ハイパフォーマンス
コンピューティング 46－9
（1993. 4. 26）

拡張記憶の拡張主記憶としての仮想化

岡部 寿男　川端 英之　津田 孝夫
京都大学 工学部 情報工学教室

主記憶容量を超える多量のデータを扱う大規模数値計算をベクトル計算機上で高速実行するために、拡張記憶が用いられている。しかし、拡張記憶と主記憶との間のデータ転送の詳細をプログラムに記述することはユーザにとって負担となる。これに対し本稿では、巨大配列を扱うプログラムを拡張記憶を用いるように自動変換する方式を提案しており、現在、プログラム変換をソースレベルで行なうFORTRANプリプロセッサを開発している。LU分解のプログラムを変換して実測した結果、データ転送コストが総実行時間の3割から5割に抑えられ、大規模計算が実用的な性能で実行できることが示された。

VIRTUALIZATION OF SEMICONDUCTOR EXTENDED STORAGE AS EXTENDED MAIN MEMORY

Yasuo OKABE, Hideyuki KAWABATA and Takao TSUDA
Dept. of Information Science, Kyoto University
KYOTO 606-01, JAPAN

Extended Storage of vector supercomputers is secondary storage on which huge array that cannot be fitted in main memory is stored. As is usual with files on magnetic disks, it is not an easy job for users to write explicit data transfer between extended storage and main memory. The authors propose a method of automatic translation from an ordinary program which processes a huge array into what does I/Os of the array data from and to extended storage. A preprocessor which does the translation at Fortran source level is available. A program of large-scale LU factorization is experimentally performed in practical time, and the data transfer cost is not more than 50%.

1 はじめに

近年、ベクトル計算機の演算性能は単一 CPU で 10 GFLOPS に迫ろうとしている。演算性能の向上にともない、演算器と主記憶装置とのデータ転送速度の高さがこれまで以上に重要となる。しかし、高速アクセスの可能な主記憶装置は、技術的あるいはコスト的な要因からその実装可能容量に限界がある。主記憶に入り切らないデータを処理する場合には、従来二次記憶として磁気ディスク装置が用いられてきた。しかし、磁気ディスクは主記憶と比較してアクセス速度が非常に遅いため、両者の間でデータ転送が頻発する場合にはベクトル計算機の高速演算性能を生かすことができない。そこで、各社のベクトル計算機には拡張記憶と呼ばれる高速半導体補助記憶装置が装備されている [1][2]。

拡張記憶は、FORTRAN の READ/WRITE 入出力文を用いて磁気ディスク装置と同様に用いることができる。巨大配列データを拡張記憶に割り付け、計算の進行に応じて主記憶と拡張記憶との間でデータを転送することによって、主記憶だけでは実行することのできない計算が可能となる。データ転送回数や転送単位のサイズを最適化したアルゴリズムを設計した場合、データ転送時間をほぼ完全にベクトル演算の陰に隠すことが可能である [3]。しかし、主記憶と拡張記憶との間のデータ転送の詳細をプログラムに記述することは、ユーザにとって大きな負担となる。

これに対し我々は、主記憶のみを想定して記述されたプログラムを、拡張記憶に巨大配列データを割り付けデータ転送を適切に行ないながら計算を実行するように、自動的に変換する手法を提案している [4][5]。この方式は、データ転送命令をコンパイル時に目的コードに埋め込み、適切な時点でデータ転送を行ないながら計算が実行されるようにプログラムを変換する。現在、FORTRAN プログラムをソースレベルで変換するプリプロセッサの開発を進めている。これにより、ユーザは拡張記憶を意識することなく、主記憶に入り切らない巨大な配列データを扱う数値計算プログラムを実行させることができるようになる。これにより、拡張記憶の主記憶の延長としての一種の仮想化が実現されるといえる。

本稿では、様々な配列参照の形態に柔軟に対応するために配列を “格子状” に分割する方法を提案する。単純な “短冊状” の分割に比べ、配列がどの次元方向へベクトル参照されても参照される部分を含む分割単位のみが主記憶に読み込まれる点で効率が良い。さらに、格子状に分割することによりベクトル長が短くなることをさけるため、リストベクトルを用いたデータ参照方法を示す。さらにプログラム変換をソースレベルで行なう FORTRAN プリプロセッサの実現について述べる。

