第138回お試しアカウント付き講習会
「スーパーコンピューター超入門」
2020年9月18日

利用実習編

本資料はWindowsマシン上で
PuTTYを使われる方向けのものです

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

2020/9/23 v2.0（修正版）

実習内容

1. スーパーコンピューターへのログイン
   初めてスパコンに入る

2. 簡単なプログラムの作成、ジョブの実行
   初めてスパコンでファイルを操作、
   初めてスパコンでプログラムをコンパイル、そして計算

3. サンプル並列プログラムの実行（積分計算）
   初めて並列化による計算の高速化を実感

4. 計算した結果を自分のコンピューターに転送（時間あれば）

スーパーコンピューター (スパコン) 利用のイメージ

通常、スパコンではログインノードと呼ばれる玄関口から実行命令を出します

端末

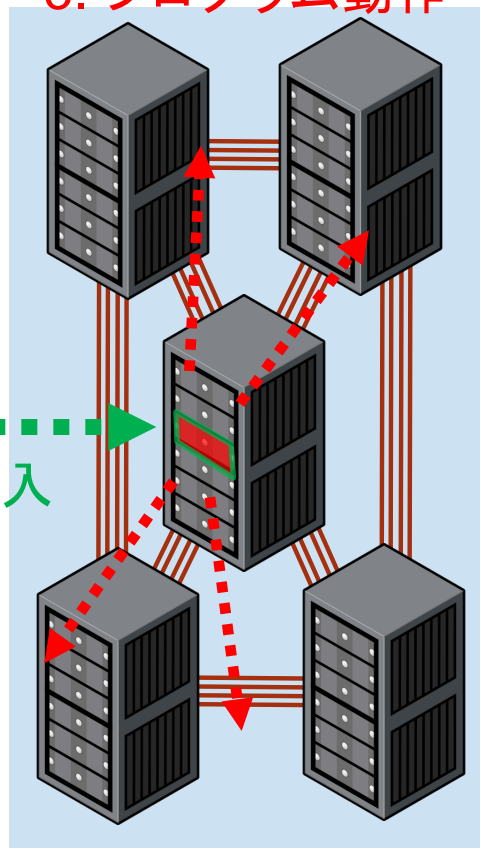
1. ログイン
SSH

ログインノード

```
[tut138@obcx02~]$ ls -l
drwxr-x--- 2 shiba group 10 1
Apr 13:00 test.out
[tut138@obcx02~]$ ./test.out
Hello world
[tut138@obcx02~]$ qsub a.sh
```

2. ジョブ投入

3. プログラム動作



計算ノード
= 本体

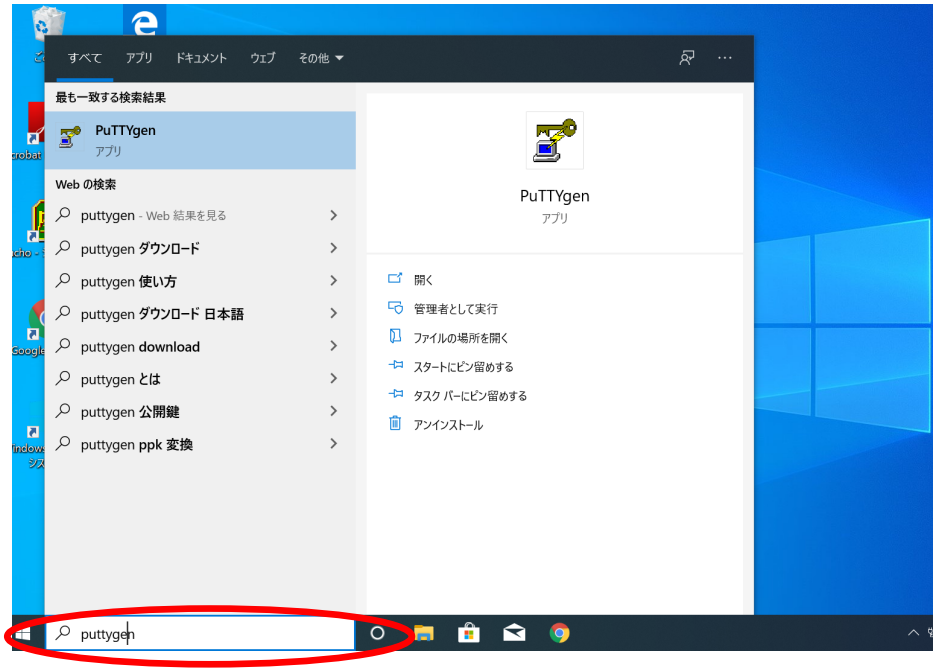
SSH で接続するには
鍵の準備が必要！

鍵の作成 (PuTTYの方)

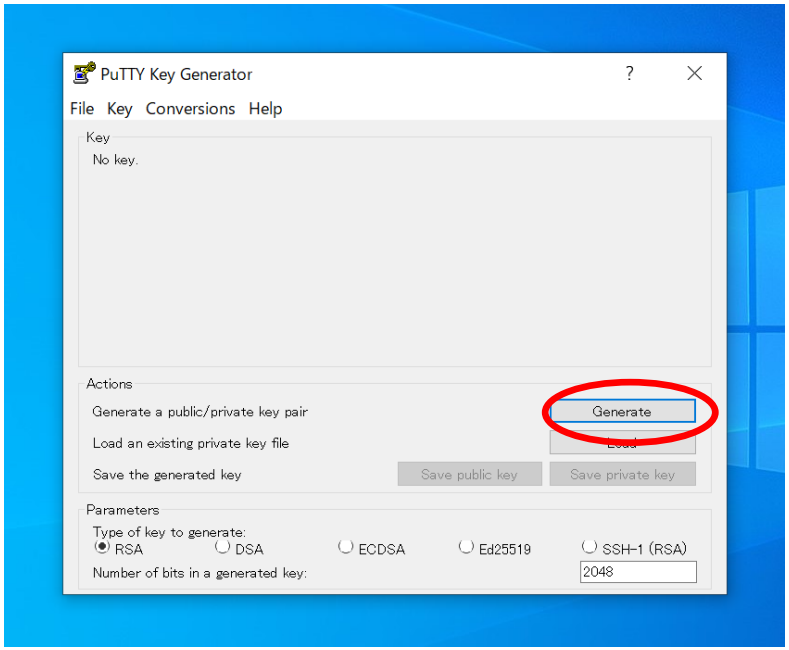
PuTTYgen を用いて公開鍵、秘密鍵を作成します

(1) PuTTYgen を起動

アイコンがなければ左下のメニューから検索



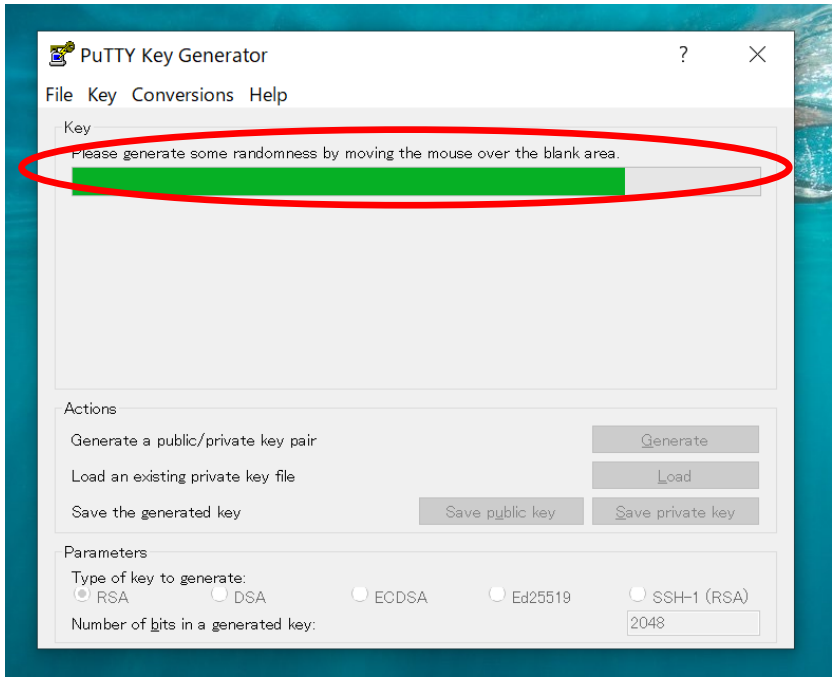
(2) Generate a public/private Key をクリック



鍵の作成 (PuTTYの方)

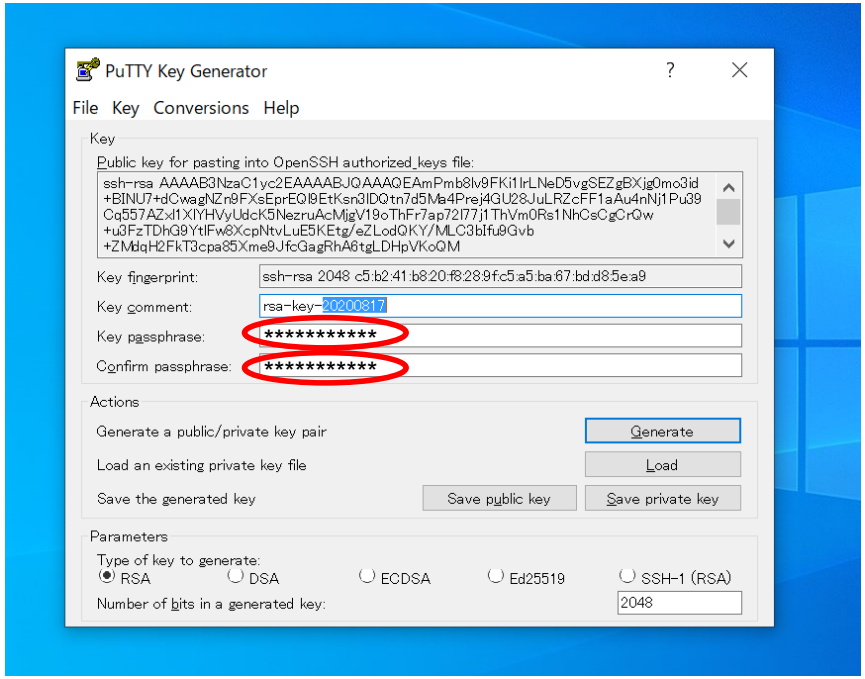
PuTTYgen を用いて公開鍵、秘密鍵を作成します

(3) 緑色のバーの近くで マウスを動かす



公開鍵、秘密鍵が作成されました

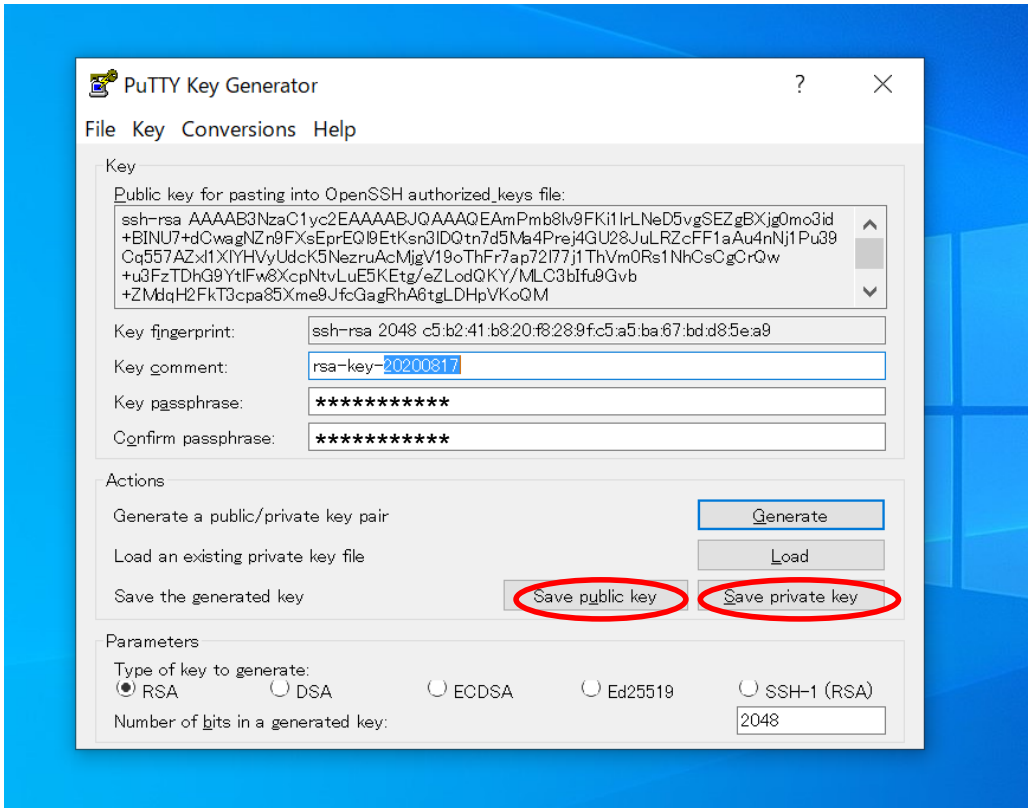
(4) 秘密鍵のパスフレーズを決める



鍵の作成 (PuTTYの方)

