マルチスレッドによる
メッセージ通信型並列処理言語SPLANの実現

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[1~3]はメッセージ通信型並列処理言語であり、CSP[4]とPASCALに基づいて設計された言語である。この言語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いる。プログラムはPASCALプログラムを並列実行の単位としたプロセスからなり、それぞれは動的に割り当てられる。各プロセスは生成時に割り当てられたチャネルを通して通信を行う。本報告ではSPLANの特徴および、ワークステーションLUNA88k2におけるMachのC Treadsパッケージを用いたこの言語の実現について述べる。

AN IMPLEMENTATION OF SPLAN
USING MULTI-THREADS ON A MULTIPROCESSOR

Shusuke Okamoto　and　Hajime Iizuka

Department of Information Sciences, Seikei University
3-1 Musashino-Shi Kichijoji Kitamachi 3-chome, Tokyo, 180 Japan
Email: okamoto@seikei.ac.jp

SPLAN is a CSP and PASCAL based programming language for parallel computers. It is designed to be the concurrent programming language which is independent of the parallel architecture. Its program consists of concurrent processes. Each process is generated dynamically and can communicate another using channels which are allocated on generation time. This paper describes the language features as well as its implementation on a multiprocessor.

1 まえがき

近年のVLSIの技術の進歩は並列処理計算機の実現を可能とし、実際に様々なアーキテクチャを持った商用機が出現している。しかし、それらの計算機を前にし、特定の応用に対してその計算機の機能を十分に引き出すプログラムを作ろうと考えたとき、それを容易にするソフトウエア環境が未だそろっていないことが大きな問題となっている。この問題は計算機を操る手段であるプログラミング言語といったレベルでもいえる。どのマシンも既存のプログラムのための最適な並列化コンパイラを持たないから、並列処理計算機を効率良く動作させるには新たに並列プログラムを書く他はない。しかし、実際に使われている多くの並列処理言語は、実現されている計算機に強く依存していることが多く、移植性といった面まで考慮に入れられていない。メッセージ通信型並列処理言語SPLAN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並列プログラムが記述できる言語を目指して設計された。

SPLANプログラムはトランスレータによって実現される計算機の特徴を最大限に利用するCのコードに変換され、それがCのコンパイラによりコンパイルされる。そして実現される計算機により実行系がこの時リンクされる。現在、SPLANは商用の分散記憶型マルチプロセッサnCUBE2用および同じく分散記憶型であるTRANSPUTER用のものが実現されており、それらにおいて、SPLANのソースコードを変更することなくコンパイルおよび実行を行うことができる。

本報告では、3番目の例としてLUNA 88k2ワークステーションにおけるMachのマルチスレッドを利用したSPLANの実現について述べる。

2 言語の特徴

この章ではSPLANの特徴について簡単にまとめる。

2.1 概要

SPLANはCSPとPASCALを基に作られた並列処理言語である。プログラムは同期および非同期の通信をおこなうプロセスからなる。一つ一つのプロセスの定義はPASCALと同様の形で行われ、これをプロセッサ割り当ての単位として動的に生成される。

通信は双方向チャネルを通して1対1で行われる。プロセスは、頭書きのチャネルパラメタ（PASCALのINPUT, OUTPUTにあたる）および局所的に宣言したチャネルを用いる。これにより他のプロセスの名前がコード中に現れることがなく、他とは独立にプロセスの記述ができる。

各チャネルはそれぞれプロトコル型を持ち、そのチャネルが扱うデータの種類およびその方法が決められている。

プログラムの実行は、mainプロセスから始まり、gen文というプロセス生成とチャネルの接続を行う文を用いて、並列プロセスを構成していく。プロセスの終了は非同期的に行われ、すべてのプロセスが終了したときがプログラムの終了となる。

また、外部宣言によりC言語などの逐次の標準関数などを利用することができる。ただし、これは関数の引き数と戻り値の情報等を正確に宣言できる場合に限る。

2.2 プロセス

SPLANプログラムはいわばPASCALプログラムの集まりである。プロセスはそのため従来のPASCALに似て、入れ子の手続き／関

数とともに以下のように宣言される。

```
process P(ﾁｬﾈﾙﾊﾟﾗﾒﾀ);
var 変数;
begin 文の並び end.
```

チャネルパラメタは後に述べるgen文により、実際のチャネルが生成時に割り当てられ、通信相手が決まる。プログラマはこのチャネルパラメタをPASCALの入出力ファイルのイメージで用いることにより通信を行う。

