XcalableMP入門

中尾 昌広（理化学研究所 計算科学研究センター）

もくじ

- XcalableMP（XMP）の概要説明
- XMPプログラミング
- Lattice QCDの実装

XMP誕生の背景

- 分散メモリシステムに対するプログラミングにはMPIが一般的
  - MPIはライブラリベース
  - プログラミングコストが大きい
- 目標
  - 高性能と高生産性を兼ね備えた並列プログラミング言語の開発

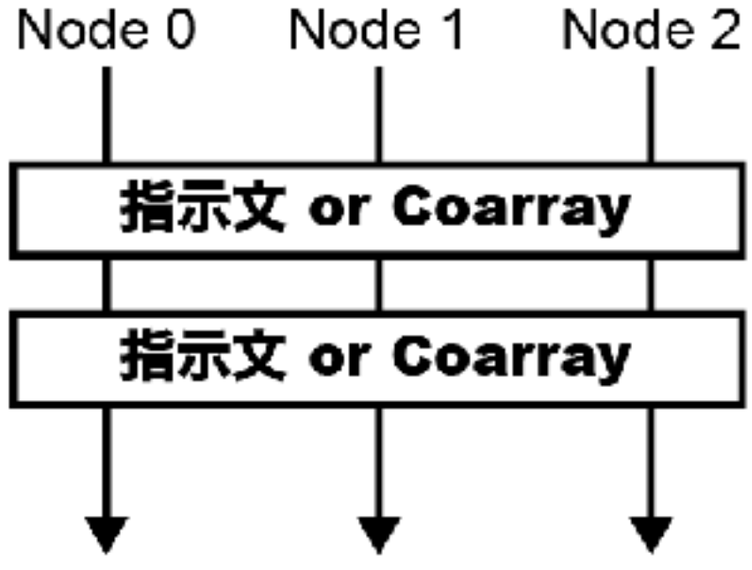
XMPの概要（1/2）

- PCクラスタコンソーシアムのXMP規格部会で仕様策定
- MPIに代わる並列プログラミングモデル
- FortranとCに対して，指示文とCoarray記法を用いた並列拡張
  - 指示文なので，既存の逐次コードからの移行が容易
  - C++にも対応中
- MPIとの連携機能もあるので，既存のMPIコードの部分的なXMP化も可能（OpenMPとの連携も可能）

http://xcalablemp.org

XMPの概要（2/2）

- SPMD (Single Program Multiple Data)
  - MPIと同じ
  - 各実行主体（XMPではノードと呼称, MPIのランクのようなもの）が同じコードを独立に重複して実行する
  - 通信は指示文やCoarray記法によって明示的に行われる
    - 通信箇所が明確なので，性能チューニングが容易（HPFの教訓）



- 2つのメモリモデル
  - 指示文によるグローバルビュー
  - Coarray記法によるローカルビュー

もくじ

- XcalableMP（XMP）の概要説明
- XMPプログラミング
- Lattice QCDの実装

グローバルビュープログラミング

- グローバルビュープログラミングは「解くべき問題全体を記述し，それを各ノードが分担する方法を示す」
  - 例：問題1～100を4人で分担して解け
- ローカルビュープログラミングは「各ノードが解くべき問題を個別に示す」（MPIはローカルビュー）
  - 例：ノード1は問題1～25を解け．ノード2は……
- 基本的に指示文を挿入するだけなので，並列化は簡易
- グローバルビューのための指示文
  - データマッピング：分散配列の定義など
  - ワークマッピング：Loop文の分割など
  - 通信・同期

Example of XMP programming in C

```
int a[MAX], sum = 0;
#pragma xmp nodes p[*]
#pragma xmp template t[MAX]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
      :
#pragma xmp loop on t[i] reduction(+:sum)
  for(int i = 0; i <MAX; i++){
    a[i] = foo(i);
    sum += a[i];
}
```

分散配列の定義．配列a[]を
複数ノードにブロック分割する

ループ文の分割と集約演算

ほぼ同じ指示文を
CとFortranに提供

Example of XMP programming in F

```
integer :: a(MAX), sum = 0
!$xmp nodes p(*)
!$xmp template t(MAX)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
    :
!$xmp loop on t(i) reduction(+:sum)
  do i = 1, MAX
    a(i) = foo(i)
    sum = sum + a(i)
enddo
```

分散配列の定義．配列a[]を
複数ノードにブロック分割する

ループ文の分割と集約演算

ほぼ同じ指示文を
CとFortranに提供

XMPとMPIとの比較

XMP

```
int array[MAX], res = 0;
#pragma xmp nodes p[*]
#pragma xmp template t[MAX]
#pragma xmp distribute t[block] onto p
#pragma xmp align array[i] with t[i]

int main(){
#pragma xmp loop on t[i] reduction(+:res)
   for (int i = 0; i < MAX; i++){
      array[i] = func(i);
      res += array[i];
   }
  return 0;
}
```

少しの手間で、逐次イメージを
残したまま、並列化できる

MPI

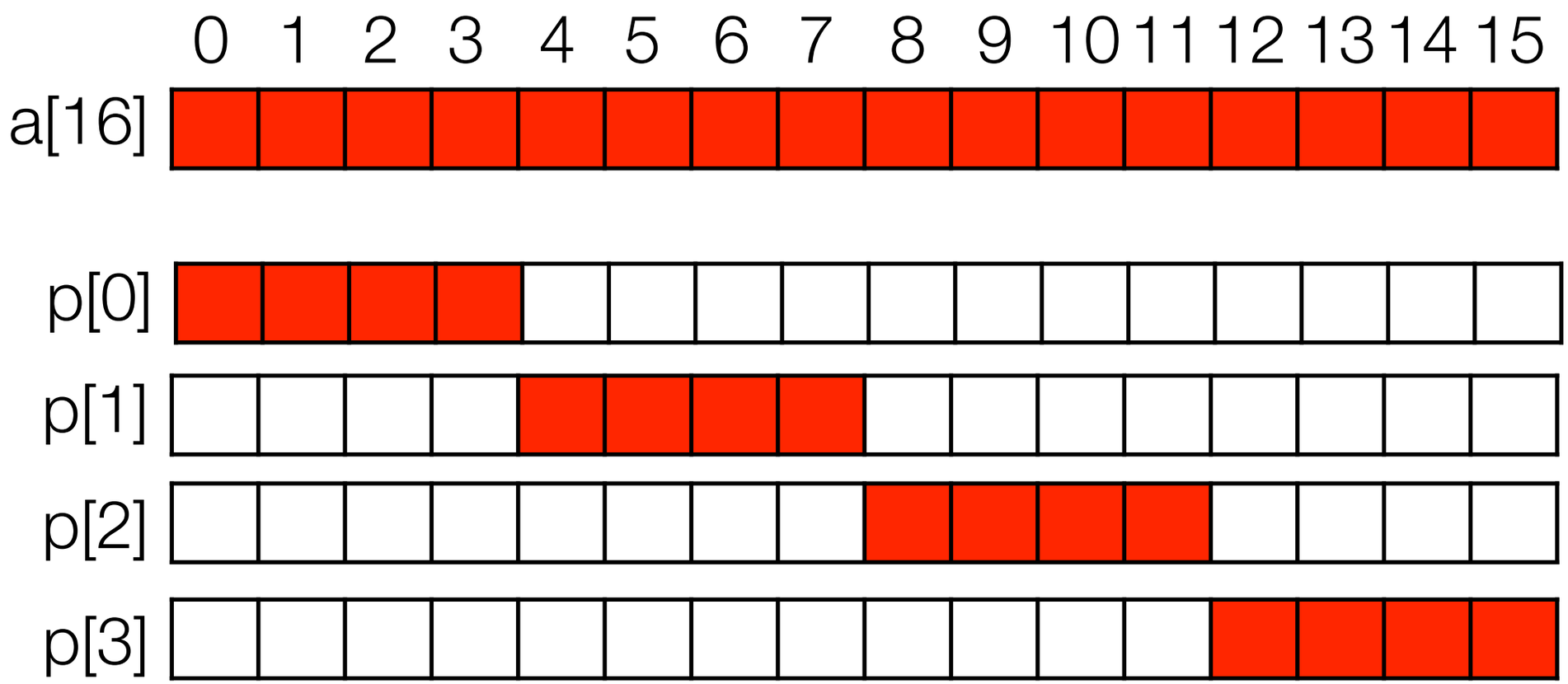
```
int array[MAX], res = 0;

int main(int argc, char **argv){
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int dx = MAX/size;
  in llimit = rank * dx;
  int ulimit = (rank != (size -1))? llimit + dx : MAX;
  int temp_res = 0;

  for(int i = llimit; i < ulimit; i++){
    array[i] = func(i);
    temp_res += array[i];
  }

  MPI_Allreduce(&temp_res, &res, 1, MPI_INT,
                        MPI_SUM, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

loop指示文

- loop指示文はfor文の直前に挿入する

