HPF トランスレータ fhpf における 分散種別を一般化したコード生成手法

岩下英俊† 青木正樹†

High Performance Fortran（HPF）仕様には様々なデータ分散種別があり用途に合わせて利用者が選択できるが，複雑な分散についてはコンパイラが十分に性能を出しきれていない．本稿ではループインデックスのローカル化のアルゴリズムを示す．この手法は SPMD コード生成における分散種別による処理の違いを極小化し，ループ構造や配列の添え字式を分散種別によらず簡潔にする．実測を通して効果を示す．

A Code Generation Technique Common to Distribution Kinds on the HPF Translator fhpf

Hidetoshi Iwashita† and Masaki Aoki†

High Performance Fortran (HPF) language specification has various distribution kinds, which can be chosen according to the purpose of use. However the compilers are not good enough at such complicated distributions as users rarely use. This article shows the algorithm of the localization of loop index. It minimizes most part of SPMD code generation in common to different distribution kinds and simplifies the loop constructions and the array subscripts regardless of the distribution kinds. Measurement result is also in shown.

1. はじめに

HPC アプリケーションの開発の生産性を上げるためには，MPI ではない高級言語の標準化と普及が必要であると我々は考えている．HPF 言語[3] に実用的な拡張を加えた HPF/JA 言語仕様[4] はその 1 つであり，地球シミュレータセンターをはじめとするハイエンドの計算機で使われている．我々は HPF 推進協議会のメンバとともに，HPF の普及推進のための活動を続けており，これが今後の並列言語の国際的な標準化・規格化につながると考えている．

HPF のターゲットと利用者層を広げるため，我々はプラットフォーム無依存の HPF コンパイラ fhpf を開発してきた．これは MPI 呼び出しを含む Fortran プログラムを出力とするトランスレータであり，生成コードは任意の標準的な Fortran と MPI 環境を持つ PC クラスタやブレードサーバ上で利用できる．その目指すものは，扱いやすさ，実行時オーバヘッド，コンパイラ自体の開発生産性など，すべてにおいて軽いことである[6]．

本稿では，fhpf コンパイラの SPMD コード生成部の実現論理と実装方法を紹介する．これは，**インデックスのローカル化**という考え方に基づいたループパラメータ，配列の宣言形状，配列アクセスの添え字式などの変換の手法である．この手法の狙いは，並列実行ループ中に出現する主要な分散配列の添え字式の形を，分散種別（block，cyclic，gen_block，indirect など）によらず簡潔な式（通常はその DO 変数自身）に変換することである．

以下，2 章で本研究の背景と目的を示し，3 章で我々が提案するアルゴリズムを紹介する．fhpf トランスレータへの実装と評価をそれぞれ 4 章と 5 章で述べる．6 章は関連研究，7 章はまとめである．

2. 背景

2.1 分散の種別

HPF で定義される分散の種別を**図 1** に示す．同図で p はプロセッサの番号であり，k 番目の配列要素が分散配置される先を図示している．

(a) block 分散 分散次元についてもデータの連続性を残しているので，差分法など隣接する要素や近

† 富士通株式会社ソフトウェア事業本部
Software Unit, Fujitsu Limited

表 1 分散パラメータ
Table 1 Distribution parameters.

N_{Td}	テンプレート T の次元 d の寸法．いい換えると，分割されるインデックスの長さ．
P_{Td}	テンプレート T の次元 d に対応するプロセッサ群の次元の寸法．いい換えると，インデックスの分割数．
w_{Td}	block 分散または block-cyclic 分散のとき，その block 幅．
W_{Td}	不均等 block 分散のとき，そのブロック幅のリストであり，長さ P_{Td} のベクトル．$W_{Td}(k)$ $(0 \leq k < P_{Td})$ はプロセッサ k に割り当てられるインデックスの幅となる．
M_{Td}	不規則分散のとき，マッピング配列であり，長さ N_{Td} のベクトル．$M_{Td}(k)$ の値はテンプレートの各要素に対応する正規化されたプロセッサ要素番号であり，$0 \leq M_{Td}(k) < P_{Td}$ となる．
L_{Td}	不均等 block 分散のとき W_{Td} から誘導される長さ $(P_{Td}-1)$ のベクトル：$L_{Td}(p) = \sum_{q=0}^{p-1} W_{Td}(q)$ $(p = 0 \cdots P_{Td}-1)$．いい換えると，不均等 block 分散のグローバルインデックスでの下限値を表す配列．

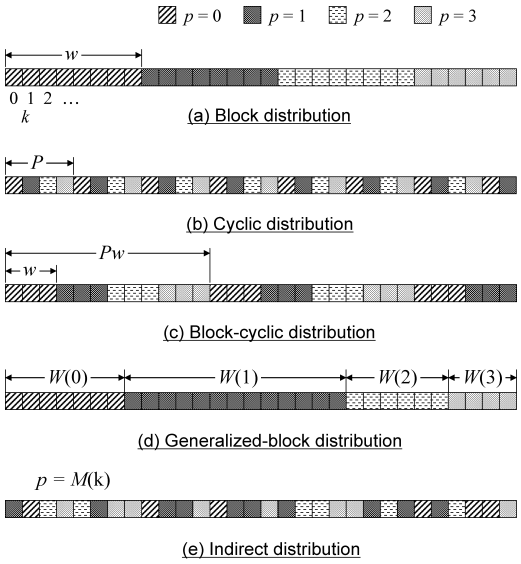


図 1 分散種別
Fig. 1 Distribution kind.

