プログラミングの基礎（makeを使った分割コンパイルと並列処理）

2011年6月9日

鴨志田 良和 (東京大学情報基盤センター)

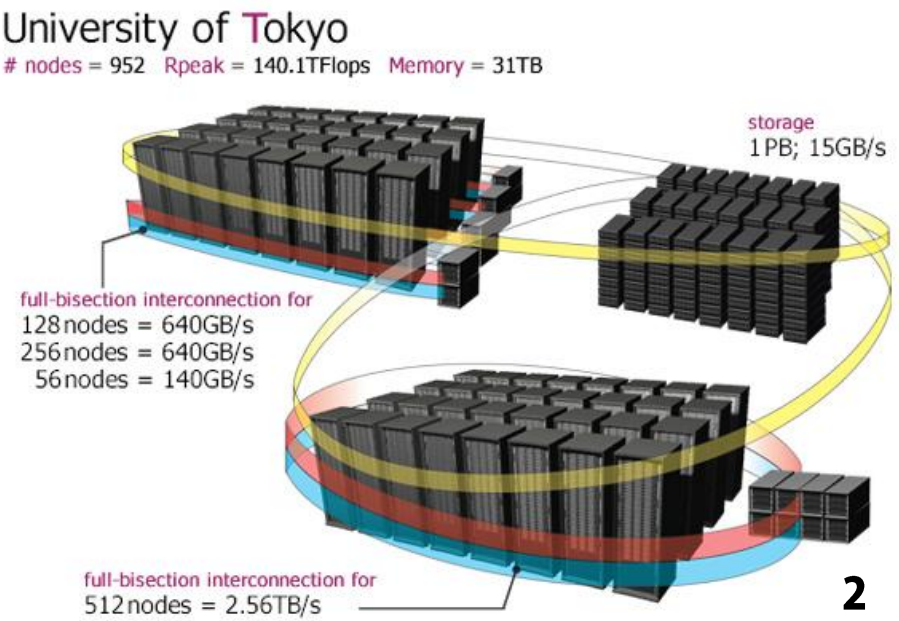
東京大学情報基盤センター
Information Technology Center, The University of Tokyo

目次

- 使用可能なファイルシステム
- バッチキューの操作(上級編)

- makeを使った分割コンパイル

- makeを使った並列処理



この講習の目的

- **HA8000クラスタシステム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キュー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キュー操作

利用可能なファイルシステム

- **HA8000クラスタシステム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
/home/ログイン名(*)	HSFS	共有
/short/ログイン名	HSFS	共有
/nfs/all/ログイン名(**)	NFS	共有
/lustre/ログイン名	Lustre	共有
/tmp	ローカルディスク	非共有

(*) 負荷分散のため、ユーザごとに/hsfs/home[1-4] のいずれかを使用
(**) 申込みにより、/nfs/personal/ログイン名、/nfs/グループ名/ログイン名も利用可能

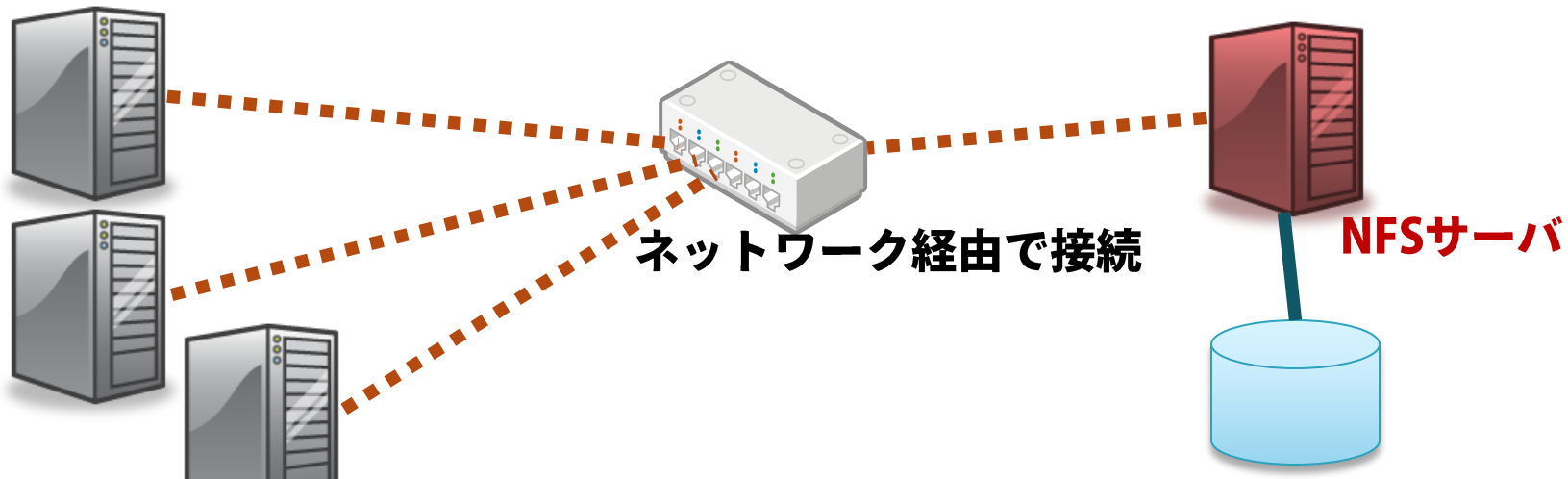
目的に応じて使い分けると効率的に作業ができる

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
- HA8000クラスタシステムにおける設定
  - /tmpに置いたファイルは、ログインノードでは1～2日で、計算ノードではジョブの終了時に削除される

