大規模系での高速フーリエ変換1

高橋大介
daisuke@cs.tsukuba.ac.jp
筑波大学計算科学研究センター

講義内容

- 高速フーリエ変換（FFT）
- ブロック三次元FFTアルゴリズム
- 並列FFTアルゴリズム

高速フーリエ変換(FFT)

- 高速フーリエ変換(FFT)は，離散フーリエ変換(DFT)を高速に計算するアルゴリズム.
- 理学分野での応用例
  - 偏微分方程式の解法
  - 畳み込み，相関の計算
  - 第一原理計算における密度汎関数法
- 工学分野での応用例
  - スペクトラムアナライザ
  - CTスキャナ・MRIなどの画像処理
  - 地上波デジタルテレビ放送や無線LANで用いられているOFDM(直交周波数多重変調)では，変復調処理にFFTを用いている.

離散フーリエ変換（DFT）

- 離散フーリエ変換（DFT）の定義

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}$$

$$0 \le k \le n-1, \quad \omega_n = e^{-2\pi i/n}$$

行列によるDFTの定式化（1/4）

- $n = 4$ のとき，DFTは以下のように計算できる．

$$y(0) = x(0)\omega^0 + x(1)\omega^0 + x(2)\omega^0 + x(3)\omega^0$$

$$y(1) = x(0)\omega^0 + x(1)\omega^1 + x(2)\omega^2 + x(3)\omega^3$$

$$y(2) = x(0)\omega^0 + x(1)\omega^2 + x(2)\omega^4 + x(3)\omega^6$$

$$y(3) = x(0)\omega^0 + x(1)\omega^3 + x(2)\omega^6 + x(3)\omega^9$$

行列によるDFTの定式化（2/4）

- 行列を用いると，より簡単に表すことができる．

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

- n^2 回の複素数の乗算と，$n(n-1)$ 回の複素数の加算が必要．

行列によるDFTの定式化(3/4)

- $\omega_n^{jk} = \omega_n^{jk \bmod n}$ の関係を用いると，以下のように書き直すことができる.

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^0 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

行列によるDFTの定式化(4/4)

- 行列の分解により，乗算回数を減らすことができる．

$$\begin{bmatrix} y(0) \\ y(2) \\ y(1) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & \omega^0 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ 0 & 0 & 1 & \omega^1 \\ 0 & 0 & 1 & \omega^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & \omega^0 & 0 \\ 0 & 1 & 0 & \omega^0 \\ 1 & 0 & \omega^2 & 0 \\ 0 & 1 & 0 & \omega^2 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

これを再帰的に行うと，演算量を $O(n \log n)$ にできる(データ数 n は合成数である必要がある)

