平成 14 年度

修士学位論文

SOM の並列学習と
そのデータ駆動型プロセッサへの
効率的な実装法

Parallel Implementation of Self-Organizing Map on a Data-Driven Processor

1055114　　福永　諭

指導教員　　岩田　誠

2003 年 1 月 31 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報システム工学コース

要 旨

SOM の並列学習と
そのデータ駆動型プロセッサへの
効率的な実装法

福永　諭

自己組織化マップ（Self-Organizing Map:SOM）はデータ分析の分野においてよく知られている手法である．しかし，ユニットの数が増えるにつれて，計算量が膨大になるため，並列処理による高速化が求められる．一般的な並列手法としては競合層分割法が挙げられるが，これは更新時の負荷が不均一になるという欠点を抱える．この問題を解決したものが，入力層分割法である．しかし，入力層の数だけ PE を準備する必要のあるこの手法は，実装時にプログラムの規模が大規模になってしまう．本研究では既に入力層分割法を発展させ，プログラム規模が大きくなり過ぎない一般化入力層分割法（Generalized Input layer Parallel Model:GIPM）を開発している．本論文では，この GIPM を採用した SOM（以降 GIPM-SOM）と，同様にして競合層を一般化した，一般化競合層分割法（以降 GCPM-SOM）をデータ駆動型プロセッサである DDMP（Data Driven Media Processor）上に実装し比較した．その結果，GIPM-SOM の方が，処理時間及び並列処理性能において，高性能であることを明らかにした．

キーワード　自己組織化マップ，SOM，データ駆動型プロセッサ，並列処理，一般化入力層分割法

Abstract

Parallel Implementation of Self-Organizing Map on a Data-Driven Processor

Satoshi Fukunaga

The Self-Organizing Map(SOM) is a popular tool for data analysis. However, the execution time of the training phase is long for many application domains that either deal with very large data sets or require real-time responses. In order to resolve these problems, many researchers have developed the parallel SOM algorithm. Popular parallel SOM algorithm is used Competitive layer Parallel Model(CPM). But CPM algorithm make load imbalance among PEs occurs. To solve this problem the Input layer Parallel Model(IPM) was proposed. Then we present a novel scheme consisting of a new parallel training algorithm Generalized Input layer Parallel Model(GIPM). In this thesis, GIPM algorithm and Generalized Competitive layer Parallel Model(GCPM) algorithm implementation on a data-driven multiprocessor system, and compare its performance. As a result, it was shown that GIPM is high-speed and high performance more than GCPM.

key words SOM, Parallel Algorithms, Data Driven Processor, Generalized Input layer Parallel Model

目次

図目次

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-Organizing Map,SOM）は，クラスタリング等の様々な分野での応用が提案されている [1]．しかしながら，SOM は入力層，競合層の 2 層を用いており，各層間は完全結合で結ばれるため，ユニット数を増やすと計算量の爆発が起こる．そのため，並列処理による高速化手法が提案されている．しかしながら，従来の並列学習法は，競合層を分割して各プロセッシングエレメント（PE）に割り当てる競合層分割法を基本としており，SOM の学習アルゴリズムが持つ特性のために学習後半で PE の負荷が不均一となり効率的ではない．この問題を解決するために入力層分割法 [2][3] という手法が考案されている．この手法は入力層を分割して各 PE に割り当てるために，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．これにより負荷の不均一が発生しない．しかし，入力層分割法は，PE 数に比例してプログラム規模が大きくなってしまうという問題を抱える．我々は既に入力層分割法を発展させた，一般化入力層分割法（GIPM）を考案している [4][5]．この手法は，複数の入力層をひとまとまりとして PE に割り当てることで，プログラムの大規模化を防いでいる．本研究では，GIPM を採用した GIPM-SOM と，競合層の分割を GIPM 同様一般化した GCPM-SOM の両アルゴリズムを，データ駆動プロセッサである 32bitDDMP[6] へ実装し，GIPM-SOM の有効性を示す．

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，自己組織化によるクラスタリングを行うアルゴリズムである．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へ効果的に表現する．このことから，ユーザは各データの関係を容易に見ることが可能となる．それだけでなく，これまでユーザの考え付かなかった潜在的な相関をも得ることができるかもしれない．この様な特徴を利用し，SOM は画像・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，新しい未知の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 アルゴリズム

最も基本的な SOM アルゴリズムは Online-SOM と呼ばれ，学習には主に「入力の提示」,「勝利者・近傍の決定」,「重みの更新」の三つの基本動作が存在する．マップの生成には入力層と競合層の 2 層を用い，二つの層のニューロン同士はそれぞれ完全結合で結ばれている．

また，層間の結合には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されている．入力層ニューロンは入力データの特徴と同じn次元，競合層は出力を視覚的に見せるため 2 次元に配列され，入力層の次元に合わせn個の要素を持っている．

アルゴリズムの最初のステップでは，入力データを入力層に提示し，競合層の各ニューロンの重みとの距離をとり，この距離が最小となる競合層ニューロンを勝利者とする（式 2.1）．距離測度には一般的にはユークリッド距離を用いるが，目的によって，ベクトル間の角度を問題にする場合には方向余弦，コード表現の一致度を比較する場合にはハミング距離，記号列間の距離を測るのにはレーベンシュタイン距離，実験心理学で用いる都市ブロック距離，文書間の関連性を表すタニモトの測度，ファジィ論理で用いられるルカシービッチ測度など，必要な情報を引き出すために様々な距離測度が考案されている．最も一般的な距離測度である，ユークリッド距離を用いた SOM アルゴリズムを下記の式で示す．$\boldsymbol{x}(t)$は入力データ，$\boldsymbol{w}_i(t)$はi番目の競合層ニューロンの重みとする．

$$c(t) = \arg\min_i\{||\boldsymbol{x}(t)-\boldsymbol{w}_i(t)||\} \tag{2.1}$$

次に，式（(2.2）を使い，ステップtにおける重みを$\boldsymbol{x}(t)$に近づくように更新する．ただし，$h_c(t)$は勝利ニューロン c の学習率係数である．

$$\boldsymbol{w}_c(t+1) = \boldsymbol{w}_c(t) + h_c(t)(\boldsymbol{x}(t)-\boldsymbol{w}_c(t)) \tag{2.2}$$

このとき，勝利者の周囲には近傍領域が発生し，この近傍領域に含まれるユニットの重みも準勝利者として，勝利者に比べ更新値は少ないものの式（2.3）を用い更新される．

$$\boldsymbol{w}_i(t+1) = \boldsymbol{w}_i(t)+h_{ci}(t)(\boldsymbol{x}(t)-\boldsymbol{w}_i(t)) \tag{2.3}$$

ここで$h_{ci}(t)$は近傍関数と呼ばれ，式（2.4）によって定義される．$t = 0, 1, 2, ...$は離散時間座標である．

$$h_{ci}(t) = h(||\boldsymbol{r_c} - \boldsymbol{r_i}||, t) \tag{2.4}$$

ここで，$\boldsymbol{r_c}$，$\boldsymbol{r_i}$はそれぞれ競合層におけるユニットcとiの位置ベクトルであり，$||\boldsymbol{r_c}-\boldsymbol{r_i}||$が増加するにつれて$h_{ci} \to 0$ とする．これは，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ことを意味する．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．ノードcの周りのユニットを近傍集合とし，N_cと定義する（これによって時間の関数としての$N_c = N_c(t)$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 2.5）．α_0は，αの初期値であり，一般的に 0.2〜0.5 の値を選ぶ．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．

$$\alpha(t) = \alpha_0(1 - \frac{t}{T}) \tag{2.5}$$

また，近傍領域$N_c = N_c(t)$についても同様に，

$$N_c(t) = N_c(0)(1 - \frac{t}{T}) \tag{2.6}$$

と表すことが出来る．$N_c(0)$ は初期値である．これまでの一連の作業を，図（2.1）に図示する．Online-SOM アルゴリズムにおいて，一つのデータを提示し，処理を終了するまでの処理単位を 1 Record，全てのデータについての処理を終えるまでの処理単位を 1 Epoch とする．更新時に顕著な変化がなくなるまで Epoch を繰り返すことで，各競合層ニューロンが持つ重みを参照ベクトルとする空間が形成される．図（2.2）に，作成されたマップの例を示す．

2.2　SOM アルゴリズム

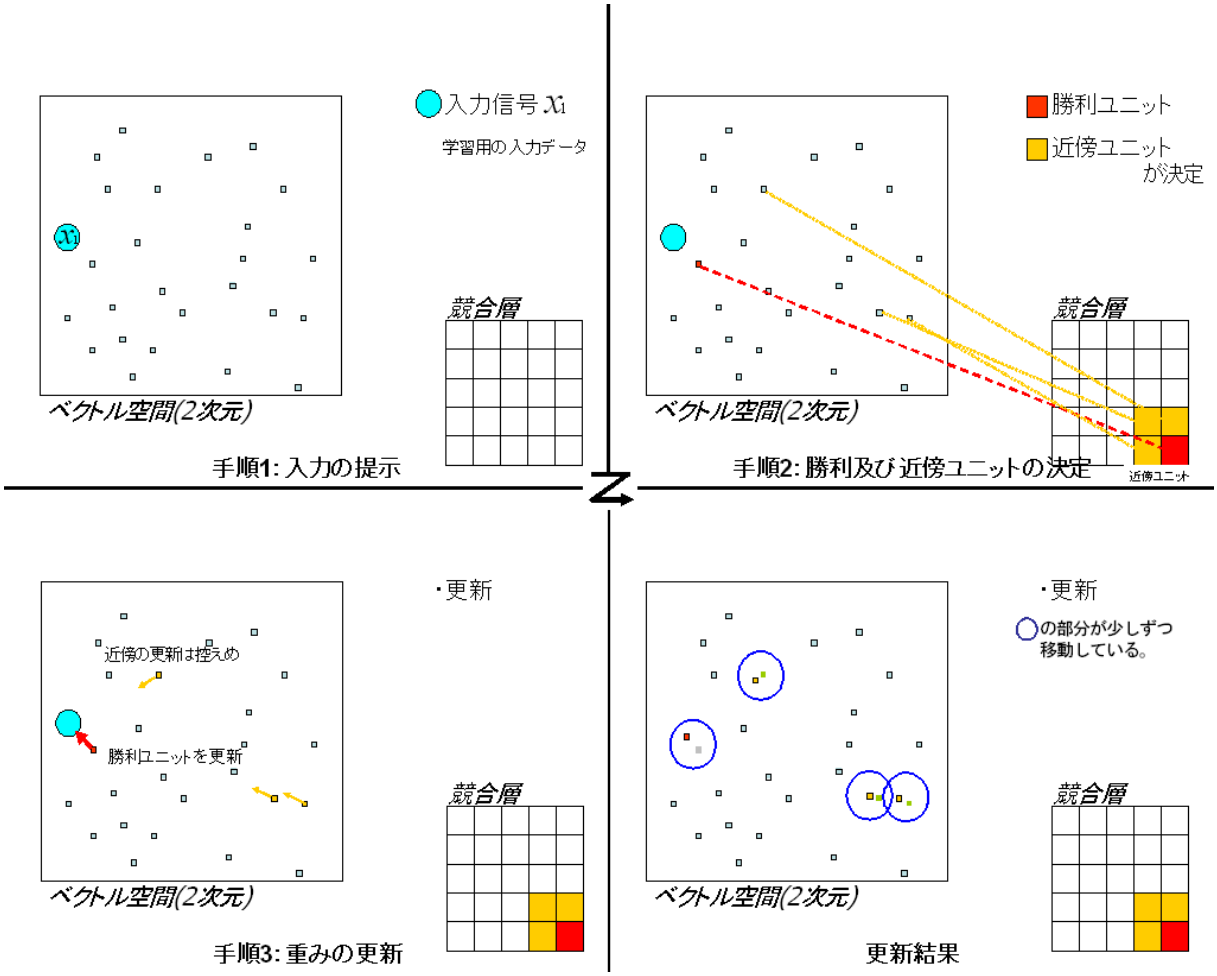

図 2.1　SOM アルゴリズムの手順

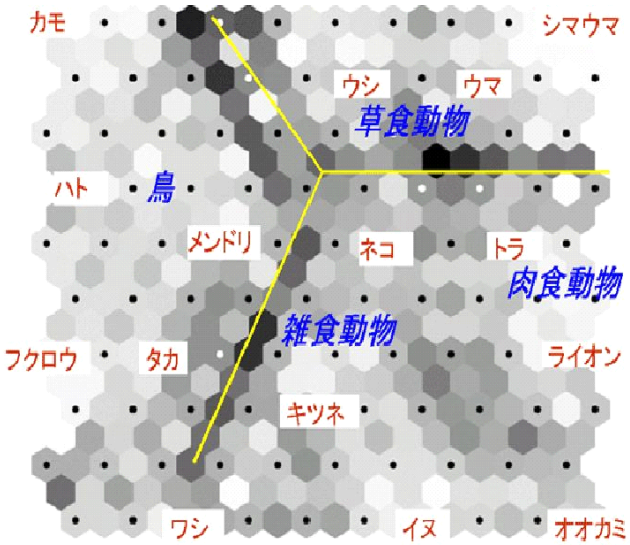

図 2.2　SOM によって生成されるマップ

第 3 章

並列 SOM

SOM は入力層と競合層の各ニューロンが完全結合で結ばれているため，大容量データを処理しようとした場合，計算量が膨大になり学習に時間がかかり過ぎるという問題がある．この問題を解決するために並列処理による高速化手法が提案されている．

