高性能並列プログラミング言語処理系 XcalableMP

筑波大学 計算科学研究センター
中尾昌広

T2K Symposium 2010@Tokyo Univ.

発表内容

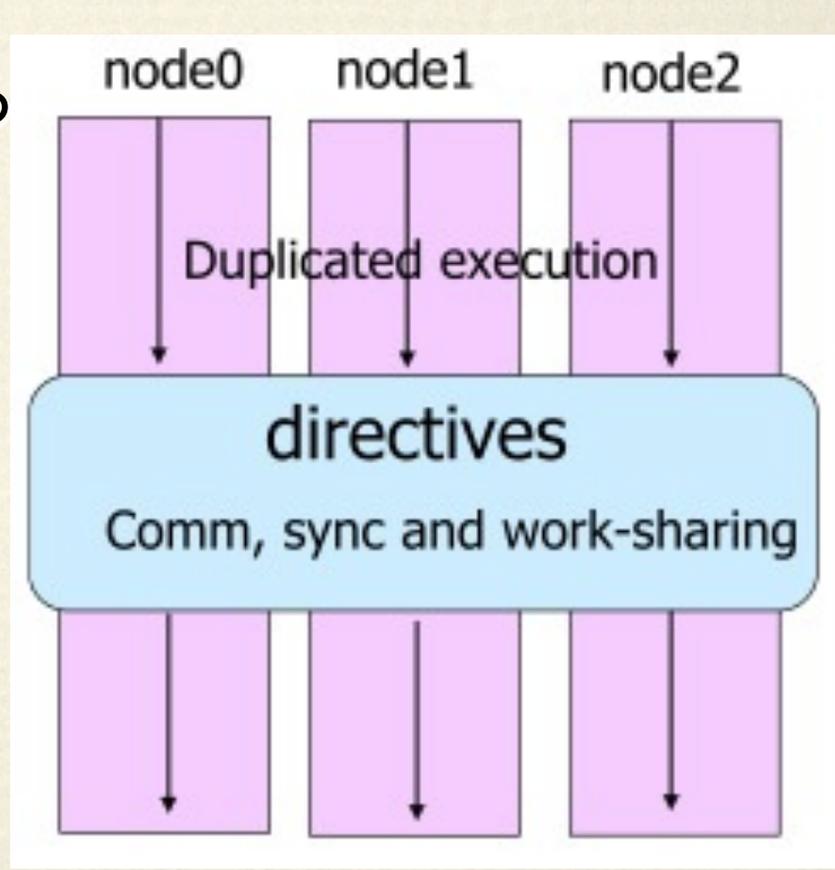
- XcalableMPの概要
- XcalableMPにおける並列化の性能評価と使ってみた感想

XcalableMPの開発背景

- 分散メモリを前提とした並列プログラミング環境が必要
  - 現在はMPIが主流。しかし、、、
    - プログラミングコストが高い
    - 記述が煩雑になりがち（コードの管理が大変）
- 研究室のPCクラスタからThe K computer（次世代スパコン）まで並列化が簡易に行える言語が必要

XcalableMPの特徴

- 既存言語（CとFortran）に指示文を挿入することで，並列化機能の提供を行う
  - 簡便な指示文で定型的なデータ並列を可能にし，明確な並列化モデルを提示する
  - プログラミングコストを低減
  - Performance Awareness ：通信がどこで発生するかが明瞭（パフォーマンスチューニングを行いやすい）



XcalableMPのコード例

```
#pragma xmp nodes p(*)
#pragma xmp template t(0:MAX-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align array[i] with t(i)
```

（データの定義）

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

#xmp pragma reduction (+:sum)
}
```

（ループ内並列）

（結果の集約）

MPIで同じコードを書くと

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

    dx = MAX/size;
    llimit = rank * dx;
    if(rank != (size -1)) ulimit = llimit + dx;
    else ulimit = MAX;

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

    MPI_Allreduce(&sum, &tmp, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Finalize( );
}
```

XcalableMPの実行概要

コンパイル

```
$ xmpcc a.c
```

実行

```
$ mpirun -np 4 a.out
```

ソースコード

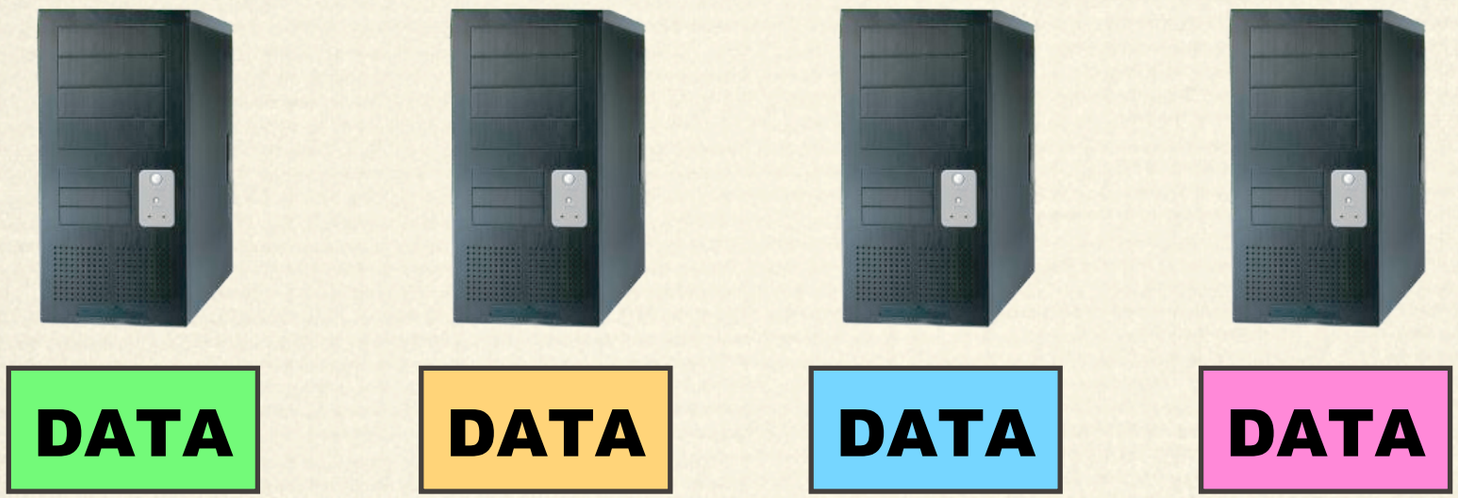
実行ファイル

XcalableMPプログラミング

- プログラミングモデルの種類
  - グローバルビューによるプログラミングモデル
    全体的なデータの分散や通信を考慮した並列化
  - ローカルビューによるプログラミングモデル
    ローカルのデータおよびノード間通信を意識した並列化

