

# プログラミングの基礎 (ファイルシステム、バッチジョブ、 makeについて)

2014年6月19日

大島 聡史 (東京大学情報基盤センター)  
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

# 目次

- ▶ ファイルシステム
- ▶ バッチジョブの操作(上級編)
  - ステージング
  - ジョブの詳細な情報の把握
  - コマンドラインオプションの利用
  - ステップジョブ
- ▶ makeの利用
- ▶ makeの応用
  - makeを使った並列処理
  - GXP/GXP makeの利用

# この講習の目的

- ▶ Oakleaf-FXにログインして効率的に作業を行えるようになることを目指し、ファイルシステムやジョブの操作について学ぶ
- ▶ 大規模なプログラムを作成する際に必須となる、分割コンパイルの方法について学ぶ
- ▶ makeを使用した並列処理の方法について学ぶ

# ファイルシステムと バッチジョブ操作

# 利用可能なファイルシステム

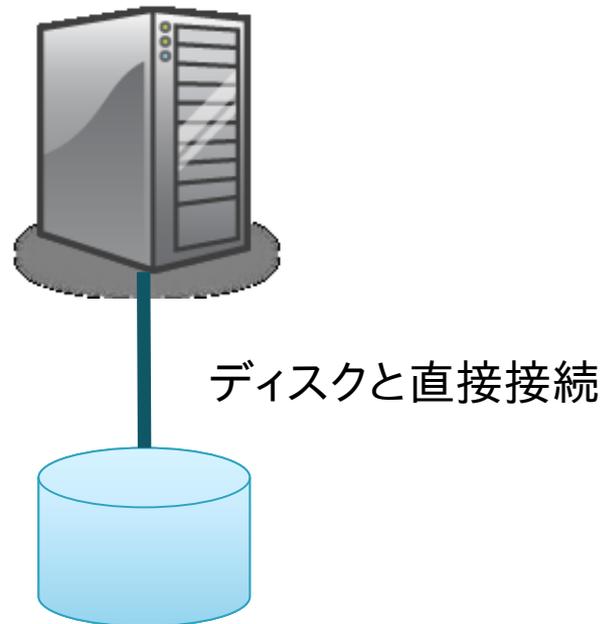
- ▶ Oakleaf-FXで利用可能なファイルシステムは以下のとおりである

PATH	種類	共有/非共有	備考
/home/ログイン名	FEFS	共有	共有ファイルシステム
/group[1-3]/グループ名 /ログイン名(*)	FEFS	共有	
/mppx[bc]/ログイン名(*)	FEFS	共有	外部ファイルシステム (一時利用)
/work	FEFS	-	ローカルファイルシステム (ステージング用)
/tmp	NFS	非共有	

(\*) 負荷分散のため、グループ、ユーザ毎に/group[1-3], /mppx[bc] のいずれかを使用

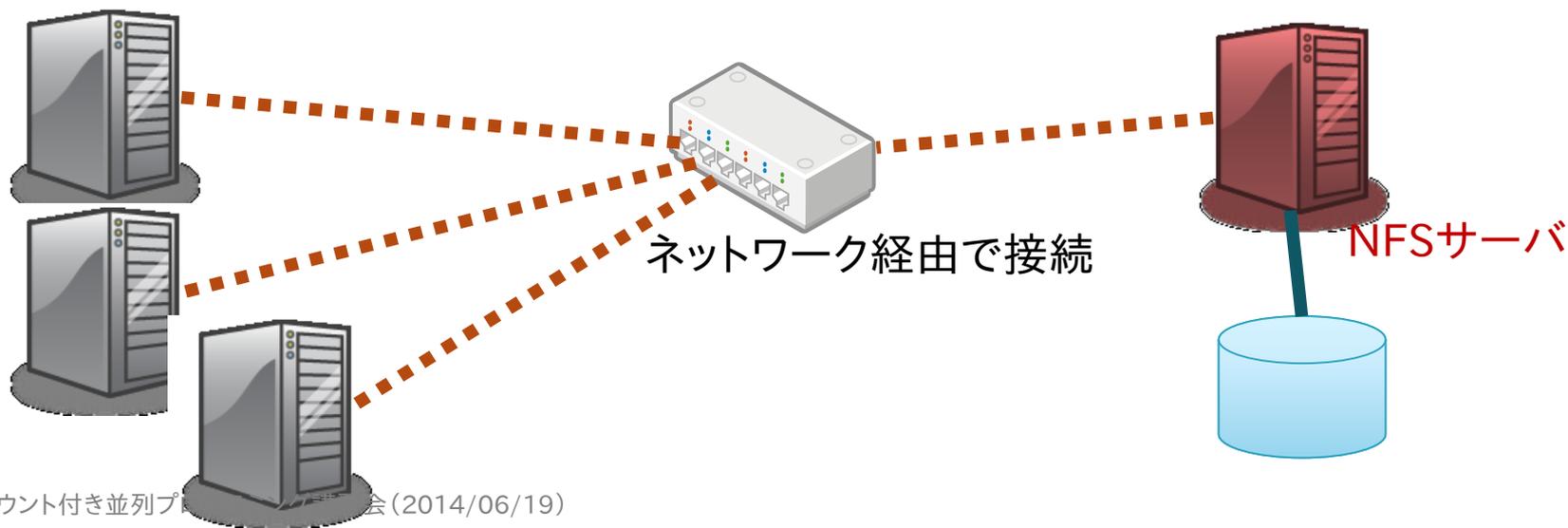
# ローカルディスク

- ▶ 他のノードから直接アクセスできない記憶域
  - Oakleaf-FXでは、計算ノードとインタラクティブノードにはローカルディスクはない



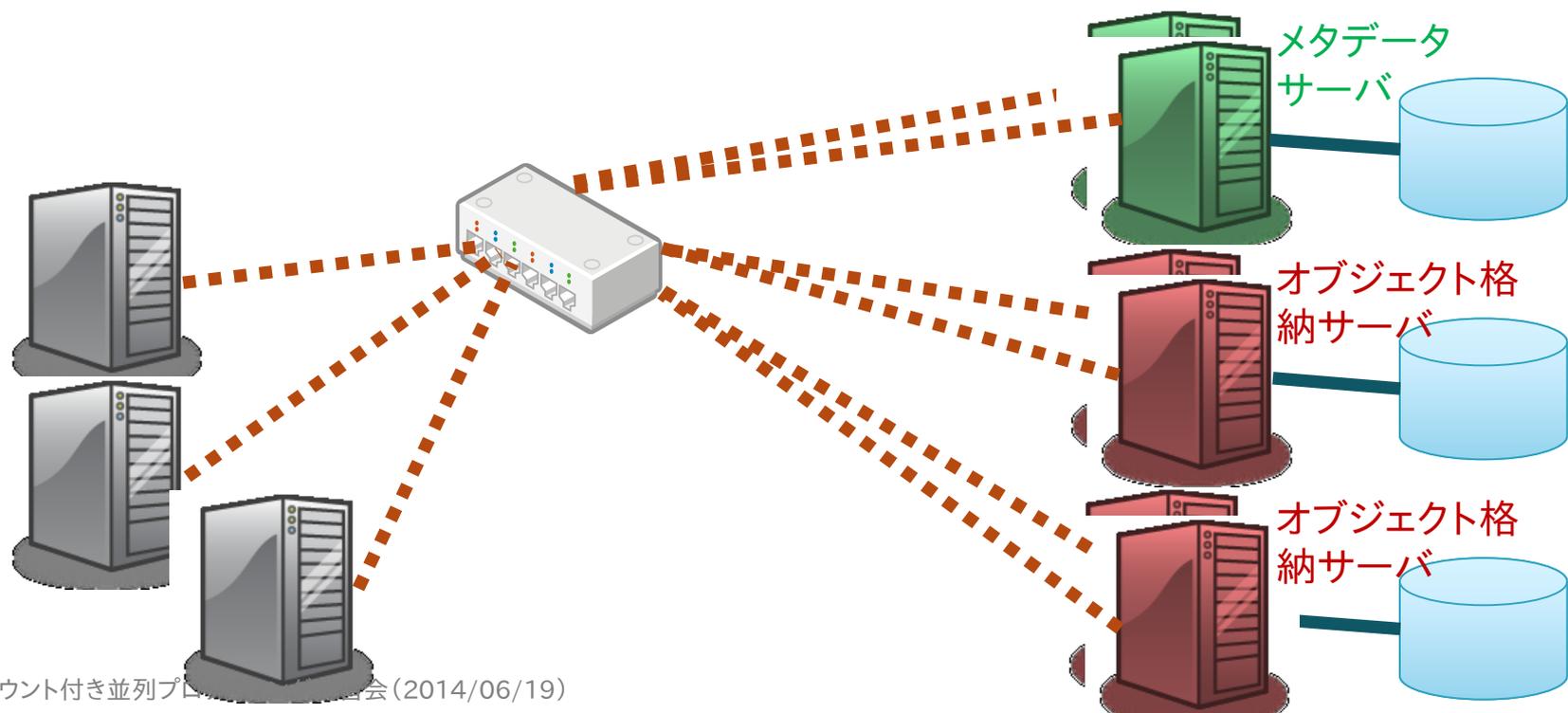
# NFS

- ▶ 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
- ▶ 動的な負荷分散機能がない(サーバは1台)
- ▶ Oakleaf-FXにおける設定
  - OS起動等のために、1ラック(96ノード)ごとに1台使用
  - /tmp (NFS領域) には書き込みを行わないことを推奨



# 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(lfs setstripeなど)

# Oakleaf-FXの分散ファイルシステム

## ▶ 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
- 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、最大ファイル数等の拡張

## ▶ 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)

# 利用可能な容量(quota)

- ▶ 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
- ▶ 残り容量の確認コマンド (show\_quota)

```
$ show_quota
```

```
Disk quotas for user t00004
```

