解　説

3. 超並列マシンとその応用

3.3 SIMD 型商用超並列コンピュータとその応用†

喜連川 優†† 湯浅太一†††

1. はじめに

単一プロセッサの性能限界が明らかになりつつある今日，並列処理による処理能力の向上への努力がなされている．メインフレームやワークステーションでのマルチプロセッサ化，スーパコンピュータにおけるベクトルパイプの多重化など並列化は次第に定着しつつある．

これらの並列度は 4～32 程度であり比較的実現容易であるのに対し，本特集の対象とする超並列処理に関してはいまだ模索段階にあるといえる．超並列マシンに関しては，ソロモンマシンなどコンピュータが開発された当時からイメージは存在していたものの，その実現，特に商品化に関しては，ICL (AMT) の DAP など必ずしも多大な成功をおさめているとはいえない．超並列マシンが成功するためには，いかに多くの有用なアプリケーションを吸収できるかにかかっているといって過言ではない．

VLSI 化の技術の進歩により，単にプロセッサをたくさん寄せ集めたマシンを作ることは今日それほど大きな問題ではなくなりつつある．問題はアプリケーションとの整合性であり，どのような問題が超並列に処理可能なのか，あるいは今後の潜在的応用としてどの程度の大きさのマーケットが期待できるのかといったポイントが重要と考える．

本稿では，コネクションマシンならびにマスパーに代表されるいわゆる商用超並列 SIMD マシンに関し，応用面からその有効性を検討することにする．

2. 商用超並列 SIMD マシンのアーキテクチャ

きわめて並列度の高いシステムの構築を想定する際，当該システムの制御手法の決定が重要な課題となる．1000 台以上の超並列プロセッサシステムを効率良く動作させ問題を解くことは，同期，プログラミング，デバッグいずれもきわめて困難であり，なるべくシンプルな制御手法が望まれる．その最も極端なアプローチが SIMD といえよう．SIMD では全プロセッサに対して同一の命令がブロードキャストされ選択されたプロセッサが一斉に当該命令を実行し，それ以外のプロセッサは動作しない．一つのプロセッサの粒度をどの程度にするか，プロセッサ間の通信機構，距離をどのようにするか，アーキテクチャ設計上種々の選択が可能となる．コネクションマシン，マスパーに関するハードウェアアーキテクチャの詳細については本特集号のほかの論文を参照されたい．基本的な特徴についてまとめると，CM-2 では一つのプロセッサのデータパスは 1 bit であり，全ての処理はビットシリアルに行われ，また，マスパー MP-1 では 4 bit 単位の処理となる．集積度の制限からプロセッサを複雑にするとそれだけ台数が減少することになり，どの程度にすれば最適かという問題は未解決である．結合網は CM-2 では 64 K 台のプロセッサを 1 チップ内では 16 台のプロセッサを 4 × 4 のメッシュ状に結合し，チップ間では 12 次元のハイパキューブ網で結んでいる．一方 MP-1 では 16 K 台のプロセッサを多段クロスバを用いて結合している．プロセッサの構成を単純化し，プロセッサ数の大幅な増大化をねらっているため，必然的にプロセッサ間を結

† Commercial Massive Parallel SIMD Computer and Its Application by Masaru KITSUREGAWA (University of Tokyo, Institute of Industrial Science), Taiichi YUASA (Toyohashi University of Technology).

†† 東京大学生産技術研究所
††† 豊橋技術科学大学

✲：用語解説にあることを示す記号

合する通信リンクのコストも増大する．一つのチップのピン数の制限などからリンクのバンド幅は低くせざるをえず，通信コストは通常の演算コストに比べ1桁から2桁程度大きくなってしまう．たとえばCM-2 (8K プロセッサ) で64 ビットデータのランダムな通信をハイパキューブ網を使って全プロセッサ間で行うと3.5ミリ秒も必要となる〔文献2)〕．第2部4章ベンチマーク評価．

各プロセッサに付加されるメモリ容量はCM-2では現行モデルでは 256K ビットであるが1M ビットに増加される予定である．一般に従来の連想プロセッサやアレイプロセッサではメモリはきわめて小容量であったのに対し，半導体メモリの進歩を反映し，大容量化が進んでいる．浮動小数点演算については，CM-1 ではサポートされていなかったが CM-2 では 32 個のプロセッサに対して一つの Weitek のチップを，全体で2K個付加し，数値演算スーパコンとしての色彩を強くしている．ビットシリアルなプロセッサとビットパラレルな Weitek チップの間のデータ形式の変換はスプリントチップと呼ばれる専用 LSI で実現されている．

また，この種の超並列マシンには高速な入出力デバイスが不可欠である．低性能な入出力システムではプロセッサアレイにデータをロードするのに多大の時間を要することになり，処理時間のほとんどが I/O 時間となりがちであるからである．MP-1 では現時点では高速入出力はサポートされていないが（予定はされている），CM-2 ではデータボルトと呼ばれる高速入出力二次記憶装置が用意されている．これは最近開発が急速に進んでいるいわゆる"ディスクアレイであり，32 個のデータ用ディスク，7個のエラーチェック用ディスク，3個のスペアディスクから構成される．データはビットレベルで各ディスクにインタリーブされ，RAID アーキテクチャのレベル2に相当する[19]．ディスクアレイとは近年ディスクの小型化，低価格化が進む中で，比較的高価な大型ディスクをアレイ化した多数の小型ディスクで置きかえようとするアプローチであり，古くから動画処理用などに用いられていたが，最近特に信頼性を向上させる工夫が施され高速データ転送を必要とするスーパコンなどの応用で注目されている．

3. 商用超並列マシンのプログラミング環境

前章で述べたように，コネクションマシンとマスパはそれぞれ 1bit と 4bit のプロセッサで構成されている．これらのプロセッサを 1bit あるいは 4bit 単位で操作する低レベルのアセンブリ言語がいずれのマシンにも提供されているが，実用レベルのアプリケーションをこれらの言語によって記述するのはきわめて困難である．そこで，C, Lisp, Fortran といった既存のプログラミング言語を拡張し，超並列 SIMD 計算のための機能を追加した言語が提供されている．これらの言語は，既存の逐次プログラムを自動的に並列化するためのものではなく，抽象度の差こそあれ，プログラマが並列実行を意識してコーディングを行うことによって並列計算が初めて可能となる．自動並列化が望ましいことはいうまでもないが，プロセッサ・メモリへのデータの配置方法など困難な問題が数多く残されており，今後の研究課題となろう．