提案する手法の有効性を検証するために、LU 分解を例にとり上げ、HITAC S-3800/380 上で実行速度の測定を行なった。その結果 3000 × 3000 の行列について演算速度は約 1GFLOPS であった。これは同じプログラムを主記憶のみを用いて実行した場合と比較して 3 割～4 割の性能であり、データ転送時間が総実行時間（CPU 時間）に占める割合は、3 割～5 割に抑えられている。

以下、2 章で準備としてベクトル計算機の拡張記憶装置について述べる。3 章で我々の提案する拡張記憶の “仮想化” について述べ、4 章で仮想化のためのプログラム変換を FORTRAN ソースレベルで自動的に行なうプリプロセッサについて述べる。5 章では、HITAC S-3800/380 上での性能評価を示す。

2 拡張記憶装置

2.1 高速補助記憶装置としての拡張記憶

ベクトルスーパーコンピュータにおける磁気ディスク装置と主記憶との間の、容量およびアクセス速度の差を埋めるために、多くの機種では拡張記憶と呼ばれる半導体二次記憶装置を装備することができるようになっている [6][7]。拡張記憶はダイナミック RAM で構成され、主記憶に対してアクセス速度は数分の一程度であるが、容量は最大主記憶容量の数倍から数十倍が実装可能である。また、従来の磁気ディスク装置と比較して数百倍のアクセス速度を持つ。表 1に、現在の主要ベクトル計算機の拡張記憶のサポート方式を示す。 拡張記憶は、磁気ディスク装置と同様の外部記憶装置の一つとして用いることができる。ユーザは FORTRAN プログラム中で、通常のファイルに対するアクセスと同様に拡張記憶上のファイルに対して READ/WRITE 入出力文により入出力を行なうことができる [8]。

拡張記憶と主記憶との間でのデータ転送は、FORTRAN の READ/WRITE 入出力文で行なう。このデータ転送においては、ある程度の大きさをまとめて転送する方が高い転送速度が得られる。例えば S-820 では 8M バイト程度、S-3800 では 32M バイト程度のレコード長によるデータ転送で、最大転送速度の 8～9 割の速度に達する。このため、拡張記憶に割り付ける配列はある程度の

表 1: 各社の拡張記憶の比較

機種・名称	最大容量 最大転送速度	主記憶 容量	転送方式	サポート方式
HITAC S-3800/380 拡張記憶 (ES)	16GB 4GB/sec	2048MB	SPU による 同期/非同期転送	read/write
HITAC S-820/80 拡張記憶 (ES)	12GB 2GB/sec	512MB	SPU による 同期/非同期転送	同上
NEC SX-3/44 拡張記憶 (ES)	16GB 3GB/sec	2048M	SPU による 同期転送	同上
FACOM VP2600/20 システム記憶	8GB —	2048MB	同上	SSU 配列 read/write
CRAY Y/MP8/8 半導体記憶 (SSD)	4GB 2.5GB/sec	1024MB	同上	SSD 配列 read/write
FACOM VP400E ベクトル記憶	0.67GB 10GB/sec	256MB	ベクトルレジスタへ 直接転送	コンパイラが 自動割り付け
IBM ES/3090VF 拡張記憶 (ES)	2GB 0.1GB/sec	512MB	SPU による 同期/非同期転送	仮想記憶

SPU : Scalar Processing Unit

大きさのレコードに分割する必要がある。その際小量の配列要素に対する参照が必要な場合でも、それらを含むレコード中のデータ全てをまとめて転送しなくてはならない。

2.2 拡張記憶配列としての利用

拡張記憶上のファイルデータの扱いは従来の磁気ディスク装置などの補助記憶装置上のデータと同様で、主記憶のデータのように直接参照することはできない。このため、計算実行中に主記憶と拡張記憶との間で明示的にデータ転送を行なうようプログラム中に明示的に記述する必要がある。これはユーザにとっては大きな負担である。

我々は、「拡張記憶の仮想化」と呼ぶ拡張記憶や主記憶容量について意識することなく記述したプログラムを拡張記憶を利用するように自動的に変換する方式を提案している [4]。さらに、変換を FORTRAN ソースレベルで実現するプリプロセッサを開発し、その有効性を示した [3]。本稿では、[3] の方式の問題点であった一つの行列が同時に複数の次元方向に沿ってベクトル参照される場合の性能低下をを解決するため、配列の「格子状分割」とリストベクトルアクセスを組み合わせる方法について示す。