Directory	used(MB)	limit(MB)	nfiles
/home/t00004	194,229	204,800	338,682
/group/gc26/t00004	0	-	90,915
/group/gv52/t00004	101,520	-	221,398
/group/gv56/t00004	0	-	0
/mppxb/t00004	202	-	34,153

```
Disk quotas for group gc26 gv52 gv56
```

Directory	used(MB)	limit(MB)	nfiles
/group/gc26	8,732,088	57,344,000	971,278
/group/gv52	131,701	8,192,000	273,851
/group/gv56	1,481,982	8,192,000	233,079

# 課題1

- ▶ それぞれのファイルシステムでファイル展開コマンドを実行せよ
  - 実行時間にどのような差があるか？
- ▶ 各ファイルシステムに割り当てられたquota値を確認せよ



# 課題で使用するファイル

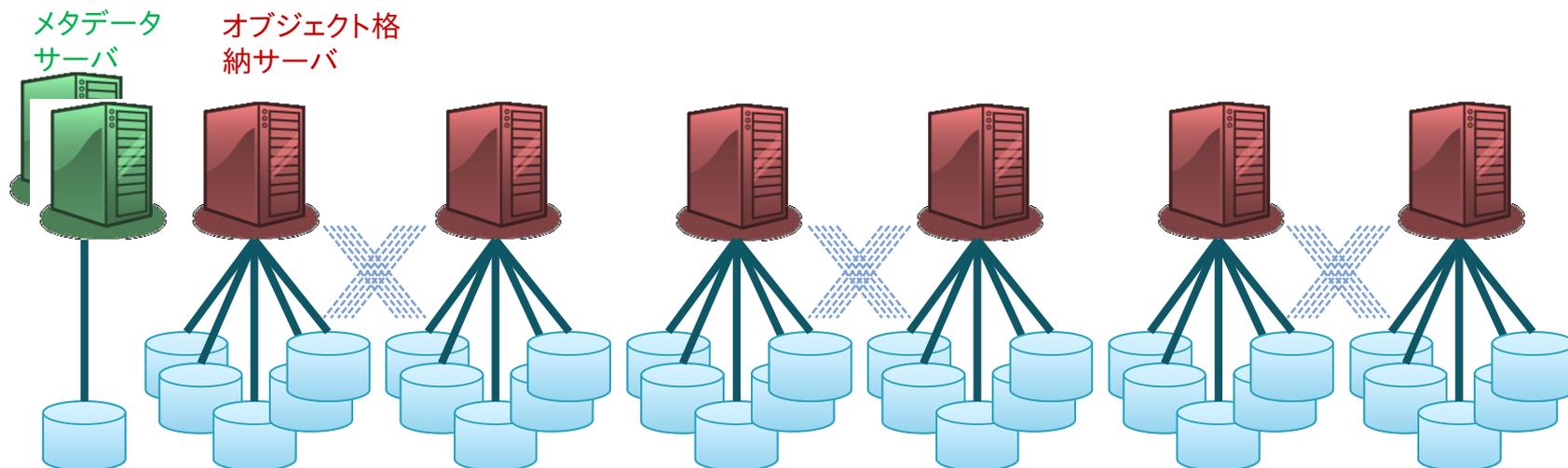
- ▶ `/home/t00001/public` に、この講習会で使用したプログラム、課題の解答などのファイルを置きました。ご利用ください。

# 解説1

- ▶ `./create_tar.sh` ファイル数
  - 指定された数のファイルが格納されたtar.gzファイルを作成
- ▶ ジョブスクリプト `extract_test.sh` (pjsubで実行する)
  - `extract`関数 (`extract` ディレクトリ)
    - 指定したディレクトリにファイルを展開し、所要時間を表示
    - `PJM_O_LOGNAME`という環境変数に、ユーザ名が格納されている
    - `-l "test.tar.gz …"`はステージングのための指定(後述)
- ▶ ファイルシステムのquotaを確認するには、`show_quota`コマンドを使用する

# Lustre/FEFSのデータ配置

- 複数のOST (Object Storage Target: 仮想的なディスク) で構成
- 各OSTは1つのRAIDグループに対応
  - 共有: RAID6 (9D+2P) x 480、ローカル: RAID5 (4D+1P) x 600、外部: RAID6 (8D+2P) x 236
- メタデータの格納先 (MDT: Metadata Target)はRAID1

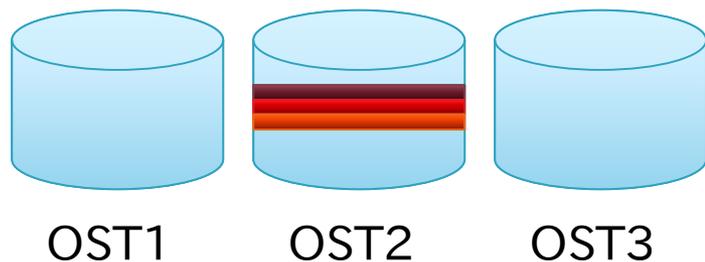


# 参考：Lustreのデータ配置の指定

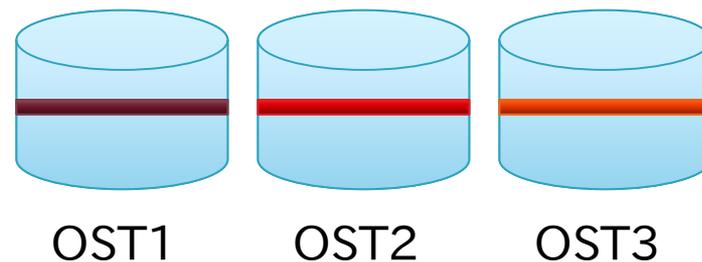
## ▶ データ配置の指定

- 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
- デフォルトではひとつのOSTに配置
- `lfs getstripe / lfs setstripe`コマンドで参照・変更可能

ひとつのOSTに配置



複数のOSTに配置



# 参考: Lustreのデータ配置の指定(例)

- ▶ `lfs setstripe -s size -c count` ファイル名
  - *size* 毎に *count* 個のOSTに渡ってデータを分散配置する設定にした空のファイルを作成する

(`lustre_stripe`ディレクトリに、ここで使用したスクリプトがあります)

```
$ dd if=/dev/zero of=/mppxc/t00004/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 35.6352 s, 121 MB/s
```

OST数が1の場合の書き込み性能

```
$ rm /mppxc/t00004/4G.dat
```

```
$ lfs setstripe -s 1M -c 50 /mppxc/t00004/4G.dat
```

ストライプ設定の変更(50個のOSTにデータを分散)

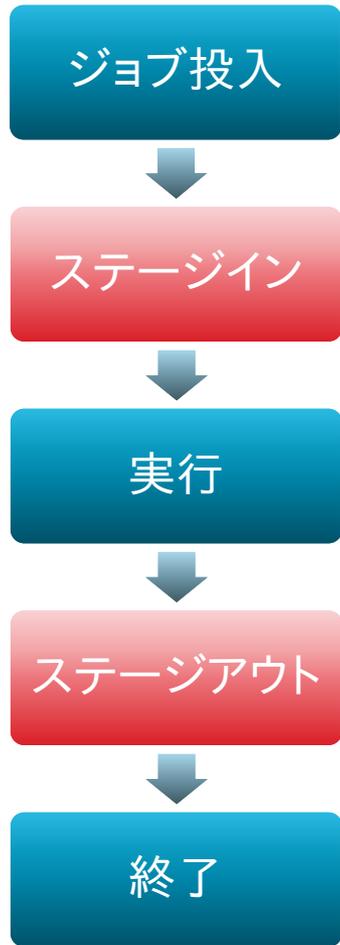
```
$ dd if=/dev/zero of=/mppxc/t00004/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 17.6508 s, 243 MB/s
```

OST数が50の場合の書き込み性能

# バッチジョブの操作（上級編）

1. ステージング
2. ジョブの詳細な情報の把握
3. コマンドラインオプションの利用
4. ステップジョブ

# 1. ステージング



- ▶ 共有ファイルシステムとローカルファイルシステムの間で、ジョブの入力ファイル、出力ファイルを転送する手法
- ▶ ジョブが利用するファイルシステムをローカルファイルシステムにすることで、入出力の競合を減らすことが可能
  - 多くのI/O処理を行う大規模なプログラムにて大きな効果を発揮する
- ▶ **ステージイン**
  - 入力ファイルなどをローカルファイルシステムに転送
- ▶ **ステージアウト**
  - 出力結果などを共有ファイルシステムに転送