```
#pragma xmp loop on t[i]
for(int i=0;i<16;i++){
```

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16];
#pragma xmp align a[i] with t[i]
```



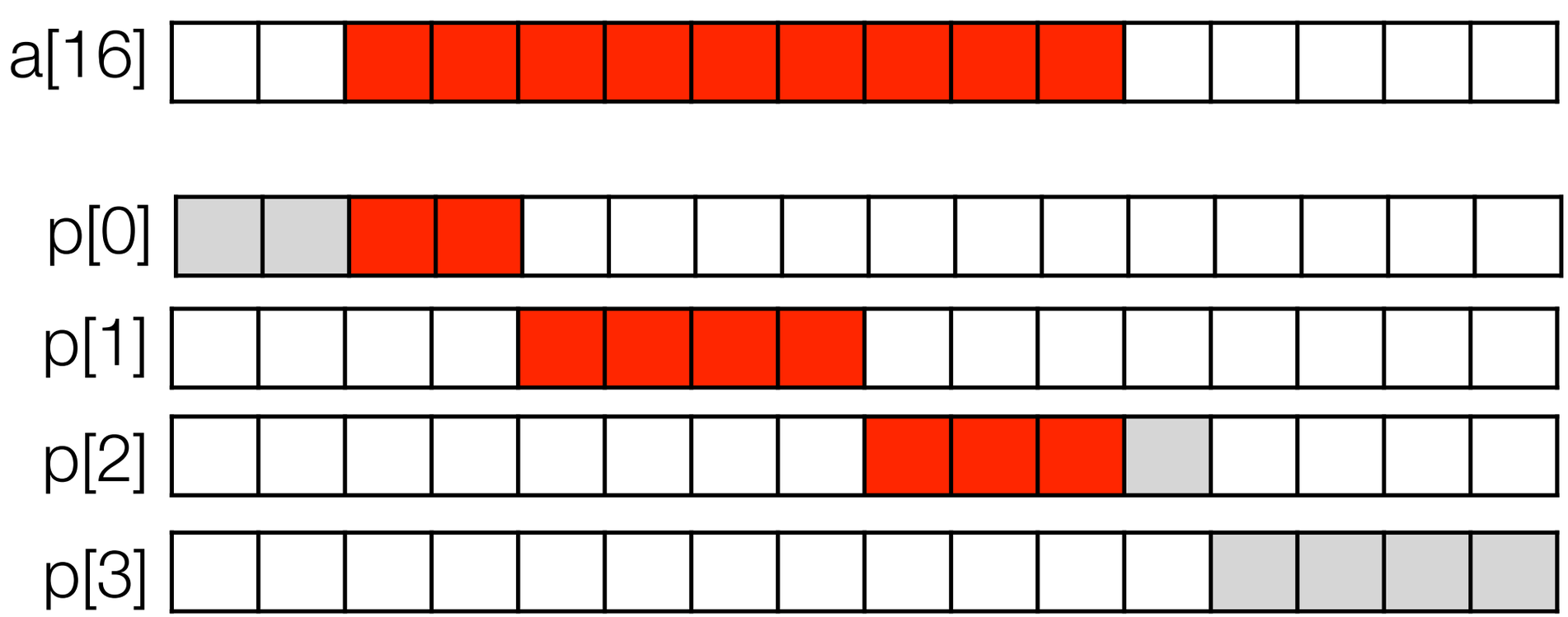
各ノードは赤いセルを並列に処理する.

loop指示文

- loop指示文はfor文の直前に挿入する

```
#pragma xmp loop on t[i]
for(int i=2;i<11;i++){
```

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16];
#pragma xmp align a[i] with t[i]
```



各ノードは赤いセルを並列に処理する.

多次元のloop指示文

[C]

```
#pragma xmp nodes p[4][2]
#pragma xmp template t[20][20]
#pragma xmp distribute t[block][block] onto p
int a[20][20];
#pragma xmp align a[i][j] with t[i][j]

#pragma xmp loop on t[i][j]
for(int i=0;i<20;i++){
  for(int j=0;j<20;j++){
      a[i][j] = func(i, j);
  }
}
```

[F]

```
!$xmp nodes p(2,4)
!$xmp template t(20,20)
!$xmp distribute t(block,block) onto p
integer :: a(20,20);
!$xmp align a(j,i) with t(j,i)

!$xmp loop on t(j, i)
do i=1, 20
  do j=1, 20
    a(j,i) = func(j,i)
  end do
end do
```

XMP指示文 + OpenMP指示文

[C]

```
#pragma xmp loop on t[i]
#pragma omp parallel for
for(int i=0;i<20;i++){
    a[i] = i;
}
```

[F]

```
!$xmp loop on t(i)
!$omp parallel do
do i=1, 20
   a(i) = i
end do
!$omp end parallel do
```

[C]

```
#pragma omp parallel for
#pragma xmp loop on t[i]
for(int i=0;i<20;i++){
    a[i] = i;
}
```

[F]

```
!$omp parallel do
!$xmp loop on t(i)
do i=1, 20
   a(i) = i
end do
!$omp end parallel do
```

指示文の順序は，どちらが先でも構わない

bcast指示文とreduction指示文

- bcast指示文
  - 全ノードに任意のローカル変数を放送する（MPI_Bcastに相当）

[C]

```
#pragma xmp bcast (b)
#pragma xmp bcast (b) from p[2]
```

[F]

```
!$xmp bcast (b)
!$xmp bcast (b) from p(3)
```

- reduction指示文
  - ノード間で集約処理を行う（MPI_Allreduceに相当）

[C]

