」を使わない入出力処理の場合は可搬性が保証されない
  - 一番記述量が少ないのはMPI_File_open / MPI_File_wite_at / MPI_File_close による書き込みだが、可搬性は低い
- nativeではOakleaf-FXとReedbushの結果は一致しない
- external32では結果が一致する
  - Reedbushの出力結果がOakleaf-FXの出力結果と同様に変化するようだ

構造体（の配列）の入出力

- 構造体を使いたい場合は一手間必要
  - 独自のデータ型を作成し、それを用いて処理を行う（MPI-IOに限った話ではない）

```
#define N 4
struct SMyData
{
  int key;
  double value;
};
struct SMyData data[N];
int iblock[2];
MPI_Aint idisp[2];
MPI_Datatype itype[2];
MPI_Datatype MPI_MYTYPE;

disp = sizeof(struct SMyData)*rank * N;
for(i=0; i<N; i++){
  data[i].key   = rank+1;
  data[i].value = (double)(16*(rank+1) + i+1);
}
```

```
iblock[0] = 1; iblock[1] = 1;
itype[0] = MPI_INT; itype[1] = MPI_DOUBLE;
MPI_Get_address(&data[0].key, &idisp[0]);
MPI_Get_address(&data[0].value, &idisp[1]);
idisp[1] -= idisp[0]; idisp[0] -= idisp[0];
MPI_Type_create_struct
  (2, iblock, idisp, itype, &MPI_MYTYPE);
MPI_Type_commit(&MPI_MYTYPE);

MPI_File_open
 (MPI_COMM_WORLD, filename,
MPI_MODE_RDWR|MPI_MODE_CREATE,
MPI_INFO_NULL, &mfh);
MPI_File_set_view
 (mfh, disp, MPI_MYTYPE, MPI_MYTYPE, "native",
MPI_INFO_NULL);
MPI_File_write(mfh, data, N, MPI_MYTYPE, &st);
MPI_File_close(&mfh);
```

共有ファイルポインタ

- ファイルポインタを共有することもできる
  - MPI_File_read/write_sharedで共有ファイルポインタを用いて入出力
  - 読み書き動作が他プロセスの読み書きにも影響する、「カーソル」の位置が共有される
  - 複数プロセスで順番に（到着順に）１ファイルを読み書きするような際に使う
    - ログファイルなど？
- 集団入出力
  - MPI_File_read/write_ordered
  - 順序が保証される
    - ランク順に処理される
    - 並列ではない

```
#define COUNT  2
MPI_File fh;
MPI_Status st;
int buf[COUNT];
MPI_File_open
  (MPI_COMM_WORLD, “datafile”,
MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_read_shared
(fh, buf, COUNT, MPI_INT, &st);
MPI_File_close(&fh);
```

少し高度なジョブの操作について

- ジョブの詳細な状態の把握
- コマンドラインによるジョブの制御
- ステップジョブ・チェーンジョブ

ジョブの詳細な状態の把握

- ジョブの詳しい状態を確認する
  - Oakleaf/Oakbridge-FX : pjstat -s ジョブID
  - Reedbush : rbstat -s ジョブID
  - ジョブIDを指定しない場合は実行前・実行中の、自分のすべてのジョブが対象
- 実行中のジョブのノード割り当て、ランク割り当ての確認
  - Oakleaf/Oakbridge-FX : pjstat -X ジョブID

➢ pjstat/rbstatには他にも様々なオプションがある。これら以外のオプションはpjstat --help、rbstat -h、man/オンラインドキュメントを参照のこと。

