「COINSにおける並列化」

渡邊　坦（COINSコンパイラ・インフラストラクチャ協会）
藤瀬哲朗（三菱総合研究所）
福岡岳穂（ソニー・コンピュータエンタテインメント）
岩澤京子（拓殖大学）
弓場敏嗣（電気通信大学）

はじめに

並列化は主として高速化を目指すものであるが，多重化による信頼性向上など，他の目的のために行われることもある．最近は省電力の手段としても並列化が注目されている．並列化の仕方はアーキテクチャによってもアプリケーションによっても異なり，さまざまな方式が研究されている段階なので，まだ人手による指示なしでコンパイラだけで並列化することは困難といえよう．COINSでは，各種コンパイラの共通インフラストラクチャとするために，並列化のためのごく基本的な機能をいくつか備えている．具体的には，ループの各繰り返しを並列に実行できるdo-all型と呼ばれるループの並列化とSMP（Symmetric Multi-Processor）向けの粗粒度並列化，ならびにSIMD向け並列化のための解析と変換の機能がある．それらは自動並列化というよりも並列化に必要な機能の一部を提供するものである．SIMD並列化については本連載の10月号で説明したので，ここでは次の並列化について説明する．

（1）do-all型ループの並列化

並列化の候補としてあげられたループに対して，制御の流れに沿って実行文がアクセスするメモリ領域を解析し，ループの各繰り返しを並列に実行できるか否かを判定する．そして，並列化可能なものについて，OpenMP[2),3)] プログラムに変換するか，あるいは並列実行できる機械語プログラムに変換する．

（2）SMP向けの粗粒度並列化

プログラムを解析して，関数やループなどの比較的大きい固まりで並列に実行できる部分を見つけ出し，OpenMPプログラムに変換する．ただし，並列化する部分の切り出しなどを行いやすいように，事前に少し人手で変換されていないと効果を上げられないことが多い．

ここでいうOpenMPプログラムは，より正確にいえば#pragmaによる並列化指示文が含まれたCプログラムの形をとり，その実行モデルはOpenMPの規格で定められるものである．ここではそれをOpenMP/Cプログラムということにする．それはOpenMPコンパイラを通じて並列実行される．

OpenMPでは並列に実行される仕事は，スレッドといわれる並列処理の論理的な単位として実行されることが多い．一般に，マルチプロセスでは相異なるアドレス空間で並列実行されるが，1つのプロセスで複数のスレッドが実行されるマルチスレッドでは，同じアドレス空間で並列実行される．do-all型ループの並列化とSMP向け粗粒度並列化は，必ずしもマルチプロセスとして実行する必要はなく，マルチスレッドとして並列実行できる．SIMD並列化は1つの命令内で並列実行されるので，スレッドに分ける必要もない．

以下，do-all型ループの並列化とSMP向け粗粒度並列化の概要，ならびにその実現上の要点を説明する．紙数が限られているので，詳細についてはCOINSのWebページ[1)] を参照されたい．

do-all型ループの並列化

● ループの並列化手順

COINSに実装されているループの並列化では，各繰り返しを独立に並列実行できるdo-all型と呼ばれるforループを検出して並列化する．この解説ではそれを単にループ並列化と略称することもある．これでは，**図-1**に示すように，ソースプログラムに近い構造を持つ高水準中間表現HIRを解析し，そこに含まれるforループをある形に正規化した後で，並列化のためのOpenMP/Cプログラムを生成するか，あるいは並列実行できる機械

