自動並列化コンパイラ TINPAR について

田中 慎司

1997/05/08

1 はじめに

シミュレーションや数値解析などの大規模科学技術演算に用いられるスーパーコンピュータに対し複雑かつ大規模な演算をより快適に行うためにさらなる演算処理能力の向上が要求されている。そのような中で並列計算機は、現在主流のベクトル計算機に代わる次世代のスーパーコンピュータとして大きな期待が寄せられている。しかし、並列計算機の能力を引き出すには、その上で動かすプログラムもそれに対応していなければならない。ところが、そのようなプログラムの開発は、通信・同期処理などために高度かつ複雑になる。そのため、既存の言語で書かれた逐次実行のプログラムを、並列計算器用のプログラムに変換する処理計の開発が望まれている。

マルチプロセッサシステム上で単一のプログラムを並列処理するために必要な並列化コンパイラの諸技術について解説する。

2 並列化コンパイラの基礎技術

2.1 タスクとその粒度

タスクとは、プロセッサに割り当てる処理の単位のことである。プログラムを分割してタスクを生成する。タスクの粒度とは、タスクの大きさのことである。粒度をどのように設定するかは、並列処理を向上させる上で非常に重要な項目である。

細粒度の並列処理 細粒度の並列処理は、機械命令レベルの並列処理のことである。

中粒度の並列処理 中粒度の並列処理は、Doループなどのループに対して行われる。各イタレーションを別々のプロセッサに割り当てて並列に実行する。

粗粒度の並列処理 粗粒度の並列処理は、Doループやサブルーチンなどの並列性を引き出すものである。

2.2 並列性

一般の逐次プログラムには、計算結果を得る上で、本質的ではない記述上の順序関係が存在する。これを並列性という。

```
T1:A=50
T2:B=A*10
T3:C=B*2
T4:A=A+B
T5:B=A+C
```

上記の例では、T3 と T4 は、順序を入れ換えてもよく、また同時に実行してもよい。つまり、並列性があるということである。

並列性の検出は、結果を得るのに本質的でない記述上の順序関係を除去し、元プログラムの意味を保つために必要最小限の制約 (先行制約) を求めることにより行う。先行制約はグラフなどによって表現する。

2.3 データ依存

条件分岐が存在しないプログラムに対しての先行制約は、タスク間でのデータ依存関係に

よって記述される。

```
T1:A=1.0
T2:B=A+3.1415
T3:A=1/3*(C-D)
T4:A=(B*3.8)/2.718
```

フロー依存 T2 では、T1 で定義された変数 A の値が使われる。その関係を T1 から T2 へのフロー依存という。

逆依存 T2 で、変数 A の値が使われているが、T3 で A に代入をしている。これを T2 から T3 への逆依存という。

出力依存 T3 で代入されている変数と同じ変数に T4 でも代入している。これを出力依存という。

2.3.1 制御依存

条件分岐がタスク館に存在する場合には、データ依存関係に加えて制御依存関係が生じる。

コントロールフローグラフ上のノード X と Y において、

1. X から Y に至るパスで、そのパス上のすべてのノード (X と Y をのぞく) を Y がポストドーミネートしているようなパスが存在し、
2. X は Y にポストドーミネートされていない、

場合の限り、Y は X に制御依存している。

2.3.2 スケジューリング

タスクのプロセッサへのスケジューリングにおいてすることは、各プロセッサで実行すべきタスクの決定と、タスクの実行順序の決定である。

また、スケジューリングをいつ実行するかによって以下の二つに分類できる。

スタティックスケジューリング スケジューリングをコンパイル時に行う。実行すべきタスクと各タスクの処理時間が実行前に正確に得られる場合には、精度のよいスケジューリングが可能となる。

ダイナミックスケジューリング スケジューリングをプログラム実行時に行う。条件分岐などの存在により、どのタスクを実行すべきかが実行時に決定されたり、各タスクの実行時間が実行前には予測できないような場合に使われる。スケジューリングのための実行時のオーバーヘッドは大きい。

section 逐次ループの並列実行

2.4 DOALL

各イタレーションのループボディの処理をタスクとし、各イタレーションをそれぞれのプロセッサに割り当て並列処理をする。ループボディはプロセッサ内で逐次処理をする。

DO ループの各イタレーション間に依存関係がない、つまり、イタレーション間での並列性を利用している。

```
do I=1,N
  A(I)=B(I)+C(I)
  D(I)=A(I)-3
enddo
     ↓
doall I=1,N
  A(I)=B(I)+C(I)
  D(I)=A(I)-3
enddoall
```

```
P1
A(1)=B(1)+C(1)
D(1)=A(1)-3
```

```
P2
A(2)=B(2)+C(2)
D(2)=A(2)-3
```

