J. Plasma Fusion Res. Vol.82, No.10 (2006) 707-710

連載コラム High Performance Fortran で並列計算を始めよう

3．作ってみよう HPF プログラム(1)

岩下英俊，林 康晴[1)]，石黒静児[2)]
富士通　ソフトウェア事業本部，[1)]NEC　第一コンピュータソフトウェア事業部，[2)]核融合科学研究所

（原稿受付：2006年8月9日）

今回から3回にわたり，HPF を使った並列プログラムの書き方を解説します．ここで紹介する言語仕様は，HPF2.0 仕様[1]に実用的な拡張が施された，HPF/JA 1.0 版[2]です[1]．

3.1 HPF 言語の考え方

3.1.1 ハードウェアのモデル化

HPF がターゲットとする**分散メモリ型計算機**[2]は，Fig. 1 のようにモデル化できます．HPF では，CPU とメモリのペアをプロセッサと呼びます．アクセスするデータが自プロセッサ上にあるか，他プロセッサ上にあるかで，アクセスのスピードは全然違っていて，数百～数千倍の差があります．そのため，HPF で性能を出すためには，できるだけ自プロセッサ上だけで演算が行えるよう配置することが大変重要になります．

3.1.2 プログラムを並列化する

HPF では，動作確認ができている逐次プログラムからスタートして，並列実行する部分を増やしていく，というプログラミングスタイルを推奨します．この過程をプログラムの**並列化**と呼んでいます．

HPF では，元の逐次実行の意味を保ちながら並列化を進めることができます．これは HPF が，プロセッサ数を変えてもプログラムの意味を変えない，**割り算の並列化**（グローバルモデル）を基本とするからです．MPI プログラムでは，プロセッサ数に応じて解く問題のサイズが変わっていく，**掛け算の並列化**(SPMD モデル)になっています．この違いは Fig. 2 を見てください．

割り算の並列化が実現できるのは，コンパイラによって以下のような仮想化が提供されているおかげです．

- **グローバル空間**　…分散メモリが1つの巨大なメモリ空間であるかのように見えます．これにより，実体はプロセッサ間に分散されているということを気にしないで，従来の配列と同じようにプログラム中で参照することができます．利用者は，データの分散方法（block か cyclic か，どの次元を並列化するかなど）をコンパイラに指示します．
- **単一スレッド**　…個々のプロセッサの動作ではなく，全体の動作をプログラムします．例えば，分散されたデータの全域をアクセスするような DO ループを書けば，コンパイラはその計算をデータの分散に合うような部分範囲に分けて，各々のプロセッサで実行するコード（SPMD モデル）に変換します．

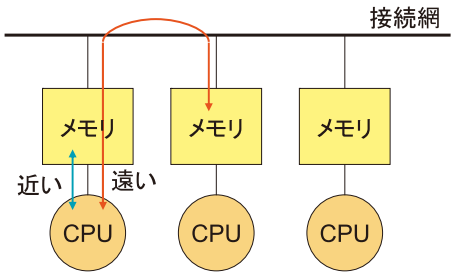

Fig. 1　HPF のターゲットハードウェア．

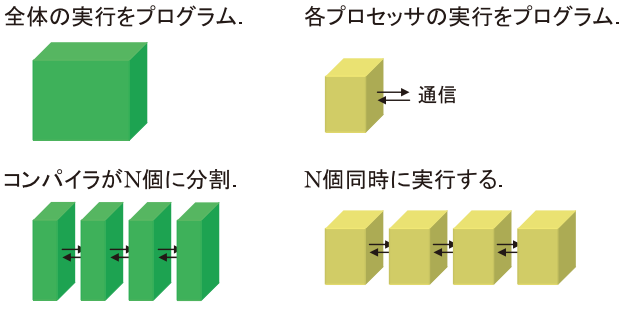

Fig. 2　割り算の並列化，掛け算の並列化．

1　NEC の HPF/ES，HPF/SX V2，HPF/EX と，前回紹介した fhpf は，HPF/JA 1.0 の主要機能をサポートしています．

2　PC クラスタ，ブレードサーバやスパコンが主なターゲットですが，共有メモリ（SMP）型計算機でも利用できます．SMP では，HPF のデータ分散を利用すれば，CPU 間のキャッシュ競合などの問題を大幅に緩和できることがあります．

Let Us Start Parallel Processing Using High Performance Fortran ! 3. Let Us Make HPF Program (1)
IWASHITA Hidetoshi, HAYASHI Yasuharu and ISHIGURO Seiji
corresponding author's e-mail: iwashita.hideto@jp.fujitsu.com

```
1       integer a(100)
2 !HPF$ DISTRIBUTE a(BLOCK)
3
4       do i=1,100
5         a(i) = i**2
6       end do
7
8       write(*,*) a
9       end
```

Fig. 3 プログラム例１.

