

課題S1解説

2012年度冬学期
中島研吾

科学技術計算Ⅱ(4820-1028)・コンピュータ科学特別講義(4810-1205)
(並列有限要素法)

課題S1 (1/2)

- 内容

- 「<\$O-S1>/a1.0～a1.3」, 「 <\$O-S1>/a2.0～a2.3 」から局所ベクトル情報を読み込み, 全体ベクトルのノルム($\|x\|$)を求めるプログラムを作成する(S1-1).
 - <\$O-S1>file.f, <\$O-S1>file2.fをそれぞれ参考にする。
- 「<\$O-S1>/a2.0～a2.3」から局所ベクトル情報を読み込み, 「全体ベクトル」情報を各プロセッサに生成するプログラムを作成する。MPI_Allgathervを使用する(S1-2)。

課題S1 (2/2)

- 内容(続き)

- 下記の数値積分の結果を台形公式によって求めるプログラムを作成する。MPI_reduce, MPI_Bcast等を使用して並列化を実施し, プロセッサ数を変化させた場合の計算時間を測定する(S1-3)。

$$\int_0^1 \frac{4}{1+x^2} dx$$

ファイルコピー

Cユーザー

```
>$ cd <$0-fem2>  
>$ cp /home/z30088/fem2/s1r.tar .  
>$ tar xvf s1r.tar
```

ディレクトリ確認

```
>$ ls  
    mpi  
>$ cd mpi/S1-ref
```

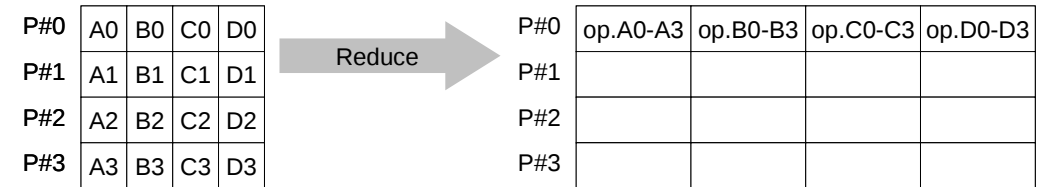
このディレクトリを本講義では **<\$0-S1r>** と呼ぶ。

<\$0-S1r> = <\$0-fem2>/mpi/S1-ref

S1-1: 局所ベクトル読み込み, ノルム計算

- 「<\$O-S1>/a1.0～a1.3」, 「<\$O-S1>/a2.0～a2.3」から局所ベクトル情報を読み込み, 全体ベクトルのノルム($\|x\|$)を求めるプログラムを作成する(S1-1)。
- MPI_Allreduce(またはMPI_Reduce)の利用
- ワンポイントアドバイス
 - 変数の中身を逐一確認しよう!

MPI_Reduce



- コミュニケーター「comm」内の、各プロセスの送信バッファ「sendbuf」について、演算「op」を実施し、その結果を1つの受信プロセス「root」の受信バッファ「recvbuf」に格納する。
 - 総和, 積, 最大, 最小 他
- **MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)**
 - sendbuf 任意 I 送信バッファの先頭アドレス,
 - recvbuf 任意 0 受信バッファの先頭アドレス,
タイプは「datatype」により決定
 - count 整数 I メッセージのサイズ
 - datatype MPI_Datatype I メッセージのデータタイプ
FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
 - op MPI_Op I 計算の種類
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
ユーザーによる定義も可能: MPI_OP_CREATE
 - root 整数 I 受信元プロセスのID(ランク)
 - comm MPI_Comm I コミュニケータを指定する

送信バッファと受信バッファ

- MPIでは「送信バッファ」,「受信バッファ」という変数がしばしば登場する。
- 送信バッファと受信バッファは必ずしも異なった名称の配列である必要はないが, 必ずアドレスが異なっていなければならない。

MPI_Reduce/Allreduceの“op”

MPI_Reduce

(sendbuf, recvbuf, count, datatype, op, root, comm)

- MPI_MAX, MPI_MIN 最大値, 最小値
- MPI_SUM, MPI_PROD 総和, 積
- MPI_LAND 論理AND

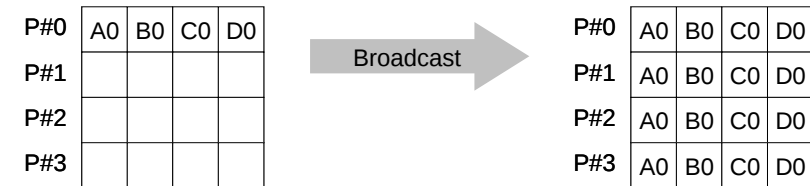
```
double X0, XSUM;
```

```
MPI_Reduce  
(&X0, &XSUM, 1, MPI_DOUBLE, MPI_SUM, 0, <comm>)
```

```
double X0[4];
```

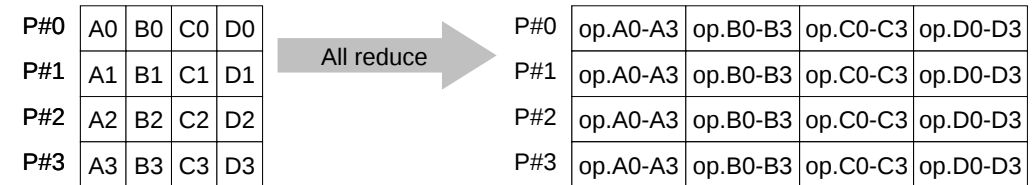
```
MPI_Reduce  
(&X0[0], &X0[2], 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>)
```

MPI_Bcast



- コミュニケーター「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から、その他全てのプロセスのバッファ「buffer」にメッセージを送信。
- **MPI_Bcast (buffer, count, datatype, root, comm)**
 - **buffer** 任意 I/O バッファの先頭アドレス,
タイプは「datatype」により決定
 - **count** 整数 I メッセージのサイズ
 - **datatype** MPI_Datatype I メッセージのデータタイプ
 FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
 C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
 - **root** 整数 I 送信元プロセスのID(ランク)
 - **comm** MPI_Comm I コミュニケータを指定する

MPI_Allreduce



- MPI_Reduce + MPI_Bcast
- 総和, 最大値を計算したら, 各プロセスで利用したい場合が多い

- MPI_Allreduce

(sendbuf, recvbuf, count, datatype, op, comm)

- sendbuf 任意 I 送信バッファの先頭アドレス,
- recvbuf 任意 0 受信バッファの先頭アドレス,
タイプは「datatype」により決定
- count 整数 I メッセージのサイズ
- datatype MPI_Datatype I メッセージのデータタイプ
- op MPI_Op I 計算の種類
- comm MPI_Comm I コミュニケータを指定する

S1-1: 局所ベクトル読み込み, ノルム計算

均一長さベクトルの場合 (a1.*): s1-1-for_a1.c

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
    int i, N;
    int PeTot, MyRank;
    MPI_Comm SolverComm;
    double vec[8];
    double sum0, sum;
    char filename[80];
    FILE *fp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

    sprintf(filename, "a1.%d", MyRank);
    fp = fopen(filename, "r");
    assert(fp != NULL);

    N=8;
    for(i=0;i<N;i++){
        fscanf(fp, "%lf", &vec[i]);
    }
    sum0 = 0.0;
    for(i=0;i<N;i++){
        sum0 += vec[i] * vec[i];
    }

    MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    sum = sqrt(sum);

    if(!MyRank) printf("%27.20E\n", sum);
    MPI_Finalize();
    return 0;
}
```

S1-1: 局所ベクトル読み込み, ノルム計算

不均一長さベクトルの場合 (a2.*): s1-1-for_a2.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
    int i, PeTot, MyRank, n;
    MPI_Comm SolverComm;
    double *vec, *vec2;
    int * Count, CountIndex;
    double sum0, sum;
    char filename[80];
    FILE *fp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

    sprintf(filename, "a2.%d", MyRank);
    fp = fopen(filename, "r");
    assert(fp != NULL);

    fscanf(fp, "%d", &n);
    vec = malloc(n * sizeof(double));
    for(i=0; i<n; i++){
        fscanf(fp, "%lf", &vec[i]);
    }
    sum0 = 0.0;
    for(i=0; i<n; i++){
        sum0 += vec[i] * vec[i];
    }

    MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    sum = sqrt(sum);

    if(!MyRank) printf("%27.20E\n", sum);
    MPI_Finalize();
    return 0;}
```

実行(課題S1-1)

```
$ cd <$0-S1r>  
$ mpifccpx -Kfast s1-1-for_a1.c  
$ mpifccpx -Kfast s1-1-for_a2.c  
  
(modify "go4.sh")  
$ pjsub go4.sh
```

S1-1: 局所ベクトル読み込み, ノルム計算 計算結果

予め求めておいた答え

```
a1.* 1.62088247569032590000E+03  
a2.* 1.22218492872396360000E+03
```

```
$> frtpx -Kfast dot-a1.f  
$> pjsub go1.sh
```

```
$> frtpx -Kfast dot-a2.f  
$> pjsub go1.sh
```

計算結果

```
a1.* 1.62088247569032590000E+03  
a2.* 1.22218492872396360000E+03
```

go1.sh

```
#!/bin/sh  
#PJM -L "node=1"  
#PJM -L "elapsed=00:10:00"  
#PJM -L "rscgrp=lecture"  
#PJM -g "gt61"  
#PJM -j  
#PJM -o "test.lst"  
#PJM --mpi "proc=1"  
  
mpiexec ./a.out
```

S1-1: 局所ベクトル読み込み, ノルム計算

SENDBUFとRECVBUFを同じにしたら...

正

```
MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM,  
              MPI_COMM_WORLD)
```

誤

```
MPI_Allreduce(&sum0, &sum0, 1, MPI_DOUBLE, MPI_SUM,  
              MPI_COMM_WORLD)
```

S1-1: 局所ベクトル読み込み, ノルム計算

SENDBUFとRECVBUFを同じにしたら...

正

```
MPI_Allreduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM,  
              MPI_COMM_WORLD)
```

誤

```
MPI_Allreduce(&sum0, &sum0, 1, MPI_DOUBLE, MPI_SUM,  
              MPI_COMM_WORLD)
```

正

```
MPI_Allreduce(&sumK[1], &sumK[2], 1, MPI_DOUBLE, MPI_SUM,  
              MPI_COMM_WORLD)
```

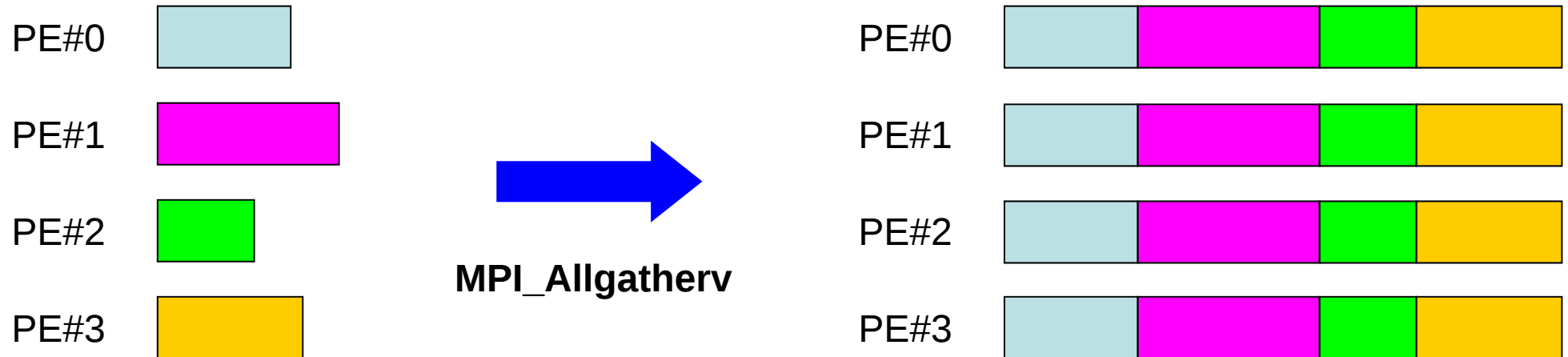
これバッファが重なっていないのでOK

S1-2: 局所ベクトルから全体ベクトル生成

- 「<\$O-S1>/a2.0~a2.3」から局所ベクトル情報を読み込み, 「全体ベクトル」情報を各プロセッサに生成するプログラムを作成する。MPI_Allgathervを使用する。

S1-2: 局所ベクトルから全体ベクトル生成

MPI_Allgathervを使う場合 (1/5)



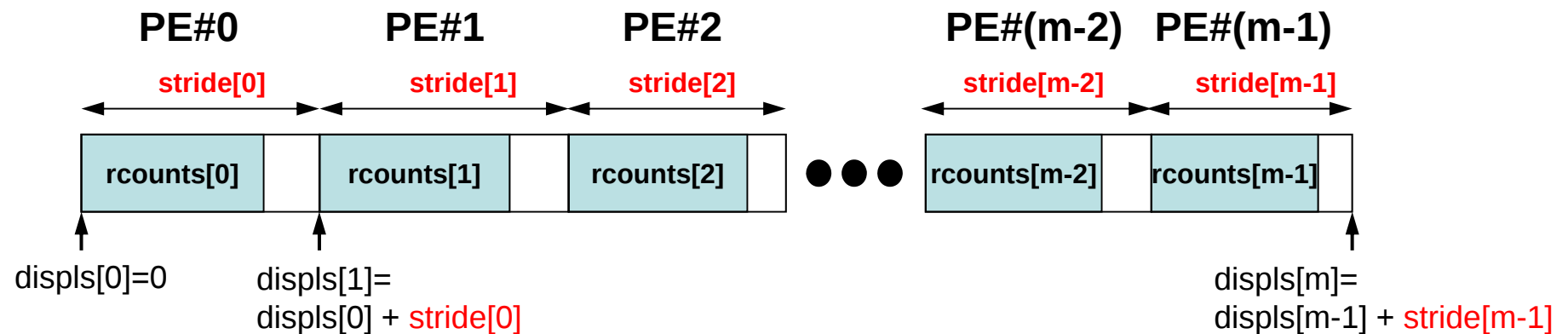
MPI_Allgather

- MPI_Allgather の可変長さベクトル版
 - 「局所データ」から「全体データ」を生成する
- MPI_Allgather (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvttype, comm)