DFTとFFTの演算量の比較

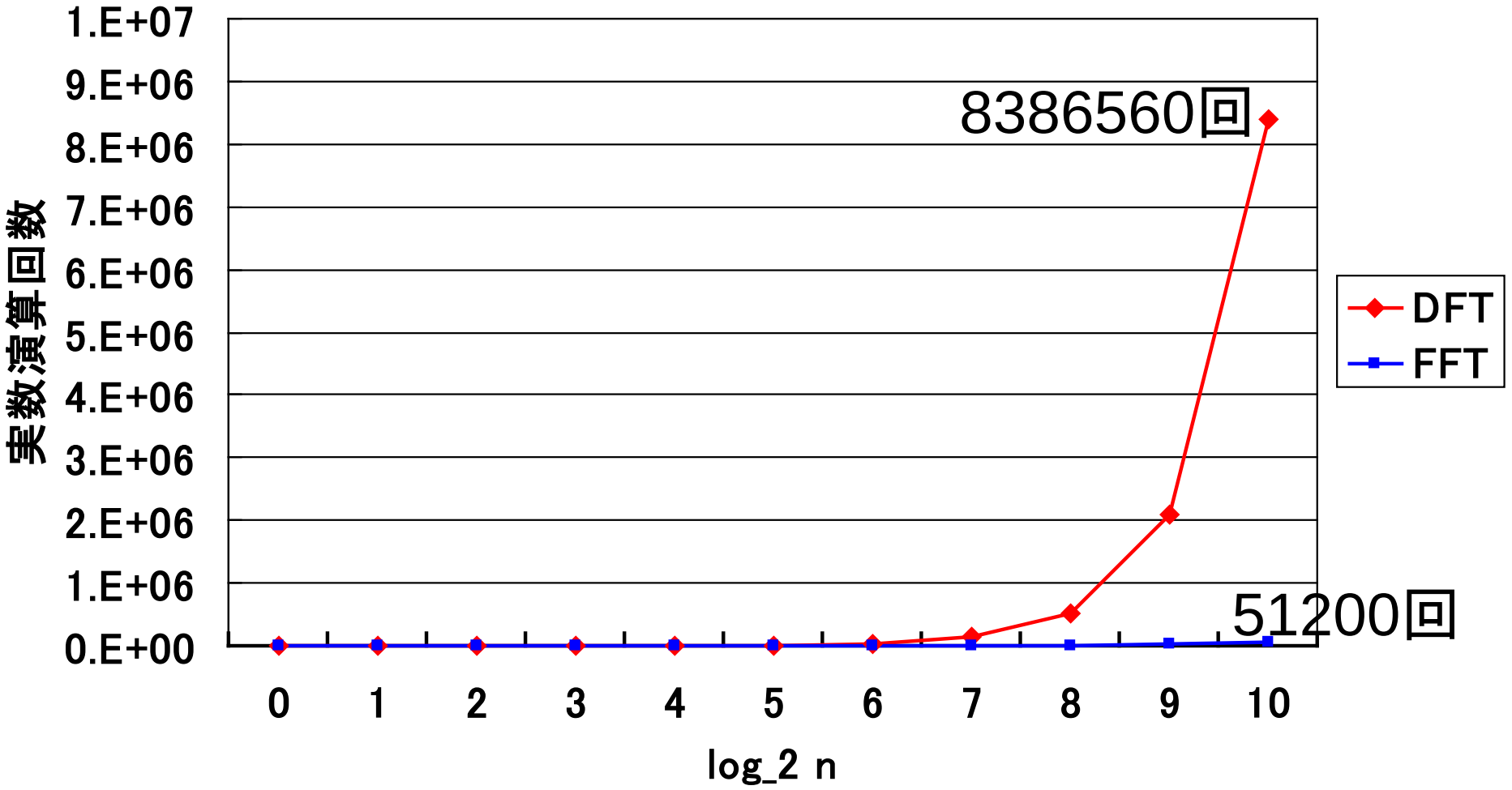
- DFTの実数演算回数

$$T_{DFT} = 8n^2 - 2n$$

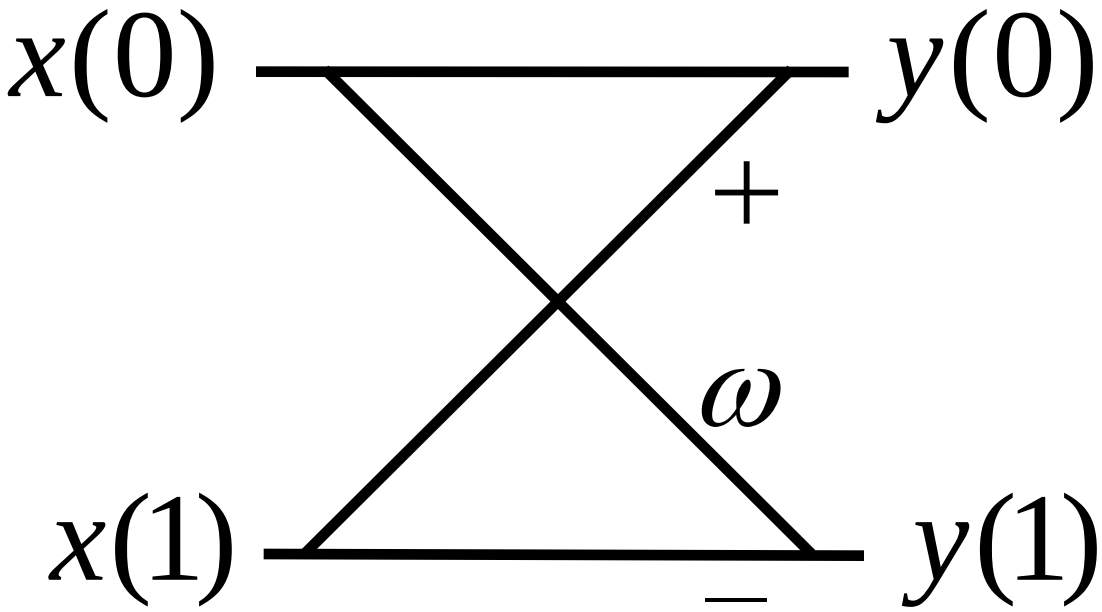
- FFTの実数演算回数
（n が2のべきの場合）

$$T_{FFT} = 5n \log_2 n$$

DFTとFFTの演算量の比較



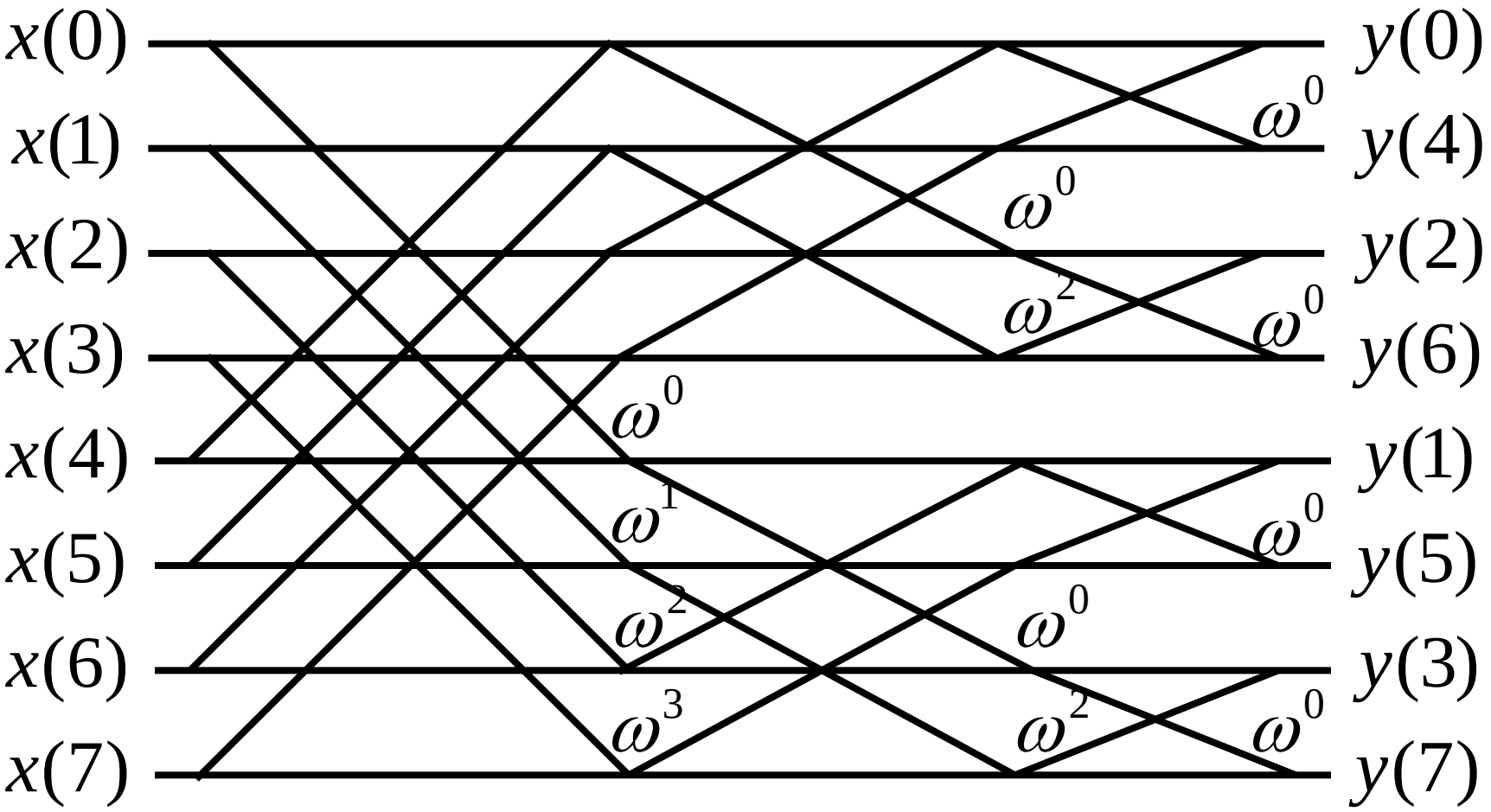
バタフライ演算



$$y(0) = x(0) + x(1)$$

$$y(1) = \omega\{x(0) - x(1)\}$$

Cooley-Tukey FFTの信号流れ図



FFTのカーネル部分の例

```
      SUBROUTINE FFT2(A,B,W,M,L)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION A(2,M,L,*),B(2,M,2,*),W(2,*)
C
      DO J=1,L
          WR=W(1,J)
          WI=W(2,J)
          DO I=1,M
              B(1,I,1,J)=A(1,I,J,1)+A(1,I,J,2)
              B(2,I,1,J)=A(2,I,J,1)+A(2,I,J,2)
              B(1,I,2,J)=WR*(A(1,I,J,1)-A(1,I,J,2))-WI*(A(2,I,J,1)-A(2,I,J,2))
              B(2,I,2,J)=WR*(A(2,I,J,1)-A(2,I,J,2))+WI*(A(1,I,J,1)-A(1,I,J,2))
          END DO
      END DO
      RETURN
      END
```

$n = n_1 n_2$に対するFFTアルゴリズム

- $n = n_1 n_2$で与えられるとする.

$$j = j_1 + j_2 n_1 \quad j_1 = 0, 1, \cdots, n_1 - 1 \quad j_2 = 0, 1, \cdots, n_2 - 1$$

$$k = k_2 + k_1 n_2 \quad k_1 = 0, 1, \cdots, n_1 - 1 \quad k_2 = 0, 1, \cdots, n_2 - 1$$

- 上記の表現を用いると, DFTの定義式を以下のように書き換えることができる.

