

(1991. 11. 21)

並列処理言語 SPLAN を 用いた並列プログラミング

岡本 秀輔 飯塚 肇

成蹊大学大学院
工学研究科情報処理専攻

SPLAN は特定の並列処理計算機に依存することなく並列プログラムを書くことを目的として設計されたメッセージ通信型の並列処理言語である。この並列処理言語は、PASCAL プログラムを並列実行の単位プロセスとし、各プロセスは仮パラメタとして宣言されたチャネルに対して同期式と非同期式の通信を行いながら、非決定性を処理する構文を使って効率よく実行を進める。全てのプロセスは動的に生成されるが、それと同時にプロセスの結合関係が明らかになるように、チャネル割当ても行われる。本報告では、この言語の設計思想及び特徴そしてこの言語を用いた基本的な並列プログラムについて述べる。

Concurrent Programming Using SPLAN

Shusuke OKAMOTO Hajime IIZUKA

Department of Information Sciences, Seikei University
3-3-1 Kichijoji Kita-machi, Musashino, Tokyo, 180 JAPAN

SPLAN is the parallel processing language using message passing. It is designed to be the concurrent programming language which is independent of the processor architecture. Its program consists of concurrent processes. Each process is generated dynamically and has both synchronous and asynchronous communications ability. This paper describes the language features as well as concurrent programs using SPLAN.

1. はじめに

近年のVLSI技術の進歩にともない様々な並列処理計算機が作られるようになってきている。それにともなって並列処理用プログラミング言語も設計されているが、それらは実現される計算機に依存していることが多く、いまだに主流となる汎用の並列プログラミング言語は存在しない。

これら計算機依存の言語で並列プログラムを作る場合、問題のモデル化の時点から実行される並列計算機の特徴を生かすための方法が取り入れられてしまい、移植性の無いものになってしまう。また、純粹にモデル化された並列アルゴリズムがあったとしても、このような言語を使う場合、その計算機の特徴をよく理解しなければプログラムを作ることができず、結果としてその計算機についてかなり詳しく知らなければ使うことが出来なくなってしまう。

一方、並列処理言語ではなく現在広く使われているCやFORTRANなどの逐次処理言語でプログラムを記述し、並列化コンパイラにより並列処理計算機を使う方法も考えられ、この分野の研究も多くなされている。この考え方はソフトウェアの資産の問題として良い答となっているし、ある並列計算機固有の知識がなくてもプログラムが作れるという点でかなり良いと思われる。しかし現在では、オブジェクト指向により自然にモデル化できるものはそのまま表現しようという流れもあり、このオブジェクト指向を追求すれば自然と並列モデルというものがプログラミングの中に入ってくることも予想されるので、この並列化コンパイラの分野だけに期待する訳にも行かない。

並列処理言語SPLAN (Small Parallel-processing LANGUAGE) はこのような現状において、特定の計算機に依存することなく容易に汎用の並列プログラムが記述できることをめざして設計された並列処理言語である。本報告では、SPLANの設計思想及び特徴について述べた後、この言語を使った基本的な並列プログラム

について例を掲げて説明していく。

2. 設計思想

2.1 逐次部分と並列部分

プログラマが陽に並列性を記述する言語で重要なことは、逐次部分と並列部分とを明確に分けることである。また、これら分けられた部分は実際の実行にも反映されなければならない。

例えばoccam2⁽¹⁾⁽²⁾での並列プログラミングでは、言語の文法上かなり細かに逐次部分と並列部分を指定することが出来るが、実際の実行ではこれがそのまま並列実行に結びつくわけではない。従って、プログラマはいつも実際に並列に実行される部分と疑似的な並列(つまり平行実行)を意識していなければならない。

SPLANでは逐次部分のことをプロセスと呼び、粗粒度のものを想定する。こうすることにより、プログラマは大まかな並列を記述して、並列モデルを実現していくことができる。実行される計算機により、細粒度の並列化が必要ならばそれを並列化コンパイラに任せる形をとることにより対応できる。この場合も粗粒度の並列プロセスが指定されているので、コンパイラは効率の良いコードを生成することができるであろう。