これらの言語では，拡張機能をまったく使用しないプログラムは，並列処理を制御するフロントエンド部のみで稼動する．フロントエンドは通常のフォン・ノイマン型マシンであり，与えられたプログラムを逐次実行する．つまり，拡張機能を使用しないプログラムは，従来の計算機上における場合と同様に実行される．拡張機能を使用した場合に限り，必要な命令列が全プロセッサにブロードキャストされて並列実行が可能となる．このためにC言語などで書かれた既存のプログラムをまずフロントエンドに移植し，並列化可能な部分を少しずつ拡張機能を使って書き直してゆき，徐々に並列化による性能向上をはかることができる．

コネクションマシン上で一般に利用されているプログラミング言語には

- Paris　● *Lisp
- C*　● CM Fortran

の4つがある[1),2)]．Paris はアセンブリ言語，C* と *Lisp はそれぞれC言語と Common Lisp に並列計算機能を付加したものである．CM Fortran は，Fortran 77 に ANSI の Fortran 8x 規格案の配列計算機能を追加した言語である．さらに実験的な言語として，*Lisp より抽象度の高い並列計

算機能を Common Lisp に追加した

- CmLisp[3]
- Paralation Lisp[4),5]

も知られている.

一方，マスパーに関しては，マシンが開発されたばかりであり，プログラミング言語はコネクションマシンほど豊富ではない．専用アセンブリ言語のほかに，MPL[6] とよばれるC言語ベースのプログラミング言語が主に使用されており，Fortran 77 ベースの MPF[7] が近々利用可能になる．基本的には，MPL はコネクションマシン上の C*, MPF は CM Fortran と同様な拡張機能を備えている．

以下，本節では C*, *Lisp, CM Fortran について概説することにする．

3.1 C*[16]

C* は，SIMD 型並列計算機の計算モデルをそのままC言語ふうの構文で記述できるようにC言語を拡張したものである．プロセッサの局所メモリにデータを配置するには，通常の変数宣言の前に poly という修飾子をつける．

```
poly int salary;
```

とすると，salary という整数変数が全てのプロセッサの局所メモリに確保される．64K 台のプロセッサを搭載した最上位モデルであれば，各プロセッサの局所メモリに変数領域を一つずつ，合計 64K 個の変数領域を確保する．これに対して，poly のない通常の変数宣言は，フロントエンドに領域を確保する．

```
int total;
```

は，プロセッサの台数に関係なく，フロントエンドのメモリに1個だけ変数領域を確保する．

salary のような poly 変数に対する演算は，全プロセッサに対する並列演算となる．たとえば，

```
salary=salary*1.03;
```

は，全プロセッサの salary 値を 3% 増加させる (3% のベースアップ)．代入文の右辺には poly 変数が使用されているので，C* コンパイラはこれを並列処理の指示だと理解し，定数 1.03 を全プロセッサにブロードキャストし，個々のプロセッサの salary の値にその定数を掛けるようなコードを生成する．さらに代入先も poly 変数なので結果を各プロセッサごとに局所メモリに格納するようなコードを生成する．

プログラムの一部を，特定のプロセッサのみを選択して実行させたい場合には，条件文を使う．たとえば，次の if 文は，salary の値が 10 万未満の全てのプロセッサに対して，salary の値を一律に 10 万とするものである（最低賃金の設定）．

```
if (salary<100000) salary=100000;
```

この if 文の条件式は poly 変数に関するものなので，その真偽を並列に求め，条件が真となるプロセッサのみが選択されて if 文の本体を実行する．代入文のための命令列はフロントエンドから全プロセッサに送られるが，選択されなかったプロセッサ（つまり，salary の値が 10 万以上のプロセッサ）はこれらの命令列を無視する．繰り返し文も考え方はまったく同じである．

```
while (P (salary))
  salary=F (salary);
```

は，個々のプロセッサに対して，salary に関するある条件 P が成り立つかどうかを調べ，P の成り立つプロセッサのみを選択して本体の代入文を実行する．その後，選択されたプロセッサ全てに対して P が成り立つかどうかをもう一度調べ，P が成り立つプロセッサのみを再度選択して本体を実行する．これを，どのプロセッサにおいても P が成り立たなくなるまで繰り返す．SIMD 計算の基本は，各プロセッサが自分の局所メモリ内にあるデータを上記のような機能を使って独立して処理するところにあるが，このように並列処理されたデータをフロントエンドに集計する（ある poly 変数の総和や論理和を求めるといった）機能が必要である．個々のプロセッサの局所メモリのデータをフロントエンドに逐一取り出してフロントエンドで処理することもできるが，プロセッサ数を考えると非常に効率が悪い．プロセッサ間のネットワークを有効に利用して集計を行うのが賢明である．たとえば，全プロセッサの salary の総和を求めるには，C* では次のように記述する．

```
total=0;
tatal+=salary;
```

total は poly でない通常の変数であった．これに全プロセッサの salary の値を加える．値を一つずつ加えてゆくのではなく，ハイパキューブ網を駆使し，全プロセッサを仮想的に2進木状に配置し，総和を求め，結果をフロントエンドに送って total に足し込むのである．プロセッサ数 n に対

して log (n) のオーダで集計が終了する.

3.2 *Lisp[17]

Lisp の計算モデルは基本的に C のそれと同じである. *Lisp のベースとなっている Common Lisp では, 大域変数 (スペシャル変数) を宣言するには, defvar 式を使う.

```
(defvar total 0)
```

とすると, total という名の大域変数が定義され, その初期値が 0 となる. *Lisp ではこの変数はフロントエンド側に配置される. プロセッサの局所メモリに配置される変数, つまり C* の poly 変数に相当する変数を, *Lisp では pvar 変数と呼んでいる. 大域的な pvar 変数を定義するには, *Lisp 固有の *defvar 式を使用する.

```
(*defvar salary (!!0))
```

とすると, 全プロセッサの局所メモリに salary という pvar が割り当てられ, その初期値が 0 にセットされる. ここで, (!!0) は, フロントエンドにある整数 0 を全プロセッサにブロードキャストすることを意味する. 原理的には, 全プロセッサに一時的に変数領域を一つずつ割り当て, その値を全て 0 にする. 上の *defvar 式は, まず salary を割り当て, (!!0) を評価し, その個々の値を, 同じプロセッサの salary にセットする. そして, *defvar の実行が終われば (!!0) の結果を格納していた領域は解放される.

