社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－ARC－167（12）
2006－HPC－105（12）
2006／2／27

LINPACK と FFT による HPF コンパイラ fhpf の生産性の評価

岩下 英俊* 岡部 寿男† 杉崎 由典* 青木 正樹*
* 富士通 (株) ソフトウェア事業本部
† 京都大学 学術情報メディアセンター

HPF コンパイラ fhpf を使って 2 つのベンチマークプログラムを並列化する事例を通して，プログラム開発の生産性について議論する．LINPACK では，コア部分の逐次プログラムの作成から始めて少量の HPF 指示行を加えた．FFT では，OpenMP プログラムの解析から始めて最小の手間で HPF への移植を行った．HPF/JA 仕様を活用した段階的な並列化の方法により，どちらの事例でも短期間で良好な並列効果が得られた．生産性は開発速度，実行性能，再利用性の 3 つの尺度で評価することができる．事例の考察から，fhpf の特長的な機能は生産性に大きく貢献していると言える．

Evaluation of Productivity on the HPF compiler fhpf Applied to LINPACK and FFT

Hidetoshi Iwashita* , Yasuo Okabe† , Yoshinori Sugisaki* and Masaki Aoki*
* Software Unit, Fujitsu Limited
† Academic Center for Computing and Media Studies, Kyoto University

Productivity of program development is discussed after a study of two cases of parallel programming with the HPF compiler fhpf. In the case of LINPACK, we made a serial program and then added a few HPF directive lines. In the case of FFT, we ported an OpenMP program into HPF after an analysis of the source code. Due to the gradual programming using the HPF/JA language, reasonable scalability was found only in a short term of development. Productivity can be estimated with three scales – development speed, performance, and reusability. The cases taught that features of fhpf contribute the productivity.

1 はじめに

HPC アプリケーションは最高性能の評価ばかりに目を奪われがちだが，生産性を真剣に考えなければ実世界アプリケーションの発展がない．PC クラスタやブレードサーバなど安価な並列計算にもターゲットを広げ，かつ計算機の専門家でない計算科学者にもどんどん高性能アプリケーションを書いて頂くためには，MPI ライブラリではない高級言語の普及が必要であると我々は考えている．

我々は HPF 推進協議会（HPFPC）[9] のメンバと共に，HPF 言語の拡張である HPF/JA 言語の普及活動を続けている．HPF はデータ並列（data parallelism）の考えに基づく言語仕様であり，データの分散さえ利用者が記述すれば，計算の並列化とデータの通信は原則として自動でやってくれると考える．これに実用性を考慮した拡張を加えた言語が HPF/JA[5] であり，コンパイラ単独で自動化が果たせなかった部分について，利用者が段階的な改善を加えることができる手段を提供している．

米国でも最近になって Co-Array Fortran, UPC などの言語などが台頭し，生産性 (productivity) をキーワードに MPI との差別化を図っている [6]．DARPA の HPCS(High *Productivity* Computer Systems) プロジェクトでも生産性向上を目指した新しい並列言語の研究開発が進んでいる [1]．国内外のこういった活動が互いに影響し合って，近い将来には生産性を重視した並列言語の共通化・標準化が進むものと期待する．

本稿は，我々が開発中の HPF コンパイラ fhpf を使った並列化の実例を通して，並列言語とコンパイラがどのように生産性向上に貢献しているかを述べるとともに生産性として何を評価すべきかを議論する．以下，2 章で fhpf コンパイラを紹介した後，3 章で fhpf による LINPACK と FFT の並列化の事例を作業手順に沿って紹介する．その結果を生産性の観点から 4 章で分析して，5 章でまとめとする．

[0]本研究は，一部 HPF 推進協議会の支援による．
[0]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．

2 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[1]．目指したのは，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 [3].

fhpf の出力コードには MPI1.1 API に準拠した MPI ライブラリの呼出しが含まれる．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．