2.3 プロトコル定義

通信に使われるチャネルの型であるプロトコルは、大域型定義部において以下のレコード型と同様な形で定義される。

```
type T = protocol
            s1:T1;
            s2:T2;
            ...
            sn:Tn
         end;
```

プロトコルTはT1,T2,...,Tnのデータ型を扱い、それぞれs1,s2,...,snという名のタグがつけられている。このタグを通信を行うときに使用して、データの種類の識別を行う。データ型には標準の型や構造体といったデータ型の他、signal という通信だけの特別な型を指定できる。これはデータを送ることなく相手プロセスに合図をするために使われる。

あらかじ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には、

```
s1(length):T1
```

の用にデータの最大長を指定しておく。

通信は基本的には非同期通信であるが

```
s1: sync T1
```

と、指定することにより同期通信を行うタグとして定義できる。この仕様は文献(1)からの変更点である。以前は通信の同期と非同期の区別は通信文で指定していたが、送信と受信の対応を間違え易いためにこのように変更した。

2.4 通信文

二つのプロセス間の通信は、前述のプロトコル型を持ったチャネルを用いて記述される。送信は

```
ch ! tag1 : exp
```

であり、対応する受信は

```
ch ? tag1 : var
```

となる。ここでchはプロセス生成時に割り当てられた同一のチャネルであり、送信側と受信側でタグがマッチする必要がある。また、expは送られるデータの式、varはそれを受ける変数であり、それぞれタグに対応する型である。

タグに対応する型がsignalならばコロン以降が省略された形をとる。

可変長の通信を行う場合は

```
ch ! tag1(exp):exp
ch ? tag1(var):var
```

のようにデータ長を指定する。

もし、指定したチャネルが大域的に宣言されたものならば放送（ブロードキャスト）となる。放送の場合、前述の同期指定は無効になる。

2.5 alt文

複数のプロセスからの通信を待ちたい場合には、CSPの選択コマンドに似た、非決定性を扱うalt文を用いる。alt文は以下の

ように使用される。

```
alt G1;G2;..;Gn end
```

これによりG1,G2,..,Gnのガードの中の一つが確実に実行される。各ガードは

```
受信 & 論理式 -> 文
```

の形をしており、受信が可能でかつ論理式が真のとき、このガードが選択され、文が実行される。"& 論理式"は省略可能である。

また、alt文は以下のようにelse文を指定することができる。

```
alt G1;G2;..;Gn;
     else 文の並び end
```

この場合、G1からGnの実行の可能性が何等かの順序で調べられた後、そのどれもが実行できない場合にelse部が実行される。このelse文により通信の時間切れや通信の優先順序などを陽に指定することができる。

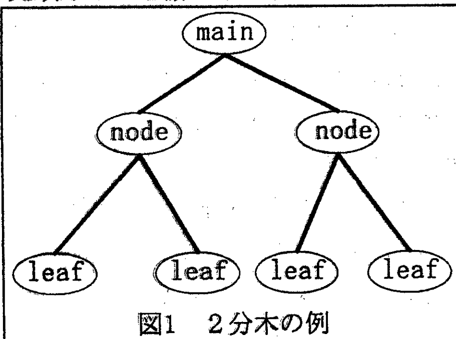

図1　2分木の例

2.6 プロセス生成

gen文を用いることにより、新たなプロセスの生成とともに、それらプロセスが使用するチャネルの割り当てを行う。つまり、ネットワーク形成する。例えば図1のような2分木を形成するにはプロセスmainにおいて以下のような記述する。

```
type arc=protocol ...end;
...
process main;
  chan right,left:arc;
  ...
  gen(A:array[0..3] of arc)
      node(left,A[0],A[1]);
      node(right,A[2],A[3]);
      leaf(A[i]),[i:0..3]
  end;
```

チャネル配列Aはプロセスmainが後に使わないために括弧の中で一時的に宣言され、ちょうど2度パラメタとして使用される。また、後にmainが使用するチャネルleftとrightは1度だけパラメタとして現れる。また、[i:0..3]の指定によりプロセスleafはA[0],A[1],A[2],A[3]をそれぞれ割り当てられた4個のプロセスが生成される。