用語や記号が定義されている個所に下線を付す．

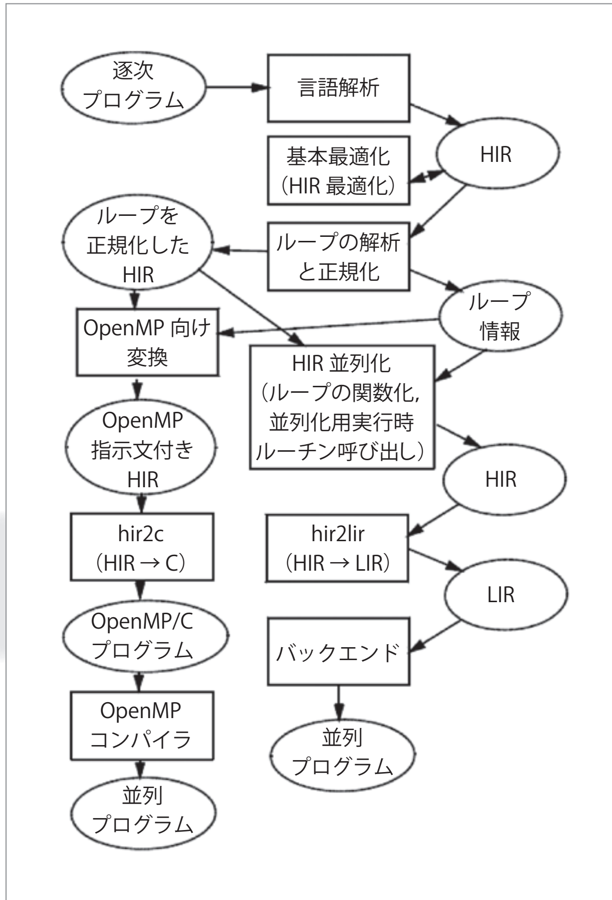

図 -1　do-all 型ループ並列化の処理の流れ

語を生成する．ループの正規化では，

```
for (i = 0; i < n; i = i+1) { .... }
```

のように，繰り返し制御用変数（i）を 0 から 1 ずつ増加させる形の for ループに変換する．OpenMP/C プログラムを生成するときは，それを OpenMP コンパイラを使って，並列化オブジェクトコードを生成して実行させる．OpenMP コンパイラとしては，COINS で利用している Omni[3)] などがある．機械語を生成するときは，並列化用の実行時ルーチンとリンクして並列実行させる．

●do-all 型ループの扱い方

ループの各繰り返しの間に依存関係がなければ，その各々，またはそれらをいくつかまとめたグループの各々は，並列に実行できる．たとえば，ループ

```
for (i = 0; i < 1000; i++) {
  a[i] = a[i]*a[i];
}
```

では，a[0] = a[0]*a[0]; a[1] = a[1]*a[1]; ... は互いに独立に実行できる．ところで，ループを制御する変数 i は，前回の i の値に 1 を加えるという形で変化するので，i の値はループの前回の値に依存する．このように繰り返しのたびにある固定数 c だけ増減する整数型の変数をインダクション変数という．インダクション変数は，一般に，何回目のループであるかを表すループインデックスを使って

初期値 + c * ループインデックス　　…（式 1）

と表現することができる．この例では，i の値域を 0 から 499 までと 500 から 999 までというようにいくつかの区域に分割し，

```
for (i = 0, i < 500, i++) { ... }
```

をスレッド 1 に，

```
for (i = 500, i < 1000, i++) { ... }
```

をスレッド 2 にというように振り分け，i をスレッドごとに別々の変数領域に置くならば，スレッド間に依存関係はなくなる．1 つのループ内にインダクション変数が他にもある場合，それらは上記の i のような基準とするループインデックスから（式 1）で算出できるので，繰り返しの前回の値を使わない形にできる．したがって，上記のようなループは，インダクション変数をスレッドごとに別々に持つことによって，do-all 型ループとして並列実行できる．

配列要素の総和を求める

```
for (i = 0; i < n; i++) {
  sum = sum + a[i];
}
```

のような，複数のデータから 1 つの値を算出するループをリダクション演算と呼び，算出する値を表す sum のような変数をリダクション変数と呼ぶ．COINS がサポートするリダクション演算には，

(a) プラス演算子による総和演算
(b) マイナス演算子によって初期値から配列要素の値を次々と引いてゆく演算
(c) 乗算演算子により配列要素の値を掛け合わせてゆく演算

がある．この例では，sum が各繰り返し間で依存関係を持つので，そのままでは並列実行できないように見える．この型のループも，繰り返しの範囲を分割してスレッドごとに割り当て，各スレッドで別々の変数に部分和を求め，それらを合計して総和を求める形にするならば並列化可能となるので，do-all 型ループとして扱うことができる．他の演算子によるリダクション演算についても同様に，部分配列に対するリダクション演算の値から全体の配列に対する値を合成することができる．

繰り返しの間に依存関係のある変数でも，スレッドごとに個別に持つならば並列実行に伴う干渉を避けることのできるものがある．このような変数はスレッドごとに異なるメモリに割り付ける．これを「その変数を private 化する」という．各繰り返しの中で値が設定されたあと参照される変数，すなわち i 回目の繰り返しで参照され

```
#pragma omp parallel for [clause , ... ]
for (index = first ; test_expr ; increment_expr )
{
  body of the loop
}
```

図-2 OpenMP parallel-for 構文

る値は同じ i 回目の繰り返しで計算しなおされている変数については，private 化することによって依存関係をなくすことができる．インダクション変数は private 化する．

ループの最後の回で設定した値がループを出てから参照される変数は，last-private 変数という．これについては，private 化して計算したあと，do-all 型ループを出たところで最終回の値を元の変数に設定すればよい．

●データの依存解析とループの正規化

ループ並列化の処理では，制御フローグラフの経路に沿って，変数がアクセスするメモリ領域を解析する．インダクション変数とリダクション変数以外について，異なるループ繰り返しで同じ領域をアクセスすることがない場合は，繰り返しの順序をどのように変更しても逐次に実行したときと等しい結果となるので，並列化可能であると判断する．そうではなく，i 回目の結果を i+1 回目で使うような，ループ繰り越し依存関係があるときは，do-all 型の並列化はできないと判断する．そのため，すべての変数と配列のメモリアクセス領域を解析し，private 化の可能性とともに領域の重なりの有無を調べて，ループごとに並列化可否を判定する．そのために，まず簡単な検査で解析対象とするループを選別し，そこで除外されなかったループについて，最内側ループから順番に外側に向かって，ループごとに次の（1）～（4）の処理を行う．

