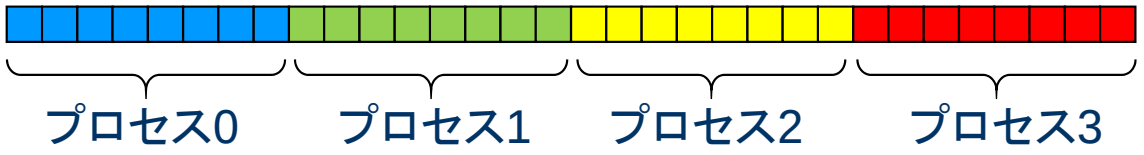
計算科学演習Ⅰ
MPIを用いた並列計算（III）

神戸大学大学院システム情報学研究科
横川 三津夫
yokokawa@port.kobe-u.ac.jp

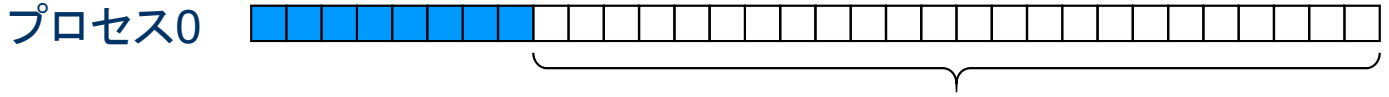
演習M2-2：ベクトルの正規化【再掲】

- n次元ベクトル $\boldsymbol{x}$ の第 i 要素を i とする（$x(i) = i$）.
- このとき，$\boldsymbol{x}$ を正規化したベクトル $\boldsymbol{x}/\|\boldsymbol{x}\|_2$ を求めるプログラムを作成せよ.
  - $\|\boldsymbol{x}\|_2$ は $\boldsymbol{x}$ の各要素の2乗和の平方根である.
  - ベクトルは，ブロック分割で各プロセスに配置する.

- 各プロセスの担当する要素（nprocs はMPIプロセス数）
  - `istart = (n/nprocs)*myrank + 1`
  - `iend   = (n/nprocs)*(myrank+1)`



- ベクトルの格納方法
  - 各プロセスは長さ n の配列を持ち，そのうち自分の担当部分のみを使う



プロセス0では，ここの部分が使われない

解答例

```
program dnorm2
use mpi
implicit none
integer, parameter :: n=1000
integer :: i,istart,iend
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP) :: sum_local, sum, error_local, error, const
real(DP) :: x(n)
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 )
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
sum_local = 0.0d0
do i = istart, iend
  x(i)      =  dble(i)
  sum_local = sum_local + x(i)*x(i)
end do
call mpi_allreduce( sum_local, sum, 1, MPI_REAL8, MPI_SUM, MPI_COMM_WORLD, ierr )
const = 1.0d0/sqrt(dble(n*(n+1)*(2*n+1))/6.0d0)
sum   = 1.0d0/sqrt(sum)
error_local = 0.0d0
do i = istart, iend
   x(i) = x(i)*sum
   error_local = error_local + abs( x(i) - i*const )
end do
call mpi_reduce( error_local, error, 1, MPI_REAL8, MPI_SUM, 0, MPI_COMM_WORLD, ierr )
if( myrank == 0 ) write(6,*) "Error = ", error
call mpi_finalize(ierr)
stop
end program
```

配列x(n)の宣言

配列xのうち，自分の担当する部分の要素をセット
要素の２乗の部分和を計算

部分和の合計の平方根の逆数

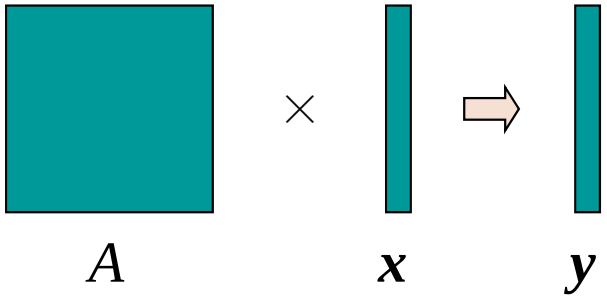
自分の担当する要素を正規化する

今週の講義の概要

1. 行列ベクトル積の並列計算
2. ブロッキング関数とデッドロック
   - mpi_sendrecv
   - mpi_isend，mpi_irecv，mpi_wait

