2021.08.31 第10回SPring-8データ科学研究会
／第63回SPring-8先端利用技術ワークショップ

「富岳」を用いた大規模イメージングデータの分散並列処理の試み

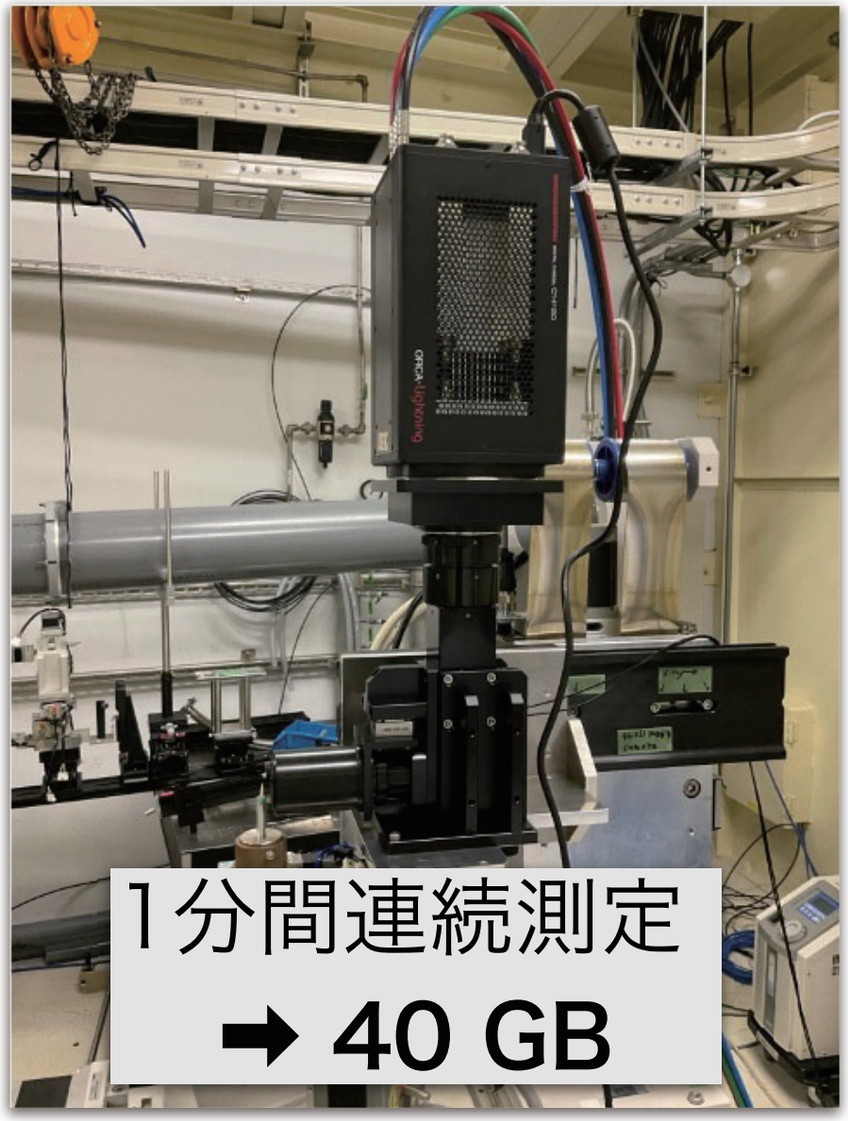
高垣 昌史 (JASRI 産業利用・産学連携推進室)

目次

背景と目的

背景

放射光実験技術の高度化に伴うデータの大規模化



イメージング用高速CMOSカメラ
浜松ホトニクス ORCA-Lightning

画素数：(w)4608 × (h)2592
フレームレート：30 枚/秒 (16 bit)

数十 GB/測定

2017年度～

GPU導入時の想定を超え始めた

極めて近い将来

数百 GB/測定

(以後も増大の予想)

計算資源の課題と対策

オンプレミス
施設管理の計算資源の限界

スケーラビリティ
データ量増加への耐性, コスト, 電力...

確保/維持は困難

➡ 外部の大型計算機の利用

「富岳」の活用

- 恒常的なスケーラビリティ確保
- 既存プログラムの高速化
- **低コスト**での移行 ⬅ "移植"よりも容易に

本試行の目的

本講演の趣旨

「富岳」を利用した実験データ処理の実際

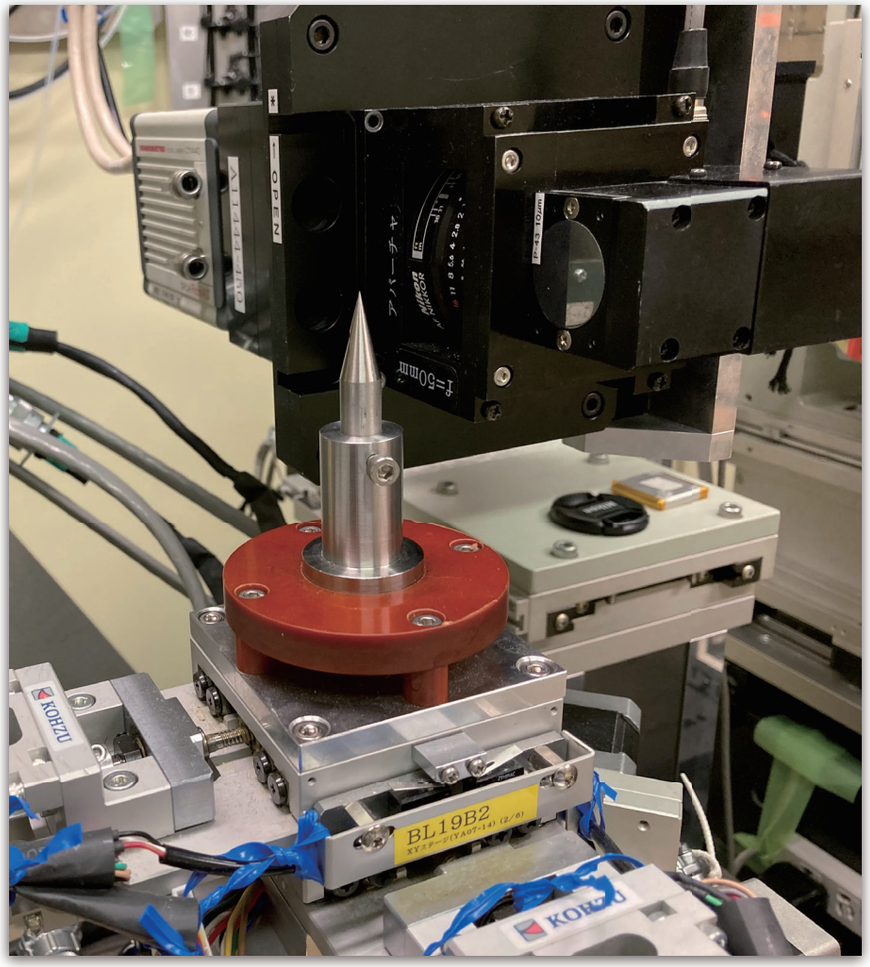
- SPring-8 大規模データ処理系として
  CT/ラミノグラフィ処理系の実装に向けての検証

- **放射光ユーザの開発/処理環境として**
  実験支援の立場から情報発信

プログラムと性能評価

テストプログラム
イメージング(CT)データの規格化処理

数千ファイル/測定を, ファイル単位で並列処理



CT レイアウト例 (BL14B2)