PuTTYgen を用いて公開鍵、秘密鍵を作成します

(5) 公開鍵と秘密鍵を保存 + 公開鍵をクリップボードにコピー



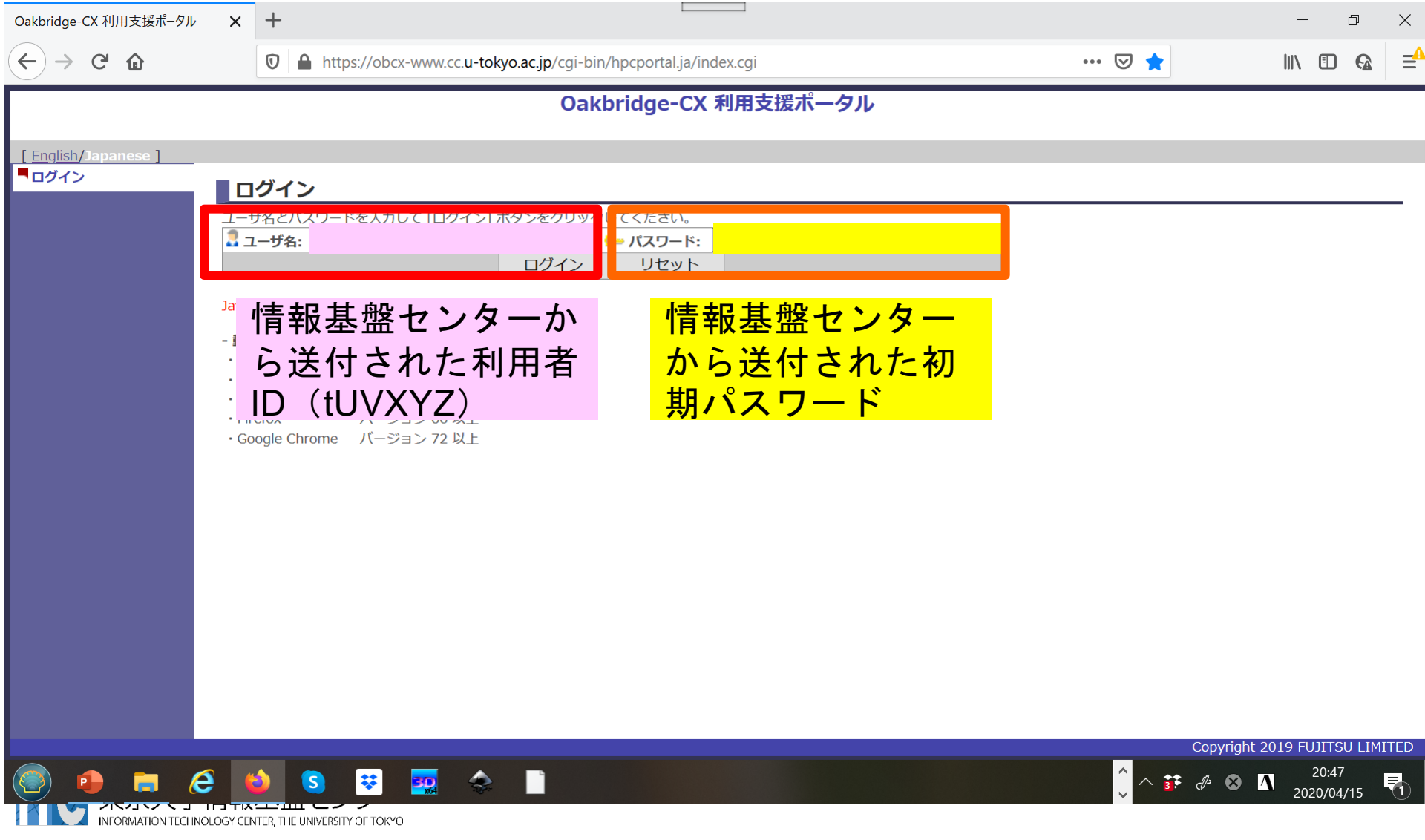
保存先ディレクトリ情報を忘れずにメモ。あとで使います。

# SSH Public Key Authentication
# SSH公開鍵認証
SSH= Secure Shell

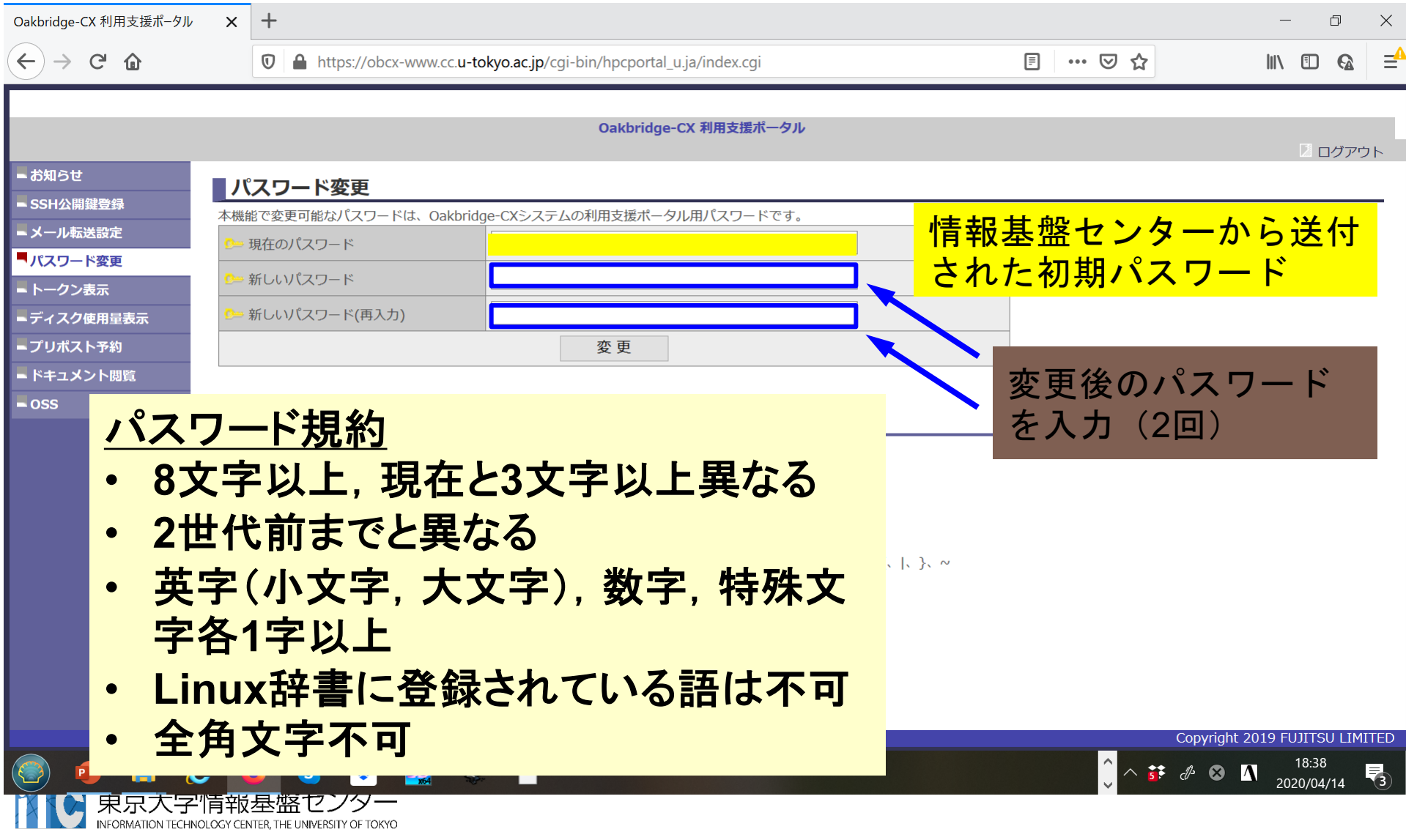
- id_rsa
  - Private Key（秘密鍵）：PC上
  - 文字通り「秘密」にしておくこと
    - 他の人に送ってはいけない
    - 基本的には作成した場所からコピーしたり移動することもしないこと
- id_rsa.pub
  - Public Key（公開鍵）：スパコン上
  - コピー可能，他の人にe-mailで送ることも可能
- もし複数のPCからスパコンにログインする場合は，各PCごとに「公開鍵・秘密鍵」のペアをssh-keygenによって作成
  - 各スパコンに複数の公開鍵を登録することは可能
  - スパコン上の公開鍵のうちの一つがPC上の「秘密鍵＋Passphrase」とマッチすると確認されるとログインできる

①スパコンポータルサイトにログイン

https://obcx-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal.ja/index.cgi



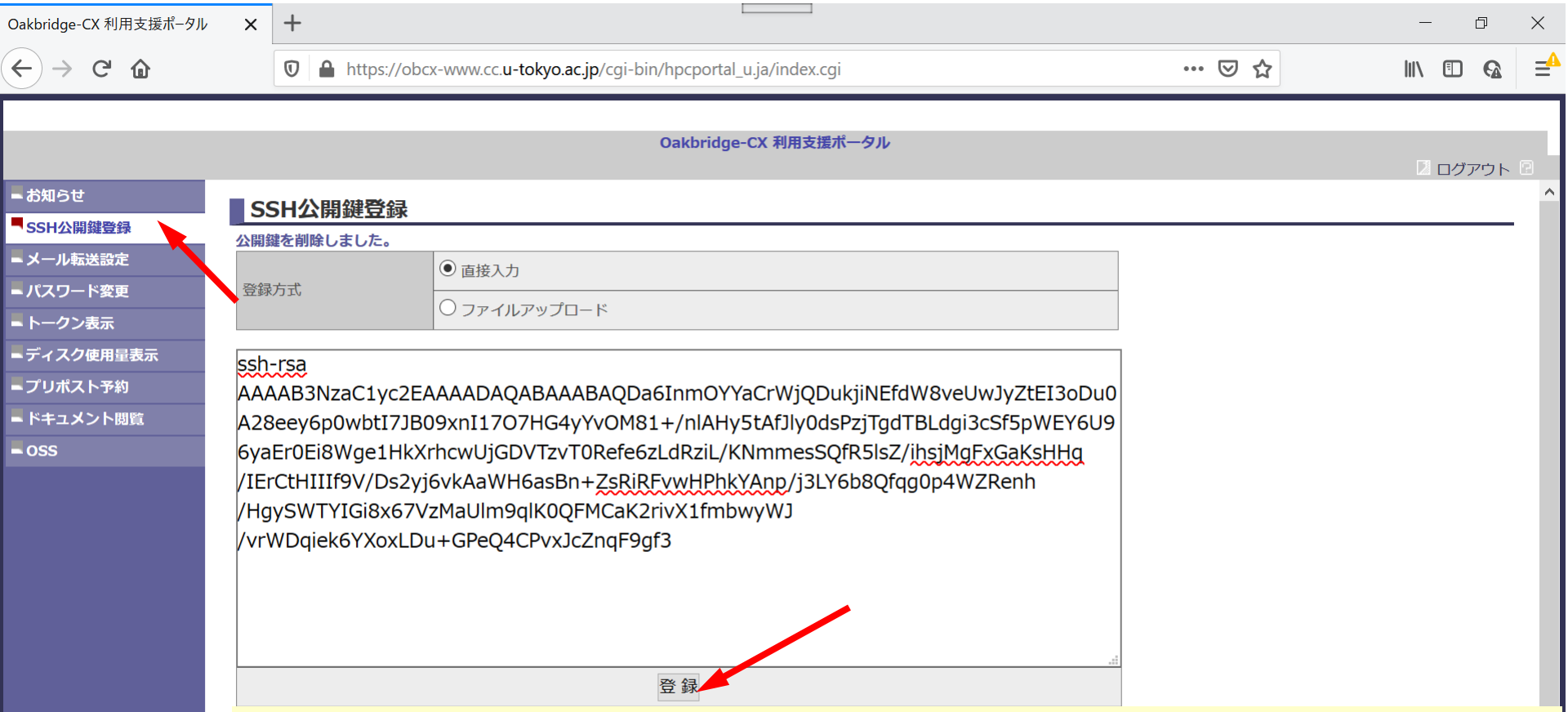
②初期パスワードの変更



パスワード規約

- 8文字以上，現在と3文字以上異なる
- 2世代前までと異なる
- 英字（小文字，大文字），数字，特殊文字各1字以上
- Linux辞書に登録されている語は不可
- 全角文字不可