C* のところであげた 3 %のベースアップは, *Lisp では次のように書く.

```
(*set salary (*!! salary (!! 1.03)))
```

ここで *!! は, 掛け算をする Common Lisp の組み込み関数* の並列版であり, 引数として与えられた pvar 変数を, プロセッサごとに掛け合わせる. * の定義を拡張して, もし引数が pvar であれば並列に掛け算するようにも定義できたであろうが, インタプリタの実行効率や, コンパイラの簡潔さのために, 並列用の関数を別に定義したものと考えられる.

最低賃金を 10 万円に設定するには

```
(*if (<!! salary (!! 100000))
  (*set salary (!! 100000)))
```

とする. また, プロセッサの局所メモリのデータをフロントエンドに集計するには, 専用の関数を使う. たとえば, salary の総和を求めるには,

```
(*sum salary)
```

とする. *Lisp で拡張された関数名は, pvar を値とするものには "!!", そうでないものには "*" をつけて区別していることに注意されたい.

3.3 CM Fortran[18]

Fortran 8x では, 配列をあたかも 1 個のデータとして取り扱うことが許されている. 配列 A, B, C に対して,

$A=B+C$

とすれば, B と C の対応する要素の和を, 配列 A の対応する要素に代入することになる. 通常の逐次型マシンであればこれをループに変換して実行するわけであるが, コネクションマシンやマスパーでは, あらかじめ配列の個々の要素を各プロセッサの局所メモリに割り当てておき, 加算と代入を一斉に行えばよい. CM Fortran と MPF は, このような Fortran 8x の配列単位の演算を並列実行するように Fortran 77 を拡張したものであり, 拡張部分については極力 Fortran 8x に準拠するように設計されている.

3 % のベースアップは, ベクタ (1 次元の配列) SALARY に対して,

```
SALARY=SALARY *1.03
```

とすればよい. また, 最低賃金の設定は, Fortran 8x の WHERE 文を使って

```
WHERE (SALARY. LT. 100000) SALARY=
100000
```

とする. SALARY の総和を求めるには組み込み関数の SUM を使う.

```
TOTAL=SUM (SALARY)
```

CM Fortran も MPF も, 基本的には Fortran 8x の配列演算機能をそのまま採用している. このために, 逐次マシン上の Fortran 8x 処理系を使って容易にテスト実行できたり, 逐次マシンとプログラムを共有できるなど C* や *Lisp にはない利点を有している.

4. 超並列 SIMD マシンのアプリケーション

本章では, コネクションマシン上に実現されたいくつかのアプリケーションを紹介する. それらが SIMD 型超並列マシン上でどのようなアルゴリズムによって実現され, どの程度の性能を有するかについて紹介する.

これらのアプリケーションは,

(1) 大容量のデータに対し,

（2） 比較的簡単な操作を
（3） 繰り返し実行する．
という点で共通しており，超並列 SIMD マシンの得意とするものばかりである．また，現状ではプロセッサ間通信の性能があまり高くないために
（4） できるかぎりプロセッサ間通信を避ける．
ように実現されている点でも共通している．

4.1 流体運動シミュレーション

流体運動のシミュレーションは，航空力学や気象予測など幅広い分野で必要とされている．従来は，膨大な方程式をスーパコンピュータなどを駆使して解くのが普通であったが，ここでは，流体の粒子の動きを模倣することによる解法を紹介する．

流体は数多くの微小粒子の集合体であり，その運動は，微粒子同士の衝突による力学的作用をマクロ的にとらえることによってシミュレート可能である．簡単のために，流体が同質の粒子から構成されるものとする．シミュレーション空間には粒子が点存し，それぞれが固有の運動方向に移動する．二つの粒子が衝突すれば，力学法則に従ってそれぞれの粒子の運動方向が変化する．さらに，シミュレーション空間には，粒子の動きを遮る障害物が存在し，障害物に衝突した粒子は，やはり力学法則に従った運動方向に変化する．障害物とは，航空機の翼など，シミュレーションによってその空力抵抗などを知りたい実験対象のことである．有効なシミュレーションを行うには膨大な数の粒子を想定する必要があるが，個々の粒子の運動は比較的単純であり，単純なプロセッサを多数結合した SIMD 型超並列マシンが有効に利用できる[8]．

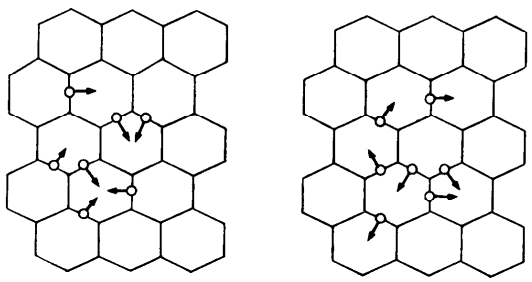
シミュレーションの対象をどのように実際のプロセッサに割り当てるかによって，実行速度は大きく左右される．もし各微粒子を1台のプロセッサに割り当てれば，次の瞬間にその粒子がほかの粒子あるいは障害物に衝突するかどうかの判断に多大の計算時間を要するであろう．次の瞬間におけるその粒子の位置と，ほかの全ての粒子及び障害物の位置を比較する必要があるからである．この問題の場合は，むしろ，シミュレーション空間を小領域に分け，各小領域を1台のプロセッサに割り当てるほうが賢明である．各小領域に現在存在する粒子の全てに対して，それらの次の瞬間における位置を決定するのは容易なことである．もしそのいずれかが別の小領域に移動すれば，プロセッサ間通信によってその情報を伝達すればよい．移動先は，その粒子が存在していた小領域の近傍であり，隣接する小領域を，物理的に近接するプロセッサに割り当てることによって，通信コストの高いグローバルネットワーク（コネクションマシンの場合はハイパキューブ）ではなく，メッシュ結合による高速近傍通信が利用できる．

