計算科学演習I　第10回講義
「MPIを用いた並列計算(III)」

2013年6月27日

システム情報学研究科 計算科学専攻
横川 三津夫, 山本有作

1．前回の宿題の解説

2．部分配列とローカルインデックス

3．双方向通信

4．ノンブロッキング通信

5．2次元の温度分布の計算

演習2-1

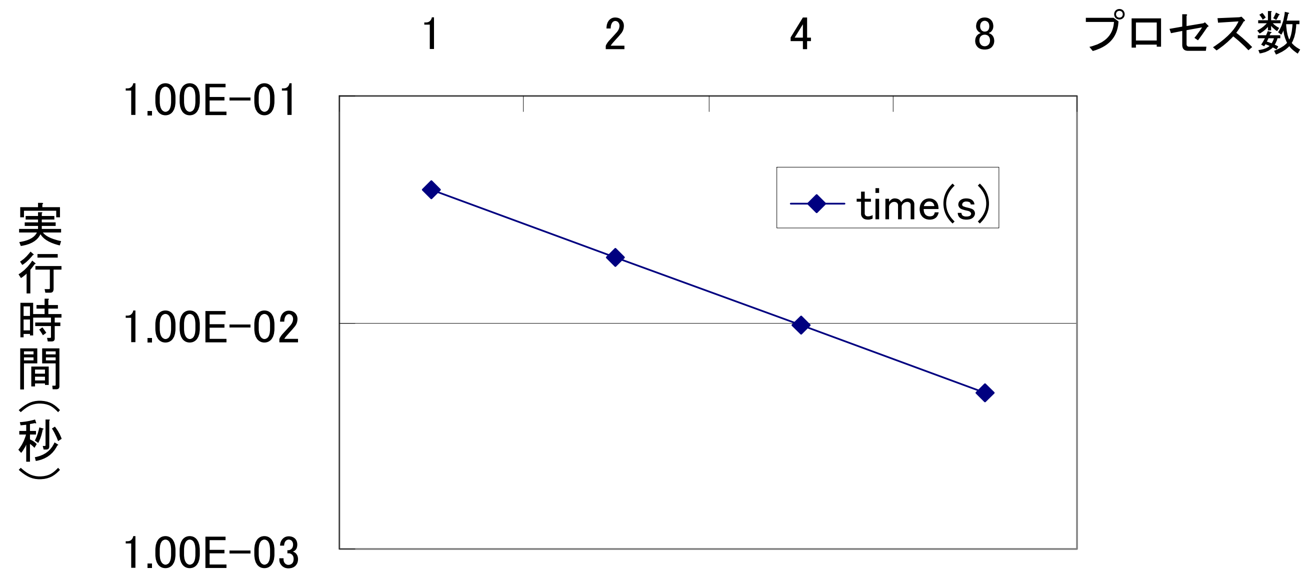
- 演習1-5 のプログラム（dsumn.f90）を次のように変更せよ
  - mpi_bcast の前と mpi_reduce の後に mpi_wtime を挿入し，和の計算の時間を測定して，ランク 0 で出力するようにせよ
    - 後者の mpi_wtime については，mpi_reduce により同期が取られるため，mpi_barrier を入れなくてよい
- n=10,000,000 として 1，2，4，8プロセスで実行し，それぞれ結果が正しいことを確かめよ。また，計算時間の変化を調べよ

解答例(/tmp/130627/dsumn_time.f90)

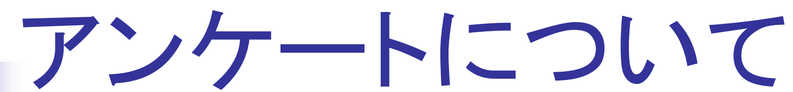
```
program dsumn
  use mpi
  implicit none
  integer :: n,i,istart,iend
  integer :: nprocs,myrank,ierr
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP) :: sum0, sum1
  real(DP), parameter :: zero = 0.0
  real(DP) :: time1,time2,e_time          時間測定用の変数の定義(倍精度実数)
  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=10000000
  call mpi_barrier(MPI_COMM_WORLD,ierr)
  time1=mpi_wtime()                       時間測定開始
  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)
  time2=mpi_wtime()                       時間測定終了(mpi_reduce でバリアを代用)
  e_time=time2-time1
  if (myrank==0) print *, 'sum =', sum1, 'time =', e_time
  call mpi_finalize(ierr)
end program dsumn
```

