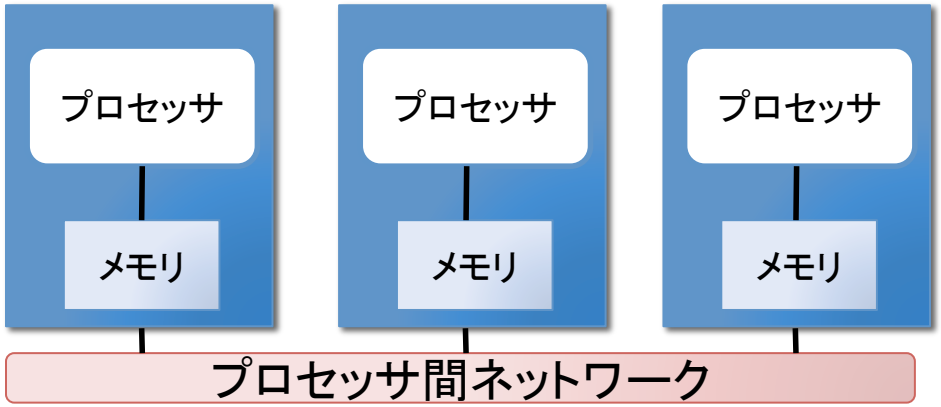
ハイパフォーマンスコンピューティング(6)

プログラムの並列化

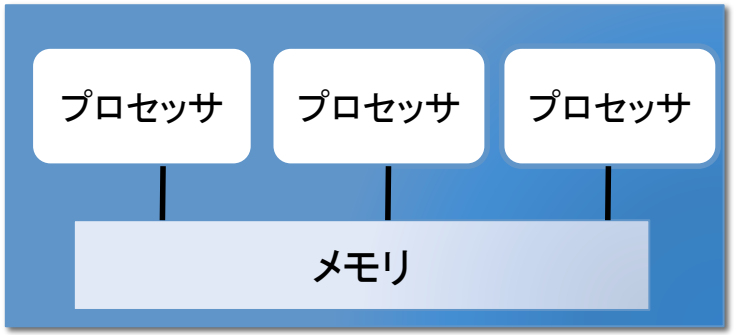
- プログラムの並列化には主に2つの種類があります。
  - 共有メモリ型（OpenMPなど）

分散メモリ型 (MPI)

