計算機アーキテクチャ　147－23
ハイパフォーマンス
コンピューティング　89－23
（2002．3．8）

# NPB を用いた HPF/JA 拡張の VPP 上での評価

浅岡 香枝[†]　平野 彰雄[†]　岡部寿男[††]　金澤正憲[†]

[†] 京都大学大型計算機センター , [††] 京都大学大学院情報学研究科

公開されている APR 社の HPF コンパイラ向けの NAS Parallel Benchmarks コードのうち EP,FT,BT,SP の 4 本について、HPF2.0 および HPF/JA 拡張仕様を用いて移植した。移植後のコードを Fujitsu VPP800 の HPF/VPP コンパイラを用いて評価した結果、すべてのコードにおいて十分な加速率を得ることができ、また、EP,BT については、MPI で実装される NPB 2.3 のコードよりも良い性能を得ることができた。

# Evaluation of the HPF/JA Extensions on Fujitsu VPP using the NAS Parallel Benchmarks

Kae ASAOKA[†], Akio HIRANO[†], Yasuo OKABE[††]
and Masanori KANAZAWA[†]

[†]Data Processing Center, Kyoto University,
[††]Graduate School of Informatics, Kyoto University

We have ported 4 codes(EP,FT,BT,SP) in APR's HPF implementation of NAS Parallel Benchmarks, so that it is conformable to HPF 2.0 and HPF/JA specification. The porting is done with the full usage of HPF 2.0 and HPF/JA new features, while the base Fortran codes remain almost unmodified. Then we have measured the performance of these benchmark codes on Fujitsu VPP800 by HPF/VPP compiler. We have achieved fairly good acceleration ratio for all codes, and better absolute performance than NPB 2.3 parallel implementation with MPI for EP and BT.

## 1　はじめに

HPF(High Performance Fortran)[4] は、Fortran を拡張したデータ並列言語である。HPF/JA[5] は、JAHPF(Japanese Association for High Performance Fortran)[10] がまとめた HPF 2.0 [4] に対する拡張仕様である。

我々は、HPF/JA の有効性を評価することを目的に、公開されている APR 社の NAS Parallel Benchmarks [1][7] コード (以下、APR のコードと呼ぶ) のうち EP,FT,BT,SP の 4 本について、HPF2.0 および HPF/JA 拡張仕様に基づいて移植した。また、京都大学大型計算機センターで運用している並列ベクトル計算機 Fujitsu VPP800 上で HPF/JA をサポートする HPF/VPP コンパイラ [2] を用いて性能を測定し、評価した。

本稿では、まず、HPF および HPF/JA について概説する。次に、コードを移植するための基本方針、また、ベンチマークコード EP, FT,BT,SP の移植において適用した手法を述べる。最後に、移植後のコードの性能および HPF/JA の有効性を評価する。

## 2　HPF と HPF/JA

HPF の言語仕様は、1992 年、HPFF(High Performance Fortran Forum)[8] によって検討され、1993 年に HPF 1.0 が定められた。そして 1997 年には HPF 2.0 が定められ、この時、基本仕様と公認拡張仕様という形に言語仕様が整理された。

国内でも、1997 年にスーパコンピュータベンダ 3 社のコンパイラ開発者および学術研究者で構成され

るJAHPFが発足し、HPFの普及およびコンパイラの実現などを目的に仕様の検討を行い、1999年にHPF/JA 1.0拡張仕様が定められた。これはHPF 2.0に対する拡張仕様であり、並列記述性の向上および適用範囲の拡大とユーザによるきめ細かい並列化および最適化の記述を可能にすることでコンパイラの処理能力不足を補うものである。

HPF2.0とHPF/JA仕様の包含関係を図1に示す。HPF/JAでは、HPF 2.0の基本仕様に加え公認拡張仕様をカバーし、さらに独自の拡張を加えている。ただし、公認拡張仕様のうちいくつかは明示的に除外することで、コンパイラの開発を容易にしている。

