TR－A－0045

Alliant, Convex, Ncubeの アーキテクチャとパフォーマンス

城　和貴

Kazuki Joe

1989. 1. 12

ＡＴＲ視聴覚機構研究所

目次

序章

その誕生以来、より速く走ることを運命づけられてきた計算機は、画像処理・音声認識から近年のニューラルネットワークに至るまで、膨大な計算量を迅速に処理することが期待され、要求され、義務づけられている。そして、その高速化を実現するために、さまざまの計算機アーキテクチャが提案され、製品化されてきている。これら高速化のためのアーキテクチャには、スーパーコンピュータに代表されるようなパイプライン、ベクトル化処理[1]、近年のワークステーションに代表されるようなＲＩＳＣアーキテクチャ、ミニコンピュータ以下の複数のＣＰＵを同時に実行させる並列計算機等[2]、が知られている。

特に、スーパー・ミニコンピュータの分野では、これら高速化の手法を取入れ、低価格でスーパーコンピュータに迫るパフォーマンスを実現しているものもある。このミニ・スーパーコンピュータと呼ばれる新しい分野での代表的な計算機には、Alliant、Convex、Ncube、等がある。Convex は Cray-1 に代表されるパイプライン・ベクトル化の手法に基づく計算機であり、既存の Fortran プログラムの自動ベクトル化により効率の良い処理が可能である。Alliant は共有メモリ方式の密結合型の並列計算機であり、それぞれのＣＰＵでベクトル化が可能なため、プログラムの組み方により、かなりの計算能力を引き出せる。Ncube はメッセージ通信による完全分散メモリ方式の疎結合型の並列計算機であり、最大１０２４個のＣＰＵを有効に利用できるアルゴリズムに関しては絶大なパフォーマンスを可能とする。

本稿ではＡＴＲで所有している Convex, Alliant, Ncube, のアーキテクチャについての概説をする。ただし、ＡＴＲ所有の機種は Convex C1, Alliant FX/8, Ncube-10（512node）であるが、アーキテクチャの説明にはそれぞれの最高機種である Convex C240, Alliant FX/80, Ncube-10（1024node）を用いる。

第１章　Ｃｏｎｖｅｘ　Ｃ２４０　のアーキテクチャ[3][4][5][6]

Convex C240 はサイクル・タイム 40ns のプロセッサを４台登載した密結合型の並列計算機で、各プロセッサ内のベクトル・ユニットを利用することにより、最大 200MFLOPS の能力を出せる。図1.1 はシステム・アーキテクチャの概要を示す。

Convex C240 System Architecture

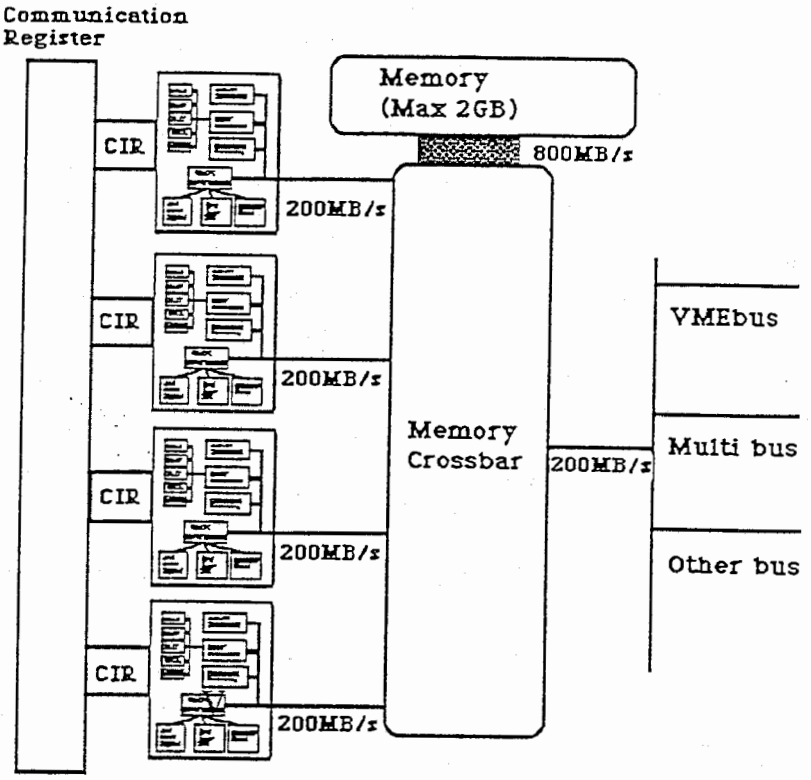


Fig 1.1

１．１　命令解読部

命令解読は Scalar Processor Functional Units にある Instruction Processing Unit（ＩＰＵ）で、それぞれ独立に行われる。（図1.2 参照）各ＩＰＵは 8KB のインストラクション・キャッシュを持ち、①インストラクション・フェッチ、②命令の解読と読み込まれた（マイクロ）命令のキャッシュ・イン、③解読された命令の配布、を２段もしくは３段のパイプライン処理により実行する。解読された命令は、その種類に応じて、スカラー・ユニット、ベクトル・ユニット、ＩＰＵ自身（分岐命令や NOP 命令の場合）に配布される。また、ＩＰＵは常に現在のプログラム・カウンタ・レジスタをチェックし、命令のプリ・フェッチを行うことにより、インストラクション・キャッシュを有効に使っている。（図1.3 参照）

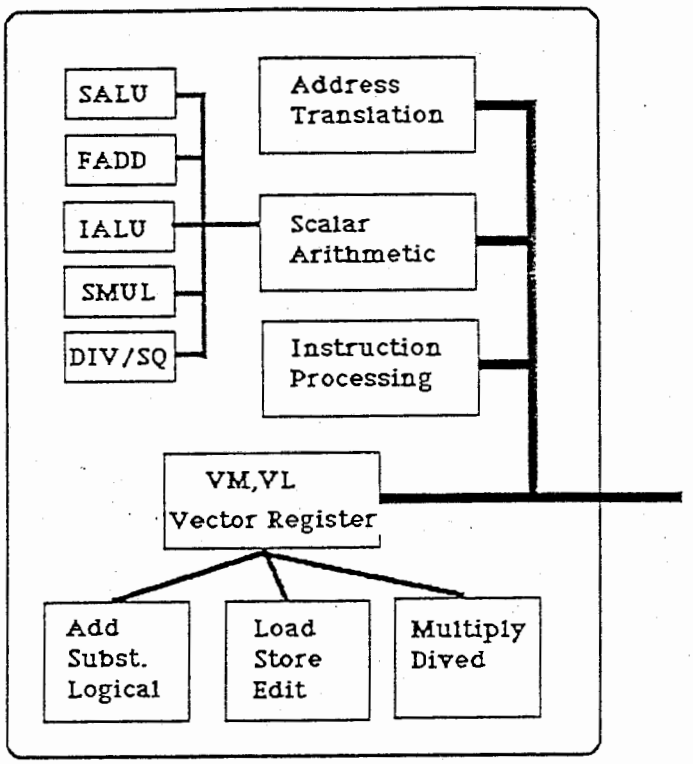


Fig 1.2

Convex の命令解読部自体には、特に他のマシンと異なる特徴はないが、後述する効率の良いメ　モリ・インターリーブによって、安定した命令パイプラインが実現されているようだ。

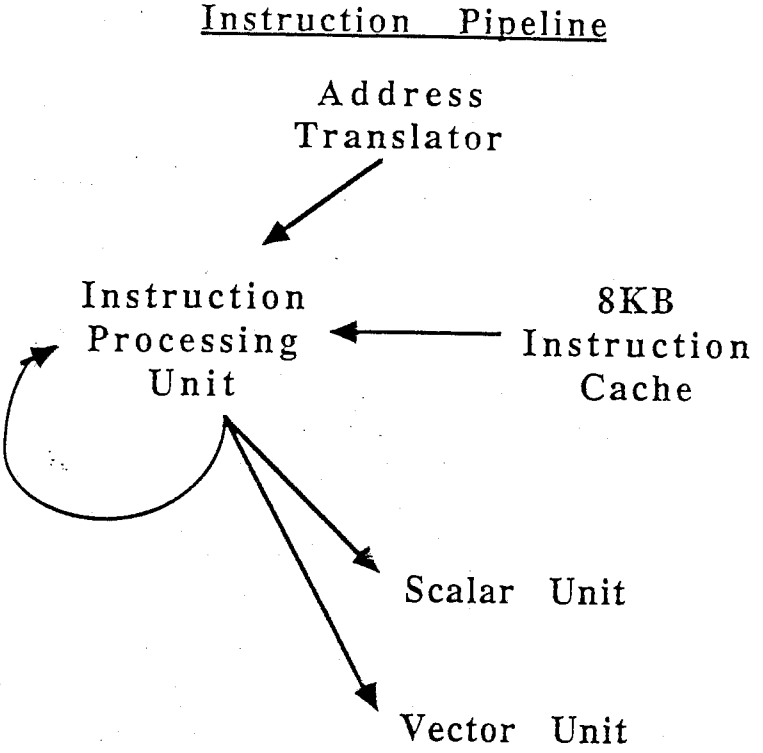
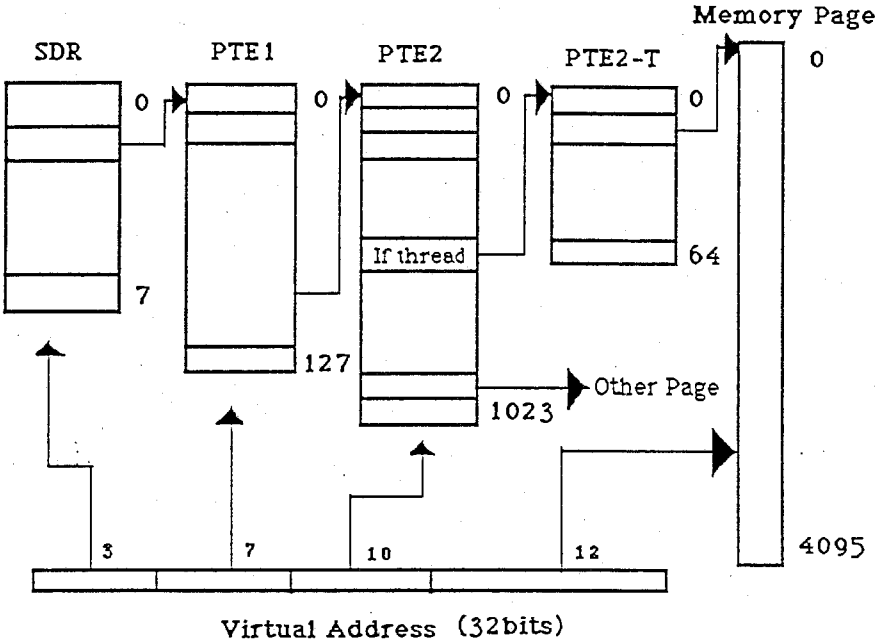


Fig 1.3

1．2　アドレス変換部

アドレス変換は Scalar Processor Functional Units にある Address Translation Unit （ＡＴＵ）で、それぞれ独立に行われる。（図1.2参照）ＡＴＵは与えられた仮想アドレスを物理アドレスに変換する。このときの物理アドレスは、メモリ上を指しているか、ページ・ファイル上にページアウトされているかのどちらかである。与えられる３２ビットの仮想アドレスは、図1.4で示されるように、上位から順に３、７、１０、１２ビットのオフセット値を持っている。これらのオフセット値は、Segment Descriptor Register （ＳＤＲ、後述）、レベル１及びレベル２のページ・テーブル・エントリ、物理ページ内でのオフセットを指し示している。ここで、ＳＤＲはレベル１のＰＴＥ、レベル１のＰＴＥはレベル２のＰＴＥ、レベル２のＰＴＥは物理ページをそれぞれポイントするのに使われている。

Fig 1.4



Address Translation in Convex

ＡＴＵは仮想アドレスを受け取ると、それがアドレス・キャッシュにあるかどうかを調べる。アドレス・キャッシュは直前の 1024 のアドレス変換をキャッシュしており、（すなわち、レベル２ＰＴＥへのポインタを持っている）もし、キャッシュ・ヒットすればレ

ベル２ＰＴＥを参照することにより直ちにアドレス参照要求がなされる。ヒットしない場合、ＡＴＵは与えられた仮想アドレスの上位３ビットに従って、各プロセッサのコミュニケーション・レジスタからＳＤＲを受け取る。ＳＤＲはその時点での各プロセッサが仮想空間のどこをアクセスしうるかという情報を示す８つのレジスタである。ＡＴＵはＳＤＲに従ってレベル１のＰＴＥを物理メモリから得る。このとき、１２８のエントリーからなるレベル１のＰＴＥは必ず物理メモリ上に常駐している。次に、ＡＴＵは仮想アドレスの次の７ビット・フィールドと、得られたレベル１ＰＴＥからレベル２ＰＴＥを見つける。更に、ＡＴＵは次の１０ビット・フィールドと、得られたレベル２ＰＴＥから実際のページ番号を得て、最後の１２ビットをそのページでのオフセットとして、目的のアドレス変換を終る。勿論、得られたページが物理メモリ上にない場合、ページングが起きる。

Convex C2 は４つのＣＰＵを持つ並列計算機であるため、複数のプロセッサが同一のＳＤＲを持つこともある。しかしながら、（おそらくはソフトウェアのために）この様な場合にもメモリの共有をさせたくない場合もある。例えばＣ言語で、ある関数を並列にコールしたとしよう。各プロセッサは同一の関数を実行しているが、関数の中でユーザ・スタック上に作られる auto 型の変数は複数のプロセッサから干渉されてはならないことは明白であろう。この様な問題点を解決するために、Convex C2 には、レベル２ＰＴＥのあとに、スレッド・レベルＰＴＥと呼ばれる付加的なページ・テーブルが用意されている。ＡＴＵは、参照したいページが共有メモリである場合はレベル２ＰＴＥのエントリーをページ番号として、共有メモリでない場合は６４エントリーからなるスレッド・レベルＰＴＥへのオフセットとして解釈する。スレッド・レベルＰＴＥのエントリーは非共有メモリのページ番号を示す。

Convex C2 のページ・テーブル方式のアドレス変換法及びそのキャッシュはＶＡＸを始めとする一般的な仮想記憶システムの手法を用いているわけだが、スレッド・レベルＰＴＥによる付加的なアドレス変換の手法は、例えば Symbolics リスプ・マシンのダイナミック・ガベッジ・コレクションを可能とするために用いられているインビシブル・ポインタを連想させ、非常に興味深い[10]。また、この様な付加的なページ変換がパイプラインにどの様な影響を与えるのか、その影響をどの様に回避しているのか興味深いところである。

１．３　スカラ演算部

スカラ演算は Scalar Processor Functional Units にある Address/Scalar Processing Units （ＡＳＰ）で、それぞれ独立に行われる。（図1.2 参照）ＡＳＰには、図1.5 に示されるように５種類の演算ユニットと 4KB のデータ・キャッシュがある。Scalar Arithmetic/Logical Unit （ＳＡＬＵ）はデータ型の変換等、種々のスカラー演算を行う。Floating Adder （ＦＡＤＤ）は１クロックで小数の和・差を求める。Integer Arithmetic/Logical Unit （ＩＡＬＵ）は２クロック

