数理解析研究所講究録
547巻 1985年 156-172



# スーパーコンピュータとそのソフトウェア

日立中研　安村通晃（Michiaki　Yasumura）

## はじめに

大規模科学技術計算を高速に実行するコンピュータシステムとして、スーパーコンピュータの需要が近年高まってきている。この増大する需要に答えるものとして、我が国内でもスーパーコンピュータが相次いで開発され、実用目的で使用されるようになってきた。また、国家的プロジェクトとしての必要性も認識され、通産省の大型プロジェクトの一つとして、スーパーコンピュータの研究開発が進められている。

一口にスーパーコンピュータと行っても、その構成する要素は広く、素子・実装・アーキテクチャといったハードウェア面から、ベクトルコンパイラ・数値計算ライブラリ・各種プログラミングツール等のソフトウェア面、および、さまざまな応用プログラムとそのアルゴリズムの研究といったアプリケーション面まで、広い範囲を含んでいる。

本論文では、スーパーコンピュータの高速性の原理的説明という観点からアーキテクチャを、また、並列性の自動検出とプログラムの自動変換という観点からベクトルコンパイラを、それぞれ取り上げ述べることとする。

## スーパーコンピュータとは

本題に入る前に、スーパーコンピュータの定義を述べておく。

ここで考えているスーパーコンピュータとは、

（１）汎用超大型機よりはやい計算機であって、

（２）特定の応用に限定された専用機ではなく、かつ

（３）経済性、

を兼ね備えたものをいう。

第１の点から、スーパーコンピュータは、高速のスカラプロセッサを前提とすることがいえる。このことは、ユーザの立場からすれば、どんなプログラムをもってきても、スーパーコンピュータでは、汎用超大型機以上の性能が保証されている、と思ってよい。また、第２の点からスーパーコンピュータは、特定の応用分野、たとえば、信号処理とか、画像処理とかに限定されることなく、広く、科学技術計算一般に適用できるものでなければならないことがいえる。さらに、第３の点として、総価格だけでなく、価格／性能比の面でも、従来の計算機システムに比べて遜色がなく、従って実用的にも十分引き合うものであることがいえる。

次に、スーパーコンピュータの実現方式を、アーキテクチャから見て分類すると次の３つの方式がある。

（１）ベクトルプロセッサ方式

（２）アレイプロセッサ方式

（３）マルチプロセッサ方式

まず、現在最も有力とされ普及しているのが、（１）のパイプラインを基本としたベクトルプロセッサの方式である。また、（２）のアレイプロセッサ方式は、複数のプロセッサが同一の命令列を同期をとりつつ並列に実行する方式であり、ＳＩＭＤ方式[*1)]ともいわれる。（３）のマルチプロセッサ方式は、複数のプロセッサが異なる命令列を非同期的に実行するもので、ＭＩＭＤ方式[*2)]ともいわれる。

---

*1)SIMD:Single Instruction Stream Multiple Data Streams

*2)MIMD:Multiple Instruction Streams Multiple Data Streams

以上述べてきた3つの方式は、いずれも、命令列が計算の流れを制御するフォンノイマン型の計算機方式の延長線上に位置つけられる方式であるが、これらの方式の他に、最近では、

（4）データフロープロセッサ方式

がスーパーコンピュータの新しい方式として、提案されている。

データフロープロセッサは、プロセッサが複数あり、異なる命令が非同期的に実行される、という点ではマルチプロセッサと同じであるが、同期制御をデータが決めるという点でマルチプロセッサとは異なる。まだ、データフロープロセッサは、性能面で、ベクトルプロセッサより速いものが作られていないが、応用面での広さでは、原理的にはベクトルプロセッサなどよりも広い。

現在の多くのスーパーコンピュータの主流は、ベクトルプロセッサ方式であり、マルチプロセッサ方式は、小型の実験システムは作られていても、実用的なものは現れていない。これは、マルチプロセッサ方式に次のような問題点があるためである。

(i)メモリの問題：マルチプロセッサのメモリは、共有メモリにする方式と、ローカルメモリにする方式と2通りがあるが、共有メモリにすると、異なるプロセッサ間のメモリアクセスに競合が生じるし、ローカルメモリにすると、システム全体のメモリの量が増えるという問題と、グローバルデータのアクセス方法という問題が生じる。