リダクション演算の応用：行列ベクトル積

- 問題
  - n×n行列 A の第 (i,j) 要素を $i+j$ とする（$a_{i,j}=i+j$）．
  - n次元ベクトル $\boldsymbol{x}$ の第 i 要素を i とする（$x_i=i$）．
  - このとき，$\boldsymbol{y}=A\boldsymbol{x}$ を計算する．



- プログラムのカーネル（主要な部分）
  - ベクトル $\boldsymbol{y}$ の要素を1個ずつ計算する．
  - $\boldsymbol{y}$ の第 i 要素は，A の第 i 行とベクトル $\boldsymbol{x}$ との内積

$$y_i = \sum_j a_{i,j}\, x_j$$

逐次プログラム M-5（mv_s.f90）

```
program mv
implicit none
integer, parameter :: n=100
integer :: i, j
real(kind=8), dimension(n,n) :: a
real(kind=8), dimension(n)   :: x, y
real(kind=8) :: error, ans
do i = 1, n
  x(i) = i
end do
do i = 1, n
  do j = 1, n
    a(i,j) = dble(i+j)
  end do
end do
do i = 1, n
  y(i) = 0.0d0
  do j = 1, n
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
error = 0.0d0
do i = 1, n
  ans   = dble( i*n*(n+1)/2 + n*(n+1)*(2*n+1)/6)
  error = error + abs( y(i) - ans )
end do
print *, 'error =', error
end program mv
```

x(i), a(i,j) の初期設定

y=Axの計算

結果の確認

abs() は絶対値関数

演習M3-1 ：M-5プログラムの実行

- mv_s.f90 をコンパイルして，実行せよ

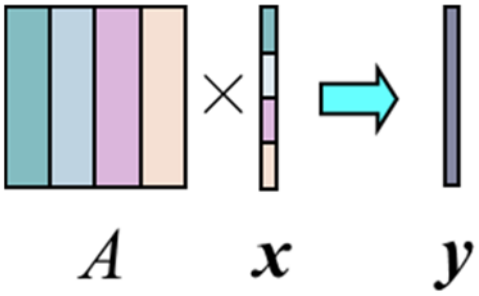
```
$ cp /tmp/mpi2/M-5/mv_s.f90 ./
$ f95 -o mv_s mv_s.f90
$ ./mv_s
```

- 結果が次のように表示されることを確認せよ.

```
$ ./mv_s
 error =    0.0000000000000000
```

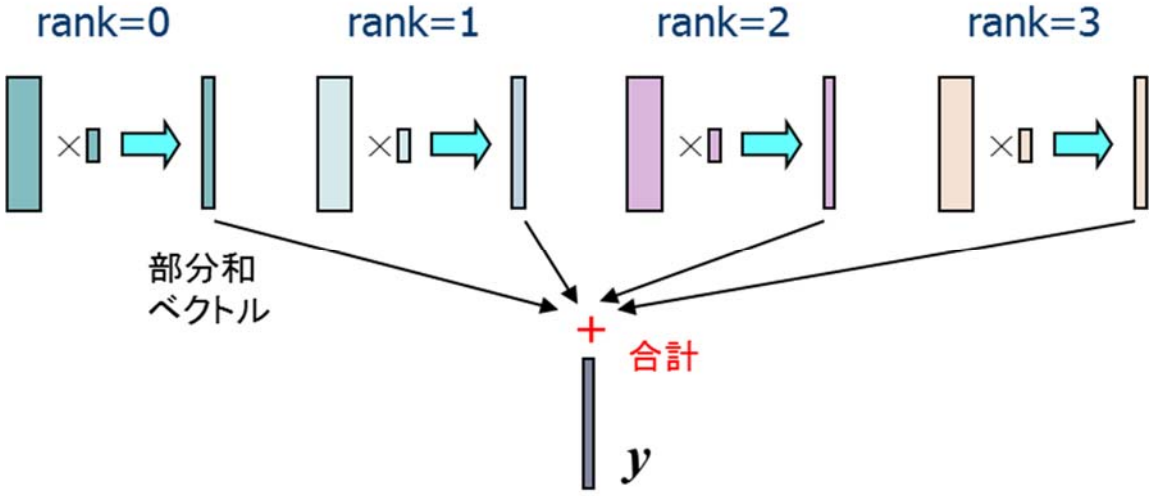
行列ベクトル積の並列化

- 行列とベクトルの分割
  - A は列方向に短冊形に分割（ブロック列分割）する.
  - $\boldsymbol{x}$ は行方向にブロック分割する.



