第84回　お試しアカウント付き
並列プログラミング講習会
「MPI基礎：並列プログラミング入門」

ファイル入出力の基礎

東京大学情報基盤センター

内容に関するご質問は
hanawa @ cc.u-tokyo.ac.jp
まで、お願いします。

この講習の目的

- Reedbushを効率的に利用するための技術を習得する

- 内容
  1. ファイルシステムについて
  2. MPI-IOの基礎
  3. makeの活用

演習で使用するプログラムなどのありか

- Reedbush-U
  - /lustre/gt00/z30105/

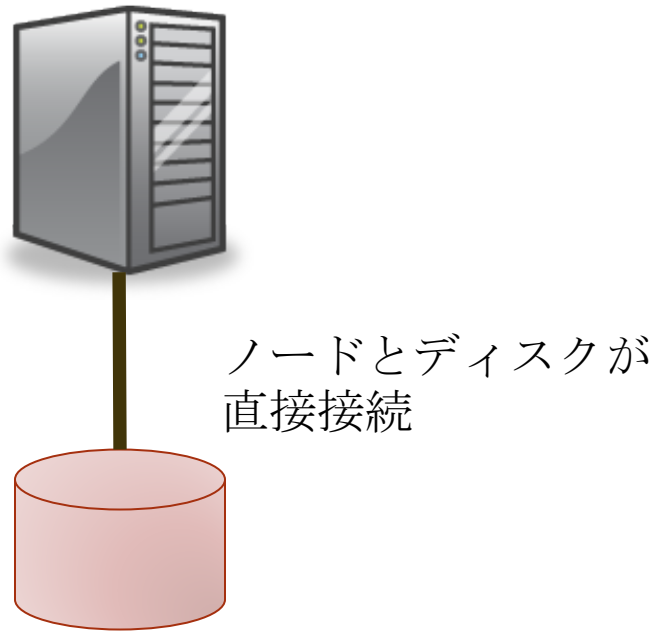
ファイルシステムについて

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
    - Reedbushは計算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
- 並列ファイルシステム
  - 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

