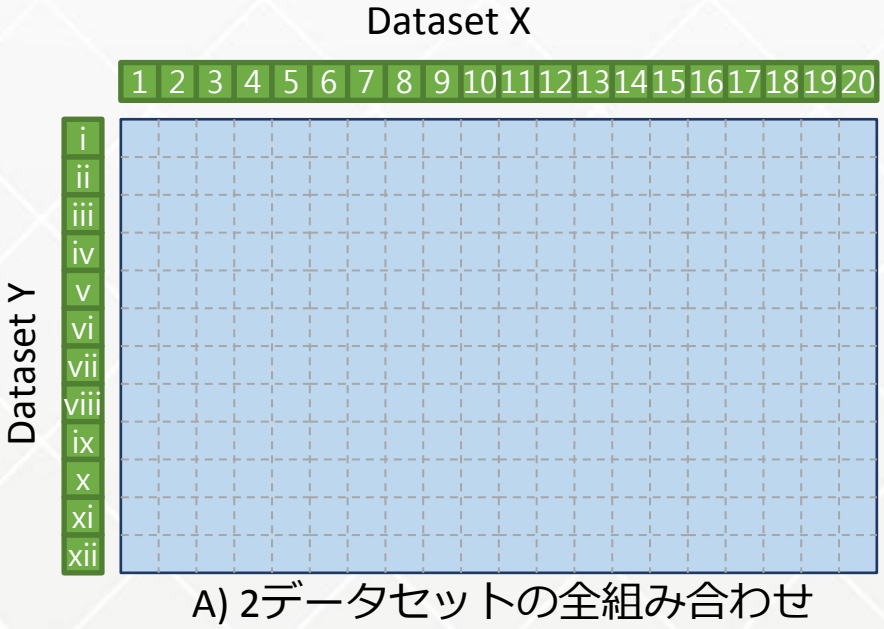
Carp(pCarp,sCarp)利用説明

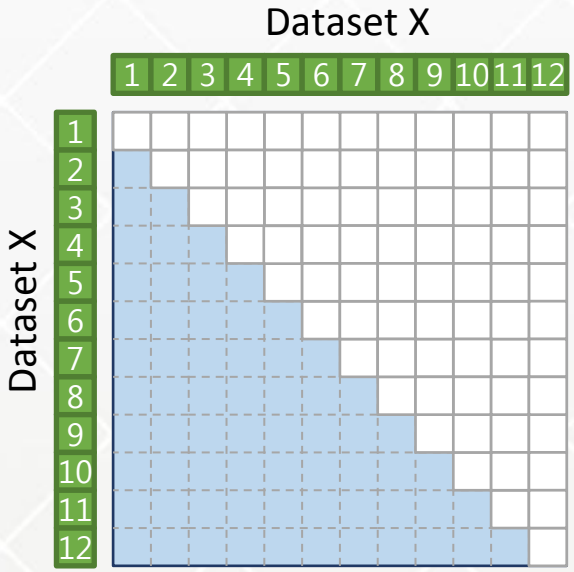
システムソフトウェア研究チーム
吉永一美

概要

- **(p)Carp : (Parallel) Cartesian Product**
- **以下の2種の計算の並列化を支援するフレームワーク**
  - A) 2つのデータセットからの全データの組み合わせ(直積集合)に対する計算
  - B) 1つのデータセットの中での全組み合わせに対する計算
- **ロードバランス・I/Oの削減を実現した並列化**



A) 2データセットの全組み合わせ



B) 1データセット内の全組み合わせ

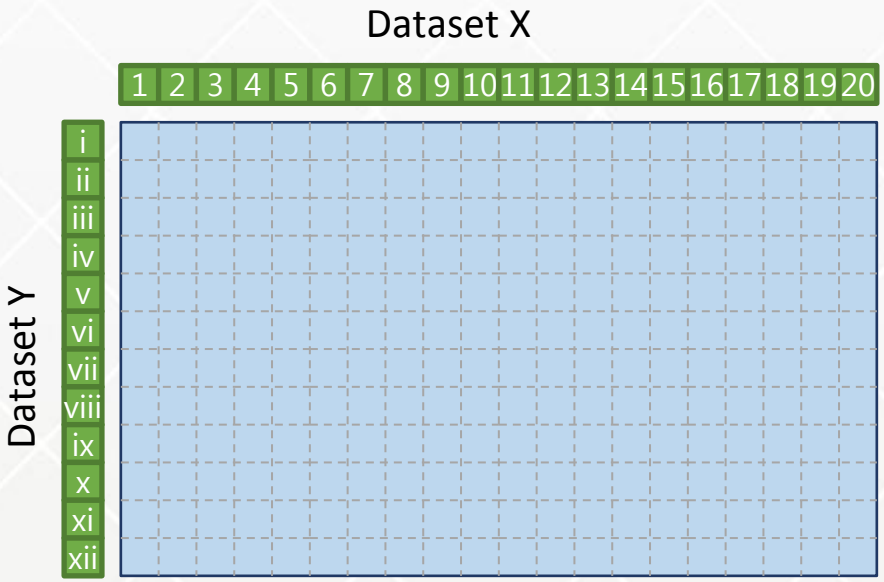
概要

- **(p)Carp : (Parallel) Cartesian Product**

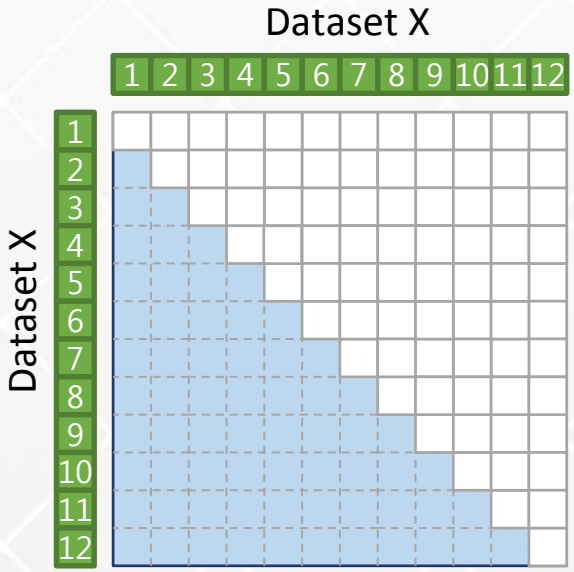
N 1つのデータ(レコード)

上と左のレコードを組み合わせて計算する領域

(Bの場合) 計算しない領域（対角要素はどちらかだけ計算すればよい）



A) 2データセットの全組み合わせ



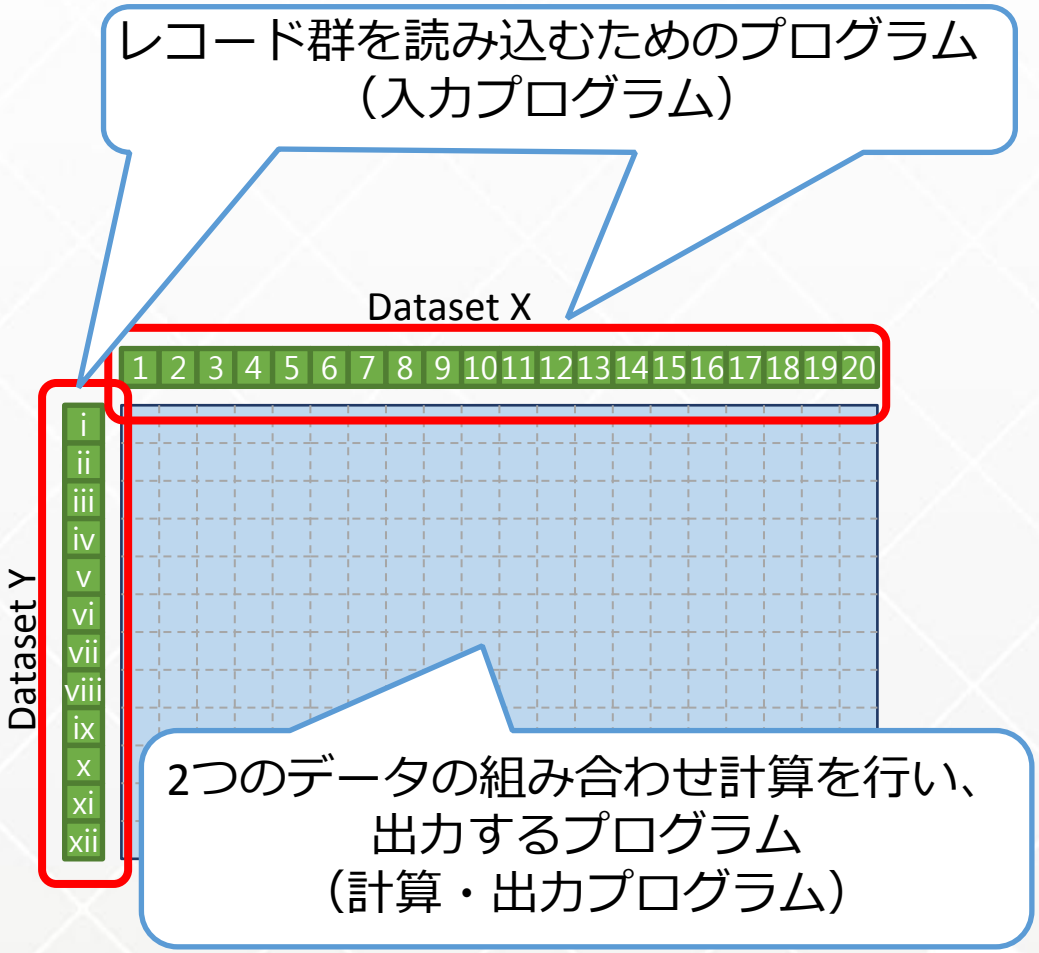
B) 1データセット内の全組み合わせ

ユーザが作成するプログラム

- **Carpを利用するためには、ユーザは2種類のシリアルプログラムを作成する**

1. **入力プログラム**
2. **計算・出力プログラム**
   - 後述のCarp APIを利用して作成する

- **Carpがこれらのプログラムを呼び出し、実行を行う**
- **MPIによる通信部分の記述は不要**
  - pCarp(Parallel-Carp)で実行すると自動で通信(並列実行)
  - sCarp(Serial-Carp)を用いることで、手元の逐次環境(MPIが無い環境)でも容易に開発可能



ユーザが作成するプログラム
(1) 入力プログラム

- **レコード群をCarpへと渡すプログラム**
- **自分の担当するレコード群を読み込み(または生成し)、Carpへと1レコードずつ渡す**
  - 全プロセスを合わせると全レコードを読み込むように、分担して読み込む。基本的に「1/プロセス数」で分割すればよい。
- **レコードのサイズは可変でよい**
- **レコードの区切りをCarpへ伝えるために、レコードをCarpへ渡す前に必ずレコードサイズを渡す(後述のAPIによる)**
- **Carpから呼び出されると、コマンドライン引数の末尾に「ランク」と「総プロセス数」が追加される**
  - 例：mpiexec　pCarp -x “input_prog1　a.txt” -y …..

