分散型システム記述用言語 Concurrent C の処理系の試作

安藤 誠　辻野 嘉宏　荒木 俊郎　都倉 信樹
（大阪大学基礎工学部）

1. まえがき

筆者らは，既に，複数のプロセッサから成る疎結合分散型システムのための記述用言語として，Concurrent Cを提案している。[1),2)] Concurrent Cは，システム記述用言語Cに並行処理機能を付加した言語であり，複数プロセッサから成るシステム全体に対して，一体化したシステム・プログラムの開発が可能である．本稿では，Concurrent Cの処理系の実現方法について述べる．なお，このConcurrent Cシステムは，NOVA3とMP/100（micro NOVA）結合システムのもとで試作された．

2. Concurrent Cの特徴

Concurrent Cの主な特徴は，(1)プロセッサの概念をもつ，(2)並行処理機能は効率的であり，かつ信頼性が高い，(3)システム記述向きである，(4)Cと上位互換性をもつ，である．

Concurrent Cのプログラムは，いくつかのプログラム・ユニットとプロセスから構成され，動的に生成されたプロセス（さまざまな機能単位）の集まりによって，目的とした処理が行なわれる（詳細は文献1)参照）．プログラム・ユニットは，Cの通常の関数と変数およびプロセス関数（キーワードprocessの付いた関数）の集まりである．プロセスは，プロセス関数名を指定して生成されるが，プロセッサを越えて動くことはできない．これらの基本的プログラム構造のもとに，次のような言語機能がCに比べて拡張されている．

(1)プロセッサの指定可能な動的プロセスの作成（activate文）．

(2)同一プロセッサ内の複数のプロセスにより大規模なデータ構造を共有するためのモニタとモニタの内部にある関数（モニタ外から呼ばれ，モニタ内の共有変数の参照を行なう関数）の実行順を制御する制御式（control expression）．

(3)send文，receive文による低レベルなメッセージ形式によるプロセス間通信機能．

(4)複数受信待ちできるselect文．

(5)send文，select文は時間切れ処理が指定できる．

(6)ストレージ・クラスexport，importの追加により，プログラム・ユニット間での名前（プロセスと関数の名前）の可視性を指定できる．

3. Concurrent Cコンパイラ・システムの構成

3.1 プログラム・ユニットの構造

プログラム・ユニットは，複数のコンパイル・ユニットをリンクすることにより構成され，実行時，下記の3つの領域に分けられる．

1）データ領域：プログラム・ユニット中で定義された静的変数の領域である．Concurrent Cの複数のプロセス間では，これらの変数は共有されないので，プロセスが生成されるたびに，そのプロセスのコードのあるプログラム・ユニットのデータ領域が，新プロセスに新らしく割付けられる†．

2）モニタ・データ領域：モニタ内で定義されたストレージ・クラスがexternの静的変数の領域である．これらの変数は，ある条件でグループ化されたプロセス群††によって共有される．従って，モニタ・データ領域は，グループを作る主となるプロセスが生成されたときに，新らしく割付けられる．

3）コード領域：複数のプロセスが同一コードを共有する方が，主記憶の利用効率もよいので，コード領域は再入可能（reentrant）とした．

†…実行時には，このデータ領域に動的変数の領域としてのスタックが付け加えられて，プロセスのデータ領域となる．

††…プログラム・ユニットの外に輸出（export）しているプロセス関数名で生成されるプロセスを主プロセス（chief process）と呼び，主プロセスとその子孫（プロセスの生成に親子関係があるとする）が，1つのグループとなる．

これらの3領域は，それぞれ別々のタイミングで主記憶に割付けられたり，解放されたりする．ここで，Concurrent Cの実行時環境として，NOVA3のように，少ない主記憶しかなく，しかも再配置を容易にするメモリ管理機構のない計算機の場合がある．そこで，3領域とも，(1)主記憶-補助記憶間でスワップイン・スワップアウト可能であること，(2)完全に再配置可能(relocatable)であることが要求される．(2)のために，各領域内のアドレスを，各領域の先頭からのオフセットとして評価する(実行時にも，再配置性を保つように注意している(5.1節参照))．

