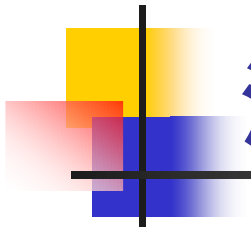
RIKEN AICS Summer School
演習3・4「MPIによる並列計算」

2013年8月7日

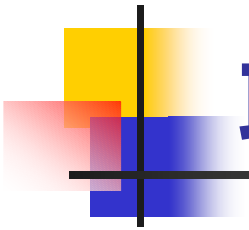
神戸大学 大学院システム情報学研究科　山本有作
理化学研究所 計算科学研究機構　下坂健則



演習の目標

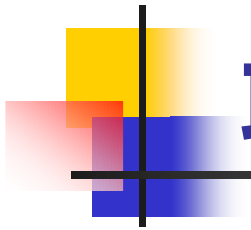
- 講義6「並列アルゴリズム基礎」で学んだアルゴリズムのいくつかを，MPI を用いて並列化してみる

- これを通じて，基本的な並列化手法と，MPI 通信関数の使い方を身に付ける



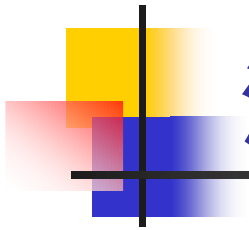
取り上げる例題と学習項目

- **行列ベクトル積**
  - **ベクトルに対するリダクション演算**
  - **グローバルインデックスとローカルインデックス**
  - allocatable **配列の利用**
- **行列どうしの積**
  - **2次元分割**
  - Fox **のアルゴリズム**
  - **コミュニケータの分割**
  - **部分ブロードキャスト**
  - mpi_sendrecv



取り上げる例題と学習項目（続き）

- **3次元FFTにおける通信部分**
  - mpi_alltoall
- **熱伝導方程式の数値解法**
  - MPI_PROC_NULL

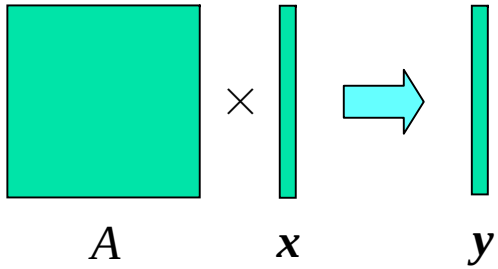


演習で使うプログラムとシェル

- FX10 **の以下のディレクトリからコピーすること**
  - /home/cs/yamamoto/ss130807/

課題1：行列ベクトル積

- 問題
  - A を第 (i, j) 要素が i+j の行列, x を第 i 要素が i のベクトルとする
  - このとき, $y = Ax$ を計算したい



- ループの構造
  - ベクトル y の要素を1個ずつ計算する
  - y の第 i 要素は A の第 i 行とベクトル x との内積

```
do i=1, n
  y(i)=zero
  do j=1, n
    y(i)=y(i)+a(i,j)*x(j)
  end do
end do
```

逐次版プログラム (mv.f90)

```
program mv
  implicit none
  integer, parameter :: n=100
  integer :: i,j
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(n,n) ::  a
  real(DP), dimension(n) :: x,y
  real(DP) :: ans,err
  real(DP), parameter :: zero=0.0
  do i=1, n                                  ┐
    x(i)=i                                   │
  end do                                     │
  do i=1, n                                  │  A, x の設定
    do j=1, n                                │
      a(i,j)=i+j                             │
    end do                                   │
  end do                                     ┘
  do i=1, n                                  ┐
    y(i)=zero                                │
    do j=1, n                                │  y = Ax の計算
      y(i)=y(i)+a(i,j)*x(j)                  │
    end do                                   │
  end do                                     ┘
  err=0.0d0                                  ┐
  do i=1, n                                  │
    ans=dble(i*n*(n+1)/2+n*(n+1)*(2*n+1)/6)  │  結果の確認
    err=err+abs(y(i)-ans)                    │
  end do                                     │
  print *, 'error =', err                    ┘
end program mv
```

演習1-1

- mv.f90 をコンパイルして実行せよ
  - cp /home/cs/yamamoto/ss130807/mv.f90 .
  - cp /home/cs/yamamoto/ss130807/sample1.sh .
  - frtpx mv.f90　　逐次プログラムのコンパイル (FORTRAN)
  - pjsub sample1.sh　　逐次プログラムの実行 (ジョブ投入)

