並列処理（並列コンピューティング）

- 同じ仕事量を複数人でこなせば、時間短縮
  - 封筒の袋詰め、仕分け、など
- 複数のCPUを用いて同じような並列処理が出来ないか？

CPU1台で処理

CPU4台で処理

効率的な処理が可能？

並列コンピュータ

- 複数プロセッサが主記憶を共有する、共有メモリ型
- 各プロセッサが独自のメモリを持つ、分散メモリ型

本実験では、分散メモリ型の並列処理を実現する MPI を学ぶ

MPI = Message Passing Interface

プロセッサ間のデータ通信に関する標準規格

並列処理の実装はプログラマの責任

MPI

- 分散メモリ型の並列処理では、プロセッサ間の通信が欠かせない
- プロセッサ間のデータ通信をライブラリとして提供
- 様々な実装があるが本実験では OpenMPI を用いる
- 各プロセッサはそれぞれ独自のメモリを持つので、同じ名前の変数でも異なる値を持ちうる（MPIプログラミングでの注意点）

ウェブ上には様々なMPIライブラリの解説がある

様々な並列処理

具体例：数列の和の計算 $$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

プロセッサ2台を用いて数列和を並列実行する場合

CPU 0

奇数の和

```
sum = 0;
for(i=1; i<=N; i+=2)
  sum += i;
```

CPU 1

偶数の和

```
sum = 0;
for(i=2; i<=N; i+=2)
  sum += i;
```

CPU0 と CPU1 の計算結果 sum の和が求める解
（プロセッサ間の通信が必要）

MPIの基礎

- MPIでは、各プロセッサは固有ID（rank）を持つ
- 各 rank の動作をプログラマが記述する
- 必要であれば、各プロセッサ間の通信をプログラマが記述

実行に参加するプロセッサの数 size = 4

MPIひな型

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

int main( int argc, char *argv[] )
{
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);

    printf( "Hello world from process %d of %d on %s\n", rank, size,  processor_name );

    MPI_Finalize();

    return 0;
}
```


OpenMPI実行のための環境

- OpenMPIコマンド群へのパスの設定
- 他のノードにパスワードの入力なしにログインできること（他のCPUにジョブを展開するために必要）

上記の設定方法については別途説明する

MPI プログラムの実行

MPIプログラムのコンパイルには、 mpicc を使う

% mpicc mpi-1.c

MPIプログラムの実行には、 mpirun を使う

% mpirun -np 2 ./a.out

ログインしているCPUで2つのノードで並列実行
-np でノード数を設定する。MPIひな形をノード数を増やして実行してみる

% mpirun -np 2 --host gpx10, gpx11 ./a.out

--host でプログラムを実行するPC端末を指定する

MPIひな型の実行

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

int main( int argc, char *argv[] )
{
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);

    printf( "Hello world from process %d of %d on %s\n", rank, size,  processor_name );

    MPI_Finalize();

    return 0;
}
```

各ノードが持つ変数 rank, size などは、それぞれ異なる値を持つことに注意！

例1

```
int main( int argc, char *argv[] )
{
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);

    printf( "Hello world from process %d of %d on %s\n", rank, size, processor_name );

    if( rank == 0 )
      printf("I am rank %d. Do you see me?\n", rank);
    else
      printf("I am rank %d. Hello!\n", rank);

    MPI_Finalize();

    return 0;
}
```

このプログラムを様々なノード数で実行してみよ

プロセッサ間通信

プロセッサ間の1対1通信を行う関数 MPI_Send と MPI_Recv

データ送信

```
MPI_Send(
  void *buf,                 送信するデータのアドレス
  int count,                 送信するデータの数
  MPI_Datatype datatype,     送信するデータの型 MPI_INT, MPI_DOUBLE
  int dest,                  送信先プロセッサ rank
  int tag,                   タグ
  MPI_Comm comm              送受信グループ MPI_COMM_WORLD
)
```