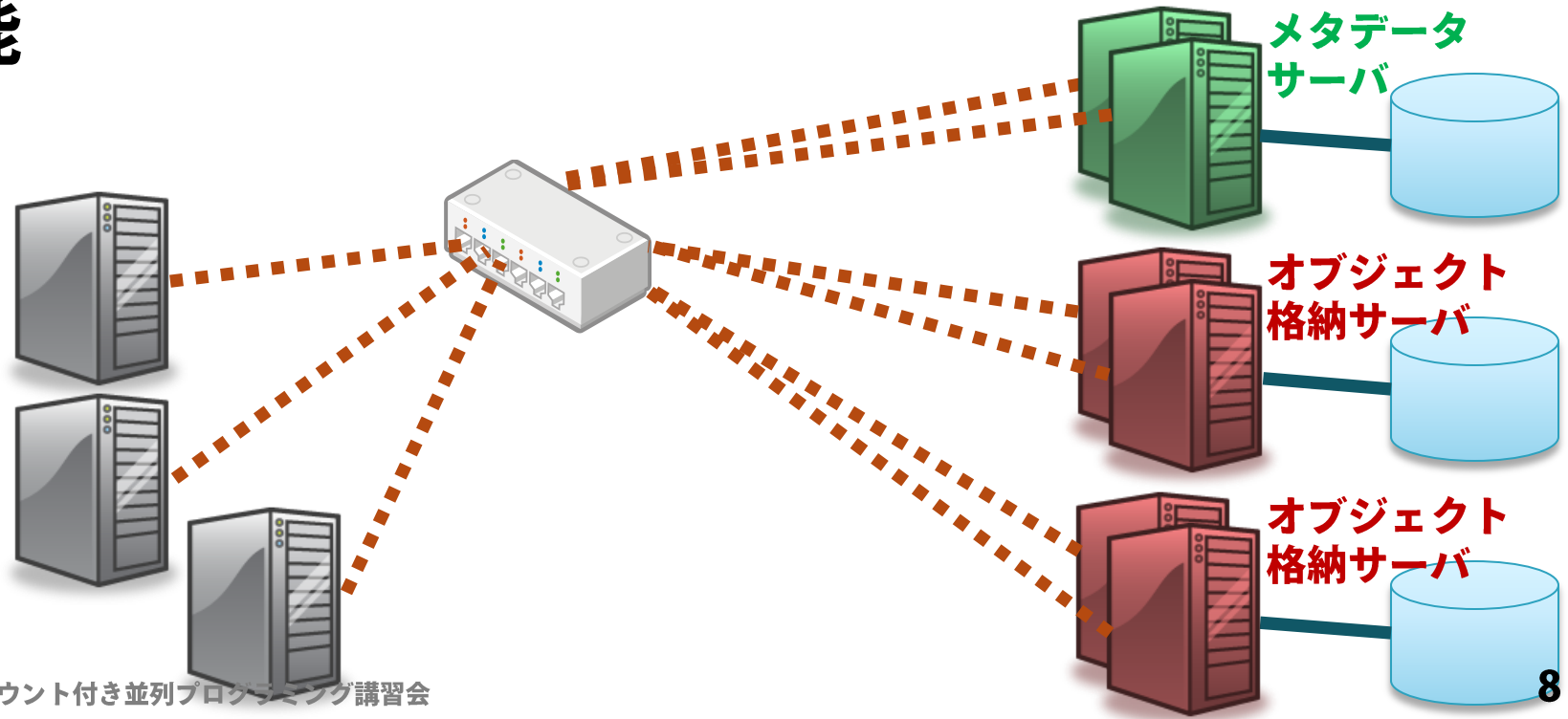
NFS

- 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)
- HA8000クラスタシステムにおける設定
  - ユーザ・グループごとに分けてサーバを4台使用



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、サーバの故障に対応可能**



ファイルシステム選択の基準

- ローカルディスク
  - 一時的なデータ置き場
- NFS
  - ログインノードにおける作業時やファイル操作のレスポンスが重視される場合に適している
  - 多数のクライアントからの並列アクセスには適さない
- 分散ファイルシステム(HSFS, Lustre)
  - 複数のファイルサーバにデータを分散できるため、多くのプロセスからアクセスする場合に効率がよい
  - 構成が複雑なため、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転送効率を上げることができる(lfs setstripeなど)

HA8000の分散ファイルシステム

- **HSFS(Hitachi Striping Filesystem)**
  - 複数ノードから大きなブロックサイズ(数MB以上?)で入出力を行う場合に効率がよいとされている
  - 現在はノードあたりの最大帯域幅は1Gbps
    - 今後改善の計画あり
  - /shortは5日後に削除される(一時的なデータ置き場)
- **Lustre**
  - 2010年3月からサービス開始
  - 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**