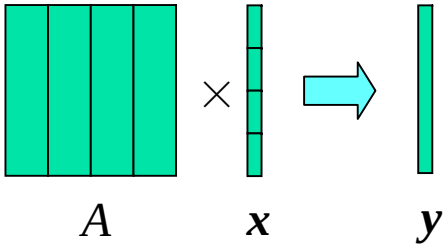
C言語版のコンパイルは　fccpx -O1 -Klib mv.c　とする

- 結果が正しいことを確認せよ
  - more sample1.sh.oxxxx

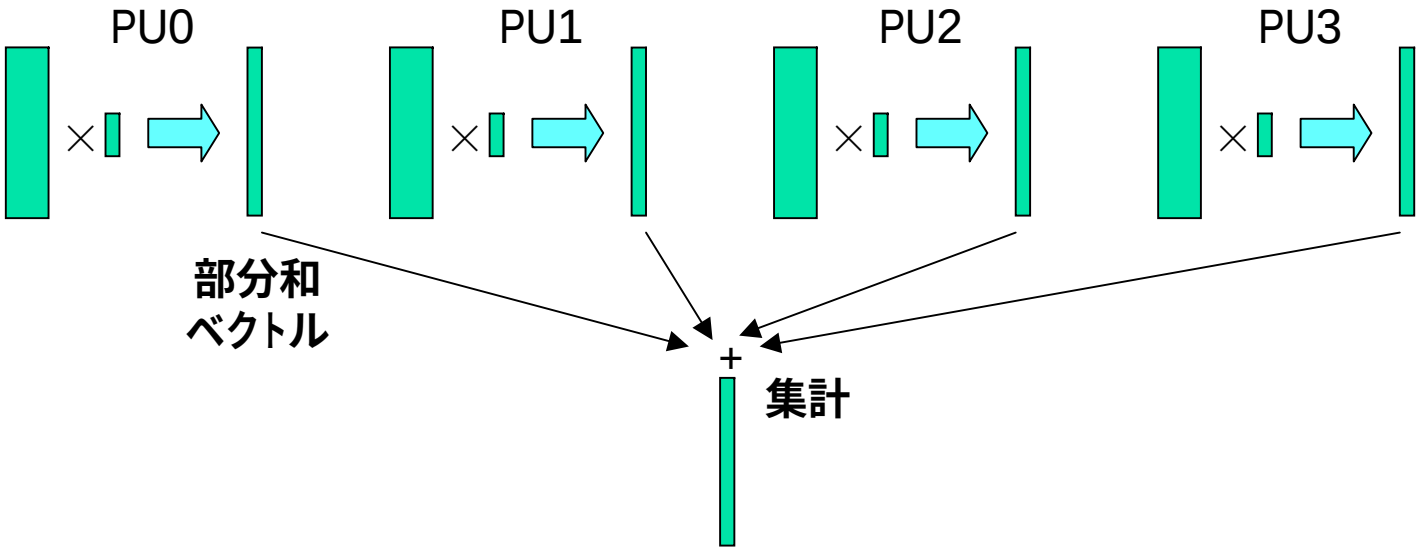
    ⇨ error = 0.0000000000000000+00

行列ベクトル積の並列化

- **データ分割**
  - 右図のように，A はブロック列分割，
    x はブロック分割されているとする



- **計算方法**
  - まず，各PUが自分の持つ A, x の要素のみを使い，部分和ベクトルを計算
  - 部分和ベクトルを mpi_reduce でPU0に集計することにより，y を計算



mpi_reduce

- **機能**
  - **全プロセスの持つデータに対してリダクション演算を行い，結果を** root**に送る**
  - **ブロッキング通信**
  - **使用方法**

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

sendbuff        : 送信バッファの先頭アドレス
recvbuff        : 受信バッファの先頭アドレス (rootでのみ使用)
count           : 送信するデータの要素数
datatype        : 送信するデータの型
op              : リダクション演算の種類
root            : リダクション演算の結果が送られるプロセスのランク
comm            : コミュニケータ
ierr            : エラーコード (出力)
```

mpi_reduce（続き）

- **リダクション演算**
  - **加算，乗算，最大値のように，複数のデータを入力として1個の出力データを求める演算**
  - **演算の種類**：mpi_sum，mpi_prod, mpi_max, mpi_min **など**

- **ベクトルに対するリダクション演算も可能**
  - **ベクトルの各要素に対してリダクション演算を行い，その結果を要素とするベクトルを生成**
  - x_1，x_2，…，x_m **をそれぞれ長さ** n **のベクトルとするとき，それらの和** $x = x_1 + x_2 + \ldots + x_m$ **を求める計算など**
  - **引数** count **に，ベクトルの長さ** n **を入れればよい**

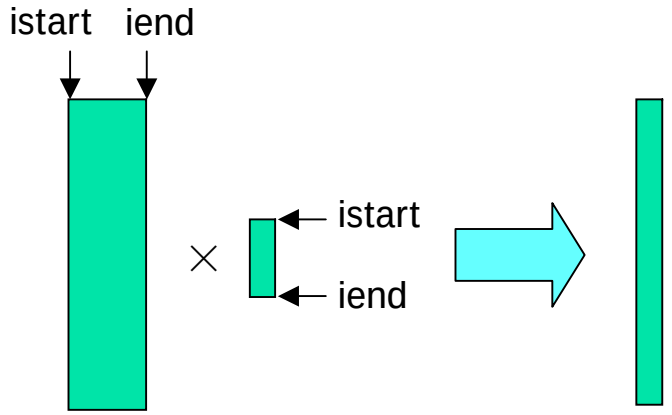
Mpi_Reduce (C言語版)

- **機能**
  - **全プロセスの持つデータに対してリダクション演算を行い，結果を** root**に送る**
  - **ブロッキング通信**
  - **使用方法**

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

sendbuff        : 送信バッファの先頭アドレス
recvbuff        : 受信バッファの先頭アドレス(rootでのみ使用)
count           : 送信するデータの要素数
datatype        : 送信するデータの型
op              : リダクション演算の種類
root            : リダクション演算の結果が送られるプロセスのランク
comm            : コミュニケータ
```

演習1-2

- mv.f90 を並列化せよ



- 書き換えのポイント
  - MPI 関連の定義，初期化，終了処理
  - 各プロセスの計算範囲の設定
    - istart=n*myrank/nprocs+1
    - iend=n*(myrank+1)/nprocs
  - A, x について，自プロセスが担当する部分のみを初期化
    - A：第 istart 列 ～ 第 iend 列
    - x：第 istart 要素 ～ 第 iend 列
  - 計算ループにおいて，自プロセスの持つ要素のみを使って計算
    - j=istart, iend とする
    - 結果の部分和ベクトルを y でなく配列 yp に入れる
  - 部分和の合計
    - mpi_reduce で配列 yp を合計し，PU0 の配列 y に入れる
    - mpi_reduce の第3変数 count は n とする

MPI プログラムのコンパイルと実行

- **コンパイル**
  - mpifrtpx xxx.f90　　FORTRAN
  - mpifccpx -O1 -Klib xxx.c　　C
- **実行**
  - /home/cs/yamamoto/ss130807/sample_mpi.sh **をコピーして使用**
    - **昨日の演習で使ったシェルスクリプトを用いてもよい**
  - 8**行目の** “-n 4” **の部分でプロセス数を指定**
  - **実行は** pjsub sample_mpi.sh

部分配列とローカルインデックス

- **部分配列の利用**
  - **前ページのプログラムでは，各プロセスが A, x 全体を格納できる配列を確保し，そのうち自分の担当部分のみに値を入れて使用**
  - **実際に使用する範囲のみを確保すれば，メモリを節約できる**
    - A: **第** istart **列 ～ 第** iend **列**
    - x: **第** istart **要素 ～ 第** iend **要素**
  - **これを実現するには，**allocatable **配列を利用すればよい**

- **ローカルインデックス**
  - allocate **文により，**x**のインデックスが** istart **から始まるようにできる**
  - **これにより，プログラムをほとんど変えずに部分配列を利用可能**
  - **サイクリック分割等の場合は，やや複雑なインデックス変換が必要**

部分配列を用いたプログラム

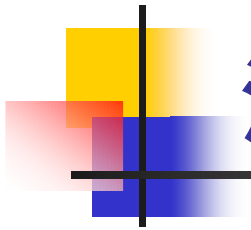
```
program mv_reduce2
  use mpi
  implicit none
  integer, parameter :: n=100
  integer :: i,j,istart,iend
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(:,:), allocatable :: a
  real(DP), dimension(:), allocatable :: x

  (他の配列の確保, MPIの初期設定, istart, iendの設定など)

  allocate(a(n,istart:iend))
  allocate(x(istart:iend))

  (A, x の初期設定)
  (自プロセスの持つ要素のみを使った内積の部分和計算)
  (mpi_reduceにより結果ベクトル y を計算)
  (プロセス0で結果のチェック)

  deallocate(a)
  deallocate(x)
  call mpi_finalize(ierr)
end program mv_reduce2
```

