高速フーリエ変換とその並列化 (I)

2003年6月6日

山本有作

今回の講義の目標 (1)

- FFTの原理と基本的な技法を学ぶ。
  - 信号処理，偏微分方程式の求解など，広い応用範囲
  - メーカー提供のライブラリ，フリーウェアなどが多数存在
  - しかし，FFTには用途に応じて様々な変種が存在
    - 実数データのFFT，分散メモリ向けのFFT，など
  - 使いたいタイプのFFTが，ライブラリにあるとは限らない。

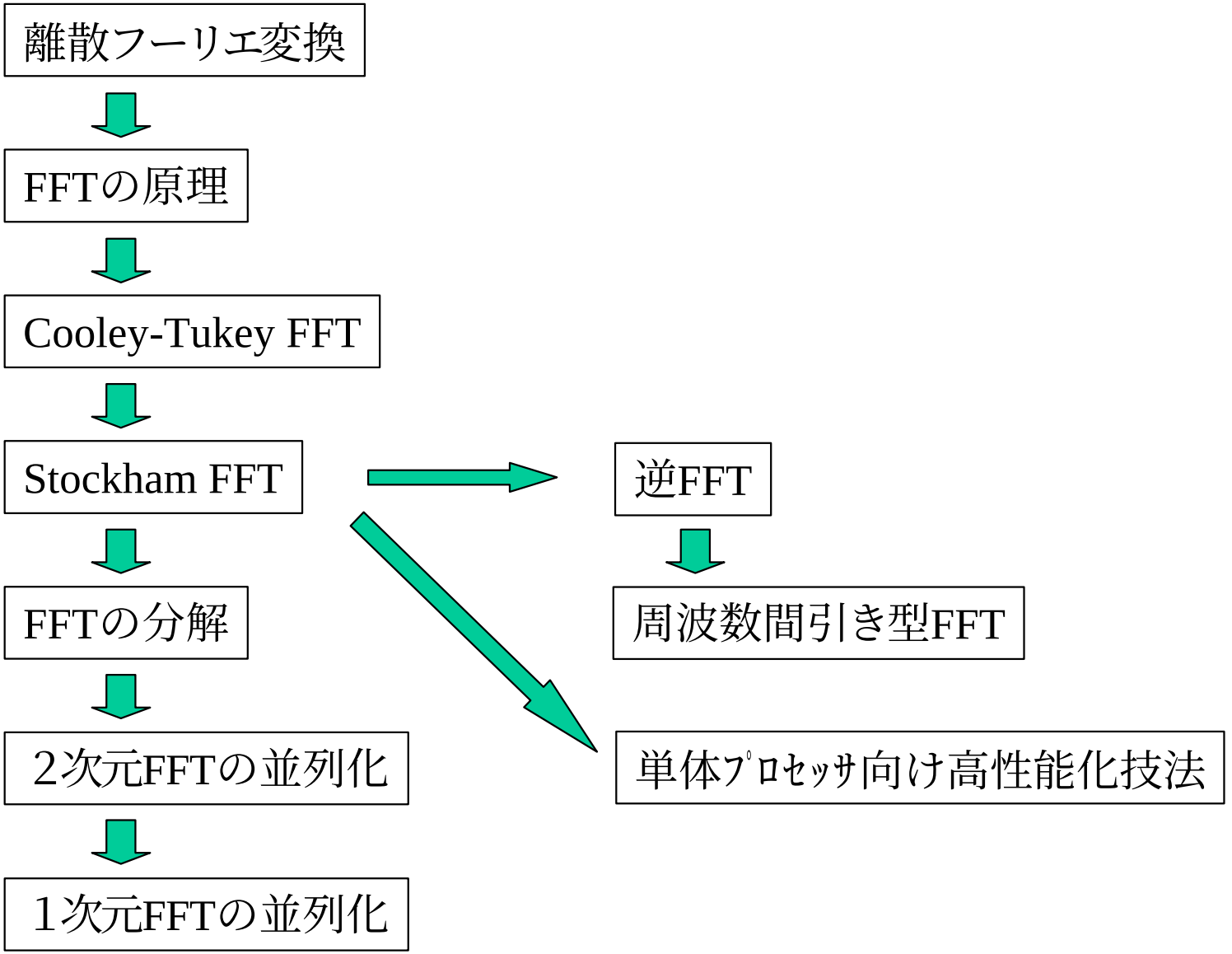
  - FFTの原理と基本的な技法を理解し，必要に応じて既存のソフトウェアを改造して使う力を身に付ける。
  - FFTに関するテクニカルタームを学ぶ。

今回の講義の目標 (2)

- FFTを通じて並列処理及び高性能計算の基本的技法を学ぶ。
  - アルゴリズムの並列化技法
  - 並列アルゴリズムの性能分析法
  - 通信オーバーヘッド削減のための技法
  - ベクトルプロセッサ，キャッシュマシン向けの高性能化技法

もくじ

今回の講義の流れ


離散フーリエ変換
FFTの原理
Cooley-Tukey FFT
Stockham FFT
FFTの分解
2次元FFTの並列化
1次元FFTの並列化
逆FFT
周波数間引き型FFT
単体ﾌﾟﾛｾｯｻ向け高性能化技法

1．FFTの基礎

- 離散フーリエ変換
- FFTの原理
- Cooley-Tukey FFT
- Stockham FFT
- 逆FFTと周波数間引き型FFT
- 多次元FFT
- FFTの分解

1.1　離散フーリエ変換 (1)

- DFTの定義
  - N個の複素数データ a_0, a_1, …, a_{N-1}に対し，次の式で定義される c_0, c_1, …, c_{N-1}をその離散フーリエ変換（Discrete Fourier Transform）と呼ぶ。

$$c_k = \sum_{j=0}^{N-1} a_j \exp(-2\pi ijk / N)$$

  - いま，$w_N = \exp(-2\pi i / N)$　とおくと，上式は次のように書き直せる。

$$c_k = \sum_{j=0}^{N-1} a_j w_N^{jk}$$

  - 定義式どおりに計算すると，DFTの計算量は$O(N^2)$である。

- 逆DFT
  - DFTの逆変換は，次の式により計算できる（確認せよ）。

$$a_j = (1/N) \sum_{k=0}^{N-1} c_k \exp(2\pi ijk / N)$$

  - これを逆DFTと呼ぶ。

1.1　離散フーリエ変換 (2)

- DFTの意味
  - 区間 $[0, 2\pi]$のN等分点で定義された関数 $f(x_n) = a_n$ を複素指数関数 $\exp(ikx)$　(k=0, 1, … , N–1) の重ね合わせに分解
  - 分解の係数 c_nを求める演算がDFT
  - 逆に，係数 c_n からN等分点での値 a_nを求める演算が逆DFT

- FFTの応用
  - 信号処理
  - 微分方程式の解法
  - 畳み込み，相関の計算
  - 多項式の積，多倍長整数の乗算　→　πの計算

1.2　FFTの原理 (1)

- DFTの分解とFFT
  - *N*が2のべき乗のとき，DFTの式は，次のように2つの項に分解できる。

$$
\begin{aligned}
c_k &= \sum_{j=0}^{N-1} a_j \exp(-2pijk/N) \\
&= \sum_{j=0}^{N/2-1} a_{2j} \exp(-2pi \cdot 2jk/N) + \sum_{j=0}^{N/2-1} a_{2j+1} \exp(-2pi \cdot (2j+1)k/N) \\
&= \sum_{j=0}^{N/2-1} e_j \exp(-2pijk/N') + \exp(-2pik/N) \sum_{j=0}^{N/2-1} o_j \exp(-2pijk/N')
\end{aligned}
$$