# ジョブ実行時ディレクトリ

- ▶ ステージングを利用する場合、ジョブの実行時ディレクトリは投入時ディレクトリとは異なるディレクトリになる

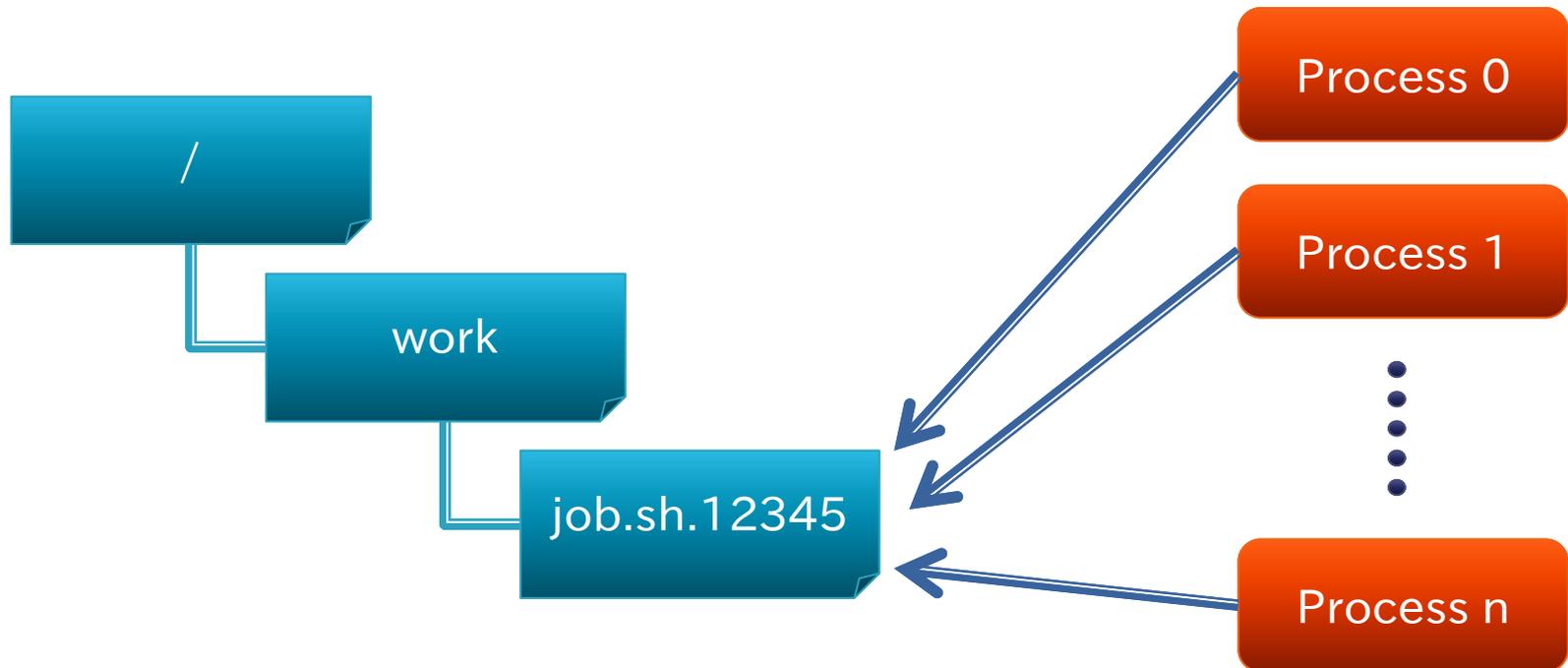
	ジョブ実行時ディレクトリ	\$PJM_O_WORKDIR	\$PJM_JOBDIR
非ステージングジョブ	A	A	A
ステージングジョブ (共有モデル)	B	A	B
ステージングジョブ (非共有モデル)	B/ <i>rank</i>	A	B

- ▶ A: ジョブ投入時ディレクトリ
- ▶ B: `/work/jobname.jobid`
- ▶ 非ステージングジョブでは、`/work/jobname.jobid`というディレクトリは作成されない

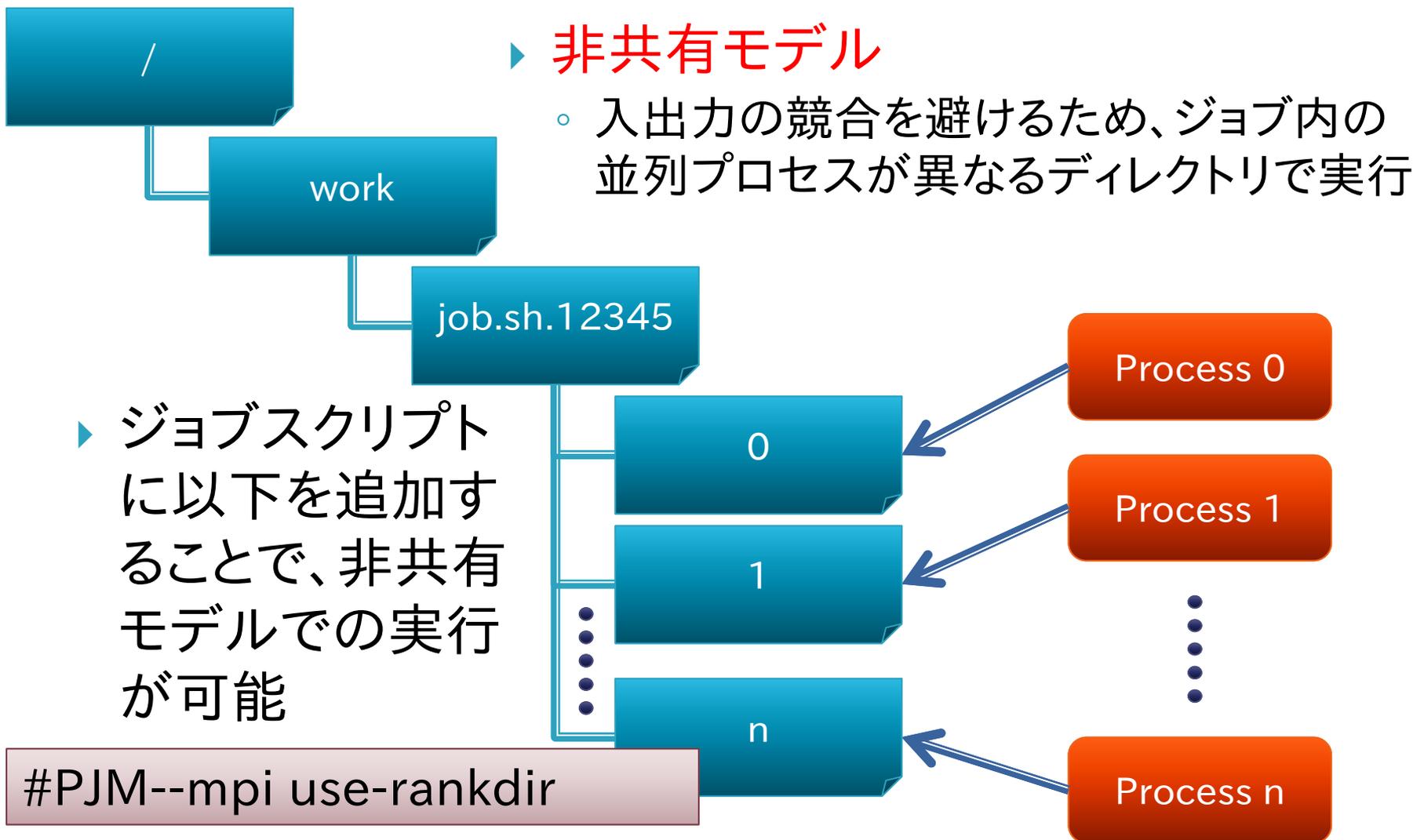
# 共有モデル

## ▶ 共有モデル

- ジョブ内の並列プロセスが同じファイルに対して入出力
- ジョブ内で同一ファイルを共有



# 非共有モデル



## ▶ 非共有モデル

- 入出力の競合を避けるため、ジョブ内の並列プロセスが異なるディレクトリで実行

- ▶ ジョブスクリプトに以下を追加することで、非共有モデルでの実行が可能

```
#PJM--mpi use-rankdir
```

# ステージインのオプション

- ▶ #PJM-I “*srcfile dstfile*” ← 「“」「”」も必要
  - *srcfile*を*dstfile*に名前変更してステージイン
- ▶ #PJM-I "*srcfile1 srcfile2 ... dstdir/*"
  - *srcfile\**を*dstdir*ディレクトリに(存在しなければ作成して)ステージイン
  - 最後の「/」も必須
- ▶ #PJM-D "*srcdir dstdir*"
  - *srcdir*以下のファイルを*dstdir*ディレクトリに(存在しなければ作成して)ステージイン
- ▶ *srcfile, srcdir*を相対パス指定したときはジョブ投入ディレクトリが起点になる

# ステージアウトのオプション

- ▶ #PJM-O "*srcfile dstfile*"
  - *srcfile*を*dstfile*に名前変更してステージアウト
- ▶ #PJM-O "*srcfile1 srcfile2 ... dstdir/*"
  - *srcfile\**を*dstdir*ディレクトリに(存在しなければ作成して)ステージアウト
  - 最後の「/」も必須
- ▶ #PJM-E "*srcdir dstdir*"
  - *srcdir*以下のファイルを*dstdir*ディレクトリに(存在しなければ作成して)ステージアウト
- ▶ *dstfile, dstdir*を相対パス指定したときはジョブ投入ディレクトリが起点になる

# 複雑なファイル名の指定

- ▶ 以下の表記を使用して、ステージングのファイル名にジョブID等を含めることが可能
- ▶ %j
  - ジョブID
- ▶ %n
  - ジョブ名
- ▶ %r
  - ランク番号(非共有モデル利用時)
  - %03rの様な指定も可能(rank=1の時、001等)

# ステー징ングジョブ用コマンド

- ▶ `pjstgchk`
  - ステーキング書式の文法チェック
- ▶ `pjcat [-e | -o] -f`
  - 標準出力・エラー出力の表示
  - `-f`は`tail -f`と同様の動作(継続して表示)
- ▶ `pjlist [-a] [-l] [-R] JOBID [rank]`
  - ジョブ実行時ディレクトリのファイルリストの表示
- ▶ `pjget [-f] [-p] [-r] JOBID [rank:] src dst`
  - ジョブ実行時ディレクトリ上のファイルをコピー
  - `-f`は既存ファイルを削除してからコピー、`-p`, `-r`は`cp`コマンドと同様

# ステージング実行例 1

- ▶ MPIプログラムを実行し、ログをジョブIDがついたディレクトリに保存

```
#PJM--mpi use-rankdir  
#PJM-I "a.out input.dat ./"  
#PJM-O "stderr.%r logs_%j/"  
#PJM-O "stdout.%r logs_%j/"
```

```
mpiexec --stdout-proc stdout ¥  
--stderr-proc stderr ./a.out input.dat
```

# ステージング実行例 2

- ▶ MPIランクごとに異なるファイル名のデータをステージイン

```
#PJM--mpi use-rankdir  
#PJM-I "program ./"  
#PJM-I "rank=0 master.dat ./"  
#PJM-I "rank=1- worker_%r.dat ./"
```

```
mpirexec ./program ...
```