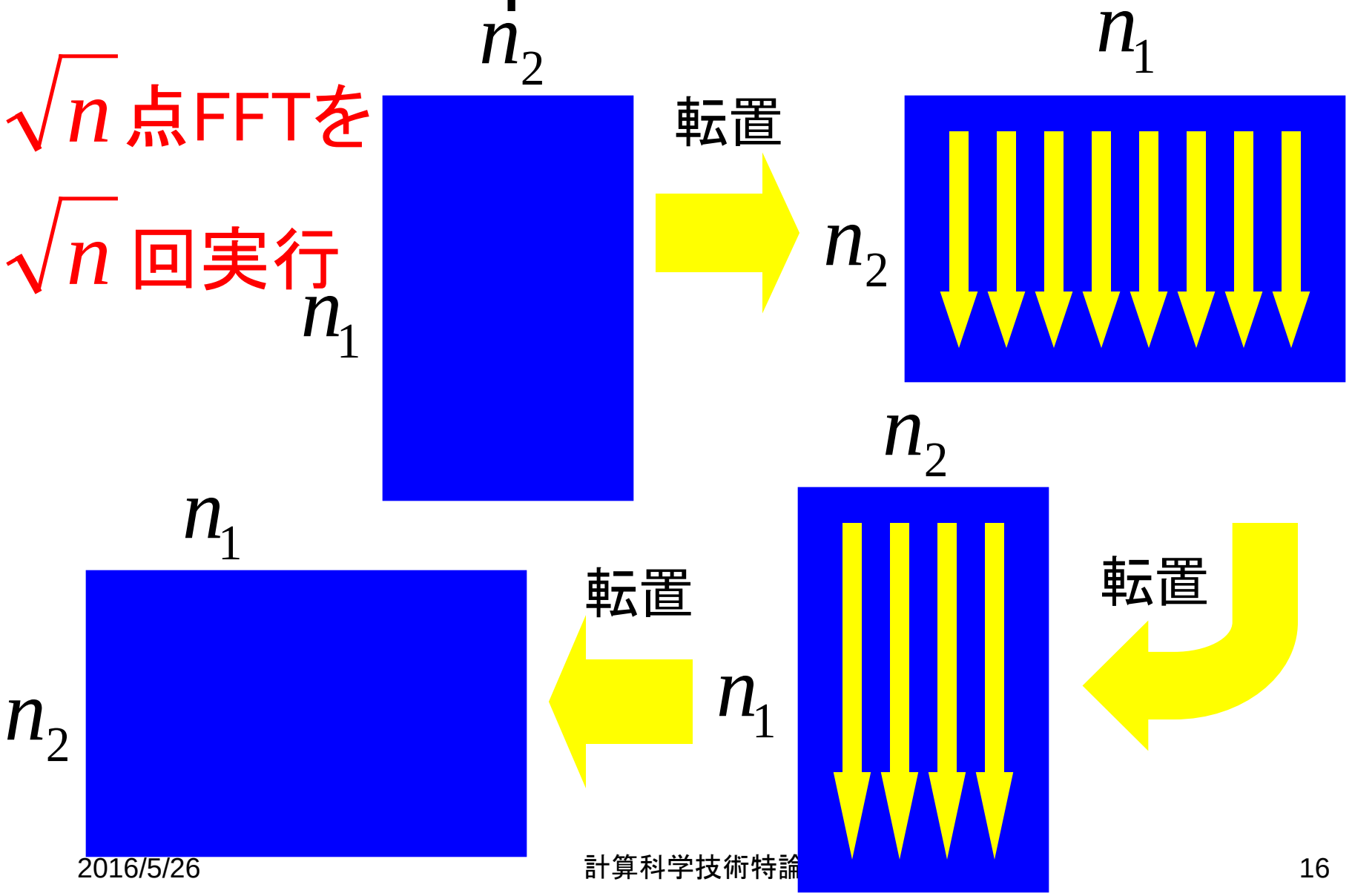
$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \left[\sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \right] \omega_{n_1}^{j_1 k_1}$$

- n 点FFTを n_1 点FFTと n_2 点FFTに分解している.

Six-Step FFTアルゴリズム

1. 行列の転置
2. n_1 組の n_2 点 multicolumn FFT
3. ひねり係数($\omega_{n_1 n_2}^{j_1 k_2}$)の乗算
4. 行列の転置
5. n_2 組の n_1 点 multicolumn FFT
6. 行列の転置

Six-Step FFTアルゴリズム



Six-Step FFTのプログラム例

```
      SUBROUTINE FFT(A,B,W,N1,N2)
      COMPLEX*16 A(*),B(*),W(*)
C
      CALL TRANS(A,B,N1,N2)                 N1 x N2行列をN2 x N1行列に転置
      DO J=1,N1
         CALL FFT2(B((J-1)*N2+1),N2)        N1組のN2点multicolumn FFT
      END DO
      DO I=1,N1*N2
         B(I)=B(I)*W(I)                     ひねり係数（W）の乗算
      END DO
      CALL TRANS(B,A,N2,N1)                 N2 x N1行列をN1 x N2行列に転置
      DO J=1,N2
        CALL FFT2(A((J-1)*N1+1),N1)         N2組のN1点multicolumn FFT
      END DO
      CALL TRANS(A,B,N1,N2)                 N1 x N2行列をN2 x N1行列に転置
      RETURN
      END
```

ブロック三次元FFTアルゴリズム

背景

- 多くのFFTアルゴリズムはデータがキャッシュに載っている場合には高い性能を示す.
- しかし，問題サイズがキャッシュより大きくなった場合，著しく性能が低下する.
- FFTアルゴリズムにおける目標としては，
  - いかにしてメモリアクセスを上位のメモリ階層に封じ込めるか. つまり，いかにキャッシュミスの回数を減らすか.
  - かつ主記憶のアクセス回数をできるだけ減らす.
- 通常の三次元 FFTでは，
  - 3回のmulticolumn FFT
  - 3回の行列の転置 → ボトルネック

  が必要.