ここで，$N' = N/2$，$e_j = a_{2j}$，$o_j = a_{2j+1}$。

更に，この式をkに関して前半と後半に分け，
$\exp(-2\pi i(k+N/2)/N) = -\exp(-2\pi ik/N)$ を用いると，

$$
\begin{aligned}
c_k &= \sum_{j=0}^{N/2-1} e_j \exp(-2pijk/N') + \exp(-2pik/N) \sum_{j=0}^{N/2-1} o_j \exp(-2pijk/N') \\
c_{k+N/2} &= \sum_{j=0}^{N/2-1} e_j \exp(-2pijk/N') - \exp(-2pik/N) \sum_{j=0}^{N/2-1} o_j \exp(-2pijk/N')
\end{aligned}
\qquad (*)
$$

  - 従って，*N* 点のDFTは2つの *N*/2 点のDFTと，その結果に複素指数関数 $\exp(-2\pi ik)$ を掛けて足し合わせる処理に分解できる。
  - この分解を再帰的に行ってDFTを計算する方法を，高速フーリエ変換（FFT; Fast Fourier Transform）と呼ぶ。

1.2　FFTの原理 (2)

- FFTの計算量
  - FFTを用いてN点のDFTを求めるときの計算量を$T(N)$とすると，複素指数関数 $exp(-2\pi ik)$ を掛けて足し合わせる処理の計算量は実数の乗算が$2N$回，加算が$3N$回の合計$5N$回だから，

$$T(N) = 2T(N/2) + 5N$$

  - $T(1) = 0$ に注意してこれを解くと（前回の講義参照），

$$T(N) = 5N \log_2 N$$

  - 従って，FFTを使うと $5N \log_2 N$ の計算量でN点のDFTが計算できる。

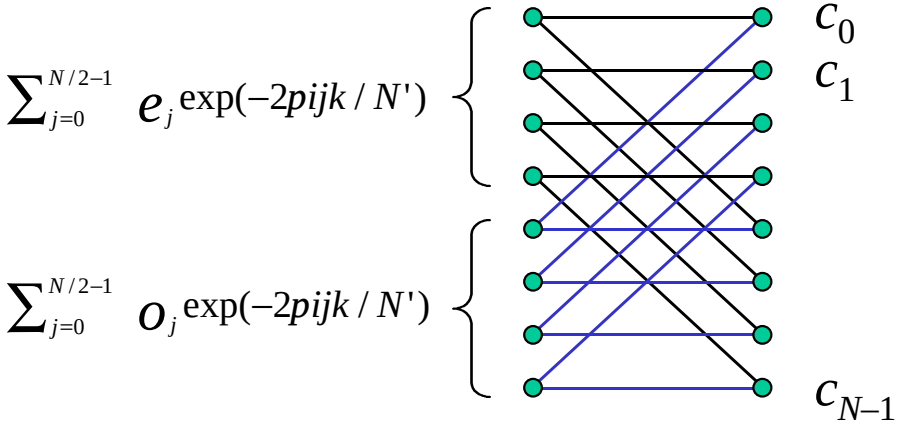
1.3 Cooley-Tukey FFT (1)

- 1次元配列への格納方式
  - DFTの分解の式 (*) において，第1項，第2項に相当する$N/2$点のDFTをそれぞれ1次元配列の前半，後半に格納する方式を考える。

$$c_k = \sum_{j=0}^{N/2-1} e_j \exp(-2\pi ijk/N') + \exp(-2\pi ik/N) \sum_{j=0}^{N/2-1} o_j \exp(-2\pi ijk/N')$$

$$c_{k+N/2} = \sum_{j=0}^{N/2-1} e_j \exp(-2\pi ijk/N') - \exp(-2\pi ik/N) \sum_{j=0}^{N/2-1} o_j \exp(-2\pi ijk/N')$$

(*)

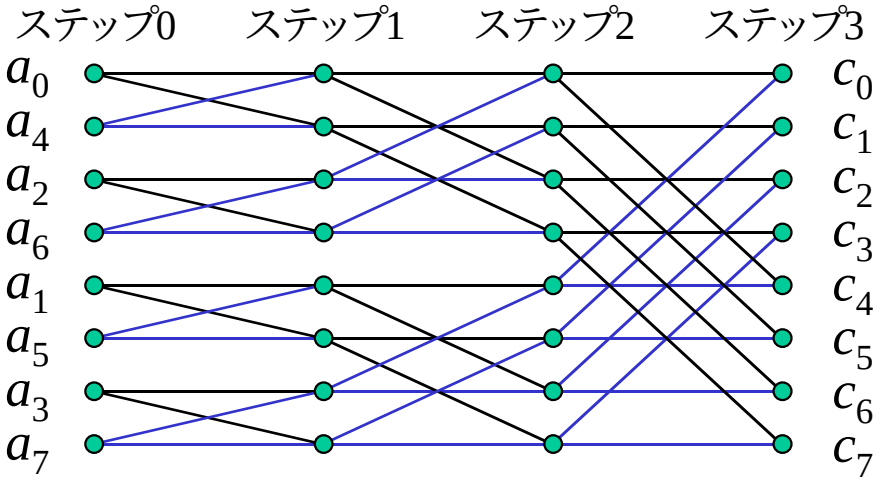


そのまま加える。

$\exp(-2\pi ik/N)$ を掛けて加える。

1.3 Cooley-Tukey FFT (2)

- Cooley-Tukey FFT
  - 同じ格納方式を，N/2点のDFTのそれぞれに対しても再帰的に適用していくことにより，各ステップでの中間結果の格納場所が定まる。
  - 計算式として ⊛ を用い，中間結果をこのように1次元配列に格納して計算を進める方法を Cooley-Tukey FFT と呼ぶ（Cooley & Tukey, 1965）。

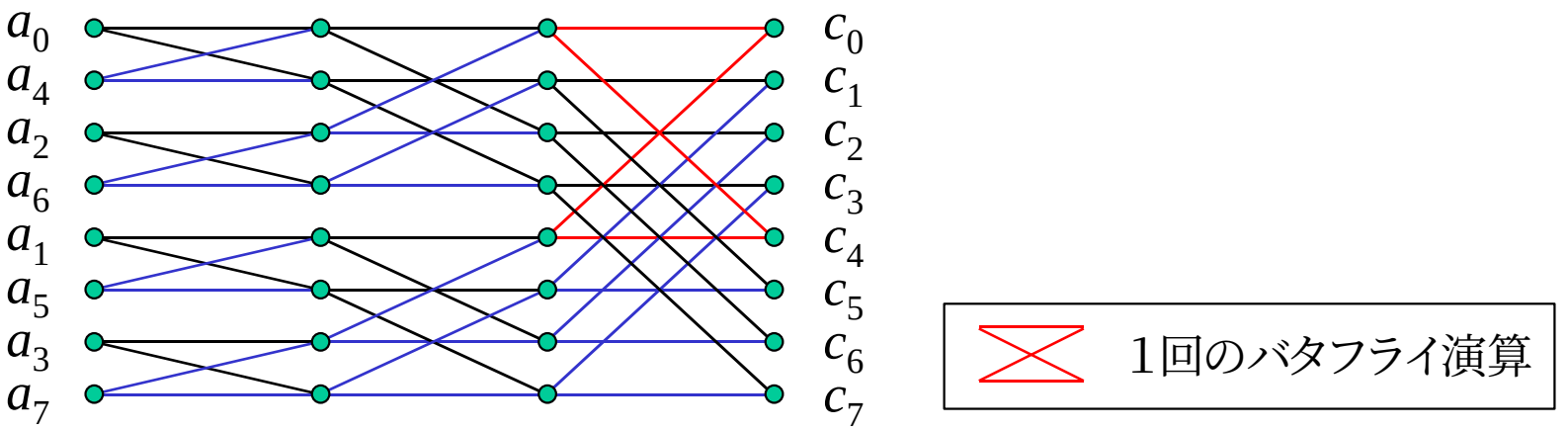


N=8の場合のCooley-Tukey FFT

  - また，1次元配列への格納形式と計算過程を表現するこのようなグラフをシグナル・フロー・グラフと呼ぶ。
  - また，各ステップでは2個の要素から2個の要素を計算する処理を繰り返し行う。この処理をバタフライ演算と呼ぶ。