HSFS	`la -a`
NFS	`quota -v`
Lustre	`lfs quota /lustre/`
/tmp	`df /tmp`

課題1

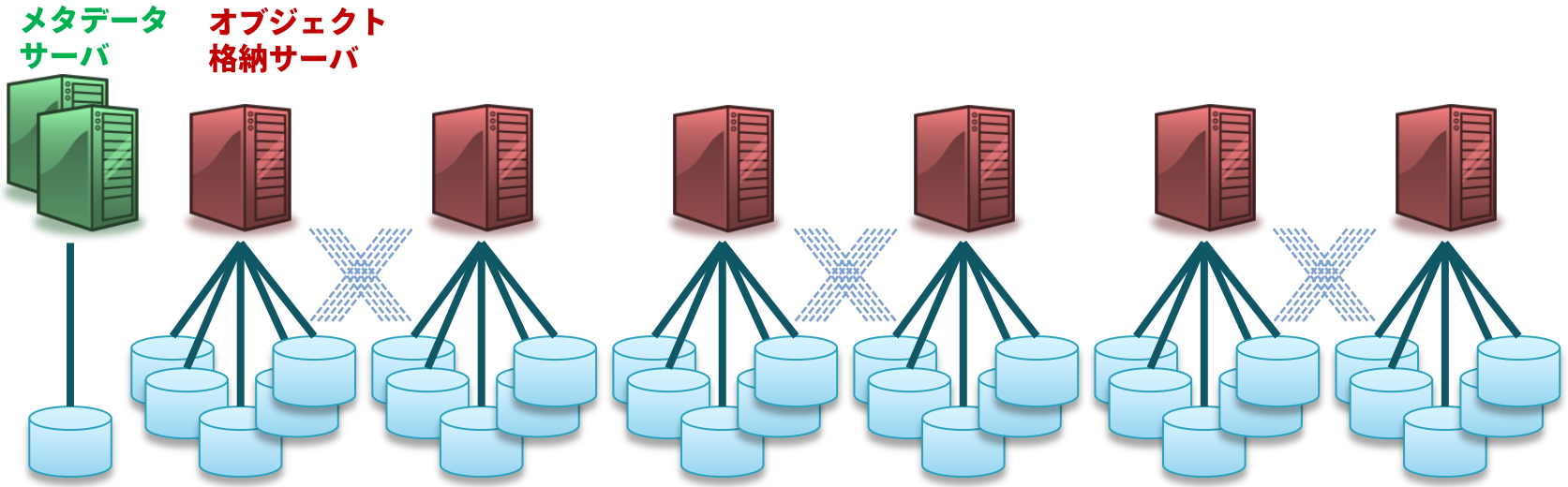
- それぞれのファイルシステムでファイル展開コマンドを実行せよ
  - 実行時間にどのような差があるか？
- 各ファイルシステムの残り容量を確かめよ

課題で使用するファイル

- **/home/t00004/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。**

Lustreのデータ配置

- HA8000のLustreファイルシステムは30個のOST(Object Storage Target: 仮想的なディスク)で構成
- それぞれのOSTはRAID6構成(8D+2P)
- メタデータの格納先(MDT: Metadata Target)はRAID1構成



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

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=/lustre/$USER/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 41.0291 seconds, 105 MB/s
```

OST数が1の場合の書き込み性能

```
$ rm /lustre/$USER/4G.dat
$ lfs setstripe -s 1M -c 4 /lustre/$USER/4G.dat
```

ストライプ設定の変更(4つのOSTにデータを分散)

```
$ dd if=/dev/zero of=/lustre/$USER/4G.dat bs=1M count=4096
4096+0 records in
4096+0 records out
4294967296 bytes (4.3 GB) copied, 10.9823 seconds, 391 MB/s
```

OST数が4の場合の書き込み性能

キュー操作上級編(1/2)

- qstat -f ジョブID (qstat -l)
  - 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

- qscript ジョブID
  - 投入したスクリプトを確認するコマンド

qstat -f の出力例

```
BATCH REQUEST:  253443.batch1
    Name:  test.sh
    Owner: uid=33071, gid=30144
    Priority: 63
    State: 1(RUNNING)
    Created at:  Tue Jun 30 05:36:24 2009
    Started at:  Tue Jun 30 05:36:27 2009
    Remain : 14 minutes 47 seconds

QUEUE
    Name:  tutorial

RESOURCES
    Per-proc. CPU time limit      = UNLIMITED
                          warning = 0 seconds
    Per-req.  CPU time limit      = UNLIMITED
                          warning = 0 seconds
    Per-proc. core file size limit= 0 kilobytes
    Per-proc. data size limit     = 28 gigabytes
                          warning = 0 kilobytes
    Per-proc. perm file size limit= UNLIMITED
                          warning = 0 kilobytes
    Per-proc. memory size limit   = 28 gigabytes
    Per-req.  memory size limit   = 28 gigabytes
    Per-proc. stack size limit    = 2 gigabytes
    Per-proc. volafile size limit = 0 kilobytes
    Per-req.  volafile size limit = 0 kilobytes
    Per-proc. working set limit   = 28 gigabytes
    Per-req.  etime limit         = 15 minutes
                          warning = 0 seconds
    Per-proc. execution nice pri. = 0

FILES
    Stderr: None
    Stdout: None
    stderr to stdout: No

MAIL
    Address: t00004@ha8000-3.cc.u-tokyo.ac.jp
    When:

MISC
    Partition: None
    Node: 1
    Jobtype: T1
    Rerunnable: Yes
    Performance information: Yes
    Shell: None
    Account name: f
    Qsub at: /nfs/all/t00004
```

キュー操作上級編(2/2)

- `qsub -N 2 -q debug` **スクリプト名**
  - **debugキューの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**
  - **注意:** `qscript`**で表示されるものと実際のオプションが異なる場合がある**
  - `qstat -f`**を使って確認すれば、正しい情報が得られる**

課題2

- qsub -q tutorial 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

- qscriptでスクリプトを確認せよ
- qstat -f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

課題3

- 実行されたジョブのノード数(-N)とジョブタイプ(-J)を標準出力に表示するスクリプトを書け
  - ヒント
    - qstat -f の出力から、必要な行をgrepする
    - “Node: 8” という行の、8の部分を出力 → awk ‘{print $2}’

makeで分割コンパイル

make

- プログラムの分割コンパイル等を支援するソフトウェア
- 変更があったファイルのみを再コンパイル
- 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
  - 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
```

