演習準備

2014年3月5日

神戸大学大学院システム情報学研究科
森下浩二

演習準備の内容

- 神戸大FX10（π-Computer）利用準備
  - システム概要
  - ログイン方法
  - コンパイルとジョブ実行方法
- MPI復習
  1. MPIプログラムの基本構成
  2. 並列実行
  3. 1対1通信、集団通信
  4. データ・処理分割
  5. 計算時間計測

神戸大FX10（π-Computer）利用準備

神戸大FX10（π-Computer）

- 富士通PRIMEHPC FX10：1ラック
  - SPARC64™ IXfx プロセッサ x 96ノード
  - 総理論演算性能：20.2TFLOPS
  - 総主記憶容量：3TByte
- 1ノード諸元表（京との比較）

	FX10 (SPARC64™ IXfx)	京 (SPARC64™ VIIIfx)
コア数	16	8
L1キャッシュ（コア）	32KB(D)/32KB(I)	←
共有L2キャッシュ	12MB	6MB
動作周波数	1.65GHz	2.0GHz
理論演算性能	211.2GFlops	128GFlops
メモリ容量	32GB	16GB

FX10 へのログイン方法

- 公開鍵認証によりログイン

- 手順の詳細は別紙を参照
  1. 鍵ペア（公開鍵・秘密鍵）の作成
  2. 仮の鍵ペアでログイン
  3. 自身の公開鍵を登録
  4. 自身の鍵ペアでログイン出来ることを確認
  5. 仮の公開鍵を削除

コンパイル方法

F : Fortran　C : C言語

- 逐次プログラム

```
$ frtpx sample.f90        F
$ fccpx sample.c          C
```

- OpenMP（ノード内スレッド並列）

```
$ frtpx -Kopenmp sample.f90        F
$ fccpx -Kopenmp sample.c          C
```

- MPI（ノード間プロセス並列）

```
$ mpifrtpx sample.f90      F
$ mpifccpx sample.c        C
```

ジョブ実行方法

- ジョブスクリプトの作成
  - single.sh：逐次ジョブ

```
#!/bin/sh
#PJM -L "rscgrp=school"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j
#
./a.out
```

←シェルを指定
←利用リソースグループ名
←利用ノード数
←最大経過時間（hh:mm:ss）
←標準エラー出力をマージして出力
←プログラムの実行

- ジョブの投入

```
$ pjsub single.sh
```

ジョブ実行方法（OpenMP）

- ジョブスクリプトの作成
  - parallel_omp.sh：スレッド並列（OpenMP）ジョブ

```
#!/bin/sh
#PJM -L "rscgrp=school"
#PJM -L "node=1"
#PJM -L "elapse=10:00"
#PJM -j
#
export OMP_NUM_THREADS=16
./a.out
```

←シェルを指定
←利用リソースグループ名
←利用ノード数
←最大経過時間（hh:mm:ss）
←標準エラー出力をマージして出力
←OpenMP並列数を指定
←プログラムの実行

環境変数 OMP_NUM_THREADS にOpenMP並列数を設定

ジョブ実行方法（MPI）

- ジョブスクリプトの作成
  - parallel_mpi.sh：プロセス並列（MPI）ジョブ

```
#!/bin/sh
#PJM -L "rscgrp=school"
#PJM -L "node=4"
#PJM -L "elapse=10:00"
#PJM -j
#
mpiexec ./a.out
```

←シェルを指定
←利用リソースグループ名
←利用ノード数
←最大経過時間（hh:mm:ss）
←標準エラー出力をマージして出力
←MPIプログラムの実行

利用ノード数にMPIによるプロセス並列数を設定

ジョブの管理

- ジョブの状態表示

```
$ pjstat [option]
```

  - “-v”オプション：詳細なジョブ情報を表示
  - “-H”オプション：終了したジョブ情報を表示
  - “-A”オプション：全ユーザのジョブ情報を表示
- ジョブのキャンセル

```
$ pjdel [JOB_ID]
```

  - [JOB_ID]はジョブ投入時に表示（“pjstat”でも確認可）
  - 例）[JOB_ID]が12345のジョブをキャンセル

```
$ pjdel 12345
```

ジョブ結果の確認

- 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ディレクトリに出力される
  - 標準出力ファイル: ジョブ名.oXXXXX
  - 標準エラー出力ファイル: ジョブ名.eXXXXX
    - デフォルトのジョブ名はジョブスクリプトのファイル名
    - XXXXXには[JOB_ID]が入る
    - ジョブスクリプト内で“#PJM -j”を指定した場合には、標準エラー出力はマージされ標準出力ファイルのみ出力される
  - 例）p.7の single.sh を投入し、[JOB_ID]に12345が割り当てられた場合: single.sh.o12345 が出力

ジョブ実行手順の例

1. プログラムの作成 — `$ vi sample.f90`
2. プログラムのコンパイル — `$ frtpx sample.f90`
   - コンパイルし実行ファイル(a.out)を作成
3. ジョブスクリプトの作成 — `$ vi single.sh`
4. ジョブの投入 — `$ pjsub single.sh`
   - ジョブ投入時の出力例: `[INFO] PJM 0000 pjsub Job XXXXX submitted.`
   - **XXXXX** が割り当てられたジョブ番号
5. ジョブ状態の確認 — `$ pjstat`
6. 結果の確認 — `$ cat single.sh.oXXXXX`