2.7 プロセスの終了

プロセスの終了はterminate文により明示的に行うことができる。また、コードの最後に制御が達したプロセスも同様のふるまいをする。

3 MachのC Threadsパッケージ

LUNA88k2 の OS である Mach では、C Threadsパッケージ[5]（以下CThreadsと呼ぶ）を用いることにより、ハードウエアに依存した細かな知識を必要とせずにMachのスレッドを利用したアプリケーションを開発することができる。提供される機能には並行処理をコントロールする多重スレッド、共有変数、クリティカル領域の相互排除、スレッドの同期化の条件変数などがある。CThreadsは後に述べるようにC言語の記述性をある意味で制限するものであるが、SPLANトランスレータのターゲットとするにはその制限は特に大きな問題にはならない。逆にトランスレータの移植を簡単にするという点で望ましい。

この章では今回用いたCTreadsの基本機能を紹介したのち、いくつかの問題点について述べる。

3.1 スレッド

Machのプロセスの概念はタスクとスレッドの２つのレベルからなる。タスクは実行環境であり資源割り当ての基本単位である。しかし、実際にはタスクが計算を実行するわけではなく、実行はタスクに含まれるスレッドが行う。スレッドはスケジュールの基本単位である。あるタスク内の全てのスレッドはそのタスクの持つ全ての機能にアクセスでき、スレッドどうしは保護されていない。したがってこの共有された資源は各スレッドがそれを考慮に入れて使わねばならない。

CTreadsのスレッドの生成と終了には

```
cthread_fork(func,arg)
cthread_exit(result)
```

をそれぞれ用いる。cthread_forkとUNIXのforkとの違いは、スレッドを生成することの他、スレッドが実行を開始する関数およびその関数への引き数をひとつ指定できることである。cthread_exitはプログラムの終了ではなくスレッドの終了を明示的に示すためのものである。

ここで生成されたスレッドはＣにおける静的変数を全て共有変数とみなす。これらは大域変数の他に、関数内のstatic変数も含まれる。また、stdout等も共有するために複数のスレッドがprintfを同時に実行すると予期せぬ結果となる。

3.2 相互排除と同期

相互排除を達成するためにmutexと呼ばれる２値セマフォが提供されている。mutexを操作するためにいくつかの関数があるが、その基本はmutex mに対して

```
mutex_lock(&m)
mutex_unlock(&m)
```

の二つである。名前のとおりmutex_lockは、mutex mをロックし、mutex_unlockはそのロックを解除する。ロックに失敗した場合は、mの使用者がそのロックを解除するまでロックを試みる。

同期にはmutexと併せて用いる条件変数とそれを操作する関数

```
condition_wait(&c,&m)
condition_signal(&c)
```

等がある。ここで引き数のcは条件変数、mはmutexである。condition_waitはmのロックに成功した後、何らかの条件を他のスレッドが満たしてくれるのを待つために、条件変数cとともに使われる。この関数を呼び出すと、mutex mのロックを解除し、他のスレッドが同じ条件変数cによるcondition_signalを呼ぶまで一時停止状態になる。基本的な使い方は以下のようになる。

```
/* one thread */
mutex_lock(m);
...
while(/* 条件が真でない */)
    condition_wait(c,m);
...
mutex_unlock(m);
--------------------------------
/* another thread */
mutex_lock(m);
...
condition_signal(c);
mutex_unlock(m);
```

3.3 CThreadsの問題点

CThreadsは今まで見てきたようにＣの静的変数を全て共有変数としてしまうために、Ｃの記述力を制限するという大きな問題を持つ。例えばライブラリ化された関数を使用する場合、その関数がファイル内静的変数を使用しているならば、関数の使用にはmutexを用いなければならない。さもなければ、ライブラリを修正する必要がある。標準I/Oライブラリがその例である。また前述のとおり、関数内の静的変数を持つ場合も同様である。

次にmutexのlockとunlock、conditionのwaitとsignalといったそれぞれの対応が、プログラムが複雑になるにつれ非常に複雑になる。if文やwhile文といった制御の流れが変わる場合、unlockのし忘れが起こりやすくデッドロックを引き起こす。これはＣのプログラミングをさらに複雑なものとしてしまい、他人のプログラムのデバッグをほとんど不可能にする。

4 SPLANの実現

はじめにも述べたがSPLANはＣ言語へのトランスレータという形によりnCUBE2等で実現されている。したがって今回もこの形を取った。この章ではメッセージ通信を中心にSPLANの実現について述べる。