– <u>sendbuf</u>	任意	I	送信バッファの先頭アドレス,
– <u>scount</u>	整数	I	送信メッセージのサイズ
– <u>sendtype</u>	MPI_Datatype	I	送信メッセージのデータタイプ
– <u>recvbuf</u>	任意	0	受信バッファの先頭アドレス,
– <u>rcounts</u>	整数	I	受信メッセージのサイズ(配列: サイズ=PETOT)
– <u>displs</u>	整数	I	受信メッセージのインデックス(配列: サイズ=PETOT+1)
– <u>recvttype</u>	MPI_Datatype	I	受信メッセージのデータタイプ
– <u>comm</u>	MPI_Comm	I	コミュニケータを指定する

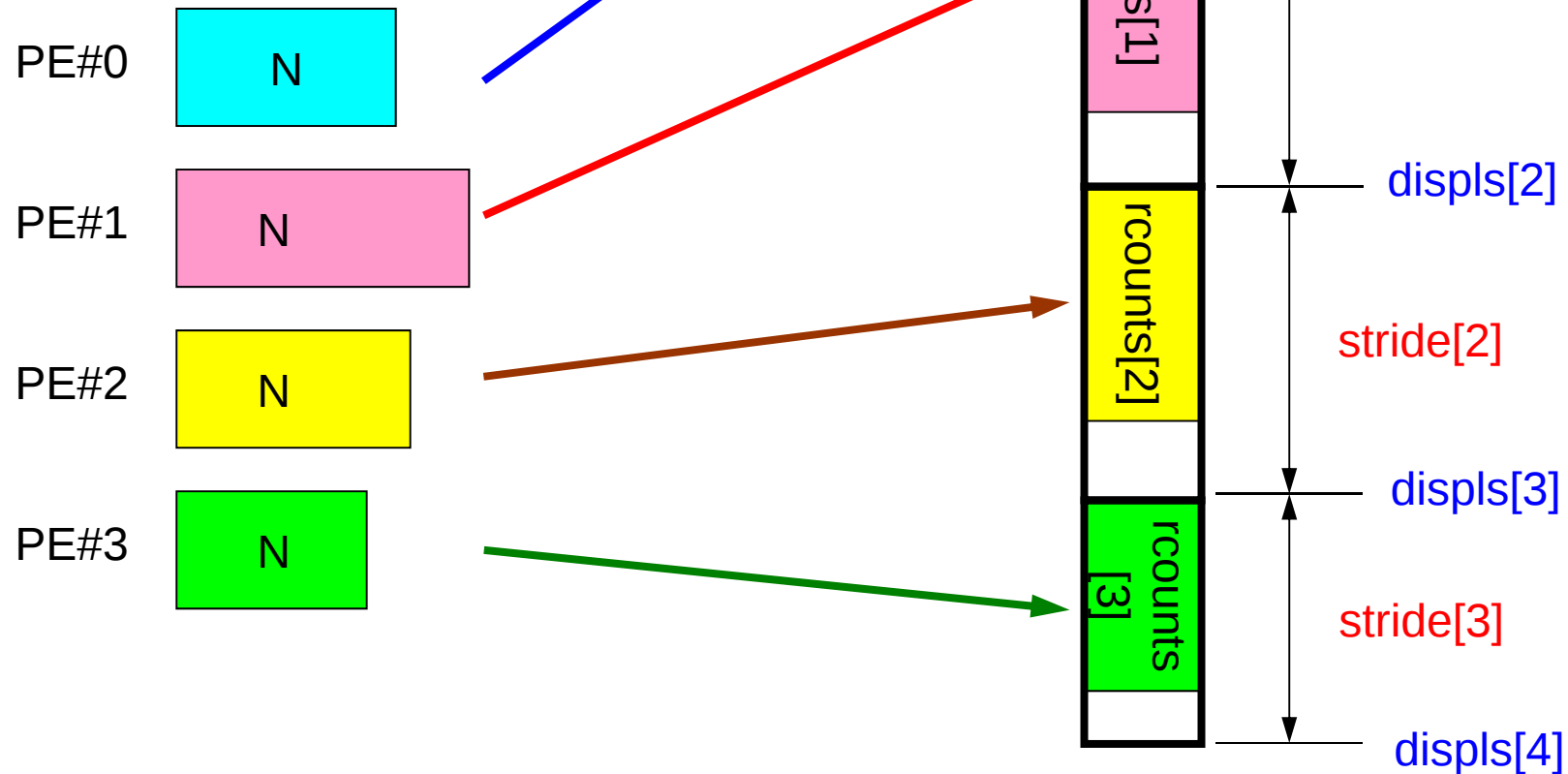
MPI_Allgatherv(続き)

- `MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)`
 - `rcounts` 整数 I 受信メッセージのサイズ(配列:サイズ=PETOT)
 - `displs` 整数 I 受信メッセージのインデックス(配列:サイズ=PETOT+1)
 - この2つの配列は、最終的に生成される「全体データ」のサイズに関する配列であるため、各プロセスで配列の全ての値が必要になる:
 - もちろん各プロセスで共通の値を持つ必要がある。
 - 通常は $\text{stride}(i) = \text{rcounts}(i)$