pjstat -s の出力例

```
Oakleaf-FX scheduled stop time: 2012/06/29(Fri) 09:00:00 (Remain:
2days 17:26:40)

 JOB ID                     : 288534
 JOB NAME                   : STDIN
 JOB TYPE                   : INTERACT
 JOB MODEL                  : NM
 RETRY NUM                  : 0
 SUB JOB NUM                : -
 USER                       : t00004
 PROJECT                    : gt00
 RESOURCE UNIT              : oakleaf-fx
 RESOURCE GROUP             : interactive_n1
 APRIORITY                  : 127
 PRIORITY                   : 63
 SHELL                      : /bin/bash
 COMMENT                    :
 LAST STATE                 : RNA
 STATE                      : RUN
 PRM DATE                   : 2012/06/26 15:33:07
 ...
 MAIL ADDRESS               : t00004@oakleaf-fx-6
 STEP DEPENDENCY EXP        :
 STEP EXITING WAIT MODE     : 2
 FILE MASK                  : 0022
 STANDARD OUT FILE          : -
 STANDARD ERR FILE          : -
 INFORMATION FILE           : -
 PJSUB DIRECTORY            : /home/t00004/private/
 FILE SYSTEM NAME           :
 APPLICATION NAME           : :submitted on oakleaf-fx-6
 ACCEPT DATE                : 2012/06/26 15:33:05
 QUEUED DATE                : 2012/06/26 15:33:06
 EXIT DATE                  : -
 LAST HOLD USER             :
 HOLD NUM                   : 0
 HOLD TIME                  : 00:00:00 (0)
 JOB START DATE             : 2012/06/26 15:33:07<
 JOB END DATE               : -
 JOB DELETE DATE (REQUIRE)  : -
 JOB DELETE DATE            : -
 STAGE IN START DATE        : -
 STAGE IN END DATE          : -
 STAGE IN SIZE              : 0.0 MB (0)
 STAGE OUT START DATE       : -
 STAGE OUT END DATE         : -
 STAGE OUT SIZE             : 0.0 MB (0)
 NODE NUM (REQUIRE)         : 1
 CPU NUM (REQUIRE)          : 8
 ELAPSE TIME (LIMIT)        : 02:00:00 (7200) <DEFAULT>
 MEMORY SIZE (LIMIT)        : 28672.0 MiB (30064771072)
 DISK SIZE (LIMIT)          : 240000.0 MB (240000000000)
 ...
 NODE NUM (ALLOC)           : 1:1x1x1
 MEMORY SIZE (ALLOC)        : 28672.0 MiB (30064771072)
 CPU NUM (ALLOC)            : 16
 ELAPSE TIME (USE)          : 00:00:12 (12)
 NODE NUM (UNUSED)          : 0
 NODE NUM (USE)             : 1
 NODE ID (USE)              : 0x030F0006
 TOFU COORDINATE (USE)      : (8,4,0)
 MAX MEMORY SIZE (USE)      : 0.0 MiB (0)
 CPU NUM (USE)              : 0
 USER CPU TIME (USE)        : 0 ms
 SYSTEM CPU TIME (USE)      : 0 ms
 CPU TIME (TOTAL)           : 0 ms
 DISK SIZE                  : 0.0 MB (0)
 I/O SIZE                   : 0.0 MB (0)
 FILE I/O SIZE              : 0.0 MB (0)
 EXEC INST NUM              : 0
 EXEC SIMD NUM              : 0
 TOKEN                      : -
```

rbstat -s の出力例

```
JOB ID              : 1193
JOB NAME            : job.sh
JOB TYPE            : BATCH
USER                : z30097
PROJECT             : pz0097
QUEUE (REQUIRE)     : u-small
COMMENT             : -
STATE               : RUNNING
ARRAY INDICES       : -
HOLD TYPES          : n
JOIN PATH           : n
MAIL SEND FLAG      : a
MAIL ADDRESS        : -
EXEC HOST           : n106*36+n107*36+n124*36+n125*36
OUTPUT PATH         : reedbush-u4:/lustre/pz0097/z30097/work/gitprojects/mpitest/mpitest1/job.sh.o1193
ERROR PATH          : reedbush-u4:/lustre/pz0097/z30097/work/gitprojects/mpitest/mpitest1/job.sh.e1193
RERUNABLE           : False
SUBMIT ARGUMENT     : job.sh
NODE NUM (REQUIRE)  : 4
CPU NUM             : 144
ELAMSE TIME (LIMIT) : 00:01:00
MEMORY SIZE (LIMIT) : 976GB
ELAPSE TIME (USE)   : 00:00:03
MEMORY SIZE (USE)   : 1MB
CPU TIME (USE)      : 00:00:00
SUBMIT DATE         : 07/02 14:02:19
START DATE          : 07/02 14:02:21
END DATE            : -
EXIT STATUS         : -
RUN COUNT           : 1
ENERGY              : -
POWER AVERAGE       : -
POWER MAX           : -
POWER MIN           : -
DEPENDENCY          : -
TOKEN               : 0.0
```

pjstat -X の出力例

- 同一ノードには同一のNODEIDが表示される
  - 2ノード、8プロセス（1ノードあたり4プロセス）の場合の例

```
$ pjstat -X

JOBID        RANK NODEID
288538          0 0x010A0006
                1 0x010A0006
                2 0x010A0006
                3 0x010A0006
                4 0x02020006
                5 0x02020006
                6 0x02020006
                7 0x02020006
```