3.2 変換フェーズの概略

Concurrent Cコンパイラ・システムの作成では，既に作成したCコンパイラ[3]の一部を利用した．図3.1に本システムの構成図を，表3.1に追加されたCコードを示す．

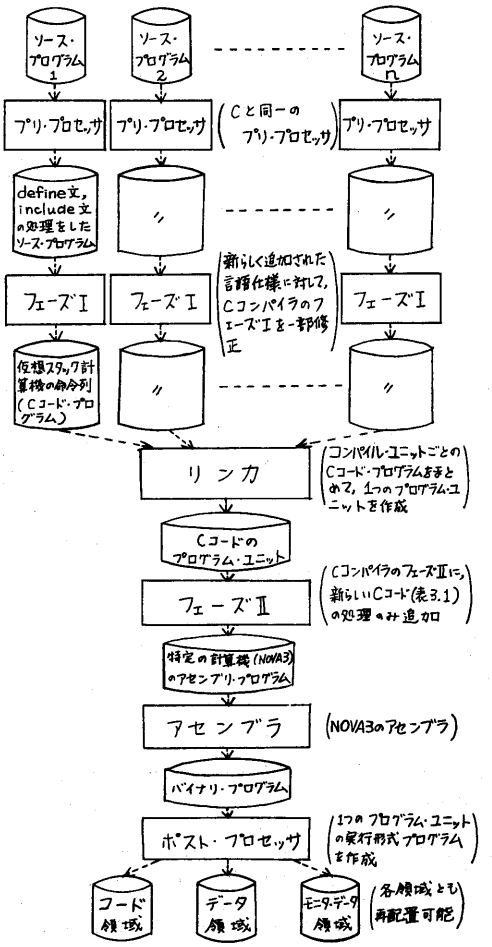

図3.1 Concurrent Cコンパイラ・システムの構成図

表3.1 追加Cコード表

ニーモニック	機能
ACT	プロセスを生成する．
SND	メッセージを送信する．
REC	メッセージを受信する．
WAT	複数メッセージの選択的待ちを行なう．
MIN	最短時間をもつ時間切れ処理を探す．
TAS	モニタ内の相互排除のためのP操作．
RST	モニタ内の相互排除のためのV操作．
LDM	モニタ・データ領域を主記憶にロードする．

4. 並行処理機能のコンパイル法

追加された機能であるactivate文，send文，receive文，select文について，フェーズIの出力(Cコード列)を例示する．追加機能の一つであるモニタの制御式については省略する(文献4)参照)．

4.1 activate文

図4.1に，プロセッサk上に，プログラム・ユニット"PU_i"の中のproc1というプロセス関数から実行を始めるプロセスを1つ生成し，そのプロセスにパラメタとしてi,jの値を渡し，生成したプロセスのプロセス識別子をlong型変数pid1に格納するというactivate文をCコードに変換した例を示す．

ソース・プログラム
```
import process proc1( ) from "PUi" ;
activate pid1 = proc1 (i,j) on k ;
```

cコード・プログラム
```
LAD  pid1    pid1のアドレスをスタックに積む．
MST†         スタックに印を付ける．
LOD  i       iの値をスタックに積む．
LOD  j       jの値をスタックに積む．
LOD  k       プロセッサ番号kをスタックに積む．
LAD  ℓ1      プログラム・ユニット名のあるアドレスをスタックに積む．
ACT  ℓ2      プロセスを生成し，その識別子をスタックに積む．
STU          識別子をpid1へ格納．

ℓ1 : プログラム・ユニット名(文字列) "PUi"
ℓ2 : プロセス関数名(文字列) "proc1"
```