3.1 競合層分割法

競合層分割法 [2] は文字通り，競合層を分割し並列化する手法である．この並列学習法は競合層を分割して各 PE に割り当てる．しかし，この手法は SOM アルゴリズムの特性上，更新時に各 PE に負荷の不均一が発生し，効率的ではない．なぜならば，更新は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，他の PE はアイドル状態で処理の終了を待つことになるからである（図 3.1）．

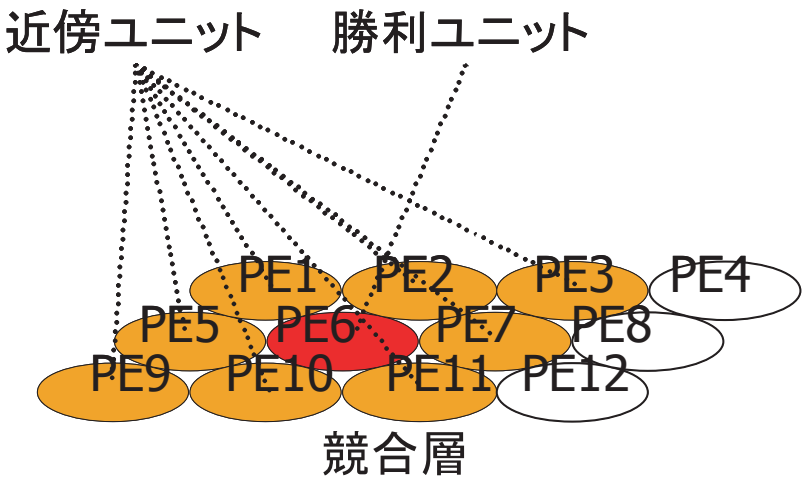

図 3.1　競合層分割法の PE 割り当て

3.2　入力層分割法

これまで行われた並列手法である競合層分割法による並列学習では，PE 間の負荷が不均一であったり，画像などの高次元入力では十分な高速化が行えないなどの問題が存在する．この問題を解決するために，PE 間の負荷の不均一が原理的に発生しない入力層分割による並列化が考案されている [2][3]．入力層分割法は，入力データの次元数 n 個分の入力層ユニットを持つ．そして，PE は入力層のユニット一つを担当し，競合層の各ユニットへの重みを持つ（図 3.2）．したがって，競合層のどのユニットがウィナー，あるいは近傍となって重みの更新を行っても，すべての PE が同等の処理をする為に各 PE の負荷は均一になる（図 3.3）．

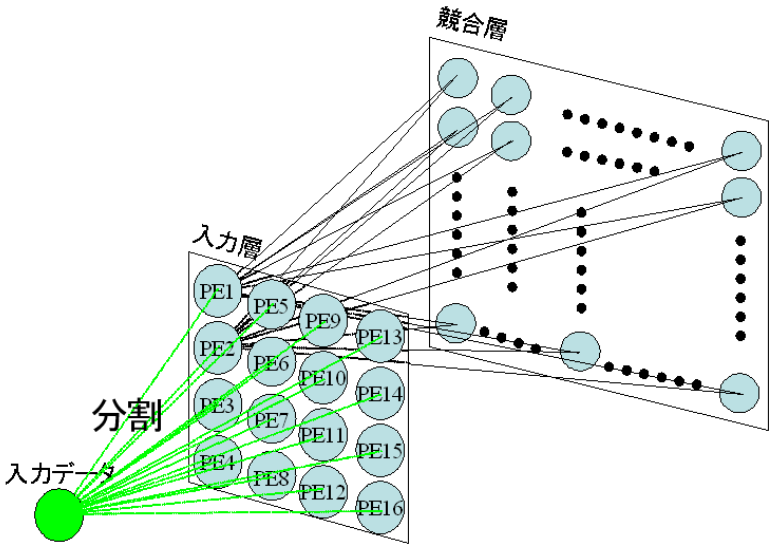

図 3.2　入力層分割法の概要

入力層分割法のアルゴリズムを示すにあたって対象とする SOM は入力層，競合層の 2 層を，それぞれ，$M \times M$，$N \times N$ 個のユニット数からなるとする．ある時刻 t における入力ユニット i と出力ユニット j 間の重みを$\boldsymbol{W}_i^j (i = 1, \cdots, M^2, j = 1, \cdots, N^2)$ とする．この SOM を $M \times M$ 個の PE に分割し，各 PE は N^2 次元の重みベクトルを持つ．学習パターン $R = \{t_1, \cdots, t_{[}M^2]\}$ が入力層ユニットに入力された場合を考える．各 PE は学習パターンで対応する要素を用い，入力ベクトル要素 t_i と重みベクトル W_i^j との間のユークリッド

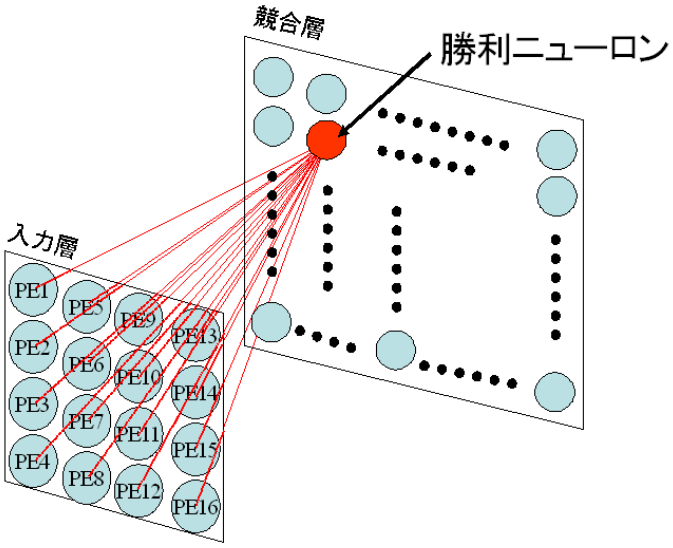

図 3.3　入力層分割法の更新処理

距離 d_i^j を式（3.1）に基づき計算する．

$$d_i^j = \sqrt{(t_i - W_i^j)^2} \qquad j = 1, \cdots, N^2 \tag{3.1}$$

各 PE_i が d_i^j を計算した後で，i について d_i^j を式（3.2）により，集計し，結果を全 PE にブロードキャストする．

$$d^j = \sum_{i=1}^{M^2} d_i^j \tag{3.2}$$

その結果，競合層ユニット j と学習パターン間の距離 d^j をすべての PE が持つことになる．各 PE は独立に d^j が最も小さいものをウィナー c として選び，式（3.3）に表される近傍関数 h_{cj} を用いて c とその近傍 $N_c(t)$ の重みを更新する．

$$h_{ci} = \begin{cases} \alpha(t) & i \in N_c \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

したがって，各 PE は同一個数の重み要素を更新することになり，負荷の不均一は原理的に発生しないことになる．

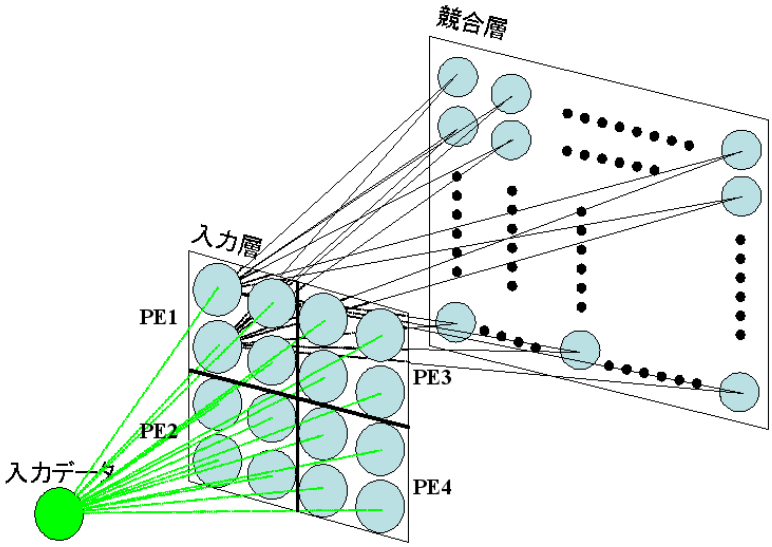

図 3.4　GIPM の概念図

3.3　一般化入力層分割法

これまで，並列手法として競合層分割法と，入力層分割法について述べてきた．入力層分割法は PE の負荷が均一になるという長所を持つが，入力データの次元が増えるに比例して PE 数が増えるため，プログラムの規模が大きくなってしまうという問題を抱える．そこで我々は一般化入力層分割法（Generalized Input layer Parallel Model:GIPM）を開発した．図（3.4）に GIPM の概念を示す．それぞれの PE は複数の入力層ニューロンをひとまとまりとし，それぞれの入力層ニューロンは全ての競合層のニューロンへの重みを持つ．学習フェイズではまず初めに，複数の次元ごとに分割された入力データのベクトルと，それに対応する競合層ニューロンの重みベクトルとの部分距離を計算する．その後，得られた値を HPE（Host Processing Element）と呼ばれる PE に送る．HPE が入力ベクトルと，それぞれの競合層ニューロンの重みベクトルとの総合距離を計算した後，それらを各 PE に送る．PE が総合距離を受け取れば，勝利者およびその近傍が決定され，対応する競合層の重みが更新される．

ここで入力層 M 個および競合層 N 個のニューロンを持っている Online-SOM を考慮し，GIPM のアルゴリズムを示す．P 個の PE と，1 つの HPE を備えたマルチプロセッサー・

システム上にこの SOM を実装しようとする．一般性を失わずに M を P で均等に分割出来ると仮定し，そのために k 番目の PE は $I = M/P$ 個の入力層ニューロンを担当する．すなわち，ステップ s で入力ベクトル$\boldsymbol{x}(s) =\{\boldsymbol{x}_1(s),\boldsymbol{x}_2(s),\ldots,\boldsymbol{x}_M(s)\}^T$が与えられ，$k$ 番目の PE が分割された入力ベクトル

$$\begin{aligned}\boldsymbol{x}^k(s) &= \{\boldsymbol{x}_1^k(s),\boldsymbol{x}_2^k(s),\ldots,\boldsymbol{x}_I^k(s)\}^T \\ &= \{\boldsymbol{x}_{(k-1)I+1}(s),\boldsymbol{x}_{(k-1)I+2}(s),\ldots,\boldsymbol{x}_{kI}(s)\}^T\end{aligned}$$

の処理を担当する．肩文字の T は行列の転置を表す.

ステップsにおいて$\boldsymbol{w}_{ij}^k(s)$は，k番目の PE でのi番目の入力層ニューロンとj番目の競合層ニューロンとの間のスカラー重みを示す．ここで，$i \in \{1,2,\ldots,I\}$，$j \in \{1,2,\ldots,N\}$である．アルゴリズムを実行する前に全ての重みは初期化される．分割された入力ベクトルとk番目の PE によって処理される重みベクトルとの距離は以下の式によって計算される:

$$d_{ij}^k(s) =|\boldsymbol{x}_i^k(s)-\boldsymbol{w}_{ij}^k(s)\,|\,. \tag{3.4}$$

GIPM の学習アルゴリズムは下の式で表される:

Step 1　k 番目の PE が部分距離を合計する

$$d_j^k = \sum_{i=1}^{I} d_{ij}^k(t)$$

そしてその値を HPE に送信する．ここで，$j \in \{1,2,\ldots,N\}$.

Step 2　HPE での計算は以下の式を用いて行われ

$$d_j(t) = \sum_{k=1}^{P} d_j^k(t)$$

得られた値を全ての PE に送信する．ここで，$j \in \{1,2,\ldots,N\}$.