```
#pragma xmp reduction (+:b)
```

[F]

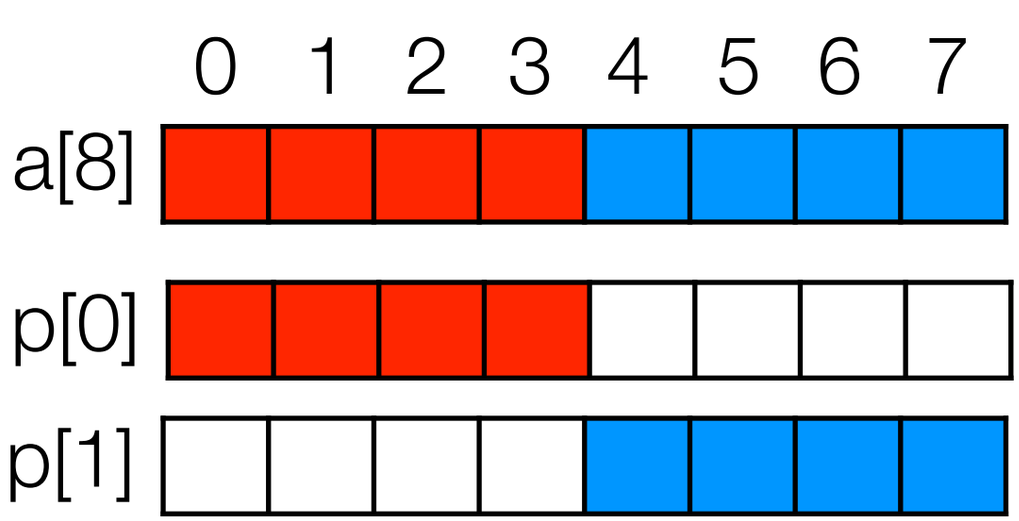
```
!$xmp reduction (+:b)
```

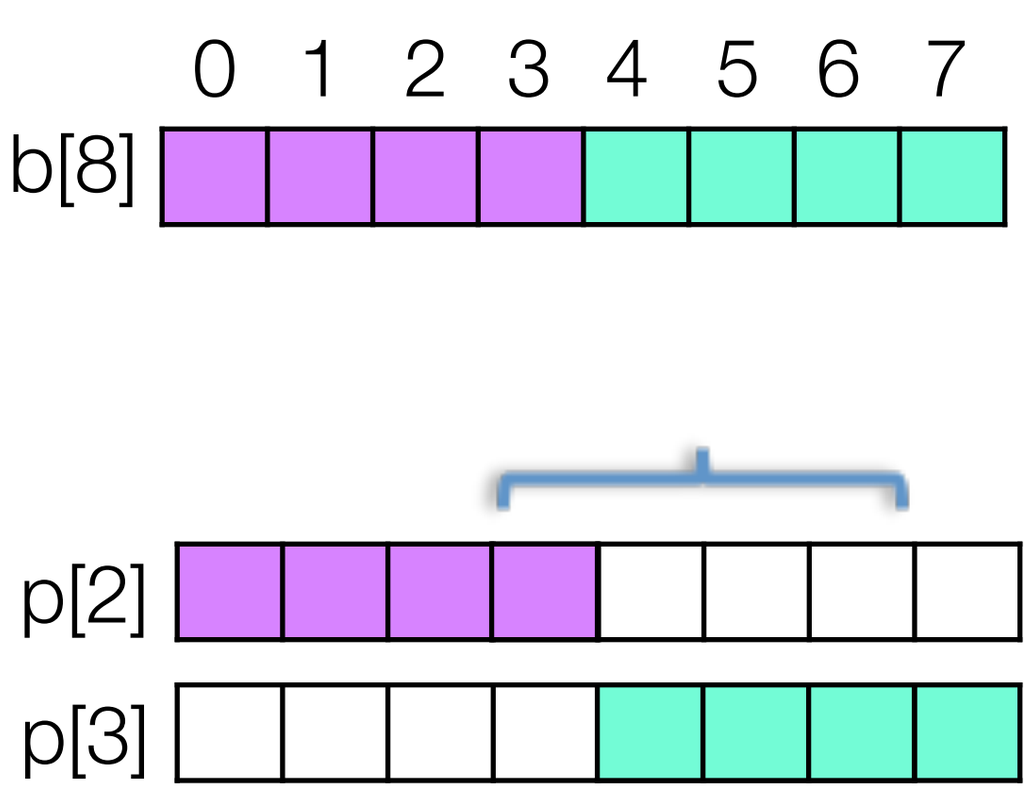
gmove指示文

- 分散配列に対する通信
  - 分散配列のある要素がどのノードが持っているかを知らなくてよい

[C]
```
#pragma xmp gmove
a[2:4] = b[3:4];
```

array-name[*base* : *length*];



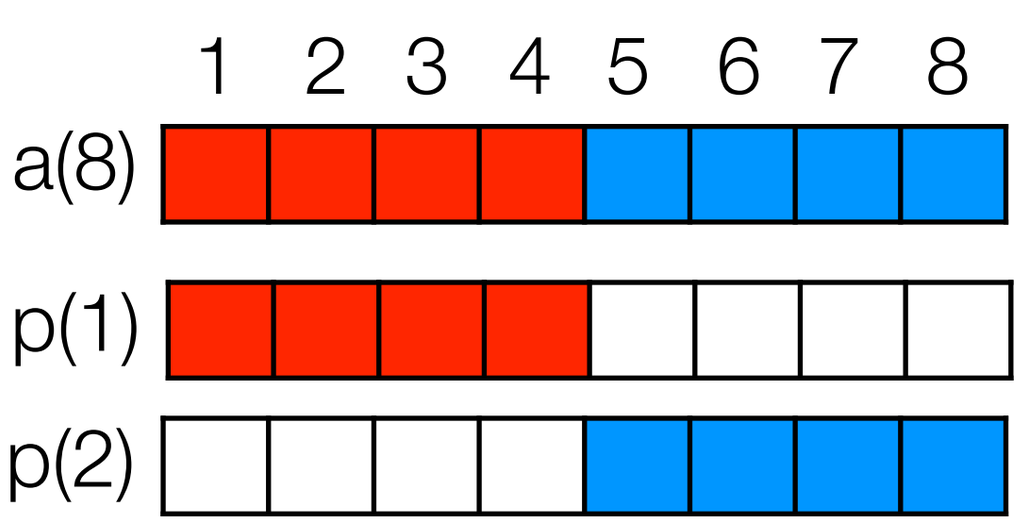


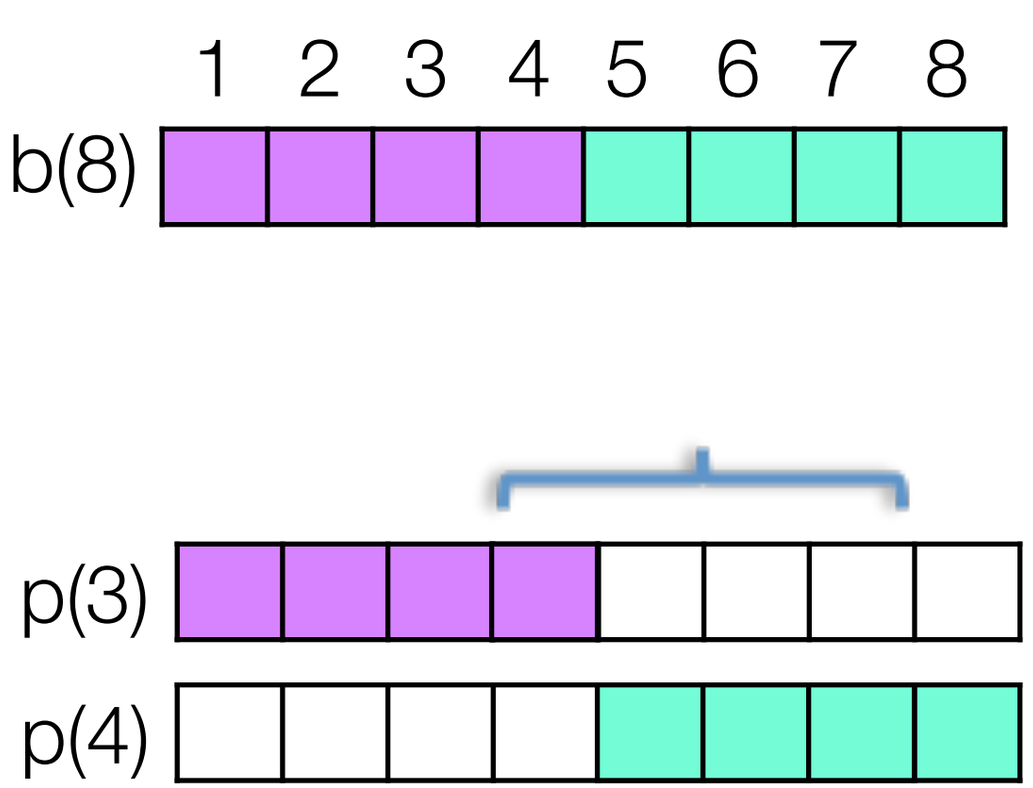
gmove指示文

- 分散配列に対する通信
  - 分散配列のある要素がどのノードが持っているかを知らなくてよい

```
!$xmp gmove                                    [F]
a(3:6) = b(4:7)
```

array-name(base : end *)*





shadow/reflect指示文

- ステンシルアプリケーションを作成するときに便利な機能
- shadow指示文は分散配列に袖領域を追加する
- reflect指示文は隣接ノードが持つ端の値を自ノードにコピーする

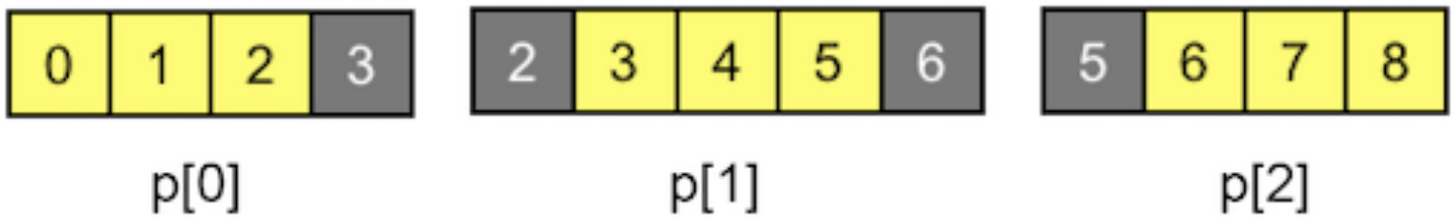
[C]