1.3 Cooley-Tukey FFT (3)

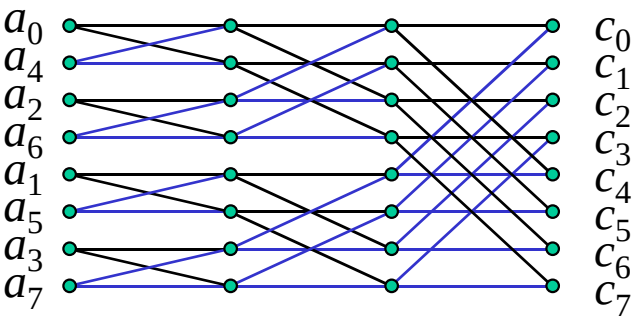
- Cooley-Tukey FFTの長所
  - Cooley-Tukey FFTでは，各バタフライ演算において，入力と出力とが1次元配列の同じ位置に格納される。
  - このことを利用すると，ステップL+1 の中間結果をステップL の中間結果に上書きでき，配列は1個で済む。
  - この特徴を持つFFTを，in-place FFT と呼ぶ。



1.3 Cooley-Tukey FFT (4)

- Cooley-Tukey FFTの短所
  - 入力$\{a_j\}$が1次元配列中で自然な順序に並ばない。
  - 入力a_jのインデックスjを2進数$j_{p-1}\ldots j_1 j_0$ $(p = \log_2 N)$，a_jの1次元配列中での位置を$i_{p-1}\ldots i_1 i_0$で表すと，格納方式の定義より，

$j_0 = 0$ なら $i_{p-1} = 0$，$j_0 = 1$ なら $i_{p-1} = 1$
$j_1 = 0$ なら $i_{p-2} = 0$，$j_1 = 1$ なら $i_{p-2} = 1$
⋮
$j_{p-1} = 0$ なら $i_0 = 0$，$j_{p-1} = 1$ なら $i_0 = 1$



  - したがって，$i_{p-1} = j_0$，$i_{p-2} = j_1$，…，$i_0 = j_{p-1}$。
  - すなわち，入力$\{a_j\}$はビット逆順に並ぶ。
  - 元々の入力が自然な順序に並んでいる場合，並べ替えが必要。

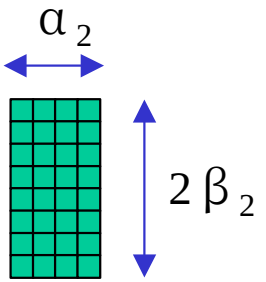
- 疑問
  - 入力も出力も自然な順に並ぶFFTの計算方式はないか？

1.4 Stockham FFT (1)

- 目標
  - 入力・出力とも自然な順序で並ぶ（self-sorting）FFTを構成する。

- 配列 $X_L(j, k)$ の定義
  - $\alpha_L = 2^L$, $\beta_L = 2^{p-L-1}$ とすると，X_L は大きさ $2\beta_L \times \alpha_L$ の2次元配列
  - $X_L(j, *)$ は入力データを $2\beta_L$ 個おきに抜き出した α_L 個の部分列 a_j, $a_{j+2\beta L}$, …, $a_{j+2(\alpha_L-1)\beta_L}$ のDFT

- 配列 $X_L(j, k)$ の性質
  - $L = 0$ のとき $X_0(j, 0) = a_j$, $L = p$ のとき $X_p(0, k) = c_k$
  - すなわち，$X_0(j, 0)$, $X_p(0, k)$ はそれぞれ自然な順序で並べられた入力データ，出力データと見なせる。



$N = 32$, $L = 2$ のときの $X_2(j, k)$

X_0 から始めて X_1, X_2, …, X_p を順に計算していくことができれば，self-sortingなFFTが構成できる。

1.4 Stockham FFT (2)

- 配列 $X_L(j, k)$の性質（続き）
  - DFTの分解の式 (*) をX_L, X_{L+1}を使って書き直すことにより, $X_L(j, k)$ は次の漸化式を満たすことが示せる（確認せよ）。

$$
\begin{aligned}
X_{L+1}(j, k) &= X_L(j, k) + X_L(j+\beta_L, k)\cdot \omega_N^{k\beta_L} \\
X_{L+1}(j, k+\alpha_L) &= X_L(j, k) - X_L(j+\beta_L, k)\cdot \omega_N^{k\beta_L} \\
\text{ただし } j &= 0, 1, \ldots, \beta_L - 1, \quad k = 0, 1, \ldots, \alpha_L - 1
\end{aligned}
$$

- Stockham FFT
  - この漸化式を用いて, $X_0 \to X_1 \to \ldots \to X_{p-1} \to X_p$ を順に計算していく方式を, Stockham FFT と呼ぶ。

- Stockham FFTの特徴
  - Self-sorting である。並べ替えが不要のため, 高性能計算に向く。
  - In-place でない。計算にはサイズ N の配列が2個必要。

1.4 Stockham FFT (3)

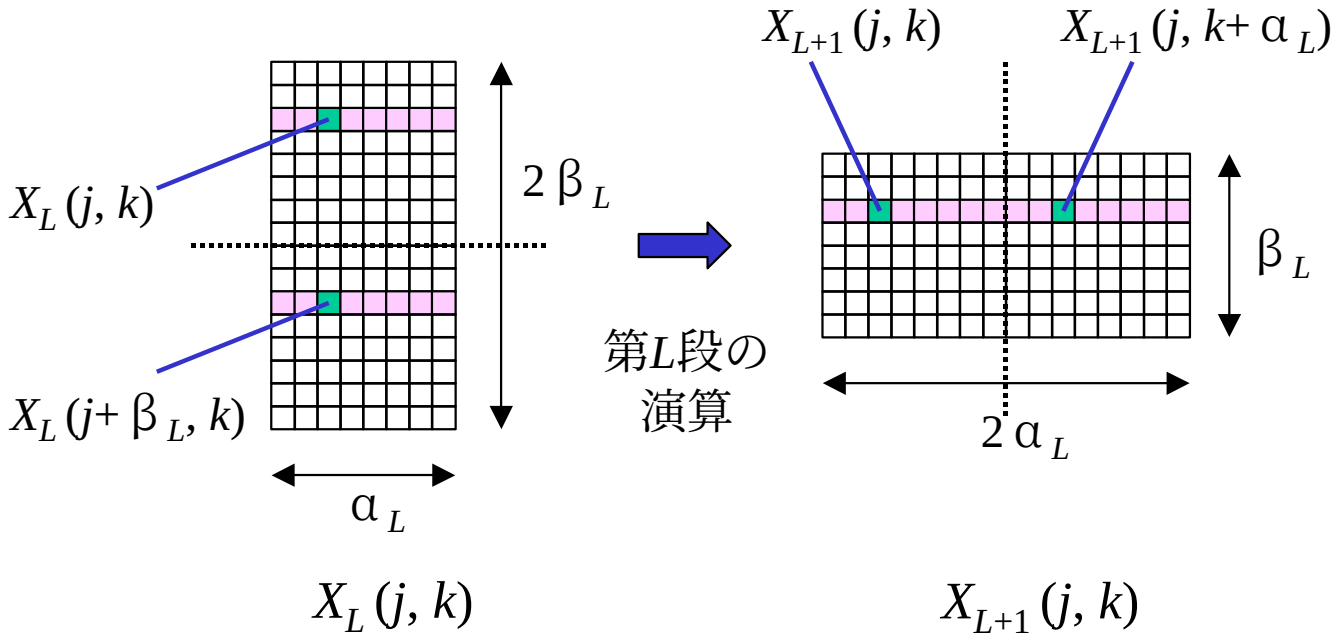
- Stockham FFT のプログラム

```
      DO 10 L = 0, p–1
        α = 2^L
        β = 2^(p–L–1)
        DO 20 k = 0, α–1
          DO 30 j = 0, β–1
            X_{L+1}(j, k)     = X_L(j, k) + X_L(j+β, k)・ω_N^{kβ}
            X_{L+1}(j, k+α) = X_L(j, k) – X_L(j+β, k)・ω_N^{kβ}
30        CONTINUE
20      CONTINUE
10    CONTINUE
```