MPI復習（Fortran編）

1. MPIプログラムの基本構成

F

```
program main
   use mpi    ←MPIモジュールを読み込み
   implicit none
   integer :: nprocs, myrank, ierr

   call mpi_init( ierr )   ←MPIの初期化処理
   call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
                                   ←MPIプロセス数を nprocs に取得
   call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
                                   ←自身のプロセス番号を myrank に取得

   （この部分に並列実行したい処理を記述）

   call mpi_finalize( ierr )    ←MPIの終了処理
end program main
```

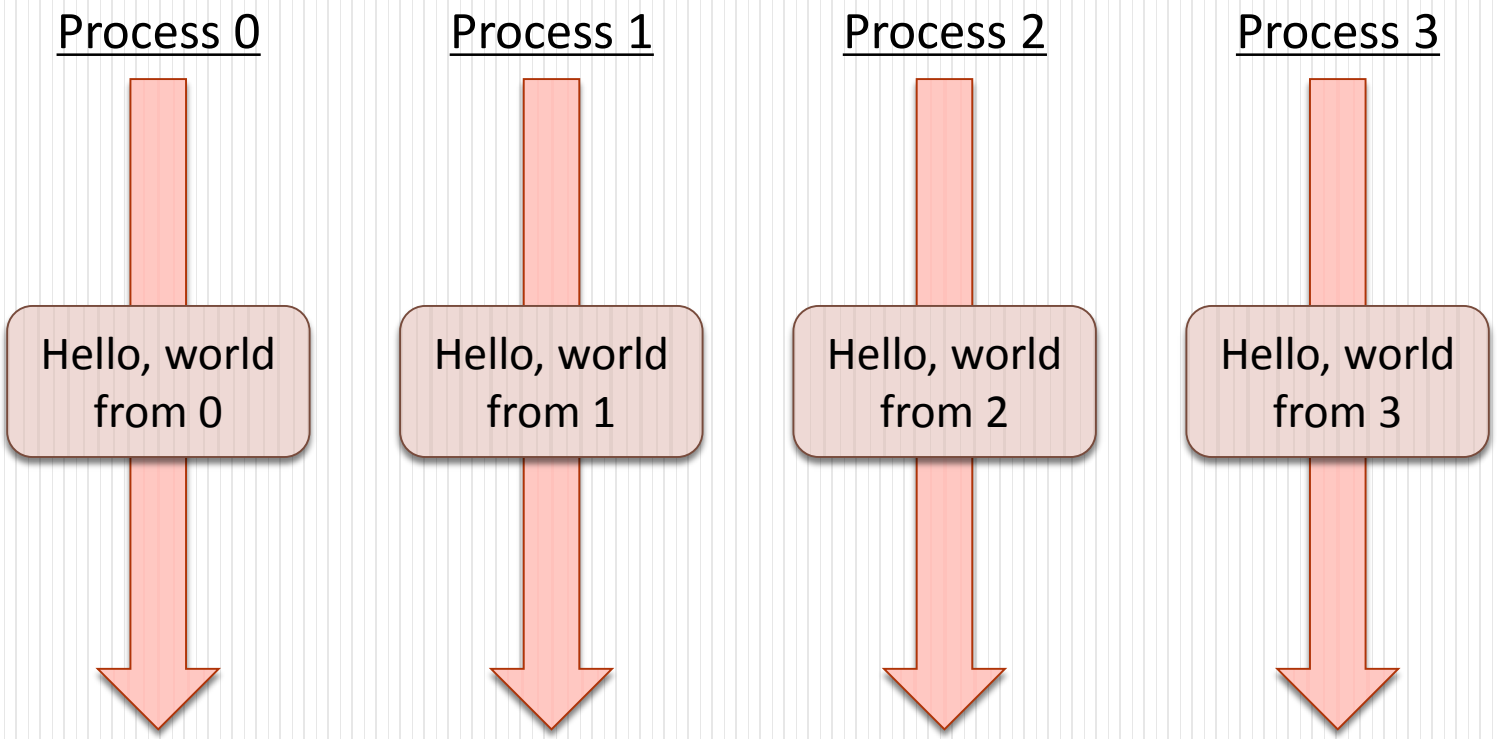
それぞれのプロセスが何の計算をするかは、nprocs や myrank の値で場合分けし、うまく仕事が割り振られるようにする

MPIプログラムの基本構成（説明）

- `mpi_init( ierr )`
  - MPIの初期化処理をする（MPIプログラムの最初に必ず書く）
- `mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )`
  - MPIの全プロセス数を取得し、2番目の引数 nprocs（整数型）に取得する
  - MPI_COMM_WORLDはコミュニケータと呼ばれ，最初に割り当てられるすべてのプロセスの集合
- `mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )`
  - 自分のプロセス番号（0からnprocs-1のどれか）を、2番目の引数 myrank（整数型）に取得する
- `mpi_finalize( ierr )`
  - MPIの終了処理をする（MPIプログラムの最後に必ず書く）