③公開鍵登録（id_rsa.pub）

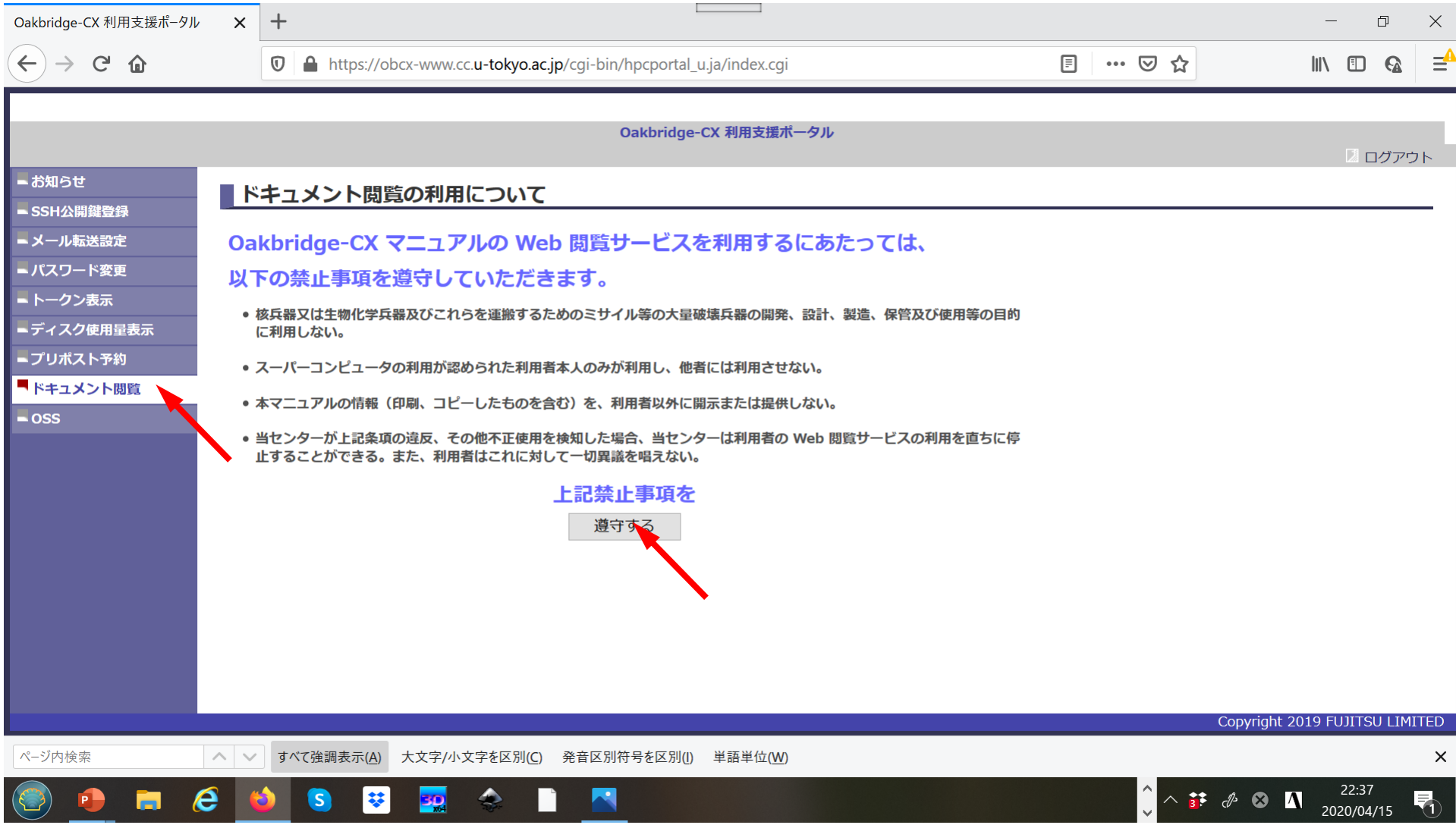


1. 「SSH公開鍵登録」を選択
2. 先ほどCopyした公開鍵（ id_rsa.pub ）を貼り付ける
3. 「登録」をクリック

よくトラブルが起きますので、先に進めなければ質問してください

スパコンには複数の公開鍵を登録できる

ポータルサイトでのマニュアル等閲覧（1/2）

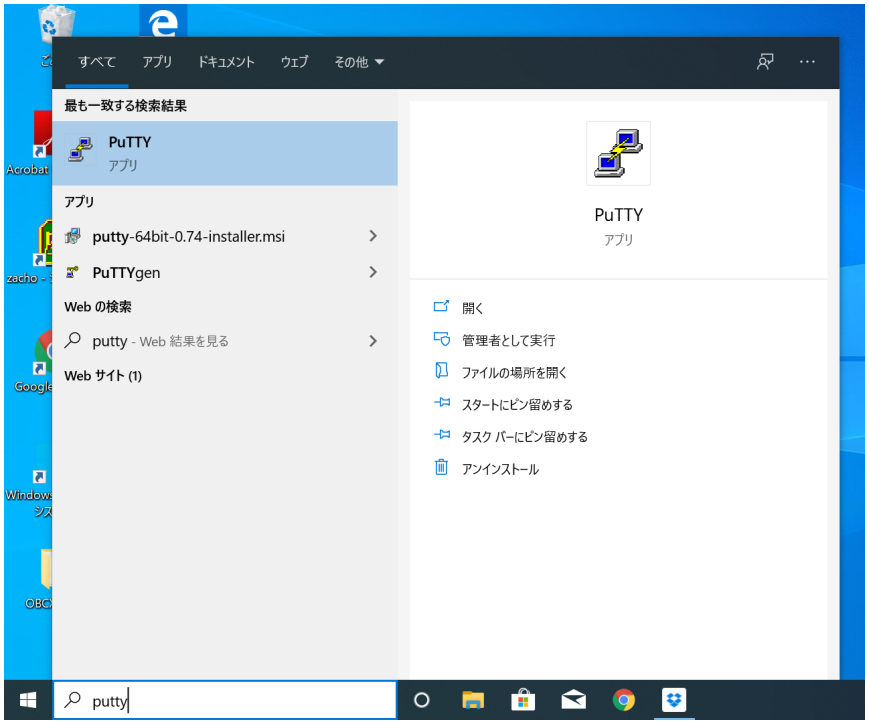


ポータルサイトでのマニュアル等閲覧（2/2）

スパコンにログイン

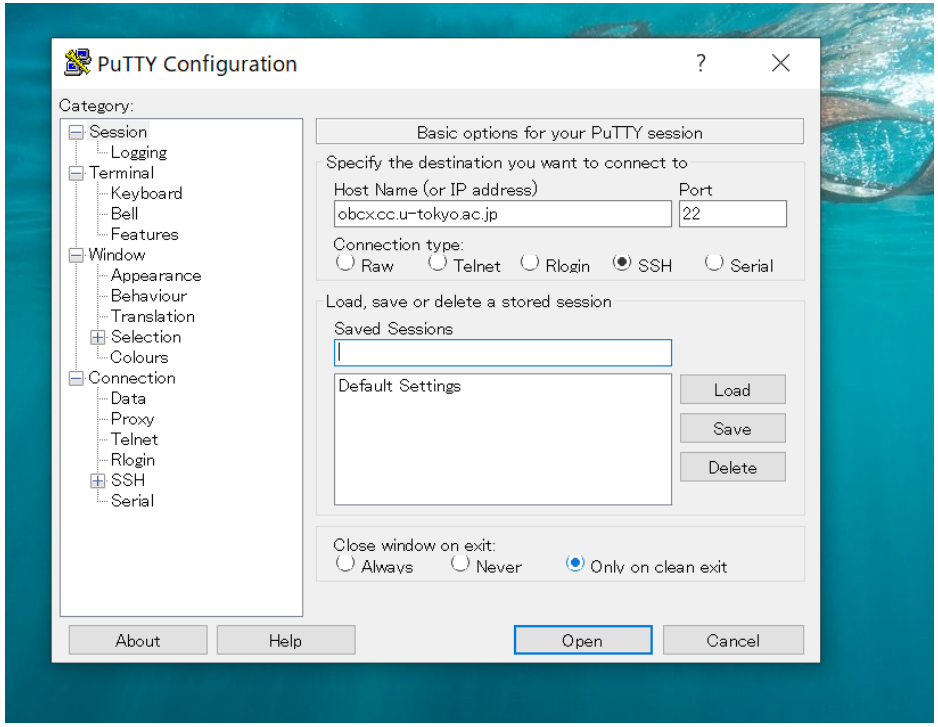
(1) PuTTYを起動

アイコンがなければ左下のメニューから検索



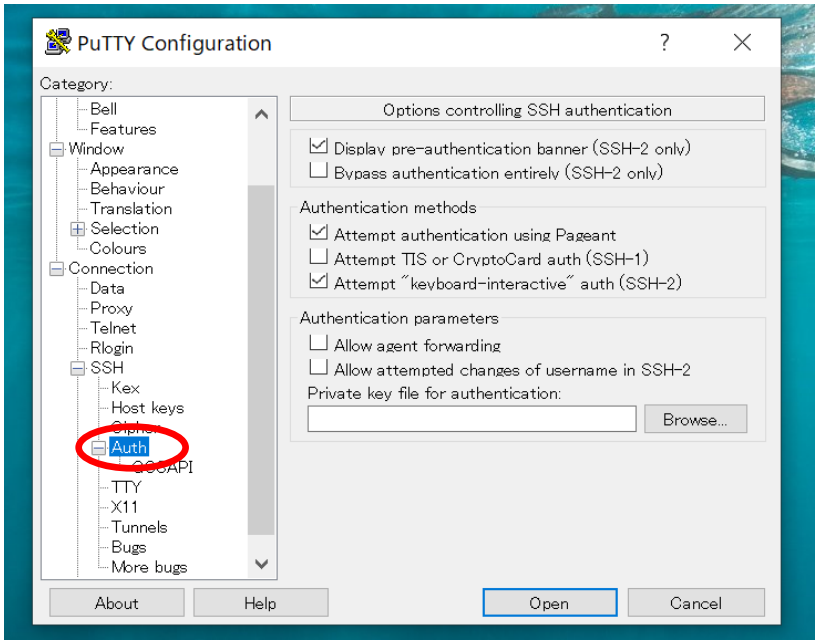
(2) 接続先 (Host Name) を設定

obcx.cc.u-tokyo.ac.jp

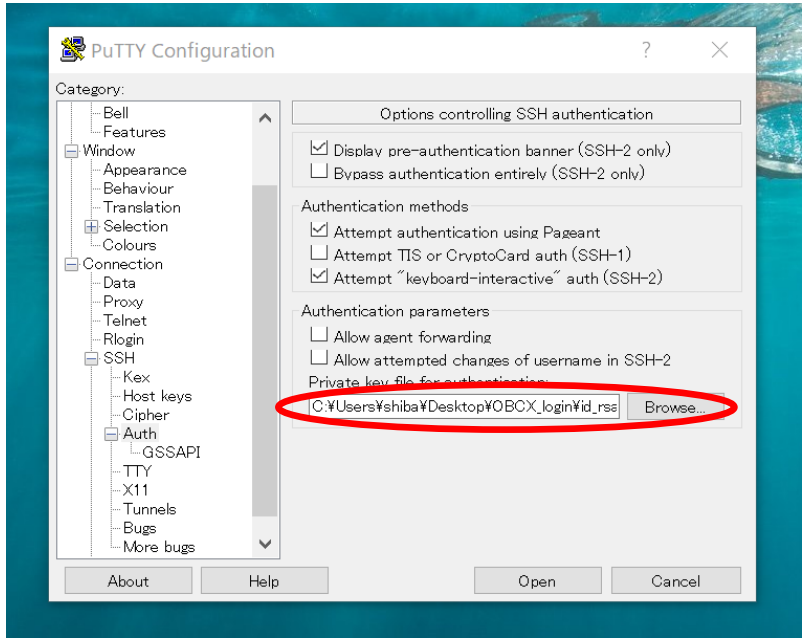


スパコンにログイン

(3)秘密鍵を指定する



Connection → SSH → Auth と移動



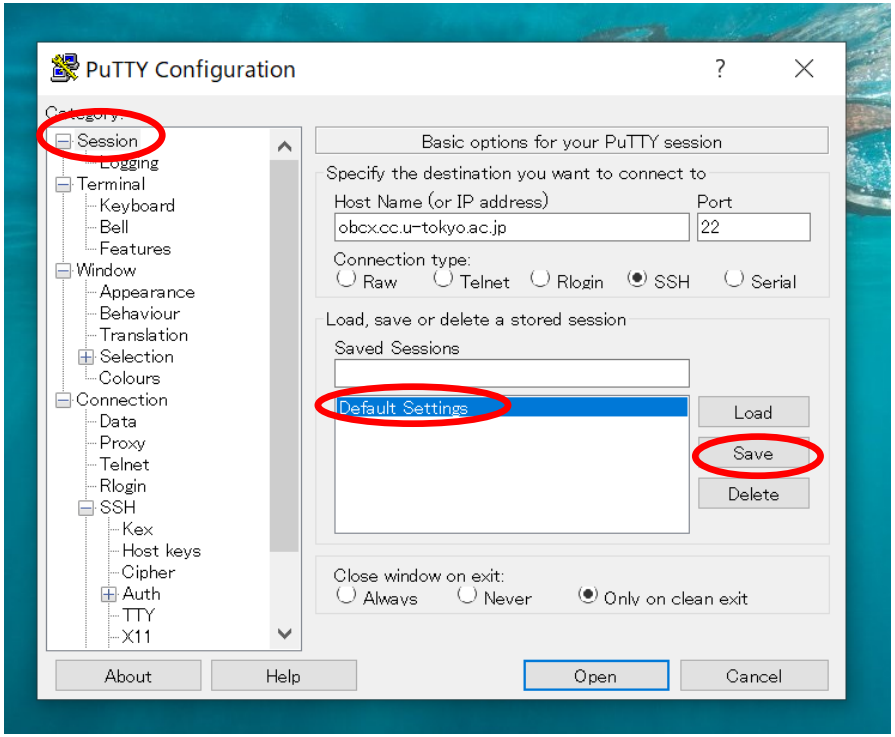
先ほど保存した秘密鍵を指定 “id_rsa”

