計算科学演習

MPI 基礎

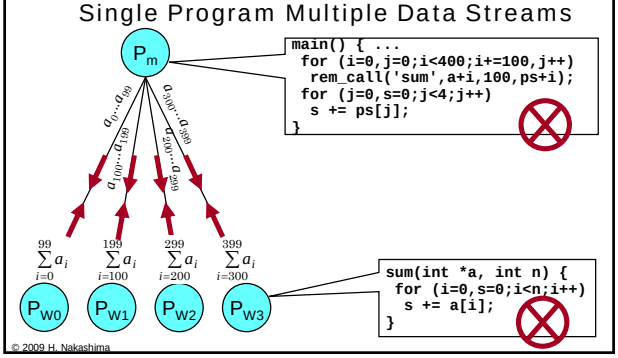
学術情報メディアセンター
情報学研究科・システム科学専攻
中島　浩

目次

プログラミングモデル

SPMD

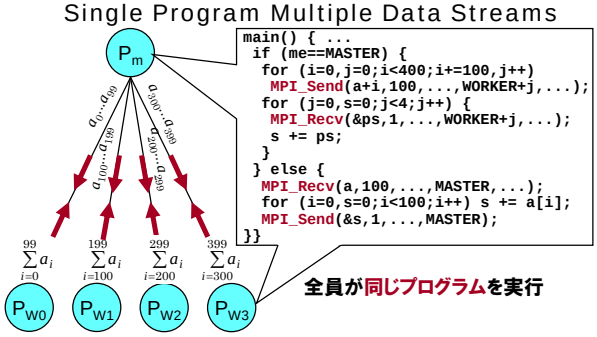
- SPMD=
  Single Program Multiple Data Streams



プログラミングモデル

SPMD

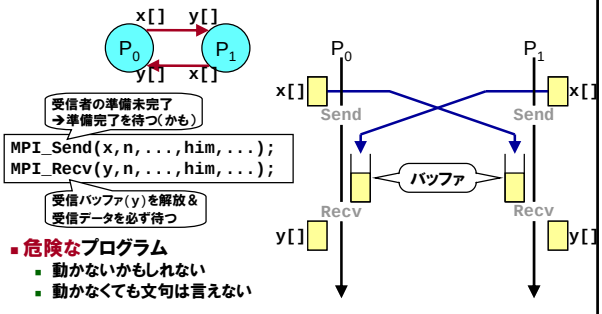
- SPMD=
  Single Program Multiple Data Streams



プログラミングモデル

同期通信／非同期通信 (1)

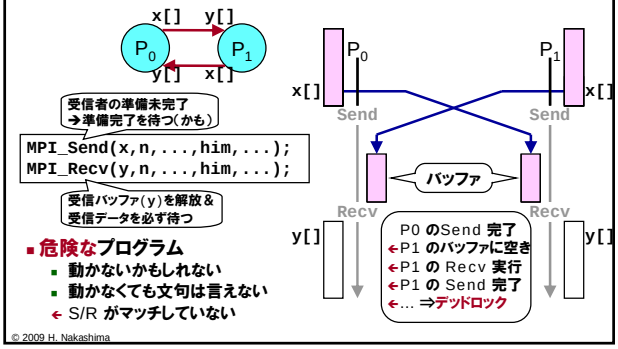
- （バッファ付）同期通信 (1/3)



プログラミングモデル

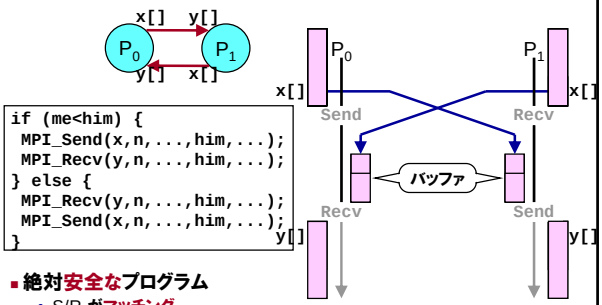
同期通信／非同期通信 (2)