    → input_progの引数は　　1: a.txt　2: ランク(0-3)　　3: 総プロセス数(4)
  - ランクと総プロセス数を用いて読み込む領域をユーザプログラム内で指定する場合などに使用する

ユーザが作成するプログラム
(2) 計算･出力プログラム

- **計算･出力プログラムでは、Carpから2つのレコードを受け取り、計算を行い、結果を出力する(この繰り返し)**
- **結果の出力は各プロセスが行うことを前提**
  - 全結果をまとめるような機能はない
- **入力プログラムと同様、引数にランクと総プロセス数が追加される**
- **レコード区切りのためにサイズが渡されるため、必ずサイズを受け取ってから、そのサイズを指定してレコードの受け取りを行う**

APIs

- **入力プログラム用API**
  - int carp_put_datasize(int size)
  - int carp_write(void *buf, int size)
  - int carp_write_record(void *buf, int size)
- **計算・出力プログラム用API**
  - int carp_get_datasize(int *size);
  - int carp_read(void *buf, int size);
  - int carp_read_discard();
  - int carp_read_skip(int size);
  - int carp_read_skip_record();
- **入力/計算・出力プログラム共用**
  - int carp_finalize()

- Fortranでは各関数の最後の引数としてint *ierrが追加
- 入力プログラム/計算・出力プログラムは最初に呼び出されたCarp関数で決定
  - 入力用API利用後に計算・出力用APIを利用した場合、CARP_ERRORが返る(その逆も同様)

入力プログラム用API

- **int carp_put_datasize(int size)　：これから書き出すレコードのサイズを指定する**
  - Parameters
    - size - size of next record
  - Return value
    - CARP_SUCCESS – 正常終了
    - CARP_DATA_ERROR - 前のレコードとして指定したサイズ分の書き出しが終了していない

- **int carp_write(void *buf, int size) : Carpへデータの書き出しを行う**
  - Parameters
    - buf - 書き出すデータの先頭アドレス
    - size - 書き出すデータのサイズ
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR - carp_put_datasizeで指定したレコードサイズを上回る
データの書き出しを行った場合

- **int carp_write_record(void *buf, int size)**
  - 下記のコードと同様の処理を行う

```
carp_put_datasize(size);
carp_write(buf,size);
```

  - Return value
    - CARP_SUCCESS – 正常終了
    - CARP_DATA_ERROR - 前のレコードとして指定したサイズ分の書き出しが終了していない

レコードの取り扱いについて補足

- Carpでは、carp_put_datasizeで渡されたsizeバイト分を1レコードとして扱う
- そのため、put_datasize1回に対してwriteを複数回に分けて行う事ができる
  - 例

```
int total, size1, size2;
int data1[10];
char data2[100];
size1 = sizeof(int) * 10 ; size2 =
sizeof(char) * 100;
total = size1 + size2;

carp_put_datasize(total);
carp_write(data1,size1);
carp_write(data2,size2); //←ここまでが1レコード
```

出力・計算プログラム用API

- **int carp_get_datasize(int *size)**
  - 次に読み込むレコードのサイズを受け取る
  - Parameters
    - size - 次のレコードのデータサイズ
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR - 以前のレコードの読み込みが完了していない
    - CARP_DATA_FINISHED – 全てのレコードの処理が終了した
- **int carp_read(void *buf, int size)**
  - Carpからデータの読み込みを行う
  - Parameters
    - buf - 読み込みバッファの先頭アドレス
    - size - 読み込みサイズ
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR - carp_get_datasizeで受け取ったレコードサイズを超えて読み込みを行おうとした
  - 入力プログラムの時と同様、複数回に分けて読み込んで良い

出力・計算プログラム用API

- **int carp_read_skip(int size)**
  - 現在読み込み中のレコードのデータをsizeバイト分切り捨てる
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR - レコードサイズを超えて切り捨てサイズを指定した
- **int carp_read_skip_record()**
  - 現在読み込み中のレコードの残り全てを切り捨てる
  - carp_read_skipで読み込み中レコードの残りサイズを指定した場合と同じ
  - Return value
    - CARP_SUCCESS- 正常終了
    - CARP_DATA_ERROR –
- **int carp_read_discard()**
  - 残りのレコードを全て切り捨てる(現在読み込んでいるものだけでなく、以降全て)
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR -

共用API

- **int carp_finalize()**
  - 終了処理を行う
  - Return value
    - CARP_SUCCESS - 正常終了
    - CARP_DATA_ERROR

      入力プログラム：最後に指定されたレコードの書き出しが終了していない

      計算・出力プログラム：Carp側から読み込むべきレコードがまだ残っている

      (レコードの読み込み途中の場合も含む)

各プログラムの引数へのformat文指定

- **各ユーザプログラムの引数には、%r（ランク番号）・%t（合計プロセス数）及び%R（ランク番号・自動桁揃え）の指定が可能**
  - ユーザプログラムには対応した値に置換されて渡される
  - 桁数指定も可能

    例：全20プロセスで実行、ランク2のプロセスでは
    - data%r.dat　：data2.dat
    - data%3r.dat　：data　2.dat　（3桁揃え・空白詰め）
    - data%03r.dat：data002.dat　（ 3桁揃え・ゼロ詰め）
    - data%R.dat　：data02.dat (最大ランク番号と同じ桁数に揃え、ゼロ詰め)

- **使用例：**

```
mpiexec pCarp -x "./reader_x datax_%02r.dat" -y "./reader_y datay_%02r.dat" -c "./computation ¥"output_%2r.dat¥""
```

(空白詰めは¥"で囲む必要あり、あまり推奨しない)

サンプルプログラム

- **単純な全組み合わせ、文字列連結処理のサンプル**
- **/opt/aics/carp/sample 以下**
- **CrossJoin_(C, Fortran, Python)**
- **CrossJoin_Format (C言語のみ)**
  - 処理としてはどちらもほぼ同じ
  - Formatの方は、Format文引数を使用するサンプル

サンプルプログラム
CrossJoin

- **SQLのCROSSJOIN操作を行うサンプル**
  - 単純に2つのレコードを連結して出力する
  - 入力プログラムの第一引数としてファイルリストを渡す
    - 1行目：データファイルの総数
    - 2行目以降：データファイルのパス
      - データファイル内に実際のレコードが記述(1行1レコード)
  - 入力プログラムは、総プロセス数とランクを用いて、ファイルリストの中から担当すべきファイルを決定し、実際にそのファイルを読み込む
  - 出力プログラムは、受け取ったデータを連結して出力する

サンプルプログラム
CrossJoin
データファイル群

ファイルリスト

```
20
datafiles/data00.txt
datafiles/data01.txt
datafiles/data02.txt
datafiles/data03.txt
datafiles/data04.txt
datafiles/data05.txt
datafiles/data06.txt
datafiles/data07.txt
datafiles/data08.txt
datafiles/data09.txt
datafiles/data10.txt
datafiles/data11.txt
datafiles/data12.txt
datafiles/data13.txt
datafiles/data14.txt
datafiles/data15.txt
datafiles/data16.txt
datafiles/data17.txt
datafiles/data18.txt
datafiles/data19.txt
```

data00.txt

```
data00-0
data00-1
data00-2
data00-3
data00-4
data00-5
data00-6
data00-7
data00-8
data00-9
data00-10
data00-11
data00-12
data00-13
data00-14
data00-15
data00-16
data00-17
data00-18
data00-19
```

data01.txt

```
data01-0
data01-1
data01-2
data01-3
data01-4
data01-5
data01-6
data01-7
data01-8
data01-9
data01-10
data01-11
data01-12
data01-13
data01-14
data01-15
data01-16
data01-17
data01-18
data01-19
data01-20
data01-21
data01-22
data01-23
data01-24
```