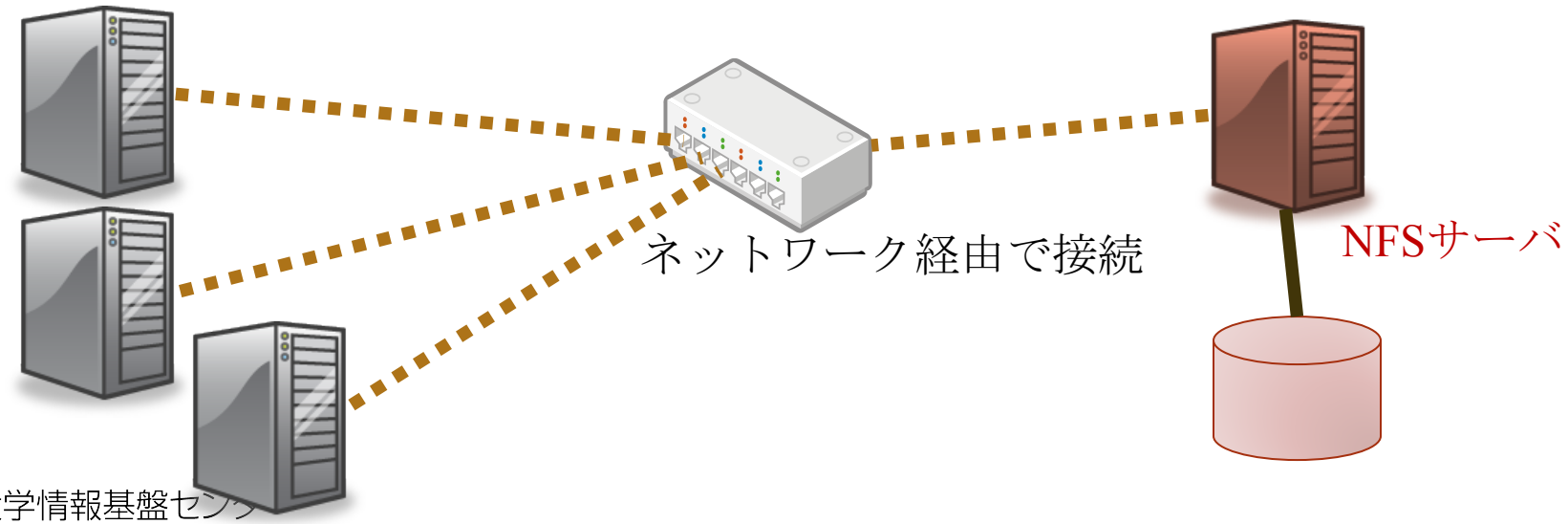
ローカルディスク

- 他のノードから直接アクセスできない記憶域
  - Reedbushは計算ノードにローカルディスクを持たない
  - 一般的なPCの内蔵・外付けストレージなど



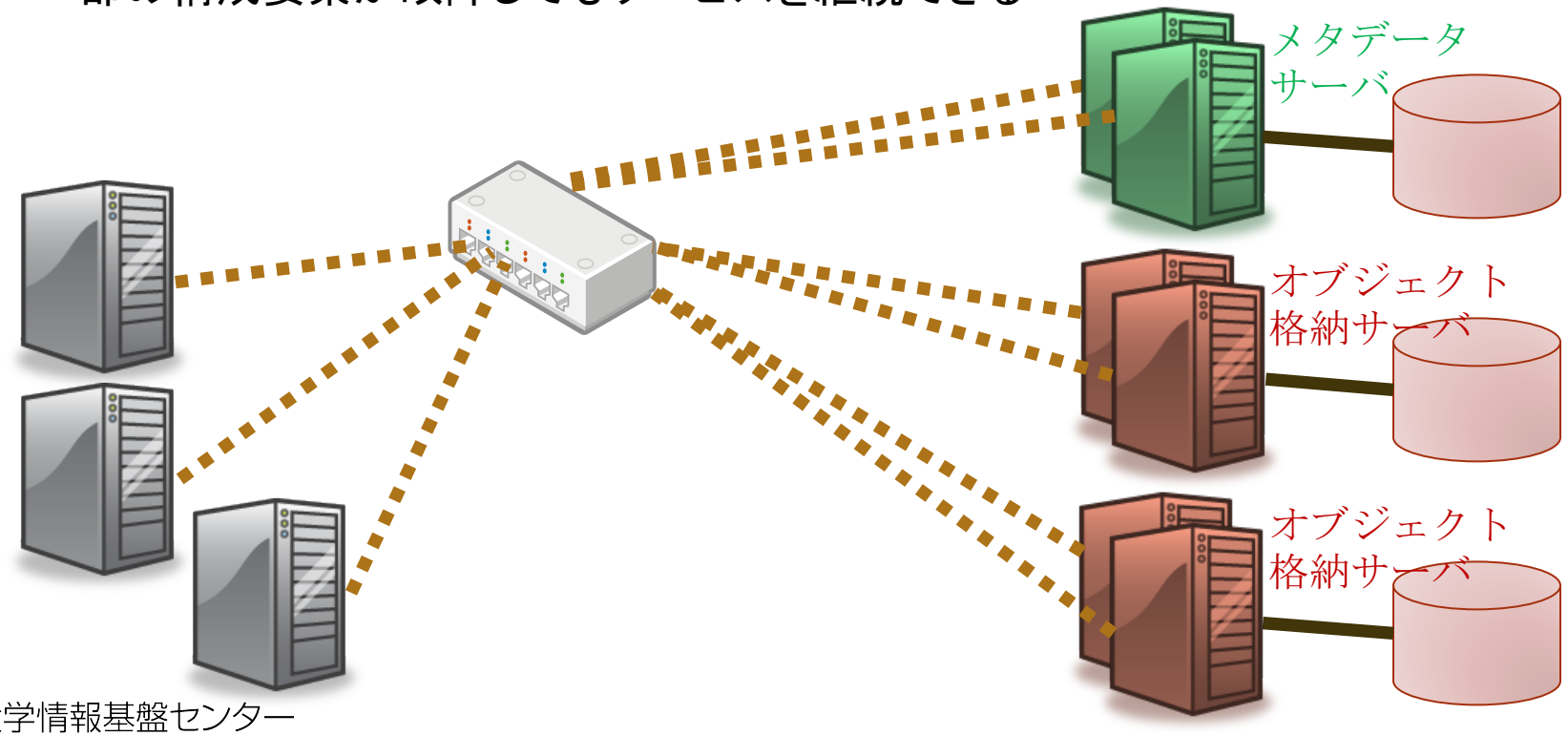
NFS (Network File System)

