XcalableMPによる NAS Parallel Benchmarksの実装と評価

○中尾 昌広[†]，李 珍泌[‡]，朴 泰祐[†‡]，佐藤 三久[†‡]

†筑波大学 計算科学研究センター

‡筑波大学大学院 システム情報工学研究科

研究背景

- 大規模な演算を行うためには，
  分散メモリ型システムの利用が必須

- Message Passing Interface（MPI）
  - 並列プログラムの大半はMPIを利用
  - 様々な実装：OpenMPI, MPICH, MVAPICH, MPI.NET
  - プログラミングコストが高いため，生産性が悪い

新しい並列プログラミングモデルとして
XcalableMPが提案されている

XcalableMP

- 分散メモリ型システム用の並列プログラミングモデル
- OpenMPのように指示文を用いた並列化 + α
- 科学技術計算でよく用いられるCとFortran言語に対応
- プログラミングコストを低減し，生産性を上げる

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

指示文により，
ループ文を分散して
各ノードで処理可能

XcalableMPのプログラム例

発表内容

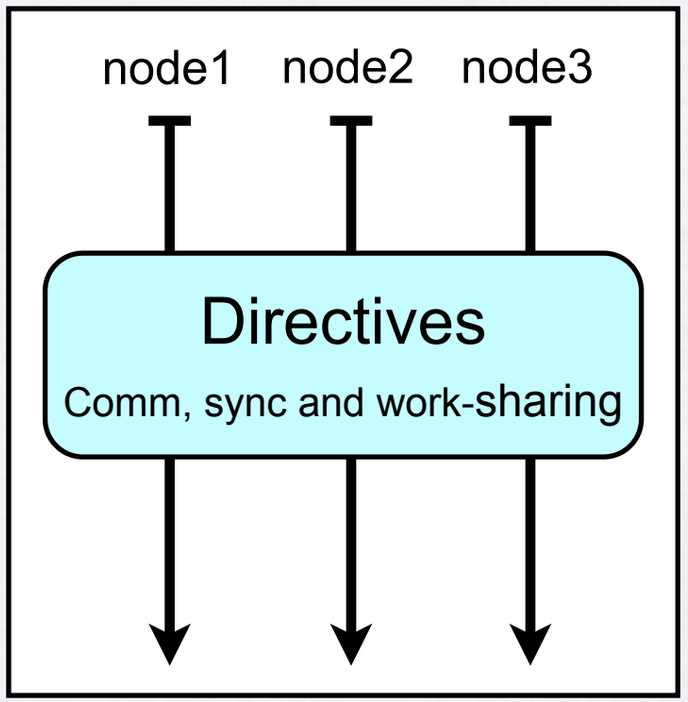
- 研究目的
  - XcalableMP（XMP）の記述性と性能を明らかにする
- 研究内容
  - NAS Parallel Benchmarks（NPB）をXMPで実装し，性能評価を行う
    - Embarrassingly Parallel（EP）：乱数発生
    - Integer Sort（IS）：整数ソート
    - Conjugate Gradient（CG）：共役勾配法

この後の発表の流れ

- XMPの概要と文法
- XMPによるNPBの実装
- 性能測定
- まとめ

XMPの概要

- 実行モデルはSingle Program Multiple Data
- High Performance Fortranなどを参考に開発
- Performance Awareness
  - 通信が発生する箇所は明示的に指示
  - それ以外はローカルメモリにアクセス
- 2つのプログラミングモデル
  - グローバルビュー
    定型的な通信（集団通信，同期など）
  - ローカルビュー
    （MPI_Put/Getのような）片方向通信の記述

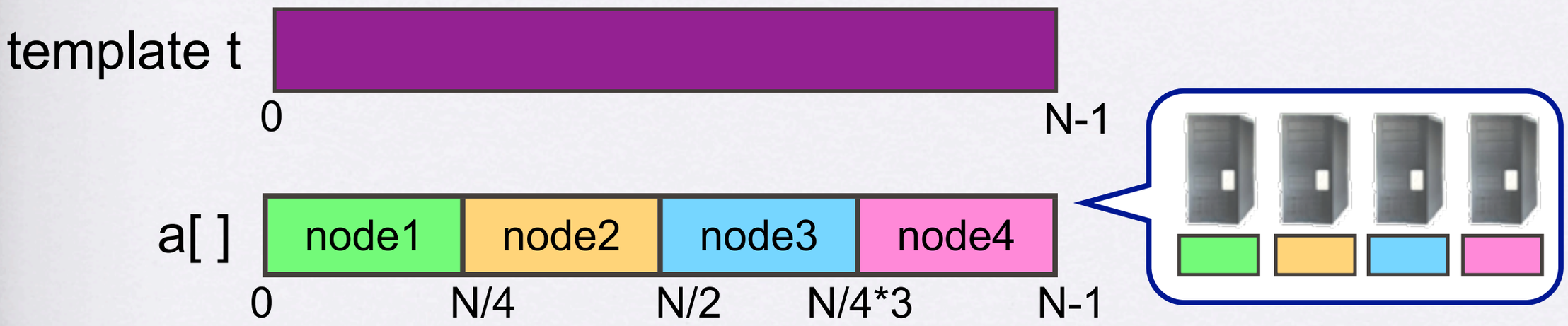


グローバルビュー（template）

データ分散とループ文処理のために用いる仮想的なindex空間

```
#pragma xmp loop on t(i)
for( i = 0; i < N; i++)
    a[ i ] = func( i );
```

```
#pragma xmp nodes n(4)                  // ノード集合を定義（今回は4ノード）
#pragma xmp template t(0:N-1)           // 0からN-1のindexを持つtemplateを作成
#pragma xmp distribute t(block) onto n  // templateをノード集合nに分配する
#pragma xmp align a[i] with t(i)        // 配列aをtemplateに整列する
```

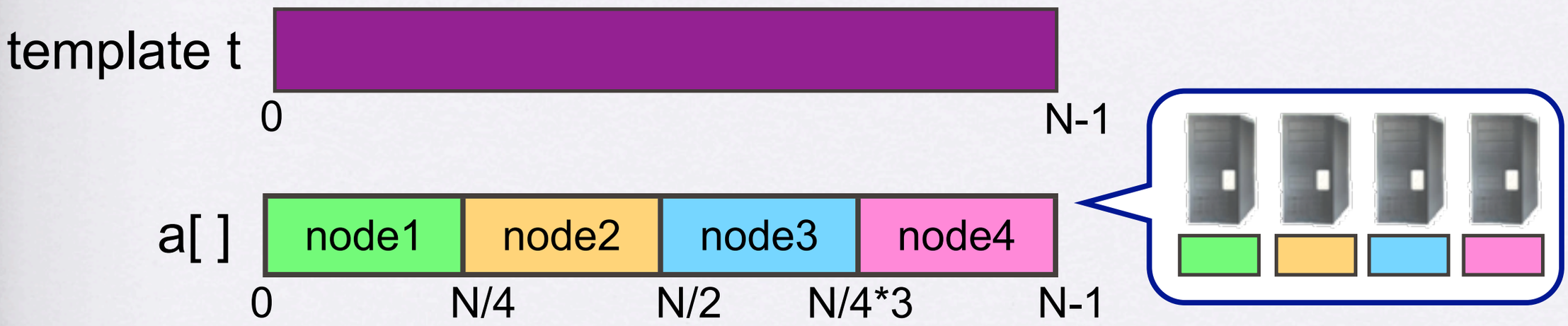


グローバルビュー（template）

データ分散とループ文処理のために用いる仮想的なindex空間

```
#pragma xmp loop on t(i)
for( i = 0; i < N; i++)
    a[ i ] = func( i );
```

```
#pragma xmp nodes n(4)              // ノード集合を定義（今回は4ノード）
#pragma xmp template t(0:N-1)       // 0からN-1のindexを持つtemplateを作成
#pragma xmp distribute t(block) onto n  // templateをノード集合nに分配する
#pragma xmp align a[i] with t(i)    // 配列aをtemplateに整列する
```

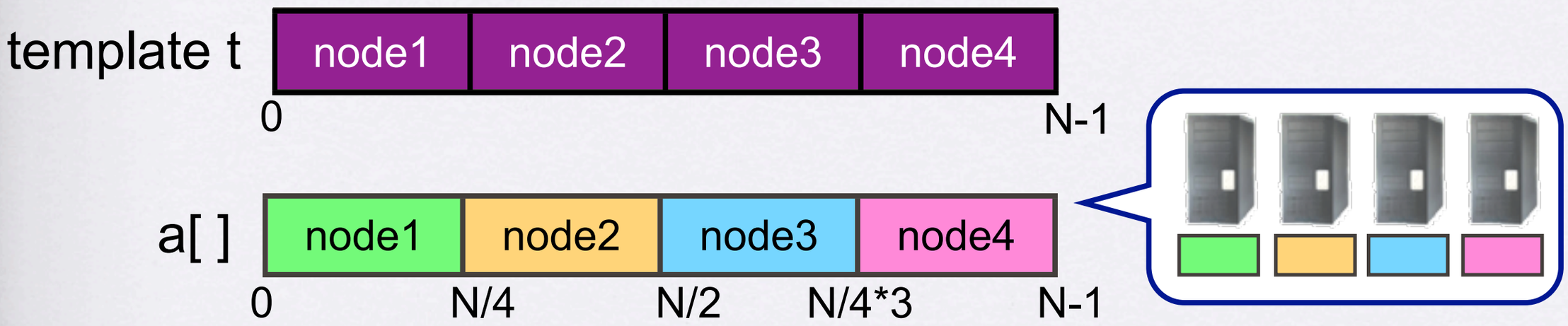


グローバルビュー（template）

データ分散とループ文処理のために用いる仮想的なindex空間

```
#pragma xmp loop on t(i)
for( i = 0; i < N; i++)
    a[ i ] = func( i );
```

```
#pragma xmp nodes n(4)                  // ノード集合を定義（今回は4ノード）
#pragma xmp template t(0:N-1)           // 0からN-1のindexを持つtemplateを作成
#pragma xmp distribute t(block) onto n  // templateをノード集合nに分配する
#pragma xmp align a[i] with t(i)        // 配列aをtemplateに整列する
```

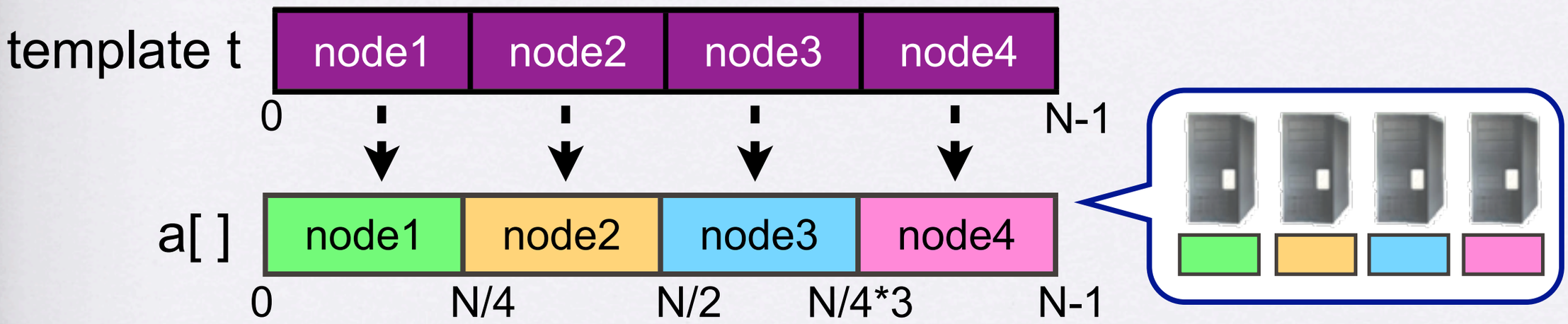


グローバルビュー（template）

データ分散とループ文処理のために用いる仮想的なindex空間

```
#pragma xmp loop on t(i)
for( i = 0; i < N; i++)
    a[ i ] = func( i );
```

```
#pragma xmp nodes n(4)                    // ノード集合を定義（今回は4ノード）
#pragma xmp template t(0:N-1)             // 0からN-1のindexを持つtemplateを作成
#pragma xmp distribute t(block) onto n    // templateをノード集合nに分配する
#pragma xmp align a[i] with t(i)          // 配列aをtemplateに整列する
```



