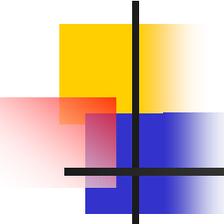


計算科学演習I 第9回講義
「MPIを用いた並列計算(II)」

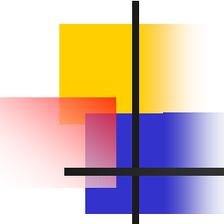
2010年6月24日

システム情報学研究科 計算科学専攻
山本有作



今回の講義の概要

1. 前回の宿題の解説
2. MPI プログラムの時間測定
3. 集団通信(続き)
4. 双方向通信とデッドロック
5. ノンブロッキング通信



演習1-3

- sum100.f90 を4プロセス用に書き換え, 実行せよ

- 書き換えのポイント

- 各プロセスが計算する部分 and の範囲を変更

```
istart=myrank*25+1  
iend=(myrank+1)*25
```

- プロセス0は, プロセス1, 2, 3のそれぞれから部分和を受信
 - mpi_recv を, 相手プロセスを変えて, 3回コール
 - 部分和を isum1 に1個受信するたびに, それを変数 isum に加える

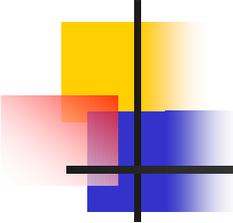
解答例 (/tmp/100617/sum100_4.f90)

```
program sum100_4
  use mpi
  implicit none
  integer :: i,istart,iend,isum,isum1,ip
  integer :: nprocs,myrank,ierr
  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)
  istart=myrank*25+1
  iend=(myrank+1)*25
  isum=0
  do i=istart, iend
    isum=isum+i
  end do
  if (myrank/=0) then
    call mpi_send(isum,1,MPI_INTEGER,0,100,MPI_COMM_WORLD,ierr)
  else
    do ip=1, 3
      call mpi_recv(isum1,1,MPI_INTEGER,ip,100,MPI_COMM_WORLD,istat,ierr)
      isum=isum+isum1
    end do
  end if
  if (myrank==0) print *, 'sum =', isum
  call mpi_finalize(ierr)
end program sum100_4
```

各プロセッサが計算する部分和の範囲を変更

プロセス1~3はそれぞれ部分和をプロセス0に送信

プロセス0は部分和をプロセス1~3から受信
受信した部分和を自分の部分和に足し込む

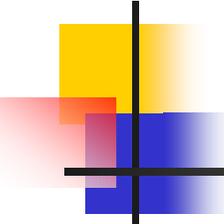


演習1-5

- `sumn.f90` を次のように書き換え, 実行せよ
 - 変数 `isum`, `isum1` を倍精度実数 `sum0`, `sum1` に変更せよ
 - それに伴い, `mpi_reduce` も倍精度で計算するようにせよ
- 書き換えのポイント
 - 倍精度実数型の定義(陰山先生の講義参照)

```
integer, parameter :: SP = kind(1.0)
integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
real(DP) :: sum, sum1
```

- `mpi_reduce` の変更
 - `datatype` を `MPI_DOUBLE_PRECISION` にする



解答例 (/tmp/100617/dsumn.f90)

```
program dsumn
  use mpi
  implicit none
  integer :: n,i,istart,iend,isum,isum1
  integer :: nprocs,myrank,ierr
  integer, dimension(MPI_STATUS_SIZE) :: istat
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP) :: sum0, sum1
  real(DP), parameter :: zero = 0.0
  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) n=10000
  call mpi_bcast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  istart=n*myrank/nprocs+1
  iend=n*(myrank+1)/nprocs
  sum0=zero
  do i=istart, iend
    sum0=sum0+i
  end do
  call mpi_reduce(sum0,sum1,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
                 MPI_COMM_WORLD,ierr)
  if (myrank==0) print *, 'sum =', sum1
  call mpi_finalize(ierr)
end program dsumn
```

倍精度実数型の定義と
変数・パラメータの定義

MPIプログラムの時間測定

- プログラム中のある部分の経過時間の測定

```
program time
  use mpi
  implicit none
  integer nprocs,myrank,ierr
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP) :: time1,time2,e_time
  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)
  ⋮
  call mpi_barrier(MPI_COMM_WORLD,ierr) 時間測定開始
  time1=mpi_wtime()
  ⋮ } 時間測定対象部分
  call mpi_barrier(MPI_COMM_WORLD,ierr)
  time2=mpi_wtime()
  e_time=time2-time1
  ⋮
  call mpi_finalize(ierr)
end program time
```