- 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
  - 大規模システムの主要な共有ファイルシステムとしては不適



並列ファイルシステム

- 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
    - 一部の構成要素が故障してもサービスを継続できる



並列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でも十分な性能が得られる（スケーラビリティが高い）
- 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

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	並列 (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 バッチジョブの投入はここから行う
/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		

※高速ファイルキャッシュシステム（バーストバッファ）については上級者向けの講習会で取り扱う予定

Reedbushの並列ファイルシステム

- Lustreファイルシステム
  - 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)
  - オープンソース開発されており様々なベンダーが採用している
  - Oakleaf-FX (富士通FEFS: Lustreファイルシステムの上位互換)，Oakforest-PACS

利用可能な領域と容量

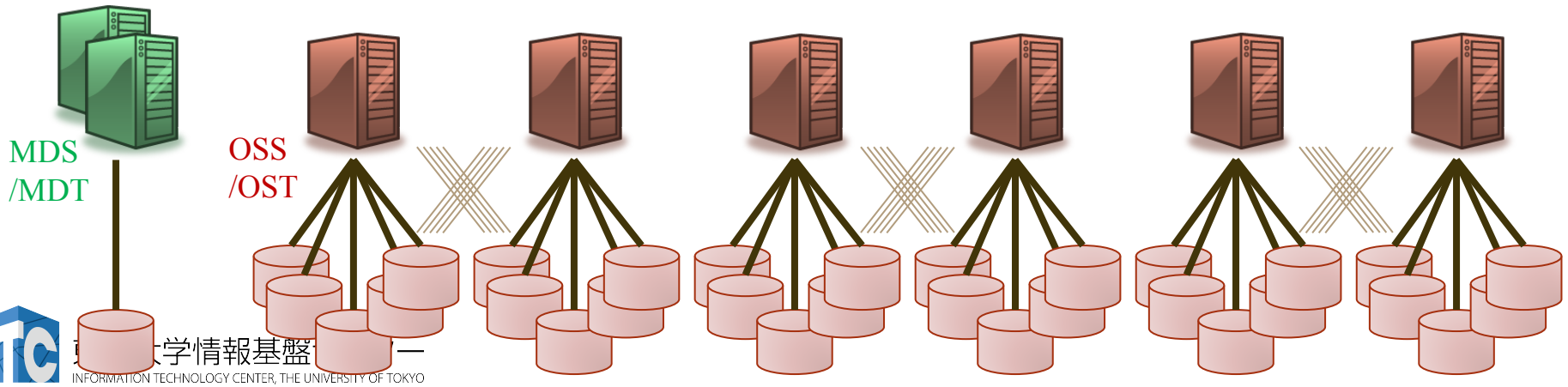
- 個人またはグループに対して利用可能容量の制限（quota）が設定されている
- show_quotaコマンドにより利用可能な領域と容量制限が確認可能
  - 実行例

```
$ show_quota
USER : z30097(pz0097)
    Directory              used(GB)  limit(GB)  nfiles
    /lustre/pz0097/z30097        74       8500      6588
 ------------------------------------------------------
USER : z30097(gz00)
    Directory              used(GB)  limit(GB)  nfiles
    /lustre/gz00/z30097          74       8500      6588
GROUP: gz00
    Directory              used(GB)  limit(GB)  nfiles
    /lustre/gz00               3698      16000   1282379
      |---*z30097                74       ----      6588
（以下省略）
```

Lustreの物理構成とデータ配置

- メタデータを格納するMDS/MDT (Meta Data Server/Target) とデータを格納するOSS/OST (Object Storage Server/Target) により構成される
  - MDS/MDTにはOSS/OST上のデータ配置についての情報が格納される
- Reedbushの構成例
  - MDT：RAID6 (8D2P) × 2 + 4 スペア
  - OSS：「RAID6 (8D2P) × 35 + 10 スペア」を3セット