傍の要素との相関が大きいアルゴリズムの記述に適している．ブロック幅 w の指定がない場合には，$w = \lceil N/P \rceil$ と定義される（N はデータの大きさ，P はプロセッサ数）．

(b) cyclic 分散 負荷バランスや計算範囲が実行ループごとに違ったり実行時まで不明だったりする場合に，負荷分散をほぼ均等にできる．マッピングは周期 P で反復する．

(c) block-cyclic 分散 block と cyclic の中間的な方法．block では負荷バランスが不均等だが，cyclic では近傍データの通信が多くなりすぎる場合などに適している．マッピングは大周期 Pw と小周期 w を持つ．

(d) 不均等 block 分散 計算負荷が計算領域に対して特定の傾向で偏っている場合（三角行列の計算など）や，並列計算機の能力が不揃いの場合（ヘテロ環境など）に適している．ブロック幅はプロセッサ数の長さのベクトル W となる．

(e) 不規則分散 データとプロセッサに不規則な対応付けをしたい場合（空間中を浮遊する粒子の計算など）に適している．長さ N のマッピング配列 M に，インデックス k に対応するプロセッサ番号を列挙する．

多次元の分散を考えるときには，N は配列データのある分散している次元の長さ，P は仮想プロセッサ配列の対応する次元の長さ，と読み替えればよい．HPF では分散次元を一次元ごとに独立に扱って重ね合わせることができる．

2.2 割 付 け

HPF 文法では，インデックスのプロセッサへの対応付けは前節で述べたように定義されているが，プロセッサ内でのインデックスの配置は規定されていない．そのため，分散配列の各プロセッサ上での割付け配置は実装に依存する．たとえば，cyclic 分散と block-cyclic 分散と不規則分散では 1 つのプロセッサに対応するインデックスが不連続であるが，実装では圧縮して連続アドレスに置いてもよいし，不連続のまま割り付けてもよい．後者の場合，インデックス変換の煩雑な計算が回避できるが，並列度が上がるほどメモリ使用の無駄が大きくなるという欠点が大きい．

我々の実装では，分散種別によらず連続的な割付けを選択する．同時に，インデックスの昇順を守るルールとすることで，以下の利点が生まれる．

- 下限と上限のペアで表現されるインデックスの区間は，分散後のイメージでも必ず下限と上限のペアだけで表現できる．

具体的なインデックス変換方法は 3.2 節で紹介する．

2.3 分散パラメータの定義

ある次元の分散を同定するパラメータには，分散種別以外に，**表 1** に示す分散パラメータ

$$D_{Td} = \{N_{Td}, P_{Td}, w_{Td}, W_{Td}, M_{Td}, L_{Td}\}$$

が必要である．ただし L_{Td} は独立なパラメータではないが，実装の便宜上ここに加える．テンプレート T は HPF 言語仕様[3),4)] で定義されるもので，配列要素

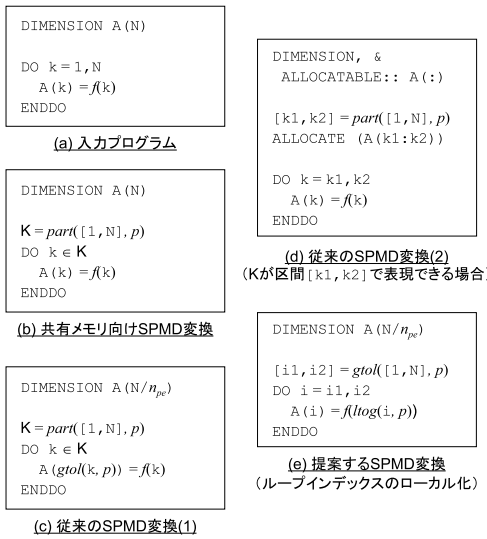


図 2 SPMD 変換の方法
Fig. 2 SPMD conversions.

またはループ反復とプロセッサとのマッピングを仲介する，実体のない配列である．以降，下付き添え字としての T と d は文脈から自明であるとき省略する．

2.4 SPMD 変換

HPF に限らず OpenMP [7] や XPFortran（旧 VPP Fortran [10]）など，グローバル名前空間をサポートする並列言語を取り扱う並列化コンパイラは，利用者の記述したグローバルイメージのプログラムを SPMD（Single Program/Multiple Data）イメージのコードに変換することが主な仕事である．これを**図 2** に示す例で説明する．分散種別は一般化されている．

同図 (a) は並列化可能なループを含むプログラムである．$f(\mathtt{k})$ は `k` を含む任意の式とする．ここでは初期値 `1` と終値 `N` の二つ組パラメータを持つ DO ループを例示している．一般に，初期値 k_1 と終値 k_2 の二つ組パラメータが表現する DO ループは，直感的には DO 変数 `k` の値を

$$k_1,\ k_1+1,\ k_1+2,\ \cdots,\ k_2$$

と変化させながら反復して実行される．これに増分 $k_3 \neq 0$ を加えた三つ組パラメータを持つ DO 文 `DO k=`k_1, k_2, k_3 が表現する DO ループは，直感的には DO 変数 `k` の値を

$$k_1,\ k_1+k_3,\ k_1+2k_3,\ \cdots,\ k_1+\left\lfloor\frac{k_2-k_1}{k_3}\right\rfloor k_3$$

と変化させながら反復して実行される．二つ組の DO ループは三つ組の DO ループで $k_3=1$ とした場合と等価である．

さて，(a) に対する SPMD 変換を考える．変数 `A` を共有メモリ上に置く場合，たとえば OpenMP で適切な並列化指示を加えて翻訳すると，(b) に示すイメージの中間コードが生成される．同図で p は自分のプロセッサ番号であり，プログラムの実行開始時に決定する．*part* は，インデックスの集合のうちプロセッサに対応付けられる部分集合を得る関数であり，実際にはコード中にインライン展開されるか，ライブラリ呼び出しで実現される．この変換の結果は，block-cyclic や不規則分散の場合，1 つの DO ループでは表現できないインデックスの集まりとなることがある．共有メモリでの実装では `A` はプロセッサ間でそのまま共有されるので，ループパラメータ以外の実行文や配列宣言を変更する必要はない．