スパコンにログイン

(4) 設定を保存

左上 “Session” へ移動

→ “Default Setting” を選択, Save

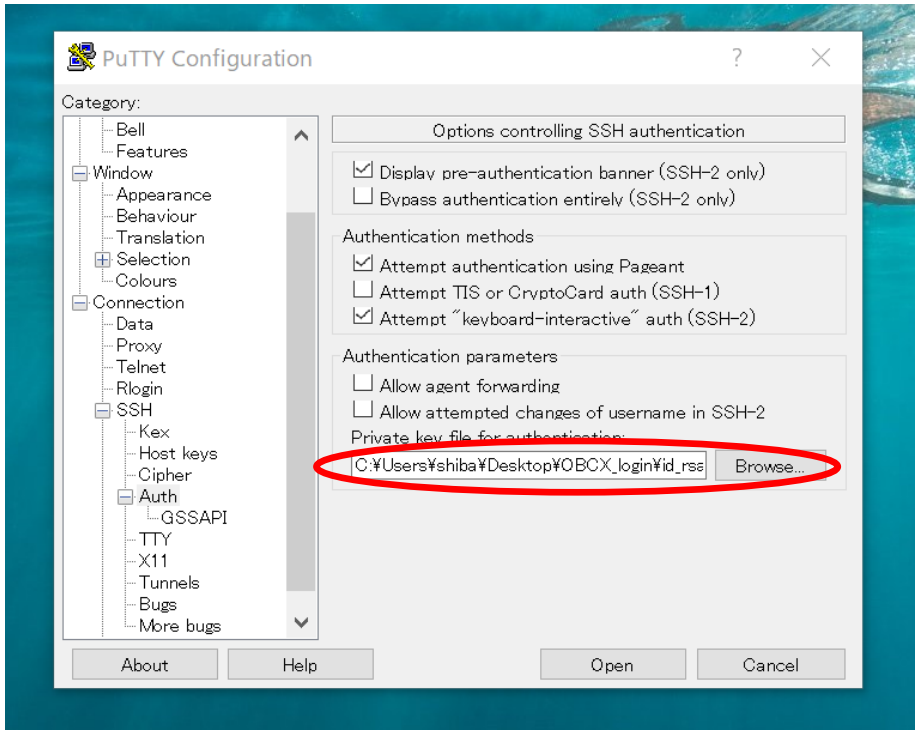


次回以降すぐLoadで開始できます

(5) 実際にログイン

先ほど作成保存した秘密鍵を指定

“id_rsa”

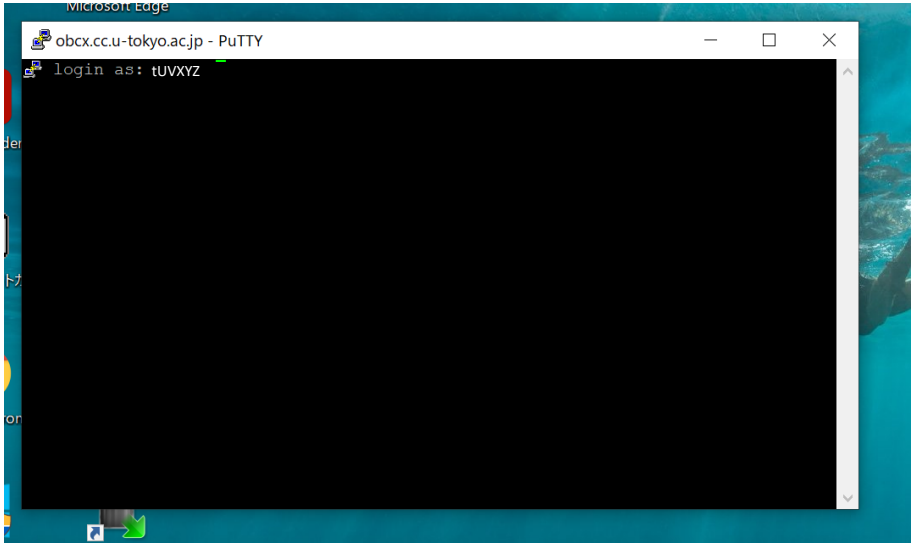


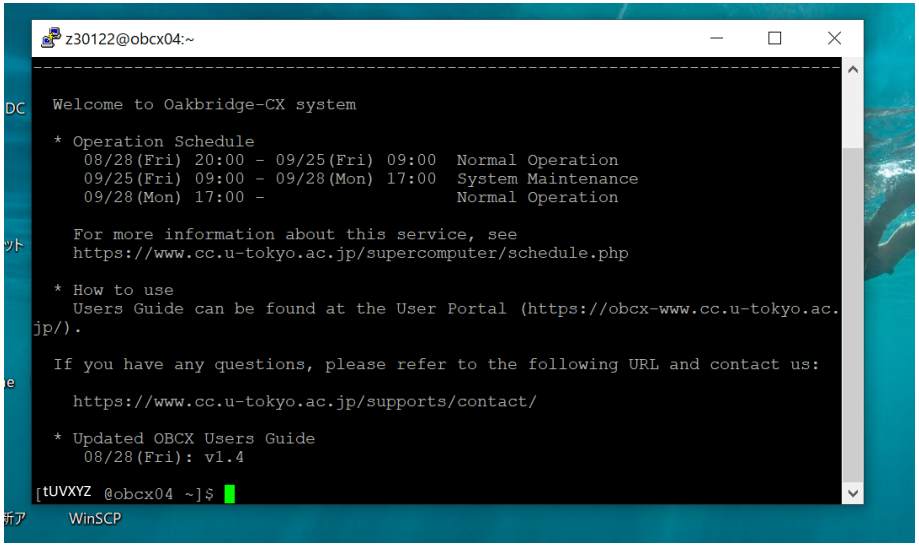
スパコンにログイン

(5) 実際にログイン〔続〕

ユーザー名（tUVXYZ）を入力

スパコンのログインノード上で
操作ができるようになりました。





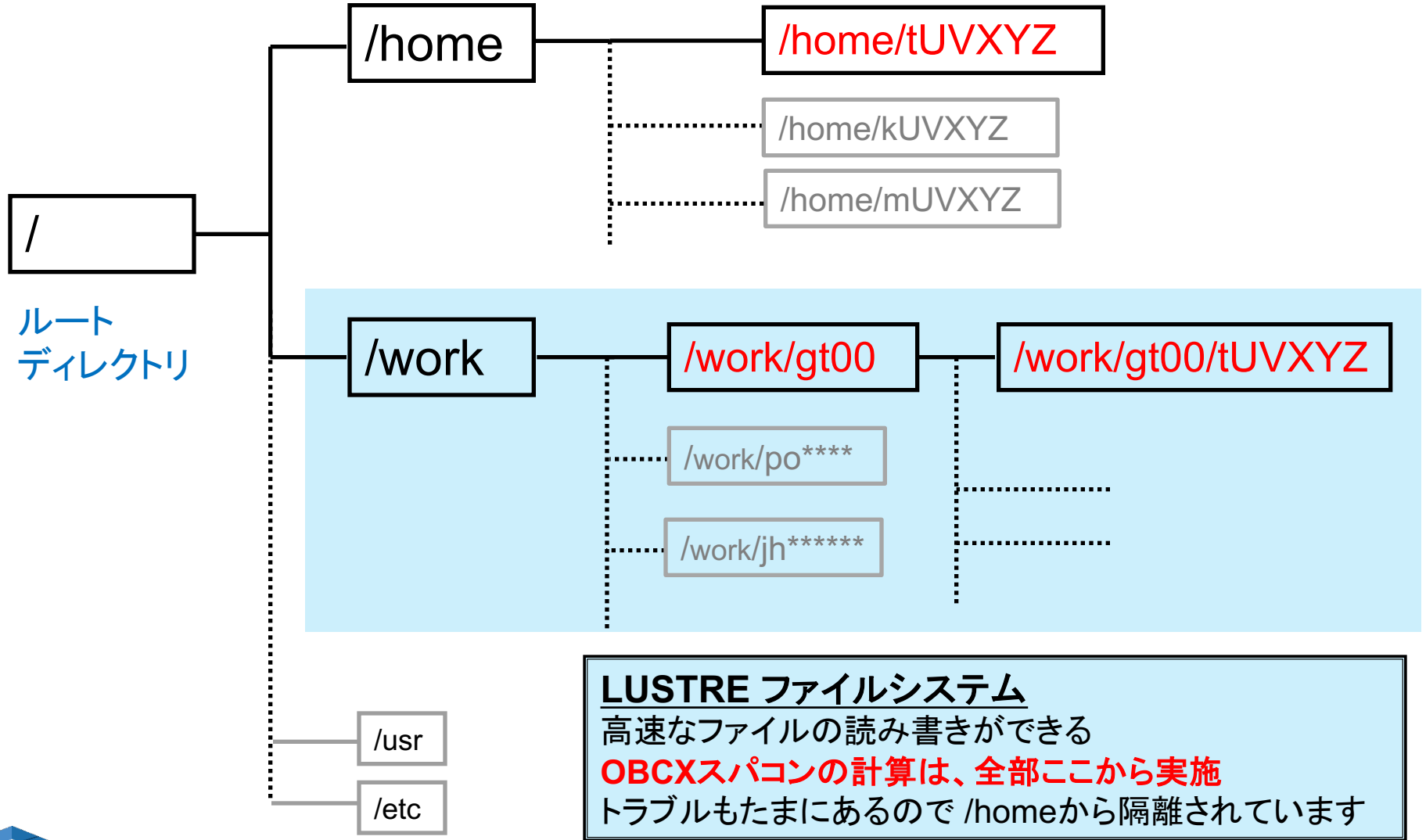
ログインしたら

ログインノード上のLinux の操作画面になります
→ 命令文を打ちます

```
[tUVXYZ@obcx01 ~]$ pwd  Return

/home/tUVXYZ
```

pwd ー 現在のディレクトリ(フォルダ)
を位置を返すコマンド

OBCXのディレクトリ構成



LUSTRE ファイルシステム
高速なファイルの読み書きができる
OBCXスパコンの計算は、全部ここから実施
トラブルもたまにあるので /homeから隔離されています

Linux のコマンド集（簡単なもの）

コマンド	
cd	現在のディレクトリの変更
ls	ファイル・ディレクトリの表示
pwd	現在のディレクトリの表示
mv	ファイル・ディレクトリの移動
cp	ファイル・ディレクトリのコピー
rm	ファイル・ディレクトリの消去（注意して使う）
mkdir	新たな空きディレクトリの作成
cat	ファイル内容の表示（全体）
less	ファイル内容の表示（一画面ごと）
head	ファイル内容の表示（先頭部分）
tail	ファイル内容の表示（末尾部分）
grep	ファイルに含まれる文字列を検索
echo	変数や文字列の値を出力
exit	セッションを終了する

ログインしたら

```
$ pwd    Return

/home/tUVXYZ

$ cd /work/gt00/tUVXYZ  Return
$ pwd    Return

/work/gt00/tUVXYZ

$ cd     Return
$ pwd    Return

/home/t00XYZ
```

1. ログインしたら「/home/tUVXYZ」に入る
2. /homeは容量が少ないので「/work/gt00/tUVXYZ」に移動すること
3. 「cd」でホームに戻れます

プログラム例題：フィボナッチ数列

漸化式で定義

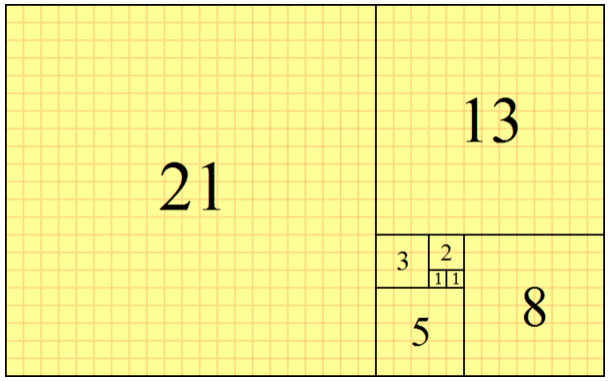
$$F_0 = 1$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

繰り返し計算で数列が得られる
1, 1, 2, 3, 5, 8, 13, 21 ….

