平成 13 年度

学士学位論文

変形 Batch SOM とその DDMP による実装

Variant Batch SOM and it's implementation on DDMP

1020306　　中城 敏幸

指導教員　　Ruck THAWONMAS

2002 年 2 月 8 日

高知工科大学 情報システム工学科

要 旨

変形 Batch SOM とその DDMP による実装

中城 敏幸

自己組織化マップ (Self-Organizing Map，SOM) は，多次元のデータを二次元上に配置して視覚化する手法である．その手法として一般的なものは，入力層，競合層の値を比較し，勝利ノードおよびその近傍を更新する Online SOM と呼ばれるものであるが，他にもその学習方法によって様々なアルゴリズムが存在する．本研究ではその中から，Data Driven Media Processor(DDMP) での実装を目指し，より DDMP に適した方法として，Batch SOM を基にした変形 Batch SOM を提案, 実装し，Online SOM との結果と比較，検証する．

キーワード　自己組織化マップ，SOM，DDMP，Online SOM，Batch SOM，変形 Batch SOM，勝利回数近似

Abstract

Variant Batch SOM and it's implementation on DDMP

Toshiyuki Nakajo

A self-organization map (SOM) is the technique which arranges the data of many dimensions to secondary dimensions, it is one of the means to visualize. The basic SOM algorithm is called Online SOM, an input layer and a competitive layer are used ,each value is compared, a victory unit and the near unit are determined, each value is updated. SOM algorithm various otherwise exists. By this research the SOM algorithm suitable for DDMP is considered.It is Variant Batch SOM is proposed this time. It improves and carries out Batch SOM. and the validity is investigated as compared with OnlineSOM.

key words Self-Organization Map, SOM, DDMP, Online SOM, Batch SOM, Variant Batch SOM, Number-of-times approximation of a victory

目次

図目次

表目次

第 1 章

はじめに

Kohonen により提案された自己組織化マップ (Self-Organaizing Map,SOM)[1] は，クラスタリング等のさまざまな分野での応用が提案されている．そのアルゴリズムは，最も基本的な Online SOM をはじめ，様々な手法が存在している．その中でも今回は Data Driven Media Processor(DDMP)[2] での実装を目的とし，最も適した方法を考察する．今回はその手法の一つとして，Batch SOM アルゴリズム [3] を採用した．その理由は，学習時の更新回数が Online SOM に比べ少なく，プロセッシングエレメント (PE) にかかる負担を軽減できるからである．しかし，Batch SOM はその更新式に除算を使用しており，DDMP での実装は困難である．そこで，本研究では Batch SOM の特徴を持ち，なおかつ DDMP での実装を可能にする変形 Batch SOM を提案する．変形 Batch SOM はその除算数を DDMP が持つ「ビットシフト」機能を用いて，擬似的な除算が可能な値に近似して除算を行うというものである．この方法を用いることにより，学習の収束という点ではオリジナルの Batch SOM に劣るものの，Online SOM に比べれば，処理時間の短縮・学習の収束共に高性能であることが期待できる．

第 2 章

SOM

2.1 SOM とは

SOM は，情報の可視化の手法の一つであり,「教師なし学習」を行うニューラルネットワークの 1 つで，与えられたデータから特徴を自動的に抽出し，分類を行うためのマップを自己組織的に作成できる技術である．このマップは二次元上の距離情報の他，色の濃淡などの表現を用いることにより，多次元の特徴を持つデータを二次元のマップとして配置・表現することができ，それぞれの関係をわかり易くすることができる．この特徴を利用し，画像・音声や指紋等の認識や工業製品の生産プロセスの制御に利用されている．また，新しいデータをすでに作成されたマップに投入・比較し，そのデータがマップの中でどの位置になるかを判断することで，そのデータの特徴を瞬時に知ることもできる．

一般的に SOM という名称は一連の工程によって生成されたマップそのものを指す言葉であるが，ここでは SOM という言葉を「自己組織化マップを生成するもの」,「自己組織化マップを生成するアルゴリズム」という意味で用いている．

2.2 SOM のアルゴリズム

マップの生成には入力層・競合層の 2 層を使い，それらは互いに完全結合で結ばれている．層間の結合にはそれぞれ多次元ベクトルデータの重みが与えられており，学習前に乱数で初期化する．ある多次元ベクトルデータの入力信号$\boldsymbol{x}$が入力層に投入されたとき，競合層のユニット i は自分が持つ重み$\boldsymbol{w}_i$ と入力信号間の距離を計算する．この距離測度には多くの場

合ユークリッド距離が用いられる．この距離を各競合層ユニットごとに算出し，距離が最小となるユニット c をベストマッチデータ，即ち勝利ユニット (ウィナー) として決定する．

$$c = argmin_i\{||\boldsymbol{x} - \boldsymbol{w_i}||\} \tag{2.1}$$

この際，勝利ユニットの周りには近傍と呼ばれる領域が設定される．この近傍領域に含まれるユニットは，いわば準勝利者として，勝利者に比べ更新値は少ないものの勝利者と同じく更新の対象とされる．ここでの h_{ci} とは近傍関数と呼ばれ，離散時間座標 $t(t = 0, 1, 2, ...)$ によって次のように定義される．

$$h_{ci}(t) = h(||\boldsymbol{r_c} - \boldsymbol{r_i}||, t) \tag{2.2}$$

r_c，r_i はそれぞれ競合層におけるユニット c と i との位置ベクトルであり，$\parallel r_c$ - $r_i \parallel$ が増加するにつれ，$h_{ci} \to 0$ とする．より位置ベクトルの近いものが近傍としての更新値を多く得られるという事になる．この近傍領域は最初は大きくとっておき，時間軸と共に減少させていく．これは学習の収束を促すためである．この領域内のユニットを近傍集合とし，Nc と定義する (これによって，時間の関数としての $Nc = Nc(t)$ が定義できる)．もし，$i \in N_c$(i が N_c 内のノード) なら $h_{ci} = \alpha(t)$，$i \notin N_c$(i が N_c 外のノード) なら，$h_{ci} = 0$ である．この時，$\alpha(t)$ の値を学習率係数と言う $(0 < \alpha(t) < 1)$．$\alpha(t)$ は普通，時間軸上において単調減少させる（式 2.3）．α_0 は，α の初期値であり，一般的に 0.2〜0.5 の値を選ぶ．この値によって学習の収束率が変わってくるため，設定には注意を払う必要がある．T は，行われるべき学習での予定された全更新学習回数である．

$$\alpha(t) = \alpha_0(1 - \frac{t}{T}) \tag{2.3}$$

また，近傍領域 $N_c = N_c(t)$ についても同様に，

$$N_c(t) = N_c(0)(1 - \frac{t}{T}) \tag{2.4}$$

と表すことが出来る．$N_c(0)$ は初期値である．

そして，勝利者のベクトルデータが入力値のベクトルデータに近づくように更新を行う．勝利ユニットを含めた更新式を以下に示す．

$$\boldsymbol{w_i}(t+1) = \boldsymbol{w_i}(t) + h_{ci}(t)(\boldsymbol{x}(t) - \boldsymbol{w_i}(t)) \tag{2.5}$$