†---MSTで，スタックに印を付けることで，ACT,SND等の実行時ルーチンの処理後，どこにスタック・ポインタを戻せばよいかがわかる．

図 4.1 activate文の変換例

4.2 send文

プロセス識別子がpidのプロセスへi,jというパラメタを送り，もしtime秒以内に対応する受信が実行されなければ，ST_1の処理をするというsend文の変換例を図4.2に示す．

ソース・プログラム
```
send (i,j)  to  pid  for (time)  ST1  ;
```

cコード・プログラム
```
      MST           スタックに印を付ける.
      LOD  i        iの値をスタックに積む.
      LOD  j        jの値をスタックに積む.
      LOD  pid      プロセス識別子pidをスタックに積む.
      LDC  0        タグ指定がないので, 0をスタックに積む.
      LOD  time     時間切れ指定timeの値をスタックに積む.
      SND  ℓ1       メッセージの送信を行なう.
                    時間切れでなければ, ℓ1から実行.
    [ST1のcコード]  時間切れの際の処理.
ℓ1:[次の文のcコード]
```

†-----送信では, 相手プロセスのどの受け口かを指定するために, メッセージにタグを付ける. 一方, 受信側は, タグを指定して同一のタグの付いたメッセージが送られるのを待つ.

図 4.2 send文の変換例

4.3 receive文

図4.3にメッセージの送信側のプロセス識別子がpidで, 付加されたタグがrequestであるメッセージが送られてきた場合に, 整数型変数p1と文字型変数aに格納するreceive文の変換例を示す.

ソース・プログラム
```
receive (p1,a)  from  pid : request  ;
```

cコード・プログラム
```
      MST           スタックに印を付ける.
      UJP  ℓ1       次に, from以下の評価をするため, ℓ1へ飛ぶ.
ℓ2 : LAD  p1        p1のアドレスをスタックに積む.
      LDC  2        p1のサイズ(バイト数)をスタックに積む.
      LAD  a        aのアドレスをスタックに積む.
      LDC  1        aのサイズ(バイト数)をスタックに積む.
      UJP  ℓ3       ℓ3へ飛ぶ.
ℓ1 : LOD  pid       プロセス識別子pidをスタックに積む.
      LOD  request  タグrequestをスタックに積む.
      UJP  ℓ2       次に, パラメタの評価をするため, ℓ2へ飛ぶ.
ℓ3 : REC            メッセージの到着を待つ.
```

図 4.3 receive文の変換例

4.4 select文

図4.4のselect文は, case節に書かれた論理式が成立しているreceive節のみについて, 指定されたプロセス識別子からの指定されたタグを伴なうメッセージが送られてくるのを, 複数受信待ちするものである. default節のtime1, time2の短い方の時間以内に, いずれかの受け口のもとにメッセージが送られれば, そのreceive節の後のSTiを実行する. そうでないときは, default節の後のSTjを実行する.

ソース・プログラム(BEはBoolean Expression, STはSTatement)
```
select  {
 case BE1  receive(i) from pid1 . ok : ST1 ; break ;
 case BE2  receive(j) from pid2 : ST2 ; break ;
 default for (time1) : ST3 ; break ;
 default for (time2) : ST4 ; break ;
        }
```

