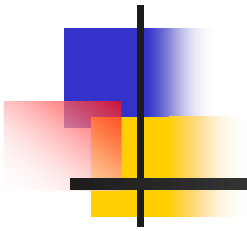
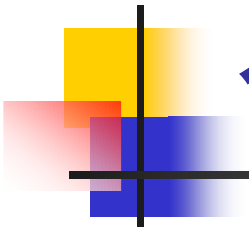


計算科学演習I　第8回講義
「MPIを用いた並列計算(I)」

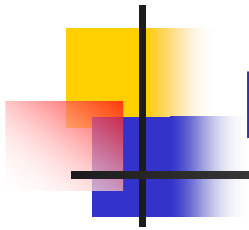
2012年6月7日

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. MPI とは
2. MPI プログラムの構成要素
3. 簡単な MPI プログラムの例
4. 集団通信

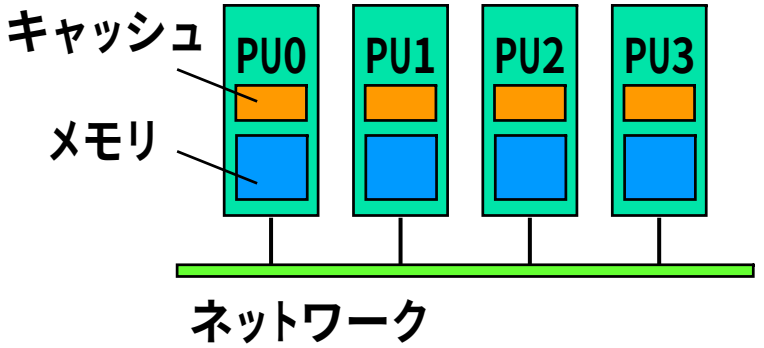


MPIとは

- **分散メモリ型並列計算機上での並列プログラミングのためのプロセッサ間通信ライブラリ**
  - **ベース言語** (FORTRAN/C/C++) **に**プロセッサ間通信**を行うサブルーチン/関数を加えることで，並列プログラミングができるように拡張**
- 1992**年頃より米国の計算機メーカー・大学を中心に標準化**
- MPI **の規格**
  - 1995　MPI: Message Passing Interface Standard
  - 1997　MPI2: Extensions to the Message Passing Interface

分散メモリ型並列計算機（復習）

- アーキテクチャ
  - 各々がメモリを持つ複数のPUを ネットワークで接続
  - 各PUはそれぞれ自分の持つメモリ のみにアクセス可能

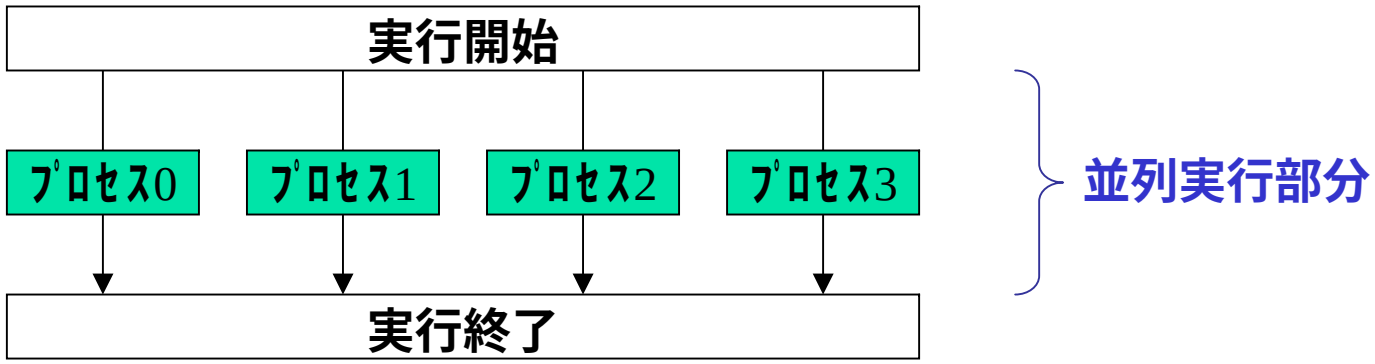


- 特徴
  - 数千～数万PU規模の並列が可能
  - PU間へのデータ分散を意識したプログラミングが必要

- プログラミング言語
  - FORTRAN/C/C++ ＋ MPI を使用

MPIの実行モデル

- 実行モデル
  - 実行開始から終了まで，全プロセスが同じプログラムを実行
  - 各プロセスは固有のプロセス番号 (ランク) を持つ
    - P 個のプロセスで実行するとき，ランクは 0 から P-1 までの整数
  - 演算対象データやIF文による分岐などは，プロセスごとに異なってよい