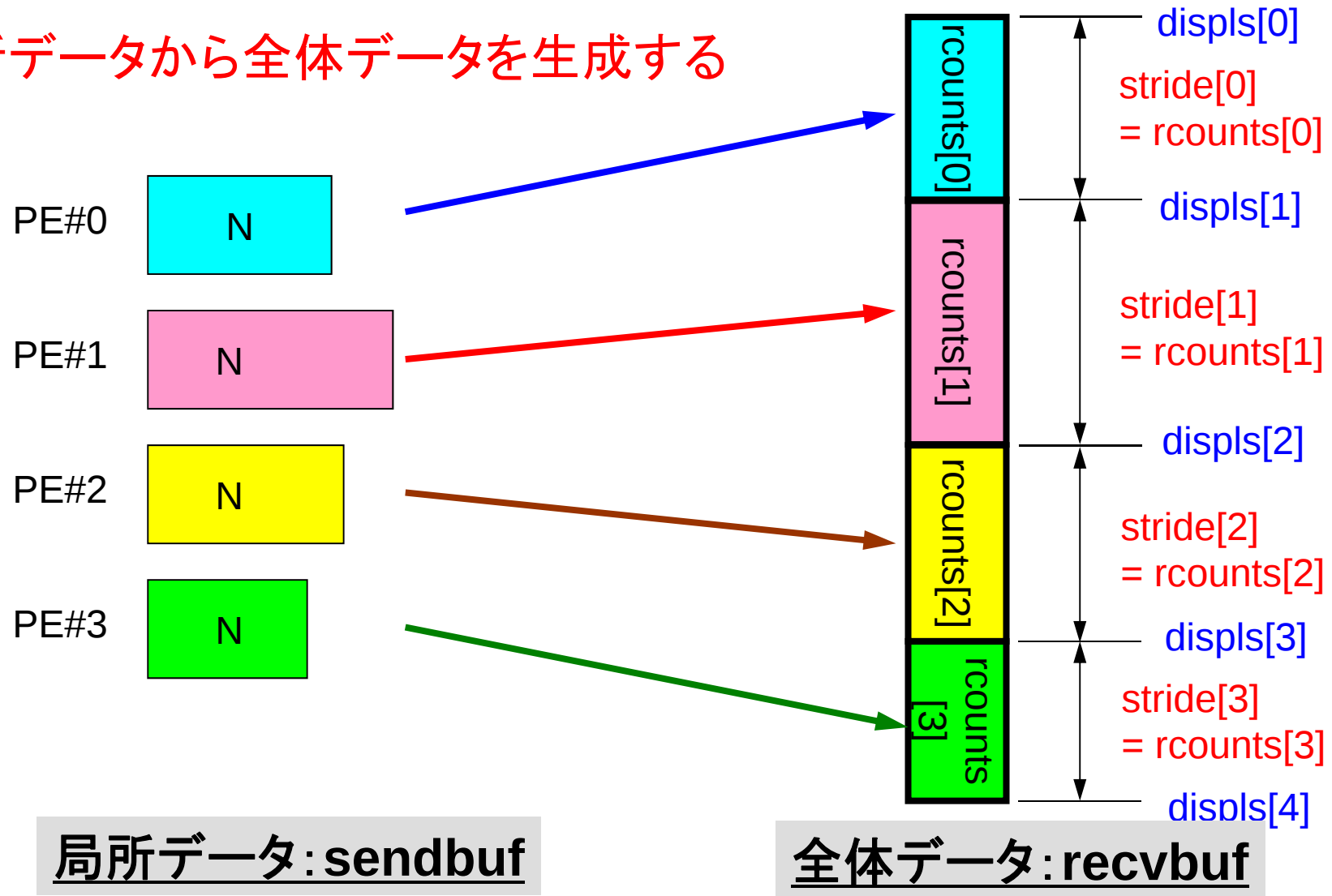
MPI_Allgatherv でやっていること

局所データから全体データを
生成する



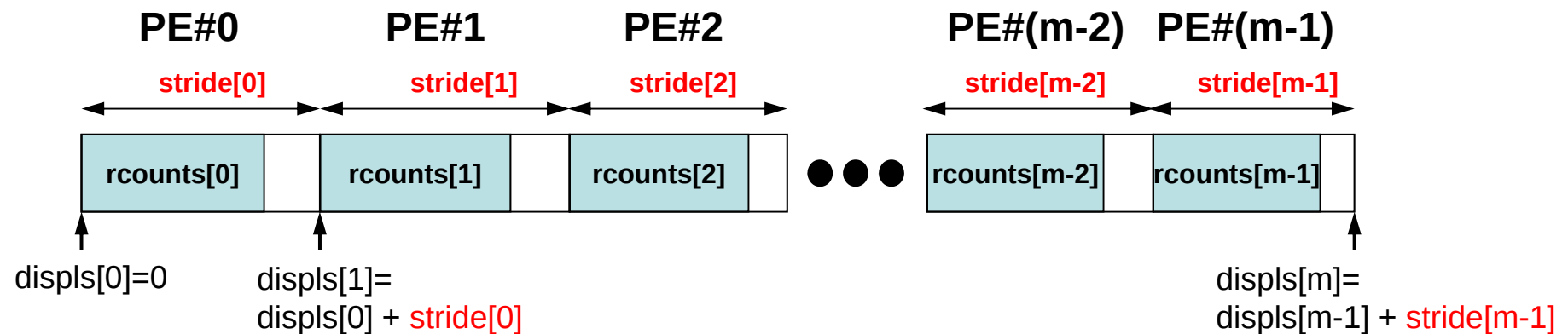
MPI_Allgatherv でやっていること

局所データから全体データを生成する



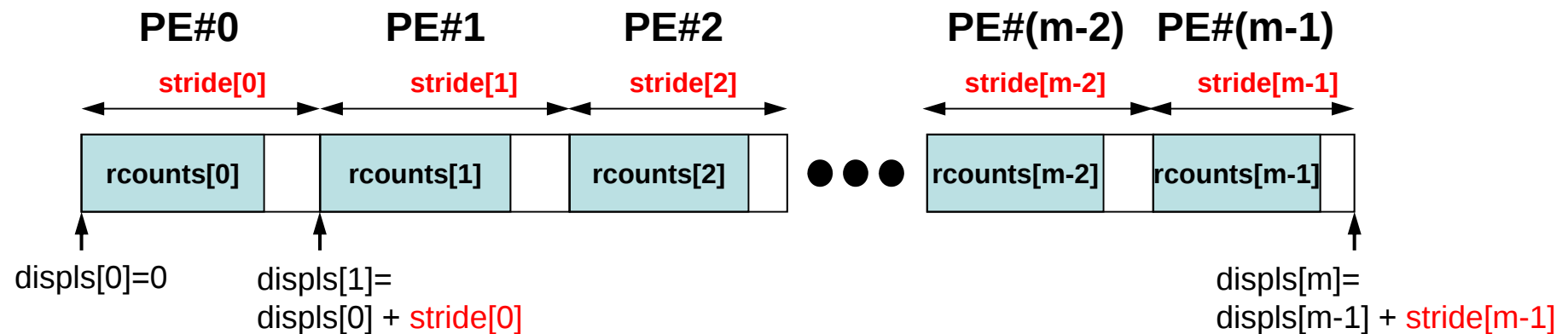
MPI_Allgatherv詳細(1/2)

- `MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)`
 - `rcounts` 整数 I 受信メッセージのサイズ(配列: サイズ=PETOT)
 - `displs` 整数 I 受信メッセージのインデックス(配列: サイズ=PETOT+1)
- **`rcounts`**
 - 各PEにおけるメッセージサイズ: 局所データのサイズ
- **`displs`**
 - 各局所データの全体データにおけるインデックス
 - `displs(PETOT+1)`が全体データのサイズ



MPI_Allgatherv詳細(2/2)

- **rcounts**と**displs**は各プロセスで共通の値が必要
 - 各プロセスのベクトルの大きさ N を**allgather**して, **rcounts**に相当するベクトルを作る。
 - **rcounts**から各プロセスにおいて**displs**を作る(同じものができる)。
 - $\text{stride}[i] = \text{rcounts}[i]$ とする
 - **rcounts**の和にしたがって**recvbuf**の記憶領域を確保する。



MPI_Allgatherv使用準備

例題:<\$O-S1>/agv.c

- “a2.0”～”a2.3”から, 全体ベクトルを生成する。
- 各ファイルのベクトルのサイズが, 8,5,7,3であるから, 長さ23(=8+5+7+3)のベクトルができることになる。