一方，分散メモリでは，ループ計算とともに `A` の実体を分割配置する必要がある．また SPMD とは同じ 1 つのプログラムですべてのプロセッサに対応することであるから，静的な配列宣言では上下限値に可変の値を持ち込めない以上，プロセッサごとに配列の宣言形状を違えることはできない．そこで従来の実装では，(c) のように宣言形状は同一として添え字式でグローバルからローカルへのインデックス変換 *gtol* を行うか，(d) のように割付けを動的にすることによって上下限値をプロセッサごとに変えている．後者は Fortran90 のような高級言語を生成コードに利用する場合に可能な表現であるが，結果的には後続の Fortran コンパイラによって (c) と同等の実行コードに変換されることになる．我々は従来 (c) の実装を行っていた．

我々は，(e) のようにループインデックスもまたローカルに変換する方法を提案する．この方法では，ループと変数の分散が整合するとき，その次元の添え字式は分散種別によらず DO 変数そのものになることが保障される．副作用として，そのような添え字式以外の場所で出現する DO 変数については，(e) のようにローカルからグローバルへの変換 *ltog* を受ける．すなわち，この方法は従来法 (c) と比べて，ループと変数の分散が整合した場合の性能を重視した変換であるといえる．整合の意味は 3.3 節で詳述する．洗練された HPC アプリケーションでは，並列 DO ループの DO 変数はほとんどの場合，分散配列の添え字式として（あるいはそのような添え字式を計算する式の中で）用いられるので，*ltog* 変換が性能上問題となるほど生じることは少ないと考えている．

2.5 インデックスのローカル化の利点

従来法と比較し，提案する方法には以下のような利点がある．

(1) 性能上重要な変数のアクセスについて，SPMD

表 2 G-L 変換式と分散後のサイズ
Table 2 Global-local transformation and distribution size.

分散種別	$i = \mathrm{gtol}_D(k[,p])$	$k = \mathrm{ltog}_D(i,p)$	分散後のサイズ
block	$k \bmod w$	$i + pw$	w
cyclic	$\lfloor \frac{k}{P} \rfloor$	$iP + p$	$\lceil \frac{N}{P} \rceil$
block-cyclic	$w\lfloor \frac{k}{Pw} \rfloor + (k \bmod w)$	$Pw\lfloor \frac{i}{w} \rfloor + pw + (i \bmod w)$	$w\lceil \frac{N}{Pw} \rceil$
不均等 block	$k - L(p)$	$i + L(p)$	$\max_{p=0}^{P-1} W(p)$
不規則	`GTOL(`k`)`	`LTOG(`i`,`p`)`	—

変換で添え字計算のコストが増えない．これは分散種別によらない．

(2) 二つ組の並列 DO ループ `DO k=k`$_1$`,k`$_2$ は，分散種別によらず二つ組の DO ループに SPMD 変換できる．

(3) 三つ組の並列 DO ループ `DO k=k`$_1$`,k`$_2$`,k`$_3$ は，block または cyclic または不均等 block 分散なら，三つ組の DO ループに SPMD 変換できる．

(1) は，block-cyclic や不規則分散など従来法での SPMD 変換が複雑になる場合には性能差が大きくなる．

(2) について，図 1 から分かるように，一般にはプロセッサ p に対応するグローバルインデックスの集合を上限と下限だけで表現することはできない．そのため従来法では，cyclic はプロセッサ数 P を増分とする三つ組 DO ループとなり，block-cyclic では大小 2 つの周期を表現するために 2 重ループとなる．不規則分散ではループを多重にしても表現できず，マスク付きのループとするか間接アクセスとするしかなかった．

3. アルゴリズム

3.1 前提とする前処理

配列の形状や，配列とループの相互間のマッピングの整列関係は，事前に以下の条件を満たすように正規化する．

- プロセッサ配列の寸法はどの次元も下限値が 0.
- 配列変数とテンプレートの分散次元の下限値は 0.
- 配列変数の分散次元の寸法は，対応するテンプレートの次元の寸法と一致．
- 配列変数の分散次元や並列ループのマッピングは，テンプレートの 1 つの次元への identical な整列で内部表現されている．

並列ループ内でのリモートアクセス，すなわち，自プロセッサに配置されているとコンパイル時に断定できないデータの参照は，事前に検出して取り除いておく．検出されたリモートアクセスは，並列ループの前後に括り出す．

我々の実装における正規化処理の詳細は，文献 5), 8) を参照されたい．リモートアクセスの検出は，正規化処理と同時に行うことができる．

3.2 準　　備

グローバルインデックス k とローカルインデックス i を相互に変換する関数 *gtol* および *ltog* は，分散パラメータ D によって**表 2** のように求めることができる．block，cyclic，および block-cyclic 分散のこれら変換式の導出は付録 A.1 で与える．不規則分散では，テーブル `GTOL` と `LTOG` はマッピング配列 M から生成する．

gtol を区間数（二つ組）と三つ組に拡張し，以下の関数が得られる．

$(i_1, i_2) = gtol2_D(k_1, k_2, p)$
$(i_1, i_2, i_3) = gtol3_D(k_1, k_2, k_3, p)$