目的

- ボトルネックとなる行列の転置は，ブロック化することにより、キャッシュヒット率を上げることが可能.
- しかし，通常の三次元FFTではmulticolumn FFTと行列の転置が分離されている.
  - 主記憶からキャッシュにロードされたデータは，行列の転置だけ，もしくはmulticolumn FFTにしか用いられていない.
- 行列の転置を単にブロック化するだけでは，キャッシュ内のデータの再利用性が低い.

方針

- 行列の転置で用いられているキャッシュ内のデータをmulticolumn FFTでも活用し，キャッシュ内のデータの再利用性を高くする.
- 主記憶からキャッシュにいったんロードしたデータは，できるだけキャッシュに載っているようにする.
- キャッシュ内のデータは再利用できるだけ再利用し，本当にいらなくなった時点で主記憶に書き戻す.
- 配列にパディングを用いて，キャッシュ内の貴重なデータが主記憶に逃げないようにする.

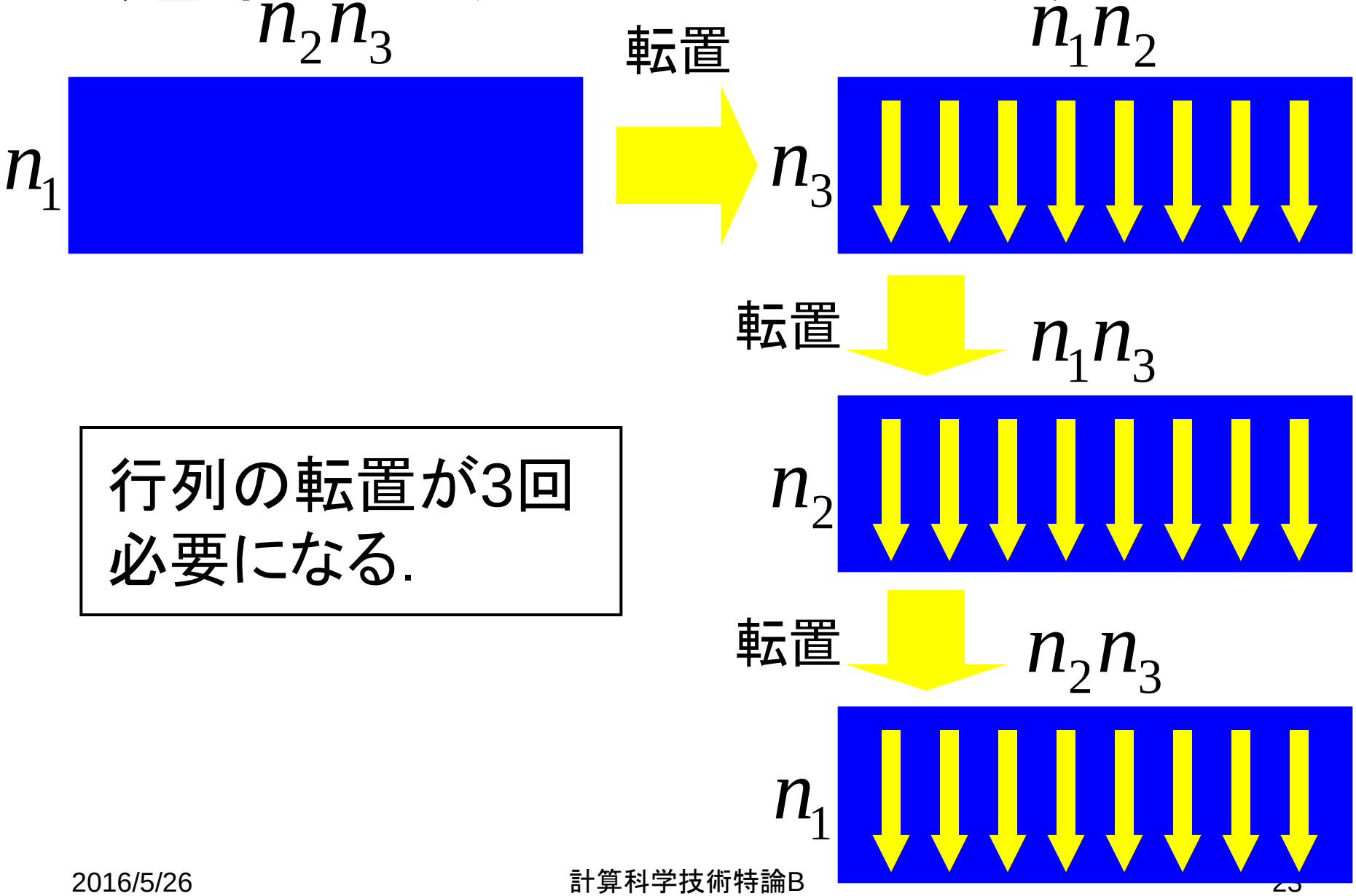
三次元FFT

- 三次元離散フーリエ変換（DFT）の定義

$$y(k_1,k_2,k_3)=\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\sum_{j_3=0}^{n_3-1}x(j_1,j_2,j_3)\,\omega_{n_3}^{j_3k_3}\omega_{n_2}^{j_2k_2}\omega_{n_1}^{j_1k_1}$$