(1) インダクション式の変換と配列添字の正規化
　ループインデックスを表す基本インダクション変数（たとえば i）を導入し，ループを
　`for (i= 0; i < n; i= i+1) { .... }`
　の形に変形する準備をし，「j = j±c（c はループ不変式）」の形で定義されるその他のインダクション変数 j をすべて
　`j=（j の初期値）± i * c;`
　という形に変換する準備をする．
(2) メモリアクセス領域の計算
　制御フローに沿って，基本ブロックの情報から，ループ i 回目の繰り返し本体でのアクセス領域，さらにループ全回でのアクセス領域を計算する．
(3) 並列化可否の判定
　ループ全回でのアクセス領域の重なりを調べ，各繰り返しを独立して実行できるかどうかを判断する．
(4) HIR 変換
　並列化できると判断したループに対して，上記で準備したインダクション式の書き換えや，変数の private 化などを行う．インダクション変数はすべて private もしくは last-private と指定する．

先に述べた，簡単な検査で解析対象から除外するループとは，次のものである．
(a) 収束計算のように繰り返し回数がループ内の計算結果に依存する．
(b) ループの外への飛び出しや外からの飛び込みがある（ループの出口や入り口が複数ある）．
(c) 副作用なしと宣言された組み込み関数以外の関数を呼び出している．
(e) if-then-else に構造化できない if-goto による分岐がある．
(f) 実数や整数の単純変数や配列という制限に収まらない型のデータが定義・参照される可能性がある．

なお，後述するように，COINS には並列化の候補とするループをプラグマで指定する機能があるが，候補とされたループであっても，上記の特性を持つものは除外される．

● OpenMP/C プログラムの生成

(1) OpenMP の構文

COINS のループ並列化では，OpenMP の指示文のうち，parallel-for 構文だけを扱う．OpenMP の具体形式は C 言語対応，Fortran 対応などによって異なり，C 言語対応の parallel-for は**図-2** の形式をとる[2]．

OpenMP の並列実行モデルは，主に共有メモリの fork-join 型であり，OpenMP コンパイラ Omni[3] は，親となるマスタースレッドが子供のスレッドのチームを生成して並列処理を実現している．parallel-for 構文は，ループの繰り返しをスレッドに割り当てて並列に実行させる指定である．

parallel-for の for 文の先頭部分は，符号付き整数のループインデックスを用いて

for（初期値代入文；上限値との比較条件式；増分式）

という形にしなければならない．上限値や増分式は全スレッドで同じ値でなければならないので，実際にはループ不変式となる．`#pragma` の `[clause, ...]` では，private 変数やリダクション演算を指定する．COINS ではこの構文に合うようにループを正規化する．OpenMP

```
int k, c[50];
int main()
{
  int x[50];
  int i,j,n, sum;
  n=50;
  k=100;
  sum = 0;
#pragma parallel doAll
  for (i=0; i<n ; i++) {
    x[i]= k + i;
    c[i]= i * i;
    k=k-2;
    sum = sum + x[i];
  }
  for (j = 0; j < n; j++)
    printf(" x[%d]=%d c[%d]=%d", j, x[j], j, c[j]);
  printf("¥nk = %d sum = %d ¥n", k, sum);
  return 0;
}
```

図-3 do-all 型ループ並列化の例題

```
int k, c[50];
int main( )
{
  int x[50];
  int i, j, n, sum;
  n = 50;
  k = 100;
  sum = 0;
#pragma omp parallel for lastprivate(k,i)
     reduction(+:sum)
  for ( i = 0; i < n; i = i + 1) {
    k = (-2) * i + 100;
    x[i] = k + i;
    c[i] = i * i;
    sum = sum + x[i];
  }
  _lab4:;
  k = (-2) * i + 100;
  for ( j = 0; j < n; j = j + 1) {
    (&printf)(" x[%d]=%d c[%d]=%d",
      j,x[j],j,c[j]);
  }
  _lab8:;
  (&printf)("¥nk = %d sum = %d ¥n",k,sum);
  return 0;
}
```

図-4 生成される OpenMP/C プログラム
（見やすくするために整形）

の詳細な仕様については，文献 2）やその Web ページ [3] を参照されたい．

（2）OpenMP による並列化の例

COINS で OpenMP/C のプログラムを生成するには，次のコマンドを用いる．

```
java coins.driver.Driver
    -coins:parallelDoAll=OpenMP foo.c
```

これは，ループ並列化により解析と変換を行い，必要な OpenMP 指示文の情報を付加した HIR を作り，それを COINS の hir2c で OpenMP/C プログラムに変換して処理を終了する．上記の例だと foo.c というソースプログラムに対して，foo-loop.c という OpenMP/C プログラムを生成する．