繰り返し計算はスーパーコンピューター利用の出発点



From Wikipedia
https://ja.wikipedia.org/wiki/フィボナッチ数

プログラム例題：フィボナッチ数列 92個

1	75025	7778742049	806515533049393
1	121393	12586269025	1304969544928657
2	196418	20365011074	2111485077978050
3	317811	32951280099	3416454622906707
5	514229	53316291173	5527939700884757
8	832040	86267571272	8944394323791464
13	1346269	139583862445	14472334024676221
21	2178309	225851433717	23416728348467685
34	3524578	365435296162	37889062373143906
55	5702887	591286729879	61305790721611591
89	9227465	956722026041	99194853094755497
144	14930352	1548008755920	160500643816367088
233	24157817	2504730781961	259695496911122585
377	39088169	4052739537881	420196140727489673
610	63245986	6557470319842	679891637638612258
987	102334155	10610209857723	1100087778366101931
1597	165580141	17167680177565	1779979416004714189
2584	267914296	27777890035288	2880067194370816120
4181	433494437	44945570212853	4660046610375530309
6765	701408733	72723460248141	7540113804746346429
10946	1134903170	117669030460994	
17711	1836311903	190392490709135	
28657	2971215073	308061521170129	
46368	4807526976	498454011879264	

パソコンでも計算は一瞬

エディタでプログラムを作成する

さて、実際に作業してみましょう。
プログラムを編集するディレクトリを作成します。

```
[tUVXYZ@obcx01 ~]$ cd                  Return
[tUVXYZ@obcx01 ~]$ mkdir fibonacci     Return
[tUVXYZ@obcx01 ~]$ cd fibonacci        Return
```

ログインノード上では、Emacs, vim, または nano のエディタを利用してください。

例1： Emacs を使用，Fortranプログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.f90    Return
```

例2： vim を使用，Python プログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ vim fibonacci.py       Return
```

例3： nano を使用, C プログラムを作成

```
[tUVXYZ@obcx01 fibonacci]$ nano fibonacci.c       Return
```

Emacs

Emacsの基本的な操作方法について

編集画面

コマンド確認画面

$emacs で起動

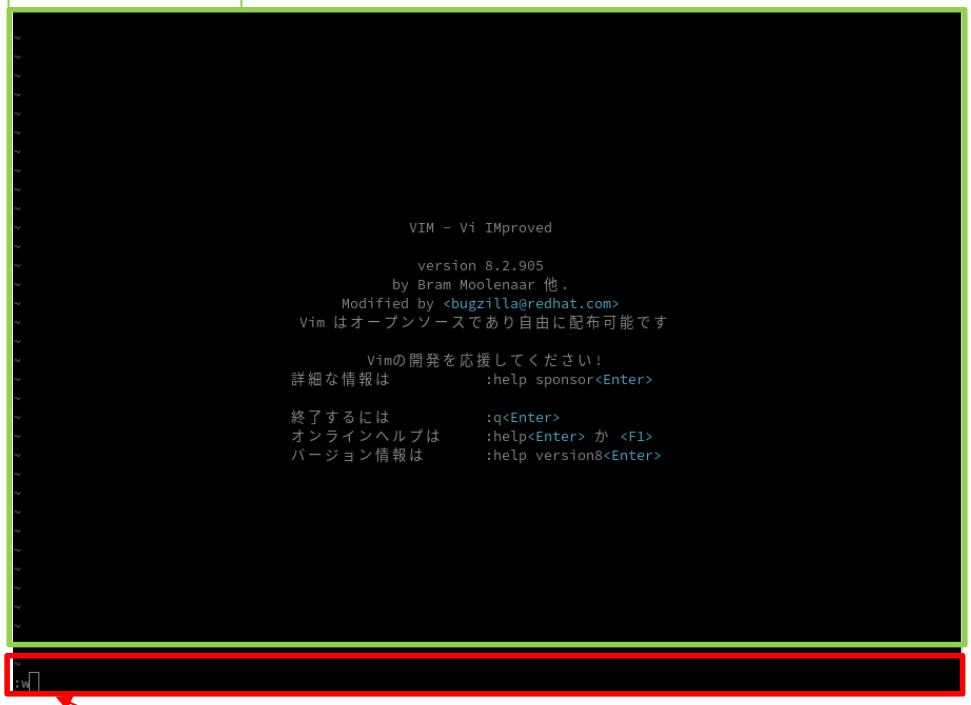
- C→Ctrlキー、M→Metaキー（ESC）、- → 同時押し
- Ctrl-zを押さないように注意

操作	コマンド
ファイルを開く	C-x, C-f ファイル名 Ret
別名保存	C-x, C-w ファイル名 Ret
上書き保存	C-x, C-s
Mark Set	C-space
コピー	Mark Set, 範囲選択, M, w
切り取り	Mark Set, 範囲選択, C-w
貼り付け	C-y
検索	C-s
Undo	C-x, u
キャンセル	C-g
終了	C-x, C-c

Vim

Vimの基本的な操作方法について

編集画面



モードおよびコマンド確認画面
- ブランク → ノーマルモード
- : → コマンドモード
  - :w Ret 上書き保存
- --挿入-- → 入力モード

$vim で起動

Vimではノーマルモード、入力モードとコマンドラインモードを切り替えて使う

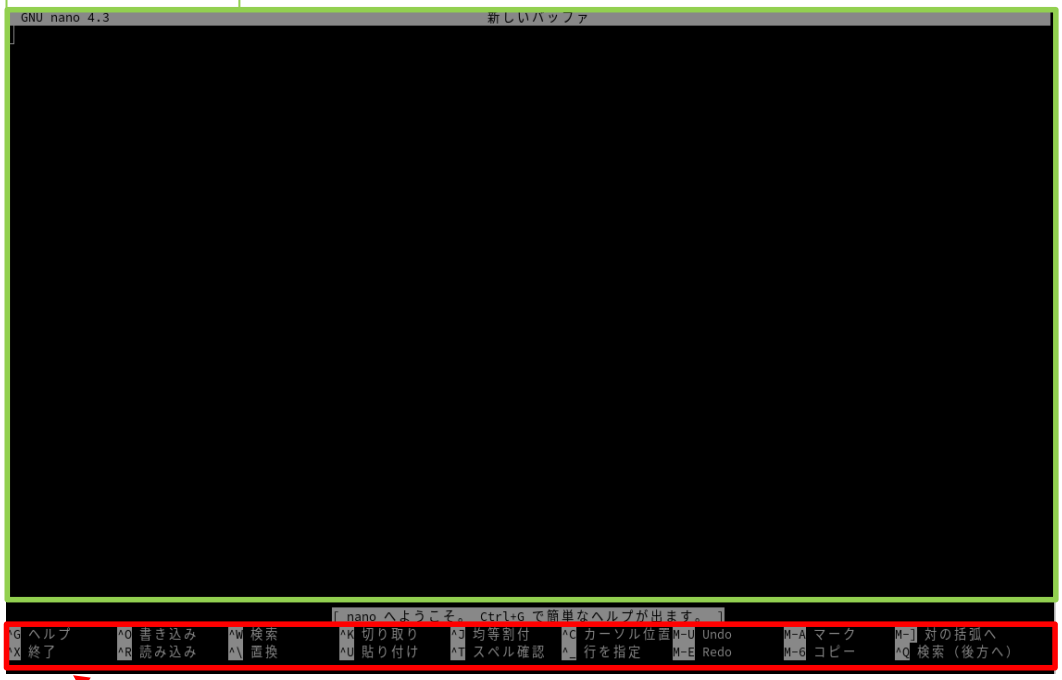
現在のモード	移行先モード	キー
ノーマル	入力	i, a, Ins
ノーマル	コマンドライン	:
入力	ノーマル	esc, ctrl-c

操作	コマンド
ファイルを開く	e Ret
別名保存	w ファイル名 Ret
上書き保存	w Ret
1行コピー	yy
1行切り取り	dd
貼り付け	p
検索	/*検索対象* Ret
終了	q

Nano

Nanoの基本的な操作方法について

編集画面



コマンド確認画面
- ^ → Ctrl
- M → Esc
- - → 順次押す

$nano で起動
- ^G → CtrlとGキーを同時押し
- M-U → Escキー、Uキーを順番に押す

操作	コマンド
ファイルを開く	^R
別名保存	^O ファイル名 Ret
上書き保存	^O
マーク	M-A
マークキャンセル	M-A
コピー	M-6
切り取り	^K
貼り付け	^U
検索	^W
終了	^X

プログラム例題：フィボナッチ数列

C言語

プログラム作成 `[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.c` Return

```
#include <stdio.h>

int main(void) {
    int i;
    long a, b, tmp;

    a=1;
    b=1;
    printf("%ld\n", a);

    for (i=2; i<=92; i++) {
        tmp = b;
        b = a + b;
        a = tmp;
        printf("%ld\n", a);
    }
}
```

注）￥マークはバックスラッシュと同じと思って（今は）結構です。

プログラム例題：フィボナッチ数列

Fortran言語

プログラム作成 `[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.f90` Return

```
program main

    implicit none
    integer(kind=4)  i
    integer(kind=8)  a,b,tmp

    a = 1
    b = 1

    do i=2, 92
       tmp = b
       b= a + b
       a = tmp
       print *, a
    end do

end program main
```

プログラム例題：フィボナッチ数列

Python

プログラム作成 `[tUVXYZ@obcx01 fibonacci]$ emacs fibonacci.py` Return

```
a, b = 1, 1
print(a)

for i in range(1,92):
    a, b = b, a+b
    print(a)
```

注） python では
- 64ビット整数など変数型の選択を自動的にやってくれます
- 多変数を同時に代入計算できます。
- コンパイルはする必要ありません。

モジュールの利用 (1)

コンパイラ等の選択

gnu コンパイラ，インテル製コンパイラ, python インタープリター, …

ライブラリを利用した計算の高速化

離散フーリエ変換，線形代数・連立１次方程式, …

使用中モジュールの表示

```
[tUVXYZ@obcx01 ~]$ module list
Currently Loaded Modulefiles:
  1) impi/2019.5.281    2) intel/2019.5.281
```

Return

Intel MPI ライブラリ　　Intel コンパイラ

コンパイラ, MPI ライブラリを切り替えるために一度 消去します

```
[tUVXYZ@obcx01 ~]$ module purge
[tUVXYZ@obcx01 ~]$ module list
No Modulefiles Currently Loaded.
```