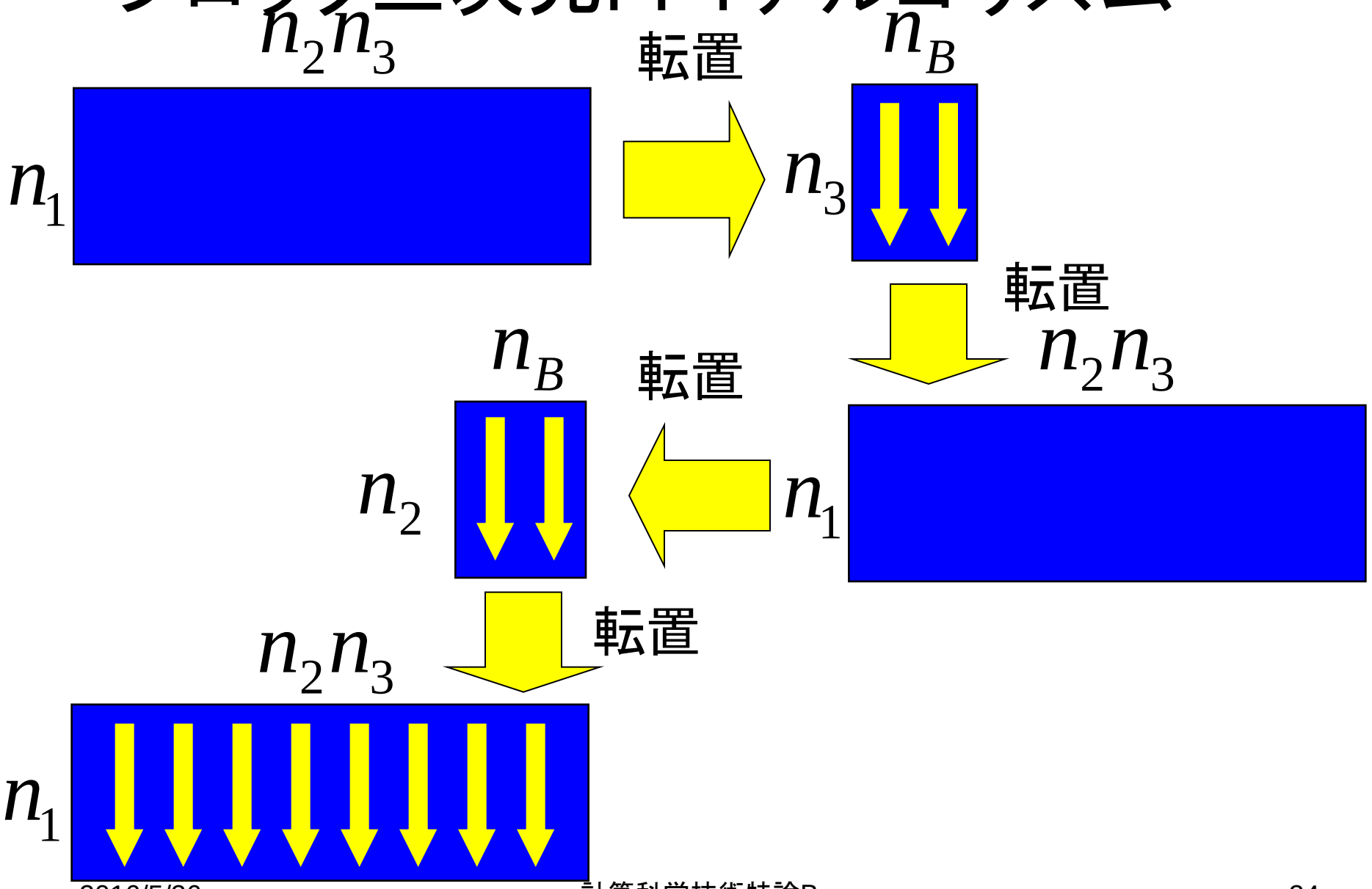
$$0\le k_r\le n_r-1,\ \omega_{n_r}=\exp(-2\pi i/n_r)$$

通常の三次元FFTアルゴリズム



行列の転置が3回
必要になる.

ブロック三次元FFTアルゴリズム



ブロック三次元FFTの利点

- Cooley-Tukey FFT等の通常のFFTアルゴリズムでは, $n = n_1 n_2 n_3$ とすると,
  - 演算回数: $5n \log_2 n$ （基数2の場合）
  - 主記憶アクセス回数: $4n \log_2 n$
- ブロック三次元FFTでは,
  - 演算回数: $5n \log_2 n$
  - 主記憶アクセス回数: 理想的には $12n$
    $n_1 n_2$ 点のデータがキャッシュに入る場合: $8n$

In-Cache FFTアルゴリズム

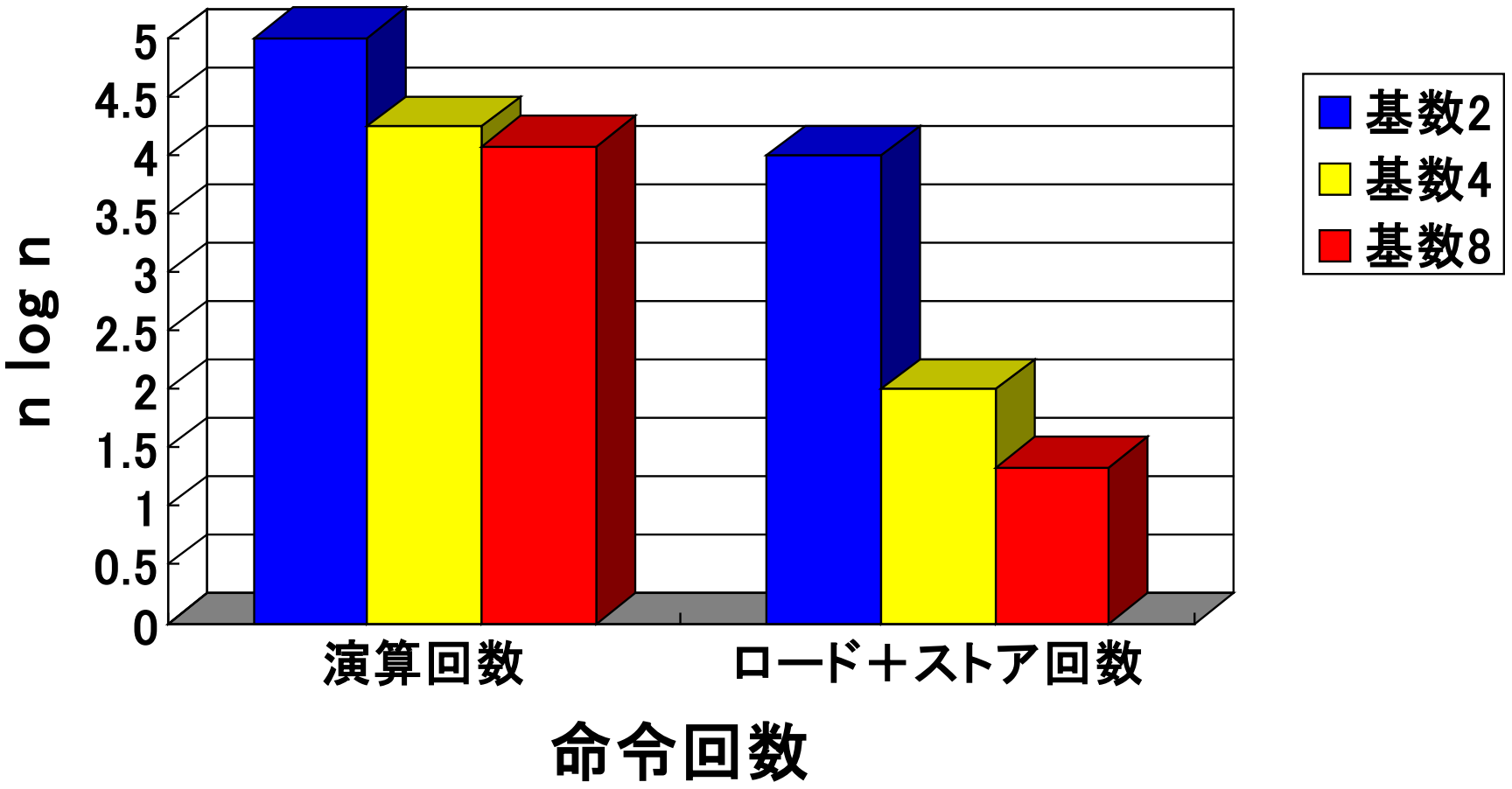
- Multicolumn FFTにおいて，各column FFTがキャッシュに載る場合のin-cache FFTとして，
  - Cooley-Tukeyアルゴリズム（ビットリバース処理が必要）
  - Stockhamアルゴリズム（ビットリバース処理は不要）