- ▶ ランク番号は範囲で指定することができる
  - 書式:rank=N1-N2
    - N1省略時:0
    - N2省略時:MPIプロセス数-1

## 2. ジョブの詳細な状態の把握

### ▶ `pjstat -s` ジョブID

- ジョブの、より詳しい状態を確認するコマンド
- 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

### ▶ `pjstat -X` ジョブID

- 実行中のジョブのノード割り当て、ランク割り当てを確認するコマンド

- `pjstat`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 `man pjstat`や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              : -
```

# 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

# 3.コマンドラインオプションの利用

- ▶ `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`コマンドを使って確認すれば正しい情報が得られる

# 課題2

- ▶ `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キーで終了
- ▶ `pjstat -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ているか
  - `env` を `mpiexec env` に変更すると、どのような環境変数が設定されるか

# 解説2

- ▶ 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\_**で始まる環境変数が設定される
  - **PJM\_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\_**または**OMPI\_**で始まる環境変数が設定される

# 課題3

- ▶ 実行されたジョブのノード数を NODES 環境変数に、総プロセス数を PROCS 環境変数に設定するにはどうすればよいか？
  - ヒント
    - MPIプログラムを実行すれば上記の情報はわかる
    - `eval `echo x=1`` を実行すると、シェル変数 `x` に 1 が設定される



# 解説3

- ▶ 環境変数の説明
  - `FLIB_NUM_PROCESS_ON_NODE`
    - ノードあたりのMPIプロセス数(の最大値)
  - `OMPI_UNIVERSE_SIZE`
    - 上記の値とノード数の積
  - `OMPI_MCA_orte_ess_num_procs`
    - MPIプロセス数
- ▶ バッチジョブ内とインタラクティブジョブ内では、`mpiexec`実行時のリダイレクトやバッククォートの動作が異なるので注意すること

## 4. ステップジョブ

- ▶ 複数のジョブの間で実行の順序関係や依存関係を指定可能
- ▶ ステップ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

```
pjsub --step --sparam "sn=1" step1.sh
[INFO]PJM 0000 pjsub Job 12345_1 submitted.
$ pjsub --step --sparam "jid=12345, sn=2, sd=ec!=0:after:1" step2.sh
[INFO]PJM 0000 pjsub Job 12345_2 submitted.
$ pjsub --step --sparam "jid=12345, sn=3, sd=ec!=0:all" step3.sh
[INFO]PJM 0000 pjsub Job 12345_3 submitted.
$ pjsub --step --sparam "jid=12345, sn=4, sd=ec==0:one:1" step4.sh
[INFO]PJM 0000 pjsub Job 12345_4 submitted.
```

# ステップジョブの実行例



## ▶ step2

- 依存するサブジョブ1の終了コードが0以外の場合( $ec \neq 0$ )、自分と自分に依存するサブジョブすべてを(after)削除

## ▶ step3

- 依存するサブジョブ2(省略時は直前のサブジョブ)の終了コードが0以外の場合( $ec \neq 0$ )、依存関係にかかわらず後続のサブジョブすべてを削除(all)

## ▶ step4

- 依存するサブジョブ1の終了コードが0の場合( $ec == 0$ )、自分を削除(one)

`pjstat -E`で、サブジョブを展開して表示可能  
終了したジョブの場合は、`pjstat -H -E`

# makeの利用

# make

- ▶ 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
- ▶ 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
- ▶ Oakleaf-FXだけではなく、一般的なLinux環境の多くで利用可能
  
- ▶ この講習会では GNU make (version 3.81)を使用する

# 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を入れる

## ▶ 実行

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

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 ...  
                  コマンド  
                  ...

- ▶ 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行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
```

## ▶ 実行

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

## ▶ hello.cを書き換え

例: world! を world!! に  
書き換え

## ▶ makeを再実行

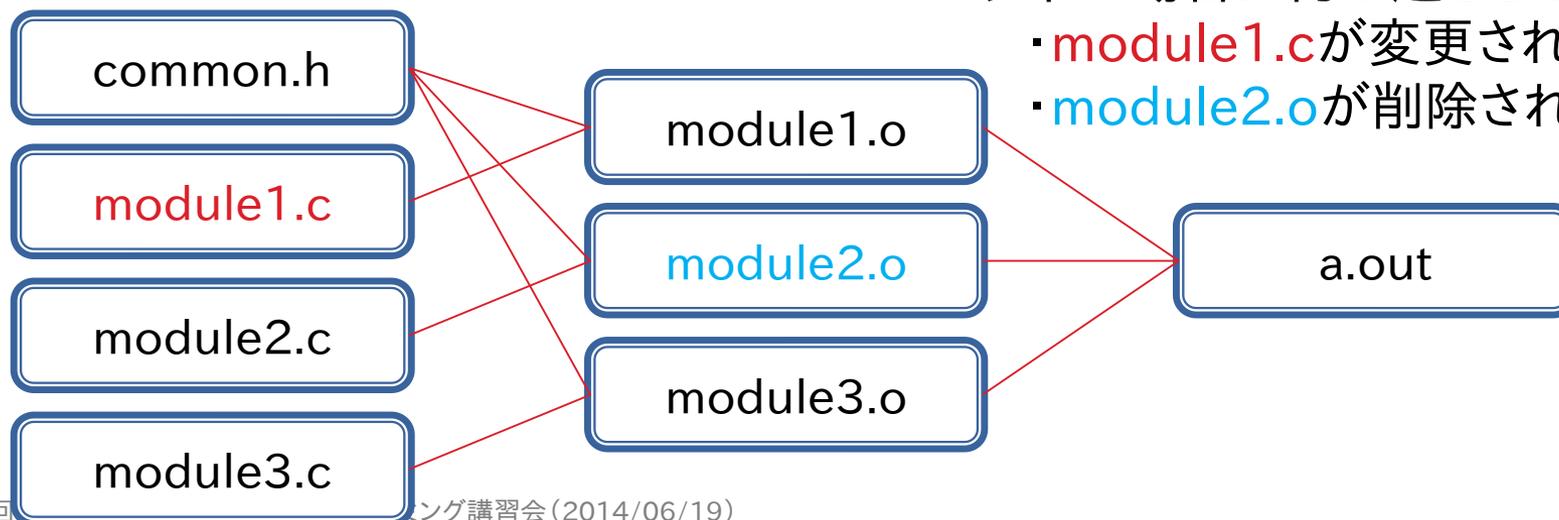
```
$ make
gcc -c hello.c
gcc -o hello hello.o main.o
```

# 分割コンパイル

- ▶ 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の指定

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

## ▶ 長い行

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

## ▶ PHONYターゲット

```
.PHONY: clean  
clean:                                     (cleanというファイルがあっても実行する)  
      rm -f hello hello.o main.o
```

## ▶ 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)

# 課題4

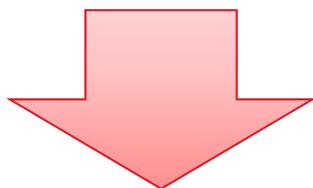
- ▶ コマンドの前のタブを、スペースにした場合、どのようなエラーが出力されるか
- ▶ .PHONY: X があるときとない時で、make X の動作に違いがあることを確認せよ

# 解説4

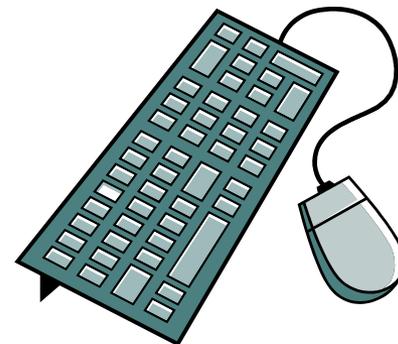
- ▶ 誤ってタブを、スペースにした場合、“missing separator” というメッセージが表示される
  - 環境設定などにより異なる場合もある
  - より親切な警告メッセージが表示される場合もある

# 高度なMakefileの書き方

- ▶ 変数、関数の使用・特別なルールの書き方



- ▶ Makefileのより簡潔な記述
- ▶ より柔軟な出力やエラー制御



# 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  
                $OBJECTSとすると、$(OBJECTS)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=-Iidir1 -Iidir2
```

CFLAGSは -Iidir1 -Iidir2 -O -gに展開される

# makeの動作の制御

- ▶ 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

## ▶ 利用可能なディレクティブ

- ifeq, ifneq, ifdef, ifndef

# 関数

## ▶ 変数と似た参照方法で利用可能

`VALUE=$(subst xx,yy,aaxxbb)`      VALUEにaayybbが代入される

`CONTENTS=$(shell cat data.txt)`      CONTENTSにはdata.txtの中身が代入される

`SECOND=$(word 2, This is a pen)`      SECOND=isと同じ

`CDR=$(wordlist 2,$(words $(LIST)), $(LIST))`  
CDRには\$LISTの2番目以降の単語のリストが代入される

## ▶ 他の関数の例

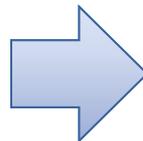
- `dir, notdir`: シェルのdirname, basenameに似た動作
- `suffix, basename`: 拡張子とそれ以外の部分に分ける
  - シェルのbasenameとは違う
- `wildcard`: ワイルドカードを展開

# 特殊な変数

- ▶ 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

<code>\$@</code>	ターゲット名
<code>\$&lt;</code>	最初の依存ファイル
<code>\$?</code>	ターゲットより新しい依存ファイル
<code>\$+</code>	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

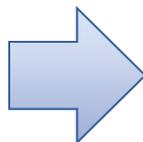
- ▶ 指定したパターンにマッチしたらコマンドを実行する

```
%.o : %.c
```

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

- `***.o` は `***.c` に依存する

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

# 変数の評価タイミング

