プログラミング一言語・基礎・実践一研究報告　9－9
（1992．　10．　30）

SIMD 型超並列プログラミング言語「並C」とそのコンパイラ

貴島 寿郎　　湯淺太一
豊橋技術科学大学

並Cは，開発中の SIMD 型超並列計算機 SM-1 上で使用されるプログラミング言語である．並C は ANSI 標準の C 言語に SIMD 型超並列処理機能を拡張したものであり，SM-1 のアーキテクチャを反映した低水準の並列処理機能をサポートする．本稿では，並Cの基本的な超並列処理機能を紹介し，稼働中の並Cコンパイラの概要を述べる．

Design and Implementation of An Extended C Language NAMI-C for SIMD Massively Parallel Programming

Toshiro Kijima　　Taiichi Yuasa
Toyohashi University of Technology
1-1 Hibarigaoka, Tenpaku-cho, Toyohashi, Aichi 441, Japan

Nami-C is a programming language to support programming on SM-1, a SIMD massively parallel computer. Nami-C is an extension of the ANSI C programming language with functions for massively parallel computation. It supports low-level primitives for data parallel processing on SM-1. The paper introduces the language and reports the current implementation of the compiler.

1 はじめに

並C[1, 2]（「なみしい」と発音する）は，開発中のSIMD型超並列計算機SM-1[3]上で使用されるプログラミング言語である．並CはANSI標準のC言語にSIMD型超並列処理機能を拡張したものであり，他のSIMD型超並列C言語MPL [4]やC* [5]と同様，プログラムの並列実行部分は明示的に記述される．拡張された機能を使用してい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逐次実行される．

並CはSM-1上でのアセンブラに代替する言語として設計されている．そのためPE間通信やバス上の演算など，SM-1のアーキテクチャを反映した低水準な並列処理機能をサポートしている．

現在並Cのコンパイラが SPARCstation 上で稼働している．このコンパイラは GNU C コンパイラ [6] (gcc) を改造することにより実現されている．

2章ではターゲット・マシンであるSM-1の概要を述べる．3章から5章では並Cの基本的な超並列処理機能を紹介する．6章では稼働中の並Cコンパイラの概要についても述べる．

2 SM-1について

SM-1は多数（現行モデルでは1024個）のプロセッサ (Processing Element，以下PEと略す) からなるPEアレイと，フロントエンド (Front End，以下FEと略す) より構成されている（図1参照）．各PEは大容量（現行モデルでは1Mバイト）のローカルメモリを持っている．PEアレイはPE間通信のための以下のようなネットワークを持つ．

- 2次元メッシュ (NEWS) ネットワーク
- shuffle-exchange ネットワーク
- ORバス（縦，横，全PE）

3 データ形式

並Cのデータ形式には，FE上に配置され逐次的に処理されるスカラ・データと，PEのローカルメモリ上に配置され，複数のPEで同時に処理されるパラレル・データの2種類がある．並Cではこれらを区別するために型が拡張されている．

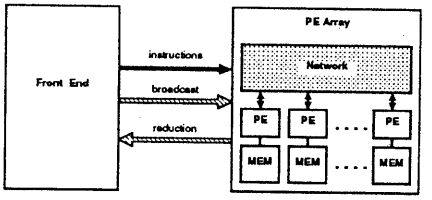

図 1: SM-1 の構成

宣言　スカラ・データを格納するための変数をスカラ変数，パラレル・データを格納するための変数をパラレル変数と呼ぶ．並Cでは，通常のCと同様な型のパラレル変数を宣言できる．パラレル変数の宣言はデータ型に $ を後置する．$ を後置しない場合はスカラ変数の宣言となる（図2参照）．

ポインタ　ポインタを宣言する場合，1) ポインタが指すデータがスカラかパラレルかということと，2) ポインタ自身がスカラ・データかパラレル・データかということを指定する（図2参照）．