で整数の和・差、比較演算を行う。Integer & Floating Multiplication （ＳＭＵＬ）は７～１１クロックで整数及び小数の積を計算する。Integer & Floating Divider and Square rooter （ＤＩＶ／ＳＱ）は２２～７９クロックで整数及び小数の商を、４０～６１クロックで平方根を求める。

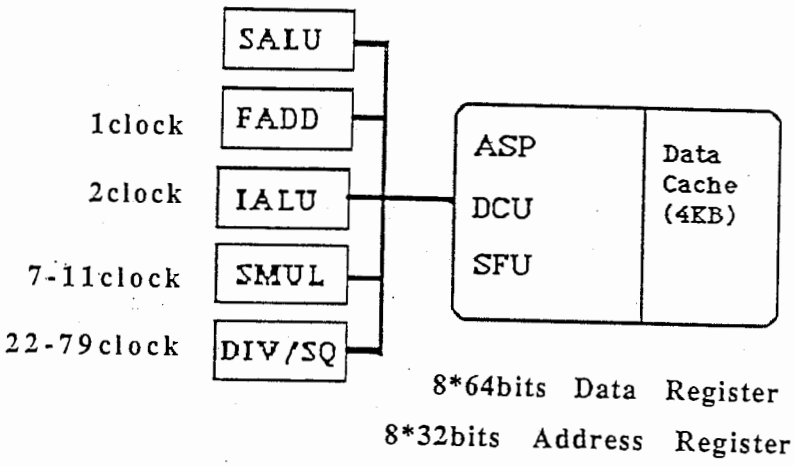


Fig 1.5

これらの演算ユニットは同時に作動するので、スカラー演算における命令パイプラインも効率よく行われる。また、後述するベクトル演算部とスカラ演算部も同時に作動するので、ベクトル・スカラ演算におけるパイプラインやチェイニングが可能である。

ＡＳＰはそれぞれ独立したデータ・キャッシュを持っている。各データ・キャッシュは６４ビット・エントリーで、ライト・スルー方式である。ＡＳＰのアドレス参照はＡＴＵへ渡されると同時にデータ・キャッシュの探索も行われる。もし、ヒットすれば１クロックで目的のデータを得るし、ミスすればＡＴＵによる参照を待つ。（このとき、当然ＡＴＵではアドレス・キャッシュが探索される。）

また、複数のＡＳＰが同時に同一メモリをキャッシングすることを防ぐために、各データ・キャッシュにはローカル及び他の３ＣＰＵ、そしてＩ／Ｏシステムのための５つのタグが（ハードウェア・レジスタとして）ついており、参照の競合を禁止している。

Convex C2 ではＡＳＰの５種類の計算ユニットを利用することにより最小２段、最大７段の単一演算パイプラインが実行出来ることになる。（スカラーのパイプラインであるので、ベクトル・プロセッサに対する演算チェイニングのような働きをする。）従って、Convex C2 上でのプログラムの最適化はベクトル化の次に、このスカラ・パイプラインをいかに上手に使いこなすかに左右される。

１．４　レジスタ

スカラ・ユニットには８個の６４ビット・スカラ・レジスタと、８個の３２ビット・アドレス・レジスタがあり、各ユニットから１クロックでロード／セーブ出来る。

ベクトル・ユニットには８個の１２８＊６４ビット・ベクトル・レジスタと、１２８ビットのベクトル・マスク・レジスタ、３２ビットのベクトル・レングス・レジスタがある。

1．5　ベクトル演算部

ベクトル演算部は 1.4 で述べた各種ベクトル・レジスタと３種類のゲート・アレイからなる。（図1.2 参照）ベクトル・レジスタは処理させるベクトルの奇数要素用と偶数要素用の２つが対になって使用される。そして、それらを処理する３種類のゲート・アレイも２組づつあり、ダブル・バッファリングを行って２並列の処理を行う。

Load/Store/Edit ユニットはベクトル・レジスタへのデータの受渡しを担当する。このとき、スカラ・ユニットのデータ・キャッシュは使用されず、直接メモリ参照が行われる。（データ量の大きいベクトルをキャッシュに置くことはキャッシュのスラッシングを引き起こす可能性が高いからと考えられる。）

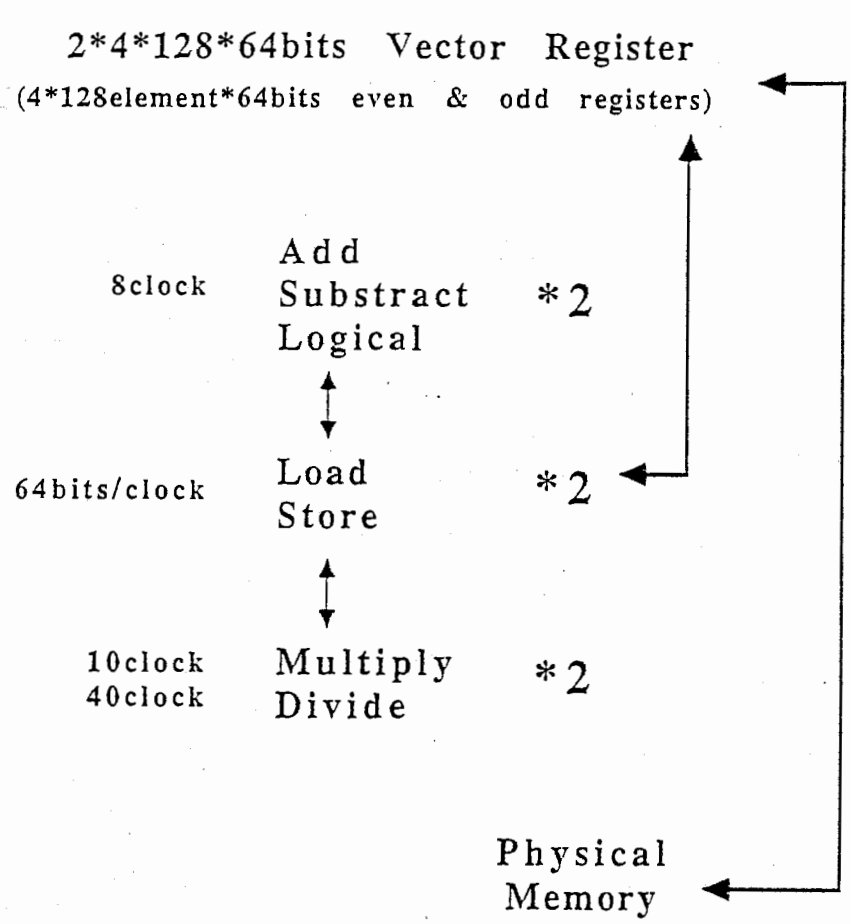


Fig 1.6

このとき、ベクトル・レジスタとメモリとの間の転送速度は６４ビット／クロックである。（ただし、メモリ・インターリーブに大きく左右される。）このようにして得られたベクタ・レジスタに対して Add/Substract/Logical ユニットは和・差・比較演算その他を３２０ｎｓ以下で、Multiply/Divide/Sqrt ユニットは積を４００ｎｓ以下で、商及び平方根を１６２０ｎｓ前後で実行する。

各ゲート・アレイは同時に作動するので、２段から３段のチェイニングが可能である。即ち、各ベクタ・ユニットごとに最高２並列３多重のチェイニング処理が可能なわけである。（図1.6 参照）

ところで Convex C-1 にはＰキャッシュと呼ばれる 64KB のデータ・キャッシュが使われていた。そしてベクタのロード・ストアは、ストライドが連続もしくはメモリ・インターリーブに一致しなければＰキャッシュを使用していた。データに依存するとはいえ、どちらがどれだけ効率がよいのか興味深いところである。

１．６　メモリ・システム

1.5 で述べたように、Convex C2 はベクトル・レジスタのロード・セーブにキャッシュを使わない。（しかも、４並列である）そのため、メモリとＣＰＵ間のバスはかなり高速でなければならない。図1.7 に示すように、４つのＣＰＵ及びＩ／Ｏシステムは、クロスバーによってメモリ・コントロール・モジュール（ＭＣＭ）につながっている。ＣＰＵ及びＩ／Ｏシステム側の各々のバス速度は 200 MB/s である。（６４ビットのデータを 40ns で転送できる。即ち、ベクトル・ユニットでのパイプラインに遅延をかけないことになる。）また、ＭＣＭ側は１００MB/s のバス８本が８つのＭＣＭにつながっており、見かけ上 800 MB/s の超高速バスを形成している。

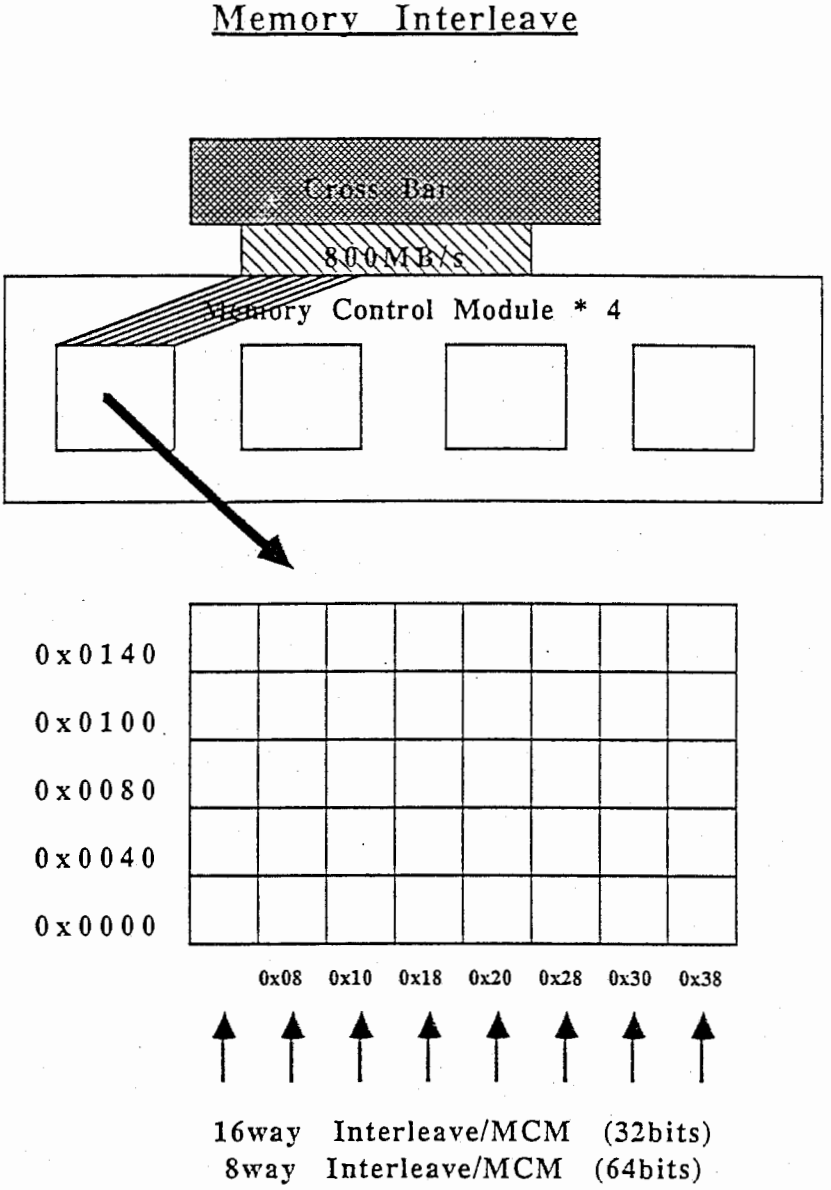


Fig 1.7

各ＭＣＭは、最大２５６MB の物理メモリを８つのブロックに分割して制御している。このため、全体で８～６４way のメモリ・インターリーブを可能としている。この強力なメモリ・インターリーブが Convex C2 のスカラ／ベクトル・パイプラインを支えていると言っても過言ではないだろう。

Convex アーキテクチャの特徴の一つにバイト・ロードがある。スーパー・コンピュータを始めとするベクタ・パイプライン方式のマシンは６４／３２ビット・ロードが大半をしめている。しかしながら、例えばリモート・センシング等の画像処理では、入出力画像の各画素は１バイトで表されることが圧倒的に多い。このような場合、６４／３２ビット・ロードのマシンではメモリを無駄にするか、データ型変換のための無駄な計算をしなければならない。Convex のバイト・ロードが、どれくらいのパフォーマンスを実現しているのか不明だが、少なくともこのような問題に対する一つの答えを提供していると言って良いだろう。

1．7　並列処理[6]

Convex C2 は Automatic Self Allocating Processors（ＡＳＡＰ）と呼ばれる並列処理環境を提供している。基本的なコンセプトは、稼働中のプロッセッサのプロセスを Fork して、稼働していないプロッセッサに同一のコンテキストを持つプロセスを作り、付加分散を計ることである。（この手法は後述する Alliant の Dynamic Complex モードに酷似している。）各プロセッサは、コミュニケーション・レジスタと呼ばれるプロセス・コンテキスト格納場所を（スケジューラの命令に従って）自由にアクセスできる。もし、同一のコミュニケーション・レジスタを複数のプロセッサがアクセスすると、同じプロセスが別々のプロセッサ上で、仮想空間を共有しつつ（注：共有しない場合もある。1.2 参照）、タスクを分担して行う。（図1.8 参照）

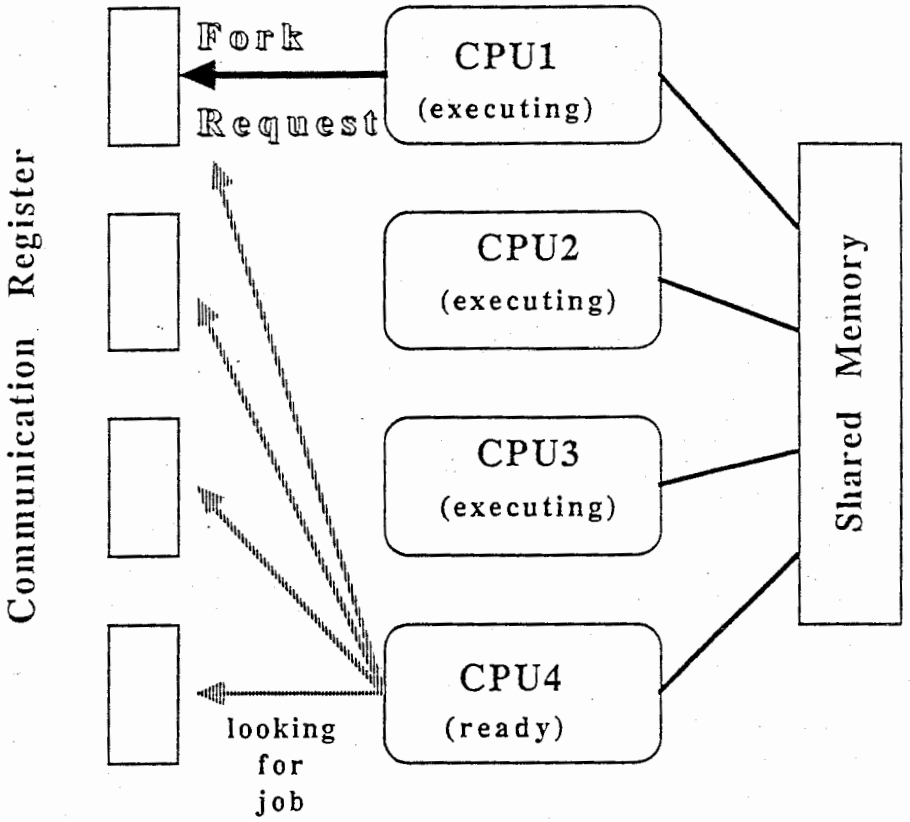


Fig 1.8

一般に、共有メモリ方式の密結合型並列処理系では、いわゆる相互排除問題が重要であるが[7]、Convex C2 ではハードウェア・レジスタにセマフォを組み込むことにより、この問題に対処しているようである。プロセッサ側からはウエイト命令が、メモリ側からはシグナル命令が送られ、共有メモリのクリティカル・エリアの管理を行っている。もし、デッド・ロックに陥れば、デッド・ロック・トラップが起動するようだが、詳細は不明である。