Step 3　k 番目の PE では勝利者を計算し，

$$c = \arg\min d_j(t)$$

勝利者とその近傍 $N_c(t)$ の重みを，次の式を使って更新する．

$$w_{ij}^k(t+1) = w_{ij}^k(t) + h_j(t)[x_i^k(t) - w_{ij}^k(t)],$$

$$h_j(t) = \begin{cases} \alpha(t), & \text{if } j \in N_c(t) \\ 0, & \text{otherwise,} \end{cases}$$

$\alpha(t)$ はステップ t における学習率係数である．

Step 4　学習率係数 α を徐々に減らしていき，Step1 からの作業を繰り返す．

GIPM を用いることにより，IPM で問題となっていた，PE 数が入力の次元数に比例して増加するプログラムの大規模化問題が解決するのである．

GIPM を用いることにより，入力データの次元数に比例していて増えていた PE 数を抑えることができる．その結果、プログラム全体の規模を GIPM は複数の入力層をひとまとまりとし PE に割り当てることで，処理に必要とする PE 数の増加を抑えることができ，その結果プログラムの規模を抑えることができる．

3.4　一般化競合層分割法

本研究では，入力層分割法を元にして最適化を行った GIPM の性能が，従来の競合層分割法に比べて有効であるかを確認する．そのために，GIPM と同様に，PE 一つに複数の競合層を割り当てる，一般化競合層分割法（Generalized Competitive layer Parallel Model:GCPM）を考案した（図 3.5）．

先程と同じように入力層 M 個および競合層 N 個のニューロンを持っている Online-SOM を考慮し，P 個の PE と，1 つの HPE を備えたマルチプロセッサー・システム上にこの GCPM を実装しようとする．一般性を失わずに N を P で均等に分割出来ると仮定し，そのために k 番目の PE は $C = N/P$ 個の競合層ニューロンを担当する．この場合の GCPM の学習アルゴリズムを，以下に示す．

Step 1　k 番目の PE が担当する N についての重みベクトルを計算．

$$d_j = \sqrt{(t - W_j)^2} \qquad j = 1, \cdots, N^2$$

計算後，その値を HPE に送信する．

Step 2　HPE は受け取った値を全ての PE に送信

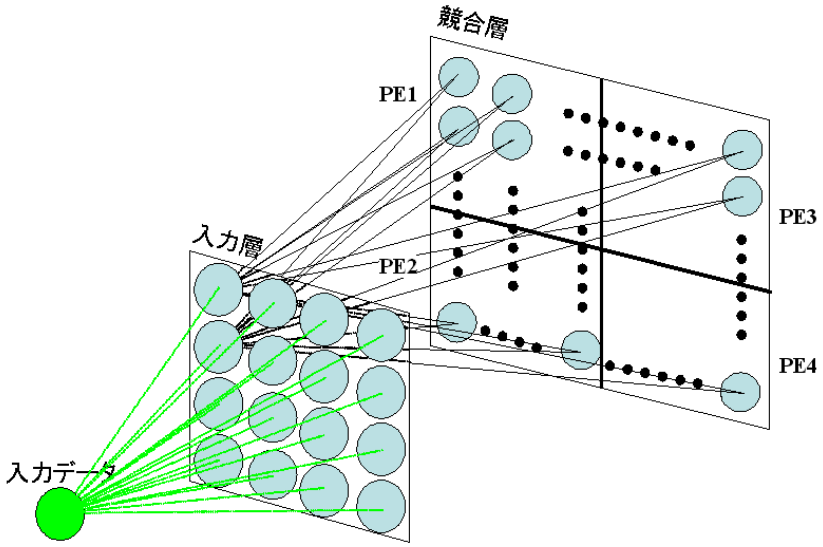

図 3.5　GCPM の概念図

Step 3　k 番目の PE では勝利者を計算し，

$$c = \arg\min d_j(t)$$

勝者ニューロン c に対応した PE が，勝利者とその近傍 $N_c(t)$ の重みを，次の式を使って更新する．

$$W_j(t+1) = W_j(t) + h_j(t)[x_j(t) - W_j(t)],$$

$$h_j(t) = \begin{cases} \alpha(t), & \text{if } j \in N_c(t) \\ 0, & \text{otherwise,} \end{cases}$$

$\alpha(t)$ はステップ t における学習率係数である．

Step 4　学習率係数 α を徐々に減らしていき，Step1 からの作業を繰り返す．

第 4 章

データ駆動型プロセッサ

4.1 データ駆動型プロセッサとは

本研究では SOM の実装に，データ駆動型プロセッサである 32bit DDMP を使用した．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，各データが独自に待ち合わせを行い，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ることにある（図 4.1）．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．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のである．これは，処理対象となるデータがパケットの形式をとり，そのタグ部分にデータの行き先や，識別情報を持つことにより，可能となるものである．この特徴により，従来のノイマン型プロセッサでオーバーヘッドとなっていたスケジューリングが全く不要となる．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

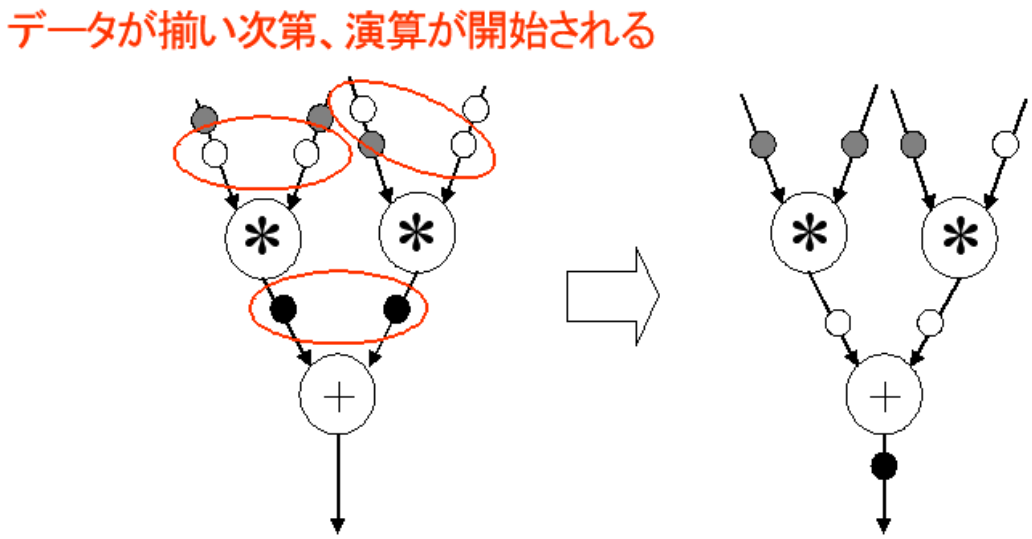

図 4.1　データ駆動型処理の例

なり，電力対性能比の向上が達成されている．以下に，本研究で使用した 32bit DDMP の nPE 構成を示す．ALP は算術演算用プロセッサ，LCP は論理演算用プロセッサ，CVP は外付け SDRAM へのアクセス制御用プロセッサである．今回の実装では，32bitDDMP への SOM アルゴリズム実装の初段階として，SOM に必要な算術演算，論理演算全てを使用可能な ALP5 個を使い実装を行った．

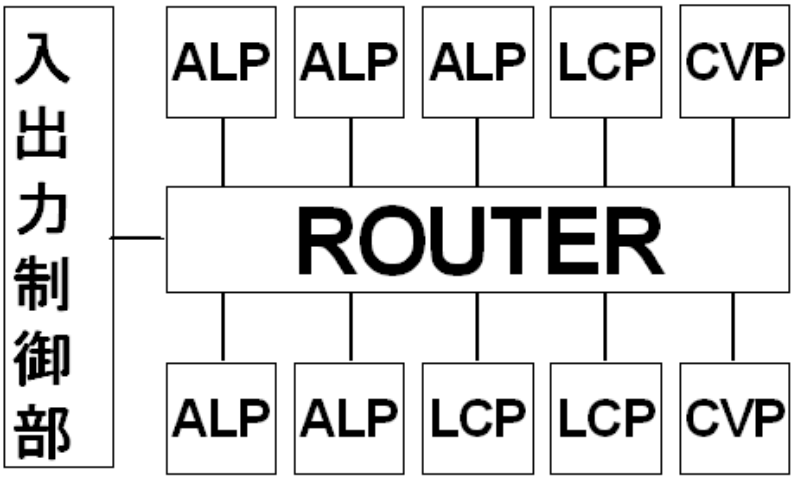

図 4.2　32bit DDMP の nPE 構成

4.2　SOM の実装について

32bit DDMP への実装には，データフローグラフを用い，図的にプログラミングを行うこととなる（図 4.3）．SOM の並列アルゴリズムに目を向けてみると，GIPM，GCPM 共に，各競合層と入力データとの距離計算及び，更新処理はニューロン毎に完全独立であり，並列処理を行うことが可能である．さらに，データ駆動型の特徴から，複数のデータが同一のプログラムを使用することが可能であることから，実装アルゴリズムに同一の処理機構を持つ複数の Line を設けた．3 章で述べた PE 一つが，1 Line となる．それぞれの Line には，Line 数で分割された入力層及び競合層に関するデータが投入され処理が行われる．SOM の処理手順については，データフローの観点から，Online-SOM の基本動作である「入力の提示」,「勝利者・近傍の決定」,「重みの更新」の 3 つの動作を,「距離測定」,「勝利者・近傍の決定」,「重みの更新」の 3 処理により実現した．このアルゴリズムを図示したものが，図 4.4 である．

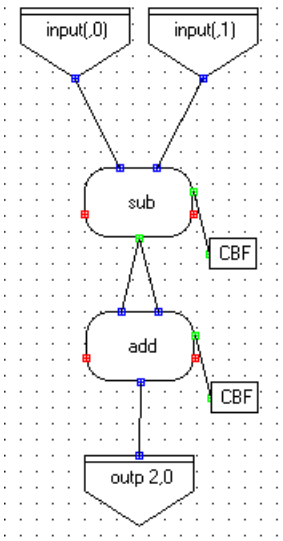

図 4.3　データフローグラフ

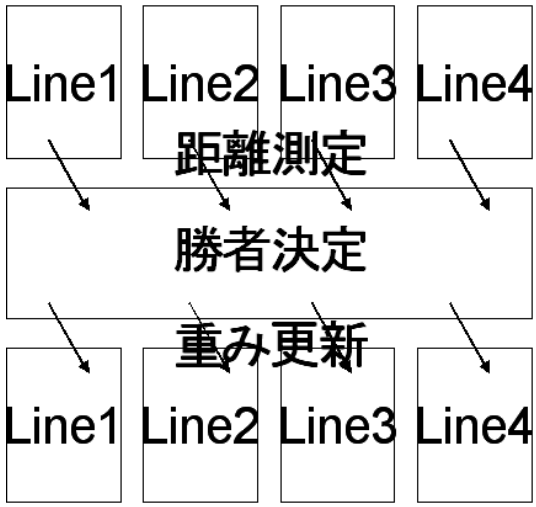

図 4.4　実装アルゴリズム

GIPM-SOM に関して言えば，距離計算部分では入力層毎に同時並列処理，競合層毎にパイプライン処理が行われることとなる．

第 5 章

実装

32bitDDMP 上に，一般化入力層分割法を適用した GIPM-SOM と，一般化競合層分割法を適用した GCPM-SOM を実装する．本研究ではプログラム作成から実験シミュレーションまでを，Windows 環境で動作する，32bitDDMP 総合開発ツール DADT（DDMP Application Designer's Toolkit）を用いて行った．

5.1 実装の準備

DADT を使用した，プログラムの作成からシミュレーションまでの手順を以下に示す．

Step1 フローグラフの作成 DADT の中のフローグラフエディタ機能を使い，プログラムを記述する．このとき，複数のノードをひとまとまりのモジュールとし，プログラムを階層的に記述することが可能である．

Step2 アセンブリ記述したプログラムを実行形式にアセンブリする

Step3 シミュレーション実行ファイルに変換されたプログラムと，投入する入力データパケットファイルを用い，DADT の中の DDMP Simulator を使用しシミュレーションを行う．

フローグラフは，幾つものノードを Arc（アーク）と呼ばれる矢印で接続し，データの流れを記述していく．記述に使用されるノードの種類と，その名称を図（5.1）にまとめる．図の 2 入力ノードの「2 変数入力」では，左右の入力が揃ってはじめて演算が開始され，片方のみの場合は，他方のデータが揃うまで待機する．また，2 入力ノードは，左右の入力のう