拡張記憶と主記憶との間のデータ転送を自動化する他の方式として、富士通 VP2000 シリーズのシステム記憶を用いた SSU 配列と呼ばれるものがある [9] (Cray Y/MP における SSD 配列と呼ばれるものも同様の方式である)。これは、拡張記憶に割り付ける巨大配列を FORTRAN プログラム中で COMMON ブロックにまとめ、ジョブ制御文により COMMON ブロック名を指定することにより、プログラム中で通常の配列とほぼ同様に扱うことができるものである。拡張記憶との間のデータ転送命令はコンパイル時にオブジェクトに埋め込まれる。配列上の連続する要素に対するベクトル参照の場合には、連続転送と呼ばれる極めて高速なデータ転送が行なわれる。しかし、SSU 配列の使用にはさまざまな制限が伴うだけでなく、配列要素を一定間隔でベクトル参照する場合や、SSU 配列要素をスカラデータとして参照する場合には、ディスタンス転送と呼ばれる より遅い転送方式が用いられ、データ転送に演算時間の数十倍をも要することもある [9]。このように、SSU 配列の使用には格別の注意を払う必要があり、少なくとも現時点では、主記憶の延長として自然に用いることができるとは言い難い。

3 拡張記憶の仮想化

提案する拡張記憶の仮想化方式では、計算は、拡張記憶中に巨大配列を割り付けておき、巨大配列の参照時に拡張記憶上のデータを主記憶上の作業領域に読み込み、参照し、書き戻す必要があれば拡張記憶に転送する、という流れで実行される。

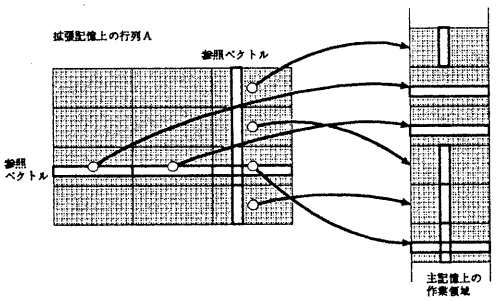

図 1: フレームとブロックの対応

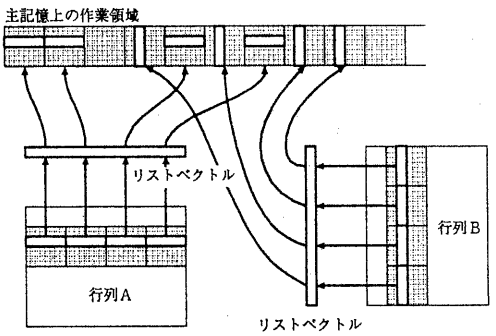

図 2: 作業領域の共有

3.1 配列の分割方式

配列の参照においては、ある一つの次元の方向に沿って端から端まで連続した要素のアクセスが行なわれることが多いことに注目すると、配列をある次元の方向に沿って分割し（二次元であれば短冊状の分割）、その次元の方向に沿った連続参照に先立ってまとめてデータを転送することが考えられる [5]。これにより、例えば2次元配列のある列のデータがループ中で参照されることがあらかじめ分かれば、分割単位を一つ主記憶中に読み込むことで、内側ループ中で連続して参照することができる。しかし、分割の方向と直行する別の次元方向に沿った配列データへの参照の際には、参照されないデータの転送量が増大する。

我々は、巨大配列を、各次元方向に沿って（二次元であれば格子状に）分割し、その分割単位を転送単位とすることを提案している [10]。ここではこの分割単位をブロックと呼び、この分割方法を格子状分割と呼ぶ。巨大配列に対してある次元の方向に連続した参照がある場合、それら一連の配列データを含むだけのブロックをまとめて主記憶に転送する。これにより、参照する次元の方向による実行速度の偏りが生じることがなくなる。複数の参照式において同時に参照されるブロックは重複して転送する必要はない (図 1)。配列の単一要素に対する参照の場合はその要素を含む単一ブロックのみを主記憶に転送すればよい。このように、巨大配列を格子状に分割する方法は、短冊状の分割に比べて、様々な次元の方向に沿って巨大配列を参照する時のデータ転送の際に生じる無駄が少ない。