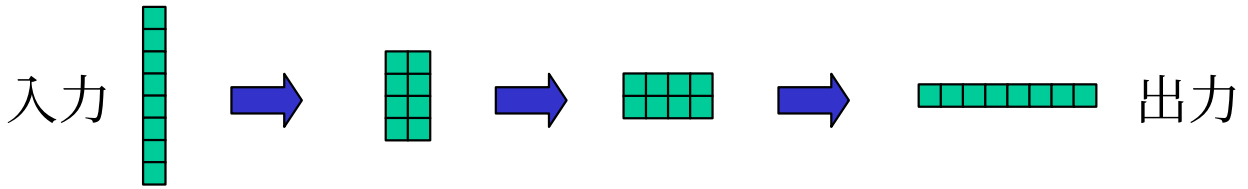
- 計算の並列性
  - DO 20, DO 30のループについて完全並列
  - この2つのループの入れ替えも可能
  - 共有メモリ型計算機での並列化は容易

1.4 Stockham FFT (4)

- 第L+1ステップの計算の図解 (N=128, L = 3)



- 全計算ステップの図解 (N = 8)



1.5　逆FFTと周波数間引き型FFT (1)

- 逆FFTの計算方法 (I)
  - 1.1(1)で述べたように，DFT，逆DFTはそれぞれ次の式で計算できる。

DFT　$$c_k = \sum_{j=0}^{N-1} a_j \exp(-2pijk / N)$$

逆DFT　$$a_j = (1/N) \sum_{k=0}^{N-1} c_k \exp(2pijk / N)$$

  - 従って逆FFTの計算は，FFTの計算式において，$\omega_N = \exp(-2\pi i/N)$ を $\omega_N{}^*$（共役複素数）で置き換え，計算結果をNで割ればよい。

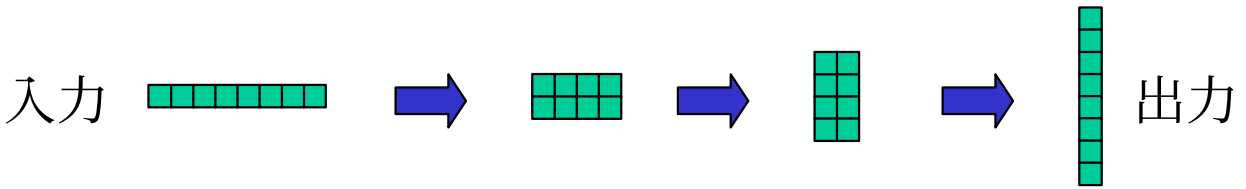
- 簡便な計算方法
  - 次のようにして計算すれば，FFTのプログラムを逆FFTに転用できる。
    (1) 入力データ c_k を複素共役 $c_k{}^*$ にする。
    (2) $c_k{}^*$ のFFTを計算する。
    (3) 結果を複素共役にする。
    (4) $1/N$ をかける。

1.5　逆FFTと周波数間引き型FFT (2)

- 逆FFTの計算方法 (II)
  - Stockham FFT において，X_L から X_L+1 を求める式を逆に解き，L に関するループを逆回しにすることによっても，逆FFTは計算できる。

- この方法による逆FFTのプログラム

```
      DO 10 L = p–1,  0, –1
        α = 2^L
        β = 2^(p–L–1)
        DO 20 k = 0,  α –1
          DO 30 j = 0,  β –1
            X_L (j, k)      = (X_L+1 (j, k) + X_L+1 (j, k+ α )) / 2
            X_L (j+ β , k) = (X_L+1 (j, k) – X_L+1 (j, k+ α ))・ ω_N^(–kβ) / 2
30        CONTINUE
20      CONTINUE
10    CONTINUE
```



1.5　逆FFTと周波数間引き型FFT (3)

- 周波数間引き型FFT
  - 逆DFTの定義式において，ω_Nを$\omega_N{}^*$で置き換え，計算結果に N を掛けると，DFTの定義式となる。
  - 従って，逆FFTの計算方法 (II) において，ω_Nを$\omega_N{}^*$で置き換え，計算結果に N を掛けると，再び順方向のFFTを計算するアルゴリズムが得られる。これを周波数間引き型FFTと呼ぶ。
  - これに対して，　㊟ 式に基づくFFTを時間間引き型FFTと呼ぶ。

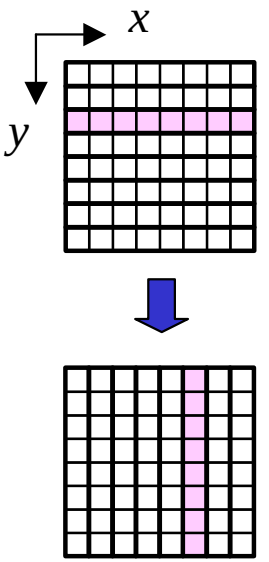
- 両FFTの使い分け
  - 時間間引き型FFTと周波数間引き型FFTは，最内側の式の形が異なるため，プロセッサによっては一方が他方より性能が出やすいことがある。
  - 対象とするプロセッサによって，性能の出やすいほうを選べばよい。

1.6　多次元FFT

- 2次元DFTの計算方法
  - $N_x \times N_y$ 個の複素数データ $\{a_{jx,\,jy}\}$ に対し，次の式で定義される $\{c_{kx,\,ky}\}$ をその2次元DFTと呼ぶ。

$$c_{kx,ky} = \sum_{jy=0}^{Ny-1} \sum_{jx=0}^{Nx-1} a_{jx,jy} \exp(-2pi\, j_x k_x / N_x) \exp(-2pi\, j_y k_y / N_y)$$

  - これは，次のように2ステップに分けて計算できる。



ステップ1

$$c'_{kx,jy} = \sum_{jx=0}^{Nx-1} a_{jx,jy} \exp(-2pi\, j_x k_x / N_x)$$

ステップ2

$$c_{kx,ky} = \sum_{jy=0}^{Ny-1} c'_{kx,jy} \exp(-2pi\, j_y k_y / N_y)$$

  - ステップ1は N_y 組のデータに対する N_x 点のFFT，ステップ2は N_x 組のデータに対する N_y 点のFFTとして計算できる。
  - このとき，計算量は $5\, N_x N_y \log_2 (N_x N_y)$ となる。

- 多次元FFT
  - 3次元以上のデータに対しても，各方向に対する1次元FFTの繰り返しにより，多次元FFTが計算できる。

1.7　FFTの分解 (1)

- 1次元DFTの２次元への分解
  - 1次元DFTにおいて，$N = lm$ と分解できるとする。このとき，インデックス j，k を

$$\boxed{\begin{array}{ll} j = rl + s & (s = 0,\ \ldots,\ l-1,\ \ r = 0,\ \ldots,\ m-1) \\ k = pm + q & (q = 0,\ \ldots,\ m-1,\ \ p = 0,\ \ldots,\ l-1) \end{array}}$$