これらは並列 DO ループのループパラメータや，部分配列の添え字三つ組の変換に使われる．たとえば不均等 block 分散の *gtol2* は以下のように計算できる．

$$i_1 = \begin{cases} k_1 - L(p) & \text{if } k_1 > L(p) \\ 0 & \text{otherwise} \end{cases}$$

$$i_2 = \begin{cases} k_2 - L(p) & \text{if } k_2 < L(p+1) \\ W(p) - 1 & \text{otherwise} \end{cases}$$

gtol3 は block-cyclic 分散または不規則分散の場合には値を持たない．三つ組が必ずしも三つ組に変換できないためである．

3.3 原　　理

提案する方法の原理を**図 3** に沿って説明する．並列実行ループ `L1` のインデックス k の動く範囲（すなわち二つ組または三つ組で表される整数の集合）を $\mathbf{K}$ とし，ループの分散を D_1 とすると，同図 (a) に示すようなループの SPMD 変換が可能である（ループインデックスのローカル化）．ループ変数の動く範囲はプロセッサごとに分割されるとともにローカルインデックスに変換されるが，*gtol* の逆関数である *ltog* により，元の k の値が再生されている．一方，配列変数 `A` の宣言形状（すなわち下限と上限で表される整数の集合）を $\mathbf{S}$ とし，`A` の分散を D_2 とすると，配列のアクセス `A(`s`)` の SPMD 変換は (b) のように表現できる．これらを合成すれば，ループ `L1` の中でアクセス `A(`k`)` がある場合の変換は (c) のようになること

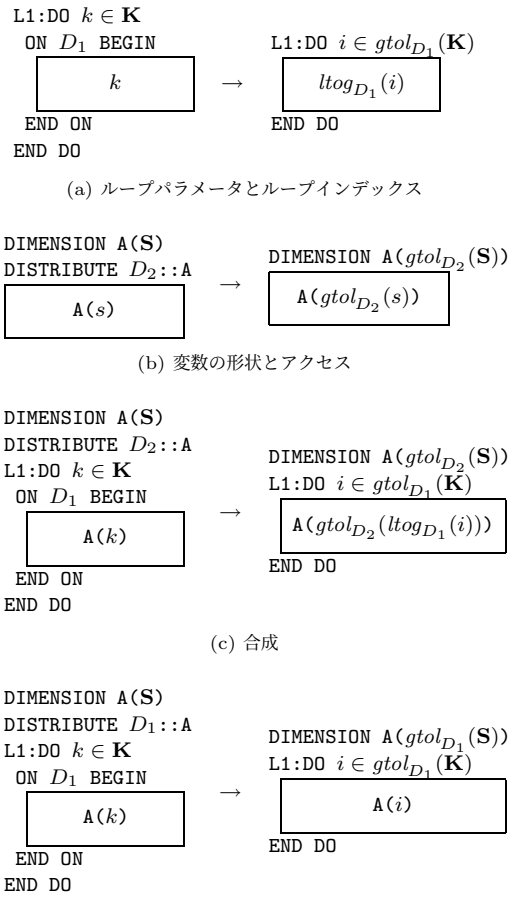


図 3 原理の説明
Fig. 3 Description of the principle.

が分かる．ここでもし D_1 と D_2 が等価であるなら，$gtol$ と $ltog$ は逆関数の関係にあるため，(c) の添え字式は (d) に示すように i に簡単化できる．すなわち，事前の正規化を前提として，ループと変数の同じインデックスが同じテンプレートに identical に整列する場合には，分散種別にかかわらず添え字式が DO 変数自体に簡単化されることが分かる．

上式の変形は一般に数式処理を必要とするため，最簡形が得られない場合がある．そこで実装上の工夫として，あらかじめループと変数の分散が一致していると分かっている添え字については，$ltog_{D_1}$ と $gtol_{D_2}$ による変換をともに省略することとする．

3.4 変換フロー

分散種別に共通なフローを**図 4** に示す．変換の実行手順は以下のとおりである．

手 順 1

テンプレート T のすべての分散次元 d について，

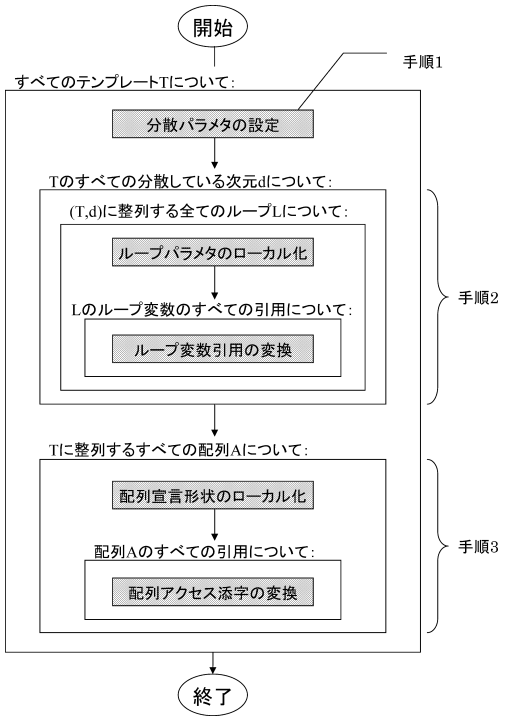


図 4 SPMD 変換の流れ
Fig. 4 Flow of SPMD conversion.

分散パラメータ D_{Td} を実行時に計算して変数に設定する実行コードを，手続きの入口に生成する．変数名はテンプレートの次元ごとに一意の名前とする．ただし，値が静的に得られるパラメータについてはそのようなコードを生成する必要はなく，以降の処理で生成変数を引用する代わりに値を直接引用する．

さらに，不規則分散の場合には，マッピング配列 M からテーブル GTOL と LTOG を作成するプログラム断片をここに展開する．

手 順 2

テンプレート T の分散次元 d に整列するすべてのループ L について，以下を実施する．

(1) ループパラメータをローカルインデックスに変換するためのコードを，そのループの直前に生成する．計算式には，ループパラメータが二つ組のとき *gtol2* を使用し，三つ組のとき *gtol3* を使用する．ただし，三つ組でかつ block-cyclic または不規則分散の場合には，二つ組または三つ組の DO ループに変換できないので，初期値，終値を *gtol2* で変換してマスク付きのループ実行で実現する．

(2) ループの中でループ変数の引用 i を検出し，グローバルインデックスに変換する式 $ltog(i)$ に

置き換える．
ただし，そのループ変数がローカルアクセスの添字式として引用されていて，かつ，その次元で配列が T の次元 d に整列している場合には，この変換は行わない．