4.1.1 粒子運動モデル

実際にコネクションマシンで実現されているシミュレーションプログラム[8]は2次元空間を対象とし，次のような簡単な粒子運動モデルを採用している．まず，シミュレーション平面全体を正六角形の小領域に分割する．微小粒子は全て小領域の境界線上にあり，境界線に垂直に移動するものとする（図-1(a) 参照）．各小領域の6つの辺のそれぞれの上には，その小領域に進入しつつある粒子が高々1個しかないものとする．同じ小領域に入ってきた粒子は小領域の中心で衝突して方向を変え（あるいは直進し），次の瞬間には小領域の境界線上に達し，境界線に垂直に進んでゆく（図-1(b) 参照）．小領域のいくつかは障害物である．粒子は障害物には進入することができず，境界線に垂直に反発される．つまり，障害物である小領域に進入しつつある粒子は，次の瞬間には同じ位置にあって進行方向が反転することになる．

4.1.2 実装手法

各小領域を1台のプロセッサが担当し，ある時点にその領域のどの辺から粒子が進入しつつあるかを知り，次の瞬間にその小領域のどの辺から粒子が出てゆくかを計算する．これらの粒子は，次



（a） 現在の状態 （b） 次の瞬間の状態

図-1 粒子運動モデル

のステップでは隣接する小領域に進入することになるので，担当するプロセッサにその旨を連絡する．この一連の処理を全プロセッサが同時に実行することにより，膨大な数の小領域の処理を高速に実行できるのである．

各小領域の処理には，どの辺から粒子が進入しつつあるかを記憶する6ビット(1辺1ビット)のデータ incoming，その結果，次の瞬間にどの辺から粒子が出ていくかを記憶する6ビットのデータ outgoing，それにその小領域が障害物かどうかを記憶しておく1ビットのデータ obstaclep があればよい，outgoing の値は incoming の値によって決定される．incoming の値を6ビットの整数 x とみなした場合の outgoing の値を $f(x)$ とするとき，各ステップの計算は次の C* コードにより実行できる．

```
if obstaclep
  outgoing=incoming;
else outgoing=f (incoming);
```

$f(x)$ は，その定義域が $0 \leqq x \leqq 63$ と小さいことから，あらかじめ全プロセッサの局所メモリにテーブルの形で記憶しておき，インデキシングによって一斉に求めることができる．

六角形の小領域と実際のプロセッサの対応は，**図-2** のようにすればよい．図中，黒丸がプロセッサを表し，上下左右の直線がメッシュ結合を表す．各小領域には，6つの小領域が隣接する．それらの隣接小領域を担当するプロセッサは当該プロセッサの上下左右に位置する4台のプロセッサと，右上及び左下に位置する2台のプロセッサである．後者の2台は直接に接続されていないので，通信は前者のいずれかを経由して行うことになる．

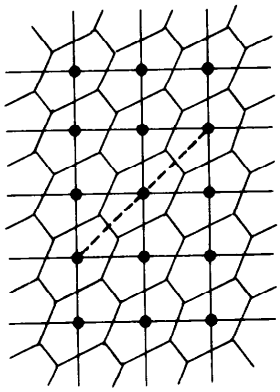


図-2 六角形の小領域とプロセッサの対応

4.1.3 性 能

CM-1 を利用した際，シミュレーション平面を約 4000×4000 小領域に分割し，3000 万個の微粒子を操作した結果，1秒間に 60 ステップ以上の計算が行えたという報告がある．個々の小領域の計算には1ステップあたり約 70 回の論理演算を必要とする．したがって1秒あたり 6.72×10^{10} 回の計算を行ったことになる．

4.2 等高線地図自動生成

同じ景色を異なる2地点から写真撮影した場合，2枚の写真の間には微妙なズレが生じる．そのズレは，カメラから近距離にある対象物ほど大きく，対象物が遠距離になるほどズレは小さくなってゆく．人間の眼でも同じことがいえ，これによって遠近感を感じることができる．等高線地図の作成にも同じ原理が利用されている．まず，航空機などを利用して，上空の異なる2地点から地上を撮影する．その結果得られた2枚の写真（立体写真）のズレを解析して等高線地図を作成するのである．従来は人間の職人芸によって解析が行われていたが，精度の向上や時間短縮のために計算機による処理が望まれている．

以下では，コネクションマシン上に実装された等高線地図自動作成プログラムを紹介する[8]．このプログラムに対する入力は，2枚の立体写真である．直感的に理解しやすいように，これらの画像を，あたかも人間の両眼がとらえたかのように取り扱い，「右」及び「左」の画像と呼ぶことにする．

4.2.1 実 装 手 法

プログラムは次の5つのステップから構成されている．

1. 入力された2枚の立体写真から，輪郭線を抽出する．
2. 「右」の輪郭線図を1ピクセルずつ左にシフトしながら「左」の輪郭線図と照合することによって，輪郭線上の各点のズレを求める．
3. 撮影高度と撮影地点間の距離をもとに，ズレの大きさから輪郭線上の各点の高度を求める．
4. そのほかの地点の高度を補間法によって求める．
5. 同一高度の地点を結び等高線を描く．

等高線地図の自動作成は画像処理の一種である．従来の大多数の画像処理専用並列計算機がメッシュ状に結合された SIMD 型プロセッサによ

って構成されていたことから明らかなように，画像処理はコネクションマシンなどが最も得意とする分野の一つである．各ピクセルを1台のプロセッサに割り当てることによって，上記1，4，5の各ステップが容易にしかも効率よく実現できることは基本的な画像処理技法としてよく知られている．また，ステップ3は単純な三角関数の計算であり，全ピクセル（したがって全プロセッサ）に対して一斉に実行できる．残されたステップ2について少し詳しく説明する．

ステップ2は具体的には次のように処理される．まず左右の輪郭線図の同一位置にあるピクセルが同一プロセッサに割り当てられるように輪郭線図の全ピクセルを配置しておく．この状態どちで，各プロセッサは，自分の担当するピクセルがどちらも輪郭線上にあるかどうかを判断する．輪郭線上のピクセルを1，そうでないピクセルを0で表現しておけば，この処理は論理積をとることを意味する．次に，「右」の輪郭線図を1ピクセル分だけ左にシフトする．この処理は，全プロセッサが自分の左隣りのプロセッサに，自分が担当していた「右」のピクセルの情報を送信することを意味する．そして，再び論理積をとり，さらに「右」のピクセルを左にシフトする，ということを繰り返すのである．