各プロセッサは割り当てられたイタレーションを実行したあと、他のプロセッサと同期をとって次の処理へ進む。

DOALL のスケジューリングには以下のものがある。

1. スタティックスケジューリング

   (a) 各プロセッサにプロセッサ台数間隔のイタレーションを割り当てる。

   ```
   do I=p,N,numberP
   ```

   ただし、p はプロセッサ番号、numberP はプロセッサ台数。

   ```
   P1
   do I=1,N,8
   P2
   do I=2,N,8
   ...
   P8
   do I=8,N,8
   ```

   (b) 各プロセッサに連続したイタレーションを割り当てる。

   ```
   do I=[N/numberP]*(p-1)+1,
            min(N/numberP*p,N)
   P1
   do I=1,20
   P2
   do I=21,40
   ```

2. ダイナミックスケジューリング

   (a) セルフスケジューリング (Self-Scheduling)) ループ制御変数 I を全プロセッサからアクセスできる共有変数 global_I として、各プロセッサが global_I をインクリメントしながら次のイタレーションを実行していく。

   ```
   lbl:( I=global_I;
         global_I=global_I+1)
       if I>N exit
         ループボディ
       goto lbl
   ```

   (b) チャンクスケジューリング (Chunk-Schedulig) 複数イタレーションを単位としてプロセッサに割り当てることにより共有変数 global_I へのアクセスオーバーヘッドを軽減する。

   ```
   lbl:( J=global_I;
         global_I=global_I+k)
       if J>N exit
       do I=J,min(J+k-1, N)
         ループボディ
       enddo
       goto lbl
   ```

   (c) ガイデッドセルフスケジューリング (Guided Self-Scheduling) ループ処理が進むに従って、チャンクサイズを次第に小さくすることにより、共有変数 global_I へのアクセスを軽減するとともに、ロードバランスを保つ。

   チャンクサイズ

   ```
   k=[Nrem/numberP]
   ```

   (共有変数 Nrem は、未割り当てイタレーション数で初期値は N)

   ```
   lbl:(k=[Nrem/numberP];
        Nrem=Nrem-k)
   ```

```
if k=0 exit
(J=global_I;
   global_I=global_I+k)
do I=J,J+k-1
  ループボディ
enddo
goto lbl
```

2.5 DOACROSS

イタレーション間にデータ依存が存在する場合、DOALL は付加のため、イタレーション間で同期をとることにより並列実行する DOACROSS が使われる。

3 並列化コンパイラ TINPAR

TINPAR は、メッセージ交換型並列計算機のための並列化コンパイラである。TINPAR は、owners computes rule に従って、拡張 Tiny Language で書かれた逐次プログラムから C 言語による SPMD プログラムを生成する。owners computes rule とは、各プロセッサが、それぞれが所有するデータのための演算を担当する戦略のことである。拡張 Tiny Language は、Fortran に類似した簡潔な構文規則を持つ言語である。

3.1 内部構成

TINPAR は、以下のように独立した 6 つのモジュールから構成される。

1. 構文解析部
   ソースプログラムの構文解析を行い処理系の内部表現に展開する。

2. 前処理部
   逐次プログラムにはそのままでは並列化に適さない部分が存在することがある。前処理部ではこのような部分に対してプログラム変換を行うことにより並列化に適した形に変形する。

3. 並列化部
   owners computes rule に従ってソースプログラムを並列化する。具体的にはデータを所有するプロセサを判定するための if 文やプロセッサ間のデータ参照を行うための通信コードの挿入を行う。また、この時点では分割されたデータに対して分割前のデータ全体が格納できる大きさのメモリ領域を確保する。このため最適化時には、分割前の単純なアドレッシングを使用することができ、データ依存関係解析が容易になる。

4. 最適化部
   並列化部で用いられる並列化手法は究めて単純なため、生成コードには無駄が多く効率が良くない。そこで、最適化部では、不要なコードの削除、通信コードのリスケジューリングといった最適化を行う。

5. メモリ再割り当て部　分割されたデータに対して、プロセッサが所有するデータ分だけメモリ領域を確保するように、メモリの再割り当てとアドレッシングの修正を行う。

6. コード生成部
   処理系の内部表現から C 言語で記述された目的コードを生成する。

4 並列化手法

強力な最適化手法の適用を前提に、TINPAR では owner computes rule にもとづく単純な並列化手法を採用している。ここでは並列化部およびメモリ再割り当て部で用いられる並列化/メモリ再割り当ての手法について述べる。