これら，一連の更新作業を図示したものが図 2.1 である．

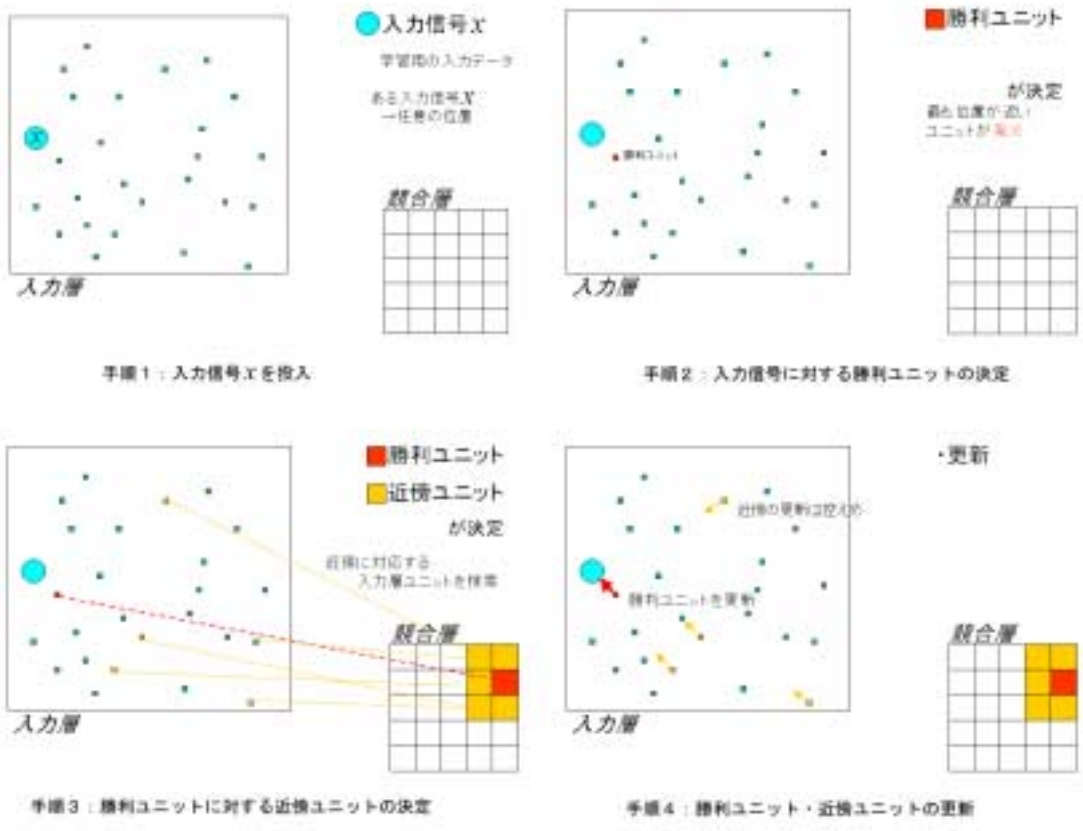

図 2.1　SOM アルゴリズム

この更新作業を，入力層のユニット変化が少なくなるまで繰り返すことにより，競合層内の重みデータが自己組織化される．その重みのデータを色の濃淡などに置き換え，出力したものが SOM マップである．生成されたマップの例を図 2.2 に示す．

2.2.1　Online SOM

Online SOM は前項で説明した更新作業を入力信号が投入されるごとに逐次行う．マップが生成されるまでには複数の入力信号群を投入する必要がある．この入力信号が投入され，Online SOM が更新を行うまでの処理単位をここではレコードと呼ぶ．複数のレコードを

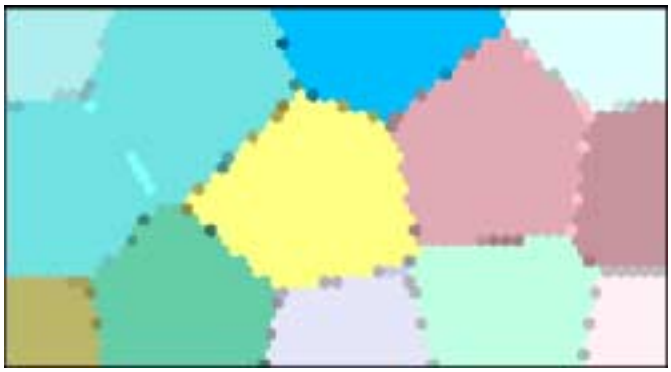

図 2.2　生成された SOM マップの例

入力することによって，様々なユニットが勝利者として，また，近傍として選択され，更新されていく．入力信号群がすべて投入し終わる，つまり，全レコードが終了すると，再び最初のレコードから処理を繰り返す．この，最初のレコードから全レコードが終了するまでの処理単位をエポックと呼ぶ．このエポックは前述したように重みの変化がなくなるまで行われ，マップの形成に至る．

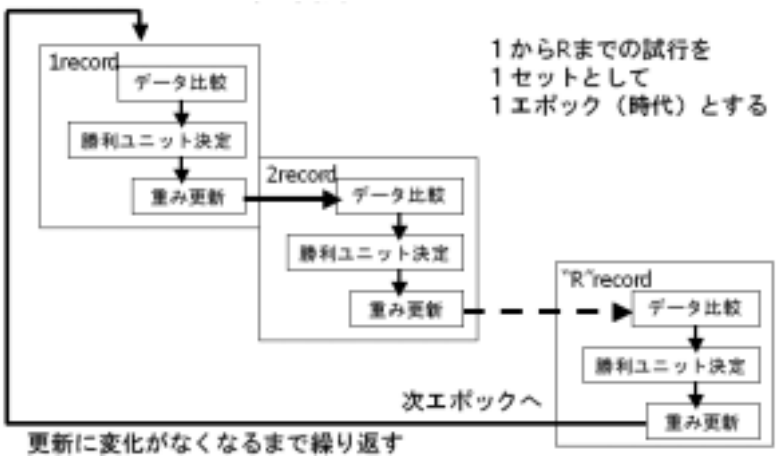

図 2.3　Online SOM アルゴリズム

この Online SOM は従来型 SOM とも呼ばれ，最も基本的な SOM アルゴリズムである．しかしながら，非常に多くの次元・データを持つマップを作ろうと試みた場合，入力信号の数は膨大になり，それにつれて更新の回数も増えてゆく．また，入力信号と各ユニットとの距離は前回の更新結果により変化する．つまり，データの入力順序によってはエポック終了時の結果が大きく異なる．ここに，例として図 2.4(株式会社 Xanagen 社ホームペー

ジ:http://www.xanagen.com/より転載)[4] を示す．図中の左のマップは Online SOM による細菌の遺伝子の解析結果を，また右はその入力信号を逆順に投入した際に生成されるマップを示しているが，この 2 つのマップからも Online SOM において入力信号の投入順序が学習に与える影響は無視できないことがわかる．

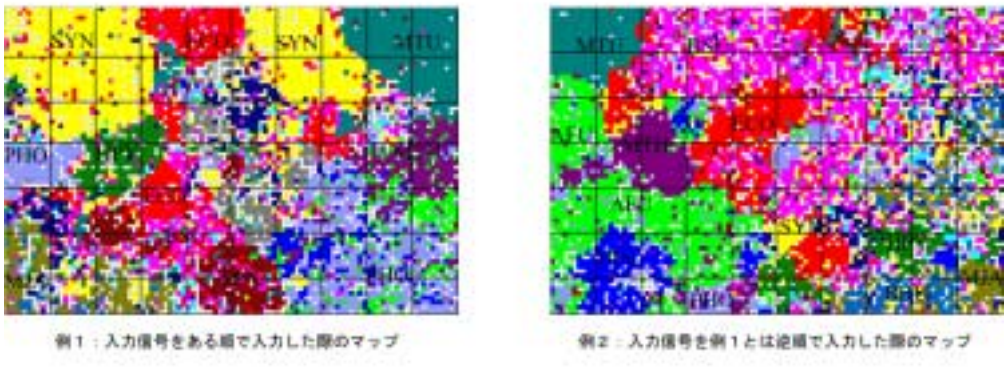

図 2.4　信号の入力順序による生成マップの違い

2.2.2　Batch SOM

Batch SOM が Online SOM と異なるのは，各重みの更新をレコードの終了時ではなく，エポックの終了時にのみ行うという点である．(図 2.5)

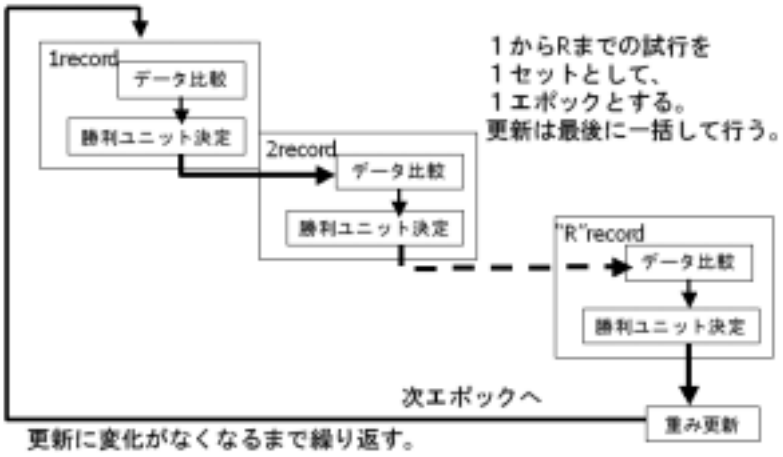

図 2.5　Batch SOM アルゴリズム

そのため，各ユニットには勝利回数，勝利値の値をそれぞれ用意し，勝利者として選ばれた場合にはそれぞれの値を追加するという処理を行う．勝利者として選ばれた場合，そのユニットは入力値$\boldsymbol{x}$に近傍関数 h_{ck} をかけた値を勝利値として得ることができる．全レコードが終了すると各ユニットに対して値の更新を行う．この際の更新式は以下のようになっている．また，勝利回数としている値は厳密に言えば単純なユニットの勝利回数ではなく，各レコード毎に近傍関数により与えられた値である．今回は，近傍を考えず勝利ユニットを 1 つのみにした場合の考え方からこう呼ぶこととする．

$$\boldsymbol{w}_k = \frac{\sum_{t'=t_0}^{t'=t_f} h'_{ck}(t')\boldsymbol{x}(t')_s}{\sum_{t'=t_0}^{t'=t_f} h'_{ck}(t')} \tag{2.6}$$

式に示すとおり，各更新値は勝利値を勝利回数で割った形になっている．この為，Batch SOM では，Online SOM のように学習率係数 $\alpha(t)$ を指定する必要がない．