例として，**図-3** のソースプログラムを並列化すると **図-4** に示すようなプログラムが生成される．hir2c で生成したプログラムそのものは，マクロ定義や型指定などを含んで見づらいので，ここでは少し整形して示す．

#pragma parallel doAll は，その直後のループを並列化候補とするという指定である．並列化で高速化できるか否かを自動判定することはむずかしいので，プラグマで候補をあげてもらったあと，並列化の条件に合っているか否かを自動判定する．この例では，配列 x と c の要素には繰り返しにまたがる依存関係はない．k は -2 ずつ変わるのでインダクション変数であり，sum は総和を計算するためのリダクション変数である．k は大域変数で，そのままではスレッド間で読み書きの競合が発生するため，private 化する必要がある．インダクション変数 k は，先の（式 1）を使って，ループインデックス i から計算されている．付加された OpenMP の指示文

```
#pragma omp parallel for lastprivate(k,i)
    reduction(+:sum)
```

は，上記のことを表している．reduction(+:sum) の + は，sum を求めるときの演算子を表す．

● 並列実行のフレームワーク

コンパイラは一般にプロセッサアーキテクチャ対応に作るものであるが，並列化の仕方は OS やメモリ構成，通信方式などによって異なるので，並列化をコンパイラだけで行うのではなく，実行環境に即した実行時ルーチンと組み合わせる方が適用性を高くでき，共通インフラストラクチャ向きである．

COINS では，OpenMP コンパイラ等を使わない場合にも並列実行を容易に実現できるようにするため，実

```
 1) #pragma parallel doAllFunc main
 2) #ifndef MAIN
 3) #define MAIN
 4) #endif
 5) #include "coinsParallelFramework.h"
 6) int printf(char*, ...);
 7) int k, c[50];
 8) int main()
 9) {
10)   int x[50];
11)   int i,j,n, sum;
12) #pragma parallel init
13)   n=50;
14)   k=100;
15) #pragma parallel doAll
16)   sum = 0;
17)   for (i=0; i<n ; i++) {
18)     x[i]= k + i;
19)     c[i]= i * i;
20)     k=k-2;
21)     sum = sum + x[i];
22)   }
23)   for (j = 0; j < n; j++)
24)      printf(" x[%d]=%d c[%d]=%d", j, x[j], j, c[j]);
25)   printf("¥nk = %d sum = %d ¥n", k, sum);
26) #pragma parallel end
27)   return 0;
28) }
```

図 -5 並列化コード生成用のプログラム例

行環境に依存しない並列化フレームワークを定めている．そのインタフェースに合わせた実行時ルーチンが用意されていれば，COINS で生成された機械語コードが並列実行できる．組み込みマイコンなどでは，さまざまな OS が使われ，OS のないことも多い．COINS はそのような環境でも使うことができる．

COINS の並列化フレームワークは，実行時ルーチンを容易に作ることができるよう，単純化してあり，

スレッド定義
スレッド初期化
スレッド起動
スレッド終了確認
通信
排他制御

を基本とする．POSIX Thread[4] などの利用できる環境では，インタフェースを合わせるだけで COINS の生成するコードを並列実行できる．

並列化フレームワークの詳細は COINS の Web ページ[1] に譲るとして，ここでは例で説明する．図 -3 のプログラムから並列実行用の機械語を生成するには，**図 -5** のように，このフレームワークに合わせて並列化用のプラグマ等を挿入する．

図 -5 の左端には説明用に行番号をつけてある．1 行目のプラグマ `parallel doAllFunc` では，ループ並列化の対象とする関数を列挙する．5 行目は並列化のフレームワークで使う変数や関数プロトタイプの宣言を取り込む include 文である．その中の変数宣言に対しては，コンパイル単位が `main` を含むか含まないかで実体を定義するか外部参照とするかなどの違いがあるので，2 ～ 4 行目で，これは `main` を含むコンパイル単位であることを示している．12 行目のプラグマ `parallel init` は，並列化の準備を行う実行時ルーチンの呼び出しに変換される．15 行目のプラグマ `parallel doAll` は，並列化の候補とするループがこの後ろに続くことを示している．26 行目のプラグマ `parallel end` は，並列実行の処理を終了させる処理を表す．

do-all 型ループの並列化では，スレッドを起動（fork）する側をマスタースレッド，起動される側をスレーブスレッドという．HIR の並列化変換部では次の処理を行う．

(1) ループ部分を関数に作り替える．
(2) ループのインデックス値域をスレッドごとにどう配分するかなどを決める実行時ルーチンを呼び出す．
(3) スレッドに渡す引数を用意する．
(4) スレーブをいくつか起動する．
(5) マスターでもループのインデックスの 1 つの値域に対する実行を行う．
(6) スレーブの終了確認待ち（join）をする．
(7) スレーブの算出する値からリダクション演算の値を合成する．