いずれにせよ、この手の並列処理を行うには非常に優れたスケジューラが不可欠である。Convex C2 では、Alliant のようにユーザがプロセッサの優先順位を指定したり、特定のプロセッサにだけ処理を行わせることが出来ないようだが、並列処理アーキテクチャの名前どうり（ＡＳＡＰ = As Soon As Possible）のスケジューラであるかどうか、興味深い。

Alliant FX/80 はサイクル・タイム 85ns のプロセッサを８台登載した密結合型の並列計算機で、各プロセッサ内のベクトル・ユニットを利用することにより、最大 188 MFLOPS の能力を出せる。図2.1 はシステム・アーキテクチャの概要を示している。この図に示されるように、Alliant は、インタラクティブなタスクをこなす最大１２個のＩＰ（Interactive Processor）と、数値計算を専用に行　う８個のＣＥ（Computation Element）との２種類のプロセッサから構成されている。
以下はシステムの中核をなすＣＥ部分の説明である。

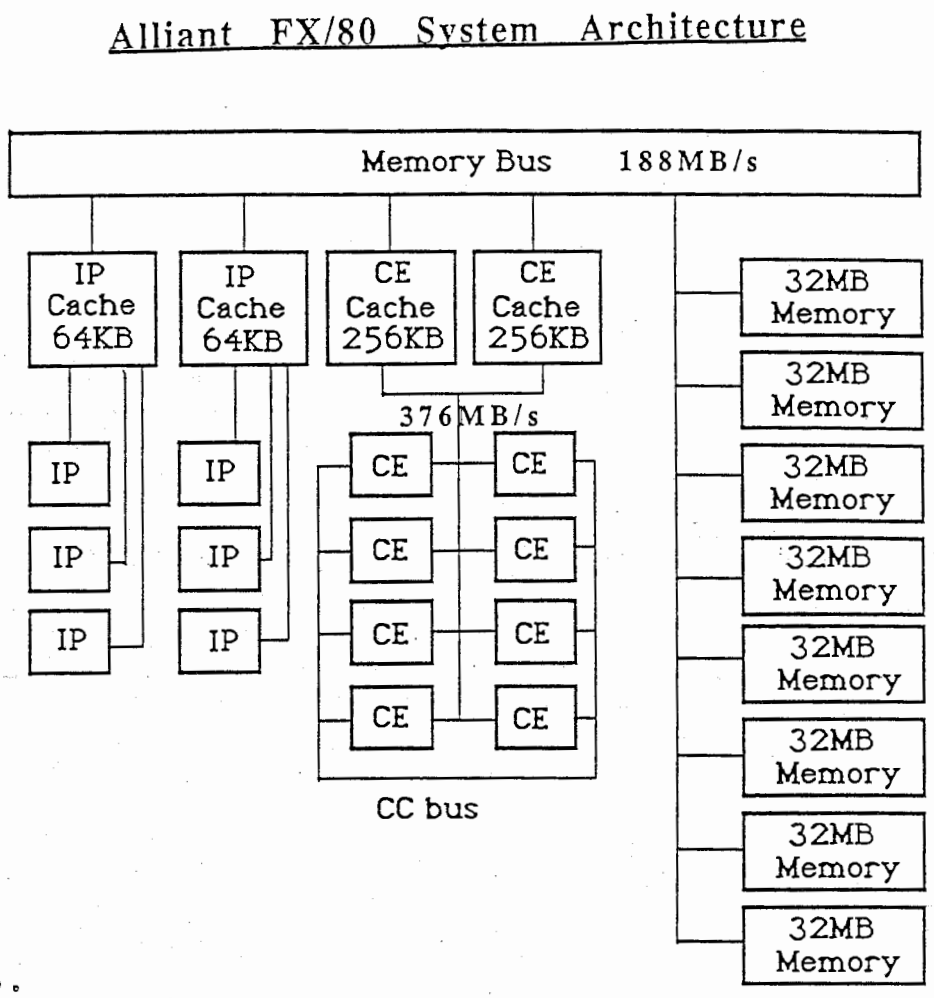


Fig 2.1

２．１　命令解読部

命令解読はＩＢＯＸセクションでそれぞれ独立に行われる。（図2.2 参照）命令解読に関わる部分は、ＩＢＯＸ・コントロール・ストア（ＩＣＳ）、アドレス・ユニット、インストラクション・パーザー、インストラクション・キャッシュである。アドレス・ユニットは後述するアドレス・トランスレーション・ユニットを通してメモリから、もしくはインストラクション・キャッシュから命令のフェッチを行い、インストラクション・パーザーに送る。このとき、目的とする命令がメモリから読み込まれていれば、16KB のインストラクション・キャッシュに転送される。インストラクション・パーザーはフェッチされた命令の解読を行い、ＩＣＳを通して整数演算部、ベクトル演算部等へのディスパッチを行う。この時の命令パイプラインはインストラクション・パーザーによって管理され、その段数は４である。（図2.3 参照）

Alliant の命令解読部の詳細はパーザーとＩＣＳを解析しなければならない。パーザーはマクロ・インストラクションを解読後、ＩＣＳにマイクロ・インストラクション実行の命令を発行し、ＩＣＳは自身のユニット内にあらかじめロードされているマイクロ・プログラム・エリアのオフセットを受け取ることにより、該当するマイクロ・インストラクションを解読し、整数演算部、小数演算部、ベクトル演算部、アドレス・ユニット、コンカレント・コントロール・ユニット（後述）等にインストラクションのディスパッチを行っていると推測されるが、どのような形でパイプラインを安定に保っているか等、不明であり、興味深いところでもある。

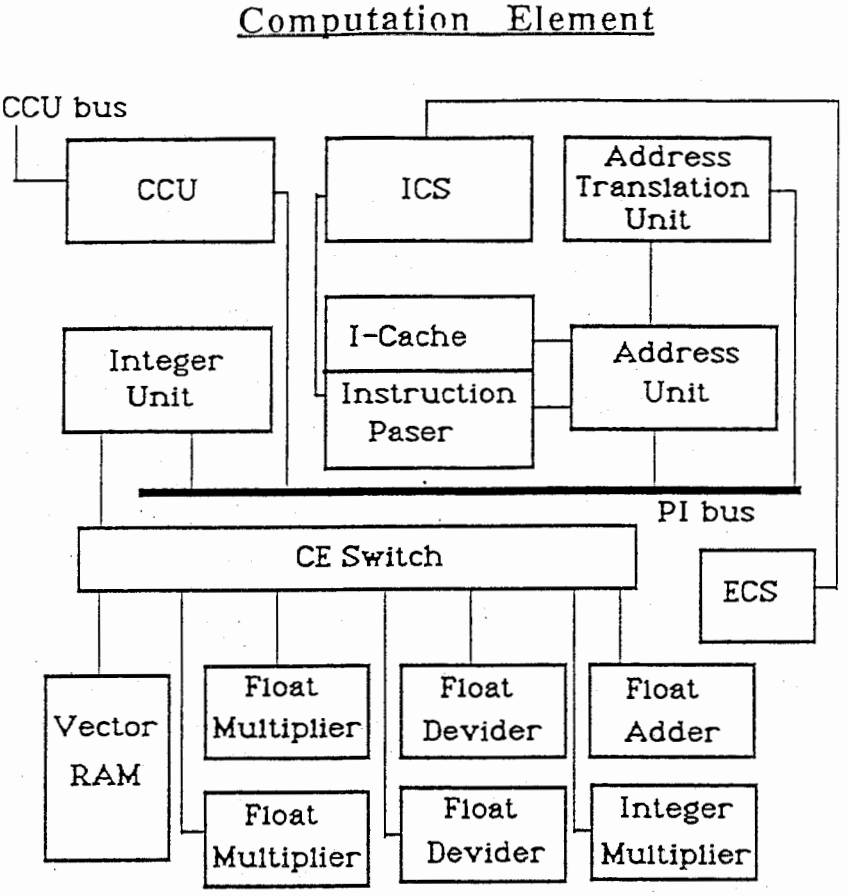


Fig 2.2

２．２　アドレス変換部

アドレス変換はＩＢＯＸにある Address Translation Unit でそれぞれ独立に行われる。（図2.2 参照） Address Translation Unit は与えられた仮想アドレスを物理アドレスに変換する。このときの物理アドレスは、メモリ上を指しているか、ページ・ファイル上にページアウトされているかのどちらかである。与えられる３２ビットの仮想アドレスは、上位から順に、１０ビットのセグメント番号、１０ビットのページ番号、１２ビットのバイト・オフセットに分けられる。各プロセスはアクセス・コントロール・テーブルと呼ばれる４つのメモリ管理用の内部レジスタ（スーパーバイザ・プログラム及びデータ、ユーザ・プログラム及びデータ）を持ち、それぞれ該当するセグメント・テーブルの物理アドレスをポイントしており、アドレス・トランスレーション・キャッシュが利用できない場合は、このレジスタからアドレス変換が行われる。Address Translation Units はセグメント・テーブル、ページ・テーブルを利用して、Convex と同様、通常のページ・テーブル方式によるアドレス変換を行う。キャッシュが利用できる場合は、ユニット内のバッファに蓄えられた６１４４のページ・テーブル・エント

リと１５３６のセグメント・テーブル・エントリから直接物理アドレスを求める。

Alliant で各プロセッサが同一のプロセス・コンテキストを持つ場合、Convex とは全く異なり、非共有メモリ・アクセスの概念はない。（1.2参照）各ＣＥに割り当てられたプロセスは、そのＣＥのＣＣＵの３２個のレジスタにプロセス・ステータスが格納される。その１つが Cactus Stack Pointer（Cactus = サボテン）を指し示している。つまり、各ＣＥ上でのプロセス・コンテキストは、ユーザ・スタック・エリアとして、全てのＣＥ上のタスク（もちろん、同一プロセス）から参照できるグローバル・エリアと、各ＣＥ上のタスクにユニークなローカル・エリア（Cactus）とに分かれている。このローカル・エリアは他のＣＥからアクセス不可能であり、これにより非共有メモリが実現されている。また、Fortran でベクトルの積和計算等を行うときのテンポラリの変数は、コンカレント・ディレクティブの指定により非共有となるわけだが、この場合は Cactus Stack を使うのではなく、プリコンパイラによりテンポラリの配列が自動的に作られ、各ＣＥに送られるわけである。（このプリコンパイラはレベル０の Fortran コンパイラと呼ばれている。）

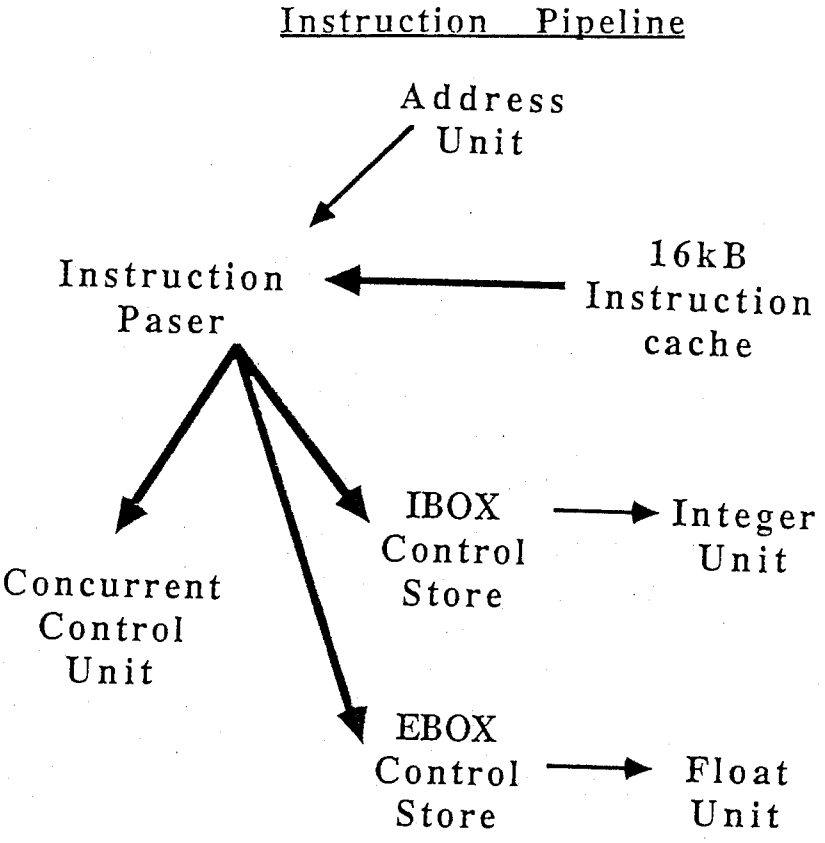


Fig 2.3

２．３　整数演算部

Alliant ではスカラ小数演算を後述するＥＢＯＸで行っているため、Convex のスカラ演算部に相当するのはＩＢＯＸの Integer Unit ということになる。このユニットは６８０２０のインストラクションを実行する。データ・キャッシュは後述する大規模なキャッシュ・システムをベクトル／小数演算部と共有している。

各インストラクションの実行クロックは６８０２０と同等と推定されるが、演算パイプラインを行っているか、あるいはベクタ／小数演算部とのパイプライン／Chaining を行っているか等は不明である。

もし、この部分でのパイプライン処理がないのであれば、Convex と比較した場

合の Alliant の特徴と言ってよいだろう。なぜなら、前者はもともとパイプライン／ベクトル処理から派生してきたマシンであり、後者は並列処理から派生してきたマシンだからである。

２．４　レジスタ

ＣＥには小数演算用に８個の６４ビット・レジスタが、ベクトル演算用に８個の６４ビット３２要素・ベクトル・レジスタが用意されている。また汎用レジスタとして、８個の３２ビット・データ・レジスタ、８個の３２ビット・アドレス・レジスタが用意されている。ベクトル・レングス及びベクトル・マスク・レジスタにはデータ・レジスタを用いている。（1.4 参照）

２．５　ベクトル／小数演算部

ベクトル及び小数演算はＥＢＯＸセクションで行われる。（図2.2 参照）ＥＢＯＸは、ＥＢＯＸコントロール・ストア、ベクトルＲＡＭ、小数加算器・乗算器・除算器、整数乗算器から構成されており、小数乗算器及び除算器はそれぞれ２個ある。ＥＢＯＸコントロール・ストアは、パイプラインが安定するようにこれらを制御する。ベクトル／小数演算のパイプラインについても詳細は不明だが、構成からすると、２並列多重方式のように推測される。なお、ベクトル演算は３２要素ごとに行われる。