時間測定結果

- プロセス数と実行時間の関係



実行時間はほぼプロセス数に反比例して減少

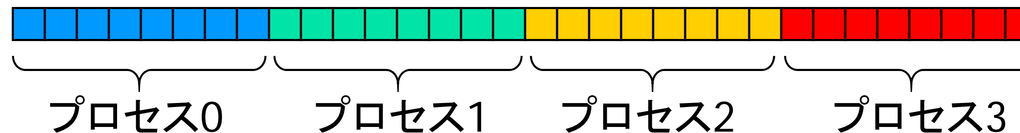


- Wiki ページのアンケート(6/13分)への協力をお願いします
  - 全体的な難易度について
  - 全体的な分量について
  - 各課題(演習2-1, 2-2, 2-4)の難易度について

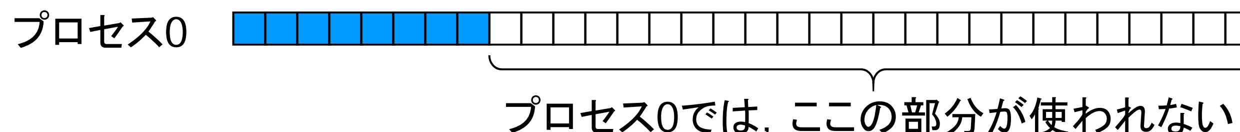
演習2-2

- 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。なお，ベクトルはブロック分割で格納せよ

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



- 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



演習2-2（続き）

- 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

- 課題
  - n=1000 としてプロセス数を変えて計算せよ
  - 正規化されたベクトルの要素は，次のようになる

$$x(i) = i / (n*(n+1)*(2*n+1)/6)^{1/2}$$

これと比較し，計算が正しくできていることを確かめよ

解答例(/tmp/130627/dnorm2.f90)

```
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) ::  sum0,sum1
  real(DP), dimension(n) :: x
  real(DP), parameter :: zero=0.0, one=1.0
  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*myrank/nprocs+1
  iend=n*(myrank+1)/nprocs
  sum0=zero
  do i=istart, iend
    x(i)=real(i,DP)
    sum0=sum0+x(i)*x(i)
  end do
  call mpi_allreduce(sum0,sum1,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ierr)
  sum1=one/sqrt(sum1)
  do i=istart, iend
    x(i)=x(i)*sum1
  end do
  call mpi_finalize(ierr)
end program dnorm2
```

配列xの定義

配列xのうち，自分の担当する部分に要素を入れる
要素の2乗の部分和を計算

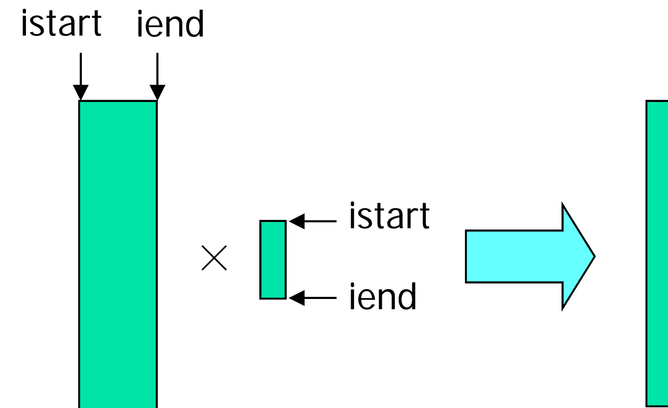
部分和の合計を計算し，平方根を取る

自分の担当する要素を平方根で割る

* dnorm2.f90 には結果チェックの計算も含まれているが，上記プログラムリストでは省略

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, $\boldsymbol{x}$ 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
    - $\boldsymbol{x}$: 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_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