2. 並列処理

“Hello, world from（プロセス番号）”を並列に出力する



プログラム1

- ソースファイル：mpi1.f90

```
program mpi1
   use mpi
   implicit none
   integer :: nprocs, myrank, ierr

   call mpi_init( ierr )
   call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
   call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )

   write(*, *) "Hello, world from", myrank

   call mpi_finalize( ierr )
end program mpi1
```

F

並列処理：実習

実習1

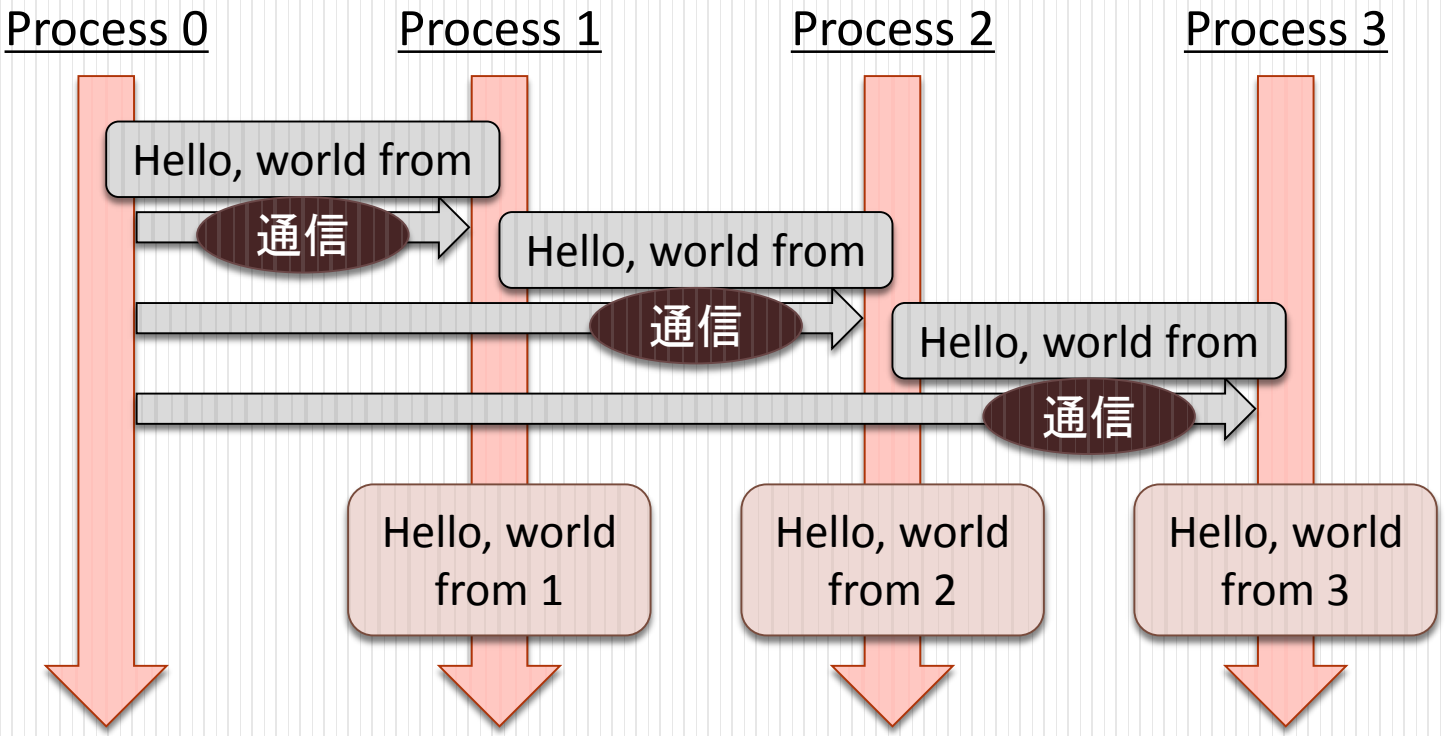
プログラム1: mpi1.f90 を作成し、コンパイルした後、4 プロセスで実行し、結果を確認してください

- 実行結果の例
  ※必ずしもプロセスの順番に出力されるとは限らない

```
Hello, world from      1
Hello, world from      0
Hello, world from      3
Hello, world from      2
```

3. 通信

MPIプロセス番号 0 から受け取ったメッセージ “Hello, world from”に自分のプロセス番号を追加して出力する



通信関数

- 1対1通信
  - mpi_send（送信）
  - mpi_recv（受信）
  - etc…
- 集団通信
  - mpi_bcast（ブロードキャスト）
  - mpi_reduce（リダクション）
  - mpi_allreduce（リダクション+ブロードキャスト）
  - etc…

1対1通信：送信関数

- `mpi_send( buff, count, datatype, dest, tag, comm, ierr )`

`buff:`	送信するデータの変数名（先頭アドレス）
`count:`	送信するデータの個数（整数型）
`datatype:`	送信するデータの型（MPI_INTEGER, MPI_REAL8, MPI_CHARACTER など）
`dest:`	送信先のプロセス番号
`tag:`	メッセージ識別番号
`comm:`	コミュニケータ（例えば、MPI_COMM_WORLD）
`ierr:`	戻りコード（整数型）