サンプル：CrossJoin(C)
入力プログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include "carp.h"

int main(int argc, char *argv[]){
  FILE *flp,*fp;
  char *filelistpath;
  int rank, totalprocs;
  int totalfiles, readcnt, startpos;
  char readfilepath[50];
  if ( argc != 4 ){
    return -1;
  }
  filelistpath = argv[1];
  rank = atoi(argv[2]);
  totalprocs = atoi(argv[3]);

  flp = fopen(filelistpath,"r");
  fscanf(flp, "%d¥n", &totalfiles);

  int rem = totalfiles % totalprocs;
  readcnt = totalfiles / totalprocs + ( rem > rank ?
1 : 0 );
  startpos = totalfiles / totalprocs * rank  + ( rem <
rank ? rem : rank );

  int cnt;
  int datasize;
  char data[100];
  // skip filelist
  for( cnt = 0; cnt < startpos; cnt++ ){
    fscanf(flp,"%s",readfilepath);
  }
  // read file(s)
  for( cnt = 0; cnt < readcnt; cnt++ ){
    fscanf(flp,"%s",readfilepath);
    fp = fopen(readfilepath, "r");
    while(fscanf(fp,"%s¥n",data)!=EOF){
      datasize = sizeof(char) * (strlen(data)+1);
      if ( carp_write_record(data, datasize) ==
CARP_DATA_ERROR ) {
        fprintf(stderr,"[%d]error at
write_record¥n",rank);
      }
          }
    fclose(fp);
  }

  if (carp_finalize() != CARP_SUCCESS ){
    fprintf(stderr,"[%d]error at finalize¥n", rank);
  }
  fclose(flp);
  return 0;
}
```

サンプル：CrossJoin(C)
入力プログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include "carp.h"

int main(int argc, char *argv[]){
  FILE *flp,*fp;
  char *filelistpath;
  int rank, totalprocs;
  int totalfiles, readcnt, startpos
  char readfilepath[50];
  if ( argc != 4 ){
    return -1;
  }
  filelistpath = argv[1];
  rank = atoi(argv[2]);
  totalprocs = atoi(argv[3]);

  flp = fopen(filelistpath,"r");
  fscanf(flp, "%d¥n", &totalfiles);

  int rem = totalfiles % totalprocs;
  readcnt = totalfiles / totalprocs + ( rem > rank ?
1 : 0 );
  startpos = totalfiles / totalprocs * rank  + ( rem
rank ? rem : rank );

  int cnt;
  int datasize;
  char data[100];
  // skip filelist
  for( cnt = 0; cnt < startpos; cnt++ ){
    fscanf(flp,"%s",readfilepath);
  }
```

引数処理

```
  filelistpath = argv[1];  // ファイルリストのパス
  rank = atoi(argv[2]);    // 自ランク(Carpにより追加)
  totalprocs = atoi(argv[3]); // 総ランク数
```

ファイルリストを開き、総ファイル数を取得

```
  flp = fopen(filelistpath,"r");
  fscanf(flp, "%d¥n", &totalfiles);
```

```
  if (carp_finalize() != CARP_SUCCESS ){
    fprintf(stderr,"[%d]error at finalize¥n", rank);
  }
```

読み込むデータファイル数と、何番目から読み込むのかを計算

```
  int rem = totalfiles % totalprocs;
  readcnt = totalfiles / totalprocs +
( rem > rank ? 1 : 0 );
  startpos = totalfiles / totalprocs *
rank  + ( rem < rank ? rem : rank );
```

サンプル：CrossJoin(C)
入力プログラム


```
#include <stdio.h>
#include <stdlib.h>
#
#
#
int main(int argc, char *argv[]){
  FILE *flp,*fp;
  char *filelistpath;
  int rank, totalprocs;
  int totalfiles, readcnt, startpos;
```

データファイルを開いてレコードの読み込み

```
  // read file(s)
  for( cnt = 0; cnt < readcnt; cnt++ ){
    fscanf(flp,"%s",readfilepath);
    fp = fopen(readfilepath, "r");
    while(fscanf(fp,"%s¥n",data)!=EOF){
      datasize = sizeof(char) * (strlen(data)+1);
      if ( carp_write_record(data, datasize) ==
CARP_DATA_ERROR ) {
        fprintf(stderr,"[%d]error at
write_record¥n",rank);
      }
    }
    fclose(fp);
  }

  if (carp_finalize() != CARP_SUCCESS ){
    fprintf(stderr,"[%d]error at finalize¥n", rank);
  }
```

終了処理

```
 if (carp_finalize() !=
CARP_SUCCESS ){
    fprintf(略);
  }
```

```
  flp = fopen(filelistpath,"r");
  f
  i
  r
1 :
  s
ran
  i
  i
  c
  /
  f
  }
```

データを1行ずつ読み込み、Carpへ送る(carp_write_recordを使用)

```
  while(fscanf(fp,"%s¥n",data)!=EOF){
      datasize = sizeof(char) * (strlen(data)+1);
      if ( carp_write_record(data, datasize) == CARP_DATA_ERROR ) {
        fprintf(stderr,"[%d]error at write_record¥n",rank);
      }
    }
  fclose(fp);
  }
```

サンプル：CrossJoin(C)
計算･出力プログラム


```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "carp.h"

int main(int argc, char *argv[]){
  FILE *fp;
  int rank = atoi(argv[1]);
  // output file
  char filename[50];
  sprintf(filename,"./result_%d.txt",rank);
  fp = fopen(filename,"w");
  if(fp == NULL) return -1;
  int datasize;
  char data_str1[100], data_str2[100];
  int err;

  while(1){
    err = carp_get_datasize(&datasize);
    if ( err != CARP_SUCCESS ){
      if ( err == CARP_DATA_FINISHED ){ // EOF
        break;
      }
      fprintf(stderr,"[%d]error at get_datasize 1¥n",
rank);
      break;
    }
    err = carp_read(data_str1,datasize);
    if ( err != CARP_SUCCESS ){
      fprintf(stderr,"[%d]error at read data 1¥n", rank);
      break;
    }
    err = carp_get_datasize(&datasize);
    if ( err != CARP_SUCCESS ){
      if ( err == CARP_DATA_FINISHED ){ // EOF
        break;
      }
      fprintf(stderr,"[%d]error at get_datasize 2¥n",
rank);
      break;
    }
    err =carp_read(data_str2,datasize);
    if ( err != CARP_SUCCESS ){
      fprintf(stderr,"[%d]error at read data 2¥n", rank);
      break;
    }
    fprintf(fp,"(%s,%s)¥n",data_str1,data_str2);
  }
  fclose(fp);

  if (carp_finalize() != CARP_SUCCESS ){
    fprintf(stderr,"[%d]error at finalize¥n", rank);
  }
  return 0;
}
```

サンプル：CrossJoin(C)
計算･出力プログラム

レコードを２つ受け取って組み合わせて出力、を繰り返す
while(1)で無限ループ
1つ目のレコードのサイズを受け取り

```
err = carp_get_datasize(&datasize);
    if ( err != CARP_SUCCESS ){
      if ( err == CARP_DATA_FINISHED ){
        // 全てのレコードの処理が終了
        break;
```

```
while(1){
  err = carp_get_datasize(&datasize);
  if ( err != CARP_SUCCESS ){
    if ( err == CARP_DATA_FINISHED ){ // EOF
      break;
    }
    fprintf(stderr,"[%d]error at get_datasize 1¥n", rank);
    break;
  }
```

```
  err = carp_read(data_str1,datasize);
  if ( err != CARP_SUCCESS ){
    fprintf(stderr,"[%d]error at read data 1¥n", rank);
    break;
  }
  err = carp_get_datasize(&datasize);
  if ( err != CARP_SUCCESS ){
    if ( err == CARP_DATA_FINISHED ){ // EOF
      break;
```

1つ目のレコードを受け取り

```
err = carp_read(data_str1,datasize);
    if ( err != CARP_SUCCESS ){
      fprintf(略);
      break;
    }
  }
  fprintf(fp,"(%s,%s)¥n",data_str1,data_str2);
}
fclose(fp);

if (carp_finalize() != CARP_SUCCESS ){
  fprintf(stderr,"[%d]error at finalize¥n", rank);
}
return 0;
}
```

サンプル：CrossJoin(C)
計算･出力プログラム

レコードを２つ受け取って組み合わせて出力、を繰り返す
while(1)で無限ループ

2つ目のレコードサイズの受け取り

2つ目のレコードを受け取り

2つのレコードに対して処理
(この例では連結してファイルに書き出すのみ)

```
#i
#i
#i
#i
int main(int argc, char *argv[]){
  FILE *fp;
  int rank = atoi(argv[1]);
  // output
  char filen
  sprintf(filename, "./result_%d.txt",rank);
  fp = fopen(filename,"w");
  if(fp == NULL) return -1;
  int datasize;
  char data_st
  int err;

  while(1){
    err = carp_get_datasize(&datasize);
    if ( err != CARP_SUCCESS ){
      if ( err == CARP_DATA_FINISHED ){ // EOF
        break;
      }
      fprintf(stderr,"[%d]error a
rank);
      break;
    }
```

```
    err = carp_read(data_str1,datasize);
    if ( err != CARP_SUCCESS ){
      fprintf(stderr,"[%d]error at read data 1¥n", rank);
      break;
    }
    err = carp_get_datasize(&datasize);
    if ( err != CARP_SUCCESS ){
      if ( err == CARP_DATA_FINISHED ){ // EOF
        break;
      }
      fprintf(stderr,"[%d]error at get_datasize 2¥n",
rank);
      break;
    }
    err =carp_read(data_str2,datasize);
    if ( err != CARP_SUCCESS ){
      fprintf(stderr,"[%d]error at read data 2¥n", rank);
      break;
    }
    fprintf(fp,"(%s,%s)¥n",data_str1,data_str2);
  }
                                          ¥n", rank);
  return 0;
}
```

京へのインストール状況

- **/opt/aics/Carp/以下にインストールされている(ログインノード･計算ノード共に)**
  - bin/pCarp：並列版, 計算ノード用
  - bin/sCarp：逐次版, pps等でのデバッグ実行用(gcc,x86でコンパイル済)
  - include/carp.h, carpf.h：ヘッダファイル
  - lib/libpcarp.a, libscarp.a ： 静的ライブラリファイル
  - 多ノードでの実行の場合、直接全ノードが /opt/aics/Carp/bin/pCarp を実行するとファイルアクセスの混雑により性能が低下する可能性がある
  - pCarpをステージインしてランクディレクトリに置くことでこれを回避可能
  - なお、/opt/aics/ 以下はステージイン対象に設定できないため、ステージインする場合は /volume41/data/aicsapp/Carp/bin/pCarp を指定する