スペースでなくタブにする

- **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.
```

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

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場合にコマンドを実行**
  - **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

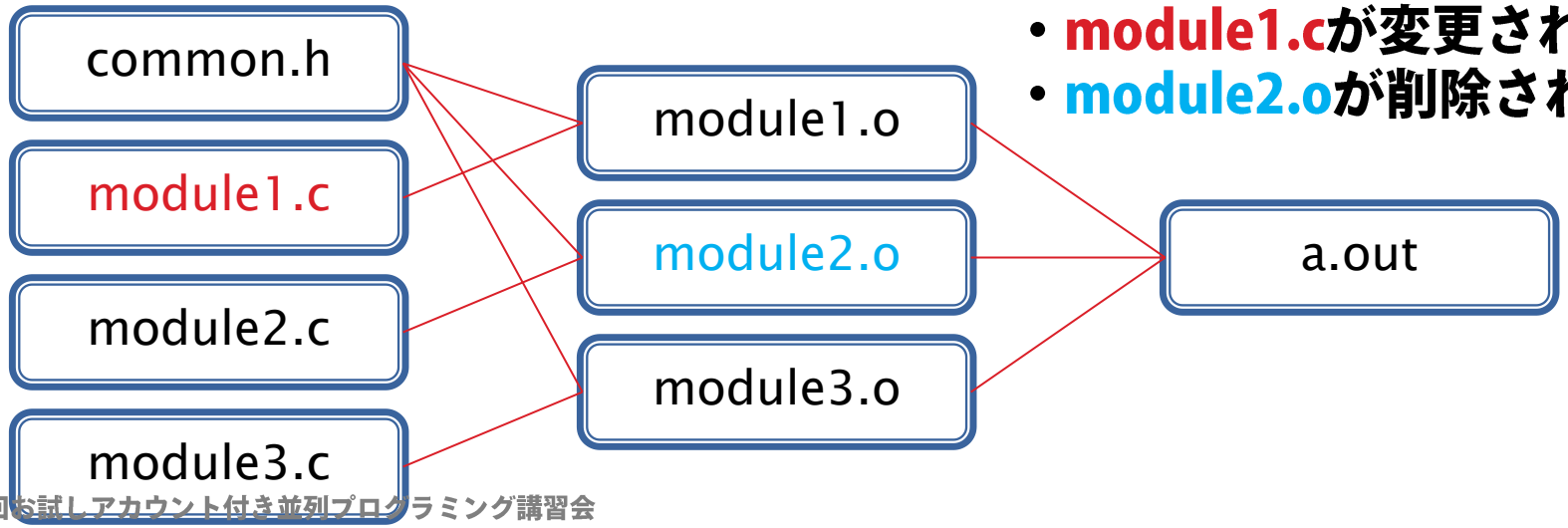
- **2回目のmakeで起きていたこと**
  - **main.oのコンパイルは、main.cに変更がなかったため行われなかった**
- **Makefileに依存関係を適切に記述することで、変更があった部分だけを再コンパイルすることができる**

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**

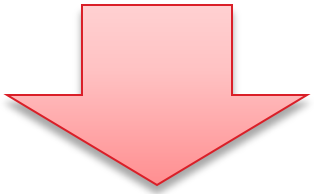
```
$ make -C hello2 target       (cd hello2; make targetと同様)
```

課題4

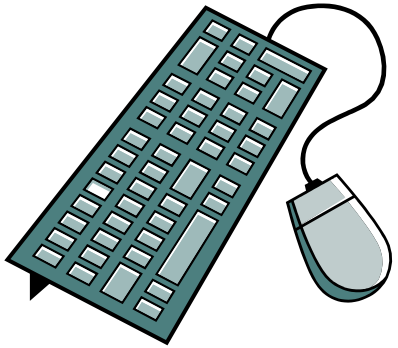
- **コマンドの前のタブを、スペースにした場合、どのようなエラーが出力されるか**
- **.PHONY: Xがあるときとない時で、make Xの動作に違いがあることを確認せよ**

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**
$OBJECTS**とすると、**$(O)BJECTS**と同じことになる**

- **再帰的な展開**

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

CFLAGS**は** -Idir1 -Idir2 -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**

```
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
```

- **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

$@	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 $<
```

変数の評価順序

```
DATE1  = $(shell date)
DATE2 := $(shell date)
DATE3  = `date`
```

- DATE1
  - 参照されるたびにdateが実行される
  - 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
  - 実行されるタイミングは最初
- 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

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月のデータだけを処理する

- 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

makeの並列処理への応用

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を使うとうれしいこと
  - 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 し直せば続きからやってくれる