MPIの実行モデル (続き)

- メモリ空間
  - プログラム中で変数・配列を定義すると，同じ名前の変数・配列がプロセス毎に独立に割り当てられる
    - 同じ名前の変数・配列でも，プロセス毎に違う値を持つことができる
  - 他のプロセスの持つ変数・配列にはアクセスできない

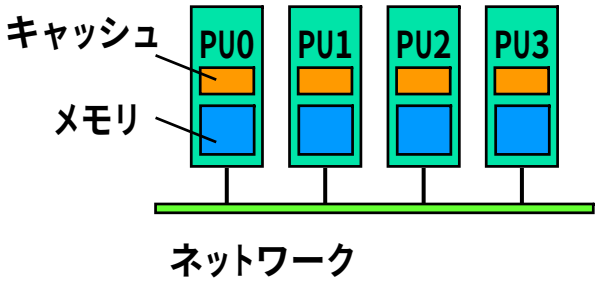
```
program main
integer :: a
integer :: b(100)
```

変数a，配列b(100)が，各プロセスに対してそれぞれ独立に割り当てられる

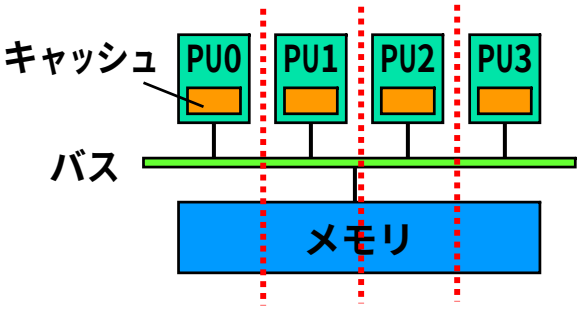
- プロセッサ間通信
  - 他プロセスの持つ変数・配列の値を参照するには，プロセッサ間通信によりその変数・配列の値を送ってもらう必要がある
  - このための通信関数の集合が MPI

MPIの実行環境

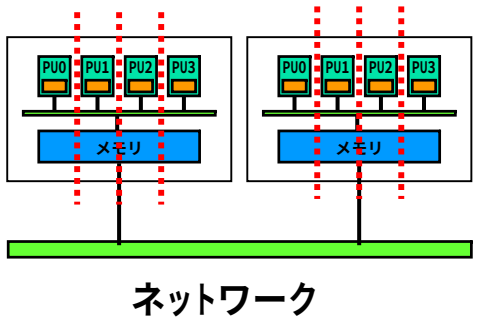
- **分散メモリ型並列計算機**
  - MPI がもともと想定している環境
  - 各プロセスが1個のPUに割り当てられる

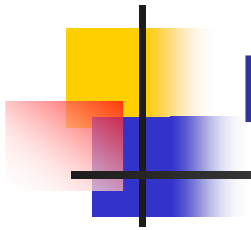


- **共有メモリ型並列計算機**
  - メモリを分割して各PUに割り当て，仮想的な分散メモリ環境を作る
  - 既存のMPIプログラムを共有メモリ型並列計算機で実行するときに便利



- SMP クラスタ
  - 各ノードのメモリを分割して各PUに割り当て，仮想的な分散メモリ環境を作る
  - 本講義でもこの環境を用いる



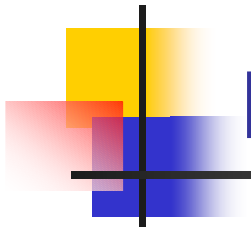


MPI プログラムの構成要素

- 元のプログラム
  - 通常の FORTRAN または C/C++ で書かれたプログラム
  - 分散メモリを意識して書かれている必要あり
- 環境設定・環境取得のための関数
  - MPI の初期化・終了処理を行う関数
  - 自プロセスのランク，全プロセス数を取得する関数
- 通信関数
  - プロセス間で通信を行うためのサブルーチン/関数
  - 1対1通信と集団通信がある