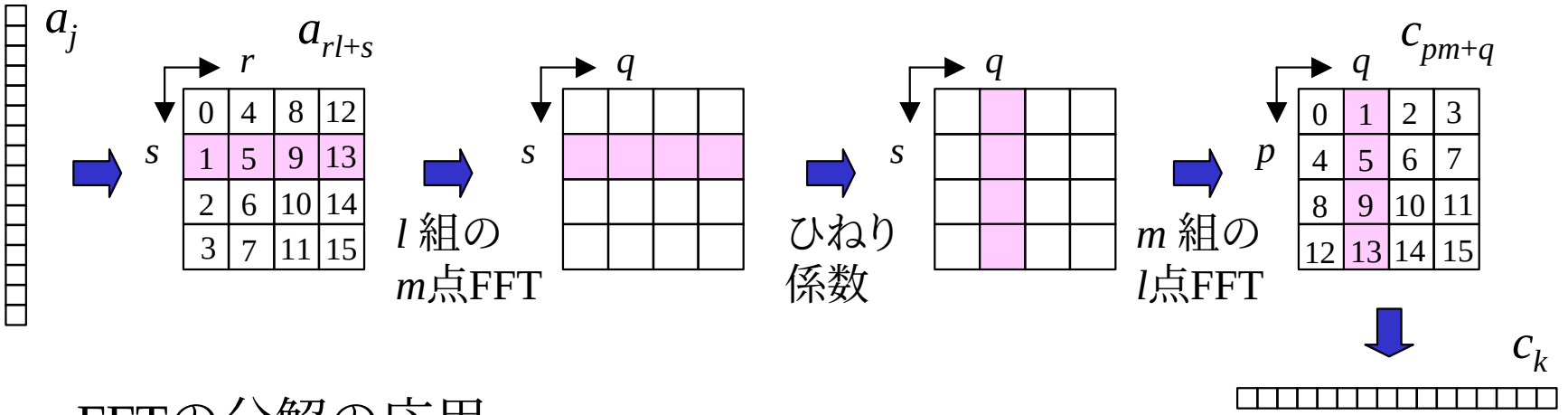
  - と書き直すと，N点DFTの計算式は次のように変形できる。

$$\begin{aligned} c_k = c_{pm+q} &= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} a_{rl+s}\exp(-2\pi i(pm+q)(rl+s)/N) \\ &= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} a_{rl+s}\exp(-2\pi ips/l)\exp(-2\pi iqr/m)\exp(-2\pi iqs/N) \\ &= \sum_{s=0}^{l-1}[[\sum_{r=0}^{m-1} a_{rl+s}\exp(-2\pi iqr/m)]\exp(-2\pi iqs/N)]\exp(-2\pi ips/l) \end{aligned}$$

  - ここで，内側の Σ は l 組のデータに対する m 点DFT，外側の Σ は m 組のデータに対する l 点DFTの形をしている。

1.7　FFTの分解 (2)

- 1次元DFTの2次元への分解（続き）
  - 従って，$N = lm$ 点のDFTの計算は次のように分解できる。
    (1) l 組のデータに対する m 点FFT
    (2) 第 s 組の第 q 要素にひねり係数 $\exp(-2\pi iqs / N)$ を掛ける。
    (3) m 組のデータに対する l 点FFT



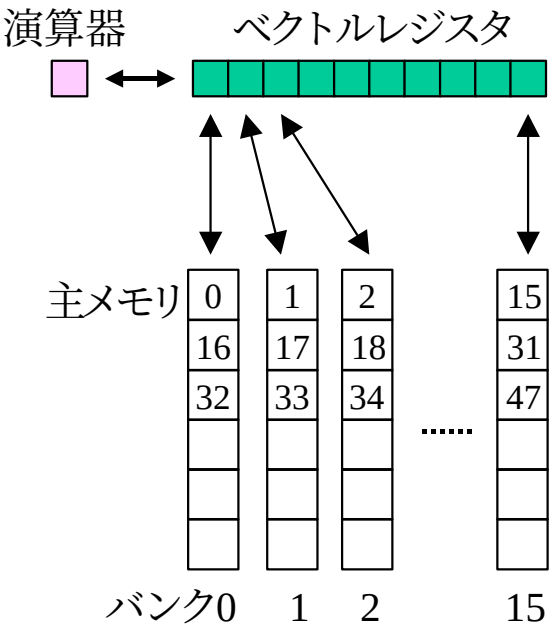
- FFTの分解の応用
  - キャッシュ再利用性の向上，分散メモリ向け並列化を行う際に有効
  - 分解を再帰的に適用することにより，3次元以上への分解も可能

2．単体プロセッサ向けの高性能化技法

- ベクトルプロセッサとその高性能化技法
- キャッシュマシンとその高性能化技法
- 最内側ループ長の増加
- バンクコンフリクトの削減
- レジスタ再利用性の向上
- キャッシュ再利用性の向上

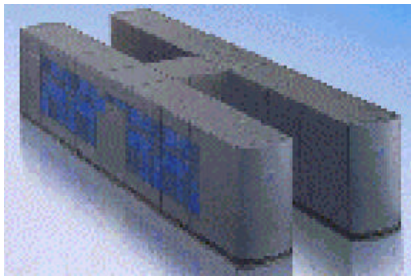
2.1　ベクトルプロセッサとその高性能化技法

- ベクトルプロセッサの特徴
  - ベクトルレジスタ／演算器による長いベクトルの高速演算
  - バンク構成による高い主メモリのスループット



- (擬似) ベクトルプロセッサの例
  - NEC SX-7 (地球シミュレータ)，富士通 VPP5000
  - 日立 SR2201，SR8000，Intel Pentium 4

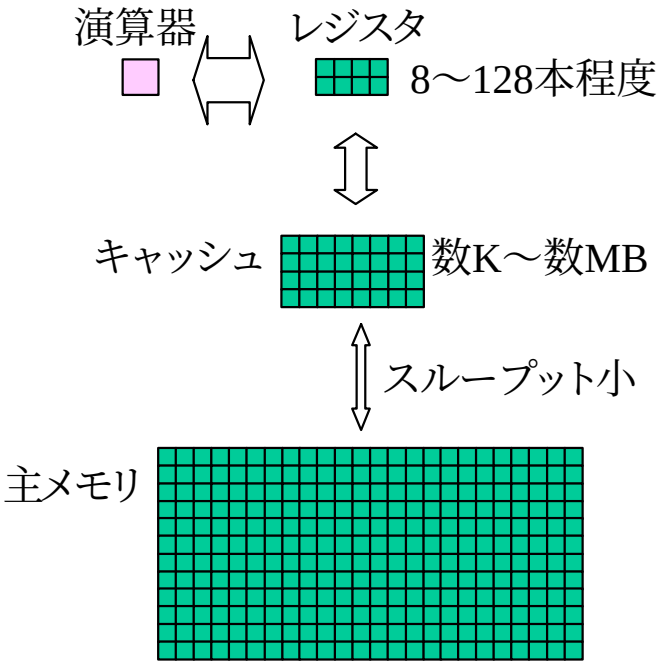
- 高性能化技法
  - 最内側ループ長の増加
    - ループ交換，ループ融合
  - バンクコンフリクトの削減
    - 2の大きなべき乗でのストライドアクセスの回避



日立　SR8000

2.2　キャッシュマシンとその高性能化技法

- キャッシュマシンの特徴
  - レジスタ – キャッシュ – 主メモリという階層的な記憶装置
  - レジスタ内のデータは高速に演算可能
  - 主メモリ – キャッシュのスループット小



