

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

## [Oakforest-PACS(OFP)編]

第85回 お試しアカウント付き  
並列プログラミング講習会  
「ライブラリ利用：科学技術計算の効率化入門」

スパコンへのログイン・  
テストプログラム起動

東京大学情報基盤センター 特任准教授 伊田 明弘

---

# スパコンへのログイン・ ファイル転送・基本コマンド

# Oakforest-PACSへログイン

---

- ▶ ターミナルから、以下を入力する  
`$ ssh ofp.jcahpc.jp -l txxxxx`  
「-l」はハイフンと小文字のL、  
「txxxxx」は利用者番号
- ▶ 接続するかと尋ねられるので、 `yes` を入力する
- ▶ 鍵の設定時に入れた  
自分が決めたパスワード(パスフレーズ)  
を入力する
- ▶ 成功すると、ログインができる

# Oakforest-PACSのデータをPCに取り込む

---

- ▶ ターミナルでscpコマンドを使う
- ▶ `$ scp txxxxx@ofp.jcahpc.jp:~/a.f90 ./`
  - 「txxxxx」は利用者番号
  - ▶ OFP上のホームディレクトリにある `a.f90` をPCのカレントディレクトリに取ってくる
  - ▶ ディレクトリごとと取ってくる場合は `-r` を指定する
- ▶ `$ scp -r txxxxx@ofp.jcahpc.jp:~/SAMP ./`
  - ▶ OFP上のホームディレクトリにあるSAMPフォルダを、その中身ごと、PCのカレントディレクトリに取ってくる

# PCのファイルをOakforest-PACSに置く

---

- ▶ 同様にターミナルでscpコマンドを使う
- ▶ `$ scp ./a.f90 txxxxx@ofp.jcahpc.jp:`  
「txxxxx」は利用者番号
  - ▶ PCのカレントディレクトリにある `a.f90` を、OFP上のホームディレクトリに置く
  - ▶ ディレクトリごと置くには、`-r` を指定する
- ▶ `$ scp -r ./SAMP txxxxx@ofp.jcahpc.jp:`
  - ▶ PCのカレントディレクトリにあるSAMPフォルダを、その中身ごと、OFP上のホームディレクトリに置く

# Oakforest-PACSにおける注意

---

- ▶ **/home** ファイルシステムは容量が小さく、ログインに必要なファイルだけを置くための場所です。
  - ▶ **/home** に置いたファイルは計算ノードから参照できません。ジョブの実行もできません。
- ▶ 転送が終わったら、**/work**に移動(mv)してください。
- ▶ または、直接 **/work**を指定して転送してください。
  
- ▶ ホームディレクトリ: **/home**/gt00/txxxxx
  - ▶ **cd** コマンドで移動できます。
- ▶ Workディレクトリ: **/work**/gt00/txxxxx

# UNIX備忘録

---

- ▶ emacsの起動: emacs 編集ファイル名
  - ▶  $\hat{x} \hat{s}$  ( $\hat{\phantom{x}}$ はcontrol) : テキストの保存
  - ▶  $\hat{x} \hat{c}$  : 終了  
( $\hat{z}$  で終了すると、スパコンの負荷が上がる。絶対にしないこと。)
  - ▶  $\hat{g}$  : 訳がわからなくなったとき。
  - ▶  $\hat{k}$  : カーソルより行末まで消す。  
消した行は、一時的に記憶される。
  - ▶  $\hat{y}$  :  $\hat{k}$ で消した行を、現在のカーソルの場所にコピーする。
  - ▶  $\hat{s}$  文字列 : 文字列の箇所まで移動する。
  - ▶  $\hat{M} x$  goto-line : 指定した行まで移動する。

# UNIX 備忘録

---

- ▶ **rm** **ファイル名** : ファイル名のファイルを消す。
  - ▶ **rm \*~** : test.c~ などの、~がついたバックアップファイルを消す。使う時は慎重に。\*~ の間に空白が入ってしまうと、全てが消えます。
- ▶ **ls** : 現在いるフォルダの中身を見る。
- ▶ **cd** **フォルダ名** : フォルダに移動する。
  - ▶ **cd ..** : 一つ上のフォルダに移動。
  - ▶ **cd ~** : ホームディレクトリに行く。訳がわからなくなったとき。
- ▶ **cat** **ファイル名** : ファイル名の中身を見る
- ▶ **make** : 実行ファイルを作る  
(Makefile があるところでしか実行できない)
  - ▶ **make clean** : 実行ファイルを消す。  
(clean がMakefileで定義されていないと実行できない)