- **MPIは、プロセッサ間のデータ通信規格。**
- **本講義では、その実装の1つであるMPICHまたはOpen MPIを使用。**



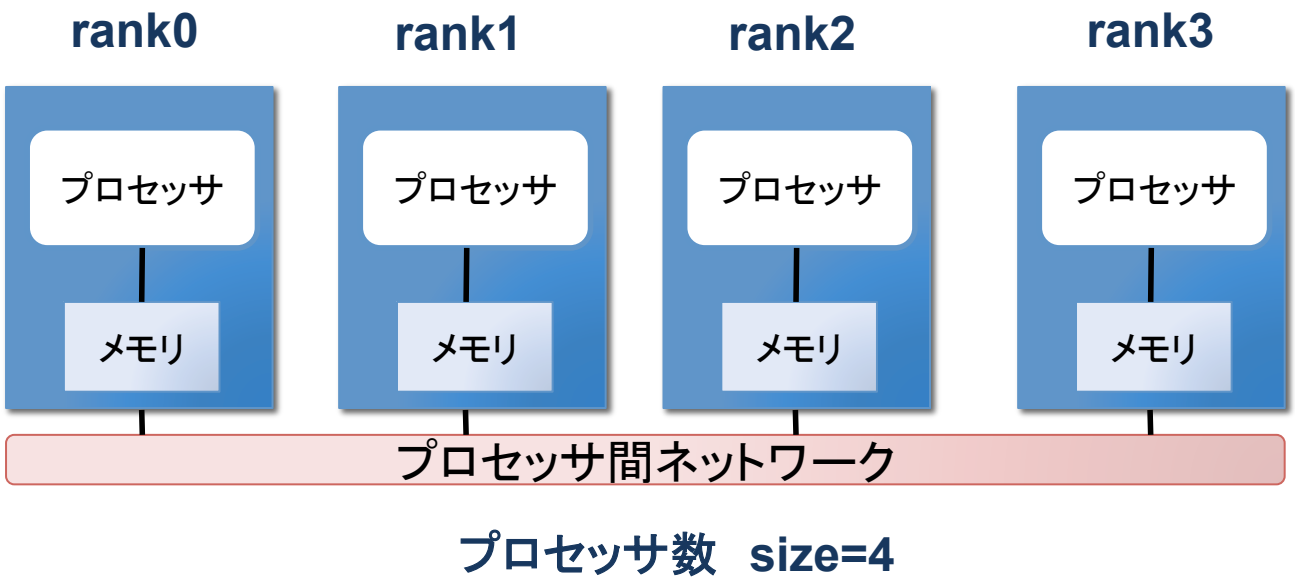
分散メモリ型並列イメージ



共有メモリ型並列イメージ

MPIとは

- **MPI(Message Passing Interface)**
- MPIでは、各プロセッサは固有ID(rank)を持つ
- rankに従う各プロセッサの動作をプログラムで記述。



環境設定：mpich2の起動方法

- 個人環境準備
  1. エディタでホームディレクトリに mpd.hosts を作成
     - ファイルの中身

     ```
     127.0.0.1  localhost
     ```
  2. エディタでホームディレクトリに .mpd.conf を作成（ファイル名の先頭のピリオドを忘れないように）
     - ファイルの中身

     ```
     MPD_SECRETWORD=hpc
     ```
  3. .mpd.conf のパーミッションを変更
     - $ chmod 700 .mpd.conf
- mpd起動
  1. mpdデーモン起動
     - $ mpd &
  2. mpdプロセス確認
     - $ mpdtrace -l
  3. mpdの終了
     - $ mpdallexit

MPIプログラム起動の確認

- MPIのための環境を準備したら、起動するかを確認する。

test.c

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

int main(int argc, char* argv[])
{
    printf("Hello.\n");

    return 0;
}
```

```
$ mpicc -O2 test.c
$ mpirun -np 4 ./a.out
```

MPIプログラムのコンパイルと実行

- コンパイル
  - $ mpicc [コンパイルオプション] [ファイル名]

- 実行
  - $ mpirun –np [プロセス数] [実行ファイル名]

実行例

```
$ mpicc -O2 test.c
$ mpirun -np 4 ./a.out      ・・・4並列の場合
Hello.
Hello.
Hello.
Hello.
```

MPIプログラムの基本

・**MPI_Init**により、MPIの実行環境の初期化を行い、最後に**MPI_Finalize**を実行する。
・.MPI_Initにはmain関数の引数のポインタを渡す。

```
#include <mpi.h>
    MPI_Init(&argc, &argv);
    …
    MPI_Finalize( );
```

・MPIプログラムの各実行プロセスには、**ランク**が設定され、ランク毎に処理を分岐させる。プロセスのランクは以下で取得する。

```
MPI_Comm_rank(MPI_COMM_定数, &my_rank);
```

【定数】
MPI_COMM_WORLD：すべてのプロセス
MPI_COMM_SELF：その定数を呼び出しているプロセス

・プロセスの総数は以下で取得する。

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

MPIプログラムの基本

・簡単なプログラムを通して、MPIの基本を学習する。

mhello.c

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

int main(int argc, char* argv[])
{
    int my_rank, p;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    printf("Hello, I am %d of %d.\n", my_rank, p);
    if (my_rank == 0) printf("my_rank is 0.\n");
    MPI_Finalize( );

    return 0;
}
```

送信・受信（1対1）

・MPI_Sendはメッセージを送信する関数である.

MPI_Send(①&a_send, ②n, ③MPI_INT, ④1, ⑤tag, ⑥MPI_COMM_WORLD);

・MPI_Sendの各パラメータの説明を以下に示す.

①送信するデータのアドレス.
②送信するデータ数.
③送信するデータの型（MPI_INT, MPI_DOUBLE）.
④送信先プロセッサのrank.
⑤メッセージのタグ.
⑥送受信グループ

タグという整数の情報を付加することで、受信側はタグの値を参照して,受信メッセージに対してどういった処理を施すべきかを判断できるようになる

・MPI_Recvはメッセージを受信する関数である.