A, x を不定サイズの配列として定義

A, x の領域を確保
A の列番号, x の要素番号が istart から始まるようにする

この部分は変更不要

A, x を解放



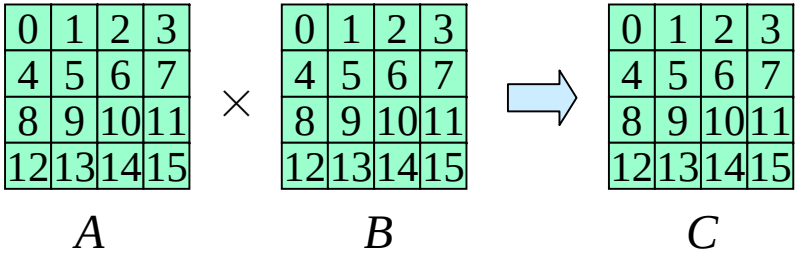
演習1-3

- 演習1-2で作ったプログラムを，部分配列を使うように書き換えよ。
- mpi_wtime を使い，行列ベクトル積の計算部分（内積の部分和の計算と mpi_reduce）の時間測定ができるようにせよ
  - 時間測定のしかたは昨日の演習のスライド参照
- n=1000 とし，プロセス数を変えて実行時間の変化を調べよ

課題2：行列どうしの積

- **問題設定**
  - $n \times n$ の行列 A と B をかけ，行列 C を求める
  - プロセス数 P は平方数で，n は $\sqrt{P}$ で割り切れるとする

- **データ分割**
  - A, B が行・列の**両方向**に**ブロック分割されているとする**
  - このとき，C も同じ分割形式で**求めるとする**



- **アルゴリズム**
  - Fox のアルゴリズムを用いる

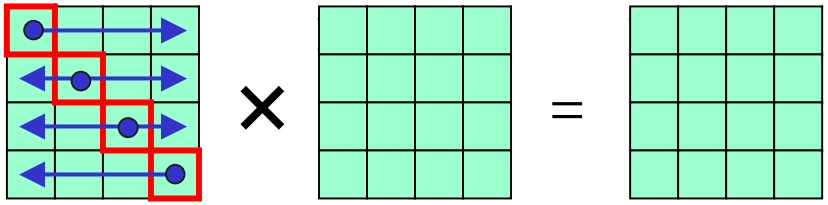
Fox のアルゴリズム (復習)

- **アルゴリズム**

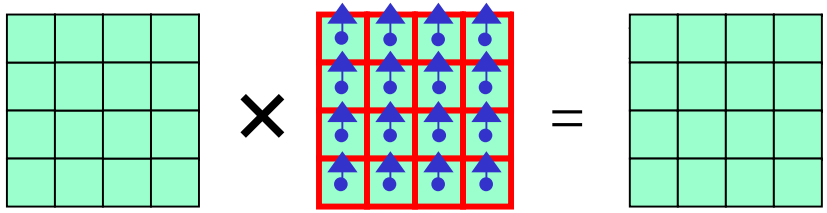
```
program fox
(MPIの初期化と環境情報取得)
(配列の確保: A,B,C,A_TMP,B_TMP)          自分の担当する部分行列を入れる配列
(自分の行番号, 列番号をそれぞれI_num,J_numとする)    0 ≦ I_num, J_num < √P
(配列A,Bにデータを入れる。配列Cを0に初期化)
do K = 1, √P
   IF (J_num = MOD(I_num+K-1,√P)) (配列AをA_TMPにコピー)
   (配列A_TMPを行方向のプロセス間でブロードキャスト)
   IF (K > 1) THEN
      (同じ列の1個上のプロセスにB_TMPを送る)
      (同じ列の1個下のプロセスからBにデータを受け取る)
   END IF
   (Bを配列B_TMPにコピー)
   C += A_TMP×B_TMP
end do
(配列Cの出力)
(MPIの終了)
stop
end
```

Fox のアルゴリズムの通信パターン

- A_{TMP} の横方向ブロードキャスト
  - 同じI_{num}を持つプロセスに対してコミュニケータ (プロセスのグループ) を定義
  - コミュニケータを用い，mpi_bcast により部分ブロードキャストを行う

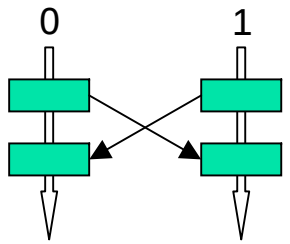


- B_{TMP} の縦方向への巡回型通信
  - 1つ上の PU へのデータ送信と，1つ下の PU からのデータ受信とを同時に行う



同時送受信

- 2プロセスの場合
  - プロセス0はプロセス1にベクトル a0 を送る
  - プロセス1はプロセス0にベクトル a1 を送る



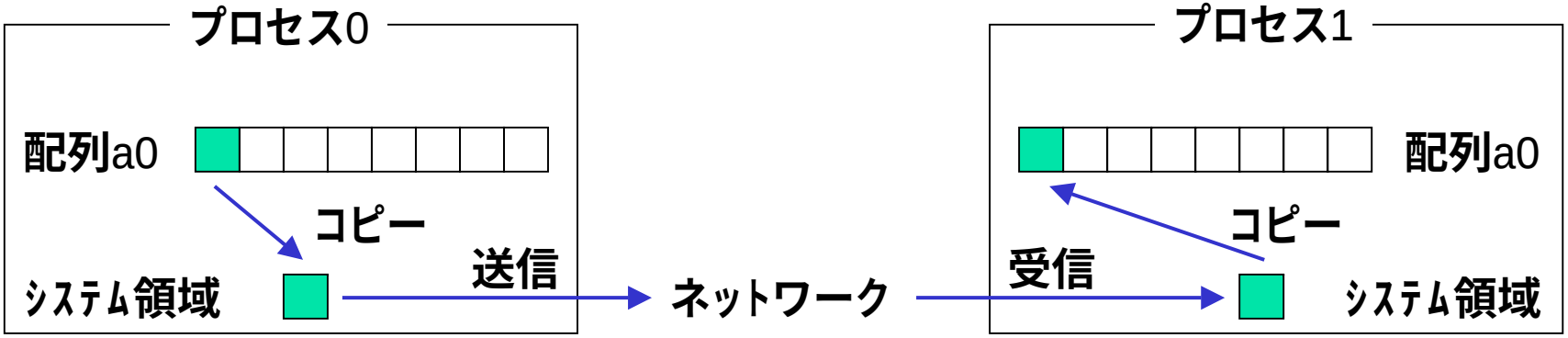

```
program main
parameter(n=1000000)
double precision a0(n),a1(n)
                    ⋮                    宣言および初期化
if (myrank.eq.0) then
   call mpi_send(a0,n,MPI_DOUBLE,1,100,MPI_COMM_WORLD,ierr)
   call mpi_recv(a1,n,MPI_DOUBLE,1,200,MPI_COMM_WORLD,istat,ierr)
else
   call mpi_send(a1,n,MPI_DOUBLE,0,200,MPI_COMM_WORLD,ierr)
   call mpi_recv(a0,n,MPI_DOUBLE,0,100,MPI_COMM_WORLD,istat,ierr)
end if
                    ⋮                    a0, a1を使った処理
stop
end
```