- （バッファ付）同期通信 (2/3)



プログラミングモデル
同期通信／非同期通信 (3)

- （バッファ付）同期通信 (3/3)

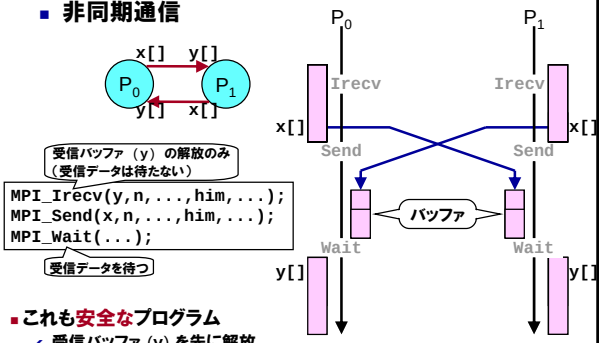



```
if (me<him) {
 MPI_Send(x,n,...,him,...);
 MPI_Recv(y,n,...,him,...);
} else {
 MPI_Recv(y,n,...,him,...);
 MPI_Send(x,n,...,him,...);
}
```

- 絶対安全なプログラム
  - S/R がマッチング

プログラミングモデル
同期通信／非同期通信 (4)

- 非同期通信



受信バッファ（y）の解放のみ（受信データは待たない）

```
MPI_Irecv(y,n,...,him,...);
MPI_Send(x,n,...,him,...);
MPI_Wait(...);
```

受信データを待つ

- これも安全なプログラム
  - 受信バッファ (y) を先に解放

MPI 概論
プログラム構造 in C

```
#include <mpi.h>
#define N ...
#define MCW MPI_COMM_WORLD
int main(int argc, **char argv) {
  int np,me;  double sbuf[N],rbuf[N];
  MPI_Status st;
  ...
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MCW,&np);
  MPI_Comm_rank(MCW,&me);
  ...
  MPI_Send(sbuf,N,MPI_DOUBLE,(me+1)%np,0,MCW);
  ...
  MPI_Recv(rbuf,N,MPI_DOUBLE,(me+1)%np,0,MCW,&st);
  ...
  MPI_Finalize();
}
```

全 MPI プロセスの集合(communicator)を表す定数(後述)

MPI プロセス初期化＆真のargc/argv 取得

プロセス集合の大きさ

プロセス集合中の id (rank)

メッセージタグ(後述)

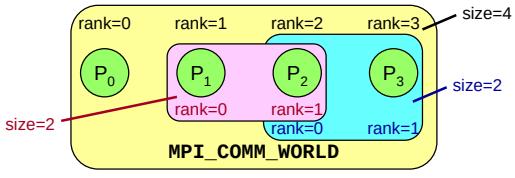
終了情報

MPI 概論
プログラム構造 in Fortran

```
program main
include 'mpif.h'
integer,parmeter::MCW=MPI_COMM_WORLD,N=...
integer::np,me,st,err
double precision::sbuf(N),rbuf(N)
  ...
  call MPI_Init(err)
  call MPI_Comm_size(MCW,np,err)
  call MPI_Comm_rank(MCW,me,err)
  ...
  call MPI_Send(sbuf,N,MPI_DOUBLE_PRECISION,&
                mod(me+1,np),0,MCW,err)
  ...
  call MPI_Recv(rbuf,N,MPI_DOUBLE_PRECISION,&
                mod(me+1,np),0,MCW,st,err)
  ...
  call MPI_Finalize(err)
end program
```

CのMPI関数の戻り値に相当

MPI 概論
Communicator と Rank



- MPI_COMM_WORLD
  = 全プロセスを含むcommunicator
- 必要なら別の communicator も作成可能
  - comm. ごとに固有の rank / size を持つ
  - comm. 内全員への集合通信など