2.2 逐次処理言語との関係

並列処理言語と言っても、逐次処理部については従来のプログラミングと何等変化をつける必要はない。したがって、SPLANではPASCALの拡張という形をとった。一般の使われ具合からすればC言語を用いたほうが良いと思われるが、Cを用いなかったのは以下の理由による。

Cは関数が単位でありそれらをまとめた逐次部分をひとかたまりとするには、ファイル別にするという方法になる。並列処理で使われている多くのC言語では、複数のmain()関数を別々のファイルに記述するという方法がとられているが、これではプロセスごとの管理がファイルの管理となってしまうし、関数のためのファイ

ル分割とプロセスのためのファイル分割が混ざり煩雑となってしまう。

また、関数に修飾子を等をつけこれをプロセスとして扱う方法もあるが、これもプロセスと関数を同レベルで定義することになり、上述と同様の問題が持ち上がる。

このようにC言語を並列処理言語として拡張しようとする、構文や意味的に多少の無理が生ずるために、多くの人に受け入れられるものとはならない。これに対して、PASCALはプログラム全体が program という記述により元々一つの閉じたものとなっているので、これをそのままプロセスとすればすっきりした拡張ができる。またプログラムの引き数 (input, output等) はファイルを示しているが、この部分も手続きや関数の引き数と同様に型を指定してやれば他の情報 (メッセージ用のチャネル等) を違和感なく渡してやることができる。

2.3 プロセス間通信

並列プログラムを構成するプロセスは、何らかの方法で他のプロセスと情報を交換しなければならない。このプロセス間通信の方法は、プログラムの作り方に大きく影響を与えるだけでなく、その言語を実現する計算機をも限定してしまう。したがってプロセス間通信の方法の選択は重要である。

プロセス間通信の方法としては、一般に共有変数、モニタ、メッセージ通信、遠隔手続き呼び出しがあげられる⁽³⁾。また最近では、Linda⁽⁴⁾に代表されるようなタブルスペースによる方法もある。

SPLANは、この中でメッセージ通信を採用している。その理由としては様々な計算機における実現の容易さを重視した。この実現の容易さは効率の良さにもつながる。

まず、共有変数とモニタであるが、これらは共有メモリ型の計算機よりの概念であり、効率よく分散メモリ型の計算機で実現するのは難しい。また、計算機を構成するプロセッサの台数が今後増えて行けば、共有メモリ型の計算機も

局所メモリを持たざるを得ないことは容易に想像できるので、共有変数とモニタは排除される。残りの3つの方法は、どれも長所と短所を持っており一概にどの方法が一番良いかを論ずることは難しい。しかし、遠隔手続き呼び出しはそれ自体が高機能すぎる点とタブルスペースは分散メモリ環境におけるタブルの管理という点でどちらも効率の良い実現がメッセージ通信よりも複雑な機構が必要であると思われるので、SPLANではメッセージ通信を採用することとした。

2.4 プロセスの生成と終了

プロセスがいつ実行を開始し、いつ終了するのかという点も並列処理言語では重要な点である。この方式は、プロセスのプロセッサ割当ての方法に影響を与え処理効率に大きく関わってくる。

考えられる単純な方式は、プログラムのはじめに全てのプロセスが実行を開始し、最後に全てが終了するものである。ただし終了に関しては、あるプロセスが終了するときに、通信をする可能性のある相手プロセスに伝えておけば問題はないので特に終了を揃える必要はない。

この実行開始を揃える方式では、静的なプロセッサ割当てにより実行時にどのプログラムコードをどのプロセッサにロードしておけば良いかが決められるので、プロセッサのメモリ負担も軽くすることができ効率の良い実行ができる。しかし、アルゴリズムが多段階の処理により構成されていて、その各段階におけるプロセスの結合関係が変わってくる場合には、多くの問題がでてきてしまう。またこのようなアルゴリズムをプログラマが意図的に避ける傾向になってしまうかも知れない。

SPLANでは、様々なアルゴリズムに柔軟に対応し、かつ容易にそれが記述できることをめざし、各プロセスは動的に生成されることとした。終了に関しては、前述のように特に揃えなくても問題を回避する方法があるので、プロセス各々が独立に終了することとした。