(ii)同期のオーバヘッドの問題：マルチプロセッサは非同期的に動くため、その同期のオーバヘッドは避けられない。これは、マルチプロセッサシステムを汎用的に作れば作る程大きくなる傾向にある。

(iii)コミュニケーションの問題：異なるプロセッサ間の通信法として、メモリを用いる方法、パケットネットワークを用いる方法、クロスバースイッチを用いる方法の3通りの結合方法があるが、特に後2者の場合、伝送量と伝送線の量が問題となる。同時に、結合のトポロジーも問題であり、応用によって、N次元立方体結合とか、トリー状結合とか、最適のものが変わりうるので、汎用的にすると、トポロジーそのものも可変にする必要性も出てくる。

(iv)プロセッサ利用率の問題：応用のもつ問題のサイズがプロセッサ台数に丁度合わないときは、プロセッサのうち何台かはあそんでしまうか、応用プログラムをうまく分割してやるか、いずれかの問題が生じる。

(v)ソフトウェアの問題：マルチプロセッサのために、既存のプログラムから完全に自動化するようなソフトウェアはまだできていないし、また、できたプログラムのデバッグも大きな問題である。

このようなマルチプロセッサの問題は、部分的には、アレイプロセッサに、または、データフロープロセッサにもあてはまる。

実用的なスーパーコンピュータの性能がどの位であるかを、図1に示す。[1] 図1の横軸は年代であり、縦軸は、スーパーコンピュータのピーク性能である。（実効的な性能は一般にこのピーク性能より低いが、この点については、

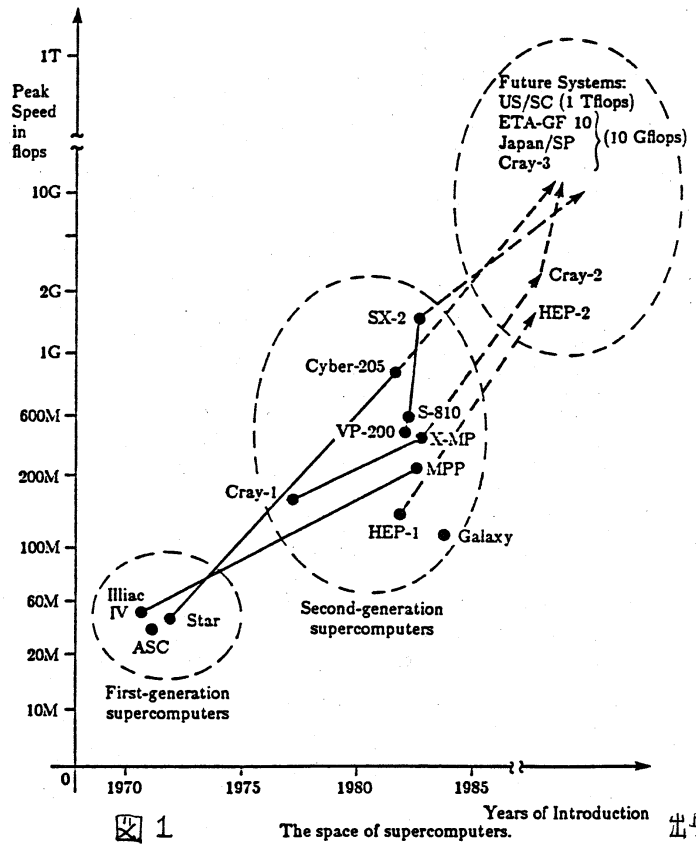



図1 The space of supercomputers. 出典1) by K. Hwang

後で触れる。）性能の単位は、FLOPS*)で、現在最も速い汎用超大型機でさえ、高々10MFLOPS程度である。これに対して、1970年代前半に作られた、第1世代のスーパーコンピュータは、約数十MFLOPSのピーク性能をもつ。1970年代後半から1980年代前半に作られた第2世代のスーパーコンピュータは、100MFLOPSから約1GFLOPSのピーク性能をもつ。第1世代のスーパーコンピュータの代表格であるIlliac IVは、アレイプロセッサ方式であったが、第2世代の代表格であるCray-1はベクトルプロセッサ方式である。また、最近相次いで発表された国産のスーパーコンピュータ（富士通のVP、日立のS-810、日電のSX）は、いずれもベクトルプロセッサ方式である。Cray-1の拡張版である、Cray X-MPは、ベクトルプロセッサを基本とした、2～4台マルチプロセッサシステムである。Denelcor社のHEPがマルチプロセッサ方式である点が、やや変わっている。今後しばらくは、ベクトルプロセッサ方式が主流の時代が続くと思われる。通産省の大型プロジェクトでは、1990年代の始めに、10GFLOPSのスーパーコンピュータを目指しているが、その内容はあまり明らかになっていない。以下では、スーパーコンピュータとして特に断らないかぎり、ベクトルプロセッサとして話を進める。