ファイル構成

I ：オブジェクトイメージ (X線透過像)×3,159

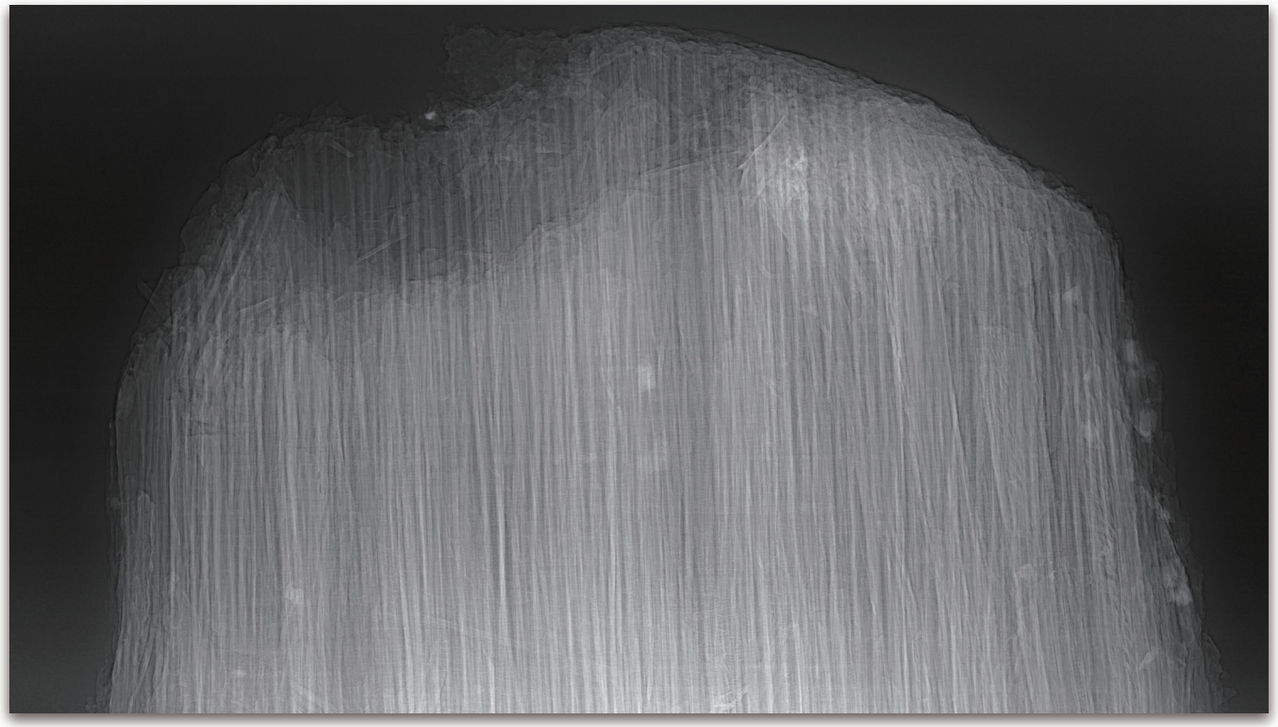
I_0 ：ダイレクトイメージ (X線強度像)×2

I_n ：ダークイメージ (検出器ノイズ像)×2

規格化 (線吸収係数像)

$$I_{\mu t} = \ln\left(\frac{I_0 - I_n}{I - I_n}\right)$$

オブジェクトイメージの規格化



オブジェクトイメージ

2平均ダイレクトイメージ

2平均ダークイメージ

ファイルフォーマットとサイズ

	入力データ	出力データ
ファイルフォーマット	TIFF	
色調	グレースケール	
画素数	(w) 4608 × (h) 2592	
データ型	UINT16	FLOAT32
1ファイルサイズ	約 23 MB	約 46 MB
全ファイルサイズ	**約 70 GB**	**約 140 GB**

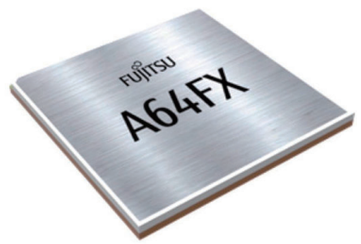
測定時間 2.5分

「富岳」開発/実行環境

- **RedHat Enterprise Linux**

  ssh ログイン, scp データ転送 ...

- **富士通FXプロセッサ + コンパイラ**



- **XcalableMP**

  C/Fortran 拡張型並列処理言語

- **emacs, make, ...**

- **ジョブスケジューラ**

XcalableMP

https://xcalablemp.org/

習得コストを下げる

- 分散メモリ環境を対象とした並列処理言語

  Message Passing Interface

- MPIコードを出力

- C/Fortran を拡張

  ディレクティブ

- 指示文ベース … 「逐次型 → 並列型」の移行が容易

  (例) `#include <stdio.h>`

プログラム normalize

Linux のコマンドラインアプリケーション

```
$ normalize -i INPUT_DIR -o OUTPUT_DIR -n START,N_OBJS
```

-i　I, I_0, I_n の保存されているディレクトリ

-o　$I_{\mu t}$ を保存するディレクトリ

-n　I の開始番号とファイル数

I のファイル名

o0000.tif, o0001.tif, ...

(例) `$ normalize -i ~/data/in -o ~/data/out -n 0,3159`

o0000.tif ～ o3158.tif を規格化

並列処理による高速化

逐次処理

$I(f)$入力

$f = 0 \sim 3{,}158$

規格化

$I_{\mu t}(f)$出力

for ループ

for ループの分解

ファイルを順に処理

並列処理

3,159 並列

$I(0)$ 入力 → 規格化 → $I_{\mu t}(0)$ 出力

$I(1)$ 入力 → 規格化 → $I_{\mu t}(1)$ 出力

$I(2)$ 入力 → 規格化 → $I_{\mu t}(2)$ 出力

$I(3)$ 入力 → 規格化 → $I_{\mu t}(3)$ 出力

全ファイルを同時処理

逐次処理バージョン - 全コード

```
int main(int _argc, char **_argv) {
  char *common_files[] = {"d01.tif", "d02.tif", "q0.tif", "qlast_d.tif"};

  /*============================================================================
    コマンドライン引数
   */
  char in[LEN_PATH], out[LEN_PATH], file_no[16];
  check_opts(_argc, _argv, in, out, file_no);
  size_t obj_start, n_objs;
  sscanf(file_no, "%lu,%lu", &obj_start, &n_objs);

  /*============================================================================
    画像情報 - UINT16 グレースケールを想定
  */
  unsigned width, height;

  char path[LEN_PATH];
  sprintf(path, "%s/%s", in, common_files[0]);
  read_size_tiff(path, &width, &height);

  size_t size = width * height;
  size_t bytes_16u = size * sizeof(unsigned short);
  size_t bytes_32f = size * sizeof(float);
```

```
  /*============================================================================
    暗電流，ダイレクトの読み込み
  */
  unsigned short *p_dark0_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_dark1_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct0_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct1_16u = (unsigned short *)GC_MALLOC(bytes_16u);

  unsigned short *flist[] = {p_dark0_16u, p_dark1_16u, p_direct0_16u, p_direct1_16u};
  for (size_t i = 0; i < 4; i++) {
    char path[LEN_PATH];
    sprintf(path, "%s/%s", in, common_files[i]);
    read_image_tiff(path, (char *)flist[i]);
  }

  /*============================================================================
    規格化
  */
  unsigned short *p_obj_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  float *p_norm_32f = (float *)GC_MALLOC(bytes_32f);

  for (size_t f = obj_start; f < obj_start + n_objs; f++) {
   char obj_name[256];
   sprintf(obj_name, "o%04lu.tif", f);
   char path_in[LEN_PATH];
   sprintf(path_in, "%s/%s", in, obj_name);
   read_image_tiff(path_in, (char *)p_obj_16u);

   for (size_t p = 0; p < size; p++) {
      float dark = ((float)p_dark0_16u[p] + (float)p_dark1_16u[p]) / 2.0;
      float direct = ((float)p_direct0_16u[p] + (float)p_direct1_16u[p]) / 2.0;
      p_norm_32f[p] = -1 * log(((float)p_obj_16u[p] - dark) / (direct - dark));
    }

    char path_out[LEN_PATH];
    sprintf(path_out, "%s/%s", out, obj_name);
    write_image_tiff(path_out, width, height, 32, SAMPLEFORMAT_IEEEFP, (char *)p_norm_32f);
  }

  return EXIT_SUCCESS;
}
```

ここを並列化

逐次処理バージョン - 規格化部

```
unsigned short *p_obj_16u = (unsigned short *)GC_MALLOC(bytes_16u);
float *p_norm_32f = (float *)GC_MALLOC(bytes_32f);

for (size_t f = obj_start; f < obj_start + n_objs; f++) {
  char obj_name[256];
  sprintf(obj_name, "o%04lu.tif", f);
  char path_in[LEN_PATH];
  sprintf(path_in, "%s/%s", in, obj_name);
  read_image_tiff(path_in, (char *)p_obj_16u);

  for (size_t p = 0; p < size; p++) {
    float dark = ((float)p_dark0_16u[p] + (float)p_dark1_16u[p]) / 2.0;
    float direct = ((float)p_direct0_16u[p] + (float)p_direct1_16u[p]) / 2.0;
    p_norm_32f[p] = -1 * log(((float)p_obj_16u[p] - dark) / (direct - dark));
  }

  char path_out[LEN_PATH];
  sprintf(path_out, "%s/%s", out, obj_name);
  write_image_tiff(path_out, width, height, 32, SAMPLEFORMAT_IEEEFP, (char *)p_norm_32f);
}
```

ファイル毎のループ (3,159回)

このループを分解 ➡ 3,159並列

I 入力

ピクセル毎のループ(規格化)