2.5 メッセージ通信に必要な機能

SPLANでのメッセージはoccam2で見られるようなチャンネルを通して通信されるものとした。これにより個々のプロセスは、自分に局所的な名前のチャンネルに対してメッセージを書いたり読んだりすることが出来るので、手続き関数レベルのモジュール化と同様にプロセスレベルでも高いモジュール化を実現することができる。

チャンネルによるメッセージ通信を行う場合、そのチャンネルが扱うデータの型を指定できる方が安全な通信を行うことができる。そこでチャンネルで扱うデータの種類をプロトコルとして宣言し、このプロトコルをチャンネルの型として定義することにする。

通信方法としては、同期通信の他に非同期通信も行えるようにする。同期通信と非同期通信は、どちらも効率の良い並列プログラムには重要なものである。もちろんどちらか1つをサポートしていれば他方を実現するのはそれほど難しくないが、頻繁に使う両者を言語のプリミティブとしたほうがプログラマの負担を減らすので良い。また通信効率を上げるために、通信による非決定性を処理するための構文を使えるようにする。

また、明確に分けられたプロセス間において、どのプロセスとどのプロセスが通信をしているのかが分かりやすく記述できることはプログラムの可読性を高くする。これは従来の並列処理言語ではあまりみられないことであるが、ある問題を並列プロセスにモデル化した場合、それをそのままプログラムにコーディングするには、この機能が優れていることが非常に重要な点となってくる。SPLANではこの機能を実現する構文を追加し、プロセスの結合関係の明確化をはかる。

3. 言語の特徴

3.1 概要

SPLANのプログラムは、双方向チャンネル

を通して一対一通信を行うプロセスからなる。各プロセスは動的に生成され、PASCALの局所的に宣言された手続きまたは関数、そして外部宣言された標準関数を用いて逐次的に実行を進める。そのために、コンパイラまたはプロセス管理プログラムが、このプロセスを単位としてプロセッサに割当てることができる。

プロセスは、頭書きの仮パラメタとして宣言され、局所的な名前をもつチャンネルを使って通信することができる。2個のプロセスをつなぐ実チャンネルは、プロセス生成時に"gen文"によって割当てられる。これによってプロセスの通信関係は明らかとなり、加えてプログラムはプロセスを他のプロセスとは独立に定義することができる。

SPLANは、同期式と非同期式のプロセス間通信を備えている。そして安全に通信を行うために、使用される全てのチャンネルはoccam2と類似したタグ付きプロトコルを持つ。

プログラムの実行は、"main"というプロセスから始まり、他のプロセスを"gen文"をつかって生成する。"main"を含めて全てのプロセスは、いつでも"gen文"により他のプロセスを生成することができる。全てのプロセスは並列に動作し、その終了は非同期に行われる。つまり、全てのプロセスが終了したときがそのプログラムの終了となる。

逐次処理で使われる標準関数、型、定数は、C言語のものを使えるようにしてもよい。この場合は、関数の引数と戻り値の情報、型、定数をあらかじめ定義しておく。

3.2 表記法

SPLANのプログラミングとは、複数の逐次プロセスを記述することである。この逐次の部分は前述のように、複数のPASCALプログラムを記述する形をしている。しかし、PASCALでは記述力に欠けるところがあるので、その部分については上位互換を保ちながら多少の拡張がなされている。ここでは、並列処理用に拡張された部分に焦点をしばり説明する。

3.2.1 プロセスの宣言

プロセス一つは基本的に P A S C A L のプログラムを宣言する。ただし、宣言の頭書きが図 1 のように変わる。

```
process プロセス名 ( 仮チャネル宣言のリスト )
...
begin
...
end.
```

図 1 プロセスの宣言

3.2.2 プロトコル宣言

通信に使われるチャネルは全て型を持ち、その型がチャネルの扱うデータの型を現す。プロトコル宣言は、そのチャネルの型名の宣言である。ここでは、扱われるデータの型名とそれに対応するタグを宣言する。このタグは通信を行うときに使われ、どの種類のデータを扱うかを指定するのに使われる。