*）Floating Oerations Per Second（一秒当たりの浮動小数点演算数）

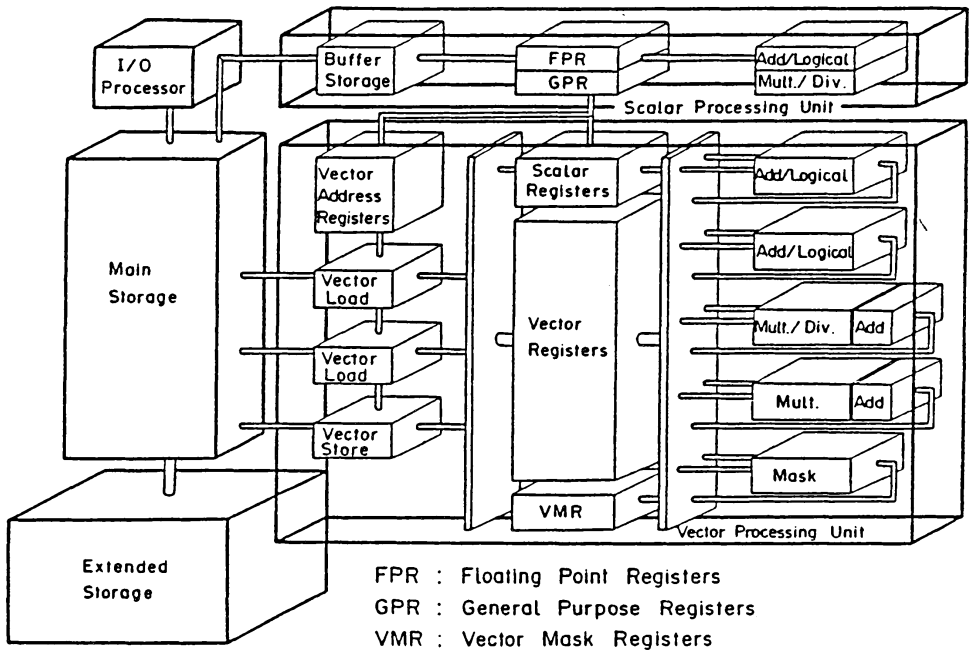
スーパーコンピュータのアーキテクチャ

スーパーコンピュータの高速性を、アーキテクチャの面から述べると、高速の演算機構と、そのための高速のデータの供給の2点であることが分かる。

高速の演算機構としては、パイプライン方式の演算器により、ベクトルの各要素を、1要素当たり1クロックのピッチで演算する方式を基本としている。現在のスーパーコンピュータでは、1クロックは約10nsec程度である。パイプライン方式とは、原理的には近代産業の基本である流れ作業と同じ方式であり、最初の要素が処理を完了するまでは、数クロック要するが、各要素が処理されるピッチは1クロックになるのである。このような高速の演算パイプラインを複数本持つことにより、最近のスーパーコンピュータは、さらに性能を稼いでいる。

しかし、この高性能の演算パイプラインに十分間に合うだけのデータ供給能力をスーパーコンピュータが供えていないと、トータルとしての高性能は実現できない。そのためには、まず、高いデータ供給能力を持つ高速のメモリが必要である。メモリは、通常複数のバンクと呼ばれる独立した部分に分けられるが、このバンクの数が汎用機では十程度であるのに対して、数十から百以上まで細かく分割されている。さらに、汎用機では、メモリへのフェッチ・ストアは通常1クロックに1個と限られているが、最近のスーパーコンピュータでは、複数のフェッチ・ストアを1クロック中に処理する能力も持っている。このような、メモリの高性能化だけでは不十分であり、最近のスーパーコンピュータは計算の途中結果を保持するためのベクトルレジスタと呼ばれる、プログラムで参照可能な高速の内部バッファを持っている。さらに、外界からのデータ供給能力を増やすために、入出力のデータ転送速度を増やす工夫も行っている。たとえば、複数のディスクに並列に読み書きする機能とか、ディスクとは別の半導体で作られた2次記憶を設けるとか、などである。