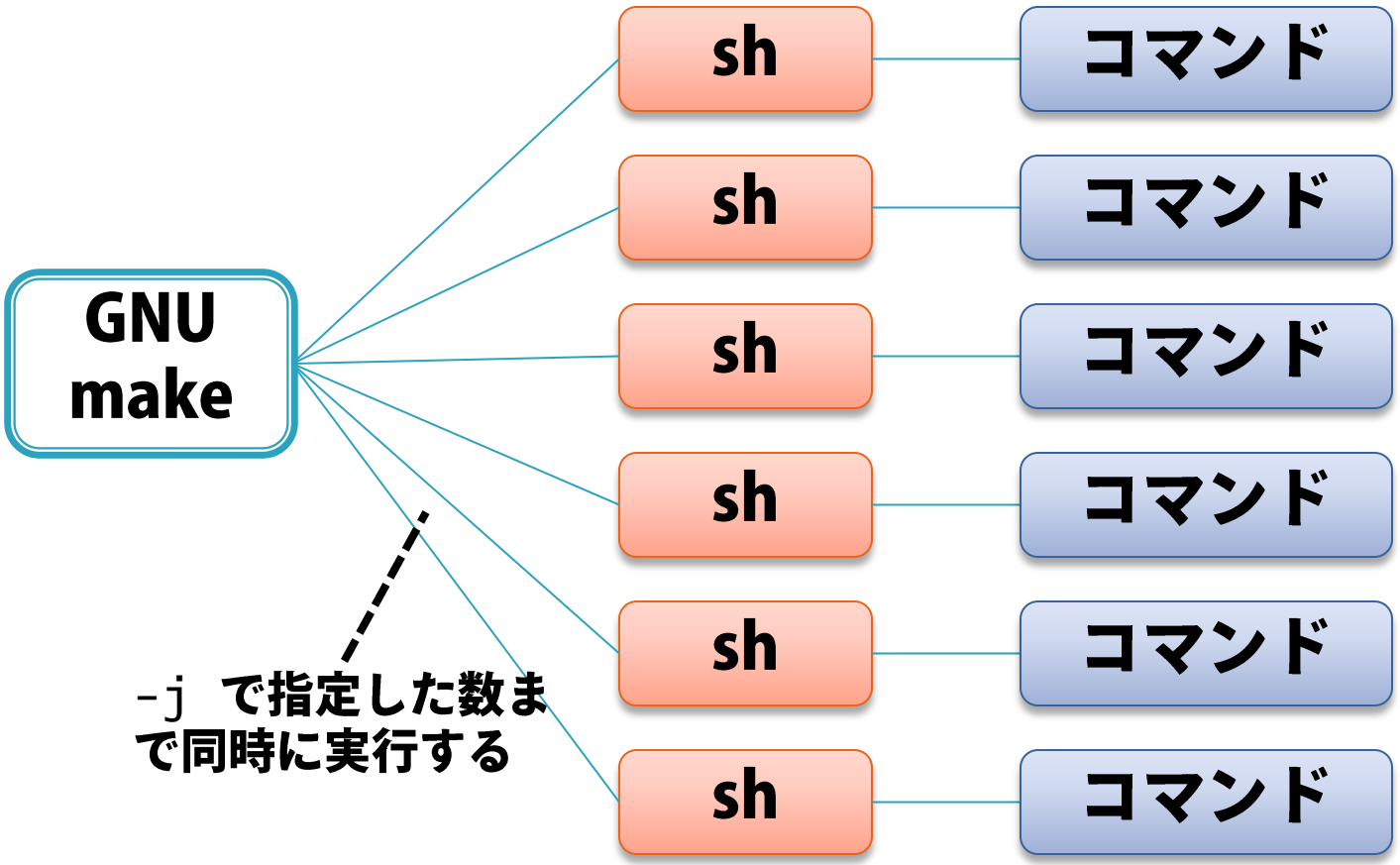
並列make

- 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(RHEL5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(∞)
- 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:**

```
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**

```
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

(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**

複数ノードで並列make

- HA8000クラスタシステムの場合、1ノードでは使える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
- 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

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

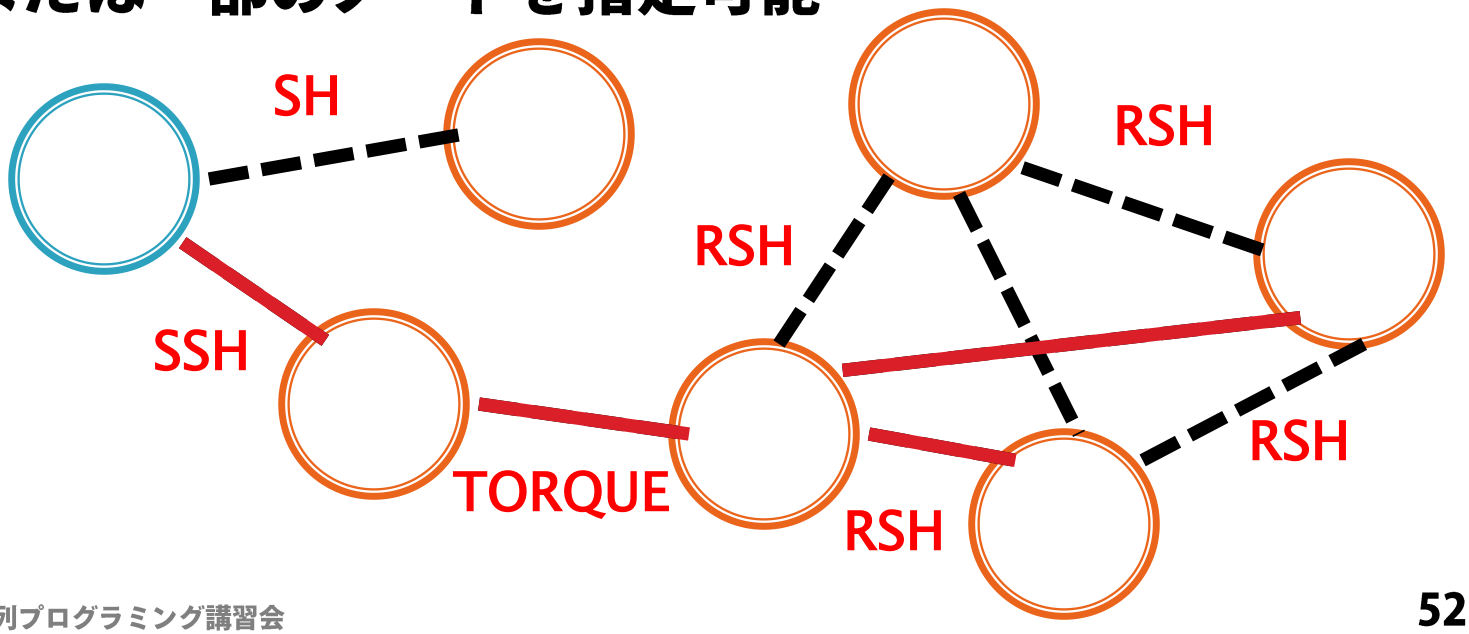
- HA8000上のインストール先

  各ノードの /opt/itc/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



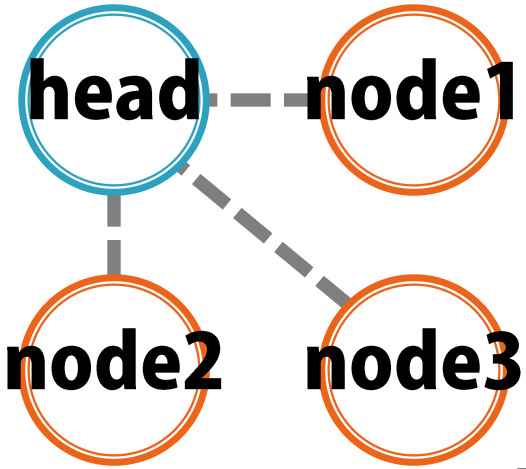
HA8000上でGXPを使用する(1/2)

```
#@$-N 4
#@$-J T1

NODES=`qstat -f $PBS_JOBID|grep Node:|awk '{print$2}'`

gxpc --root_target_name head
gxpc use torque_psched head node
gxpc explore node[[1--$NODES]]

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

gxpc quit
```