コマンドラインによるジョブの制御

- ジョブ投入時の引数指定によりジョブの内容を変更（上書き）できる
- pjsub –L node=2,rscgrp=tutorial スクリプト名
  - スクリプト内の記述に関わらずtutorialリソースグループの2ノードを使用して実行
  - ジョブスクリプトに書いたものより、コマンドライン引数で指定したオプションのほうが優先される
- 投入したスクリプトに記述された設定と実際のオプションが異なる場合がある
- pjstat/rbstatコマンドを使って確認すれば正しい（実際に有効となっている）情報が得られる

演習

- ジョブの投入・実行と環境変数に関する実験
  - Oakleaf-FX
    - `pjsub -L rscgrp=tutorial,node=1`
      コマンドを実行し、標準入力に env|sort; sleep 30を入力してCtrl-D
    - pjstat -sで詳細情報を確認せよ
  - Reedbush
    - `qsub -q u-tutorial -l select=1 -W group_list=gt00 -l walltime=00:01:00`
      コマンドを実行し、標準入力にenv|sort;sleep 30を入力してCtrl-D
    - rbstat -sで詳細情報を確認せよ
  - ジョブ終了後、STDIN.o～に出力された内容を確認せよ
    - どのような環境変数が設定されたか？
  - env を mpiexec env / mpirun env に変更すると、どのような環境変数が設定されるか？

関係ファイル：jobディレクトリ内

解説と補足

- 標準入力から与えたジョブスクリプトのジョブ名はSTDINになる (-Nオプションで変更可能)
- ジョブ内では、PJM_/PBS_で始まる環境変数が設定される
  - PJM_O_/PBS_O_で始まる環境変数には、pjsubした環境の情報が格納される
- 更に、MPIプロセス内では、FLIB_・OMPI_・I_MPI_で始まる環境変数が設定される

ステップジョブ・チェーンジョブ

- Oakleaf/Oakbridge-FX：ステップジョブ
- Reedbush：チェーンジョブ
- 複数のジョブの間で実行の順序関係や依存関係を指定できる
- ステップジョブ・チェーンジョブは複数サブジョブから構成され、各サブジョブは同時に実行されることはない
  - 前のジョブが終了したら自動的にジョブが投入される、ような状況
- 実行順序の制御が必要な用途の例
  - 入力データを生成するジョブを実行したあとで、計算ジョブを実行する
  - 長いジョブを分割して実行するために、チェックポイントを取って終了し、次の（続きの）ジョブを実行する
- 単純な例として「あるジョブが正常終了したら実行される」ジョブを投入することを考える
  - さらに複雑なケースについてはマニュアル等を参照

関係ファイル：stepjobディレクトリおよびchainjobディレクトリ内

Oakleaf/Oakbridge-FX：ステップジョブ

- pjsubに--stepオプションを指定して利用する
  - 例

```
$ pjsub --step job1.sh
[INFO] PJM 0000 pjsub Job 2632030_0 submitted.
$ pjstat
JOB_ID     JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE       TOKEN NODE:COORD
2632026    job1.sh    RUNNING pz0097   -              07/02 23:14:12    00:01:30         - -
$ pjsub --step --sparam "jid=2632030,sd=ec!=0:one" job2.sh
[INFO] PJM 0000 pjsub Job 2632030_1 submitted.
$ pjstat
JOB_ID     JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE       TOKEN NODE:COORD
2632026    job1.sh    RUNNING pz0097   -              07/02 23:14:12    00:01:40         - -
```

  - --sparam以下に細かい動作の内容を指定する
  - jid=2632030
    - 依存するジョブのIDを指定

※jidのみを指定した場合には、単純に前のジョブが終了すると後続のジョブが実行される

  - sd=ec!=0:one
    - ジョブを削除する条件（実行しない条件）を指定
    - この例は「依存するジョブの終了ステータスが0ではない場合はこのジョブを削除する」を意味する
    - 依存するジョブが異常時にはexit 1で終了するようにしておけば、後から投入されたジョブは実行されずに削除される