分散された配列を1つの配列のように扱うことが可能

グローバルビューの通信

- 集団通信
  - Reduce, Broadcast, Gatherなど
- gmove
  - 分散配列のための代入指示文
- シャドウ
  - 袖領域のための通信

シャドウについては論文集を参考にして下さい

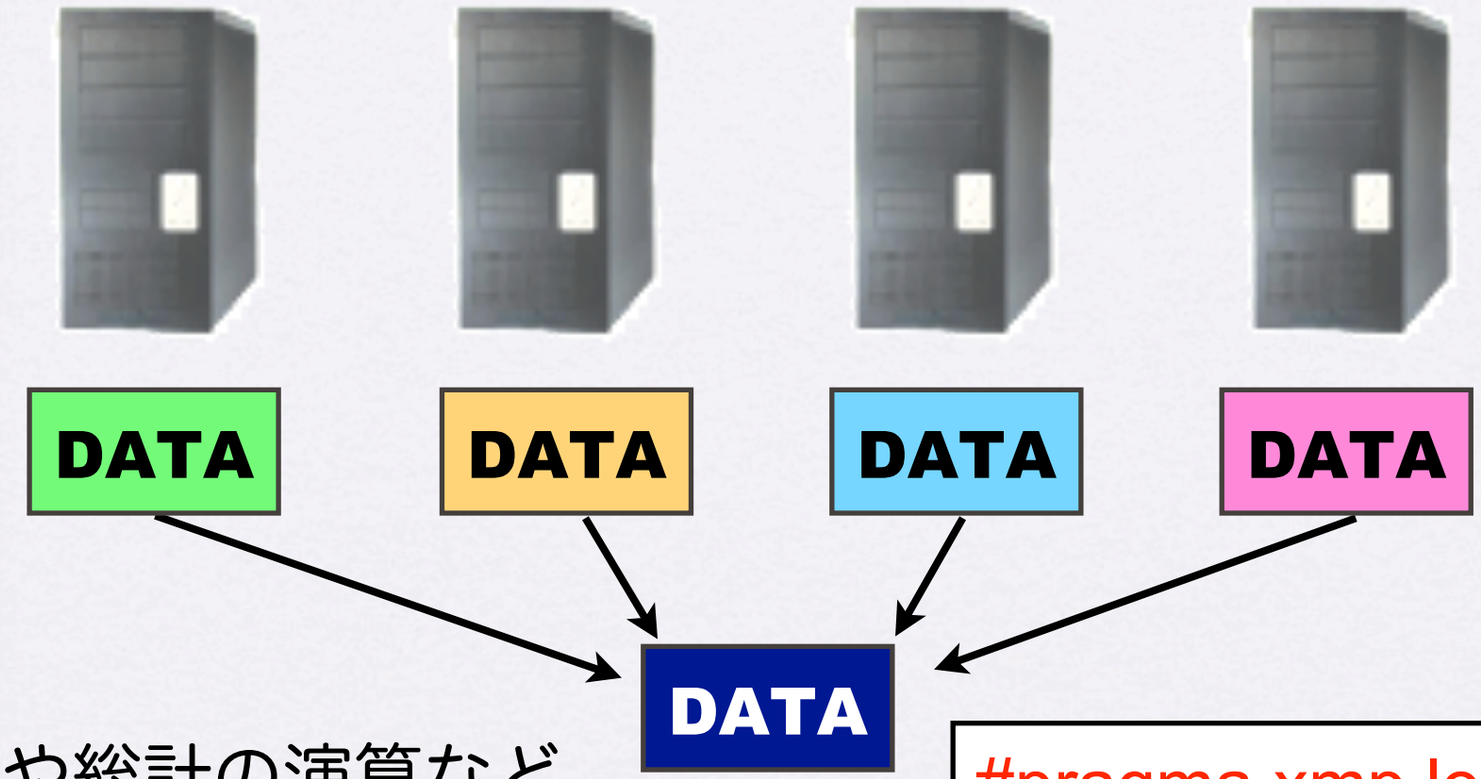
グローバルビューの通信

- 集団通信
  - Reduce, Broadcast, Gatherなど
- gmove
  - 分散配列のための代入指示文
- シャドウ
  - 袖領域のための通信

シャドウについては論文集を参考にして下さい

グローバルビュー（集団通信）

リダクション（集約）



最大値や総計の演算など

MPI_Allreduceと等価

```
#pragma xmp loop on t(i)
  for(i = 0, sum=0; i < N; i++){
      sum += array[i];
    }
#pragma xmp reduction (+:sum)
```

グローバルビュー（ gmove）

各ノードに分割された配列の代入文

分割された配列b[]の要素すべてを
分割された配列a[]に代入したい

```
#pragma xmp gmove
a[:] = b[:];
```

array[*lower* : *upper*]

配列arrayの要素 *lower*から *upper* までが処理対象

array[:]の場合は，すべての要素を処理対象とする意味

どのデータがどのノードに配置されているかを，意識する必要がなく，通信を用いたコピーを行える

グローバルビュー（ gmove）

各ノードに分割された配列の代入文

分割された配列b[]の要素すべてを
分割された配列a[]に代入したい

```
#pragma xmp gmove
a[:] = b[:];
```

array[*lower* : *upper*]

配列arrayの要素 *lower*から *upper* までが処理対象

array[:]の場合は，すべての要素を処理対象とする意味

どのデータがどのノードに配置されているかを，
意識する必要がなく，通信を用いたコピーを行える

ローカルビュー

- ローカルデータとノード間通信を意識したプログラミング
- XMPではローカルビューとしてCo-array記法を導入し，片側通信を実現
- Fortran版のXMPはCo-Array Fortranと互換（C言語では文法を拡張）

```
#pragma xmp coarray b
 ・ ・ ・
a[0:3] = b[3:6]:[1];
```

配列の次元を拡張（ノード番号を表す）

ノード1が持つb[3:6]のデータを
a[0:3]に代入

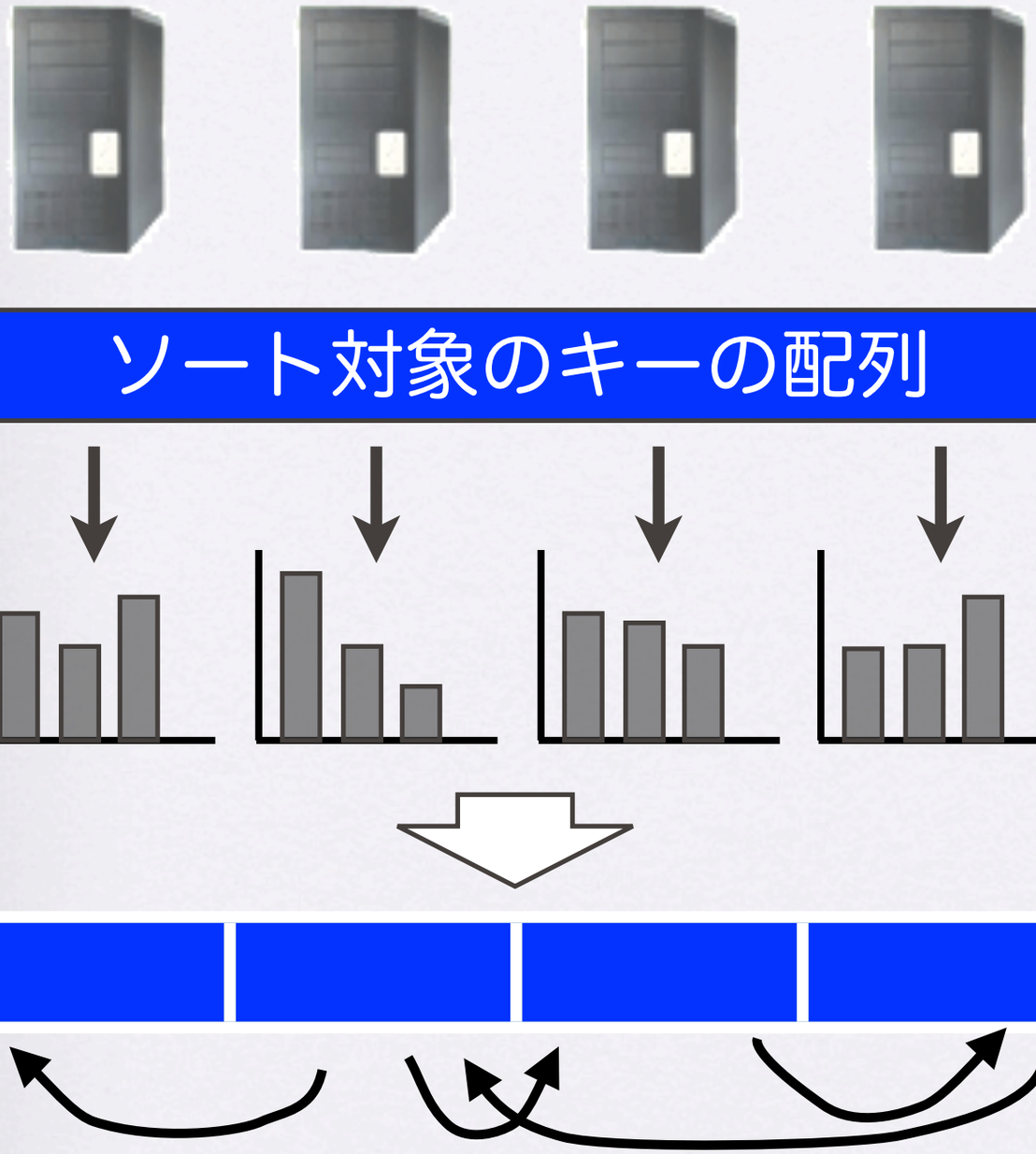
より柔軟な並列アルゴリズムの記述が可能

XMPによるNPB実装

- NAS Parallel Benchmarks（NPB）をXMPで実装
- 並列化の方法
  - MPI版とOpenMP版におけるNPBの並列アルゴリズムをXMPで実装
- 対象問題
  - Embarrassingly Parallel（EP）：乱数発生
  - Integer Sort（IS）：整数ソート
  - Conjugate Gradient（CG）：共役勾配法

OpenMPを参考にした実装やEPについては論文集を参考にして下さい

MPI版のISの概要


ソート対象のキーの配列
7 1 8 2 4 6 5
3 0 9 4 7 2 6
ヒストグラムを作成 + ソート
値の入れ替え + ソート
値が小さいキーほど左側のノードへ

IS-XMPのソース

key_array[]は分散された配列

```
#pragma xmp coarray key_buff2                          （coarrayを宣言）
 ・ ・ ・
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ )                            （ヒストグラムを作成）
     bucket_size[key_array[i] >> shift]++;
 ・ ・ ・
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ ) {                          （ソート）
     key = key_array[i];
     key_buff1[bucket_ptrs[key >> shift]++] = key;
}
 ・ ・ ・                                               （値の受信位置の計算）

for(i=0;i<NUM_PROCS;i++)                               （値の入れ替え）
     key_buff2[a[i]:b[i]]:[i] = key_buff1[c[i]:d[i]];
```