- このプログラムは正しく動くか？

同時送受信 (続き)

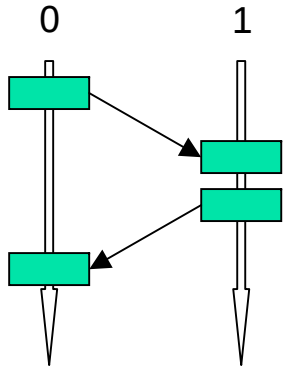
- 実はデッドロックの可能性あり
  - プロセス 0 は, a0 を一部分ずつシステム領域にコピーしてから送信
  - システム領域中のデータが送信され, 相手に受信されるまでは, 次の部分を送信できず, 待機状態となる
  - ところが, 相手も先に a1 の送信を行おうとするため, 同じ理由で待機状態となる

⇒ デッドロックが発生

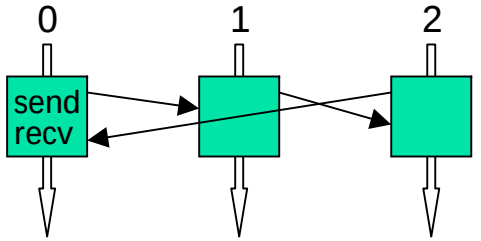


デッドロックの回避法

- **送受信の順序の変更**
  - **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**
  - **問題点： 時間が2倍かかってしまう**



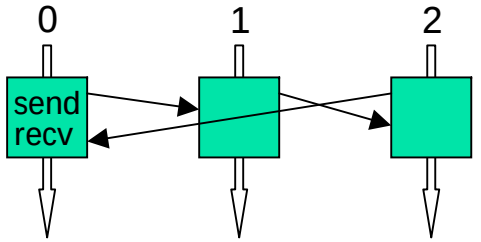
- mpi_sendrecv **の利用**
  - mpi_send **と** mpi_recv **をまとめて行うルーチン**
  - **デッドロックは生じない**
  - **1回の送受信の時間で済む**
  - **送信相手と受信相手が異なってもよい**
  - **使用方法**




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

MPI_Sendrecv（C言語版）

- **機能**
  - mpi_send と mpi_recv をまとめて行う
  - 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
  - 使用方法




```
int Mpi_Sendrecv(void *sendbuff, int sendcount,
                 MPI_Datatype sendtype, int dest, int sendtag,
                 void *recvbuff, int recvcount,
                 MPI_Datatype recvtype, int source,
                 int recvtag, MPI_Comm comm,
                 MPI_Status status)
```

Fox のアルゴリズムでの巡回型通信

- K>1 のときのみ，mpi_sendrecv により以下の送受信を行う

- **送信**
  - **送信すべきデータ：配列** B_{TMP}
  - **送り先：同じ列の1個上のプロセス**
    - **すなわち，自分よりランク番号が$\sqrt{P}$だけ小さいプロセス**
    - **ただし，一番上のプロセスは一番下のプロセスに送信**

- **受信**
  - **受信データの格納先：配列** B
  - **送り元：同じ列の1個下のプロセス**
    - **すなわち，自分よりランク番号が$\sqrt{P}$だけ大きいプロセス**
    - **ただし，一番下のプロセスは一番上のプロセスから受信**

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

B

部分ブロードキャスト

- コミュニケータの分割
  - 各プロセスが，それぞれカラー番号と呼ばれる整数型変数 icolor を持っているとする
  - カラー番号により，プロセスを複数のサブグループに分けたいとする
  - このとき，自分が属するサブグループのコミュニケータを求める関数が mpi_comm_split
  - 各プロセスにはサブグループ内でのランク番号が付く
    - ランク番号は，引数 key が小さい順に，0，1，2，… となる
    - mpi_comm_split では，このランク番号自体は出力されない

```
call mpi_comm_split(old_comm,icolor,key,new_comm,ierr)

old_comm        : 元のコミュニケータ
icolor          : カラー番号
key             : サブグループにおける自プロセスのランクを決めるための変数
new_comm        : 自分が属するサブグループのコミュニケータ
                  (サブグループごとに違うコミュニケータが出力される)
ierr            : エラーコード (出力)
```

MPI_Comm_split (C言語版)

- **機能**
  - **プロセスを，カラー番号により，複数のサブグループに分割する**

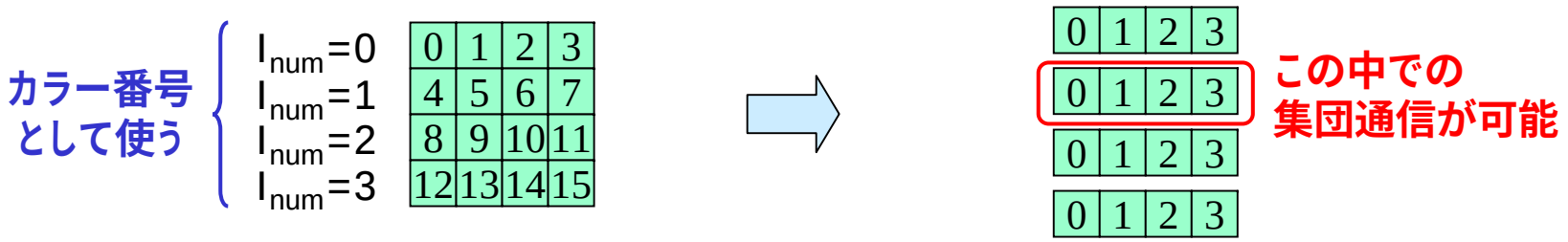
```
Int Mpi_Comm_split(MPI_Comm old_comm, int icolor, int key,
                   MPI_Comm new_comm)

old_comm        : 元のコミュニケータ
icolor          : カラー番号
key             : サブグループにおける自プロセスのランクを決めるための変数
new_comm        : 自分が属するサブグループのコミュニケータ
                  (サブグループごとに違うコミュニケータが出力される)
```

コミュニケータの分割の例

- **同じ行番号を持つプロセスグループへの分割**
  - **元のコミュニケータ**：mpi_comm_world
  - **カラー番号**：**行番号** I_{num}
  - key：myrank

```
call mpi_comm_split(mpi_comm_world,inum,myrank,new_comm,ierr)
```

  - **サブグループ内での自プロセスのランクは列番号** J_{num} **となる**
  - new_comm と J_{num} **とを使うことで，行内での集団通信が可能**
    - mpi_comm_world と myrank を new_comm と J_{num} **で置き換えるだけ**