時間測定終了

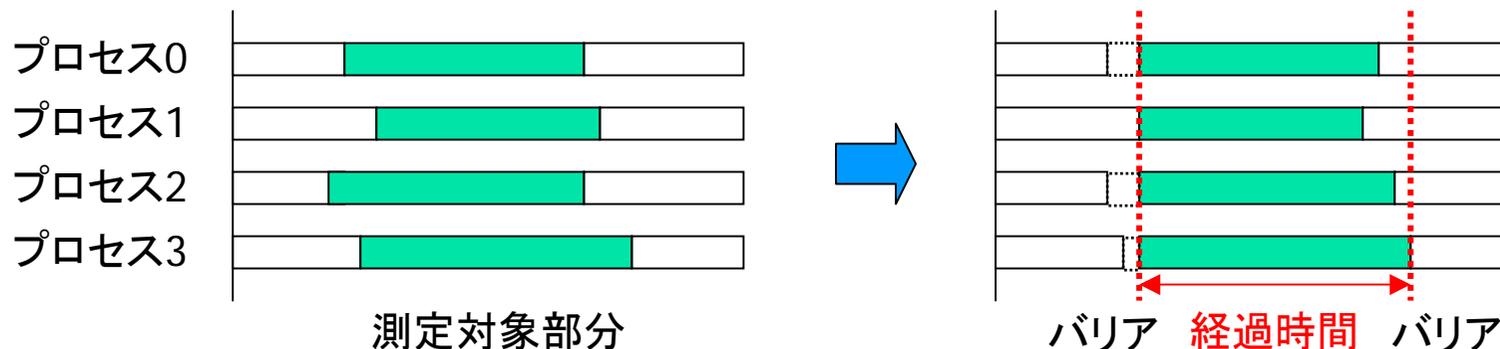
プログラムの解説

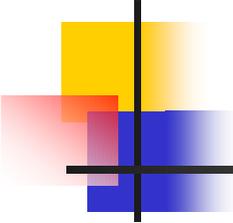
■ 各プロセス内部での時間測定

- `mpi_wtime()`
 - ある時点を基準とした経過秒数を浮動小数点で返す関数

■ プログラム全体の経過時間の測定

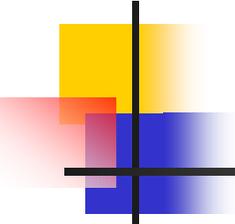
- プログラムの各部分は、プロセスにより開始・終了時間が異なる
- ある部分の経過時間(=最も長いプロセスの時間)を測定するには、最初と最後の `mpi_wtime` の後に `mpi_barrier` を挿入し、同期を取る
- `mpi_barrier(comm,ierr)`
 - `comm` 内の最も遅いプロセスがバリアに到達するまで、全プロセスが待つ





演習2-1

- 演習1-5 のプログラム (dsumn.f90) を次のように変更せよ
 - `mpi_bcast` の前と `mpi_reduce` の後に `mpi_wtime` を挿入し、和の計算の時間を測定して、ランク 0 で出力するようにせよ
 - 後者の `mpi_wtime` については、`mpi_reduce` により同期が取られるため、`mpi_barrier` を入れなくてよい
- $n=10,000,000$ として 1, 2, 4, 8 プロセスで実行し、それぞれ結果が正しいことを確かめよ。また、計算時間の変化を調べよ

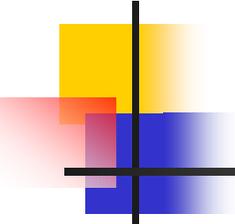


リダクション演算（続き）

- `mpi_allreduce`
 - 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、結果を全プロセスに送る
 - ブロッキング通信
 - 使用方法

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

sendbuf : 送信バッファの先頭アドレス
recvbuf : 受信バッファの先頭アドレス
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
comm : コミュニケータ
ierr : エラーコード(出力)