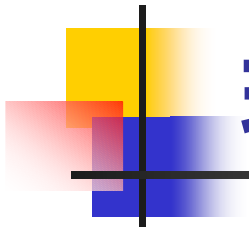
MPI プログラムの構成要素 (続き)

- **基本データ型と定義済み演算**
  - MPI_INTEGER, MPI_REAL, MPI_SUM など
  - 通信関数のデータ型指定や, 通信と演算を同時に行う関数 (総和など) での演算の指定に用いる
- **インクルードファイル**
  - プログラムの最初で use mpi と記述
- **実行時の引数**
  - プログラムの実行に用いるプロセス数を指定



MPI の特徴

- 移植性
  - MPI でプログラムを書くことで，様々な分散メモリ型並列計算機での実行が可能
  - 共有メモリ型並列計算機でも実行可能
- スケーラビリティー
  - 適切なアルゴリズムに基づいて並列プログラムを書けば，数千～数万個のプロセッサを持つ並列計算機の能力を引き出すことが可能
- 多様な集団通信
  - 総和，ブロードキャストなど多様な通信関数が予め用意されている
- 性能解析ツール
  - 負荷分散，通信時間等の詳細な情報を取得するツールが使用可能



現在利用可能な MPI の実装

- MPICH
  - **米国のアルゴンヌ国立研究所が提供しているフリーな** MPI
  - **現在では** MPICH2 **が主流**
  - http://www.mcs.anl.gov/research/projects/mpich2/
- OpenMPI
  - MPI **に関する様々な先行プロジェクトの成果を基に開発された** MPI
  - http://www.open-mpi.org/
- **各計算機メーカーの提供する** MPI
  - **並列計算機メーカー各社が，自社の計算機向けに，それぞれ** MPI **の実装を提供している**

簡単な MPI プログラムの例 (1)

- **並列版** “Hello World”

```
program hello
  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)
  print *, 'Hello, World. My rank is', myrank
  call mpi_finalize(ierr)
end program hello
```

ファイル mpif.h のインクルード

MPI の初期化

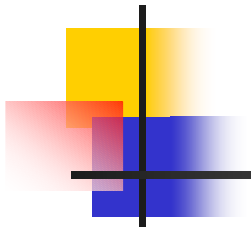
全プロセス数の取得

自分のプロセス番号の取得

MPI の終了処理

- **実行結果** (4プロセスの場合)

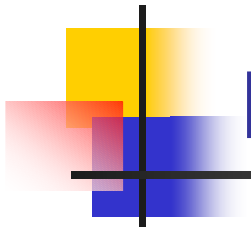
```
Hello, World. My rank = 0
Hello, World. My rank = 1
Hello, World. My rank = 2
Hello, World. My rank = 3
```



プログラムの解説

- mpi_init と mpi_finalize
  - **プログラムの最初と最後にコールし，それぞれ**MPI**の初期化，終了処理を行う**
- mpi_comm_size(comm,nprocs,ierr)
  - comm **はプロセスグループを指定する**コミュニケータ
    - MPI_COMM_WORLD **とすると，全プロセスを含むグループの意味**
  - comm **内の全プロセス数が** nprocs **に入る**
  - ierr **にはエラーコードが入る**
- mpi_comm_rank(comm,myrank,ierr)
  - comm **内での自分のプロセス番号が** nprocs **に入る**

これらの関数は，どんな MPI **プログラムでも，サンプルプログラムと同じ順番でコールすればよい**



MPIプログラムのコンパイル方法

- mpif90 コマンドを使用
  - mpif90 hello.f90 → 実行ファイル a.out ができる
  - mpif90 -o hello hello.f90 → 実行ファイル hello ができる

MPIプログラムのバッチジョブ投入方法

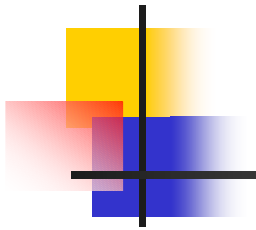
キュー名	ノード数	メモリ容量指定	同時実行ジョブ数上限(本)
PCS-A	1-8	1GB/node	16
PCL-A	9-64	2GB/node	16