京でのユーザプログラムのコンパイル方法

pCarp用

```
fccpx -I<INCLUDE_DIR> -L<LIB_DIR> -o hoge
hoge.c -lpcarp
frtpx -I<INCLUDE_DIR> -L<LIB_DIR> -o hoge
hoge.f -lpcarp
```

sCarp用

```
gcc  -I<INCLUDE_DIR> -L<LIB_DIR> -o hoge hoge.c
-lscarp
gfort -I<INCLUDE_DIR> -L<LIB_DIR> -o hoge
hoge.f -lscarp
```

※ INCLUDE_DIR=/opt/aics/Carp/include
LIB_DIR=/opt/aics/Carp/lib

Carpによる実行

- **2データ群の全組み合わせ計算**

```
mpiexec pCarp -x "in_prog1 [opt]" -y "in_prog2 [opt]" -c
"calc_prog [opt]"
```

- **1データ群の中での全組み合わせ計算**

```
mpiexec pCarp -x "in_prog1 [opt]" -c "calc_prog [opt]"
```

  - yオプションを省略すると、1データ群の中での全組み合わせに対する計算

- **sCarpの場合も同様(mpiexecではなく直接実行)**

- **実行時に環境変数を設定することで、Carp⇔ユーザプログラム間の通信に用いる共有メモリ(リングバッファ)のサイズを変更可能**
  - CARP_SHM_SIZE：リングバッファ1つのサイズ(Default:1MB)
  - CARP_SHM_RING_LEN：リングバッファの長さ(Default: 32)
  - 京ではデフォルト値のままでよい

注意事項(1)
動的メモリ割付した領域の利用について(1)

- **動的割付した変数の利用→可能**
- **構造体を作成・その中に動的割付をした変数が含まれている場合に問題が発生する**
  - 一括での構造体送信は不可
  - 変数ごとに分割してcarp_write/readを行うことで対応可能
  - C/Fortranどちらの場合でも同様

注意事項(1)
動的メモリ割付した領域の利用について(2)

```
type TEST
    integer(4) :: x
    integer(4) :: s(100)
end type TEST

type(TEST) :: t
integer(4) :: size
// 構造体内に動的割り付けが無いので
// tを一括して送信出来る

size = 4 * 101
call carp_put_datasize(size,
ierr)
call carp_write(t, size,
ierr)
```

```
type TEST
    integer(4) :: x
    integer(4),pointer :: s(:)
end type TEST

type(TEST) :: t
integer(4) :: size
// 動的割り付け
allocate(t%s(100))

// 構造体内に動的割り付けがあるので
// tは個別に送信する必要がある
size = 4 * 101
call carp_put_datasize（size,ierr)
call carp_write(t%x, 4, ierr);
call carp_write(t%s, 4*100, ierr);
```

注意事項(2)
ノード数＞レコード数時の実行について(1)

- **Carpの実行の前提：各レコードをプロセスに振り分けて読み込み**
  - X方向のレコード群：読み込んだノードにて保持
  - Y方向のレコード群：隣接プロセスへ順次通信
  - 両レコード群の組み合わせに対して演算

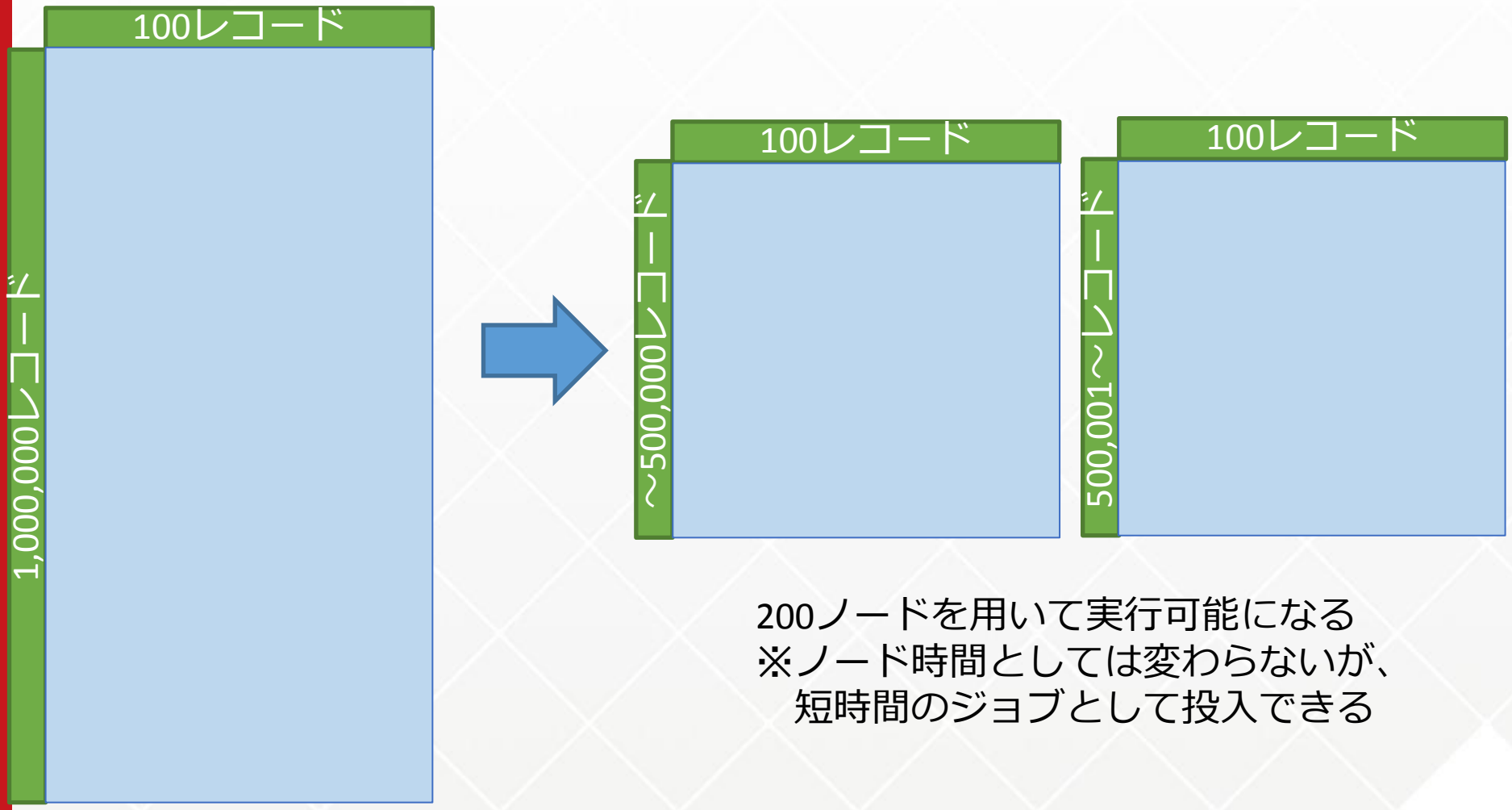
- **X方向のレコード数がノード数よりも少ない場合**
  - 常に計算対象のないノードが存在
    (1レコードも保持していないため)
- **Y方向のレコード数がノード数よりも少ない場合**
  - 各ステップで一部のノードは計算対象がない
    (1レコードも送られてこないステップがあるため)

- **ノード時間の無駄になるので、X,Yのうち少ない方のレコード数以下のノード数で動かすことを推奨する**

注意事項(2)
ノード数＞レコード数時の実行について(2)

- **どうしてもレコード数を上回るノード数で動かしたい場合**
  - 例：X方向のレコード数: 100　/ Y方向のレコード数: 1,000,000
    - このままでは100ノード以下で動かさなければノード時間が無駄になる
  - 複数ジョブに分割して実行することで、レコード数を上回るノード数で無駄なく実行できる
    - 例：上記ジョブを2つのジョブに分割する
    - X方向のレコード数: 100のまま、両ジョブで共通とする
    - Y方向のレコード数: 500,000ずつ、2ジョブ分に分割
      - つまり、100 x 500,000 通りの計算を行うジョブを2個作成
    - 100ノードジョブを2個同時に流す、または、200ノードジョブ内で100プロセスのmpiexecを2個並べて同時実行する。
  - これはTATの短縮にも繋がる
    - 分割前：100ノードで実行すると、1ノードで計算する組み合わせ：1×100万＝100万通り
      - 通常の実行ではこれ以上ノード辺りの計算量を削減できない
    - 分割後：200ノードで実行すると、1ノード当たりの計算量：50万通りの組み合わせに削減
      →各ノードの実行時間が1/2になる　（総ノード時間積は変わらない）

注意事項
ノード数＞レコード数時の実行について(3)



200ノードを用いて実行可能になる
※ノード時間としては変わらないが、
　短時間のジョブとして投入できる

注意事項
ノード内並列について

- **ノード内並列化(スレッド並列化)はユーザ自身がプログラム内で記述する**

- **pCarpでのプログラム実行時には、各ノードで以下が動作する**
  1. **常時動作**
     A) pCarp本体プロセス
  2. **読み込み完了まで動作**
     A) 入力プロセス(データセット１：X方向)
     B) [X入力終了後：入力プロセス(データセット２：Y方向)]
  3. **読み込み完了後(計算中)に動作**
     A) 計算・出力プロセス
     B) pCarp内のMPI通信スレッド

- **計算プログラムのノード内並列化時はpCarpのメインスレッドとMPI通信スレッドが存在することを念頭に置く**

自前でのビルド方法(1)

- **配布URL**

```
http://www-sys-
aics.riken.jp/releasedsoftware/ksoftware/carp.html
```

- **For K, aics-FX10**
  - トップでmakeコマンドを実行するとデフォルト設定によりコンパイルされる
  - 計算ノードとログインノードでアーキテクチャが異なるため、京,FX10ではデフォルトでは以下でビルドされるように設定済み
    - pCarp,pCarp用ライブラリ,pCarp用サンプル
      : 計算ノード用 富士通コンパイラでビルド
    - sCarp,sCarp用ライブラリ,sCarp用サンプル
      : ログイン・post/preノード用 gnuコンパイラでビルド