Oakleaf/Oakbridge-FX：ステップジョブ（続き）

- ジョブの確認について
  - pjstatで見えるJOB_IDは最初のジョブのID
  - JOB_NAMEは実行時のサブジョブ
  - pjstatを見てもステップジョブの存在はわからない
  - pjstat -sの「SUB JOB NUM」をみればステップジョブの存在と進行度が確認できる
    - 0/2、1/2など
- 出力されるファイルについて
  - Oakleaf/Oakbridge-FXでは特に指定しない場合、出力ファイル名が「JOB_NAME.{o,e}JOB_ID」になる
  - ステップジョブの場合には「JOB_ID」のあとにさらに「_サブジョブID」が追加される

Reedbush：チェーンジョブ

- qsubに-Wオプションを指定して利用する
  - 例
```
$ qsub job1.sh
2021.reedbush-pbsadmin0
$ rbstat
JOB_ID  JOB_NAME   STATUS  PROJECT  QUEUE     START_DATE       ELAPSE   TOKEN    NODE
2021    job1.sh    RUNNING pz0097   u-debug   07/03 00:07:29   00:00:00 0.0      1
$ qsub -W depend=afterok:2021 job2.sh
$ rbstat
JOB_ID  JOB_NAME   STATUS  PROJECT  QUEUE     START_DATE       ELAPSE   TOKEN    NODE
2021    job1.sh    RUNNING pz0097   u-debug   07/03 00:07:29   00:00:25 0.0      1
2022    job2.sh    HELD    pz0097   u-debug   -                -        (0.0)    1
```
  - dependの以下に細かい動作の内容を指定する
  - afterok:2021
    - ジョブを実行する条件を指定
    - ジョブ番号2021のジョブが正常終了したらこのジョブを実行する

※他にafternotok, afteranyなど色々な条件がある

  - rbstatコマンドでチェーンジョブの投入状況がわかる
    - HELDは依存ジョブの終了を待っている状態
    - 条件によりHジョブだけが残ってしまうことがある→qdelで消す

makeの活用について

makeとは？

- プログラムの分割コンパイル等を支援するツール（ソフトウェア）
- 変更があったファイルのみを再コンパイルする、等の指定が可能であり、大規模なプログラムを書くときに便利
- 本質的にはワークフロー言語の実行エンジン
  - コンパイルに限らず、処理の依存関係を記述して、依存関係に従ってコマンドを実行できる
- 一般的なUn*x系OS・Linux環境の多くで利用可能
  - makeの実装による違いもあるため注意すること

- この講習会では GNU make (version 3.81/3.82)を使用する
  - Oakleaf/Oakbridge-FX：3.81
  - Reedbush：3.82

Hello, world!

- hello.c

```
#include <stdio.h>
int main(int argc, char** argv) {
  printf("Hello, world!¥n");
  return 0;
}
```

- Makefile

```
hello: hello.c
	gcc -o hello hello.c
```

  - スペースではなくタブ

- 実行

```
$ make hello
gcc -o hello hello.c
```

さらにもう一度makeを実行するとどうなるか？

```
$ make hello
make: `hello' is up to date.
```

※コマンド(gcc)は実行されない

Makefileの構造

- ルールは、ターゲット、依存するファイル、コマンドで記述される

  ターゲット: 依存するファイル …

  タブ コマンド

  タブ …

  - 誤ってタブをスペースにした場合、“missing separator” などのメッセージが表示される
    - 環境設定などにより異なる場合もある
    - より親切な警告メッセージが表示される場合もある
- makeの実行
  - make ターゲット
  - ターゲットを省略した場合は、Makefileの最初のターゲットが指定されたものとして実行される
    - Makefileを1行目から順番に見ていって最初に出現したターゲット、という意味

コマンドが実行される条件

- 以下のいずれかが満たされるとコマンドを実行
  - ターゲットが存在しない
  - (ターゲットのタイムスタンプ)
    < (依存するいずれかのファイルのタイムスタンプ)

- 依存するファイルXが存在しない場合、make Xを先に実行
- コマンドを実行した後の終了ステータスが0以外の場合は続きの処理を実行しない

少し複雑な例

- hello.c

```
#include <stdio.h>
void hello(void) {
  printf("Hello, world!¥n");
}
```

- main.c

```
void hello(void);
int main(int argc, char** argv) {
  hello();
  return 0;
}
```

- Makefile

```
hello: hello.o main.o
        gcc -o hello hello.o main.o
hello.o: hello.c
        gcc -c hello.c
main.o: main.c
        gcc -c main.c
```