元のコミュニケータ mpi_comm_world **内でのランク**

新しいコミュニケータ new_comm **内でのランク**

Fox のアルゴリズムによる行列乗算 (fox.f90)

```
program fox
  use mpi
  implicit none
  integer, parameter :: n=100
  integer :: i,j,k,kk,myrank,nprocs,nblock,nblock2,npsqrt
  integer :: inum,jnum,newcomm,idst,isrc,itag,ierr,ig,jg

  (配列の宣言, MPIの初期化など)

  dnprocs=nprocs
  npsqrt=sqrt(dnprocs)
  nblock=n/npsqrt

  nblock2=nblock*nblock

  inum=myrank/npsqrt
  jnum=mod(myrank,npsqrt)

  コミュニケータを分割し, 同一行に属するプロセスグループのコミュニケータnew_commを作成

  (配列a, b, cのアロケーション, 初期化など)

  idst=mod(myrank-npsqrt+nprocs,nprocs)
  isrc=mod(myrank+npsqrt,nprocs)
```

プロセス数の平方根npsqrtと, プロセス当たりのブロックサイズnblockを計算

自プロセスの行番号inum, 列番号jnumを計算

mpi_sendrecvのためのidst, isrcの設定

Fox のアルゴリズムによる行列乗算

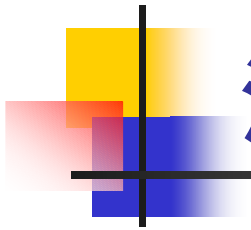
```
  do kk=1,npsqrt
     if (jnum == mod(inum+kk-1,npsqrt)) atmp(:,:) = a(:,:)
      iroot = mod(inum+kk-1,npsqrt)

         (new_commを使い，配列ATMPを行方向のプロセス間でブロードキャスト)

     if (kk > 1) then

         (mpi_sendrecvを使い，配列BTMPを縦方向に巡回型通信)

     end if

     btmp(:,:) = b(:,:)

     do j=1,nblock
        do i=1,nblock
           do k=1,nblock
              c(i,j)=c(i,j)+atmp(i,k)*btmp(k,j)
           end do
        end do
     end do

  end do

  (結果のチェック)
  (配列の解放，MPIの終了処理)

end program fox
```

配列AをATMPにコピー

行方向ブロードキャストのための送り元プロセスを設定

配列BをBTMPにコピー

C += ATMP×BTMP

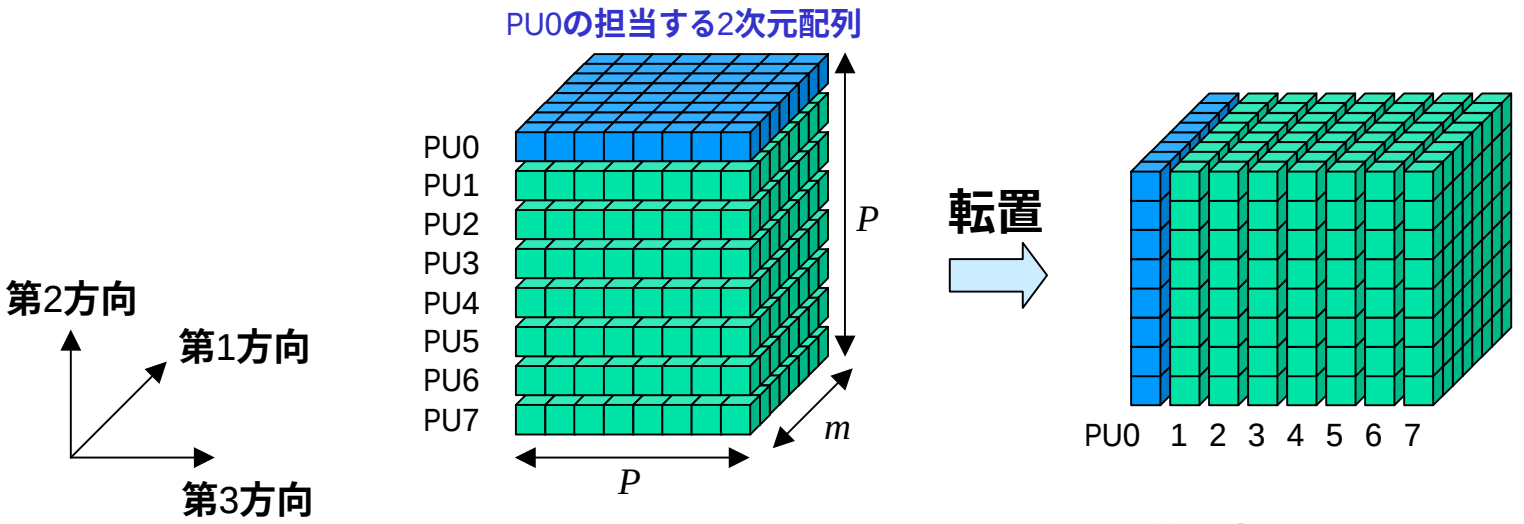


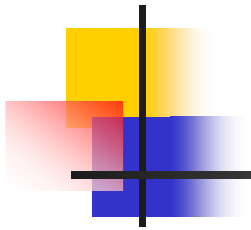
演習2-1

- Fox のアルゴリズムのプログラムを，赤字の部分を補って完成させよ
- 4 プロセス，16 プロセスで実行し，結果が正しいことを確かめよ。error が 0 となり，result OK というメッセージが出ればよい
- 余裕があれば，n=1200 とし，プロセス数を 4，9，16 と変えて実行して計算時間の変化を調べよ。また，加速率を求めよ

課題3：3次元FFTにおける通信部分

- **問題設定**
  - u を大きさ $m \times P \times P$ の3次元配列とする
  - P 個のプロセスがあり，u の第2方向が分割されているとする
    - すなわち，各プロセスは $m \times P$ の2次元配列を持つ
  - このとき，u をプロセス間で**転置**し，第3方向がプロセス間に分割されるようにしたい





mpi_alltoall

- **機能**
  - **各プロセスは，**sendbuf **の先頭から** sendcount **個ずつ，それぞれプロセス** 0, 1, …, nprocs-1 **に向けてデータを送信する**
  - **同時に，プロセス** 0, 1, …, nprocs-1 **から** recvcount **個ずつ受け取ったデータを，**recvbuf **の先頭から順番に格納する**
  - **使用方法**

```
call mpi_alltoall(sendbuff,sendcount,sendtype,recvbuf,
                  recvcount,recvtype,comm,ierr)

sendbuff        : 送信バッファの先頭アドレス
sendcount       : 送信するデータの要素数
sendtype        : 送信するデータの型
recvbuff        : 受信バッファの先頭アドレス
recvcount       : 受信するデータの要素数
recvtype        : 受信するデータの型
comm            : コミュニケータ
ierr            : エラーコード(出力)
```