3.2 作業領域上のデータの参照

配列を分割して、ブロック毎に主記憶に転送した場合、本来等間隔に配置されるべき配列要素が、作業領域上では区分的にしか等間隔に配置されない (図 2)。要素が主記憶上で等間隔に配置されていないデータへの一連のアクセスはそのままではベクトル化できないので、単純な格子状分割では配列アクセス時のベクトル長が短くなってしまう。これに対し、リストベクトルによる間接参照を用いることにすれば、大きいベクトル長によるベクトル参照を可能にすることができる (図 1、図 2)。

リストベクトルによる間接参照を行なう場合リストベクトルを設定するためのオーバヘッドが必要である。そこでリストベクトルの設定については、ループ中での配列参照がどの次元の方向に沿って行なわれるかを把握しておき、その次元の添字式の値の差をスカラ変数に保持しておくことで、ループの繰り返し毎のリストベクトルの再設定を回避することができる。

3.3 データ転送制御

巨大配列に対して格子状分割を施し拡張記憶に割り付けるとともに、主記憶上にはデータを転送して参照するための作業領域を設ける。作業領域はフレームと呼ぶ区画に分割し、各ブロックを任意のフレームに転送することができるようにする (図 2)。

作業領域、すなわちフレーム全体は、空きフレームリストと呼ぶリスト構造で管理する。拡張記憶からブロックを転送する際には、空きフレームリストからフレームを取り外す。値の参照の終了後には、そのフレーム中のデータはそのままにしてフレームをリストにつなぎ、空のフレームとみなす。そのフレーム中のデータを別の時

点で参照する必要が生じた場合には、フレームをリストから外すだけでよい。そのフレームに新たなデータを読み込むときに、必要であればそのフレーム中の元のデータを拡張記憶に書き戻す。複数の配列参照式が一つのブロックを同時に参照する場合には一つのフレーム中のデータを共用する。

フレームは、書き戻す必要のあるデータが入ったものとそうでないものとを区別し、フレームをリストからとり外す際に、書き戻す必要のないデータが入っていたフレームから優先的に取り外すようにすることにすれば、拡張記憶、主記憶間のデータ転送量を減じることができる。

3.4 データ転送のタイミング

配列データの転送は、配列の添字式に含まれる変数のうち一つを除いた全てのものが定義された時点で、参照される次元の方向にまとめて行なうことができる。プログラムの制御構造を変更せず、もとのプログラム中にデータ転送のための手続き呼びだしを挿入する形態を考えると、例えば多重ループの最内側で頻繁に転送制御ルーチンが呼び出されるような状況を避けることが望ましい。

DOループの制御変数を添字式に用いて、ある次元の方向に沿ってループ中で配列を参照する場合には、制御変数の上下限式により最内側ループの繰り返し中で必要となるブロックを特定することができる。それらのデータを、最内側ループに入る前の時点でまとめて転送すればよい (図3)。さらに、一回のデータ転送ルーチン呼び出しによって最内側ループの複数回転分のデータを転送されることになるので、外側ループで実際にデータ転送ルーチンを呼び出す回数はループ繰り返し回数に比べずっと少ない。

```
      DO 10 I=1,N              この時点で、配列Aの二次元目
        ★              ← の方向に 1*3-2+I から N*3-2+I
      DO 10 J=1,N              までの一連の要素を含むブロッ
      X=A(I+2,J*3-2+I)+X クを作業領域に転送することが
10    CONTINUE                 できる。
```

図 3: データ転送ルーチンの挿入 (2次元目の添字式が先に確定する場合)

配列の参照が一つの次元の方向に沿ったものでない場合には、次のような方法をとる。

```
      DO 10 I=2, N
        DO 12 J=0,N/2-1
          ★                        ← データ転送
12      CONTINUE
        DO 11 J=0,N/2-1
          A(J+I,2*J+1) = B(J+I,J+1)
11      CONTINUE
10    CONTINUE
```

図 4: データ転送ルーチンの挿入 (ループ分割による方法)

- 配列の全ての添字式が制御変数の線形式の場合、制御変数の上下限式を用いて、内側ループの一回の実行で必要となるブロックを計算し、外側でまとめて転送する。
- 添字式が制御変数の非線形式の場合、ループを分割し、一つ目のループに置いて最内側ループで参照されるブロックを調べてデータ転送およびリストベクトルの生成を行ない、二つ目のループで実際に配列参照を行なう (図4)。