```
#pragma xmp nodes p[3]
#pragma xmp template t[9]
#pragma xmp distribute t[block] onto p
int a[9];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1:1]
...
#pragma xmp reflect (a)
```

[F]

```
!$xmp nodes p(3)
!$xmp template t(9)
!$xmp distribute t(block) onto p
integer :: a(9)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)
...
!$xmp reflect (a)
```

lower-bound : upper-bound



shadow指示文は
袖領域（灰色のセル）
を追加する

shadow/reflect指示文

- ステンシルアプリケーションを作成するときに便利な機能
- shadow指示文は分散配列に袖領域を追加する
- reflect指示文は隣接ノードが持つ端の値を自ノードにコピーする

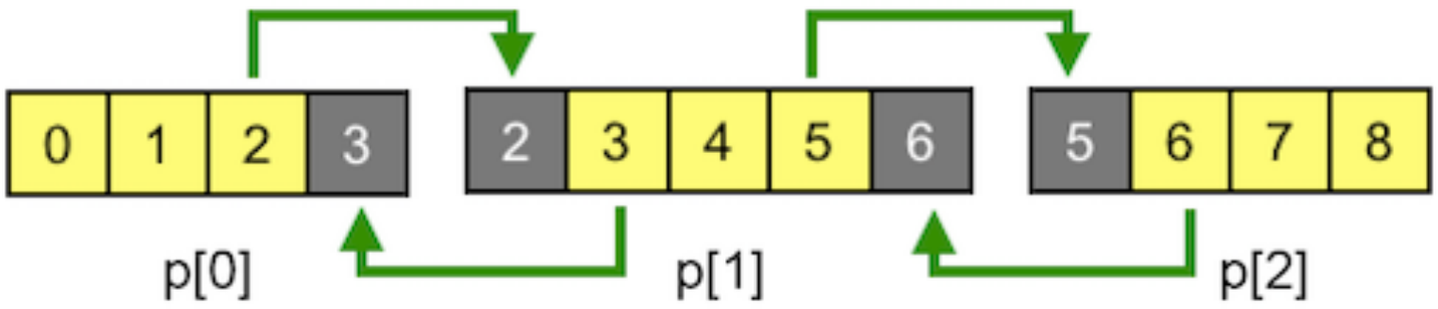
[C]

```
#pragma xmp nodes p[3]
#pragma xmp template t[9]
#pragma xmp distribute t[block] onto p
int a[9];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1:1]
...
#pragma xmp reflect (a)
```

[F]

```
!$xmp nodes p(3)
!$xmp template t(9)
!$xmp distribute t(block) onto p
integer :: a(9)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)
...
!$xmp reflect (a)
```

lower-bound : upper-bound



reflect指示文は隣接ノードの端の値を袖領域にコピーする

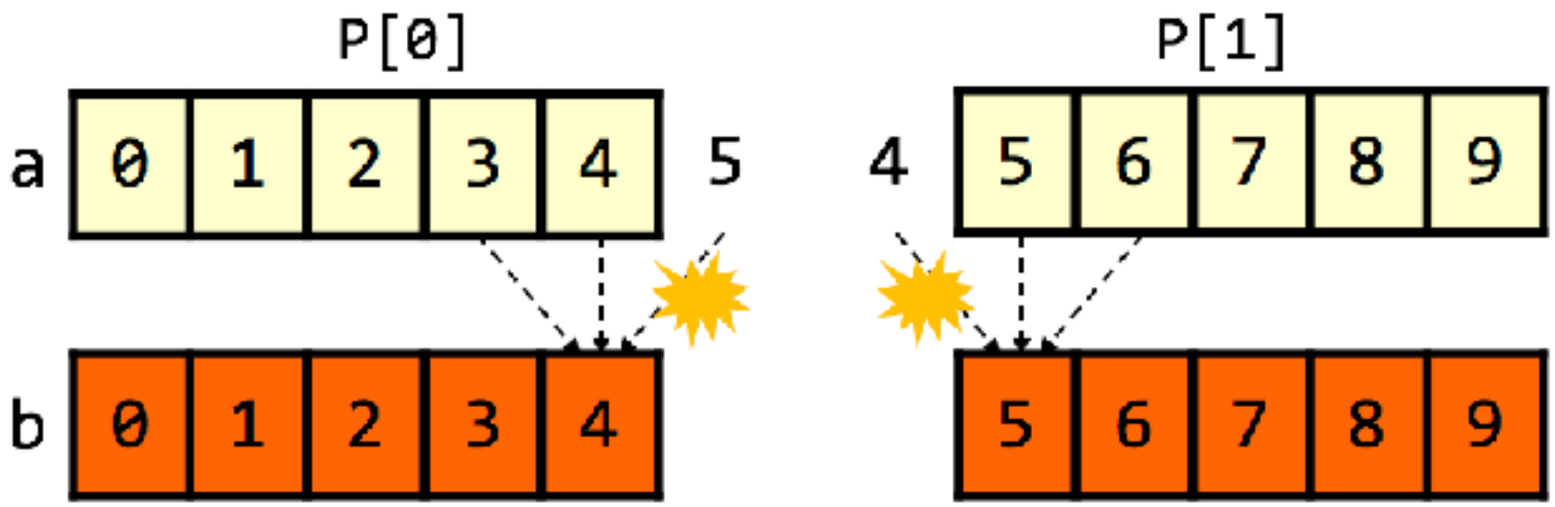
shadow/reflect指示文

[C]

```
#pragma xmp loop on t[i]
for(int i=1;i<9;i++){
  b[i] = a[i-1] + a[i] + a[i+1];
}
```

[F]

```
!$xmp loop on t(i)
do i = 2, 8
  b(i) = a(i-1) + a(i) + a(i+1)
end do
```



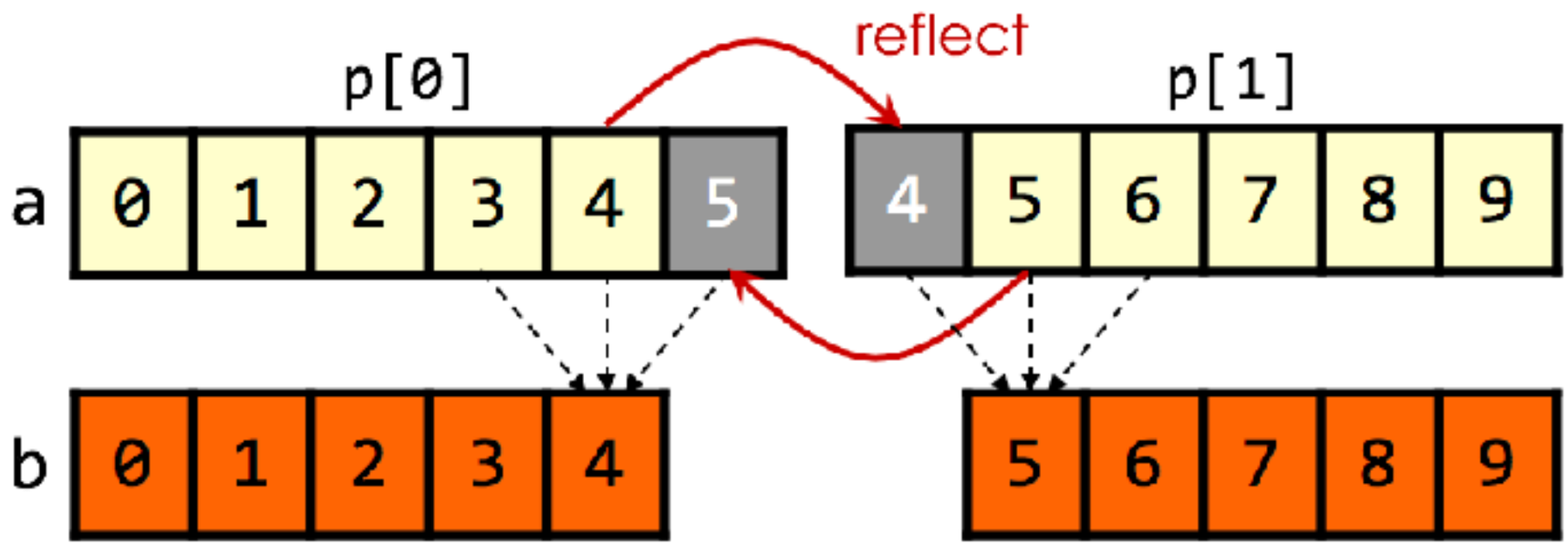
shadow/reflect指示文

[C]