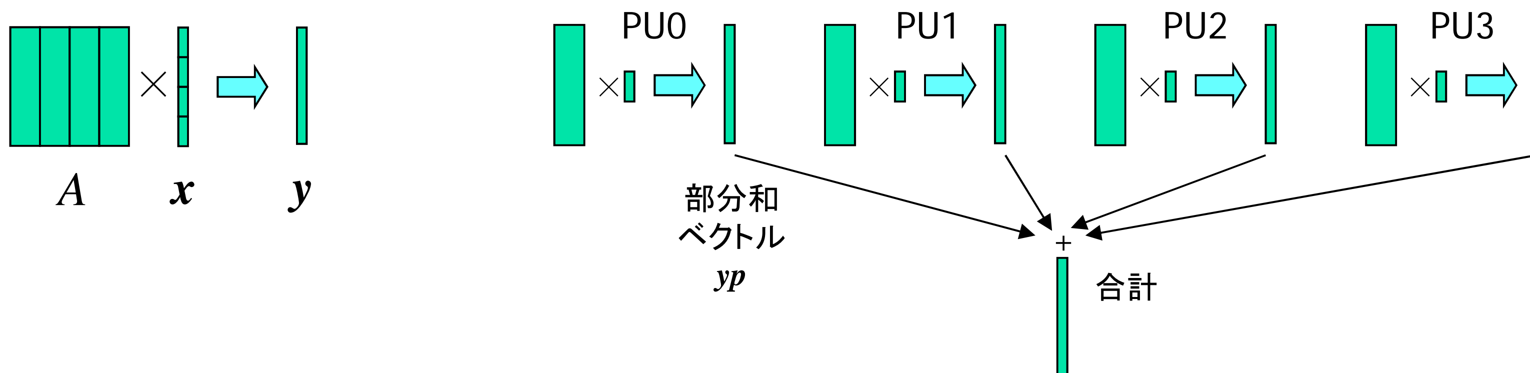
解答例（/tmp/130627/mv_allreduce.f90）

```
program mv_allreduce
  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(n,n) :: a
  real(DP), dimension(n) :: x,y,yp
  real(DP) :: ans,err
  real(DP), parameter :: zero=0
  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)
```

自プロセスの担当する範囲を表わす変数の定義

部分和を格納する変数の定義

（次ページに続く）



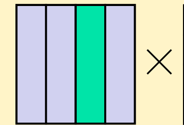
解答例(続き)

```
  istart=n*myrank/nprocs+1
  iend=n*(myrank+1)/nprocs
  do i=istart, iend
    x(i)=i
  end do
  do i=1, n
    do j=istart, iend
      a(i,j)=i+j
    end do
  end do
  do i=1, n
    yp(i)=zero
    do j=istart, iend
      yp(i)=yp(i)+a(i,j)*x(j)
    end do
  end do
  call mpi_allreduce(yp,y,n,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ierr)
  if (myrank==0) then
    err=zero
    do i=1, n
      ans=real(i*n*(n+1)/2+n*(n+1)*(2*n+1)/6,DP)
      err=err+abs(y(i)-ans)
    end do
    print *, 'error =', err
  end if
  call mpi_finalize(ierr)
end program mv_allreduce
```

自プロセスの担当する範囲を計算

A, x のうち，自プロセスの担当する範囲のみを初期化

例えば，rank 3では，緑の部分だけしか使っていない．

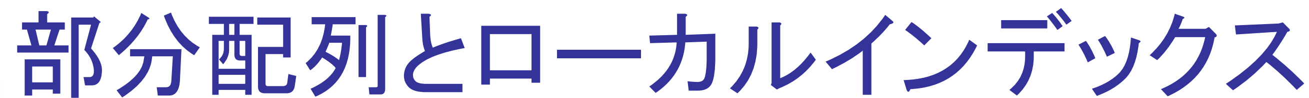


部分和ベクトル yp の計算

yp を合計して y を得る

プロセス0で結果をチェック

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, $\boldsymbol{x}$ 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_allreduce2
  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
  real(DP), dimension(n) :: y,yp
  real(DP) :: ans,err
  real(DP), parameter :: zero=0
  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=n*myrank/nprocs+1
  iend=n*(myrank+1)/nprocs