図2 S-810 Block Diagram (model 10)



最近のスーパーコンピュータのアーキテクチャの実例を図2に示す。これは、日立のS-810/10の例であるが、国産のスーパーコンピュータのアーキテクチャはいずれもほぼこれと同じ構成をしている。汎用高速のスカラプロセッサに、ベクトルレジスタと複数パイプライン演算器を中心としたベクトルプロセッサが付いている。ベクトルレジスタと高速のメモリとの間には、複数本のメモリフェッチ・ストアの口が付いている。また、後で述べる条件文を処理するための、マスク演算器やマスクレジスタと呼ばれるものも含まれている。主メモリには、大容量の拡張記憶を附けることもできる。

現在のスーパーコンピュータのアーキテクチャの基本は、パイプライン演算器であるが、さらに高速性を得るために、データフロー的なアプローチおよび、アレイプロセッサ的なアプローチも併用して並列性を増している。図3（a）は、複数ベクトルの実行状況（横軸は時間）を示す。ベクトルの各要素は、先頭の要素が終わりしだい、その演算結果を使う次の演算へと引き継がれていく。これを継続という。さらに、まったく独立なベクトルデータは、先の演算を追い越して実行されることもある。これらは、複数の演算器をデータフロー的に実行させている結果可能になるわけである。当然、同一の演算器を使用する場合は、前の演算がすべて完了するまで待たされることもある。図3（b）は、演算器のアレイプロセッサ的な実行状況を示す。図の例では、ベクトルの偶奇の要素を、二つの演算器に振り分けて実行させる要素間並列方式により、倍の性能向上を得ることができることを示している。また、内積や総和等の命令も、一要素ずつ演算していくと遅くなるが、部分和を取るトーナメント方式により、高速化を実現している。

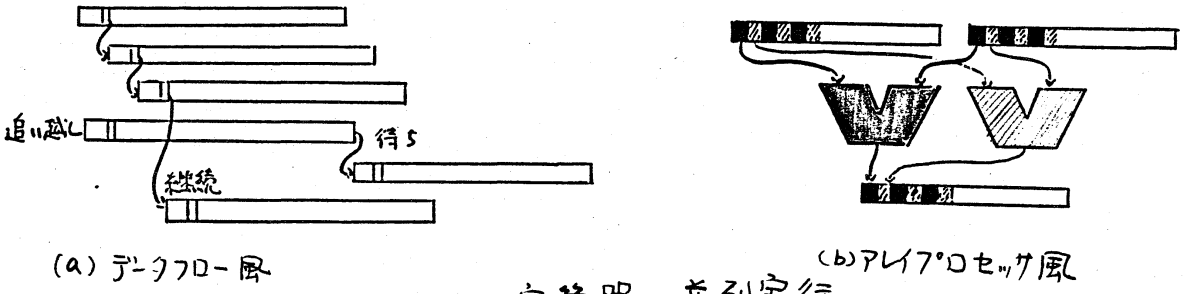



図3 演算器の並列実行

ベクトルコンパイラ

以上述べてきた、スーパーコンピュータのアーキテクチャを有機的に活かすものとして、スーパーコンピュータのソフトウェア類がある。これらのソフトウェアとしては、既存のプログラムから並列性を検出してベクトルオブジェクトを自動生成するベクトルコンパイラとベクトル化のチューニング等を行う各種のツール、各スーパーコンピュータ向けに作られた数値計算パッケージ等が、スーパーコンピュータのシステムの一部として提供されている。

このうち、自動ベクトル化の中心であるベクトルコンパイラについて以下では述べる。現在のスーパーコンピュータのベクトルコンパイラのほとんどが、FORTRAN77用であり、また数学関数（ｅｘｐや１ｏｇなど）もベクトル命令として展開している。ベクトルコンパイラをFORTRAN以外の言語に適用することは比較的易しいことであるが、現在はFORTRANしかないと言うのは、単にユーザニーズの結果に過ぎない。（以下の例でも、FORTRANのプログラムを例に使う）