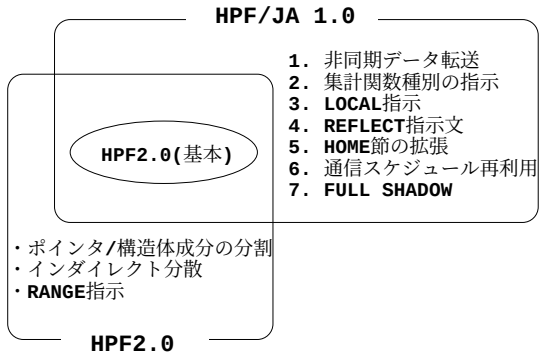


図 1. 仕様の包含関係

なお、HPF2.0の指示文が!hpf$で始まるのに対し、HPF/JAの指示文は!hpfjで始まり、HPF2.0の指示文と区別できるようになっている。

## 3 ベンチマークコードの移植

### 3.1 基本方針

今回の移植は、APRのコードを元にした。これらのコードはFortran77で書かれており、HPFの指示行はほとんど用いられておらず、一部のコードにはAPRのHPFコンパイラ用拡張指示文が含まれている。我々は、これらに対して、コードのFortran部分をできるだけ変えずに、かつ、HPFおよびHPF/JA指示文を可能な限り詳細に挿入することで、VPP800において高性能を得ることを目標とした。

### 3.2 EP

#### 3.2.1 EPのアルゴリズム

EP (Embarrassingly Parallel)は、モンテカルロ法などで用いられる乱数生成のためのベンチマークである。基本的なプログラム構造を図2に示す。

```
      dimension x(2*nk)
              :
      do 150 k=1,nn
              :
         t3 = randlc(t1,t2)
                   ! 乱数の種(t1)を求める
         call vranlc(2*nk,t1,a,x)
                   ! 乱数列(x)を求める
              :
         do 140 i=1,nk
              :
           .....  ! 乱数のペアを求める
           l=...  ! 属する領域を求める
           if (l .eq. 0) then
             q0=q0+1.0d0
           else if (l .eq. 1)then
             q1=q1+1.0d0
                :
             q9=q9+1.0d0
           end if   ! カウントアップ
 140     continue
 150   continue
```

図 2. EPのプログラム構造

図2に示すように、doループ150が基本ループであり、このループ内で、乱数の種を求める関数**randlc**と乱数列を求める**vranlc**サブルーチンを呼出し、得られた乱数列から乱数のペアを求め、それが10個の領域のどれに属するかを集計するという処理を**nn**回繰り返す構造をしている。

したがって、集計処理以外は、独立に求めることが可能であり、doループ150で並列化できる。

#### 3.2.2 テンプレート配列の導入による並列化

HPFでは、図2のdoループ150の前に、並列化を明示するindependent指示文を挿入しても並列化できない。これは、HPFが分散配列のデータマッピングを基準に計算処理を分割する仕様に基づいているためである。

したがって、図3に示すように、doループ150の繰り返し数である**nn**と同じ大きさのテンプレート配列**dummy**を宣言してブロック分割し、これをdoループ150を並列化するための分割基準とした。また、領域の集計は、プロセッサ間にまたがる総和計算となるので、reduction節に集計のための変数**q0**, **q1**,...,**q9**を指定している。

#### 3.2.3 手続きのローカル化とベクトル化

HPFでは、Fortran言語仕様に外部手続き種別という概念が追加されており、extrinsic節で宣言される。外部手続き種別には、標準のHPF_GLOBAL、HPF_LOCALおよびFORTRAN_LOCALといったものがある。

```
!hpf$ processors p(4)
!hpf$ template dummy(nn)
                  ! テンプレート配列の宣言
!hpf$ distribute dummy(block) onto p
                  ! ブロック分割の指定
            :
!hpf$ independent,
!hpf$& new(k,i,t1,t2,t3,x,l,...),
!hpf$& reduction(q0,q1,...,q9)
      do 150 k=1,nn       ! 並列化ループ
!hpf$ on home(dummy(k)) begin
                  ! ループの分割基準の指定
              :
          t3 = randlc(t1,t2)
                  ! 乱数の種 (t1) を求める
          call vranlc(2*nk,t1,a,x)
                  ! 乱数列 (x) を求める
              :
          do 140 i=1,nk
              :
            .....  ! 乱数のペアを求める
            l=...  ! 属する領域を求める
            if (l .eq. 0) then
             q0=q0+1.0d0
            else if (l .eq. 1)then
             q1=q1+1.0d0
              :
             q9=q9+1.0d0
            end if  ! カウントアップ
 140      continue
!hpf$ end on
 150  continue
```