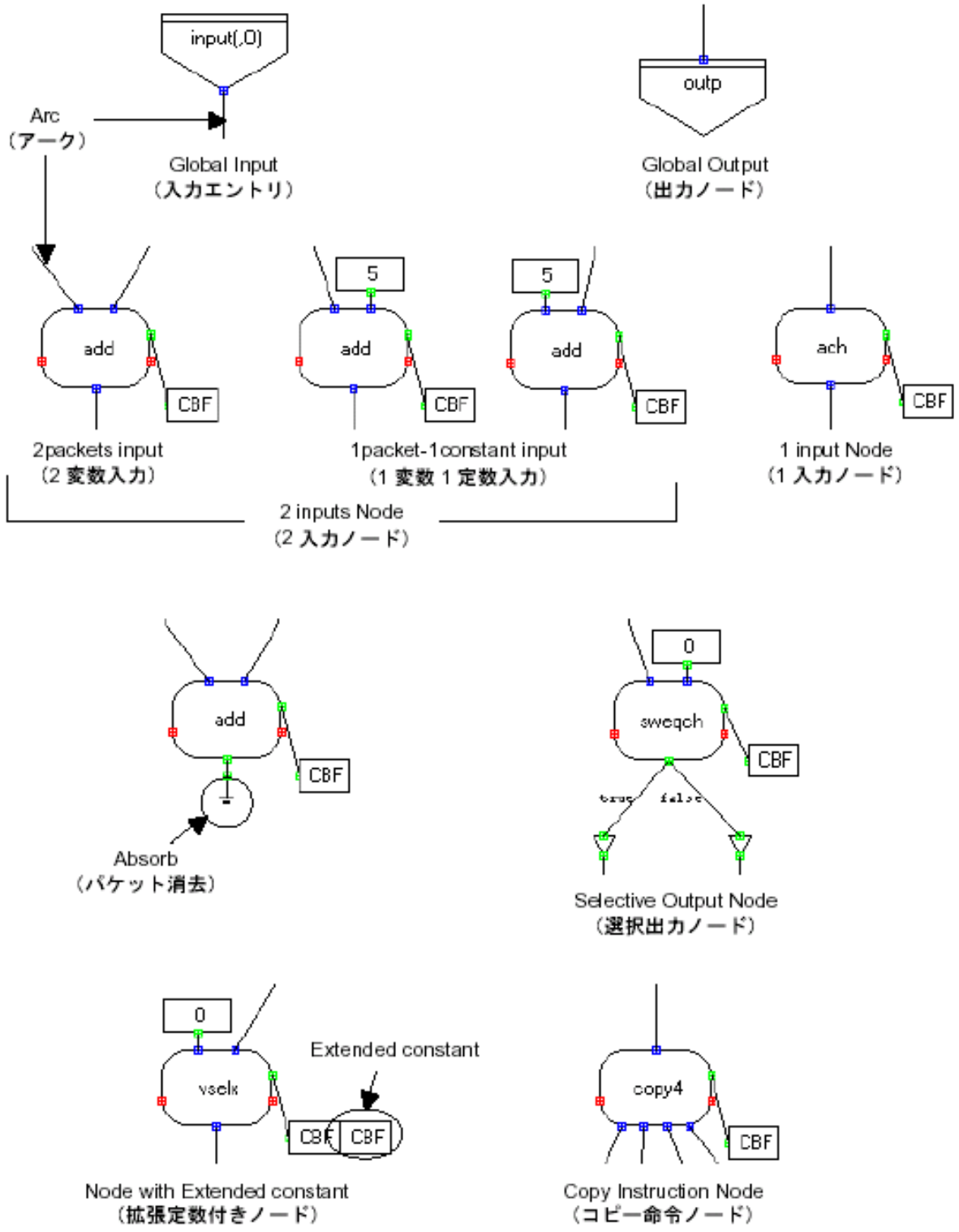

図 5.1　フローグラフに用いられる各部の名称

ちどちらかに定数を追加して記述することで,［1 変数 1 定数入力］となる．この場合，片方は定数として常に存在するので，他方のデータが到着し次第演算が開始される．

入力データパケットファイルは，図（5.3）に示されるようなテキスト形式のファイルである．このファイルに入力データの値と，データ識別タグである世代番号，どの入力エントリに投入するか等を記述し，シミュレーションを行う．入力データパケットファイルの数値記述は，10 進数もしくは 16 進数で行う必要がある．世代番号は，全部で 28bit の値からなり，BK#，FS#，CH#の 3 つの領域に論理的に分割される．本研究では，BK#を 2bit，FS#を 14bit，CH#を 12bit に分割する設定で実装を行った（図 5.2）．

最後に，表（5.1）に，プログラム開発及び実験に必要なファイルをまとめる．

図 5.2　世代番号 28bit の割り当て

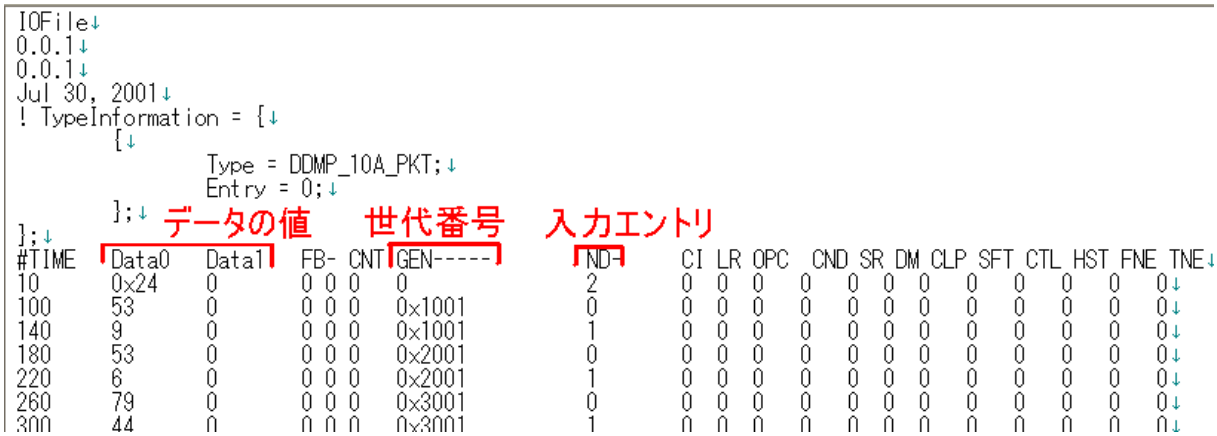

図 5.3　入力データパケットファイル

5.2　GIPM-SOM の実装

この節ではどの様なフローグラフ記述で GIPM-SOM を実装したかについて詳しく述べていく．この節で例にとりあげる GIPM-SOM プログラムは，4 Line 構成で，入力層 48 個，

拡張子	用途	使用箇所
*.dfg	プログラムの最上位階層を記述したファイル	Step 1
*.dfm	プログラム中に中に使われるモジュールを記述したファイル	Step 1
*.fs	プログラムのアセンブラソースファイル	Step 2
*.fe	プログラムオブジェクトファイル	Step 2
*.in	入力データパケットファイル	Step 3

表 5.1　実験で使用するファイル

競合層 36 個を持つこととする．

前述した通り，本研究では実装の際に使用する nPE は，5 つの ALP である．実装において，まず，ノードの演算内容を修飾する CBF 値と，メモリのアドレッシング領域についての宣言を行う（図 5.4）．IPPL 命令は内蔵メモリのアドレス計算において，1 つの FS#あたりの CH#の数を決める．よって，本プログラムでは，一つ目の入力パケットで，競合層数 N と同じ値を入れることで，5 つの ALP のメモリアドレスを最適化している．

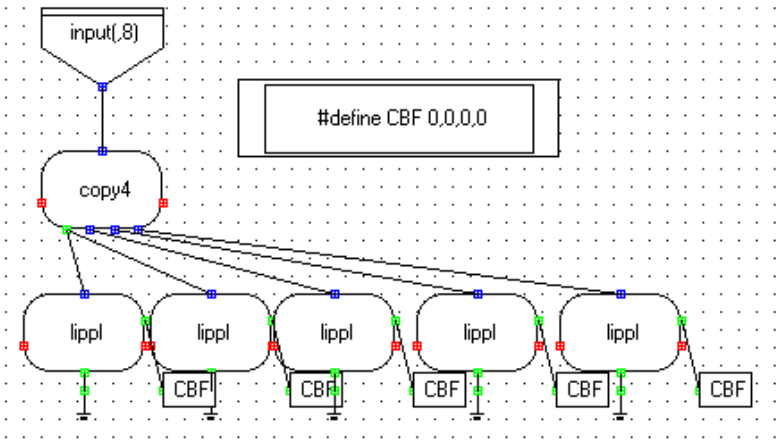

図 5.4　CBF 値とメモリアドレッシング領域の宣言部分

5.2.1　距離測定部分

フローグラフの最上段に位置する，データの入力部分は，4.2 節で述べていた Line 毎に二つの入力エントリを持つ（図 5.5）．この時，入力エントリの値は，0 から連番にする必要がある．偶数番の入力エントリには，各 Line が担当する入力層のデータが入り，奇数番

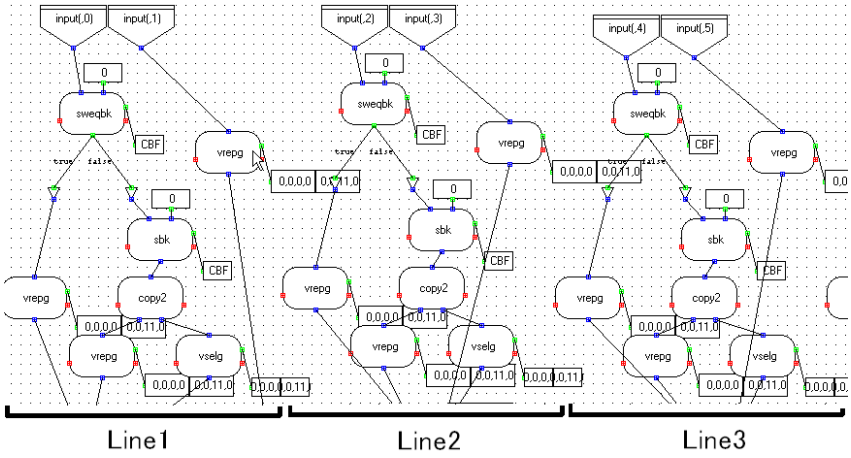

図 5.5　入力部分

の入力エントリには，そのデータに対応する競合層のデータが入る．入力データの世代番号は，FS#領域を入力層の識別に，CH#領域を競合層の識別に使用している．4 Line 構成の GIPM-SOM の場合，Line1 には，入力層 1〜12 個目と，全ての競合層 1〜36 個とを完全結合で結んだ世代番号を持つデータが投入される．投入されたデータは，後々の処理に備えて，vrepg 命令でメモリに格納される．メモリに格納されたデータは，その値と世代番号をそのままに，mod モジュールへと流れてゆく．2 Record 目のデータは，世代番号の BK#領域に 1 の値を持ち，偶数番号の入力エントリだけに投入される．その後，sweqbk 命令で分岐された後，1 レコード目で更新された競合層データを呼び出し，次の処理へと進むこととなる．

入力部分の処理を終えたデータは，図（5.6）の mod モジュールへと進む．モジュールを展開した物が，同図左側の部分である．mod モジュールでは，各ラインが自分の担当する入力層についての，入力層ユニットと競合層ユニットとの重みの値の差をとり，それらをすべて合計して出力する．本来ならば，ユークリッド距離を求めるのがベストであるが，処理が