ベクトルコンパイラは、Ｋｎｕｔｈが明示的に述べた経験則、すなわち、プログラムのごくわずかの部分が実行時間の大部分を占める、という事実に基づいている。FORTRANプログラムの場合、図４のようなＤＯループの実行時間が、全プログラム実行時間の多くの時間を占める。さて、このＤＯループにおいて、通常の実行順序は、各インデックス値について、すべての演算を実行していく、という順序である。この実行形態をスカラによる実行形態

```
FORTRAN DO loop          vector mode

    DO 10 i=1,N
    A(i)=B(i)+C(i)   => (Ai=Bi+Ci, i=1,N)
10  D(i)=A(i)*E(i)      (Di=Ai*Ei, i=1,N)
```

図４　Loop distribution(vectorization)

と呼ぶ。これに対して、ベクトルでの実行形態とは、各演算をすべてのインデックス値について、まず実行して行くことを言う。スカラで，図の縦方向の実行形態が、ベクトルでは横方向の実行形態に変っている。これは一種のプログラム変換であり、この変換をベクトル化と呼ぶ。見方を変えれば、スカラでのＤＯループは大きなループであったが、ベクトル化後は、各演算毎の小さなループに分配されることから、ベクトル化をループ分配(loop distribution)と呼ぶこともある。

ベクトル化は、一種のプログラム変換であるから、変換の前後でプログラムの意味が変ってはならない。すなわち、プログラムの正しさが保証されなければならない。このベクトル化の保証を行うのが、ベクトルコンパイラの基本的な役割である。このことを、図５の例を用いて説明する。図５のＤＯ１０およびＤＯ２０はベクトル化を行うとスカラのときとは結果が異なりしたがってベクトル化できないケースである。一方、ＤＯ１５とＤＯ２５は、ベクトル化後も結果が変わらずしたがってベクトル化が可能な例である。

*)ここで言うプログラムの意味とは、最終的な結果に対してだけの弱い意味で使っている。

```
   DO 10 i=1,N                 DO 15 i=1,N
   A(i-1)=B(i)+C(i)            A(i+1)=B(i)+C(i)
10 B(i)=A(i)*D(i)           15 B(i)=A(i)*D(i)

   DO 20 i=1,N                 DO 25 i=1,N
   A(i)=S*B(i)                 S=A(i)*B(i)
20 S=C(i)+A(i)              25 B(i)=S+C(i)

   Unsuitable case             Suitable case
```

図5 Examples of data dependency

配列に対しては、添字の”大小関係”が問題であり、変数に関しては、定義と参照の順序関係が問題である。この関係をまとめたものが、表1であり、高貫ら[5]は、配列と変数のベクトル化を、このようなデータ依存関係に分類してベクトル化を行うアルゴリズムを示した。さらに、安村等[6]は、このアルゴリズムに、ループ分割による部分のベクトル化[8]や、文交換法などのプログラム変換法を、併用することによりベクトル化のアルゴリズムを広げている。Illinois大のＫｕｃｋ等[9]はこれとは別に、グラフを用いたデータ依存解析法を示している。図6に示すように、彼らはデータの依存関係を、データ依存、反依存、出力依存、制御依存の4種類に分けて依存グラフを作り、そのグラフ上での依存関係を減らすという方式のベクトル化のアルゴリズムを提唱している。

表1　Basic data dependency relations

| dependency | array | variable | vec |
|---|---|---|---|
| independent | Fi∩Fj=0 | ref only | o |
| dependent: suitable | Fi ≧ Fj | def->ref | o |
| dependent: special | VIP, etc. | VIP, etc. | o |
| dependent: unsuitable | Fi ≦ Fj | ref->def | x |
| unknown | Fi ? Fj | - | x |

≦ means "less than" relation in ordered set of subscripts
Fi, Fj are the set of subscript value of an array
VIP: Vector Inner Product