この，エポックの終了時のみに更新を行う Batch SOM では，Online SOM に比べ，単純にレコードの数だけ更新処理が抑えられるため，処理時間の短縮が期待できる．また，すべての入力信号が投入されるまで更新が行われないために，更新の結果が入力信号の順番に影響されないというのも大きな特徴である．

しかしながら，この Batch SOM を DDMP をはじめ，ハードウェア上の演算制御では，ほとんどの場合，割り算は他の四則演算に比べて処理に時間がかかるうえ，実装するのが困難である．この為，割り算の使用は好ましくない．DDMP にいたっては割り算のモジュールが用意されておらず，また，既存の演算子の組み合わせでこれを再現しようとすると複雑になり，PE の負荷が増えたり，サイクル数の増加につながるなど，現段階では Batch SOM を DDMP では実装する事は困難である．

そこで，本研究では，この欠点を補った変形 Batch SOM を提案する．

2.3 変形 Batch SOM

2.3.1 基本アルゴリズム

変形 Batch SOM は前節の Batch SOM のアルゴリズムを改良したアルゴリズムである．Batch SOM では更新式に割り算を使用していたが，変形 Batch SOM では割り算の変わりにビットシフトという手法を使う．図 2.6 のようにすることで，結果的に割り算の結果を得ることができる．しかしながら，この方法では除算数は必然的に 2^n に限られる．よって，勝利回数をこれらの値に近似する．

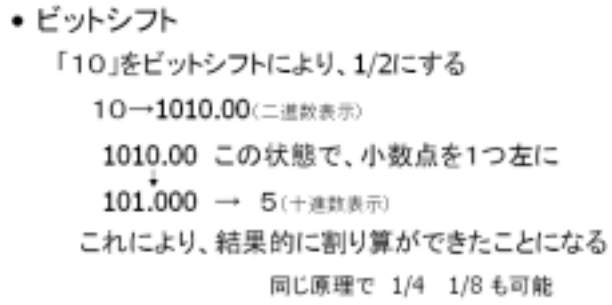

図 2.6 ビットシフトの例

2.3.2 勝利回数近似

図 2.7 勝利回数近似テーブル

図 2.7 に示したとおり，勝利回数を近似する．例えば，勝利回数が 3 回の場合は除算数を 4 にする．ここで，2 にしないのは学習を過剰に行わないようにするためである．(図 2.8 を参照)

この勝利回数近似により，DDMP に Batch SOM のアルゴリズムを実装することが容易になる．また，DDMP 以外のケースでもプログラムの記述の簡略化，更新時の処理時間の

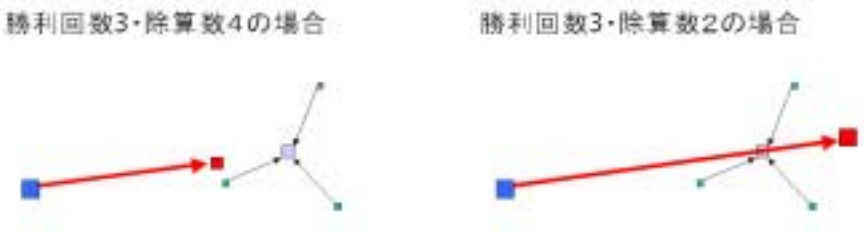

図 2.8　近似数の違いによる学習結果の違い

短縮等が期待できる．しかしながら，勝利回数を近似することによって更新時の各ユニットの移動が少なからず縮められてしまうため，全体的な処理終了までの時間 (＝エポック数) が増え，収束率が低下してしまうなどの欠点が現在のところ考えられる．

第 3 章

DDMP

3.1 DDMP とは

DDMP はマルチメディア信号処理向けに開発された，データ駆動型信号処理プロセッサである．主な特徴として，データ駆動処理方式の採用があげられる．パイプライン処理方式は，ハードウェアの処理能力向上手段として，理想的であると考えられている．しかし，パイプライン処理方式が最大の効果を発揮するのは，パイプラインの中に，一定のデータ流量が定常的に確保されていることが大きな要件となる．

さらに，パイプラインの複数の段からの外部資源へのアクセス競合などによって，パイプライン段相互間のインターロッキングが要求される環境では，いわゆるバブルの発生によってパイプライン処理の効果が減殺されるばかりではなく，パイプライン制御機能のいたずらな複雑化を招く．これらの要請をシステム的に保証する最も明確な手段は，パイプライン内を流れる複数のデータの組の処理が相互に完全に独立であるような処理原理の導入である．

データ駆動型の大きな特徴は，個々の演算に必要なデータがそろった時点でそれぞれの演算が起動され，いったん起動された演算相互間には，何ら依存関係がないことにある．すなわち，いったん起動された演算は，ほかの演算とまったく無関係に実行される．また，自己タイミング方式の特徴によって，信号を処理していない場合の消費電力はほぼ零となり，電力対性能比の向上が達成されている．

3.2 DDMP での実装

以下に，DDMP のフローグラフシミュレーターでの Online SOM，Batch SOM，および本研究で提案する変形 Batch SOM の実装の手順について記す.

DDMP の専用型プロセッサの構成は，OCP(Operation and Control Procesor) と VMP(VideoMimoryControlProcesser) が各 2 つ，計 4 つのプロセッサからなっており (図 3.1)，シミュレーション時には設定ファイルを用いて，この構成は任意に変更できる．本研究では，OCP を 4，8 個実装し，それぞれに対応して VMP を，また，各それぞれにホストとして更に 1 つずつ OCP を接続するというプロセッサ構成で実装を行った．そして，それぞれ実装されたアルゴリズムを DDMP のフローグラフシミュレータ上で実行し，その学習時間を評価する.

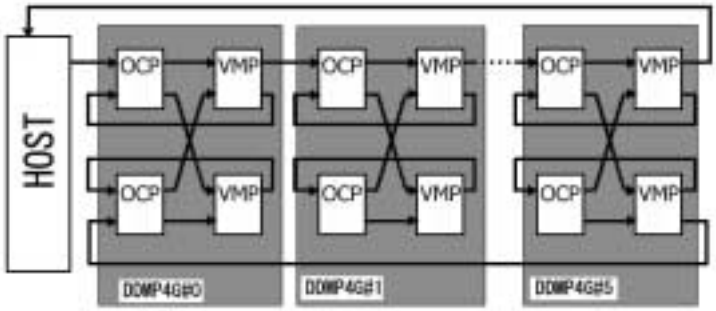

図 3.1 DDMP のプロセッサ接続構成例

3.2.1 準備

プログラムの実装，実行に必要なファイルを表 3.1 に記す.

この他に，OCP を 4 つ接続したプロセッサで評価するためのコンフィグファイル (config.emu) も用意した．また，DDMP のプログラムを作成するにあたって，各試行ごとにパケットファイルを作成する必要がある．テキストエディタで拡張子 inp のファイル (図 3.2) を作成し，入力パケット生成ツール (mkpkt_4g.exe) を実行すれば拡張子 pkt のファイルを作ることが出来る．最上行の#10 は, パケットの投入感覚を表し，この場合，投入感覚は 4

拡張子	用途	起動アプリ
＊.Fg ＊.mod	フローグラフエディタで作成したプログラム	フローグラフエディタ fed 4g.exe
＊.Fe	アセンブル出力ファイル (オブジェクトファイル) データフローグラフ (DFG) 実行形式	シミュレーター fsim 4g
＊.Fg	フローグラフエディタ中間ファイル (アセンブラソース)	テキストエディタ
＊.inp	テキスト形式で記述 入力パケット用ソースファイル	テキストエディタ
＊.pkt	標準型入力パケットファイル	

表 3.1 DDMP で使用するファイルの一覧

サイクルである．その次の行の%1 は，一度に投入するパケット数である．この場合は，1 度に一つずつのパケットを投入することになる．

```
1  #10↓
2  %1↓
3  !        FD#  LN#  PX#  PE#  ENT#  DATA↓
4  ! N!     0    0    0    0    1     10↓
5           0    0    0    0    3     17↓
6           0    0    0    0    5     19↓
7           0    0    0    0    7     13↓
8           0    1    0    0    1     26↓
9           0    1    0    0    3     34↓
10 ↓
11 [EOF]
```

図 3.2 *.inp ファイルの例

フローグラフのプログラムは，入力ユニット，出力ユニット，宣言文，演算ユニット，モジュールの 5 つからなる (図 3.3)．データはまず入力ユニットへ入力され，それから，各演算ユニットへと流れていく．そして，モジュール (ここで言うモジュールとは，演算ユニットの集合のことである) などを通り，最終的には，出力ユニットから出力される．宣言文は，プログラム中でわかり易くリネームしている PE の番号を実際のアセンブラソースを生成するときに本来の PE 番号に変更する意味を持つが，今回はそのファイルを外部ファイルとし

て genelal.h というファイルからインポートしている．図 3.3 中で説明すると，フローグラフを作成する際には，実際の出力ユニットには 54 という PE 番号が割り振られているが，ここでは Host に出力するという意味で genelal.h というファイル中に#define host 54 という記述を加えることで扱いやすく表現している．