a2.0~a2.3

PE#0

8
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0

PE#1

5
201.0
203.0
205.0
206.0
209.0

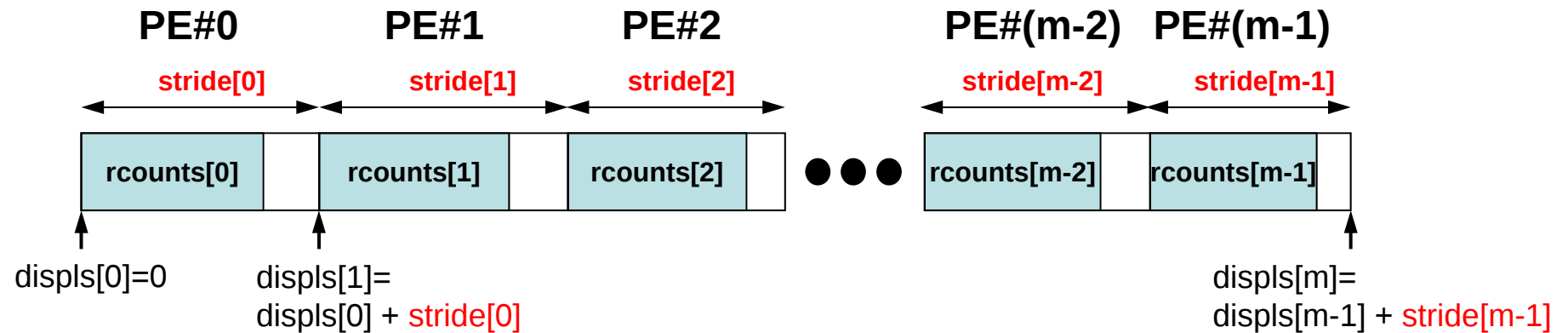
PE#2

7
301.0
303.0
305.0
306.0
311.0
321.0
351.0

PE#3

3
401.0
403.0
405.0

S1-2: 局所⇒全体ベクトル生成: 手順



`size[recvbuf]= displs[PETOT]= sum[stride]`

- 局所ベクトル情報を読み込む
- 「rcounts」, 「displs」を作成する
- 「recvbuf」を準備する
- Allgatherv

S1-2: 局所⇒全体ベクトル生成(1/2)

s1-2.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
    int i, PeTot, MyRank, n;
    MPI_Comm SolverComm;
    double *vec, *vec2, *vecg;
    int *Rcounts, *Displs;
    double sum0, sum;
    char filename[80];
    FILE *fp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

    sprintf(filename, "a2.%d", MyRank);
    fp = fopen(filename, "r");
    assert(fp != NULL);

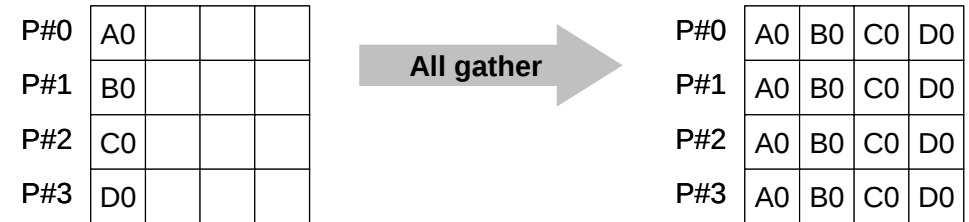
    fscanf(fp, "%d", &n);
    vec = malloc(n * sizeof(double));
    for(i=0; i<n; i++){
        fscanf(fp, "%lf", &vec[i]);
    }

    Rcounts = calloc(PeTot, sizeof(int));
    Displs = calloc(PeTot+1, sizeof(int));

    MPI_Allgather(&n, 1, MPI_INT, Rcounts, 1, MPI_INT, MPI_COMM_WORLD);
```

各PEにおけるベクトル長さの情報が「Rcounts」に入る

MPI_Allgather



- MPI_Allgather = MPI_Gather + MPI_Bcast
- MPI_Allgather (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, comm)
 - sendbuf 任意 I 送信バッファの先頭アドレス,
 - scount 整数 I 送信メッセージのサイズ
 - sendtype MPI_Datatype I 送信メッセージのデータタイプ
 - recvbuf 任意 0 受信バッファの先頭アドレス,
 - rcount 整数 I 受信メッセージのサイズ
 - recvtype MPI_Datatype I 受信メッセージのデータタイプ
 - comm MPI_Comm I コミュニケータを指定する

S1-2: 局所⇒全体ベクトル生成 (2/2)

s1-2.c

```

Displs[0]=0;
for(i=0;i<PeTot;i++){
    Displs[i+1] = Displs[i] + Rcounts[i];
}

vecg = calloc(Displs[PeTot], sizeof(double));

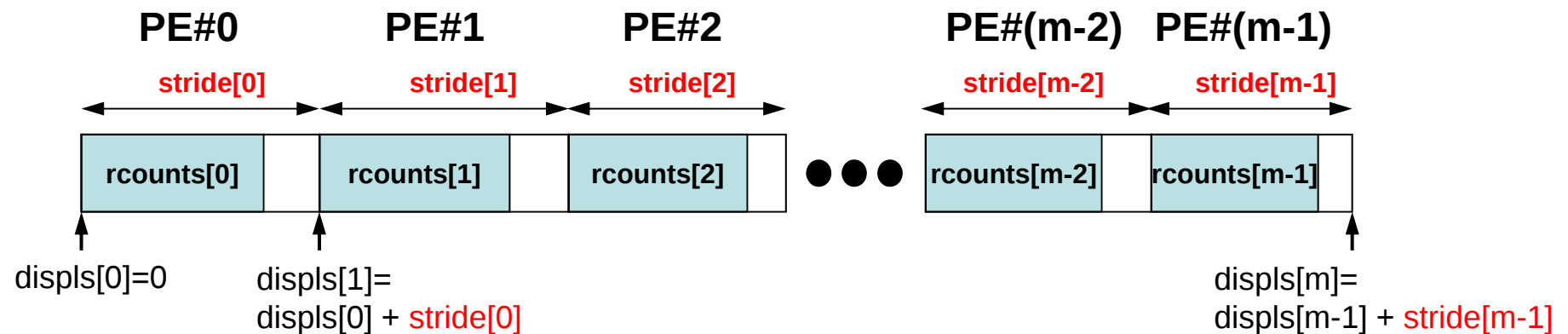
MPI_Allgatherv(vec, n, MPI_DOUBLE, vecg, Rcounts, Displs, MPI_DOUBLE, MPI_COMM_WORLD);

for(i=0;i<Displs[PeTot];i++){
    printf("%8.2f", vecg[i]);
}
printf("\n");

MPI_Finalize();
return 0;
}

```

「Displs」生成



`size[recvbuf] = displs[PETOT] = sum[stride]`

S1-2: 局所⇒全体ベクトル生成 (2/2)

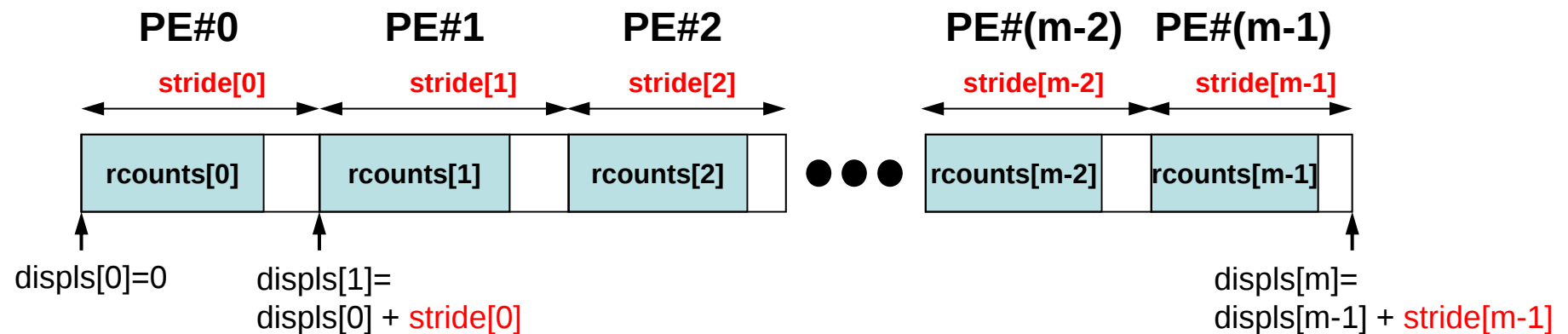
s1-2.c