1対1通信：受信関数

- `mpi_recv( buff, count, datatype, source, tag, comm, status, ierr )`

`buff:`	受信するデータのための変数名（先頭アドレス）
`count:`	受信するデータの個数（整数型）
`datatype:`	受信するデータの型（MPI_INTEGER, MPI_REAL8, MPI_CHARACTER など）
`source:`	送信元のプロセス番号
`tag:`	メッセージ識別番号
`comm:`	コミュニケータ（例えば、MPI_COMM_WORLD）
`status:`	受信状態を格納するサイズMPI_STATUS_SIZEの配列（整数型）
`ierr:`	戻りコード（整数型）

プログラム2

- ソースファイル：mpi2.f90

F

```
program mpi2
   use mpi
   implicit none
   integer :: nprocs, myrank, ierr, i, mst(MPI_STATUS_SIZE)
   character (len=17) :: msg

   call mpi_init( ierr )
   call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
   call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )

   if( myrank == 0 ) then
      msg  = "Hello, world from"
      do i = 1, 3
         call mpi_send( msg, len(msg), MPI_CHARACTER, i, 0, MPI_COMM_WORLD, ierr )
      end do
   else
      call mpi_recv( msg, len(msg), MPI_CHARACTER, 0, 0, MPI_COMM_WORLD, mst, ierr )
      write(*, '(a,i5)') msg, myrank
   end if

   call mpi_finalize( ierr )
end program mpi2
```

集団通信：ブロードキャスト

- root に指定したプロセスが持つ buff の値を，comm 内の他のプロセスの buff に配布する

• mpi_bcast(buff, count, datatype, root, comm, ierr)	
buff:	root が送信するデータの変数名（先頭アドレス） 他のプロセスは、同じ変数名でデータを受け取る
count:	送受信するデータの個数（整数型）
datatype:	送受信するデータの型：MPI_INTEGER, MPI_REAL8, MPI_CHARACTER など
root:	送信元のプロセス番号
comm:	コミュニケータ（例えば、MPI_COMM_WORLD）
ierr:	戻りコード（整数型）

プログラム3

- ソースファイル：mpi3.f90

```
program mpi3
   use mpi
   implicit none
   integer :: nprocs, myrank, ierr
   character (len=17) :: msg

   call mpi_init( ierr )
   call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
   call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )

   if( myrank == 0 ) then
      msg  = "Hello, world from"
   end if

   call mpi_bcast( msg, len(msg), MPI_CHARACTER, 0, MPI_COMM_WORLD, ierr )

   if( myrank /= 0 ) then
      write(*, '(a,i5)') msg, myrank
   end if

   call mpi_finalize( ierr )
end program mpi3
```

F

通信：実習

実習2

プログラム2: mpi2.f90, プログラム3: mpi3.f90 を作成し、コンパイルした後、4 プロセスで実行し、結果を確認してください

- 実行結果の例
  ※必ずしもプロセスの順番に出力されるとは限らない

```
Hello, world from      1
Hello, world from      3
Hello, world from      2
```

4. データ・処理分割

大きさ n の2個のベクトルの内積を計算するプログラムを並列化

F

```
program main
   implicit none
   integer :: i
   integer, parameter :: n=10000
   real(8) :: v(n), w(n)
   real(8) :: ipr

   do i = 1, n
      v(i) = dsin(i*0.1d0)
      w(i) = dcos(i*0.1d0)
   end do

   ipr = 0.0d0
   do i = 1, n
      ipr = ipr + v(i)*w(i)
   end do

   write(*, '(a,f20.15)') "answer:", ipr

end program main
```

- 各プロセスにデータ・処理を分散
  - 例えば，n=10000 のベクトルを4個のプロセスで計算する場合
    - プロセス0:　　1- 2500 のループ部分を処理
    - プロセス1: 2501- 5000 のループ部分を処理
    - プロセス2: 5001- 7500 のループ部分を処理
    - プロセス3: 7501-10000 のループ部分を処理
- 各プロセスの部分和のリダクションが必要
  - mpi_reduce または mpi_allreduce

集団通信:リダクション

- comm 内のすべてのプロセスからデータを rootへ集め，演算(op)を適用する

- `mpi_reduce( sendbuff, recvbuff, count, datatype, op, root, comm, ierr )`

sendbuff:	送信するデータの変数名(先頭アドレス)
recvbuff:	受信するデータの変数名(先頭アドレス)
count:	送受信するデータの個数(整数型)
datatype:	送受信するデータの型
op:	データに適用する演算の種類:MPI_SUM(総和)、MPI_PROD(掛け算)、MPI_MAX(最大値)など
root:	送信先のプロセス番号
comm:	コミュニケータ(例えば、MPI_COMM_WORLD)
ierr:	戻りコード(整数型)

集団通信:リダクション

- comm 内のすべてのプロセスのデータに対して演算(op)を適用し、その結果をすべてのプロセスへ配布する

- `mpi_allreduce( sendbuff, recvbuff, count, datatype, op, comm, ierr )`

`sendbuff:`	送信するデータの変数名(先頭アドレス)
`recvbuff:`	受信するデータの変数名(先頭アドレス)
`count:`	送受信するデータの個数(整数型)
`datatype:`	送受信するデータの型
`op:`	データに適用する演算の種類:MPI_SUM(総和)、MPI_PROD(掛け算)、MPI_MAX(最大値)など
`comm:`	コミュニケータ(例えば、MPI_COMM_WORLD)
`ierr:`	戻りコード(整数型)