● ループ並列化の変換例

COINS でループを並列化して機械語コードを生成するには，

```
java coins.driver.Driver -S
     -coins:parallelDoAll=n foo.c
```

というコマンドでコンパイルする．`n` としては並列度を指定する．`foo.c` は対象とするプログラムを示す．図 -5 の例をコンパイルすると，スレーブで実行する関数として**図 -6** に対応する HIR を生成する（マスターで実行する部分の概要は前節で示した通りである）．ここ

```
1) void * main_loop_0( int _threadId_0,
       long _indexFrom_0,long _indexTo_0,
       int * sum_0,void * _voidPtr_0 )
2) {
3)    int i, n, k_0, sum;
4)    coinsParallelPpreprocessForDoAllThread();
5)    sum = 0;
6)    k_0 = _k_global_0;
7)    { i = 0; }
8)    for ( i = _indexFrom_0; i < _indexTo_0;
            i = i + 1) {
9)       k_0 = (-2) * i + 100;
10)      _x_global_0[i] = k_0 + i;
11)      c[i] = i * i;
12)      sum = sum + _x_global_0[i];
13)   _lab25:;
14)   }
15)   _lab29:;
16)   *sum_0 = sum;
17)   coinsParallelPostprocessForDoAllThread();
18) }
```

図 -6　スレーブが実行するループ部分（HIR を C に変換し整形）

で，下線 (_) で始まる変数と _0, _1, ... で終わる記号はコンパイラが生成した変数や関数である．

図 -6 の 1 行目は次のことを表している．

(a) main の 0 番目のループに対する関数は main_loop_0 として呼ばれる．

(b) このスレッドの受け持つインデックス値域は _indexFrom_0 から _indexTo_0 である．

(c) リダクション演算の部分和を書き戻す先は sum_0 である（_voidPtr_0 はリダクション変数が 2 つあるときに使われるもので，この例では使われない）．

5 行目ではスレッドの前処理を行う．k は大域変数であるが last-private 変数なので，局所変数 k_0 として private 化し，初期値を _k_global_0 として受け取る．i は private 化されている局所変数で初期値を 0 と設定される．

8 行目のループではインデックスの値域が _indexFrom_0 から _indexTo_0 までと書き換えられている．配列 x はマスターの局所変数であるが，アクセスする要素はスレッドごとに異なり競合しないので，そのアドレスを _x_global_0 として渡してもらい，各スレッドで読み書きする．_k_global_0，_x_global_0 は自動生成された大域変数であり，その値はマスターでスレーブを起動する前に設定する．配列 c は大域変数なので，11 行目でそのまま使っている．12 行目で計算するリダクション変数 sum の部分和は，書き戻し先として sum_0 で指定された所へ 16 行目で書き込んでいる．17 行目では，スレーブの終了告知などをする後処理を行う．

並列化フレームワークとして実現した実行時ルーチンは，インライン展開などの機能で呼び出し側コードに埋め込めばオーバーヘッドを削減できる．並列化変換部は，現在，データ共有度が非常に高い上記方式のみを実現しているが，初期化部に共有メモリに関するパラメータを設けるなどすれば，スレッド間のデータ共有度を下げたコード生成も可能になる．

COINS による並列化を，FPGA（Field Programmable Gate Array）に複数個のソフト CPU コア MicroBlaze[5] を実装した廉価なマルチコア環境で実現してみた．その実行時ルーチンは OS を使わない形で作成した．MicroBlaze 用のコンパイラは，COINS でそれ用のマシン記述を作ることにより，約 2 カ月で実現できた．

3 つの MicroBlaze プロセッサを 1 チップに載せたものでは，Laplacian filter のようなメモリアクセスの負荷が大きくないプログラムであれば，それなりの実行性能（Laplacian filter で 2.9 倍）が得られ，オーバーヘッドの少ない並列化が実現できることを確認できた．

SMP 向け粗粒度並列化

並列化の方法として，プログラムをループや副プログラムなどの粗粒度タスク（macro task）に分解して並列実行させる粗粒度並列化[6), 7)] がある．COINS では，SMP を対象とした粗粒度並列化を行い，結果を OpenMP/C プログラムとして出力する粗粒度並列化コンパイラ CoCo（Coins based Coarse grain parallelizing Compiler）[8)] を実現している．以下に，その概要を述べる．

● 粗粒度並列化の処理の流れ

粗粒度タスクをループや副プログラムなどといったが，正確には，プログラムのある一部分で，その実行を開始する文がその部分の先頭の行だけである（途中への飛び込みがない）ものと定義される[6)]．したがって，基本ブロックも 1 つの粗粒度タスクとして扱うことができる．CoCo では，プログラムを粗粒度タスクに分解し，それらの間の制御フローとデータ依存関係をマクロフローグラフ[6), 7)] と呼ぶ非循環の有向グラフとして表現して，各タスクの実行開始条件を求める．実行開始条件とは，「プログラム実行中に，どのような条件が揃えばそのタスクを実行してよいか」ということを，タスクごとに論理式で表現したものである．実行時には，実行開始条件が成立したものを順次スレッドとして起動するこ