4 プログラム自動変換の実現

4.1 ソースレベルでの自動変換

提案する手法の実現として、拡張記憶を意識することなく記述されたFORTRANプログラムを、拡張記憶を利用するようにソースレベルで自動的に変換するプリプロセッサを開発中である。

プリプロセッサは、以下のようなプログラム変換を行なう。

1. 巨大配列の宣言文を検出し、拡張記憶に割り付けるように宣言を変更する。
2. 巨大配列の参照文を検出し、主記憶上に設ける作業領域への参照に置き換える。ここで、リストベクトルを用いた間接参照を利用する。
3. 巨大配列の参照文から、データ転送を行なうべき位置を決定し、データ転送ルーチンの呼びだし文、あるいはその他の補助関数の呼びだし文を挿入する。また、それらの補助関数のパラメータを設定し、ソースプログラムに付加して出力する。

ユーザは、拡張記憶に割り付けるべき巨大配列の指定、及び、配列の大きさなどを考慮して作業領域の宣言を設定するための各種のパラメータを、プリプロセッサ指示行 (特殊な形式のコメント文) を用いて記述できる。指示行以外の部分では全く拡張記憶を意識する必要はない。

4.2 プログラム変換の例

プリプロセッサによる変換の例として、入力プログラム例を図5に、変換後のプログラムを図6に示す。

```
1         DO 10 I=I0,I1
2         DO 10 K=K0,K1
3         DO 10 J=J0,J1
4             A(K,8)=1.0D0
5             B(I,J)=3.0D0
6    10   CONTINUE
```

図 5: 変換前のプログラム

```
 1        NXTBLK(2)=(I0-1)/BLWD(B,1)
 2        CURBLK(2)=-1
 3        CALL XXCL(A,1,1,8,1,.TRUE.,1,K0,K1)
 4        DO 10 I=I0,I1
 5        IF(NXTBLK(2).NE.CURBLK(2)) THEN
 6           CALL XXCL(B,2,I,1,1,.TRUE.,2,J0,J1)
 7           CURBLK(2) = NXTBLK(2)
 8        ELSE
 9           XXDIFF(2)= I-LASTI1(2)
10        ENDIF
11        NXTBLK(2) = ((I+(1))-1)/BLWD(B,1)
12        DO 10 K=K0,K1
13   *VOPTION INDEP(XXTT), VIST
14        DO 10 J=J0,J1
15           XXTT(XXI(1,K)+XXDIFF(1))=1.0D0
16           XXTT(XXI(2,J)+XXDIFF(2))=3.0D0
17    10   CONTINUE
```

図 6: 変換後のプログラム

図5の4,5行目の拡張記憶に割り付ける配列の参照文が、作業領域（一次元配列 XXTT）の参照文に置換され（図6、15、16行目）ている。配列Aについてははじめから8列目の参照であることが分かるため、最外側のループの外でデータ転送を行なうことが可能である (図6、3行目)。配列Bについては、最後に定まる添字式が二次元目のJであり、図5の1行目を過ぎた時点で配列のI行目に対するデータ転送を行なうことができる (図6、1,2,5～11行目)。とくに配列Bは、制御変数Iが添字式に含まれるため、最外側のループ中でIの値によりデータ転送の必要性の有無を判断することができる。実際にデータ転送が必要な時点になってからデータ転送ルーチンを呼び出すということが可能である（図6、6行目)。

出力されるプログラムはFORTRAN77であり、プリプロセッサによる変換の修正や改良などを手作業で行なうことも容易である。

5 評価

本稿の手法の有効性を検証するために、LU分解を例にとり上げ実行速度の測定を行なった。実行は、東京大学大型計算機センターのベクトル計算機 HITAC S-3800/380 上で、自動ベクトル化コンパイラ FORT77/HAP を用いて行なった (表2)。実行結果は全て単一 CPU によるものである。CPU 時間および VPU 時間はそれぞれスカラプロセッサユニット、ベクトルプロセッサユニットの動作時間を表し、CPU 時間は VPU 時間を包含する。また、データ転送時間は全て CPU 時間に包含される。

LU分解プログラムのアルゴリズムは2行2列同時クラウト法 [4] で、もともとベクトル実行に対し親和性の高いものである。行列のサイズは 3000 × 3000 および 4000 × 4000 で、ブロック長 (＝レコード長) はそれぞれ 301 × 301 および 401 × 401 である。フレーム数は作業領域の広さを表す。表2においては、フレーム数50で作業領域の大きさが巨大配列の大きさの半分となる。