Mpi_Alltoall (C言語版)

- **機能**
  - **各プロセスは，**sendbuf **の先頭から** sendcount **個ずつ，それぞれプロセス** 0, 1, …, nprocs-1 **に向けてデータを送信する**
  - **同時に，プロセス** 0, 1, …, nprocs-1 **から** recvcount **個ずつ受け取ったデータを，**recvbuf **の先頭から順番に格納する**
  - **使用方法**

```
int Mpi_Alltoall(void *sendbuff, int sendcount,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)

sendbuff        : 送信バッファの先頭アドレス
sendcount       : 送信するデータの要素数
sendtype        : 送信するデータの型
recvbuff        : 受信バッファの先頭アドレス
recvcount       : 受信するデータの要素数
recvtype        : 受信するデータの型
comm            : コミュニケータ
```

mpi_alltoall を用いた配列の転置 (alltoall.f90)

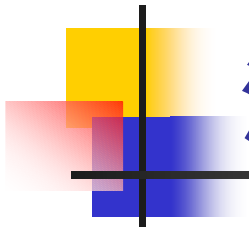
```
program alltoall
  use mpi
  implicit none
  integer, parameter :: m=99
  integer :: j,k
  real(DP), dimension(:,:), allocatable :: u,un

  (MPIの初期化など)

  allocate(u(m,nprocs))
  allocate(un(m,nprocs))

   do k=1, nprocs
    do j=1, m
      u(j,k)=real(j+k-1,DP)
    end do
  end do

  mpi_alltoallを用いた配列の転置

  err=0.0_DP
  do k=1,nprocs
    do j=1,m
      err=err+(abs(un(j,k))-myrank-j)
    end do
  end do

  (チェック結果の出力, MPIの終了処理など)

end program alltoall
```

u: 元の配列, un:転置結果

配列u, unの確保

配列uにデータを入れる

転置結果のチェック



演習3-1

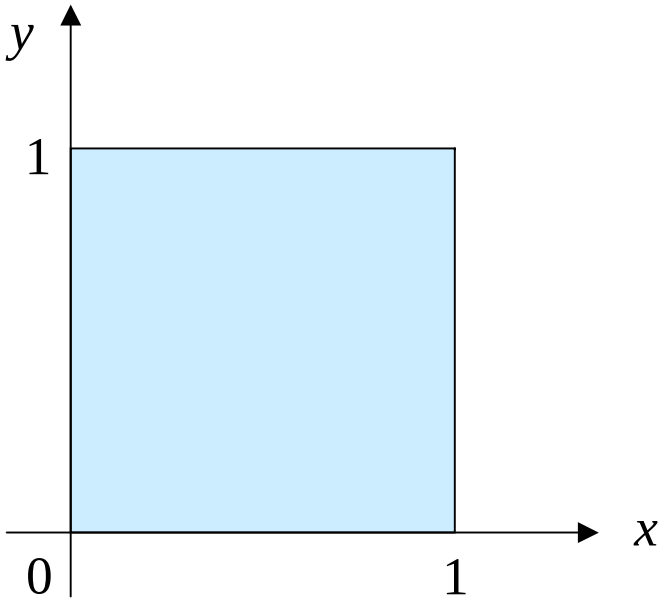
- 配列の転置のプログラムを，赤字の部分を補って完成させよ
  - 2次元配列 u(m,nprocs) は，1次元配列 u(m*nprocs)と見なせる
  - この配列の先頭から m 要素ずつを，プロセス 0, 1, …, nprocs-1 に送信すればよい
  - 同様に，2次元配列 un(m,nprocs) を1次元配列 un(m*nprocs)と見なして受信する

- 4 プロセス，16 プロセスで実行し，結果が正しいことを確かめよ。error が 0 となり，result OK というメッセージが出ればよい

課題4：熱伝導方程式の数値解法

- **問題**
  - **2次元正方形領域** $[0,1]\times[0,1]$ **での熱伝導を考える**
  - **境界をすべて0℃に固定**

$$u(0, y) = 0$$
$$u(1, y) = 0$$
$$u(x, 0) = 0$$
$$u(x, 1) = 0$$



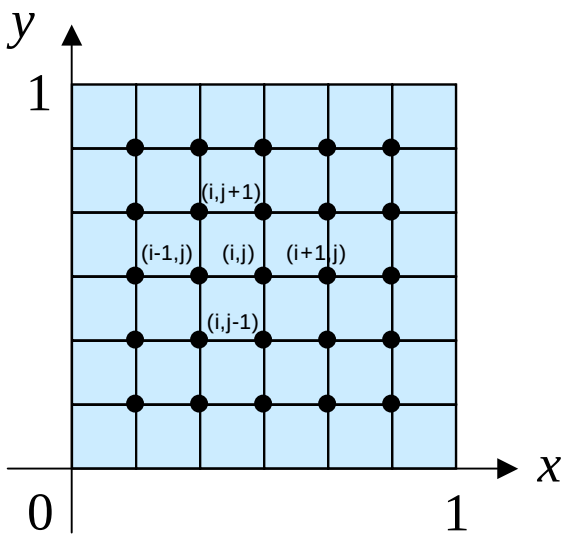
  - **領域全体に一定の熱を加える**

このとき，十分な時間が経った後での温度分布はどうなるか？

熱伝導方程式の数値解法（続き）

- **熱伝導方程式**
  - $\partial u / \partial t = \nabla^2 u + f$　（u: 温度, f: 熱源の効果）
- **離散化**
  - 領域内を格子に区切り，格子点上での温度のみを考える
  - さらに，離散的な時間ステップでの温度のみを考える
- **離散版の熱伝導方程式**
  - 時間ステップ n での格子点 (i, j) の温度を $u_{ij}^{(n)}$ とすると，