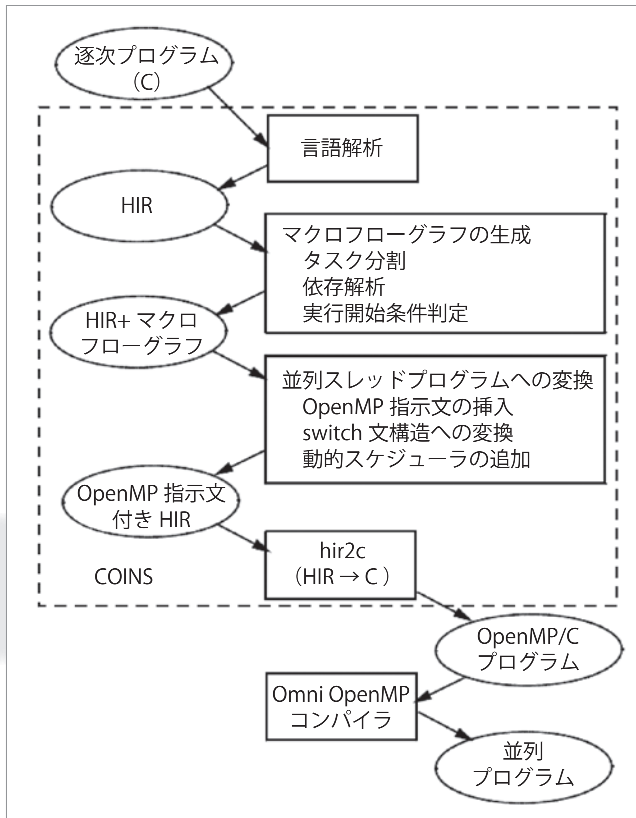

図 -7 SMP 向け粗粒度並列化の処理の流れ

とで並列処理を制御する．全体の処理の流れを**図 -7** に示す．

● マクロフローグラフの生成

並列化の処理を例で説明する．**図 -8** のプログラムを粗粒度タスク MT1 〜 MT4 に分解し，行 5, 6, 7 を MT1，行 8 〜 11 を MT2，行 13 〜 16 を MT3，行 17 を MT4 と表したとすると，そのマクロフローグラフは**図 -9** のようになる．ここで，実線矢印は制御の流れ，点線矢印はデータの生成・利用を表すフロー依存関係を表す．

この場合，開始条件は，

MT1　true　// 最初の時点で実行可能
MT2　1　// MT1 がすむと実行可能
MT3　1　// MT1 がすむと実行可能
MT4　2 ^ 3　// MT2 と MT3 がすむと実行可能

と表される．

● 並列スレッドプログラムへの変換

上記の処理を行った結果を並列実行されるプログラムへ変換するため，HIR 上で，(a) スケジューラの組み入れ，(b) 開始条件を満たすスレッドの番号取得，(c) スレッド番号に合わせて対応する処理を行う switch 文の生成，ならびに，(d) 並列化のための OpenMP 指示文の挿入を行う．上のプログラムは**図 -10** のように変換される．

```
 1) int main()
 2) {
 3)   int i, n=1000, sum, max;
 4)   int a[1000];
 5)   for (i = 0; i < 1000; i=i+1) {
 6)     a[i] = i*(1000-i);
 7)   }
 8)   sum = 0;
 9)   for (i = 0; i < 1000; i=i+1) {
10)     sum = sum + a[i];
11)   }
12)   max = a[0];
13)   for (i = 1; i < n; i=i+1) {
14)     if (a[i] > max)
15)       max = a[i];
16)   }
17)   printf(" sum=%d max=%d¥n", sum, max);
18) }
```

図 -8 粗粒度並列化の例題

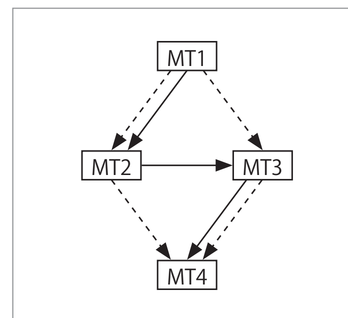

図 -9 マクロフローグラフ
(図 -8 に対応)

● 評価

CoCo で並列化したプログラムを次の実験環境で走行させて並列化の効果を調べた．

OS	Solaris 8
プロセッサ	Sun UltraSPARC2 (450MHz) * 4
メモリ	1GB
Java コンパイラ	JDK1.4.2
C コンパイラ	gcc 2.95.3
OpenMP コンパイラ	Omni OpenMP Compiler 1.4

並列化の対象プログラムとして，SPEC95 ベンチマークプログラムの swim（Shallow water modeling）および tomcatv（Mesh generation with Thompsons solver）をとりあげた．