- キャッシュマシンの例
  - Intel Pentium III，IBM Power 4
  - AMD Athlon，DEC Alpha，など

- 高性能化技法
  - レジスタ再利用性の向上
    - ループ展開
  - キャッシュ再利用性の向上
    - ブロッキング

2.3　最内側ループ長の増加

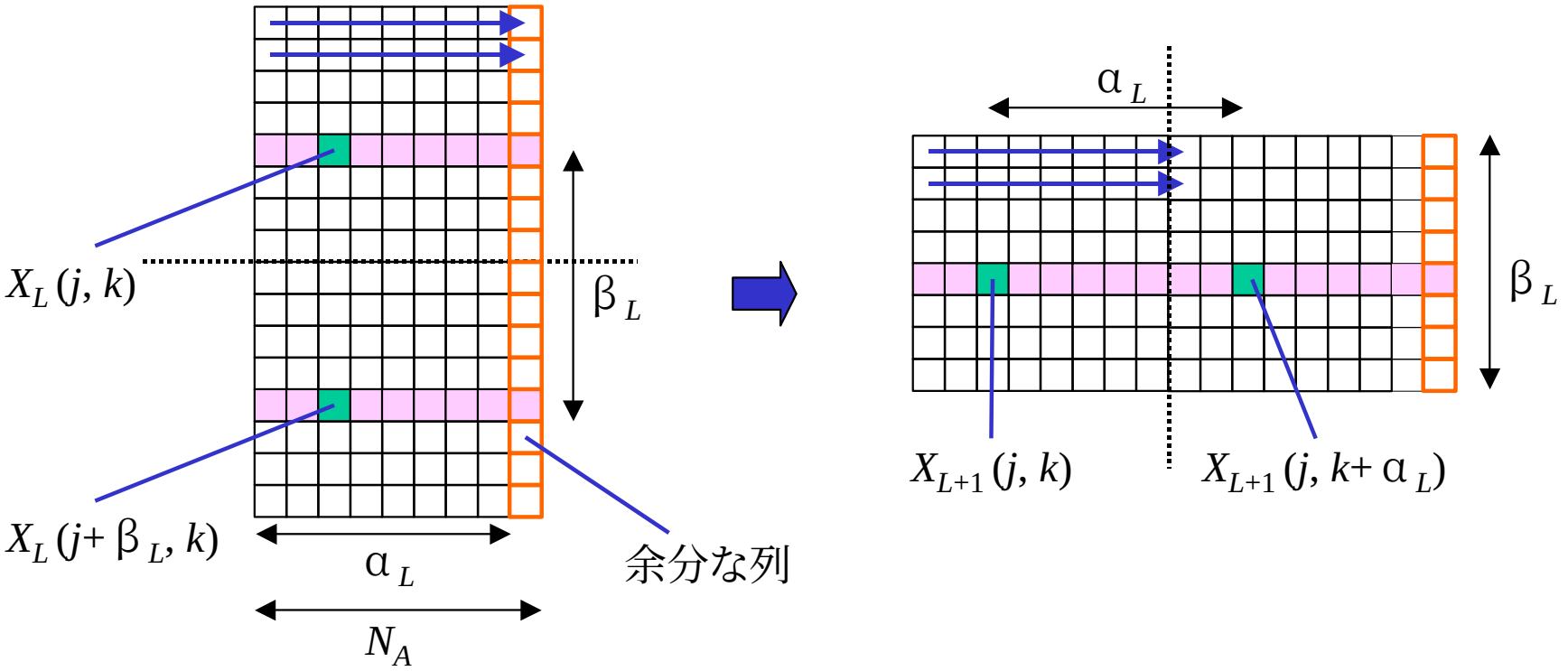
- Stockham FFT の特徴
  - 最内側のループ長が $N/2$, $N/4$, …, 1と変化

```
      DO 10 L = 0, p–1
         α = 2^L
         β = 2^(p–L–1)
         DO 20 k = 0, α –1
            DO 30 j = 0, β –1
               X_{L+1} (j, k)     = X_L (j, k) + X_L (j+ β , k)・ ω_N^{kβ}
               X_{L+1} (j, k+ α ) = X_L (j, k) – X_L (j+ β , k)・ ω_N^{kβ}
30          CONTINUE
20       CONTINUE
10    CONTINUE
```

- 最内側ループ長の増加
  - α > β となった時点で DO 20 と DO 30 のループを交換することにより, 最内側ループ長は常に $O(N^{1/2})$以上に保てる。

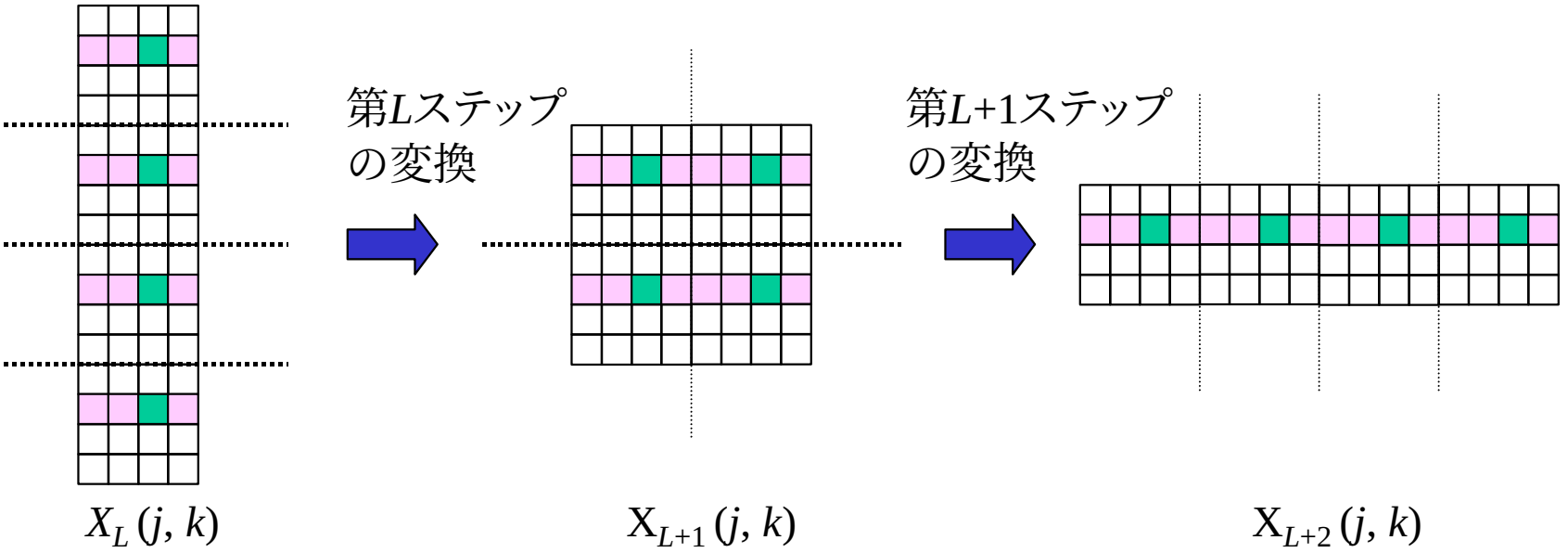
2.4　バンクコンフリクトの削減

- 整合寸法の調整によるバンクコンフリクト削減
  - k方向が連続アドレスの場合，Stockham FFTでの $X_L(j, k)$ と $X_L(j + \beta_L, k)$ のアドレス差は $\alpha_L * \beta_L = N/2$
  - 2次元配列 X_L の整合寸法 N_A を調節することにより，これを $N_A * \beta_L$ としてバンクコンフリクトを削減できる。



2.5　レジスタ再利用性の向上 (1)

- 4基底FFT
  - Stockham FFTにおいて，2ステップ分の変換を一度に行うことで，ロード／ストアが削減でき，レジスタ再利用性が向上する。
  - これは，4個の要素に対して演算を行うので，4基底FFTと呼ぶ。
  - 今まで説明してきたFFTは，2基底FFTと呼ぶ。



  - 同様の方式により，8基底FFTも構成できる。

2.5　レジスタ再利用性の向上 (2)

- 4基底FFTの計算式
  - $\alpha_L = 2^L$,　　$\beta_L = 2^{p-L-2}$　とおくと，カーネルは次のようになる。
  - この段階ではロード／ストア削減のみ。演算量は2基底と同じ。