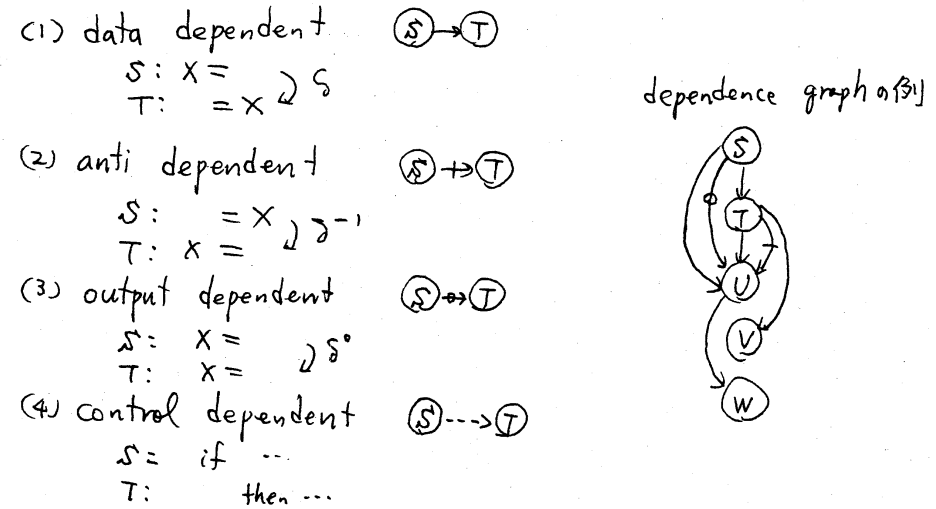
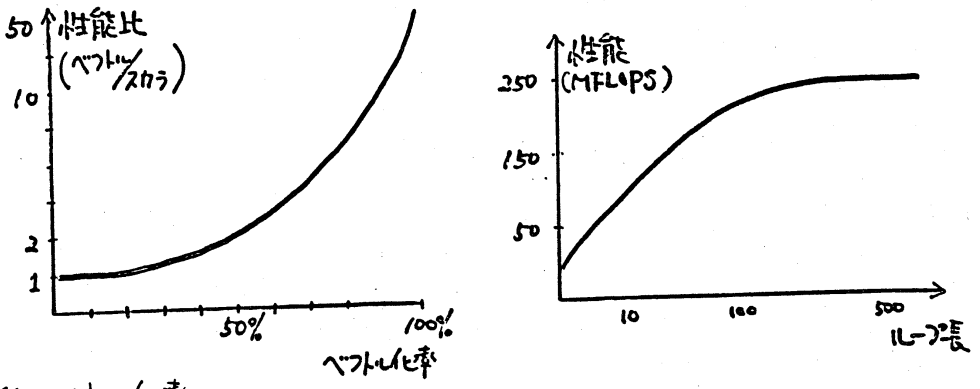



図6　Kuckらのデータ依存解析法

条件文のベクトル化

すでに述べたとおり、スーパーコンピュータのピーク性能と実効的な性能とは必ずしも一致しない。スーパーコンピュータの実効性能を決めるものにはさまざまな要因があるが、その主なものを図7に示す。

まず、第1にベクトル化率であるが50%しかないと、スーパーコンピュータの性能はスカラプロセッサの高々2倍までしかいかない。ベクトル化率が90%になってやっと高々10倍の性能比である。従って、ベクトルコンパイラににとって、ベクトル化率を向上させることが（ベクトル化の基本アルゴリズムは別として）、最も重要な課題である。次に、スーパーコンピュータでは、演算のピッチは十分短くても、一要素を完全に実行しおわるまでは、スカラプロセッサと同等以上の時間が一般にはかかる。従って、次に大切なのは、演算対象のベクトルの長さ、すなわち、ループ長である。通常、現在のスーパーコンピュータでは、ループ長が数百で性能が飽和する。飽和性能の半分の性能に達するループ長を $n_{\frac{1}{2}}$ とし、これを性能評価尺度の一つとする考え方[12]もある。そのほか、スーパーコンピュータの実効性能を左右するものとしては、ベクトル要素のアドレスが連続か非連続かというようなメモリアクセスに対する問題や、演算器の利用率といった、ハードウェアリソースとの関係で決まる問題もある。これらの項目は、応用プログラムの特性と、コンパイラの機能・能力とに関係している。




(1) ベクトル化率

$$\gamma = \frac{T_V(\text{ベクトル化可能部実行時間})}{T(\text{全スカラ実行時間})}$$

(2) ループ長

(3) その他
・アドレス連続/非連続
・演算器利用率, 等

図7 実効性能決定要因

ベクトル化率を上げる上で、まず重要なのが（配列の添字解析の問題を除けば）、文の種類である。なかでも、条件文に対するベクトル化が最初の課題である。

条件文のベクトル化には、マスク演算と呼ぶハードウェア機構を通常は用いる。これは、スカラにおける、条件毎の制御の移動を、マスクベクトルと呼ばれる、1ビット毎の制御データに変更したものを演算の制御に用いる方式であり、制御フローをデータフロールに変換した方式とも見ることができる。（図8）