〈swim による評価〉

swim はほぼ逐次処理される 9 つの粗粒度タスクからなるので，そのままでは並列効果が得られない．ところ

```
1) int main( )
2) {
3)   変数宣言
4)   スケジューラの初期化
5)   int taskNum; /* MT番号 */
6) #pragma omp parallel private(i)
           firstprivate(_mdf_task)
7)   while (1) {
8)     if ( 出口のMT4の処理終了 ) break;
9)     taskNum = getTask( ); /* 次に実行するMTの
                                   番号取得 */
10)    switch (taskNum) {
11)    case 1:  /* MT1 */
12)      for (i = 0; i < 1000; i=i+1) {
13)        a[i] = i*(1000-i);
14)      }
15)      開始条件更新（MT1終了）
16)      break;
17)    case 2:  /* MT2 */
18)      sum = 0;
19)      for (i = 0; i < 1000; i=i+1) {
20)        sum = sum + a[i];
21)      }
22)      開始条件更新（MT2終了）
23)      break;
24)    case 3: /* MT3 */
25)      max = a[0];
26)      for (i = 1; i < n; i=i+1) {
27)        if (a[i] > max)
28)          max = a[i];
29)      }
30)      開始条件更新（MT3終了）
31)      break;
32)    case 4: /* MT4 */
33)      printf(" sum=%d max=%d¥n", sum, max);
34)      開始条件更新（MT4終了）
35)      break;
36)    } /* switchの終わり */
37)  } /* whileの終わり */
38) } /* mainの終わり */
```

図-10 SMP向け粗粒度並列化の変換例

で，その実行時間の97%は粗粒度タスク8で占められており，それは1つの大きいループで，1度の繰り返しの中でさらに複数のループが実行される．そこで粗粒度タスク8を4つの部分に手動で分割したうえでCoCoを使ってコンパイルし，生成されたOpenMP/Cプログラムを上記実行環境でプロセッサ数を4まで変えて走行させた．

その結果，以下のように，逐次実行にくらべ，2並列では1.5倍，4並列では1.9倍の速度向上が得られた．

	プロセッサ数	実行時間（sec）	台数効果
変換前	1	2.52	1.0
	1	2.53	1.0
変換後	2	1.61	1.5
	3	1.60	1.5
	4	1.28	1.9

〈tomcatvによる評価〉

tomcatvは8つの粗粒度タスクで構成されるが，そのままでは並列性があまりないので，実行時間比率の非常に大きい粗粒度タスク5を手動で4つの部分に分割し，CoCoでコンパイルし，実行させた．その結果，以下のように，逐次実行にくらべ，2並列では1.4倍，4並列では1.7倍の速度向上が得られた．

	プロセッサ数	実行時間（sec）	台数効果
変換前	1	0.67	1.0
	1	0.67	1.0
変換後	2	0.47	1.4
	3	0.45	1.5
	4	0.39	1.7

〈SMP向け粗粒度並列化のまとめ〉

CoCoはCOINSを用いてSMP向けの粗粒度並列化の研究的試作を行ったものである．現在はまだmain関数しか並列化しないとか，HIRを並列化向きに変換する機能が十分でないなどの理由により，並列化するにはソースプログラムを手作業で変換する必要があったりする．並列化できる範囲を広げるには，並列化機能の強化ばかりでなく，基盤部の機能強化も必要であるが，COINSは拡張性に優れているので，そのような機能強化は比較的行いやすいと思われる．この試作により，COINSを利用すると，並列化の研究・開発が容易になることが分かった．

おわりに

この連載によって，簡単なコンパイラの作り方から始め，実用的なコンパイラで行っている処理の概要まで，かなり具体的に説明してきた．コンパイラを作るのはだいぶたいへんではあるが，1人あるいは数人でもできないわけではない．とくに，COINSのような基盤が利用できればかなりの品質のものを短期間で作ることができる．COINSを用いて学生実験などで簡単なコンパイラを作成した例や，新しいコンパイル方式の研究を行った例はいくつかある．

コンパイラには興味があるが，自分のやりたいことができるようになるまでの土台作りが大変ではないかと思っている方や，研究開発や製品開発でコンパイラが1つの主要項目になるが，そこまで手を広げるのはむずかしいと思って逡巡しておられる方もいるのではないかと思われる．このような場合，COINSは1つの有力な支援手段となるであろう．

COINSの機能や性能にはまだ不十分な点もあるが，その改良・拡充はCOINSコンパイラ・インフラストラクチャ協会で継続的に進めている．これがコンパイラ関連分野の技術の普及と向上に役立つことがあるならば幸いである．