表記は、図 2 のように構造体の宣言と類似な形をしている。

```
type プロトコル名 = protocol
    タグ : 型名 ;
    タグ : 型名 ;
    ...
end;
```

図 2 プロトコル宣言

型名には標準の型や構造体の型といったデータ型のほかに、signal という型を記述することができる。これは通信専用の型でデータを送ることなく相手プロセスに合図を送るときに使われる。

文字列のように予め通信量が決まっていないような、可変長の通信を行いたい場合は、以下のようにタグの次にデータの最大長を指定しておく。

タグ (最大長) : 型名

例えば、2 種類の整数と 1 種類の実数を扱う

ようなチャネルの型を、“pipe” という名で宣言する場合は図 3 のように宣言する。

```
type pipe = protocol
    int1, int2 : integer;
    real1 : real
end;
```

図 3 プロトコル宣言の例

このように 2 個の整数それぞれにタグをつけておけば、1 個目と 2 個目の通信でデータの追い越しがあっても正しく受信を行うことができる。

3.2.3 通信

通信は、基本的には図 4 のように記述される。

```
チャネル ! タグ : 式
チャネル ? タグ : 変数
チャネル !! タグ : 式
チャネル ?? タグ : 変数
```

図 4 通信文 1

ここで“!” は非同期送信を表し、“?” は対応する受信を表す。“!!” や“??” の場合は、この通信は同期を伴うことを意味する。

送信側と受信側でタグがマッチしたときにはじめて通信が行われる。

また、タグの対応する型が signal ならば図 5 のように記述されるになる。

```
チャネル ! タグ
チャネル ? タグ
チャネル !! タグ
チャネル ?? タグ
```

図 5 通信文 2

signal 型は、実行の同期や同期を取らないまでも相手がどこまで実行を進めたかを知するために使うことができる。

可変長の通信を行う場合は図 6 のようにデー

タ長に関する式や変数を指定して記述される。

```

チャンネル ! タグ ( 式 ) : アドレス
チャンネル ? タグ ( 変数 ) : アドレス
チャンネル !! タグ ( 式 ) : アドレス
チャンネル ?? タグ ( 変数 ) : アドレス

```

図6 通信文3

送信側は送るデータの大きさとデータの先頭アドレスを指定し、受信側では、括弧内の変数にその大きさが代入されデータの大きさを知ることができる。ただし、データのアドレスは標準のPASCALでは得ることができないのでこの部分に関しては、modula-2⁽⁵⁾の様に拡張されている。

3.2.4 非決定性

複数のプロセスからの通信を待っている場合に、各プロセスの実行状況により非決定性が生まれる。この非決定性を処理するために、SPLANではCSP⁽⁶⁾の選択コマンドに似たalt文を持つ。これは図7のように記述される。

```

alt
    受信 -> 文 ;
    受信 -> 文 ;
    ...
else 文
end

```

図7 alt文

この文は、受信が可能なところの通信を行った後、対応する”文”を実行する。もし受信可能なものがない場合はelse部を実行する。else部はなくてもよいが、この場合どれかの受信が成功するまで繰り返し調べられる。

また、受信に条件をつけたい場合は、受信のすぐ後に”& 論理式”を記述することもできる。

3.2.5 プロセス生成

プロセスの動的生成は図8の”gen文”に

よって行われる。

```

gen ( 生成用チャンネルリスト )
    プロセス名 ( 実チャンネルリスト ) ;
    ...
    プロセス名 ( 実チャンネルリスト )
end

```

図8 gen文

この文は、プロセスの生成とともに実チャンネルの割当て、つまりプロセスどうしの連結を行う。

生成側のプロセスで、新しく生成されるプロセスとの通信に使用しないチャンネルは、”生成用チャンネルリスト”で宣言され、プロセスを連結するために二つのプロセスの”実チャンネルリスト”に現れる。その他のチャンネルは生成側のプロセスとの通信に使われるので、一度だけ”実チャンネルリスト”に現れる。また、同じ種類のプロセスを複数生成したい場合は、プロセス名の後に繰り返しを示す記述が出来る。