```
DATE1 = $(shell date)
```

```
DATE2 := $(shell date)
```

```
DATE3 = `date`
```

## ▶ DATE1

- 参照されるたびにdateが実行される(毎回再評価する)
- 実行されるタイミングは最初(アクションが実行される前)

## ▶ DATE2

- (参照されなくても)1度だけdateが実行される(右辺が変わっていなければ再評価しない)
- 実行されるタイミングは最初(アクションが実行される前)

## ▶ DATE3

- 最初は`date`という文字列が展開されるだけ
- dateが実行されるのは各アクションが実行されるとき

# 課題5

- ▶ 以下のルールDATE1をDATE2,DATE3に変更して実行せよ。2つechoの出力に違いはあるか？

```
test:
    echo $(DATE1)
    sleep 1
    echo $(DATE1)
```

- ▶ DATE1,DATE2は、一見すると出力が同じであるが、どうすれば動作の違いを説明できるか？
- ▶ **DATE4 := `date`**
  - はどれと同じ動作になるか

# 解説5

- ▶ dateコマンドだけでは明確な違いがわかりにくいいため、以下のようにして情報を取得
  - より細かい単位で表示 (+%N でナノ秒単位の表示)
  - date実行時にログを出力する
  - 実行開始時刻を付加する、時刻表示前にsleepする
- ▶ lab5/2
  - date1, date2: 「最初」に時刻を変数に代入している、実行するコマンドを表示する時点で展開されている
  - date3, date4: 実行時に代入
  - date2: 二度目の実行時に再度実行「しない」
  - date3 = date4

# 課題6

- ▶ wildcard関数を使用して以下の処理を行うMakefileを記述せよ
  - 入力データの中から、2009年8月と9月のデータだけ进行处理する
- ▶ 入出力データの仕様
  - 入力ファイル名に日付が含まれている(YYYYMMDD.in)
  - 出力データは拡張子を.inから.outに変え、内容をコピーする

# 解説6

- ▶ `test: *.out`から、`%.out: %.in`というルールが適用される
- ▶ `*.out: *.in` のアクションとして、`cp *.in` が実行される
  - sh(bash)の場合、`*.in`は存在する複数のファイルに展開され、`*.out`は存在しないため、展開はされず、`*.out`のままとなる
  - 結果として、複数のファイルを '`*.out`' というディレクトリにコピーするコマンドとなり、失敗する
- ▶ Wildcard関数を使用した例(test3)では、正しいOUTFILELISTが生成されるため、入力ファイルの数と同数のcpコマンドが実行され、正しい結果が得られる

# makeの応用 (makeを使った並列処理、 GXP/GXP makeの利用)

# 並列処理への応用

- ▶ 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を使う上での便利な点
  - 実行するコマンドの依存関係を簡単に記述可能
  - 簡単な並列化
    - 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

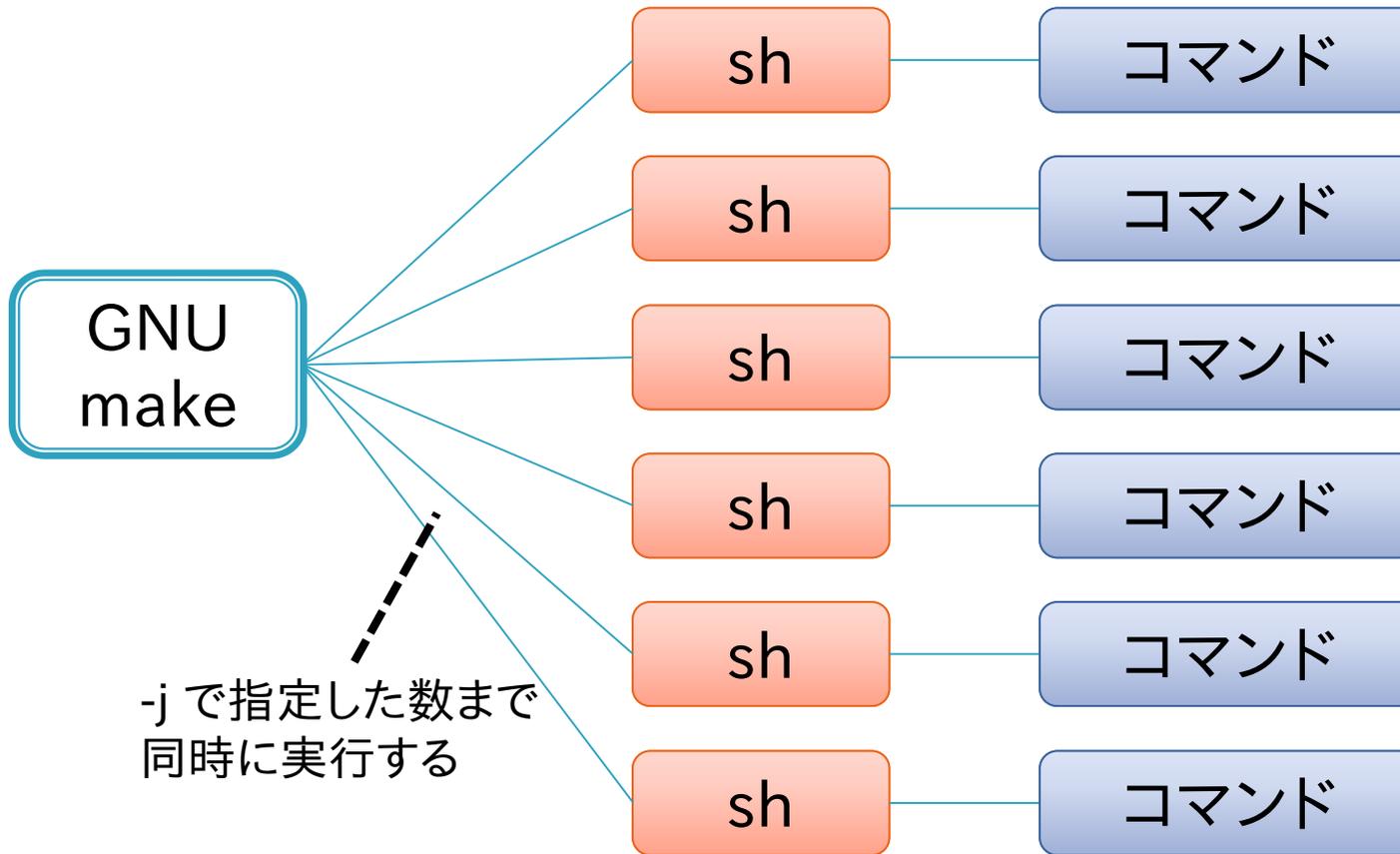
```
all: a b
```

```
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
- 最大並列度の最大値は 4096 (RHEL6における制約)
  - 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( $\infty$ )

## ▶ 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

# 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
```

- ▶ 本来は依存関係を明示する必要がある

(wrong\_makefiles/wrong1.mak に、ここで使用した Makefile があります)

# 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がある

```
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

(wrong\_makefiles/wrong2.mak に、ここで使用した Makefile があります)

# 課題7 (1ノードの例)

## ▶ Makefile

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

```
all: $(FILES)
```

```
%.dat:
```

```
    sleep 5
    touch $@
```

- 変数や%を使わない場合どのようなMakefileになるか
- make と make -j の実行時間を比較せよ

# 解説7

- ▶ test.mkに特殊変数展開前、test2.mkに展開後のファイルを用意した
- ▶ `time make -j 数値`とすれば並列度を変更して計測できる
- ▶ 様々な並列度で試してみていただきたい

# 複数ノードで並列make

- ▶ Oakleaf-FX の場合、1 ノードで使える CPU コア数は 16 まで
- ▶ 多数のノードを使用すれば、よりたくさんの処理を行うことが可能
- ▶ make は複数ノードを使った処理に対応していない
- ▶ **GXP make** を使用すると複数ノードで並列make を実行可能
  - GXP make は並列シェル GXP と一緒に配布されているソフトウェア
  - Make の処理を、マスターワーカー型の並列処理として複数ノードで実行可能
  - 各ノードでファイルが共有されていることが前提

# GXP

- ▶ 並列分散環境を簡単に扱うための、並列版シェル
  - 多数のノードのインタラクティブな利用
  - 並列ワークフローの実行 (GXP make)

- ▶ 詳しい情報

<http://www.logos.t.u-tokyo.ac.jp/gxp>

<http://sourceforge.net/projects/gxp>

- ▶ ダウンロード方法

```
$ cvs -d ¥  
:pserver:anonymous@gxp.cvs.sourceforge.net:/cvsroot/gxp ¥  
co gxp3
```