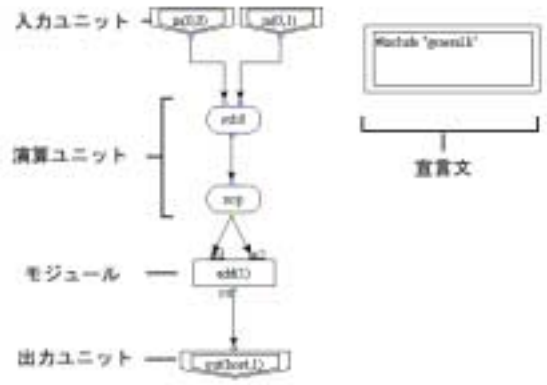

図 3.3 フローグラフの例

フローグラフエディタで実装していく場合に気をつけなければならないのは，データ駆動処理方式の特長によって，ある演算ユニットにおいて，流れてくる 2 つのデータの条件が一致したときに演算が行われる点である．この条件は，演算ユニットによって様々であるが，パケットの持つ世代番号情報である，フィールド (FD)，ライン (LN)，ピクセル (PX) の 3 つが完全に一致しないことには，演算は行われない．また，フローグラフ中のそれぞれのユニットには，必ず一つの PE が割り当てられていなくてはならない．

3.2.2 Online SOM の実装

DDMP による Online SOM の実装は入力層分割法 [5] という手法を用いた．これは，各 PE が入力層のユニット一つを担当し，競合層の各ユニットに対して重みを持ち，競合層のどのユニットが勝利，あるいは近傍となって重みを更新してもすべての PE が同等の処理をするため負荷は均一となるというものである．従来行われてきた SOM の並列化の手法である競合層分割法における，更新を勝利ユニットとその近傍ユニットを割り当てられた PE だ

けが行い，勝利ユニットとその近傍の割り当てられた PE のみに負荷のかかる，負荷の不均一を防ぐことができる．

以下に，Online SOM を DDMP に実装する際の大まかな流れを示す．

1. レコード数分の入力層の値および競合層の値をメモリに保存する
2. 各ＰＥで入力層のユニットと競合層のユニットとの誤差をとる
3. 2 で求めた誤差をホストに送信
4. ホストで競合層の各ユニットについての誤差を集計し，勝利ユニット及びその近傍ユニットを決定
5. 4 で得たデータを各プロセッサに送信し，各プロセッサで勝利ユニット及びその近傍ユニットに関連した重みを更新
6. 更新された競合層ユニットの値と保存していた入力層データを次のレコード用に用意する
7. 全レコードが終了するまで，2-6 を繰り返す
8. 全エポックの終了まで 2-7 を繰り返しプログラムを終了する

ここで，手順の最初に値を保存するという作業について補足する．DDMP ではパケットの持つ世代番号情報が一致しないと演算が開始されなというのは前述で説明した．そのため，次レコード用の入力層の値が入力されても，現在のレコードで競合層の重みが更新され，次レコードとしてパケットが流れてくるまでこのパケットは演算待ちの状態になる．この状態で 1 つの PE に大量のデータが演算待ちのまま溜まり続けると PE がオーバーフローを起こしてしまう．そこで，一旦値をメモリ内に格納し，そのパケットを破棄する．そして，前レコードの競合層の重み更新が完了した時点でメモリから値を読み出し，競合層ユニットと比較する．こうすることで，余計なパケットの待ち時間を解消することができ，より多数のレコードの試行も可能になった．また，この値をそのまま残しておき，次エポックでの試行にも利用できる．

以下に実際に実装された DDMP のフローグラフがどのようになっているか，各機能別に

解説していく．

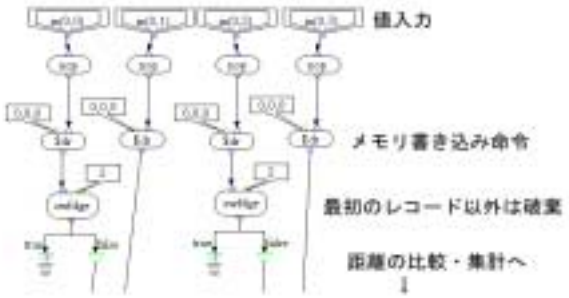

図 3.4 フローグラフ：入力部

図 3.4 が，各データの入力部分である．この図の，in(0,0) が入力部分であるが，この入力 0 番および 2 番に入力信号の位置情報を，1 番，3 番にそれぞれのベクトルの位置情報を投入する．そして，それぞれの値をメモリに格納する．ここでのメモリ書き込みモジュール $dr の左の定数，0,0,0 は，オフセット値であり，この場合は世代番号情報をそのままメモリアドレスとして書き込んでいる．そして，先ほど説明したとおり，入力信号入力側には次々と次レコード用の情報が流れ込んでくるが，最初のレコード情報のみを比較部分に流すため，世代番号情報のうち，FD の値が 2 以上のものはここでは値を記憶するだけで，比較部分には流さない．図中の swfdge は右定数の値以上の FD を持つデータが流れてきたときに真を選択するという演算子で，この場合，FD の値が 1 のもののみを比較部分に流している．ここで，今回の実装におけるデータの世代番号の割り振りを表 3.2 に示す．

次に，比較部分の説明に移る．

図 3.5 では，先ほどの入力部から流れてきた，入力層および競合層のベクトルの値を，まず MoD というモジュールに流す．このモジュールを展開したものが同図内の左側に囲ってある部分である．ここでの詳細な説明は省くが，startLN から endLN までの流れてきた入力層・競合層の値の差をとり，それらをすべて合計して出力するという機能を持っている．そして，各モジュールから出力された値をさらに合計し，最終的には全 16 次元分の差を合計した値が求められる．本来，距離という概念からすれば，この値からユークリッド距離を

FD	LN	PX	割り当て
1	0	1	Record1 でのユニット 1 の総合距離
1	1	1	Record1 でのユニット 1・次元 1 の値
1	2	1	Record1 でのユニット 1・次元 2 の値
・	・	・	
・	・	・	
2	0	1	Record2 でのユニット 1 の総合距離
2	1	1	Record2 でのユニット 1・次元 1 の値
2	2	1	Record2 でのユニット 1・次元 2 の値
・	・	・	
・	・	・	
100	16	0	エポック数-1 の値

表 3.2 Online SOM で使用するパケットデータの世代番号割り振り

求めるのがベストであるが，処理が複雑になり，サイクル数の増大につながることや，各ユニット値の比較が目的であるため，ここでは合計した値をそのまま比較している．また，これらの値の合計の処理には当然世代番号をあわせる必要があるため，すべてのモジュールから出力される LN の値を 0 に統一している．こうして，最終的には表 3.2 で示したような，すべてのユニットの LN0 には総合距離が集計されている事になる．この値を一度メモリに格納し比較部分は終了である．この値を基にして勝利者の決定を行う．

図 3.6 での一番上のモジュールは，先ほど比較部分で算出された各ユニット毎の入力信号との距離を比較し，いちばん距離が少ないもののみを出力する．ここでの min では左右のパケットが持つ値を比べ，値の少ないものを出力している．しかし，この処理では比較の段階で世代番号をあわせる必要があるため，最小距離が算出されるものの，その値を持っているユニットの情報は失われてしまう．そのユニットを特定するため，パケットを 2 通りにわけ，a のモジュールには値を初期化したパケットものを，b のモジュールには勝利値を持った