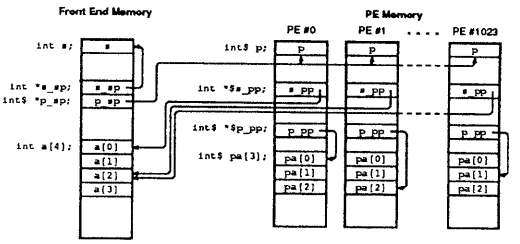

図 2: 変数およびポインタのアロケーション

4 基本的な並列処理機能

4.1 並列演算

算術演算　算術演算のオペランドがすべてスカラ・データである場合，演算は通常のCと同様に行われる．

例：
```
int     s1, s2;

s1 + s2
```

オペランドがパラレル・データであった場合，その演算は各 PE ごとのデータに対して行われる．スカラ・データとパラレル・データとの演算では，スカラ・データはパラレル・データへ暗黙のうちに変換され，それぞれの PE 上で演算が行われる．

例：
```
int $   p1, p2;

p1 * p2
p1 << s1
```

代入　パラレル変数へ代入する場合，右辺はパラレル・データとスカラ・データのいずれも許される．パラレル変数への代入は同じ PE 上に格納されている値が代入される．右辺がスカラ・データである場合，パラレル・データへ暗黙のうちに変換される．

例：
```
p2 = p1;
p1 = s1;
```

スカラ変数へ代入する場合，右辺はスカラ・データのみ許される．

例：
```
s1 = s2;
s1 = p1;      /* ILLEGAL !! */
```

4.2 制御構造

並 C では，プログラムの流れを制御する通常の C の制御構造のほかに，PE を選択的に実行させるための制御構造を持つ．PE の実行が許されているか否かを示す状態をその PE のアクティビティと呼び，アクティビティを制御する構文をアクティビティ制御構文と呼ぶ．実行を許されている PE をアクティブな PE，許されていないものをインアクティブな (アクティブでない) PE という．並 C のアクティビティ制御構文には以下のようなものがある．

- 条件文 (pif, pswitch)
- ループ (pwhile, pdo-pwhile, pfor)
- その他
  (pbreak, pcontinue, preturn, pall)

ここでは pif 文をとりあげる．pif 文はある条件を満たす PE のみに処理を実行させる．例えば

```
pif (p1 < 100000)
        p1 = 100000;
else
        p1 = p1 * 1.03;
```

という場合，p1 の値が 10 万未満であるすべての PE に対して p1 の値を 10 万にし，それ以外の PE に対しては p1 の値を 3% 増加する．

4.3 リダクション

パラレル・データに何らかの処理を行い，1 つのデータにまとめることをリダクションという．例えば

```
s1 = +$ p1;
```

とすると，各 PE の p1 の合計が求められ，s1 に代入される．並 C では，スカラ・データへのリダクションを行うための演算子としてビットごとの AND，OR，排他的 OR (&$, |$, ^$)，総和 (+$)，総積 (*$)，および最大値と最小値 (>$, <$) が用意されている．

5 その他の機能

並 C の持つその他の並列処理機能は以下の通りである [1, 2]：

アクティビティ制御を伴う演算子 &&$, ||$, ?$:

アクティビティに対応した実行制御 ifany 文，ifnone 文，!$

PE 間通信演算子 @N , @E , @W , @S, @sh , @rs , @ex

バス・リダクション演算子 &-$, |:$, >+$, <-$ など

ブロードキャスト演算子 ?-$, ?:$, ?+$

これらについての説明は省略する．

6 コンパイラの実現

現在並 C のコンパイラが SPARCstation 上で稼働している．このコンパイラは gcc (バージョン 1.40) を基に，以下のような改造を施すことにより実現されている．

1. 構文解析部の拡張
2. パラレル・データに対応するための型属性の追加
3. 拡張された制御や演算子に対応する中間コードの拡張とコード生成部の追加

4. スタック領域割り当て部の拡張

5. データフロー解析部の変更

特にデータフロー解析部に関しては，アクティビティ制御のために変数間の依存関係の解析が複雑化している．以下ではこのデータフロー解析における問題点と解決方法について述べる．

6.1 命令の種類

コード生成部で生成される命令列は SIMD 型計算機向きのものであるから，1つの命令列の中に FE が実行する命令と PE が実行する命令とが混在している．それらの命令は以下の6種類に分類できる．