3 並列プログラミングの事例

HPC Challenge ベンチマーク（HPCC）にある High-Performance LINPACK (HPL)[8] と 1 次元 FFT[7] を題材とした．どちらもデータの初期化には，HPCC の Web ページからリンクされる C で書かれた乱数生成ルーチン hpl_init を言語間結合して使用した．結果の精度検証，性能測定区間などは HPCC の規約に準じた．

目的は最高性能を得ることではなく，妥当な性能が妥当な開発工数で得られるかどうか確認することである．プログラムの改造や新規作成を伴う並列化チューニングを行うが，その過程でコスト対効果を常に考えて，1ヶ月程度の短期で着実な成果を出すようにする．作業効率を度外視した性能改善や，当たれば幸いの発想での試行錯誤は行わない．

測定環境は以下の 2 つである．

- PRIMEPOWER HPC2500，1.3GHz の版．96CPU の 1 ノード．ノード内共有メモリ実効 256GB．ソフトは富士通 Parallelnavi Fortran と富士通 MPI.
- ブレードサーバ．ノード内は Pentium III(670MHz)1 個と 377MB メモリ．10 ノード 1 ラックでラック間 1Gbps．ソフトは MPICH1.2.6 と富士通 Fortran.

3.1 新規作成からの並列化 (HPL)

コア部分である LU 分解と前方・後方代入のサブルーチンを新規に作成した．プログラム全体の骨組みは，VPP500 向けに VPP Fortran で記述されたプログラム [4] を利用した．

[1]HPFPC に会員登録して頂いた方には無償で配布している．詳しくは Web ページ [9] 参照.

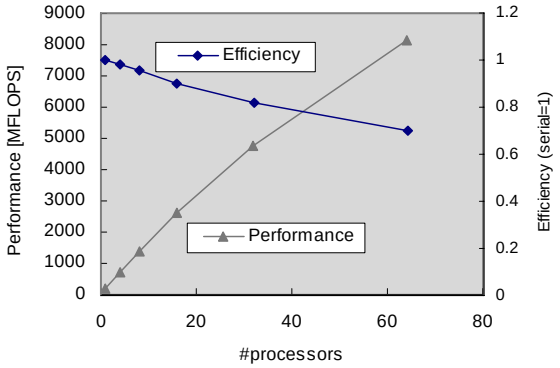

図 1: HPL の並列効果 (N=4000)

3.1.1 プログラミング

LU 分解と前方・後方代入のサブルーチンについて，Web で公開されている情報 [10, 11] などを参考に，基本に忠実なスタイルで Fortran で記述した．その後，HPF 指示行を加えて並列化する方針で進めた．

LU 分解の第 k ステップで発生する通信は，1 次元め (行) を分散する場合，**1a)** pivot 行を見つけるため最大値の reduction，**1b)** pivoting 処理で行の swap，**1c)** U 行列計算で行の broadcast が生じる．一方，2 次元め（列）を分散する場合，**2a)** pivot 行番号 `ipiv` の broadcast，**2b)** U 行列計算で列の broadcast が生じる．この比較により通信量の少ない 2 次元めの分散を選択した．実装ではさらに工夫し，列の broadcast を早く行って冗長計算で `ipiv` を計算することで `ipiv` の broadcast を不要にした．

分散種別は，LU 分解でも前方・後方代入でも列ごとに負荷か不均等であるので，cyclic 分散を採用した．アルゴリズム上プロセッサ数に特別な制約はないので，プロセッサ数の決定はコンパイル時ではなく実行時に持ち越すことができる．

小さなプログラムなので，LU 分解と前方・後方代入のサブルーチンをそのまま A に載せる．

3.1.2 性能評価

図 1 に PRIMEPOWER HPC2500 を使った測定結果を示す．左側の座標は性能値，右側の座標は Fortran での翻訳実行を基準とした実行効率を示す．元数 N が比較的小さな場合には良好な並列効果が得られることが分かった．