自前でのビルド方法(2)

- **コンパイラを指定したビルド(その他の環境用等)**
  - makeの引数に指定
    - COMPILER=<pCarp用コンパイラ>
    - COMPILER_SCARP=<sCarp用コンパイラ>
      - コンパイラ名にはgcc, fccpx, gccpx, clang, icc が指定可能
  - 例：FX10で、sCarpも計算ノード用にコンパイルしたい場合
    - $ make COMPILER_SCARP=fccpx
  - 例：全てgcc(+mpi)でコンパイルしたい場合
    - $ make COMPILER=gcc COMPILER_SCARP=gcc
- **Carp⇔ユーザプロセス間通信方式の切り替え**
  - 規定値：共有メモリを介した通信
  - USE_POPEN=YES をオプションに追加してmakeすると、共有メモリを使わずLinux PIPEを用いて通信する

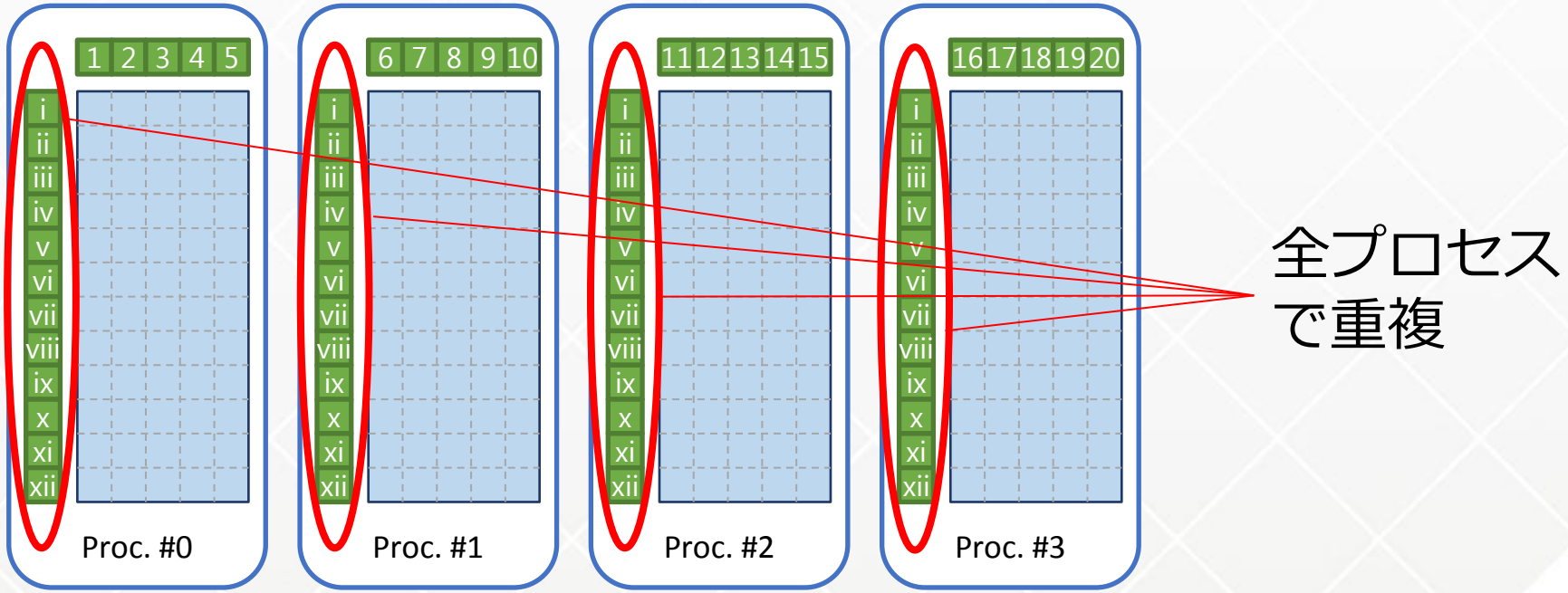
ファイル構成

- **src/**
  - ソースディレクトリ
- **mk/**
  - Makefile用設定ファイル
- **bin/pCarp**
  - 並列実行プログラム
- **bin/sCarp**
  - 1ノードでの実行プログラム
- **lib/[libpcarp.a , libscarp.a]**
  - ユーザプログラムである、入力プログラム・計算(出力)プログラムからリンクするライブラリ
- **include/[carp.h,carpf.h]**
  - C,Fortran用ヘッダファイル
- **sample/CrossJoin_[C,F]**
  - サンプル(c,fortran)
  - それぞれにreader,crossjoin 及び reader_scarp, crossjoin_scarp がビルドされる

Carpの内部動作の説明

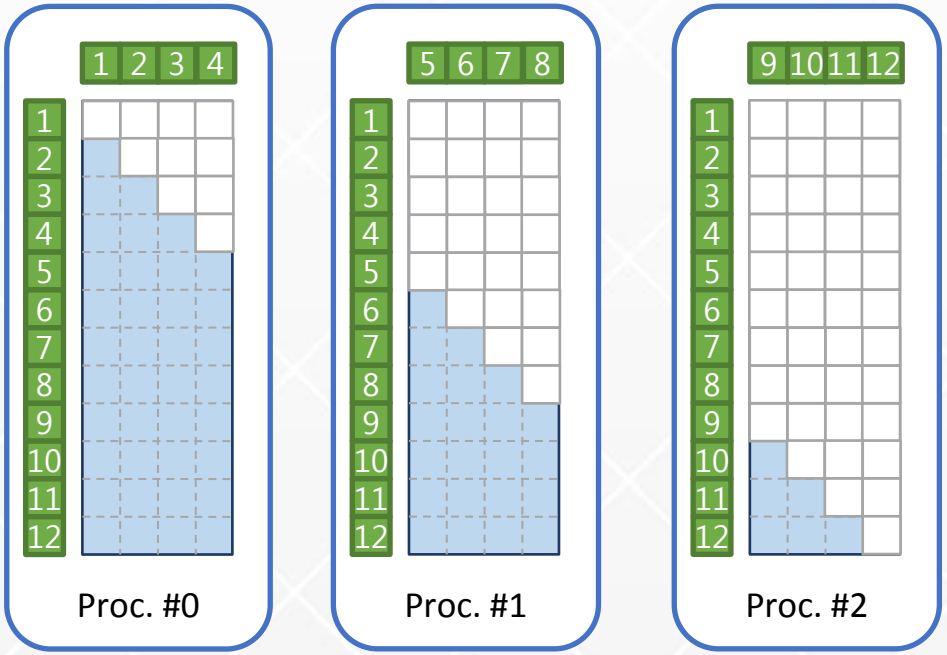
単純な組み合わせ計算の並列化における問題点(1)

- ファイルからの重複読み込みによるI/O量の増加→時間の増大



単純な組み合わせ計算の並列化における問題点(2)