MPI 概論
送受信データ型(基本型)

MPI	C	MPI	C
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short	MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long	MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float		
MPI_DOUBLE	double		

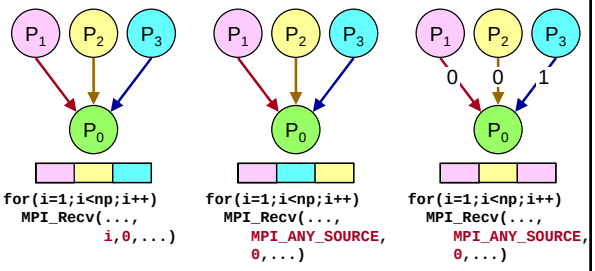
MPI	Fortran	MPI	Fortran
MPI_INTEGER	integer	MPI_DOUBLE_PRECISION	double precision
MPI_LOGICAL	logical	MPI_COMPLEX	complex
MPI_REAL	real	MPI_CHARACTER	charcter(1)

- Cでもポインタ型はない
  - プロセスごとに固有のメモリ空間
  - 異なるメモリ空間ではポインタ(アドレス)は無意味

MPI 概論
メッセージタグ (1)

- MPI_Send(sbuf,N,MPI_DOUBLE,him,0,MCW)
  call MPI_Send(sbuf,N,MPI_DOUBLE_PRECISION,&
  him,0,MCW,err)
- MPI_Recv(rbuf,N,MPI_DOUBLE,him,0,MCW,&st)
  call MPI_Recv(sbuf,N,MPI_DOUBLE_PRECISION,&
  him,0,MCW,st,err)
  - メッセージの種類を示す整数
    - ≒メールの subject
    - 種類によって適切なタグ付けをすると便利
  - 通常:送信者＝特定, タグ＝特定
  - 特殊な使い方
    - 送信者＝任意 (MPI_ANY_SOURCE) and/or
      タグ＝任意 (MPI_ANY_TAG)

MPI 概論
メッセージタグ (2)



MPI 概論
主要な一対一通信関数

（MPI_ は略）

- Send(*void sbuf, int cnt, MPI_Datatype type,
  int dst, int tag, MPI_Comm comm)
- Recv(*void rbuf, int cnt, MPI_Datatype type,
  int dst, int tag, MPI_Comm comm,
  MPI_Status *st)
- Get_count(MPI_Status *st, MPI_Datatype type,
  int *cnt)

実際の受信データ数取得

MPI 概論
主要な一対一通信関数

（MPI_ は略）

- Isend(*void sbuf, int cnt, MPI_Datatype type,
  int dst, int tag, MPI_Comm comm,
  MPI_Request *req)

非同期送信　完了確認用データ構造

- Irecv(*void rbuf, int cnt, MPI_Datatype type,
  int dst, int tag, MPI_Comm comm,
  MPI_Request *req)

非同期受信　完了確認用データ構造

- Wait(MPI_Request *req, MPI_Status *st)

完了確認

- Waitall(int n, MPI_Request *reqs,
  MPI_Status *sts)

n個の完了確認　完了確認用データの配列

- Sendrecv(*void sbuf, int scnt, MPI_Datatype stype,
  int dst, int stag, void *rbuf, int rcnt,
  MPI_Datatype rtype, int src, int rtag,
  MPI_Comm comm, MPI_Reuest *st)

安全な双方向＆循環通信

MPI 概論
主要な一対一通信関数

（Fortran版）

- Send(sbuf,cnt,type,dst,tag,comm,err)
- Recv(rbuf,cnt,type,dst,tag,comm,st,err)
- Get_count(st,type,cnt,err)
- Isend(sbuf,cnt,type,dst,tag,comm,req,err)
- Irecv(rbuf,cnt,type,dst,tag,comm,req,err)
- Wait(req,st,err)
- Waitall(n,reqs,sts,err)