ノード数を取得

各ノードでGXPDを起動

コマンドの実行



torque_pschedはtorque
ジョブ内のプロセス起動

HA8000上でGXPを使用する(2/2)

- **Exploreするところまでを実行するスクリプト ha8000_gxp.sh を使用すれば、より簡単に記述可能**

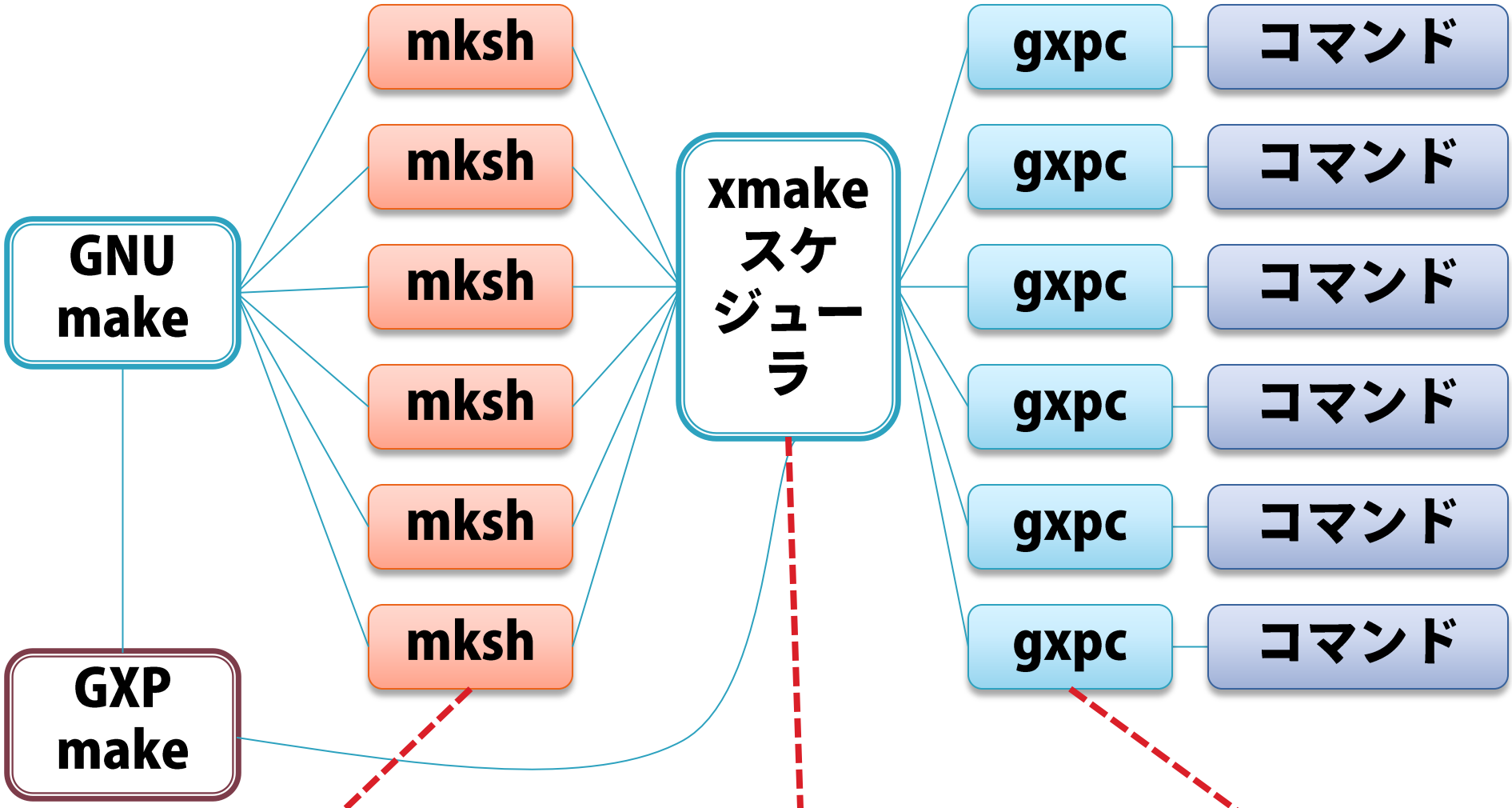
```
#@$-N 4
#@$-J T1

. /opt/itc/bin/ha8000_gxp.sh

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

gxpc quit
```

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

GXP makeの動作の仕組み



GXP make サンプルスクリプト

```
#@$-N 4
#@$-J T1

cd $QSUB_WORKDIR
. /opt/itc/bin/ha8000_gxp.sh
```