1. 実行

```
$ make
gcc -c hello.c
gcc -c main.c
gcc -o hello hello.o main.o
```

2. hello.cを書き換え

例: world! を world!! に書き換え

3. makeを再実行

```
$ make
gcc -c hello.c
gcc -o hello hello.o main.o
```

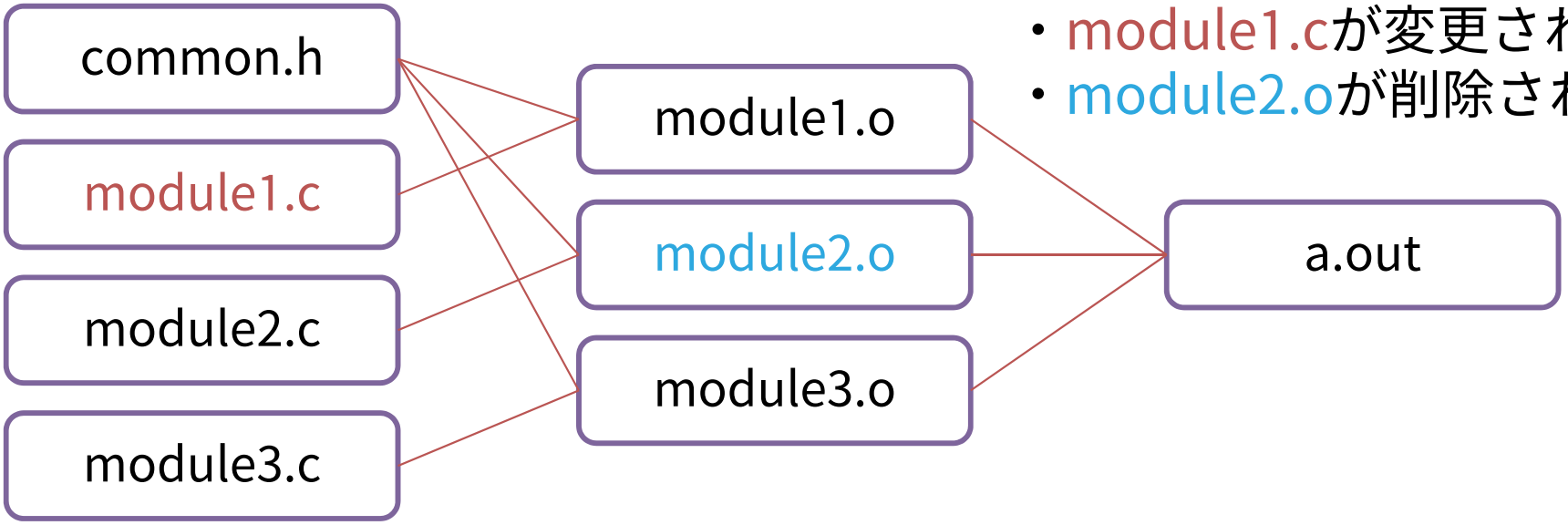
部分コンパイル

- 2回目のmakeで起きていたこと
  - main.oのコンパイルは、main.cに変更がなかったため行われなかった
- Makefileに依存関係を適切に記述することで、変更があった部分だけを再コンパイルすることができる

依存関係の記述

```
module1.o: module1.c common.h
        gcc -o module1.o module1.c -c
module2.o: module2.c common.h
        gcc -o module2.o module2.c -c
module3.o: module3.c common.h
        gcc -o module3.o module3.c -c
a.out: module1.o module2.o module3.o
        gcc -o a.out module1.o module2.o module3.o
```

以下の場合に何が起きるか考えよ
- module1.cが変更された場合
- module2.oが削除された場合



makeのtips

- Makefile（makeの対象となるファイル）の指定

```
$ make -f test.mk
```

- 長い行の折り返し

```
hello: hello.o main.o
        gcc -g -Wall -O3 ¥
          -o hello hello.o main.o
```

  - 半角円記号（¥）と　半角バックスッシュ（＼）は同じもの（フォントなどの都合）
  - 折り返した後もタブは必要

- PHONYターゲット

```
.PHONY: clean
clean:
        rm -f hello hello.o main.o
```

（偶然、運悪く）cleanというファイルが存在していたとしても必ず実行される

- ディレクトリを移動してmake

```
$ make -C hello2 target
```

cd hello2; make target と同様
実行後は元のディレクトリに戻る

変数の使い方

- 代入方法

```
OBJECTS=main.o hello.o
```

- 参照方法

```
hello: $(OBJECTS)
```

${OBJECTS}でもよい
$OBJECTSとすると、$(O)BJECTSと同じことになる

- 変数代入時における変数の参照（展開）

```
CFLAGS=$(INCLUDES) -O -g
INCLUDES=-Idir1 -Idir2
```

CFLAGSは -Idir1 -Idir2 -O -g に展開される（置き換えられる）

動作の制御

- 実行しようとするコマンドを表示しない

```
test1:
	@echo Test message
```

- コマンド終了時ステータスを無視する（実行結果に問題があっても次のコマンドを実行する）

```
test2:
	-rm file1 file2 file3
```