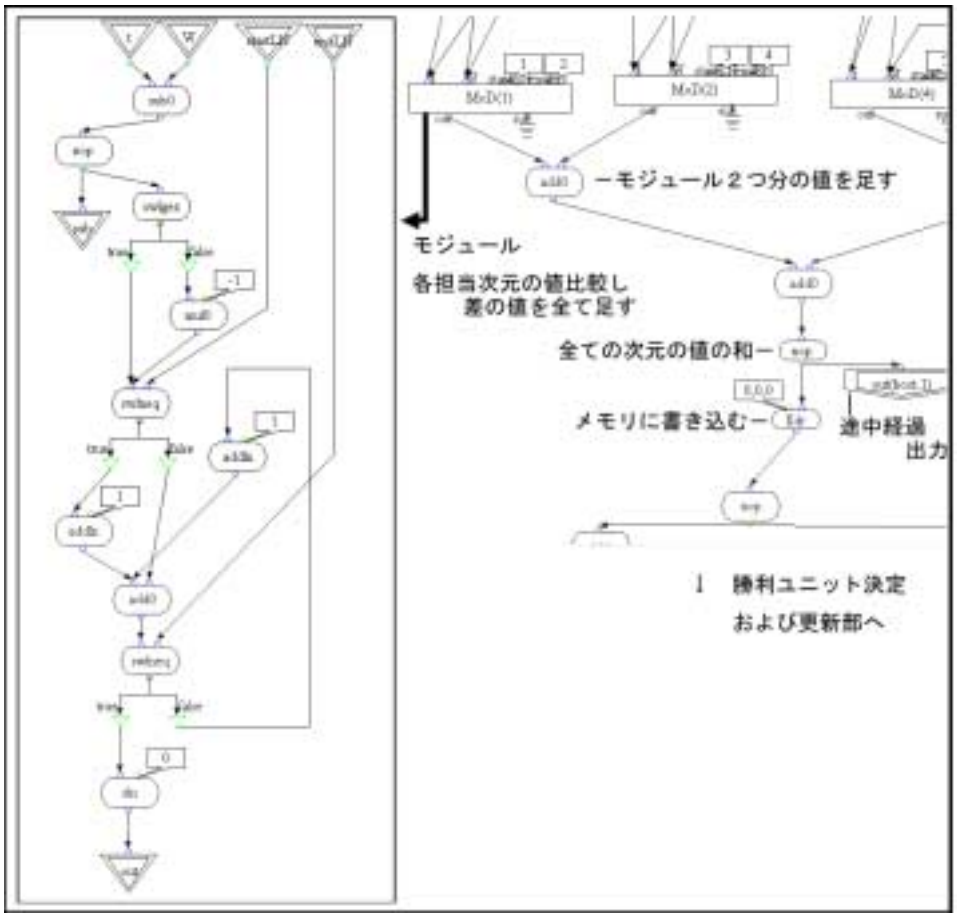

図 3.5　フローグラフ：距離の比較・集計

ままのパケットを流す．この，expand_PX はユニットの数だけパケットを複製するモジュールである．a のモジュールで複製されたパケットには先ほど格納した，それぞれのユニットの値を戻し，そして，b で複製された，勝利した値を持ったパケットそれぞれのユニットが持っている値を比べる．その値を持っているユニットが勝利ユニットだと判定することができる．その処理はモジュール，select1win にて行っているが，ここでは同時に勝利パケットが 2 つだった場合に，勝利ユニットを 1 つに絞る働きも持っている．また，このモジュールから出力されるパケットは勝利ユニットの番号を値に持つパケットである．

先ほど出力されたパケットを 2 つに分け，spx を通すことで，パケットの世代番号を勝利ユニットの世代番号に設定する．ここで一度値を初期化し，すべての次元について値を更新するためにまず 1 を設定，そして下のモジュールで次元数だけ複製する．ちなみに，この

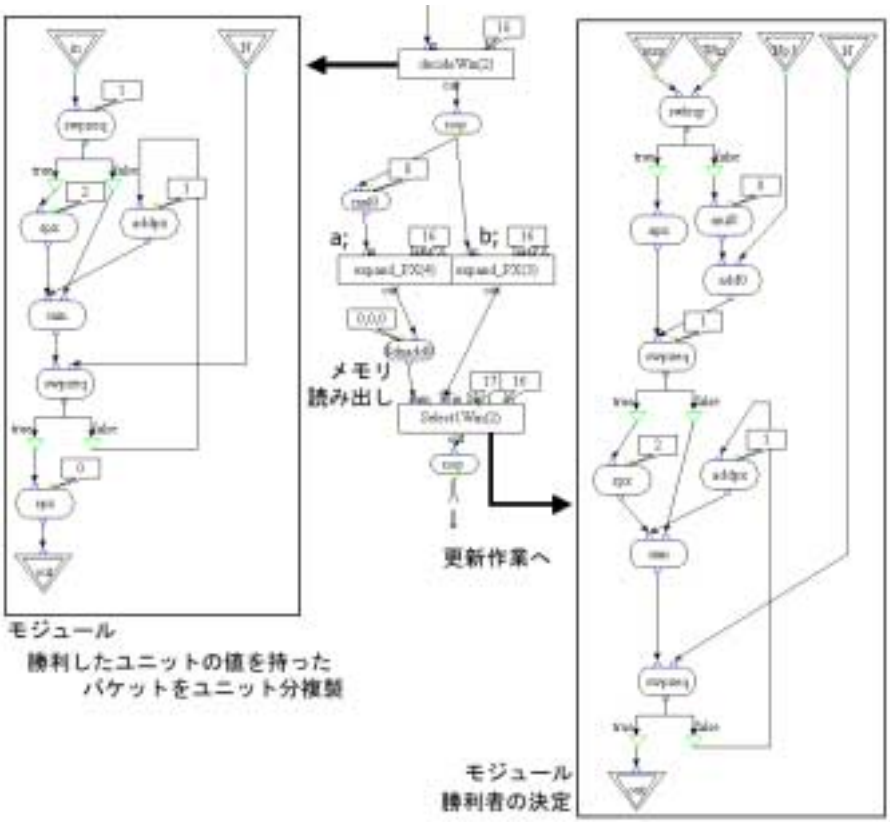

図 3.6　フローグラフ：勝利ユニット決定

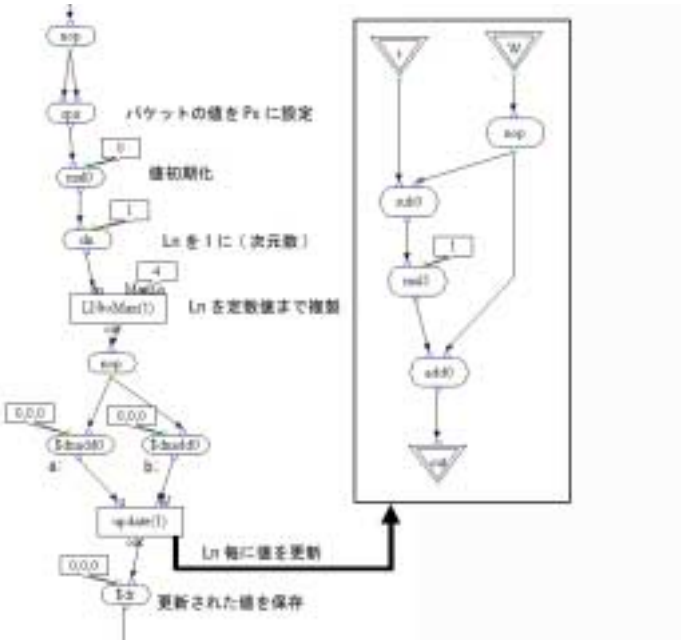

図 3.7　フローグラフ：更新

比較・集計部分では複数の PE に処理を分担させ，並列処理を行っている．図 3.7 の例では，この PE が 1 から 4 の次元までを担当していることになる．そして世代番号の設定の終わったパケットから順次値を更新する．a では入力信号の値を，b には競合層の値を代入し，モジュール update で更新を行う．更新式にあるように，まず入力信号と競合層の値の差をとり，その値に学習率係数をかける．ここでは，mul1 がこの処理を担当しているが，この mul1 はパケットの値に定数をかけた結果に左ビットシフトを加えるというもので，この場合 1 をかけた後，1 ビットシフトしていることから，結果的に 0.5 をかけた事になる．この 0.5 というのが学習率係数である．そしてその値を元の競合層の値に加えて更新は終了である．この値を新たにメモリに上書きする形で書き込む．この更新された値は次のレコードの比較の際に呼び出される．