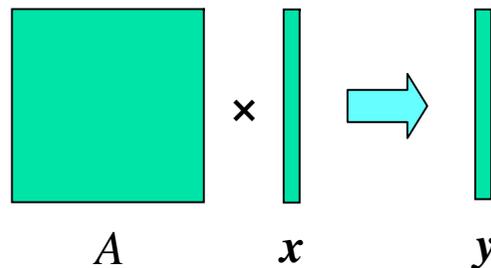
演習2-2

- x を、長さが n で第 i 要素が i のベクトルとする ($x(i) = i$)。このとき、 x を正規化したベクトル $x / \|x\|_2$ を求める MPI プログラムを作成せよ。ただし、 $\|x\|_2$ は x の要素の2乗の和の平方根である。また、結果のベクトルはブロック分割で格納されるようにせよ
- 考え方
 - 演習1-5 のプログラム (dsumn.f90) をベースに修正する
 - まず、各プロセスが自分の担当分の要素について、2乗和を計算
 - プロセス間での総和を求める。ただし、結果は全プロセスで必要なので、`mpi_reduce` でなく `mpi_allreduce` を使う
 - 各プロセスは `mpi_allreduce` の結果を用いて、自分の担当する要素について正規化を行う
- $n=1000$ としてプロセス数を変えて計算し、結果の一部 ($x(n)$ など) を出力して、プロセス数によらずに同じ結果が得られることを確認せよ

リダクション演算の応用：行列ベクトル積

■ 問題

- A を第 (i, j) 要素が $i+j$ の行列とし, x を演習2-2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
```

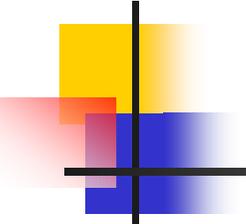
逐次版プログラム (/tmp/100624/mv.f90)

