gxp3を使った
お手軽並列分散処理と
それに関するいくつかの注意点

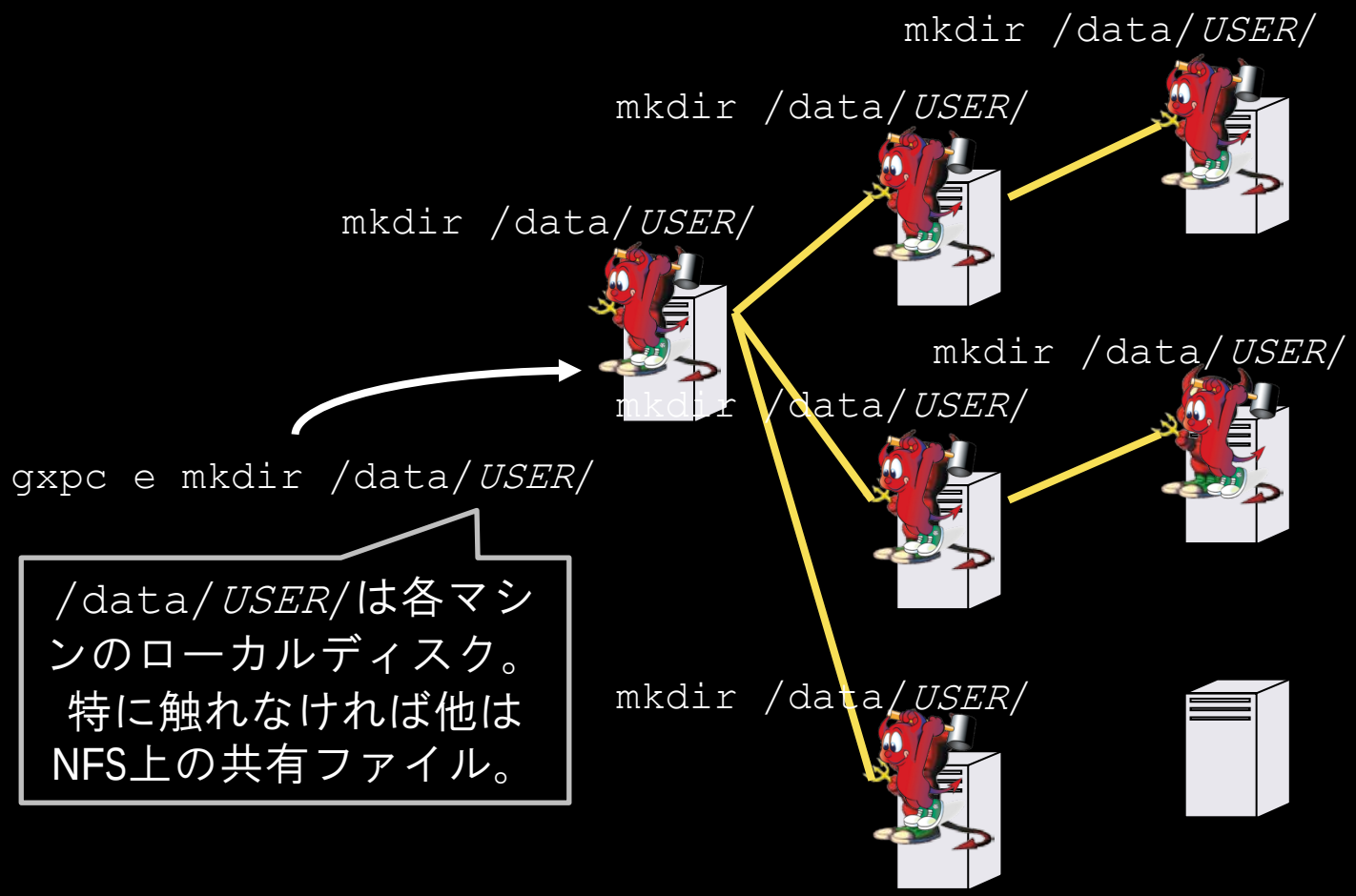
Last Update: 4 July 2013

MURAWAKI Yugo

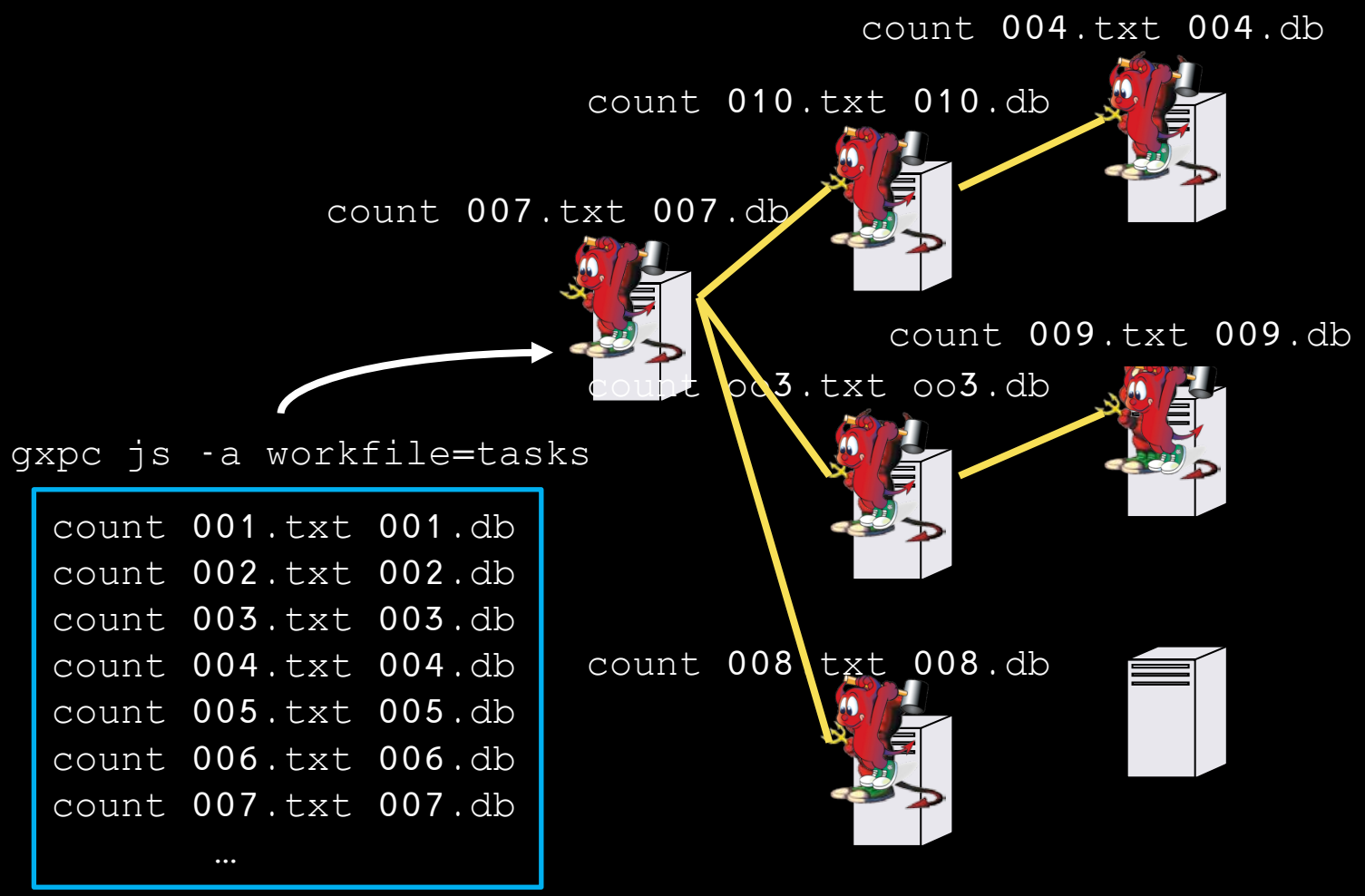
前回の復習

- gxp3はお手軽並列分散処理ツール
- gxp3はクラスタ上の各計算機にデーモンを立ち上げ、デーモンのネットワークを作る
- デーモンを介してコマンドを実行させる
- `gxpc e` (alias: `e`) で同じコマンドを一斉に実行させる (前処理・後処理用に)
- 本処理は`gxpc js`あるいは`gxpc make`で

gxpの起動～gxpc e



GXP js/GXP make



今日のお題

- 簡単なワークフローの例
  - GXP js
  - GXP make
- 複雑なワークフローの例
- ~~炎上デモ~~
- 注意点

簡単なワークフローの例

- テキスト中の単語頻度を集計する

  `count` *`input output`*

- 処理したいジョブが100個

```
nice -19 count 001.txt 001.db
nice -19 count 002.txt 002.db
nice -19 count 003.txt 003.db
                …
nice -19 count 100.txt 100.db
```