$I_{\mu t}$ 出力

並列化 = for ループの分解

並列処理バージョン - 全コード

XMP利用

並列数指示文

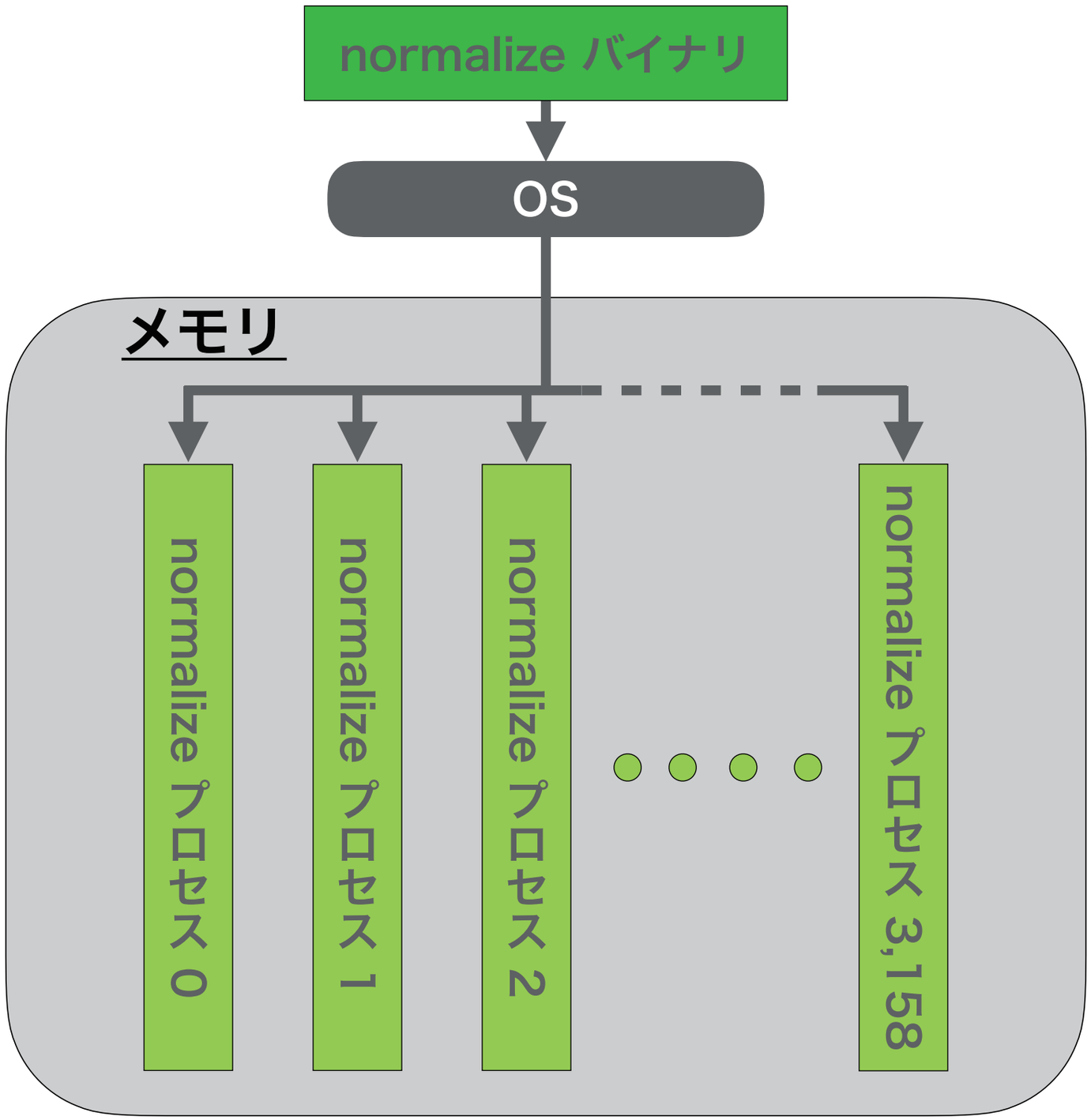
#include <xmp.h>
XcalableMP ライブラリの利用

#pragma xmp nodes p[*]
normalize が複数立ち上がる
(数はコマンドライン引数で指定)

for ループが無い

```c
#include <xmp.h>
int main(int _argc, char **_argv) {

#pragma xmp nodes p[*]

  char *common_files[] = {"d01.tif", "d02.tif", "q0.tif

  /*=============================================
    コマンドライン引数
   */
  char in[LEN_PATH], out[LEN_PATH], file_no[16];
  check_opts(_argc, _argv, in, out, file_no);
  size_t obj_start = atoi(file_no);

  /*=============================================================
    画像情報 - UINT16 グレースケールを想定
   */
  unsigned width, height;

  char path[LEN_PATH];
  sprintf(path, "%s/%s", in, common_files[0]);
  read_size_tiff(path, &width, &height);

  size_t size = width * height;
  size_t bytes_16u = size * sizeof(unsigned short);
  size_t bytes_32f = size * sizeof(float);

  ==========================================

  [unsigned short *)GC_MALLOC(bytes_16u);
  [unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct0_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct1_16u = (unsigned short *)GC_MALLOC(bytes_16u);

  lark1_16u, p_direct0_16u, p_direct1_16u};

  ]);

  /*=============================================================
    規格化
   */
  unsigned short *p_obj_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  float *p_norm_32f = (float *)GC_MALLOC(bytes_32f);

  size_t idx = xmpc_node_num(); // プロセス番号
  size_t f = idx + obj_start; // オブジェクトのファイル番号

  char obj_name[256];
  sprintf(obj_name, "o%04lu.tif", f);
  char path_in[LEN_PATH];
  sprintf(path_in, "%s/%s", in, obj_name);
  read_image_tiff(path_in, (char *)p_obj_16u);

  for (size_t p = 0; p < size; p++) {
    float dark = ((float)p_dark0_16u[p] + (float)p_dark1_16u[p]) / 2.0;
    float direct = ((float)p_direct0_16u[p] + (float)p_direct1_16u[p]) / 2.0;
    p_norm_32f[p] = -1 * log(((float)p_obj_16u[p] - dark) / (direct - dark));
  }

  char path_out[LEN_PATH];
  sprintf(path_out, "%s/%s", out, obj_name);
  write_image_tiff(path_out, width, height, 32, SAMPLEFORMAT_IEEEFP, (char *)p_norm_32f);

  return EXIT_SUCCESS;
}
```

マルチプロセス実行


normalize バイナリ
OS
メモリ
normalize プロセス 0
normalize プロセス 1
normalize プロセス 2
normalize プロセス 3,158

並列処理バージョン - 全コード

```c
#include <xmp.h>
int main(int _argc, char **_argv) {
#pragma xmp nodes p[*]

  char *common_files[] = {"d01.tif", "d02.tif", "q0.tif", "qlast_d.tif"};

  /*======================================================================

  */
  char in[LEN_PATH], out[LEN_PATH], file_no[16];
  check_opts(_argc, _argv, in, out, file_no);
  size_t obj_start = atoi(file_no);

  /*======================================================================
    画像情報 - UINT16 グレースケールを想定
  */
  unsigned width, height;

  char path[LEN_PATH];
  sprintf(path, "%s/%s", in, common_files[0]);
  read_size_tiff(path, &width, &height);

  size_t size = width * height;
  size_t bytes_16u = size * sizeof(unsigned short);
  size_t bytes_32f = size * sizeof(float);

  /*======================================================================
    暗電流, ダイレクトの読み込み
  */
  unsigned short *p_dark0_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_dark1_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct0_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  unsigned short *p_direct1_16u = (unsigned short *)GC_MALLOC(bytes_16u);

  unsigned short *flist[] = {p_dark0_16u, p_dark1_16u, p_direct0_16u, p_direct1_16u};
  for (size_t i = 0; i < 4; i++) {
    char path[LEN_PATH];
    sprintf(path, "%s/%s", in, common_files[i]);
    read_image_tiff(path, (char *)flist[i]);
  }

  /*======================================================================
    規格化
  */
  unsigned short *p_obj_16u = (unsigned short *)GC_MALLOC(bytes_16u);
  float *p_norm_32f = (float *)GC_MALLOC(bytes_32f);

  size_t idx = xmpc_node_num(); // プロセス番号
  size_t f = idx + obj_start; // オブジェクトファイル番号

  char obj_name[256];
  sprintf(obj_name, "o%04lu.tif", f);
  char path_in[LEN_PATH];
  sprintf(path_in, "%s/%s", in, obj_name);
  read_image_tiff(path_in, (char *)p_obj_16u);

  for (size_t p = 0; p < size; p++) {
    float dark = ((float)p_dark0_16u[p] + (float)p_dark1_16u[p]) / 2.0;
    float direct = ((float)p_direct0_16u[p] + (float)p_direct1_16u[p]) / 2.0;
    p_norm_32f[p] = -1 * log(((float)p_obj_16u[p] - dark) / (direct - dark));
  }

  char path_out[LEN_PATH];
  sprintf(path_out, "%s/%s", out, obj_name);
  write_image_tiff(path_out, width, height, 32, SAMPLEFORMAT_IEEEFP, (char *)p_norm_32f);

  return EXIT_SUCCESS;
}
```

XMP利用

並列数指示文

以降を全て並列化

1プロセス = 1ファイル

for ループが無い

並列処理バージョン - 規格化部

```
unsigned short *p_obj_16u = (unsigned short *)GC_MALLOC(bytes_16u);
float *p_norm_32f = (float *)GC_MALLOC(bytes_32f);

size_t idx = xmpc_node_num(); // プロセス番号
size_t f = idx + obj_start; // オブジェクトのファイル番号
```

ループカウンタの代わり

ループから出しただけ

I 入力

```
char obj_name[256];
sprintf(obj_name, "o%04lu.tif", f);
char path_in[LEN_PATH];
sprintf(path_in, "%s/%s", in, obj_name);
read_image_tiff(path_in, (char *)p_obj_16u);
```