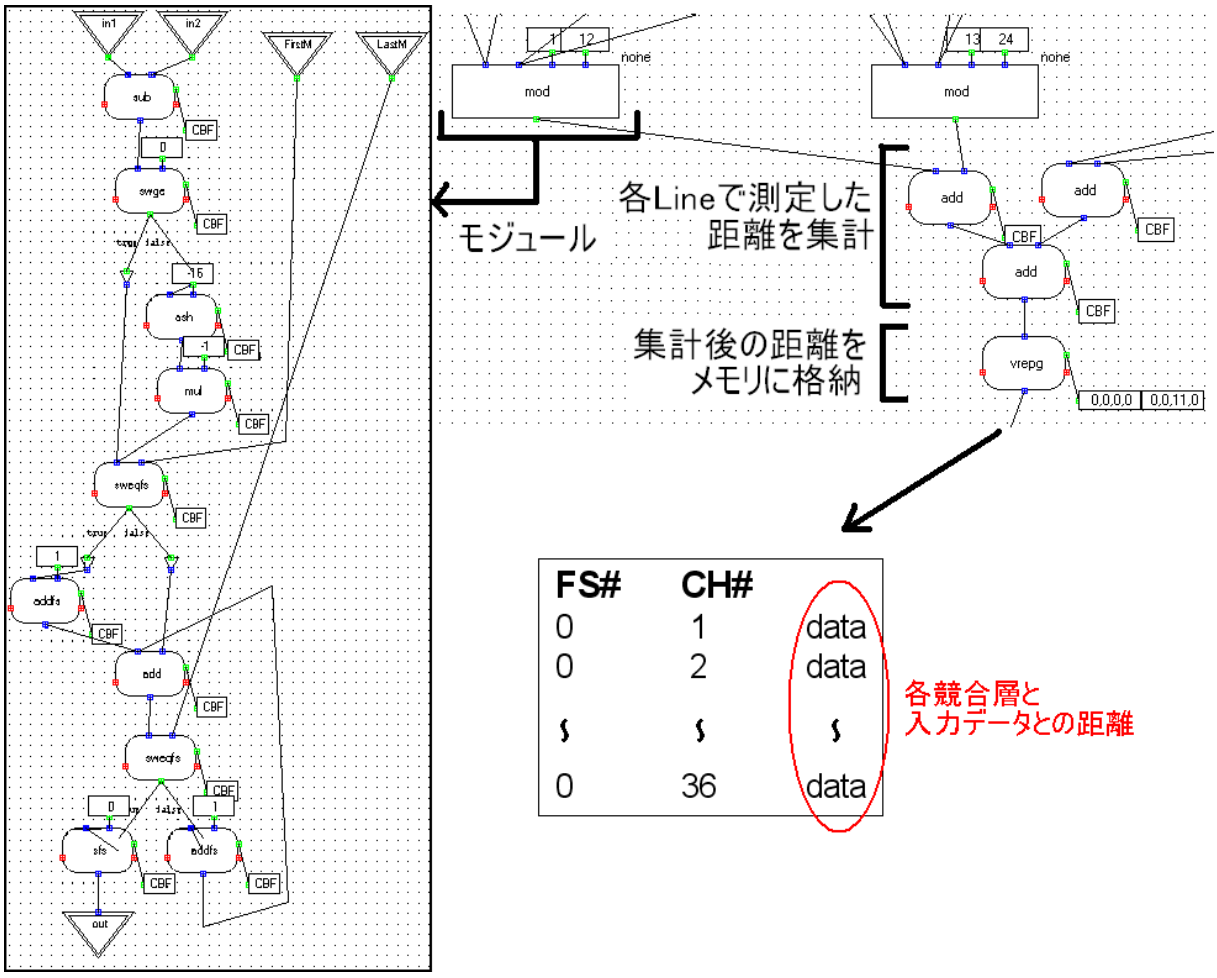

図 5.6　距離測定及び集計部分

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，ここでは単純にをそのままの値を比較している．また，この後の処理で，世代番号をあわせる必要があるため，mod から出力される FS#の値を 0 に統一している．こうすることで，各 Line の mod モジュールから流れてきたデータの集計が可能となり，各競合層データそれぞれと，投入された入力データとの距離が算出される．そして，その値を vrepg 命令でメモリに格納する．ここでメモリに格納されるデータ数は，競合層数 N 個となる．ここまでの処理が，図（4.4）の距離測定部分になる．

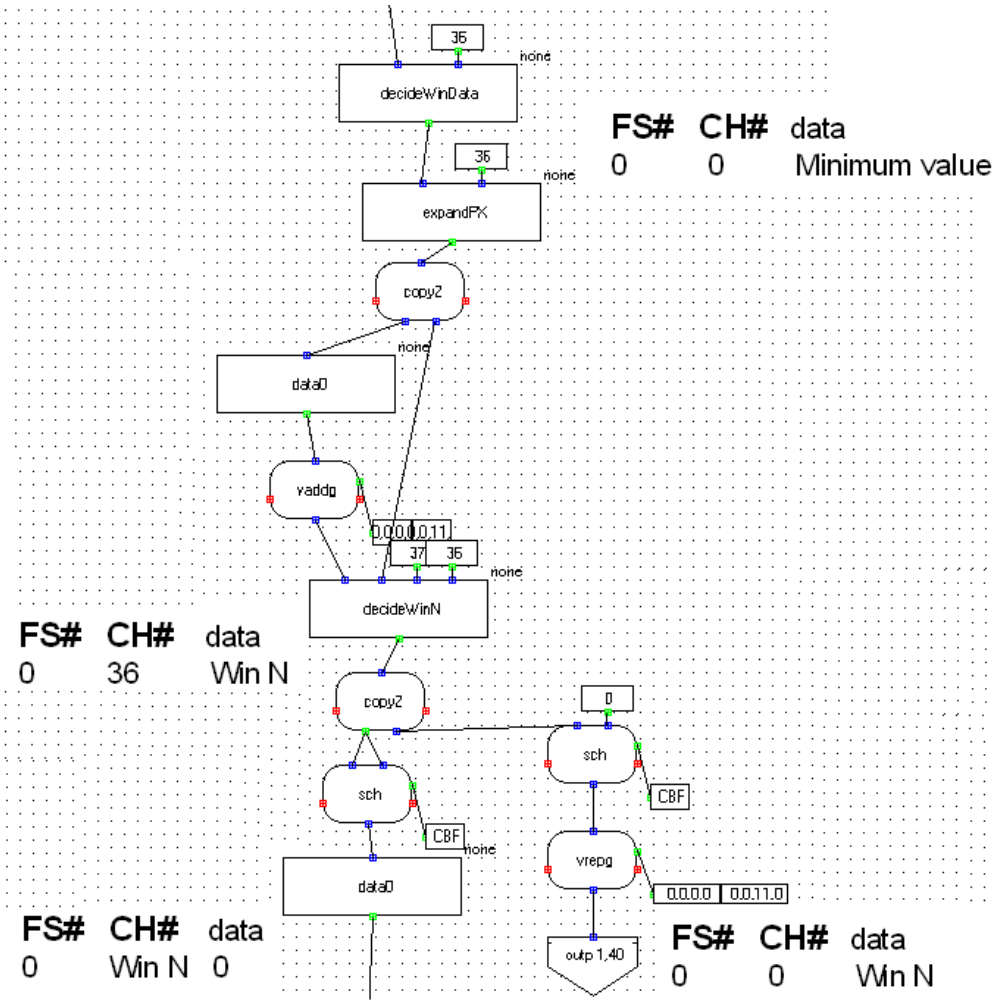

図 5.7　勝利者決定部分

5.2.2　勝利ユニット 決定部分

次に，各競合層の合計距離を元に，勝利ユニットの決定を行う（図 5.7）．図中の Win N とは勝利ユニットとなった競合層の CH#番号である．集計を終えたデータは，decideWindata（図 5.8）へ通され，そこで最小値，つまり最も入力データに近い競合層の値が決まる．そして，expandPX によって CH#の値を 1〜36 まで増やされた後に，copy 命令でデータフローを複線にする．一方は次の処理モジュールである decidewinN（図 5.9）へ，もう一方は，data0（5.10）モジュールで値を 0 にした後，先ほど格納した，競合層ユニット毎の集計データをメモリへ取りに行く．decidewinN モジュールでは，集計された値と，decideWindata

によって求められた最小値を比較することによって，どの競合層が勝利ユニットになったかを明らかにする．勝利ユニットの決定には，最小値の決定→その値を持つ競合層ユニットの決定と，手順を踏むこととなる．また，decidewinN 直後の分岐により，勝利ユニットの競合層番号 CH#を値に持つデータパケットを，メモリに格納する．これは，勝利ユニットの CH#を元に，近傍の決定を行うためである．

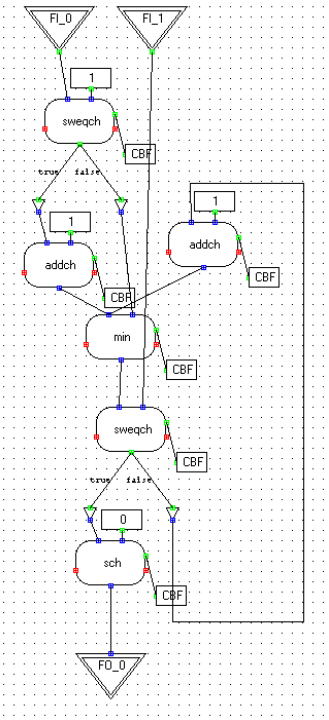

図 5.8　decidewindata モジュール

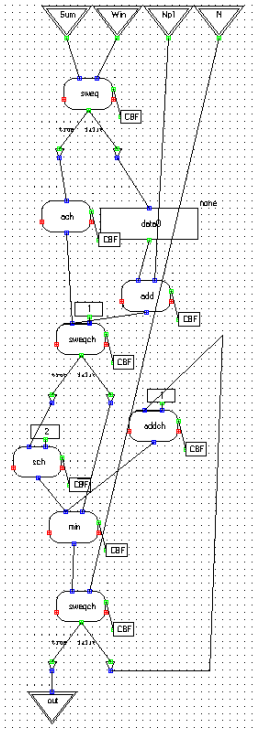

図 5.9　decidewinN モジュール

5.2.3　重み更新部分

勝利ユニットが決定した後は，勝利ユニット及び，その近傍にあたる競合層の重みの更新を行う．図（5.11）に更新に用いるアルゴリズムを示す．それぞれの Line に流れる勝利ユニットの競合層番号を持ったデータは，LNtoMax モジュールにて，担当する入力層数分に増やされる．そして，その世代番号を持って，メモリから入力層及び競合層のデータを引き出し，update モジュール（図 5.12）にて，値の更新が行われる． 更新された競合層の値は，再びメモリに書き込まれて，廃棄される．そして Line 4 の FS#の M 番目のパケットが，