グローバルビューの説明の前に

- 定型的な並列化方法



データ（配列）を各ノードに分散配置し，それぞれを処理するようにプログラムする

```
count = N / num_procs;

for( i = 0; i < count; i++)
    a[ i ] = func( i );
```

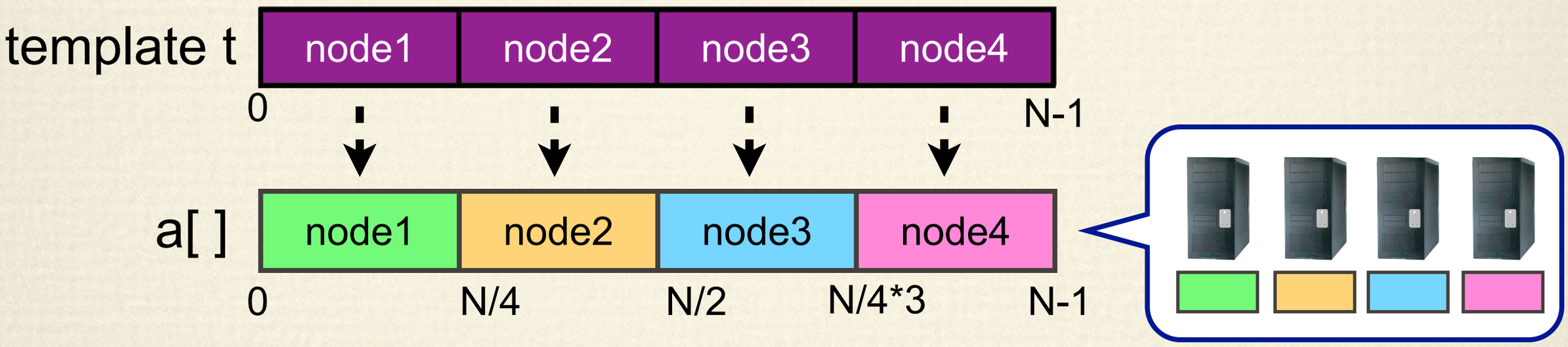
元々１つだった配列を
分散配置させて処理している

グローバルビューの特徴

- 仮想的なindex空間を用いて，１つの配列のように操作する

→ #pragma xmp template t(0:N-1)
→ #pragma xmp distribute t(block) onto p
→ #pragma xmp align a[i] with t(i)

（データの定義）




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

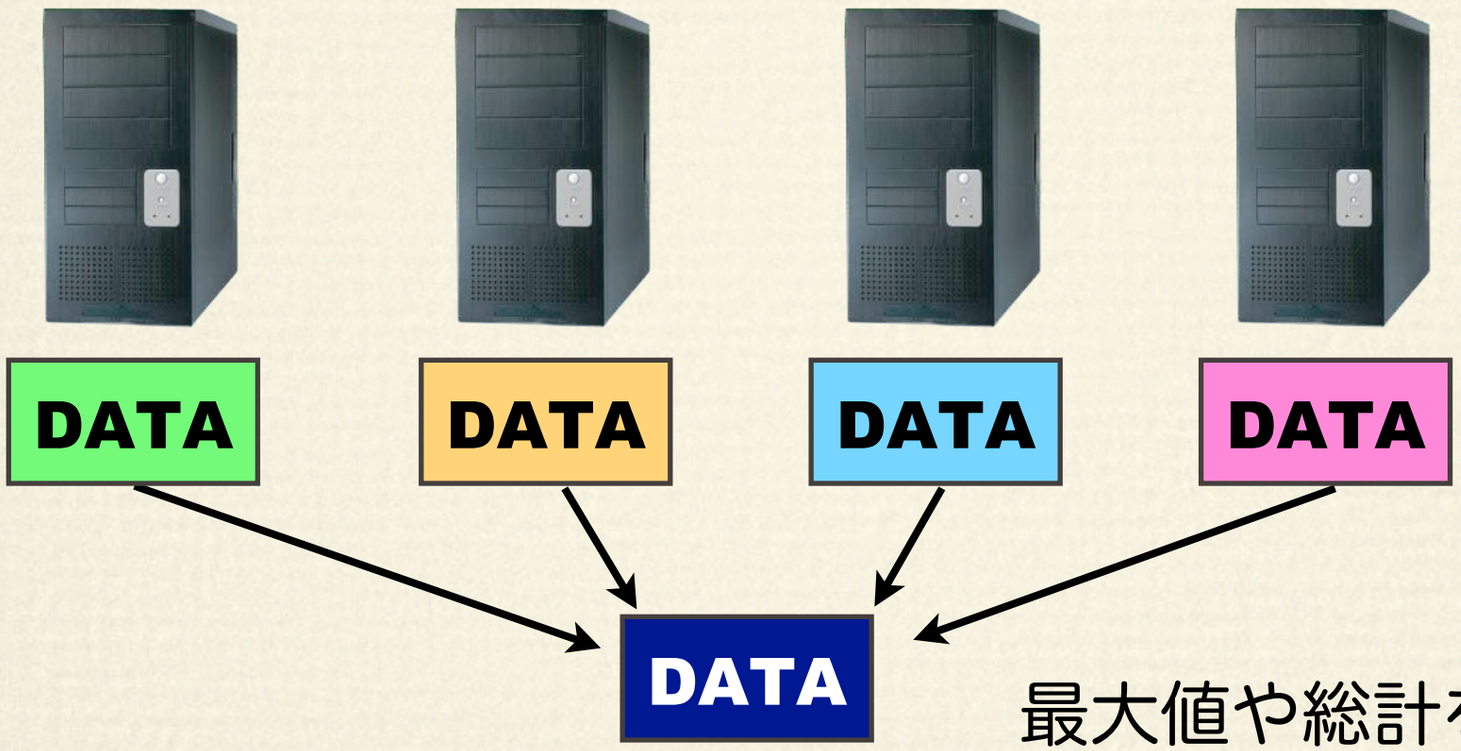
・逐次コードの変更なしで並列化可能
・１つの配列のように操作できるため，
　並列化プログラミングの負担が少ない