IS-XMPのソース

key_array[]は分散された配列

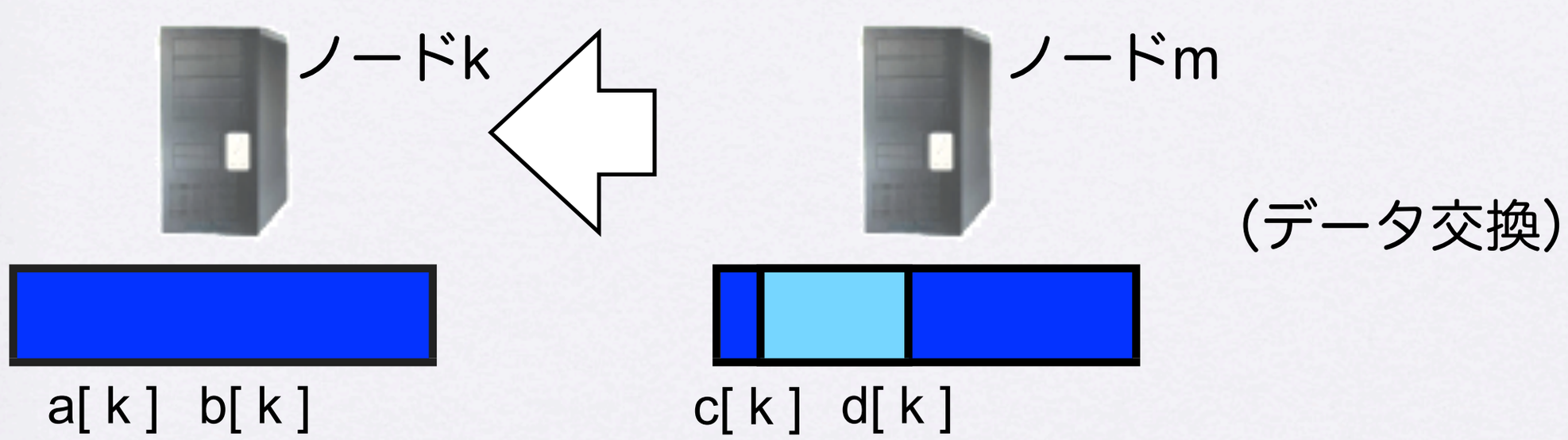
```
#pragma xmp coarray key_buff2                          （coarrayを宣言）
 ・ ・ ・
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ )                            （ヒストグラムを作成）
     bucket_size[key_array[i] >> shift]++;
 ・ ・ ・
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ ) {                          （ソート）
     key = key_array[i];
     key_buff1[bucket_ptrs[key >> shift]++] = key;
}
 ・ ・ ・                                               （値の受信位置の計算）

for(i=0;i<NUM_PROCS;i++)                               （値の入れ替え）
     key_buff2[a[i]:b[i]]:[i] = key_buff1[c[i]:d[i]];
```

Co-array詳細

・・・　　（値の受信位置の計算）

```
for(i=0;i<NUM_PROCS;i++)
    key_buff2[ a[ i ] : b[ i ] ] : [ i ] = key_buff1[ c[ i ] : d[ i ] ];
```

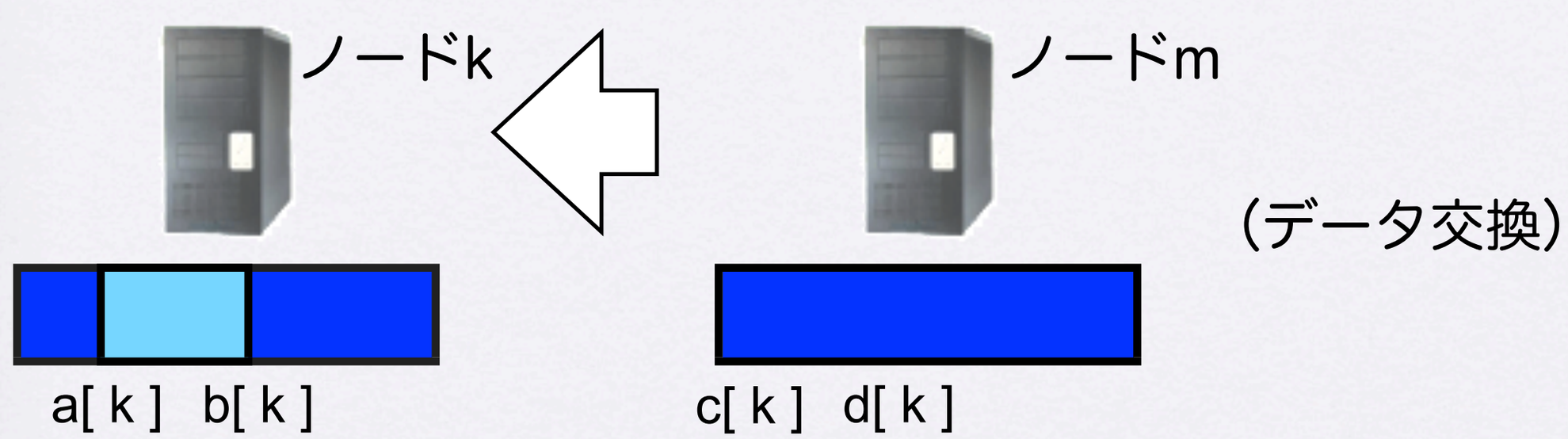


受信位置（a[]とb[]）の計算を行う理由は，片側通信の場合，
送信データと受信データの両方の位置の情報が必要なため
（ MPI版のISでは集団通信であるAlltoallvのみを利用している）

Co-array詳細

・・・　　（値の受信位置の計算）

```
for(i=0;i<NUM_PROCS;i++)
    key_buff2[ a[ i ] : b[ i ] ] : [ i ] = key_buff1[ c[ i ] : d[ i ] ];
```



受信位置（a[]とb[]）の計算を行う理由は，片側通信の場合，
送信データと受信データの両方の位置の情報が必要なため
（ MPI版のISでは集団通信であるAlltoallvのみを利用している）

実験方法と環境

XMP版とMPI版のNPBとの性能比較を行う（問題サイズはB）

PC Cluster System

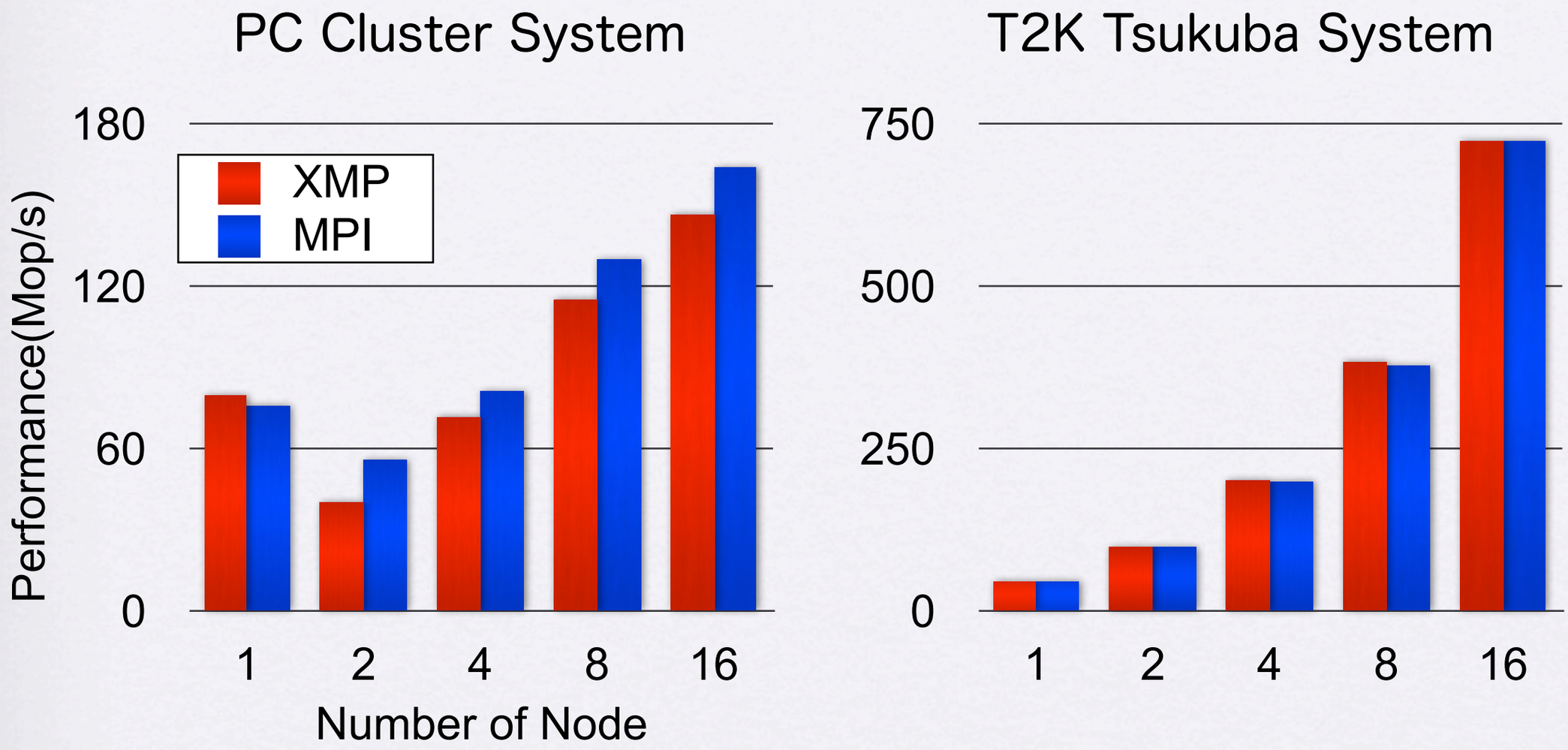
Intel Core2 Quad 3.0GHz
Gigabit Ethernet

T2K Tsukuba System

AMD Opteron Quad 2.3GHz
Infiniband DDR（× 4rails）

ネットワークのオーバヘッドを計測するために,
1ノード1プロセスで実験を行った

実験結果（IS）



XMPの方が10%程度性能は低い

値の交換に用いる方法の違いのため

XMPとMPIはほぼ同じ性能

CG（共役勾配法）のXMP化

2次元疎行列

a[ ] → CG（共役勾配法）→ 最小固有値

プロセスは2次元に配置

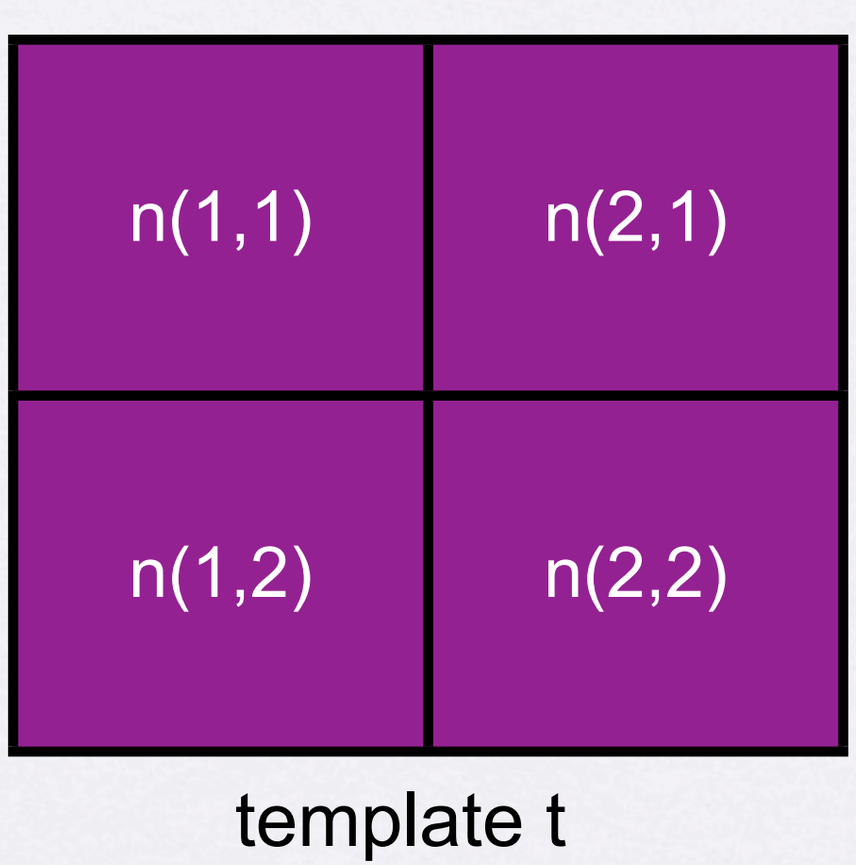
```
#pragma xmp template t(0:N-1,0:N -1)
#pragma xmp distribute t(block,block) on n

double x[N], z[N], p[N], q[N], r[N], w[N];

#pragma xmp align [i] with t(i,*):: x,z,p,q,r
#pragma xmp align [i] with t(*,i):: w
```

4プロセスの例　x[], z[], p[], q[], r[]

w[]