上記の処理を繰り返せば，「左」の輪郭線上の各ピクセルに対して，「右」の輪郭線図を何ピクセル分シフトした場合に輪郭線上のピクセルと合致するかが判断できる．しかし，入力画像のノイズや偶然のために，複数回合致する場合が少なからず生じる．そのいずれが本当に求めたいズレであるかを判断しなければならない．この問題を，コネクションマシン上のプログラムでは，ピクセル同士が本当に合致すれば，その近傍のピクセル同士も合致するはずだ，と仮定することによって解決している．すなわち，近傍のピクセルが最も多く合致したシフト数を，そのピクセルのズレとして採用している．

4.2.2 性　能

CM-1 において，上記のプログラムを 512×512 ピクセルの入力画像に適用し，ステップ2において 30 ピクセル分までシフトさせた結果，ステップ2が2分以内で実行できたと報告されている[8]．容易に想像できるように，ステップ2の実行がこのプログラムの実行時間の大半を占める．したがって，ほぼ2分程度で1枚の等高線地図が作成できるものと推測できる．動画を入力して実時間で等高線地図を作成するには，この 1000 倍以上の速度が要求されるが，その達成は今後の課題となろう．

4.3 ドキュメント検索

ダウジョーンズ社は，コネクションマシンを使い高度なドキュメント検索システムを実用化している．

コネクションマシン上のドキュメント検索システム[9]は，「関連性フィードバック」を使った，ユーザが使いやすいシステムである．その検索の方法は二つの段階に分かれる．

第一段階で，ユーザはいくつかのキーワードを入力する．キーワードには重み数値が付加されて全てのプロセッサにブロードキャストされる．それぞれのプロセッサは内部に一つのドキュメントとその得点をもっており，ブロードキャストされた単語が自分のドキュメントに出てくる場合，単語の重み数値をドキュメントの得点に加算する．頻繁に出てくる単語は小さな重み数値をもち，稀にしか出てこない単語は大きな重み数値をもつ．全てのキーワードを流し終わった後，得点の高いほうからいくつかのドキュメントを選んでユーザに示す．

第2段階で，ユーザは示されたドキュメントの一つ一つについて，そのドキュメントが自分の求める情報に関係する場合 “good”，そうでない場合は “bad” の判定をする．そして，“good” として選ばれたドキュメントの中の全ての単語をキーワードとして，第1段階と同じ処理が行われる．このようにして，最終的にユーザの求めるドキュメントが全てリストアップされる．「関連性検索」と呼ばれるこの方法は，ユーザが使いやすく，検索の正確さにおいても優れている．しかし，極端に大きな計算力が必要であるため，今までに実際のシステムで使われたことがなかったが，超並列マシンの出現により，現実のものとなったといえる．

4.3.1 実装手法

ドキュメントは，そのままのテキストとしてプロセッサに保持されるのではなく，ある種の加工が施される．もとのドキュメントは，まず専用の

プログラムに通されて重要な単語だけが選択される．どのドキュメントにも出てくるような単語が捨てられて，結果的に3分の1程度の量に圧縮される．次に，残った単語が「サロゲートコーディング」と呼ばれる方法によってビットベクタのデータ構造へと変換される．サロゲートコーディングは，複数の単語を固定長のビットフィールドに符号化するものであり，次のように行う．

ⅰ） まず，ビットベクタを0で初期化する（図-3(a)）．

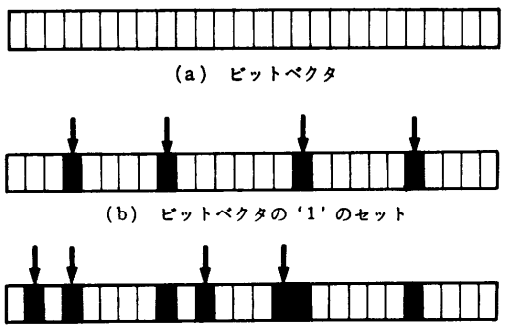
ⅱ） 符号化する単語を複数のハッシュ関数にかけ，それぞれのハッシュ値をインデックスとして対応するビットを1にセットする（図-3(b)）．

ⅲ） 対象とする全ての単語について同様の処理を行う（図-3(c)）．

CM-2のドキュメント検索システムでは，120個の単語を4096ビットのビットベクタに符号化している．用いるハッシュ関数は10個であり，ビットベクタの4096ビットのうち，大体，120×10＝1200ビットがセットされることになる．

ある単語がドキュメントに含まれるかどうかを判定するには，その単語をビットベクタを作ったときと同じハッシュ関数にかけ，全てのハッシュ値に対応するビットが1であったときに“可能性有”とする．

この場合，たまたま全てのビット位置がほかの単語によってセットされており，誤って陽性の判断をする可能性があるが，このことがシステム全体に深刻な影響を与えることはない．なぜならその確率が低く $1/3^{10}$ 程度であり，また，ある単語について間違った判断をしたとしても，それは多くの単語のうちの一つであるから致命的なミスにはならないからである．



(a) ビットベクタ

(b) ビットベクタの‘1’のセット

(c) 全単語によるビットベクタのセット

図-3

4.3.2 性　能

ドキュメント検索システムをフルセットのCM-2上で動かした場合，2G Bytesのデータベースに対する100単語の質問が必要とする時間は72 msecと報告されている[9]．重み付きの関連性フィードバックを用いたシステムであるので，検索の質が高く使いやすいものになっている．しかし，上記のシステムは全てメモリ上で処理することを前提として設計されているので，2G Bytesを超えるようなデータベースを扱うことはできない．そこで，より大きなデータベースのために二次記憶システムを使ったドキュメント検索システムが開発されている．そこでは，ビットベクタを二次記憶上で作り，繰り返しメモリ上にロードしてくる方法をとっている．ユーザと会話的処理をするには遅過ぎるので，バッチ処理の形式をとっている．このシステムで，