主な小数演算のサイクル数は、和差積が１、商が２、ｃｏｓ演算が９～１７、ｓｉｎ演算が８～１７、ＬＯＧ演算が１８、平方根が４である。また、主なベクトル演算のサイクル数は、和差積が２＋０．５＊ベクトル長、商が６＋６＊ベクトル長、積和計算が３＋０．５＊ベクトル長である。

Alliant のベクトル演算の特徴としては、ベクトルのロード・ストアに専用のインストラクションを持っていることが上げられる。Scatter/Gather と呼ばれるこのインストラクションは、後述するデータ・キャッシュを通して、ベクトル・レジスタとメモリとの間のベクトルの受渡しを、ハードウェアにより高速に行う。

ここで賢明なる読者諸君は、どうして Convex のベクトル・インストラクション・サイクルがベクトル長に無関係で、Alliant のそれが関係するのか疑問に思われるだろう。これはベクトル演算に於て、Convex はチェイニングを行っており、Alliant は行っていないからである。すなわち、Convex のベクトル・ユニットのロード・ストア・ユニットが、あるベクトル要素をロードしてくると、その要素は即座にベクトル演算が行われ、それと並行して次の要素のロードが行われるわけである。すなわち、ベクトル・パイプラインのシーケンスが十分に安定していれば、ベクトル・インストラクションのサイクル・タイムは各演算ユニットの速さだけに左右される。もし、その他の要因があるとすれば、メモリ・インターリーブの競合によりロード／ストアに遅延をきたす場合であろう。これに対して、Alliant のロード／ストアは前述したように Scatter/Gather を用いて行われ、最高３２要素までのデータ・キャッシュ・アクセスが完了してからベクトル演算が実行されるため、ベクトル長がベクトル・インストラクションに影響を与える。

これらの問題はコストとのかねあいがからみ複雑である。つまり Convex はベクトル・マシンであり、Alliant は並列マシンである、と言うことであろう。

Convex C2 は 1.5 で示したように、ベクトル演算でのデータ・キャッシュを行っていない。Convex の６４ way メモリ・インターリーブを利用したメモリの直接アクセス方式と Alliant の大容量データ・キャッシュ＋ Scatter/Gather 方式では、一体どちらが効率が良いのか（コスト・パフォーマンスとのかねあいで）誠に興味深い。一般的には、大容量キャッシュはヒット・ミスを減らす反面、キャッシュ・アクセス・タイムを増大させる。従って、基本マシン・サイクル・タイムを短縮させにくくなる。この点からすると、Convex はマシン・サイクルを小さくする方向で、Alliant は現状維持のサイクル・タイムでプロセッサの数を増やす方向で、それぞれのアーキテクチャを発展させるものと思われる。

２．６　メモリ・システム

Alliant FX/80 は最大 256MB のメモリを登載でき、４ way のメモリ・インターリーブを可能としている。メモリ・バスは１８８MB/s で、データ・キャッシュからクロスバーへとつながっている。（図2.4 参照）データ・キャッシュはライト・バック方式の 128KB のものが４個装備され、全体で 512KB の大容量のデータをキャッシュしている。８個のＣＥは、それぞれ独立して、この４個のデータ・キャッシュを使うわけだが、ＣＥからクロスバーを通ってデータ・キャッシュまでの転送速度は３７６MB/s である。

なお、Alliant にはＣＥのためのキャッシュの他に、１２８KB のデータ・キャッシュがＩＰのために用意されている。

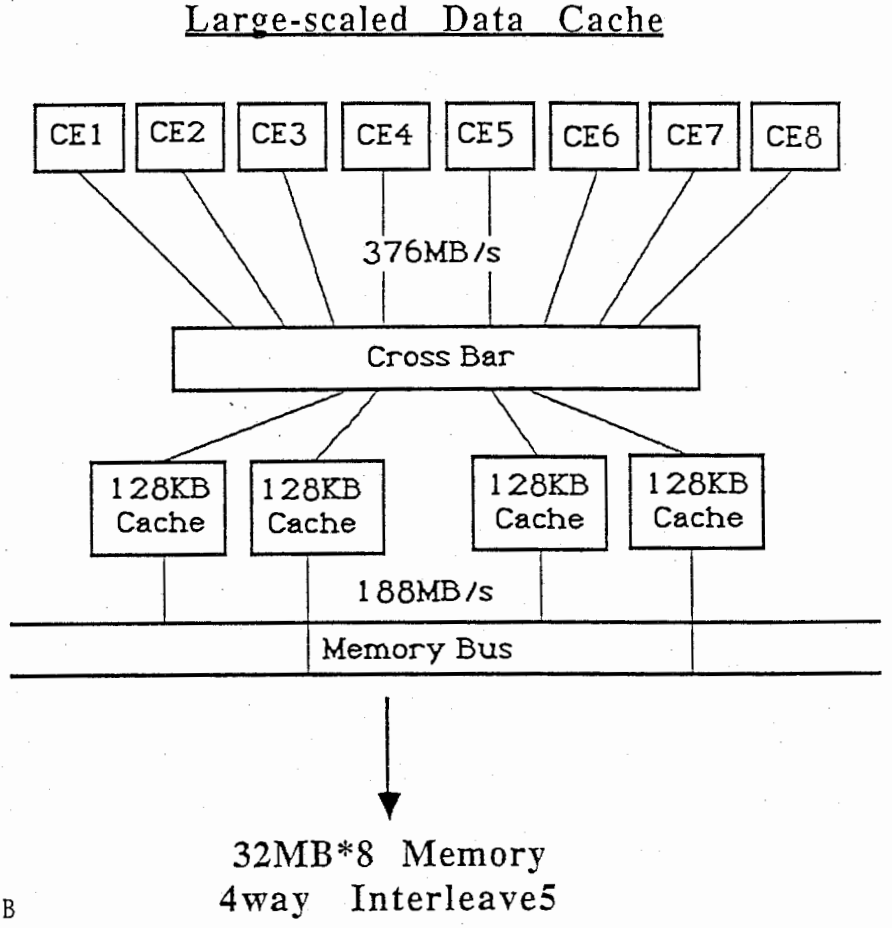


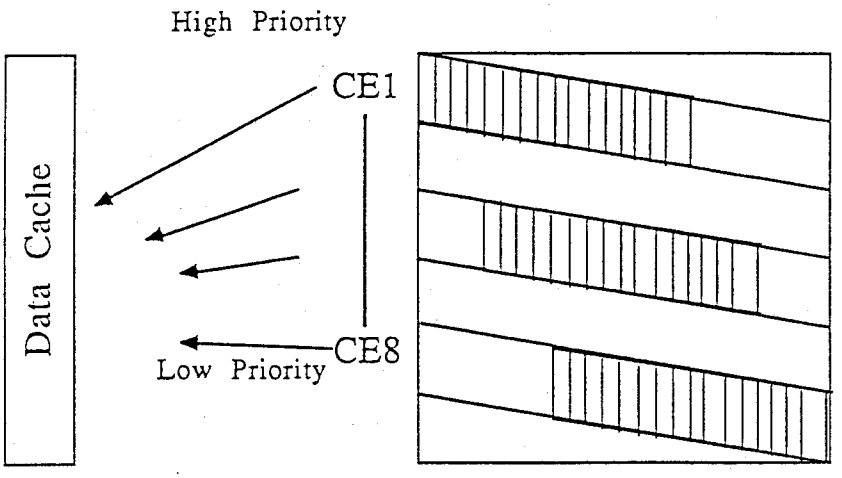
Fig 2.4

Alliant は Convex と同様、バイト・ロードが用意されている。勿論、先に述べた Scatter/Gather インストラクションもバイト指定が可能なため、効率の良いリソースの利用が出来る。

Alliant は 2.7 で述べるように８個のＣＥが独立して共有メモリをアクセスするわけだが、メモリ・アクセスの大半は大容量のデータ・キャッシュにヒットするものと考えられる。従って、１つのデータ・キャッシュ・ユニットにおける複

Memory Access Contention

数ＣＥの参照の競合が問題となる。Alliant FX/8までは、図2.5(a)で示すように、ＣＥの番号による優先順位によってこの問題を回避してきた。この手法は図2.5(a)で示されるように、タスクの並列化にパフォーマンスが依存する。Alliant FX/80では、図2.5(b)で示すように、優先順位を常にローテーションさせる手法をとっている。この優先順位のローテーションは１４サイクルごとにハードウェアによって行われ、タスクがどの様に分割されていても適当なパフォーマンスを発揮出来る。



Fug 2.5(a)

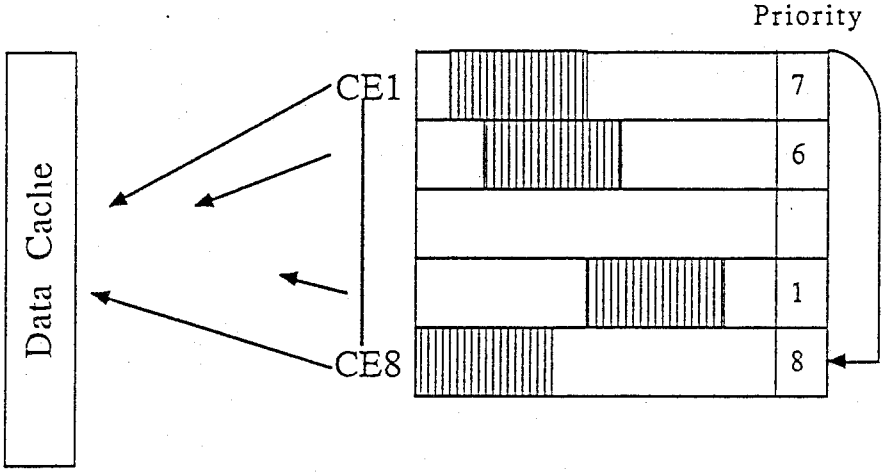


Fig 2.5(b)

一般に並列マシンのキャッシングには、従来のライト・スルー方式のキャッシュ・マネージメントが使えないことが知られている[11]。このキャッシュ・コヒーレンス問題を解決するには、

①リードのみもしくはローカル・リードのみのキャッシュを用いる。
（当然共有メモリのリード・ライトに時間がかかる。）

②プロセッサ間でキャッシュを共有する。
（プロセッサとキャッシュの間にインターコネクション・ネットワークを置く必要があり、そのためメモリ・アクセスが多少遅くなる。）

③キャッシュと共有メモリ上でのデータ・ブロックにフラグを置き、ダイナミックなコヒーレンス・チェックを行う。（フラグ操作にオーバーヘッドが生じるので、小数のプロセッサからなるシステムでしか使われない。）

の３方式がある。明かに、Alliant は②であり、Convex は③を採用している。

Alliant の問題点は６４バイト＊４のキャッシュ・ブロック（各６４バイトがそれぞれ１２８ＫＢのキャッシュに割り当てられている。）とＣＥとのスイッチ

ング・ネットワークが、ＣＥの数に制約を受けるという点である。つまり、現行の８ＣＰＵシステムを３２ＣＰＵシステムに拡張するとき、最初に問題になる点がこれであろう。Alliant の新機種はこの問題点を解決し、３２ＣＰＵ（＝４クラスタ）システムとなるものと予想される。

Convex の問題点はフラグ操作のオーバーヘッドである。ベクトル演算にはキャッシュを利用しないとはいえ、４ＣＰＵ以上のプロセッサにこのようなフラグ操作を効果的に（しかもライト・スルーで）行わせることはむずかしいと思われる。

２．７　並列処理

Alliant はもともと密結合型の並列処理系であり、機能拡張としてベクトル処理、パイプライン処理を施したという印象が強い。これは、もともとベクトル処理系であった Convex C1 が機能拡張して４台のプロセッサを登載した並列処理系になったことと好対象である。

Alliant FX/80 の強力な並列処理のメカニズムの基盤をなすのは、ＣＥの中にあるコンカレンシー・コントロール・ユニット（ＣＣＵ）である。ＣＣＵはＩＢＯＸのＰＩバスを通じて、インストラクション・パーザー、アドレス・ユニット、整数演算ユニットからの制御情報を受け取る。そして、各ＣＥのＣＣＵはＣＣＵバスを通して、同期を取り合っている。

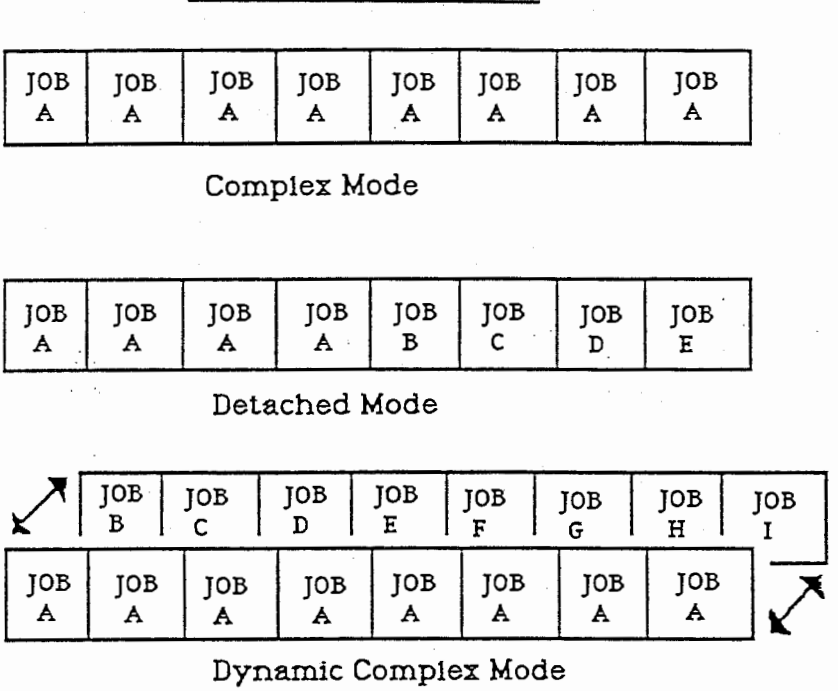


Fig 2.6

各ＣＣＵは３２個のレジスタを持ち、並列にタスク分割されたプロセスの状態を記述するのに１６個が、同期等のハードウェア的な並列処理メカニズムのために残りの１６個が使用される。プロセスの状態としては、ＰＣ、ＳＰ、Cactus SP、イタレーションの回数、現在のイタレーション数、次のイタレーション数、同期のためのセマフォに似たウェイト／シグナルなどが高速のＣＣＵバスを通して伝えられる。（このＣＣＵバスに関する資料は全くない）

ユーザはプログラム中から自分のジョブを
①全てのＣＥを使ったり（複合モード）、
②一部のＣＥを指定したり（分離モード）、
③処理させるＣＥをスケジューラにまかせたり（ダイナミック複合モード）できる。（図2.6 参照）また、各ＣＥに対するリスケジュールのタイミングの指定等もオンラインで可能である。

ダイナミック複合モードにおいては、若干のオーバー・ヘッドがあるとは言え、8台の密結合したプロセッサのロード・バランシングが可能となっている事実は驚嘆に値すると言えよう。

第３章　Ｎｃｕｂｅ／１０のアーキテクチャ[2][9]