CG（共役勾配法）のXMP化

2次元疎行列

a[ ] → CG（共役勾配法）→ 最小固有値

プロセスは2次元に配置

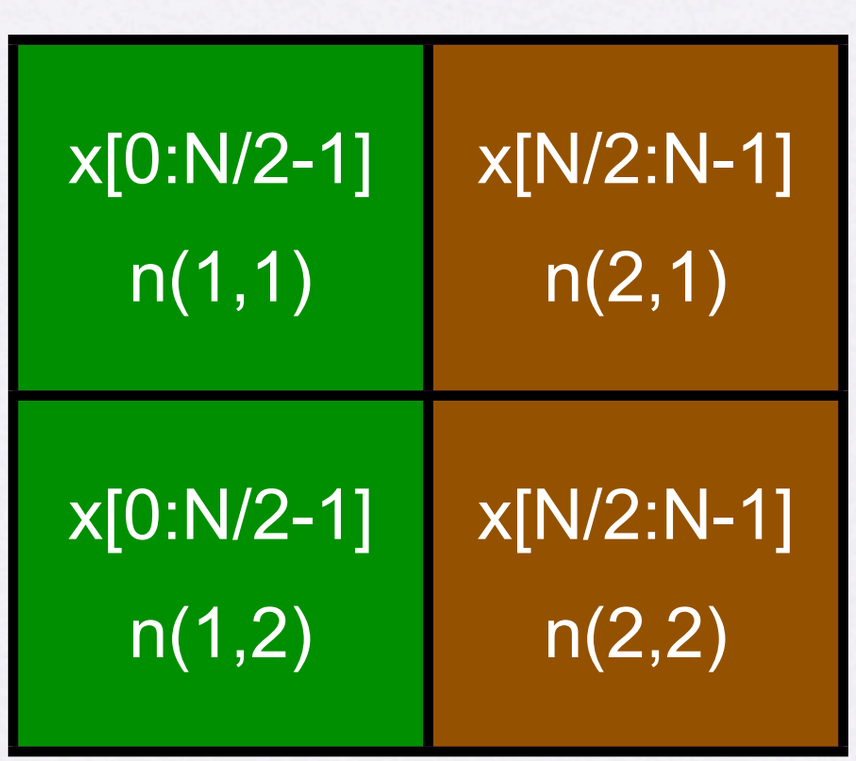
```
#pragma xmp template t(0:N-1,0:N -1)
#pragma xmp distribute t(block,block) on n

double x[N], z[N], p[N], q[N], r[N], w[N];

#pragma xmp align [i] with t(i,*):: x,z,p,q,r
#pragma xmp align [i] with t(*,i):: w
```

4プロセスの例　x[], z[], p[], q[], r[]

w[]



template t

CG（共役勾配法）のXMP化

2次元疎行列

a[ ] → CG（共役勾配法）→ 最小固有値

プロセスは2次元に配置

```
#pragma xmp template t(0:N-1,0:N -1)
#pragma xmp distribute t(block,block) on n

double x[N], z[N], p[N], q[N], r[N], w[N];

#pragma xmp align [i] with t(i,*):: x,z,p,q,r
#pragma xmp align [i] with t(*,i):: w
```

4プロセスの例　x[], z[], p[], q[], r[]

w[]

template t

CG（共役勾配法）のXMP化

2次元疎行列

a[ ] → CG（共役勾配法）→ 最小固有値

プロセスは2次元に配置

```
#pragma xmp template t(0:N-1,0:N -1)
#pragma xmp distribute t(block,block) on n

double x[N], z[N], p[N], q[N], r[N], w[N];

#pragma xmp align [i] with t(i,*):: x,z,p,q,r
#pragma xmp align [i] with t(*,i):: w
```

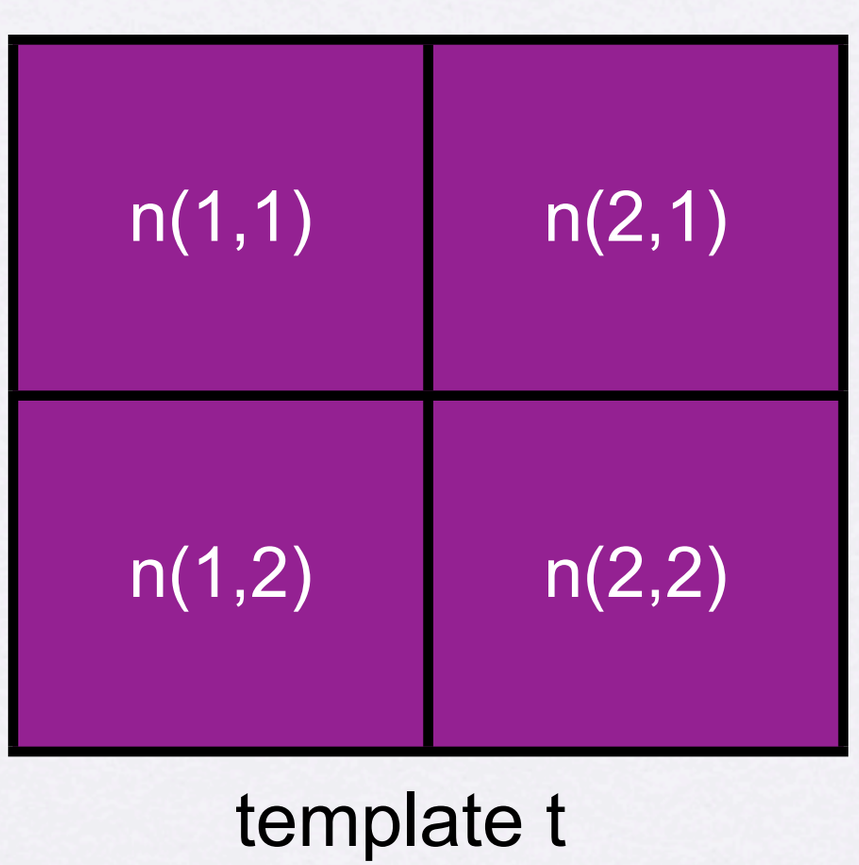
n(*,1)

w[]

n(*,2)

***は重複の意味**

4プロセスの例　x[], z[], p[], q[], r[]



CG-XMPのソース

w, p, qは分散された配列

p[j], q[j] with t (j, *)
w[j] with t (*, j)

```
#pragma xmp loop on t(*, j)
    for(j = 0; j < lastrow-firstrow+1; j++) {
        sum = 0.0;
        for(k = rowstr[j]; k <= rowstr[j+1]; k++) {
                sum = sum + a[k]*p[colidx[k]];
        }
        w[j] += sum;
    }

#pragma xmp reduction(+:w) on n(*,:)

#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの集約）

（ベクトルの交換）

rowstrとcolidxを各ノードが分担して処理するように設定（MPIと同様）
次元の対応が異なるqとwをgmoveを用いて代入する

CG-XMPのソース

w, p, qは分散された配列

p[j], q[j] with t (j, *)
w[j] with t (*, j)

```
#pragma xmp loop on t(*, j)
    for(j = 0; j < lastrow-firstrow+1; j++) {
        sum = 0.0;
        for(k = rowstr[j]; k <= rowstr[j+1]; k++) {
                sum = sum + a[k]*p[colidx[k]];
        }
        w[j] += sum;
    }