手　順　3

テンプレート T に整列するすべての分散配列 A について，以下を実施する．

(1) T のすべての分散次元 d に対応する配列 A の次元について，表 2 に示した分散後のサイズ（その次元方向の要素数）に変換する．不規則分散では分散後のインデックスの最大数を下回らない範囲でできるだけ小さなサイズにする．

(2) A の引用をすべて検出し，すべての分散次元 d に対応する添え字式 s をローカルインデックスに変換する式 $gtol(s)$ に置き換える．
ただし，s がループ変数そのものであり，かつ，そのループが T の次元 d に整列している場合には，この変換は行わない．

4. 実　　装

4.1 HPF トランスレータ fhpf

HPF コンパイラ fhpf は，HPF/JA 仕様を入力とし SPMD 形式の Fortran プログラムコードを出力するトランスレータである．fhpf の出力コードには MPI1.1 API に準拠した MPI ライブラリの呼び出しが含まれる．片側通信や MIMD 処理など MPI2 仕様の機能は含まれない．fhpf の処理系は独自の実行時ライブラリを持たず，必要な実行時ルーチンはすべて Fortran プログラムとして出力コードの中に展開される．また，出力される Fortran プログラムも標準の Fortran90 仕様の範囲なので，任意の Fortran コンパイラと MPI ライブラリを持つ環境であれば，どこでも翻訳し実行することができる．

4.2 fhpf への実装

fhpf トランスレータの流れはおおむね以下のとおりである．

(1) HPF パーサ
(2) 前処理最適化
(3) 正規化
(4) 通信一括化
(5) SPMD コード生成
(6) MPI 呼び出し生成
(7) Fortran 出力

(2) ではローカルアクセスの判定や，軽いループ自動並列化と自動分散を行う．3.1 節で述べた前処理は，(3) と (4) で実施される．それを前提として，3 章のアルゴリズムは (5) に組み込まれている．変換の実例を付録 A.2 に示す．

5. 評　　価

5.1 コンパイラの性能

Himeno ベンチ M モデルの HPF 版を 3 次元分散で作成し，MPI 版と性能比較した結果を**図 5** に示す．測定環境は 670 MHz Pentium III の Linux ブレードサーバ，MPICH V1.2.6 と，富士通の Linux Parallel Language Package V1.0 に含まれる Fortran である．MPI 版は Himeno ベンチのサイト[13] で公開されている Fortran77+MPI 版をそのまま使用した．HPF 版は同サイトの Fortran77 版から作成している．64 ノードを使った 25 通りの 3 次元構成の実測において，fhpf を使った HPF 版は MPI 版に対し平均 96%の高性能を達成している．

5.2 分散種別の違いによる性能の評価

LINPACK で分散種別を変えて並列加速度を測定した結果を**図 6** に示す．富士通 PRIMEPOWER HPC2500[16] の 1 ノードと，それに搭載されている富士通製の Fortran と MPI を使用した．プログラムは IJ 行列の J 方向に対して 1 次元分散で HPF 化したものであり，タイリングやパネル化などの性能を出すための特別な工夫は施していないが，任意の分散方法で実行可能であるという特徴がある[9]．分散種別をさらに増やして，富士通 PRIMEQUEST 480[15] で 8 並列以下について測定した結果を**図 7** に示す．同図は 1 プロセッサあたりの性能を，Fortran による逐次翻訳実行を 1 とした相対値で表している．

結果は，この問題サイズ（N = 4000）に対して非常に良いスケーラビリティを示している．HPF 翻訳による 1 並列実行と逐次実行の間でほどんど性能差がないことから，提案手法が生じさせるオーバヘッドコストは非常に小さいといえる．図 7 ではスーパリニアな速度向上が見られ，問題のサイズと並列数によっては並列化のオーバヘッドコストよりもキャッシュ効率化の効果が上回ることもあると分かる．各々の分散種別について，以下のように考察する．

- cyclic 分散は，期待どおり block 分散よりも良い性能を示している．プログラムの計算コストの大部分を占める LU 分解では，J 方向に対して負荷が不均一であり，block 分散では後ろを担当するプロセッサほど負荷が大きくなる．このようなアプリで簡単に性能を出すには，block よりも cyclic が向いていることが検証された．

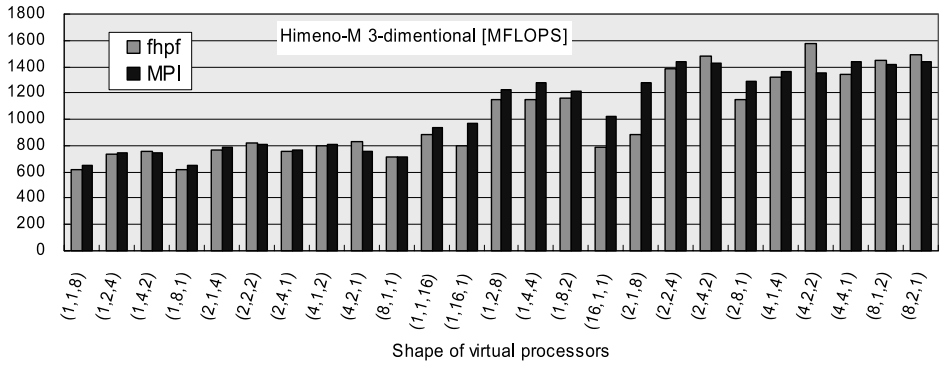


図 5 Himeno ベンチの対 MPI 性能比較（ブレードサーバ）
Fig. 5 Comparison to MPI on Himeno benchmark (on blade server).