  が考えられる．
- 高い基数のFFTを積極的に用いることにより，メモリアクセス回数を減らす．
  - 基数4，8のFFTを組み合わせて使用．
  - 基数8よりもメモリアクセス回数の多い，基数4のFFTを最大2ステップに抑えることにより，さらにメモリアクセスを減らす．

最内側ループにおける ロード＋ストア，乗算および加算回数

	基数2	基数4	基数8
ロード＋ストア回数	8	16	32
乗算回数	4	12	32
加算回数	6	22	66
総浮動小数点演算回数 $(n\log_2 n)$	5	4.25	4.083
浮動小数点演算命令数	10	34	98
浮動小数点演算命令数とロード＋ストア回数の比	1.25	2.125	3.063

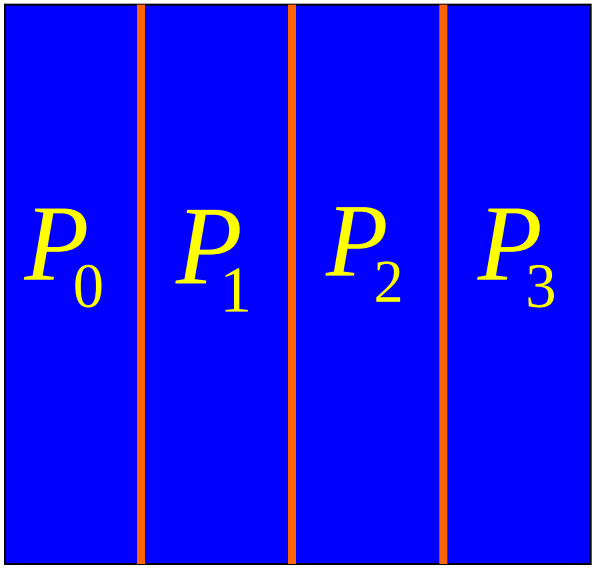
FFTカーネルを用いた場合の n点FFTの必要演算回数

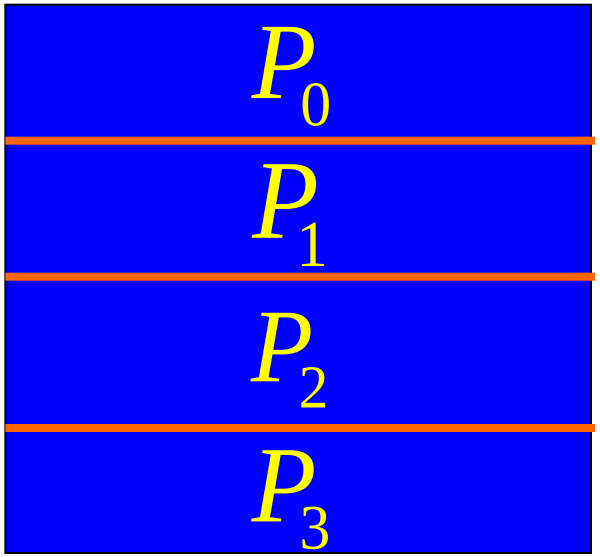


並列FFTアルゴリズム

配列の分割方法(1/4)

- MPIで並列化を行う際には，各ノードで配列を分割して持つようにすると，メモリを節約することができる．
- ブロック分割
  - 連続する領域をノード数で分割



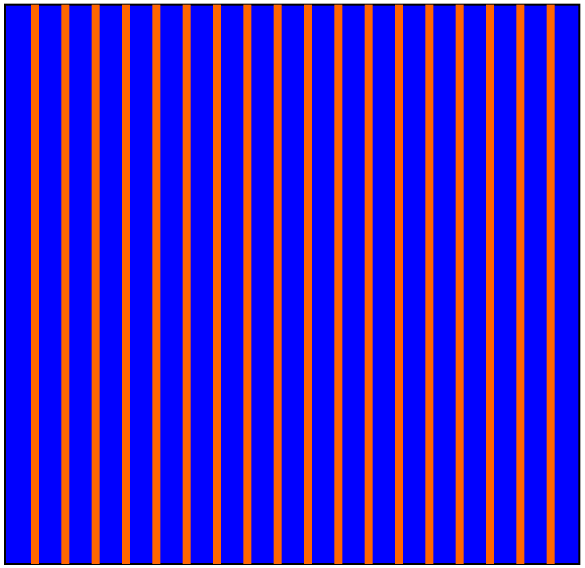


列方向に分割したブロック分割　　行方向に分割したブロック分割

配列の分割方法(2/4)

- サイクリック分割
  - 1列(または1行)ごとに分割
  - ブロック分割に比べてロードバランスが取りやすい

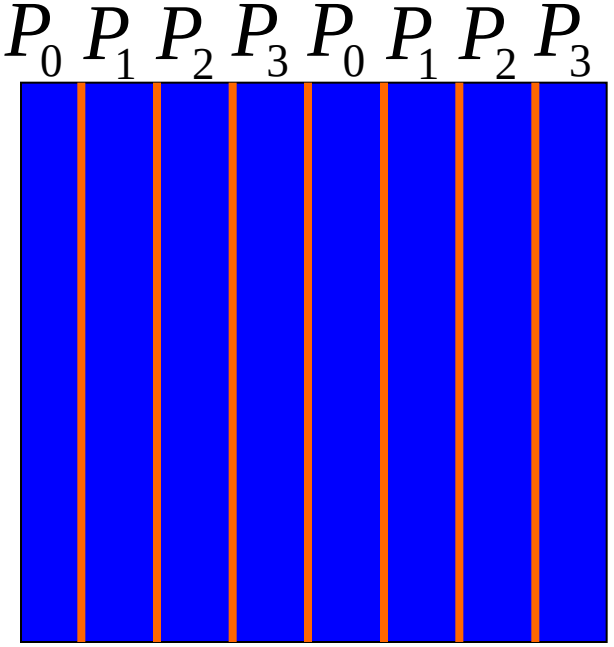
0123012301230123…



列方向に分割したサイクリック分割

配列の分割方法(3/4)