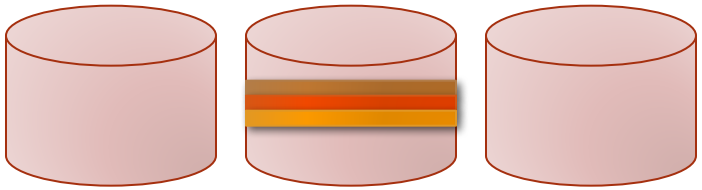
D: データ
P: パリティ



Lustreのデータ配置の指定

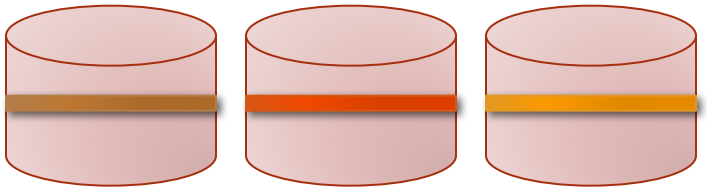
- データ配置の指定
  - 1ファイルのデータをひとつのOSTに配置するか、複数のOSTに分散して配置するかはユーザが指定できる
    - デフォルトでは1ファイルあたりひとつのOSTに配置、ファイル単位で使用するOSTが決められる
    - lfs getstripe / lfs setstripeコマンドで参照・変更可能
  - 複数プロセスから単一のファイルに対する処理を高速化したい場合には複数のOSTを使うような指定が必要