#pragma xmp reduction(+:w) on n(*,:)
```

（ベクトルの集約）

```
#pragma xmp gmove
    q[:] = w[:];
```

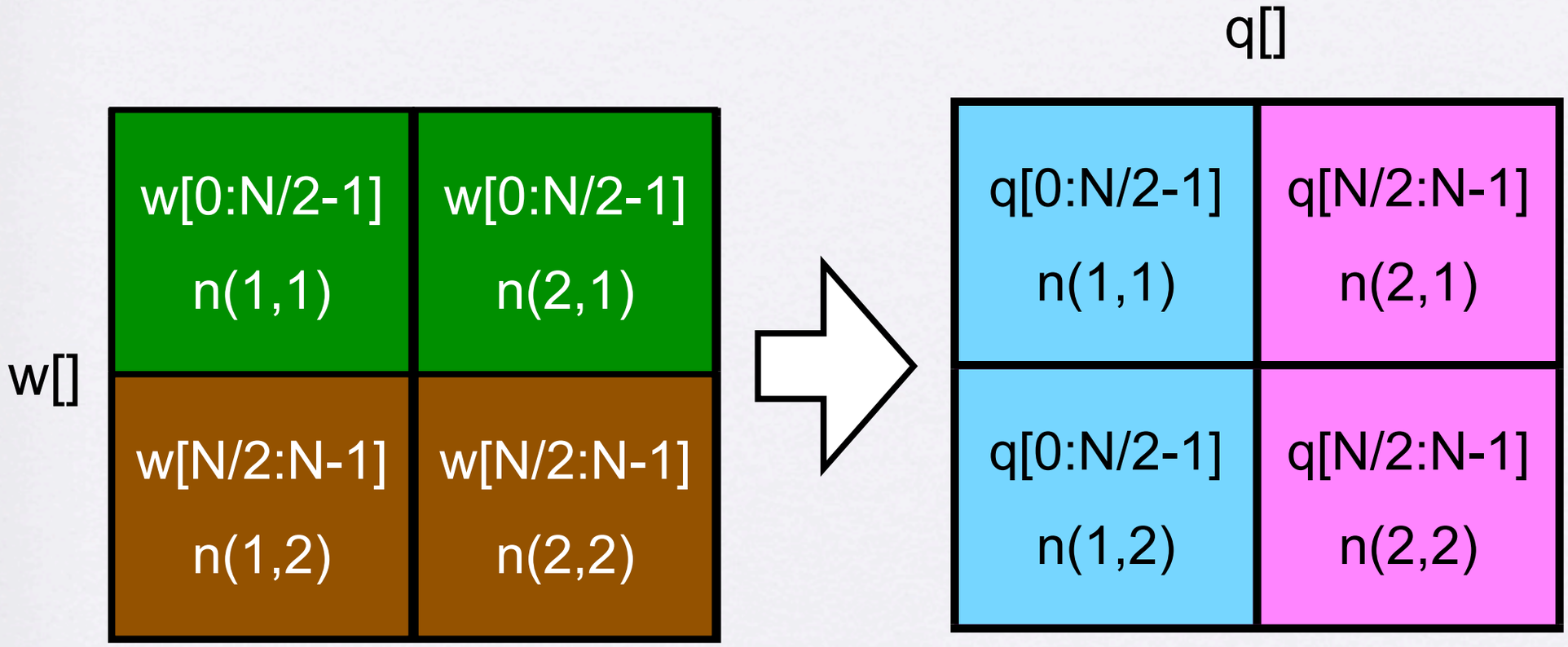
（ベクトルの交換）

rowstrとcolidxを各ノードが分担して処理するように設定（MPIと同様）
次元の対応が異なるqとwをgmoveを用いて代入する

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

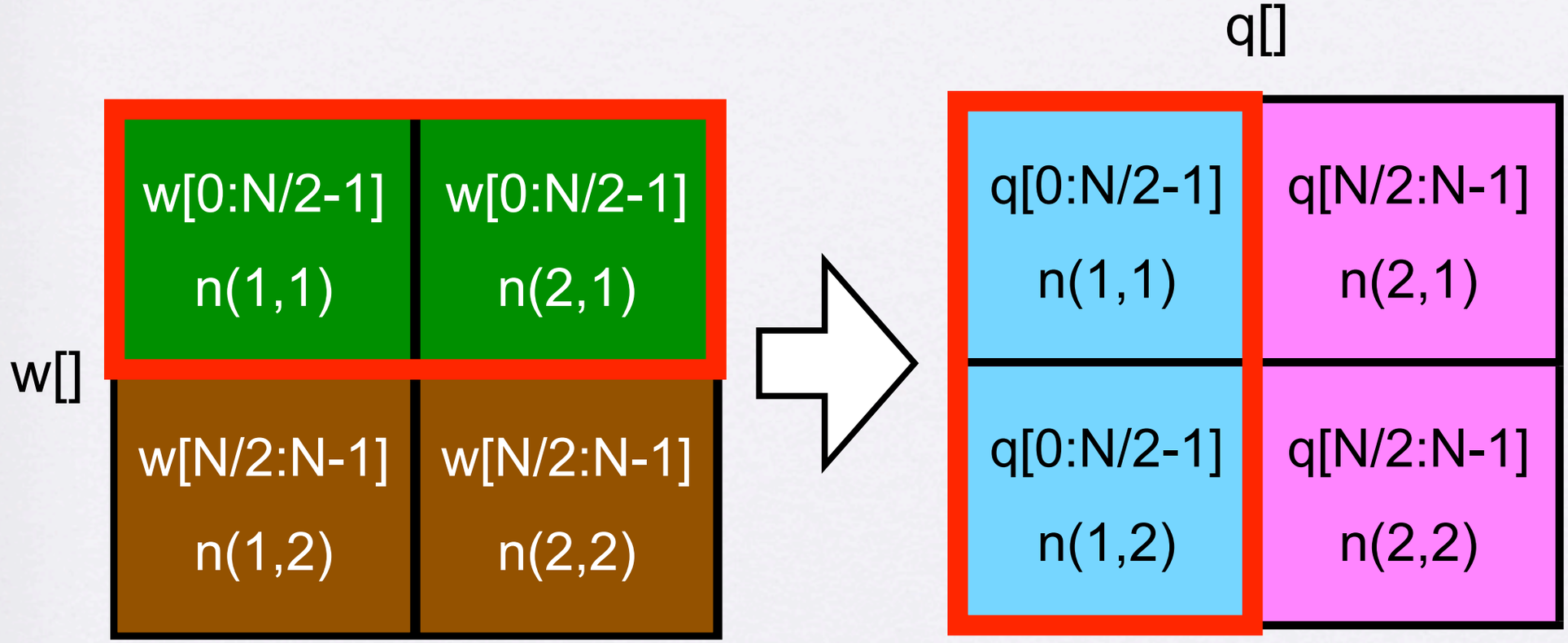


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

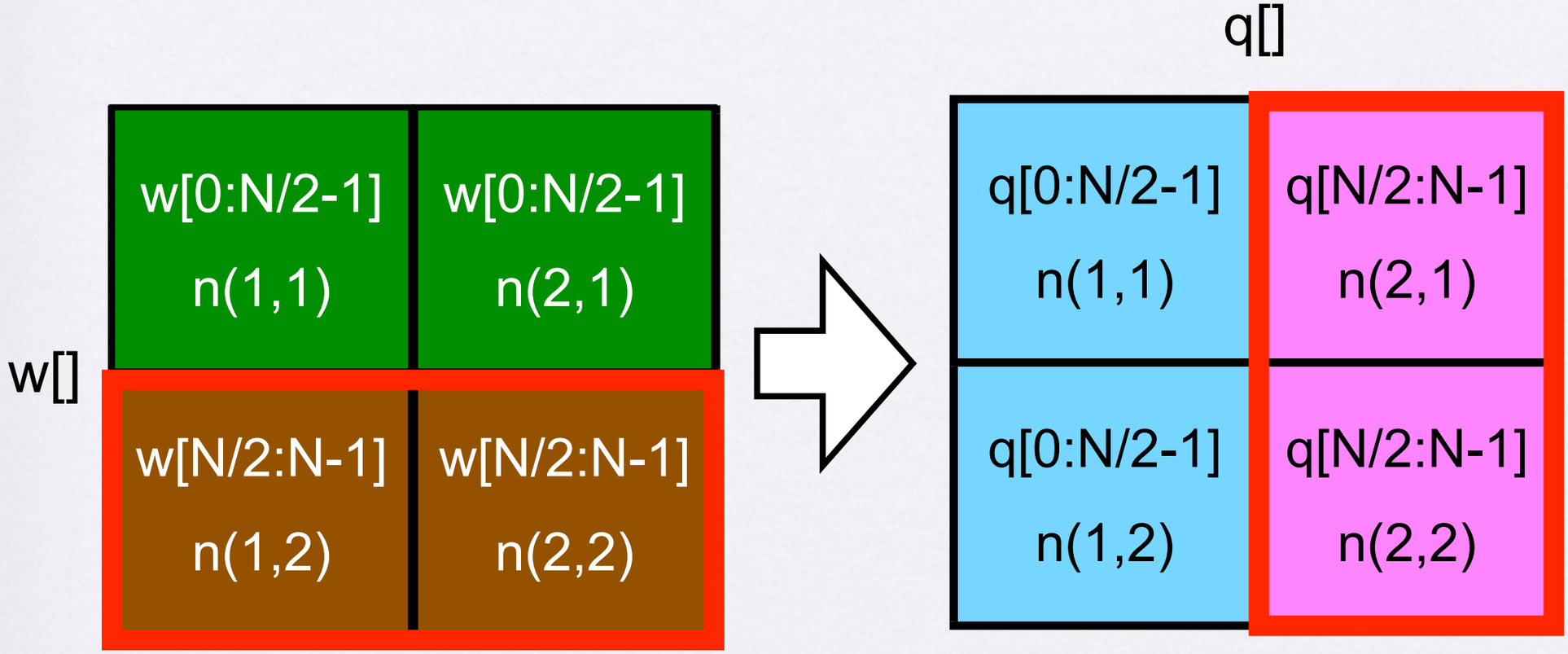