クラスタ利用時のルールとしてniceをかける

GXP js 1/4

- ジョブ100個を*tasks*ファイルに保存
- jsを実行

```
% gxpc js -a work_file=tasks
```

- スケジューラを介してジョブが各計算機にばらまかれ、実行される

GXP js 2/4

- よく使うオプション: cpu_factor

```
% gxpc js -a work_file=tasks -a cpu_factor=0.5
```

(各ノードにおける並列数の最大値。8 coreのノードなら、最大で 8*0.5=4 ジョブまで並列実行)

- ジョブ数 < 総コア数 のときは特に cpu_factor を指定すべき
  - gxpは均等にジョブをばらまかず、特定のノードにコア数限度まで投入しがち
    - e.g. 8コアのノード3台に対して、10個のジョブを (8, 2, 0) のように割り振りがち
  - cpu_factorで使用コア数を制限すれば均される

GXP js 3/4

- ジョブ一覧を標準入力から与えることも可能
  - シェルスクリプト書いて、コマンドラインから実行

```
for ((i=1;i<=100;i+=1)); do
  INPUT=`printf %03d.txt $i`
  OUTPUT=`printf %03d.db $i`
  echo nice -19 count $INPUT $OUTPUT
done | gxpc js -a work_fd=0
nice -19 db-merge --from=1 --to=100 --output=merged.db
```

他の方法
1. `for i in {001..100}; do`
2. `for i in `seq -w 1 100`; do`
(1. は zsh のみ)

続けてマージ処理をシェルスクリプトに書ける

GXP js 4/4

- 一度に最後まで実行できなかったらどうする?
- 問題: 今のままでは最初から再実行してしまう
- 解決策: 出力が存在したらジョブにしない

```
for ((i=1;i<=100;i+=1)); do
  INPUT=`printf %03d.txt $i`
  OUTPUT=`printf %03d.db $i`
  if [ ! -s "$OUTPUT" ]; then
    echo nice -19 count $INPUT $OUTPUT
  fi
done | gxpc js -a work_fd=0
nice -19 db-merge --from=1 --to=100 --output=merged.db
```

GXP make 1/4

- (普通の) Makefileを書く

```
INPUTS := $(wildcard *.txt)
OUTPUTS := $(patsubst %.txt,%.db,$(INPUTS))
$(OUTPUTS) : %.db : %.txt
    nice -19 count $< $@
merged.db : $(OUTPUTS)
    nice -19 merge-db --from=1 --to=100 --output=$@
all: merged.db
```

- `make -n all` でテスト (dry-run)
- `gxpc make -j num all` で本実行
  - *num* (e.g. 20) は最大並列数
    - 省略可だが指定すべき
    - コア数よりも少し大きいぐらいの値を指定する

GXP make 2/4

- gxpのオプションは - - のあとに指定

```
% gxpc make -j 20 all -- -a cpu_factor=0.5
```

- makeなら途中からの再実行は自動で行なってくれる
- 一部のジョブが失敗しそうなときはmakeの -k オプションを使う
  - 一部のジョブが失敗しても、依存関係上先に進めなくなるまでジョブを投下し続ける

GXP make 3/4

- recipeの各行は別のホストで実行されうる

```
$(OUTPUTS) : %.db : %.txt
    nice -19 count $< /data/USER/tmp/$@.tmp
    mv /data/USER/tmp/$@.tmp $@.tmp
```

/data/*USER*/tmp/$@.tmp
が見つからない!

- 対策1: コマンドを && でつなぐ

```
$(OUTPUTS) : %.db : %.txt
    nice -19 count $< /data/USER/tmp/$@.tmp && ¥
      mv /data/USER/tmp/$@.tmp $@.tmp
```

- 対策2: コマンドをシェルスクリプトに書き出して、makeからはシェルスクリプトを呼び出す (おすすめ)

```
$(OUTPUTS) : %.db : %.txt
    nice -19 count.sh $< $@
```

count.shの中身は必ず一つのノードで実行される

GXP make 4/4

- 開始時にローカルホストのload_avgが跳ね上がることがあるが、問題ない
  - makeの仕様上、ローカルホストでタスクごとにダミーコマンドを実行する
- デバッグが大変
  - typoがあっても「Xを生成するルールがない」と怒られるだけで、原因がすぐに分からない
  - `make -p` でデータベースをdumpして調べる
    - 他に良い方法があったら教えて!