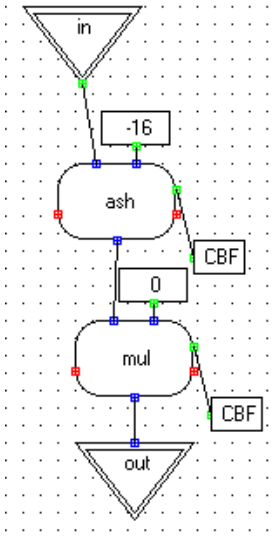

図 5.10　data0 モジュール

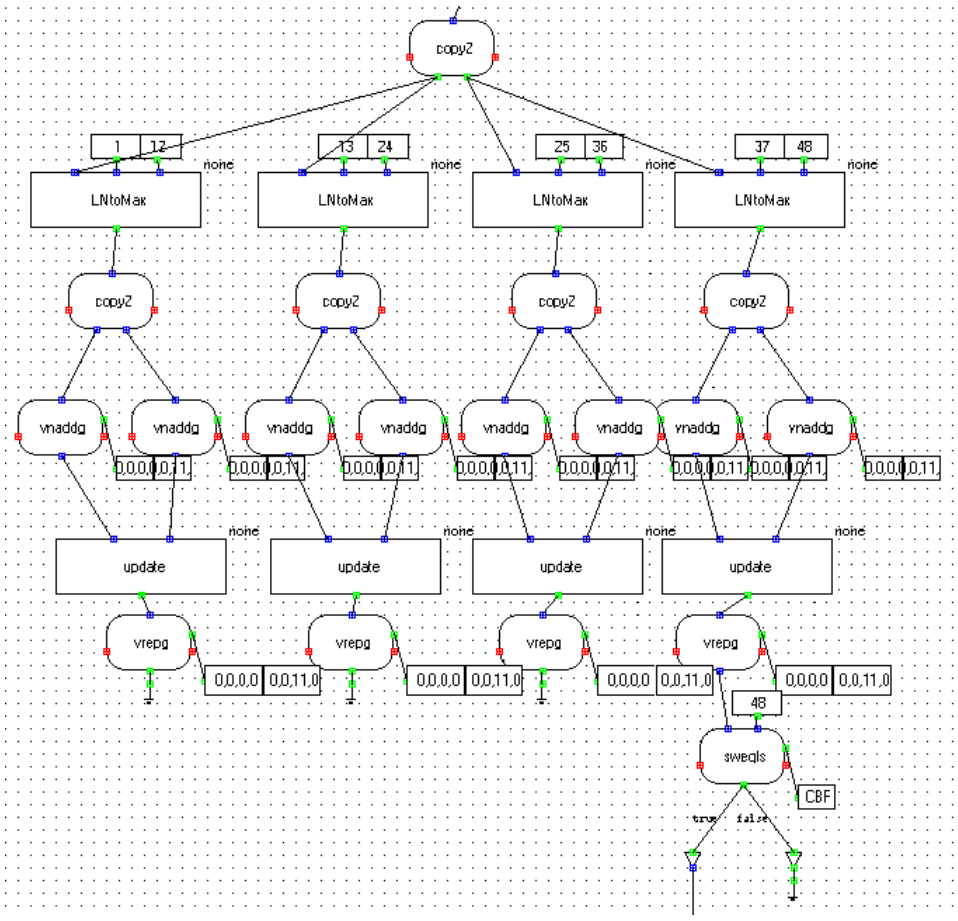

図 5.11　勝利ユニットの更新処理部分

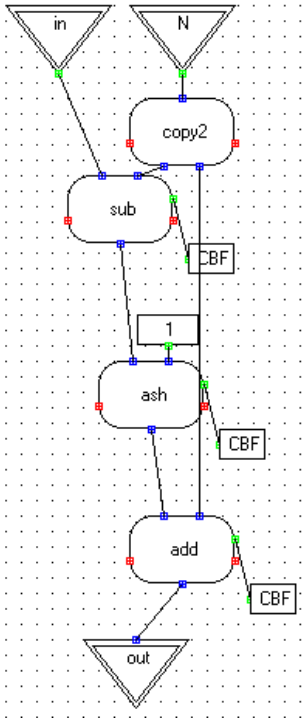

図 5.12　update モジュール

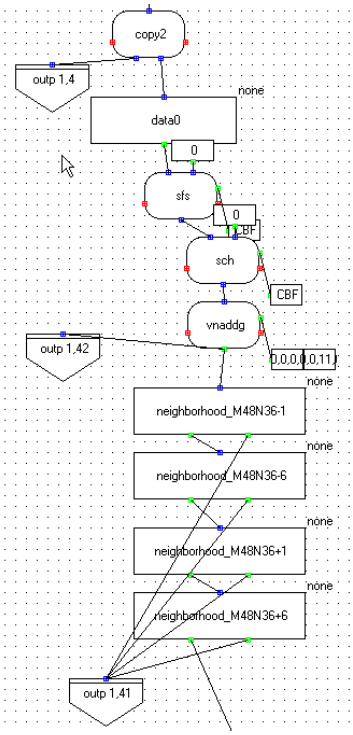

図 5.13　近傍処理部分

次処理である近傍の更新へと引き継がれる（図 5.13）．この処理部分に流れてくるデータパケットは，データの値及び，FS#値，CH#値をリセットされ，vnaddg 命令によって，メモリから勝利ユニットの CH#番号を取ってくる．この値を持って，近傍モジュールに処理をうつしていく．この図に示されている近傍モジュールは，それぞれ，近傍判定の命令ノード群と，図（5.11）と同じ命令群を擁する．

ここで，本プログラムでの近傍について述べておく．本研究では図（5.14）の配置の競合層を持つものとし，近傍の範囲を一律，勝利ユニットの周り 4 近傍とした．近傍の判定は，勝利ユニットとなったマスとの，数値の増減で判定を行っている．CH#21 のユニットが勝利した場合，CH#15，CH#20，CH#22，CH#27 の周囲 4 マスのユニットが近傍ユニットとして決定される．もし，CH#12 のユニットが勝利した場合，CH#6，CH#11，CH#18 の 3 ユニットが近傍となる．

近傍の更新を終えた後は，図（5.15）Record 数と Epoch 数の判定を行う．ここでは，流れてくるデータパケットの値をリセットし，メモリからカウンタの値を呼び出す．この部分

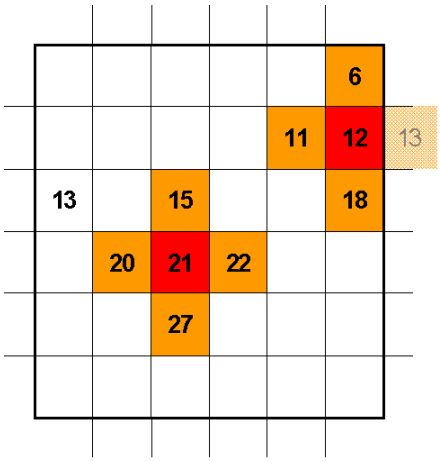

図 5.14　近傍について

を通過するたびに値を 1 増やし，Record 数が指定した数になると，sweq 命令にて Epoch 判定にまわす．Epoch 数が指定した数に達すると，処理は終了である．

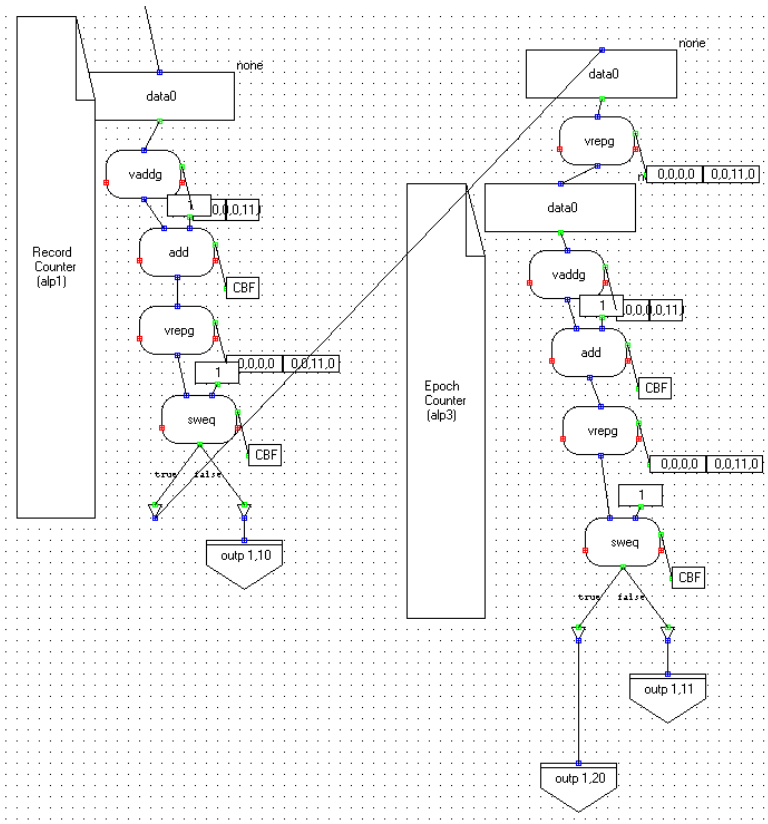

図 5.15　Record&Epoch カウンタ

5.3　GCPM-SOM の実装

次に GCPM-SOM の実装について述べる．前節と同様に，4 Line 構成で，入力層 48 個，競合層 36 個を持つ GCPM-SOM について考えることとする．宣言部分は，図（5.4）とまったく同じものを使っている．

5.3.1　距離測定部分

データの入力部分は図（5.5）とまったく同じものを使用している．しかし，GCPM-SOM は競合層を分割しているため，入力データの形式が GIPM-SOM とは異なる．GCPM-SOM が 1 つの Line で処理するデータは，複数の競合層ニューロンと，それと完全結合する入力層ニューロンである．よって，Line1 には，入力層 1〜48 個と，競合層 1〜9 個とを完全結合で結んだ世代番号を持つデータが投入される．これに伴い，距離測定モジュールも変わってくる．図（5.16）に，距離測定及び集計部分までを示す．集計作業を終えた後，データはメモリに格納される．

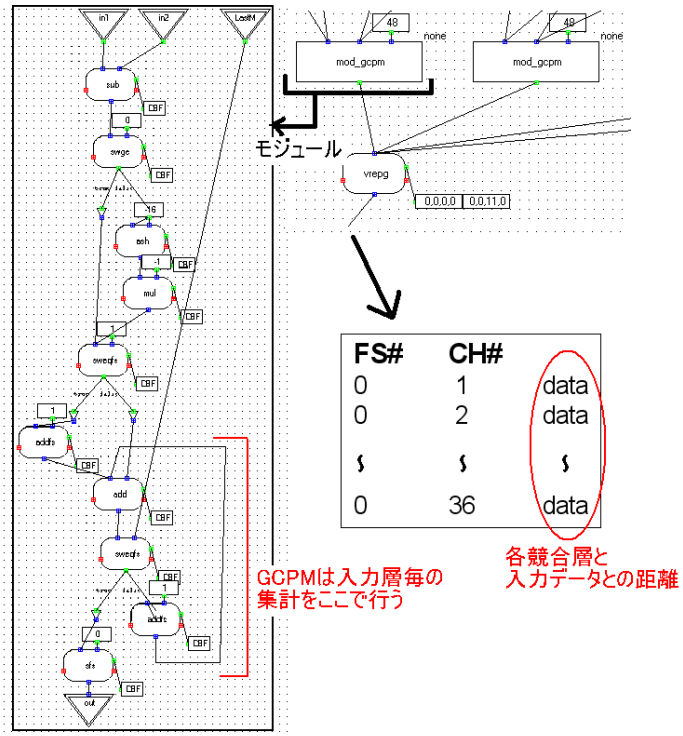

図 5.16　GCPM-SOM の距離測定部分

5.3.2　勝利ユニット 決定部分

次に勝者決定部分に移るが，GCPM-SOM の勝者決定部分は，GIPM-SOM のそれとまったく一緒である．これについては，5.2.2 節を参照頂きたい．

5.3.3　重み更新部分

GIPM-SOM と最も異なるのが，この重みの更新部分である．GCPM-SOM の重み更新部分の記述を図（5.17）に示す．まず，流れてきたデータを各 Line にコピーし，PXtoMax モジュールを使い，各々の Line が担当する競合層の分だけデータを拡張する．その後，copy 命令と sweqch 命令の組み合わせで，勝利ユニットが，自分の担当する競合層の中にあるかどうかを確認する．自分の担当する競合層ユニットが，勝利ユニットでなかった場合，データは廃棄されてその Line での処理は終了する．もし，勝利ユニットに該当する競合層ユニッ

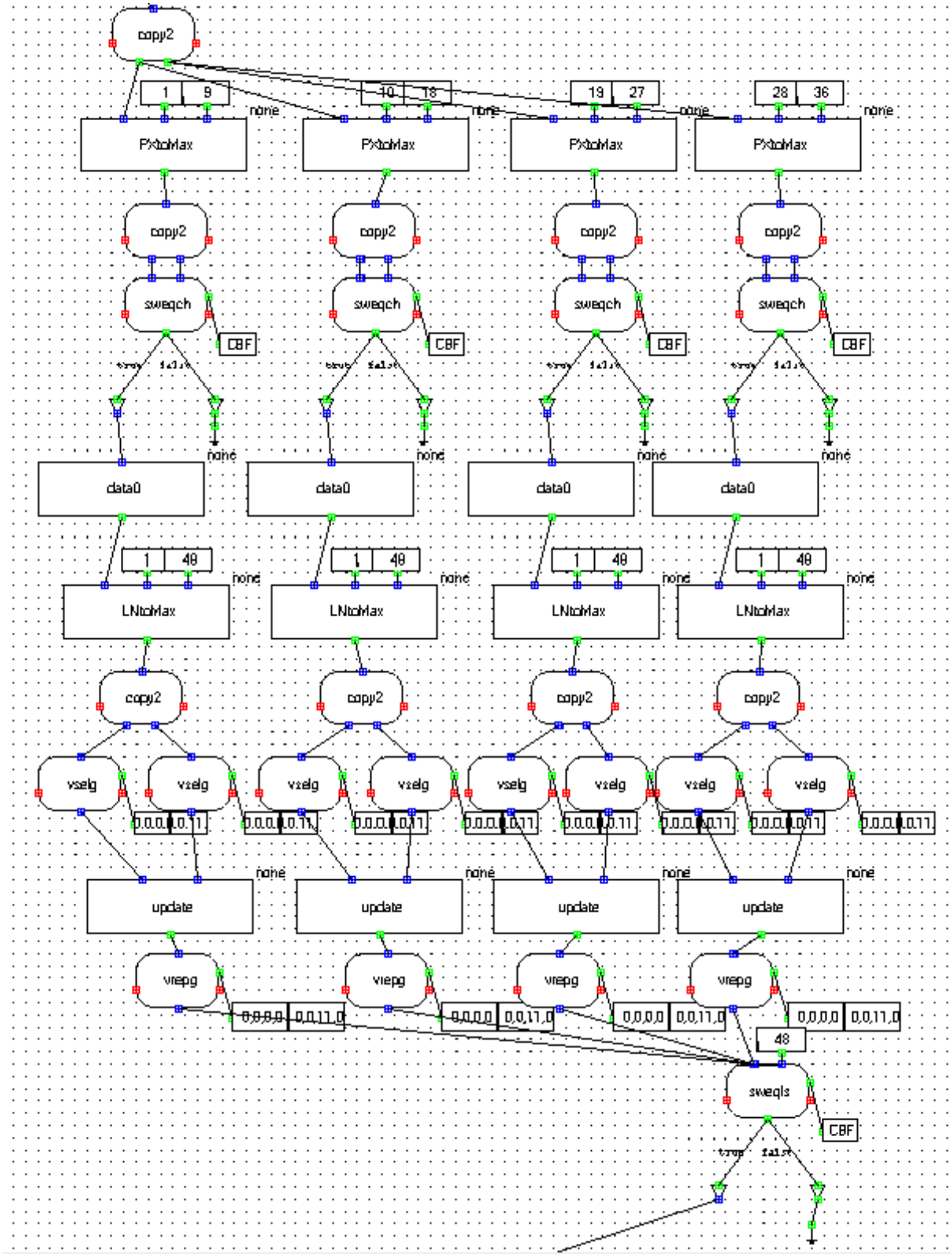

図 5.17　GCPM-SOM の重み更新部分

トが存在する場合，データの値を 0 にし，その世代番号を持って，メモリから対応する入力層及び競合層のデータを読み出し，update モジュールにて重みの更新を行う．そして，更新された重みは再びメモリに格納され，sweqfs 命令でふるいにかけられる．ここで FS#の M 番目のパケット一つだけを残し，その他のパケットは廃棄される．