Return　Return

利用可能なモジュールの一覧を表示できます

```
[tUVXYZ@obcx01 ~]$ module avail
```

Return

```
------ /home/opt/local/modulefiles/L/mpi/intel/2019.5.281/impi/2019.5.281 ------
alps/2.3.0(default)                       phdf5/1.10.5(default)
feram/0.26.04(default)                    pnetcdf/1.11.2(default)
frontflow_blue/8.1(default)               ppohBEM/0.5.0(default)
frontistr/4.5(default)                    ppohDEM_util/1.0.0(default)
modylas/1.0.4(default)                    ppohFDM/0.3.1(default)
mpi-fftw/3.3.8(default)                   ppohFEM/1.0.1(default)
netcdf-fortran-parallel/4.4.5(default)    ppohFVM/0.3.0(default)
netcdf-parallel/4.7.0(default)            pt-scotch/6.0.6(default)
openmx/3.8(default)                       revocap_coupler/2.1(default)
parmetis/4.0.3(default)                   superlu_dist/6.1.1(default)
petsc/3.11.2(default)                     xtapp/rc-150401(default)
phase/2019.01(default)

---------- /home/opt/local/modulefiles/L/compiler/intel/2019.5.281 ----------
R/3.6.0(default)                  metis/5.1.0(default)
akaikkr/cpa2002v010(default)      mt-metis/0.6.0(default)
bioconductor/3.10(default)        netcdf/4.7.0(default)
blast/2.9.0(default)              netcdf-cxx/4.3.0(default)
bwa/0.7.17(default)               netcdf-fortran/4.4.5(default)
fftw/3.3.8(default)               paraview/5.6.1(default)
gsl/2.5(default)                  povray/3.7.0.8(default)
hdf5/1.10.5(default)              ppohAT/1.0.0(default)
hdf5/1.8.21                       revocap_refiner/1.1.04(default)
impi/2019.5.281(default)          samtools/1.9(default)
intelpython/2.7                   scotch/6.0.7(default)
intelpython/3.6(default)          superlu/5.2.1(default)
mesa/19.0.6(default)              superlu_mt/3.1(default)
metis/4.0.3                       xabclib/1.03(default)

--------------------- /home/opt/local/modulefiles/L/core ---------------------
acusolve/2019.1.0(default)          intel/2018.3.222
advisor/2019.3.0.591490             intel/2019.3.199
advisor/2019.4.0.597843             intel/2019.4.243
advisor/2019.5.0.602216(default)    intel/2019.5.281(default)
anaconda/2-2019.03                  intel/2020.1.217
anaconda/3-2019.03(default)         itac/2019.4.036
bioperl/1.007002(default)           itac/2019.5.041(default)
bioruby/1.5.2(default)              julia/1.4.0(default)
cmake/3.0.2                         llvm/7.1.0(default)
cmake/3.14.5(default)               massivethreads/0.97
```

モジュールの利用 (2)

Intel コンパイラの 最新のものを load してみます

```
[tUVXYZ@obcx01 ~]$ module load  intel/2020.1.217 Return
[tUVXYZ@obcx04 ~]$ module list  Return
Currently Loaded Modulefiles:
  1) impi/2019.7.217    2) intel/2020.1.217
```

Intel MPI ライブラリも、一緒に自動でload されました

Python も比較的最近の バージョン3.7.3 を使えるようにしましょう。

```
[tUVXYZ@obcx01 ~]$ module load python/3.7.3 Return
[tUVXYZ@obcx01 ~]$ module list   Return
Currently Loaded Modulefiles:
  1) impi/2019.7.217    2) intel/2020.1.217    3) python/3.7.3
```

スーパーコンピューター (スパコン) 利用のイメージ

通常、スパコンではログインノードと呼ばれる玄関口から実行命令を出します

端末

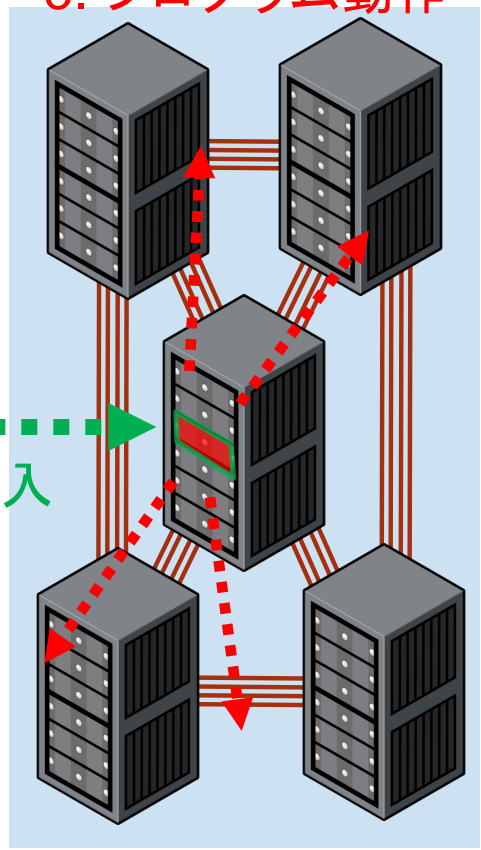
1. ログイン
SSH

ログインノード

```
[tut138@obcx02~]$ ls -l
drwxr-x--- 2 shiba group 10 1
Apr 13:00 test.out
[tut138@obcx02~]$ ./test.out
Hello world
[tut138@obcx02~]$ qsub a.sh
```

2. ジョブ投入

3. プログラム動作



計算ノード
= 本体

SSH で接続するには
鍵の準備が必要！

ジョブスクリプト = スパコンへの指示書

プログラムをコンパイルしたあとにログインノード上で
直接 プログラム実行することはしないでください！
— 負荷がかかると、他のユーザーの同時作業の妨げとなります

かわりに計算ノードに計算をしてもらうための「指示書」が必要となります。

C, Fortran 向け (fibonacci.sh)

```
#!/bin/bash
#PJM  -L  rscgrp=tutorial    [リソースグループ]
#PJM  -L  node=1             [使用するノードの数]
#PJM  -L elapse=0:01:00      [実行時間上限（1分）]
#PJM  -g  gt00               [バジェットグループ名]
#PJM  -N  fibonacci          [今回のジョブの名前]
#PJM  -o   stdout.txt        [標準出力先ファイル名]
#PJM  -j                     [エラー出力を標準出力にマージ]

module purge
module load intel/2020.1.217
./fibonacci.out
```

Python 向け (fibonacci.sh)

```
#!/bin/bash
#PJM  -L  rscgrp=tutorial
#PJM  -L  node=1
#PJM  -L  elapse=0:01:00
#PJM  -g  gt00
#PJM  -N  fibonacci
#PJM  -o  stdout.txt
#PJM  -j

module load python/3.7.3
python ./fibonacci.py
```

ジョブスクリプトを用意しました。

ご自身で入力いただくと時間を要しますので、
用意されたものをコピーしてください。

```
[tUVXYZ@obcx01 ~]$ cd                Return
[tUVXYZ@obcx01 ~]$ cd fibonacci      Return
```

C, Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_c/fibonacci.sh .
```

Return

Python を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_python/ fibonacci.sh .
```

Return

エディターでの編集が間に合わなかった方は

サンプルとなるプログラムも用意してありますので、
ご自身でコピーしてください。

C を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_c/fibonacci.c .
```

Return

Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_fortran/fibonacci.f90 .
```

Return

Python を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ cp /work/gt00/share/z30122/fibo_python/fibonacci.py .
```

Return

プログラムのコンパイル

C または Fortran を使用される方

→ プログラムをコンパイルして、実行ファイル（バイナリ）を生成します。

C を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ icc fibonacci.c -o fibonacci.out
```

Return

プログラム名　　実行ファイル名

Fortran を使われる方

```
[tUVXYZ@obcx01 fibonacci]$ ifort fibonacci.f90 -o fibonacci.out
```

Return

スパコンでのジョブの実行

OBCX スパコンでは、/home からプログラム実行できません
→ /work ディレクトリ領域内にプログラム等一式をコピー

```
[tUVXYZ@obcx01 ~]$ cd  Return
[tUVXYZ@obcx01 ~]$ ls  Return
fibonacci
[tUVXYZ@obcx01 ~]$ cp -r fibonacci  /work/gt00/tUVXYZ/ Return

[tUVXYZ@obcx01 ~]$ cd  /work/gt00/tUVXYZ/fibonacci Return
```

作成したジョブスクリプトを投入、
計算ノードで実行してもらいます。

```
[tUVXYZ@obcx01 fibonacci]$ pjsub fibonacci.sh  Return
```

ジョブの確認、結果の表示

実行されているジョブを確認します。
（一瞬で計算が終わるので、ジョブ一覧に出ないかも）

```
[tUVXYZ@obcx01 fibonacci]$ pjstat
```

得られた結果を表示します。

```
[tUVXYZ@obcx01 fibonacci]$ more result_fibo.txt
```

[Return で下の方に進めていくことができます]

結果から 747031 という文字列を探します

```
[tUVXYZ@obcx01 fibonacci]$ grep 747031 result_fibo.txt
```

お試しアカウントについての注意

今回発行されたアカウント
→ 講習会終了後も約1ヶ月間、OBCXスパコンを使用可能

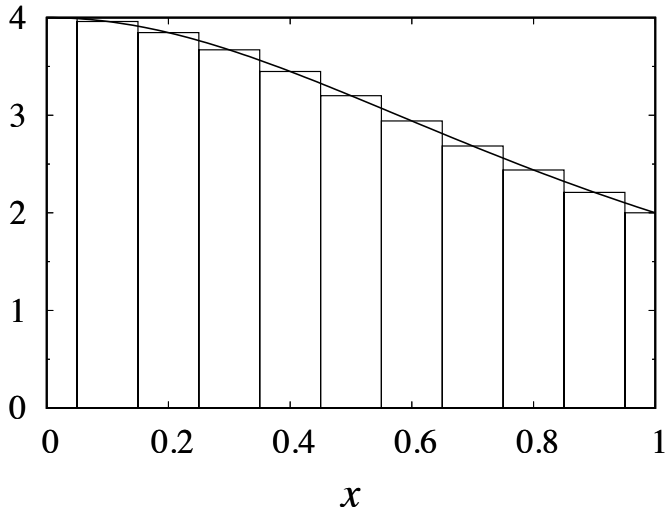
講習会終了後は リソースグループ lecture を使用してください。
リソースグループ tutorial は本日 13:00-17:00 のみです。

ジョブスクリプト
fibonacci.sh

```
#!/bin/bash
#PJM  -L  rscgrp=lecture        [リソースグループ]
#PJM  -L  node=1
#PJM  -L elapse=0:01:00
#PJM  -g  gt00
#PJM  -N  fibonacci
#PJM  -o   stdout.txt
#PJM  -j

module purge
module load intel/2020.1.217
./fibonacci.out
```

並列計算の例題：区分求積で円周率計算

$$I = \int_0^1 \frac{4}{1+x^2} dx = \pi$$



和の短冊
↓
1億倍以上
細かくする

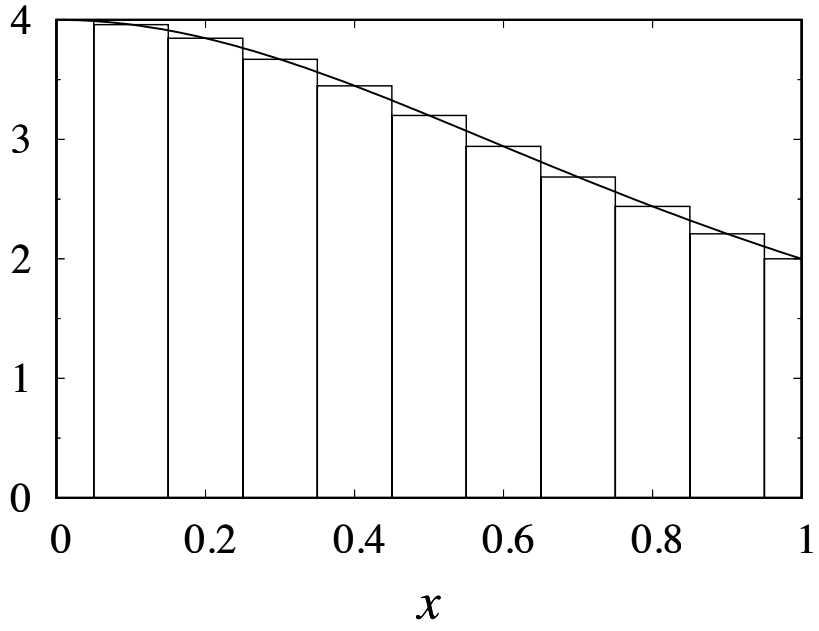
数値積分を並列化 → 性能が向上する。

C, Fortran, Python, 3通りのサンプルプログラムを用意しました。

実習： プログラムの並列度を変えながら実行
実行時間から高速化の度合いを見る

並列計算の例題：区分求積で円周率計算

$$I = \int_0^1 \frac{4}{1+x^2} dx$$



変数変換 $x = \tan\theta$

$$dx = \frac{d\theta}{\cos^2\theta}$$

$$\Rightarrow I = \int_0^{\pi/4} \frac{4}{1+\tan^2\theta} \frac{d\theta}{\cos^2\theta} = 4\int_0^{\pi/4} d\theta = \pi$$