MPI_Recv(①&b_recv, ②n, ③ MPI_INT, ④ 0,⑤ tag, ⑥MPI_COMM_WORLD, ⑦&status);

・MPI_Recvの各パラメータの説明を以下に示す.

①受信データが格納されるアドレス
②受信するデータ数
③受信するデータの型（MPI_INT, MPI_DOUBLE）.
④送信元プロセッサのrank.
⑤メッセージのタグ.
⑥送受信グループ
⑦受信結果の情報を格納するアドレス

MPI_Send, MPI_Recvの例

sendrecv.c

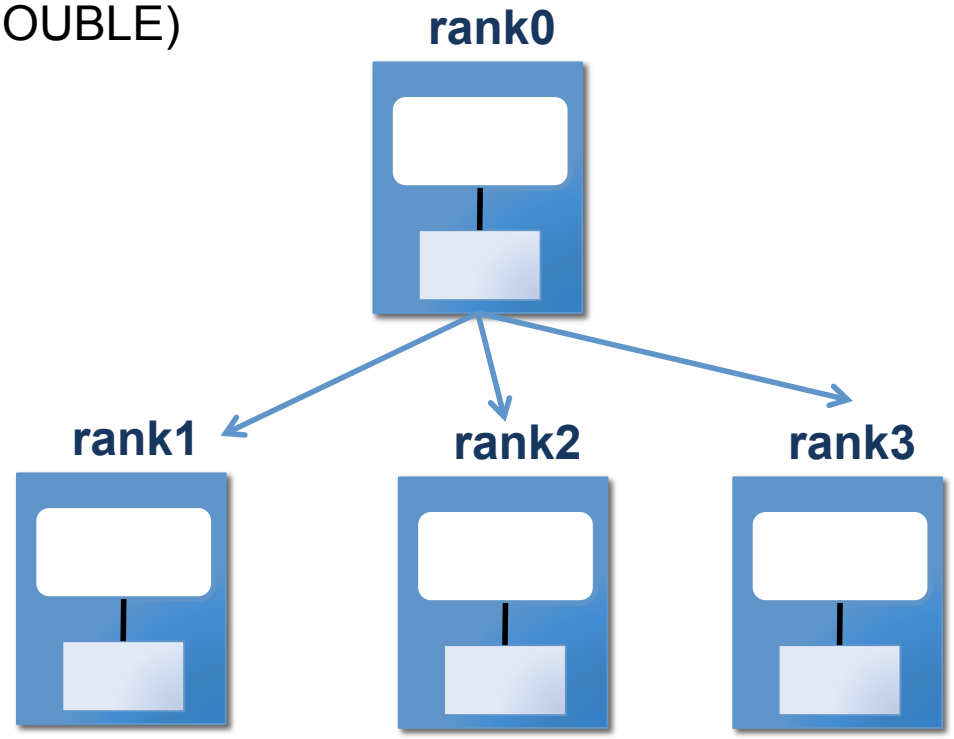
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int my_rank, p, a_send = 0, b_recv = 0, tag =1000;
    MPI_Status status;  /* MPIによる通信状態を保存するための変数 */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if (my_rank == 0) {
      a_send = 5;
/* rank0 sends data to rank1 */
      MPI_Send(&a_send, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
    } else if (my_rank == 1) {
/* rank1 receives data from rank0 */
      MPI_Recv(&b_recv, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
    }
    printf("my_rank = %d: a_send = %d, : b_recv = %d\n", my_rank, a_send, b_recv);
MPI_Finalize( );
    return 0;
}
```

プロセッサ間通信

・複数ノードへの一斉送信・受信を行う

MPI_Bcast(①&a, ②n, ③MPI_INT, ④0, ⑤MPI_COMM_WORLD);

①受信データが格納されるアドレス
②受信するデータの個数
③受信するデータ型(MPI_INT,MPI_DOUBLE)
④送信元のプロセッサ rank
⑤通信を行うグループの指定

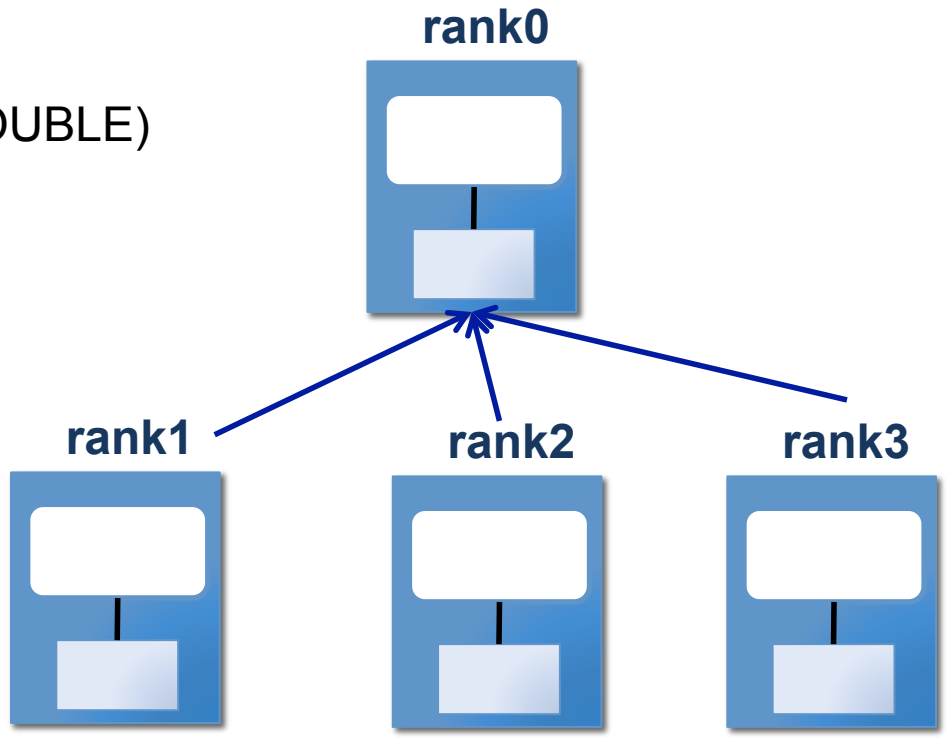


プロセッサ間通信

・各ノードの結果を集約する。

MPI_Reduce(①&my_s, ②&s, ③1, ④MPI_DOUBLE, ⑤MPI_SUM, ⑥0, ⑦MPI_COMM_WORLD);