cコード・プログラム
```
      MST           スタックに印を付ける.
    [BE1のcコード]  BE1の評価を行なう.
      FJP  ℓ4       偽なら, 次のcase節(ℓ4)へ飛ぶ.
      UJP  ℓ2       真なら, from以下の評価のためℓ2へ飛ぶ.
ℓ1 : LAD  i         iのアドレスをスタックに積む.
      LDC  2        iのサイズ(バイト数)をスタックに積む.
      UJP  ℓ3       ℓ3へ飛ぶ.
ℓ2 : LOD  pid1      プロセス識別子pid1をスタックに積む.
      LOD  ok       タグokをスタックに積む.
      LAD  ℓ1       対応する送信が行なわれたときに, 実行を
                    開始する番地(ℓ1)をスタックに積む.
      UJP  ℓ4       次のcase節(ℓ4)へ飛ぶ.
ℓ3 : REC            メッセージを受信する.
    [ST1のcコード]  受信した後で, 文ST1を実行する.
      UJP  ℓ14      select文の次へ飛ぶ.
ℓ4 : [BE2のcコード] BE2の評価を行なう.
      FJP  ℓ12      偽なら, ℓ12へ飛ぶ.
      UJP  ℓ6       真なら, from以下の評価のためℓ6へ飛ぶ.
ℓ5 : LAD  j         jのアドレスをスタックに積む.
      LDC  2        jのサイズ(バイト数)をスタックに積む.
      UJP  ℓ7       ℓ7へ飛ぶ.
ℓ6 : LOD  pid2      プロセス識別子pid2をスタックに積む.
      LDC  0        タグ指定がないので, 0をスタックに積む.
      LAD  ℓ5       対応する送信が行なわれたときに, 実行を
                    開始する番地(ℓ5)をスタックに積む.
      UJP  ℓ12      ℓ12へ飛ぶ.
ℓ7 : REC            メッセージを受信する.
    [ST2のcコード]  受信した後で, 文ST2を実行する.
      UJP  ℓ14
ℓ8 : LOD  time1     制限時間time1をスタックに積む.
      LAD  ℓ9       時間切れ時の処理番地(ℓ9)をスタックに積む.
      MIN           スタックに積まれた2組の時間のうち, 短かい方の時間とその処理番地をスタックに残す.
      UJP  ℓ10      次のdefault節(ℓ10)へ飛ぶ.
ℓ9 : [ST3のcコード] time1時間切れ時に, 文ST3を実行する.
      UJP  ℓ14
ℓ10 : LOD  time2    制限時間time2をスタックに積む.
      LAD  ℓ11      時間切れ時の処理番地(ℓ11)をスタックに積む.
      MIN
      UJP  ℓ13      ℓ13へ飛ぶ.
ℓ11 :[ST4のcコード] time2時間切れ時に, 文ST4を実行する.
      UJP  ℓ14
ℓ12 : LDC  MAXTIME  MAXTIMEをスタックに積む.
      LAD  ℓ14      時間切れ時の処理番地(ℓ14)をスタックに積む.
      UJP  ℓ8       最初のdefault節(ℓ8)へ飛ぶ.
ℓ13 : WAT           複数の受信待ちを登録して, 対応する送信を待つ(詳細は, 6.3.2節参照).
ℓ14 :[次の文のcコード]
```

図 4.4 select文の変換

5. Concurrent Cの実行環境

Concurrent Cのプログラム起動時には, システムの生成したプロセスが, 指定されたプログラム・ユニットのmain関数から実行を開始する. 以後, プロセスが新らしいプロセスを動的に生成して, 数が増加する. プロセスの生成は, 他のプロセッサ上に行なうこともできる. 全てのプロセスが終了

したときに，プログラムは終了する．

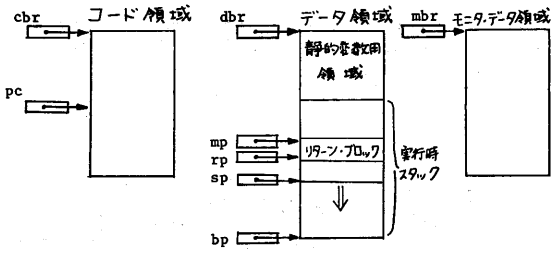

図5.1 プロセスの実行時の状況

5.1 プロセスの実行時の状態 実行中のプロセスは，それぞれ図5.1に示すような3つの領域と8種のレジスタ(表5.1参照)を持つ．データ領域の一部である実行時スタックは，必要に応じて伸びるので，データ領域の大きさは可変である．