※cvsで入手したものにパスを通せばすぐに使えます

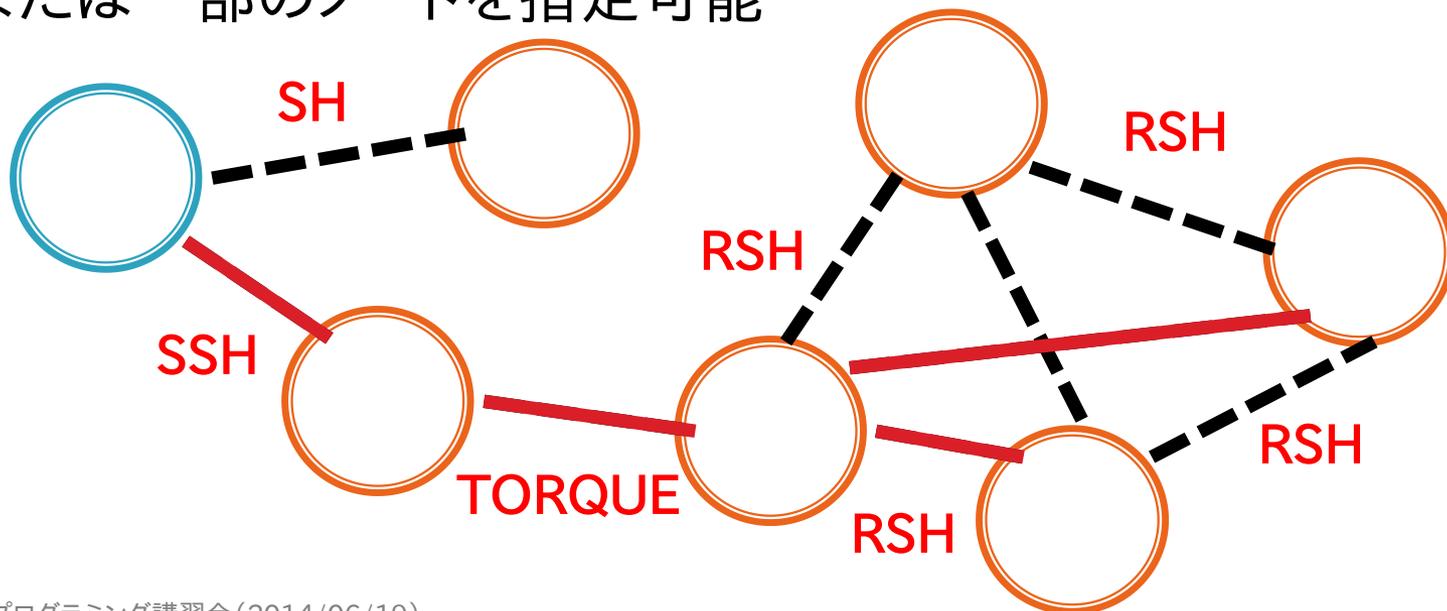
- ▶ Oakleaf-FX 上のインストール先

`/home/t00001/public/gxp3`

# GXPの動作

- ▶ 各計算ノードでデーモンプロセス(GXPD)を起動
  - ノード集合と、GXPD の起動方法を指定 (use)
    - SSH, PBS, GridEngine 等が利用可能。拡張も可能
  - ノード集合を指定して、GXPD を起動 (explore)
- ▶ e(execute) コマンドでユーザプロセスを起動
  - 全部または一部のノードを指定可能

use  
explore  
execute



# バッチジョブ内でGXPを使用する

```
#PJM-L node=4  
#PJM--mpi proc=4
```

```
NODES=ノード数
```

ノード数を取得

```
gxpc --root_target_name head  
gxpc rsh YYY XXX %target% %cmd%  
gxpc use YYY head node  
gxpc explore node[[1--$NODES]]
```

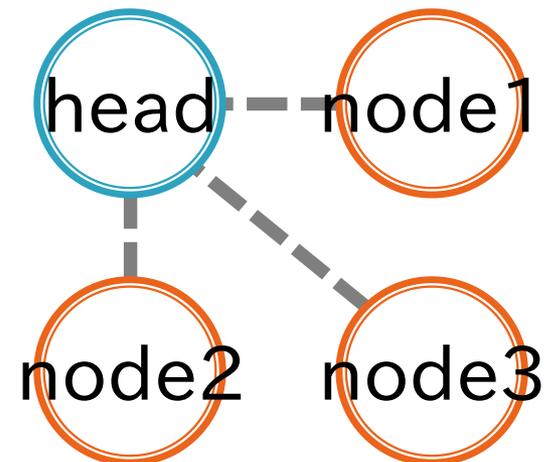
各ノードでGXPDを起動

```
gxpc cd `pwd`  
gxpc e 'echo $GXPC_EXEC_IDX `hostname`'
```

コマンドの実行

```
gxpc quit
```

XXX は、各ノードでプロセスを起動する  
ためのrsh-likeなコマンド



# Oakleaf-FX上でGXPを使用する

- ▶ Oakleaf-FX では、プロセス起動に rsh 等を使用できないため、MPI プロセス経由で起動する

```
gxpc --root_target_name head
```

```
gxpc rsh mpi_redirect redirect_client %target% %cmd%
```

```
gxpc use mpi_redirect head node
```

```
mpiexec redirect_server &
```

← redirect\_server をバックグラウンドで起動

```
NODES=`redirect_client getsize`
```

← ノード数を取得

```
gxpc explore node[[1--$NODES]]
```

```
gxpc cd `pwd`
```

```
gxpc e 'echo $GXPC_EXEC_IDX `hostname`'
```

```
gxpc quit
```

```
redirect_client shutdown
```

← redirect\_server を終了

```
wait
```

# より簡単な方法

- ▶ 用意された、初期化からExploreするところまでを実行するスクリプト、終了処理を実行するスクリプトを使用すれば、より簡単に記述可能

```
#PJM -L node=4
```

```
. /home/t00001/public/fx10_gxp/gxp_init.sh
```

```
gxpc cd `pwd`
```

```
gxpc e 'echo $GXPC_EXEC_IDX `hostname`'
```

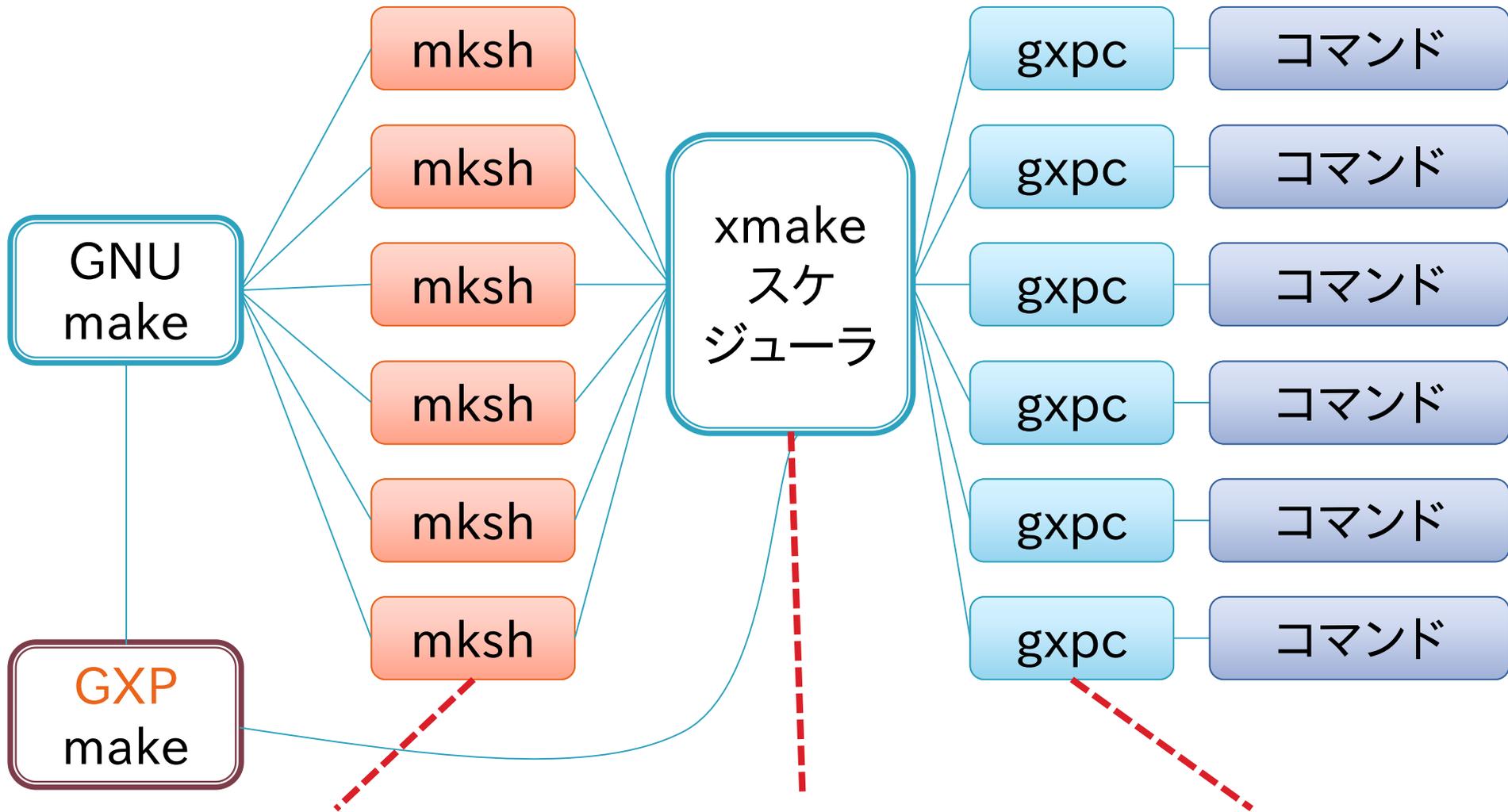
```
. /home/t00001/public/fx10_gxp/gxp_finalize.sh
```