コンピュータにおける演算の仕組み

(ノイマン型) コンピューター上では、全ての数がビット [0,1] の集まりで表現される

単精度浮動小数点 = 32ビットで1つの実数を表現

$\log_{10}(2^{23}) \sim 7$ 桁程度 (10進) が表現可

$$(-1)^{\mathrm{sign}}(1.b_1b_2\ldots b_{23})_2 \times 2^{e-127}$$

変数型	FORTRAN: real(kind=4) C, C++: float Python: 基本はなし

倍精度浮動小数点 = 64ビットで1つの実数を表現

$\log_{10}(2^{52}) \sim 15$ 桁程度 (10進) が表現可

$$(-1)^{\mathrm{sign}}(1.b_1b_2\ldots b_{52})_2 \times 2^{e-1023}$$

変数型	FORTRAN: real(kind=8) C, C++: double Python: 無指定で倍精度

プログラムの解説 C

pi.c [非並列版]

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

int main(void) {
  int i;
  int ndiv = 10000000;


  double width = 1.0 / (double)ndiv;
  printf("width = %.15f\n", width);
  double sum = 0.0;
  double x = 0.0;


  for (i=0; i<ndiv; i++) {
    x = (i+0.5)*width;
    sum += width * 4.0 / (1.0 + x*x);
  }


  printf("PI = %.15f\n", sum);
  return 0;
}
```

pi_mpi.c [並列版]

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

int main(int argc, char* argv[]) {
   int ndiv = 5600000000;
   int ierr,  myrank, nprocs;
   int ndiv_local, i;
   double x, width, sum, total_sum;
   double t1, t2;

   width = 1.0 / (double)ndiv;

   ierr = MPI_Init(&argc, &argv);
   ierr = MPI_Comm_rank(MPI_COMM_WORLD,
                        &myrank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD,
                        &nprocs);

  t1 = MPI_Wtime();

  sum = 0.0;
  ndiv_local = ndiv / nprocs;

  for (i = myrank*ndiv_local;
        i < (myrank+1)*ndiv_local; i++) {
        x = (i + 0.5) * width;
     sum = sum + width * 4.0 / (1.0 + x*x);
  }

  MPI_Reduce(&sum, &total_sum, 1, MPI_DOUBLE,
                  MPI_SUM, 0, MPI_COMM_WORLD);

  t2 = MPI_Wtime();

  if (myrank == 0) {
    printf("PI(MPI) = %.18f\n", total_sum);
    printf("Number of cores utilized = %d\n", nprocs);
    printf("Execution time = %.8f (sec.)\n", t2 - t1);
  }

  ierr = MPI_Finalize();

  return 0;
}
```

コンパイル

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiicc pi_mpi.c -o pi_mpi.out
```

プログラムの解説 Fortran

pi.f90 [非並列版]

```
program main

  implicit none

  integer i, ndiv
  real(kind=8) unit, width, sum, x

  ndiv = 560000000
  width = 1.0 / dble(ndiv)

  print *, "width"
  print '(F18.14)', width

  sum = 0.0d0
  x = 0.0d0

  do i = 1, ndiv
    x = ( dble(i-1) + 0.5) * width
    sum += width * 4.0 / (1.0 + x*x)
  end do

  print *, "sum"
  print '(F18.14)', sum

end program main
```

pi_mpi.f90 [並列版]

```
program main

  Use mpi
  implicit none

  integer ndiv, ierr, myrank, nprocs
  integer ndiv_local, i;
  real(kind=8) unit, width, sum, x, total_sum
  real(kind=8) t1, t2

  ndiv = 5600000000
  width = 1.0 / dble(ndiv)

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD,
                     myrank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD,
                     nprocs, ierr)

  sum = 0.0d0
  ndiv_local = ndiv / nprocs

  t1 = MPI_Wtime()

  do i=myrank*ndiv_local+1, (myrank+1)*ndiv_local
    x = ( dble(i-1) + 0.5)*width
    sum = sum + width * 4.0 / (1.0 + x*x)
  end do

  call MPI_REDUCE(sum, total_sum, 1, MPI_REAL8,
MPI_SUM, 0, MPI_COMM_WORLD, ierr)

  t2 = MPI_Wtime()

  if (myrank .eq. 0) then
    print "('PI(MPI) = ', F18.16)", total_sum
    print "('Number of cores utilized = ', i0)", nprocs
    print "('Execution time = ', F12.8)", t2 - t1
  endif

  call MPI_FINALIZE(ierr)

end program main
```

コンパイル

```
[tUVXYZ@obcx01 calc_pi_fortran]$ mpiifort pi_mpi.f90 -o pi_mpi.out
```

プログラムの解説 Python

pi.py [非並列版]

```
program main
  implicit none

  integer i, ndiv
  real(kind=8) unit, width, sum, x

  ndiv = 560000000
  width = 1.0 / dble(ndiv)

  print *, "width"
  print '(F18.14)', width

  sum = 0.0
  x = 0.0

  do i = 1, ndiv
     x = ( dble(i-1) + 0.5) * width
     sum = sum + width * 4.0 / (1.0 + x*x)
  end do

  print *, "sum"
  print '(F18.14)', sum

end program main
```

pi_mpi.py [並列版]

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
myrank = comm.Get_rank();
nprocs = comm.Get_size();

ndiv = 560000000
width = 1.0/ndiv

t1 = MPI.Wtime()

sum = numpy.zeros(1)
total_sum = numpy.zeros(1)
ndiv_local = ndiv // nprocs

for i in range (myrank*ndiv_local, (myrank+1)*ndiv_local):
    x = width * (i+0.5)
    sum[0] = sum[0] + width * 4.0 / (1.0 + x*x)

comm.Reduce([sum, MPI.DOUBLE], total_sum, op=MPI.SUM, root=0)

t2 = MPI.Wtime()

if comm.rank == 0:
    print("PI(MPI) = ", total_sum[0])
    print("Number of cores utilized = ", nprocs)
    print("Exectution time = ", t2 - t1, " (sec.)"
```

プログラムなど一式を自分の /work へコピー

```
[tUVXYZ@obcx01 ~]$ cd /work/gt00/tUVXYZ
[tUVXYZ@obcx01 tUVXYZ]$ mkdir calc_pi_mpi
[tUVXYZ@obcx01 tUVXYZ]$ cd calc_pi_mpi
[tUVXYZ@obcx01 calc_pi_mpi]$ pwd
/work/gt00/tUVXYZ/calc_pi
```

Fortran の方

```
$ cp /work/gt00/share/z30122/pi_fortran_mpi/* .
```

アスタリスク = ワイルドカード
「ここはなんでも良い」

C の方

```
$ cp /work/gt00/share/z30122/pi_c_mpi/* .
```

Python の方

```
$ cp /work/gt00/share/z30122/pi_python_mpi/* .
```

並列プログラムのコンパイル

C および Fortran では プログラムをコンパイルします。

C の方

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiicc pi_mpi.c -o pi_mpi.out
```

Fortran の方

```
[tUVXYZ@obcx01 calc_pi_mpi]$ mpiifort pi_mpi.f90 -o pi_mpi.out
```

Python の方

事前に次の作業が必要です。

```
[tUVXYZ@obcx01 ~]$ cd /work/gt00/tUVXYZ/calc_pi_mpi
[tUVXYZ@obcx01 calc_pi_mpi]$ emacs setenv.sh   (or vim setenv.sh)
```

次のところを変更してください

```
export PYTHONUSERBASE=/work/gt00/tUVXYZ/.local
```

⇩

自分のユーザー名に

次にsetup.sh スクリプトを実行 （numpy, mpi4py をインストールします）

```
[tUVXYZ@obcx01 ~]$ ./setup.sh
```

並列プログラムのジョブスクリプト

性能が向上することを見ます。
様々な並列度のジョブスクリプトを用意しました。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0001.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0004.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0028.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n1c0056.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n2c0112.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n4c0224.sh
[tUVXYZ@obcx01 calc_pi_mpi]$ pjsub run_n8c0448.sh
```

- 1ノード, 1コア
- 1ノード, 4コア
- 1ノード, 28コア
- 1ノード, 56コア
- 2ノード, 112コア
- 4ノード, 224コア
- 8ノード, 448コア

ジョブの実行状況の確認

自分の全てのジョブが終わるまで、待機します。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ pjstat
```

ジョブが終了すると、出力ファイルが出ているはずです。

```
[tUVXYZ@obcx01 calc_pi_mpi]$ ls
```

pi_mpi.c	result_n2c0112.txt	run_n1c0056.sh
pi_mpi.out	result_n4c0224.txt	run_n2c0112.sh
result_n1c0001.txt	result_n4c0448.txt	run_n4c0224.sh
result_n1c0004.txt	run_n1c0001.sh	run_n8c0448.sh
result_n1c0028.txt	run_n1c0004.sh	
result_n1c0056.txt	run_n1c0028.sh	

“result_n*c****.txt” が、結果を記載した出力ファイルです。

結果を確認、実行時間を比較してみましょう

出力ファイルには実行時間の記載があります。
出力ファイルの冒頭部分を一斉に見てみましょう。

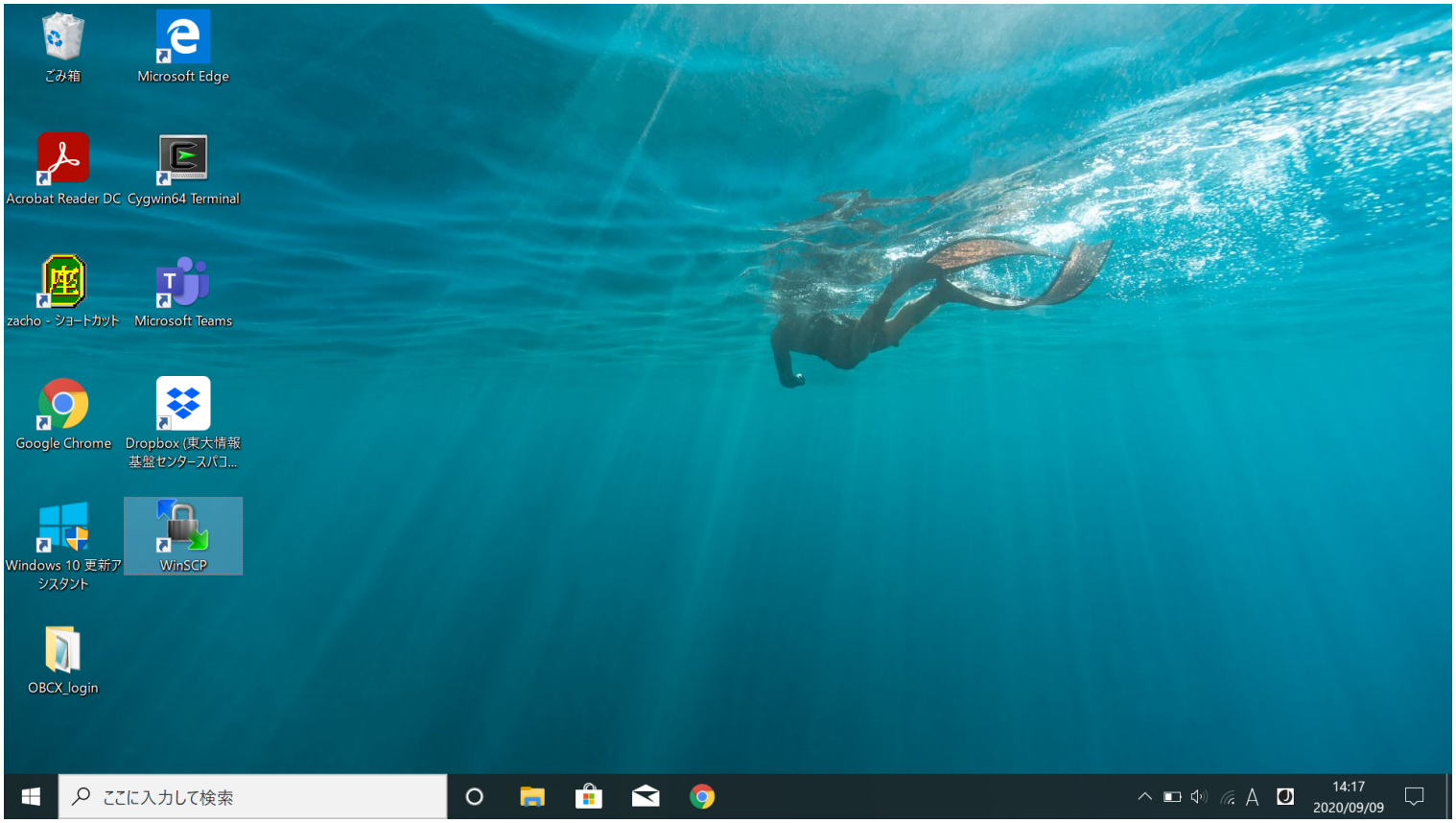
```
[tUVXYZ@obcx01 calc_pi_mpi]$ head result* txt
```

並列化で実行時間が短縮されています。

```
=> result_n1c0001.txt <==
PI(MPI) = 3.141592653590673301
Number of cores utilized = 1
Execution time = **.******** (sec.)