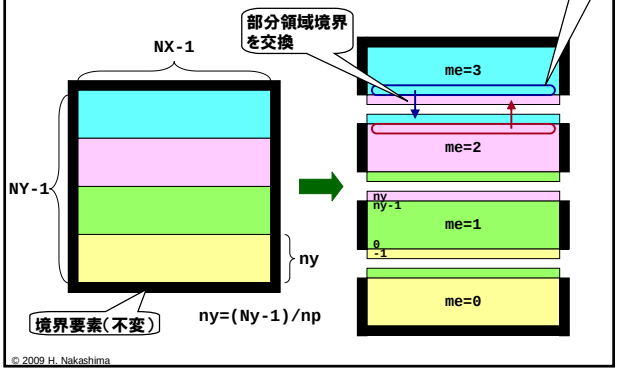
reqs,sts は n 要素(以上)の配列

- Sendrecv(sbuf,scnt,stype,dst,stag,&
  rbuf,rcnt,rtype,src,rtag,comm,st,err)

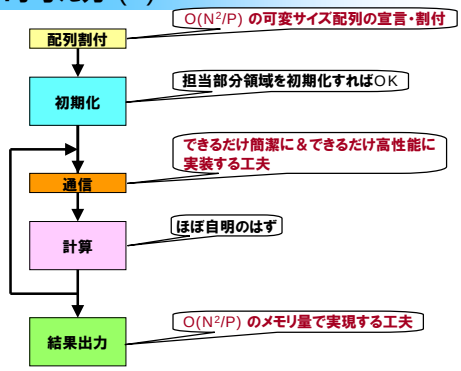
sbuf,rbuf は任意のデータ(配列)
他は全て integer(の配列)

1次元分割並列化:基本
基本的考え方 (1)

MPI_Send(&u[0][0],NX-1...)
call MPI_Send(u(0,0),NX-1...)



1次元分割並列化:基本
基本的考え方 (2)



1次元分割並列化:基本
可変サイズ配列の宣言＆割付 (1/3)

- 部分領域の大きさ (ny+2) はプロセス数に依存
  ➔固定サイズの配列を宣言・割付してはならない
- [X0,X1]×[Y0,y1] なる配列 a の宣言＆割付
  (X0, X1, Y0 は定数、y1は可変)
  - Fortran は簡単

```
double precision,allocatable::a(:,:)
allocate(X0:X1,Y0:y1)
```

  - C は結構面倒

```
double a[y1-Y0+1][X1-X0+1];
```

これは文法的に誤り

1次元分割並列化:基本
可変サイズ配列の宣言＆割付 (2/3)

- [X0,X1]×[Y0,y1] なる配列aの宣言＆割付in C
  - とりあえずプログラムを短くするために...

```
#define W (X1-X0+1)
int h=y1-Y0+1;
```

  - 以下はダメ←部分領域が連続しないので通信に困る

```
double **a;
a=(double**)malloc(h*sizeof(double*))-Y0;
for(i=Y0;i<=y1;i++)
  a[i]=malloc(double*)(W*sizeof(double))-X0;
```

  - 以下は良くない←a[j][i] の参照がやや遅い

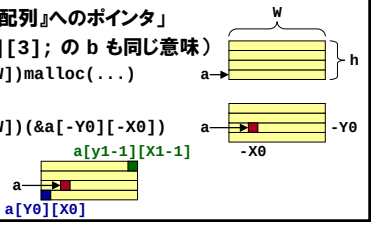
```
double *a[];
a=(double**)malloc(h*sizeof(double*))-Y0;
a[Y0]=(double*)malloc(h*W*sizeof(double))-X0;
for(j=Y0+1;j<=y1;j++) a[j]=a[j-1]+W;
```


1次元分割並列化:基本
可変サイズ配列の宣言＆割付 (3/3)

- [X0,X1]×[Y0,y1] なる配列aの宣言＆割付in C
  - 以下がベスト←a[j][i] の参照がやや速い

```
double (*a)[W];
a=(double(*)[W])malloc(h*W*sizeof(double*));
a=(double(*)[W])(&a[-Y0][-X0]);
```