```

Displs[0]=0;
for(i=0;i<PeTot;i++){
    Displs[i+1] = Displs[i] + Rcounts[i];
}
vecg = calloc(Displs[PeTot], sizeof(double));
MPI_Allgatherv(vec, n, MPI_DOUBLE, vecg, Rcounts, Displs, MPI_DOUBLE, MPI_COMM_WORLD);
for(i=0;i<Displs[PeTot];i++){
    printf("%8.2f", vecg[i]);
}
printf("\n");
MPI_Finalize();
return 0;
}

```

「recvbuf」



$\text{size}[\text{recvbuf}] = \text{displs}[\text{PETOT}] = \text{sum}[\text{stride}]$

S1-2: 局所⇒全体ベクトル生成 (2/2)

s1-2.c

```

Displs[0]=0;
for(i=0;i<PeTot;i++){
    Displs[i+1] = Displs[i] + Rcounts[i];
}

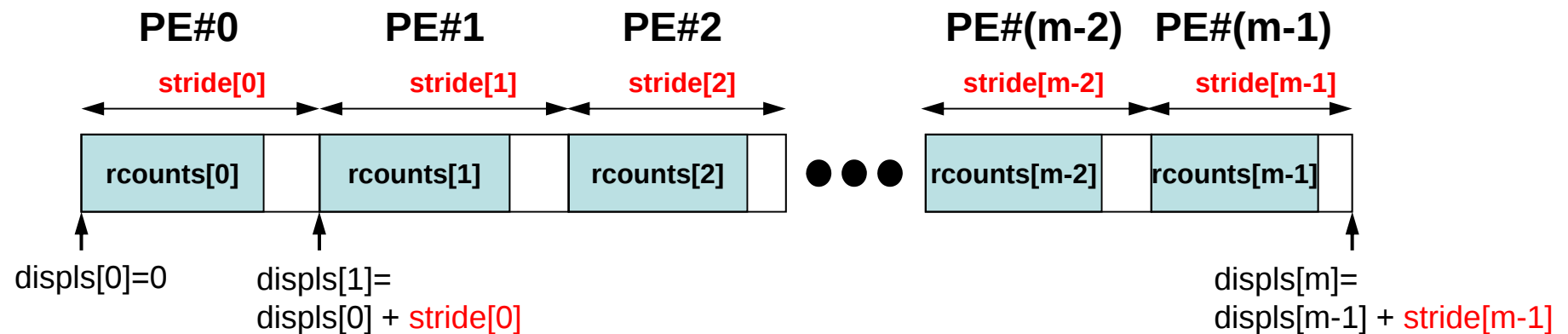
vecg = calloc(Displs[PeTot], sizeof(double));

MPI_Allgatherv(vec, n, MPI_DOUBLE, vecg, Rcounts, Displs, MPI_DOUBLE, MPI_COMM_WORLD);

for(i=0;i<Displs[PeTot];i++){
    printf("%8.2f", vecg[i]);
}
printf("\n");

MPI_Finalize();
return 0;
}

```



実行(課題S1-2)

```
$ mpifccpx -Kfast s1-2.c
```

```
(modify "go4.sh")
```

```
$ pjsub go4.sh
```

S1-2: 結果

my_rank	ID	VAL
0	1	101.
0	2	103.
0	3	105.
0	4	106.
0	5	109.
0	6	111.
0	7	121.
0	8	151.
0	9	201.
0	10	203.
0	11	205.
0	12	206.
0	13	209.
0	14	301.
0	15	303.
0	16	305.
0	17	306.
0	18	311.
0	19	321.
0	20	351.
0	21	401.
0	22	403.
0	23	405.

my_rank	ID	VAL
1	1	101.
1	2	103.
1	3	105.
1	4	106.
1	5	109.
1	6	111.
1	7	121.
1	8	151.
1	9	201.
1	10	203.
1	11	205.
1	12	206.
1	13	209.
1	14	301.
1	15	303.
1	16	305.
1	17	306.
1	18	311.
1	19	321.
1	20	351.
1	21	401.
1	22	403.
1	23	405.

my_rank	ID	VAL
2	1	101.
2	2	103.
2	3	105.
2	4	106.
2	5	109.
2	6	111.
2	7	121.
2	8	151.
2	9	201.
2	10	203.
2	11	205.
2	12	206.
2	13	209.
2	14	301.
2	15	303.
2	16	305.
2	17	306.
2	18	311.
2	19	321.
2	20	351.
2	21	401.
2	22	403.
2	23	405.

my_rank	ID	VAL
3	1	101.
3	2	103.
3	3	105.
3	4	106.
3	5	109.
3	6	111.
3	7	121.
3	8	151.
3	9	201.
3	10	203.
3	11	205.
3	12	206.
3	13	209.
3	14	301.
3	15	303.
3	16	305.
3	17	306.
3	18	311.
3	19	321.
3	20	351.
3	21	401.
3	22	403.
3	23	405.

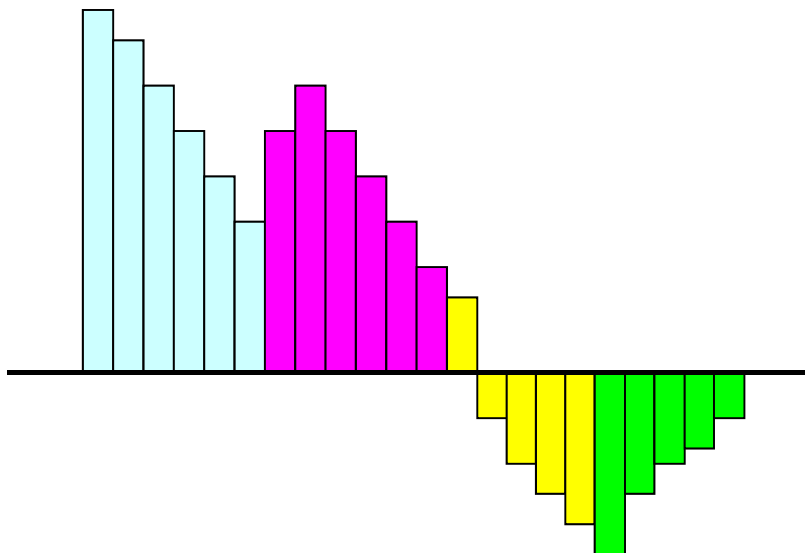
S1-3: 台形則による積分

- 下記の数値積分の結果を台形公式によって求めるプログラムを作成する。MPI_REDUCE, MPI_BCASTを使用して並列化を実施し, プロセッサ数を変化させた場合の計算時間を測定する。

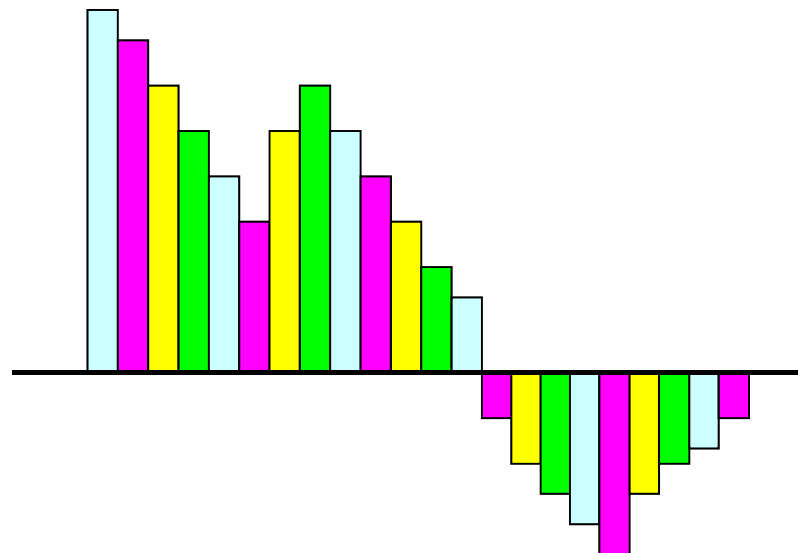
$$\int_0^1 \frac{4}{1+x^2} dx$$

S1-3: 台形則による積分 プロセッサへの配分の手法

タイプA



タイプB



$\frac{1}{2} \Delta x \left(f_1 + f_{N+1} + \sum_{i=2}^N 2 f_i \right)$ を使うとすると必然的に「タイプA」となるが...

S1-3: 台形則による計算

TYPE-A(1/2): s1-3a.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "mpi.h"

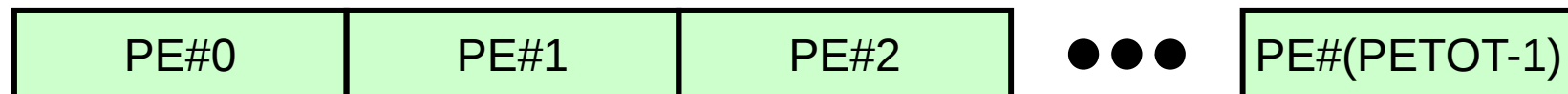
int main(int argc, char **argv){
    int i;
    double TimeStart, TimeEnd, sum0, sum, dx;
    int PeTot, MyRank, n, int *index;
    FILE *fp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
    MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

    index = calloc(PeTot+1, sizeof(int));
    fp = fopen("input.dat", "r");
    fscanf(fp, "%d", &n);
    fclose(fp);
    if(MyRank==0) printf("%s%8d¥n", "N=", n);
    dx = 1.0/n;

    for(i=0; i<=PeTot; i++){
        index[i] = ((long long)i * n)/PeTot;
    }
}
```