しかし図 2 から分かるように，N を大きくしていくと性能が飽和する．一般に LINPACK では，計算量のオーダが $O(N^3)$ と大きいので，N が大きくなると通信オーバヘッドが相対的に減少して性能が出やすくなる．しかし本プログラムでは，3.1.1 項 2b の通信が大きいことと

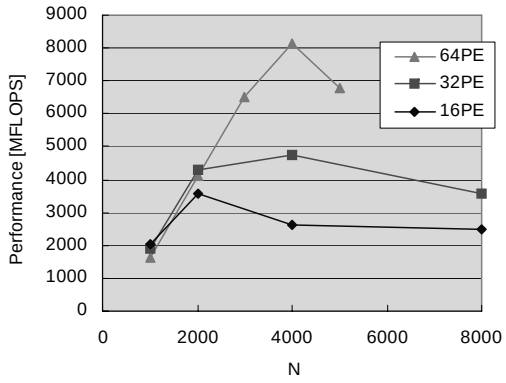

図 2: HPL の問題サイズによる性能の違い

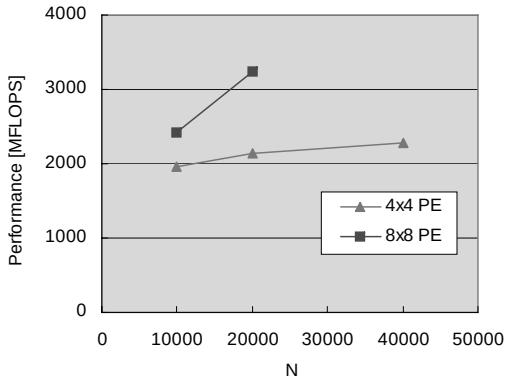

図 3: 2 次元分散 HPL の問題サイズによる性能

(トータルで $O(pN^2)$), 並列効果のない冗長実行部分が多いため, スケール効果が得られないようである.

3.1.3 スケール効果の改善

大きな N に対して性能を出すためには, 通信量のオーダを $O(N^3)$ より小さくし, かつプログラム中の並列効果のない部分を最小にする必要がある. 我々は, プログラムを 2 次元 cyclic 分散に書き換えた. 2 次元分散では 3.1.1 項で述べた 1a から 2b のすべての通信が発生するが, reduction や broadcast の対象範囲が p から $\sqrt{p}$ に変わるので, p が十分大きくなれば通信量は減少する (p はプロセッサ数).

この結果を図 3 に示す. 1 次元分散の場合に比べて 10 倍以上大きな N に対しても性能が上がり続けていることが分かる. しかし絶対性能はあまり高くない. より多数のプロセッサでより大きな N で計測すれば 1 次元分散よりも性能が出る可能性はあるが, 1 プロットの計測に必要な時間が数時間を越えてくる ($O(N^3)$ で大きくなる) ため, 開発効率とコスト面からここで計測を断念した.

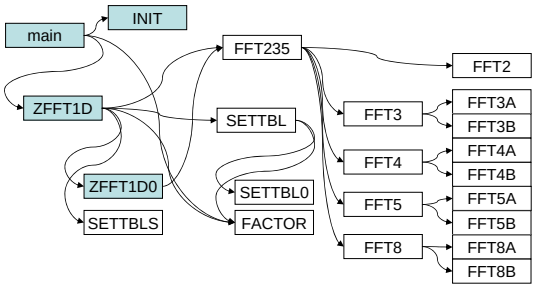

図 4: FFT の call graph

3.2 OpenMP からの移植 (FFT)

HPCC の Web site からリンクされている ffte-4.0[7] の 1D-complex 版を利用した. このプログラムは OpenMP Fortran で書かれているので, 並列実行ループの指示はあるが, データの分散に関わる指示はない. これを分散メモリ環境で動作するように HPF に移植する.

プログラムは初期化部分を除いても 1000 行程度あり, 短期間にすべては理解できないと考えた. プログラムの構造の分析から始め, 並列化のために必要な最小限だけを理解して進める.