さて、条件文のベクトル化のために、ベクトルコンパイラは、条件文を含むDOループの制御フローを解析し、かつ条件文下の配列や変数のデータ依存関係を解析する必要がある。条件文下の配列のデータ依存関係は、条件文のない場合とほぼ等価であるが、条件文下の変数のデータ依存関係は、条件文のないときと全く異なってくる。その例を図9に示す。ここで、ケースBとCは、ベクトル化しても依存関係は変らないが、ケースAは、依存関係が変るためベクトル化できない。以前は、変数は文面上定義が参照に先行しなければならない、と述べたが、条件文の場合には、動的な実行順序の上で定義が参照に先行しなければならない、というように変更しなければならないことを示す。このために、条件文を含むDOループのプログラムフロー図に対して、変数の定義・参照関係をそのグラフ上で解析する必要があることを示す。このための方式として、表2に示すように、深さ優先探索法と、横方向優先法とがある。深さ優先探索法は、単純であるが時間がかかるため、解析の範囲を制限したりする必要がある。これに対して横方向優先法は、高速で適用範囲も広いが、局所的な情報を保持する必要があるため、安村等[8]は、データフロー変数やデータフロー演算子なるものを導入して、この方式を採用している。

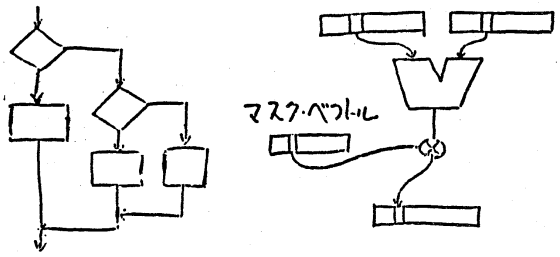



図8　制御の流れからデータ(マスクベクトル)の流れへの変換

```
IF( ) THEN      IF( ) THEN      IF( ) THEN
  v=...           v=...           v=...
ELSE            ELSE                ..=v
  ..=v            v=...         ELSE
ENDIF           ENDIF             ...
...=v           ..=v            ENDIF
```

Case A　　Case B　　Case C

図 9 Data dependency of variables under IF

表 2 Two approaches of data flow analysis with IFs

| strategy | depth-first | breadth-first |
|---|---|---|
| method | for each path(p), analyze the order of def. and ref. | for each vertex(v), determine the relation between def. and ref. |
| time | O(Np) | O(Nv) |
| workspace | none | required |
| technique | region-restriction if-then-else reduct. | data-flow variables data-flow operators |
| vectorize | simple IF statements | nested IF statements |

プログラム変換とベクトルコンパイラ

今迄述べてきたプログラム変換のほかに、ベクトルコンパイラにとっては、スーパーコンピュータの実効性能を高めるためのさまざまのプログラム変換[10]が知られている。その一覧を表3の示す。（1）は基本的なベクトル化の変換であり、（2）はことなるループを一つに融合して性能を上げようとする（1）の逆変換である。（3）は、多重ループに対して、いくつかのプログラム変換が存在することを示す。（4）は、ループのインデックスに対してその一部ないし全部を展開するものである。（5）は、ベクトルレジスタのベクトル長に制限があるとき、DOループをブロック化して、ベクトルレジスタ内に収めるための変換である。（6）は、関数・サブルーチンのインライン展開である。（7）は、内積や総和等のマクロ演算を検出して、その命令を適用するための変換である。

このように、ベクトルコンパイラにとって、プログラム変換とは、きわめて重要な技術である。プログラム変換を容易に行うために、対象とするプログラムや言語に関数的性質があることが望ましい。

表3プログラム変換

（1）ループ分配(distribution)，分割(splitting)
（2）ループ融合(fusion)
（3）ループ入れ替え(reordering)，一重化(collapse)
（4）ループ展開(unrolling)
（5）ループブロック化(blocking)
（6）関数インライン化(expansion)
（7）マクロ演算検出(macro detection)