“input.dat”で分割数Nを指定
中身を書き出して見よう:n



index[0]

index[1]

index[2]

index[3]

index[PETOT-1]

index[PeTot]
=N

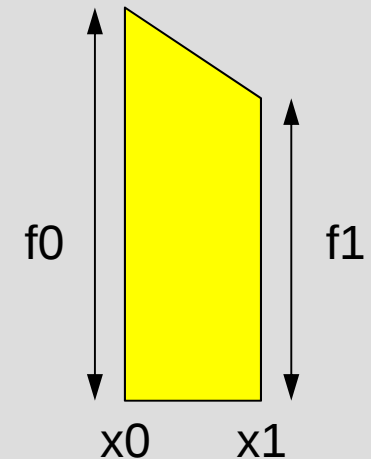
S1-3: 台形則による計算

TYPE-A(2/2): s1-3a.c

```

Times = MPI_Wtime();
sum0 = 0.0;
for(i=index[MyRank]; i<index[MyRank+1]; i++)
{
    double x0, x1, f0, f1;
    x0 = (double)i * dx;
    x1 = (double)(i+1) * dx;
    f0 = 4.0/(1.0+x0*x0);
    f1 = 4.0/(1.0+x1*x1);
    sum0 += 0.5 * (f0 + f1) * dx;
}

```



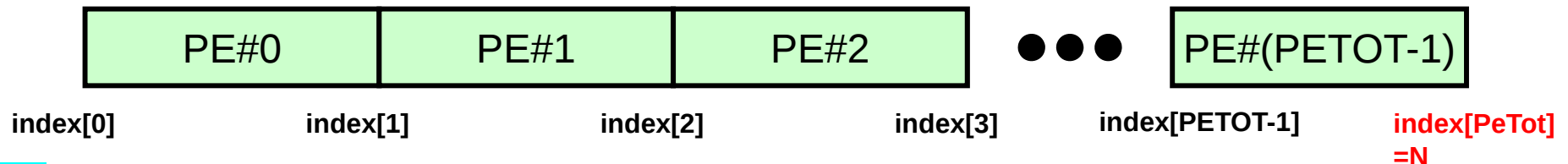
```

MPI_Reduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
TimeE = MPI_Wtime();

if(!MyRank) printf("%24.16f%24.16f%24.16f\n", sum, 4.0*atan(1.0), TimeE - Times);

MPI_Finalize();
return 0;
}

```



S1-3: 台形則による計算

TYPE-B : s1-3b.c

```

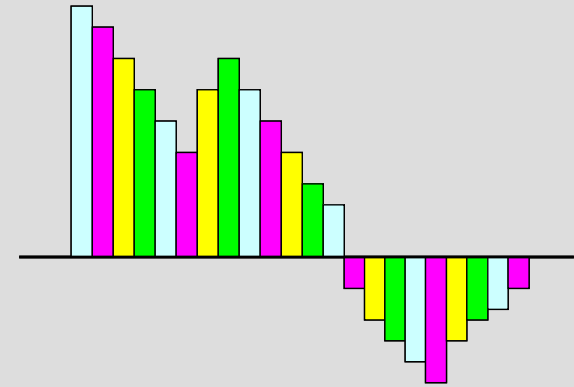
Times = MPI_Wtime();
sum0 = 0.0;
for(i=MyRank; i<n; i+=PeTot)
{
    double x0, x1, f0, f1;
    x0 = (double)i * dx;
    x1 = (double)(i+1) * dx;
    f0 = 4.0/(1.0+x0*x0);
    f1 = 4.0/(1.0+x1*x1);
    sum0 += 0.5 * (f0 + f1) * dx;
}

MPI_Reduce(&sum0, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
TimeE = MPI_Wtime();

if(!MyRank) printf("%24.16f%24.16f%24.16f\n", sum, 4.0*atan(1.0), TimeE-Times);

MPI_Finalize();
return 0;
}

```



コンパイル・実行(課題S1-3)

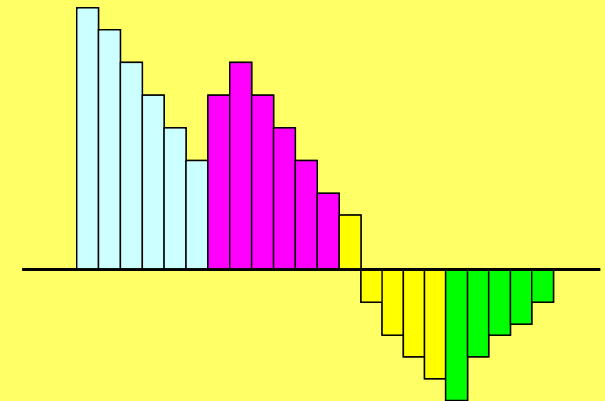
```
$ mpifccpx -Kfast s1-3a.c
```

```
$ mpifccpx -Kfast s1-3b.c
```