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



複数OST分の読み書き性能が得られる可能性がある

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

※ファイルを作成する際に指定する必要があるため、最初に削除している

```
$ rm /path/to/data.dat
$ lfs setstripe -s 1M -c 50 /path/to/data.dat
$ dd if=/dev/zero of=/path/to/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 関係ファイル：stripeディレクトリ内

- 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プロセスのみと同じ速度は出ていないが）性能が大幅に改善しており並列に実行した意味があった

※これはOakleaf-FXでの性能例。残念ながらReedbushでは性能が向上しない。

ファイルシステムについて：まとめ

- 多くのスーパーコンピュータシステムは並列ファイルシステムを持つ
  - 適切に活用すれば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Ifs 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などには重要
- ファイルシステムの使い分けも重要
  - 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ことで最大の性能が得られることがある
  - Reedbushは複数のファイルシステムを備える
    - 基本的にLustreのみを使う
    - 高いI/O性能が必要な場合はバーストバッファの利用を検討する

MPI-IOについて

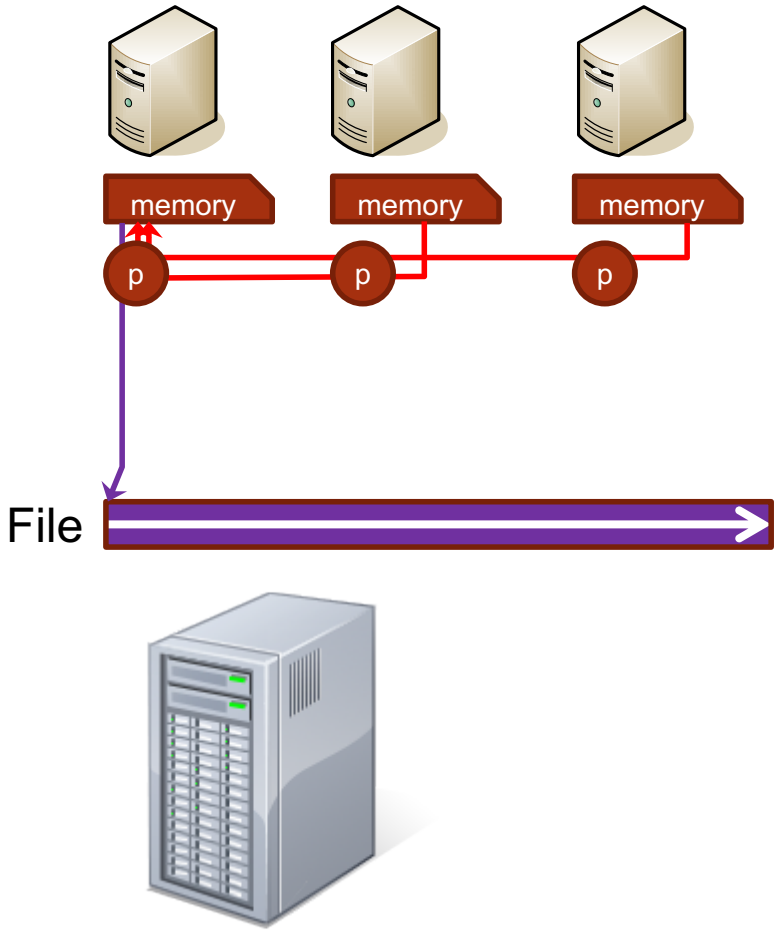
MPI-IO とは

- 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）
  - 例：利用者はファイル上のデータが配列上のどこに配置されて欲しいかだけを指定
    → MPI-IOにより「まとめて読み込んでMPI通信で分配配置」されて高速に処理が行われる、かも知れない

- 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

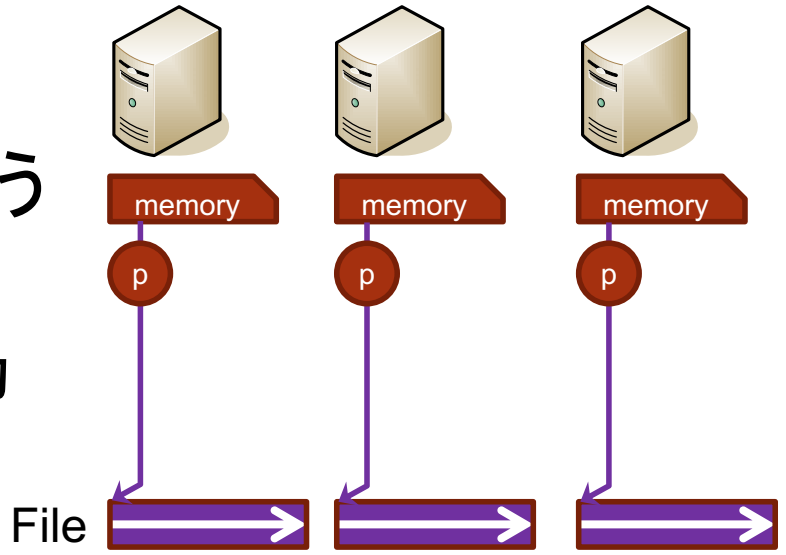
非MPI-IO：並列アプリによるファイルI/Oの例1

- 逐次入出力
  - 1プロセスのみがI/Oを行う、(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



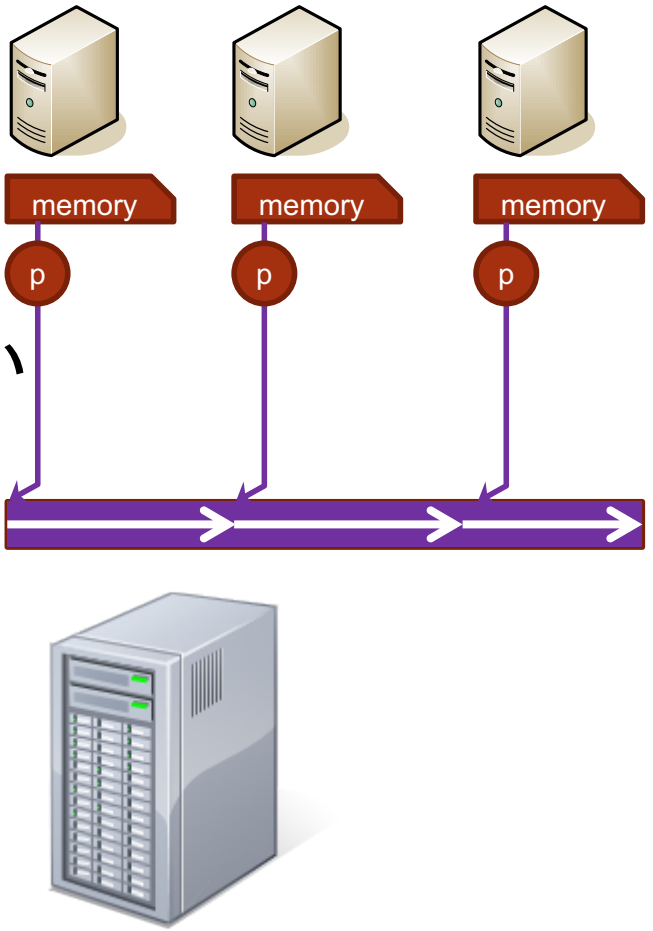
非MPI-IO：並列アプリによるファイルI/Oの例2

- 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
  - 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
  - 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
    - ※いずれにせよReedbushではあまり性能が上がらない

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 関数によりファイルをオープン
2. 読み書きの処理を実行（様々な関数が用意されている）
3. MPI_File_close 関数によりファイルをクローズ

- 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）に基づいて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; // 各プロセスが書き込む場所を設定
data = 1+rank; // 書き込みたいデータを設定（MPIランク+1）

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);
```

