勉強会資料「自動並列化コンパイラ」

藤原雅宏

平成８年４月２５日

1　はじめに

次世代のスーパコンピュータの構成方式として並列計算機が期待されている。並列化コンパイラは、逐次型言語あるいはその拡張言語で書かれたプログラムを並列化し、並列計算機での実行を可能にするものである。今回は、並列化コンパイラの中でも、逐次型言語を自動的に並列プログラムに変換する自動並列化コンパイラについて述べる。

2　自動並列化コンパイラの役割

並列計算機向けの並列プログラムは、逐次プログラムに比べ開発やデバッグが困難で、そのアルゴリズムを理解するのも難しい。その原因の１つには、プロセス間の同期や通信等を考慮しなければならない点があげられる。又、計算機により並列アーキテクチャが様々なので、特定の計算機を対象としたプログラムとなってしまい、プログラムの可搬性の低下が避けられない。この点、自動並列化コンパイラでは、ユーザーは並列処理を意識することなく従来の逐次計算機と同様の感覚で並列計算機を使用する事ができる。反面、自動並列化コンパイラは、逐次プログラムの隠れた並列性を見つけることはできるが、並列コンピュータに適した新しいアルゴリズムを構築することは困難で、並列処理を明示的に書かれたプログラム程の処理速度の向上は望めない。

3　自動並列化コンパイラ

一般に自動並列化コンパイラは逐次型言語で書かれたプログラムを次の手順で並列化する。

1) データフロー解析

2) 並列実行可能な部分の検出

3) プログラムを並列処理できる形へ変換

4) 並列実行可能部分のプロセッサへのスケジューリング

5) スケジューリング結果に基づいた並列化コードの生成

3.1　データフロー解析

自動並列化を行なう際、コンパイラはまず命令間、サブルーチン間、Do ループ間のデータ依存解析を行なう。データの依存性とは、データの定義・参照の実行順序の関係であり、次に示す 3 種類がある。

1) フロー依存 :ある文で定義した変数を他の文で参照する。

2) 出力依存 :ある文が定義した変数を他の文が再定義する。

3) 逆依存 :ある文が参照した変数を他の文が定義する。

これらは、並列実行の可能性を検出するために使用される。

3.2　並列実行可能な部分の検出

プログラム中の並列実行可能な部分を前節のフロー解析の結果をもとに見つけ出す。見つけ出した部分は、並列処理単位の大きさにより細粒度、中粒度、粗粒度の 3 つに分類できる。

3.2.1　細粒度の並列処理

一般に細粒度の並列処理は、機械命令レベルの並列処理を指し、算術式の計算を行なう際に交換、結合、分配法則を用いて命令間の並列性を引き出す計算木を作成するツリーハイトリダクションなどがこれにあたる。

3.2.2　中粒度の並列処理

中粒度の並列処理は、主に Do ループに対して行なわれる。並列化された Do ループにおいて、繰り返しは異なるプロセッサで並列に実行される。

3.3　粗粒度タスクの並列処理

粗粒度タスクの並列処理は、プログラム中に含まれる Do ループ、サブルーチン間などの比較的粒度の粗いタスク間の並列性を引き出すものである。

3.4　プログラムを並列処理できる形へ変換

コンパイラは、前節で導き出されたデータ依存性を元に、逐次プログラムから並列に実行可能な部分を抽出する。ここでは、Do ループの並列化について考えてみる。ループのイタレーション間にわたるデータ依存解析を行い依存関係を検出する。この結果イタレーション間のデータ依存の有無で、ループが Doall 可能か Doacross 可能かに分類する。図 1では、Do ループは I=1,I=2,···,I=N の各イタレーションは全く独立に計算可能で、各イタレーションを別々のプロセッサに割り当て実行することができる。(Doall 処理) 一方、J に関しては、J=k の s1 を処理するときに、J=k-1 の s3 の結果を待たねばならない。(Doacross 処理)

```
        do I=1,N
                do J=2,N
S1:                     C(I,J)=A(I,J-1)+B(I,J)
S2:                     D(I,J)=3.0*C(I,J)
S3:                     A(I,J)=C(I,J)
                end do
        end do
```