表5.1 レジスタの種類

レジスタ	機能
プログラム・カウンタ (pc)	次に実行すべきコードを指すポインタ．
スタック・ポインタ (sp)	現在の実行時スタックの先頭を指すポインタ．
マーク・ポインタ (mp)	実行中の関数の局所的変数の参照に用いられるベース・レジスタ．mpの指す領域の先頭は，関数のリターン・ブロック†である．
ボトム・ポインタ (bp)	実行時スタック領域の最後(データ領域の最後でもある)を指すポインタ．
ブロック・ポインタ (rp)	実行時スタックの先頭に最も近いリターン・ブロック†を指すポインタ．
データ・ベース・レジスタ (dbr)	プロセスごとの静的変数を参照するのに用いられるベース・レジスタ．
モニタ・データ・ベース・レジスタ (mbr)	モニタ内の静的変数を参照するのに用いられるベース・レジスタ．
コード・ベース・レジスタ (cbr)	プロセスの動くプログラム・ユニットのコード部の主記憶上の場所を知るのに用いられるベース・レジスタ．

(†.... 関数の復帰に必要な情報を置く領域．関数呼出し前のpc(関数からの戻り番地)，mp，rp，関数のリターン値が置かれる．)

スケジューリング・アルゴリズムにより，次に実行されるプロセスがPと決まると，スケジューラが，Pの3領域を，主記憶になければロードする(ロードの際に主記憶が不足すれば，スワッピング・アルゴリズムによりプロセスをスワップ・アウトして，ロードの領域を確保する)．さらに，レジスタ類に実アドレスを設定して，制御をPに渡す．その後，プロセスの実行は，次のように行なわれる．(1) 静的変数のロードとストアに対しては，dbrあるいは，mbrとコード中にある変数の論理アドレス(データ領域，あるいは，モニタ・データ領域の先頭からのオフセット)を加えることにより，実アドレスを計算する(モニタ・データ領域の変数の最上位ビットに1を立てることにより，2つのデータ領域の参照を区別する)．(2) 変数のアドレスを評価した値は，論理アドレスである．従って，スタックに積まれる値も，変数に格納される値も論理アドレスである．(3) 静的変数の参照時には，プロセスの実行中に他のプロセスの領域を壊さないように，参照範囲を検査している．(4) 関数の呼出しの際には，cbrとコード中のその関数の先頭アドレス(コード領域の先頭からのオフセット)を加えることにより，実アドレスを計算して呼出しが実行される．このとき，関数のリターン・ブロック中の戻り番地には，コード領域中の論理アドレスが入れられている．(5) ジャンプの際にも，cbrと飛び先の論理アドレスの加算により，実アドレスを計算して飛び先が決定される．(6) 算術演算は，Cで行なわれていたのと同様である．(7) 並行処理機能(プロセス生成，メッセージ送受信など)は，スケジューリング，メモリ管理などが必要となるので，カーネルで処理する(6節参照)．(1)～(7)の処理は，それぞれのCコードに対応する実行時ルーチンを呼出すことにより行なわれる．

(2)，(4)に示したように，実行時にも，再配置性が保持されている．

5.2 プロセス実行時に必要なデータ構造

1つのプロセッサ内には，複数のプロセスが存在し，スケジューラによって制御が切り替えられる．これらの複数のプロセスをスケジューラによって管理するために，プロセス・リストがある．その1つのエントリ(プロセス・エントリ)は，図5.2のような構造をもち，その各項目には，プロセスの状態と，一度スケジューリングで中断されたプロセスを再開するために必要な情報などが入れられている．各プロセスの各領域へは，直接のポ

インタではなく，メモリ・エントリ(図5.4)や，プログラム・ユニット・エントリ(図5.3)へのポインタになっている．これにより，メモリ・エントリを見ることで主記憶の管理の大半が可能となる．また，プロセスがどのプログラム・ユニットに属しているかを調べたり，1つのプロセッサ上に同じプログラム・ユニットが2つ以上ロードされるのを避けるため，コードの場合に，メモリ・エントリとの間に，プログラム・ユニット・エントリが設けられている．