- 少し解説
  - a は「『W要素の配列』へのポインタ」
    (double b[2][3]; の b も同じ意味)
  - a=(double(*)[W])malloc(...)
  - a=(double(*)[W])(&a[-Y0][-X0])



1次元分割並列化:基本
部分領域の交換 (1/2)

- 方法:(1) 一斉に北に送り (2) 一斉に南に送る
  - 以下はダメ←デッドロックはしないが大渋滞の可能性

```
int north=me+1, south=me-1;
if (north<np) MPI_Send(..., north, ...);
if (south>=0) MPI_Recv(..., south, ...);
if (south>=0) MPI_Send(..., south, ...);
if (north<np) MPI_Recv(..., north, ...);
```

  - MPI_Sendrecv() を使う安全なシフト

```
if (north>=np)    MPI_Recv(..., south, ...);
else if (south<0) MPI_Send(..., north, ...);
else MPI_Sendrecv(..., north, ..., south, ...);
if (south<0)        MPI_Recv(..., north, ...);
else if (north>=np) MPI_Send(..., south, ...);
else MPI_Sendrecv(..., south, ..., north, ...);
```

➔ 少し煩雑

1次元分割並列化:基本
部分領域の交換 (2/2)

- すっきりした方法: MPI_PROC_NULL の利用
  - C版

```
int north = me<np-2 ? me+1 : MPI_PROC_NULL;
int south = me>0    ? me-1 : MPI_PROC_NULL;
...
MPI_Sendrecv(..., north, ..., south, ...);
MPI_Sendrecv(..., south, ..., north, ...);
```

  - Fortran版

```
integer :: north,south
north=me+1;  south=me-1
if (north>=np) north=MPI_PROC_NULL
if (south<0)   south=MPI_PROC_NULL
...
call MPI_Sendrecv(..., north, ..., south, ...)
call MPI_Sendrecv(..., south, ..., north, ...)
```

send しても recv しても
何も起こらない null process

1次元分割並列化:基本
結果の出力 (1/5)

- 各プロセスが個別に出力する方法 (1)
  - 以下はダメ←同時に同じファイルに書いてはダメ

```
udata=fopen("...","w");
for(...) for(...) fprintf(udata,...);
```

    - ちなみに標準出力への同時出力は何とかしてくれる
      →デバッグ等に使ってもよい
    - ただし複数プロセスの出力はグチャグチャに混ざる(稀に行内でも)
      →行頭に rank を付記するとわかりやすい
  - これもダメ←同時に同じファイルを書込openしてはダメ

```
udata=fopen("...","w");
for(r=0;r<np;r++){
  MPI_Barrier(MCW);
  if(r==me) {
    for(...) for(...) fprintf(udata,...);
} }
```

全プロセスのバリア同期

1次元分割並列化:基本
結果の出力 (2/5)

- 各プロセスが個別に出力する方法 (2)
  - 以下は良くない←ファイルのopen/closeは遅い

```
for(r=0;r<np;r++){
  MPI_Barrier(MCW);
  if(r==me) {
    udata=fopen("...",me==0?"w":"a");
    for(...) for(...) fprintf(udata,...);
    fclose(udata);
} }
```

rank0 は書込open
他は追加書込open

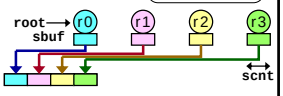
  - 個別出力の良い方法は明日のお楽しみ

1次元分割並列化:基本
結果の出力 (3/5)

- rank0 がまとめて出力する方法 (1)
  - 以下は良くない←N^2のメモリが必要

```
wbuf=(double(*)[NX+1])malloc((NY+1)*...)+...;
MPI_Gather(&u[...][...],ny*(NX+1),MPI_DOUBLE,
           wbuf,ny*(NX+1),MPI_DOUBLE,0,MCW);
if(me==0) {
  udata=fopen(...);
  for(...) for(...) fprintf(udata,...);
  fclose(udata);
}
```