ピクセル毎のループ(規格化)

```
for (size_t p = 0; p < size; p++) {
  float dark = ((float)p_dark0_16u[p] + (float)p_dark1_16u[p]) / 2.0;
  float direct = ((float)p_direct0_16u[p] + (float)p_direct1_16u[p]) / 2.0;
  p_norm_32f[p] = -1 * log(((float)p_obj_16u[p] - dark) / (direct - dark));
}
```

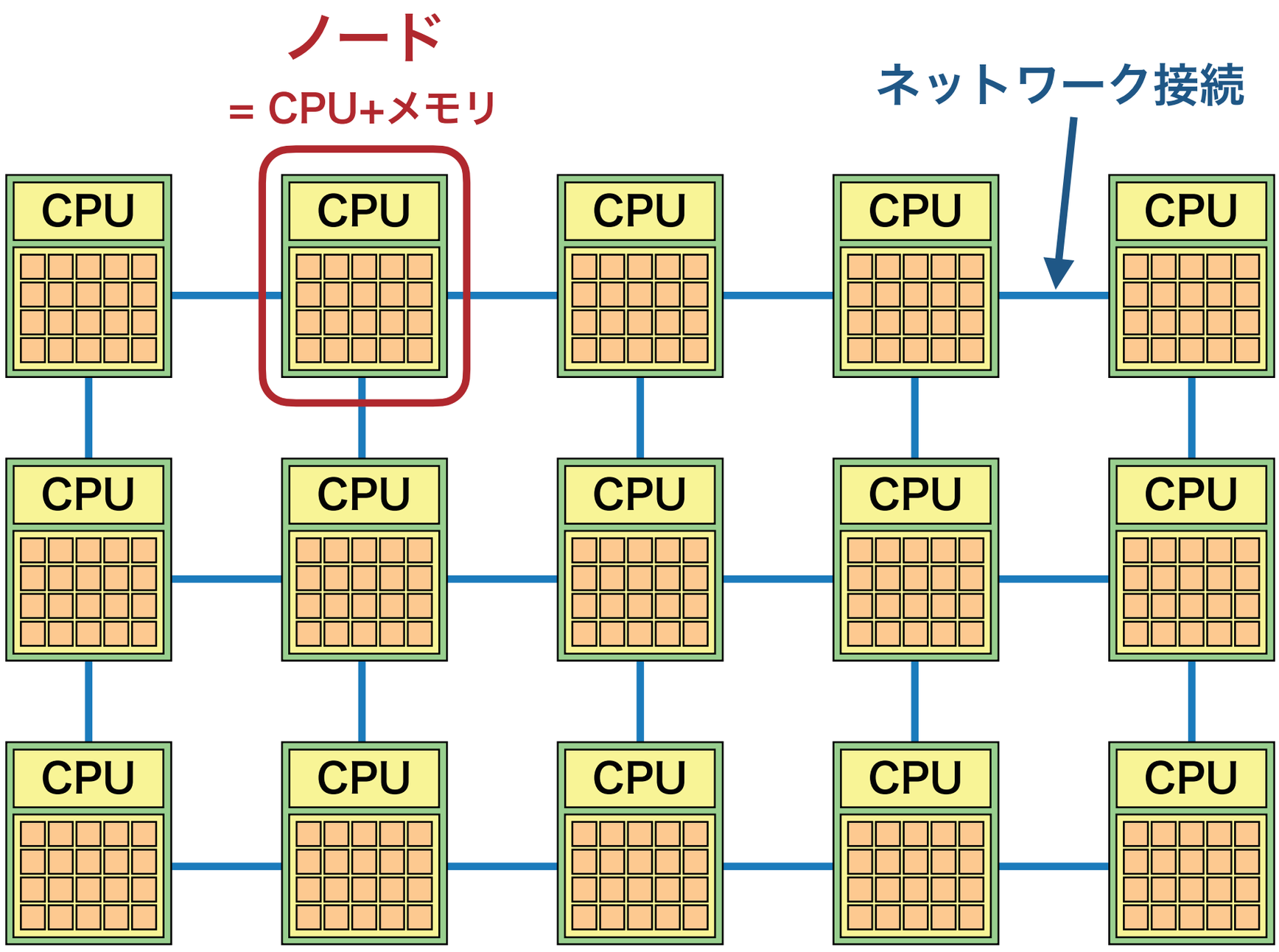
$I_{\mu t}$ 出力

```
char path_out[LEN_PATH];
sprintf(path_out, "%s/%s", out, obj_name);
write_image_tiff(path_out, width, height, 32, SAMPLEFORMAT_IEEEFP, (char *)p_norm_32f);
```