Ncube は前述した Alliant, Convex と根本的にアーキテクチャの異なるマシンである。すなわち、大規模なパイプライン／ベクトル処理による高速化も、プロセッサ間での同期を取りながらの並列処理も行わない。つまり、メモリを共有する密結合型の並列処理は行わないということである。

Ncube/10 は公称値 2.5 MIPS／ 0.4 MFLOPS の能力を持つプロセッサを最大１０２４個装備できる完全分散処理型の大規模疎結合並列処理系である。ノードと呼ばれるこのプロセッサは、Alliant のＣＣＵ、Convex のコミュニケーション・レジスタに相当する同期のためのユニットを一切持たない。各ノードは、ユーザのプログラムからのみ他ノードやホスト・プロセッサと通信を行う。勿論、通信のためのプロトコルもユーザにまかされている。図3.1 はシステムの概要を示す。

Ncube/10 System Architecture

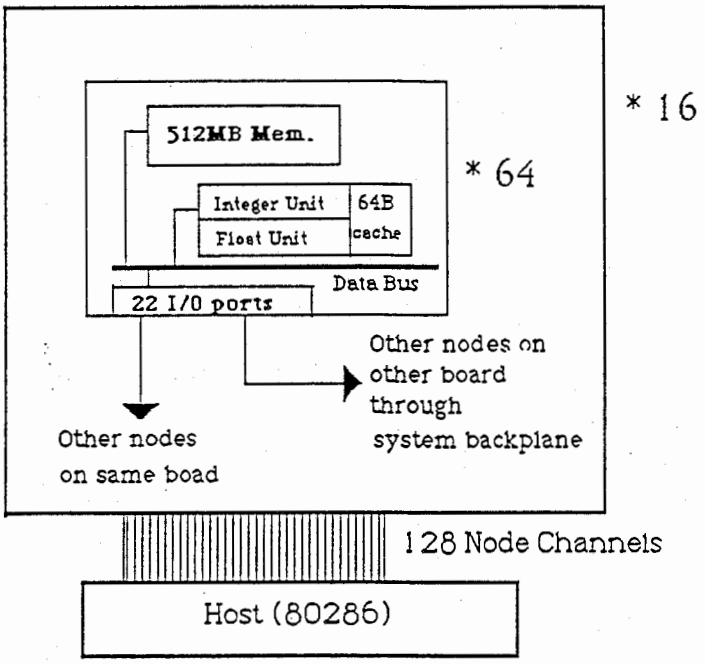


Fig 3.1

３．１　ホスト・プロセッサ

Ncube/10 のホスト・プロセッサは８０２８６で、ノード・プロセッサとユーザとのインターフェイスの役割をはたしている。ホストは、先ず必要な数のノード・プロセッサをアロケートし（アロケートするノードの数は 2^n でなければならない。）、ノード・プロセッサ用のプログラムをロードし、必要に応じてデータを転送する。各ノードはホスト経由でディスクからのデータや、ユーザからのデータを受け取る以外は全く独立している。従って、このホスト・プロセッサはある意味では Alliant のＩＰに相当すると言って良いだろう。

なお、ホストは Axis と呼ばれる Unix もどきのＯＳで作動する。また、ノードとの通信は Vortex と呼ばれる Axis の拡張システムで行われる。Vortex は、基本的には Axis のスペシャル・ファイル /dev/ncube に fread , fwrite を行っているにすぎないようだ。

３．２　ノード・プロセッサ

Ncube のノード・プロセッサは、８Ｍｈｚの一般の３２ビット・マイクロ・プロセッサに高速のＩ／Ｏチャネルを付け足したようなデザインになっている。従って、各ノードでは通常の数値計算、データ転送、制御命令のほかに、他のノードとの間での通信を行うことが出来る。

各ノードは５１２KB の物理メモリを持ち、物理メモリ・システムで作動する。また、通信用の Vertex と呼ばれる３２KB（メッセージ・バッファも含む）のシステム以外はＯＳを持たず、ユーザ・プログラムがホストからダウン・ロードされてくると、直ちにそのプログラムの実行を行う。

レジスタは、汎用レジスタが１６個、プロセッサ・レジスタが１２個、Ｉ／Ｏレジスタが２２個あり、すべて３２ビット長である。汎用レジスタは整数計算、小数計算の他にアドレス・レジスタとしても使われる。倍精度計算を行う時は、２個の汎用レジスタが一組で使われる。プロセッサ・レジスタは、スタック・ポインタ、プロセッサＩＤ、フォルト・レジスタを始めとする各種プロセッサ状態及び、ＩＯ割り込みのために使われる。（ここで言うＩ／Ｏとは、プロセッサ間通信のこと）また、Ｉ／Ｏレジスタは、他ノードとの通信を行う際のバッファ・アドレスとメッセージ長を示すのに使われる。

Ncube は前述したように大規模なパイプライン・ベクトル化は行っていないが、各ノードにおいてパフォーマンスを上げるために次のような工夫がなされている。

①小数計算の際には次のオペランドのプリフェッチを行う疑似パイプライン
②３２バイトのインストラクション・キャッシュ

ただし、この程度の機能ではとてもコンパイラ・オプティマイザの対象になるとは考えられず、結局ノード・プロセッサのアセンブラを使わなければベスト・パフォーマンスは発揮できないものと思われる。

ノード間の通信は２２個のＤＭＡ・Ｉ／Ｏチャネルによって行われる。チャネル０から９までは他ノードからの入力に、３２から４１は出力に、３１と６３はホスト及びＩ／Ｏポート（例えばＮｃｕｂｅ・グラフィック・システムのフレーム・バッファ）との入出力に使われる。（チャネル１０～３０と４２～６２は将来の拡張のためにリザーブされている）各チャネルは送受信のためのメッセージ・アドレスとメッセージ長を示すレジスタを２個づつ持っている。各チャネルのＤＭＡ転送速度は１８０MB/s で（正確には、システム・バックプレインのＩ／Ｏホードの能力が１８０MB/s）、プロセッサ・レジスタの状態に従ってメッセージの送受信、インタラプト、ペンディング・ウェイト等を、ノード・プロセッサと独立して行う。また、各ノードはブロードキャスト命令により、１１の出力チャネルから、同時に同じメッセージをＤＭＡ転送することも出来る。

Ncube を利用するときの問題点は、既存のアルゴリズムがそのまま使えるかどうかわからない、ということである。また、アルゴリズム的には並列であっても、各ノードが計算のたびに大量のデータ転送を頻繁に行うものであれば、パフォーマンスの低下は免れない。しかしながら、問題とするアルゴリズムとデータ構造をうまくハイパーキューブに変換出来れば圧倒的なパフォーマンスが可能となる。

3．3　ハイパーキューブ構造

Ncube での高速処理を実現するためには、いかにしてハイパーキューブ構造を実現するか、ということが最も重要である。以下に、その具体例及びパフォーマンスに影響を与える要因について考察する。

3．3．1　ハイパーキューブ構造の具体例

図3.2 は4次元までのハイパーキューブ構造を図示している。各ノード間の矢印はデータ転送を意味し、ノードの濃淡が受け取る順番を意味している。つまり、同じ濃淡パターンを持つノードは同時にデータを受け取ることになり、2^n個のノード全てにデータを送るのに要する転送回数はnである。各ノードはデータの転送が終わると、それぞれ並列に計算を始める。そして計算結果を矢印と反対の方向に並列に転送し、最終的に1つのノードに集められる。

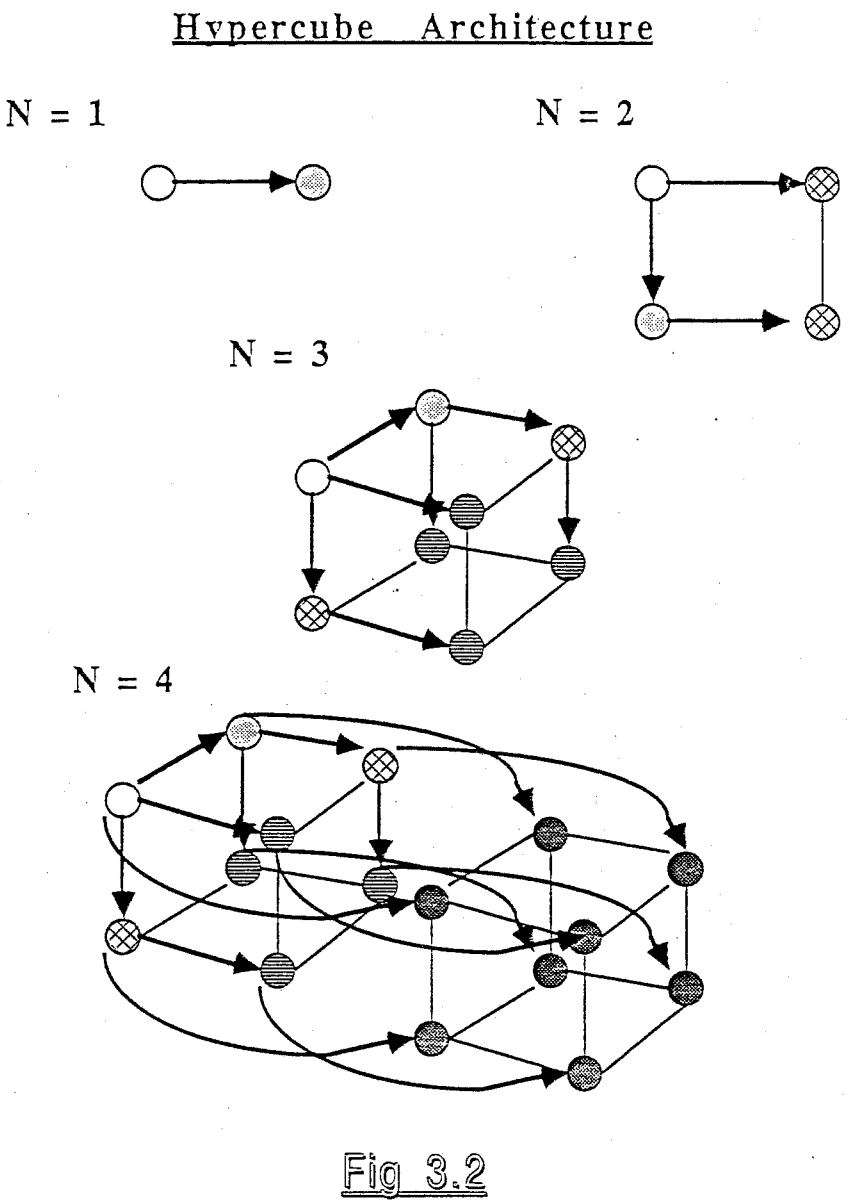


Fig 3.2

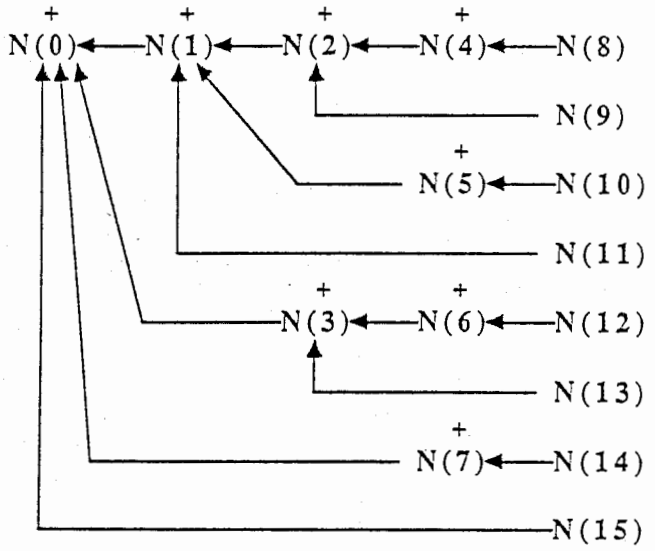
例えば何らかの積分計算をハイパーキューブ上にインプリメントするときには図3.3のように行う。このとき、各ノードは与えられた小区間での積分計算を行い、矢印方向に計算結果を送る。計算結果を受け取ったノードは、自分の計算結果と足し合わせた結果を矢印方向に送る。以下、ノード0が計算結果を受け取るまでこの転送／足し算の繰り返しが行われる。この結果、ハイパーキューブ化しなかった時に較べて、2^n-1回のデータ転送及び足し算がn回分のデータ転送及び足し算に要する時間で行われる。

このような構造を Ncube 上に実際にインプリメントするときには、ノード／ホスト上でのプログラミングに十分注意しなければならない。例えば、もし効率の良いハイパーキューブ構造をインプリメントしたいのなら、ノードの（物理的な）ロケーションにも注意を向けるべきである。なぜなら、ノード・プロセッサは６４個ごとに同一ボード上にあり、ボードを経由してのデータ転送はボード内でのそれよりも時間がかかるからである。つまり、各ボード上のノードにつけられているシーケンシャルなノードＩＤを使えば、ボード間の通信、すなわちノード 64*k+i と 64*k+j との通信は i = j = 0 の時のみ行われるようにプログラムしなければならないわけである。

下記のハイパーキューブ上におけるデータ分配のプログラム例はこのようなことを考慮して作られている。

Example of Hypercube Decomposition

$$\int f(x)\,dx = \sum \int_{x_i} f(x)\,dx$$



Result $2^n - 1$ Data Move & Add

↓

N Data Move & Add

Fig 3.3

```
#define SCATTER some-number
scatter (nodep,ar,len)
int nodep,len;
long ar[];
{ int parent,child,typep=SCATTER,children;
  for (parent = 1024;
       parent & nodep == 0;
       parent >>= 1);
  parent <<= 1;              /*親ノードを見つける*/
  nread((char *)ar,len*4,&parent,&typep,cflag);
  for (children = 1;
       children & nodep == 0;
       children <<= 1)
  { child = nodep | children;  /*子ノードを見つける*/
    typep = SCATTER;
    nwrite((char *)ar,len*4,child,typep,cflag);
  }
}
```

３．３．２　ハイパーキューブ構造のパフォーマンス[12]

ハイパーキューブのパフォーマンスに影響を与える要因には次のものがある。

①アルゴリズム

どの様なグッド・アルゴリズムであっても、並列度が１のものでは並列コンピュータを使う意味はない。例えば、多くのソートなど並列度１のアルゴリズムは並列用のものを新しく開発しなければならない。

②ソフトウェアのオーバーヘッド

いくらアルゴリズムが良くても、実際にインプリメントしたときのデータ構造が最適化されなければ、オーバーヘッドが生じる。例えばマルチ・インデックス方式のアレイ・アクセスなどは配列アクセスのたびに無駄なアドレス計算を必要とする。（そのために、これまでの Fortran コンパイラのオプティマイザは、このようなデータ構造を持つデータのアクセスを最適化するように作られている。）

③ロード・バランシング[13]

解くべき問題が静的なものである場合、全体の計算量が概算でもとまり、ＣＰＵの数で等分することが出来る。ところが一般にこのような方式では柔軟な負荷分散はむずかしく、動的にタスクを分割することが求められる。このとき、安易に動的なタスク分割しか考慮しないと、特定のプロセッサに負荷が集中する恐れがある。従って、アルゴリズム・レベルでの動的な、しかも均等な負荷分散が求められる。これをロード・バランシングという。