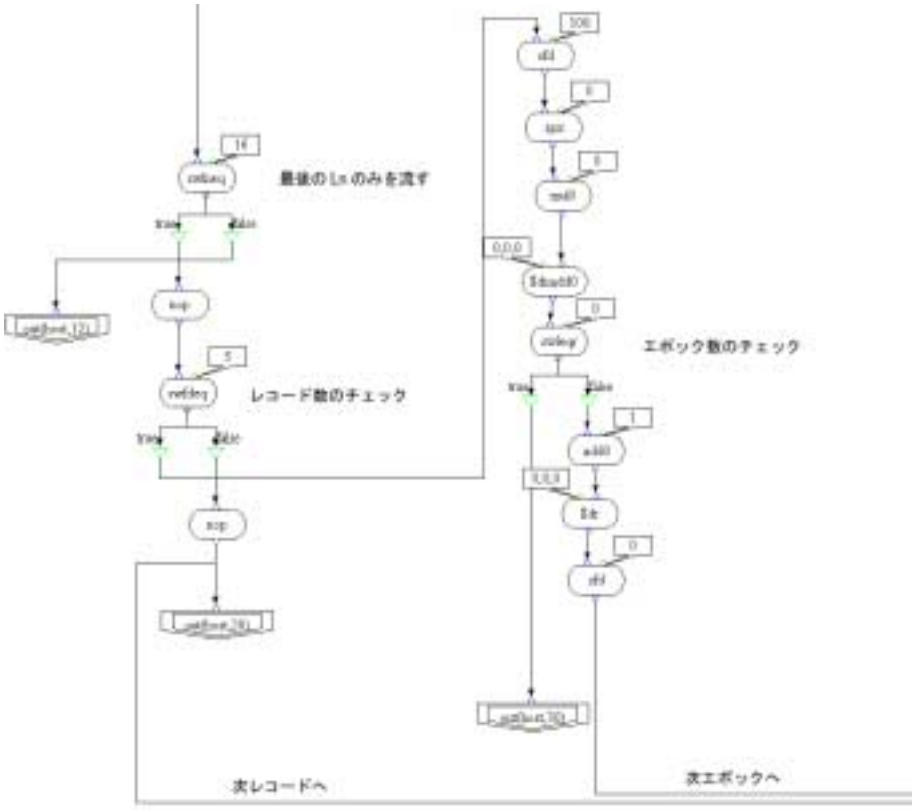

図 3.8　フローグラフ：次レコード・エポックへ

ここで，図 3.8 は更新済みのユニットから一番最後の次元のもののみを取り出し，それを

次レコードのきっかけとして用意しようというものである．図では 16 次元のユニットのみを取り出し，現在のレコードの値をみる．この演算子の定数 (図では 5) のレコードまで達していない場合は，次のレコードの作業に入るため，パケットの複製および値のセットを行う．また，レコードが達している場合は図右側のエポック数の判定を行う．DDMP には世代番号情報を 3 種類しかなく，現在のところすべての世代番号情報は使用中であるので，エポック数の場合のみ，FD，LN，PX がそれぞれ，100，16，0 にエポック情報を割り当てている．その理由としては，レコード数 (FD) が現在のところ 100 を超えないこと，次元数 (LN) の最後のパケットを判定に使用していることから再設定しないでよいように，ユニット番号 (PX) はどれにも当てはまらないように，などがある．このエポック情報に割り当てたパケットの値 +1 が現在のエポック数となる．設定されたエポック数に満たない場合，この値に 1 を加え，レコードの値をリセットした後，パケットの複製および値のセットを行っている．

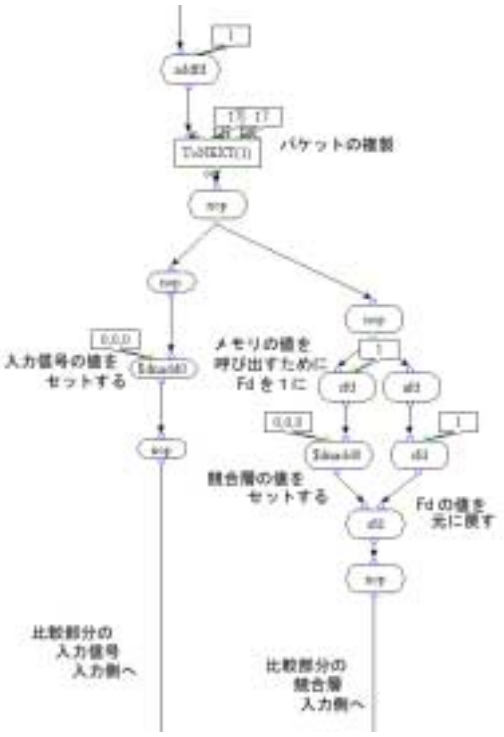

図 3.9　フローグラフ：比較部分に備えての値のセット

図 3.9 にて，次のレコードに備えて入力信号側，競合層側それぞれの値をセットし，準備

する．まず，一番上の addfd では，次のレコードに進むために値を 1 追加している．ここで，エポック終了後の次エポック・最初のレコードの場合には，ここに送られてくるパケットの FD は 0 になっている．そして，次のモジュール ToNEXT では，パケットの数を次元数・ユニット数だけ複製している．ここで，定数が 17 になっているが，これは必要とされる各値 +1 に設定しているためである．ここで，パケットを 2 つにわけ，片方では入力信号の値を，もう片方では競合層の値をセットしている．ここで，入力信号側ではレコード数の分だけデータが用意されているので FD の値を変更する必要はないが，競合層側では値が FD1 でしか用意されていないため，一度 FD を 1 に戻す必要がある．そこで，パケットを 2 通りにわけ，片方で FD を 1 にして，値をセット，もう片方は afd にて現在の FD の値をパケットの持つ値として取り出し，値のセットが終わったパケットに sfd で再設定している．この作業を加えることによって，FD の値を保存したまま値のセットが行える．そして，次レコードへの比較準備が完了する．

以上が，入力層分割法で行った Online SOM の DDMP への実装図である．

3.2.3 変形 Batch SOM の実装

変形 Batch SOM を実装する際には，Online SOM との比較を考え更新処理以外の部分を Online SOM と共通とした．よって，図 3.4 から図 3.6，および図 3.8，図 3.9 の部分は共通である．

以下に，変形 Batch SOM の処理の大まかな流れと，変形 Batch SOM で使用する際のデータの世代番号の割り振りを表 3.3 に示す．

1. レコード数分の入力層の値および競合層の値をメモリに保存する
2. 各ＰＥで入力層のユニットと競合層のユニットとの誤差をとる
3. 2 で求めた誤差をホストに送信
4. ホストで競合層の各ユニットについての誤差を集計し，勝利ユニット及びその近傍ユニットを決定

5. 競合層ユニットの値と保存していた入力層データを次のレコード用に用意する
   また，4 で得たデータを各プロセッサに送信し，各プロセッサで勝利ユニットの勝利回数，およびその近傍ユニットの勝利回数を加算
6. 全レコードが終了するまで，2-5 を繰り返す
7. 各ユニットに保存されている値から，競合層ユニットの更新を行う
8. 全エポックの終了まで 2-7 を繰り返しプログラムを終了する

FD	LN	PX	割り当て
0	0	1	ユニット 1 の勝利回数
0	1	1	ユニット 1・次元 1 の勝利値
0	2	1	ユニット 1・次元 2 の勝利値
・	・	・	
・	・	・	
1	0	1	Record1 でのユニット 1 の総合距離
1	1	1	Record1 でのユニット 1・次元 1 の値
1	2	1	Record1 でのユニット 1・次元 2 の値
・	・	・	
・	・	・	
100	16	0	エポック数-1 の値

表 3.3 Batch SOM で使用するパケットデータの世代番号割り振り

変形 Batch SOM 特有の実装図を以下に示す.

図 3.10 は勝利ユニット決定後の勝利回数の加算時の処理である．ここで使用しているモジュール Winner Px16 and Count の前半部分は勝利ユニットの決定部分で，Online SOM と共通であるが，その後パケットを 2 つにわけ a のルートで勝利回数をメモリに格納している．まず，アドレスを合わせ，勝利値 1 を与える．将来近傍が実装されれば近傍関数をか