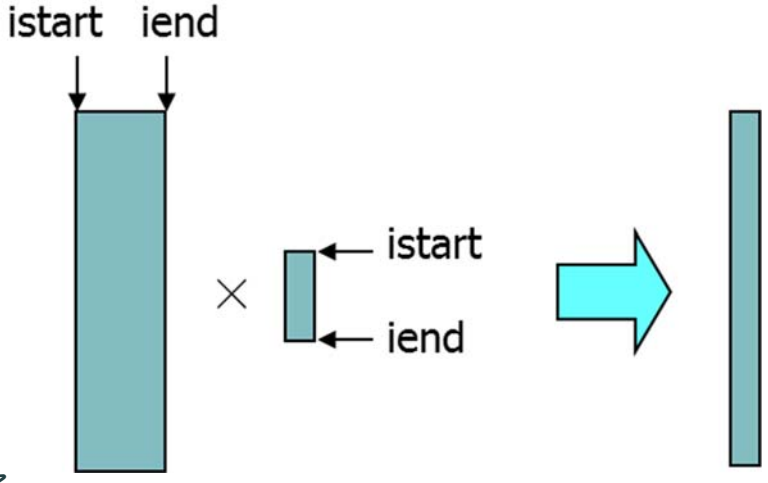
- プログラムの方針
  - 各プロセスが持つ自分の要素のみを使って，部分和ベクトルを計算
  - 部分和ベクトルを mpi_reduce関数によりプロセス0で集計する.

$$y_i = \sum_{\text{rank=0}} a_{i,j}\, x_j + \sum_{\text{rank=1}} a_{i,j}\, x_j + \sum_{\text{rank=2}} a_{i,j}\, x_j + \sum_{\text{rank=3}} a_{i,j}\, x_j$$



演習M3-2：mv_s.f90を並列化せよ（M-6）

- プログラム書き換えの方針
  - MPIの定義，初期化，終了処理を忘れないこと．
  - 各プロセスの計算範囲を求める
    - `istart = (n/nprocs)*myrank + 1`
    - `iend   = (n/nprocs)*(myrank+1)`



  - A，$\boldsymbol{x}$ について，各プロセスが担当する部分のみ初期化する．
    - A ：第 istart 列 ～ 第 iend 列
    - $\boldsymbol{x}$ ：第 istart 要素 ～ 第 iend 要素
  - 部分和ベクトルは，各プロセスの持つ要素のみを使って計算
    - 部分和ベクトルは，別の配列（例えば y_tmp）を用いる．
  - 部分和ベクトルの合計
    - mpi_reduce 関数により，ランク0のプロセスで，配列 y_tmpの合計を配列yに入れる．
    - mpi_reduce関数の第3引数（count）に注意（前回資料 29ページ）

演習M3-2：続き（提出課題）

- n=1000として，プロセス数1，2，4，及び8と変化させて実行させ，結果が正しいことを確認せよ.

- そのときの計算時間の変化を調べよ.
  - 初期設定，結果の確認部分は，計測範囲に含めないこと.
  - プロセス数（n），計算時間（T_n），加速率（$S_n = T_1/T_n$）をまとめる.

n	T_n	S_n
1	xxxxxxx	1.000
2	xxxxxxx	xxxxxxx
4	xxxxxxx	xxxxxxx
8	xxxxxxx	xxxxxxx

MPIプログラム M-7 ：デッドロック

```
program deadlock
use mpi
implicit none

integer, parameter :: n=10
double precision  :: a0(n), a1(n)
integer :: nprocs, myrank, ierr
integer :: istat(MPI_STATUS_SIZE)
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
  a0 = 1.0
else
  a1 = 2.0
endif

if( myrank == 0 ) then
   call mpi_send( a0, n, MPI_REAL8, 1, 100, MPI_COMM_WORLD, ierr )
   call mpi_recv( a1, n, MPI_REAL8, 1, 200, MPI_COMM_WORLD, istat, ierr )
else
   call mpi_send( a1, n, MPI_REAL8, 0, 200, MPI_COMM_WORLD, ierr )
   call mpi_recv( a0, n, MPI_REAL8, 0, 100, MPI_COMM_WORLD, istat, ierr )
end if

call mpi_finalize( ierr )
end program deadlock
```

演習M3-3 デッドロックを確認せよ

- プログラム M-7をコピーし，以下のことを確認せよ.
  - /tmp/mpi2/M-7/deadlock.f90

- プログラム5行目の nを，10， 100としたときに，結果がどうなるか確認せよ．プロセス数は2として実行する.
  - 注意） ジョブスクリプトの #PJM –L “elapse=00:00:xx” の xx は大きくしない.