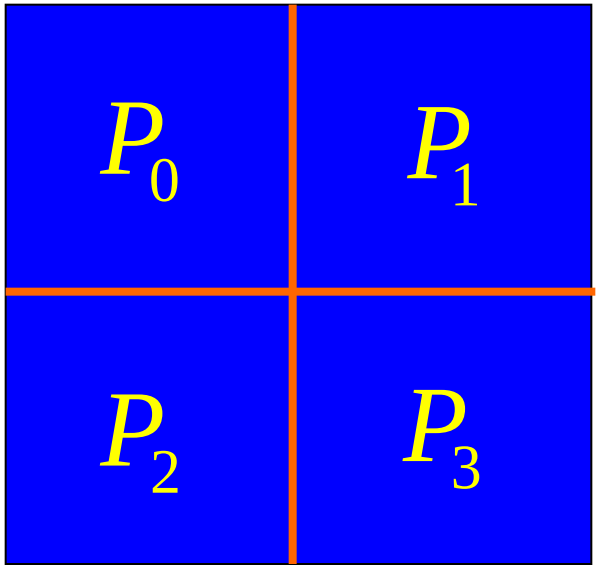
- ブロックサイクリック分割
  - 複数列(または複数行)ごとに分割
  - ブロック分割とサイクリック分割の中間



列方向に分割したブロックサイクリック分割

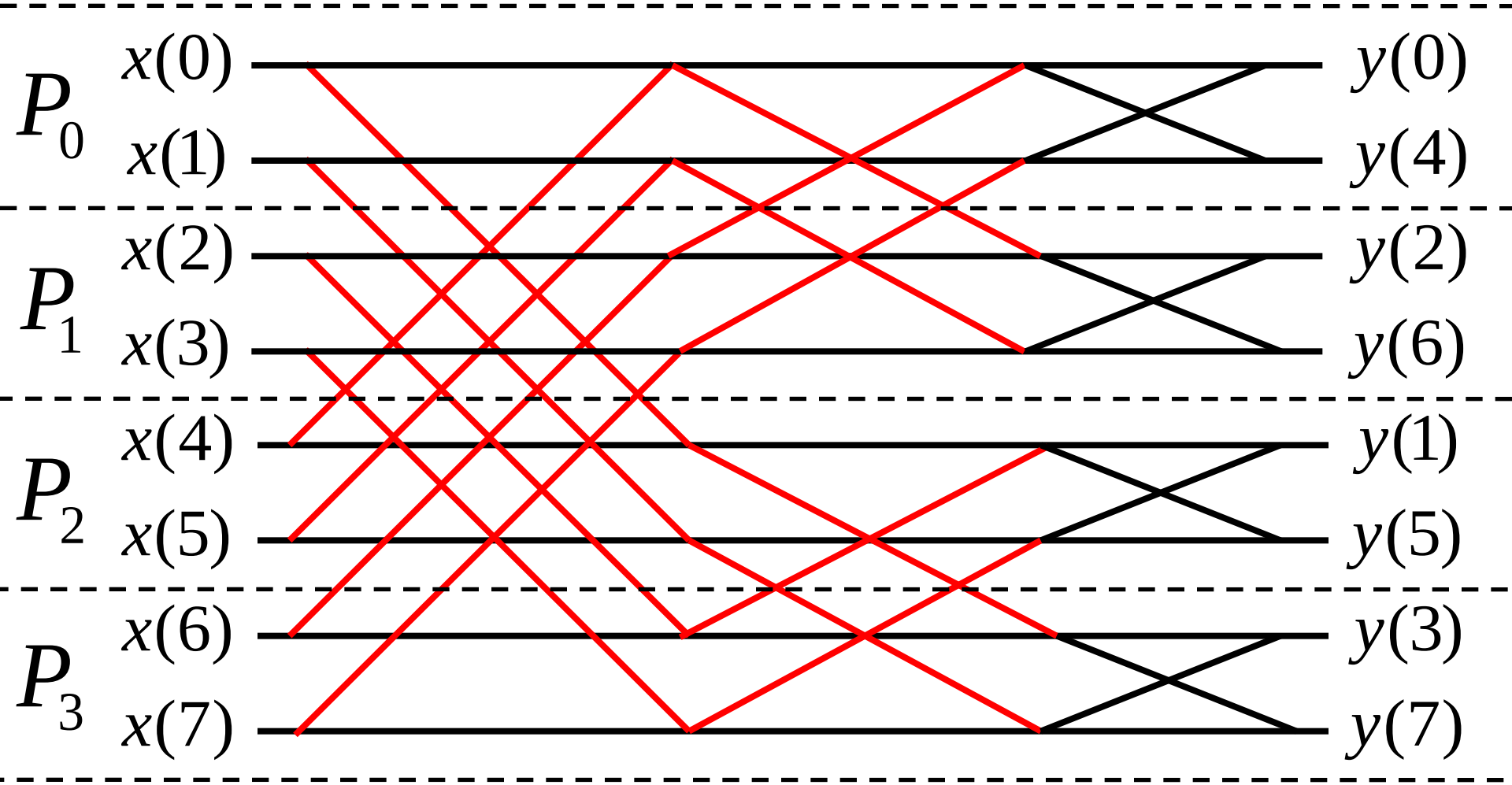
配列の分割方法(4/4)

- 二次元分割
  - 行方向と列方向の両方を分割
  - 一次元分割よりも通信量が減ることがある
  - 二次元のブロック分割，サイクリック分割，ブロックサイクリック分割が考えられる.