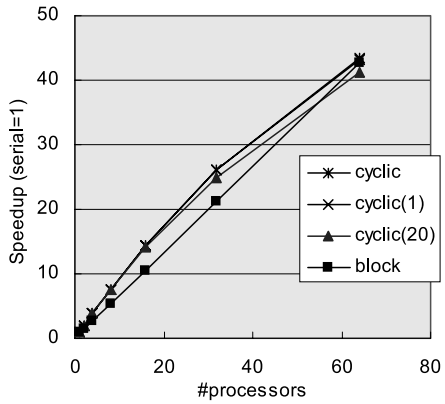


図 6 LINPACK の速度向上比（PRIMEPOWER）
Fig. 6 Speedup ratio on LINPACK (on PRIMEPOWER).

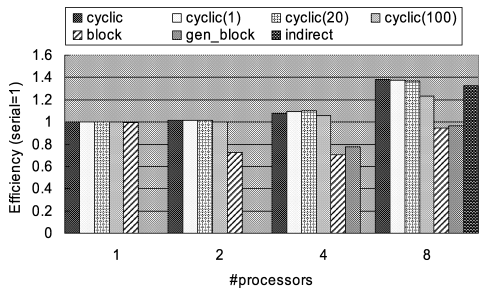


図 7 LINPACK の実行効率（PRIMEQUEST）
Fig. 7 Efficiency on LINPACK (on PRIMEQUEST).

- block-cyclic 分散（cyclic(w)）は，cyclic 分散と比較して遜色のない性能を示している．block-cyclic 分散は従来法（図 2 の c）では 2 重 DO ループを生成しなければならないなど複雑になるため，我々は実装していなかった．提案法の採用により実用的な性能で実装することができた．
- 不均等 block 分散（gen_block）は，最も性能が高くなるようにブロック幅 W（図 1 参照）を調整した結果，block 分散よりも高い性能が得られた．結果として，4 並列で最も性能が高かったブロック幅は，LU 分解の J 方向分散でプロセッサごとの計算量がほぼ均等になるブロック幅 $(1769, 842, 716, 673)$ と，均一なブロック幅 $(1000, 1000, 1000, 1000)$ の，中間付近 $(1385, 921, 858, 836)$ であった．この結果は，計算領域が時間で変化していく様子を考えれば妥当である．なお，均一なブロック幅を使った場合，block 分散との性能差が 1%未満（測定誤差の範囲）であった．このことは，block 分散と比較したコストの増加（手続き入口での分散パラメータの L_{Td} のテーブル作成や，並列ループ入口の *gtol2* 計算の複雑化）が，無視できる程度であったことを意味する．
- 不規則分散（indirect）は，8 並列について，マッピング配列 M（図 1 参照）を乱数で生成して測定した．結果として cyclic 分散に近い性能を得ている．cyclic より若干性能が劣った理由として，乱数の精度の悪さが考えられる（8 プロセッサ間で負荷配分に最大 8%の開きがあった）．なお，マッピング配列で cyclic と同じ分散を表現すると，cyclic 分散との性能差は 1%未満（測定誤差の範囲）であった．このことは，cyclic 分散と比較したコストの増加（手続き入口でのテーブル `GTOL` と `LTOG` の作成や，並列ループ入口の *gtol2* 計算の複雑化）が，無視できる程度であったことを意味する．

5.3 従来法との比較

従来法と提案する方法の性能の違いを**図 8** に示す．同図は cyclic 分散で記述した 1 次元 FFT を使って，問題サイズを変えて 8 並列実行の性能を測定したものである．提案法 (A) と従来法 (B)，(C) は，それぞれ図 2 の (e)，(c)，(d) に対応する．ただし (C) は cyclic

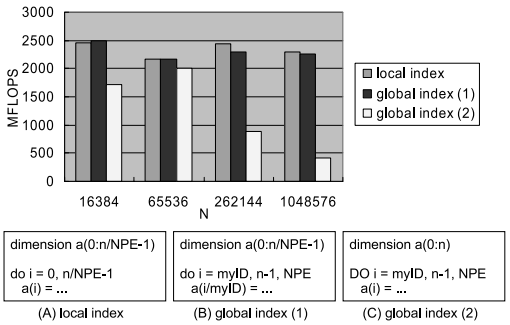


図 8 コード生成の違いによる FFT の実行性能（PRIMEQUEST）
Fig. 8 Efficiency on FFT with different code generations (on PRIMEQUEST).

分散では動的割付けができないため必要以上に大きな静的割付けを行っている．これらのプログラムの特徴的な差異を同図に示している．

(A) と比較し，(B) では最大 5.7%性能が劣る．これは主に添え字式の中に出現する整数除算のオーバヘッドであると考えられる．(C) の生成コードは一見すると最も演算量が少なく効率が良いように見えるが，キャッシュ利用効率の悪さから大きな性能低下を起こしていることが分かる．

block-cyclic 分散では，表 2 の gtol 関数の比較で明らかなようにさらに複雑な添え字変換を必要とするため，性能差はより広がると考えられる．しかし，従来法の実現がコードの手修正でも容易ではないため，実測による比較はできていない．提案法の効果は，性能向上よりもむしろ，従来は実現困難だった分散種別が実装可能になった点が大きい．

6. 関連する研究と動向

中田は著書 11) の中で SPMD 変換を理論的に説明している．この中ではループインデックスのローカル化を暗に前提としているが，分散種別に関する広範な考察はない．

従来の実装では不規則分散は性能が出しにくいとされ，HPF 言語の実用化拡張である HPF/JA 仕様では制限事項とされていた[4]．

OpenMP V2.0 仕様には多重並列化はあっても多次元並列化はない[7]．必要性は認識され始めていて，V3.0 仕様拡張の議論の中で取り上げられている．

本研究で対象とする多次元分散は分散次元ごとに独立に処理できる場合に限られる．これは HPF 仕様でサポートされる多次元分散がそのような前提で設計されているからである．Rice 大学の dHPF プロジェクトで研究されている Overpartitioning は次元間で絡み合った多次元分散であり，本稿の範囲を超える[1]．また，DARPA の HPCS で研究されている Chapel 言語でも次元間で影響し合う不均等分散が検討されている[2]．