3.2.1 並列化する手続きの選択

プログラムの構成は図 4 のようになっている. プログラムの主変数はメインプログラムの `A,B` であることは比較的容易に分かる. 方針として, 主変数のプロセッサへの分散配置を決め, それに沿った手続きの並列化を行うこととする. 他の変数は必要になったときに分散を決める.

HPF では data parallelism に基づき, 主変数を分散配置してその分散に沿った並列実行を考える. 主変数が引数や大域変数として受け渡されて処理を行っているサブプログラムが並列化の対象とされるべきである. また, そのようなサブプログラムから並列実行中に呼び出されるサブプログラムは, ネスト並列を考えない限り並列化の対象とはしない. このような考え方でサブプログラムを分類すると, 同図でハッシュを付けたサブルーチン `ZFFT1D`, `ZFFT1D0`, `INIT` が並列化されるべき手続きと分かる. また, 主変数のセグメントが受け渡されるサブルーチン `FFT235` の呼出しには注意が必要であることが分かる. 他のサブルーチンについては, 手を入れる必要はなく, Fortran プログラムとして翻訳し結合する.

3.2.2 コアルーチンの並列化

主変数 `A,B` の分散方法をコア部分 `ZFFT1D0` に対して最適となるように決定する. `ZFFT1D0` を観察した結果,

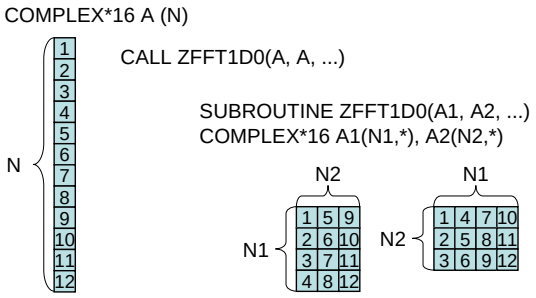

図 5: FFT の引数渡しの様子

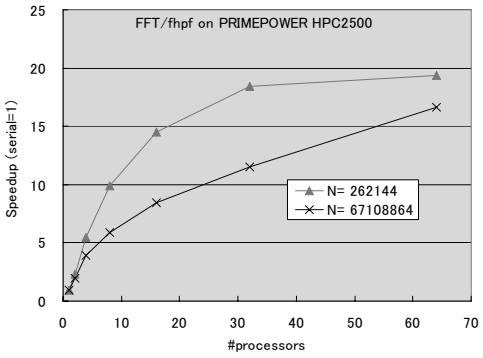

図 6: PRIMEPOWER での FFT の加速率

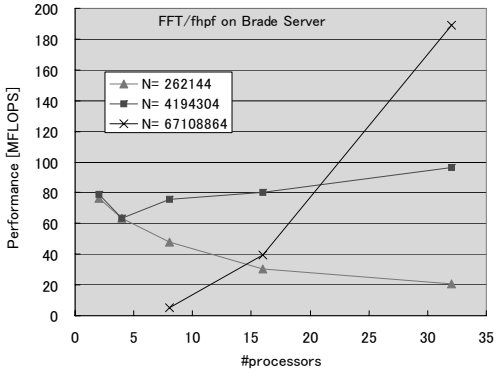

図 7: ブレードサーバでの FFT の性能

前半のループ DO I から，A(I,:) と B(I,:) のアクセスがあるので共に 1 次元めでの分散が適すると判断できる．しかし，後半のループ DO J では B(:,J) のアクセスがあるので B は 2 次元めの分散が適する．そこで，B に対応する分散次元の異なる配列 BT を宣言し，前半と後半のループの間で B から BT への全配列代入を行うことにより，1 回の transpose 通信だけでサブプログラム全体が処理できるように記述した．