プログラム4

- ソースファイル
  : mpi4.f90

F

```
program mpi4
   use mpi
   implicit none
   integer :: i, ista, iend
   integer, parameter :: n=10000
   real(8) :: v(n), w(n)
   real(8) :: ipr, ans
   integer :: nprocs, myrank, ierr
   call mpi_init( ierr )
   call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
   call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )

   ista = myrank*n/nprocs + 1
   iend = (myrank+1)*n/nprocs

   do i = ista, iend
      v(i) = sin(i*0.1d0)
      w(i) = cos(i*0.1d0)
   end do

   ipr = 0.0d0
   do i = ista, iend
      ipr = ipr + v(i)*w(i)
   end do

   call mpi_allreduce( ipr, ans, 1, MPI_REAL8, &
   &                   MPI_SUM, MPI_COMM_WORLD, ierr )

   write(*, '(a,i5,a,f20.15)') "rank:", myrank, &
   &                           ", answer:", ans

   call mpi_finalize( ierr )
end program mpi4
```

データ・処理分割：実習

実習3

プログラム4: mpi4.f90 を作成し、コンパイルした後、4 プロセスで実行し、結果を確認してください

- 実行結果の例

```
rank:     0, answer:    3.639755648373931
rank:     1, answer:    3.639755648373931
rank:     3, answer:    3.639755648373931
rank:     2, answer:    3.639755648373931
```

5. 計算時間計測

```
real(8) :: time1, time2   ←計測のための変数を倍精度実数で宣言する
 ...
call mpi_barrier( MPI_COMM_WORLD, ierr )  ←開始の足並みを揃える
time1 = mpi_wtime()                       ←開始時刻を time1 に設定

（計測する部分）

call mpi_barrier( MPI_COMM_WORLD, ierr )  ←終了の足並みを揃える
time2 = mpi_wtime()                       ←終了時刻を time2 に設定

（time2-time1 を出力）   ←time2 - time1が計測した部分の計算時間となる
```

F

- `mpi_barrier( comm, ierr )`
  - comm 内の最も遅いプロセスが到達するまで、全プロセスが待つ
- `mpi_wtime()`
  - ある時点を基準とした経過秒数を倍精度実数で返す関数

計算時間計測：実習

実習4

1. プログラム4: mpi4.f90 を、並列処理部分の計算時間を計測して出力するよう修正してください
2. 並列数を変えて実行し計算時間がどう変わるか測定し、以下の表を完成させてください

並列数 n	計算時間 T(n)（秒）	速度向上率 T(n)/T(1)（n=1の計算時間T(1)との比）
1		1.0
2		
4		
8		
16		

MPI復習（C言語編）

1. MPIプログラムの基本構成

C

```
#include "stdio.h"
#include "mpi.h"   ←MPIライブラリを読み込み
int main( int argc, char **argv ){
    int nprocs, myrank;

    MPI_Init( &argc, &argv );  ←MPIの初期化処理
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
                                        ←MPIプロセス数を nprocs に取得
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
                                      ←自身のプロセス番号を myrank に取得
   （この部分に並列実行したい処理を記述）
    MPI_Finalize();    ←MPIの終了処理
    return 0;
}
```