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

A, x を不定サイズの配列として定義

A, x の領域を確保

（次ページに続く）

部分配列を用いたプログラム（続き）

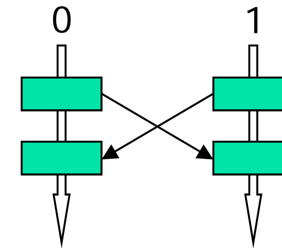
```
  do i=istart, iend
    x(i)=i
  end do
  do i=1, n
    do j=istart, iend
      a(i,j)=i+j
    end do
  end do
  do i=1, n
    yp(i)=zero
    do j=istart, iend
      yp(i)=yp(i)+a(i,j)*x(j)
    end do
  end do
  call mpi_allreduce(yp,y,n,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ierr)
  if (myrank==0) then
    err=zero
    do i=1, n
      ans=real(i*n*(n+1)/2+n*(n+1)*(2*n+1)/6,DP)
      err=err+abs(y(i)-ans)
    end do
    print *, 'error =', err
  end if
  deallocate(a)
  deallocate(x)
  call mpi_finalize(ierr)
end program mv_allreduce2
```

A の列番号，x の要素番号が istart から始まるようにしたので，この部分は変えなくてよい

A，x を解放

双方向通信

- 双方向の同時通信
  - プロセス0はプロセス1に配列変数 a0 を送る
  - プロセス1はプロセス0に配列変数 a1 を送る



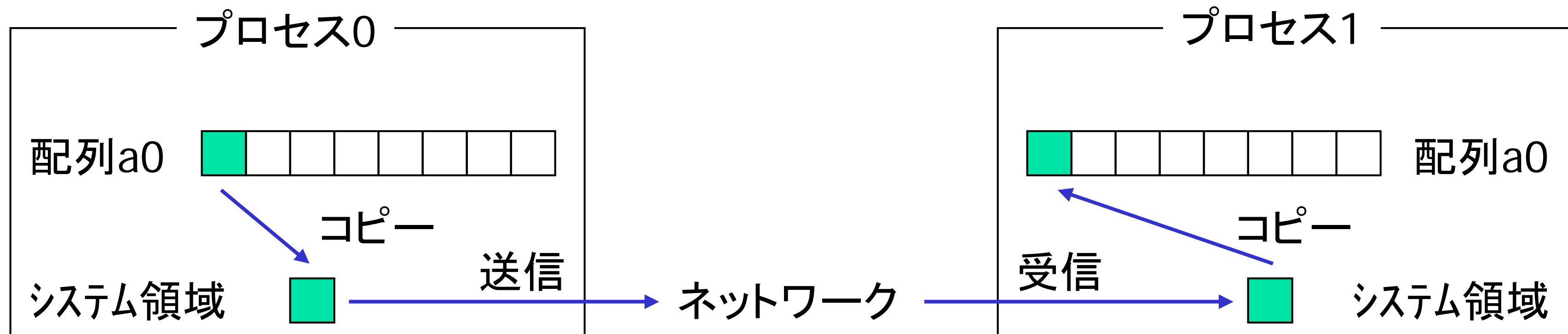

```
program main
parameter(n=1000000)
double precision a0(n),a1(n)
                    ⋮                     宣言および初期化
if (myrank.eq.0) then
   call mpi_send(a0,n,MPI_DOUBLE,1,100,MPI_COMM_WORLD,ierr)
   call mpi_recv(a1,n,MPI_DOUBLE,1,200,MPI_COMM_WORLD,istat,ierr)
else
   call mpi_send(a1,n,MPI_DOUBLE,0,200,MPI_COMM_WORLD,ierr)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

- このプログラムは正しく動くか？（#PJM –L "elapse=00:xx" を小さくすること）

双方向通信(続き)

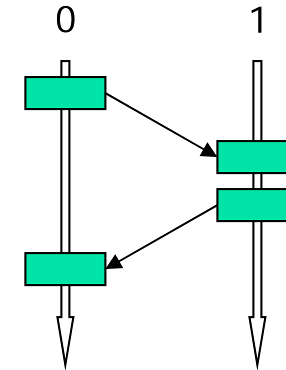
- デッドロックの可能性あり
  - 受信関数(MPI_Recv)の完了が確認されないと，送信(MPI_Send)が終了しない(ブロッキング関数)
  - プロセス 0 は，a0 を一部分ずつシステム領域にコピーしてから送信
  - システム領域中のデータが送信され，相手に受信されるまでは，次の部分を送信できず，待機状態となる
  - ところが，相手も先に a1 の送信を行おうとするため，同じ理由で待機状態となる

⇒ デッドロックが発生

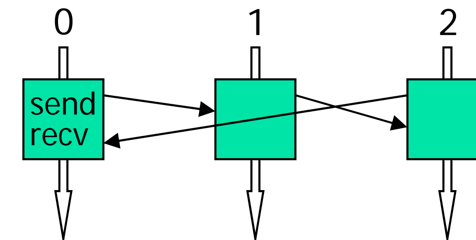


デッドロックの回避法

- 送受信の順序のシリアライズ
  - 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



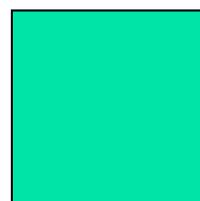
- mpi_sendrecv の利用
  - mpi_send と mpi_recv をまとめて行うルーチン
  - デッドロックは生じない
  - 1回の送受信の時間で済む
  - 送信相手と受信相手が異なってもよい
  - 使用方法



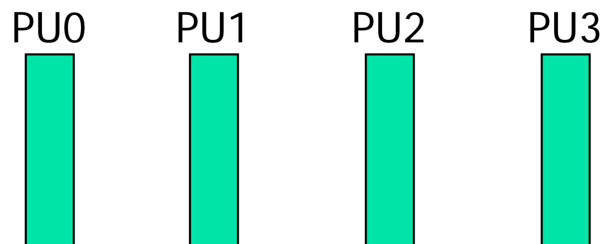