条件分岐

- コマンドの条件分岐

```
hello: $(OBJECTS)
ifeq ($(CC),gcc)
        $(CC) -o hello $(OBJECTS) $(LIBS_FOR_GCC)
else
        $(CC) -o hello $(OBJECTS) $(LIBS_FOR_OTHERCC)
endif
```

- 変数代入の条件分岐

```
ifeq ($(CC),gcc)
  LIBS=$(LIBS_FOR_GCC)
else
  LIBS=$(LIBS_FOR_OTHERCC)
endif
```

- 変数代入には行頭のタブは不要
- 行頭のスペースは無視される
- コマンド中には書けない

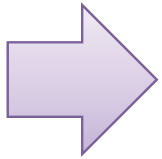
- 条件分岐に関するディレクティブ
  - ifeq, ifneq, ifdef, ifndef

特殊な変数

- make実行時にターゲット名や依存ファイル名などに展開される特殊な変数がある

$@	ターゲット名
$<	最初の依存ファイル
$?	ターゲットより新しい依存ファイル
$+	すべての依存ファイル

```
hello: hello.o main.o
        gcc -o hello ¥
        hello.o main.o
hello.o: hello.c
        gcc -c hello.c
main.o: main.c
        gcc -c main.c
```



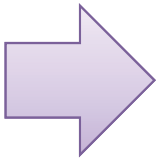
```
CC=gcc
OBJECTS=hello.o main.o
hello: $(OBJECTS)
        $(CC) -o $@ $+
hello.o: hello.c
        $(CC) -c $<
main.o: main.c
        $(CC) -c $<
```

型ルール

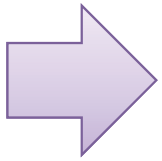
- 指定したパターンにマッチしたら対応するコマンドを実行
  - ***.o は ***.c に依存する

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

```
hello: hello.o main.o
        gcc -o hello ¥
        hello.o main.o
hello.o: hello.c
        gcc -c hello.c
main.o: main.c
        gcc -c main.c
```



```
CC=gcc
OBJECTS=hello.o main.o
hello: $(OBJECTS)
        $(CC) -o $@ $+
hello.o: hello.c
        $(CC) -c $<
main.o: main.c
        $(CC) -c $<
```



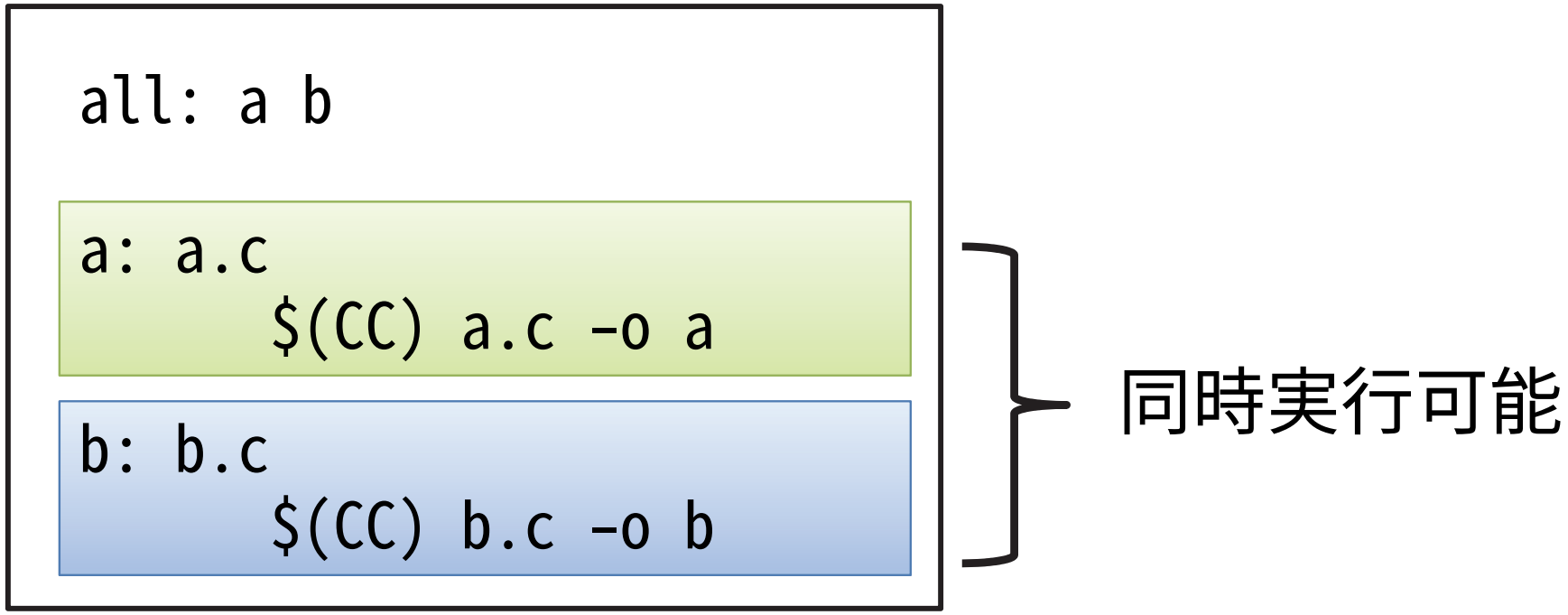
```
CC=gcc
OBJECTS=hello.o main.o
hello: $(OBJECTS)
        $(CC) -o $@ $+
%.o: %.c
        $(CC) -c $<
```