1. データの分散
2. 演算の分割とプロセッサへの割り付け
3. 通信コードの挿入

4.1 データの分散

owner computes rule に従って逐次プログラムを並列化する場合には、各プロセッサに対してデータが分散されていることが前提となる。TINPAR では、単一次元方向に対する任意のブロックサイズのブロックサイクリック分割を扱うことができる。この時点では、データ分割が施された配列に対しても配列全体を格納できる大きさのメモリ領域を各プロセッサで確保する。

```
real a(0:n - 1:*) /* a() はブロック分割 */


for i = 0, n-1 do
 a(i) = 0.0
endfor
            ↓
real a(0:n - 1:*)


for i = 0, n-1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  a(i) = 0.0
 endif
endfor
```

図 1: 演算の分割とプロセッサへの割り当て

4.2 演算のプロセッサへの割り付け

データの分散が行なわれると、owner computes rule に従ってソースプログラム中の演算を分割し、各プロセッサに割り当てる事が可能となる。この処理は、以下のように文を単位として行なうことができる。(図 1)

- 変数宣言文
  並列化部では、データ分割が施された配列の要素に対するアドレッシングを変更しないため、配列の大きさの宣言自体も変更しない。このため、メモリ再割り当て部では、そのような配列の要素数を、各プロセッサの所有する分だけになるように縮小する必要がある。
- 代入文
  左辺値すなわち代入先の変数が、データ分割された配列の要素である場合には、その代入文は当該データの所有プロセッサのみによって実行されなければならない。そこでこのような代入文には、図 1のように配列要素の所有者プロセッサ番号と自分のプロセッサ番号を比較する条件文を挿入しなければならない。ここで OWNER() 関数は指定されたデータの所有者プロセッサ番号を返し、GET_CELL_ID() 関数は自分のプロセッサ番号を返す。また、右辺にデータ分割された配列の要素が現れる場合にはメッセージ通信のためのコードを挿入する必要がある。
- for、if、および while 構文
  全プロセッサで実行されるため、変更を受けない。

4.3 通信コードの挿入

TINPAR による並列化では、他のプロセッサに割り当てられているデータの参照はメッセージ通信によって実現される。このためデータ分割された配列の要素を代入文の右辺で参照している場合には、以下のような処理が行なわれる。(図 2)

1. 新たなスカラ変数を宣言する。
   参照者プロセッサによって受信される当該データの値は、一時的にこのスカラ変数に格納される。
2. SEND 文および if/endif 文を挿入する。
   当該データを送信するための SEND 文、および所有者プロセッサを判定するための if/endif 文を当該代入文の前に挿入する。

```
real a(0:n-1:*)
real b(0:n-1)
/* a(i) はブロック分割　b(i) は分割なし */

for i = 0, n-1 do
 b(i) = a(i)
endfor
            ↓
real a(0:n-1:*)
real b(0:n-1)
real tmp_a

for i = 0, n-1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  for cell_id = 0, プロセッサ数-1 do
   SEND(cell_id, a(i))
  enddo
 endif
 RECV(OWNER(a(i)), tmp_a)
 b(i) = tmp_a
endfor
```

図 2: 通信コードの挿入

3. RECV 文を挿入する。
   当該データを受信するために、当該代入文の直前に RECV 文を挿入する。

4. 当該代入文の右辺の参照を置き換える。
   当該データの参照を 1. で宣言したスカラ変数への参照に置き換える。

5 並列化に関する部分の最適化

TINPAR は、C 言語によるオブジェクトコードを生成する。このため対象並列計算機に依存した低レベルの最適化は C コンパイラにまかせることが可能である。そこで TINPAR の最適化部では、並列化に密接した最適化のみを行う。

5.1 ランタイムオーバーヘッドの削減

TINPAR の並列化部が生成するコードには、データ分割が施された配列要素の所有者を判定するための if 文のように、実行時にオーバヘッドとなる文が多数挿入されている。ここでは、これらのうち不要なものを削除し、ランタイムオーバーヘッドの削減するための最適化手法について述べる。

5.1.1 ループの実行範囲の縮小

並列化部の生成するコードには、図 3のような for ループが多数現れる。このようなループでは、if 文の条件式が成立しない場合には無意味なイタレーションを実行することになる。そこで、そのような条件式が成立する範囲にループの実行範囲を狭めることによって空回りとなるイタレーションを削除する。

5.1.2 恒真/恒偽となる if 文の削除

ループの実行範囲の縮小を行った結果、if 文の条件式が恒真または恒偽となる場合があるので、コンパイル時に if 文の条件式を静的評価することによって、このような不必要な if 文を削除する。

```
for i = 0, n-1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  a(i) = ...
 endif
endfor
```