## 終わりに

スーパーコンピュータとそのソフトウェアについて、特に、アーキテクチャとベクトルコンパイラに焦点を当てて述べてきた。ここで述べたベクトルコンパイラを中心とするスーパーコンピュータのほかに、最近では、データフローマシンがスーパーコンピュータとしても設計され始めた。この2つのアプローチの違い[11]を図10に示す。両者とも、依存グラフを内部的に作る点では同じだが、ベクトルコンパイラを用いる方式では、その後、プログラム変換を十分に行い、均質なデータ（ベクトル）をベクトルプロセッサに送り込む。これに対して、データフローのアプローチでは、非均質なデータをプロセッサに送り込み、動的な演算制御が行われる点が、根本的に異なる。

現在は、ベクトルプロセッサのアプローチが、性能と実用性の両面で進んでいるが、今後、データフロープロセッサの研究と開発の進展も楽しみである。

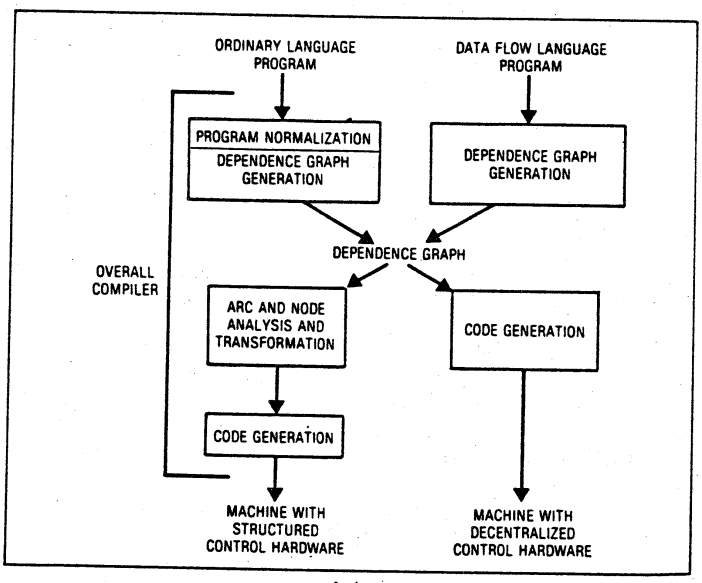



図10 Comparison of methods. 出典 11) by D. Gajski, et al

参考文献

(1) K. Hwang, "Evolution of ModernSupercomputers", Tutorial of Supercomputer: Design and Applications, IEEE, 1984.

(2) 小高俊彦，小林二三幸，長島重夫，最大性能が６３０ＭＦＬＯＰＳで１Ｇバイトの半導体記憶が付くスーパーコンピュータＨＩＴＡＣ　Ｓ－８１０，日経エレクトロニクス，pp159-184, 1983. 4. 11.

(3) K. Miura, K. Uchida, "FACOM Vector Processor System: VP-100/VP-200", Proc. of NATO Advamced Research Workshop on High Speed Computing", pp. 127-138, June 1983.

(4) S. Nagashima, Y. Inagami, T. Odaka, S. Kawabe, "Design Consideration for a High-speed Vector Processor: The Hitachi S-810", Proc. of Internatinal Conf. of Computer Design, pp238-244, 1984.

(5) R. Takanuki, Y. Umetani, I. Nakata, "Some Compiling Algorithms for Array Processor", Proc. of 3rd USA-JapanComputer Conf., pp273-279, 1978.

(6) M. Yasumura, Y. Tanaka, Y. kanada, A. Aoyama, "Compiling Algorithms and Techniques for the S-810 Vector Processor", Proc. of International Conf. on Parallel Processing, 1984.

(7) S. Kamiya, F. Isobe, H. Takashima, M. Takiuchi, "Practical Vectorization Techniques for the FACOM-VP", Proc. of Information Processing' 83, pp389-394, 1983.

(8) 安村通晃，梅谷征雄，堀越彌，自動ベクトルコンパイラにおける部分ベクトル化の方式，情報処理，VOL. 24NO. 1, PP15-21, 1983.

(9) D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolf, "Dependence Graphs and Compiler Optimizations", Proc. of the 8th ACM Symp. on Princ. of Programming Languages, pp207-218, 1981.

(10) D. B. Loveman, "Program Improvement by Source-to-Source Transformation", J. of the ACM, vol. 20 no. 1, Jan. 1977.

(11) D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages", IEEE Computer, pp58-69, Feb., 1982.

(12) R. W. Hockney, C. R. Jessehope, "Parallel Computers", Bristol, 1981.