④通信のオーバーヘッド

Alliant や Convex のように、共有メモリ方式の密結合型並列マシンの場合、プロセッサ間のデータは共有されているので、データの通信について考慮しなければならないことは少ない。ところが Ncube をはじめとするメッセージ・パッシング方式の疎結合型並列マシンでは、データ通信の時間を無視することは出来ない。つまり、上記の問題に加えて、データのやりとりを最小限にするアルゴリズムを考えなければならない。

①はパフォーマンスを考慮する以前の問題であり、出来るだけ共有データの少ない、しかも並列度の高いアルゴリズムを選ぶべきである。②はプログラミングのむずかしさとのトレードオフになるが、使用するコンパイラのオプティマイザを熟知して、無駄のないアセンブラ・コードを出力するデータ構造、制御構造を使用しなければならない。③については本稿では述べないが、アニーリング、有限要素法、ニューラルネット等の分野で精力的に研究が進められている。現実問題としては④が最も基本的かつ必要とされるものであろう。

ハイパーキューブ構造をもつタスクの通信によるオーバーヘッドf_cは、

$$f_c = \text{計算量} / \text{通信量}$$

で表される。目的のタスクが各ノードで十分に効率よく動いていると仮定すると、スピード・アップ率Sはf_cと全ノード数Nを用いて

$$S = N / (1 + f_c)$$

と表される。従って、完全分散処理の場合はSはノード数に比例することになる。しかし、実際の問題ではこのような完全分散のタスクは少ないので、f_cやSをあらかじめ求めてから、どの様にデコンポジションを行えばよいか検討すべきであろう。

例えば、ラプラシアンによる2次微分を考えてみよう。次式

$$\nabla^2 \phi = 0$$

が各ノードにおいて、一辺の長さ$\sqrt{n}$、要素数nの2次元配列に適用されるとする。このときの実際の計算を

$$\phi^{new}_{x,y} = \frac{1}{4} (\phi^{old}_{x-1,y} + \phi^{old}_{x+1,y} + \phi^{old}_{x,y-1} + \phi^{old}_{x,y+1})$$

とすると、各ノード上の配列の周辺部のみが他ノードとの通信を必要とするため、各ノードごとに4＊n回の数値計算と4＊$\sqrt{n}$回／ワードの通信が必要となる。各ノードでの1回の数値計算に要する時間をt_{calc}、1ワードの通信に要する時間をt_{comm}とすると、

$$f_c = (4 * \sqrt{n}) * t_{comm} / (4 * n)\ t_{calc}$$
$$= 1 / \sqrt{n} \cdot t_{comm} / t_{calc}$$

となる。t_{comm} / t_{calc}はハードウェアによって一定に定まるので、スピード・アップ率は

$$S \sim N / (1 + k / \sqrt{n})$$

（kはマシンに依存する定数）

となり、ターゲット・マシンの性能に従ったデコンポジションを行うことにより、ノード数に比例したスピード・アップが期待できる。

第４章　Ａｌｌｉａｎｔ、Ｎｃｕｂｅ、Ｃｏｎｖｅｘのパフォーマンス

ユーザにとって計算機のスピードとは、その計算機のアーキテクチャ自体よりもはるかに魅力的なものだろう。しかしながら、計算機のピーク性能が必ずしもその計算機のスピードに一致しないのは周知のことである。以下に Ncube, Alliant, Convex のパフォーマンスについての考察をする。

４．１　Ｎｃｕｂｅ／１０のパフォーマンス

Ncube はそのアーキテクチャが他の計算機と根本的に異なるため、単純にベンチマーク・テスト等でパフォーマンスを推定することは出来ない。そこで、各インストラクションのクロック数から全体の能力を計算して見ることにした。表１はノード・プロセッサの主なインストラクション及びアドレッシングの時間を示す。

・データ転送　　(array1[i] <- array2[i], i=1,N)

ノード・プロセッサのアセンブラは次のようになる。

```
REP     R0
MOVW    (R2)+,(R4)+
```

表１より、REP 命令に４クロック、R0 アドレッシングは０クロック、MOVW 命令に２クロック、(R2)+ アドレッシングに７クロック、(R4)+ アドレッシングに４クロック、合計 4+2+7+4 = 17 クロックである。プロセッサのクロックは８Ｍｈｚであることから、１ワード（４バイト）のデータ転送に要する時間は１２５ns ＊１７＝２．１５us となり、各ノードのローカル・メモリ内でのデータ転送は約１．９MB/s である。

・整数計算　　(array1[i] <- array1[i]+array2[i], i=1,N)

ノード・プロセッサのアセンブラは次のようになる。

```
REP     R0
ADDW    (R2)+,(R4)+
```

表１より、REP 命令に４クロック、R0 アドレッシングは０クロック、ADDW 命令に２クロック、(R2)+ アドレッシングに７クロック、(R4)+ アドレッシングに４クロック、合計 4+2+7+4 = 17 クロックである。プロセッサのクロックは８Ｍｈｚであることから、整数型配列１要素の足し算をするのに要する時間は２．１５us で、各ノードでの整数計算能力は０．４６５MIPS、Ncube 全体で４７６MIPS となる。

・小数計算　　(array1[i] <- array1[i]*array2[i], i=1,N)

ノード・プロセッサのアセンブラは次のようになる。

```
        REP     R0;                     ! 4+0
        MULR    (R2)+,(R4)+;            ! 21+7+4
```

即ち、小数型配列１要素のかけ算をするのに要する時間は４．５us となり、各ノードでの小数計算能力は０．２２２MFLOPS、全体で２２７MFLOPS となる。

・積和計算　　(a <- a + array1[i]*array2[i], i=1,N)

ノード・プロセッサのアセンブラは次のようになる。

```
L1:     MOVW    (R0)+,R10;      ! 3+7+0
        MULR    (R2)+,R10;      ! 21+7+0
        ADDR    R10,R4;         ! 30+0+0
        REP     R6;             ! 4
        JMP     L1;             ! 5
```

即ち、積和計算の１回の和及び積を計算する時間は９．６us となり、各ノードでの小数計算能力は０．２０８MFLOPS、全体で２１３MFLOPS となる。

結局、適切なデコンポジションが施されてハイパーキュブ構造が構成されれば、Ncube/10 は実行時４５０MIPS／２００MFLOPS 程度のパフォーマンスが実現されるものと推定される。

４．２　Ａｌｌｉａｎｔ、Ｃｏｎｖｅｘのパフォーマンス

Alliant, Convex については、ベクトル化、パイプライン化、コンカレント化等があって、Ncube のようなパフォーマンスの概算は難しい。そこで、ＡＴＲにて利用できる Alliant FX/4 及び Convex C1 について簡単なベンチマーク・テストを実行してみた。表２はベンチマーク・テストの結果を示す。

Alliant FX/4 はサイクル・タイム１７０ns のプロセッサが４個並列処理を行うマシンで、最大性能は約４０MFLOPS、Convex C1 はサイクル・タイム１００ns のプロセッサ１個からなるベクトル処理系で、最大性能は同じく約４０MFLOPS である。

表２の結果からすぐにわかることは、ベクトル長の短いときのパフォーマンスの差だろう。Alliant は３２要素ごとのベクトル化を４プロセッサで並列に実行する。つまり、３２＊４以下のベクタ長では効率が悪いと言える。逆に、Convex では、１２８要素ごとのベクトル化ではあるが、短い場合でもそこそこのパフォーマンスが出ている。ベクタ長の長い場合、単純な計算ではほぼ同程度のパフォーマンスが出ている。このパフォーマンスの差は 2.5 で述べたチェイニングの有無によるものと推測される。しかしながら、積和計算、間接アドレッシング等では Alliant の方がかなりのパフォーマンスを出している。これは、ベクタ長１０００くらいの大きさでは、Alliant のデータ・キャッシュが確実にヒットするためと思われる。また、sqrt,sin,cos,log の初等関数では、Convex が圧倒的に速

くなっている。これは、初等関数については Alliant が十分なベクトル化を行っていないためと考えられる。

このベンチマーク・テストの結果からだけ考えれば、Alliant FX/4 と Convex C1 は共に実行時３～１０MFLOPS 程度であると言って良いだろう。

Integer		Float		Addressing (src)						Addressing (dst)				
Opr	Cyc	Opr	Cyc		Byte	Half	Word	Real	Long	Byte	Half	Word	Real	Long
ADD	2	ADD	28	Immediate	0	0	0	0	2	-	-	-	-	-
SUB	2	MUL	21	Rn	0	0	0	1	3	0	0	0	1	3
CALL	12	DIV	35	(Rn),(Rn)+	5	5	7	7	12	4	4	4	5	11
CMP	2	CMP	10	(Rn)++	6	6	8	8	13	5	5	5	6	12
JMP	5	NEG	10	A(Rn)	8	8	10	10	15	7	7	7	8	14
MOV	2	MOV	10	@A(Rn)	16	16	18	18	23	15	15	15	16	22
REP	4	SGN	10	A(SP)	7	7	7	9	14	6	6	6	7	13
RET	10	SQT	35	@A(SP)	15	15	17	17	22	14	14	14	15	21

Table 1 Instruction & Address mode timing of Ncube

length / Operation	Alliant FX/4					Convex C1				
	10	50	100	200	1000	10	50	100	200	1000
a(i) = b(i) + s	0.62	2.55	4.20	5.63	8.14	2.00	7.14	9.09	7.70	8.13
a(i) = b(i) * c(i)	0.58	2.25	3.52	4.4(	5.93	1.67	6.25	8.33	5.88	7.81
a(i) = b(i)*c(i) + d(i)*e(i)	1.42	4.79	6.86	7.95	9.48	1.43	3.57	5.26	3.39	3.29
s = s + a(i) * b(i)	0.39	2.57	4.35	7.71	12.9	1.25	3.33	4.35	3.18	3.07
a(i) = b(j(i)) + s	0.53	1.95	2.91	3.30	4.37	1.25	3.57	5.00	3.51	3.38
a(j(i)) = b(i)*c(i)	0.56	2.01	2.97	3.53	3.88	1.25	2.27	2.94	2.78	2.82
SQRT	0.16	0.42	0.55	0.62	0.65	1.42	3.13	3.13	3.23	3.13
SIN	0.13	0.32	0.40	0.43	0.47	1.60	2.50	3.23	3.17	3.13
EXP	0.13	0.28	0.33	0.35	0.36	1.60	3.30	3.23	3.17	3.13
LOG	0.13	0.30	0.37	0.39	0.41	1.60	2.94	3.22	3.18	3.15

Table 2 Benchmark Test of Alliant FX/4 and Convex C1
(Each number means MFLOPS)

おわりに

本稿で述べたようなミニ・スーパーコンピュータに限らず、計算機には独自のアーキテクチャがあり、そのアーキテクチャに合ったタスクを与えなければ、せっかくの計算能力が生かしきれない。或は、本稿ではふれなかったが、コンパイラによる自動ベクトル化・並列化の性能も考慮しなければならない。すばらしい計算機資源は十分に活用されるべきである。

謝辞

マシンを使用させていただいたＡＴＲ光電波通信研究所の諸氏に感謝致します。また本稿の執筆のために数多くの資料を用意していただいた、東京エレクトロンの加納氏、日本アライアントの大塚氏、住商電子の山上氏に深く感謝致します。アライアント社の Harry Jonson 氏にはシステム・アーキテクチャのあらゆる質問に即座に答えてもらい、感謝すると同時に、彼のコンピュータに対する知識の豊富さに驚かされました。また、ＳＣＣの山上雅之君にはベンチマーク・テスト等で手伝ってもらったことを深く感謝致します。

【参考文献】
[1] R.W.Hockney, C.R.Jesshope,『並列計算機』、共立出版
[2] 富田、『並列計算機構成論』、昭晃堂
[3] M.Chastain, G.Gostin, J.Mankovich, S.Wallach, "The Convex C240 Architecture"
[4] "Convex C Series Hardware Technical Overview", Convex Computer Corporation
[5] "Convex Hardware & Architecture Handbook", Convex Computer Corporation
[6] B.Funkhouser, "Understanding the convex parallel system",TAC Note, 1988
[7] M.Ben-Ari,『並行プログラミングの原理』、啓学出版
[8] "FX/Series Architecture Manual", Alliant Systems Corporation
[9] "Ncube Users Handbook", Ncube
[10] 城、緒形、『シンボリックス・ソフトウェア環境の内部構造と機能』、TR-A-0013
[11] 笠原、成田、『並列処理技術』、コンピュートロール、１９
[12] G.C.Fox, S.W.Otto, "Concurrent Computation and the Theory of Complex Systems",
[13] G.C.Fox, W.Furmanski,"Load Balancing Loosely Synchronous Problems with a Neural Network", CCCP363B

付録　　用語集

（注：　この用語集はあくまでも本稿理解のためのものであり、一般的な説明を行っているものではありません。）

アドレス・キャッシュ

アドレス変換を行うユニットにある。通常の仮想記憶システムにおいて、インストラクション・キャッシュやデータ・キャッシュがいくら効果的に働いても仮想アドレス＝＞物理アドレスの変換に時間がかかれば無意味である。アドレス・キャッシュは、この変換を（つまり、仮想アドレス：物理アドレスの組を）過去数百から数万記憶しておくことにより、変換にかかる時間を節約する。ただし、キャッシュという名前はついているが、他のキャッシュ・メモリと同じくらい高価なＲＡＭを使っているとは限らない。

アドレス変換

仮想記憶空間の任意の仮想アドレスは物理アドレスに一意に対応しなければならない。（Convex のスレッド・ページは別）従って、この変換の方法も一意に決まらなければならない。通常はページ・テーブルと呼ばれるテーブルに従って、ベース・アドレスを求め、オフセット値を加えることにより物理アドレスが得られる。これをアドレス変換という。アドレス変換には当然ＣＰＵの負荷を必要とする。このためにアドレス・キャッシュが用いられる。

アドレス・レジスタ

アドレスを指し示す専用のレジスタ。昔の計算機にはメモリのアドレスを直接指定することは出来なかった。つまり、インストラクション実行時には、どこのデータを取ってくるか、計算結果をどこにしまうか、という指定は、専用のレジスタを使わなければならなかった。現在、主流となっている仮想記憶方式の計算機では、メモリのアドレスを直接指定出来るようになっているため、アドレス・レジスタは汎用レジスタとして、データ・レジスタと同一のものになっている。ところが最近のＲＩＳＣではＳｕｎ４のようにアドレス・レジスタを使うようなものも出てきている。いろいろなアドレス修飾によってパイプラインの流れが一定しないよりは、アドレス・レジスタ方式によって安定したパイプラインを目指そうという戦略と思われる。

アドレッシング

プログラム中で任意のアドレスを指定すること。例えば、レジスタの内容、レジスタの指し示す仮想アドレス、仮想アドレスの内容を他の仮想アドレスと見なす間接アドレッシング、仮想アドレスをベース、レジスタ値をオフセットとするアドレス修飾等、いろいろあり、それぞれ目的とするデータをフェッチするのに要する時間が異なる。

アロケート

タスクの実行中にリソース（メモリ、デバイス等）を占有すること。例えば、C言語でのアロケートは malloc 以外にも、auto 型変数がある。この変数は、宣言されるブロックの実行時にユーザ・スタック上に取られる。ところで、Ncube や Connection Machine のような多数のプロセッサを持つコンピュータでは、プログラム中から使うべきプロセッサを指定しなければならない。このこともアロケートと言う。