# GXP make

- ▶ makeで実行される各コマンドをGXP経由で実行
  - -jオプションと組み合わせて、ノードにまたがってmakeを並列実行することができる
  - 各ノードでファイルが共有されている必要がある
- ▶ gxpc make …
  - … には、GNU makeに渡すことができるすべてのオプションを渡すことができる
- ▶ Oakleaf-FXでのGXP makeの実行
  - CPU数の自動取得に失敗するため、作業ディレクトリ上に以下の内容でgxp\_js.confというファイルを作成

cpu 16

# GXP makeの動作の仕組み



直接コマンドを実行せずスケジューラに登録するだけ

どのノードでコマンドを実行するか決める

指定されたノードでコマンドを実行する

# GXP make サンプルスクリプト

```
#PJM-L node=4  
#PJM--mpi proc=4
```

```
./home/t00001/public/fx10_gxp/gxp_init.sh
```

GXPDの起動

```
gxfc cd `pwd`  
gxfc make -j 64
```

並列makeの実行

```
./home/t00001/public/fx10_gxp/gxp_finalize.sh
```

GXPDの終了

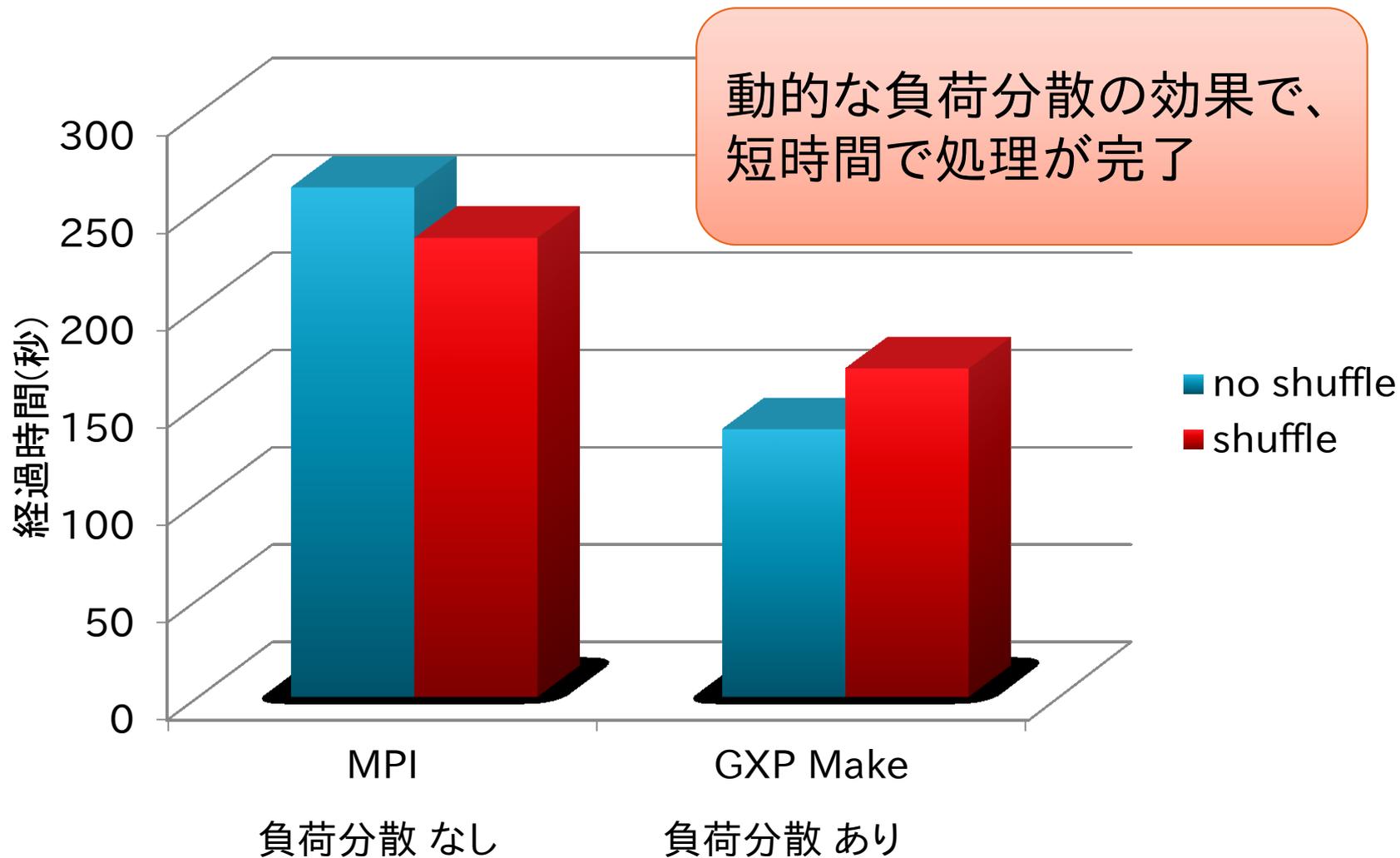
# 課題8

- ▶ 以下に述べる並列処理を実行せよ
- ▶ 処理の内容
  - 複数の入力ファイルがある(in/inpXX-Y.dat)
  - 入力ファイルごとに、その内容に従って「処理」を行い、1つの出力ファイルを生成する(out/outXX-Y.dat)
    - 入力ファイルの内容により、処理時間は異なる
  - それぞれのタスクは独立で、並列実行可能
- ▶ 以下のそれぞれの場合を実際に試して、実行時間の違いの理由を考えよ
  - 処理するファイルをプロセスごとに固定する場合(MPI)
  - マスターワーカー型の負荷分散を行う場合(GXP make)

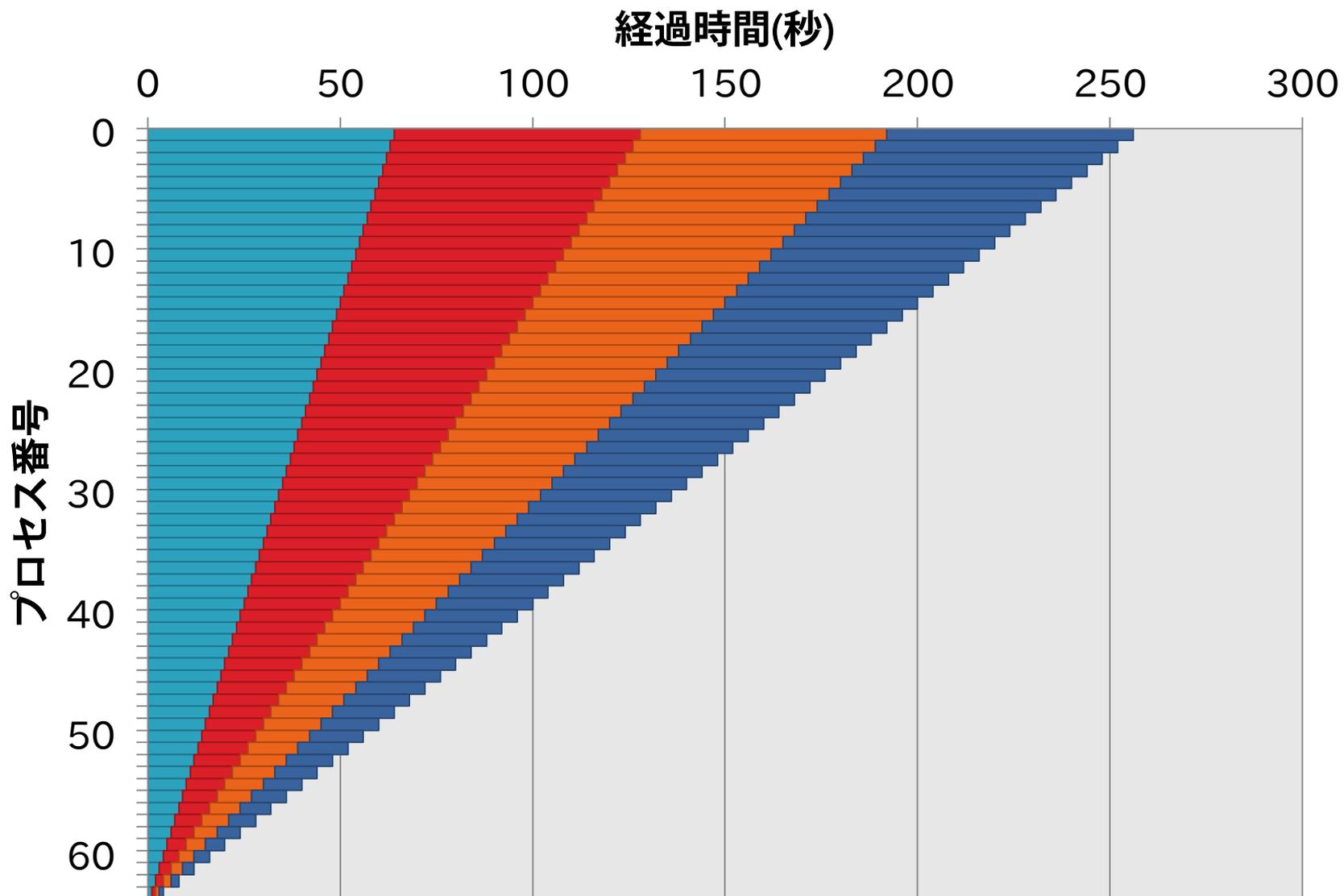
# 解説8

- ▶ サンプルプログラムの説明
  - nolb.cがMPI版
    - make nolbでコンパイル可能
    - pjsub.shがジョブスクリプト
  - GXP make版のジョブスクリプトはpjsub\_gxp.sh
  - make infiles で入力ファイルを作成
  - ./shuffle.sh で入力ファイルの内容をシャッフル