GXPDの起動

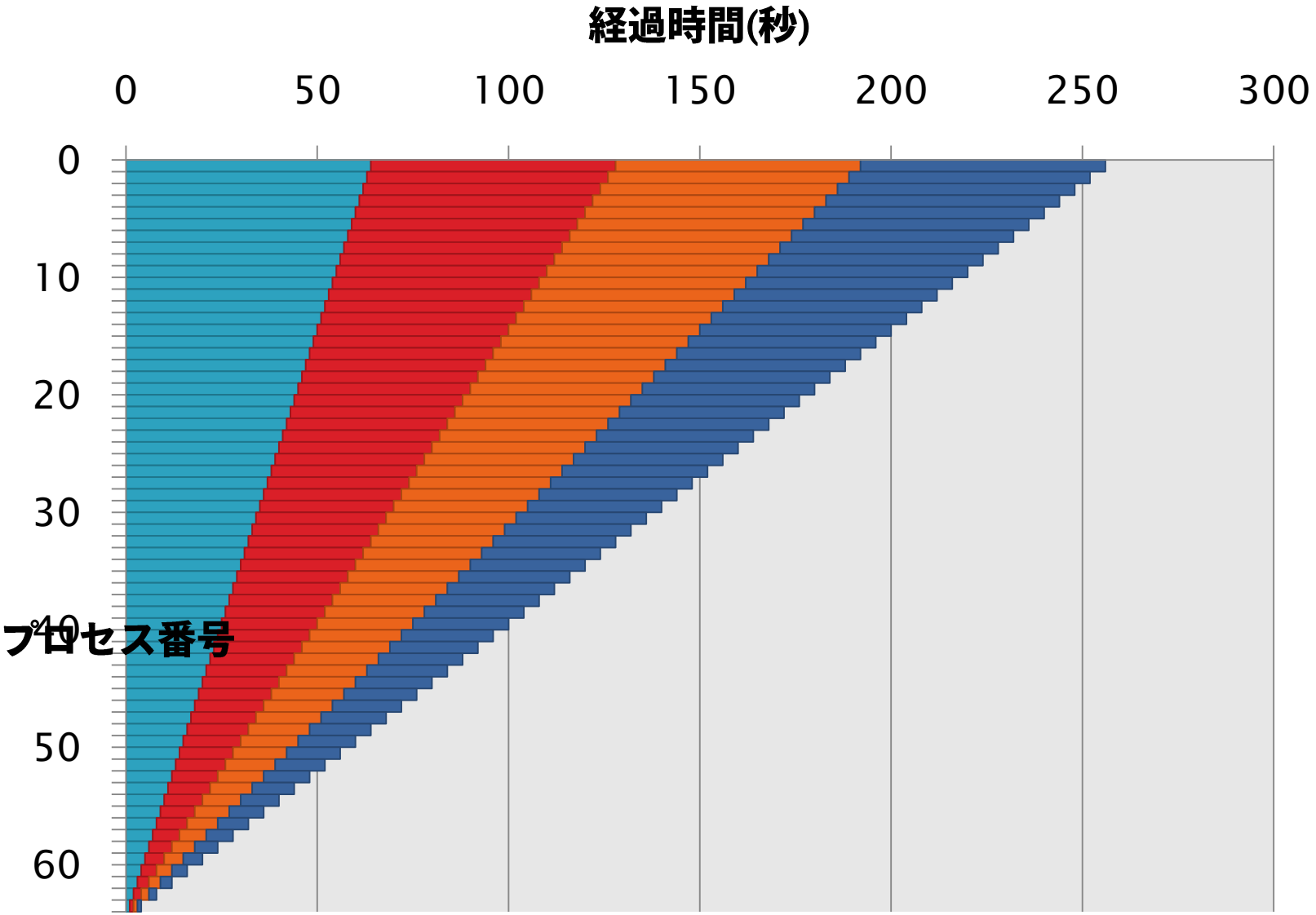
```
gxpc cd `pwd`
gxpc make -j
gxpc quit
```