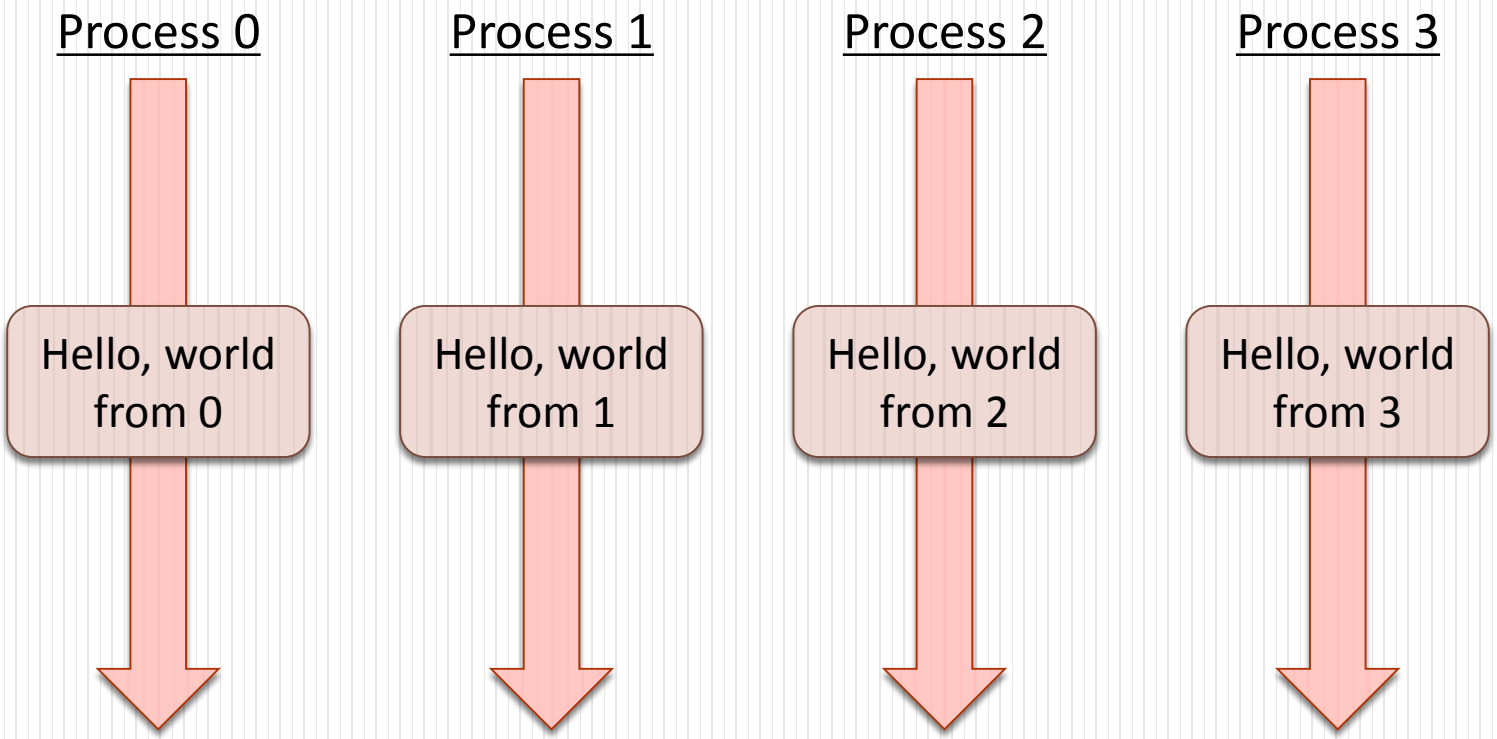
それぞれのプロセスが何の計算をするかは、nprocs や myrank の値で場合分けし、うまく仕事が割り振られるようにする

MPIプログラムの基本構成（説明）

- `int MPI_Init( int *argc, char ***argv )`
  - MPIの初期化処理をする（MPIプログラムの最初に必ず書く）
- `int MPI_Comm_size( MPI_Comm comm, int *nprocs )`
  - MPIの全プロセス数を取得し、2番目の引数 nprocs（整数型）に取得する
  - MPI_COMM_WORLDはコミュニケータと呼ばれ，最初に割り当てられるすべてのプロセスの集合
- `int MPI_Comm_rank( MPI_Comm comm, int *myrank )`
  - 自分のプロセス番号（0からnprocs-1のどれか）を、2番目の引数 myrank（整数型）に取得する
- `int MPI_Finalize( void )`
  - MPIの終了処理をする（MPIプログラムの最後に必ず書く）

2. 並列処理

“Hello, world from（プロセス番号）”を並列に出力する



プログラム1

- ソースファイル：mpi1.c

C

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

int main( int argc, char **argv ){
   int nprocs, myrank;

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

   printf( "Hello, world from %3d¥n", myrank );

   MPI_Finalize();
   return 0;
}
```

並列処理：実習

実習1

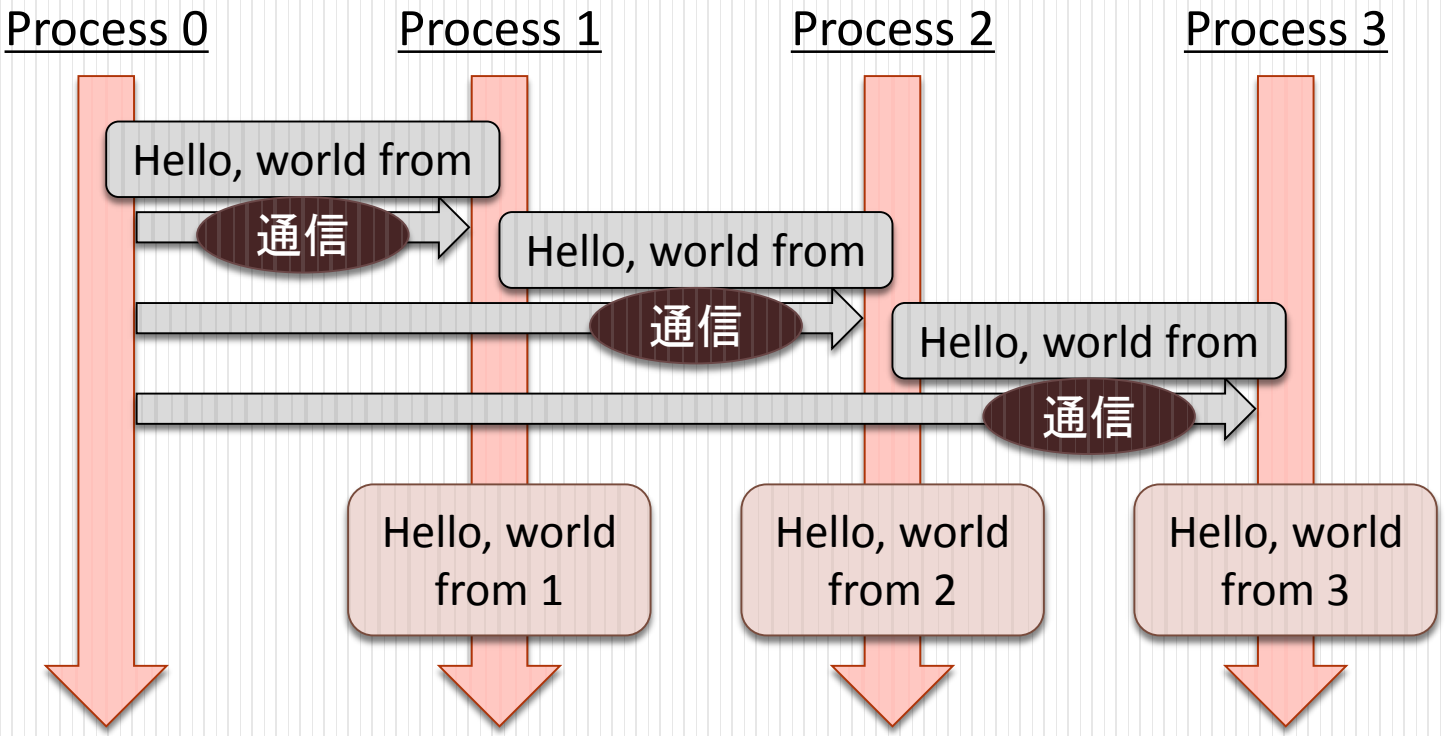
プログラム1: mpi1.c を作成し、コンパイルした後、
4 プロセスで実行し、結果を確認してください

- 実行結果の例
  ※必ずしもプロセスの順番に出力されるとは限らない