```
call mpi_sendrecv(sendbuff,sendcount,sendtype,dest,sendtag,
                  recvbuff,recvcount,recvtype,source,recvtag,
                  comm,status,ierr)
```

mpi_sendrecv の応用

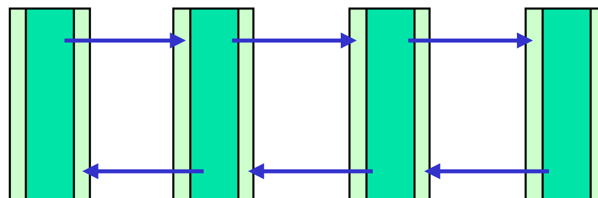
- 問題
  - 2次元配列がブロック列分割されているとする
  - このとき，自分の要素に隣接する隣プロセスの要素を持ってくる



2次元配列



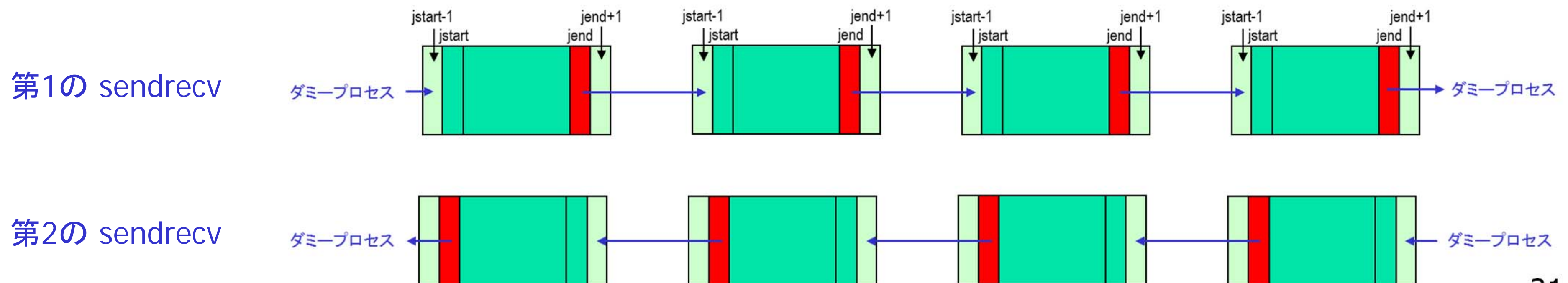
ブロック列分割



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）

mpi_sendrecv の応用（続き）

- 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 列に受信
  - 次に，左隣に jstart 列を送り，右隣から jend+1 列に受信
  - 両端のプロセスは，ダミープロセス（MPI_PROC_NULL) と送受信するようにする．
    - MPI_sendrecvで同じように記述できる．



プログラム例(/tmp/130627/sendrecv.f90)