その後，GIPM-SOM と同様に，近傍ユニットの更新，Record，Epoch 数のカウント判定と処理を移していく．記述方法は，近傍の更新モジュールが擁する，更新部分が異なるだけで，図（5.13，5.15）と同様である．

第 6 章

評価

6.1 32bit DDMP での評価

5 章で解説した GIPM-SOM と GCPM-SOM プログラムの各パラメータを変化させ，DDMP Simulator 上で実験を行った．シミュレーション結果は，Simulator のシミュレーション時刻であるステップ数で出力され，32bitDDMP 内部のパイプラインをパケットが 1 Stage だけ進む時間が 2 Step に相当する．目安としては，1 Step ＝約 3.5nsec. となる．

まず始めに，Line 数を 4 に固定し，入力層の数を 12，24，36，48，60 個，競合層の数を 16，36 個と変化させた，すべての場合の処理終了までの時間を計測した．実験対象とするデータの値は，0-99 までの一様乱数で，Record 数，Epoch 数は共に 1 である．また，採取する結果は，周囲 4 近傍全てを更新した場合の処理終了時間とする．この結果を，表 6.1 に示す．この表より，図（6.1）のグラフを導くことが出来る．

	M12	M24	M36	M48	M60
GIPM N16	34266	52505	74518	93722	113226
GCPM N16	55869	86939	117535	147847	180021
GIPM N36	61228	99197	138508	175787	214531
GCPM N36	86801	134914	186107	236698	281251

表 6.1　学習時間の比較

次に，入力層 48 個，競合層 36 個の場合と，入力層，競合層共に 36 個の場合，入力層 48 個，競合層 16 個の場合の 3 パターンについて，Line 数を 1〜8 と変化させて処理終了まで

6.1　32bit DDMP での評価

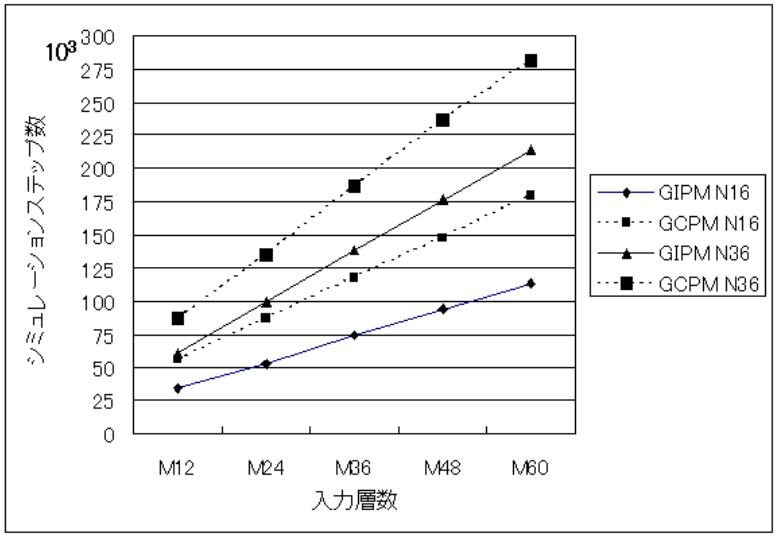

図 6.1　実験結果のグラフ

の時間を計測した．この結果を表（6.2）に示す．M48N36 は，5Line では，入力層及び，競合層をきれいに分割することが出来ず実装できない．よって，表には近似した値を入力している．M48N16 における，3，5，6，7Line も同様である． これをグラフにあらわしたもの

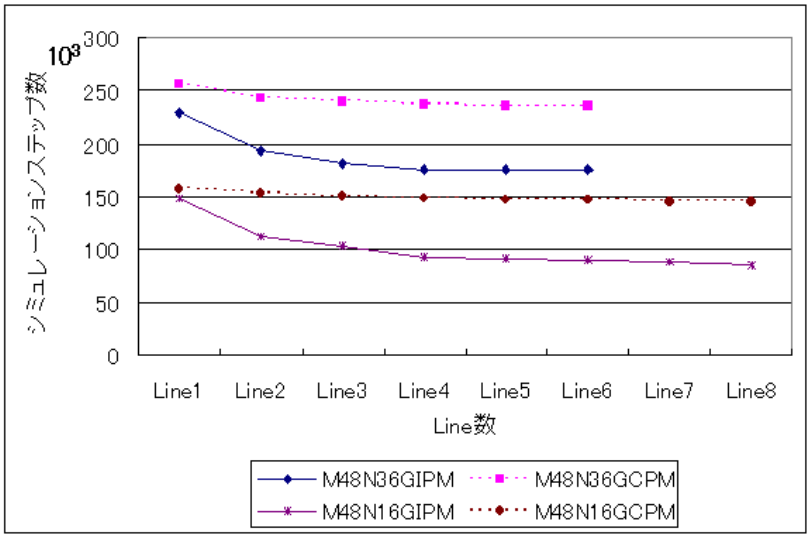

図 6.2　Line 数の変化による学習時間の移り変わり

が図（6.2）である．

また，入力層を 48 個，競合層を 36 個とし，GIPM-SOM 及び GCPM-SOM を各部位ご

	Line1	Line2	Line3	Line4
M48N36GIPM	230239	193985	182162	175787
M48N36GCPM	256700	243018	240175	236698
M36N36GIPM	177967	151636	142340	138508
M36N36GCPM	190804	190790	188437	186107
M48N16GIPM	148239	111796	102759	93722
M48N16GCPM	157289	152654	150250.5	147847

	Line5	Line6	Line7	Line8
M48N36GIPM	175326	174864		
M48N36GCPM	235790	234881		
M36N36GIPM	137661	136814		
M36N36GCPM	185766	185424		
M48N16GIPM	91807	89891	87975.5	86060
M48N16GCPM	147101	146729	146169.3	145610

表 6.2　Line 数による処理時間の変化

とにブレイクさせ，それぞれの処理に要する時間を計測した（表 6.3，6.4）．そして，これをまとめたものが，図（6.3，6.4）のグラフである．

図（6.1，6.2）のグラフから，全工程において GCPM よりも GIPM の方が，より短時間で処理を終えていることは明らかである．

また，図（6.2）から GIPM-SOM は，Line 数を増やすことにより，プログラムがより短時間で処理を終えることがわかった．これは，Line 数を増やすことで，DDMP 上の各 nPE のリソースを効率的に使えるためである．Line 数 1 では，処理しなければならないデータ数に対して，使用出来る nPE のリソースが少ないために，処理に時間がかかる．ここで使用しなかったリソースを，Line 数を増やすことで利用することができる．Line 数を 4 →

	Line1	Line2	Line3	Line4	Line5	Line6
勝利者決定終了まで	147608	141653	139838	139721	139683.5	139646
↑						
かかった時間	10573	10573	10573	10573	10573	10573
↓						
重み更新直前	158181	152226	150411	150294	150256.5	150219
↑						
かかった時間	72058	41759	31751	25493	25069	24645
↓						
ラスト	230239	193985	182162	175787	175325.5	174864

表 6.3　GIPM-M48N36 の各部位に要する時間

	Line1	Line2	Line3	Line4	Line5	Line6
勝利者決定終了まで	147608	148338	146188	146131	146381	146631
↑						
かかった時間	10571	10571	10571	10571	10571	10571
↓						
重み更新直前	158179	158909	156759	156702	156952	157202
↑						
かかった時間	98521	84109	83416	79996	78837.5	77679
↓						
ラスト	256700	243018	240175	236698	235789.5	234881

表 6.4　GCPM-M48N36 の各部位に要する時間

6.1　32bit DDMP での評価

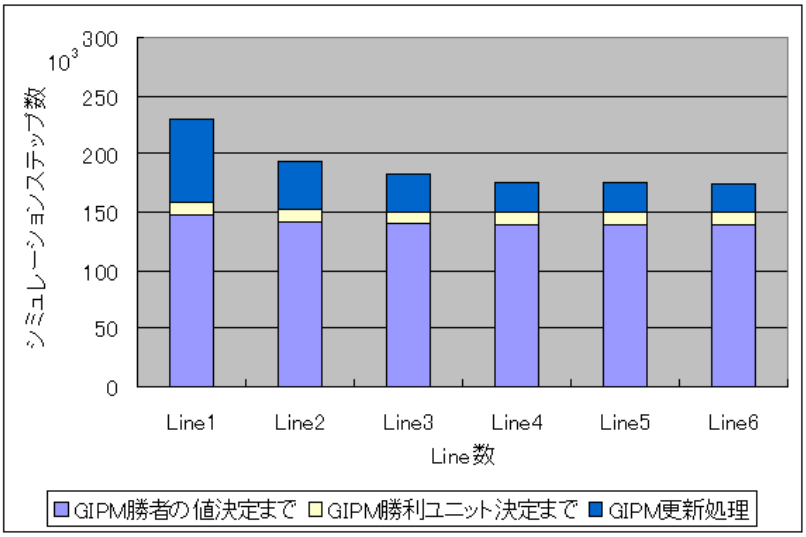

図 6.3　GIPM-SOM の各部位における所要時間

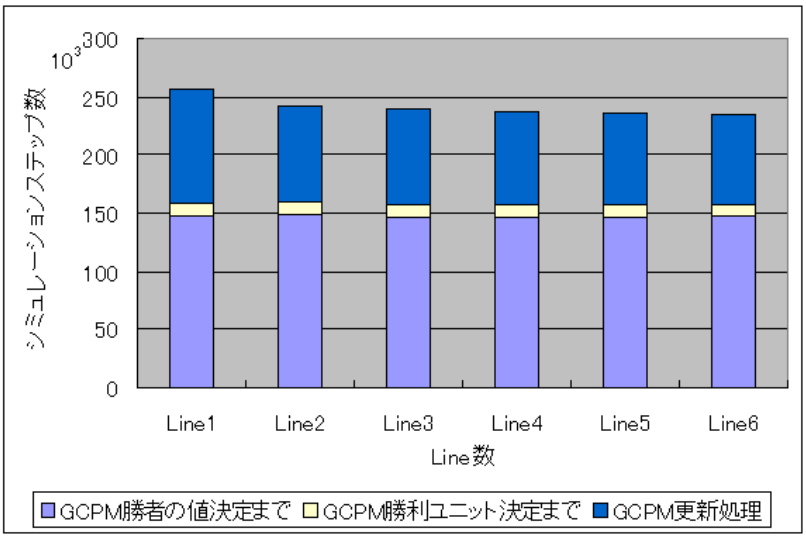

図 6.4　GCPM-SOM の各部位における所要時間

6 に増やしても，それほどスピードアップしていない．このれは，Line 数が増えることで，各 nPE での処理が増え逆に効率が悪くなるためである．GCPM-SOM に関しては，全体の Line 数が変化した場合でも，その内の 1 つの Line で全ての更新処理を行うので処理時間はほとんど変化しない．現在使用している nPE は ALP5 つであるが，将来的に全ての nPE を使用して実装することにより，総合的なリソースが増え，Line 数を増やすことでのより

高い並列性を得ることが出来るのではないかと考える．

表（6.3，6.4）や，図（6.3，6.4）のグラフは，プログラムを各工程でブレイクさせ，その必要時間を計測したものである．各工程での必要時間に目を向けてみると，両プログラムとも，勝者決定までの必要時間にあまり差はみられない．しかし，更新処理に要する時間を見ると，GIPM-SOM の方が高速であることがはっきりとわかる．これは，GIPM-SOM は使用できる Line を全て利用して並列に更新処理を行うのに対して，GCPM-SOM は Line 数が幾つになろうと，1Line で全ての更新処理を行わなくてはならないことに起因する．