for ループカウンタの代替定義

以外 変更点なし

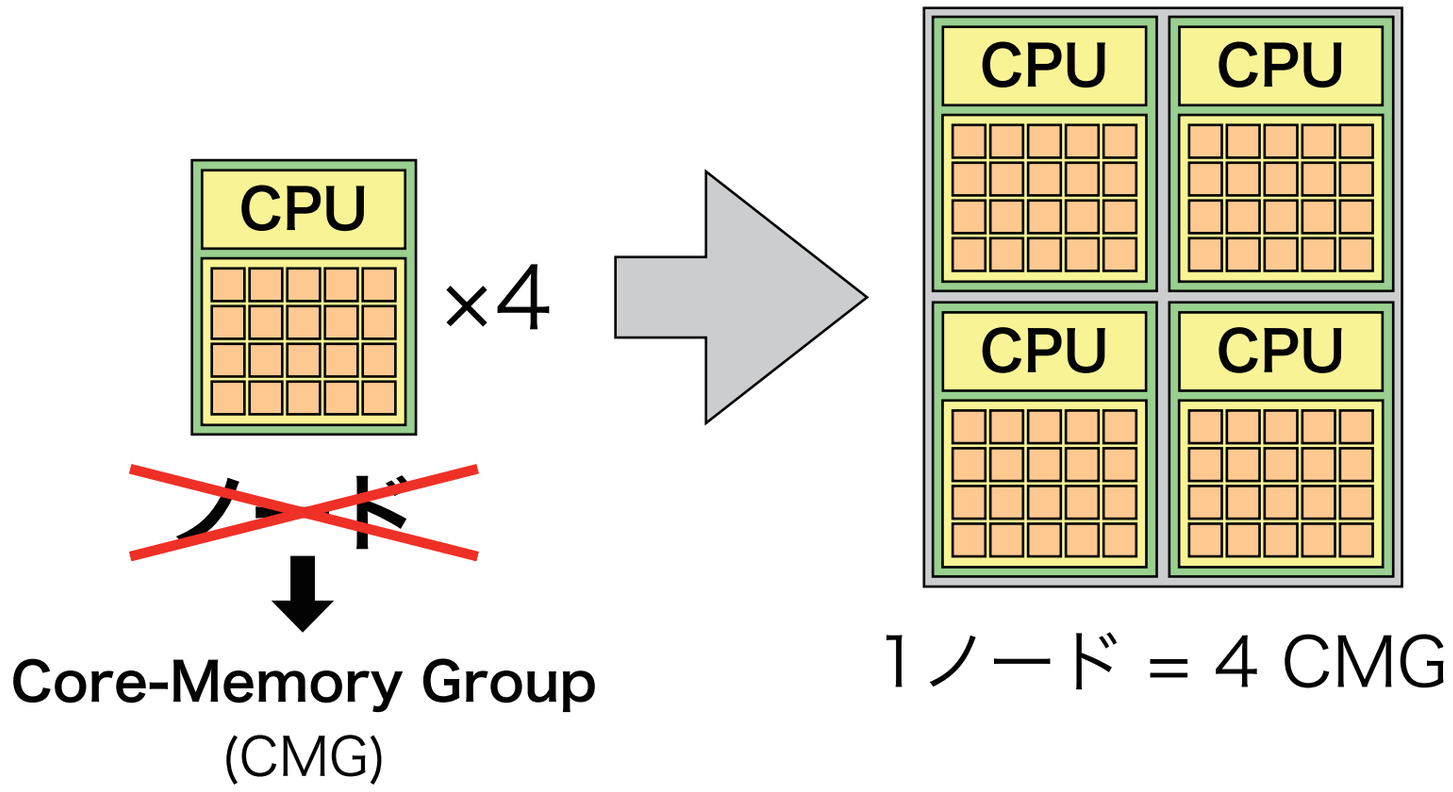
ノード - 並列化の学術的最小単位



分散メモリシステム

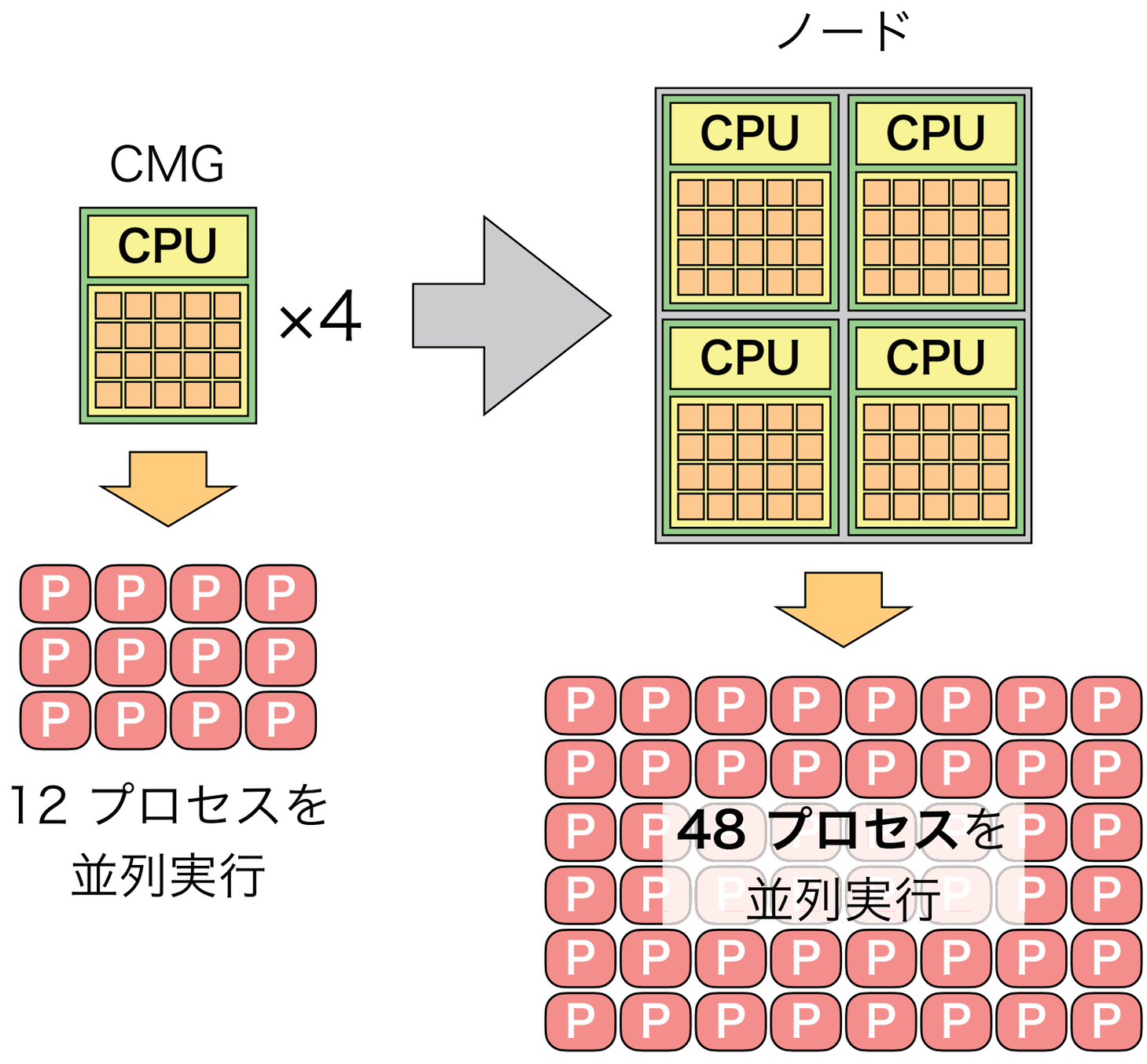
(XcalableMP の対象)

「富岳」におけるノード

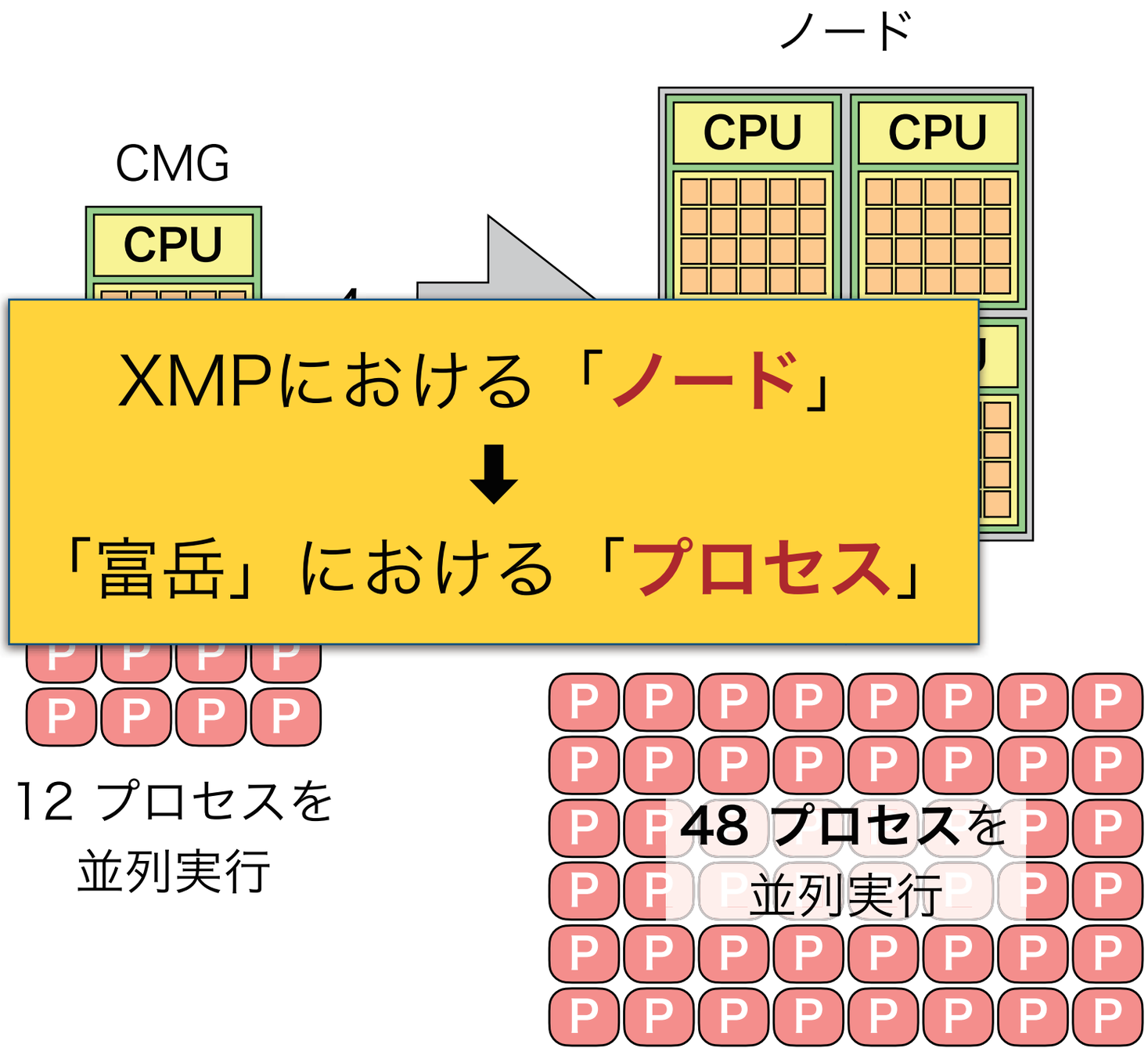


CPUを4個単位で使用する

プロセス
プログラミングの立場から見た並列単位



プロセス
プログラミングの立場から見た並列単位



並列数の指定
ジョブスクリプトの記述方法と投入

ジョブスクリプト run.sh

```
#!/bin/sh
#PJM -L "node=66"
#PJM -L "rscgrp=small"
#PJM -L "elapse=0:30:00"
#PJM --mpi "max-proc-per-node=48"
#PJM -S
#PJM --llio perf

export OMP_NUM_THREADS=1
export FLIB_FASTOMP=FALSE
export FLIB_CNTL_BARRIER_ERR=FALSE

export LD_LIBRARY_PATH=${HOME}/lib/libgc/lib:${HOME}/lib/libtiff/lib:${LD_LIBRARY_PATH}


PROG="normalize_xmp_1f1n_cache"
EXE="$HOME/src/20210719/$PROG/$PROG"
DATAROOT="/vol0003_cache/hp210225/u01868/data"
INDATA="${DATAROOT}/in"
OUTDATA="${DATAROOT}/out/${PROG}"
LOG="$PROG.log"
FILE_NO="0"
N_OBJS="3159"

time mpiexec -of ${LOG} -np ${N_OBJS} ${EXE} -i ${INDATA} -o ${OUTDATA} -n ${FILE_NO}
```

ジョブ投入

```
$ pjsub run.sh ↩
```

並列数の指定
ジョブスクリプトの記述方法と投入

ジョブスクリプト run.sh

```
#!/bin/sh
#PJM -L "node=66"
#PJM -L "rscgrp=small"
#PJM -L "elapse=0:30:00"
#PJM --mpi "max-proc-per-node=48"
#PJM -S
#PJM --llio perf

PROG="normalize_xmp_1f1n_cache"
EXE="$HOME/src/20210719/$PROG/$PROG"
DATAROOT="/vol0003_cache/hp210225/u01868/data"
INDATA="${DATAROOT}/in"
OUTDATA="${DATAROOT}/out/${PROG}"
LOG="$PROG.log"
FILE_NO="0"
N_OBJS="3159"

time mpiexec -of ${LOG} -np ${N_OBJS} ${EXE} -i ${INDATA} -o ${OUTDATA} -n ${FILE_NO}
```