いずれも前ページのMPI_File_openと
MPI_File_closeの間を置き換えて使う

演習

1. 出力したデータを読み込むMPI-IOプログラムを作成せよ
   - ヒント：基本的にwriteをreadにするだけ
2. 配列を入出力するMPI-IOプログラムを作成せよ
   - いままでの例はプロセス毎に1つの変数を出力していたが、プロセス毎に2つの変数（長さ2の配列）を出力してみよう
   - ヒント：個数を制御しているパラメタは……

- ちなみに……
  - writeとreadのバリエーションが異なっていても良い
    - 例：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なくても良い
    - 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
  - こうした対応がしやすいのもMPI-IOの利点の一つと言えるだろう

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
- 「View」を使わない入出力処理は可搬性が保証されない
  - MPI_File_open → MPI_File_wite_at → MPI_File_close は記述量が少なく簡単だが、可搬性が低い
- 具体的な例
  - ReedbushとOakleaf-FXで同じプログラム（ソースコード）を使いたい
    - nativeではOakleaf-FXとReedbushの出力結果が一致しない
    - external32では出力結果が一致する（Oakleaf-FXの結果に揃う）
  - Reedbush同士でもMPI処理系を変更すると結果が変わるかも知れない

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に限った話ではなくMPIの一般的な手順）

```
#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
    - 例：ログファイルへの追記
- 共有ファイルポインタを用いた集団入出力もある
  - 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);
```

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
  - 入力データの生成などにも活用できる
- 一般的なUn*x系OS・Linux環境の多くで利用可能
  - makeの実装によるオプション（引数）の違いには注意
  - Reedbushに用意されているmakeは GNU make version 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

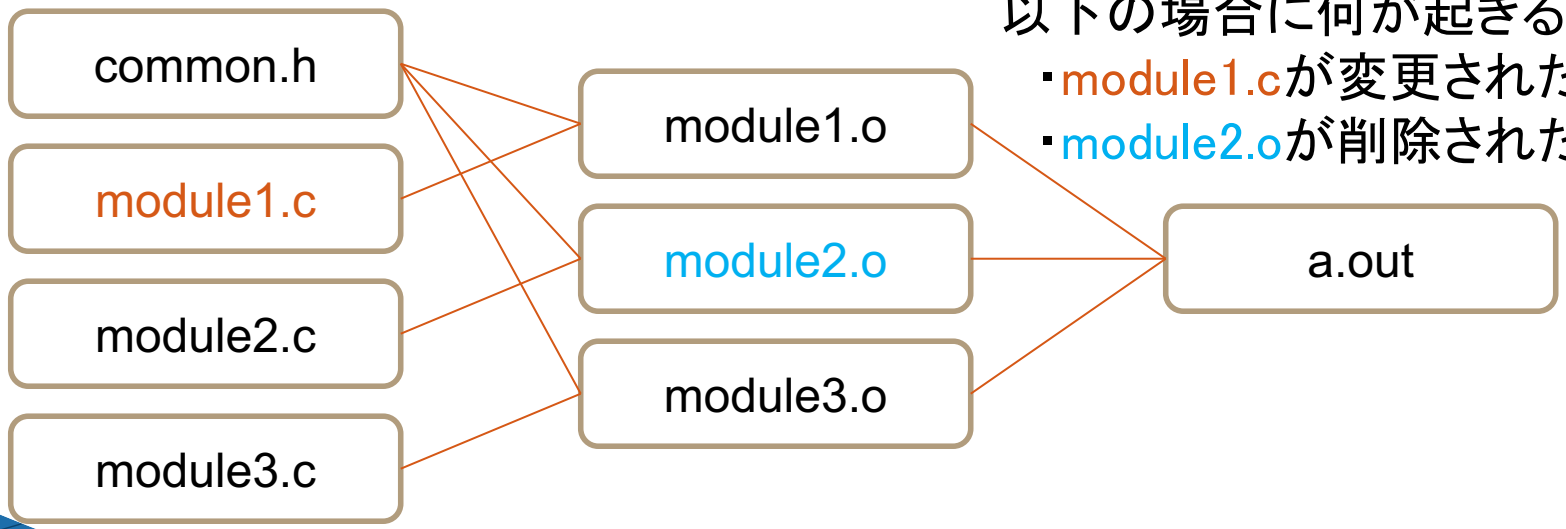
```
$ make
gcc -c hello.c
gcc -o hello hello.o main.o
```