インストラクション・キャッシュ

プログラムは通常、数十～数百インストラクションの範囲で繰り返し実行されることが多い。従って、実行中のタスクにおいて実行すべきインストラクションを、アドレス変換に従って毎回メモリから取ってくるのでは効率が悪い。インストラクション・キャッシュは実行中のインストラクションを数kバイト、キャッシュ・メモリに保存することによって、このアドレス変換にかかる負荷をなくすことを目的としている。

インターフェイス

パフォーマンスと同様、日本人が好んで用いる言葉。計算機の操作環境から、プログラムのモジュール間の接続、ハードウェア・デバイスに対するデバイス・ドライバ、ハードウェア・デバイス間のモジュール等、何でもインターフェイスである。

ウェイト命令

一般にはプロセスの実行を一時中断する。
本稿においてはセマフォに対する基本命令を意味している。

演算パイプライン

演算部におけるパイプライン。例えば、小数加算は一般に次のように5ステージで計算される。

①小数A（$m_1 2^{e1}$）、B（$m_2 2^{e2}$）の指数の大小比較（$e1 \geqq e2$とする）
②指数の桁合わせ、Bの仮数部を（$e1 - e2$）右シフト。
③AとBの仮数部の和を計算。
④仮数部の和の最上位から最初に1が現れるまでの桁数（pr）を調べる。
⑤指数部からprを引き、指数部の結果とする。仮数部はpr桁左シフト。

演算パイプラインには各ステージ間にラッチを置く同期型と、隣接ステージ間で通信を行う非同期型とがある。

仮想アドレス

仮想記憶システムにおける論理的なアドレス。通常、セグメントやページごとに分けられ、オフセット値を付け加えることにより物理アドレスに変換され、実際のデータのやりとりを行う。メモリの配置を考えなくても良いため、コンパイラ等が大変簡単に作れるようになったが、反面アドレス変換に時間がかかる。スーパーコンピュータでは仮想アドレス＝物理アドレスであるものが多い。

仮想記憶システム

ＲＡＭメモリが高価だったころの遺産。ユーザに使いもしない広大なプログラム領域を与え、ユーザはデータ領域、アドレス変換、ページング等、何も考えずにプログラムが作れるようになった。そしてそれがプログラマーの質を落とす要因となっているとの意見もある。従って、仮想記憶システムのオーバーヘッドを考えながら作られたプログラムでなければプログラムではないと言えよう。

キャッシュ・イン

キャッシュ・メモリにメイン・メモリのデータを読み込むこと。

キャッシュ・コヒーレンス

複数のプロセッサがキャッシュを共有するときに生じる問題。キャッシュの同一ブロックを、あるプロセッサは書き込みアクセス、別のプロセッサは読み込みアクセスをすると矛盾が生じる。このような場合には何らかのプロトコルに従ったアクセスをしなければならない。

キャッシュ・ヒット

プログラム中からメモリ・アクセスをする際に、目的のデータがキャッシュ・メモリ上にあればアクセスが速くなる。このときキャッシュ・ヒットしたと言う。通常、メイン・メモリのアクセスの速さは 100-200 ns/word、キャッシュ・メモリのアクセスの速さは 5-50 ns/word くらいである。当然キャッシュ・メモリの方が高価である。

キャッシュ・メモリ

プログラム中からのメモリ・アクセスには局所性がある。ループ・カウンタなどの変数はレジスタに割り付けられるとしても、その他の変数をアクセスする際に、その変数自身もしくは仮想アドレスでの近傍を再びアクセスする可能性が高いことが統計的に知られている。従って、任意の仮想アドレスをアクセスする際に、そのアドレス及び近傍のデータを高速記憶領域に置いておけば、以後のアクセスが速くなる。この高速記憶領域をキャッシュ・メモリと言い、近傍のデータをキャッシュ・ブロックと言う。キャッシュ・ブロックは通常６４～１２８バイトからなる。キャッシュ・ヒットしているかどうかのチェックはキャッシュ・メモリの内容を示すテーブルを使う。キャッシュ・ミスの場合はＬＲＵ方式などでメイン・メモリとの間でキャッシュ・ブロックの置き換えが行われる。

共有メモリ方式

並列計算機のうち、複数のプロセッサから同時に共有のメモリをアクセスできるような形態のもの。このとき、デッド・ロック等の問題点をなくすために、共有のメモリを持つプロセッサ間では高速のバスによって常に同期を取り合う必要がある。

クロック

たいていの場合、１クロック＝ＣＰＵのサイクル・タイムと考えればよい。システム内にはハードウェア・タイマーが内蔵されており、パイプライン等の各ステージ間でのラッチによる同期を実現している。従って、ゲート・ユニットを１回通るのが１クロックと言える。

クリティカル・エリア

共有メモリ方式の密結合型並列計算機の場合、各プロセッサが共有しているデータが読み込みだけの場合は問題がないが、１つでも書き込みを行うプロセッサがある場合は、何らかのプロトコルに従って各プロセッサはそのエリアをアクセスしなければならない。この危ないエリアをクリティカル・エリアと呼ぶ。データベース理論の共有ファイルに対するロックとは根本的に異なることに注意。（逐次処理マシンでの複数のプロセスの各命令はやはり逐次的に処理され、並列計算機における同時アクセスとは別のものである。）

コンテキスト

環境、基本要素、状況、状態、といった意味。例えば、プロセスのコンテキストならそのプロセスの持つ仮想空間、レジスタ、占有しているデバイス等である。

コンパイラ・オプティマイザ

高級言語で書かれた冗長な命令を、機械語に翻訳するとき必要最小限度の命令に置き換えてくれる賢いプログラム。通常は、変数のレジスタへの割付だとか、ループ文中の不必要な命令の除去等である。オプティマイザは大抵コンパイル時に指定するので、オプティマイザを指定せずに同じプログラムをどれだけ速く動くように書き換えるかというゲームもある。ところで、ＵｎｉｘのＣコンパイラはデフォルトでオプティマイザが起動しないことをあなたは知っていましたか？Ｓｕｎのオプティマイザはレベル４まで指定できるのを知っていましたか？

サイクル・タイム

ＣＰＵが一命令を実行するのに必要とする時間。普通のインストラクションはマイクロ・シーケンサによって基本命令に解釈される。この基本命令実行に必要な時間がサイクル・タイムと考えてよい。

シグナル命令

一般にはプロセスに割り込みをかけること。
本稿においてはセマフォに対する基本命令を意味している。

スカラー・ユニット

スカラー計算を実行するプロセッサの一部。通常の計算機のプロセッサと考えてもよい。ベクトル計算を専用に行わせるマシンには、このほかにベクトル・ユニットがつく。スカラー計算でも、小数演算用にＦＰＡ（Floating Point Accelerator）を装備しているものもある。

スケジューラ

マルチ・プロセス・システムのＯＳで、複数のプロセスに効率よく計算機資源を分け与えてくれる特別なプログラム。スケジューラがあほだとシステムの効率が悪くなる。密結合型の計算機の場合、それぞれのプロセッサごとのプロセスのスケジュールを行うので、スケジューラの善し悪しがシステム全体に重大な影響を与える。

スーパーバイザ・プログラム

Ｕｎｉｘのシェル、ＶＭＳのＤＣＬ、シンボリックスのリスプ・リスナー・コマンド等。ユーザ・プログラムの起動やファイル操作等、ユーザとＯＳのインターフェイス部分を担当する。ある計算機を扱えるとは、大抵そのスーパーバイザ・プログラムのコマンドをいくつか知っていることを意味する。

スペシャル・ファイル

Ｕｎｉｘのファイルのうち、デバイスやライブラリ等、通常のアスキー／バイナリ・ファイルでないもの。例えばｋｍｅｍというスペシャル・ファイルは物理メモリを含んでおり、このファイルをオープンしてリードすることにより、任意のアドレスをアクセス出来る。（ただし、へたにアクセスするとシステム・クラッシュを起こしかねない。）

スラッシング

プロセスの数にくらべて圧倒的に物理メモリが不足している場合、リソースの多くを単にスワッピングに費やしている場合が起こりうる。このことをスラッシングと言う。あるいは、ＶＭＳのように各プロセスの占有できる物理メモリの量（ワーキング・セットとかレジデント・セットと言う）が指定できるマシンで、ユーザ・プログラムの必要とする仮想空間が、そのユーザ・プロセスの占める物理メモリより圧倒的に大きい場合、割り当てられた計算リソースの多くを単にページングに費やしているような場合もスラッシングと言う。ＣＰＵを常に完全に使っているようなシステムは大抵スラッシングを起こしており、それを持ってコンピュータが有効に使われていると思う管理者が多いのは誠になさけない話である。

セマフォ

Dijkstra によって提案された、密結合型並列計算機の相互排除問題への回答。定義）セマフォｓは０以外の値を取ることが出来る整数変数である。ひとたびｓが初期値を与えられると、ｓに対して許される唯一の操作は、手続き wait(s) 及び signal(s) という基本操作を呼び出すことである。

wait(s)： もしｓ＞０ならｓ＝ｓ－１、そうでなければそのプロセスの実行を一時停止する。

signal(s)： もしプロセスＰが、セマフォｓに対する以前の wait(s) により一時停止させられていたとすれば、Ｐを起動する。
さもなければｓ＝ｓ＋１。

ただし、ここでの話は並列計算機アーキテクチャに関するもので、Alliant を使うのにセマフォを知らなくても何ら問題ない。（知っていた方がよいが）

相互排除問題

プロセスＡの動作ｘと、プロセスＢの動作ｙが重複する場合、（例えば、共有メモリの書換え）ｘとｙが同時に起こらない保証が必要である。この手の問題を相互排除問題と言う。一昔前のアルゴリズム屋さんは、卓上シミュレーションによってこれら相互排除問題への回答、すなわち並列計算機の基本動作についての研究を行っていたわけである。有名なアルゴリズムには Lamport の［パン屋のアルゴリズム、オランダ・ビール・バージョン］等が知られている。

疎結合型並列計算機

メモリを共有せずに複数のプロセッサを同時に動かせる計算機。一般にメモリ共有方式の並列計算機では、既存のアルゴリズムをどのようにタスク分割するかということが問題になっても、コンパイラがループ部分を並列化する程度のものですむことが多い。このとき、共有メモリ方式だとプロセッサ間の同期のために、プロセッサの数に物理的な制限が存在する。この制限は現在の計算機の数千、数万倍速い計算機の実現に致命的な障害となりうる。疎結合型並列計算機は、メモリを共有せずに、従ってプロセッサ間の同期を取らずに、従って一つのスケジューラの管理下に置かれない、多数の独立したプロセッサ群から構成される。このときの問題点は、共有データをどの様に扱うかということである。同期をとらないプロセッサ間の通信手段は、割り込みによるメッセージ通信が考えられるが、これではメモリ・アクセスと同程度のデータ転送は期待できない。そこで、プロセッサ間のデータ転送路に何らかの構造を入れることにより、データ転送自体を少なくしようという試みが取られている。ハイパーキューブ構造はその代表的な例である。このようなアプローチの問題点は、既存のアルゴリズムが使えないことにある。数値計算の分野で開発されてきたアルゴリズムは、逐次型処理を前提としており、計算オーダもデータの大きさのみであらわされている。疎結合／ハイパーキューブ方式では、アルゴリズムの再開発、通信量を考慮に入れた計算オーダの使用等、研究課題が積載されているのが現状であるとはいえ、さまざまな分野でハイパーキューブ構造を用いたアルゴリズムが提案されている。このように計算機の形態にそくしたアルゴリズム開発が研究される一方、未来永劫 Fortran でなければ数値計算は出来ないと信じている人が日本にたくさんいることも現状である。

ダウン・ロード

計算機のプログラム・コードは通常その計算機のディスク装置等にあり、必要に応じてメイン・メモリにロードされる。もしディスク等の外部記憶装置のない計算機にプログラムを実行させたい場合は、何らかの手段を用いてプログラムをロードしなければならない。この手段はネットワークや外部デバイスからの転送によって行われることが多く、このようにしてプログラムをロードすることをダウン・ロードと言う。例えば Ncube ではホストからノードにＤＭＡ転送によりプログラムのダウン・ロードを行い、イサーネットにつながるターミナル・サーバではホスト・ノードからネットワーク経由でＯＳをダウン・ロードしている。

タスク

計算機資源を用いて行う処理。１つのタスクは複数のプロセスによって実行されることもあるし、複数のプログラムから構成されることもある。

チェイニング

ベクトル・ユニットはベクトル演算をベクトル・レジスタの大きさごとにパイプライン処理によって実行するが、各演算ユニット間の結合を行うことによりさらに高速化出来る。例えば、

X(I) = A(I) * B(I) + C(I) , (I=K*L, L はベクトル・レジスタ長)

はクロック数をｎとすると２＊ｎ＊ｋの時間で処理される。（ただし、和算・乗算が１クロックで実行出来ると仮定する。）このとき、最初のベクトル・レジスタ間の乗算の直後、２つ目のベクトル・レジスタ間の乗算と同時に最初の乗算の結果に対する和算を行えば、処理速度はｎ＊ｋとなる。このような手法をチェイニングという。これは演算どうしだけではなく、演算とロード／ストアに、ロード／ストアどうしでも可能である。いずれにしても Convex のようにチェイニング機能をそなえたベクトル・マシンでのコーディングでは、演算順序をくふうするだけでパフォーマンスの改善が期待できる。

デコンポジション

与えられた問題を並列計算機上で実現するために、タスク分割を施すこと。一般に、与えられた問題は逐次型であることが多いため簡単にはデコンポジションできない。あるいは、そもそも分割不可能なアルゴリズムもある。（無理やり分割するとかえって遅くなるという意味。）

データ・キャッシュ

キャッシュ・メモリのこと。アドレス・キャッシュと明確に区別するためにこのように呼ばれる。

デッド・ロック

二つのプロセスが一つのリソース獲得をめぐってにらみ合っている状態。相互排除問題で失敗すると、デッド・ロックに陥るか、片方のプロセスが全くリソースを利用できなくなる。

転送パイプライン

ベクトル・データをメイン・メモリからベクトル・レジスタに転送する際のパイプライン。メモリ・バスに対するアドレッシングとデータそのものの転送を数ステージに分割してパイプライン処理を行う。安定した転送パイプラインはメモリ・インターリーブによって達成される。また、転送パイプラインと演算パイプラインがチェイニングされることにより全体として高速のベクトル処理が行われる。

トラップ

ある事象が発生したときに、実行中のプロセスを強制的に一時停止させ、例外処理を行わせること。通常はエラー処理に用いられることが多い。もちろん、ユーザ・プログラムからトラップ（罠）を仕掛けることもある。

パイプライン

一つの処理を行う時に、もしその処理が数段階のステージで分割できれば、流れ作業を行わせることにより数倍速くなる。パイプラインは処理として、命令部分に関するところ、データ転送に関するところ、演算に関するところの３種類があり、それぞれ数段階のステージに分割され、高速処理が達成されている。この３種類はそれぞれ、命令パイプライン、転送パイプライン、演算パイプラインと呼ばれている。ワーク・ステーション程度の計算機は命令パイプラインを、汎用計算機ではそれに加えて転送パイプラインを、スーパー・コンピュータではさらに演算パイプラインを用いている。