第Lステップの変換

$$
\begin{aligned}
X_{L+1}(j, k) &= X_L(j, k) + X_L(j+2\beta_L, k)\cdot \omega^{2k\beta_L} \\
X_{L+1}(j, k+\alpha_L) &= X_L(j, k) - X_L(j+2\beta_L, k)\cdot \omega^{2k\beta_L} \\
X_{L+1}(j+\beta_L, k) &= X_L(j+\beta_L, k) + X_L(j+3\beta_L, k)\cdot \omega^{2k\beta_L} \\
X_{L+1}(j+\beta_L, k+\alpha_L) &= X_L(j+\beta_L, k) - X_L(j+3\beta_L, k)\cdot \omega^{2k\beta_L}
\end{aligned}
$$

第L+1ステップの変換

$$
\begin{aligned}
X_{L+2}(j, k) &= X_{L+1}(j, k) + X_{L+1}(j+\beta_L, k)\cdot \omega^{k\beta_L} \\
X_{L+2}(j, k+2\alpha_L) &= X_{L+1}(j, k) - X_{L+1}(j+\beta_L, k)\cdot \omega^{k\beta_L} \\
X_{L+2}(j, k+\alpha_L) &= X_{L+1}(j, k+\alpha_L) + X_{L+1}(j+\beta_L, k+\alpha_L)\cdot \omega^{(k+\alpha_L)\beta_L} \\
X_{L+2}(j, k+3\alpha_L) &= X_{L+1}(j, k+\alpha_L) - X_{L+1}(j+\beta_L, k+\alpha_L)\cdot \omega^{(k+\alpha_L)\beta_L}
\end{aligned}
$$

2.5　レジスタ再利用性の向上 (3)

- 4基底FFTにおける演算量削減
  - 第 L+1ステップでの三角関数乗算を，すべて第Lステップに持ってくる。
  - これにより，ωの乗算回数を削減可能

第Lステップの変換

$$
\begin{aligned}
X_{L+1}(j, k) &= X_L(j, k) + X_L(j+2\beta_L, k)\cdot \omega^{2k\beta_L} \\
X_{L+1}(j, k+\alpha_L) &= X_L(j, k) - X_L(j+2\beta_L, k)\cdot \omega^{2k\beta_L} \\
Y_{L+1}(j+\beta_L, k) &= X_L(j+\beta_L, k)\cdot \omega^{k\beta_L} + X_L(j+3\beta_L, k)\cdot \omega^{3k\beta_L} \\
Y_{L+1}(j+\beta_L, k+\alpha_L) &= X_L(j+\beta_L, k)\cdot \omega^{(k+\alpha_L)\beta_L} - X_L(j+3\beta_L, k)\cdot \omega^{(3k+\alpha_L)\beta_L} \\
&= -iX_L(j+\beta_L, k)\cdot \omega^{k\beta_L} + iX_L(j+3\beta_L, k)\cdot \omega^{3k\beta_L}
\end{aligned}
$$

$\omega^{\alpha_L\beta_L} = \exp(-\pi i/2) = -i$ を用いて変形

第L+1ステップの変換

$$
\begin{aligned}
X_{L+2}(j, k) &= X_{L+1}(j, k) + Y_{L+1}(j+\beta_L, k) \\
X_{L+2}(j, k+2\alpha_L) &= X_{L+1}(j, k) - Y_{L+1}(j+\beta_L, k) \\
X_{L+2}(j, k+\alpha_L) &= X_{L+1}(j, k+\alpha_L) + Y_{L+1}(j+\beta_L, k+\alpha_L) \\
X_{L+2}(j, k+3\alpha_L) &= X_{L+1}(j, k+\alpha_L) - Y_{L+1}(j+\beta_L, k+\alpha_L)
\end{aligned}
$$

2.5　レジスタ再利用性の向上 (4)

- ４基底FFT，８基底FFTの効果
  - ２基底に比べ，レジスタへのロード／ストア回数，演算量ともに減少

	実数加算／乗算	ロード／ストア	Byte/Flop値
２基底	6 / 4	4 / 4	6.4
４基底	22 / 12 (２基底では 24 / 16)	8 / 8	3.76
８基底	66 / 32 (２基底では 72 / 48)	16 / 16	2.61

Byte/Flop値：　演算１回当たりに何回のロード／ストアが必要かを示す指標

2.6　キャッシュ再利用性の向上 (1)

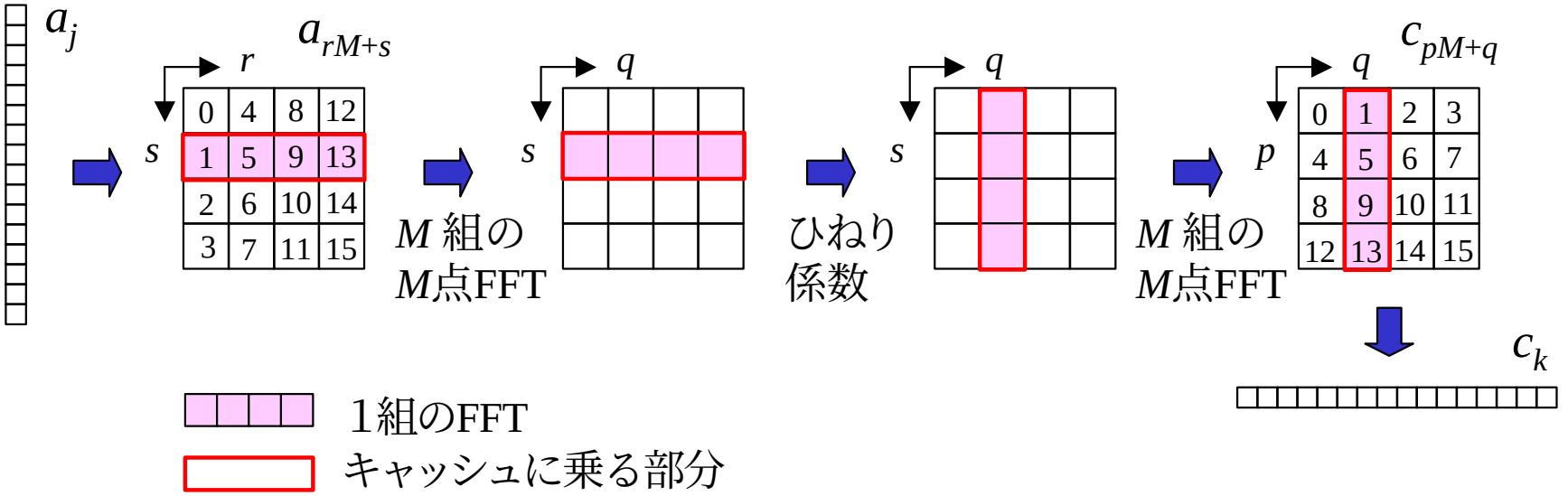
- Stockham FFTのメモリアクセスパターン
  - 各ステップで，配列 $X_L(j, k)$ の全要素（N 個）をアクセス
  - キャッシュサイズを M とするとき，$M < N$ ならば，毎ステップで主メモリに対し，$O(N)$ 回のアクセスが発生
  - FFT全体では $O(N \log_2 N)$回

- FFTの分解の利用
  - いま，$M = N^{1/2}$ と仮定する。
  - このとき，1.7 (2)で示したように，N点のFFTは次のように分解できる。
    (1)　$M = N^{1/2}$ 組のデータに対する M 点FFT
    (2)　第 s 組の第 q 要素にひねり係数 $\exp(-2\pi iqs / N)$ を掛ける。
    (3)　M 組のデータに対する M 点FFT

- 効果
  - 処理 (1)，(3) において，各組のFFTはキャッシュ上で行える。
  - 主メモリのアクセス回数は，(1)，(2)，(3) でそれぞれ$O(N)$ 回
  - FFT全体では $O(N)$回

2.6　キャッシュ再利用性の向上 (2)