# UNIX備忘録

---

- ▶ **less** **ファイル名** : ファイル名の中身を見る(catでは画面がいっぱいになってしまうとき)
  - ▶ **スペースキー** : 1画面スクロール
  - ▶ **/** : 文字列の箇所まで移動する。
  - ▶ **q** : 終了 (訳がわからなくなったとき)
- ▶ **cp** **ファイル名** **フォルダ名** : ファイルをコピーする
- ▶ **mv** **ファイル名** **フォルダ名** : ファイルを移動させる

---

# テストプログラムのコンパイルと実行 [Oakleaf-FX編]

---

# サンプルプログラムのコンパイル

# サンプルプログラム名

---

- ▶ C言語版・Fortran90版共通ファイル:  
**Samples-OFP.tar**
- ▶ tarで展開後、C言語とFortran90言語のディレクトリが作られる
  - ▶ **C/** : C言語用
  - ▶ **F/** : Fortran90言語用
- ▶ 上記のファイルが置いてある場所  
**/work/gt00/z30107 (/homeでないので注意)**

# 並列版Helloプログラムをコンパイルしよう (1/2)

---

1. **Workディレクトリ(/work/gt00/txxxxx)に移動する**
2. /work/gt00/z30107 にある Samples-OFP.tar を  
自分のディレクトリにコピーする  
`$ cp /work/gt00/z30107/Samples-OFP.tar ./`
3. Samples-fx.tar を展開する  
`$ tar xvf Samples-OFP.tar`
4. Samples フォルダに入る  
`$ cd Samples`
5. C言語 : `$ cd C`  
Fortran90言語 : `$ cd F`
6. Hello フォルダに入る  
`$ cd Hello`

# 並列版Helloプログラムをコンパイルしよう (2/2)

---

6. make する

```
$ make
```

7. 実行ファイル(hello)ができていることを確認  
する

```
$ ls
```

---

# サンプルプログラムの実行

# Oakforest-PACSスーパーコンピュータシステムでのジョブ実行形態

- ▶ 以下の2通りがあります
- ▶ **インタラクティブジョブ実行**
  - ▶ PCでの実行のように、コマンドを入力して実行する方法
  - ▶ スパコン環境では、あまり一般的でない
  - ▶ デバック用、大規模実行はできない
  - ▶ OFPでは、以下に限定
    - ▶ 1ノード(68コア):2時間まで
    - ▶ 16ノード(1,088コア):10分まで
- ▶ **バッチジョブ実行**
  - ▶ バッチジョブシステムに処理を依頼して実行する方法
  - ▶ 実行させたい処理をファイル(ジョブスクリプト)で指示する
  - ▶ スパコン環境で一般的
  - ▶ 大規模実行用
  - ▶ OFPでは、最大2048ノード(139,264コア)(24時間)

※講習会アカウントでは  
バッチジョブ実行のみ、  
最大16ノード15分まで



# Oakforest-PACSスーパーコンピュータシステムでのジョブ実行形態(2)

---

- ▶ 2つの異なるメモリモードを用意
  - ▶ Flatモード
    - ▶ MCDRAMとDDR4メモリを個別にアクセス可能
  - ▶ Cacheモード
    - ▶ MCDRAMはDDR4メモリのキャッシュとして働く
- ▶ 各ジョブキューには、**-flat**, **-cache** をそれぞれ用意
  - ▶ 講習会アカウントでは、**Flatモードだけ**が使えます。

# インタラクティブ実行のやり方

## ▶ コマンドラインで以下を入力

### ▶ 1ノード実行用

```
$ pjsub --interact -g gt00 -L rg=interactive-  
{flat,cache},elapse=01:00
```

### ▶ 16ノード実行用

※コマンドは改行せず1行で入力すること

```
$ pjsub --interact -g gt00 -L rg=interactive-  
{flat,cache},node=16,elapse=01:00
```