```
program sendrecv
  use mpi
  implicit none
  integer, parameter :: m=100
  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
```

Recvで必要は配列変数の宣言

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) right=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+jstart-1,DP))
    end do
  end if
  if (myrank < nprocs-1) then
    do i=1, m
      err=err+abs(u(i,jend+1)-real(i+jend+1,DP))
    end do
  end if
  print *, 'myrank =', myrank, 'error =', err

  deallocate( u )

  call mpi_finalize(ierr)
end program sendrecv
```

左右のプロセスのプロセス番号を計算
（存在しない場合は MPI_PROC_NULL とする）

mpi_sendrecv による送受信

正しく受信できたことを確認

演習3-1

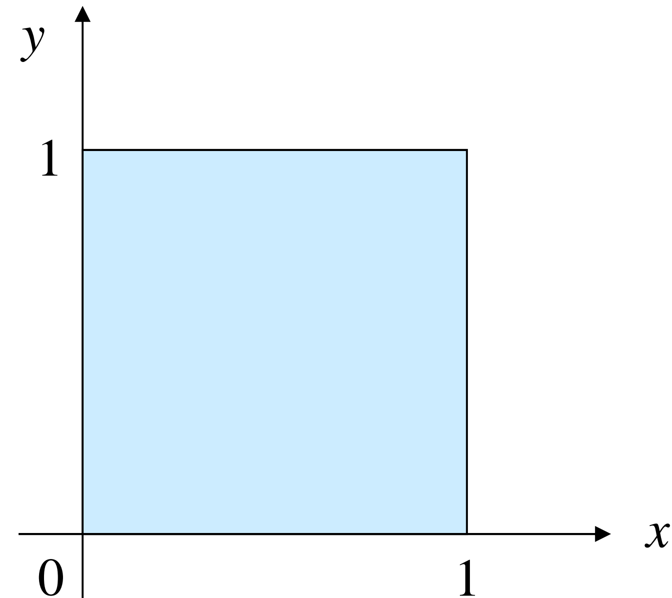
- sendrecv.f90 をコンパイルして 4 または 8 プロセスで実行し，データの送受信が正しくできていることを確かめよ
  - すべてのプロセスが error = 0.0 を出力すればよい