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 -c module1.c
module2.o: module2.c common.h
        gcc -o module2.o -c module2.c
module3.o: module3.c common.h
        gcc -o module3.o -c module3.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

例：コンパイラごとに行う処理を分ける

Makefile

```
all: hello.o
ifeq ($(CC),ICC)
        $(CC) -o hello.out hello.c $(LIBS)
else
        $(CC) -o hello.out hello.c $(LIBS)
endif

ifeq ($(CC),ICC)
  LIBS=-lxxx
else
  LIBS=-lyyy
endif
```

※2組の条件分岐はどちらが先にあっても構わない

利用例

```
$ make
cc -o hello.out hello.c -lyyy
$ CC=ICC make
icc -o hello.out hello.c -lxxx
$ export CC=ICC
$ make
icc -o hello.out hello.c -lxxx
```

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
  - 例

$@	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
$*	サフィックス（拡張子）を除いたターゲット名

```
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・耐障害性
    - 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 による並列実行
  - 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

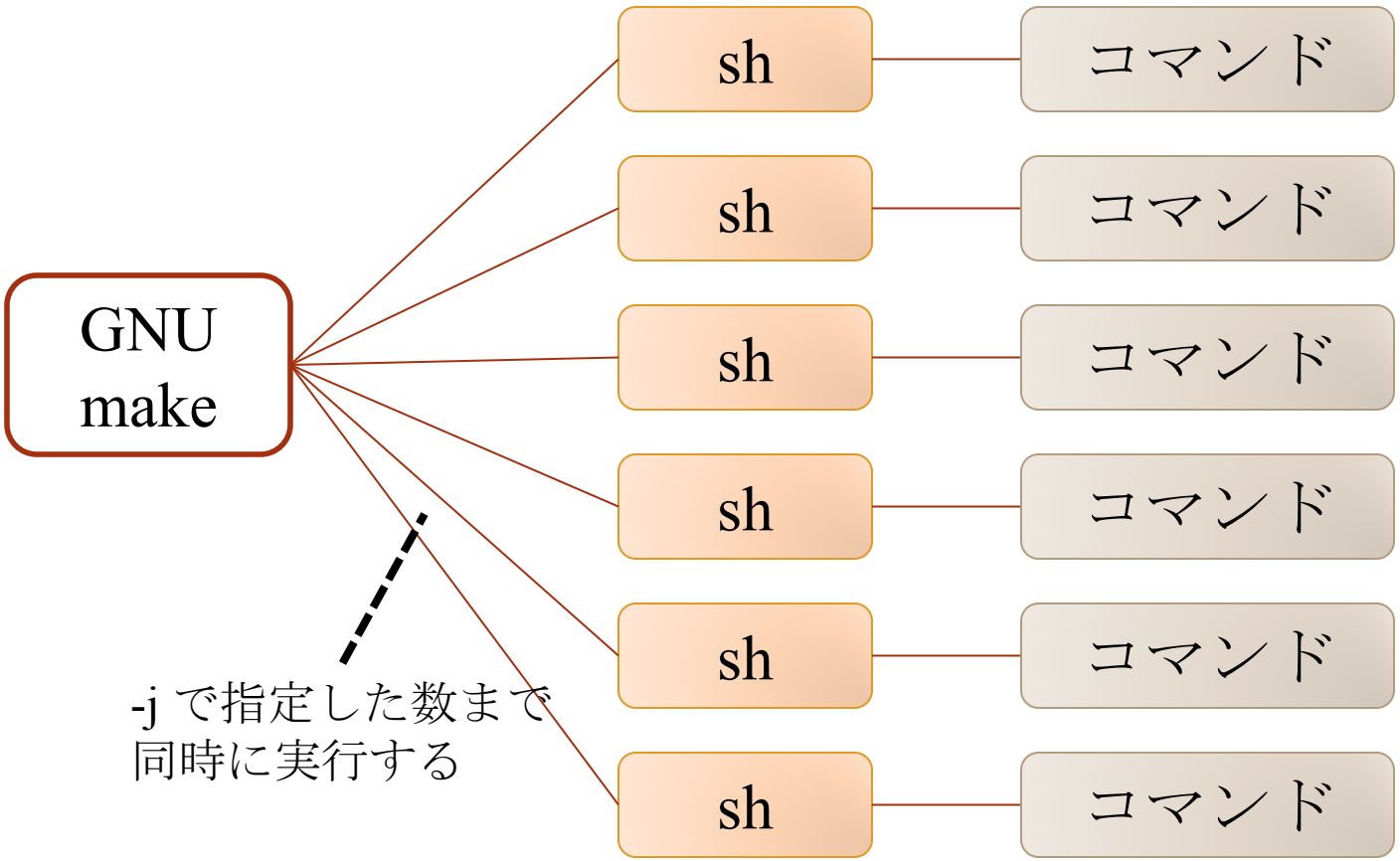
```
all: a b