※インタラクティブ用のノードがすべて使われている場合、  
資源が空くまで、ログインできません。  
※講習会アカウントでは使えません。

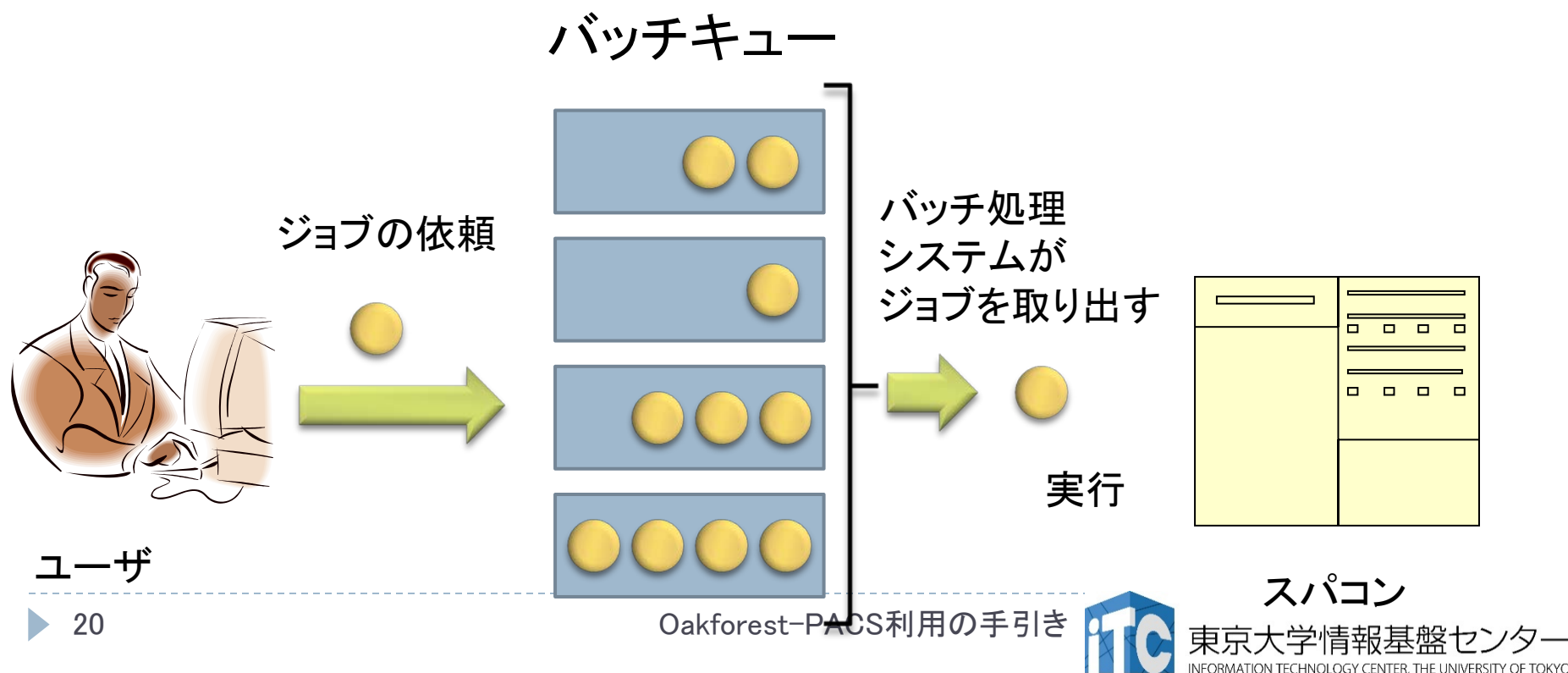
# コンパイラの種類とインタラクティブ実行およびバッチ実行

---

- ▶ OFPでは、コンパイラはバッチ実行、インタラクティブ実行で共通に使えます。
- ▶ 例) Intelコンパイラ
  - ▶ Cコンパイラ: `icc`, `mpiicc` (Intel MPIを使う場合)
  - ▶ Fortran90コンパイラ: `ifort`, `mpiifort` (Intel MPIを使う場合)
  - ▶ KNL向け最適化: `-xMIC-AVX512`
    - ▶ ログインノードやプレポスト用ノードで実行する可能性もある場合:  
`-axMIC-AVX512`

# バッチ処理とは

- ▶ スパコン環境では、通常は、インタラクティブ実行(コマンドラインで実行すること)はできません。
- ▶ ジョブは**バッチ処理**で実行します。



ユーザ

# バッチキューの設定のしかた

---

- ▶ OFPでのバッチ処理は、富士通のバッチシステムで管理されています。
- ▶ 以下、主要コマンドを説明します。
  - ▶ ジョブの投入: `pjsub` <ジョブスクリプトファイル名>
  - ▶ 自分が投入したジョブの状況確認: `pjstat`
  - ▶ 投入ジョブの削除: `pjdel` <ジョブID>
  - ▶ バッチキューの状態を見る: `pjstat --rsc`
  - ▶ バッチキューの詳細構成を見る: `pjstat --rsc -x`
  - ▶ 投げられているジョブ数を見る: `pjstat -b`
  - ▶ 過去の投入履歴を見る: `pjstat -H`
  - ▶ 同時に投入できる数／実行できる数を見る: `pjstat --limit`