他のグローバルビュー

- XcalableMPでは，ループ文のみでなく，
  データ全体に関わる操作をグローバルビューと定義
- XcalableMPが提供する指示文を用いることで，定型的な
  （MPIでもよく用いられている）集合通信を簡易に利用可能

集合通信の例

- リダクション（集約）



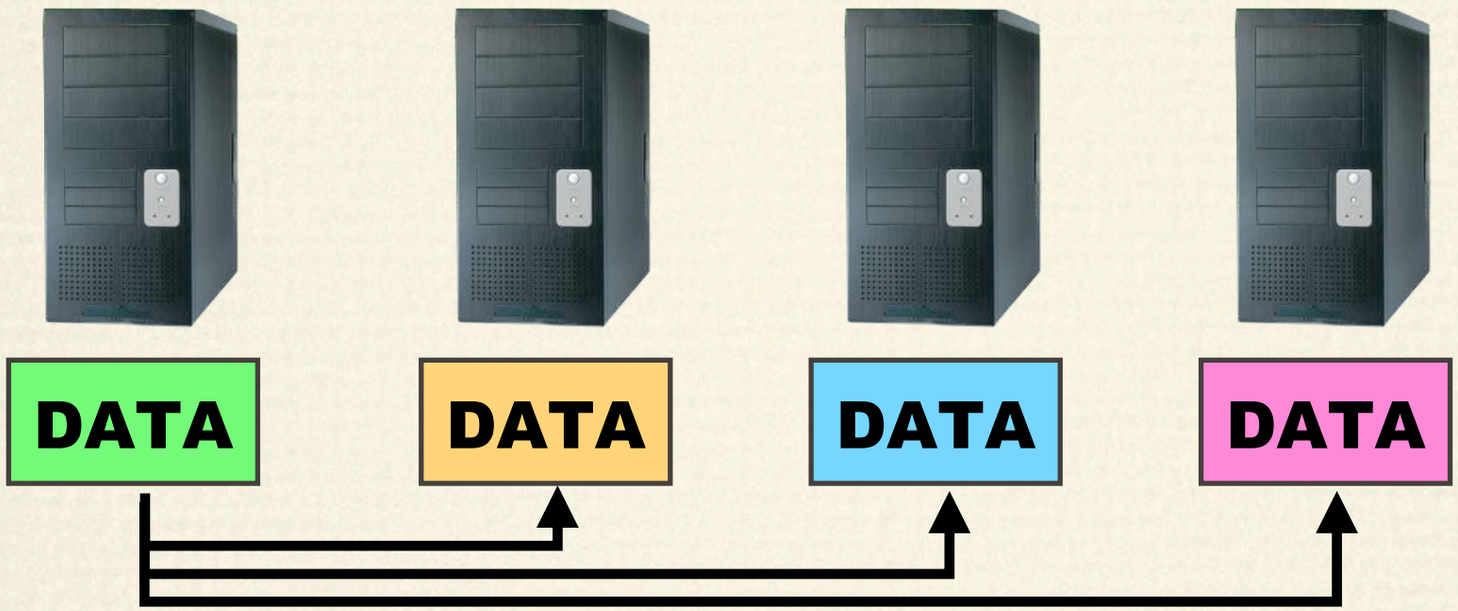
最大値や総計を出す演算など

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

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

集合通信の例

- ブロードキャスト

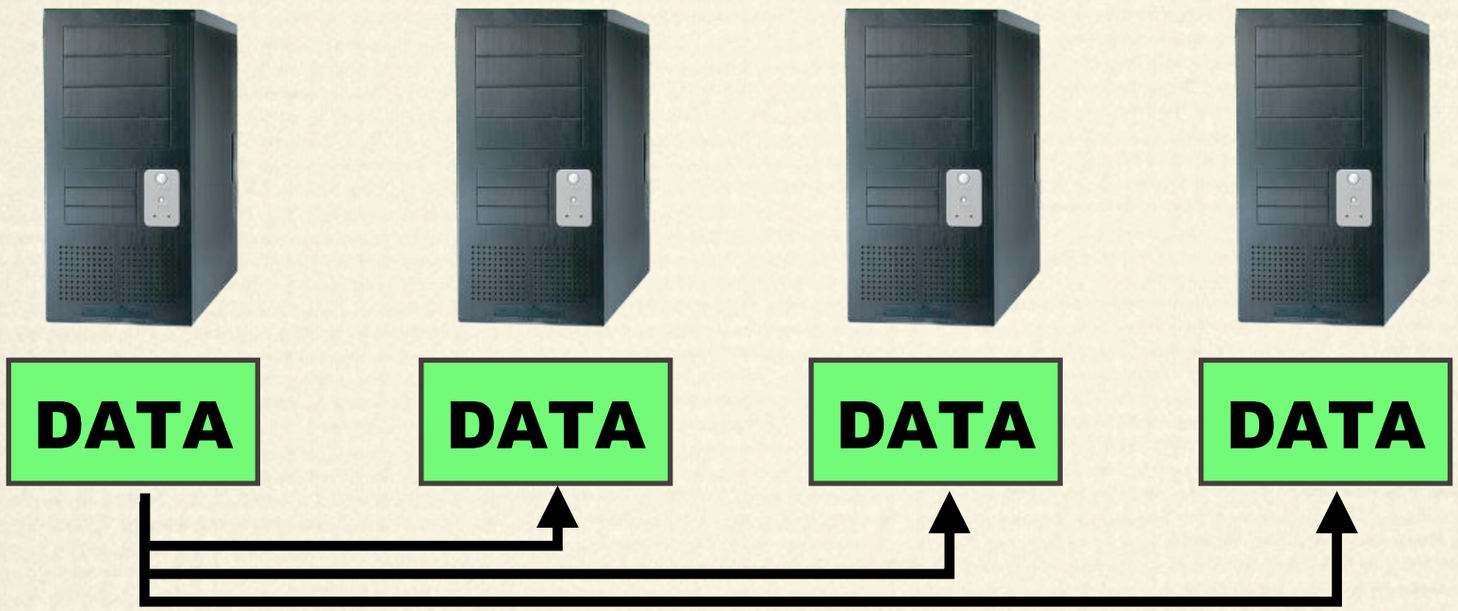



```
#pragma xmp bcast a from p(0)
```

MPI_Bcast(sendbuf, count, datatype, root, comm, ierror)