7. まとめと課題

SPMD コード生成における，インデックスのローカル化の手法を提案した．この手法は分散種別によるコンパイラの処理の違いを局所化する．また，従来は性能が出しにくいとされていた block-cyclic 分散や不規則分散について，単純な block や cyclic と同等な性能を出すことが実測でも示された．

この手法の効果は，実行性能の向上だけではなく，コンパイラ開発の生産性向上にも役立っている．コンパイラ内部や実行時処理で，分散種別によって処理を分けなければならなかった部分が非常に小さくなったためである．

今後の課題として，Fortran90 流の書式である添え字三つ組，配列を返す関数，配列組込み関数などにうまく展開して，引数渡しの効率を改善したい．また，従来高級言語ではうまく処理されずあまり使われていなかった不規則分散について，実アプリケーションでの有効性を検証していく必要がある．

謝辞 本研究の一部は，NEDO（新エネルギー・産業技術総合開発機構）の基盤研究促進事業「高信頼・低消費電力サーバの研究開発」からの委託に基づいている．性能評価では一部 HPF 推進協議会[14] に支援をいただいた．

参 考 文 献

1) Broom, B., Chavarria-Miranda, D., Jin, G., Fowler, R., Kennedy, K. and Mellor-Crummey, J.: Overpartitioning with the Rice dHPF Compiler, *Proc. HPF User Group Meeting* (*HUG2000*) (Oct. 2000).
2) Callahan, D., Chamberlain, B.L. and Zima, H.P.: The Cascade High Productivity Language, *Proc. 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (*HIPS' 04*) (2004).
3) High Performance Fortran Forum: *High Performance Language Specification Version 2.0* (1997).
4) High Performance Fortran Forum（著），富士通，日立，日本電気（訳）：High Performance Fortran 2.0 公式マニュアル，シュプリンガー・フェアラーク東京，ISBN4-431-70822-7 (1999).
5) Iwashita, H. and Aoki, M.: Mapping Normalization Technique on the HPF compiler fhpf,

Proc. International Symposium on High Performance Computing 2005, Nara, Japan (Sep. 2005).
6) Iwashita, H., Hotta, K., Kamiya, S. and van Waveren, M.: Towards a Lightweight HPF Compiler, *Proc. International Symposium on High Performance Computing 2002*, Japan, LNCS 2327 (May 2002).
7) OpenMP Architecture Review Board: *OpenMP Fortran Application Program Interface Version 2.0* (2000).
8) 岩下英俊：並列化コンパイラ fhpf の正規化変換で用いられる数式処理, 数式処理, Vol.11, No.3-4, pp.125–140 (2005).
9) 岩下英俊，岡部寿男，杉崎由典，青木正樹：LINPACK と FFT による HPF コンパイラ fhpf の生産性の評価，情報処理学会研究報告（HOKKE2006），Vol.2006, No.20, pp.67–72 (2006).
10) 岩下英俊, 進藤達也, 岡田 信：VPP Fortran：分散メモリ型並列計算機言語，情報処理学会論文誌，Vol.36，No.7，pp.1542–1550 (1995).
11) 中田育男：コンパイラの構成と最適化，朝倉書店，ISBN 4-254-12139-3 (1999).
12) FFTE: A Fast Fourier Transform Package. http://www.ffte.jp/
13) Himeno ベンチマークプログラム. http://accc.riken.jp/HPC/HimenoBMT/
14) HPF 推進協議会ホームページ. http://www.hpfpc.org/
15) 富士通 IA サーバ PRIMEQUEST 480. http://primeserver.fujitsu.com/primequest/
16) 富士通 UNIX サーバ PRIMEPOWER HPC2500. http://primeserver.fujitsu.com/primepower/

付　　録

A.1 変換式の導出

block 分散，cyclic 分散，block-cyclic 分散について，表 2 にまとめた変換関数を導出する．分散パラメータの定義は表 1 と図 1 参照．以下では，グローバルインデックスを k（≥ 0），ローカルインデックスを i（≥ 0），プロセッサ id を p とする．次式はつねに成り立つ．

$$0 \leq p < P. \tag{1}$$

A.1.1 BLOCK(w)

図 9 に示すとおり，ローカルインデックスはグローバルインデックスを $-pw$ だけシフトさせたものであるので，

$$i = k - pw \tag{2}$$

$$k = i + pw. \tag{3}$$

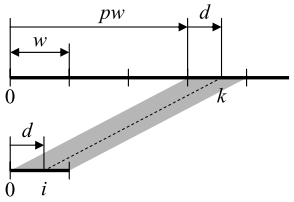


図 9 Block 分散のインデックス変換
Fig. 9 Index transformation on block distribution.

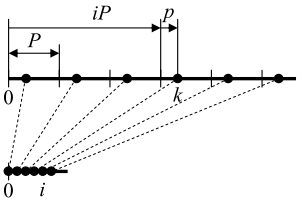


図 10 Cyclic 分散のインデックス変換
Fig. 10 Index transformation on cyclic distribution.