最後に，実装した GIPM-SOM アルゴリズムの性能を実時間で検証する．今回は，入力データの次元を 60 次元，競合層の数 36 個の GIPM-SOM について，リアルタイムでの処理時間を計算してみる．Line 数 4 で実装し，レコード数，エポック数を共に 1 として実験した結果は，表（6.1）から，281251 Step となる．実験結果のシミュレーションステップ数は 1 Step が約 3.5nsec である．SOM を実際に試行する際，収束するまでの学習回数（エポック数）は，結果を見ながらユーザが任意に設定しなければならない．この程度のサイズの SOM であれば，一般的に 500-1000 エポックで収束すると考えられるが，今回は 2000 エポックと仮定して評価する．入力データ数（レコード数）を 50 個とした場合の，リアルタイム処理時間は，下記の式にて計算される．

$$\begin{aligned} \text{リアルタイム処理時間} &= \text{シミュレーション結果} \times 3.5\mathbf{n}\,\text{sec} \times \text{レコード数} \times \text{エポック数} \\ &= 214531 \times 3.5 \times 10^{-9} \times 50 \times 2000 \\ &= \text{約}\ 75.1\,\text{sec} \end{aligned} \tag{6.1}$$

この，75.1 sec という時間は，主観的ではあるが高速で十分実用に耐えうる値であると考える．今後は，他のハードウェアと競合させ相対的な処理時間の比較を行っていく必要がある．

6.2 DDMP-4g での評価

我々は，これまでの結果の他に，本論文で用いたものとは別のデータ駆動型プロセッサである，DDMP-4g を用いて実験を行っている．元来映像信号処理アプリケーションのた

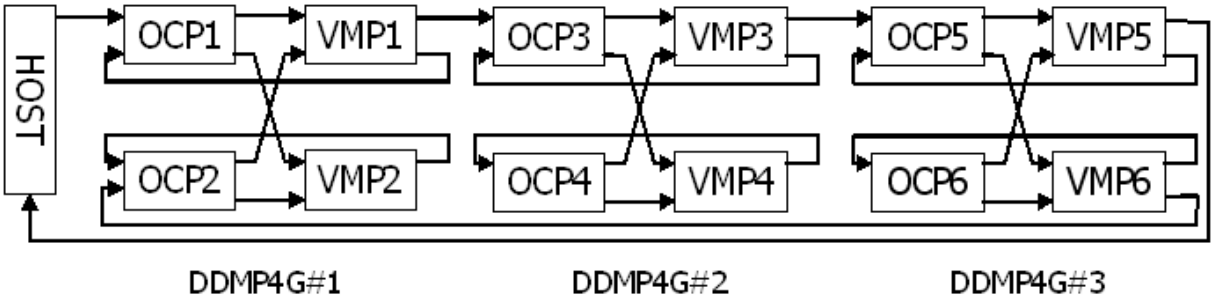

図 6.5　DDMP-4g のプロセッサ構成の例

めに開発されていた DDMP-4g は 2 台の Operation and Control Processors（OCP）と 2 台の Video Memory Processors（VMP）からなる．OCP は，一般的な信号処理機能を操作するために，VMP はタグによるメモリアクセスのために設計されている．この 4 つのプロセッサを有するチップを，図（6.5）のように複数接続したプロセッサ構成を使用した．DDMP-4g では，近傍なしの GIPM-SOM を実装し，入力層数 M を，16，32，64，96 個，競合層数 N を 8，16 個，ハードウェア上の PE 数つまり，OCP 数を 1，2，4，8 個と変化させ実験を行った． その実験結果のシミュレーションステップ数を図（6.6）にまと

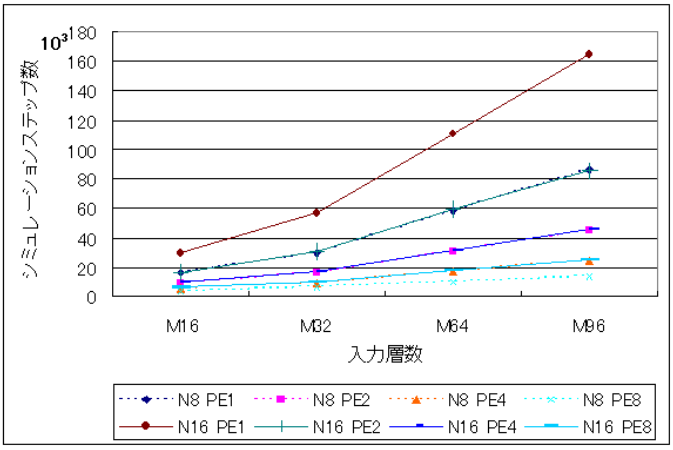

図 6.6　DDMP4g での実装結果

める．DDMP-4g では 1 Step=約 8.3nsec である．この結果を元に導き出した PE 数を増やすことによるスピードアップ率を，図（6.7）に示す．このことから DDMP-4g においても GIPM-SOM では，扱うデータ数が増えるにつれて，PE 数による並列性が発揮できるこ

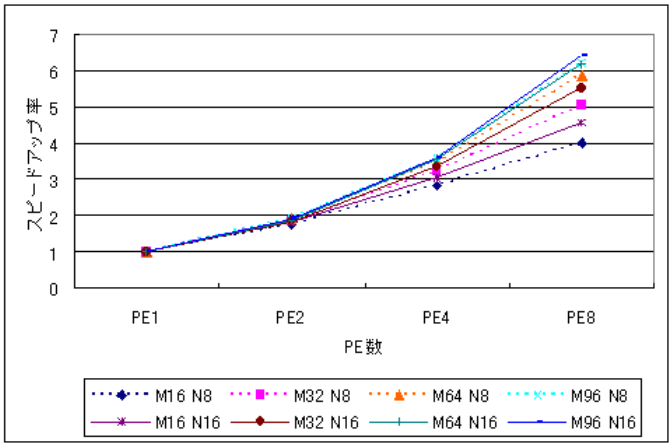

図 6.7　GIPM-SOM のスピードアップ率

とが明らかになった．プロセッサの性能は違うものの，この DDMP-4g に実装した結果と，32bitDDMP に実装した結果から，GIPM-SOM をデータ駆動型プロセッサに並列実装することの有効性を見ることが出来る．

第 7 章

おわりに

教師なし学習によって自己組織化的に特徴マップを生成することができる，ニューラルネットの一種である自己組織化マップ（Self-Organizeing Map:SOM）がある．SOM は入力層と競合層との 2 層間が完全結合で結ばれているため，ユニット数を増やすにつれて，計算処理数は膨大な数になる．この問題を解決するために，アルゴリズムの並列処理が望まれる．従来の競合層分割法（Competitive layer Parallel Model:CPM）は，競合層を PE 数で分割するために，しかし，勝利ユニットの更新を行う時に，各 PE に負荷の不均一が発生するという問題を抱えた．これを解決するために，入力層分割法（Input layer Parallel Model:IPM）が提案された．入力層分割法は，入力データを次元毎に入力層で分割し，各競合層へ完全結合で重みを持つ．こうすることにより，ある勝利ユニットが更新処理を行うとき，入力層に割り当てられた全ての PE が処理を行う．こうすることで，競合層分割法で問題になっていた，更新時の負荷の不均一が解消される．しかし，この手法も PE 数に比例してプログラム規模が大きくなるという問題がある．そこで，本研究では，複数の入力層ニューロンをひとまとまりとして PE に割り当てる，一般化入力層分割法（Generalized Input layer Parallel Model:GIPM）を開発した．GIPM は CPM の負荷の不均一問題と，IPM のプログラムの大規模化問題を解決する．そして，入力層分割法を元にした GIPM-SOM と，競合層分割法を元にした SOM アルゴリズムの性能を比較するために，一般化競合層分割法（Generalized Competitive layer Parallel Model:GCPM）を考案した．これは，複数の競合層ニューロンをひとまとまりとして PE に割り当てる．この GIPM-SOM と GCPM-SOM をデータ駆動型プロセッサである 32bitDDMP 上に実装し，性能を評価した．

その結果，全ての場合において，GIPM-SOM の方が短時間で処理を終え，さらにその差

は開いていく傾向にあることが明らかになった．これは，更新処理において GIPM-SOM は，全ての Line で処理を分割して行えるのに対し，GCPM-SOM は分割せずに処理しなければならないことに起因する．また，Line 数を増やすにつれて並列性の効果が上がることも，明らかにできた．これは Line 数を増やすことで，nPE のリソースを効率よく使用することが出来るからである．しかし，Line 数を増やし過ぎると，逆に nPE での処理が飽和してしまい，効率が悪くなってしまうこともわかった．一方 GCPM-SOM では，1 つのラインで全ての更新処理を行うので，Line 数増加による並列処理の恩恵をあまり受けることが出来ない．以上の様に，処理時間及び，実装アルゴリズムによる並列処理性の観点から GIPM-SOM の有効性を示した．

今後の課題，将来的な展望としては，次のようなものがある．

- 近傍周りのプログラムの拡張

  現在の近傍は一律，第一近傍の 4 つだけであるが，これを SOM アルゴリズムの式に基づいて，時間軸において近傍数が変化するよう改良する．
- 32bitDDMP の全ての nPE を使用，かつ，複数のプロセッサを用いての実装

  現在使用していない 32bitDDMP の nPE，3 つの LCP と 2 つの CVP を使用して，プログラムを実装する．このことにより，ハードウェア全体での利用できるリソースが増え，Line 数を増やすことでのより高い性能が期待できる．
- 実問題を通しての検証

  本研究は一様乱数のデータを用いて処理時間の検証を行った．今後は，実問題を通すことで，近傍の実装と関連しての収束性の検証，他のハードウェアで実装した場合との比較を行う．
- 階層化 SOM への組み込み

  GIPM-SOM を適用したとしても，32bitDDMP で利用できる PE のリソースには限界がある．よって，将来的な展望として，本研究のプログラムを階層化 SOM の中への組み込み利用を視野に入れている．階層化 SOM である GHSOM（Growing-Hierarchical

SOM）[7] は，Dittenbach らが提案した SOM の階層化手法である．従来型の SOM は学習前には，競合層数によるマップサイズを決定する必要がある．従来型の SOM では，このマップサイズは，我々人間の経験測により決めなければならない．しかし，GHSOM では最上位層以外の各マップは，均一なサイズからスタートし，ある判定条件を元にユニットを挿入し，適切なマップサイズまで拡大する．また，階層化においても判定条件を元に，条件を満たすユニットのみを階層化して行く．つまり，自動的にマップサイズの最適化を行えるのである．入力データが大量の場合にも各マップサイズは適度に保たれるので，アルゴリズムにおける競合層ユニット数を抑えることが出来る．GHSOM の各層における更新処理は，従来型 SOM アルゴリズムと同様であることから，現プログラムの組み込み利用が可能となる．

謝辞

本研究を実施するにあたって，たくさんの方々の御助力を頂きました．ハードウェアの知識とアドバイスを下さった，指導教員の岩田先生，本当にありがとうございました．大森先生には，梗概提出の際にお世話になりました．また，フローグラフ製作時には岩田研究室のみなさんに，相談に乗ってもらい，非常に助かりました．元指導教員のラック先生には，立命館大学から暖かい指導を受けることができました．また，同じ部屋で苦楽を共にした，平山君，友池君．いろいろとありがとう．その他，煮詰まった時に支えてくれた澤村さん，気分転換に付き合ってくれた沢山の友人達に，感謝したいと思います．本当にありがとうございました．

参考文献

[1] T. コホネン，“自己組織化マップ”，シュプリンガー・フェアラーク東京，(1999).

[2] 山森一人, 堀口進，“自己組織化ニューラルネットワークの並列学習シミュレーション”，北陸先端科学技術大学院大学 リサーチリポート IS-RR-97-0006, pp.1-16 (Mar. 1997).

[3] 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 (Mar. 2001).

[4] Ruck Thawonmas, Makoto Iwata, and Satoshi Fukunaga，“Parallel Implementation of Self-Organizing Map on a Dynamic Data-Driven Multiprocessor”，Proc. the 2001 International Symposium on Nonlinear Theory and its Applications (NOLTA'2001), Miyagi, pp. 613-616 (Oct. - Nov. 2001)

[5] 福永諭,RuckTHAWONMAS, 岩田誠，“SOM の並列学習とそのデータ駆動型プロセッサへの実装に関する研究”，日本機械学会第 12 回インテリジェント・システム・シンポジウム講演論文集,No.2-10, pp.343-344 (Nov. 2002).

[6] 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-I Vol.J81, No.2, pp.62-69 (Feb. 1998).

[7] M. Dittenbach, D. Merkl and A. Rauber，“The Growing Hierarchical Self- Organizing Map”，S. Amari and C. L. Giles and M. Gori and V. Puri, editors, Proceedings of the International Joint Conference on Neural Networks (IJCNN 2000), vol. 6, pp. 15-19, (July 2000)