66 ノード

48 プロセス/ノード

(*)48 は最大数

最大 3,168 並列まで実行可能

3,159 プロセス指定

normalize 実行

```
$ mpiexec -np 3159 normalize -i 入力 -o 出力 -n 0
```

ジョブ投入

```
$ pjsub run.sh ⏎
```

ジョブ実行時間 (秒)

逐次処理	5,117.978
並列処理	252.951
GPU	296.721

実動180秒程度

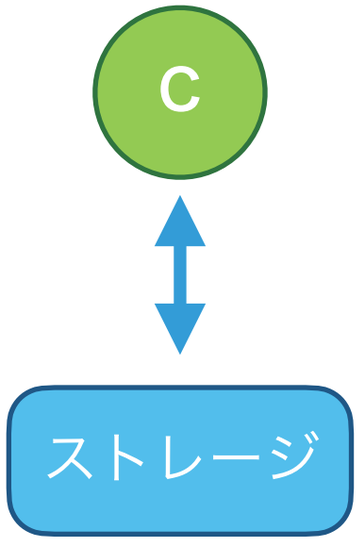
ファイル毎の平均処理時間 (秒)

	オブジェクト入力	規格化	ファイル出力
逐次処理	0.542	1.040	0.037
並列処理	79.892	1.109	2.473
GPU	0.052	0.001	0.040

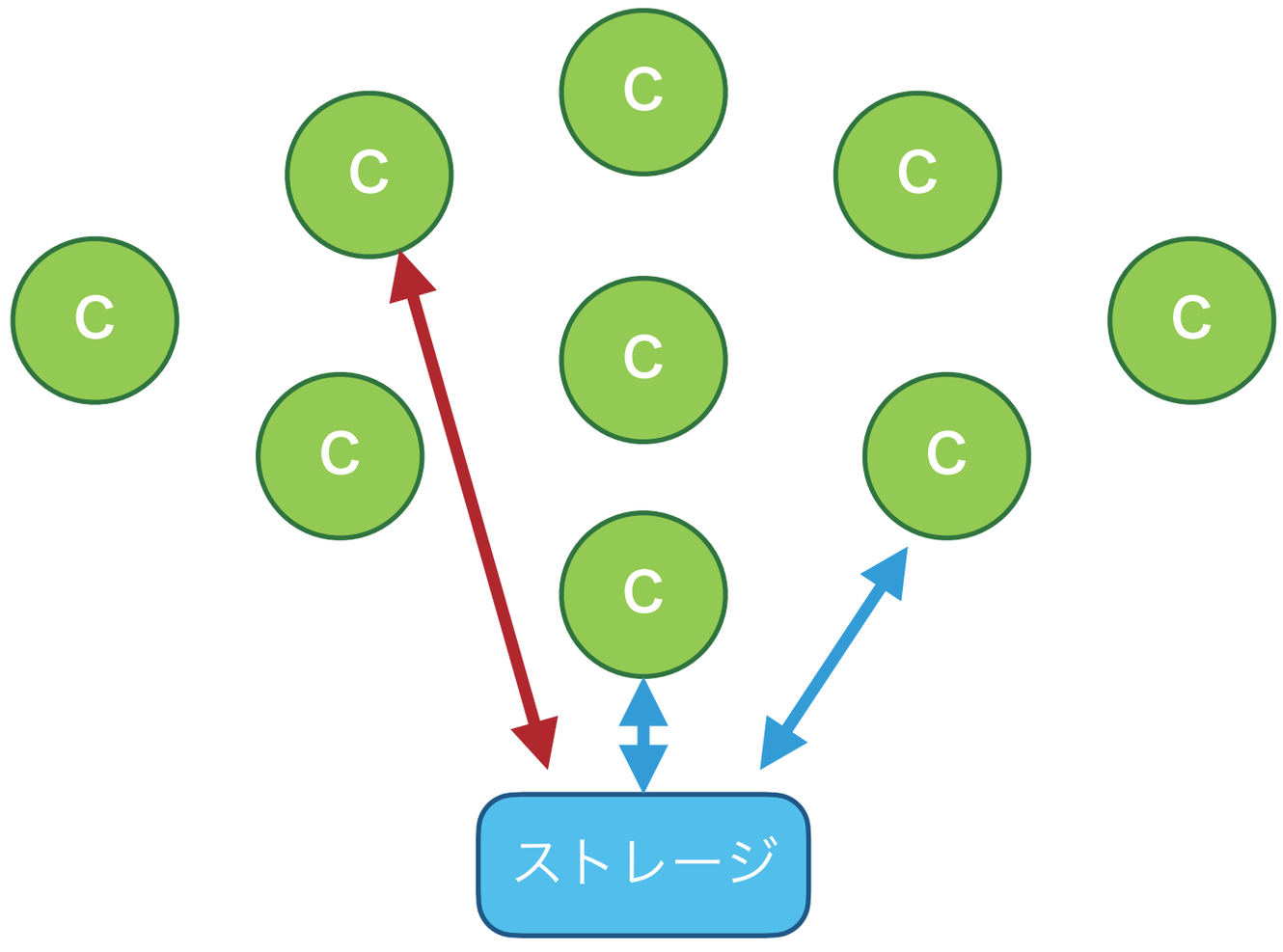
ストレージ ⇆ 計算ノード間通信

C 計算ノード

逐次処理

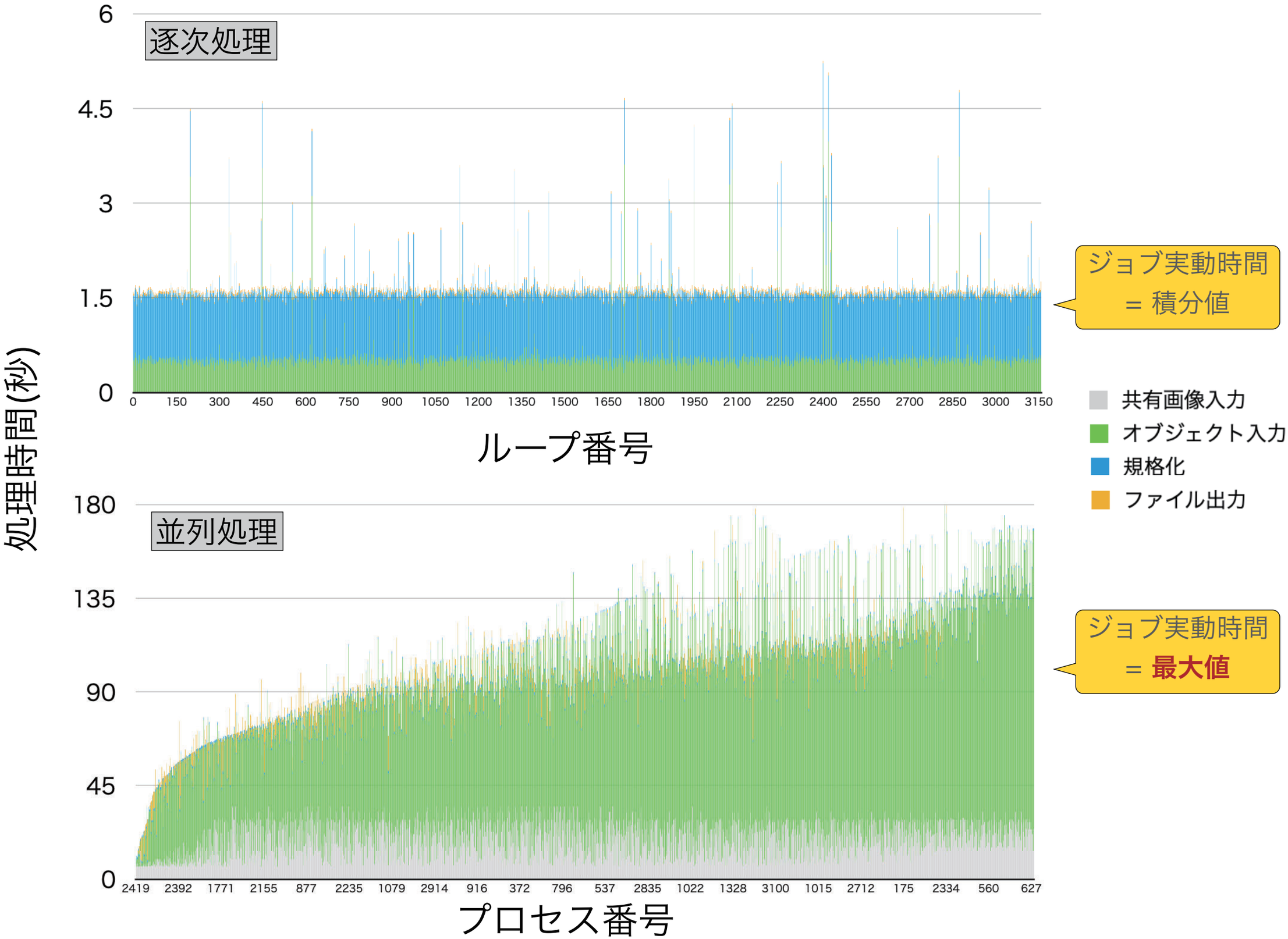


並列処理



"**遠い**" ノードもある

ファイル毎の処理時間の詳細



(参考) GPU環境

CPU	Intel Xeon Gold 5217 @ 3.0 GHz
メモリ	DDR4-2933 16 GB × 12
NVMe	Intel DC P4610 3.2 TB インターフェース: PCIe 3.1 × 4 シーケンシャルリード: 3,200 MB/s シーケンシャルライト: 3,050 MB/s
GPU	NVIDIA Quadro RTX 8000 インターフェース: PCIe 3.0 × 16 CUDA コア: 4,608 メモリ: GDDR6 48 GB FLOAT32 パフォーマンス: 16.3 TFLOPS
C++コンパイラ (兼 GPUバックエンド)	Intel C++ Compiler (icc) 19.0
GPUフロントエンド	CUDA Compilation tools (nvcc) 10.2 (Compute Capability 7.5)