集合通信の例

- ブロードキャスト

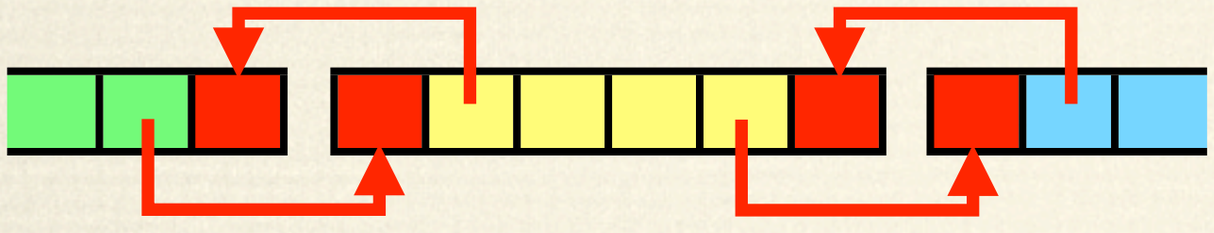



```
#pragma xmp bcast a from p(0)
```

MPI_Bcast(sendbuf, count, datatype, root, comm, ierror)

shadow

- シャドウ領域の定義と一括通信



隣の領域が計算に必要！

```
#pragma xmp shadow a [1]
・・・
#xmp pragma reflect a
```

グローバルな配列上で
オーバラップしている箇所を
簡単に指示できる

MPIの場合，ノード毎に送りたい領域のアドレスを計算し，
通信関数を用いて同期をとる必要がある

gmove

- グローバルイメージの代入文

b[3:6]をa[0:3]に代入したい

```
#pragma xmp gmove
a[0:3] = b[3:6];
```

どのデータがどのノードに
配置されているかを,
ユーザは意識する必要がない

ローカルビューの必要性

- 記述された処理に関してはMPIと同等の性能
  - ほとんどの通信はtwo-sided通信で実行できる
  - 典型的な並列化の処理を記述
    - 多くの場合（特にデータ並列では）はこれで十分
- グローバルビューの限界
  - すべての並列アルゴリズムを記述できるわけではない
  - one-sided通信への対応

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
- XcalableMPではローカルビューとしてCo-arrayを導入
- Co-arrayを用いた片側通信
- FortranはCAFと互換（Cでは新しい記法を導入）

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

ノード1が持つb[3:6]のデータを
a[0:3]に代入

配列の次元を拡張（ノード番号を表す）

より柔軟な並列アルゴリズムの記述が可能

性能評価

- 目的

  XcalableMPの性能を測定する

- 方法

  XcalableMPを用いてNAS Parallel Benchmarksを
  並列化実装し，そのMPI版との性能比較を行う

  - Integer Sort（IS）　　：バケツソートを用いた整数ソート
  - Conjugate Gradient（CG）：共役勾配法で最小固有値を計算

実験環境

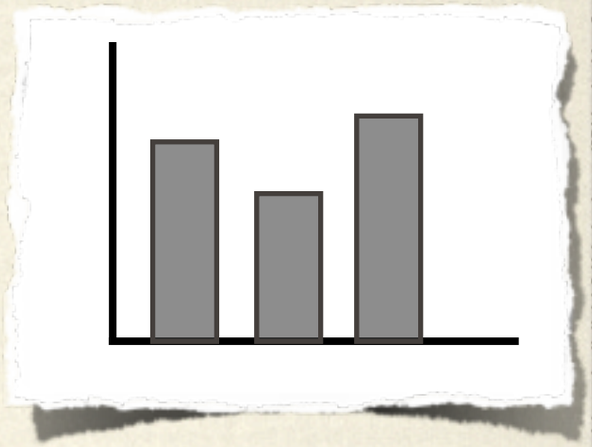
- PC Cluster System

Intel Core2 Quad 3.0GHz
Gigabit Ethernet

- T2K Tsukuba System

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

ISの並列化

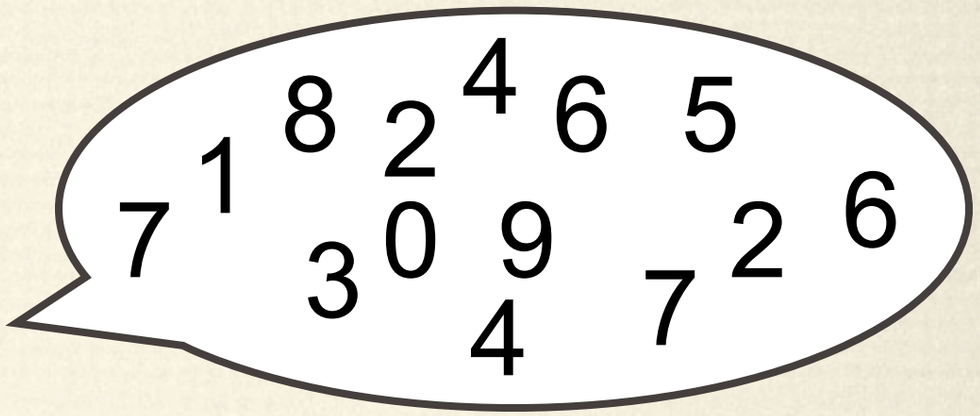


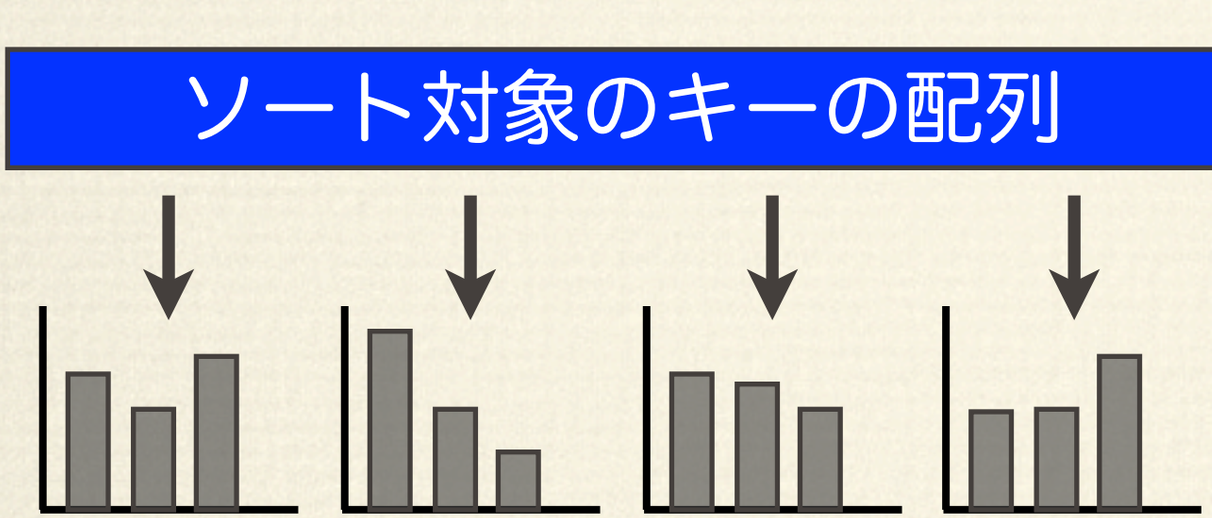
- IS-H：ヒストグラムを用いた並列化手法
  ローカルビューも行う必要がある
- IS-noH：ヒストグラムを用いない並列化手法
  グローバルビューのみで実装可能