3.2 最初の例：データ分散とループの並列化

それでは，HPF プログラムの例を見てみましょう．Fig. 3 は，小さいながらこれで完結した HPF プログラムの例です．

3.2.1 指示文の書式

HPF のプログラムは，Fortran プログラムに**指示文**を加えたもの，と考えてください[3]．2 行目がこのプログラムで唯一の指示文で，"!HPF$"で始まる形をしています．つまり，Fortran ではコメント行と見なされる形です．大文字と小文字は区別されません．

正しく書かれたHPFプログラムは，同じプログラムで指示文を無視した Fortran プログラム（**逐次解釈**）と実行結果が同じになります[4]．これは利用者にとって大変うれしい特徴です．逐次解釈による実行結果と比較することにより，正しく並列化が進んでいるか（正しく指示文を挿入できているか）を，いつでも確認することができるからです．

3.2.2 データの分散とその効果

配列 a は，**3.1.2**項で説明した仮想的なグローバル空間にあります．2 行目の **DISTRIBUTE 指示文**は，これを均等にブロック状に分散すること―ブロック分散―をコンパイラに指示しています．プロセッサ数を指定していないので，具体的に配列要素とプロセッサの対応が決まるのは，プログラムの実行開始時まで持ち越されます．

実行時のプロセッサ数が 2 であるとき，各々のプロセッサの実行イメージは，Fig. 4に示したようになります．ここでポイントは以下の 3 点です．

(1)配列 a がプロセッサに分散配置されています．これは DISTRIBUTE 指示文の直接の効果です．分散方法はここでは"BLOCK"が指定されていますが，他の分散方法は**3.2.3**項で紹介します．

(2) DO 文には何の指示も与えていませんが，変数 a の分散に応じて，DO ループの実行範囲が変換されています．これは，HPF コンパイラによる自動並列化の効果です．自動並列化とは，**ループの並列性**（**3.2.4**項）を自動で検出し，かつ，**ループ処理の分担**（**3.2.5**項）を自動で決定することです．

(3) WRITE 文は，分散された配列 a が逐次解釈と同じ順序で出力されるよう自動変換されます．実現方法はコンパイラの実装によりますが，ここではプロセッサ＃ 0 に配列 a 全体を集める通信を行ってからまとめて出力しています．このように，並列化に必要なプロセッサ間通信は，気づかないところで自動生成されることがあります．

3.2.3 分散方法のいろいろ

Fig. 3 では block 分散が使われています．この分散は，差分法など，近傍の要素を参照することが多いアプリケーションに向いています．"BLOCK"の代りに"CYCLIC"と指定すればcyclic分散となり，配列 a の要素は［1, 3, …, 99］と［2, 4, …, 100］に分けて配置されます．cyclic は DO ループ上下限が変化する場合や三角行列の計算など，block 分散では各プロセッサの計算負荷が不均等になる場合に向いています．他には，これらの中間である block-cyclic，block の拡張でブロック幅を不均等にする gen_block，そして，インデックスとプロセッサの対応を全く自由に指定できる indirect があります．

Fig. 3 ではプロセッサ数を可変としましたが，**PROCESSORS 指示文**を使ってプロセッサの名前と形状を宣言し，DISTRIBUTE 指示文で"ONTO"の後に記述すると，プロセッサ数がコンパイル時に固定されます．構文規則は最後の Fig. 8 にまとめました．プロセッサ数は可変にしておく方が便利ですが，プロセッサ数を固定すると性能が良くなる場合もあります．

3.2.4 ループの並列性

HPF コンパイラは，ループが並列に実行できるかどうか

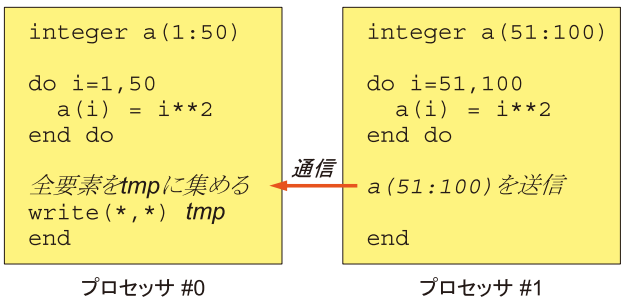

Fig. 4 例１の実行イメージ.

```
1       integer a(100)
2 !HPF$ DISTRIBUTE a(BLOCK)
3
4 !HPF$ INDEPENDENT
5       do i=1,100
6         a(i) = i**2
7       end do
8
9       write(*,*) a
10      end
```

Fig. 5 例１＋INDEPENDENT.