スカラ演算命令 変数オペランドが全てスカラ変数であるような演算命令

並列演算命令 変数オペランドが全てパラレル変数であるような演算命令

ブロードキャスト命令 ソース・オペランドがスカラ変数，ディスティネーション・オペランドがパラレル変数であるような演算命令

リダクション命令 ソース・オペランドがパラレル変数，ディスティネーション・オペランドがスカラ変数であるような演算命令

分岐命令 命令列上の制御の移り先を条件付きで，あるいは無条件に変更する命令

アクティビティ制御命令 PE のアクティビティを条件付きで，あるいは無条件に変更する命令

アクティビティ制御命令は基本的には PE-jump 命令と PE-label 命令の2種類である．PE-jump 命令はその時点でアクティブな PE のうち，条件が成立している (あるいは無条件なら全ての) PE のアクティビティを指定されたアクティビティ保存場所に格納 (すなわち，1をセット) した後，その PE をインアクティブにする．その他の PE には何もしない．PE-jump 命令の形式は，無条件 PE-jump 命令の場合は

PE-jump L

であり，条件付き PE-jump 命令の場合は

PE-jump L if $COND$

である．L はアクティビティ保存場所，$COND$ は条件を表す．また PE-label 命令はその時点でアクティブでない PE のアクティビティを指定されたアクティビティ保存場所から戻す (すなわち，1がセットされていたらアクティブにする) 命令である．その時点でアクティブな PE はアクティブなままにしておく．PE-label 命令の形式は

PE-label L

である．これらの命令は PE に対する分岐命令および分岐先ラベルと見なすことができる．例えば前述の pif 文

```
pif (p1 < 100000)
     ....
else
     ....
```

は PE-jump 命令と PE-label 命令を用いて次のように表現できる．

```
PE-jump L1 if p1 ≥ 100000
    ....
PE-jump L2
PE-label L1
    ....
PE-label L2
```

6.2 アクティビティ制御による問題

上記のような命令からなる命令列における変数間の依存関係は，スカラ変数同士やパラレル変数同士の間だけではなく，スカラ変数とパラレル変数との間にも存在する．よって変数間の依存関係の解析はスカラ変数とパラレル変数とで一括して行う必要がある．

変数間の依存関係を大域的に解析する際には，前処理として命令列の基本ブロックへの分割と制御フロー・グラフの構成を行う [7]．基本ブロックとは先頭の命令以外から制御の流れが入ったり，最後の命令以外から制御の流れが出ていくことがないような連続した命令列のことである．また制御フロー・グラフとは基本ブロックを節とし，基本ブロック間の制御の流れを辺とする有向グラフのことである．

命令列にアクティビティ制御が含まれない場合，命令列上の制御の流れは FE 側の分岐命令によってのみ変化する．よってパラレル変数とスカラ変数を区別する必要はなく，パラレル変数をスカラ変数と見なした上で従来の解析手法を適用することができる．しかし命令列にアクティビティ制御が含まれている場合は，以下のような問題が生じる．

1. PE-jump 命令はあくまで PE のアクティビティを変化させるだけであるため，FE 側の制御の流れには影響しない．よって PE-jump を実行したとしても，FE 側の制御はその PE-jump 命令の直後の命令にのみ移る．

2. 分岐命令では飛び先となるラベルが命令列中の 1 点を表しているため制御の移り先は一意に定まるが，PE-jump 命令では対応するアクティビティ保存場所に関する PE-label 命令が命令列中に複数存在し得るため，制御の移り先は一意には定まらない．

例えば 1. が問題となるのは

```
int$ p1, p2;
int  s1, s2;
  ....
pif (COND)
     /* PE-jump L if not COND */
{
     s1 = s2;
     p1 = p2;
}    /* PE-label L */
```

において *COND* が成立する PE が存在しなかった場合である．この場合 p1 = p2 は実行されないが s1 = s2 は実行される．また 2. が問題となるのは例えば

```
while (....) {
     ....
     pif (C)
          /* PE-jump L1 if not C */
     {
          ....
          if (....) {
               /* PE-label L1 */
               break;
               /* goto EXIT */
          }
          ....
     }    /* PE-label L1 */
     ....
} /* EXIT: */
```