- ドキュメント数 3200 万個
- ドキュメントサイズ 128 単語
- データベースサイズ 64 G Bytes
- 1質問当たり単語数 100

であるようなデータベースを扱う場合の性能は，40の質問に対する応答時間が142秒と報告されている．

4.4 タンパク質シーケンスマッチング

分子生物学，会話認識，暗号学など多くの分野で，記号のシーケンスを比較して相互関係を計算するという共通した問題がある．また，単なるシーケンスの比較ではなく，シーケンスの中の最も似通った部分を探す場合や，さらにその部分が不連続であるような場合もある．このような問題はサブシーケンスマッチング問題と呼ばれ，コネクションマシンはこのような問題を従来のコンピュータより遥かに高速に実行することができる[10]．

サブシーケンスマッチング問題は，次のように定義される．

集合 F に属する記号で作られた二つのシーケンス A, B が与えられている．

$A=(a_1, a_2, \cdots, a_n)$, $B=(b_1, b_2, \cdots, b_m)$, $a_i, b_j \in F$

このとき，次のようなサブシーケンス A', B' を見つける．

$A'=(a_{i1}, a_{i2}, \cdots, a_{iz})$,

$B'=(b_{j1}, b_{j2}, \cdots, b_{jz})$,

$1 \leqq i_1 < i_2 < \cdots < i_x \leqq n$,

$1 \leqq j_1 < j_2 < \cdots < j_x \leqq m$

ただし，A', B' は比較関数 C (A', B') を最大にするものとする．C の値は a_{it}, b_{jk}, 及び，A' と B' に入らなかった A と B 内の記号（ギャップ）の数によって定まる．

4.4.1 アルゴリズム

前節で述べた二つのシーケンス A, B について，H_{ij} なる行列を考える．この行列の H_{ij} なる要素の意味は，直観的には，A の (a_1, a_2, …, a_i) なるサブシーケンスと B の (b_1, b_2, …, b_j) なるサブシーケンスを比較したときの，比較関数Cの最大値ということである．この $H_{i,j}$ は，次のようにして再帰的に計算できる．

$$H_{i,j} = \max \left\{ \begin{array}{l} 0 \\ H_{i-1,j-1} + s(a_i, b_j) \\ H_{i-1,j} + w \\ H_{i,j-1} + w \end{array} \right\}$$

ただし，w はギャップ定数と呼ばれ，ギャップの増加によってCの値を減少させる働きをする．一般には負である．また，$s(x, y)$ は要素 x と y の相関関数である．

この式からわかるように，H_{ij} の計算には $H_{i-1,j-1}$, $H_{i-1,j}$, $H_{i,j-1}$ の三つに対する依存関係しかない．このため，効率良く並列度を引き出すことができる．

4.4.2 実装手法

具体的なインプリメンテーションでは，一つのプロセッサは行列 H の1行を受け持つ．各プロセッサには初めに，シーケンスの要素が適切にロードされる．プロセッサの処理は，まず最初のステップで一番左の列の要素を計算し，次のステップでその右の列，というように進んでいく．**図-4** に示すように，処理中の要素を表す斜めの直線が，時間とともに右へシフトしていく．全体の処理に要するステップ数は，図から分かるように $m+n-1$ である．

4.4.3 性　能

文献 10) によれば，上記のような行列計算一回に要する時間は，クロック 7 MHz, プロセッサ数 65536 個の CM-2 上では 700 μsec であり，この数字は，ベクトル演算の回数で比較するとシングルプロセッサの CRAY X-MP の 70〜90 倍とされている．

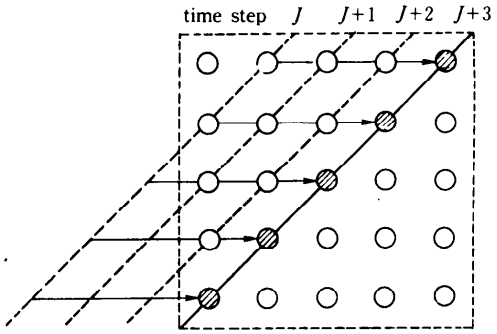


図-4 シーケンスマッチの超並列処理

5. 超並列関係データベース処理

関係データベース処理はきわめて負荷が高く，その高速実行は現時点においても大きな研究課題の一つといえる[14]．特に SIS に代表される統計データベース処理応用での*関係データベース (RDB) 高速化の要求は強い．著者らはコネクションマシン上で RDB 処理をいかに高速に実行できるか実際に実装し評価を行っているので報告する[11)~13)]．

関係データベースでは，データがタプルと呼ばれる単位で管理される．一つのタプルはいくつかのデータフィールドからなるが，ある処理に着目した場合には，一つのデータフィールドのみが検索や比較の対象となることが多い．このような処理の主役となるようなフィールドをキー属性と呼ぶ．

ある情報を表現するためのタプルの集合体を，リレーションと呼ぶ．関係データベースでは，このリレーションに対する各種の操作機能が提供される．

5.1 関係データベースにおけるジョイン処理

関係データベースの基本演算の中でも，ジョイン処理は特に重要なものであり，同時に，負荷が大きい処理でもある．これは，**図-5** に示すように，二つのリレーション R, S に対し，同じキー属性同士のタプルをつきあわせて，その直積であるリレーション J を得る演算とみなすことができる．

ジョイン処理のアルゴリズムについては，さまざまな手法が提案されているがその代表的な技法にハッシュによるものがある[15]．ハッシュジョインとは，キー属性を元にタプルをクラスタ分け

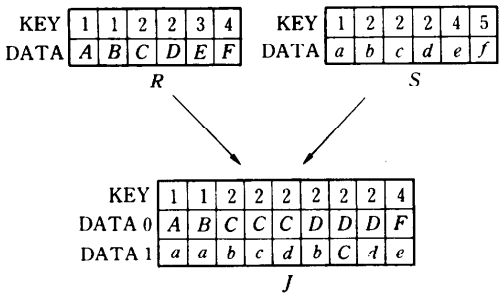


図-5　関係データベースにおけるジョイン演算

する方法である．具体的には，キー属性の値を*ハッシュ関数にかけ，そのハッシュ値によってタプルをいくつかのクラスタに分ける．キーの比較はクラスタ内だけで行えばよいので，全体としての比較回数は大きく減少する．

5.2 実装手法

詳細は文献[11)~13)]に譲り，ここでは処理の流れの概略を示す．

i）　最初に，**図-6** に示すように R のタプルは複数のクラスタに分割される．クラスタリングはおのおののタプルのキー属性にハッシュを施した値に従ってなされ，同一のハッシュ値をもつタプルは，同一のクラスタに含まれることになる．