# ジョブスクリプトの例

※実行させたい処理によって  
各項目の内容は異なります

```
#!/bin/bash
#PJM -L rscgrp=lecture-flat
#PJM -L node=2
#PJM --mpi proc=4
#PJM --omp thread=16
#PJM -L elapse=0:01:00
#PJM -g gt00

mpirun ./a.out
```

リソースグループ名  
:lecture-flat

利用ノード数

MPIプロセス数

プロセスあたりの  
スレッド数

実行時間制限  
:1分

利用グループ名  
:gt00

プログラムを実行

# 本お試し講習会でのキュー・グループ名

---

- ▶ **本演習中のキュー名:**
  - ▶ tutorial-flat
  - ▶ 最大10分まで
  - ▶ 最大ノード数は16ノード(1088コア)まで
  
- ▶ **本演習時間以外(24時間)のキュー名:**
  - ▶ lecture-flat
  - ▶ 利用条件は演習中のキュー名と同様
  
- ▶ **グループ名: gt00**

# pjstat --rsc の実行画面例

```
$ pjstat --rsc
RSCGRP                                STATUS                                NODE
regular-cache
|---- small-cache                    [ENABLE,START]                       3846
`---- medium-cache                   [ENABLE,START]                       3846
regular-flat
|---- small-flat                     [ENABLE,START]                       3846
`---- medium-flat                   [ENABLE,START]                       3846
interactive-cache
|---- interactive_n1-cache          [ENABLE,START]                       100
`---- interactive_n16-cache        [ENABLE,START]                       100
interactive-flat
|---- interactive_n1-flat          [ENABLE,START]                       100
`---- interactive_n16-flat        [ENABLE,START]                       100
debug-cache                          [ENABLE,START]                       234
debug-flat                            [ENABLE,START]                       234
prepost                              [ENABLE,START]                       12
```

使える  
キュー名  
(リソース  
グループ)

現在  
使えるか

ノードの  
利用可能数



# pjstat --rsc -x の実行画面例

```

$ pjstat --rsc -x
RSCGRP                STATUS                MIN_NODE  MAX_NODE  MAX_ELAPSE  REMAIN_ELAPSE  MEM(GB)  PROJECT
regular-cache
|---- small-cache     [ENABLE,START]        1         128      48:00:00     48:00:00       82 pz0105
`---- medium-cache    [ENABLE,START]       129        512      48:00:00     48:00:00       82 pz0105
regular-flat
|---- small-flat      [ENABLE,START]        1         128      48:00:00     48:00:00       96 pz0105
`---- medium-flat     [ENABLE,START]       129        512      48:00:00     48:00:00       96 pz0105
interactive-cache
|---- interactive_n1-cache [ENABLE,START]        1          1      02:00:00     02:00:00       82 pz0105
`---- interactive_n16-cache [ENABLE,START]        2          16      00:10:00     00:10:00       82 pz0105
interactive-flat
|---- interactive_n1-flat [ENABLE,START]        1          1      02:00:00     02:00:00       96 pz0105
`---- interactive_n16-flat [ENABLE,START]        2          16      00:10:00     00:10:00       96 pz0105
debug-cache           [ENABLE,START]        1         128      00:30:00     00:30:00       82 pz0105
debug-flat            [ENABLE,START]        1         128      00:30:00     00:30:00       96 pz0105
prepost              [ENABLE,START]        1          1      06:00:00     06:00:00      222 pz0105

```

↑  
 使える  
 キュー名  
 (リソース  
 グループ)

↑  
 現在  
 使えるか

↑  
 ノードの  
 実行情報

↑  
 課金情報(財布)  
 実習では1つのみ

# pjstat --rsc -b の実行画面例

```
$ pjstat --rsc -b
```

RSCGRP	STATUS	TOTAL	RUNNING	QUEUED	HOLD	OTHER	NODE
regular-cache							
---- small-cache	[ENABLE,START]	45	40	5	0	0	3846
`---- medium-cache	[ENABLE,START]	1	1	0	0	0	3846
regular-flat							
---- small-flat	[ENABLE,START]	150	120	30	0	0	3846
`---- medium-flat	[ENABLE,START]	7	3	4	0	0	3846
interactive-cache							
---- interactive_n1-cache	[ENABLE,START]	0	0	0	0	0	100
`---- interactive_n16-cache	[ENABLE,START]	0	0	0	0	0	100
interactive-flat							
---- interactive_n1-flat	[ENABLE,START]	1	1	0	0	0	100
`---- interactive_n16-flat	[ENABLE,START]	0	0	0	0	0	100
debug-cache	[ENABLE,START]	7	4	3	0	0	234
debug-flat	[ENABLE,START]	0	0	0	0	0	234
prepost	[ENABLE,START]	0	0	0	0	0	12