MPIはヒストグラムを用いた手法のみ提供されている

IS-Hの概要

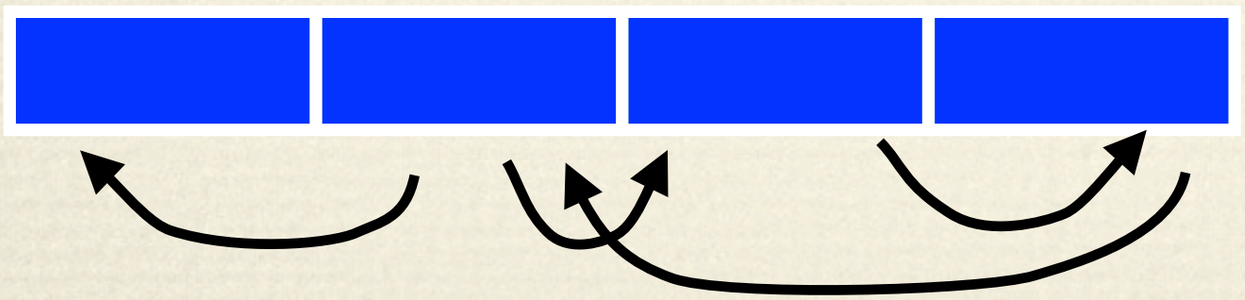
ソート対象のキーの配列





ヒストグラムを作成 + ソート

どのノードにどのキーを送るか計算



値の入れ替え
（ローカルビュー）

値が小さいキーほど左側のノードへ

IS-Hのソース

key_array[]はグローバルな配列

```
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ )
     bucket_size[key_array[i] >> shift]++;
 ・ ・ ・

#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ ) {
     key = key_array[i];
     key_buff1[bucket_ptrs[key >> shift]++] = key;
}

 ・ ・ ・

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

（ヒストグラムを作成）

（ソート）

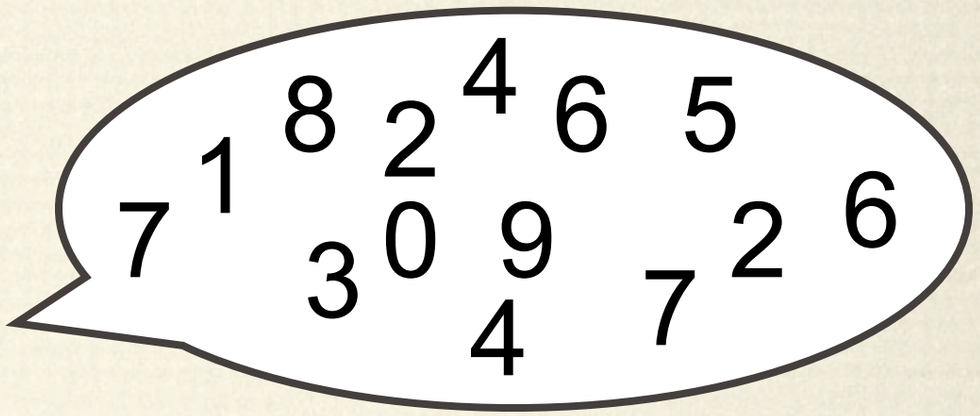
（どのノードにどのキーを送るか計算）

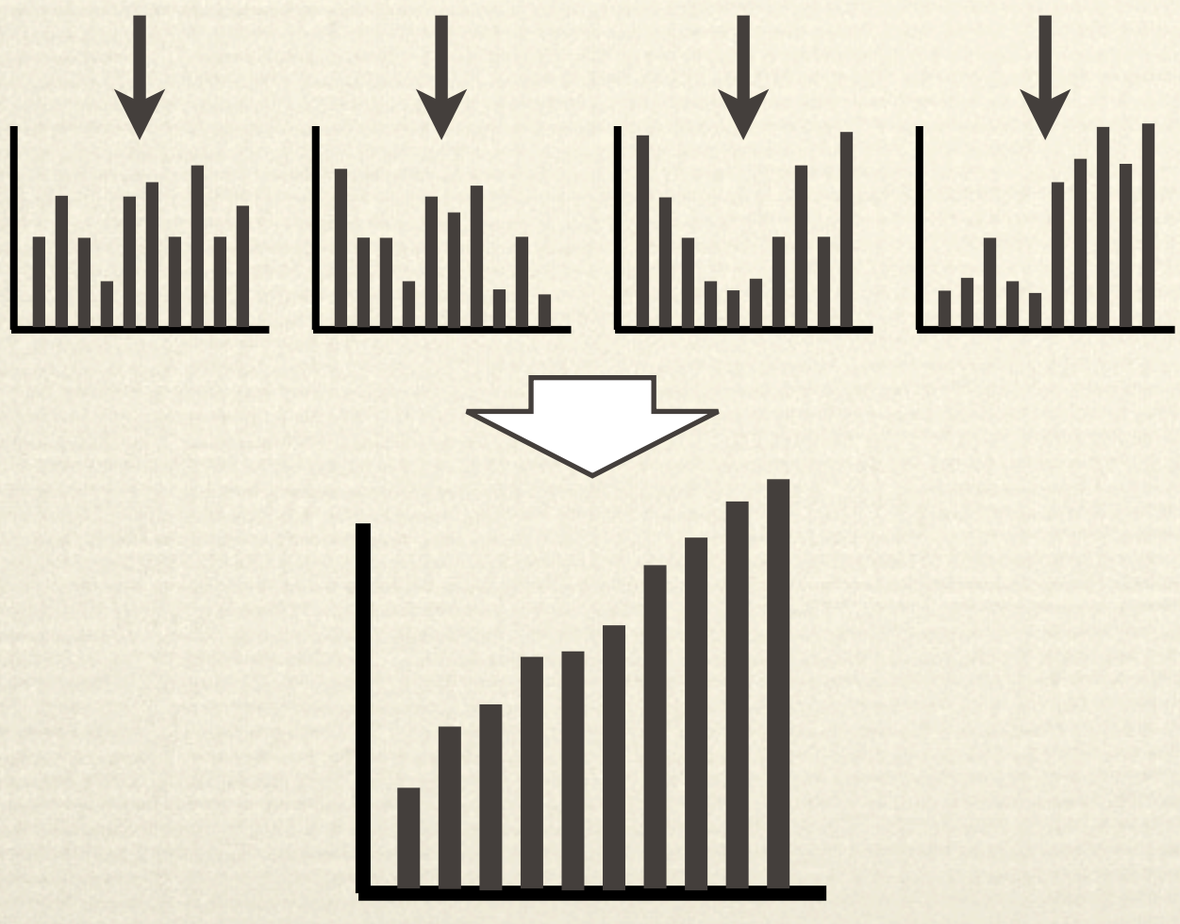
（値の入れ替え）

Co-arrayを用いた通信

IS-noHの概要

ソート対象のキーの配列





キーの値の個数を調べる +
累積の計算

全ノードの累積値を計算
(リダクション操作)

IS-noHのソース

key_array[]はグローバルな配列