例えば二つの仮チャンネル引き数をとる node という名のプロセスが、main プロセスを含めて図9のようにをpipe型のチャンネルでリング構造に結合している場合、

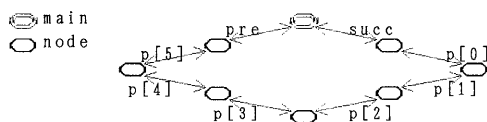


図9 リング結合の例

プロセス main における”gen文”は、図10のようになる。

チャンネル配列 p[] は、後の mainの通信に関係ないので、各 node の引数として2度ずつ現れ、チャンネル pre と suc は、1度だけ引数とし現れている。

”gen文”はチャンネルのスコープを犯さない限りどのプロセスのどこで使われてもかまわない。

```

type arraypipe = array[0..5] of pipe;
...
chan pre,suc : pipe;
...
gen ( p : arraypipe )
  node( suc , p[0] );
  node( p[i], p[i+1] ),[i:0..4];
  node( p[5], pre )
end;

```

図10 リング結合のプロセス生成

4. プログラム例

4.1 木結合型並列ソート

並列ソーティングの簡単な例として、プロセスに2分木を構成させて行わせる方法がある。このアルゴリズムは、まず2分木の根から葉の方向にデータを流し、それをまた集めることによりデータを整理させていくものである。ここではこのような木結合型にプロセスを結合させるプログラムとして静的にプロセスを生成するものと動的に行うものの2種類を示す。

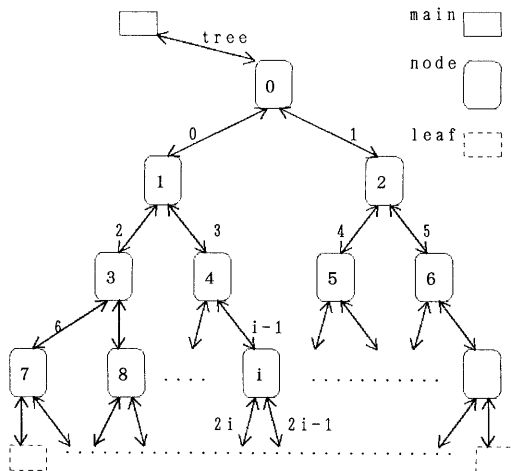


図11 2分木の例

4.1.1 静的なプロセス生成

プロセスは、データの読み込みと他のプロセス制御を行うroot (main) と、木の節を形作る node、そして木の葉の部分となる leaf の三種類により図11のようにネットワークを構成する。

それぞれのプロセスは図12のように振る舞う。ここで hand は、チャンネルの型とする。

```

process leaf( up : hand );
begin
  チャンネル up からデータを受け取る
  quick ソートをする。
  チャンネル up へデータを送る。
end.

process node( up, left, right : hand );
var lflag, rflag : boolean;
begin
  チャンネル up からデータを受け取る
  データの半分をチャンネル left へ、
  残り半部分をチャンネル right へ送る。
  lflag := TRUE; rflag := FALSE;
  while lflag OR rflag do
    begin
      alt
        left からの受信 & lflag
          -> lflag := FALSE;
        right からの受信 & rflag
          -> rflag := FALSE;
      end
    end
  end
  受け取ったデータをマージソートする。
  チャンネル up へそれを送る。
end.

```

```

process main();
chan tree : hand;
begin
  データをファイルから読み込む
  プロセス生成を行う。
  チャンネル hand へデータを送る。
  チャンネル hand からデータを受け取る。
  データをファイルへ書き込む
end.