なお，今回は作業効率を優先する立場から，ベースのプログラムでタイリングによって多重化されていたループは元の単純なループに戻した．タイリングは HPF でも有効なプログラミング技術であるが，パラメタの調整に手間がかかるし，また，キャッシュ競合に関しては HPF のデータ分散によって完全に回避されるので効果がない．

3.2.3 サブプログラムインタフェース

コアルーチン ZFFT1D0 の呼び出しでは変数 A,B についてトリッキーな引数渡しが行われている．図 5 に示すように，実引数（呼び出し側）と仮引数（サブルーチン側）の形状を敢えて違えることで，サブルーチン内の多次元インデックス計算を簡単にしているようである．

HPF では，このような形状の不一致があって，実引数と仮引数のそれぞれに分散が指定されているとき，サブルーチンの呼出し時と復帰時に自動的に再分散を起すことで正常動作を保障している．しかし主変数に対するこのような全対全通信は，大きな実行時オーバヘッドとなる．

そこで，この通信を分散方法の選択と制限付けによって完全に回避することを考えた．実引数を cyclic 分散，仮引数も 1 次元めで cyclic 分散とし，問題サイズ N を平方数に限る（すなわち N1=N2）とし，N1 はプロセッサ数で必ず割り切れるとすれば，実引数と仮引数の対応する要素は必ず同じプロセッサ上にあることが保障できる．fhpf では利用者責任によるこの保障があれば，たとえ引数の形状が違っていても通信オーバヘッドの生じない手続き呼出し／復帰を実現している．

3.2.4 性能評価

PRIMEPOWER での結果を図 6 に示す．それぞれのデータサイズでの逐次実行を 1 とする相対性能で表している．

ブレードサーバでの結果を図 7 に示す．プロセッサ数 p，データサイズ N のとき，分散メモリでは 1 プロセッサ当りの通信コスト $O(N/p)$ に対して計算量が $O((N \log N)/p)$ と速いことを反映して，データサイズが増えるほどスケーラビリティが出ていることが見て取れる．データサイズが大きくなると少数ノードではメモリ不足となり計測できない．

4 プログラムの生産性

生産性を決めるものは，生産のスピードと質であろう．HPC プログラムの開発では，開発速度と実行性能であると言える．プログラム開発をサイクルで考えると，こ

れに開発物の再利用性という観点が加わる．この 3 つの観点に沿って，3 章の事例を考察する．

4.1 開発速度の考察

ソースコード量を比較する．コメント行を除く行数はそれぞれのプログラムで以下の通りである．なお，乱数生成ルーチン（C 言語，2000 行）がデータの初期化のために共通に使用されているが，asis の使用でありカウントから除外している．

	HPL/1D	HPL/2D	FFT
計測区間/修正	98 (8)	319 (77)	158 (16)
計測区間/asis	0 (0)	0 (0)	738 (0)
main, 他/修正	152 (3)	197 (12)	101 (9)
main, 他/asis	0 (0)	0 (0)	0 (0)
計	250 (11)	516 (89)	997 (25)

括弧内は HPF 指示行の行数（内数）.「asis」は全く修正を加えていないファイルの行数であり，わずかでも手を入れたファイルは全行数を「修正」に計上.

修正したファイルの量は，概ねそのコーディングに要した手間を反映していると感じられる．HPL/2 次元分散では，Fortran 部分の書き換えの手間が大きかった．FFT では asis で済むファイルを切り出す作業を先に正確に行ったことで，修正が楽になった．

他には以下のような因子が開発速度に影響を与えると考えられる．

実行の elapsed time　HPL ではデータサイズが大きくなると 1 回の実行に数時間以上を要し，時間的，経済的に開発が制約を受けた．

開発途中での実行確認の容易さ　HPF プログラムは Fortran コンパイラで翻訳すれば常に同じ意味の逐次実行が可能である．この確認により実際に何度かバグの作り込みに気がついた．