```
#pragma xmp shadow a[1:1]
...
#pragma xmp reflect (a)
#pragma xmp loop on t[i]
for(int i=1;i<9;i++){
  b[i] = a[i-1] + a[i] + a[i+1];
}
```

[F]

```
!$xmp shadow a(1:1)
...
!$xmp reflect (a)
!$xmp loop on t(i)
do i = 2, 8
  b(i) = a(i-1) + a(i) + a(i+1)
end do
```



ローカルビュープログラミング

- グローバルビュープログラミングは「解くべき問題全体を記述し，
  それを各ノードが分担する方法を示す」
  - 例：問題1〜100を4人で分担して解け

- ローカルビュープログラミングは「各ノードが解くべき問題を
  個別に示す」（MPIはローカルビュー）
  - 例：ノード1は問題1〜25を解け．ノード2は……

- 自由度が高い分，グローバルビュープログラミングと比較してやや難しい

- ローカルビュープログラミングのための機能として，Fortran 2008から
  導入したcoarray記法をXMP/Cでもサポート

Coarray記法 in XMP/C

- Coarrayに対して他ノードからの片側通信（Put/Get）を実行できる

Fortran2008

```
real a(8)
real b(8)[*]

if(this_image() == 1) then
  b(6)[2] = b(4)
  a(0) = b(3)[2]
end if

sync all
```

b()をcoarrayとして宣言

image 1は b(4)をimage 2の b(6)にPutする

image 1はimage 2の b(3)を a(0)にGetする

同期（今までに実行した片側通信の終了を待つ）

XMP/C

```
double a[8];
double b[8]:[*];

if(xmpc_this_image() == 1){
  b[6]:[2] = b[4];
  a[0] = b[3]:[2];
}

xmpc_sync_all(NULL);
```

部分配列 in XMP/C

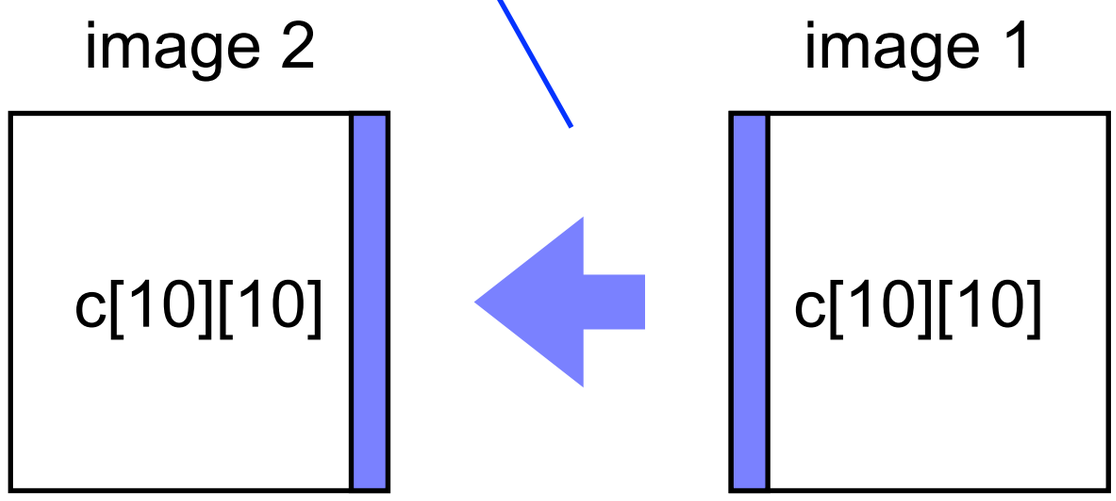
- XMP/Cにおいて複数の要素に対する片側通信のために，部分配列を導入
- Intel Cilk や OpenACCと同等

[C]

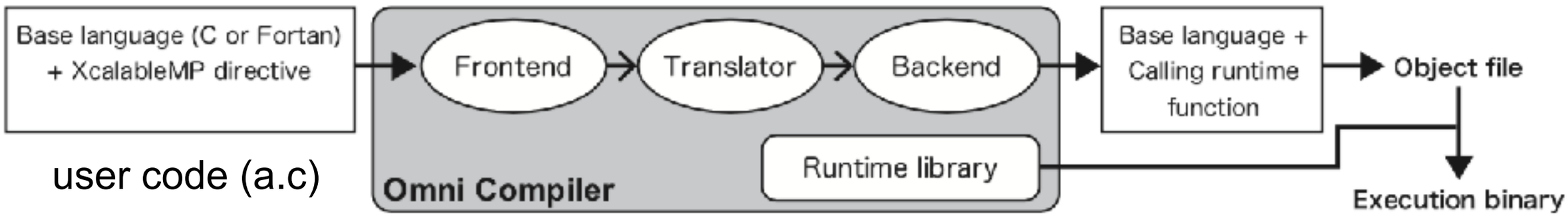
```
if(xmpc_this_image() == 1){
  b[10:5]:[2] = b[0:5];
  a[:]:[2] = b[:];
  c[:][9]:[2] = c[:][0];
}
```

b[0]-b[4]の5要素はimage 2の
b[10]-a[14]にPutされる

配列bの全要素はimage 2の配列aに
Putされる



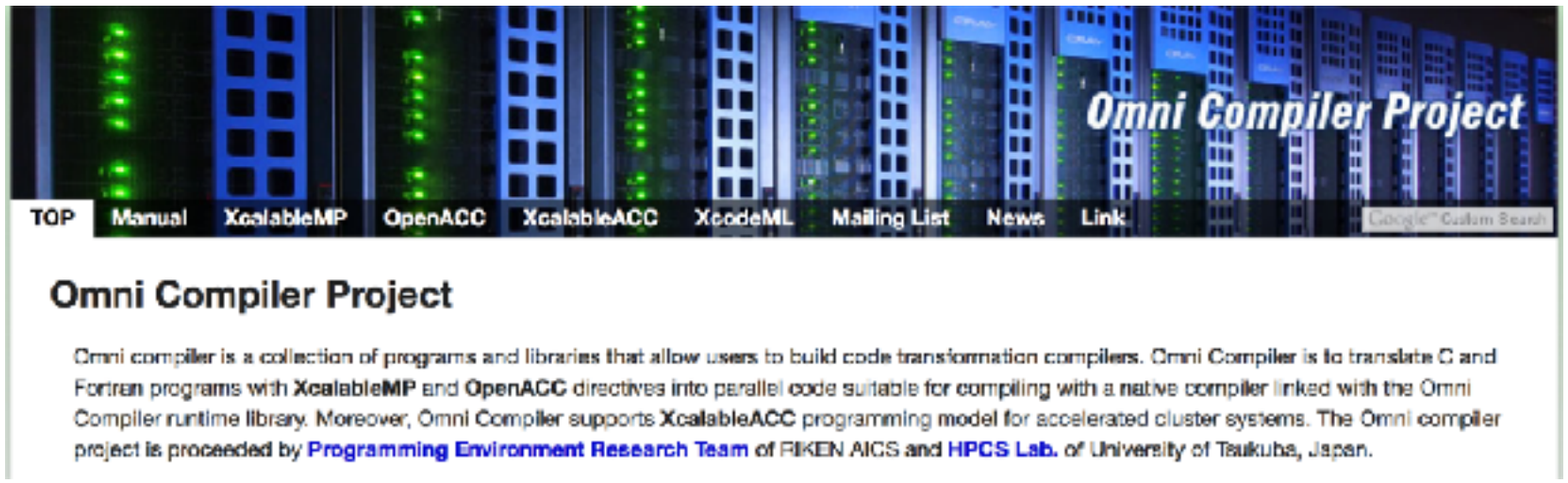
Omni Compiler