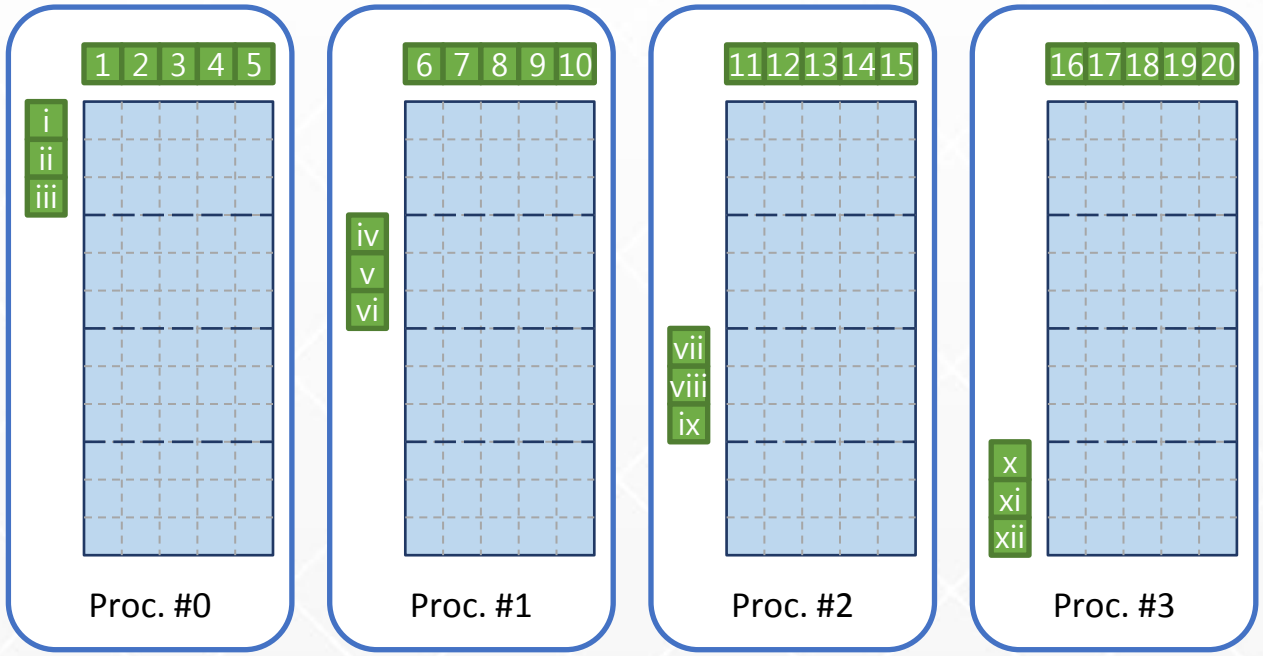
- ロードインバランス



- 単純な分割ではロードバランスが崩れる
- I/Oもバランスを保ちたい

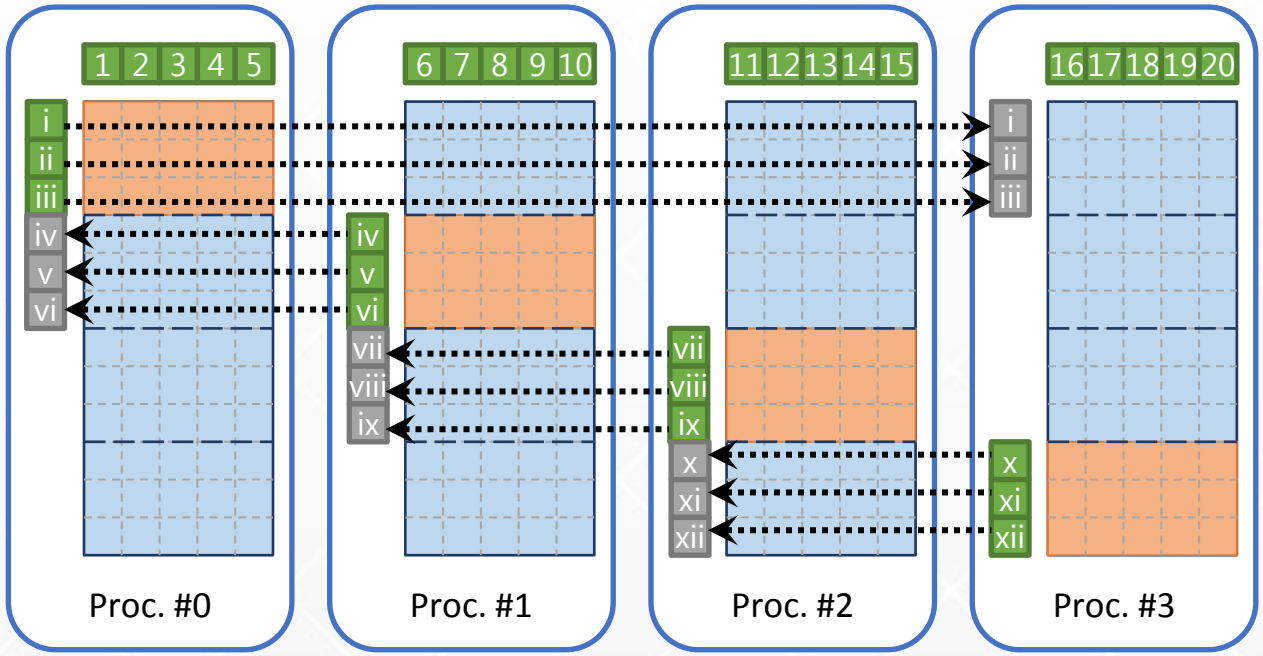
- データ読み込みの重複を避けた上で、負荷・I/Oのバランス

効率的な組み合わせ計算並列化
(2データセットでの全組み合わせ)



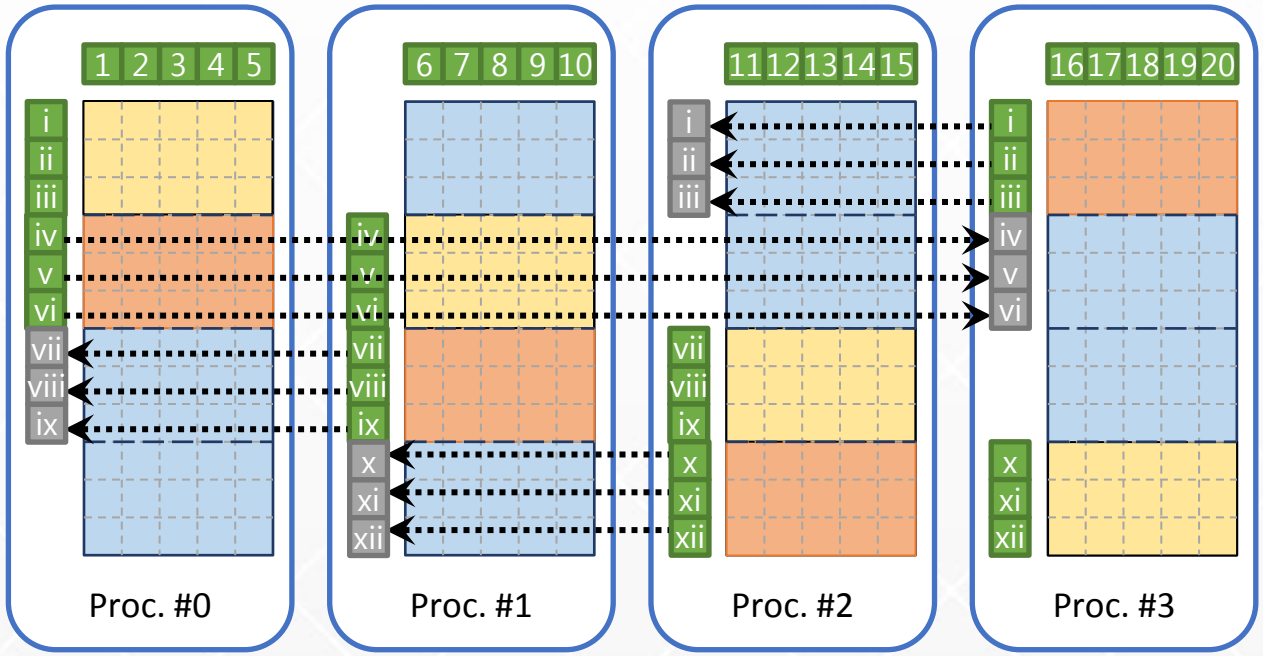
- **読み込みの重複を無くす：プロセスで分担**
  - 全体で全てのレコードが重複無く読み込まれる
  - 実行前に一意にレコードを読み込むプロセスが決定できる
    →ランクディレクトリを利用した読み込み時間の短縮も可能
- **読み込むレコード数を均一にすることでロードバランシング可能**
- **計算中にバックグラウンドで別プロセスへ転送する**

効率的な組み合わせ計算並列化
(2データセットでの全組み合わせ)



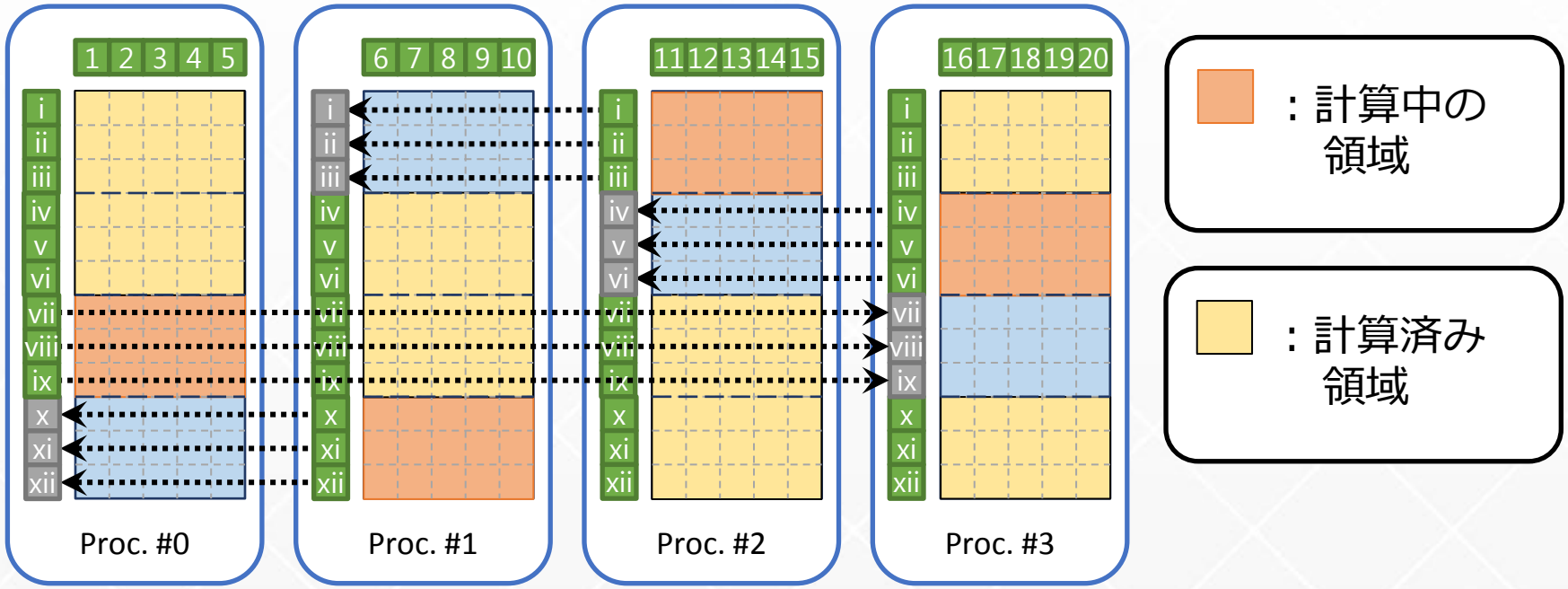
- **読み込みの重複を無くす：プロセスで分担**
  - 全体で全てのレコードが重複無く読み込まれる
  - 実行前に一意にレコードを読み込むプロセスが決定できる
    →ランクディレクトリを利用した読み込み時間の短縮も可能
- **読み込むレコード数を均一にすることでロードバランシング可能**
- **計算中にバックグラウンドで別プロセスへ転送する**

効率的な組み合わせ計算並列化
(2データセットでの全組み合わせ)