makeを用いた並列処理

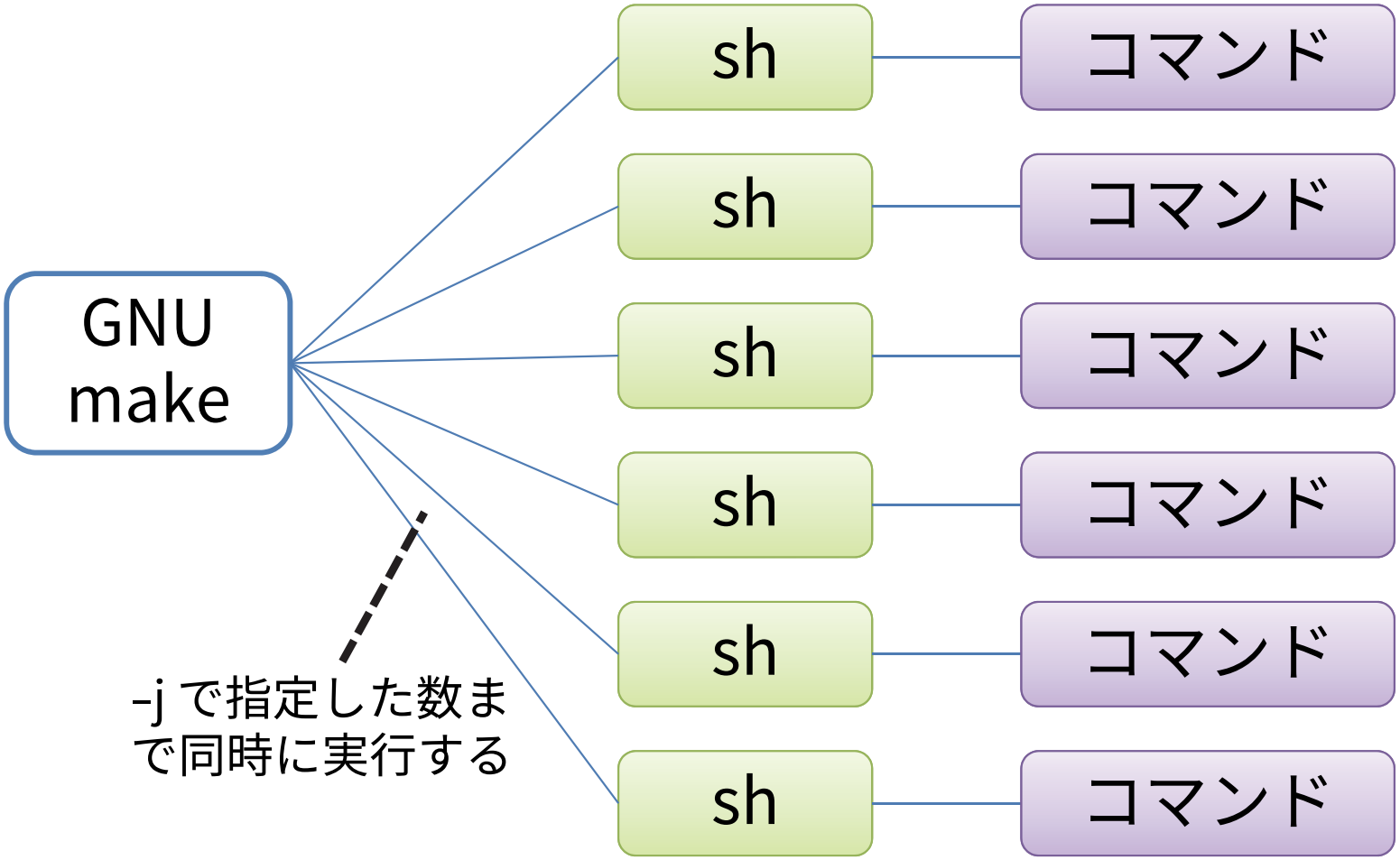
- makeは本質的にはワークフロー言語とその実行エンジン、コンパイル以外にもいろいろなことができる
- makeを使う上で便利な点
  - 実行するコマンドの依存関係を簡単に記述可能
  - 並列化が容易
    - 依存関係の解析はmakeが自動的に行ってくれる
  - 耐故障性
    - 途中で失敗しても、make し直せば続きから実行してくれる