- M-7において，send, recvの順番を次のように変えて実行し，結果がどなるか確認せよ.

【変更１】

```
    if( myrank == 0 ) then
       call mpi_recv( )
       call mpi_send( )
    else
       call mpi_recv( )
       call mpi_send( )
    end if
```

【変更２】

```
    if( myrank == 0 ) then
       call mpi_send( )
       call mpi_recv( )
    else
       call mpi_recv( )
       call mpi_send( )
    end if
```

実行結果は．．．

- 次のシステム・メッセージが出るケースがある．

```
jwe0017i-u The program was terminated with signal number SIGXCPU.
```

⇒ CPUの時間制限を越えた．
⇒ ジョブが指定した時間内に終わらなかった．

- ジョブが終了するケースと，そうでないケースがある．

→ 何故か？

ブロッキング関数とデッドロック

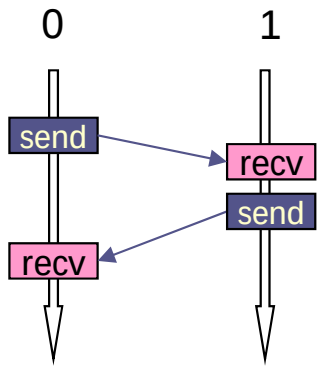
- mpi_send()，mpi_recv() はブロッキング関数
- ブロッキング関数の動作（実装による）
  - 送信／受信側のバッファ領域にメッセージが格納され，受信／送信側のバッファ領域が自由にアクセス（上書き）できるまで，呼出し元に制御が戻らない．
    - mpi_sendの場合，すべてのメッセージがMPI送信バッファに書き込みが終わった段階で，呼出し元に制御が戻る場合もある（後は，下位レイヤの通信プログラムに制御を任せてしまう）．
    - mpi_recvは，すべてのメッセージを受信するまで，呼出し元に制御が戻らない．
  - 次の行に制御が移らない．
- ブロッキング関数は，その関数の処理が終了するまで，次の処理に進まない．

演習M3-3の解説

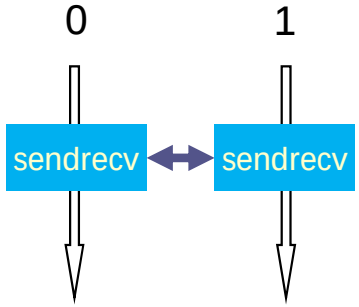
- ケース１：send-recv：send-recv かつ n=10
  - mpi_sendで送るメッセージのバイト数が小さいため，システムのバッファにすべて書き込めたので，制御が戻り，次の行が実行された，と考えられる.
  - mpiライブラリの実装に依る.
- ケース２： send-recv：send-recv かつ n=100
  - mpi_sendで送るメッセージのバイト数が大きく，すべてのメッセージがMPI通信バッファに書き込めず，相手のrecvの開始を待っているが，相手もmpi_sendを実行していて，受取ってくれないので，deadlockとなった.
- ケース３： recv-send: recv-send
  - どちらのプロセスもmpi_recv関数を実行し，データの到着を待っているが，お互いmpi_sendが実行できないので，そこで待っている間にCPUの制限時間に達した.
- ケース４： send-recv: recv-send
  - 送受信の順番が，シリアライズされたため，上手く実行できた.