3 HPF2.0 仕様は，Fortran95仕様に指示文と HPF ライブラリを追加したものです．HPF/JA ではさらに，通信を効率化するための指示文を追加しています．

4 ただし後述する集計演算では，並列数を変えると計算誤差の大きさが変わることがあります．また，計時関数や乱数生成など，振る舞いが一定でない（pure でない）手続を使用するプログラムでは，結果は一定にならない場合があります．

を自動で判断します．利用者が判断してコンパイラに教えたいときは，DO 文の直前に **INDEPENDENT 指示文**を記述します（Fig. 5）．

どのようなループが並列化できるかは，仕様書[1]に厳密に定義されていますが，ラフには，「ループの繰り返しをそれぞれ独立して実行できるようなループは，並列化できる」と理解しておけば実用上は十分でしょう．逆に，i 番目までの繰り返しを先に実行しないと(i+1)番目が正しく実行できないようなループは，並列化できません．また，i 番目と j 番目を同時に実行するとおかしくなるようなループは，並列化できません．ループ外への飛び出しを含むループもまた，並列化はできません．

Fig. 6 に並列化できるループ(○)，できないループ(×)，コンパイラでは判断できないループ（？）の例を示します．

同図(c)は，i 番目の繰り返しの後で(i+1)番目の繰り返しを実行しないと結果が違いますから，並列化できません．もしこれに誤って並列化指示を与えると，例えばi番目をプロセッサ 0 が担当し，(i+1)番目をプロセッサ 1 が担当して，i 番目より先に(i+1)番目が実行されるかもしれないので，正しい結果が保障されません．(b)は(c)に似ていますが，並列化できます．変数 a への書き込みがないからです．2 つの繰り返しでの読み出しは，どちらが先でも同時でも大丈夫です．

(e)は繰り返しの実行順序を変えても正しい結果になるのですが，同時に実行すると，同じ変数 tmp への読み書きが衝突するので，このままでは並列実行できません．ですが並列化する手段はあります．利用者が書く場合，tmp の値が繰り返しを跨いでは引き継がれない（それぞれの繰り返しの中でだけ使用される）ことに気がつけば，INDEPENDENT 指示文の **NEW 節**を使うことができます（Fig. 8）．NEW 節で指示された変数（**new 変数**）は，繰り返しごとに固有の変数と見なされます．複数の繰り返しを同時に実行する場合には別々の領域が使用されるので，並列に実行できます．

(f)の変数 ifound は(e)の tmp とは違い，ループの外に値を持ち出したいので，new 変数にはできません．書き込みの衝突は禁止されるので，複数の繰り返しで ifound への代入が行われる場合には，並列化できません．

(g)は(f)と同じ理由で，複数の繰り返しで代入が起こると並列化できないのですが，こちらはよくある重要なパターン（最大値を得る）なので，特別に並列化する手段が用意されています．シリーズの次回で詳しく説明します．

(h)は，ix(i)の値がループの繰り返しの範囲で全部異なっている場合に限って並列化できます．(i)は，サブルーチン subx の中での変数 a や common 変数の読み書きが，サブルーチンの呼出しの間で衝突していない場合に限って並列化できます．これらはコンパイラでは通常判断できません．利用者が判断して並列化を指示します．

(j)のような多重ループでは，ループ毎に並列化できるかできないかを判断します．この例では，繰り返しを跨ぐ干渉により，j のループでは並列化できないとわかります．

最後に(k)は，ループの中から外への飛び出しがあるので，並列化できません．EXIT 文など他の分岐でも同様です．STOP 文を含むループも並列化できません（並列化するループから外へ飛び出さない分岐は許されます）．

3.2.5 ループ処理の分担

自動検出か指示によって並列化されるループについて，次はループ処理の分担，つまり，何番目の繰り返しをどのプロセッサで実行するかを決めなければなりません．ここで重要な点は，通信を最小にするような選択を考えることです．自動で行う場合には，コンパイラはまず，ループ中に出現する分散配列のアクセスパターンを解析します．この例の場合，a(i)の実体が配置されるプロセッサ(a(i)のホーム）が i の繰り返しを担当するように分担すれば，通信なしで実行が可能になるとわかります．同じことを利用者の指示で行う場合には，Fig. 7 のように**ON 指示文**を使って，「a(i)のホームがこの区間を実行せよ」と書くことができます．ON 指示文には他にも様々な使い方があります