4.2 実行性能の考察

HPL/1 次元分散と FFT は，逐次実行と比較した並列化効率の評価では良好な結果を得た．逐次プログラムから出発して並列化を進めるというアプローチでは，HPF/JA 言語仕様と fhpf コンパイラは一定の成功を収めたと言える．

しかし逐次実行の性能が十分ではないため，64 並列での実行性能はいずれもハードの理論ピーク性能の 5%未満に留まっている．より性能が必要なら，タイリングやプリフェッチなど，逐次実行性能向上に有効なプログラムチューニングが必要であろう．

HPL ではデータ量の増大に対して性能の飽和が見られ，2 次元分散に変更しても開発速度から今回のアプローチでは限界が見られた．FFT では 64 並列まで性能向上が続くものの飽和気味である．さらなる性能改善には，逐次からの並列化というアプローチではなく，最初から並列を考慮したアルゴリズムからの開発が有効かもしれない [8]．そのような開発スタイルに対する HPF/JA と fhpf の評価は今後の課題である．

4.3 再利用性の考察

ここでは，生産物やその部分品がどの程度使い回せるかという尺度を再利用性と呼ぶ．再利用性は，次の開発プロジェクトへの流用のしやすさだけではなく，その開発自体のスピードにも影響することが多い．

プラットフォーム無依存性　fhpf は標準的な MPI と Fortran を出力とするので，ターゲットマシンを選ばず動作確認ができる．

台数無依存性　HPL/1 次元分散はプロセッサ数不定のプログラムとしたので，並列度を変えても再コンパイルする必要がない．FFT ではデータの形状とプロセッサ数に制約関係があるので，翻訳時にプロセッサ数を決める必要があった．

メモリのスケーラビリティ　分散メモリ計算環境では，1 ノードではメモリ不足で実行できない問題が解けたり，ノード数に比例する大きな問題が解けることが利用価値を広げる．FFT のブレードサーバでの結果が示すように，fhpf では実行時の情報を保持する構造体を極力小さくし，メモリのスケーラビリティを向上させている．

言語間結合の容易さ (interoperability)　逐次 Fortran プログラムとの結合性のよさにより，FFT では大半のサブルーチンを中身を正確に理解しないまま asis で利用することができた．データの初期化では C プログラムとも結合している．また，FFT で現れた次元数の一致しない引数渡しでは，fhpf は再マッピングによるオーバヘッドの生じない引数の分散を実現した．

5 まとめ

逐次プログラムとしての動作を確認しながら並列化を進めるというプログラミングスタイルで，HPL では逐次プログラムの記述から始める方法，FFT では既存の OpenMP プログラムを分散メモリ向けに移植する方法を試みた．いずれも良好なスケーラビリティが得られている．HPF/JA 仕様はこのようなプログラミングスタイルに向いている．

プログラムの生産性は，開発速度，実行性能，汎用性の３つの尺度で考えられる．fhpf コンパイラは，台数無依存，プラットフォーム無依存，メモリのスケーラビリティ，引数の効率的な受渡しなどの特長により，プログラム開発の生産性向上に貢献している．

本稿では fhpf コンパイラを使った生産性について述べたが，処理系を越えた機能的・性能的な互換性も，利用者の立場から見た生産性の重要な要素である．今後の課題として，HPFPC の活動などを通して取り組んでいく必要がある．

参考文献

[1] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *Proceedings of HIPS2004*. 2004.

[2] H. Iwashita and M. Aoki. Mapping Normalization Technique on the HPF compiler fhpf, *ISHPC2005*. 2005.

[3] H. Iwashita, K. Hotta, S. Kamiya, and M. van Waveren. Towards a Lightweight HPF Compiler, *ISHPC2002* (LNCS 2327). 2002.

[4] M. Nakanishi, H. Ina, and K. Miura. A High Performance Linear Equation Solver on the VPP500 Parallel Supercomputer, *SC1994*. pp.803-810. 1994.

[5] 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 年.