図 3. EP の並列化プログラム

HPF_LOCAL は、引数に分散配列が指定されると分割後の大きさで渡る。また、FORTRAN_LOCAL は、 Fortran コンパイラで生成された外部手続きの呼出しであり、各プロセッサで独立に呼出される。

図 3の **randlc** 関数および **vranlc** サブルーチンは、完全に各プロセッサで独立に呼出せる手続きであり、FORTRAN_LOCAL 手続きに変更した。

また、 APR のコードの **vranlc** サブルーチンは、ベクトル化できないものであったので、 NPB2.3-serial で提供されているベクトル向けのルーチンに置換えた。この置換えにより、約 2.68 倍の性能向上が得られた (VPP800 の 4PE での実行時)。

## 3.3 BT

### 3.3.1 アルゴリズム

BT (Block Tridiagonal simulated CFD application) は、非優位対角な 5 × 5 のブロックサイズをもつ 3 重対角方程式を解くベンチマークであり、基本部分は 3 次元の ADI(Alternative Direction Implicit) 法である。

### 3.3.2 隣接プロセッサ上データのアクセス処理

図 4(a) は、 BT に現れる演算について配列を 2 次元に簡略化したものを示している。

図 4(a) では、配列 **x**,**u** をそれぞれ 2 次元目でブロック分割し、 do ループ 100 で並列化している。分散配列 **x** は do ループ 100 において、常にプロセッサ内に閉じたアクセスとなる。しかし、分散配列 **u** は、 2 次元目の添字部分に **j-1**,**j-2**,**j+1**,**j+2** というインデックス計算が現われているため、分割境界の演算部分においては隣接プロセッサにマッピングされているデータを必要とする。

このようなデータアクセス構造を持つループのために、 HPF には shadow 指示文がある。 shadow 指示文は、隣接プロセッサ上のデータのコピーを保持するための領域 (shadow 域) を確保することを指示する。

図 4(a) に対して shadow 指示文を適用したものを図 4(b) に示す。図 4(b) では、 shadow 指示文で、分散配列 **u** の 2 次元目に対して、上端と下端に、それぞれ 2 要素分の shadow 域を確保した。

また、 reflect 指示文は、 HPF/JA の指示文であり、 shadow 域へのコピーを明示する。

さらに、図 4(b) の do ループ 100 の中では、分散配列 **u** について shadow 域を含めた各プロセッサに閉じたアクセスとするために、 HPF/JA の local 指示文を指定している。このような指示で、 do ループ 200 がベクトル化される。

図 4の do ループ 100 の実行時間 (**n**=64) は、オリジナル (a) では 24.8$m$sec、 HPF/JA 適用後 (b) では 0.113$m$sec であった (VPP800 の 4PE 実行時)。

### 3.3.3 分割配列データの転置処理

図 5(a) は、 BT に現れる 4 次元配列間の転置処理を示している。

3 次元 ADI 法は、 3 次元の各方向について順次走査する。つまり、ある 1 方向には依存関係があり他の 2 方向については独立に処理できるので並列化可能であるが、配列の分割次元方向の走査は、そのままでは並列化できない。

したがって、これに対応するために、同じ分割の作業用配列を用意し、そこに元の配列からデータを転置して格納し、元の配列の代りにこの作業用配列をアクセスすることで並列化している。

BT では、この転置処理のために図 5(a) に示す 4 重の do ループが現われる。

```
    !hpf$ processors p(4)
    !hpf$ distribute (*,block)
    !hpf$&       onto p :: x,u
                  :
    !hpf$ independent,new(i,j)
p         do 100 j = 3, n-2
p   !hpf$ on home(x(:,j)) begin
p          do 200 i = 1,n
p 5         x(i,j)=x(i,j)+u(i,j-2)
p         &         +u(i,j-1)+u(i,j)
p         &         +u(i,j+1)+u(i,j+2)
p    200   continue
p   !hpf$ end on
p    100  continue
```