①受信データが格納されるアドレス
②集約結果を格納するアドレス
③受信するデータ数
④受信するデータ型(MPI_INT,MPI_DOUBLE)
⑤集約処理の指定
⑥送信先のプロセッサ rank
⑦通信を行うグループの指定



※⑤の指定は以下の通り

MPI_SUM	和
MPI_PROD	積
MPI_MAX	最大値
MPI_MIN	最小値

実習：MPI_Bcast, MPI_Reduce

・1からnまでの総和をもとめるプログラム。プロセッサ数を増やしたとき、処理時間が短くなる事を確認しましょう

msum.c

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

#define ND 100000000
double a[ND];

int main(int argc, char* argv[])
{
    int my_rank, p, n, i, m, istart, iend;
    double s, my_s, tic, toc;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if (my_rank == 0) {
      printf("Input n\n");
      scanf("%d", &n);
    }
```

```
/* Broadcast */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    for (i = 0; i < n; i++) a[i] = (double)(i + 1);
    m = n/p; /* #data/proc */
    istart = my_rank*m;
    iend = istart + m - 1;
    if (my_rank == p - 1) iend = n - 1;
    tic = MPI_Wtime();
    my_s = 0.0;
    for (i = istart; i <= iend; i++) my_s += a[i];

    MPI_Reduce(&my_s, &s, 1, MPI_DOUBLE, MPI_SUM, 0,
                    MPI_COMM_WORLD);
    toc = MPI_Wtime();
    printf("rank = %d, my_s = %f\n", my_rank, my_s);
    if (my_rank == 0) printf("s = %f, time = %f\n", s, toc - tic);
    MPI_Finalize( );

    return 0;
}
```

（※）呼び出したタスクの経過時間を返す

MPI_Wtime(void)