```
$ xmpcc a.c -o a.out
```

- XMP指示文を含むユーザコードは，ランタイムの呼び出しに変換する
- ランタイムはCとMPIで実装（つまりMPI対応の並列計算環境で動作する）
- 変換されたコードは，GNU・Intel・PGIなどのネイティブコンパイラで翻訳
- オープンソース

https://omni-compiler.org

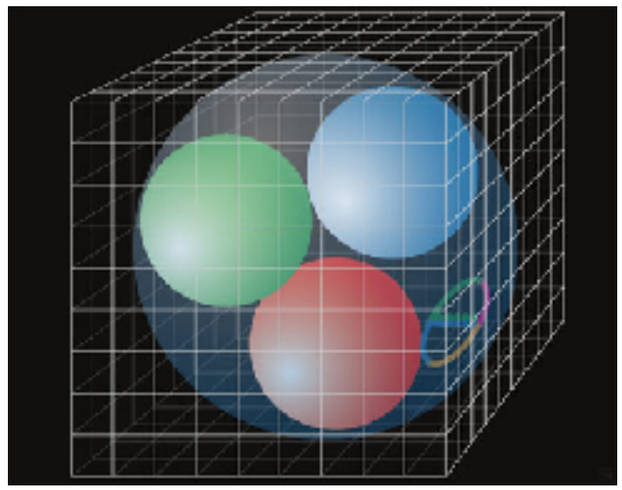


もくじ

- XcalableMP（XMP）の概要説明
- XMPプログラミング
- Lattice QCDの実装

Lattice QCD（LQCD）ミニアプリケーション

- HPC分野で重要なアプリケーションの1つ
- QCD（Quantum Chromo-Dynamics：量子色力学）は物質の最小単位であるクォークと，クォーク間における相互作用を結ぶグルーオンを表す基本方程式



- LQCDは4次元（時間+XYZ軸）の格子上で行うQCDのシミュレーション

- LQCDミニアプリケーション
  - C言語，逐次コード，850行くらい by KEK 松古さん
  - 実アプリケーションBridge++のカーネル部分を抽出

Overview of algorithm

- Pseudo-code (CG method is used)

```
S = B                    // COPY
R = B                    // COPY
X = B                    // COPY
sr = norm(S)             // NORM
T = WD(U,X)              // Main Kernel
S = WD(U,T)              // Main Kernel
R = R - S                // AXPY
P = R                    // COPY
rrp = rr = norm(R)       // NORM
do{
  T = WD(U,P)            // Main Kernel
  V = WD(U,T)            // Main Kernel
  pap = dot(V,P)         // DOT
  cr = rr/pap
  X = cr * P + X         // AXPY
  R = -cr * V + R        // AXPY
  rr = norm(R)           // NORM
  bk = rr/rrp
  P = bk * P             // SCAL
  P = P + R              // AXPY
  rrp = rr
}while(rr/sr > 1.E-16)
```

WD() is the Wilson-Dirac operator

$$D_{x,y} = \delta_{x,y} - \kappa \sum_{\mu=1}^{4} \{(1-\gamma_\mu) U_\mu(x) \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U_\mu^\dagger(x-\hat{\mu}) \delta_{x-\hat{\mu},y}\}$$

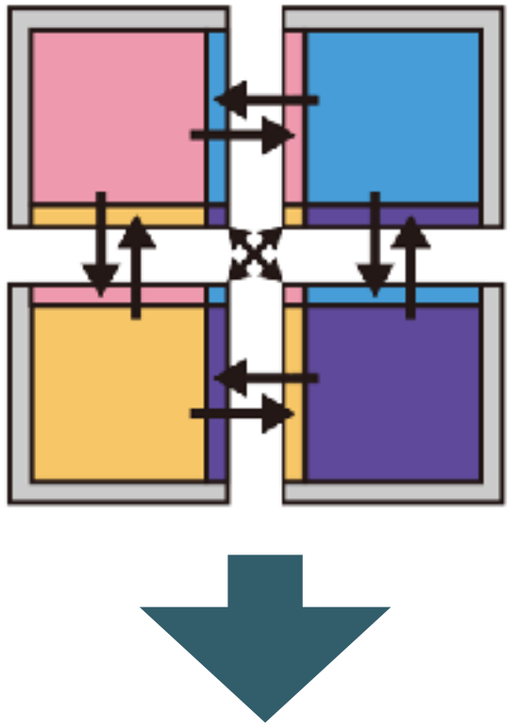
- Main kernel (most costly)
- Stencil calculation