- N =16, M = 4の場合の図解



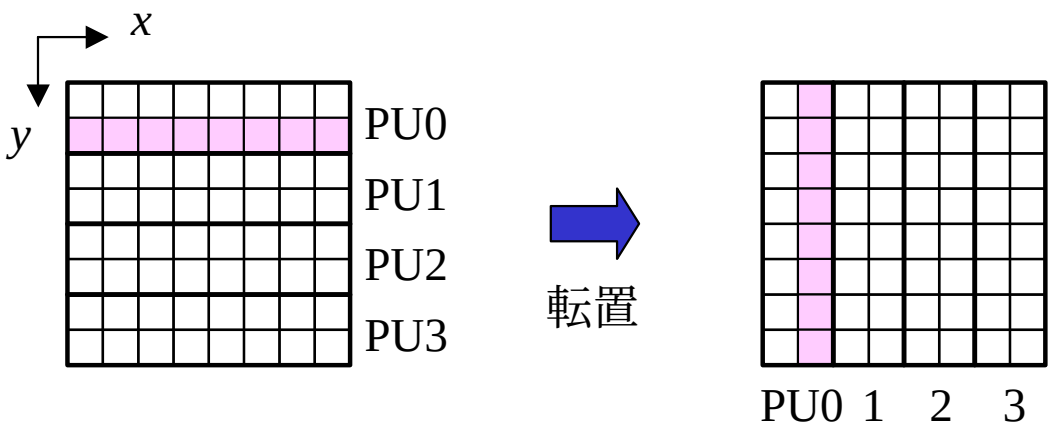
- 一般の場合
  - $N \sim M^r$ ならば，分解を r–1回再帰的に行い，1つのFFTのサイズを M 程度にすればよい。

3．分散メモリ向けの並列化技法

- 2次元FFTの並列化
- 1次元FFTの並列化

3.1　2次元FFTの並列化 (1)

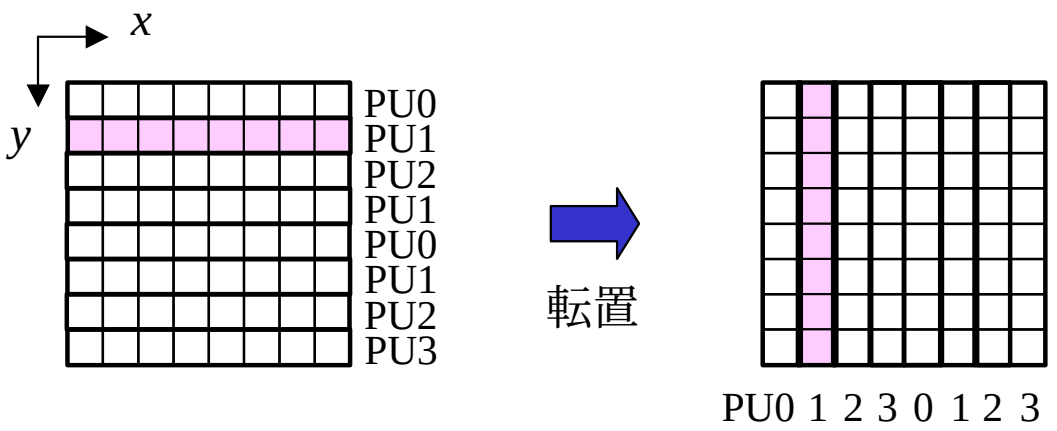
- ブロック分割による並列化
  - *y*方向にデータをブロック分割し，*x*方向のFFTを実行
  - その後，all-to-all broadcast によりデータを再分散 (転置)
  - *x*方向にデータをブロック分割し，*y*方向のFFTを実行



- 計算量とデータ通信量
  - プロセッサ数が p のとき，
    - プロセッサあたりの計算量は $5\,N_xN_y\log_2(N_xN_y)\,/\,p$
    - プロセッサあたりのデータ通信量は $N_xN_y\,/\,p$，通信回数は p–1回

3.1　2次元FFTの並列化 (2)

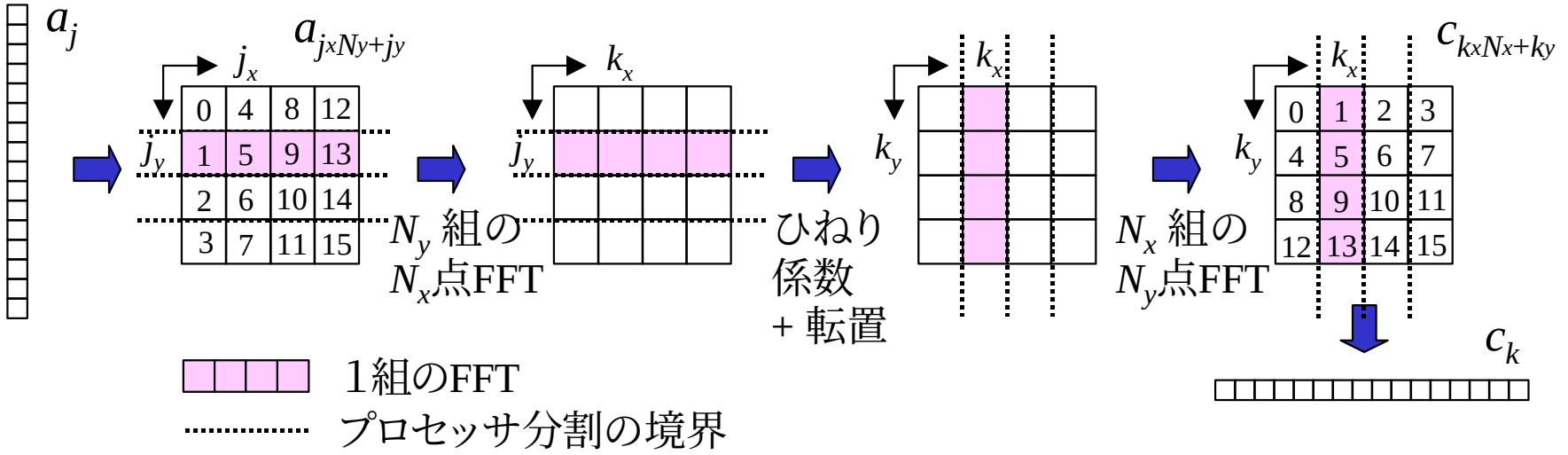
- サイクリック分割による並列化
  - *x*方向にFFTを行う処理は，*y*方向については完全独立
  - *y*方向のデータ分割は，どんな形式でもよい。



- 計算量とデータ通信量
  - ブロック分割の場合と同じ。

3.2　1次元FFTの並列化 (1)

- FFTの分解の利用
  - $N = N_x N_y$ と分解し，N点のFFTを次のように分解
    - (1)　N_y 組のデータに対する N_x 点FFT（y方向にデータ分割）
    - (2)　第 j_y 組の第 k_x 要素にひねり係数 $\exp(-2\pi i k_x j_y / N)$ を掛ける。
    - (3)　all-to-all broadcast によりデータを再分散（転置）
    - (4)　N_x 組のデータに対する N_y 点FFT（x方向にデータ分割）



  - この並列FFTアルゴリズムを，転置アルゴリズムと呼ぶ。

3.2　1次元FFTの並列化 (2)

- 計算量とデータ通信量
  - プロセッサ数が p のとき,
    - プロセッサあたりの計算量は $5\,N \log_2 N / p$
    - プロセッサあたりのデータ通信量は N / p，通信回数は p–1回

- 通信のオーバーヘッド
  - SR8000の通信／演算の性能比
    - 1ノードの演算性能は8GFLOPS
    - 通信速度は，複素数で0.0625語/s
  - $N = 2^{30}$のとき，通信と演算に掛かる時間はほぼ同程度

通信オーバーヘッドを削減する方法はないか？

次回の予定

3. 分散メモリ向けの並列化技法 (続き)
   - 通信オーバーヘッドの削減
   - ベクトル並列機向けのFFT

4. FFTの拡張
   - ２のべき乗でない N に対するFFT
   - 対称性を持つデータのFFT

5. FFTの応用
   - 偏微分方程式の求解
   - 信号処理
   - 多倍長計算