二次元ブロック分割

Cooley-Tukey FFTの並列化

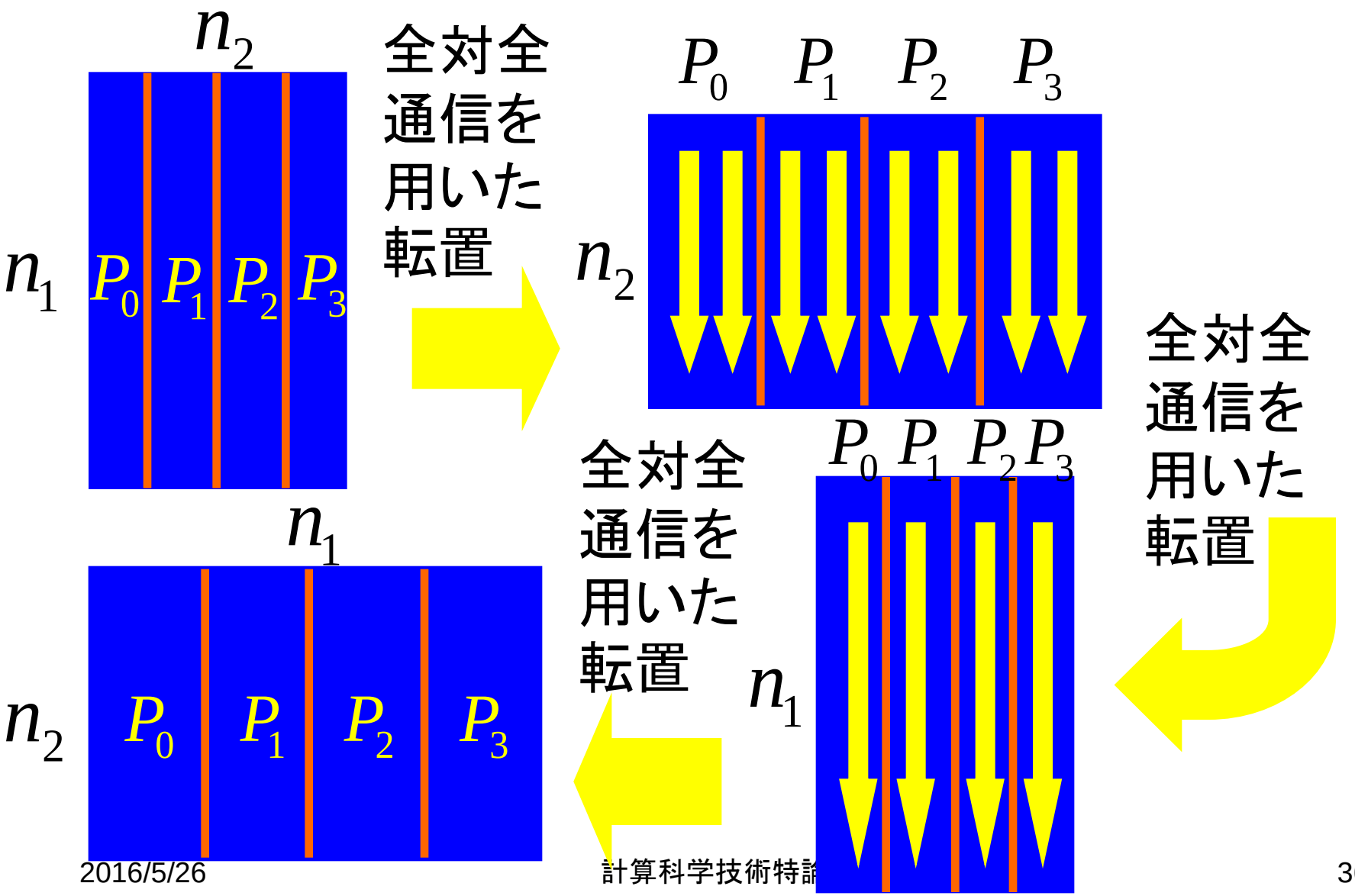


並列Cooley-Tukey FFTの通信量

- ノード数を P とすると，並列Cooley–Tukey FFTでは，$\log_2 P$ ステージの通信が必要になる.
- 各ステージでは n/P 個の倍精度複素数データの通信（MPI_Send, MPI_Recv）が行われるため，合計の通信量は，

$$T_{Cooley–Tukey} = \frac{16n}{P}\log_2 P \quad （バイト）$$

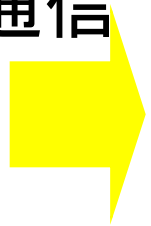
並列Six-Step FFTアルゴリズム



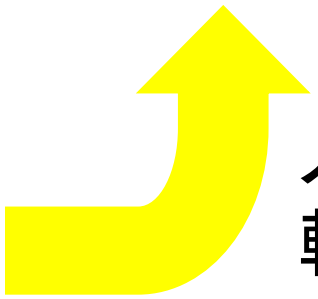
全対全通信(MPI_Alltoall)を用いた行列の転置

P0	P1	P2	P3
0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

全対全通信



P0	P1	P2	P3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15
16	18	20	22
17	19	21	23
24	26	28	30
25	27	29	31



ノード内転置

P0	P1	P2	P3
0 1	2 3	4 5	6 7
8 9	10 11	12 13	14 15
16 17	18 19	20 21	22 23
24 25	26 27	28 29	30 31

並列Six-Step FFTのプログラム例

```
      SUBROUTINE PARAFFT(A,B,W,N1,N2,NPU)
      COMPLEX*16 A(*),B(*),W(*)
C
      CALL PTRANS(A,B,N1,N2,NPU)          N1 x N2行列をN2 x N1行列に転置
      DO J=1,N1/NPU                       （MPI_ALLTOALLを使用）
        CALL FFT2(B((J-1)*N2+1),N2)       (N1/NPU)組のN2点multicolumn FFT
      END DO
      DO I=1,(N1*N2)/NPU
        B(I)=B(I)*W(I)                    ひねり係数（W）の乗算
      END DO
      CALL PTRANS(B,A,N2,N1,NPU)          N2 x N1行列をN1 x N2行列に転置
      DO J=1,N2/NPU                       （MPI_ALLTOALLを使用）
        CALL FFT2(A((J-1)*N1+1),N1)       (N2/NPU)組のN1点multicolumn FFT
      END DO
      CALL PTRANS(A,B,N1,N2,NPU)          N1 x N2行列をN2 x N1行列に転置
      RETURN                              （MPI_ALLTOALLを使用）
      END
```

並列Six-Step FFTの通信量

- ノード数を P とすると，並列Six−Step FFTでは，3回の全対全通信が必要になる.
- 全対全通信では，各ノードは n/P^2 個の倍精度複素数データを，自分以外の $P-1$ ノードに送ることになるため，合計の通信量は

$$T_{Six-Step} = 3 \cdot (P-1) \cdot \frac{16n}{P^2} \text{（バイト）}$$

並列Cooley-Tukey FFTと並列Six-Step FFTの通信量の比較

- 並列Cooley–Tukey FFTの通信量

$$T_{Cooley–Tukey} = \frac{16n}{P}\log_2 P$$

- 並列Six–Step FFTの通信量

$$T_{Six–Step} = 3\cdot(P-1)\cdot\frac{16n}{P^2}$$

- 両者を比較すると，$P > 8$ の場合には，並列Six–Step FFTの方が通信量が少なくなる.

まとめ（1/2）

- ブロックFFTアルゴリズムでは，out-of-core算法の考え方を応用し，オンチップキャッシュを「主記憶」とし，メインメモリを「半導体ディスク」として扱っている.
- キャッシュ内のデータの再利用率を高くすることにより，
  - キャッシュミスを削減.
  - その結果，主記憶のアクセス回数も削減.
- ブロックFFTアルゴリズムは，プロセッサの演算速度と主記憶のアクセス速度との差が大きい場合に，より効果的.

まとめ(2/2)

- 並列FFTアルゴリズムでは，問題の領域をどのように分割するかが鍵となる.
  - ブロック分割，サイクリック分割，ブロックサイクリック分割
- 並列FFTでは通信部分はAll-to-All通信が中心となるので，並列化は比較的容易.
- 通信量を削減するだけではなく，ブロック化等を用いたメモリアクセスの局所化も重要となる.