[6] *Co-Array Fortran*. http://www.co-array.org/

[7] *FFTE: A Fast Fourier Transform Package*. http://www.ffte.jp/

[8] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. *HPL Algorithm*. http://www.netlib.org/benchmark/hpl/ algorithm.html

[9] **HPF 推進協議会ホームページ**. http://www.hpfpc.org/

[10] http://mikilab.doshisha.ac.jp/dia/research/report/2002/0612/018/report20020612018.html

[11] http://www2.ee.knct.ac.jp/el/E4/H15-E406/lu1.html

A　HPF プログラムの実例 (HPL)

A.1　LU 分解

```
      SUBROUTINE DVALU(A,NP,N,IP,PIV)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(NP,N), IP(N)
      LOGICAL PIV
      DIMENSION A1(N)
      DIMENSION FACT(N)
!hpf$ distribute A(*,cyclic)
!hpfj asyncid id1

      DO I=1,N
        IP(I)=I
      END DO

      DO 10 ITOP=1,N-1
C--------------------------------------
C     Bloadcast one line
C--------------------------------------
!hpfj   asynchronous(id1), nobuffer
        A1(ITOP:N) = A(ITOP:N,ITOP)
!hpfj   asyncwait(id1)

C--------------------------------------
C     PIVOTING
C--------------------------------------
        IF (PIV) THEN
C-- find pivot (redundant exec.)
          IPIV=ITOP
          AMAX=A1(ITOP)
          DO 20 I=ITOP+1,N
            TMP=ABS(A1(I))
            IF (AMAX.LT.TMP) THEN
              AMAX=TMP
              IPIV=I
            END IF
 20       CONTINUE
C-- swap rows
          IF (ITOP.NE.IPIV) THEN
            I=IP(ITOP)
            IP(ITOP)=IP(IPIV)
            IP(IPIV)=I
            TMP=A1(ITOP)
            A1(ITOP)=A1(IPIV)
            A1(IPIV)=TMP
!hpf$        independent, new(TMP)
            DO 30 J=1,N
              TMP=A(ITOP,J)
              A(ITOP,J)=A(IPIV,J)
              A(IPIV,J)=TMP
 30         CONTINUE
          END IF
        END IF

C--------------------------------------
C     compute L(ITOP+1:N,ITOP) and
C      next A(ITOP+1:N,ITOP+1:N)
C--------------------------------------
        AINV=1.0/A1(ITOP)
        DO I=ITOP+1,N
          A1(I)=A1(I)*AINV
        END DO
!hpf$    independent, new(I)
        DO 40 J=ITOP,N
          IF (J.EQ.ITOP) THEN
            DO I=ITOP+1,N
              A(I,J)=A(I,J)*AINV
            END DO
          ELSE
            DO I=ITOP+1,N
              A(I,J)=A(I,J)-A1(I)*A(ITOP,J)
            END DO
          END IF
 40     CONTINUE

 10   CONTINUE     ! ENDDO ITOP
      RETURN
      END
```

A.2　前方・後方代入

```
      SUBROUTINE DLUX(A,NP,N,B,IP)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(NP,N), B(N), IP(N)
      DIMENSION BTMP(N)
!hpf$ distribute A(*,cyclic)

C-- pivot B
      DO I=1,N
        BTMP(I)=B(IP(I))
      END DO
      DO I=1,N
        B(I)=BTMP(I)
      END DO

C-- solve Ly=b, replacing vector B
      DO I=2,N
        TMP=B(I)
!hpf$   independent, reduction(TMP)
        DO J=1,I-1
          TMP=TMP-A(I,J)*B(J)
        END DO
        B(I)=TMP
      END DO

C-- solve Ux=y, replacing vector B
      DO I=N,1,-1
        TMP=B(I)
!hpf$   independent, reduction(TMP)
        DO J=I+1,N
          TMP=TMP-A(I,J)*B(J)
        END DO
        B(I)=TMP/A(I,I)
      END DO

      RETURN
      END
```