```
#pragma xmp reflect (X) width(/periodic/..) orthogonal
WD(X, ...);
```

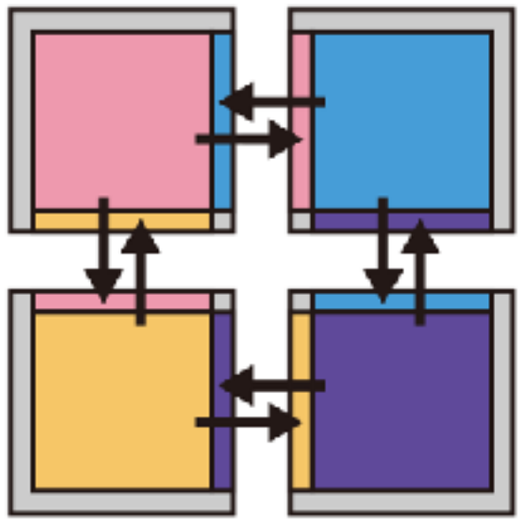
```
void WD(Quark_t X[NT][NZ][NY][NX], ... ){
    :
#pragma xmp loop on t[t][z]
#pragma omp parallel for collapse(4)
for(int t=0;t<NT;t++)
  for(int z=0;z<NZ;z++)
    for(int y=0;y<NY;y++)
      for(int x=0;x<NX;x++){
                :
```

reflect指示文

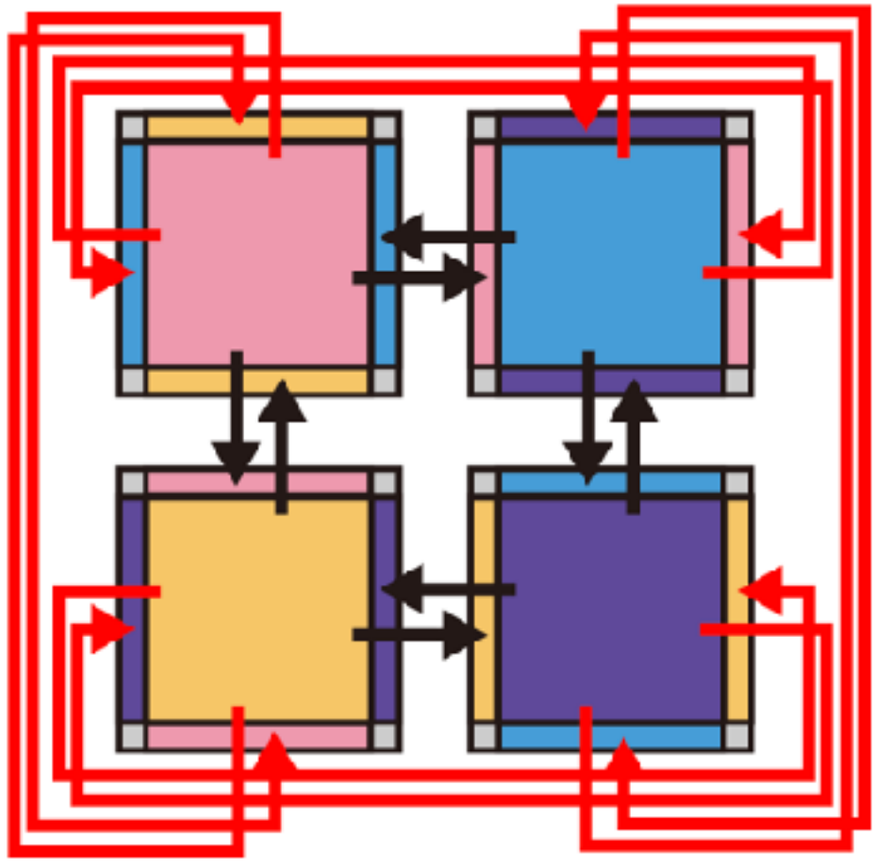
#pragma xmp reflect (a)



#pragma xmp reflect (a) orthogonal



#pragma xmp reflect (a) width(/periodic/1, ..) ¥
orthogonal



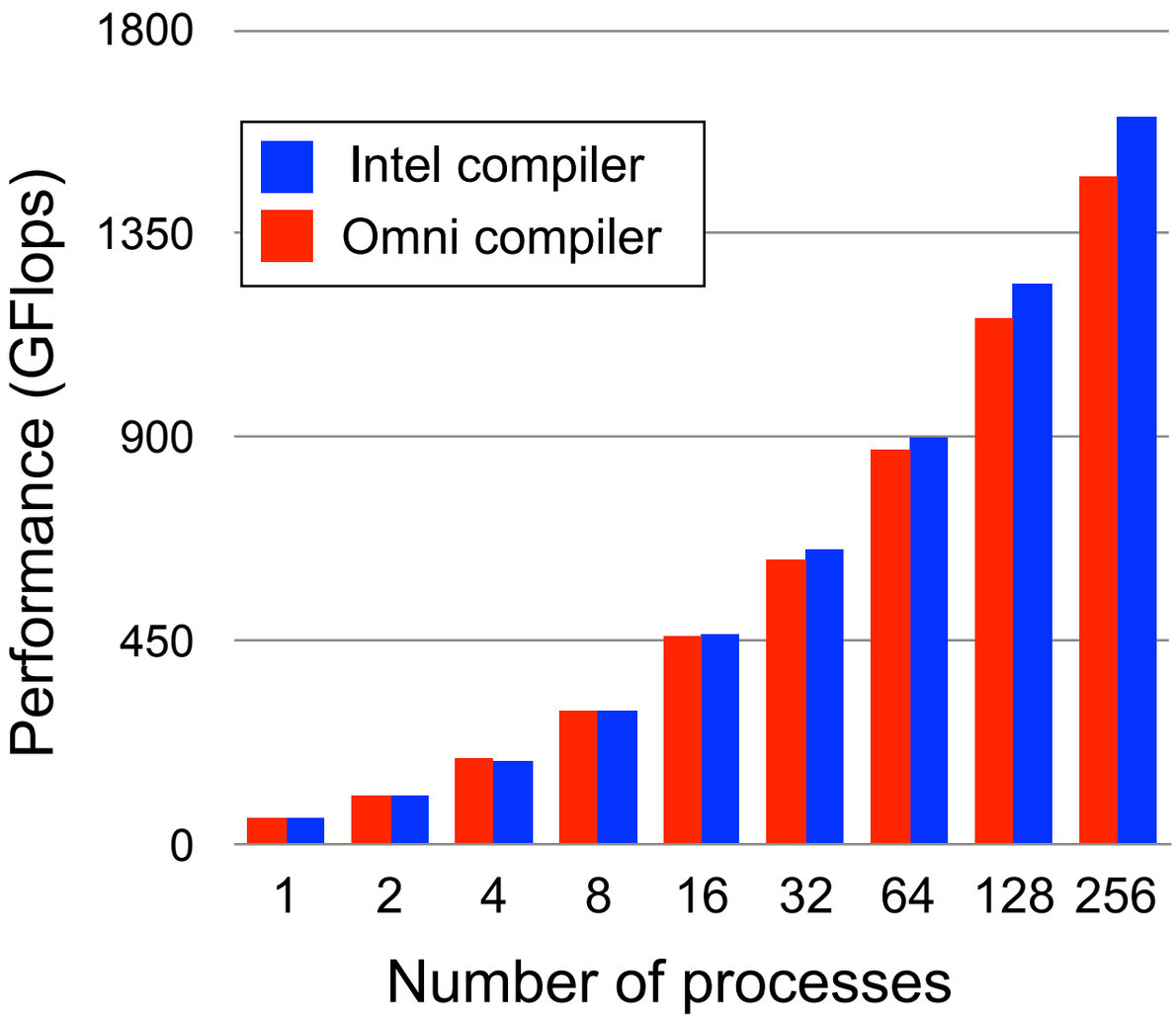
（orthogonalは実装中）

Performance Evaluation on cluster

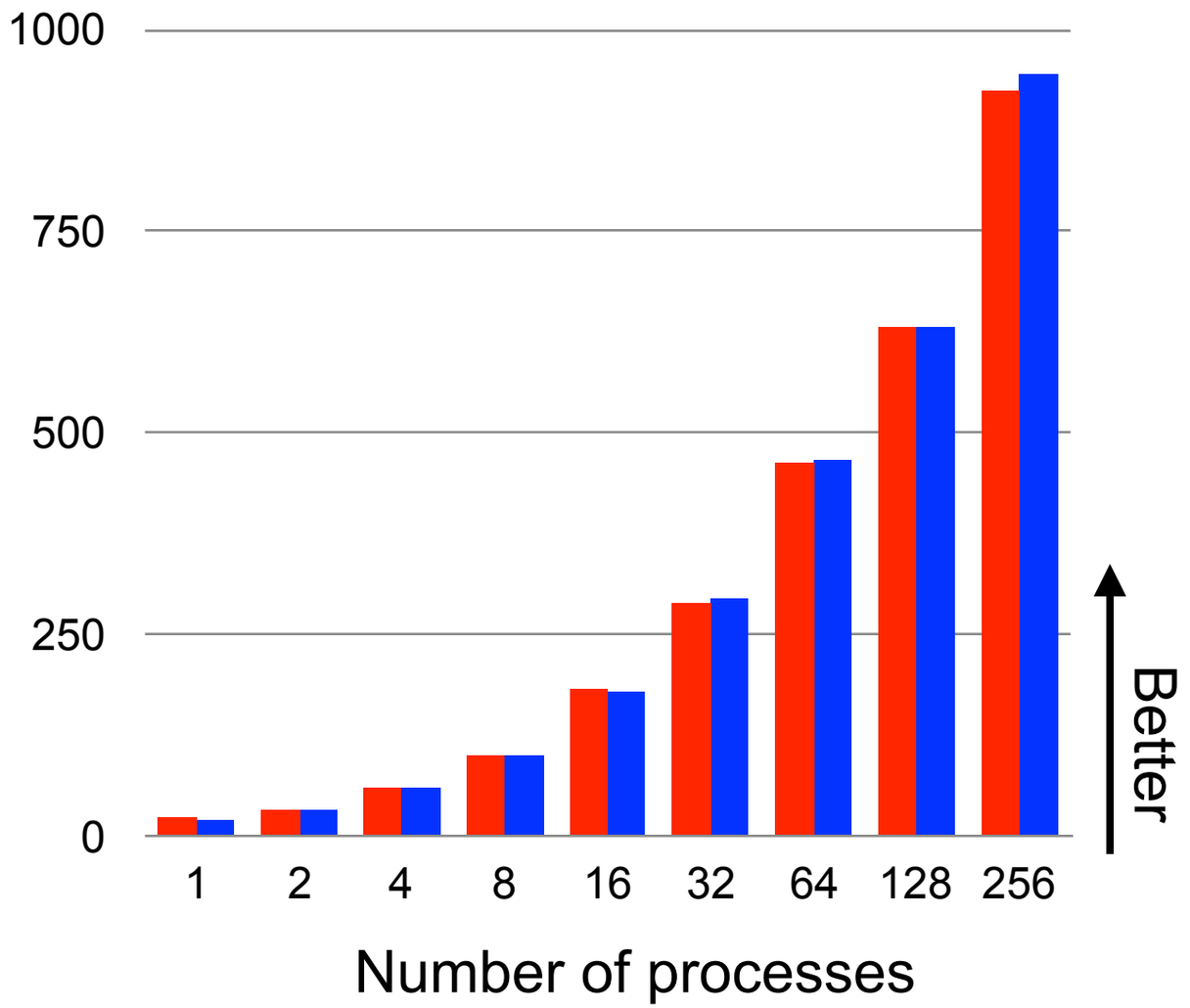
- Oakforest-PACS and COMA
  - One process per compute node on Oakforest-PACS
  - Two processes per compute node on COMA because it has two CPU sockets
- Problem size is (32,32,32,32) as (NT,NZ,NY,NX) with strong scaling

Performance Evaluation

- Oakforest-PACS