ここで，$0 \leq i < w$ を考慮すると式 (2) から p を消去できて，

$$i = k \bmod w. \tag{4}$$

A.1.2 CYCLIC

cyclic 分散の定義から，

$$p = k \bmod P \tag{5}$$

は明らか．また，**図 10** に示すとおり，プロセッサ p のローカルインデックス i に対応するグローバルインデックス k は，周期 P の等差数列

$$k = iP + p \tag{6}$$

となる．次に，不等式 (1) に式 (6) を変形した $p = k - iP$ を代入して変形すると，

$$\frac{k}{P} - 1 < i \leq \frac{k}{P} \tag{7}$$

したがって，

$$i = \left\lfloor \frac{k}{P} \right\rfloor. \tag{8}$$

A.1.3 CYCLIC(w)

block-cyclic 分散では，グローバルインデックス k に関して大周期 Pw と小周期 w が現れる．k に対応する大周期と小周期のカウントをそれぞれ m, n とし，

$$k = mPw + c \tag{9}$$

$$k = nw + d \tag{10}$$

とおく．ここで，

$$0 \leq c < Pw \tag{11}$$

$$0 \leq d < w \tag{12}$$

である．block-cyclic 分散の定義から，

$$n = mP + p \tag{13}$$

という関係があるので，これと式 (10) を使って式 (9) を整理すると，

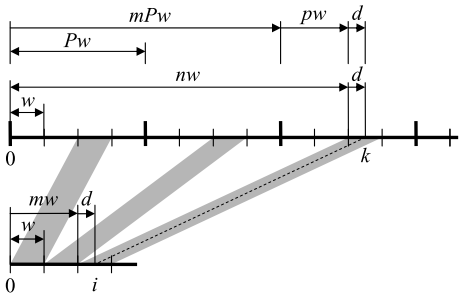


図 11 Block-cyclic 分散のインデックス変換
Fig. 11 Index transformation on block-cyclic distribution.

$$k = mPw + pw + d. \tag{14}$$

一方，ローカルインデックス i の周期は w であり，i の周期のカウントは k の大周期のカウントと一致する．また，周期からのずれはグローバルインデックスの小周期からのずれと一致する．すなわち，

$$i = mw + d. \tag{15}$$

以上の関係を**図 11** に示す．

さて，式 (9)，(15) から

$$d = k \bmod w = i \bmod w \tag{16}$$

が成り立ち，式 (13) から

$$p = n \bmod P \tag{17}$$

が成り立つ．また，不等式 (1) と式 (6) から式 (8) を得たのと同様に，不等式 (11) と式 (9)，不等式 (12) と式 (10)，不等式 (12) と式 (15) から，それぞれ以下の式が得られる．

$$m = \left\lfloor \frac{k}{Pw} \right\rfloor \tag{18}$$

$$n = \left\lfloor \frac{k}{w} \right\rfloor \tag{19}$$

$$m = \left\lfloor \frac{i}{w} \right\rfloor. \tag{20}$$

結果として，式 (15)，(18)，(16) により，

$$i = w \left\lfloor \frac{k}{Pw} \right\rfloor + (k \bmod w), \tag{21}$$

式 (14)，(20)，(16) により，

$$k = Pw \left\lfloor \frac{i}{w} \right\rfloor + pw + (i \bmod w). \tag{22}$$

A.2 fhpf トランスレータの出力例

コマンド

```
% fhpf block.hpf -f77
```

を使用した．入力ファイル `block.hpf` と生成された出力ファイル `block.mpi.f` は以下のとおりである．

入力：`block.hpf`

```
      integer A(100)
!hpf$ processors P(4)
!hpf$ distribute A(block) onto P
!hpf$ independent
      do i=n1,n2
        A(i)=i
      enddo
      end
```

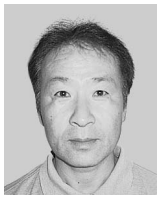
出力：`block.mpi.f`

```
 PROGRAM ORG__MAIN___
 INCLUDE 'mpif.h'
 EXTERNAL ORG___MAIN___
 INTEGER*4 ORG_RANK
 INTEGER*4 ORG_SIZE
 INTEGER*4 ierrX0
 COMMON /ORG_COMMON/ORG_RANK,ORG_SIZE
 CALL mpi_init(ierrX0)
 CALL mpi_comm_rank(MPI_COMM_WORLD,ORG_RANK,ierrX0)
 CALL mpi_comm_size(MPI_COMM_WORLD,ORG_SIZE,ierrX0)
 CALL ORG___MAIN___()
 CALL mpi_finalize(ierrX0)
 END
 SUBROUTINE ORG___MAIN___()
 INCLUDE 'mpif.h'
 INTEGER*4 a(0:24)
 INTEGER*4 i
 INTEGER*4 n1
 INTEGER*4 n2
 INTEGER*4 ORG_RANK
 INTEGER*4 ORG_SIZE
 INTEGER*4 spmdX0
 INTEGER*4 spmdX1
 INTEGER*4 spmd_startX0
 INTEGER*4 spmd_endX0
 INTEGER*4 spmd_stepX0
 INTEGER*4 ierrX0
 COMMON /ORG_COMMON/ORG_RANK,ORG_SIZE
 spmdX0 = mod(ORG_RANK,4)
 spmdX1 = 25*spmdX0
 IF (spmdX0.LE.(n1-1)/25) THEN
   spmd_startX0 = n1-1-spmdX1
 ELSE
   spmd_startX0 = 0
 ENDIF
 IF (spmdX0.LT.(n2-1)/25) THEN
   spmd_endX0 = 24
 ELSE
   spmd_endX0 = n2-1-spmdX1
 ENDIF
 spmd_stepX0 = 1
 DO i=spmd_startX0,spmd_endX0,1
   a(i) = i+spmdX1+1
 ENDDO
 END
```

(平成 18 年 1 月 27 日受付)
(平成 18 年 5 月 16 日採録)

岩下 英俊（正会員）

1963 年生．1986 年愛媛大学工学部電子工学科卒業．1988 年同大学大学院工学研究科修士課程修了．同年（株）富士通研究所入社．1997 年より富士通（株）．博士（工学）．入社以来，HPC 向け並列プログラミング言語と並列化コンパイラの研究開発に従事．IEEE 会員．



青木 正樹

1959 年生．1977 年富士通株式会社入社．以降 HPC 向け言語処理系開発に従事．現在，同社ソフトウェア事業本部 MWC 事業部第二開発部部長．