4.1 プロセスの管理

SPLANプロセスをそのままCThreadsのスレッドと対応させた。SPLANではプロセス内の局所変数は、PASCAL同様の入れ子の手続き（関数）宣言を許している。そのためＣに変換されたときに、静的変数を使用していないので問題にならない。したがって、プロセス管理としてはプログラムの終了を判定するために、実行しているプロセス数を数えるカウンタを共有変数として一つ設定し、これをＣのmain関数において監視するようにした。つまり、Ｃのmain関数はSPLANのmainプロセスをforkし、カウンタを１に設定した後、このカウンタが０になるのを待ってexit関数を呼ぶ。各プロセスは他のプロセスを生成したときにその数だけカウンタを増やし、プロセスが終了するときには各々が１減らす。

4.2 通信バッファ

SPLANプロセスは非同期通信を行うために、プロセスそれぞれが受信用のバッファを持つ必要がある。さらにそのバッファは放送をサポートするために全てのプロセスがアクセスできなければならない。そこで二つの実現方法を考案した。一つは生成されるプロセスの数を制限し、その個数分の静的な配列を用い、その配列要素各々を一つのプロセスの受信バッファすると方法（図2）。他方はハッシュとリストの原理で、固定配列から受信バッファをリンク・リスト状につなげていく方法である（図3）。前者は生成されるプロセス数が制限されるがアクセスが速く、後者はその逆である。プログラマがコンパイル時にそのどちらかを指定するのはそれほど難しい判断ではないと思われる。

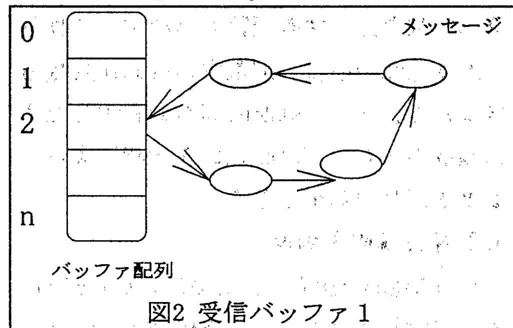

図2 受信バッファ１

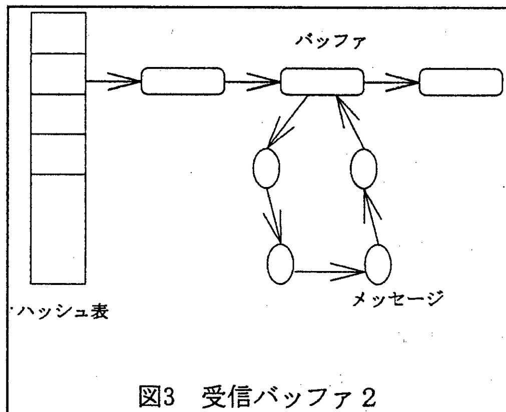

図3　受信バッファ２

それぞれの図中のメッセージは、非同期通信時に動的にバッファに入れられる。これは実チャネルとタグから作られる識別番号、送信者のID、データサイズそしてデータからなり、FIFOのリンク・リストとしてつなげられていく。

4.3 同期と非同期通信

SPLANは同期と非同期の通信を行うことができる。同期通信の場合、プロセスの待ち合わせに前述のバッファを用いるが、実際のデータは直接受け側の変数にコピーすることができる。これは以下のように行われる。①送信が先の場合、送信者は対応する受信者が待っていないことを確かめた後、図2や図3のようにメッセージを生成し、識別番号、ID、データサイズを設定する。送信データはコピーせずに送信データの変数の先頭アドレスを設定し、待ち状態に入る。受信者はリストの中から対応するメッセージを先頭から探し、データを自分の変数にコピーした後、送信者に合図を送る。②受信者が先の場合、受信バッファの専用の場所に識別番号と通信相手のIDおよびデータを格納する場所の先頭アドレスを設定し、待ち状態に入る。送信者がこれを見つけるとデータをそのアドレスにコピーした後、合図を送る。

非同期通信では、送信者が先の場合はメッセージを生成してそこにデータをコピーするが、受信者が先の場合は同期通信の②と同様の方法をとる。

このように通信を細かく場合別けした結果、データの無駄なコピーを減らすことができた。

5 実行例

現時点では今回開発したシステムの詳細な性能評価がなされていない。そこで、ここでは２つの簡単な実行例の紹介にとどめる。実行は4プロセッサのLUNA上において他のユーザがloginしていない状態で計測し、通信バッファは図2の方法を使用した。