```
#pragma xmp loop on t(i)
for( i=0; i<NUM_KEYS; i++ ){
    key_buff2[i] = key_array[i];
    prv_buff1[key_buff2[i]]++;
}

for( i=0; i<MAX_KEY-1; i++ )
    prv_buff1[i+1] += prv_buff1[i];

#pragma xmp reduction(+:prv_buff1)
```

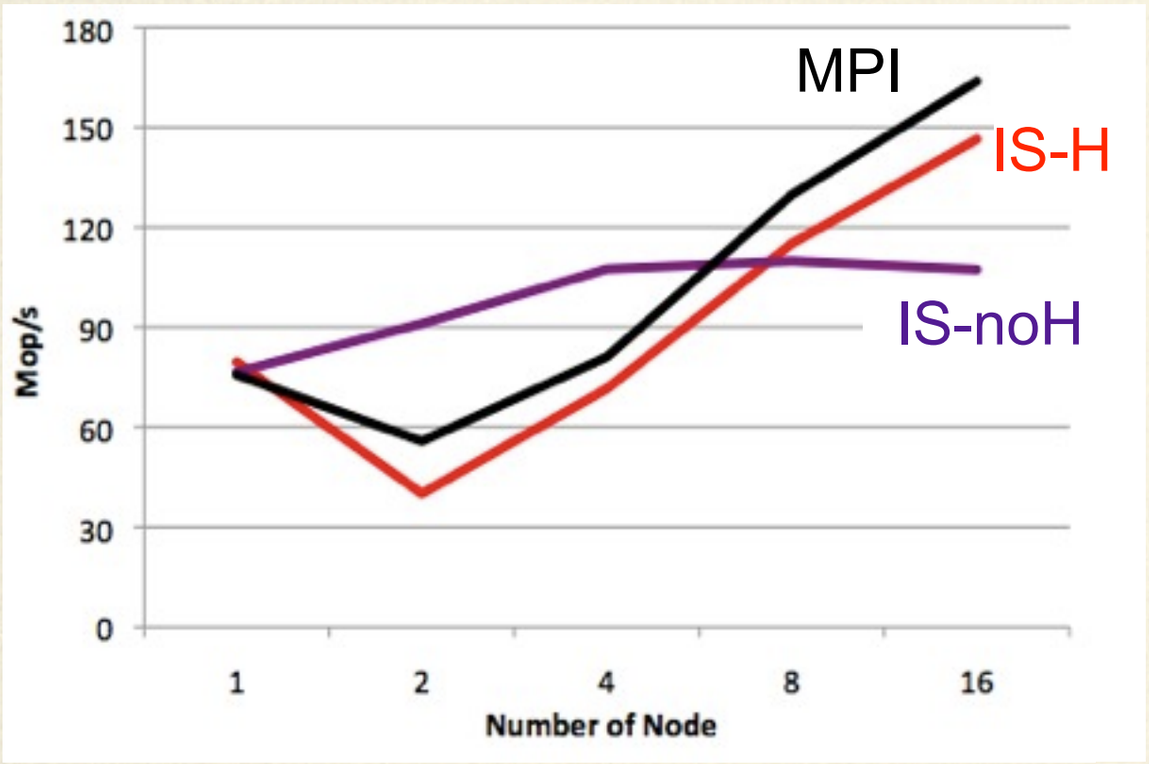
（キーの値の個数を調べる）

（累積の計算）

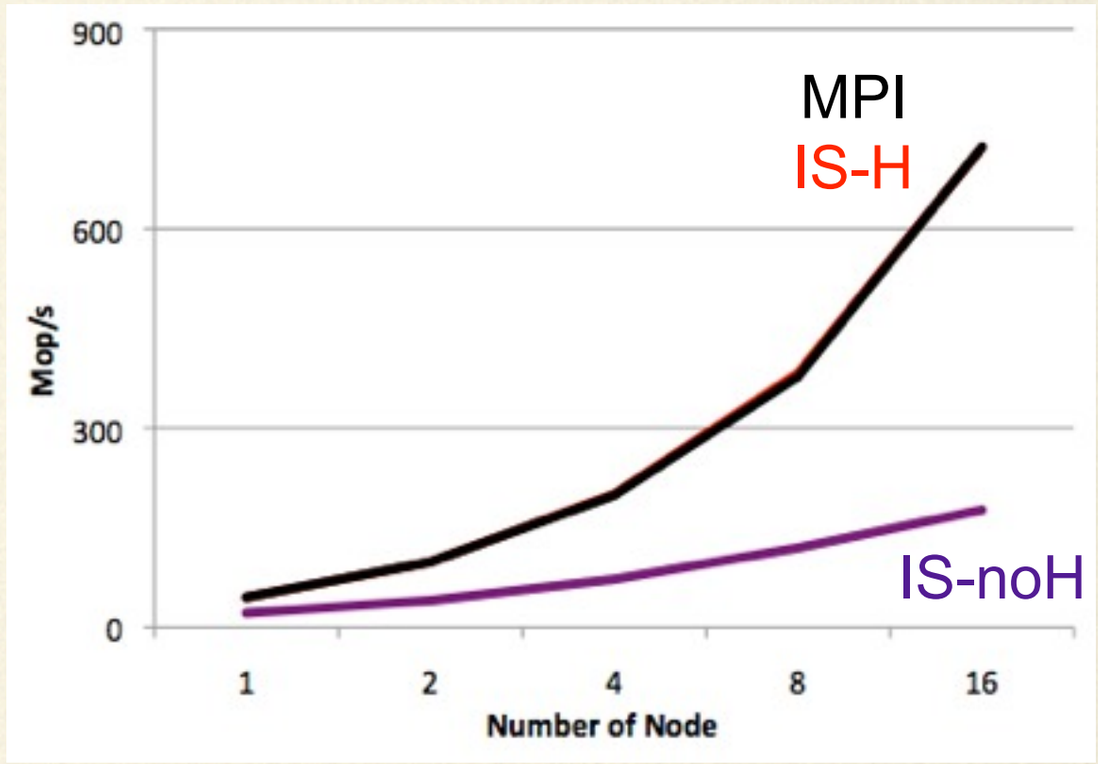
（ 全ノードの累積値を計算）

ISの結果

PC Cluster



T2K Tsukuba