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

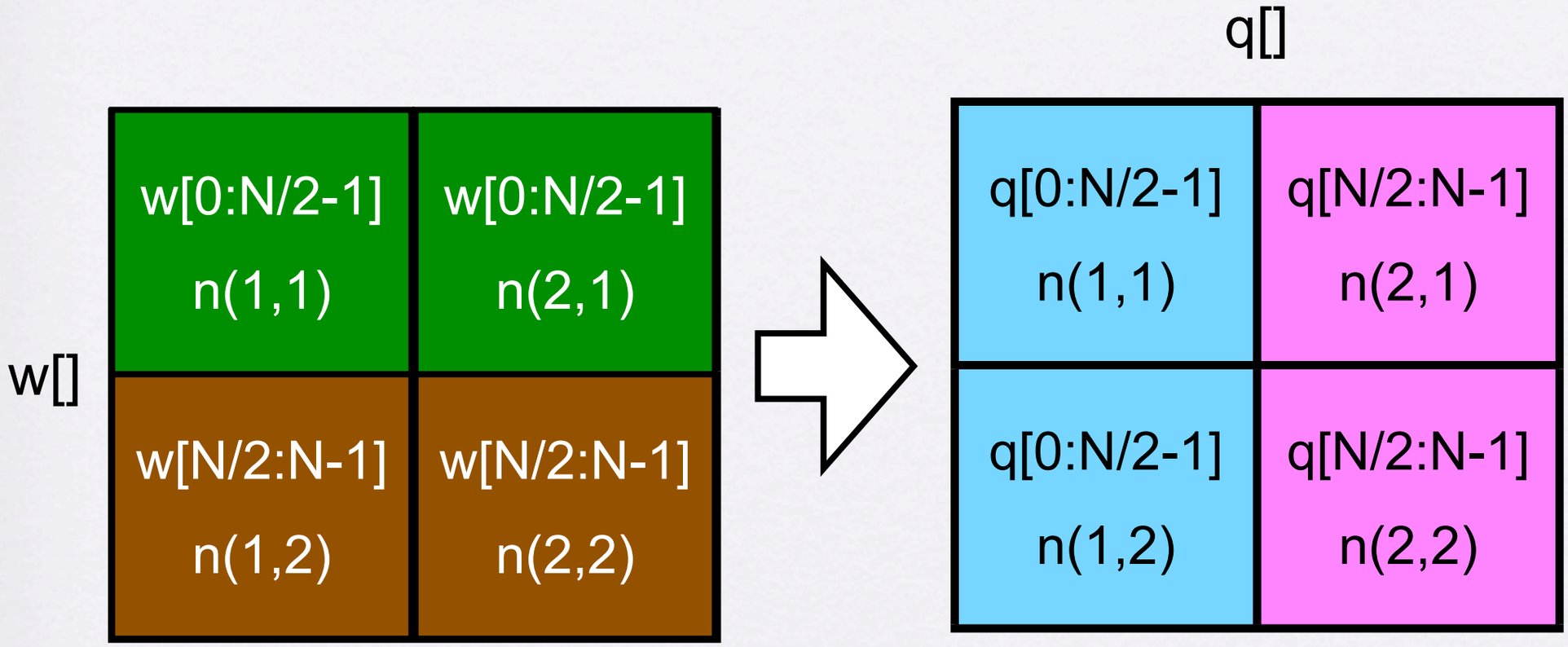


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

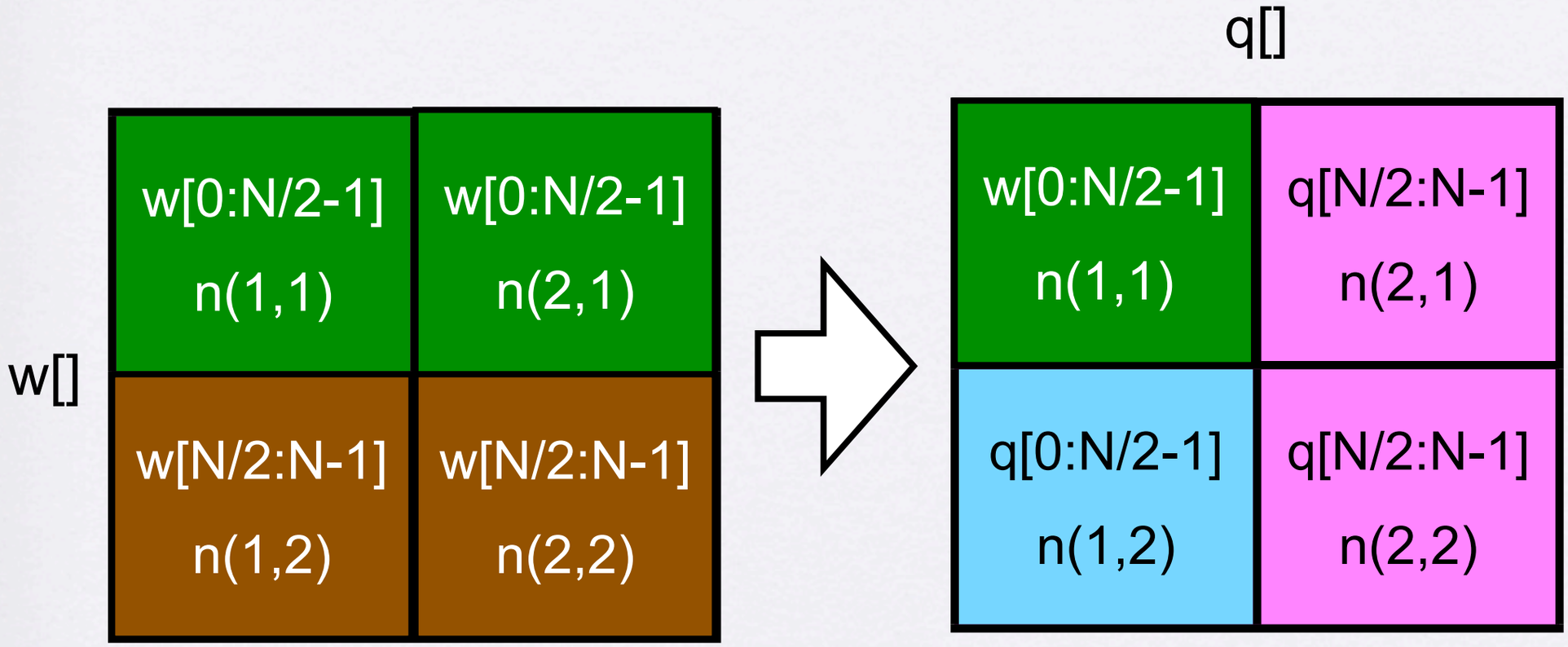


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

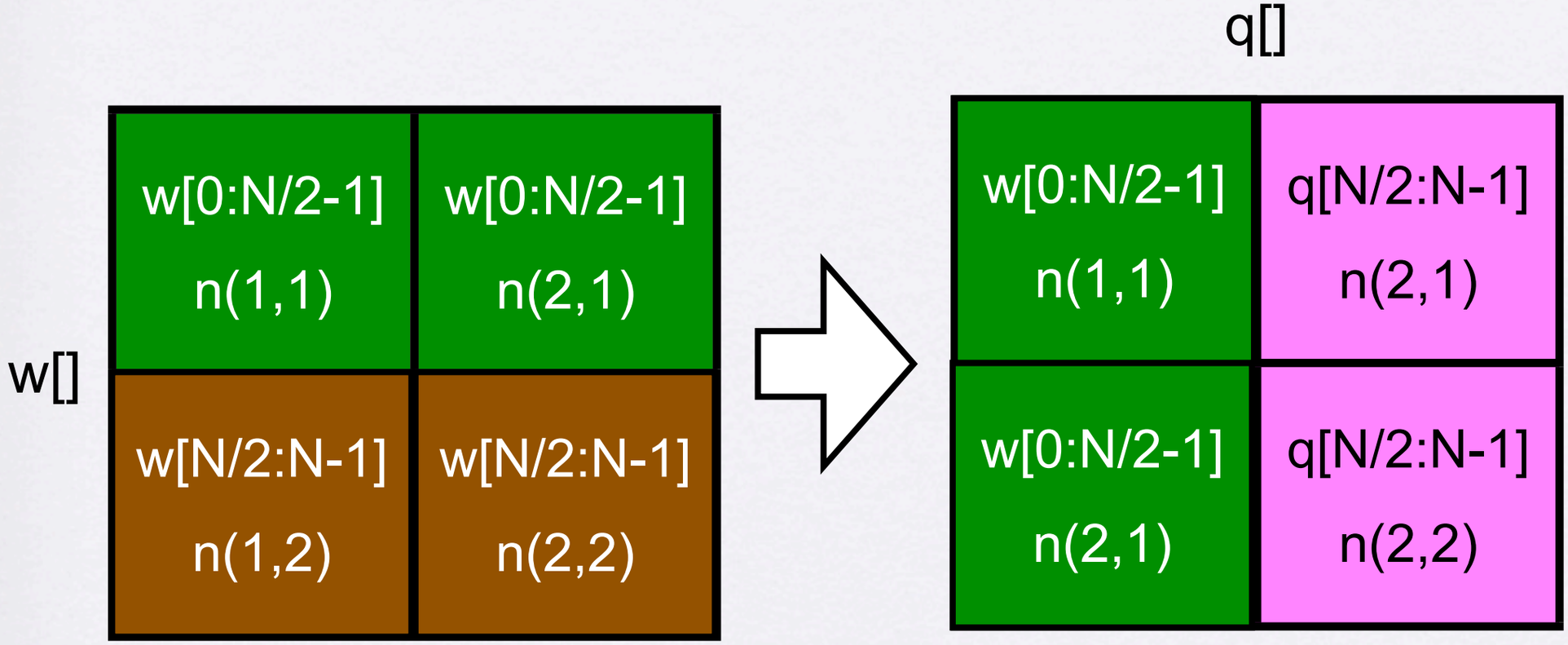


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

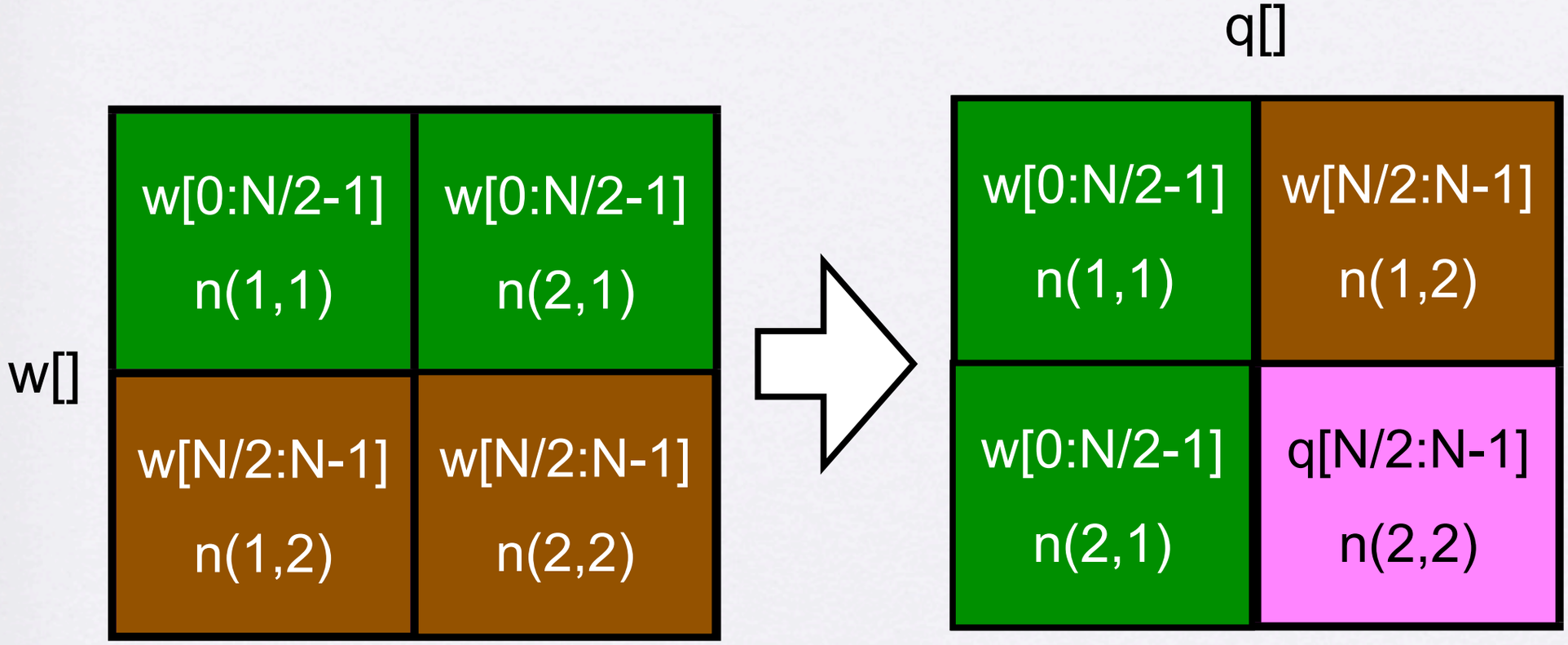


通信を伴う転置操作

CG-XMPのソース