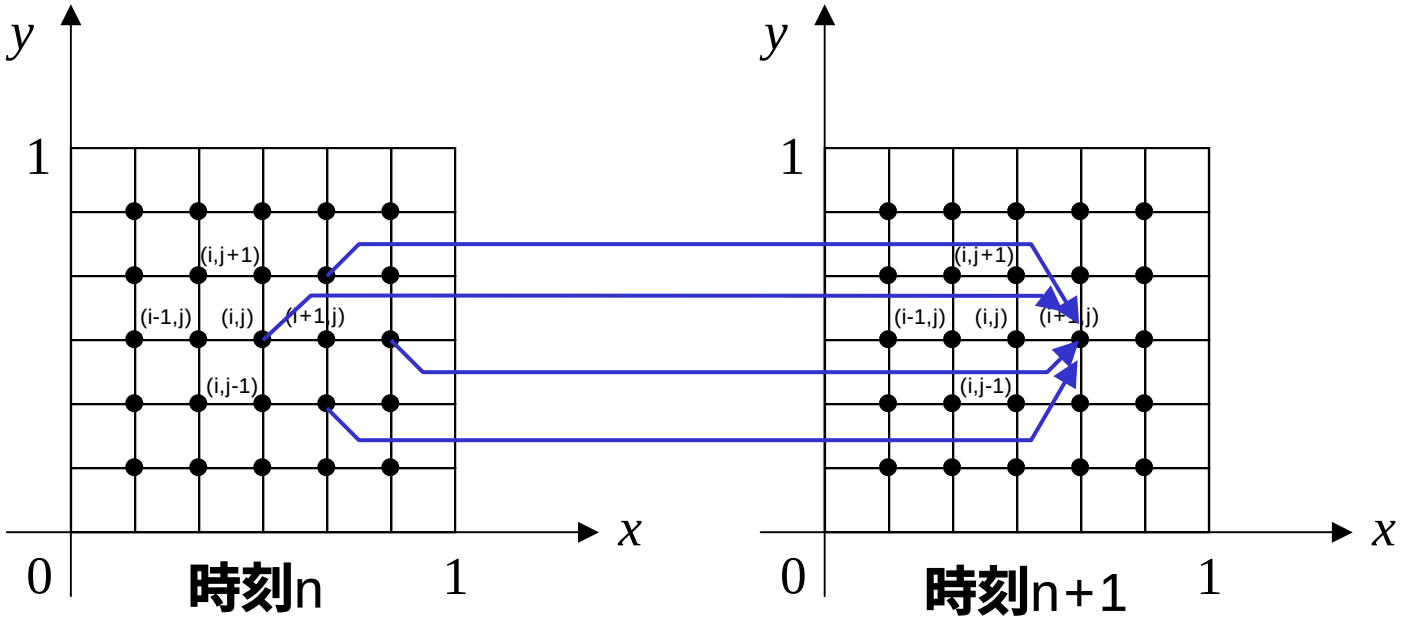
$$u_{ij}^{(n+1)} = (u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)}) / 4 + f_{ij}$$



熱伝導方程式の数値解法 (続き)

- **時間発展のアルゴリズム (ヤコビ法)**

```
do j=1, m
    do i=1, m
        u_ij^(n+1) = (u_i-1,j^(n) + u_i+1,j^(n) + u_i,j-1^(n) + u_i,j+1^(n) ) / 4 + f_ij
    end do
end do
```



逐次版のプログラム (heat1.f90)

```
program heat1
  implicit none
  integer, parameter :: m=49, nmax=20000
  integer :: i,j,n
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(:,:), allocatable :: u, un
  real(DP) :: h, heat=1.0_DP
  allocate(u(0:m+1,0:m+1))
  allocate(un(m,m))

  h=1.0_DP/(m+1)
  u=0.0_DP

  do n=1, nmax
    do j=1, m
      do i=1, m
        un(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))/4.0_DP+heat*h*h
      end do
    end do
    u(1:m,1:m) = un(1:m,1:m)
    if (mod(n,100)==0) print *, n, u((m+1)/2,(m+1)/2)
  end do
end program heat1
```

50×50の格子 (内点は49×49)

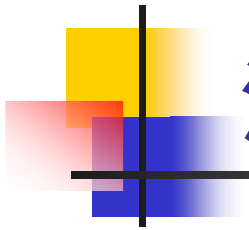
時間ステップ数20,000

u: 現在の時間ステップでの温度
(境界条件を考慮するため, 全方向に1だけ大きい配列)

un: 次の時間ステップでの温度

次の時間ステップでの温度を計算

un を新しい u とする



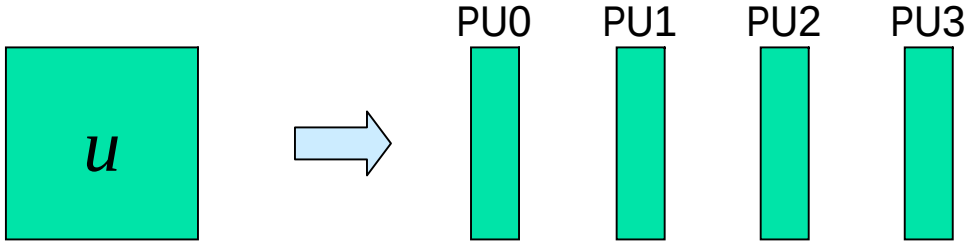
演習4-1

- heat1.f90 **をコンパイルし，実行せよ**

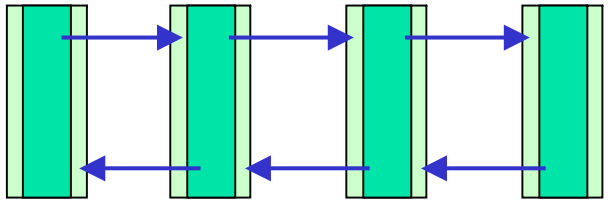
- **出力結果（点**((m+1)/2, (m+1)/2)**での**100**ステップおきの値）を調べ，それが一定値に収束していることを確認せよ**
  - **この結果は，後ほど並列プログラムのチェックに用いる**

並列化方法

- データ分割
  - ブロック列分割とする



- 計算方法
  - ある点の計算には，1ステップ前における隣の4点での値が必要
  - PU境界の点の計算のため，各ステップの最初に，両隣からデータを送信してもらう

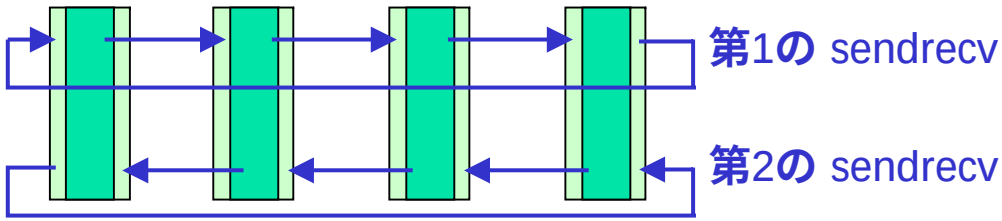


両隣のプロセスから1列を受信
(受信用の領域を確保しておく)

通信部を抜き出した並列化プログラム

- **配列の確保**
  - **自プロセスの担当範囲は** jstart ～ jend **列**
  - **受信領域を考慮し,** jstart-1 ～ jend+1 **列の領域を確保**

- mpi_sendrecv **による送受信**
  - **まず, 右隣に** jend **列を送り, 左隣から** jstart-1 **列を受信**
  - **次に, 左隣に** jstart **列を送り, 右隣から** jend+1 **列を受信**
  - **右端プロセスの右側との送受信, 左端プロセスの左側との送受信では, 相手先として** MPI_PROC_NULL **を指定**
    - **この場合, 通信は行われない**