MPIとヒストグラムを用いたISは同程度の性能を示す

CGの並列化

- CG-1d：プロセスを1次元に分割
- CG-2d：プロセスを2次元に分割

・2次元分割の方が，性能が高いとされている

・1次元分割の方が実装は容易

・MPIは2次元分割を用いた手法のみ提供されている

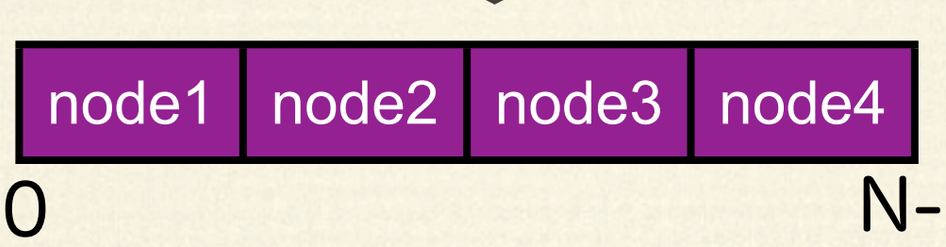
CGの概要

2次元疎行列

a[ ]

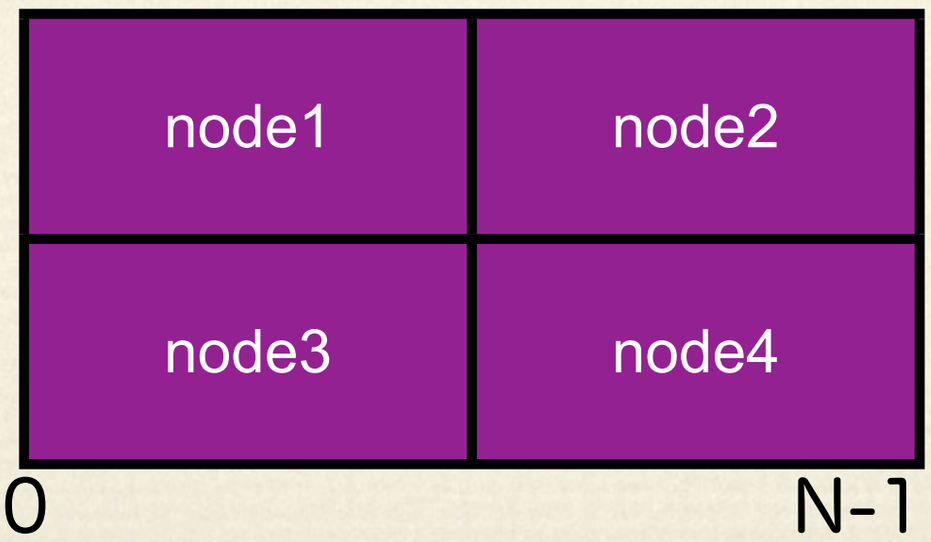
→ CG（共役勾配法） → 固有値
（特徴量の一種）

1次元分割
（CG-1d）



template t(0:N-1)

2次元分割
（CG-2d）



template t(0:N-1, 0:N-1)

CG-1dのソース

w, p, qはグローバルな配列