↓

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), 0
 if OWNER(a(i)) == GET_CELL_ID() then
  a(i) = ...
 endif
endfor
```

図 3: ループの実行範囲の縮小

```
for i = 0, n - 1 do
 if OWNER(a(i)) == GET_CELLID() then
  a(i) = ...
 endif
 b(i) = ...
endfor
```

↓

```
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(),
≦ i ≦ n - 1} do
 a(i) = ...
 b(i) = ...
endfor

for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(),
≦ i ≦ n - 1} do
 b(i) = ...
endfor
```

図 4: if 文を削除するためのループ分割

5.1.3 if 文を削除するためのループ分割

恒真又は恒偽となる if 文の削除は、図 4のような場合には適用できない。しかしこのような場合にも、ループインデックスのとりうる範囲を分割することによって、if 文の条件式を恒真または恒偽とすることができる。

5.1.4 不要な send/receive 文の削除

メッセージ交換型並列計算機では通信処理にかかるコストは一般に大きいため、不要な通信は可能な限り削除しなければならない。そこで不要な代入操作となる receive 文をデータフロー解析によって検出し、対応する send または broadcast 文とともに削除する。[3]

5.2 通信レイテンシの低減

通信レイテンシは、演算に必要なデータをほかのプロセッサから受信するための待ちによって生じる。ここでは、通信レイテンシを低減し、プロセッサの利用率を高めるための最適化について述べる。

5.2.1 SEND 文のリスケジューリング

メッセージ交換型並列計算機では、送信側プロセッサから送られたデータは、実際に使用されるまで受信側でバッファリングされる。このため送信側プロセッサは、使用されるよりも早い時期にあらかじめデータを送っておくことができる。そこで、SEND 文をリスケジューリングすることによってデータ送信を可能な限り先行させる。具体的には、SEND 文、および SEND 文を含むが RECV 文を含まないブロックを、プログラムの上流方向に移動する。図 5 これにより、通信レイテンシをある程度抑えることができる。

5.2.2 SEND/RECV 文をともに含むループのループ分配

SEND 文のリスケジューリングは、制御及びデータ依存関係による制約を受けるため、通

```
for i = 0, n -1 do
 a(i) = ...
 if OWNER(a(i)) == GET_CELL_ID() then
  SEND(OWNER(b(i)),a(i))
 endif
 if OWNER(b(i)) == GET_CELL_ID() then
  RECV(OWNER(a(i)),tmp_a)
  b(I) = tmp_a
 endif
endfor
```

↓

```
for i = 0, n -1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  SEND(OWNER(b(i)),a(i))
 endif
 a(i) = ...
 if OWNER(b(i)) == GET_CELL_ID() then
  RECV(OWNER(a(i)),tmp_a)
  b(I) = tmp_a
 endif
endfor
```

図 5: SEND 文のリスケジューリング

```
for i = 0, n -1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  SEND(OWNER(b(i)),a(i))
 endif
 if OWNER(b(i)) == GET_CELL_ID() then
  RECV(OWNER(a(i)),tmp_a)
  b(I) = tmp_a
 endif
endfor
```

↓

```
for i = 0, n -1 do
 if OWNER(a(i)) == GET_CELL_ID() then
  SEND(OWNER(b(i)),a(i))
 endif
endfor
for i = 0, n -1 do
 if OWNER(b(i)) == GET_CELL_ID() then
  RECV(OWNER(a(i)),tmp_a)
  b(i) = tmp_a
 endif
endfor
```

図 6: SEND/RECV 文をともに含むループのループ分配

信レイテンシを完全に隠蔽するには十分ではない。例えば、図 6上のような場合には、データの送信から受信までの時間間隔は非常に短く、通信レイテンシが生じると予想される。しかしこのコードに図 6下のようなループ分配を行うと、送信から受信までの時間間隔は前半ループの実行時間程度となり、レイテンシはほとんど隠蔽される。SEND および RECV 文をともに含むループを SEND 文のみを含むループと RECV 文のみを含むループに分配することによって、データ送受信間の時間的距離が拡大する。これにより、通信レイテンシを大幅に削減することができる。

参考文献

[1] H.Zima,B.Chapman 著; 村岡洋一訳:スーパーコンパイラ, オーム社,1995

[2] 三吉郁夫:メッセージ交換型並列計算機のための並列化コンパイラ TINPAR,1995

[3] 後藤慎也:データフロー解析にもとづく並列化コンパイラの通信最適化手法,1995

[4] 本多弘樹; 並列化コンパイラ技術,1996