MPI利用のシェルスクリプト例

```
#PBS –l   cputim_job=00:05:00   ← 使用CPU時間を指定
#PBS –l   memsz_job=2gb         ← 使用メモリサイズを指定 (PCS-Aでは1, PCL-Aでは2)
#PBS –l   cpunum_job=1          ← 1ノードあたりのプロセス数 (本日の演習では1に固定)
#PBS –T   vltmpi                ← Voltaire MPIを指定
#PBS –b   4                     ← 4ノード (4プロセス) を指定
#PBS –q   PCL-A                 ← 投入先のキュー名を指定 (本日の演習ではPCL-Aを使用)
cd 作業ディレクトリ (例えば、/home/users/yyamamoto/120607)
mpirun_rsh  -np  4  ${NQSII_MPIOPTS} 実行プログラム (たとえば ./a.out )
```

- サンプルシェルスクリプトはscalar:/tmp/120607/large.sh
- 各自コピー後、編集して利用

バッチジョブ関連コマンド



- **ジョブ投入**： qsub　**ジョブスクリプトファイル**
  - **例えば，ジョブスクリプトファイルが** large.sh **なら，** qsub large.sh **でシステムにジョブ投入**
- **ジョブの状態表示**： qstat
  - **投入したジョブの状態を表示（キュー状態か，** Run**か終了か）**
- **投入ジョブのキャンセル**： qdel　**ジョブ番号**
  - **ジョブ番号は，** qstat **で表示される** RequestID **に相当**

ジョブ実行結果

- ジョブの実行が終了すると、標準出力/標準エラー出力がそれぞれ通常ファイルとして出力される。

small.sh というジョブスクリプトを投入した場合

```
[ss099@scalar 120607]$ ls -l large.sh.?20
-rw-r--r--  1  ss099  ss2012    244  June 7  2012  large.sh.e20
-rw-r--r--  1  ss099  ss2012  3285  June 7  2012  large.sh.o20
```

(1) (2) (3)

(1) ジョブスクリプトファイル名

(2) 標準出力(o), 標準エラー出力(e)

(3) ジョブのリクエストIDの数字部

演習1-1

- hello.f90 を2，4プロセスで実行し，結果を確認せよ
- 作業手順の詳細

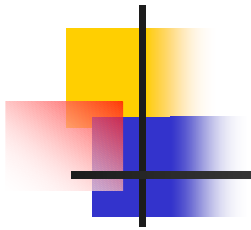
1. scalar マシンにログイン （“ssh scalar.scitec.kobe-u.ac.jp”）
2. mkdir コマンドで、適当な作業ディレクトリを作成
   (たとえば、“mkdir 120607” で 120607 という名前のディレクトリを作成)
3. そのディレクトリに移動 （“cd 120607”）
4. 必要なファイル(large.sh, hello.f90)をコピー
   (“cp /tmp/120607/large.sh .”, “cp /tmp/120607/large.f90 .”)
5. large.sh の中身を確認･編集 (作業ディレクトリの指定，利用ノード数の変更など)
6. ジョブ投入 （”qsub large.sh”）
7. ステータスの確認 （“qstat”）
8. ジョブ終了後、出力の確認 (“more large.sh.o***”)

- 結果の確認
  - 各プロセスが異なる myrank の値を出力することを確認
  - これにより，myrank という変数は，各プロセスが独立に持っていることがわかる

簡単な MPI プログラムの例 (2)

- 1から100までの整数の和を2並列で求めるプログラム

```
program sum100
  use mpi
  implicit none
  integer :: i,istart,iend,isum,isum1
  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*50+1                    各プロセスが計算すべき範囲を求める
  iend=(myrank+1)*50
  isum=0
  do i=istart, iend
     isum=isum+i                        各プロセスが部分和を計算
  end do
  if (myrank==1) then
     call mpi_send(isum,1,MPI_INTEGER,0,100,MPI_COMM_WORLD,ierr)
  else                                  プロセス1は部分和をプロセス0に送信
     call mpi_recv(isum1,1,MPI_INTEGER,1,100,MPI_COMM_WORLD,istat,ierr)
  end if                                プロセス0は部分和をプロセス1から受信
  if (myrank==0) print *, 'sum =', isum+isum1
  call mpi_finalize(ierr)               プロセス0は合計を表示
end program sum100
```



プログラムの解説

- mpi_send
  - **他のプロセスにデータを送信**
  - **ブロッキング通信**
    - **データが送信バッファから送り出された後にリターンする**
  - **使用方法**

```
call mpi_send(buff,count,datatype,dest,tag,comm,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
ierr     : エラーコード(出力)
```