```
#pragma xmp shadow p[*]
```
（シャドウの宣言）

```
 ・ ・ ・

#pragma xmp reflect p
```
（pの同期）

```
#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;
}
```
（行列・ベクトル積）

```
#pragma xmp loop on t(j)
for (j = 1; j <= lastrow-firstrow+1; j++)
    q[j] = w[j];
```
（ベクトルの代入）

CG-2dのソース

w, p, qはグローバルな配列

```
#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;
    }
```

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

（行列・ベクトル積）

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

（ベクトルの集約）

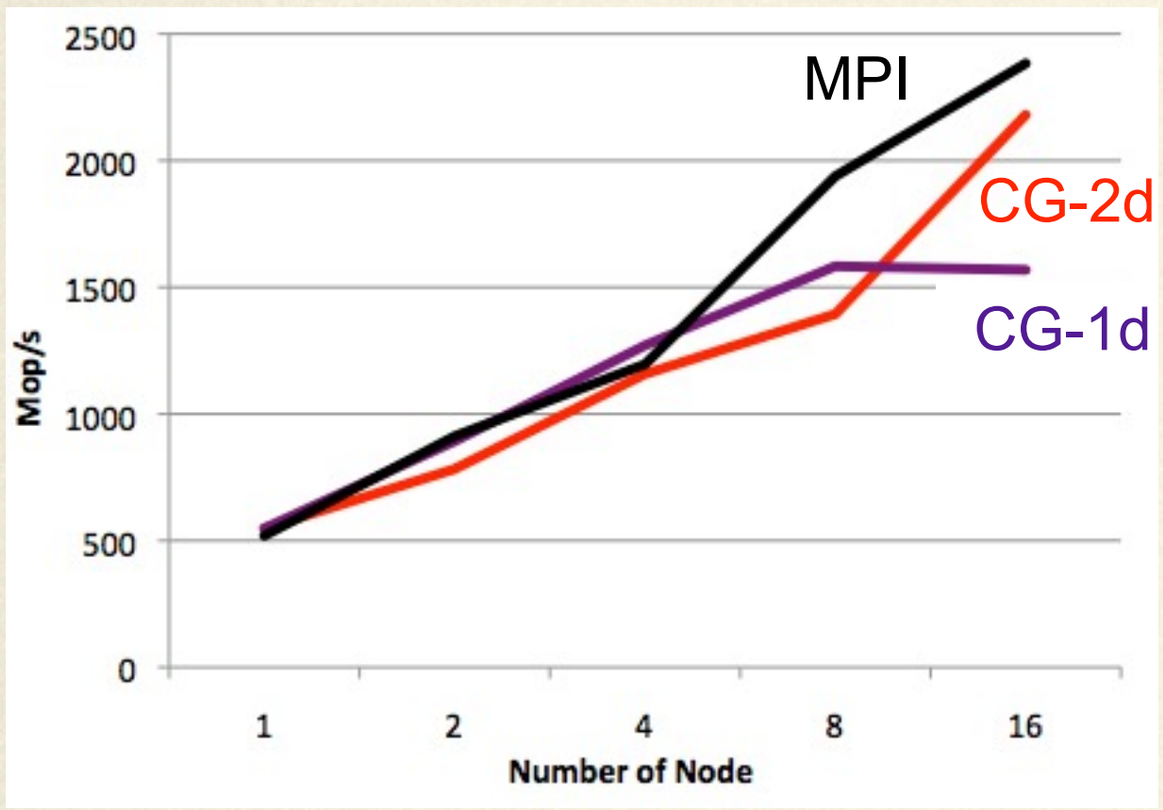
```
#pragma xmp gmove
    q[:] = w[:];
```

（ベクトルの代入）

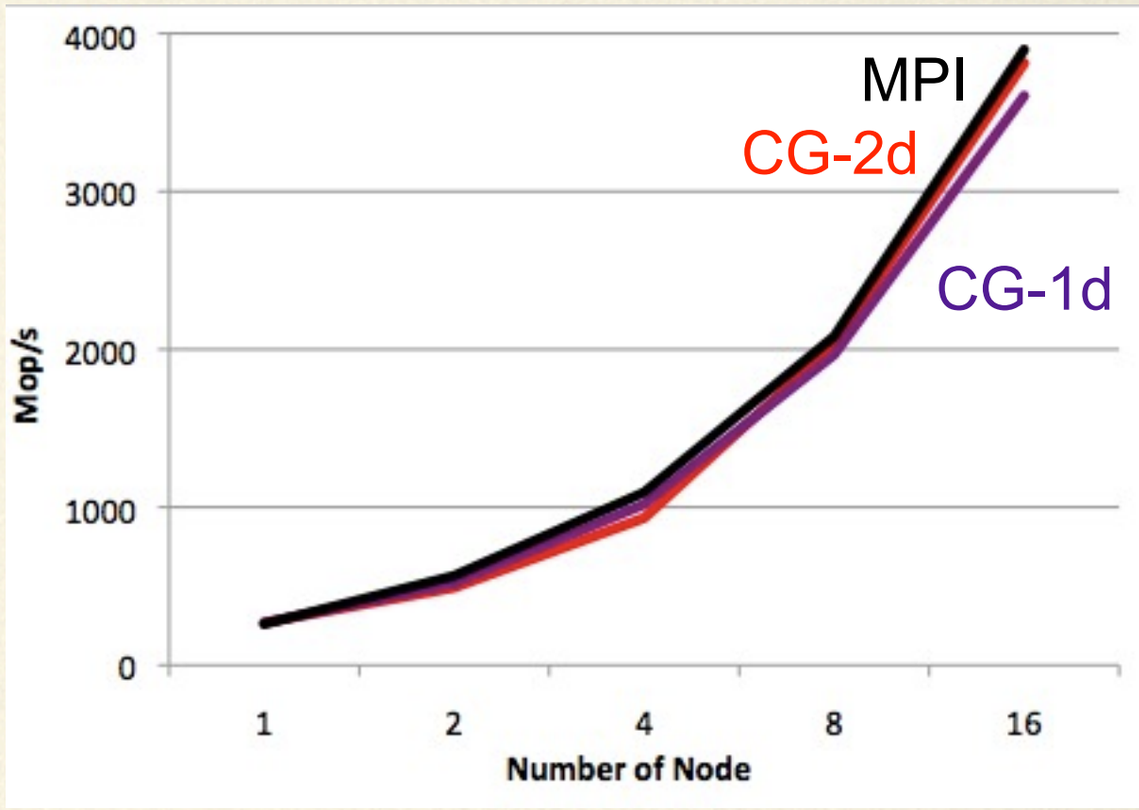
rowstrとcolidxを事前に計算し，2重ループの外側と内側ともに並列化
分け方の異なるqとwをgmoveを用いて代入（転送）する

CGの結果

PC Cluster



T2K Tsukuba



MPIとCG-2dは同程度の性能を示す

特にT2K Tsukubaではすべてが同程度の性能を示す

結果のまとめと考察

- MPIのアルゴリズムと同様のXcalableMPプログラムを作成すると，ほぼ同程度の性能がでる
  （特に高速ネットワークを用いると）
- CPUとネットワークに要する時間を調べた結果，CPU時間にはほとんど差はなく，ネットワークのみ差が現れた（CPUのオーバヘッドは少ない）
- 今後は，ノード数（コア数）を多く用いて実験し，効率の良い通信の開発を行う予定
- マルチコアノードへの対応：OpenMP-MPIハイブリッド

感想（1/2）

- 配列を分散化せず，1つの配列として扱うことが可能なので，逐次プログラミングからの敷居が低く感じた
- 逐次コードを変更しなくて良いため，コード管理が楽
- アプリケーションによっては，逐次実行を行いながら少しずつの並列化が可能なため，どの箇所を並列化すれば高速化できるのかの見極めが行いやすい

感想（2/2）

- Alltoall(v)通信などの，複雑な通信パターンの場合，XcalableMPのみでは記述が難しい
  - 既存のMPI関数を使いたい場合もある
    - 分散化された配列と既存のMPI関数とのインタフェースを作成することで，MPI関数を利用可能
- ユーザから，グローバルな配列のどの要素がどのノードに割り当てられているかわからない
  - 計算時間の見積もりや細かい操作を行いたいときがある
  - 専用のAPIが必要？

全体のまとめ

- 高性能並列プログラミング環境XcalableMPについての概要，性能評価および感想
- 性能評価では，MPIとほぼ同程度の性能を示した
- 感想として，分散された配列を1つの配列のように扱える点，並列化による高速化が行いやすい点が非常に便利と感じた