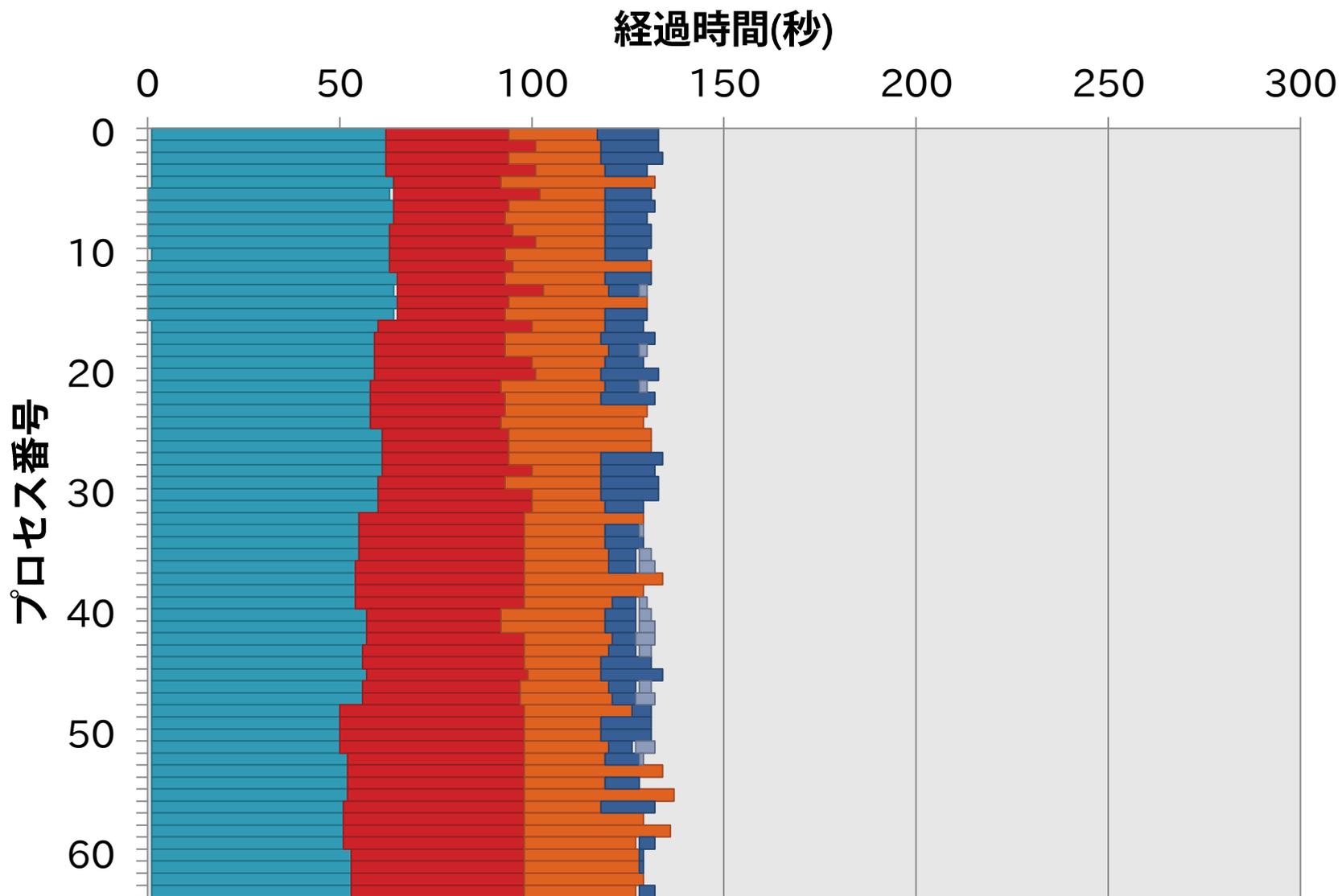
# 実行時間の比較



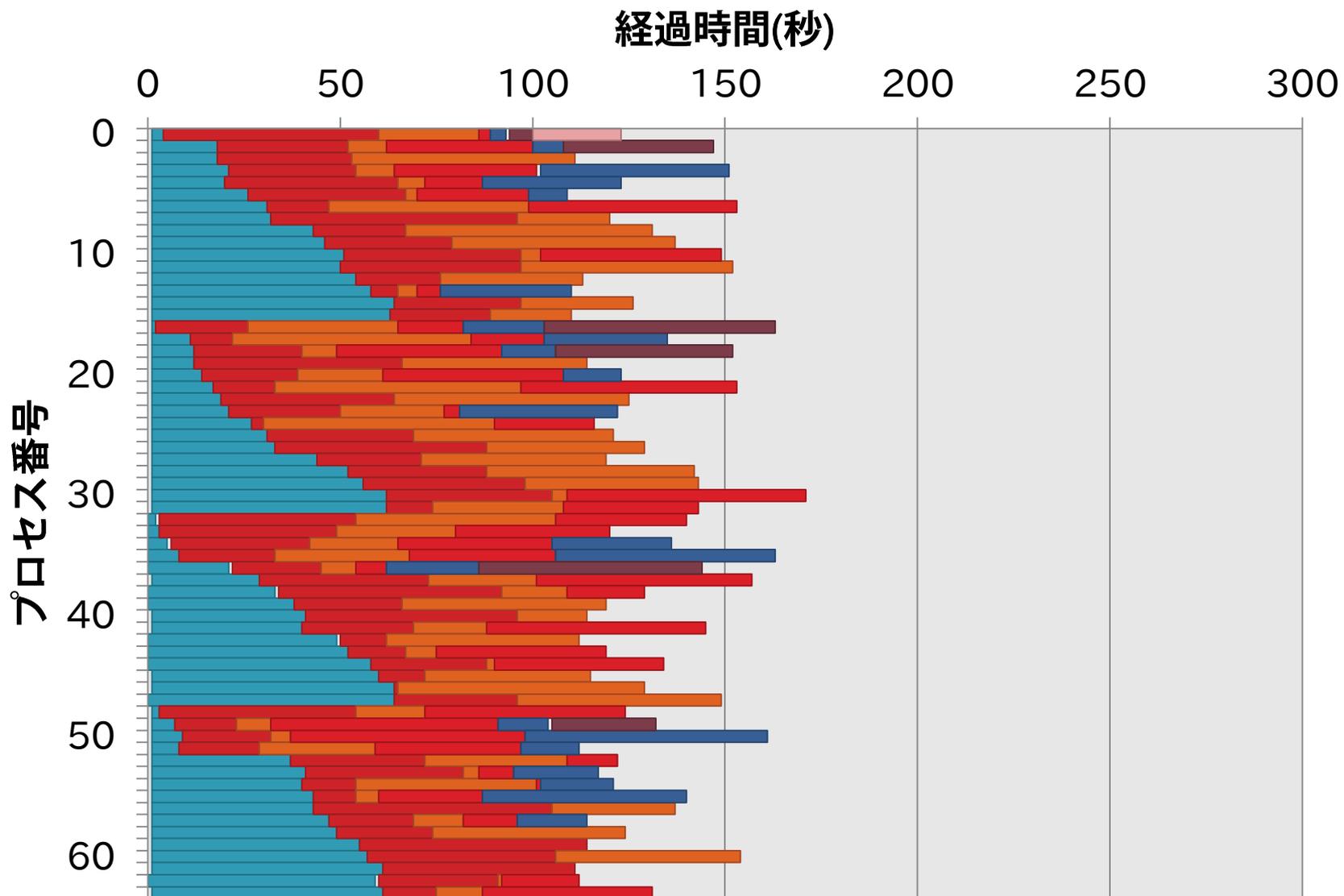
# 負荷分散を行わない場合



# 負荷分散を行った場合



# 負荷分散あり・シャッフルあり



# パラメタ並列処理(1/2)

- ▶ 容易にパラメタ並列処理を記述可能
  - GXPが提供する、パラメタ並列用のMakefileをincludeする
- ▶ 使用方法
  - parameters, target, output, cmd 変数を定義する
  - output, cmdは、:=ではなく=で値を定義する
    - これをテンプレートとして何度も展開される
  - \$(GXP\_MAKE\_PP)をinclude文で読み込む
    - (GXPインストール先)/gxpmake/gxp\_make\_pp\_inc.mk

# パラメタ並列処理(2/2)

- ▶ 例1: (2 \* 3 \* 4=24個のタスクを並列実行)
  - 以下のMakefileを書いて、`gxpc make -j baz` を実行する

```
parameters:=a b c
a:=1 2
b:=3 4 5
c:=6 7 8 9
target:=baz
output=hoge.$(a).$(b).$(c)
cmd=expr $(a) + $(b) + $(c) > hoge.$(a).$(b).$(c)
include $(GXP_MAKE_PP)
```

- ▶ 例2: (課題8の処理)
  - 複数のパラメタ並列処理の組み合わせも可能

# MapReduce

## ▶ MapReduceモデル

- Googleが提案する、大規模データの並列処理に特化したプログラミングモデル
- 1レコードに対する処理を書くと、処理系が大規模データに並列適用
- **入力データ**は、レコードの集合
- プログラムは、以下の2つの処理を定義
  - **Map**: レコード $\rightarrow$ (key, value)の集合
  - **Reduce**: (key, value)の集合 $\rightarrow$ 出力
  - 異なるレコードに対するmap処理と、異なるkeyに対するreduce処理が並列実行可能

# GXPのMapReduce機能

- ▶ GXP make上に構築されたMapReduce処理系
  - パラメタ並列と同様に、GXP が提供する Makefile を include するだけで利用可能
  - GXP が動く環境ならどこでも動く
- ▶ カスタマイズが容易
  - Makefile と、mapper, reducer などのいくつかの小さなスクリプトを書くだけ

# GXP MapReduceを制御する変数

- ▶ include \$(GXP\_MAKE\_MAPRED)の前に、以下の変数を設定する
  - input=入力ファイル名
  - output=出力ファイル名
  - mapper=mapperコマンド(ex\_word\_count\_mapper)
  - reducer=reducerコマンド(ex\_count\_reducer)
  - n\_mappers=map ワーカー数(3)
  - n\_reducers=reduce ワーカー数(2)
  - int\_dir=中間ファイル用ディレクトリ名
    - 省略時は\$(output)\_int\_dir
  - keep\_intermediates=yの時、中間ファイルを消さない
  - small\_step=yの時、細かいステップでの実行

# GXP MapReduceの使用例

- ▶ 例(word count)
  - Mapper: レコード→(単語1, 1),(単語2, 1),...
  - Reducer: それぞれのkeyについてvalueの和を出力
  - 以下のMakefileを書いて、`gxpc make -j bar` を実行する

```
input:=foo
output:=bar
mapper:=ex_word_count_mapper
reducer:=ex_count_reducer
n_mappers:=5
n_reducers:=3
include $(GXP_MAKE_MAPRED)
```

- ▶ 複数のMapReduceやパラメタ並列処理を組み合わせることも可能

# まとめ

- ▶ ファイルシステムやジョブ管理システム
  - Oakleaf-FXに固有の情報を活用することで、より効率的なシステムの利用が可能
- ▶ 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
- ▶ 並列ワークフロー処理
  - `make -j`で並列にmake処理を実行可能
  - makeを拡張したGXP makeを利用することで、大規模な並列処理を実行可能