複雑なワークフローの例

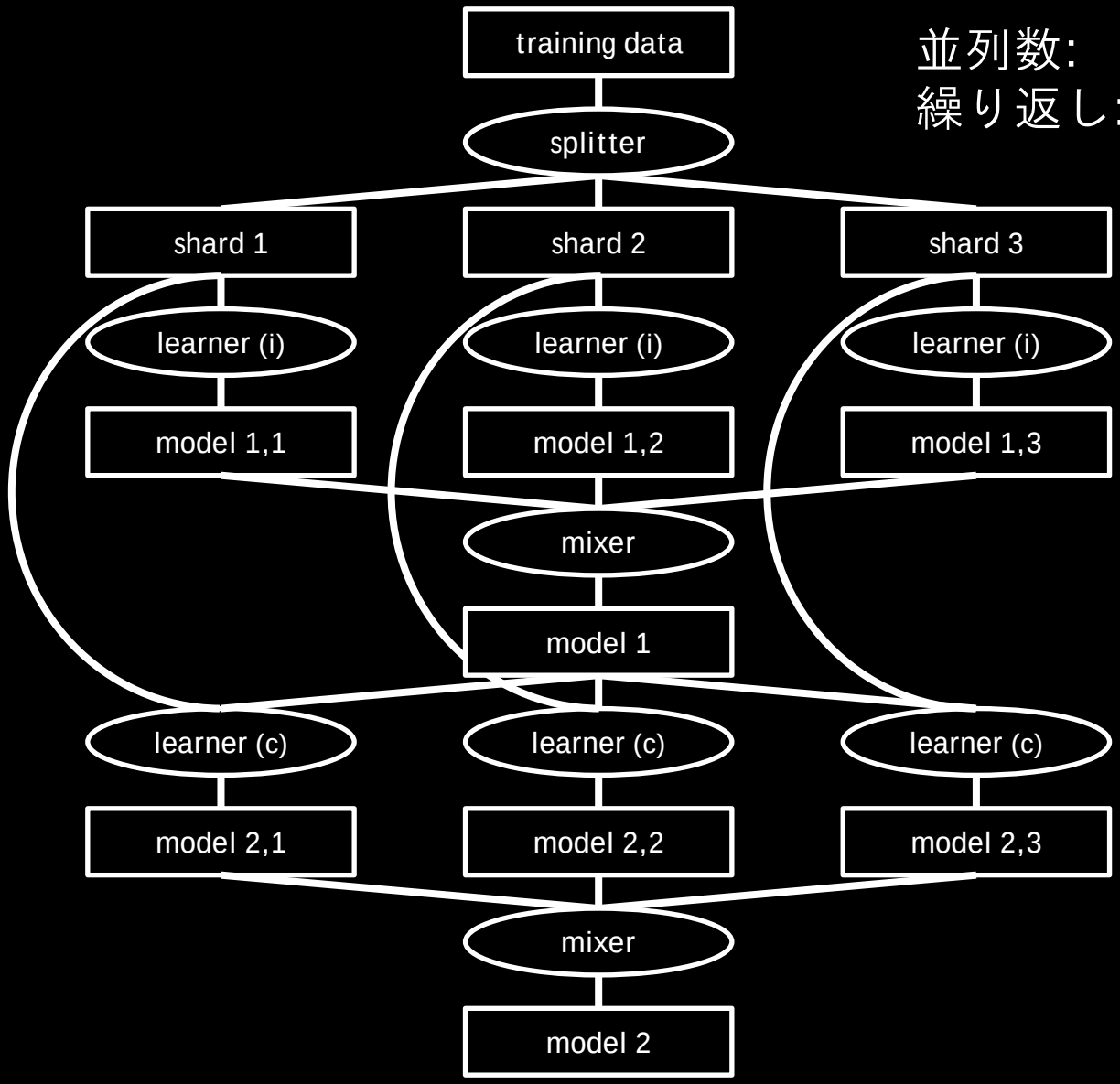
Iterative Parameter Mixing for Perceptron Training (McDonald+, 2010)

- Perceptronの学習を並列化する
- 巨大な訓練データを分割し、データの断片から並列に学習
- iterationごとにモデルをmixingすると収束性が保証される