(a) オリジナル

```
    !hpf$ processors p(4)
    !hpf$ distribute (*,block)
    !hpf$&       onto p :: x,u
    !hpf$ shadow u(0,2:2)
                  :
    !hpfj reflect u
    !hpf$ independent,new(i,j)
  p         do 100 j = 3, n-2
  p   !hpf$ on home(x(:,j)) begin
  p   !hpfj local begin
v p          do 200 i = 1,n
v p          x(i,j)=x(i,j)+u(i,j-2)
v p        &         +u(i,j-1)+u(i,j)
v p        &         +u(i,j+1)+u(i,j+2)
v p    200   continue
  p   !hpfj end local
  p   !hpf$ end on
  p    100  continue
```

(b) shadow,reflect,local 指示文の適用

図 4. 隣接プロセッサ上のデータアクセス

```
    !hpf$ processors p(4)
    !hpf$ distribute u(*,*,*,block) onto p
    !hpf$ align with u :: ut,r,rt
                  :
    !hpf$ independent,new(k,j,i,m)
p         do 400 k =1,nz
p   !hpf$ on home(u(:,:,:,k)) begin
p          do 300 j = 1,ny
p           do 200 i = 1,nx
p            do 100 m = 1,5
p 1            ut(m,i,k,j) = u(m,i,j,k)
p 1            rt(m,i,k,j) = r(m,i,j,k)
p    100     continue
p    200    continue
p    300   continue
p   !hpf$ end on
p    400  continue
```

(a) do ループによる転置処理

```
  !hpf$ processors p(4)
  !hpf$ distribute u(*,*,*,block) onto p
  !hpf$ align with u :: ut,r,rt
  !hpfj asyncid id
                :
  !hpfj asynchronous(id), nobuffer begin
1       forall(m=1:5,i=1:nx,j=1:ny,k=1:nz)
       &      ut(m,i,k,j) = u(m,i,j,k)
1       forall(m=1:5,i=1:nx,j=1:ny,k=1:nz)
       &      rt(m,i,k,j) = r(m,i,j,k)
  !hpfj end asynchronous
  !hpfj asyncwait(id)
```

(b) 非同期転送構文による転置処理

図 5. 4 次元配列の転置処理

HPF/JA には、配列間のデータ転送のために非同期転送構文が用意されている。図 5(a) の処理を非同期転送構文に置換えたものを図 5(b) に示す。

図 5の転置処理の実行時間 (**nx**=**ny**=**nz**=64) は、do ループで処理する (a) では $6,533m$sec、HPF/JA の非同期転送構文 (b) では $0.345m$sec であった (VPP 800 の 4PE 実行時)。

### 3.4 SP

SP (Scalar Pentadiagonal simulated CFD application) は、非優位対角なスカラ 5 重対角方程式を解くベンチマークで、通信に対する演算量の割合が異なる以外は BT と同じであり、3.3節で述べた手法を SP にも適用した。

### 3.5 FT

FT (3-D FFT PDE) は、3 次元高速フーリエ変換のベンチマークである。

FT では、3 次元配列の各次元方向について、1 次元 FFT を順次行っており、1 次元 FFT はその方向については依存関係があるが、他の 2 方向については独立して実行でき、並列化可能である。しかし、分割次元方向での 1 次元 FFT はそのままでは並列化できない。