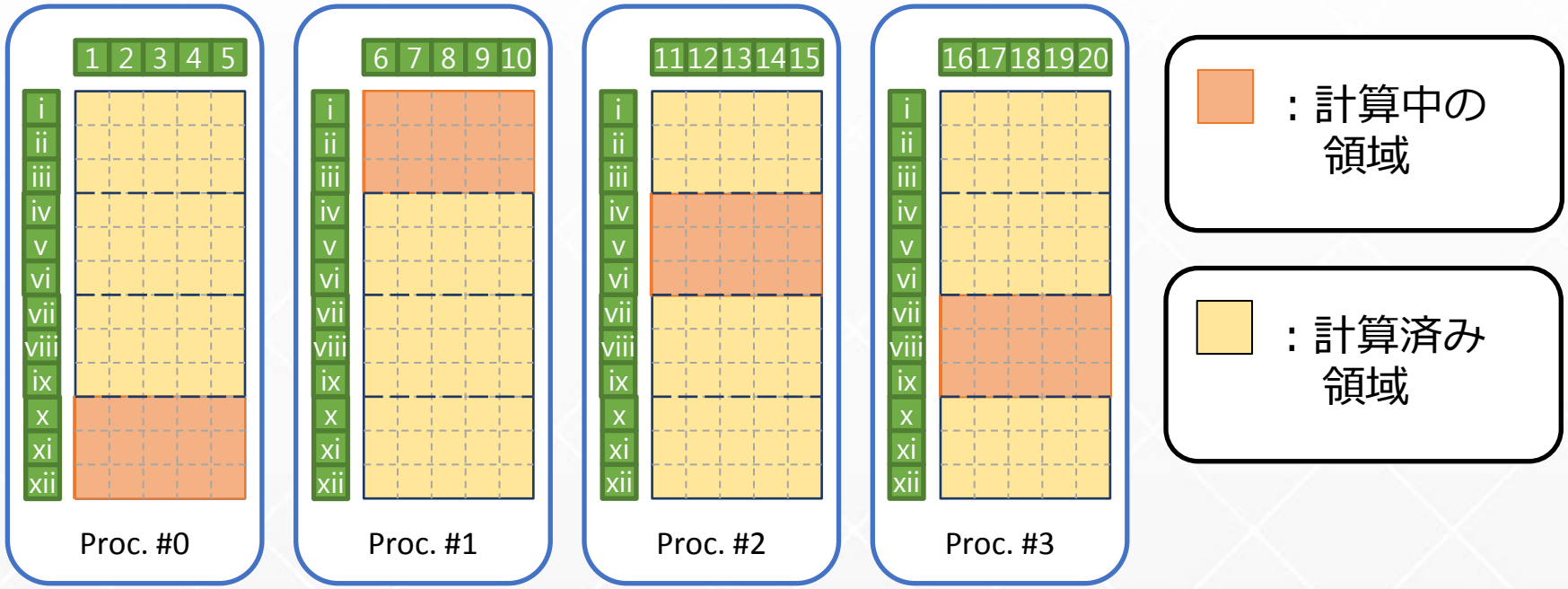
- **読み込みの重複を無くす：プロセスで分担**
  - 全体で全てのレコードが重複無く読み込まれる
  - 実行前に一意にレコードを読み込むプロセスが決定できる
    →ランクディレクトリを利用した読み込み時間の短縮も可能
- **読み込むレコード数を均一にすることでロードバランシング可能**
- **計算中にバックグラウンドで別プロセスへ転送する**

効率的な組み合わせ計算並列化
(2データセットでの全組み合わせ)



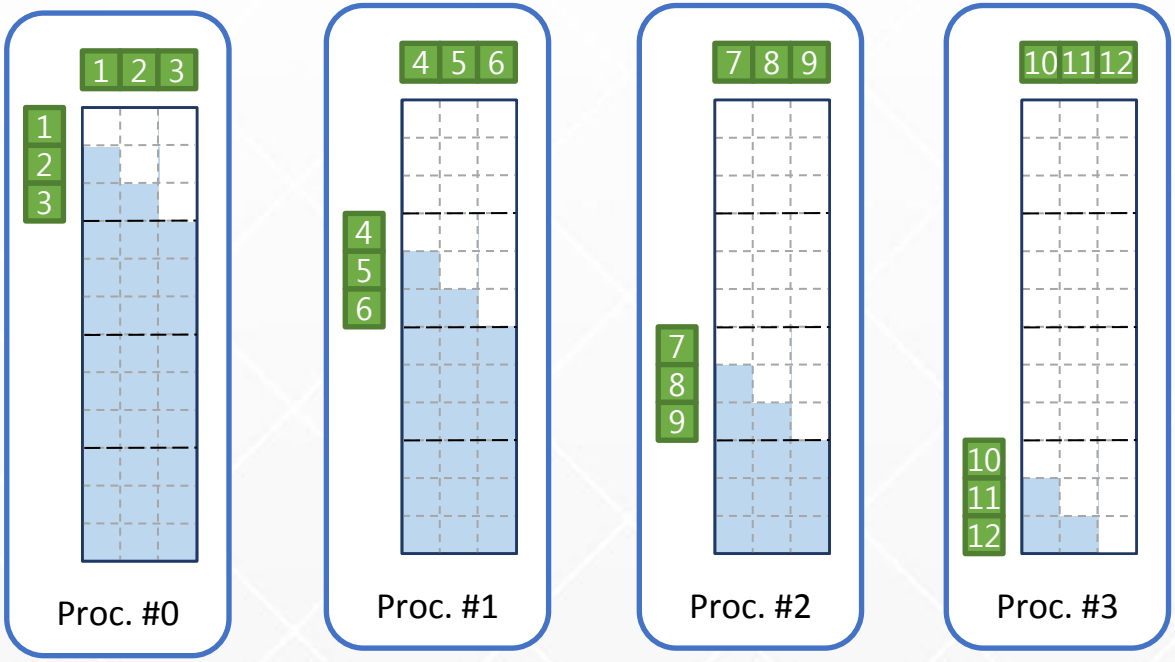
- **読み込みの重複を無くす：プロセスで分担**
  - 全体で全てのレコードが重複無く読み込まれる
  - 実行前に一意にレコードを読み込むプロセスが決定できる
    →ランクディレクトリを利用した読み込み時間の短縮も可能
- **読み込むレコード数を均一にすることでロードバランシング可能**
- **計算中にバックグラウンドで別プロセスへ転送する**

効率的な組み合わせ計算並列化
(2データセットでの全組み合わせ)



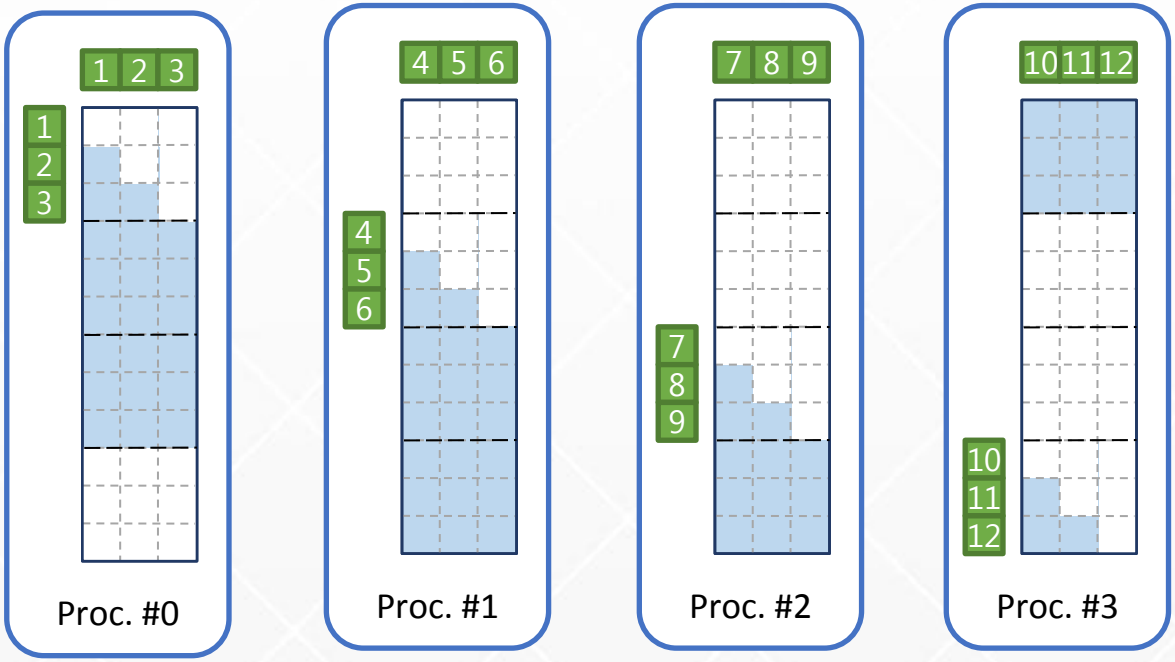
- **読み込みの重複を無くす：プロセスで分担**
  - 全体で全てのレコードが重複無く読み込まれる
  - 実行前に一意にレコードを読み込むプロセスが決定できる
    →ランクディレクトリを利用した読み込み時間の短縮も可能
- **読み込むレコード数を均一にすることでロードバランシング可能**
- **計算中にバックグラウンドで別プロセスへ転送する**

効率的な組み合わせ計算並列化
(1データセット内での全組み合わせ)