プロセッサ間通信

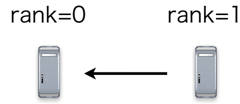
プロセッサ間の1対1通信を行う関数 MPI_Send と MPI_Recv

データ受信

```
MPI_Recv(
  void *buf,                 受信データが格納されるアドレス
  int count,                 受信するデータの数
  MPI_Datatype datatype,     受信するデータの型 MPI_INT, MPI_DOUBLE
  int source,                送信先元プロセッサ rank
  int tag,                   タグ
  MPI_Comm comm,             送受信グループ MPI_COMM_WORLD
  MPI_Status *status         受信結果の情報を格納するアドレス
)
```

プロセッサ間通信

rank1 から rank0 へ
実数値 double x を送る



rank1 での処理

```
dest = 0; //送信先 destination を指定
tag = 777; // 合い言葉の tag を指定
MPI_Send(&x, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

rank0 での処理

```
source = 1; //送信元 source を指定
tag = 777; // 合い言葉を指定
MPI_Recv(&x, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD,
&mpi_stat);
```

送信元 source、送信先 dest、タグ tag の3つを正しく指定することはプログラマの責任

1対1通信の例

```
MPI_Status mpi_stat;
int source, dest, tag;
double x;

.....

if( rank == 0 ){
  printf("Rank %d is going to receive a data\n", rank);
  source = 1;
  tag = 777;
  MPI_Recv( &x, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &mpi_stat);
  printf("Rank %d has received a data %f\n", rank, x);
}

if( rank == 1 ) {
  x = 123.0; // rank 1 で変数 x に値を代入. rank 0 では変数 x は未定であることに注意
  printf("Rank %d is going to send a data %f\n", rank, x);
  dest = 0;
  tag = 777;
  MPI_Send( &x, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
  printf("Rank %d has sent a data\n", rank);
}
```


例2

```
int main( int argc, char *argv[] )
{
    int rank, size, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status mpi_stat;
    int source, dest, tag;
    double x;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(processor_name, &namelen);

    printf( "Hello world from process %d of %d on %s\n", rank, size, processor_name );

    if( rank == 0 )
     ......
    if( rank == 1 )
     ......

    MPI_Finalize();

    return 0;
}
```

例1を拡張して、rank1 から rank0 へ適当な値を
送受信するプログラムを作成せよ。

例3：数列和の並列計算

2つのノードを用いて数列和を並列実行する $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

rank0 の処理：奇数の和

```
int sum;

sum = 0;
for(i=1; i<=N; i+=2)
  sum += i;
```

```
rank1のsumを受信**
```

rank1 の処理：偶数の和

```
int sum;

sum = 0;
for(i=2; i<=N; i+=2)
  sum += i;
```

```
rank0へsumを送信
```

```
受信したrank1のsumと自分のsumの和を表示
```

** rank0 で奇数の和を格納する変数 sum と、rank1の sum を受信する変数は
別物でなければならない！

処理時間の測定

- 並列処理をする場合、ノード間の通信コスト（ネットワークを介したデータのやり取りにかかる時間）が無視できない
- 処理時間を測定して並列処理の効果を見積もる必要がある。MPI_Wtime()

```
double start_time, finish_time;

start_time = MPI_Wtime();    開始時刻を測定

........      ........ で挟まれた箇所の処理にかかる時間を測定したい。

finish_time = MPI_Wtime();   修了時刻を測定

if( rank == 0 )
  printf("It took %lf second.\n", finish_time - start_time);
```

Rank 0 に処理時間（終了時刻 − 開始時刻）を表示させる

課題

- 1からNまでの数列和を2台のプロセッサを用いて並列計算して結果を表示するプログラムを作れ。Nはキーボードから入力するものとする（rank0が担当する。）
- 下記積分をリーマン和として計算し、円周率の近似値を求めるプログラムを、1）1台のCPUを用いる場合、2）2台のCPUを用いる場合、の二通り作成し、処理時間を比較せよ。

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