この際，タプルはコネクションマシンの汎用通信機構を使って適切なクラスタ（プロセッサに相当する）へと送信される．コネクションマシンの送信プリミティブには Paris ライブラリに send-to-queue なる命令が用意されており，ハイパキューブを利用した通信が行われる．送信後のデータの配置の様子を図-6 に示す．また，到着したメッセージの数はキューごとに用意されたカウンタに記録される．

ii）　次に，それぞれのクラスタ，すなわちそれぞれのプロセッサ内でのローカルなジョイン処理を行う（**図-7**）．ここでは，クラスタリングによって，一つのプロセッサの処理するタプルの数は十分に小さいことが期待できるので，最も簡単なジョインアルゴリズムであるネストループアルゴリズムを採用する．ネストループジョイン処理手法は，着目すべきタプルのインデックスを表す変数 IS, IR を使った二重ループ構造の基本的な処理手法である．各プロセッサが並列に処理を行うが，最も多くタプルを有するプロセッサによって全体の処理時間が決定される．

iii）　一つのプロセッサに一つのタプルが格納されるように，通信ネットワークによってタプルの置換を行い，図-5 のような最終結果を求める．

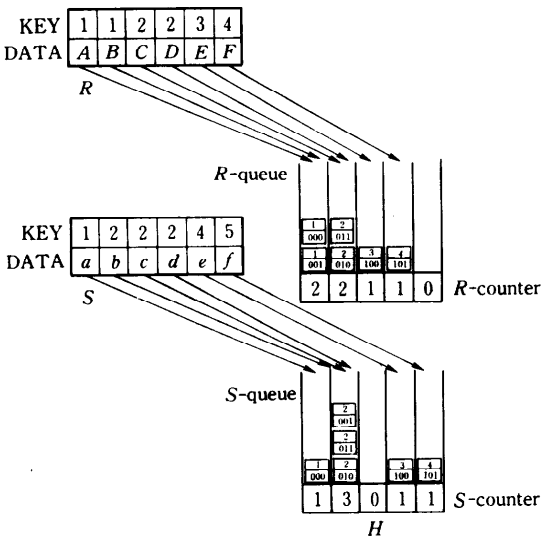


図-6　ハッシュ値に基づくプロセッサ空間内でのタプルのクラスタリング

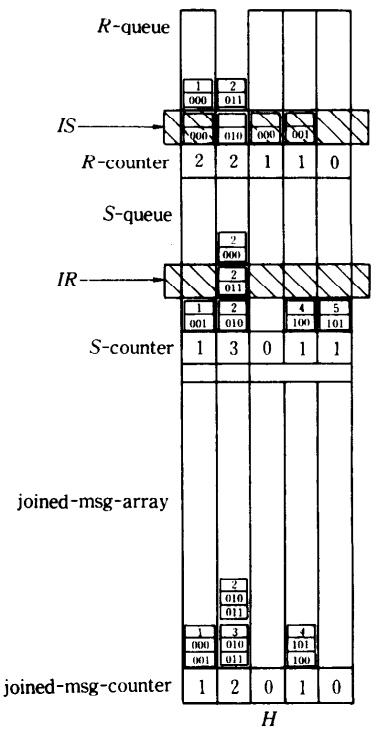


図-7　ネストループアルゴリズムによるローカルジョイン処理

また，実装したアルゴリズムでは効率上の理由からタプルのデータの実体は送信せずにポインタ処理するようになっており，最後にデータの実体を通信ネットワークによって抽出する．

5.3 性　能

次のような条件の下で性能評価を行った結果を示す．コネクションマシンプログラミングのためのプリミティブを提供する PARIS ライブラリは一つの物理プロセッサが複数の仮想的なプロセッサ (*VP*) をシミュレートする機構を備えており，仮想プロセッサ数と物理プロセッサ数の比を *VP* 比と呼んでいる．

● 使用したハードウェアは，クロック 6.7 MHz プロセッサ数 8192 個のコネクションマシン CM-2.

● タプルのサイズは，キー属性 2 バイト，データ 13 バイトの計 15 バイト長．

● ジョイン前のリレーションのプロセッサへのロード時間は測定時間に含まれない．

● 選択率 100% について，*VP* 比を 1，2，4，8 と変化させて測定を行った．

● 元となるリレーションでのキー属性値はユニークであり，その配列はランダムである．

● 測定結果に示される「タプル数」とは，おのおののリレーションのもつタプル数である．つまり，タプル数が 1024 ということは，1024 タプル× 1024 タプルのジョインで 1024 タプルが生成されることを意味する．

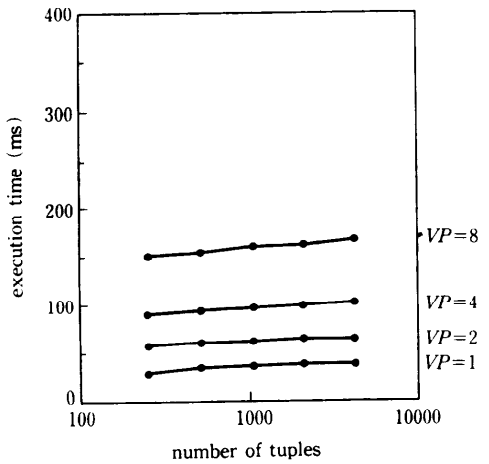


図-8 ハッシュジョインの性能

図-8 のグラフから，逐次マシン上の最も簡単なジョインアルゴリズムがタプル数の二乗に比例する多大な処理時間を必要とするのに対し，コネクションマシン上におけるジョイン処理では，タプル数に対して定数に近い時間しか必要としないことが分かる．これは，一般的にデータパラレルマシンにおいては全てのデータ要素に対して同時にオペレーションを行うことができるため，処理に必要な時間はデータ数にほとんど依存しないことによる．