プロセス識別子	プロセス・プライオリティ
mp	bp
sp	pc
rp	プロセス・ステイタス
プログラム・ユニット・エントリ へのポインタ	
メモリ・エントリ(データ用)へのポインタ	
メモリ・エントリ(モニタ・データ用)へのポインタ	
プロセス・エントリ へのポインタ (プロセス・リストのリンク用)	

(mp, bp, sp, pc, rpは全て論理アドレスが入れられる)

図5.2 プロセス・エントリの構造

プログラム・ユニット名
メモリ・エントリ(コード用)へのポインタ
プログラム・ユニット・エントリへのポインタ(リンク用)

図5.3 プログラム・ユニット・エントリの構造

一時的ファイルの名前
一時的ファイルのサイズ
ロードされたときの主記憶上の実アドレス
利用カウント

図5.4 メモリ・エントリの構造

6. Concurrent Cのカーネル

プロセスの実行時の環境を支えるのが，カーネルである．

6.1 ヘッド・ルーチン Concurrent Cの環境への移行を行なう．最初に動き出して欲しいmain関数をもつプログラム・ユニットの名前とその関数へのパラメタを，コマンド・ラインに入力すると，ヘッド・ルーチンが，プロセスを1つ生成して，上記のmain関数から実行を開始させる．

6.2 プロセスの生成 Cコード ACTに対応する実行時ルーチンのカーネルにおける処理手順を示す．①プロセス・エントリ作成，②データ領域ファイルに，activate文中のパラメタを付加して，新プロセス用の一時的データ・ファイル作成，③生成を行なったプロセスのモニタ領域を継承(生成されるプロセスが主プロセスなら，実行開始時に，新モニタ領域が割付けられる)，④プロセッサに新プロセスのコードのあるプログラム・ユニットがなければ，コード領域をプログラム・ユニット・エントリに登録．既に存在する場合，利用カウントを1増やす，⑤生成された新プロセスが実行開始できるように，レジスタ類を設定(プロセスの3領域は，スケジューラによりこのプロセスに制御が渡されるまでに，ロードされる)，⑥生成されたプロセスの識別子を，生成したプロセスのスタックの先頭に積む．ただし，他プロセッサ上にプロセスを生成する場合は，生成に必要な制御情報を送信する．プロセス生成後，プロセス識別子が返信される．

6.3 メッセージの送受信 プロセス間のメッセージの送信はSNDルーチン，受信はRECルーチン，複数メッセージの選択受信待ちはWATルーチンで処理される．

6.3.1 送信と受信(SND,RECルーチン) 送信が，対応する受信より以前に行なわれた場合，メッセージを送信表(send table)に登録する(時間切れ処理があれば，その時間も添えて登録)．逆に，受信の方が先に実行された場合，メッセージを入れるべき変数の番地と大きさ(バイト数)を受信表(receive table)に登録する．送信と受信の対応がつくまでは，プロセスはサスペンド状態である．対応がついたとき，メッセージを順に変数に格納し，プロセスをレディ状態にする．他プロセッサのプロセスへの送信では，相手プロセッサの送信表に登録する．ただし，送信の時間切れ処理がある場合，処理開始番地は自プロセッサの時間切れ表(time-out table)に登録する．メッセージの送受信における大部分の処理は，受信プロセスが存在するプロセッサ側で行なわれる．

6.3.2 複数受信待ち（WATルーチン）

スタックに積まれた複数の受信情報（4.4節参照）を受信表に登録し，時間切れ時の処理開始番地を時間切れ表に登録する．いずれかの受信に対応する送信が実行されると，そのreceive節のパラメタの評価が行なわれ，送信されたメッセージが渡される（受信が送信より以前に行なわれた場合と同様である）．このとき，対応しなかった受信（複数個ありうる）の情報は，受信表から削除する．