使える  
キュー名  
(リソース  
グループ)

現在  
使えるか

ジョブ  
の総数

実行して  
いるジョブ  
の数

待たされて  
いるジョブ  
の数

ノードの  
利用可能  
数

## 並列版Helloプログラムを実行しよう

---

- ▶ このサンプルのJOBスクリプトは `hello-pure.bash` です。
- ▶ 配布のサンプルでは、キューが”`lecture`”になっています
- ▶ `$ emacs hello-pure.bash` で、“`lecture`” → “`tutorial`” に変更してください

# 並列版Helloプログラムを実行しよう

---

1. Helloフォルダ中で以下を実行する  
`$ pjsub hello-pure.bash`
2. 自分の導入されたジョブを確認する  
`$ pjstat`
3. 実行が終了すると、以下のファイルが生成される  
`hello-pure.bash.eXXXXXXXX`  
`hello-pure.bash.oXXXXXXXX` (XXXXXXXXは数字)
4. 上記の標準出力ファイルの中身を見してみる  
`$ cat hello-pure.bash.oXXXXXXXX`
5. “Hello parallel world!”が、  
64プロセス\*4ノード=256表示されていたら成功。

# バッチジョブ実行による標準出力、標準エラー出力

- ▶ バッチジョブの実行が終了すると、標準出力ファイルと標準エラー出力ファイルが、ジョブ投入時のディレクトリに作成されます。
- ▶ 標準出力ファイルにはジョブ実行中の標準出力、標準エラー出力ファイルにはジョブ実行中のエラーメッセージが出力されます。

ジョブ名.oXXXXXX --- 標準出力ファイル  
ジョブ名.eXXXXXX --- 標準エラー出力ファイル  
(XXXXXX はジョブ投入時に表示されるジョブのジョブID)

# 並列版Helloプログラムの説明 (C言語)

このプログラムは、全コアで起動される

```
#include <stdio.h>
#include <mpi.h>
```

```
int main(int argc, char* argv[]) {
```

```
    int  myid, numprocs;
    int  ierr, rc;
```

```
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

```
    printf("Hello parallel world! Myid:%d ¥n", myid);
```

```
    rc = MPI_Finalize();
```

```
    exit(0);
```

```
}
```

MPIの初期化

自分のID番号を取得  
:各コアで値は異なる

全体のプロセッサ台数  
を取得  
:各コアで値は同じ  
(演習環境では192)

MPIの終了

# 並列版Helloプログラムの説明 (Fortran言語)

このプログラムは、全コアで起動される

```
program main
```

```
common /mpienv/myid,numprocs
```

```
integer myid, numprocs  
integer ierr
```

```
call MPI_INIT(ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
```

```
print *, "Hello parallel world! Myid:", myid
```

```
call MPI_FINALIZE(ierr)
```

```
stop  
end
```

MPIの初期化

自分のID番号を取得  
:各コアで値は異なる

全体のプロセッサ台数  
を取得  
:各コアで値は同じ  
(演習環境では192)

MPIの終了

# 時間計測方法 (C言語)

```
double t0, t1, t2, t_w;  
..  
ierr = MPI_Barrier(MPI_COMM_WORLD);  
t1 = MPI_Wtime();
```

<ここに測定したいプログラムを書く>

```
ierr = MPI_Barrier(MPI_COMM_WORLD);  
t2 = MPI_Wtime();
```

```
t0 = t2 - t1;  
ierr = MPI_Reduce(&t0, &t_w, 1,  
    MPI_DOUBLE, MPI_MAX, 0,  
    MPI_COMM_WORLD);
```

バリア同期後  
時間を習得し保存

各プロセッサで、t0の値は異なる。  
この場合は、最も遅いものの値をプロセッサ0番が受け取る



# 時間計測方法 (Fortran言語)

```
double precision t0, t1, t2, t_w  
double precision MPI_WTIME
```

```
..  
call MPI_BARRIER(MPI_COMM_WORLD, ierr)  
t1 = MPI_WTIME(ierr)
```

<ここに測定したいプログラムを書く>

```
call MPI_BARRIER(MPI_COMM_WORLD, ierr)  
t2 = MPI_WTIME(ierr)
```

```
t0 = t2 - t1  
call MPI_REDUCE(t0, t_w, 1,  
& MPI_DOUBLE_PRECISION,  
& MPI_MAX, 0, MPI_COMM_WORLD, ierr)
```

バリア同期後  
時間を習得し保存

各プロセッサで、t0の値  
は異なる。

この場合は、最も遅いもの  
の値をプロセッサ0番  
が受け取る

---

おわり

お疲れさまでした