```
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
    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(i)=y(i)+a(i,j)*x(j)
    end do
  end do
  err=0.0d0
  do i=1, n
    ans=dbl(i*n*(n+1)/2+n*(n+1)*(2*n+1)/6)
    err=err+abs(y(i)-ans)
  end do
  print *, 'error =', err
end program mv
```

} A, x の設定

} $y = Ax$ の計算

} 結果の確認



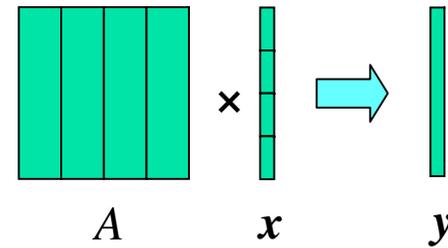
演習2-3

- mv.f90 をコンパイルして実行せよ
 - `cp /tmp/100624/mv.f90 .`
 - `pgf95 mv.f90`
 - `a.out`

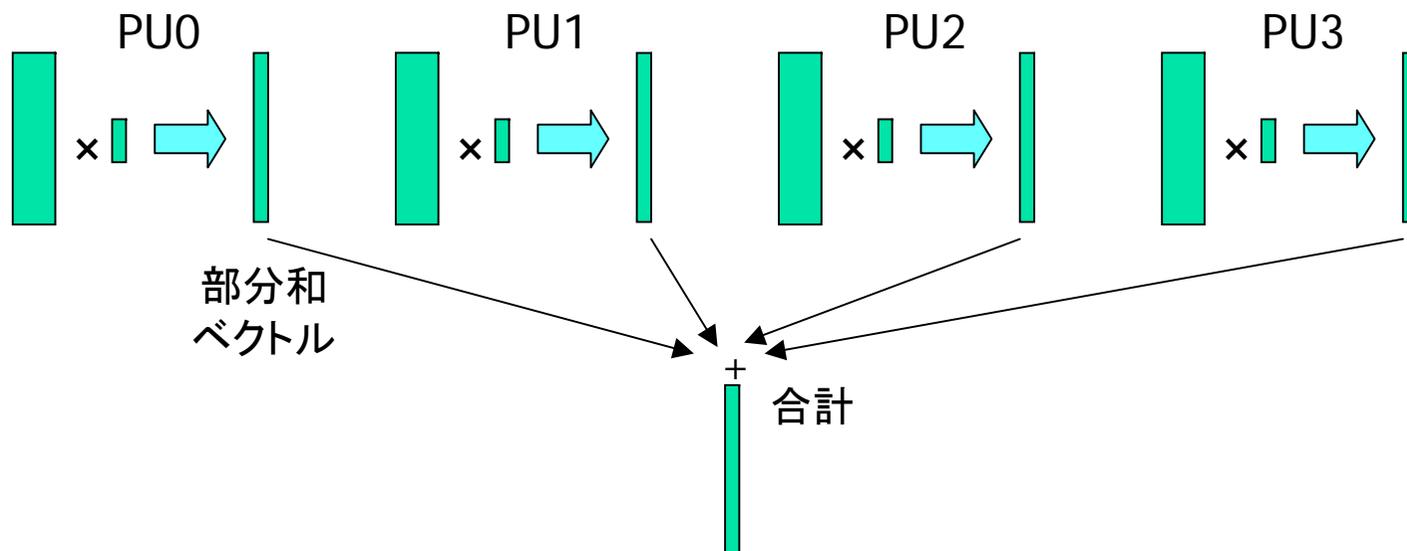
- 結果が正しいことを確認せよ
 - `error = 0.000000000000000000`

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, 部分積ベクトルを計算
 - 部分積ベクトルを `mpi_allreduce` で合計することにより, y を計算

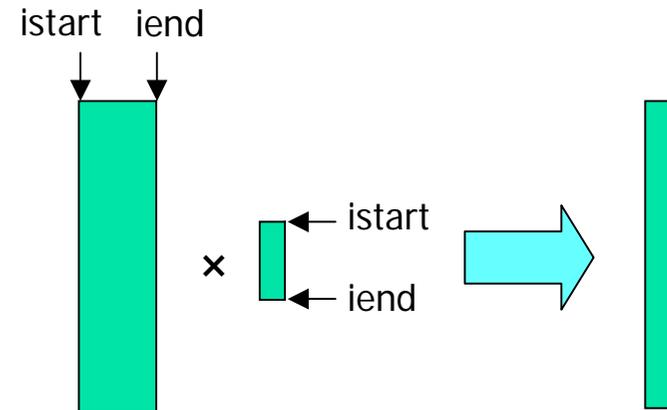


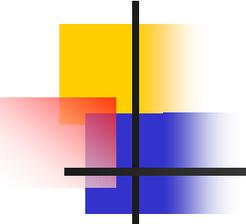
演習2-4

- 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
 - 結果の部分積ベクトルを y でなく配列 yp に入れる
- 部分積の合計
 - `mpi_allreduce` で配列 yp を合計し, 配列 y に入れる
 - `mpi_allreduce` の第3変数 `count` は n とする





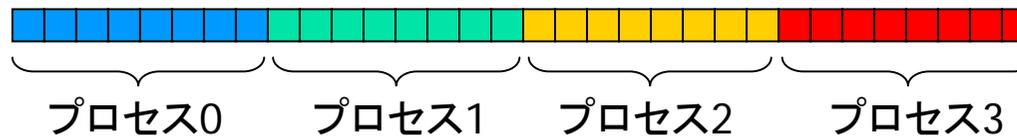
演習2-4(続き)

- $n=1000$ として 8 プロセスで実行し, 結果が正しいことを確かめよ
- 余裕があれば, プロセス数を 1, 2, 4, 8 と変えて実行し, 計算時間の変化を調べよ
 - 初期設定, 結果の確認の部分は時間測定に含めないこと

データの分割方式

- ブロック分割

- 配列をプロセス数分のブロックに分割して割り当て
- この分割を使うと、通信データ量を小さくできる場合が多い



- サイクリック分割

- 配列の要素を1個ずつサイクリックにプロセスに割り当て
- この分割を使うと、PU間の負荷均衡に役立つ場合が多い



- ブロックサイクリック分割

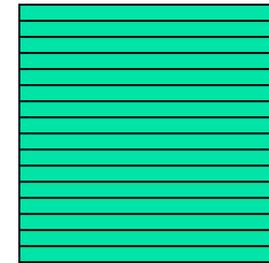
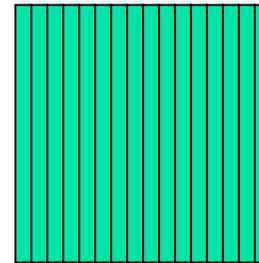


1次元分割と2次元分割

- 行列など2次元以上の配列は, どの方向を分割するかについて任意性あり

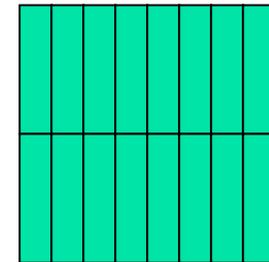
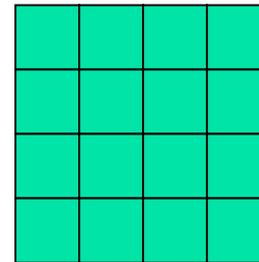
- 1次元分割

- プログラムが簡単
- プロセス数に制約なし
- 通信量が多くなる傾向あり



- 2次元分割

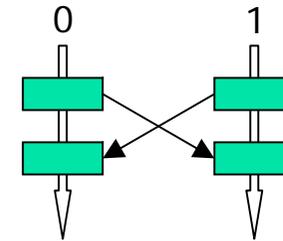
- プログラムはやや複雑
- プロセス数は合成数
- 通信量を抑えられる場合が多い



- 通信量・負荷分散などから最適な分割を決める必要あり

双方向通信とデッドロック

- 双方向の同時通信
 - 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)
      ⋮
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
      ⋮
stop
end
```