プログラム例 (sendrecv.f90)

```
program sendrecv
  use mpi
  implicit none
  integer, parameter :: m=99
  integer :: i,j,jstart,jend
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), dimension(:,:), allocatable :: u
  real(DP) :: err
  integer ::  nprocs,myrank,ierr,left,right
  integer, dimension(MPI_STATUS_SIZE) :: istat
  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)

  jstart=m*myrank/nprocs+1
  jend=m*(myrank+1)/nprocs
  allocate(u(m,jstart-1:jend+1))

  do i=1, m
    do j=jstart, jend
      u(i,j)=real(i+j,DP)
    end do
  end do
```

各プロセスの担当する列の範囲を計算

jstart-1列～jend+1列の領域を確保

自分の担当する列に値を設定

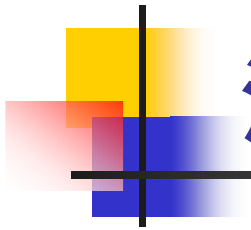
プログラム例 (続き)

```
  left=myrank-1
  if (myrank==0) left=MPI_PROC_NULL
  right=myrank+1
  if (myrank==nprocs-1) MPI_PROC_NULL
  call mpi_sendrecv(u(1,jend),m,MPI_DOUBLE_PRECISION,right,100, &
&                   u(1,jstart-1),m,MPI_DOUBLE_PRECISION,left,100, &
&                   MPI_COMM_WORLD,istat,ierr)
  call mpi_sendrecv(u(1,jstart),m,MPI_DOUBLE_PRECISION,left,100, &
&                   u(1,jend+1),m,MPI_DOUBLE_PRECISION,right,100, &
&                   MPI_COMM_WORLD,istat,ierr)

  err=0.0_DP
  if (myrank/=0) then
    do i=1, m
      err=err+abs(u(i,jstart-1)-real(i+mod(jstart+m-2,m)+1,DP))
    end do
  end if
  if (myrank/=nprocs-1) then
    do i=1, m
      err=err+abs(u(i,jend+1)-real(i+mod(jend,m)+1,DP))
    end do
  end if
  print *, 'myrank =', myrank, 'error =', err

  call mpi_finalize(ierr)
end program sendrecv
```

左右のプロセスのプロセス番号を計算
(両端のプロセスは MPI_PROC_NULL を指定)

mpi_sendrecv による送受信

正しく受信できたことを確認

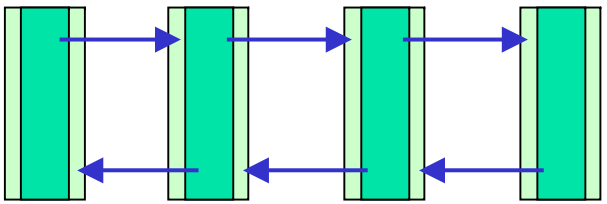


演習4-2

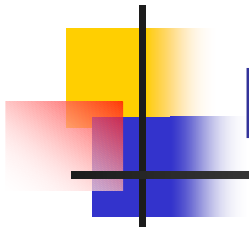
- sendrecv.f90 をコンパイルして 4 または 8 プロセスで実行し，データの送受信が正しくできていることを確かめよ
  - すべてのプロセスが error = 0.0 を出力すればよい

heat1.f90 の並列化

- 考え方
  - 2次元配列 u, un をブロック列分割
    - 配列 un は, jstart ～ jend 列の領域を確保
    - 配列 u は, 受信領域を考慮し, jstart-1 ～ jend+1 列の領域を確保
  - un の計算前に, 左のプロセスから u の jstart-1 列, 右のプロセスから u の jend+1 列を送ってもらう
    - sendrecv.f90 と同様にして, mpi_sendrecv を用いて送受信
  - un の jstart ～ jend 列の計算を行う

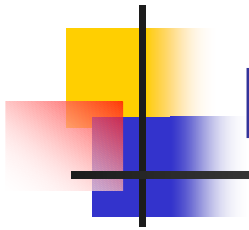


両隣のプロセスから1列を受信
(受信用の領域を確保しておく)



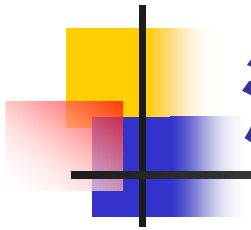
heat1.f90 の並列化 (続き)

- **書き換え I：初期化部分**
  - jstart, jend **の計算** (sendrecvf.90 **と同様**)
  - **配列の確保**
    - allocate(u(0:m+1,jstart-1:jend+1))
    - allocate(un(m,jstart:jend))
  - u **の** jstart ～ jend **列のみを**0**に初期化**
  - **左隣のプロセス番号** left, **右隣のプロセス番号** right **を定義**
    - **右端プロセスの右側との送受信, 左端プロセスの左側との送受信では, 相手先として** MPI_PROC_NULL **を指定**



heat1.f90 の並列化 (続き)

- 書き換え II：時間発展ループ内
  - 両隣のプロセスから，u の第 jstart-1 列，jend+1 列を受信
    - sendrecv.f90 と同様，mpi_sendrecv を2回繰り返す
  - プロセス 0 は，u の第 jstart-1 列を 0 に初期化
  - プロセス nprocs-1 は，u の第 jend+1 列を 0 に初期化
    - 領域左端，右端での境界条件を設定
  - jstart ～ jend 列のみについて，un を計算
  - jstart ～ jend 列のみについて，un を u にコピー
  - (m/2,m/2) 要素を担当するプロセスは，100ステップおきに値を出力
    - if (jstart <= m/2 .and. jend >= m/2) という条件を使う



演習4-3

- heat1.f90 を MPI を用いて並列化せよ
- 4 または 8 プロセスで実行し，heat1.f90 と出力結果が同じであることを確認せよ
- 余裕があれば，プロセス数を 1，2，4，8，16 と変えて実行し，計算時間の変化を調べよ。また，加速率を求めよ
- さらに余裕があれば，問題サイズ m を 100，200 と大きくして同様の実験を行い，加速率を調べよ

演習4-4 (ハイブリッド並列化)

- MPI 並列化を行ったプログラムにおいて，un を計算する i, j の2重ループを，OpenMP により並列化せよ

- FORTRAN でのコンパイルと実行
  - コンパイル用シェル：compile_heat2_hybridf.sh
    - ソースファイル名は heat2_hybrid.f90 とする
  - 実行用シェル：sample_heat2_hybridf.sh

- Cでのコンパイルと実行
  - コンパイル用シェル：compile_heat2_hybridc.sh
    - ソースファイル名は heat2_hybrid.c とする
  - 実行用シェル：sample_heat2_hybridc.sh

- MPI 並列化のみのプログラムと比較し，高速化率を求めよ