「富岳」課題実施の準備

RISTとの技術相談

http://www.rist.or.jp/

- **必要なリソース: メモリ, マシンタイム等**

  「処理イメージ ↔ 『富岳』アーキテクチャ」擦り合わせ

- **必要なライブラリ: TIFF 等**

  メモリ自動解放機能
  [今回] C言語用ガベージコレクション lib 等の新規導入

- **支援スタッフの決定**

  手続きの窓口, 技術アドバイス, プロトタイピング支援
  (Makefile, ジョブスクリプト等「富岳」固有の設定)

HPCI経由の課題申請
https://www.hpci-office.jp/

「富岳」一般試行課題(随時募集)

- 動作検証や性能評価を試行する課題枠
- 期間 6ヶ月, 無償
- 期間中, 申請タイムを自由に采配
- HPCI交付アカウントでシングルサインオン

多くの計算資源利用を一本化

シングルサインオン電子認証

https://portal.hpci.nii.ac.jp/auth-portal/

代理証明書 + パスフレーズ

週一で再発行

「富岳」への接続と
データ転送

シングルサインオンの準備

ログインツール：**Docker イメージ**で入手可能

https://www.hpci.nii.ac.jp/software/ からダウンロード

コマンドラインから

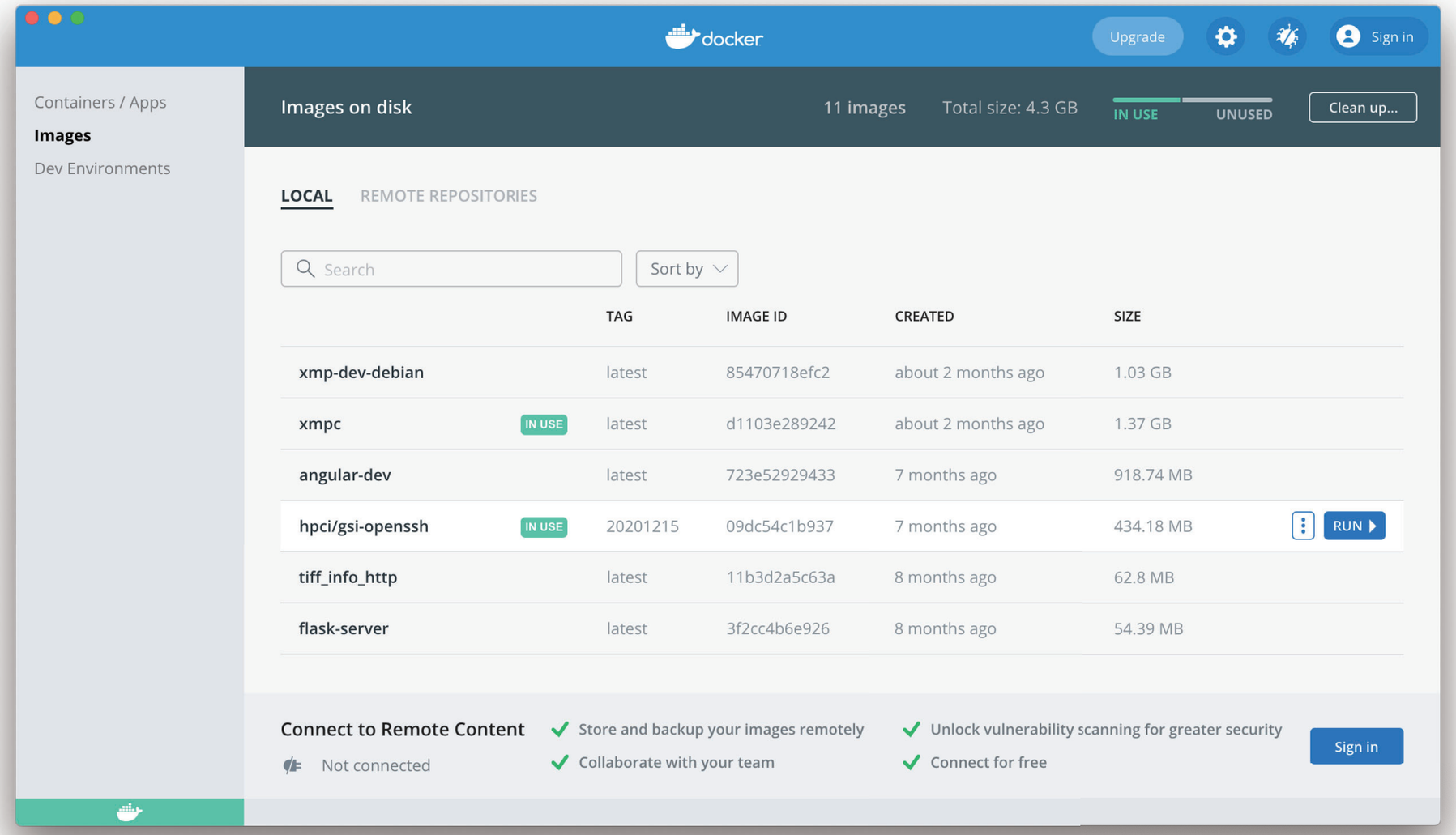
```
# docker load -i gsi-openssh-20201215.tar.bz2
```

Docker (macOSスクリーンショット)