謝辞　COINSの開発にご尽力いただいた方々とご支援いただいた方々，ならびに有益な助言をいただいた閲読者の方々と中田育男教授に深く感謝します．

参考文献

1）COINSの解説 , http://www.coins-project.org/COINSdoc/
2）Chandra, R. et al.: Parallel Programming in OpenMP, Morgan Kaufmann Publishers (2001).
3）Omni OpenMP Compiler Project : http://phase.hpcc.jp/Omni/
4）ISO/IEC 9945-1:1996 Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) [C Language]
5）Xilinx : MicroBlaze Architecture, http://www.xilinx.com/
6）本田弘樹，岩田雅彦，笠原博徳：Fortranプログラムの粗粒度タスク間の並列性検出手法，電子情報通信学会D-I, Vol.J73-D-I, No.12, pp.951-960 (Dec. 1990).
7）石坂一久，小幡元樹，笠原博徳：OpenMPを用いた粗粒度並列処理，情報処理学会計算機アーキテクチャ研究会，Vol.2000, No.139, pp.187-192 (Aug. 2000).
8）池田倫久，Ngo Tau Van，田中雅俊，福岡岳穂，片桐孝洋，本田弘樹，弓場敏嗣：粗粒度並列化コンパイラCoCoの開発，情報処理学会研究報告 IPSJ SIG Technical Report 2004-HPC-98 (7), pp.37-42 (Apr. 2004).

（平成18年11月1日受付）

▶渡邊　坦（正会員）
tan@watanabe.ai.to
1962年京都大学理学部数学科卒業．日本IBM（株），（株）日立製作所，電気通信大学を経て，現在電気通信大学名誉教授．工学博士．主として言語処理系に興味を持つ．ACM，日本ソフトウェア科学会各会員．

▶岩澤京子（正会員）
kiwasawa@cs.takushoku-u.ac.jp
1981年東京農工大学工学部卒業．（株）日立製作所中央研究所，東京農工大学工学部を経て，現在拓殖大学工学部情報工学科助教授．工学博士．コンパイラの並列化および最適化技術に興味を持つ．

▶藤瀬哲朗（正会員）
fujise@mri.co.jp
1984年電気通信大学大学院修了．同年（株）三菱総合研究所入社．1992年（財）新世代コンピュータ技術開発機構出向．現在（株）三菱総合研究所主任研究員．並列記号処理，並列化コンパイラ技術に興味を持つ．

▶弓場敏嗣（正会員）
yuba@is.uec.ac.jp
1966年神戸大学大学院工学研究科修士課程修了．（株）野村総合研究所を経て，1967年通商産業省工業技術院電子技術総合研究所（現在，（独）産業技術総合研究所）に入所．以来，計算機のオペレーティングシステム，見出し探索アルゴリズム，データベースマシン，データ駆動型並列計算機などの研究開発に従事．その間，計算機方式研究室長，知能システム部長，情報アーキテクチャ部長等を歴任．1993年より，電気通信大学大学院情報システム学研究科教授．並列処理・分散処理の科学技術一般に興味を持つ．工学博士．本会，電子情報通信学会各フェロー．日本ソフトウェア科学会，日本ロボット学会，ACM，IEEE各会員．

▶福岡岳穂（正会員）
Takeaki_Fukuoka@hq.scei.sony.co.jp
2001年電気通信大学大学院情報システム学研究科卒業．（株）管理工学研究所を経て現在（株）ソニー・コンピュータエンタテインメント．コンパイラ，並列／分散アルゴリズム，計算機アーキテクチャに興味を持つ．日本ソフトウェア科学会会員．

21世紀のコンパイラ道しるべ‥連載を振り返って

本連載担当：鈴木　貢

連載の種となったCOINSは，2000～04年度に実施された「並列化コンパイラ向け共通インフラストラクチャの研究」というプロジェクトから生み出されました．その成果を使って，コンパイラ初学者向け，あるいはリフレッシュセミナを意図して，また，少しだけCOINSの宣伝を狙って，この連載が企画されました．当初の予定では8回の連載だったのですが，最終回の内容が1つの号では収まりきれなくなったので，9回まで延長させていただきました．延長をお許しいただいた編集委員各位に感謝いたします．

プロジェクトは終わりましたが，COINS自体はCOINSコンパイラ・インフラストラクチャ協会のメンバによるボランティアベースで日々進化を続けていて，CVSの中の現在約33万行の.java（8Mバイトの.classファイル）や特にCOINSのwebページ http://www.coins-project.org/ 中のドキュメントは，着実に更新されています．8回が9回になったのも，企画の当初に予想していなかったIPAの次世代ソフトウェア開発事業でのCOINSを応用した成果を盛り込んだためです．今後のCOINSにご期待ください．

ところで9回の連載では，語りつくせなかったことが多々あるかと思います．内容の詳細については，上記webページの内容をご覧いただけると幸いです．質問やご意見・ご感想は，会員の広場や，各号の著者のメールアドレスまでお願いします．

最後に，各号で原稿に対して適切なご意見をいただいた担当エディタを，感謝の意を込めてご紹介し，本連載の締めといたします．
（敬称略）大城正典，大野　晋，小幡元樹，河辺義信，笹島宗彦，佐藤浩史，白井良成，酢山明弘，祖父江恒夫，濱　利行，松尾健史，望月　源