```

図12 プログラム例1

このプログラムでは、各ノードの通信する配列の大きさが、leaf に近くなるにつれて半分づつに減っている。そこでチャンネル型のhandは、図13のようなプロトコル宣言で可変長の通信を許すように宣言される。

```

hand = protocol
  data(MAXDATA) : integer
end;

```

図13 木結合型ソートのプロトコル宣言

MAXDATA は、このプログラマがソートできる最大数で、定数宣言部に宣言されているとする。

実際に通信を行うときは、

```
up ? data(i):  adr(a[0]);    |# 受信
up ! data(i):  adr(a[0]);    |# 送信
```

の用に行われる。ここで up は プロトコル hand のチャンネル、i は整数変数で通信データの大きさを表し、a[] は整数の配列である。可変長通信の場合に限り送信及び受信の対象となるデータはその先頭アドレスを指定しなければならないので、標準関数 adr() を用いて配列 a[] の先頭アドレスを求めている。

このプログラムの gen 文を図14に示す。

```
const n = ...;
      m = ...;
i : integer;
chan tree : hand;
...
gen( Arc : array[0..n] of hand )
  node( tree,      Arc[0],   Arc[1] );
  node( Arc[i-1],  Arc[2*i], Arc[2*i+1] ),
      [i:1..m];
  leaf( Arc[i] ) , [i:m..n]
end;
...
```

図14 木構造選ソートのプロセス生成

この gen 文の特徴、2 番目の node の繰り返しと leaf の繰り返しの数（つまり定数 m, n）を調整するだけで、簡単に木の大きさを変えられるところである。

4.1.2 動的なプロセス生成

2 分木を構成する場合、データの大きさに応じて木の大きさを変えたい場合がある。4.1 のプログラムをこの形に書き換える場合、プロセス node が、受け取ったデータに応じてさらに自分より下にプロセス node を再帰的に生成するか、自分でソートするかを判断する様にする。プロセス leaf はなくなり、プログラムは図15

のようになる。

```
process node( up : hand );
var lflag, rflag : boolean;
chan left, right : hand;
begin
  チャンネル up からデータを受け取る
  if さらにnodeを生成する then
  begin
    gen()
    node( left );
    node( right )
  end;
  データを半分の半分をチャンネル left へ、
  残り半分をチャンネル right へ送る。
  lflag := TRUE; rflag := FALSE;
  while lflag OR rflag do
  begin
    alt
      left からの受信 & lflag
        -> lflag := FALSE;
      right からの受信 & rflag
        -> rflag := FALSE;
    end
  end
  受け取ったデータをマージソートする。
end

else
  データをソートする。
  チャンネル up ヘデータを送る。
end.

process main();
chan tree : hand;
begin
  データをファイルから読み込む
  gen()
  node( tree )
end;
チャンネル hand ヘデータを送る。
チャンネル hand からデータを受け取る。
データをファイルへ書き込む
end.
```

図15 プログラム例2

この様にした場合、プロセス main は、自分に関係しない node プロセスに関しては、なにも知らなくて良いので、すっきりしたプログラム構造になる。

4.2 Conway のライフゲーム

Conway のライフゲーム⁽²⁾⁽⁷⁾とは、無限の大きさのマトリックス上の点が alive と dead の二つの状態を持つものとし、これら全ての点に対し次のルールを適用し、次の状態（世代）を決定していく一種のシミュレーションゲームである。

ルール：

1. その点が alive で周囲の点に alive が 2 個未満なら dead になる
2. その点が alive で周囲の点に alive が 2 ～ 3 個なら alive のまま
3. その点が alive で周囲の点に alive が 4 ～ 8 個なら dead になる
4. その点が dead で周囲の点に alive が 3 個なら alive になる

コンピュータ上でこのライフゲームを表現する場合、メモリの制限からマトリックスの大きさは有限になる。そして通常はその境界部分より外側は、全て dead と仮定して実現される。ここで紹介するプログラムは、右側の境界と左側の境界、そして上の境界と下の境界がつながったものつまりトーラス状を仮定した。

このプログラムは、プロセス cellmgr をチャンネル配列 ns (north & south) と we (west & east) を用いて $I \times J$ のトーラス状につなぎ、プロセス main を制御用としてチャンネル配列 ToCells を用いて全ての cellmgr とつなぐ。一つの cellmgr はマトリックス全体の $I \times J$ 分の 1 を計算することになる。