並列分散処理的には

- 並列数 (=断片数) と繰り返し回数に依存する複数ステップの処理

Iterative Parameter Mixing for Perceptron Training



並列数: $(SHARD)
繰り返し: $(ITER)

複雑なmakeの書き方 1/5

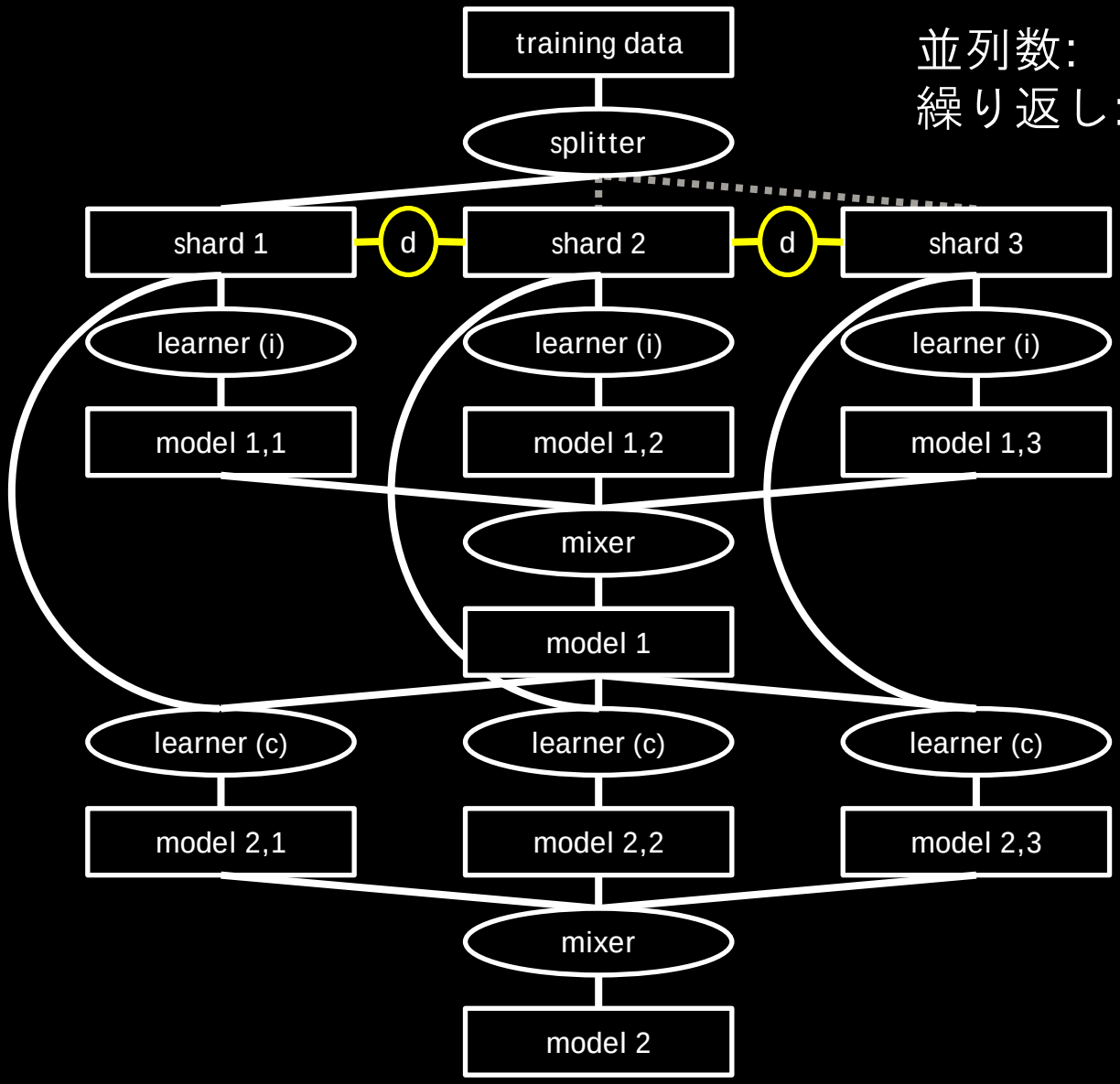
```
% gxpc make -j 20 all SHARD=3 ITER=2
```

Makefileは宣言的に記述

```
all : $(OUTPUT)

$(OUTPUT) : model.$(ITER)
    mv $< $@
```

Iterative Parameter Mixing for Perceptron Training



並列数: $(SHARD)
繰り返し: $(ITER)