これに対して、APR のコードでは、APR の HPF コンパイラ独自の指示文を用いた再分割により分割次元を変更することで並列化している。

HPF には実行時に分割を変更する `redistribute` 指示文が用意されているが、HPF/VPP コンパイラ

表 1. 実行時間と加速率 (CLASS=C)

| Benchmark | 1PE* | 1PE | 2PE | 4PE | 8PE | 16PE | 32PE |
|---|---|---|---|---|---|---|---|
| EP | 443.96 (1.00) | 473.04 (0.93) | 237.21 (1.87) | 118.76 (3.73) | 59.43 (7.47) | 29.69 (14.95) | 14.86 (29.87) |
| FT | — | — | 3684.18 (2.00) | 1843.09 (3.99) | 921.28 (7.99) | 460.64 (15.99) | 230.52 (31.96) |
| BT | 17337.00 (1.00) | 17608.00 (0.98) | 8807.90 (1.96) | 4515.00 (3.83) | 2319.40 (7.47) | 1222.00 (14.18) | 681.48 (25.44) |
| SP | 9155.30 (1.00) | 10260.00 (0.89) | 5160.40 (1.77) | 2647.30 (3.45) | 1365.90 (6.70) | 721.59 (12.68) | 411.30 (22.25) |

(* Fortran Compile) (Seconds)

では、再分割する配列はFULL SHADOWを持たなくてはならないという制限があるために、3.3.3節で述べたようにBT,SPと同様な手法で作業用配列を用意し、転置処理を行うように修正した。

また、1次元FFTの処理を行うサブルーチン **fft** は、完全に各プロセッサで独立して呼出せる手続きであるので、`FORTRAN_LOCAL` 手続きとした。

## 4 評価

### 4.1 実行性能

移植後のコードの実行性能を評価するために、VPP 800を用い、問題サイズCLASS=Cで測定した結果を表1に示す。

表1の1PE* は移植後のコードをFortranコンパイルした実行結果であり、また、1PEから32PEは、HPF/VPPでコンパイルしそれぞれのPE(Processors Element)数で実行させた結果である。実行時間を秒単位で示しており、括弧内は、Fortranコンパイルした実行時間を1.00とした場合の加速率を示している。ただしFTについては、メモリ不足により1PEでは測定不可能であったため、2PEの実行時間を2.00として加速率を算出してある。また、図6はこの加速率をグラフに表したものである。

表1および図6からも明らかなように、移植後のコードすべてにおいてPE数に応じた並列化効果が得られている。

このことから、我々が今回取り上げたAPRのコードのように並列化に十分配慮されたプログラムであれば、HPF/JA拡張仕様を用いてデータアクセスおよびデータ転送をきめ細かく指定することで、十分な性能が得られることがわかる。

なお、使用したコンパイラはFujitsu UXP/V Fortran V20L20 L01091およびFujitsu UXP/V HPF V20L20 L01041であり、測定には、マイクロ秒単位で経過時間を返すサービスルーチン **gettod** および **fjhpf_gettod** を用いた。

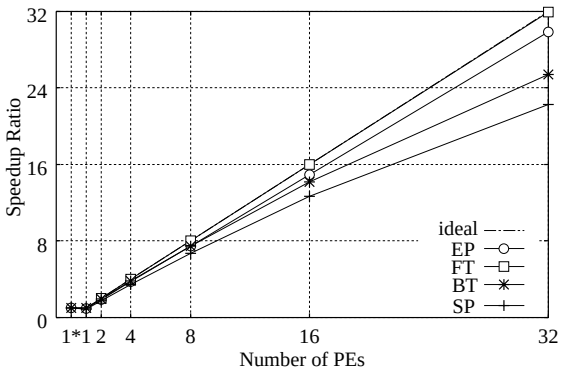


図 6. 加速率

### 4.2 HPF/JA 拡張仕様の有効性

HPF/JA拡張仕様の有効性の評価を目的に、移植後のコードおよび移植後のコードからHPF/JAの指示行を削除したものについて、PE数を4、問題サイズCLASS=Aとして測定した結果を表2に示す。ただし、EPには、HPF/JA指示文が無いので外している。

表2のAdaptorとは、我々がVPP800に移植したHPF2.0準拠のパブリックドメインのHPFコンパイラAdaptor [9]でコンパイルし、実行した結果を示す。なお、HPFコンパイラAdaptorは、MPIコードへのトランスレータとして実現されており、使用したMPIライブラリはFujitsu UXP/V MPI V20L20 L00011である。そして、測定には、Adaptorが提供するルーチン **system_clock** を用いた。