6. おわりに

SIMD 型商用超並列マシンとして主としてコネクションマシン (CM-2) をとりあげ，その応用面から有効性を探ってみた．紙面の都合上わずかの応用しかとりあげることができなかったが，コネクションマシンは執筆時点で 50 以上のサイトで稼動しており，多彩な利用が模索されている．もっとも現時点では模索段階であり，詳細な性能評価データ，特に，多数のプロセッサ間での通信コストがどの程度であるか，あるいは，ベクトルスーパコンとの比較などについては報告されていない．今後の研究成果が待たれる．コンパイラなどプログラミング環境はいまだ不十分なところも多く，また，アーキテクチャ的にも改善の余地が大きく残されており，数値計算アプリケーションに関しては比較的長い積み重ねをもつベクトルプロセッサに対しどの程度優位性を立証できるか，今後が興味深い．また，そのほかの非数値応用も有望と考えられるものが多々存在するが MIMD に比べて単純になったとはいえ，SIMD マシンのプログラミングにおいてはプログラマはアーキテクチャを念頭におくことが不可欠であり，現時点ではいまだ記述は容易とはいえず，また，その記述能力も十分ではなく，プログラム生産性を向上するツールの開発が当面の課題といえよう．

参考文献

1) Hillis, D. W.: The Connection Machine, MIT Press (1985).
2) 喜連川訳著「コネクションマシン：第 2 部」パーソナルメディア (1990).
3) Steele, G. L. Jr. and Hillis, D. W.: Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing, Proceedings of the 1986 ACM Conference on Lisp and Functional Programing

(1986).
4) Sabot, G. W.: Paralation Lisp Reference Manual, Thinking Machines Technical Report PL 87-11 (1988).
5) Sabot, G. W.: The Paralation Model: Architecture-Independent Parallel Programming, MIT Press (1988).
6) MPL Reference Manual, Maspar Computer Corporation, PN 9302-0000 (1990).
7) MPF Reference Manual, MasPar Computer Corporation, PN 9300-9003-00 (1990).
8) Introduction to Data Level Parallelism, Thinking Machines Technical Report TR 86.14 (1986).
9) Stanfil, C. and Kahle, B.: Parallel Freetext Search on the Connection Machine system, Communications of the ACM, Vol. 29, No. 12 (Dec., 1986).
10) Jones, R. et al.: Protein Sequence Comparison on the Connection Machine CM-2, Computers and DNA, SFI Studies in the Sciences of Complexity, Vol. Ⅶ (1989).
11) 松本，喜連川：データパラレルソートマージジョインアルゴリズムとコネクションマシンによるその評価，電子情報通信学会コンピュータシステム研究会，CPSY 90-80 (1990).
12) 松本，喜連川：データパラレルハッシュジョインアルゴリズムとコネクションマシンによるその評価，情報処理学会計算機アーキテクチャ研究会 ARC 85-7 (1990).
13) Kitsuregawa, M. and Matsumoto, K.: Massively Parallel Relational Database Processing on the Connection Machine CM-2, Proc. of the 2nd International Symposium on Database Systems for Advanced Applications (1991).
14) Ozkarahan, E.: Database Machines and Database Management, Prentice-Hall, N. J., U. S. A. (1986).
15) Kitsuregawa, M. Tanaka, H. and Moto-oka, T.: Application of Hash to Data Base Machine and Its Architecture, New Generation Computing, Vol. 1, No. 1, pp. 66-74 (1983).
16) C* Reference Manual, Version 4, Thinking Mchine Corp. (1987).
17) *Lisp Reference Manual, Version 5, Thinking Machine Corp. (1988).
18) CM Fortran Reference Manual, Version 5.2, Thinking Machine Corp. (1989).
19) Patterson, D. et al.: A case for Redundant Arrays of Inexpensive Disks (RAID), Proc. of SIGMOD, pp. 109-116 (1988).

(平成3年1月 31 日受付)

用語解説

関係データベース

1970年に E. F. Codd によって提案されたデータモデルに基づくデータベースシステム．一つのレコードをタプル，タプルの集合をリレーションと呼ぶ．リレーションに対する操作が関係演算として定義されており，これにより種々の問合せ処理が可能となる．関係データベースは従来のネットワークモデル，木構造モデルに比べ強固な論理的基盤，高度な非手続的インタフェース等優れた特長を有し現在ではパソコンからメインフレームに至るまでの利用が広く浸透しつつある．

ディスクアレイ

近年，磁気ディスク装置の小型化ならびに低価格化が進む中で，8インチ以上の大型ディスクを，3～5インチの小型ディスクを複数台アレイ化することで置き換えることが提案されている．特に，1988 年の D. Patterson の RAID と呼ばれる冗長化ディスクを利用したディスクアレイ構築法（その構成法によりレベル1からレベル5まで5通りの手法が提案されている）に関する論文がきっかけと見なされることが多い．それ以前にも動画像の処理を目的として多数のディスクを同期回転させたディスクアレイが多数商用化されたが，近年の超小型化が一層の拍車をかけたものと考えられる．我国においても国産メーカよりスーパコンピュータ用のディスクアレイが発表されている．

ハッシュ関数

ある情報を格納する際，その情報のキー属性に関し何らかの関数を施し，その結果得られた値をアドレスとして，その位置に当該情報を格納する手法が考えられる．本手法をハッシュ法（二次記憶を対象とする場合，直接編成法）と呼び，用いられる関数をハッシュ関数と言う．ハッシュ法はロードファクタが低い場合最も高速な探索技法である．異なるキー値が同一のアドレスに写像されることをコリジョンと呼び，元のキー分布によらずなるべくコリジョンの少ない関数が望まれる．除算法，乗算法，折りたたみ法，自乗法など種々の方式が提案され，いくつかの方式を融合して利用することが多い．

喜連川 優（正会員）

昭和30年生．昭和53年東京大学工学部電子工学科卒業．昭和58年同大学院情報工学専門課程博士課程修了．工学博士．同年，東京大学生産技術研究所講師．現在，同研究所助教授．並列コンピュータアーキテクチャ，データベースマシン，データ工学などの研究に従事．電子情報通信学会，電気学会，IEEE, ACM 各会員．

湯浅 太一（正会員）

1952年神戸生．1977年京都大学理学部卒業．1979年同大学理学部修士課程修了．1982年同大学博士課程修了．同年京都大学数理解析研究所助手．1987年豊橋技術科学大学講師．1988年同大学助教授となり現在に至る．理学博士．記号処理システムと並列処理に興味を持っている．著書「Common Lisp 入門（共著）」他．訳書「プログラミング言語 Turing（共訳）」他．