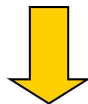
2次元の温度分布の計算

- 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





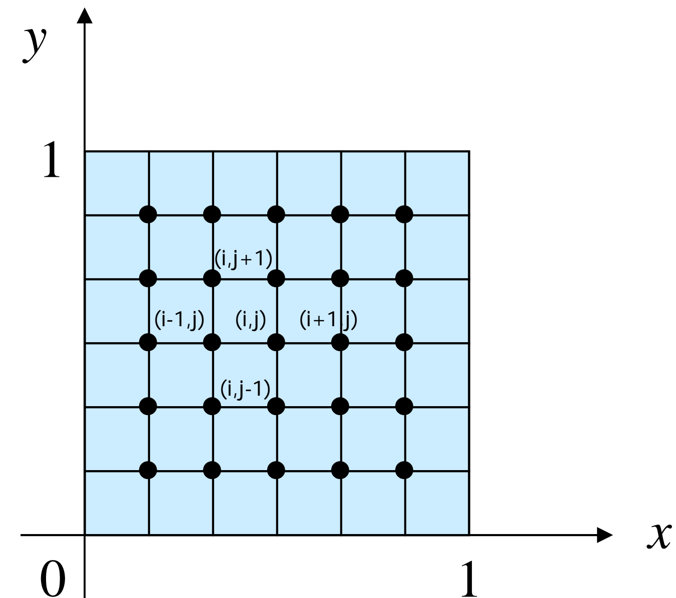
このとき，十分な時間が経った後での温度分布はどうなるか？

2次元の温度分布の計算(続き)

- 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

- 温度の従う方程式
  - 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，

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

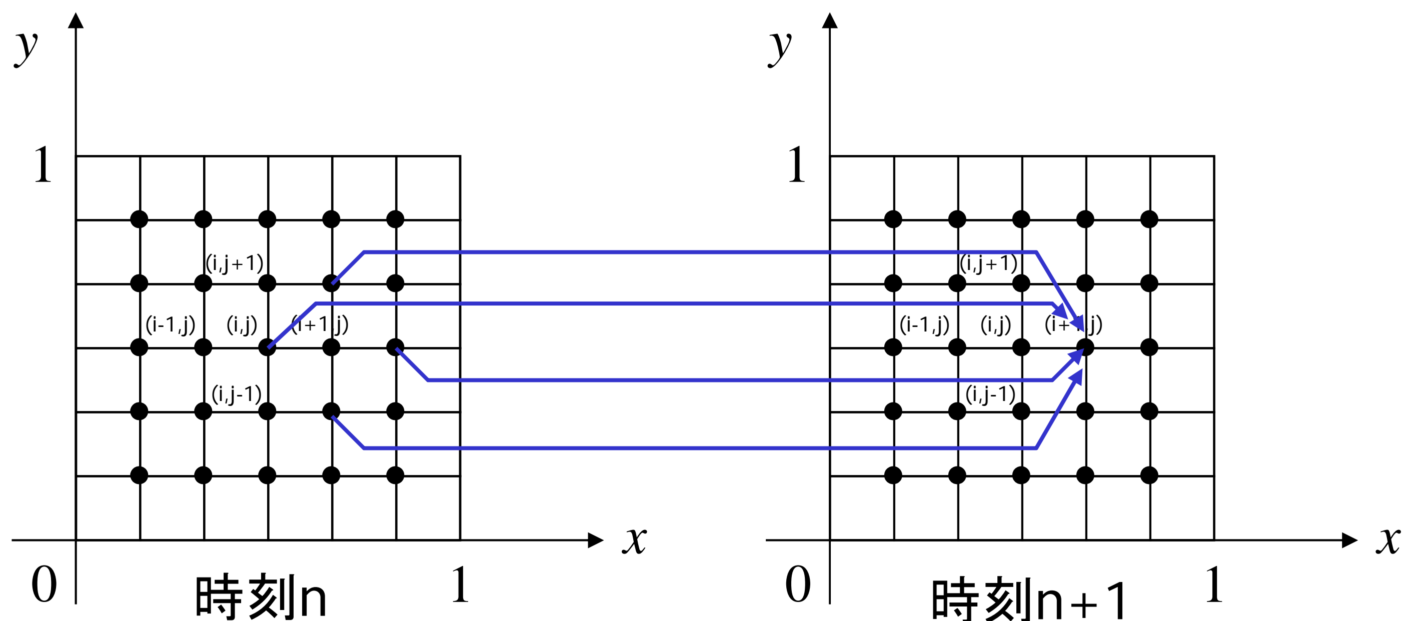


2次元の温度分布の計算（続き）

- 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
```

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



プログラム例（/tmp/130627/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
```