（普通は）
scnt==rcnt
stype==rtype



  - 収集通信関数

```
MPI_Gather(void *sbuf, int scnt,
           MPI_Datatype stype,
           void *rbuf, int rcnt,
           MPI_Datatype rtype,
           int root, MPI_Comm comm)
```

Fortran では
最後に err を付加

1次元分割並列化:基本
結果の出力 (4/5)

- rank0 がまとめて出力する方法 (2: C版)
  - (とりあえず今日の)ベスト←N^2/Pのメモリで済む

```
if(me==0){
  udata=fopen("...", "w");
  for(...) for(...)
    fprintf(udata, ..., u[...][...]);
  for(r=1;r<np;r++) {
    MPI_Recv(&u[...][...], ..., r, ...);
    for(...) for(...)
      fprintf(updata, ..., u[...][...]);
  }
  fclose(udata);
} else
  MPI_Send(&u[...][...], ..., 0, ...);
```

rank0 だけがopen
まず自分の部分領域を出力
r=1 から順に
他人の部分領域
を受信
受信した部分領域を出力
rank 0 に部分領域を送信

  - 北側の境界要素について少し工夫が必要

1次元分割並列化:基本
結果の出力 (5/5)

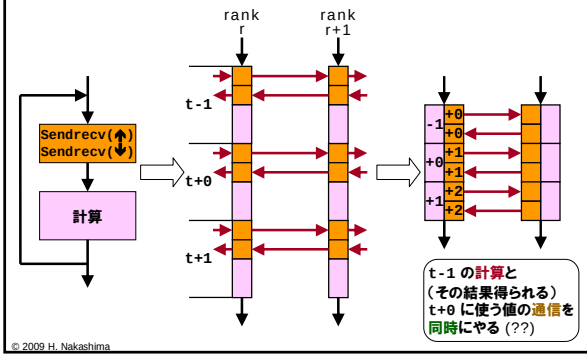
- rank0 がまとめて出力する方法 (2: Fortran版)
  - (とりあえず今日の)ベスト←N^2/Pのメモリで済む

```
if(me==0)then
  open(...)
  do ...; do ...
    write(...) ...,u(..., ...)
  end do; end do
  do r=1,np-1
    call MPI_Recv(u(..., ...), ..., r, ...)
    do ...; do ...
      write(...) ..., u(..., ...)
  end do; end do; end do
  close(...)
else
  call MPI_Send(u(..., ...), ..., 0, ...)
end if
```

rank0 だけがopen
まず自分の部分領域を出力
r=1 から順に
他人の部分領域
を受信
受信した部分領域を出力
rank 0 に部分領域を送信

1次元分割並列化:高速化
計算・通信のオーバーラップ (1/3)

- 基本的な考え方



1次元分割並列化:高速化
計算・通信のオーバーラップ (2/3)

- **計算と通信の同時実行方法 (C版)**

```
MPI_Request req[4];  MPI_Status st[4];
...
for(t=...) {
  MPI_Irecv(↑,...,&req[0]);
  MPI_Irecv(↓,...,&req[1]);
  /* 計算 part-1 */
  MPI_Isend(↑,...,&req[2]);
  MPI_Isend(↓,...,&req[3]);
  /* 計算 part-2 */
  MPI_Waitall(4,req,st);
}
```