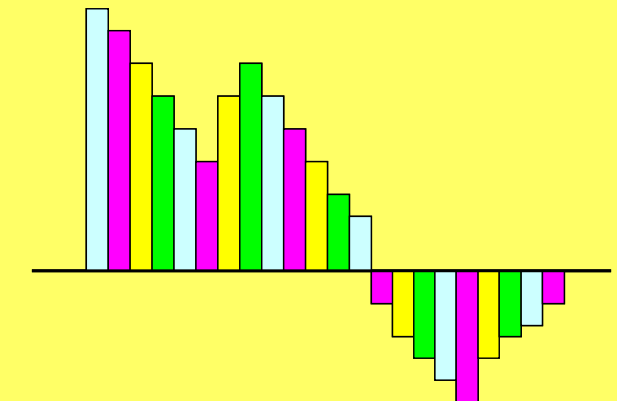
```
(modify "go.sh")
```

```
$ pjsub go.sh
```

タイプA



タイプB



go.sh

```
#!/bin/sh
```

```
#PJM -L "node=1"
```

ノード数 (≤ 12)

```
#PJM -L "elapsed=00:10:00"
```

実行時間 (≤ 15 分)

```
#PJM -L "rscgrp=lecture"
```

実行キュー名 (lecture4)

```
#PJM -g "gt64"
```

「財布」変更不可

```
#PJM -
```

```
#PJM -o "test.lst"
```

標準出力

```
#PJM --mpi "proc=8"
```

MPIプロセス数 (≤ 192)

```
mpiexec ./a.out
```

8分割

"node=1"

"proc=8"

16分割

"node=1"

"proc=16"

32分割

"node=2"

"proc=32"

64分割

"node=4"

"proc=64"

192分割

"node=12"

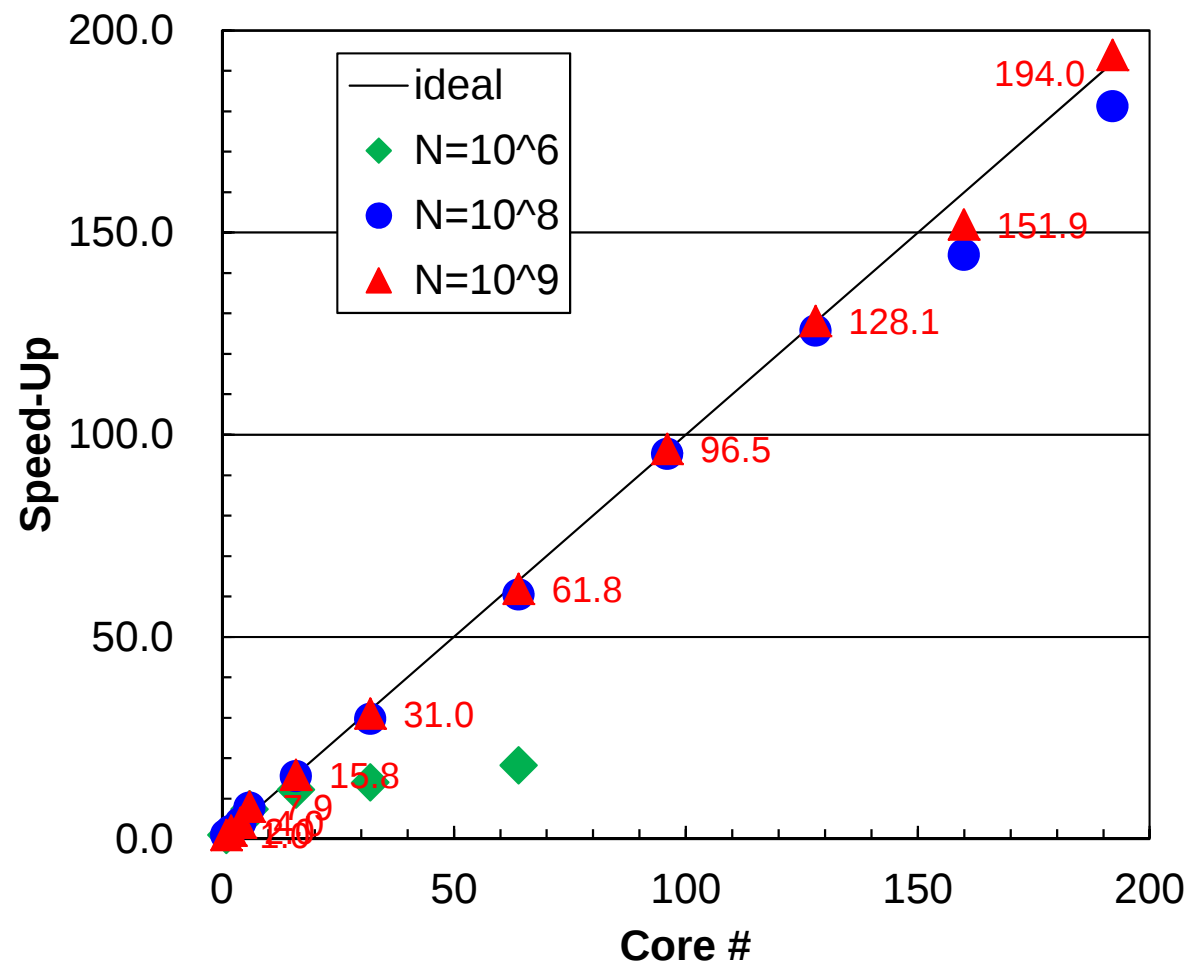
"proc=192"

S1-3: Oakleaf-FXにおける並列効果

- ◆ : $N=10^6$, ● : 10^8 , ▲ : 10^9 , — : 理想値
- 1コアにおける計測結果(sec.)からそれぞれ算出

- Strong Scaling
 - 全体問題規模固定
 - N倍のコアを使ってN分の1の計算時間で解けるのが理想

- Weak Scaling
 - ノード当たり(コア当たり)問題規模固定
 - N倍のコアを使ってN倍の規模の問題を同じ時間で解けるのが理想



理想値からのずれ

- MPI通信そのものに要する時間
 - データを送付している時間
 - ノード間においては通信バンド幅によって決まる
 - Gigabit Ethernetでは 1Gbit/sec.(理想値)
 - 通信時間は送受信バッファのサイズに比例
- MPIの立ち上がり時間
 - latency
 - 送受信バッファのサイズによらない
 - 呼び出し回数依存, プロセス数が増加すると増加する傾向
 - 通常, 数～数十 μ secのオーダー
- MPIの同期のための時間
 - プロセス数が増加すると増加する傾向

理想値からのずれ(続き)

- 計算時間が小さい場合(S1-3ではNが小さい場合)はこれらの効果を見視できない。
 - 特に, 送信メッセージ数が小さい場合は, 「Latency」が効く。