a: a.c
        $(CC) a.c -o a

b: b.c
        $(CC) b.c -o b
```

同時実行可能

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
  - 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

利用例：多数のファイルを生成する

- 問題設定：入力ファイル FILE0.txt FILE1.txt …… が必要であり、これらはsetup.outプログラムに対応する番号を引数で与えて生成されるとする
  - ./setup.out 0 → FILE0.txt が作られる
- とりあえず要求を満たすMakefileの例

```
all: FILE0.txt FILE1.txt FILE2.txt FILE3.txt
FILE0.txt:
	./setup.out 0
FILE1.txt:
	./setup.out 1
FILE2.txt:
	./setup.out 2
FILE3.txt:
	./setup.out 3
```

○　存在しないファイルだけ作ってくれるので便利
×　ファイル数が1000あったら全部書かねばならない

利用例：多数のファイルを生成する

- 解決方法の1例

```
NUMS:=$(shell seq 0 3)
FILES=$(NUMS:%=FILE%.txt)

setup: $(FILES)

%.txt:
	./setup.out $(subst FILE,,$*)
```

$(shell ～)
- ～にある文字列をコマンドとして実行して結果を取得する
- seqは第1引数から第2引数までの数字列を返す関数
  - または第1引数から第2引数刻みで第3引数までの数字列

$(NUMS:%=FILE%.txt)
- 文字列の置換
- %をワイルドカードとして前後に文字列を追加している

参考：拡張子の変更なら
$(FILES:%.c=%.o) でも
$(FILES:.c=.o) でも良い（頻出）

$(subst A,B,C)
- 文字列の置換関数subst
- Cに対して、AをBで置き換える
- $*はターゲット名からサフィックス（拡張子）を除いたもの
  - 今回の例ではFILE0 FILE1 FILE2 FILE3のいずれかを受け取りFILEを削除、結果として0 1 2といった数値のみが残る

参考

- ちなみに、今回の問題設定ではmakefileを書くよりもシェルでループを回した方が手っ取り早い。実施したい内容にあわせて使い分ける。
  - bashのループ処理を使う例

```
for i in `seq 1 10`
do
  ./setup.out ${i}
done
```

ファイルの存在を確認したい場合

```
for i in `seq 1 10`
do
  if [ ! -e FILE${i}.txt ]; then ./setup.out ${i}; fi
done
```

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
  - 並列度を明示的に指定するとどうなるだろうか？

```
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

- 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
  - Lustre、高い性能を得るにはストライプ設定も必要
  - MPI-IOを使えば並列IOをわかりやすく簡単に書くことができる

- 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できる