バス

データの転送路。例えばメイン・メモリとプロセッサの間にはメモリ・バスがあり、このバスが遅ければシステム全体のパフォーマンスが悪くなる。その他、各種デバイスとの間のバス、プロセッサ間のバス（並列計算機の場合）等がある。ユニ・バスと呼ばれるバスはＤＥＣ社が開発した汎用のデバイス・バスで、その仕様が公開されているため様々な周辺機器がユニ・バス用に作られ、ＰＤＰ，ＶＡＸというベストセラー機種が生まれた。最近のはやりはＶＭＥバスのようである。

フェッチ

メモリからデータを取り込むこと。命令パイプライン、転送パイプラインでのステージの一つ。

フォーク

Ｕｎｉｘのシステム・ルーチン。あるプロセスがプロセスＩＤを除いて自分と同一のプロセス・コンテキストをもつプロセスを作り出すこと。

物理アドレス

物理メモリ（メイン・メモリ）上のアドレス。物理メモリ・システムにおいては任意のアドレスがそのまま物理アドレスで、仮想メモリ・システムではアドレス変換が行われた結果のアドレスが物理アドレスである。当然、物理アドレス・システムの方が速い。現在、Ｓｕｎ４等のワーク・ステーションでは数十ＭＢのメイン・メモリをもつものがほとんどである。なぜ物理メモリ・システムにしないのだろうか。ＲＡＭが安くなった現在において、仮想メモリ・システムにこだわるのは納得いかない。

プロセス

計算機の処理環境の基本要素。マルチ・プロセス・システムにおいては、計算資源はユーザ・プロセスとスケジューラ及びシステムのデーモンによって使われる。（全く使われていないと空ループを回っている。）スケジューラは全てのプロセスへの計算資源の割当を決める。そして、与えられた計算資源を各プロセスが使うわけである。ユーザは計算機にタスクを実行させるときに１つ以上のプロセスを使う。これらのプロセスはそれぞれ独自の処理環境をもち、与えられたタスクの一部もしくは全部を処理するわけである。

プロトコル

一対一もしくは多対一の通信を行う際の手順。密談をするときの合言葉のようなもので、一般ユーザにはあまり関係ない。

ページ・アウト

物理メモリがすべて使われているとき、あるプロセスが新しい仮想アドレスもしくはページ・ファイルにある仮想アドレスをアクセスすると、物理メモリ中のあるページをページ・ファイルに保存して、新しいページを物理メモリ中に確保しなければならない。このようにあるページをページ・ファイルに退避させることをページ・アウトと言う。

ページ・テーブル

仮想アドレスから物理アドレスを求めるためのテーブル。ページ・テーブルの要素は別のページ・テーブルを指していたり、物理メモリのページ番号を指していたりする。仮想アドレスはこのページ・テーブルを指定する部分と、最終的に得られたページに対するオフセット値を指定する部分に分かれている場合が多い。

ページ・ファイル

すべてのプロセスの仮想空間の総和が物理メモリより多い場合、各プロセスは不必要なページを適当に物理メモリから退避させなければならない。ページ・ファイルはこれらのページが退避させられる場所である。

ページング

すべてのプロセスの仮想空間の総和が物理メモリより多い場合、各プロセスは不必要なページを適当に物理メモリから退避させなければならない。これをページングという。ページングには当然時間がかかるので、パフォーマンスを大幅に低下させる。従って、ある程度大きなプログラムを作るときには参照の局所性に注意すべきである。

ベクトル化

数値計算のプログラムでは配列計算に要する時間がその大半を占めている。そして、一次元の配列はもちろん、多次元の配列もベクトルの基本演算で表せるので、ベクトル演算さえ高速に出来れば数値計算を高速に行える。このためにベクトル・プロセッサが開発され、通常のプログラムをベクトル計算ように変換することをベクトル化という。ベクトル化は通常 Fortran コンパイラによってなされるが、ベクトル・ルーチンがユーザ・コーラブルであれば、自分でベクトル化した方が良いだろう。

ベクトル・ユニット

ベクトル・プロセッサ内蔵の計算機は、スカラー計算用のスカラー・ユニットとベクトル計算用のベクトル・ユニットを持つ。ベクトル・プロセッサはスカラー・プロセッサと同様、命令解読部からの命令に従ってベクトル計算を行う。ベクトル・プロセッサはベクトル・ユニットに1つとは限らない。例えば、Convex は2並列のベクトル・ユニットを持っている。また、独立したベクトル・ユニットが複数ある場合、その個数を多重度と言う。例えば、Alliant は8多重のベクトル・プロセッサを持つ。

ベクトル・レジスタ

ベクトル・プロセッサのためのレジスタ。ロード・ストア・ユニットはベクトル・レジスタと呼ばれる数十～数百バイトのレジスタにデータを読み書きし、ベクトル長の回数だけ演算パイプラインによるベクトル演算を行う。ベクトル・レジスタのロード・ストアとベクトル演算はチェイニングされている場合とされてない場合とがある。

マイクロ・インストラクション

マクロ・インストラクション

プログラム中で指定されるインストラクションは命令解読部でフェッチされ、さらに詳細なインストラクションに変換されてから、スカラー・ユニットやベクトル・ユニット、アドレス・ユニットに渡される。通常、フェッチされるものをマクロ・インストラクション、解読された結果をマイクロ・インストラクションと言う。

密結合型並列計算機

メモリを共有し一つのスケジューラの管理下で互いに同期をとりながら複数のプロセッサを動かせる計算機。共有データ部分のアクセスさえ正しく行われれば並列化が比較的容易なため、コンパイラ・レベルでの並列化が研究され、実現されてきている。しかしながら、現在の数倍の性能を目指すならともかく、数百倍、数千倍の性能を目指すにはプロセッサ間の同期の問題から、密結合型並列計算機では無理がある。そのため、現在の密結合型並列計算機は数台のプロセッサにベクトル・プロセッサを内蔵して、外側のループを並列化し、内側のループをベクトル化する手法が取り入れられている。疎結合型並列計算機が既存のアルゴリズムを作り直さなければ使えないことを考えると、ここ数年は密結合型並列計算機全盛の時代となると思われる。

命令解読

与えられたインストラクションをプロセッサの基本的な命令に解読すること。命令パイプラインの重要な１ステージ。

命令パイプライン

最も一般的なパイプライン。インストラクションのフェッチ、解読、分配をステージごとに行う。フェッチ及び分配はキャッシュの問題もあるが、ある程度は予測出来る。ところがマクロ・インストラクションが複雑であるほど命令解読に必要な時間が予測しにくい。この点を考えるとＲＩＳＣの簡単な命令セットは言うまでもなく命令パイプラインを効率よく働かせる。ところがコンパイラの設計を考えると、ＲＩＳＣに問題がないわけではない。これまでコンパイラ・オプティマイザは、マシンに特有の命令をいかにうまく活用するかという目的で研究されてきたが、これまでのマシン同様、ＲＩＳＣベースのマシンのためのコンパイラ・オプティマイザが実現されるべきである。そして、そのオプティマイザは、おそらく命令パイプラインとレジスタ割当に工夫をこらしたものとなるであろう。

メッセージ通信

独立した２つのプロセスの間の通信手段。通常、キュー構造をもち非同期に通信を行う。ＵｎｉｘのＩＣＰルーチン、ＶＭＳのメイル・ボックス等が有名である。この考えを拡張して、疎結合並列計算機では独立したプロセッサ間での通信手段としてメッセージ通信が使われる。そしてこの通信路をもとにしてハイパーキューブ構造などが構築される。

メモリ・インターリーブ

プロセッサのサイクル・タイムがいくら速くなっても、データ・アクセスに時間がかかれば全体のパフォーマンスはよくならない。例えば、10ns のプロセッサに、アクセス・タイム 100ns のメイン・メモリをつけたとすると、90ns の無駄が生じる。もちろん、キャッシュ・メモリ等の高速化の手法もあるが、これだけではメモリ・アクセスのボトルネックは解決しない。理想的には 10ns のアクセス・タイムのＲＡＭを使えば良いのだが、コスト・パフォーマンスの面で無理がある。そこで、メモリ・アクセスの並列化が考えられる。つまり、メモリは大抵シーケンシャルにアクセスされるという性質を利用して、これを複数のバンクに分け、独立したメモリ・コントローラで並列にアクセスすることにより、全体のアクセス・タイムを短くすることが出来る。これをメモリ・インターリーブと言う。当然メモリ・バンクがたくさんあるほど高価になるが、パイプラインを安定に保つという観点からすると、プロセッサのサイクル・タイムにつき１ワードのアクセスが出来る程度のバンク数にすべきである。また、多次元配列をデータとして使う場合、メモリ・インターリーブの数を考慮にいれて配列宣言するべきである。例えば、（５１２、５１２）の配列に対し、メモリ・インターリーブが８だとする。このとき配列ｙ成分（ＣやＬｉｓｐのときはｘ成分）のアクセスをシーケンシャルに行うと、特定のバンクにアクセスが集中し、全体のパフォーマンスを低下させる。

ユーザ・スタック

ユーザ・プログラム・エリアにおいて、仮想空間の最後から作られるスタック構造。リカーシブ・コールなどにおいてコール・フレームを作ったり、動的な記憶割付に使われたりする。もともと、汎用レジスタが数個しかなかった時代に、レジスタの値の保管場所として使われていたのだが、現在のＲＩＳＣでは多くのレジスタがあるため、そのような意味での使われかたはなく、コール・バイ・スタックにしても、レジスタ・ウィンドウ等を用いた高速手法が取り入れられている。

優先順位

複数のプロセスが稼働するシステムにおいて、計算資源を有効に活用するための各プロセスに割り当てられた順番。例えば、リアルタイムのレスポンスが要求されるプロセスでは当然優先順位は高い。逆に、計算のみが果てしなく続くようなジョブでは優先順位を低くするべきである。これらの優先順位はスケジューラによって動的に変化しているが、その変化に制限を設けてジョブを動きやすくしたり、動きにくく指定したり出来る。これは通常システム管理者にゆだねられていることが多いわけだが、システムを把握できないシステム管理者はユーザの計算環境に制限をもうけた方が節約になって良いだろうといった妄想を抱くことが多く、そのため稼働中のプロセスの優先順位を不当に低くし、あたかも自分がシステムを把握しているとの幻想を持ち、システム全体のスループットを下げていることに気が付かないことがある。そのようなシステム管理者にとってのユーザの優先順位は最低なのだろう。

ユーザ・スタック

ユーザ・プログラム・エリアにおいて、仮想空間の最後から作られるスタック構造。リカーシブ・コールなどにおいてコール・フレームを作ったり、動的な記憶割付に使われたりする。もともと、汎用レジスタが数個しかなかった時代に、レジスタの値の保管場所として使われていたのだが、現在のＲＩＳＣでは多くのレジスタがあるため、そのような意味での使われかたはなく、コール・バイ・スタックにしても、レジスタ・ウィンドウ等を用いた高速手法が取り入れられている。

ライト・スルー

ライト・バック

キャッシュ・メモリの任意のアドレスはメイン・メモリのあるアドレスに必ず対応しているわけだが、そのアドレスに対する書き込みのアクセスがあったときの対処のしかたが、ライト・スルーとライト・バックである。ライト・スルーはキャッシュへの書き込みと同時にメイン・メモリも書き変わる方式で、ライト・バックはキャッシュ・ブロックが退避されるときにメイン・メモリを書き換える方式である。

リスプ・マシン

記号処理言語ＬＩＳＰを高速に処理するために、ＭＩＴのハッカー達によって作られた究極のハッキング・マシン。近年のＡＩブームにもかかわらず、依然としてリスパーは少なく、高性能ワークステーションの追い上げにもあい、リスプ・マシンは絶滅の危機に直面している。結局、どんなにすぐれたマシンでも、それを使うユーザの理解の範囲を超えていてはいけないと言うことなのだろうか....

ロード・バランシング

複数のタスクを計算資源の無駄がないように効果的に行うこと。特に、デコンポーズされたそれぞれのタスクを動的に再配置し、各プロセッサの負荷を均等にするようなアルゴリズムを指すことが多い。

ＤＭＡ

Direct Memory Access。ディスク装置をはじめとする各種デバイスには、デバイス・ドライバと呼ばれる特別なプログラムが付随していて、プロセッサに負荷を掛けながらデータ通信を行うことが多い。ＤＭＡとは、送るべき（受け取るべき）データの先頭アドレスと長さを指定するだけで、以後のデータ転送はハードウェアで行う方式で、特別なディスク入出力からフレーム、バッファ、疎結合並列計算機のノード間の通信等、様々なところで使われている。

ＭＩＰＳ

Milion Instruction Per Second

一秒間に百万回の命令を実行出来るマシン性能。ただし、命令によって実行時間が異なるため、単なるベンチマーク・テストではＭＩＰＳ値はわからない。正確には、プログラム中にあらわれる各命令（整数演算、小数演算、論理演算、制御命令、その他特殊命令）の統計的な頻度にもとずいた値を算出すべきである。もちろん、そのようなことはむずかしいので、その代わりとして、代表的な１ＭＩＰＳマシンをＶＡＸ７８０とし、相対比で算出することが多い。

ＦＬＯＰＳ

Floating Operation Per Second

ＭＩＰＳ値は前述したとうり、あやふやな側面があってマシンの性能を必ずしも表せない。そこで、数値計算を行うマシンの性能の尺度として、小数演算命令をどれだけ行えるかというものが考えられた。ただし、メーカー公称値としてのピーク性能は、ベクトル・ユニットがパイプライン及びチェイニングを理論どうりに行えばこれだけの小数演算が可能なはずである、という希望的な値であり、ハンド・オプティマイズを何ら施していないプログラムではピーク性能の１０分の１程度しかでないのが普通である。一般に、ベクトル／コンカレントのマシンでは、コンパイラ・オプティマイザが最適化してくれる部分以外に、パイプライン、チェイニング、メモリ・インターリーブ、ベクトル長、ループ中での条件分岐、コンカレント・コールのオーバーヘッド、プロセッサ間の同期、共有メモリの管理、ページング、....などのオプティマイズの対象があり、これらをすべて把握しておかなければ効果的なプログラムは作れない。

ＭＣＰＳ

Mega Connection Per Second

ＦＬＯＰＳ値は前述したとうり、あやふやな側面があってマシンの性能を必ずしも表せない。そこで、ニューラルネット・シミュレーションを行うマシンの性能の尺度として、１秒間にどれだけのコネクションを処理できるかというものが考えられた。この値は現在のところ公式なものではなく、プログラム・レベルでのオプティマイズに左右されることもあって、全くあてにならない。

ＲＩＳＣ

Reduced Instruction Set Computer

数年前までの計算機の歴史は、コンパイラの必要とする複雑な命令をいかにしてアーキテクチャに組み込むか、というものであった。例えばＶＡＸのマシン・インストラクションには、他項式の計算を行わせたり、ＣＲＣチェックを行わせたりするものもある。あるいは２次元配列のアドレッシングを指定できるアドレス修飾もアーキテクチャの中に含まれている。このような方式では、ＣＰＵのクロックを上げることはむずかしく、命令パイプラインを安定に保つことも困難である。ＲＩＳＣはこれらの風潮に逆行したものである。つまり、マシン・インストラクションやアドレス修飾は最低限必要なものだけを用い、簡単な命令をバランス良く実行させようというものである。この結果、ＣＰＵのクロックが上げやすくなり、レジスタが大量に確保でき、安定した命令パイプラインが可能となる。もちろん、既存のコンパイラ・オプティマイザが使いものにならなくなるという問題点もあるのだが....