次の世代の計算において、境界部分については、8 方のプロセスが持つデータを得なければならない（角の点は、斜め方向のプロセスが持つ状態が必要なため）。そこでまず cellmgr は、自分が扱う 2 次元配列として自分の計算する点のデータ数より縦横それぞれ +2 の大きさで持っておく。例えば自分の計算する点が $m \times n$ ならば $(m+2) \times (n+2)$ の配列を持つようにする。自分の担当のデータを 2 次元配列の $(1,1), (1,n), (m,1), (m,n)$ の 4 点内に割当て、

残った配列の外側一周を通信用のバッファとして使う。1 世代の計算に必要なデータの通信は次のようにして行われる。

1. 点 $(1,n)$ と (m,n) を結ぶ線上の点の状態を右のプロセスへ送る。
2. 点 $(1,1)$ と $(m,1)$ を結ぶ線上の点の状態を左のプロセスへ送る。
3. 右から送られて来たデータを $(1,n+1)$ と $(m,n+1)$ を結ぶ線上の点に入れる。
4. 左から送られて来たデータを $(1,0)$ と $(m,0)$ を結ぶ線上の点に入れる。
5. 点 $(1,0)$ と $(1,n+1)$ を結ぶ線上の点の状態を上のプロセスへ送る。
6. 点 $(m,0)$ と $(m,n+1)$ を結ぶ線上の点の状態を下のプロセスへ送る。
7. 上から送られて来たデータを $(0,0)$ と $(0,n+1)$ を結ぶ線上の点に入れる。
8. 下から送られて来たデータを $(m+1,0)$ と $(m+1,n+1)$ を結ぶ線上の点に入れる。

1 ～ 8 はデッドロックが起こらないように順番を変えることができる。重要なことはこのように 4 方のプロセスと通信を行うことにより、計算に必要な斜め方向のプロセスが持つデータをも得ることができ、同時に全ての cellmgr が世代ごとの計算に同期を自動的に取ることができる点である。このような特徴は、この問題に限らず多くの分野の並列化で応用できる手法である。

各プロセスを news 型のチャンネルでつなぎ、制御用のチャンネルの型を ctrl とするとプロセス生成は図 16 のように記述される。

```
chan ToCells : array[1..I,1..J] of ctrl;
...
gen( ns, we : array[1..I,1..J] of news )
    cellmgr( ToCells[i,j],
              ns[(i mod I)+1,j],
              we[i,j], we[i,(j mod J)+1] ),
              [i:1..I][j:1..J]
end;
```

図 16 ライフゲームのプロセス生成

2 分木の時と同様に、定数 I, J を適当に変え

れることにより、トーラスの大きさを調節することができる。

5. むすび

並列処理言語 S P L A N の設計思想及び特徴そして簡単なプログラミング例について述べてきた。プロセスを動的に生成し自由にプロセス結合の形を作れるということは、計算機にマッピングする効率よいプロセッサ割当てが必要になってくるであろう。しかし、並列処理計算機における通信性能は急速に良くなっており、近い将来においてはプロセッサの結合状態に関係なく完全結合を仮定できるような計算機の登場も予想され、この問題もそれほど難しいものではなくなるであろう。

参考文献

- (1) INMOS Ltd, 'occam2 Reference Manual', Prentice-Hall(1988)
- (2) G. Jones, M. Goldsmith, 'Programming in occam2', Prentice-Hall(1988)
- (3) 宮村 勲, 「並列処理言語」, マグロウヒル (1986)
- (4) N. Carriero, D. Gelernter, 'How to write parallel programs', The MIT Press
- (5) N. Wirth, 'Programming in Modula-2', Springer-Verlag(1985)
- (6) C. A. R. Hoare, 'Communicating Sequential Processes', Comm. ACM, 21(8), 666-677(1978).
- (7) W. Poundstone 著 / 有沢 誠 訳, 「ライフゲームの宇宙」, 日本評論社(1990)
- (8) 岡本秀輔, 飯塚 肇, 'メッセージ通信型並列処理言語 S P L A N の設計', '90年 信学秋全大 6-57
- (9) 岡本秀輔, 飯塚 肇, 'A Design and An Implementation of The Parallel Processing Language Using Message Passing', IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 681~684