次の時刻に必要な値を口を開けて待っておく

両隣が次の時刻に必要な値だけを先行して計算

両隣が次の時刻に必要な値の送信開始

次の時刻に自分だけが必要な値を計算。この間に両隣から次の時刻に必要な値が飛んでくるので…

両隣からの値が届いたことと両隣への値が飛んでいった(≠両隣へ届いた)ことを確認

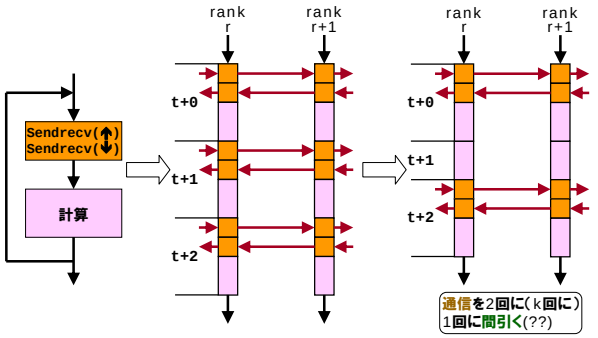
1次元分割並列化:高速化
計算・通信のオーバーラップ (3/3)

- **計算と通信の同時実行方法 (Fortran版)**

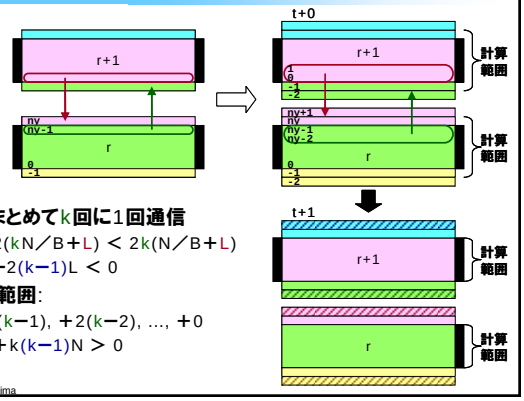
```
integer::req(4),st(4)
...
do t=...
  call MPI_Irecv(↑,...,req(1),err)
  call MPI_Irecv(↓,...,req(2),err)
  ! 計算 part-1
  call MPI_Isend(↑,...,req(3),err)
  call MPI_Isend(↓,...,req(4),err)
  ! 計算 part-2
  call MPI_Waitall(4,req,st,err)
end do
```


1次元分割並列化:高速化
通信回数削減 (1/2)

- **基本的な考え方**



1次元分割並列化:高速化
通信回数削減 (1/2)



- k行まとめてk回に1回通信
  - → 2(kN/B+L) < 2k(N/B+L)
  - → −2(k−1)L < 0
- 計算範囲:
  - +2(k−1), +2(k−2), ..., +0
  - → +k(k−1)N > 0

おまけ
実行時間計測

- **C版**

```
double time;
/* 初期化 */
MPI_Barrier(MCW);  time = MPI_Wtime();
for(t=...) {
  ...}
MPI_Barrier(MCW);  time = MPI_Wtime()-time;
/* 結果出力 */
```

一旦足並みを揃え

ある(不特定の)時刻を基準とした現在時刻

全員が終わるまでの時間を計測

- **Fortran版**

```
double precision :: time, MPI_Wtime
/* 初期化 */
call MPI_Barrier(MCW,err);  time = MPI_Wtime()
do t=...
  ...
end do
call MPI_Barrier(MCW,err);  time = MPI_Wtime()-time
/* 結果出力 */
```


おまけ
コンパイルと対話型実行

- **コンパイル**

```
% mpifcc -Kfast -o diffmpi diffmpi.c
% mpifrt -Kfast -o diffmpi diffmpi.f90
```

- **対話型実行**

```
% mpiexec -band 16 –n プロセス数 実行ファイル
```

たとえば

```
% mpiexec -band 16 –n 16 diffmpi
```

16プロセスずつまとめて計算ノード(16コア)に割り付ける

おまけ
バッチ実行

- ジョブスクリプト

```
# @$-eo
# @$-q sh20103     # または qh10160 その場合
#                  # @$-g qh10160 も必要
# @$-lp 1          # MPIでは(普通は)1
# @$-lP 16         # プロセス数と同じ
set -x
cd $QSUB_WORKDIR
mpiexec -n $QSUB_VNODE diffmpi
```

-lP で指定した値を保持する環境変数