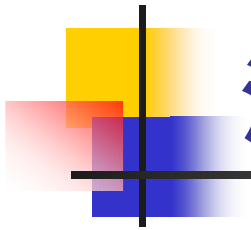
プログラムの解説 (続き)

- 各引数に関する注意
  - buff
    - 送信する領域は，メモリ上で連続アドレスでなければならない
    - 他の通信関数でも同じ
  - datatype
    - MPI_INTEGER，MPI_REAL，MPI_DOUBLE_PRECISION などが使用できる
  - tag
    - 同じ相手プロセスに対して複数の mpi_send を行うとき，それらを区別するために使用する
    - 対応する mpi_recv では，同じ tag を使わなければならない
    - 送受信の順番などによって複数の通信を区別できる場合は，tag の値は同じでもよい

プログラムの解説 (続き)

- mpi_recv
  - **他のプロセスからデータを受信**
  - **ブロッキング通信**
    - **データが受信バッファに完全に格納された後にリターンする**
  - **使用方法**

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

buff     : 受信バッファの先頭アドレス
count    : 受信するデータの要素数
datatype         : 受信するデータの型
dest     : 受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
status   : 状況オブジェクトの配列を指定
ierr     : エラーコード (出力)
```

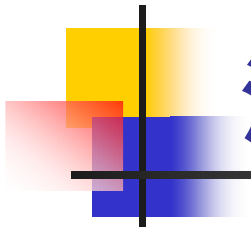


演習1-2

- sum100.f90 を2プロセスで実行し，結果を確認せよ

- **実行に当たっての注意**
  - **プログラムの所在**：/tmp/120607/sum100.f90
  - large.sh **を使ってジョブを投入すること**
  - large.sh **でプロセス数を**2**に変えること**（2**箇所**）**を忘れないように**

- **結果の確認**
  - **プロセス**0**が正しい結果を出力することを確認**



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
  - 各プロセッサが計算する部分和の範囲を変更

```
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

プログラムと実行結果を1つのファイル (kekka.txt など) にまとめ，yyamamoto にメールで送ること。タイトルは ex1-3 とすること

mail yyamamoto -s ex1-3 < kekka.txt

集団通信

- 1からnまでの整数の和を並列で求めるプログラム

```
program sumn
  use mpi
  implicit none
  integer :: n,i,istart,iend,isum,isum1
  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)
  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
  isum=0
  do i=istart, iend
     isum=isum+i
  end do
  call mpi_reduce(isum,isum1,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ierr)
  if (myrank==0) print *, 'sum =', isum1
  call mpi_finalize(ierr)
end program sumn
```

プロセス0はnを読み込む

nの値を放送

各プロセスが計算すべき範囲を求める

各プロセスが部分和を計算

部分和の総和を計算

プロセス0は合計を表示

プログラムの解説

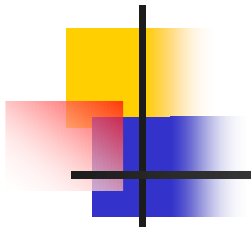
- mpi_bcast
  - **1個のプロセスから他の全てのプロセスにデータを送信**
  - **ブロッキング通信**
    - root **はデータが送信バッファから送り出された後にリターン**
    - **他のプロセスはデータが受信バッファに完全に格納された後にリターン**
  - **使用方法**

```
call mpi_bcast(buff,count,datatype,root,comm,ierr)

buff     ： (送信プロセス) 送信バッファの先頭アドレス
            (受信プロセス) 受信バッファの先頭アドレス
count    ： データの要素数
datatype          ： データの型
root     ： 送信元のプロセスのランク
comm     ： コミュニケータ
ierr     ： エラーコード (出力)
```

プログラムの解説 (続き)

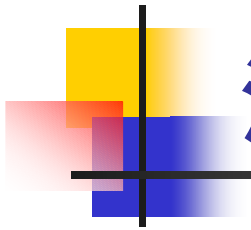
- mpi_reduce
  - **全プロセスの持つデータに対してリダクション演算を行い，結果を** root**に送る**
  - **ブロッキング通信**
  - **使用方法**

```
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)
```



リダクション演算とは

- 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
- MPIで使えるリダクション演算
  - MPI_SUM，MPI_PROD，MPI_MAX，MPI_MIN
- 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