https://www.docker.com/

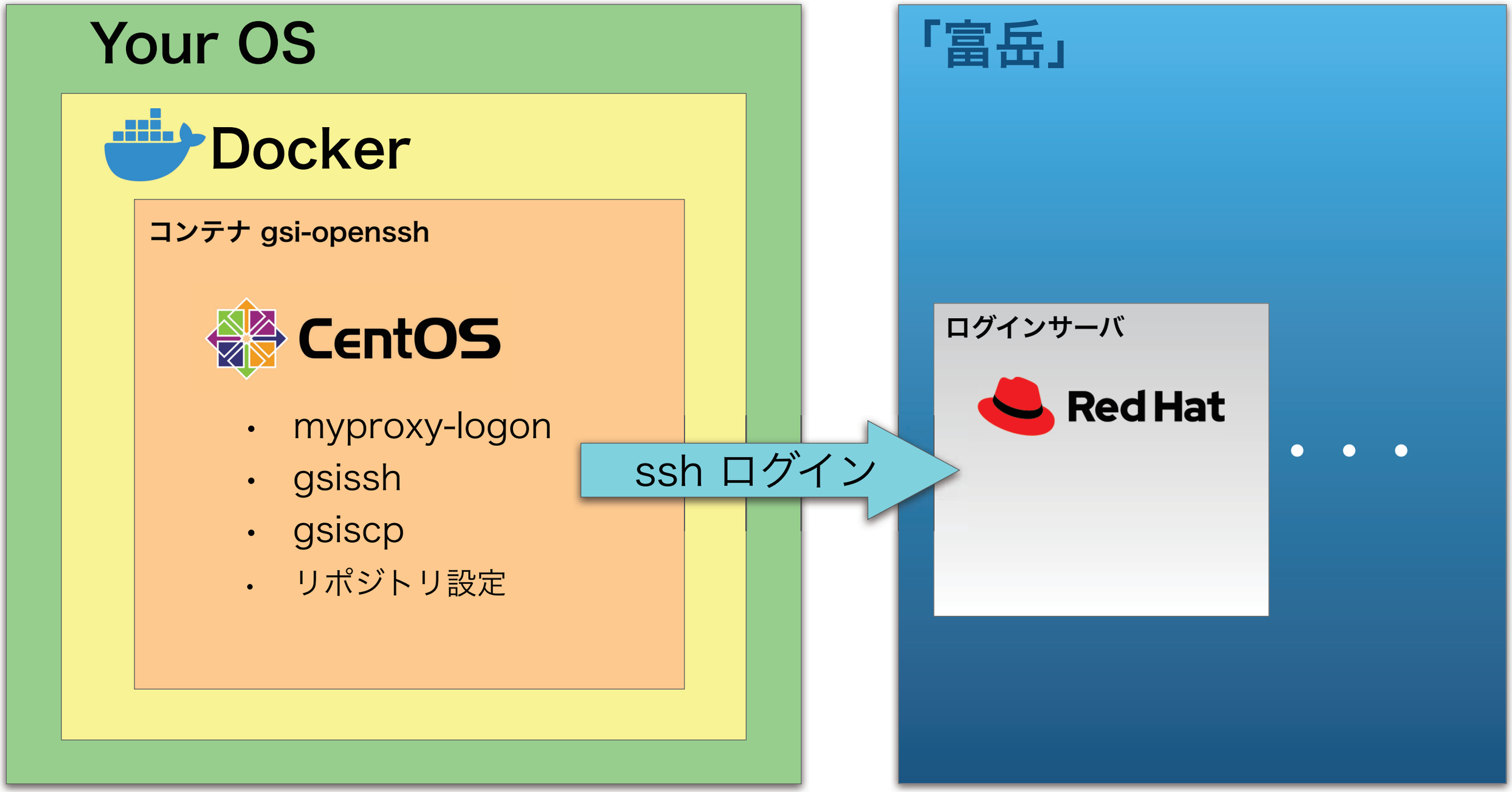
・OS機能ごとインストール ➡ 依存関係に悩まない

・ホストOS環境を荒らさない



(*)GUIは用意されているが, CLIが基本

Docker コンテナ ⇆「富岳」



Docker コンテナの立ち上げ

インストールされたイメージ名：hpci/gsi-openssh

コンテナの実行

```
# docker run -d --name gsi-openssh hpci/gsi-openssh:20201215
```

コンテナに入る

```
# docker exec -it gsi-openssh /bin/bash
```

コンテナにログインした状態 ➡

コンテナから「富岳」へログイン

週一で再発行

代理証明書ダウンロード (myproxy-logon コマンド)

```
$ myproxy-logon -s portal.hpci.nii.ac.jp -l USERNAME ↩
Enter MyProxy pass phrase:****************
A credential has been received for user USERNAME in /tmp/x509up_u2000.
```

ログイン (gsissh コマンド)

```
$ gsissh -p 2222 login.fugaku.r-ccs.riken.jp ↩
Last login: Tue Jul  6 10:32:26 2021 from ***.***.***.***
login1$
```

データ転送 (gsiscp コマンド)

```
$ gsiscp -P 2222 ~/data/o0001.tif login.fugaku.r-ccs.riken.jp:~/ ↩
                                              100%   23MB   8.8MB/s   9.5MB/s   00:05
```

データ転送の準備

クライアント環境

ホストOS	Linux
NVMe	PCIe3.1×4, 1132 MB/s(実測値)
ネットワーク	1 GbE
ホストOSデータパス	/home/public/data/20210810/lightning_sbm3x20
コンテナOSデータパス	/home/hpciuser/work/20210810/lightning_sbm3x20

マウント

```
$ docker run -d --name gsi-openssh ↙
 -v /home/public/data:/home/hpciuser/work ↙
 hpci/gsi-openssh:20201215 ↩
```

データ転送の速度評価(1)

```
$ gsiscp -P 2222 -r lightning_sbm3x20/ ↙
 login.fugaku.r-ccs.riken.jp:~/data/ ↩
```

→ ~/data/lightning_sbm3x20/ にコピー

転送時間：771 秒 (93 MB/s)

データ転送の速度評価(2)

クライアント環境

OS	Windows10
HDD	SATA, 4 TB, 7,200 rpm, 128 MB キャッシュ
ネットワーク	1 GbE

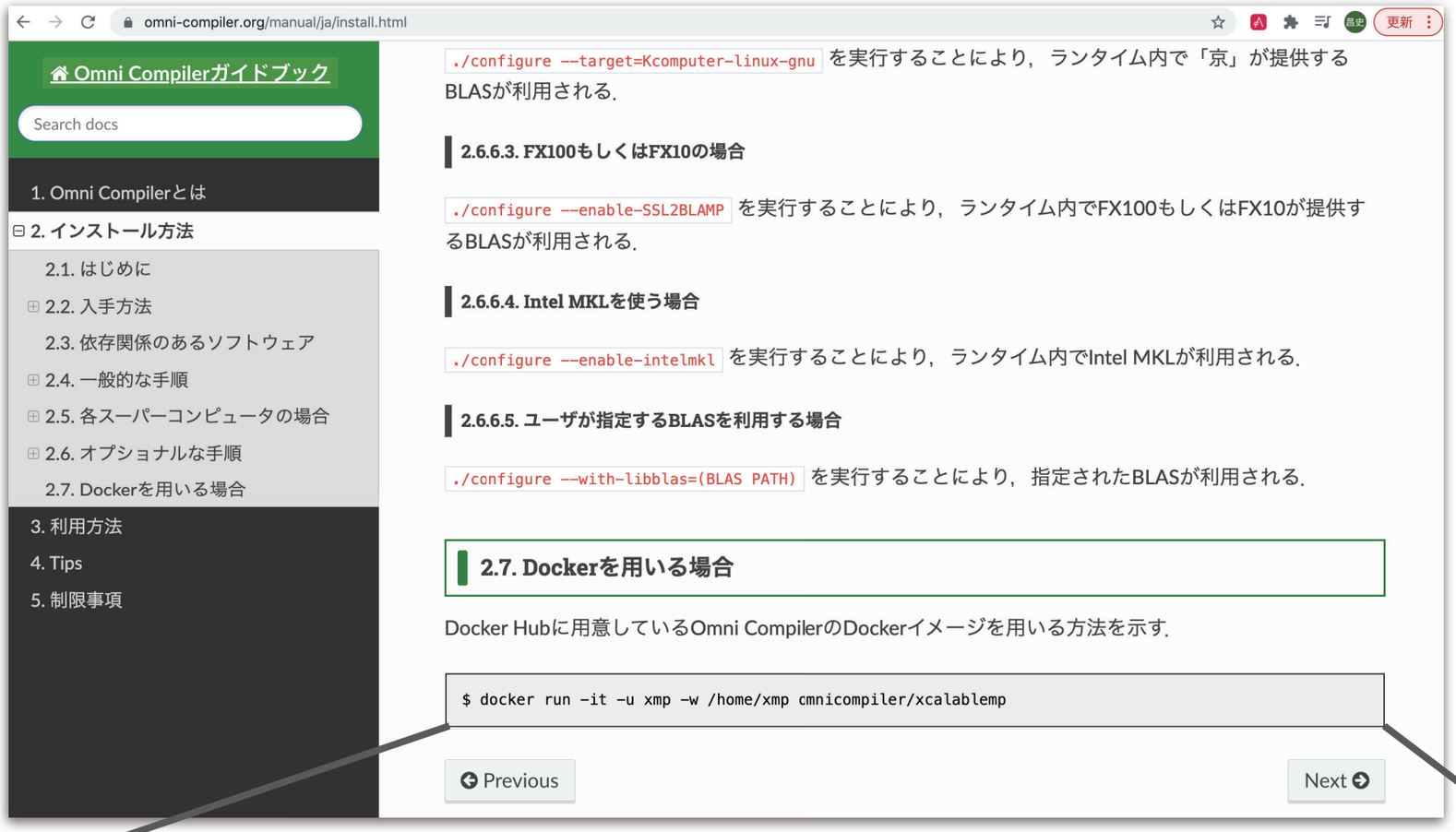
転送時間：2,556 秒 (28 MB/s)

オンプレミス開発環境

XcalableMPのインストール

Omni Compiler

https://omni-compiler.org/manual/ja/




```
$ docker run -it -u xmp -w /home/xmp omnicompiler/xcalablemp
```

Docker イメージで入手可能

Makefile と実行の例

Makefile

```
OUTPUT = normalize
SRC = main.c
CXX = xmpcc

CXXLIBS = -ltiff -lgc -lm

all : ${OUTPUT}

${OUTPUT} : ${SRC}
	@echo "##### Making" $@ "..."
	${CXX} ${CXXFLAGS} ${CXXINCLUDES} ${SRC} -o ${OUTPUT} ${CXXLIBPATH} ${CXXLIBS}
	@echo "##### done."

clean :
	rm -rf *.o ${OUTPUT}
```

XMPコンパイラ:
xmpcc

```
$ mpiexec -np 2 normalize -i ~/data/in -o ~/data/out -n 0
```

プロセス数指定

まとめ

- **イメージングの大規模データ処理を「富岳」で高速化**
  - 並列化により約**20倍** (1.4h → 4.2m / 70GB)

- **移行コストが低い**
  - 既知環境（オンプレミスのCPU環境）との親和性 (メモリ管理, ファイルI/Oなど)
  - XcalableMPの利用 ➡ 最小限の改造

- **RISTスタッフによる開発支援**
  - ライブラリ準備等の迅速な開発環境整備
  - 技術アドバイスによる最適化 (コード, リソース利用)
  - プロトタイピング支援 (Makefile, ジョブスクリプト)