49×49の格子，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

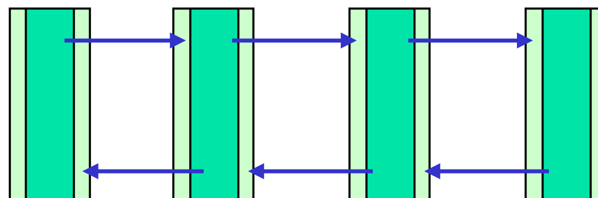
un を新しい u とする

演習3-2

- heat1.f90 をコンパイルし，実行せよ
  - pgf95 heat1.f90
  - ./a.out
- 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

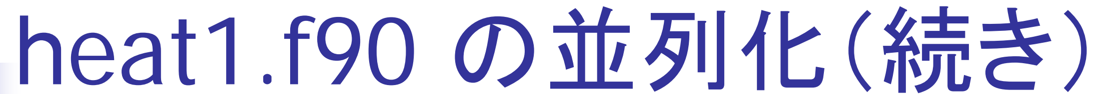
heat1.f90 の並列化

- 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

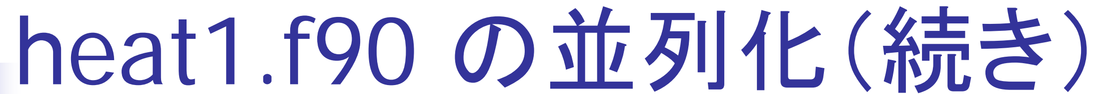


両隣のプロセスから1列を受信
（受信用の領域を確保しておく）

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)
    - mが 並列数2, 4, 8, 16 で割り切れないことに注意
    - 最後のプロセスのみ，割り切れずに残った部分の計算を行うようにする
    - if(myrank == nprocs-1) jend = m
  - 配列の確保
    - allocate(u(0:m+1,jstart-1:jend+1))
    - allocate(un(m,jstart:jend))
  - u を0.0に初期化
    - 同時に領域の左右端，上下端での境界条件も0.0に設定されることに注意
  - 左隣のプロセス番号 left，右隣のプロセス番号 right を定義
    - sendrecv.f90 と同じに，巡回的にする



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
  - 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+1)/2,(m+1)/2) 要素を担当するプロセスは, 100ステップおきにその値を出力
    - if (jstart <= (m+1)/2 .and. jend >= (m+1)/2) という条件を使う

演習3-3

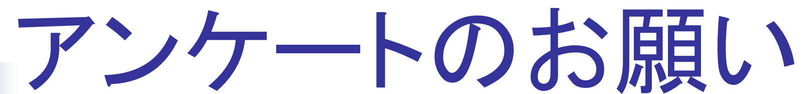
- heat1.f90 を MPI を用いて並列化せよ
- 2, 4, 8, 16プロセスで実行し，プログラムが正しく実行することを確認せよ
  - heat1.f90 の出力結果（点 ((m+1)/2, (m+1)/2) での100ステップおきの値）と，並列計算の出力結果がほぼ同じであることを確認する

演習3-4（任意）

- MPI化したheat1.f90 を MPI を用いて，プロセス数を 1，2，4，8，16 と変えて実行した時の計算時間の変化を調べよ。また，加速率を求めよ.

課題の提出方法と提出期限

- 演習3-3（必須）の提出方法
  - プログラムと実行結果を一つのファイルにまとめ，以下の方法で提出
    - cat xxx.f90 > report_3-3.txt
    - cat result.o? >> report_3-3.txt
    - report_3-3.txtの中身を確認
    - nkf –Lu report_3-3.txt | mail –s mpi_3-3 yokokawa@port.kobe-u.ac.jp
  - プログラムがうまく動かない場合でも，途中結果を提出せよ
- 演習3-4（任意）の提出方法
  - プロセス数(n)，計算時間(Tn)，加速率(Sn)テキストファイルに整理して，別メールとして提出
    - n, Tn, Sn (= T1/Tn)
- 期限：7月3日（水） 午後5時



- Wiki ページのアンケート(6/27)への協力をお願いします
  - 全体的な難易度について
  - 全体的な分量について
  - 部分配列とローカルインデックスの概念について
  - 双方向通信の使い方について
  - 演習3-3(2次元の温度分布の計算の並列化)の難易度について