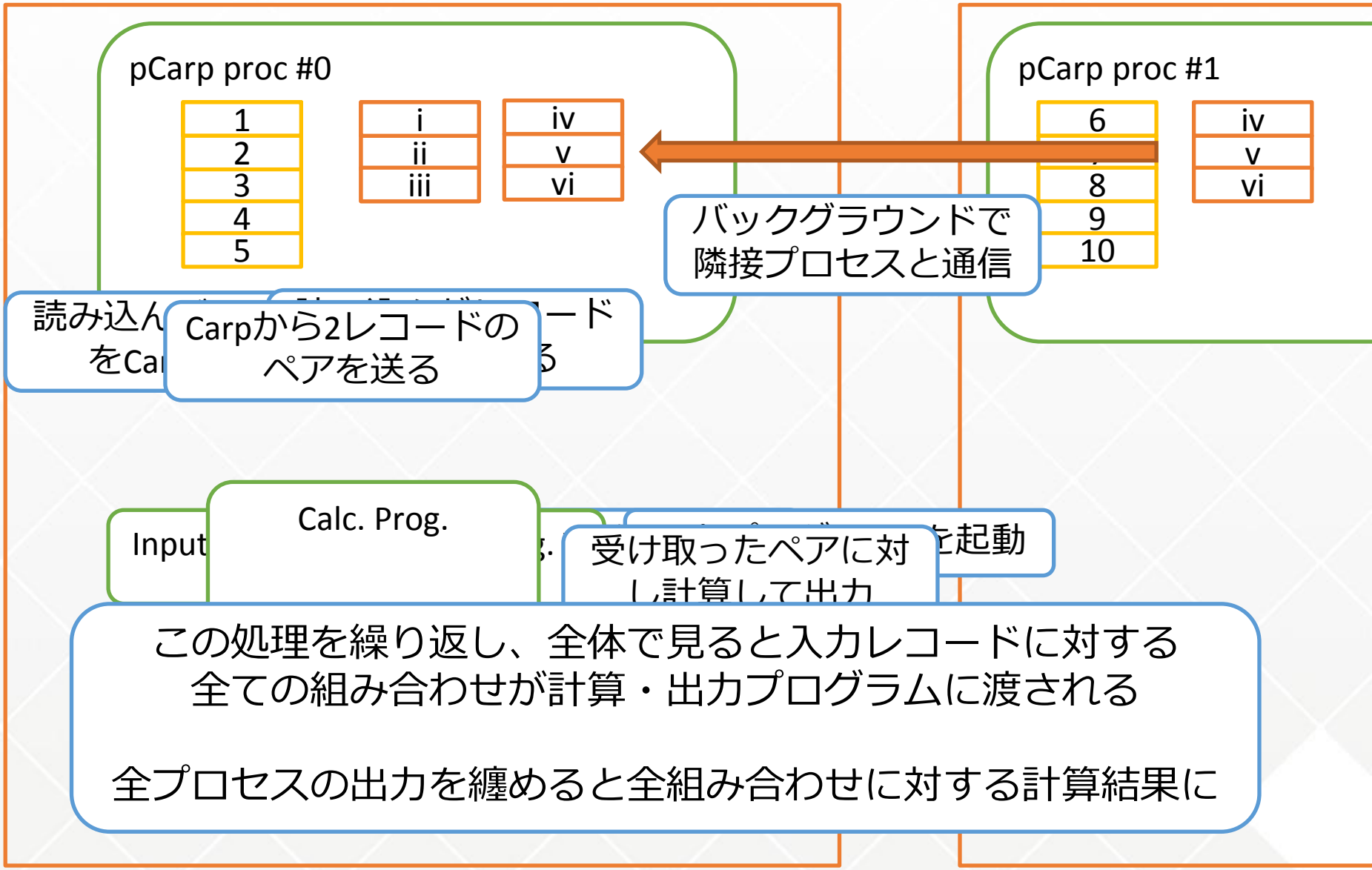
- **1データセット内での全組み合わせもほぼ同様**
- **但し、単純な分割ではロードバランスが崩れる**
- **対角要素を考慮して計算領域を変更する**
  - 奇数プロセスの場合は完全にバランシング可能
  - 偶数プロセスの場合は半分のプロセスが1セット分計算領域が多い

効率的な組み合わせ計算並列化
(1データセット内での全組み合わせ)



- **1データセット内での全組み合わせもほぼ同様**
- **但し、単純な分割ではロードバランスが崩れる**
- **対角要素を考慮して計算領域を変更する**
  - 奇数プロセスの場合は完全にバランシング可能
  - 偶数プロセスの場合は半分のプロセスが1セット分計算領域が多い

pCarp動作イメージ



サポートに関して

- **API Document**

http://www-sys-aics.riken.jp/releasedsoftware/ksoftware/carp.html

- **Carp User ML**
  - carp@riken.jp

- **ソフトウェアサポート問い合わせ先**
  - aics-sys-oss@riken.jp
    - 他ソフトと共通のため、ソフトウェア名を記載してメールすること

Tips(1)
Pythonでの利用 (非公式/サポート外)

- **共有ライブラリを作り、ctypesを用いることで、pythonでも利用可能**
  - sample/CrossJoin_P/にサンプル
    - cdll.LoadLibraryで共有ライブラリを読み込み、Carp関数を呼び出し

```
$ cd src/lib
$ make dll  ← lib/lib[ps]carp.soが作られる
$ mpiexec pCarp -x "python hoge.py …"
```

Tips(2)
シェルスクリプトを介した実行

- 入力プログラム、計算・出力プログラムを直接呼び出すのではなく、間にシェルスクリプトなどを挟んでも動作する

- 例：

```
$ mpiexec ./pCarp -x ./input.sh -y …
```

input.sh

```
#!/bin/bash
RANK=$1
PRCS=$2
./pre.sh $RANK $PRCS # 前処理等をここで行う
./input_prog $RANK $PRCS
```

CrossJoin(Fortran)
入力プログラム


```
program reader
    include 'carpf.h'
    integer  datasize
    character(50) opts
    integer  n,rank,totalprocs,ierr
    integer lines
    integer rem, skip,readcnt
    character(50) readfilepath
    character(100) datastr

    n=iargc()
    if ( n .ne. 3 ) then
        stop
    end if

    call getarg(2 , opts)
    read(opts,*) rank
    call getarg(3 , opts)
    read(opts,*) totalprocs
    call getarg(1 , opts)

    open(7, file=opts, status='old')
    read(7, *) lines

    rem = mod( lines, totalprocs)
    readcnt = lines / totalprocs
    if ( rem > rank ) then
        readcnt = readcnt+1
    end if

    skip = lines / totalprocs * rank
    if ( rem < rank ) then
        skip = skip + rem
    else
        skip = skip + rank
    end if

!   skip filelist
    do n=1,skip
        read(7,'(A)')
    end do

    datasize = 100
    do n=1,readcnt
        read(7,'(A)') readfilepath
        open(8, file=readfilepath, status='old')
        do
            read(8,'(A)', end=100) datastr
            call
carp_write_record(datastr,datasize,ierr)
        end do
100     close(8)
```

CrossJoin(Fortran)
出力プログラム

シミュレーションが未来をひらく

```
program crossjoin
    include 'carpf.h'
    integer  datasize
    character(50) opts
    integer  n,rank,ierr
    character(50) writefilepath
    character(100) datastr1, datastr2

    n=iargc()
    if ( n .ne. 2 ) then
        stop
    end if

    call getarg(1 , opts)
    read(opts,*) rank

    write(writefilepath,'("result-", i0 ,".txt")') rank
    open(7, file=writefilepath, status='replace')

    do
        call carp_get_datasize(datasize, ierr)
        if (ierr .ne. CARP_SUCCESS) then
            if (ierr .ne. CARP_DATA_FINISHED) then
                write(0,*) 'ERROR at getdatasize'
            end if
            goto 100
        end if
        call carp_read(datastr1,datasize, ierr)
        if (ierr .ne. CARP_SUCCESS) then
            write(0,*) 'ERROR at read'
            goto 100
        end if
        call carp_get_datasize(datasize, ierr)
        if (ierr .ne. CARP_SUCCESS) then
            write(0,*) 'ERROR at getdatasize'
            goto 100
        end if
        call carp_read(datastr2,datasize, ierr)
        if (ierr .ne. CARP_SUCCESS) then
            write(0,*) 'ERROR at read'
            goto 100
        end if
        write(7,*)
'(',trim(datastr1),',',trim(datastr2),')'
    end do
    call carp_finalize(ierr)
    if (ierr .ne. CARP_SUCCESS) then
        write(0,*) 'ERROR at Finalize'
    end if
100 close(7)

end program crossjoin
```

CrossJoin_Format

- **sample/CrossJoin_C_Format/**
  - Data_1/ , Data_2/
    - データ群のファイルが100ファイルずつ
  - makelist.sh (ファイルリスト作成スクリプト)
    - データディレクトリ、リスト格納ディレクトリ、プロセス数を指定すると、データファイル数を均等に分担し、各プロセスが読み込むファイルのリストを自動生成する
    - 内部でファイルリストの連番を`seq –w 0 プロセス数-1`で作成（自動で0詰めの桁揃え）
  - 実行方法
    - 各プロセスが読み込むファイルのリストを作成する

      ./makelist.sh Data_1/ List_1/ NUM_OF_PROCS

      ./makelist.sh Data_2/ List_2/ NUM_OF_PROCS
    - リストを用いて実行(前述のCrossJoinのサンプルと異なり、読み込むファイルをプログラム内部で動的に計算しない)

      mpiexec pCarp -np NUM_OF_PROCS -x "./reader List_1/list_%R.txt" -y "./reader List_2/list_%R.txt" -c "./crossjoin results_%R.txt"
  - 処理内容はCrossJoinとほぼ同じなので省略

注意事項(1)
動的メモリ割付した領域の利用について(2)
- C言語版 -

```
struct test{
        int x;
        char s[32];
};

struct test t;

// 構造体内に動的割り付けが無いので
// tを一括して送信出来る
int size = sizeof(struct
test);
carp_put_datasize(size);
carp_write(&t, size);
```

```
struct test{
        int x;
        char *s;
};

struct test t;

// 動的割り付け
t.s = (char
*)malloc(sizeof(char)*32);

// 構造体内に動的割り付けがあるので
// tは個別に送信する必要がある
int size = sizeof(int) +
sizeof(char) * 32;
carp_put_datasize（size);
carp_write(&t.x, sizeof(int));
carp_write(t.s, sizeof(char) *
```