```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）

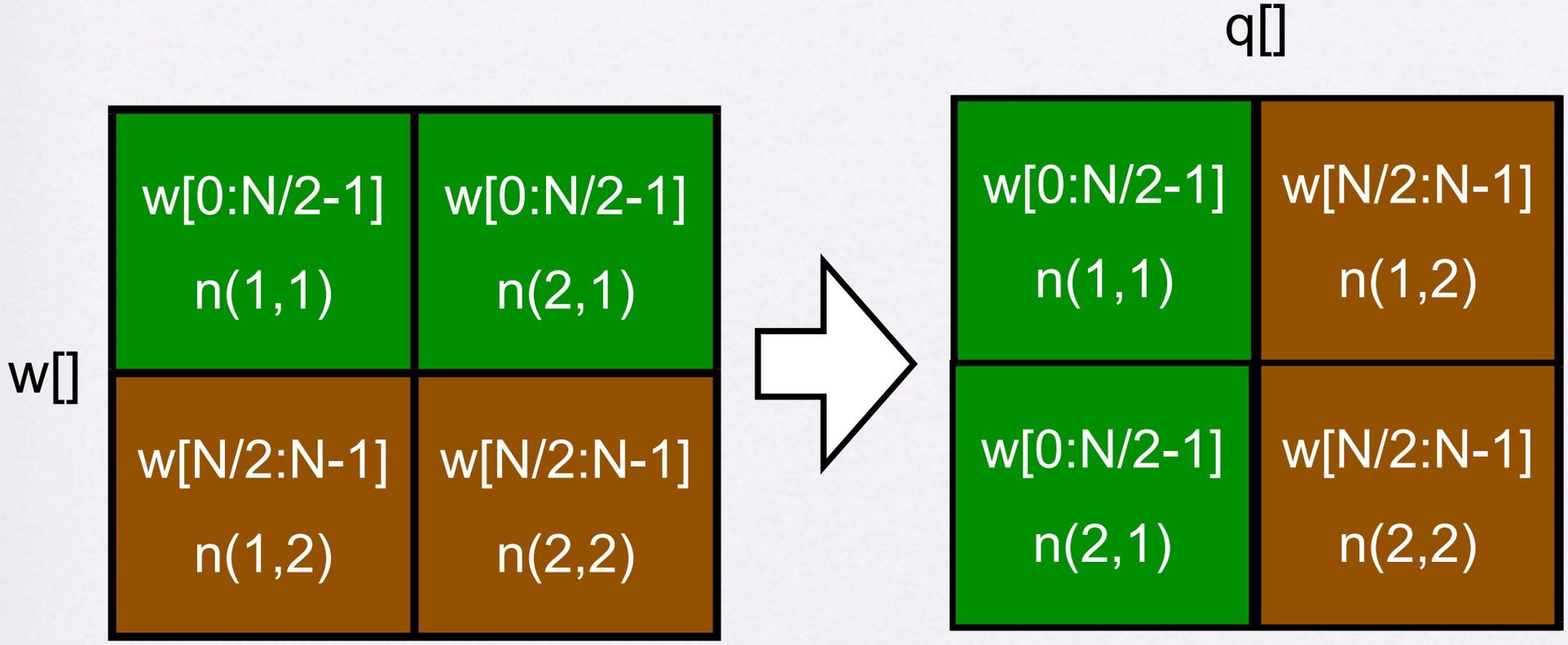


通信を伴う転置操作

CG-XMPのソース

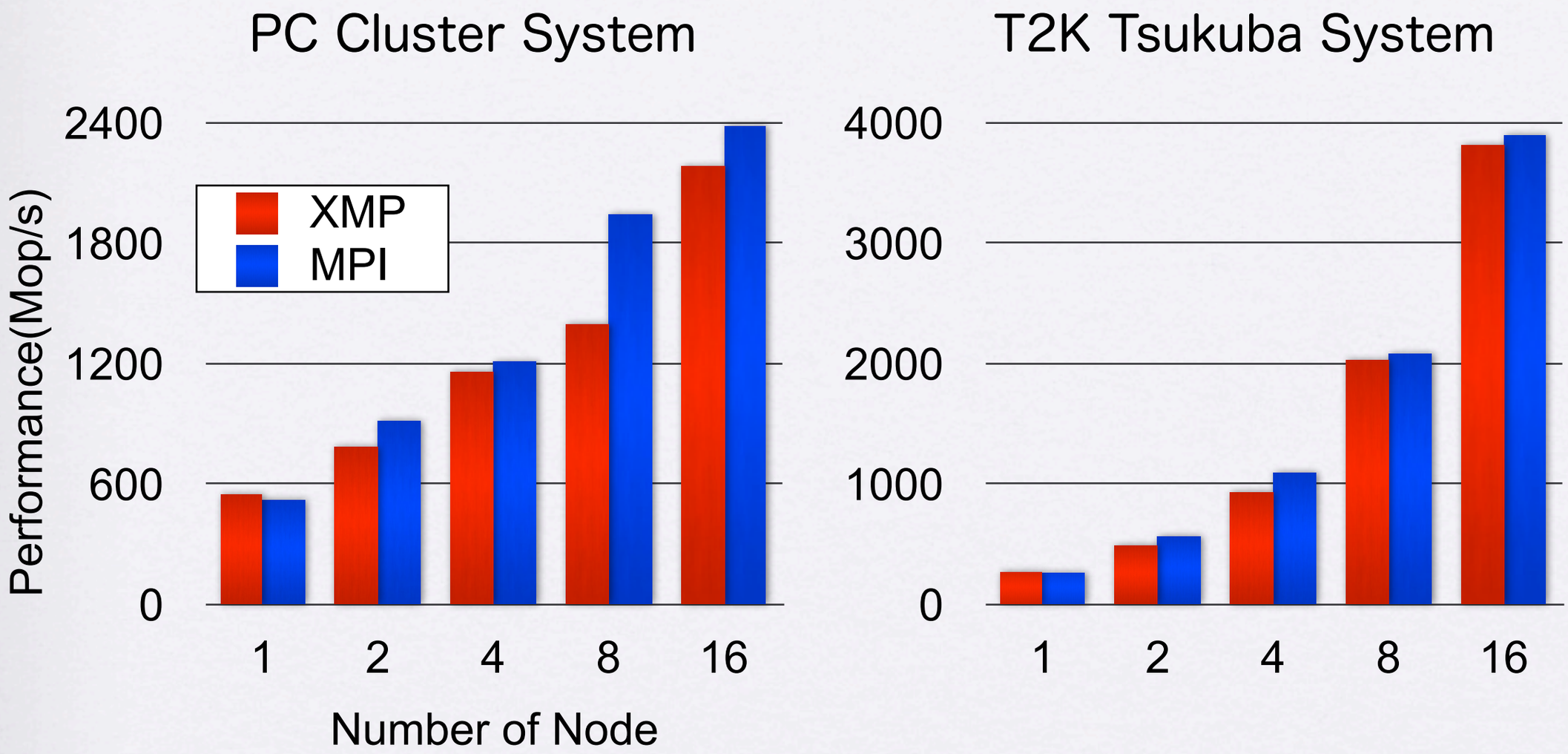
```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの交換）



通信を伴う転置操作

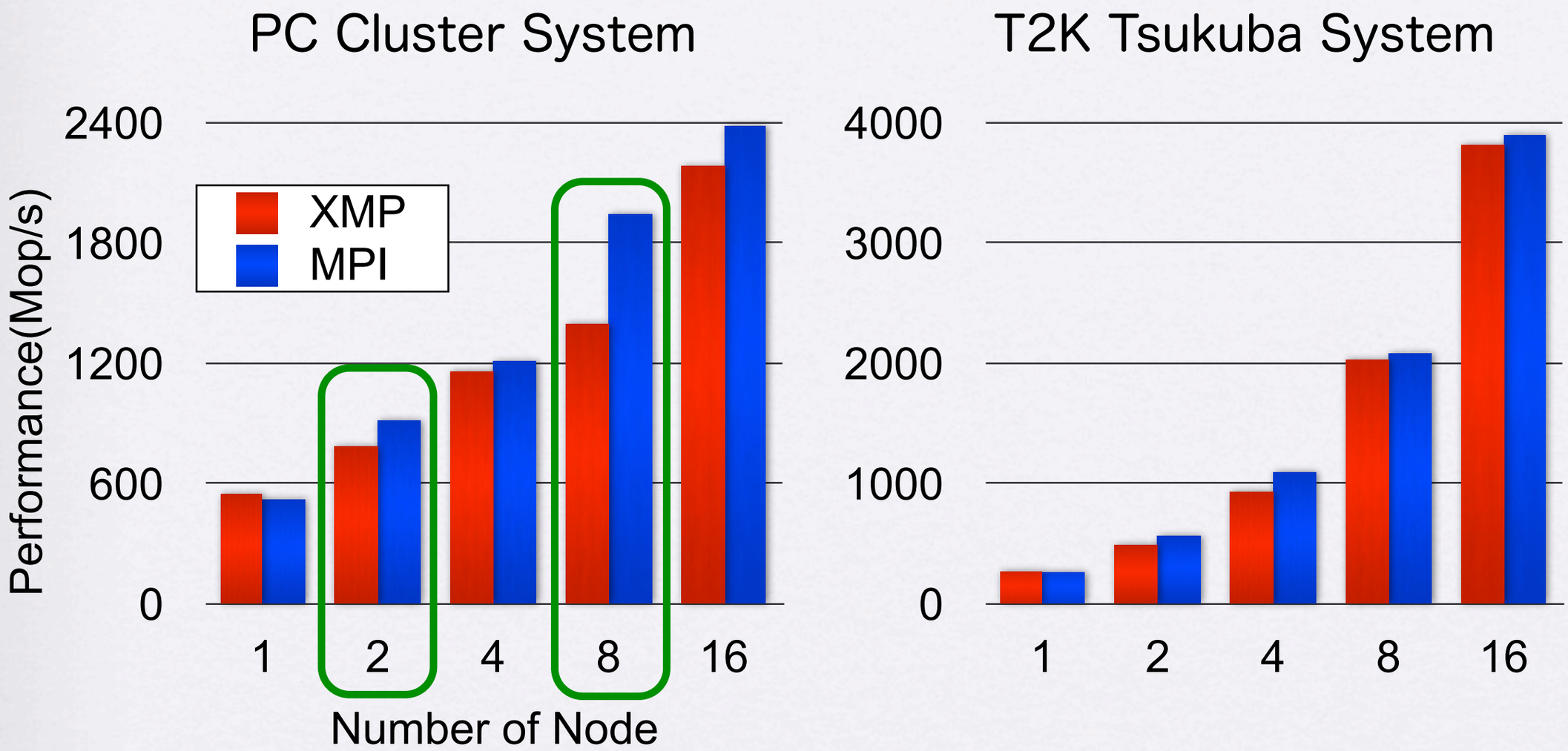
実験結果（CG）



1, 4, 16プロセスの場合,
ほぼ同じ性能

XMPとMPIはほぼ同じ性能

実験結果（CG）



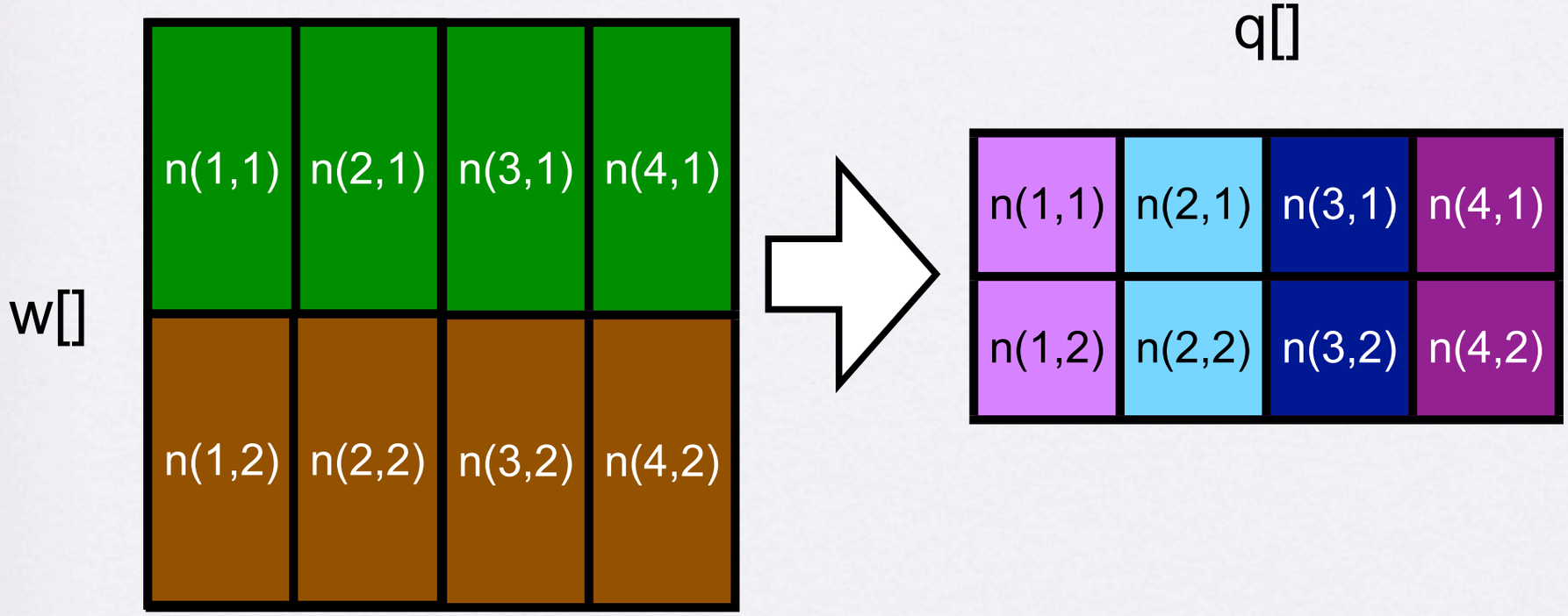
1, 4, 16プロセスの場合,
ほぼ同じ性能

XMPとMPIはほぼ同じ性能

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



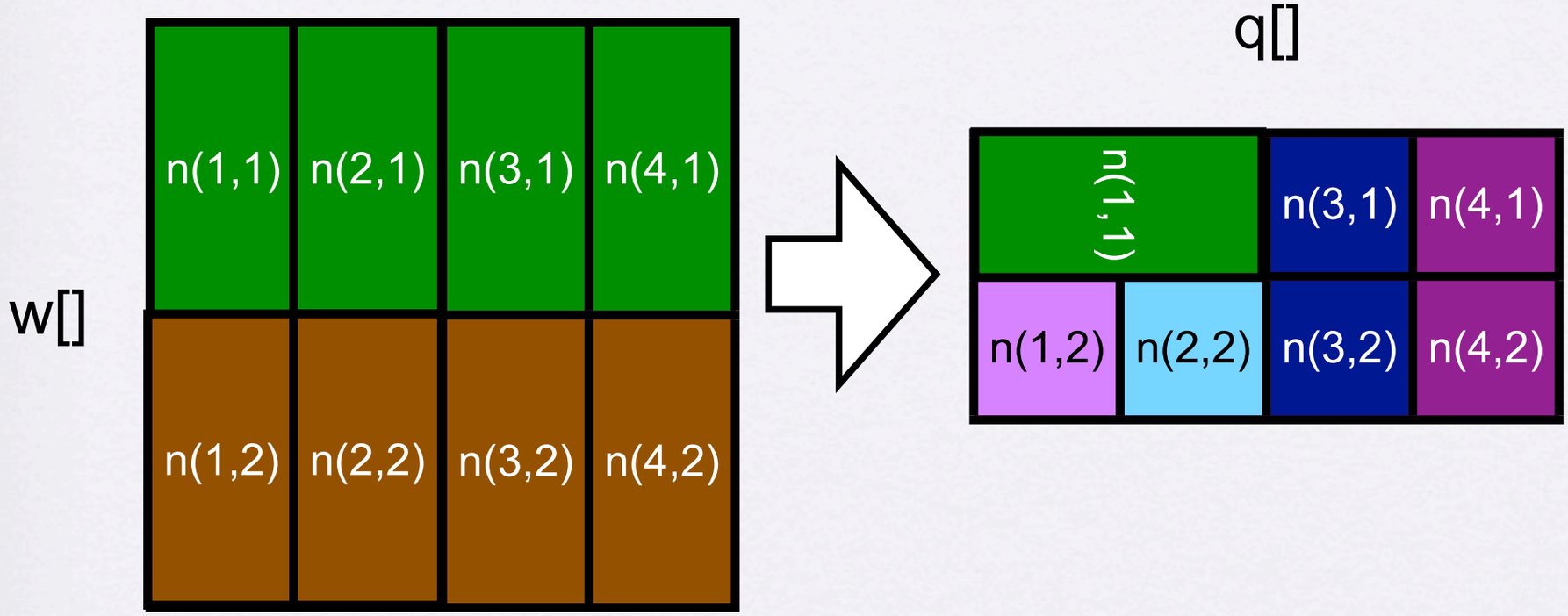
2分割×4コピー　　4分割×2コピー

- XMP版ではすべての要素をリダクションにしているのに対し，
  MPI版は計算に必要な要素のみをリダクションしているため

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



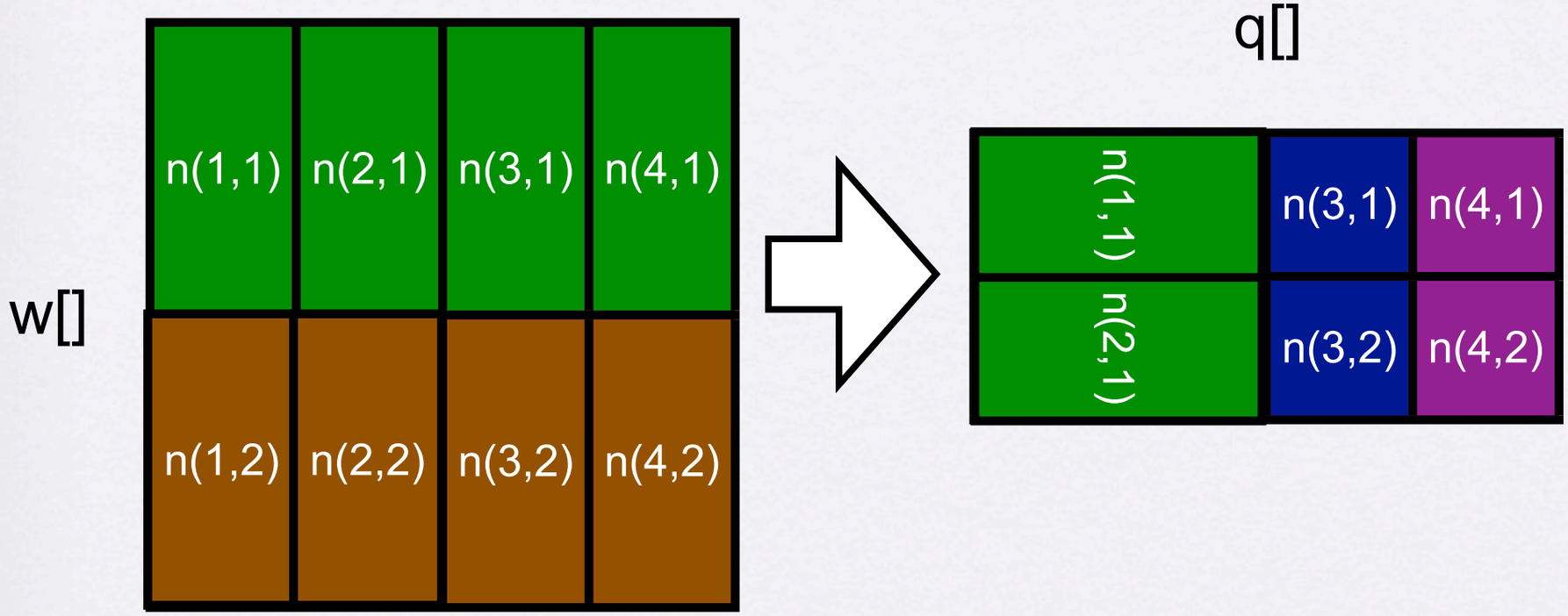
2分割×4コピー　　4分割×2コピー

- XMP版ではすべての要素をリダクションにしているのに対し，
  MPI版は計算に必要な要素のみをリダクションしているため

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