演習1-4

- sumn.f90 を4，8プロセスで実行し，結果を確認せよ

- **実行に当たっての注意**
  - **プログラムの所在**：/tmp/120607/sumn.f90
  - large.sh **を使ってジョブを投入すること**

- **結果の確認**
  - **プロセス0が正しい結果を出力することを確認**

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 sum，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（臼井先生の講義参照）

```
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

プログラムと実行結果を1つのファイル（kekka2.txt など）にまとめ，yyamamoto にメールで送ること。タイトルは ex1-5 とすること

mail yyamamoto -s ex1-5 < kekka2.txt

MPIプログラムの時間測定

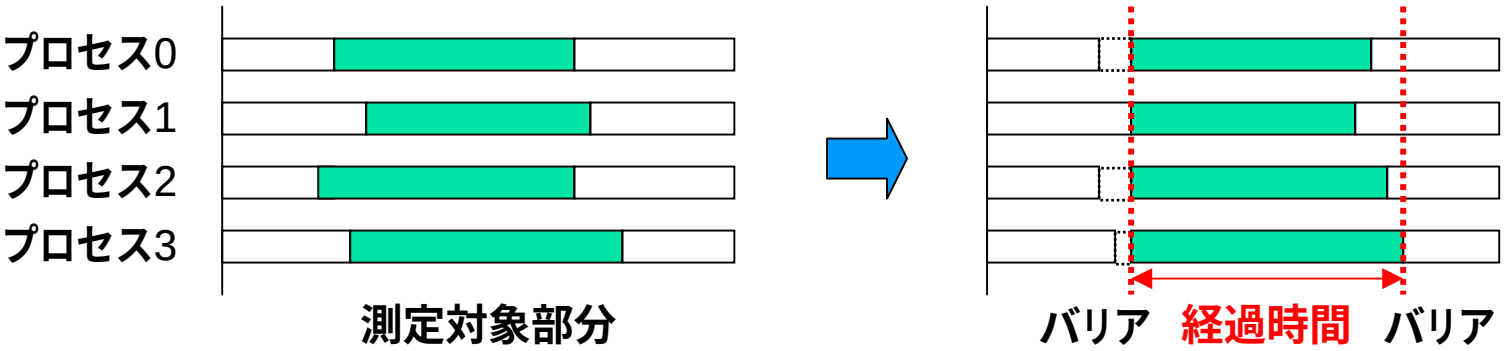
- プログラム中のある部分の経過時間の測定

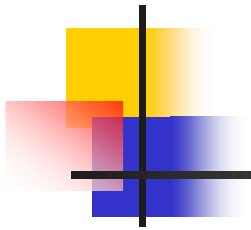
```
program time
  use mpi
  implicit none
  integer nprocs,myrank,ierr
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP) :: time1,time2,e_time
  call mpi_init(ierr)
  call mpi_comm_size(MPI_COMM_WORLD,nprocs,ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,myrank,ierr)
                    ⋮
  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()                       時間測定終了
  e_time=time2-time1
                    ⋮
  call mpi_finalize(ierr)
end program time
```

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

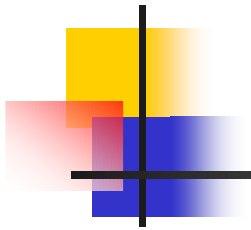
- プログラム全体の経過時間の測定
  - プログラムの各部分は，プロセスにより開始･終了時間が異なる
  - ある部分の経過時間（＝最も長いプロセスの時間）を測定するには，最初と最後の mpi_wtime の後に mpi_barrier を挿入し，同期を取る
  - mpi_barrier(comm,ierr)
    - comm 内の最も遅いプロセスがバリアに到達するまで，全プロセスが待つ





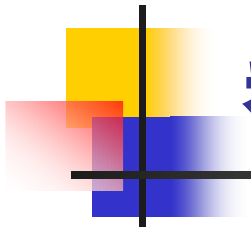
演習1-6

- **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**



レポートの締切について

- **演習**1-3, 1-5**とも，来週の計算科学演習の時間までに提出すること**



参考文献

- P.パチェコ (秋葉 博訳)：“MPI並列プログラミング”, 培風館, 2001.

- 青山幸也：“並列プログラミング入門　MPI版”, http://accc.riken.jp/HPC/training.html