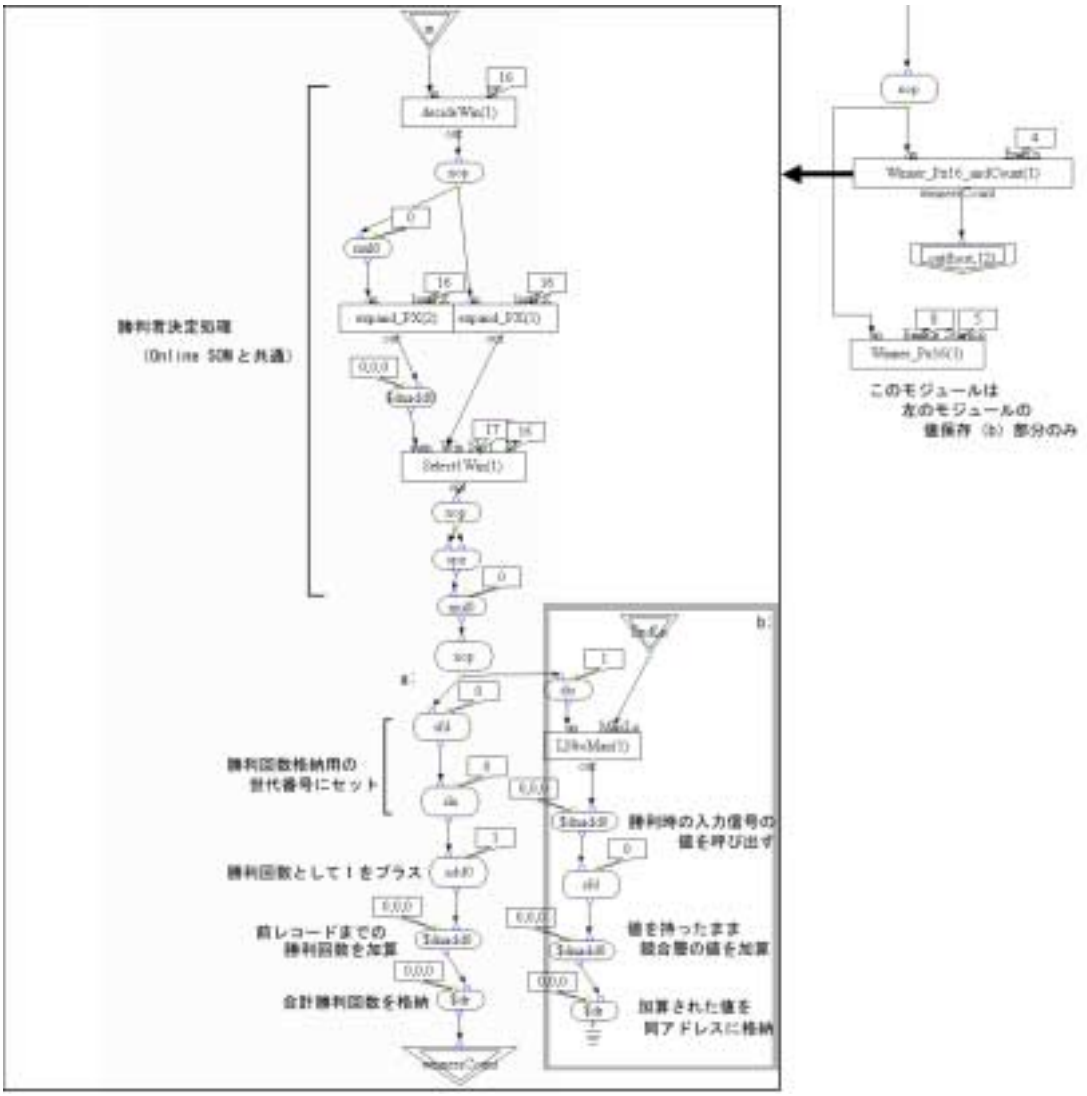

図 3.10　フローグラフ：勝利回数，および勝利値のメモリ格納

けた値を勝利値とすることになる．そして，前レコードまでの勝利値を加算し，改めてメモリに格納する．また，b のルートでは次元毎の勝利値をメモリに格納しており，基本的には勝利回数と同じ処理を行っている．また，この部分も並列化を行っており，このモジュールは次元 1 から 4 を担当しているが，この担当が最初の部分のみ (次元 1 を担当に含むもの) が勝利回数の更新を行っており，それ以降のモジュールは勝利値の更新のみを行っている．このように勝利回数と勝利値をレコード分だけ更新した後，それぞれのユニットの更新に入る．

ちなみに，この処理は各レコードの比較終了後に行われるが，同時に次のレコードにもデータを回しており，勝利回数の加算処理を待たずに次のレコードの処理に入れるため，全体的な時間短縮が図れる．

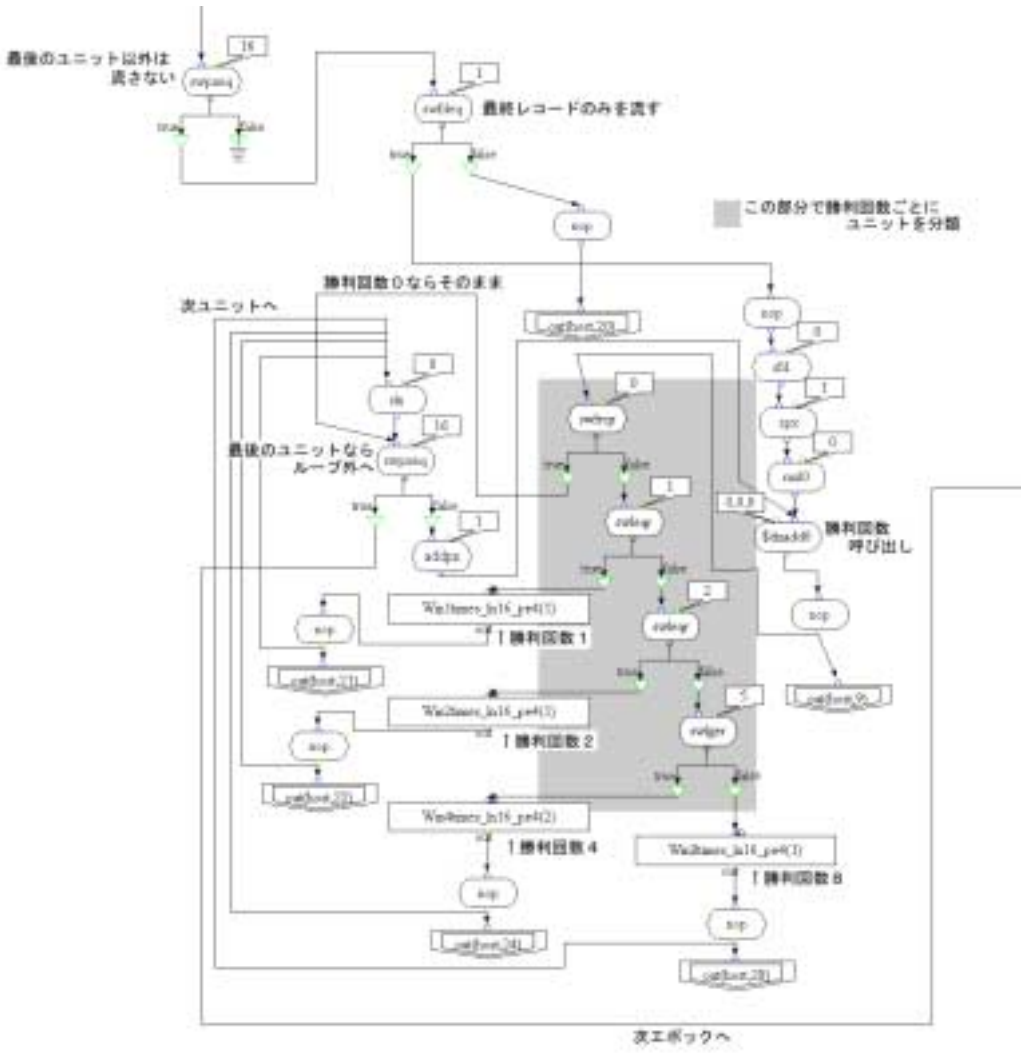

図 3.11　フローグラフ：勝利回数・勝利値を基にした更新

ここでの更新は勝利回数によって更新するモジュールに振り分ける処理を行う．まず，エポックの最終パケット以外を破棄し，最終パケットの世代番号をユニット 1 の勝利回数に割り当てられた番号にセットする．そして，ユニット 1 の勝利回数をメモリから呼び出し，その格納されている値をもとに更新するモジュールに振り分ける．例えば，ユニット 1 が 2 回

勝利していた場合，除算数は 2 であるので $Win2times$ にパケットを流す．

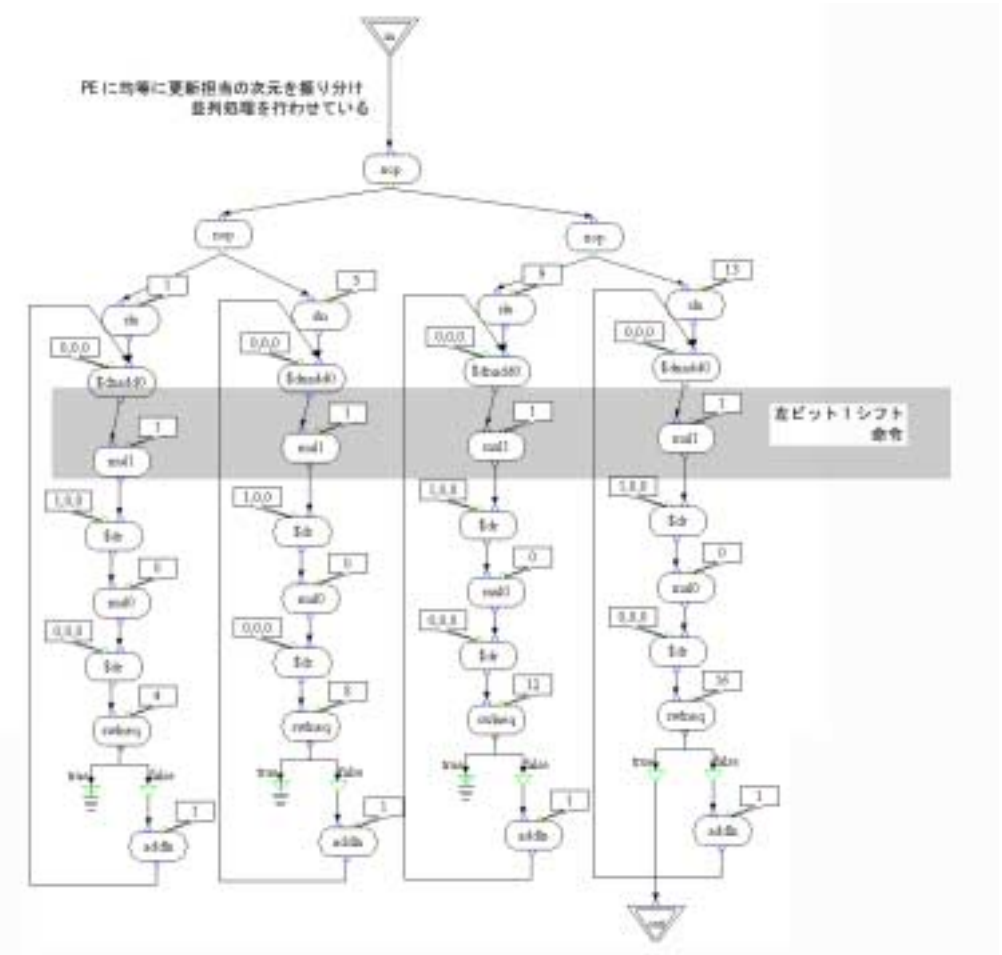

図 3.12　フローグラフ：モジュール $Win2times$

モジュール $Win2times$ に流されてきたデータはここで 4 つに分けられ，並列に更新される．ここの 4 つというのは単純に PE の数である．ここでは 16 次元を 4 つの PE で分担しているので，各部分で 4 つづつ更新を繰り返している．ここで注目すべき点は中ほどにある $mul1$ である．ここでは 1 をかけた後左ビットシフトして値を 2 で割った値と同等の結果を得ている．このモジュールでは除算数が 2 であるので，左ビットシフト数は 1 であるが，除算数が 4 の場合は 2 ビットシフトする必要があり，$mul2$ を．8 の場合は $mul3$ を使用することになる．ちなみに，一回も勝利しなかったユニット (勝利回数が 0) は処理短縮のためにも何も処理しないでそのまま次のユニットへと処理を移す．これを前ユニット数繰り返し，