並列makeの実行

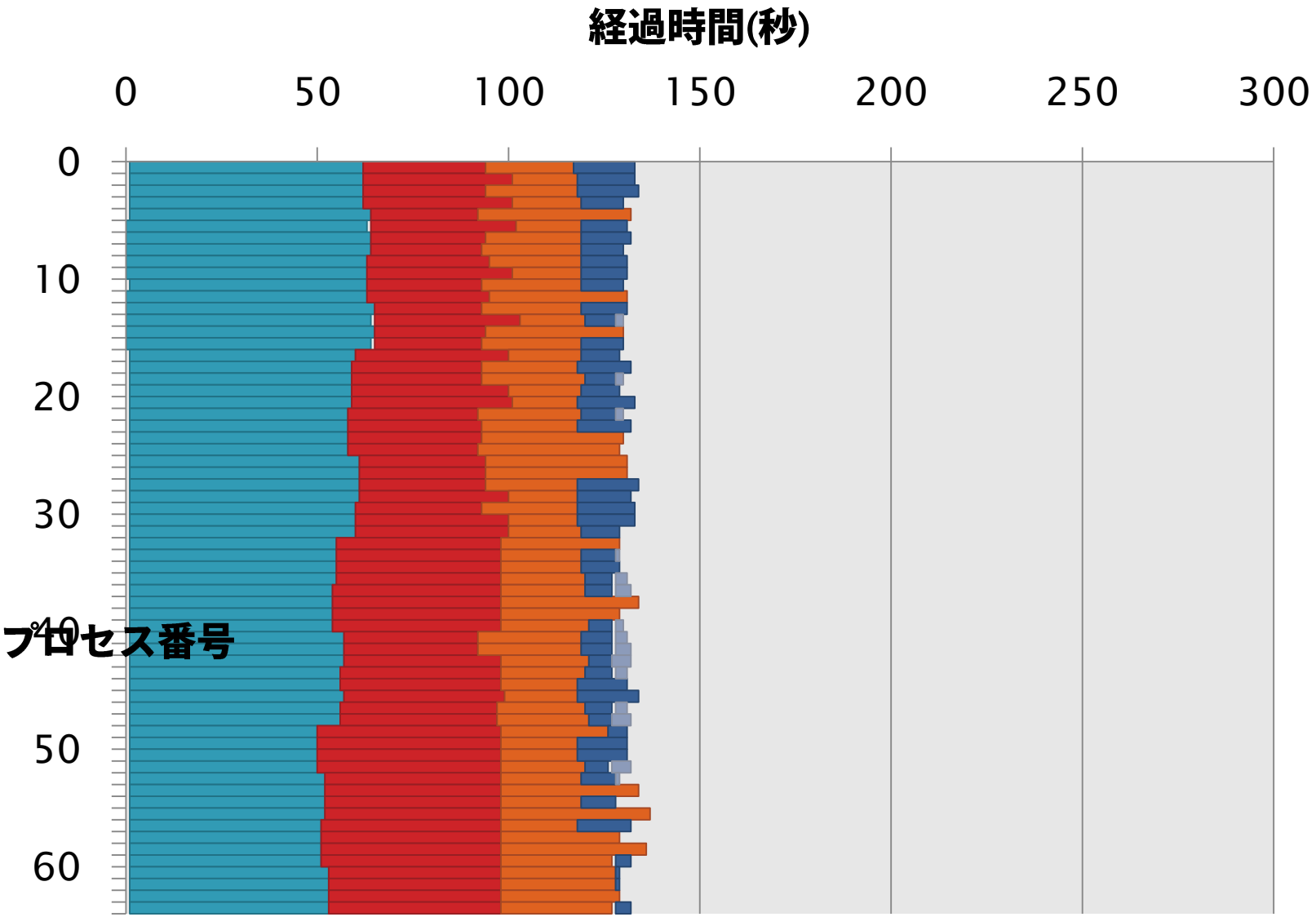
課題8

- 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)

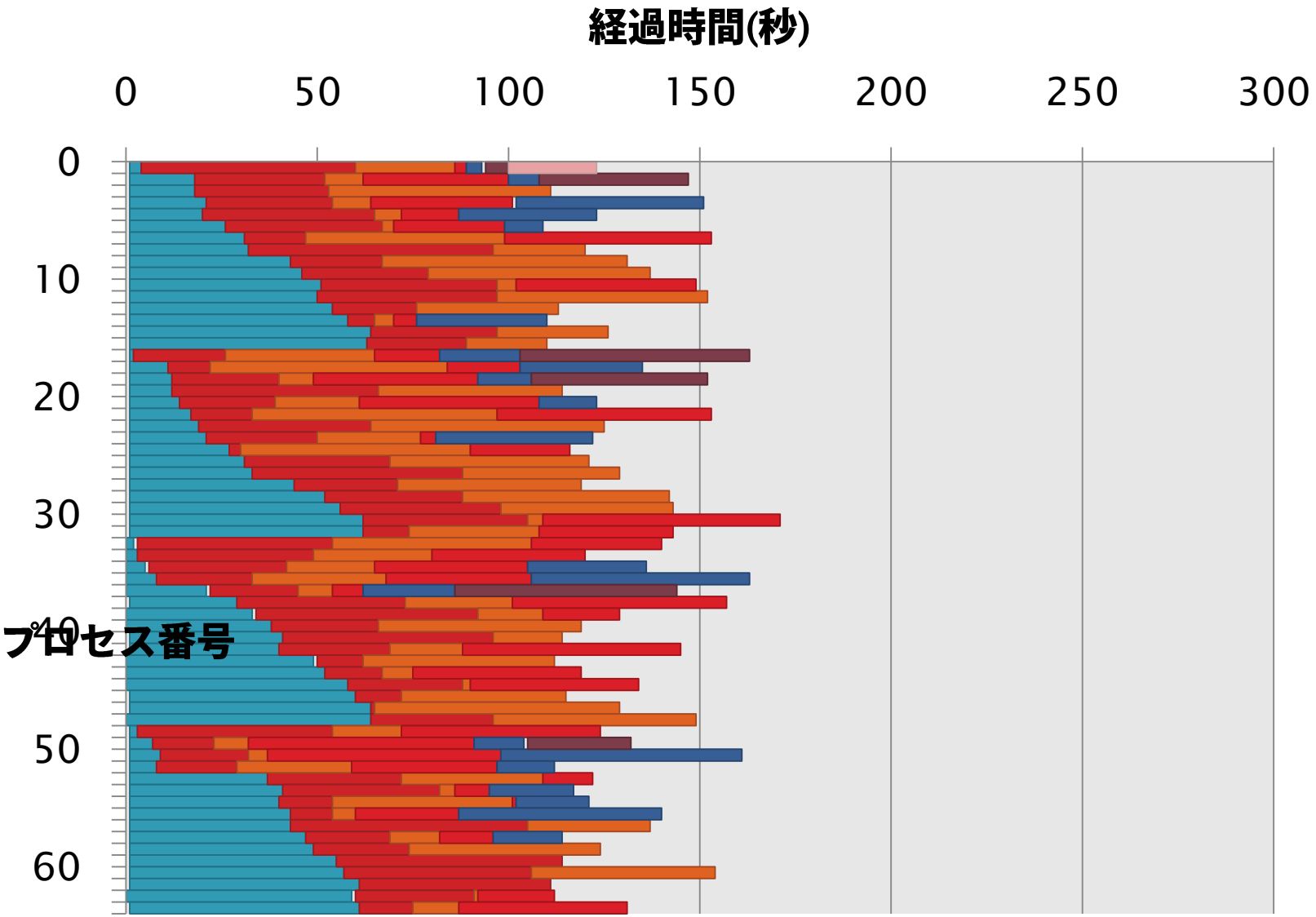
負荷分散を行わない場合



負荷分散を行った場合



負荷分散あり・ランダムな順番



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→(key, value)の集合**
    - **Reduce: (key, value)の集合→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 $(GXP_MAKE_MAPRED)の前に、以下の変数を設定する
  - input=入力ファイル名
  - output=出力ファイル名
  - mapper=mapperコマンド(ex_word_count_mapper)
  - reducer=reducerコマンド(ex_count_reducer)
  - n_mappers=map ワーカ数(3)
  - n_reducers=reduce ワーカ数(2)
  - int_dir=中間ファイル用ディレクトリ名
    - 省略時は$(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バッチキューイングシステム
  - HA8000クラスタシステム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