表 2. HPF/JA 拡張の効果 (CLASS=A,4PE)

| Benchmark | HPF/VPP | HPF/VPP (Without JA) | Adaptor |
|---|---|---|---|
| EP | ⟹ | 7.32 | 6.95 |
| FT | 42.30 | 605.78 | 42.52 |
| BT | 269.21 | (*1) | 599.36 |
| SP | 174.35 | (*1) | 535.18 |

(*1) More than 1hour (Seconds)

表 2の結果から、 HPF/VPP においては HPF/JA 拡張仕様が極めて有効に働き事実上必須であることがわかる。

また、 Adaptor の結果については、 EP,FT では良い性能が得られているが、 SP,BT では多くの時間がかかっており、 HPF/JA 拡張仕様をきちんとサポートすることで性能改善の余地があると考えられる。

### 4.3 NPB 2.3 との比較

移植後のコードの絶対性能を知るために、 MPI で実装された NPB2.3 について、 PE 数 4、問題サイズ CLASS=C で測定した結果を表 3に示す。

表 3では基本となるコードが異なるため単純に比較はできないが、我々の移植後のコードは、 EP および BT において NPB2.3 よりも高速な実行結果が得られている。

表 3. NPB2.3 との比較 (CLASS=C,4PE)

| Benchmark | HPF/VPP | NPB 2.3 |
|---|---|---|
| EP | 118.76 | 158.32 |
| FT | 1843.09 | 361.56 |
| BT | 4515.00 | 5100.24 |
| SP | 2647.30 | 240.93 |

(Seconds)

## 5 まとめ

NAS Parallel benchmark の HPF による実装として公開されている APR のコード 5 本のうち、 4 本について HPF 2.0 および HPF/JA に準拠したコードに書き直した。さらにそのコードを VPP800 上で HPF/VPP を用いて性能評価した。

その結果、良好な加速率と EP および BT については NPB2.3 MPI 版よりも高速な絶対性能を得た。

NPB を HPF により記述して性能評価した研究の例としては、 [6] が知られている。これに対し本研究では、既存のコードをできるだけ変えずに HPF2.0 および HPF/JA に対応させることを主眼に移植した。今回対象としたコードは基本的に並列化されていたので、ここで提案した簡単な方針で移植したものでも NPB2.3 MPI 版と同等の性能を得ることができた。しかし、 3.3.3節で示したように、コードのわずかな違いで性能に 10000 倍以上もの差がでること、また、 HPP/VPP の実装上の制約により、移植の方針に反して Fortran コードを書き換える必要があったことなど、言語処理系には、まだ多くの改善の余地があると考える。

今回は、 APR が公開していた NPB5 本のうち 4 本について評価を行ったが、他の NPB については課題である。また HPFbench[3] などの他の HPF ベンチマークコードについても HPF/JA 拡張仕様がどのように有効かを評価することが必要と思われる。

## 参考文献


[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga: THE NAS PARALLEL BENCHMARKS, RNR Technical Report RNR-94-007, NASA Ames Research Center (1994).

[2] 富士通株式会社: UXP/V HPF 使用手引書 V20L20 L01041 用 J2U5-0450-01 (2001).

[3] Y. Charlie Hu, Guoha Jin, S. Lennart Johnson, Dimitris Kehagias, Nadia Shalaby: HPF Bench: A High Performance Fortran Benchmark Suite, To appear in ACM Transactions on Mathematical Software (2000).

[4] High Performance Fortran Forum: High Performance Fortran Language Specification, version 2.0, http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/ (1997).

[5] Japan Association for High Performance Fortran: HPF/JA Language Specification, version 1.0, http://www.hpfpc.org/jahpf/ (1999).

[6] 太田 寛, 西谷 康人, 小林 篤, 布広 永示:HPF 処理系 Parallel FORTRAN による NAS Parallel ベンチマークの並列化, 情報処理, Vol.38, No.9, pp.1830-1839 (1997).

[7] ftp://ftp.infomall.org/tenants/apri/Benchmarks/

[8] http://www.crpc.rice.edu/HPFF/

[9] http://www.gmd.de/SCAI/lab/adaptor/

[10] http://www.hpfpc.org/jahpf/