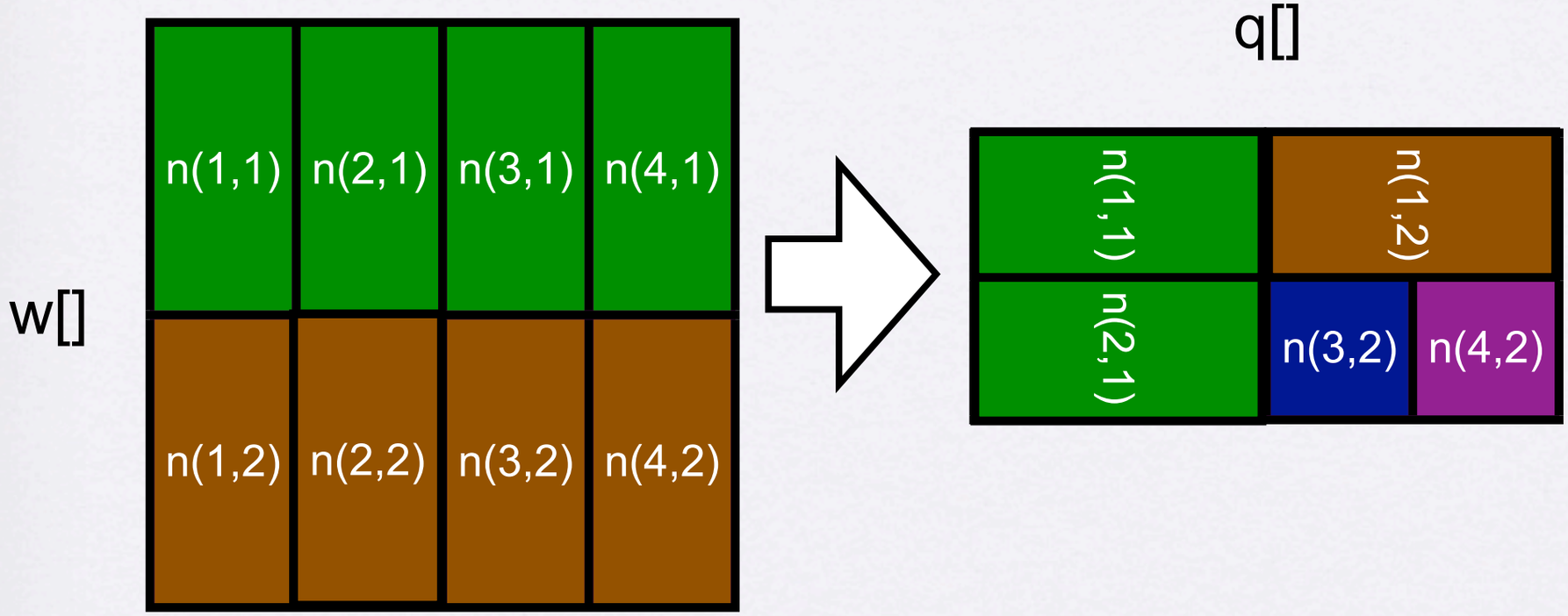
2分割×4コピー　　4分割×2コピー

- XMP版ではすべての要素をリダクションにしているのに対し，
  MPI版は計算に必要な要素のみをリダクションしているため

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



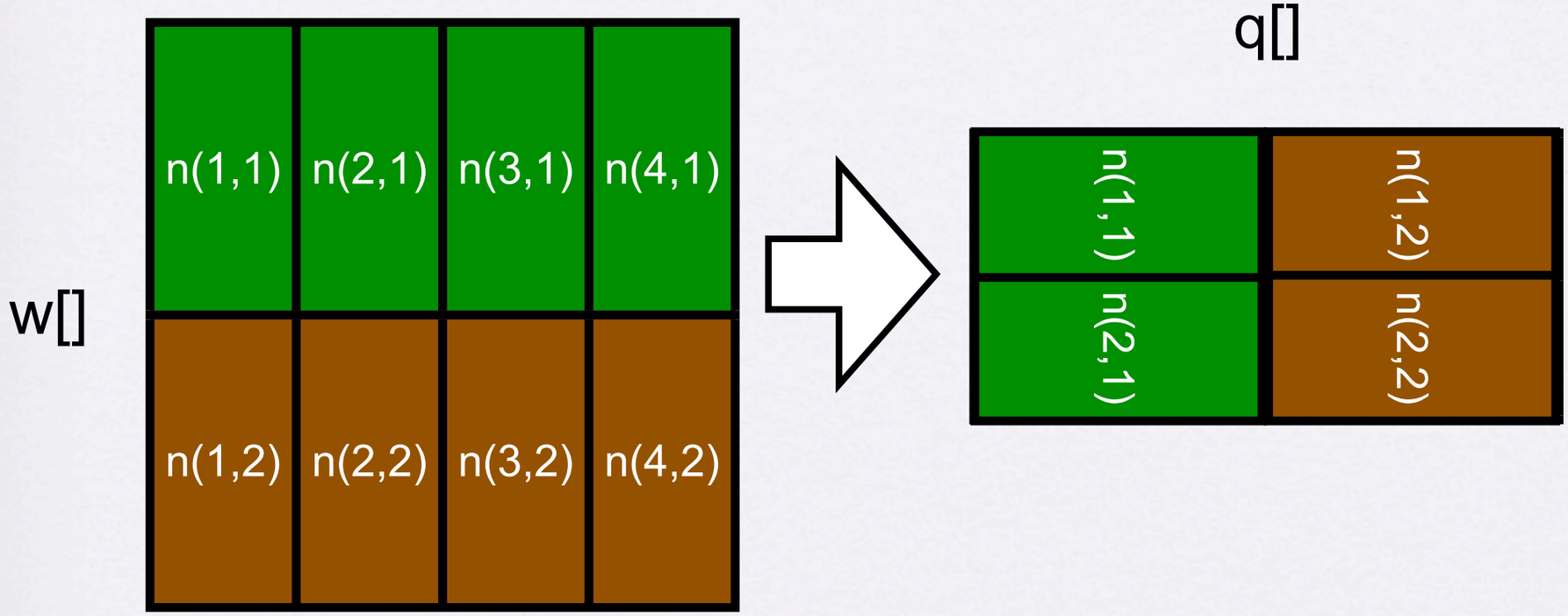
2分割×4コピー　　4分割×2コピー

- XMP版ではすべての要素をリダクションにしているのに対し，
  MPI版は計算に必要な要素のみをリダクションしているため

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



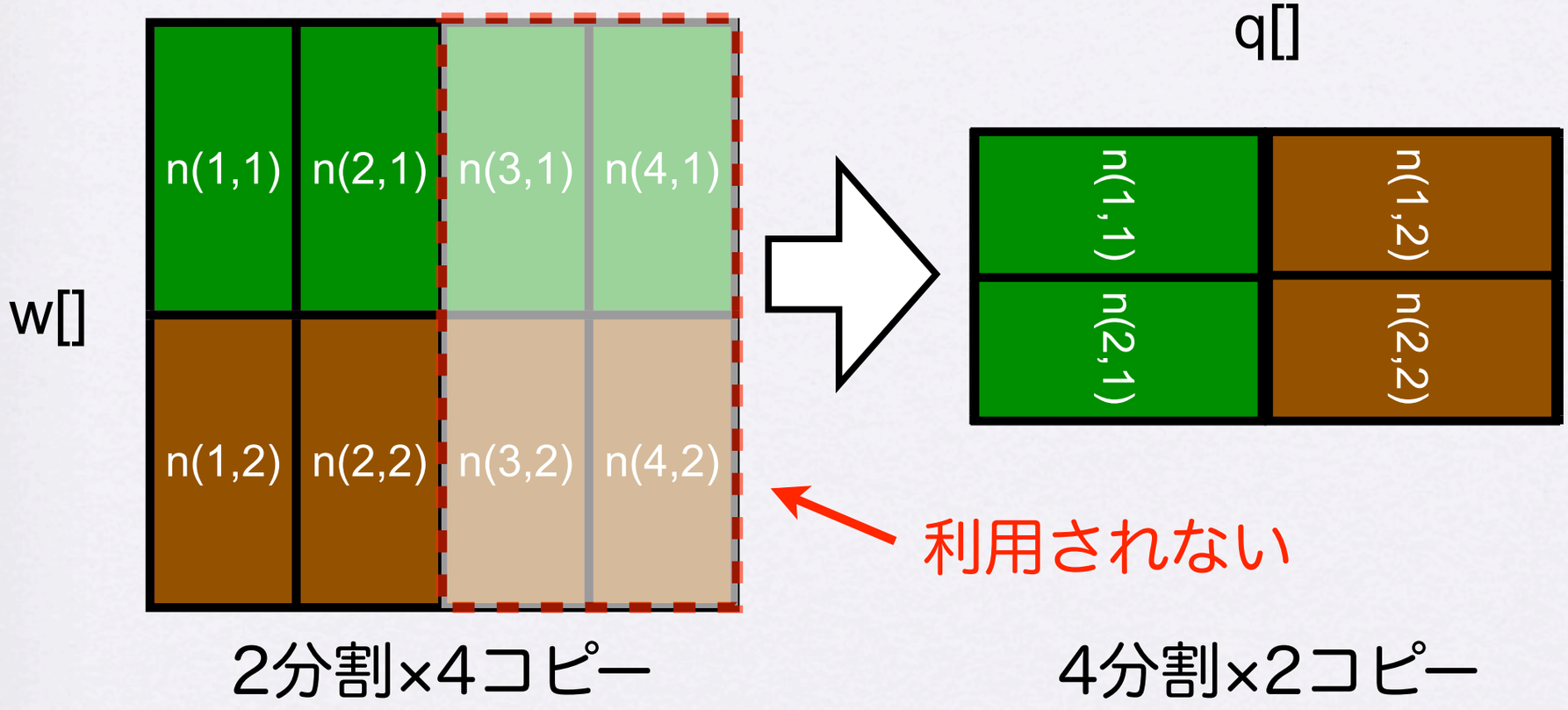
2分割×4コピー　　4分割×2コピー

- XMP版ではすべての要素をリダクションにしているのに対し，
  MPI版は計算に必要な要素のみをリダクションしているため

CGの考察

- 2と8プロセスの場合，縦と横の分割数が異なる（1，4，16では同じ）

reduction後のgmove　wとqの要素数は同じ



- XMP版ではすべての要素をリダクションにしているのに対し，MPI版は計算に必要な要素のみをリダクションしているため

まとめと今後の課題

- 分散メモリシステム用の新しいプログラミングモデルであるXMPの性能評価
- 本発表では，NPBのCGとISをXMPの実装を紹介
- MPI版のNPBとの性能比較した結果，ほぼ同等の性能
- 今後の課題
  - マルチコア対応XMPの評価
  - ノード数とプロセス数を増やして性能評価
  - 並列化をサポートするための，プロファイリングツールの開発