==> result_n1c0004.txt <==
PI(MPI) = 3.141592653589913464
Number of cores utilized = 4
Execution time = **.******** (sec.)

==> result_n1c0028.txt <==
PI(MPI) = 3.141592653589770912
Number of cores utilized = 28
Execution time = **.******** (sec.)

==> result_n1c0056.txt <==
PI(MPI) = 3.141592653589800221
Number of cores utilized = 56
Execution time = **.******** (sec.)

==> result_n2c0112.txt <==
PI(MPI) = 3.141592653589794892
Number of cores utilized = 112
Execution time = **.******** (sec.)

==> result_n4c0224.txt <==
PI(MPI) = 3.141592653589791340
Number of cores utilized = 224
Execution time = **.******** (sec.)
```

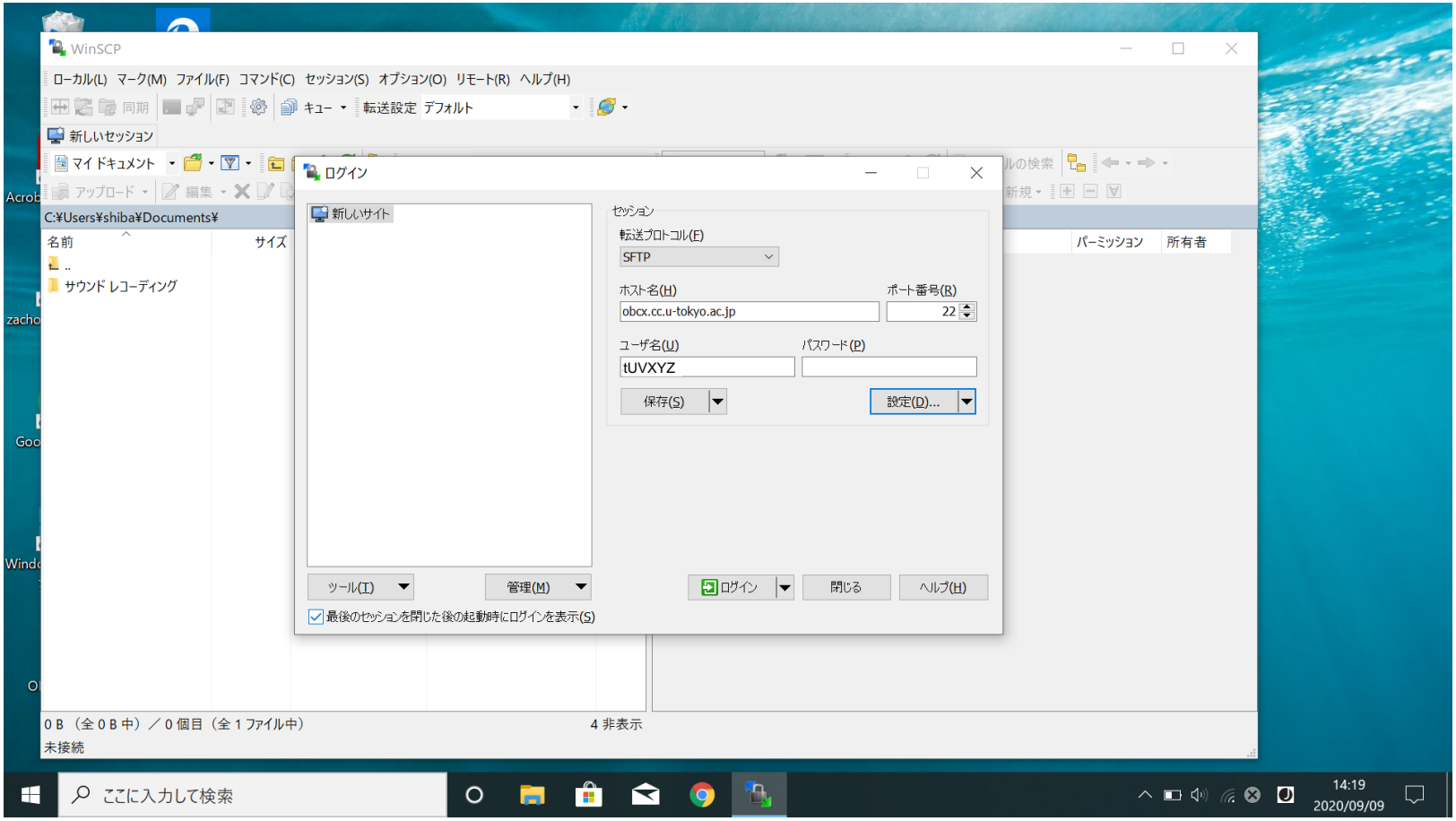
最後に：計算結果のファイルを手元に転送する

WinSCP を起動してください



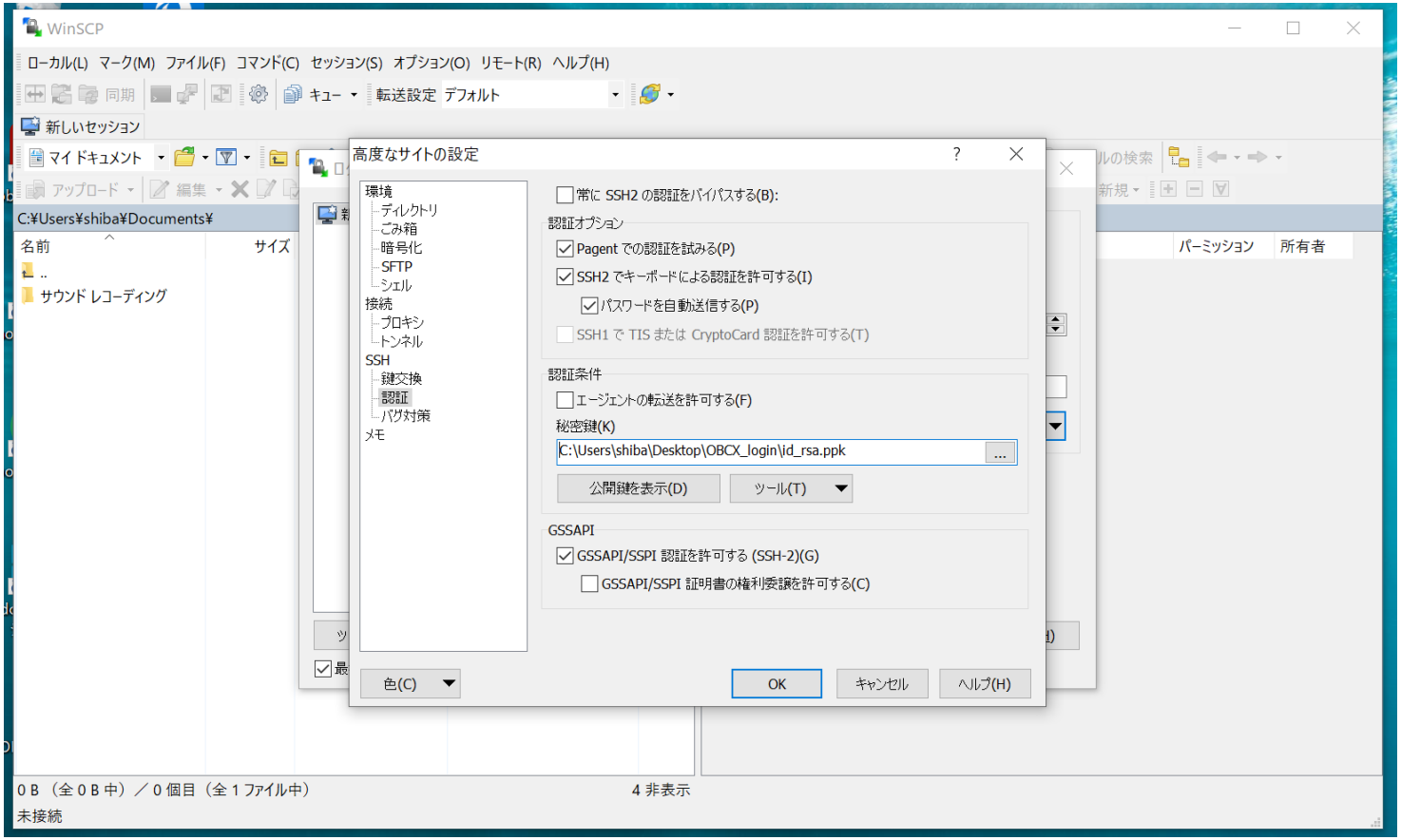
最後に：計算結果のファイルを手元に転送する

接続サーバー情報を、ユーザー名 (tUVXYZ) とともに入力します
OBCXでは鍵認証ですので、パスワードは空欄のまま



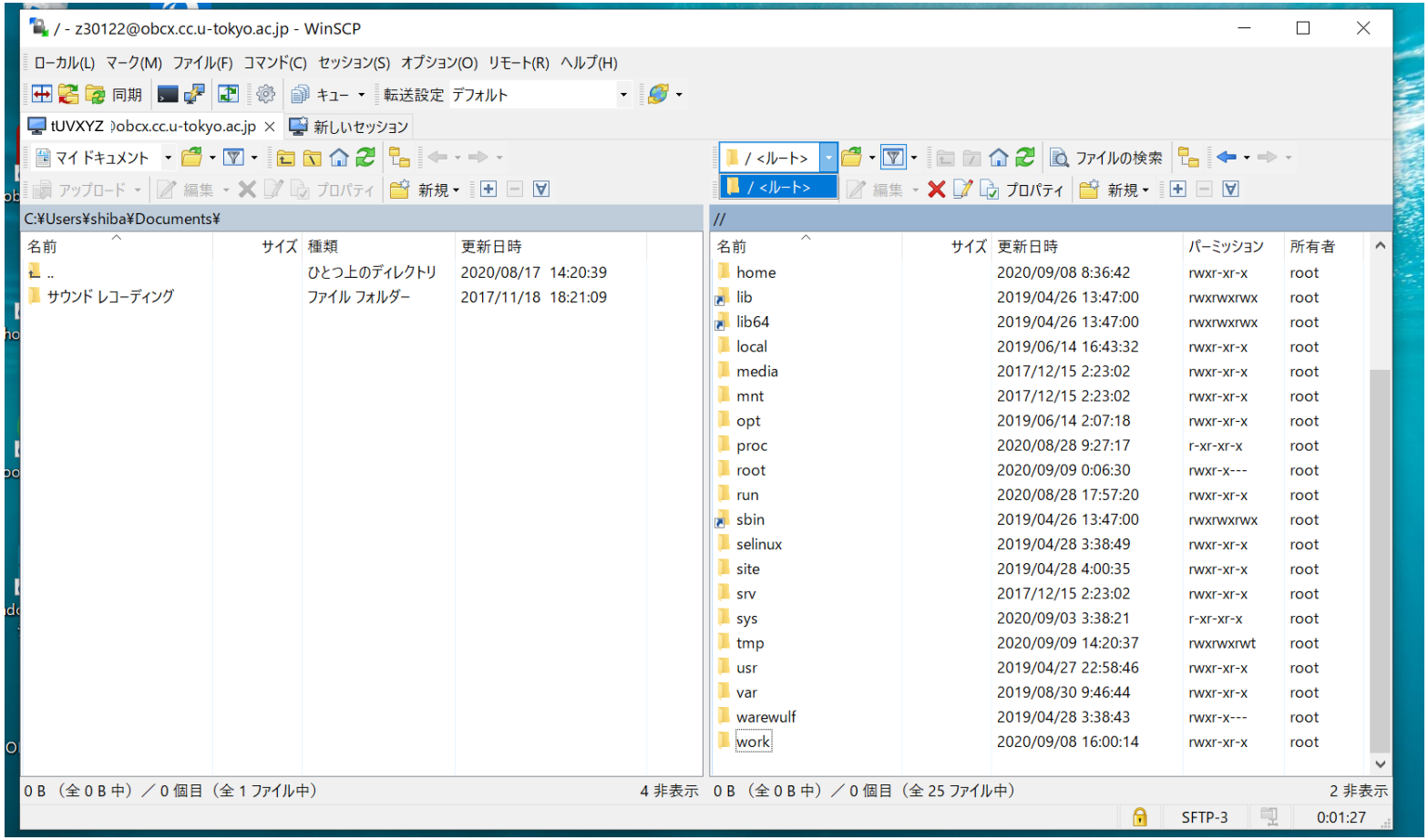
最後に：計算結果のファイルを手元に転送する

SSH > 認証　秘密鍵を指定し、チェックボックスを確認



最後に：計算結果のファイルを手元に転送する

目的のディレクトリに行くために、まず「 / <ルート>」から出発



最後に：計算結果のファイルを手元に転送する

こんな画面になります