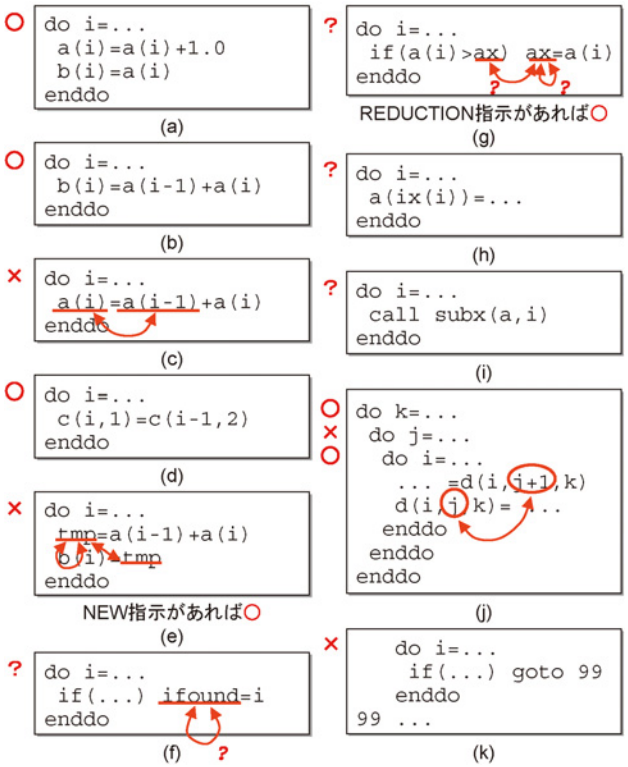

Fig. 6 並列化できるループ，できないループ．

```
1        integer a(100)
2  !HPF$ DISTRIBUTE a(BLOCK)
3
4  !HPF$ INDEPENDENT
5        do i=1,100
6  !HPF$ ON HOME(a(i)) BEGIN
7           a(i) = i**2
8  !HPF$ END ON
9        end do
10
11       write(*,*) a
12       end
```

Fig. 7 例 1＋INDEPENDENT+ON.

Fig. 8 今回解説した構文のまとめ.

が，このようにDOループ本体をすべて囲む形が最も効果的で重要です．同図は指示構文の例ですが，ネストしたDOループの外側を並列化する場合など，DOループ本体が1つの文または構文の場合には，単純指示文も使えます(Fig. 8)．

3.2.6 考察：自動 vs. 手動

さて，これらの指示文はなるべく書くべきでしょうか，なるべく自動に任せるべきでしょうか．「まず自動化に任せてみて，うまくいかなかった部分だけ人手で書く」という考え方もできますし，「人手で書く方が実行時の挙動がわかりやすいし，コンパイラの最適化のレベルに左右されなくていい」という考えもあります．筆者らは，これはどちらも正しいと考えます．自由に選択できることが重要です．何千，何万行の実プログラムの並列化では，性能的にクリティカルな部分だけは挙動を掌握するために人手で書いて，他の部分は自動に任せて生産性を上げる，という使い分けが必要になることがあります．

3.3 演習：姫野ベンチマークプログラム

ここまでの内容を，前回紹介した姫野ベンチHPF版の2つのサンプル[3]で確認してみましょう．指示文ALIGN，TEMPLATE，SHADOW，REFLECTと，REDUCTION節，LOCAL節については，次回以降解説します．

- V1，V2とも，逐次解釈で翻訳・実行ができます．HPFの1プロセッサ実行と比較すれば，HPFによる並列化に伴うオーバヘッドコストが評価できるでしょう．
- V1はプロセッサ数可変，V2は固定です．ソースを見比べてみてください．
- V1では，INDEPENDENT指示文とON指示文の組合せが1ヶ所使われ，他のループはすべて自動並列化に頼っています．V2ではループ並列性はすべて指示で与えています．見比べてください．これらの指示を加えたり外したりすると，並列化は変わるでしょうか．
- INDEPENDENT指示文には，NEW節が正しく記述されています．文法ではこれらは省略できないのですが，大抵のコンパイラでは書いていなくても自動的に判断して補います．確認してみてください．

3.4 まとめ

今回は，HPFプログラミングの基本的な考え方と，データ分散とループ並列化に関係する書き方を紹介しました．構文をFig. 8にまとめます．

当初の予定を変更し，「作ってみようHPFプログラム」をあと2回続けます．次回は，分散と並列化についてさらに深めます．次々回は，手続を越えた並列化を考えます．

参考文献

[1] High Performance Fortran 言語仕様書 Version 2.0
(http://www.hpfpc.org/jahpf/spec/hpf-j.html)
[2] HPF/JA 言語仕様書 Version 1.0
(http://www.hpfpc.org/jahpf/spec/jahpf-j.html)
[3] HPF 推進協議会（http://www.hpfpc.org/）