宣言および初期化

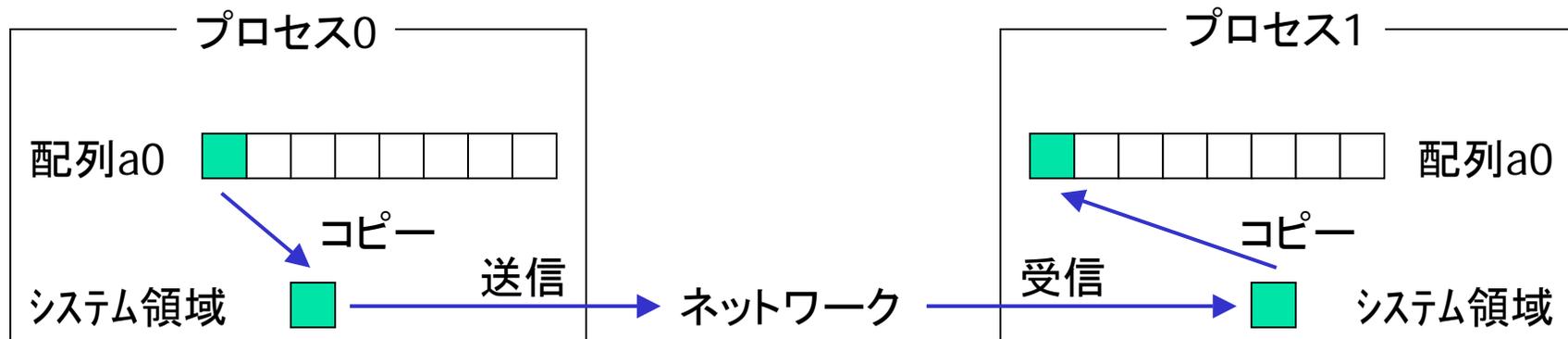
a0, a1を使った処理

- このプログラムは正しく動くか？

双方向通信とデッドロック(続き)

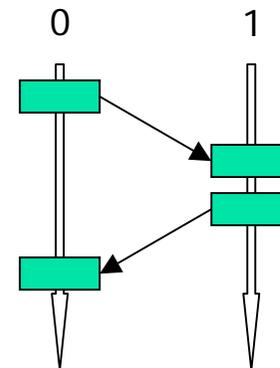
- 実はデッドロックの可能性あり
 - プロセス 0 は, a0 を一部分ずつシステム領域にコピーしてから送信
 - システム領域中のデータが送信され, 相手に受信されるまでは, 次の部分を送信できず, 待機状態となる
 - ところが, 相手も先に a1 の送信を行おうとするため, 同じ理由で待機状態となる

➡ デッドロックが発生

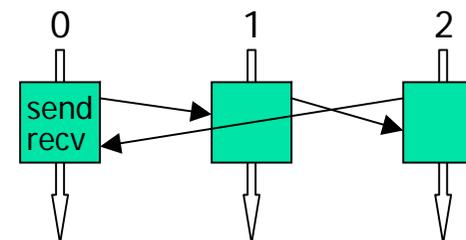


デッドロックの回避法

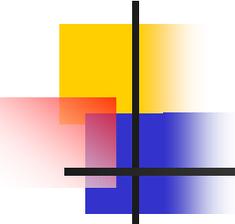
- 送受信の順序の変更
 - プロセス0: 送信してから受信
 - プロセス1: 受信してから送信
 - 問題点: 時間が2倍かかってしまう



- `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

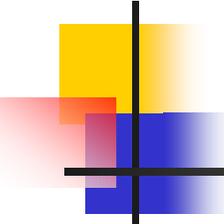


```
call mpi_sendrecv(sendbuff, sendcount, sendtype, dest, sendtag,  
                  recvbuff, recvcount, recvtype, source, recvtag,  
                  comm, status, ierr)
```