次のエポックへとパケットを送る．

ここまでの処理をエポック数だけ繰り返して，すべての処理の終了である．

これまでに示した Online SOM，変形 Batch SOM それぞれを DDMP に実装したフローグラフを使ってそれぞれの性能を比較するために，実験を行う．

第 4 章

実験

4.1 実験手順

今回の実験は各アルゴリズムを実装したフローグラフに同じサンプルパケットデータを流し，フローグラフシミュレータ上でプログラムを走らせ，その終了までのサイクル数で，プログラム終了までの時間で比較を行った．実験に使用するパケットは，表 4.1 に示した 16 通りである．また，表中の M は次元数，N はユニット数，R はレコード数，そして E はエポック数を表している．

PE 数	M×N	R	E	PE 数	M×N	R	E
4	１６×１６	1	1	8	１６×１６	1	1
	１６×１６	1	5		１６×１６	1	5
	１６×１６	2	1		１６×１６	2	1
	１６×１６	2	5		１６×１６	2	5
	１６×１６	4	1		１６×１６	4	1
	１６×１６	4	5		１６×１６	4	5
	１６×１６	8	1		１６×１６	8	1
	１６×１６	8	5		１６×１６	8	5

表 4.1　実験に使用するパケット

これをそれぞれフローグラフシミュレータに掛け，プログラム終了までのサイクル数を比較する．

4.2 結果

それぞれの実験結果は以下のようになった．

	Record 1	Record 2	Record 4	Record 8
Online SOM Epoch 1	10345	35216	84981	184338
Online SOM Ephoch 5	111374	235903	484688	984163
変形 Batch SOM Epoch 1	11422	33439	77489	165597
変形 Batch SOM Epoch 5	116574	226540	446665	887463

表 4.2 実験結果（PE4)

	Record 1	Record 2	Record 4	Record 8
Online SOM Epoch 1	7338	32473	82805	183544
Online SOM Ephoch 5	119175	234998	486573	990348
変形 Batch SOM Epoch 1	8295	30517	75126	164094
変形 Batch SOM Epoch 5	125161	225315	448385	898130

表 4.3 実験結果（PE8)

また，この実験結果を基に，変形 Batch SOM が Online SOM に対してどれほど有効であるかを示すためにスピードアップ率を算出した．この値は，各サイクル数の割合を使ったもので，算出式はこのようになる．

$$SpeedUp(\%) = (1 - \frac{\text{変形 } BatchSOM \text{ サイクル数}}{OnlineSOM \text{ サイクル数}}) \times 100 \tag{4.1}$$

式 4.1 を用い，表 4.2・4.3 の結果をスピードアップ率であらわしたものが図 4.1 のグラフである．

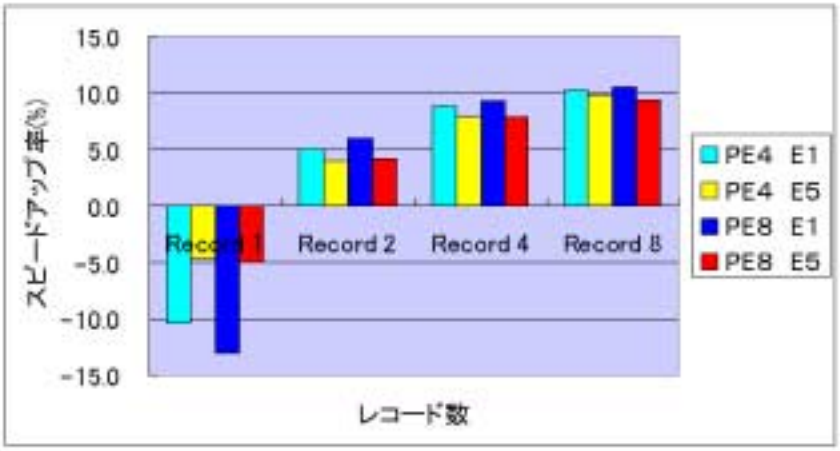

図 4.1 Online SOM に対する変形 Batch SOM のスピードアップ率

4.3 考察

実験の結果から，レコード数が増えるにしたがって変形 Batch SOM のサイクル数が Online SOM に比べて少なく，スピードアップ率の推移もレコード数の増加によって効率が上がっているのがわかる．これは，PE が 4・8 のどちらにも当てはまる．しかしながらレコードが 1 の場合に限っては逆に Batch SOM のほうがサイクル数がかかり，スピードアップ率はマイナスを示している．これはレコードが 1 つの場合，勝利するユニットも 1 つであるので Online SOM の場合は更新するユニットが 1 つで済むのに対して，変形 Batch SOM では勝利者が 1 つの場合でもすべてのユニットに対して勝利回数の調査，更新を行わなければならないためであると考えられる．またエポックが 5 の場合は，Onlien SOM より Batch SOM の方がサイクル数が少ないものの，エポックが 1 の時に比べてスピードアップ率は僅かではあるが下がっている．また，PE4・8 の場合にあまり目立った差がなかった事から，更新時，またはその他の処理における各 PE への処理の割り当て，つまり並列化が効率良く行われていなかった事が，現時点で考えられる．

第 5 章

まとめ

今回，Batch SOM を DDMP で実装することによって，Online SOM に対して Batch SOM の有効性が実験結果より確認された．しかし，今回の実装はあらかじめ設定したレコード・エポック数の処理時間を比較したものであり，実際 SOM として実用化する際には更に収束率・学習結果なども視野に入れて比較を行わなければならない．また，今回実験に使用したシミュレータの制限等からあまり大きなパケットを使用することができなかったが，実際に SOM として活用することを考えればもっと大量のパケットで実験を行わなくてはならない．

しかしながら，本研究で提案した変形 Batch SOM は割り算の部分を簡略化できたということ，それにより DDMP での Online SOM と比較実験が可能になった事は無駄ではないと考える．DDMP 以外の他のプロセッサ，もしくはプログラムでも変形 Batch SOM を実装・比較すること，また DDMP に関しても効率の良い並列化で実装することなどが今後の課題として挙げられる．

謝辞

今回，この変形 Batch SOM を DDMP に実装するにあたってたくさんの方々の御助力を頂き，またご迷惑をおかけしました．また，この論文製作に当たっては株式会社 Xanagen 様の HP から画像をお借りするなど，予想以上に多くの人を巻き込んでしまいました．お忙しい中，転載許可の返事を下さった Xanagen 社様にも感謝しています．指導教員の Ruck 先生をはじめ，フローグラフ製作時には同研究室，大学院生の福永さん，岩田先生，岩田研究室の三宮さんはじめたくさんの人たち，その他，研究での鬱憤晴らしに付き合っていただいたりと，ここには書ききれないほどの友人達．気分転換に癒してくれた車とか．この場を借りて感謝したいと思います．

ありがとう．

参考文献

[1] T. コホネン著，”自己組織化マップ”

特高平蔵／岸田悟／藤村喜久朗 訳，ISBN4-431-70700-X

[2] 岩田誠，宮田宗一，寺田浩詔

”自己タイミング・スーパーパイプライン型データ駆動プロセッサ”

電子情報通信学会論文誌 D-1, VOL.J81, No.2, pp.62-69(Feb.1998)

[3] Amalendu Roy

”A Survey on Data Clustering Using Self-Organizing Maps”

http://www.cs.ndsu.nodak.edu/~amroy/courses.html

[4] 阿部貴志, 金谷重彦, 工藤喜弘, 木ノ内誠, 大平賢

”学習順に依存しない自己組織化法の開発”

情報科学討論会要旨集., 1999., pp.146-147.

[5] 福永諭，Ruck THAWONMAS，岩田誠

”DDMP による SOM の並列化に関する研究”

高速信号処理応用技術学会春季研究会, pp.70-71, 2001 年 3 月.