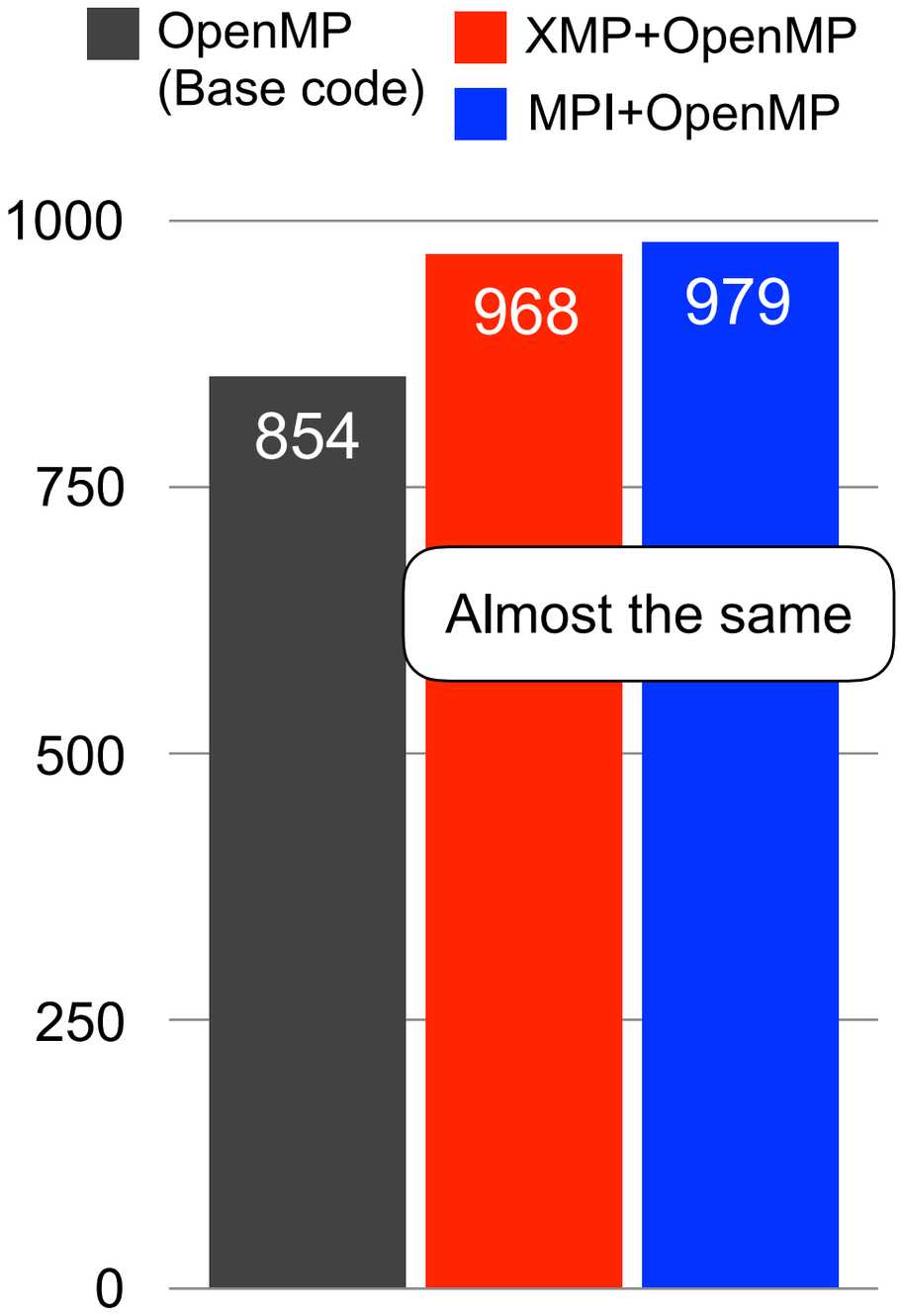
- COMA



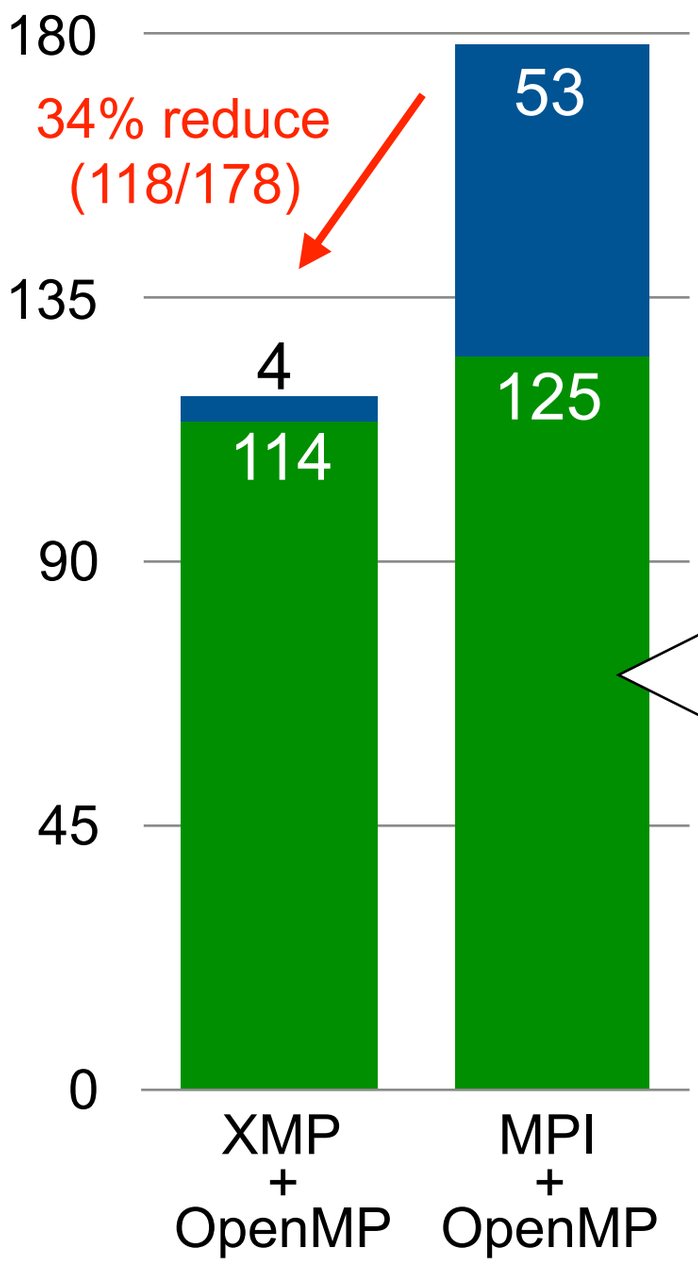
Performances of Omni compiler achieve 94 - 105% of those of Intel compiler.

Productivity Evaluation

- Source lines of codes (SLOC)



- Delta SLOC How many lines the code changed from a base code to a parallel code



Modification
Addition

← **Quantitative**

Qualitative ↓

While most of 114 lines in XMP+OpenMP is the insertion of XMP directives,
125 in MPI+OpenMP is a creation of new functions for communication.

It is easier to develop a parallel application in XMP+OpenMP than MPI+OpenMP

まとめ

- 並列言語XMPの紹介
- Lattice QCDの実装と評価
  - 十分な性能と高い生産性を達成
  - 今回紹介しませんでしたが，色々なベンチマークやミニアプリも実装しています．その内のいくつかは，下記のGitHubで公開中
    - https://github.com/omni-compiler/XMP-Applications
- 色々なXMPの拡張
  - PythonのためのXMPモジュール
  - アクセラレータ用言語OpenACCとの連携
    - XMP + OpenACC + α = XcalableACC
    - アクセラレータを搭載したクラスタシステム用言語
    - アクセラレータ間の直接通信にも対応