のように pif 文の内側から外へ break する場合である．この場合 break の前に pif 文で変化したアクティビティを戻すための PE-label 命令が挿入される．このため例中の PE-jump 命令に対応する PE-label 命令は 2 ヶ所存在することになる．

6.3 FE フロー・グラフと PE フロー・グラフ

前述したように，アクティビティ制御を含む命令列においては FE 側から見た制御の流れと PE 側から見た制御の流れは一致しなくなる．そこで FE 側と PE 側とで別々の制御フロー・グラフを構成し，解析の際に着目する変数がスカラ変数かパラレル変数かに応じてそれらを使い分けることにする．

まず命令列を基本ブロックへ分割する際は，基本ブロック内での局所的な依存関係の解析も考慮し，その区切りを

1. 分岐命令の直後
2. 分岐先ラベルが示す命令の直前
3. PE-jump 命令の直後
4. PE-label 命令の直前

とする．次にこれらの基本ブロックから FE 側および PE 側での制御フロー・グラフを構成する．FE 側から見た制御フロー・グラフを FE フロー・グラフ，PE 側から見た制御フロー・グラフを PE フロー・グラフと呼ぶことにする．FE フロー・グラフは従来の制御フロー・グラフと同様，

1. 最後の命令が無条件分岐命令以外である基本ブロックから，命令列での直後にあたる基本ブロックへ向かう辺を作る．
2. 最後の命令が条件付きあるいは無条件分岐命令である基本ブロックから，その分岐先ラベルを持つ基本ブロックへ向かう辺を作る．

とすることで求められる．また PE フロー・グラフは

1. 最後の命令が無条件分岐命令でも無条件 PE-jump 命令でもない基本ブロックから，命令列での直後にあたる基本ブロックへ向かう辺を作る．
2. 最後の命令が条件付きあるいは無条件 PE-jump 命令である基本ブロックから，対応する PE-label 命令を持つ全ての基本ブロックへ向かう辺を作る．
3. 最後の命令が条件付きあるいは無条件分岐命令である基本ブロックから，その分岐先ラベルを持つ基本ブロックへ向かう辺を作る．

とすることで求められる．このようにして構成された2つの制御フロー・グラフを変数間の依存関係の解析の際に用いる．参照されているのがスカラ変数の場合はFEフロー・グラフを，パラレル変数の場合はPEフロー・グラフを用いる．

7 おわりに

SIMD型超並列プログラミング言語「並C」の基本的な超並列処理機能を紹介し，稼働中の並Cコンパイラの概要を述べた．並CはSIMD型超並列計算機能を拡張したCommon Lisp [8] の処理系の記述言語として，あるいは現在開発中のFortran 90ベースのFortranコンパイラ[9]の目的言語としても使用されている．また現在仮想PEをサポートするための拡張が進められている．

参考文献

[1] 貴島：並列C言語“並C”言語仕様書，豊橋技術科学大学湯淺研究室，1992.

[2] 野田：SIMD型超並列プログラミング言語「並C」ユーザーズ・ガイド，豊橋技術科学大学湯淺研究室，1992.

[3] 松田，湯淺：SIMD型超並列計算機SM-1（仮称）の概要，第5回SWoPP予稿集，1992.

[4] MasPar Parallel Application Language (MPL) Reference Manual, MasPar Computer Corp., 1990.

[5] C* Programming Guide, Thinking Machines Corp., 1990.

[6] Stallman, R. M. : Using and Porting GNU CC, Free Software Foundation, Inc., 1990.

[7] Aho, A. V., Sethi, R., and Ullman, J. D. : Compilers – Principles, Techniques, and Tools, Addison-Wesley, 1986.

[8] Yuasa, T. : TUPLE: An Extended Common Lisp for Massively Parallel SIMD Architecture, In Proc. of the DPRI Symposium, 1992.

[9] 安村，大谷，渦原，小前，杉森：Bee-Fortran仕様書，慶応義塾大学安村研究室，1992.