```
Hello, world from    1
Hello, world from    0
Hello, world from    3
Hello, world from    2
```

3. 通信

MPIプロセス番号 0 から受け取ったメッセージ “Hello, world from”に自分のプロセス番号を追加して出力する



通信関数

- 1対1通信
  - MPI_Send（送信）
  - MPI_Recv（受信）
  - etc…
- 集団通信
  - MPI_Bcast（ブロードキャスト）
  - MPI_Reduce（リダクション）
  - MPI_Allreduce（リダクション+ブロードキャスト）
  - etc…

1対1通信：送信関数

- `int MPI_Send( void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`

buff:	送信するデータの変数名（先頭アドレス）
count:	送信するデータの個数（整数型）
datatype:	送信するデータの型（MPI_INT, MPI_DOUBLE, MPI_CHAR など）
dest:	送信先のプロセス番号
tag:	メッセージ識別番号
comm:	コミュニケータ（例えば、MPI_COMM_WORLD）
戻り値:	戻りコード（整数型）

1対1通信：受信関数

- `int MPI_Recv( void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )`

buff:	受信するデータのための変数名（先頭アドレス）
count:	受信するデータの個数（整数型）
datatype:	受信するデータの型（MPI_INT, MPI_DOUBLE, MPI_CHAR など）
source:	送信元のプロセス番号
tag:	メッセージ識別番号
comm:	コミュニケータ（例えば、MPI_COMM_WORLD）
status:	受信状態を格納するオブジェクト
戻り値:	戻りコード（整数型）

プログラム2

- ソースファイル
  : mpi2.c

C

```c
#include "stdio.h"
#include "string.h"
#include "mpi.h"
int main( int argc, char **argv ){
   int nprocs, myrank, i;
   char msg[18];
   MPI_Status mst;

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

   if( myrank == 0 ) {
      strcpy( msg, "Hello, world from" );
      for( i = 1; i <= 3 ; i++ ){
         MPI_Send( &msg, 17, MPI_CHAR, i, 0, MPI_COMM_WORLD );
      }
   }
   else {
      MPI_Recv( &msg, 17, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &mst );
      printf( "%s %3d¥n", msg, myrank );
   }

   MPI_Finalize();
   return 0;
}
```

集団通信：ブロードキャスト

- root に指定したプロセスが持つ buff の値を，comm 内の他のプロセスの buff に配布する

- `int MPI_Bcast( void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`

buff:	root が送信するデータの変数名（先頭アドレス） 他のプロセスは、同じ変数名でデータを受け取る
count:	送受信するデータの個数（整数型）
datatype:	送受信するデータの型
root:	送信元のプロセス番号
comm:	コミュニケータ（例えば、MPI_COMM_WORLD）
戻り値:	戻りコード（整数型）

プログラム3

- ソースファイル：mpi3.c

C

```
#include "stdio.h"
#include "string.h"
#include "mpi.h"
int main( int argc, char **argv ){
   int nprocs, myrank;
   char msg[18];
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

   if( myrank == 0 ) {
      strcpy( msg, "Hello, world from" );
   }

   MPI_Bcast( &msg, 17, MPI_CHAR, 0, MPI_COMM_WORLD );

   if( myrank != 0) {
      printf( "%s %3d¥n", msg, myrank );
   }

   MPI_Finalize();
   return 0;
}
```

通信：実習

実習2

プログラム2: mpi2.c, プログラム3: mpi3.c を作成し、コンパイルした後、4 プロセスで実行し、結果を確認してください

- 実行結果の例
  ※必ずしもプロセスの順番に出力されるとは限らない