デッドロックの回避方法

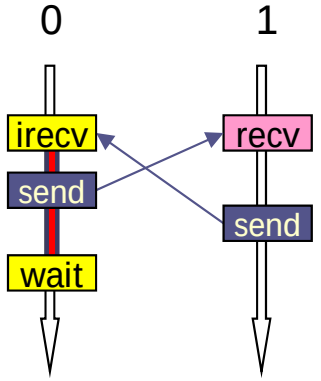
① 送受信の順序のシリアライズ（ケース４）
- プロセス0： 送信してから受信
- プロセス1： 受信してから送信



② mpi_sendrecv の利用
- mpi_send と mpi_recv をまとめて行うルーチン
- デッドロックは生じない
- 1回の送受信の時間で済む
- 送信相手と受信相手が異なってもよい



③ ノンブロッキング関数の利用
- mpi_isend
- mpi_irecv
- ノンブロッキング関数では，制御が呼出し元にすぐに戻るので，転送する変数に関係ない他の作業をすることが出来る.
  - 特に，通信と計算が同時に動作する
- mpi_waitで，関数の終了を確認する必要がある.



双方向通信：mpi_sendrecv関数

```
mpi_sendrecv( sendbuff, sendcount, sendtype, dest, sendtag,
              recvbuff, recvcount, recvtype, source, recvtag,
              comm, status, ierr )
```

- sendbuff:　送信するデータのための変数名（先頭アドレス）
- sendcount:　送信するデータの数（整数型）
- sendtype:　送信するデータの型（MPI_REAL，MPI_INTEGERなど）
- dest:　送信する相手プロセスのランク番号
- sendtag :　メッセージ識別番号．送くるデータを区別するための番号
- recvbuff:　受信するデータのための変数名（先頭アドレス）
- recvcount:　受信するデータの数（整数型）
- recvtype:　受信するデータの型（MPI_REAL，MPI_INTEGERなど）
- source:　送信してくる相手プロセスのランク番号
- recvtag:　メッセージ識別番号．送られて来たデータを区別するための番号
- comm:　コミュニケータ（例えば，MPI_COMM_WORLD）
- status:　受信の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）

ノンブロッキング送信関数 mpi_isend

```
mpi_isend( buff, count, datatype, dest, tag, comm, request, ierr )
```

※ ランク番号destのプロセスに，変数buffの値を送信する．

- buff:　　送信するデータの変数名（先頭アドレス）
- count:　　送信するデータの数（整数型）
- datatype:　送信するデータの型
  - MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISIONなど
- dest:　　送信先プロセスのランク番号
- tag:　　メッセージ識別番号．送るデータを区別するための番号
- comm:　　コミュニケータ（例えば，MPI_COMM_WORLD）
- request:　リクエスト識別番号（整数型）
- ierr:　　戻りコード（整数型）

ノンブロッキング受信関数 mpi_irecv

```
mpi_irecv( buff, count, datatype, source, tag, comm, request, ierr )
```

※ ランク番号sourceのプロセスから送られたデータを，変数buffに格納する．

- buff:　　受信するデータのための変数名（先頭アドレス）
- count:　　受信するデータの数（整数型）
- datatype:　受信するデータの型
  - MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISIONなど
- source:　　送信してくる相手プロセスのランク番号
- tag:　　メッセージ識別番号．送られて来たデータを区別するための番号
- comm:　　コミュニケータ（例えば，MPI_COMM_WORLD）
- request:　リクエスト識別変数（整数型）
- ierr:　　戻りコード（整数型）

待ちの関数 mpi_wait

```
mpi_wait( request, status, ierr )
```

※ リクエスト識別変数requestに対応した通信関数の終了を確認する．ブロッキング関数

- request:　リクエスト識別変数（整数型）
  - 対応するmpi_isend，またはmpi_irecvのリクエスト識別番号と一致させる
- status:　受信の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）

演習M3-4, M3-5（提出課題）

- プログラム M-7を，次の２つの方法で，deadlockしないプログラムにせよ.
  - mpi_irecv, mpi_waitを使う. (M3-4)
    - 16ページの③のとおり.
  - mpi_sendrecvを使う. (M3-5)
    - プロセス0，プロセス1は，それぞれ送る変数が違うことに注意.
- データがきちんと転送されていることを確認すること.

課題の提出方法と提出期限

- 演習M3-2，M3-4，M3-5の提出方法
  ① それぞれプログラムと実行結果を一つのファイルにまとめる.

```
$ cat pppppp.f90 > report-xx.txt
$ cat xxxxx.onnnnn >> report-xx.txt
```

  ② 以下の方法で，メールにより提出する.

```
$ cat report-xx.txt | mail -s “3-n:アカウント” yokokawa@port.kobe-u.ac.jp
```

Note)アカウントは自分のログインID
番号（3-n）は，演習課題番号

- 期限：7月13日（水） 午後5時