演算速度は、主記憶のみを用いて同様の計算を行なった場合と比較して3割～4割、およそ1GFLOPSを達成することができ、データ転送時間が総実行時間（CPU時間）に占める割合は、3割～5割に抑えられている。表に示されるように、高いベクトル化率を維持できていることが分かる。提案するプログラム変換によって、もとのプログラムのベクトル化適応性を損なうことなく変換が可能となっている。また作業領域が広いと、拡張記憶に対するキャッシュとしての効果が現れ、必要となるデータ転送量が少なくなる。

6 おわりに

本稿では、拡張記憶を意識せずに記述したプログラムを、拡張記憶を利用してデータ転送制御を行なうように

表 2: LU 分解実行時間測定結果

行列長：3000 × 3000　ブロック長：301 × 301

フレーム個	CPU 時間 sec			転送速度 MB/sec	演算速度 Mflops
		VPU	転送		
(a) 35	24.0	11.4	11.3	2890	751
(b) 40	21.8	11.3	7.89	3049	825
(b) 50	18.4	11.3	4.71	3055	981
(a) 50	19.1	11.4	4.88	3056	941
(c) 50	23.5	11.3	4.88	3053	767
(d) —	8.00	6.89	—	—	2250

行列長：4000 × 4000　ブロック長：401 × 401

フレーム個	CPU 時間 sec			転送速度 MB/sec	演算速度 Mflops
		VPU	転送		
(a) 40	48.6	25.8	19.2	3240	877
(d) —	16.8	14.7	—	—	2550

全て倍精度実数による演算。CPU 時間は総実行時間を表す。VPU 時間・転送時間は CPU 時間に含まれる。(a):通常実行, (b):作業領域開放ルーチンを呼ばない, (c):データ転送ルーチンを常に呼び出す従来方式, (d):主記憶のみでの実行

1993 年 4 月 東京大学大型計算機センター
HITAC S-3800/480 にて測定（シングル CPU）

自動的に変換する手法について述べた。その結果、行列積計算のような実用的なプログラムにおいて、制御構造や演算の順序を変更することなくソースレベルで変換を行なうだけでも転送時間を総実行時間の 3 割～4 割に抑えることができることが示された。

今後の課題としては、まず配列の分割において短冊状分割と格子状分割を融合させることが考えられる。また、ループアンローリングの自動化など、元のプログラムの制御構造の変更を含めたデータ転送最適化の実現が望まれる

参考文献

[1] T. Okada, S. Nagashima and S. Kawabe: Hitachi Supercomputer S-810 Array Processor System, SUPERCOMPUTERS Class IV Systems, Hardware and Software, pp.113-136 (Elsevier Science Publishers, 1986).

[2] T. Watanabe, H. Katayama and A. Iwaya: Introduction of NEC Supercomputer SX System, Ibid., pp.153–167.

[3] Tsuda, T. and Y. Okabe : Use of Semiconductor Extended Storage as Extended Main Storage for Large-Scale Supercomputing, *Proc. 2nd International Conference on Supercomputing*, pp. 176–183 (May 1987).

[4] 津田 孝夫: 数値処理プログラミング, 岩波講座ソフトウェア科学 9 (岩波書店, 1988).

[5] 岡部, 大西, 津田: スーパーコンピュータ用拡張記憶の拡張主記憶としての仮想化, 平成 2 年電気関係学会関西支部連合大会講演論文集, S56 (1990).

[6] 島崎 眞昭: スーパーコンピュータとプログラミング (共立出版, 1989).

[7] シドニーファンバック編, 長島重夫訳: スーパーコンピュータ—超高速計算のためのハードウェアとソフトウェアの全て— (パーソナルメディア株式会社, 1988).

[8] 日立製作所編: 最適化 FORTRAN77 OFORT77 E2, HAP FORTRAN77 使用の手引 (日立製作所, 1989).

[9] 永井 亨: ベクトル計算機入門 (3) —SSU, 名古屋大学大型計算機センターニュース, Vol. 23, No. 3, pp. 205–213 (Aug. 1992).

[10] 岡部, 川端, 津田: ベクトル計算機における拡張記憶の拡張主記憶としての仮想化—リストベクトルの利用による配列の格子状分割—, 日本ソフトウェア科学会第 9 会大会論文集, pp.481–484 (1992).