演習2-5

- x を演習4と同じベクトルとする。 x をブロック分割し、 P 個のプロセスがそれぞれ1ブロックずつを持っているとする。このとき、`mpi_sendrecv` を繰り返し行うことにより、各プロセスが x の全部の要素を持つようにせよ
- 考え方
 - 各プロセッサで長さ n の配列 $x(n)$ を確保し、自分の担当する部分にのみ値を入れる
 - `mpi_sendrecv` を $P-1$ 回コールし、データの送受信を行う。 i 回目の `mpi_sendrecv` において、プロセス r は、プロセス $\text{MOD}(r+i, P)$ にデータを送信し、プロセス $\text{MOD}(r-i+P, P)$ からデータを受信するようにする
 - 送信バッファ、受信バッファとしては、自分の持つ配列 x のうち、送信または受信したい最初の要素の位置を指定すればよい
 - データを受信し、配列 x の正しい位置に格納するためには、相手プロセスに対する `istart`, `iend` の値を計算しなければならないことに注意する
- $n=100$ として8プロセスで計算し、プロセス0, 4の結果をそれぞれ出力して、意図した結果が得られていることを確認せよ

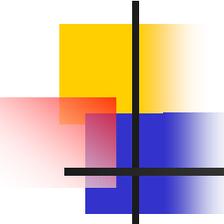


ノンブロッキング通信

- `mpi_isend`
 - 他のプロセスにデータを送信
 - ノンブロッキング通信
 - 送信バッファからのデータ送り出しを指示した直後にリターンする
 - データが送り出されている間に、他の処理を進めることが可能
 - 使用方法

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

`buff` : 送信バッファの先頭アドレス
`count` : 送信するデータの要素数
`datatype` : 送信するデータの型
`dest` : 送信相手のプロセスのランク
`tag` : メッセージID
`comm` : コミュニケータ
`request` : この送信に対して付けられる識別番号(出力)
`ierr` : エラーコード(出力)



ノンブロッキング通信(続き)

- `mpi_wait`
 - `mpi_isend`, `mpi_irecv` による送受信の完了を待つ
 - 必要な理由
 - `mpi_isend` では, 送信バッファからのデータ送り出しの完了を待たずにリターンする
 - 送信バッファへアクセスする場合は, データ送り出しの完了を確認するため, `mpi_wait` をコールする必要がある
 - `mpi_irecv` についても同様
 - 使用方法

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

```
request : mpi_isend または mpi_irecv から返される識別子(入力)
```

```
status  : 状況オブジェクトの配列を指定
```

```
ierr    : エラーコード(出力)
```

ノンブロッキング通信(続き)

- mpi_isend, mpi_irecv の使用法

```
program main
parameter(n=1000000)
double precision a(n)
    ⋮
if (myrank.eq.0) then
    call mpi_isend(a,n,MPI_DOUBLE,1,100,MPI_COMM_WORLD,ir,ierr)
    ⋮
    call mpi_wait(ir,status,ierr)
    ⋮
else
    call mpi_irecv(a,n,MPI_DOUBLE,0,100,MPI_COMM_WORLD,ir,ierr)
    ⋮
    call mpi_wait(ir,status,ierr)
    ⋮
stop
end
```

送信開始

(配列aへのアクセスを行わない処理)

送信終了確認

(配列aへのアクセスを行う処理)

受信開始

(配列aへのアクセスを行わない処理)

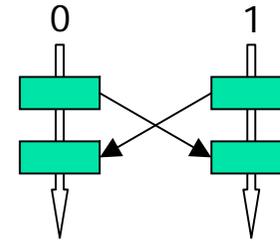
受信終了確認

(配列aへのアクセスを行う処理)

ノンブロッキング通信を用いた双方向通信

■ 双方向の同時通信

- 送信終了を待たずに受信を開始できる
- デッドロックは起こらない



```
program main
parameter(n=1000000)
double precision a0(n),a1(n)
    ⋮
if (myrank.eq.0) then
    call mpi_isend(a0,n,MPI_DOUBLE,1,100,MPI_COMM_WORLD,ir1,ierr)
    call mpi_irecv(a1,n,MPI_DOUBLE,1,200,MPI_COMM_WORLD,ir2,ierr)
else
    call mpi_isend(a1,n,MPI_DOUBLE,0,200,MPI_COMM_WORLD,ir1,ierr)
    call mpi_irecv(a0,n,MPI_DOUBLE,0,100,MPI_COMM_WORLD,ir2,ierr)
end if
call mpi_wait(ir1,status,ierr)
call mpi_wait(ir2,status,ierr)
    ⋮
stop
end
```

宣言および初期化

a0, a1を使った処理