5.1 並列ソート

最初の例はプロセスを図1に示した木構造に生成し、mainプロセスがデータを読み込んだ後、nodeプロセスそしてleafプロセスと下方に向かってデータを均等に分配していく。leafプロセスはデータを受け取るとクイックソートを用いてソートし、結果をnodeプロセスに送る。nodeプロセスはそれらをマージしてmainプロセスに送り、mainがマージして終了する。このプログラムを用いて５０万個から１００万個の文字をソートした。その結果を図4に示す。

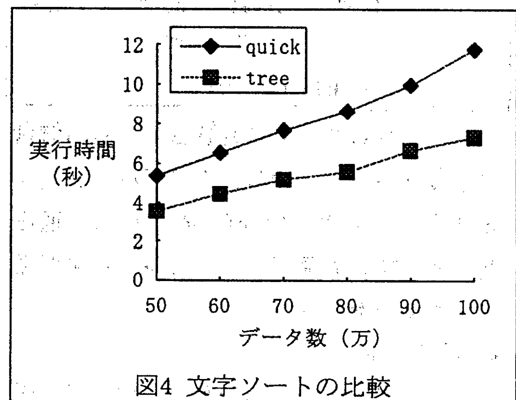

図4 文字ソートの比較

どの点でもクイックソートの２倍の速さになっていないが、このアルゴリズムは通信が非常に多く、またそれほどよく並列化されていないためである。並列ソートにおいてleafでのソートの量は逐次の1/4であるが、通信とマージによるのオーバーヘッドがかなり大きい。

5.2 並列１次元FFT

次はCooley-Tukeyのアルゴリズム[6]を単純に並列化した例である。まず４つのプロセスをそれぞれバタフライ演算と同様に結合させ、初期データをプロセス１から順にそれぞれ割り当てる。Ｎステージの変換が必要であるとして、Ｎ－２ステージまで各プロセスが独立に計算した後、Ｎ－１ステージでプロセス１と２および３と４が通信して変換を行い、Ｎステージにおいてプロセス１と３および２と４が通信してそれぞれ変換を行う。8192個から65536個の単精度浮動小数点を用いたFFTの実行結果を図5に示す。

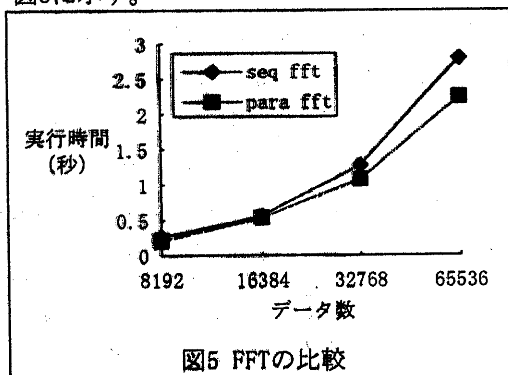

図5 FFTの比較

このアルゴリズムは計算量よりも通信量がかなり多い。しかし、このアルゴリズムを利用して２次元のFFTを考えれば、通信を増やさずに計算できるので良い結果が期待できるであろう。

6 おわりに

マルチスレッドを用いたメッセージ通信型並列処理言語SPLANの実現について述べてきた。nCUBE2およびTRANSPUTERで実行可能な並列プログラムをそのまま共有記憶のマルチプロセッサ上で実行する環境ができたことは価値があるといえるであろう。今後、評価とともに更なる改善を行っていく予定である。

参考文献

(1) 岡本秀輔,飯塚肇: “メッセージ通信型並列処理言語SPLAN”, 信学論文Vol.J75-D-I No.8 pp.575-582(1992).

(2) 岡本秀輔,飯塚肇: “nCUBE2における並列処理言語SPLANの実現”, 情処42全大pp.5-147～5-148(1991).

(3) 岡本秀輔,飯塚肇: “TRANSPUTERにおける並列処理言語SPLANの実現”, 情処43全大pp.5-91～5-92(1991).

(4) Hoare C. A. R.: “Communicating Sequential Processes”, Comm.ACM, 21(8), pp.666-677(1987).

(5) “LUNA-88K UniOS-Mach解説書”, オムロン(1990).

(6) Cooley, J. W., and Tukey, T. W., “An algorithm for the machine calculation of complex Fourier series”, Mathematics of Computation, Vol. 19, No. 90, pp.297-301(1965).