図 1: Doall 及び Doacross ループの例

3.4.1 並列実行可能部分のプロセッサへのスケジューリング

前節で明らかになった並列実行可能部分への、プロセッサの割り当て方を考える。コンパイル時に、割り当てを決定する静的なスケジューリング法では、実行時のオーバーヘッドが少ないものの、ループ中に条件分岐などがあり、各イタレーションの処理時間が大きく変動する場合には、プロセッサ間の負荷バランスが崩れることがある。一方、OS 等がプログラム実行時にプロセッサを割り当てる動的スケジューリングでは、複雑なスケジューリングアルゴリズムを使う事ができない、通信が必要であるなどの欠点がある。ここでは、動的スケジューリングの例をあげる。

- セルフスケジューリング
  共有メモリ上に処理を終えたイタレーション数を記憶するカウンタを用意し、各プロセッサが 1 イタレーションの処理を終えるとそのカウンタを参照し、次に自分が処理すべきイタレーションを 1 つ決め、カウンタをインクリメントする方式。

- チャンクスケジューリング
  セルフスケジューリングのように 1 イタレーション毎に処理すべきイタレーションを取って来るのではなく、処理を終えたプロセッサが、事前に設定された数の複数イタレーションを取ってくる方式。

- ガイデッドセルフスケジューリング
  各プロセッサが次に処理すべきイタレーションを決める際、その時点で残っているイタレーション数 R をプロセッサ数 m で割った数、すなわち [R/m] 個 ([x] は x 以上の最小の整数) のイタレーションを次に処理するイタレーションとして獲得する方式。

4 事例研究

本章では、並列コンパイラの事例として、富田研究室が開発しているメッセージ交換型並列計算機のため並列化コンパイラ TINPAR を取り上げる。

TINPAR では、指定された方法で各プロセッサに対して配列変数を分割し owner computes rule[1] に従って、拡張 Tiny language で書かれた逐次プログラムから、C 言語による並列実行可能なプログラムを生成する。

TINPAR は、次の過程を経てにより並列プログラムを出力する。

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) 並列化に関する部分の最適化

4.1 データの分散

TINPAR では、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することができる。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。

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。

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

図 2: 演算の分割とプロセッサへの割り当て

- 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

---

[1] 分割されたデータを保持するプロセッサが、そのデータをアクセスする文を実行するように、コードを生成する戦略。

```
for i = 0, n-1 do
    b(i) = a(i)      /* a(i) はブロック分割   b(i) は分割なし */
endfor
   ↓
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 3: 通信コードの挿入

め、配列の大きさの宣言自体も変更しないが、メモリ再割り当て部では、そのような配列の要素数を縮小する必要がある。

- 代入文
  左辺値の変数が、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 2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。

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。

そこでこのようなデータに対する参照には、所有者プロセッサによる送信のための send 文、参照者プロセッサによる受信のための receive 文、及び両者を判定するための if/endif 文を挿入する必要がある。(図 3)

4.4 並列化に関する部分の最適化

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ので、これらのう

ち不要なものを削除する。また、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も行なう。

4.4.1 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 2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。

4.4.2 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_CELLID() then
                a(i) = ...
        endif
        b(i) = ...
endfor
                        ↓
for i ∈ {i|OWNER(a(i)) == GET_CELL_ID(), 0 ≦ i ≦ n - 1} do
        a(i) = ...
        b(i) = ...
endfor
for i ∈ {i|OWNER(a(i)) != GET_CELL_ID(), 0 ≦ i ≦ n - 1} do
        b(i) = ...
endfor
```

図 4: if 文を削除するためのループ分割

4.4.3 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。

4.4.4 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。[5]

4.4.5 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。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ことにより、通信レイテンシを抑えることができる。

```
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 5: send/receive 文をともに含むループのループ分配

4.4.6 send/receive 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、通信レイテンシを完全に隠蔽するには十分ではありません。例えば、図 5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ます。しかしこのコードに図 5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ます。

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] 笠原博徳:並列処理技術, コロナ社,1991

[3] 大森健児:並列プログラミングの基礎, 丸善,1990

[4] 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

[5] 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