複雑なmakeの書き方 2/5

define/foreach-eval-callでルールを展開

```
shard.1 : $(INPUT)
    $(SPLITTER) --input=$< --prefix=shard $(SHARD)

define shard_dummy
shard.$(1): shard.$(shell expr $(1) - 1)
endef

$(foreach x,$(shell seq 2 $(SHARD)), ¥
  $(eval $(call shard_dummy,$(x))))
```

訓練データ分割の本処理

recipeのないダミールールを生成

呼び出し先では$(1)で参照

複雑なmakeの書き方 3/5

define / foreach-eval-callでルールを展開

```
define parallel_train_init
model.1.$(1): shard.$(1)
    $(LEARNER) --input=shard.$(1) --output=model.1.$(1)
endef

$(foreach x,$(shell seq 1 $(SHARD)), ¥
  $(eval $(call parallel_learn_init,$(x))))
```

複雑なmakeの書き方 4/5

define / foreach-eval-callでルールを展開

```
define parallel_learn_each
model.$(1).$(2): model.$(shell expr $(1) - 1) shard.$(2)
    $(LEARNER) --input=shard.$(2) ¥
      --init=model.$(shell expr $(1) - 1) ¥
      --output=model.$(1).$(2)
endef

$(foreach x,$(shell seq 2 $(ITER)), ¥
  $(foreach y,$(shell seq 1 $(SHARD)), ¥
    $(eval $(call parallel_learn_each,$(x),$(y)))))
```

複雑なmakeの書き方 5/5

define / foreach-eval-callでルールを展開

```
define mixer
model.$(1): $(foreach y,$(shell seq 1 $(SHARD)),model.$(1).$(y))
    $(MIXER) --prefix=model.$(1). --output=model.$(1)
endef

$(foreach x,$(shell seq 1 $(ITER)), ¥
  $(eval $(call mixer,$(x))))
```

処理の実際

- GXP js/GXP makeから本処理を直接呼び出すのではなく、本処理を含んだシェルスクリプトを呼び出す
- シェルスクリプトの中身
  - 前処理 (処理に必要なデータのコピーなど)
  - 本処理
  - 後処理 (不要になった一時ファイルの削除など)

IOに注意 1/4

- homeで大規模データを読み書きしない
  - 炎上したときにみんなが巻き添えをくらう
- 入出力データはhome以外のstorage serverに置く
  - 炎上しても被害の範囲が限定される

IOに注意 2/4

- 出力は一時的にローカルディスクにはいて、最後にmv/cp/scp
  - NFSへのアクセスを減らす (効果は要検証)
  - 強制終了したときに壊れた出力ファイルが残りにくい
- 巨大なファイルはgzip/bzip2で圧縮

```
$MAIN_PROGRAM --input=$INPUT --output=$TMP_DIR/tmp.$$
if [ $? = 0 ]; then
  bzip2 -c $TMP_DIR/tmp.$$ > $TMP_DIR/tmp.bz2.$$
  mv $TMP_DIR/tmp.bz2.$$ $OUTPUT
fi
rm -f $TMP_DIR/tmp.$$ $TMP_DIR/tmp.bz2.$$
```

IOに注意 3/4

- 巨大なDBを並列に引く場合などには、まずローカルディスクにDBをコピーしてそちらを引く
- bcpならGXPのノードがリレー方式でファイルをコピーするので、負荷が抑えられる

```
gxpc mw bcp orchid:$HOME/large.db /data/USER/
$MAIN_PROGRAM --db=/data/USER/large.db
```

- js/make起動時にアクセスが集中する場合は、sleepしてIOのタイミングをずらす (やっつけ技)

```
sleep `expr $RANDOM / 400`; # $RANDOM は 0 から 32767
cp /yew/USER/large.db /data/USER/
```

IOに注意 4/4

- KNPの辞書は各マシンにコピー済みなのでそれを使う
  - knprcの辞書指定部分を変更して ~/.knprc.local に保存

```
(KNP辞書ディレクトリ
    /data/share/usr/share/knp/dict
)
```

  - knp -r ~/.knprc.local で呼び出す

実例をいくつか眺めてみる

今回触れなかった話題

- 再帰make
  - 起動時にワークフローが決まらない場合に必要
- 自分でジョブスケジューラを書く
- MapReduce
- 分散ファイルシステムと併用
- vgxp