並列make

- make -j による並列化
  - 同時実行可能なコマンドを見つけて並列に実行
  - 依存関係の解析は make が自動的に行ってくれる



並列makeの動作の仕組み



並列make使用時の注意点

- make -j 最大並列度
  - 最大並列度で指定した数まで同時にコマンドを実行する
  - 最大並列度の最大値は（RHELの場合は） 4096
    - それ以上を指定すると 1 を指定したものとみなされる
  - 省略した場合、可能な限り同時にコマンドを実行する（並列度∞）
- make -j が正常に動作しない場合
  - Makefileの書き方の問題
    - 暗黙の依存関係
    - 同名の一時ファイル
  - リソース不足
    - 使用メモリやプロセス数が多すぎる
    - 最大並列度を適切に設定する必要がある
- 複数ノードを用いた並列makeはできない
  - GXPなどを利用する必要がある

暗黙の依存関係

- 逐次 make の実行順序に依存した Makefile の記述をしてはいけない
- 左のターゲットから順番に処理されることに依存したMakefileの例

```
all: 1.out 2.out
1.out:
	sleep 1; echo Hello > 1.out
2.out: 1.out
	cat 1.out > 2.out
```

- 依存関係を明示して逐次処理させる必要がある

同名の一時ファイル

- 逐次 make 実行順序に依存する Makefile の別な例
- 同名の一時ファイルを使用すると、並列実行時に競合する
  - 実行できたとしても正しい結果が得られない可能性

```
all: a b
a: a.c.gz
	gzip -dc < a.c.gz > tmp.c
	$(CC) tmp.c -o a
b: b.c.gz
	gzip -dc < b.c.gz > tmp.c
	$(CC) tmp.c -o b
```

tmp.cが競合
→異なる名前
にすれば良い

演習

- Makefileの読み方を確認し、並列実行時の挙動を理解する
- 以下のMakefileについて、変数や%を使わない場合にどのようなMakefileとなるだろうか
- 「make」と「make -j」の実行時間を予想し、実際に測定して比較せよ

```
FILE_IDS := $(shell seq 1 10)
FILES    := $(FILE_IDS:%=%.dat)

all: $(FILES)

%.dat:
	sleep 5
	touch $@
```

関係ファイル：makeディレクトリ内

まとめ

- シェル・シェルスクリプトとテキスト処理プログラムについて理解することで柔軟な処理が可能になる
  - 複雑なコマンド操作、ジョブ実行、解析処理を容易に実行可能
- ファイルシステムの特徴を理解し、MPI-IOを活用することで、入出力を効率的に行うことができる
- ジョブ管理システムを使いこなすことで効率よくジョブを操作できるようになる
  - ジョブの詳細情報の確認
  - ステップジョブ・チェーンジョブ
- make, Makefile
  - make, Makefileを利用することで、変更箇所だけを再作成する分割コンパイルが可能
  - make -jで並列にmake処理を実行可能、コンパイル以外の処理にも応用が可能