6.3.3 時間切れ時の処理

スケジューリングが行なわれるたびに，送信表および受信表の各エントリの時間を検査する．時間切れプロセスがあれば，そのプロセスの状態をサスペンドからレディに変え，時間切れ時の処理開始番地（時間切れ表に登録されている）から，プロセスを実行させる．

6.4 プロセスのスケジューリング

プロセス・リストからラウンド・ロビン方式でレディ状態のプロセスを選ぶ．スケジューリング・ポイントは，新らしく追加された並行処理に関わる実行時ルーチンの中に埋め込まれている．

6.5 メモリの管理

主記憶の制限により，スケジューリングされて実行するプロセスの，3つの領域の一部あるいは全部がロードできない場合，サスペンド状態のプロセスから，主記憶よりスワップ・アウトしていく．必要があれば，レディ状態のプロセスでもスワップ・アウトする（状態は，レディのままである）．

6.6 他プロセッサとの通信

他プロセッサとの通信は，ACT，SNDルーチンなどで用いられる．プロセッサ間の通信情報の転送は，転送先プロセス名，通信情報の種類，その中身をパラメタにして，送信を行なう関数を呼ぶ．一方，受信は，受信済カウンタを定期的にみて，通信情報が到着していれば，その解析を行なう．送受信は，割込み処理部で処理される．

6.7 カーネルの記述

NOVA3とMP/100のカーネルは，プロセッサ間の通信処理以外，全く同一である．カーネルは，そのほとんどが，Cのプログラムで作成されている．これが動くには，Cの実行時ルーチンが必要となるが，Concurrent C用実行時ルーチンを，そのまま利用している†．

7. あとがき

分散型システム記述用言語として提案したConcurrent Cの処理系の，複数のプロセッサ上での実現について述べた．処理系は，言語のコンパイラ・システムの作成と実行時環境の作成とに大別されるが，前者に4人日，後者に4人日の期間を要した．コンパイラ・システムの各フェーズともCで書かれ，また，実行時環境（15K語）もそのほとんどがC（2100行）で書かれている．アセンブリ言語は，Cの実行時ルーチン（アセンブリ言語で書かれたロード，ストア等の簡単なルーチン：2.4K語）の一部の修正（100行）と，プロセッサ間通信の処理部（470行），プロセスの切り替え時のレジスタ類の設定部分（30行）の記述だけに用いられた．NOVA3－MP/100結合システムは，Concurrent Cの実現の一つの試作であり，ここで得られた経験をもとに，複数プロセッサ上での実現を行なう予定である．

謝辞 実行時環境の一部を作成していただいた小田垣秀雄氏，香西省治氏ならびに，NOVA3とMP/100との結合に協力いただいた竹村治雄氏に感謝いたします．

文献

1) 辻野，安藤，荒木，都倉："分散型システム記述用言語 Concurrent C"，信学技報（電子計算機），EC 81-13 (1981-06)．

2) Y.Tsujino, M.Ando, T.Araki and N.Tokura: "Concurrent C: The Programming Languages for Distributed Multiprocessor Systems", Programming Languages Group Memo No.81-04, Department of Information and Computer Sciences, Osaka University (1981-09)．

3) 黒田，辻野，荘原，荒木，都倉："システム記述用言語Cのポータブル・コンパイラの作成"，情報処理学会論文誌，Vol. 21, No. 6 (1980-11)．

4) 辻野，香西，荒木，都倉："Concurrent Cのモニタにおける制御式について"，情報処理学会第24回全国大会，7L-6 (1982-03発表予定)．

† ---- cbr, dbr, mbrにそれぞれ 0, 0, 8000(16進)の値を設定しておくことにより，Concurrent Cの実行時ルーチンが利用できる．