```
Hello, world from      1
Hello, world from      3
Hello, world from      2
```

4. データ・処理分割

大きさ n の2個のベクトルの内積を計算するプログラムを並列化

C

```
#include "stdio.h"
#inlucde "math.h"

int main( int argc, char **argv ){
  int i;
  const int n=10000;
  double v[n], w[n];
  double ipr;

  for( i = 0; i < n; i++ ){
    v[i] = sin( 0.1*(i+1) );
    w[i] = cos( 0.1*(i+1) );
  }

  for( ipr = 0.0, i = 0; i < n; i++ ){
    ipr += v[i]*v[i]
  }

  printf( "answer: %20.15lf¥n", ipr );
  return 0;
}
```

- 各プロセスにデータ・処理を分散
  - 例えば，n=10000 のベクトルを4個のプロセスで計算する場合
    - プロセス0: 0- 2499 のループ部分を処理
    - プロセス1: 2500- 4999 のループ部分を処理
    - プロセス2: 5000- 7499 のループ部分を処理
    - プロセス3: 7500- 9999 のループ部分を処理
- 各プロセスの部分和のリダクションが必要
  - MPI_Reduce または MPI_Allreduce

集団通信:リダクション

- comm 内のすべてのプロセスからデータを rootへ集め，演算(op)を適用する

- `int MPI_Reduce( void *sendbuff, void *recvbuff, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`

sendbuff:	送信するデータの変数名(先頭アドレス)
recvbuff:	受信するデータの変数名(先頭アドレス)
count:	送受信するデータの個数(整数型)
datatype:	送受信するデータの型
op:	データに適用する演算の種類:MPI_SUM(総和)、MPI_PROD(掛け算)、MPI_MAX(最大値)など
root:	送信先のプロセス番号
comm:	コミュニケータ(例えば、MPI_COMM_WORLD)
戻り値:	戻りコード(整数型)

集団通信:リダクション

- comm 内のすべてのプロセスのデータに対して演算(op)を適用し、その結果をすべてのプロセスへ配布する

- `int MPI_Allreduce( void *sendbuff, void *recvbuff, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )`

sendbuff:	送信するデータの変数名(先頭アドレス)
recvbuff:	受信するデータの変数名(先頭アドレス)
count:	送受信するデータの個数(整数型)
datatype:	送受信するデータの型
op:	データに適用する演算の種類:MPI_SUM(総和)、MPI_PROD(掛け算)、MPI_MAX(最大値)など
comm:	コミュニケータ(例えば、MPI_COMM_WORLD)
戻り値:	戻りコード(整数型)

プログラム4

- ソースファイル
  : mpi4.c

C

```
#include "stdio.h"
#include "mpi.h"
#include "math.h"
int main( int argc, char **argv ){
   int i, ista, iend;
   const int n=10000;
   double v[n], w[n];
   double ipr, ans;
   int nprocs, myrank;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

   ista = myrank*n/nprocs;
   iend = (myrank+1)*n/nprocs;

   for( i = ista; i < iend; i++ ){
      v[i] = sin( 0.1*(i+1) );
      w[i] = cos( 0.1*(i+1) );
   }

   for( ipr = 0.0, i = ista; i < iend; i++ ){
      ipr += v[i]*w[i];
   }

   MPI_Allreduce( &ipr, &ans, 1, MPI_DOUBLE, MPI_SUM,
                  MPI_COMM_WORLD );

   printf( "rank: %d, answer: %20.15lf¥n", myrank, ans );

   MPI_Finalize();
   return 0;
}
```

データ・処理分割：実習

実習3

プログラム: mpi4.c を作成し、コンパイルした後、
4 プロセスで実行し、結果を確認してください

- 実行結果の例

```
rank: 0, answer:    3.639755648373932
rank: 1, answer:    3.639755648373932
rank: 3, answer:    3.639755648373932
rank: 2, answer:    3.639755648373932
```

5. 計算時間計測

C

```
double time1, time2;      ←計測のための変数を倍精度実数で宣言する
 ...
MPI_Barrier( MPI_COMM_WORLD );      ←開始の足並みを揃える
time1 = MPI_Wtime();                ←開始時刻を time1 に設定

（計測する部分）

MPI_Barrier( MPI_COMM_WORLD );      ←終了の足並みを揃える
time2 = MPI_Wtime();                ←終了時刻を time2 に設定

（time2-time1 を出力）    ←time2 - time1が計測した部分の計算時間となる
```

- `int MPI_Barrier( MPI_Comm comm )`
  - comm 内の最も遅いプロセスが到達するまで、全プロセスが待つ
- `double MPI_Wtime( void )`
  - ある時点を基準とした経過秒数を倍精度実数で返す関数

計算時間計測：実習

実習4

1. プログラム4: mpi4.c を、並列処理部分の計算時間を計測して出力するよう修正してください
2. 並列数を変えて実行し計算時間がどう変わるか測定し、以下の表を完成させてください

並列数 n	計算時間 T(n)（秒）	速度向上率 T(n)/T(1)（n=1の計算時間T(1)との比）
1		1.0
2		
4		
8		
16		