Vol. 35 No. 11　　情　報　処　理　　Nov. 1994

解　説

データ駆動による細粒度並列処理†

山　口　喜　教††

1. はじめに

本稿では，細粒度の並列処理をサポートするアーキテクチャという観点から，データ駆動のモデルやそれに関連して研究が行われた並列計算機のアーキテクチャについて解説する．データ駆動計算機は，後述するように，データの待合せと処理を基本とした並列計算モデルに基づいた計算機である．データ駆動原理と呼ばれる並列実行モデルに基づいたデータ駆動計算機は命令を単位とした粒度の細かい並列性を引き出して，細粒度の並列処理を行うことを可能とした．データ駆動計算機の研究は，このようなデータ駆動のモデルに基づいた純粋なデータ駆動計算機の研究を経て，その発展型のアーキテクチャとして研究が展開されている．本稿では，このようなデータ駆動の考え方を基にして細粒度の並列処理を行うためのアーキテクチャとして研究が行われているものを中心に解説を行う．ここでは特に，並列処理の基本的な要素のうち，同期機構とデータ構造に対するアクセス機構に焦点をあてることとする．データ駆動計算機における同期の機構については，一般的にはデータの待合せによる命令の発火機構がよく知られているが，より幅広く並列計算を実現するためにはデータの到着を待ち合わせるばかりではなく，データの存在する場所（メモリ）で同期をとる必要が生じる．なぜならば，データ駆動計算機にみられるような，ある命令と別の命令の実行順序が本質的に非同期的に決定されるものにおいては，データへの書込みと読出しの順序を規定することが不可能であるからである．

本稿では，まずデータ駆動の原理とその一般的なアーキテクチャについて紹介することによってデータ駆動計算機における細粒度並列処理の考え方を述べ，次にデータ駆動計算機から派生したマルチスレッドの細粒度並列計算機の考え方と，そのような新しい考え方を進めたいくつかの実行モデルについて紹介する．次いで，データ駆動に基づいた細粒度並列処理におけるデータ構造アクセスに関して述べる．最後に，細粒度並列処理における最適実行のためのソフトウェア技術としてデータ駆動に関連するコンパイラ技術について触れる．

† Fine Grain Parallel Execution by Data-Driven Principle by Yoshinori YAMAGUCHI (Computer Science Division, Electrotechnical Laboratory).

†† 電子技術総合研究所情報アーキテクチャ部

2. データ駆動計算機の実行方式

本章では，データ駆動的な細粒度の並列処理を理解するために必要となる，データ駆動計算機の基本的な原理について紹介し，その原理を実現するための基本的なアーキテクチャについて述べる．データ駆動計算機に関しては，すでにいくつかの解説[26),27),29),31)~33)]があるので，本稿では基本的な考え方と方式を紹介するにとどめることとする．

2.1 動作原理

データ駆動計算機においては，プログラムを構成する各命令は，それぞれの実行に必要な引数データ（本稿ではトークンと呼ぶ．また，実際のハードウェア上でやり取りされるトークンのことを本稿ではパケットと呼ぶ）がすべて到着すると，実行可能な状態になる．実行可能な状態になった命令は，その引数データおよび実行結果の宛先とともに，命令実行機構に送られる．そこでは，該当する命令が実行され，その結果データに宛先が付加される．結果データはその宛先に従って次に実行すべき命令の引数データとして転送される．データの移動によって命令の実行が起動されることから，このような計算制御機構はデータフロー (dataflow) 方式とも呼ばれている[5),33)]．したがって，データ駆動計算機はデータフロー計算

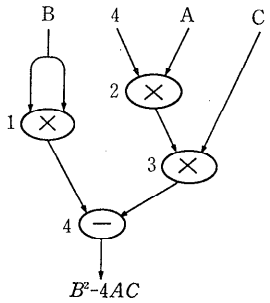


図-1　データ駆動図式の例

機とも呼ばれる．

データ受渡しの依存関係を図示したものがデータ駆動図式であり，データ駆動計算機のプログラムである[5),33)]．**図-1** に B^2-4AC のデータ駆動図式による記述例を示す．データ駆動の実行原理は，データ駆動図式において左右の入力アーク（枝）からトークンが到着すると命令は発火し，演算結果（命令実行結果）のデータをトークンとして出力アークに送出するというものである．図の B^2 の計算（番号1）はBのアークにデータが到着すると実行される．$4A$ の計算（番号2）もAのアークにデータが到着すると，片方は定数だから実行可能となる．これら二つの命令の実行は並列に行うことができる．しかし，残る二つの命令は，それぞれの引数データが入力アークに到着しないと実行できない．このように，データ駆動方式では，命令の実行順序はデータ受渡しの依存関係によってのみ定まる．

データ駆動図式のノードのもつ機能を並列計算機の要素プロセッサに対応させると，抽象的なデータ駆動計算機のおのおののプロセッサにおける処理の内容は以下のようになる．

1. 到着するトークンを一時的にバッファに格納する．
2. 到着するトークンの待合せを行う．
3. 待合せが成功すると，指定された命令をプログラム記憶から読み出しそれを実行する．
4. 得られた結果データに送り先のノードを示す宛先情報を付加し，送り出す．

各ノードを要素プロセッサに対応させると，要素プロセッサはパケット形式で受け取ったデータに対して上記の実行サイクルを繰り返すことになる．入力データの発信源あるいは出力データの送り先は，自分自身である場合もあるし，相互結合網を経由した他の要素プロセッサである場合もある．この実行サイクルの各機能をハードウェアによって構成したものがデータ駆動計算機である．

2.2　基本的なアーキテクチャ

データ駆動計算機の方式は静的アーキテクチャ[5),6)]と動的アーキテクチャ[3),9),10),20)]に分類される．静的アーキテクチャではデータ駆動図式における各アークには高々1個のトークンしか存在することが許されない．これに対して，動的アーキテクチャでは同じアーク上のトークンにタグを付加して実行環境を区別し，各アーク上に複数のトークンが同時に存在することを許す．このため，データ駆動図式におけるノードとして表現される命令を多重化して使用することができる．多重化に用いられる識別子情報はカラーと呼ばれ，データ駆動図式の各アーク上に同時に存在しうるトークン数は可能なカラー数と等しくなる．これによって，動的データ駆動方式では再帰的なプログラムを実行することが可能になる．また，手続き呼出しおよび繰返し文などの実行において，並列に動作する計算要素をカラーによって区別する制御機構の導入が容易になる．

動的なデータ駆動計算機の一般的なアーキテクチャの構成☆を **図-2** に示す．システム全体は多数台の要素プロセッサを並列配置して構成される．入力バッファは，先入れ先出しのバッファでありパケットが到着する間隔の不規則性を平滑化する

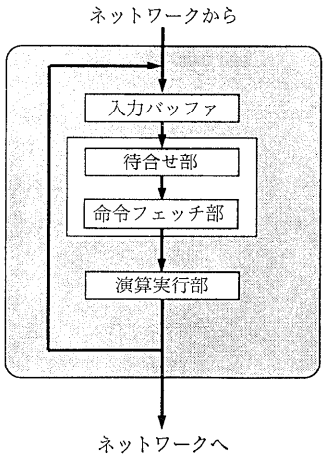


図-2　動的データ駆動計算機での要素プロセッサの基本構成

☆ ここでのアーキテクチャの構成は単純化した形で示してある．たとえば，構造記憶部という配列など構造を有するデータを効率よく処理するための専用記憶を含む場合もある．

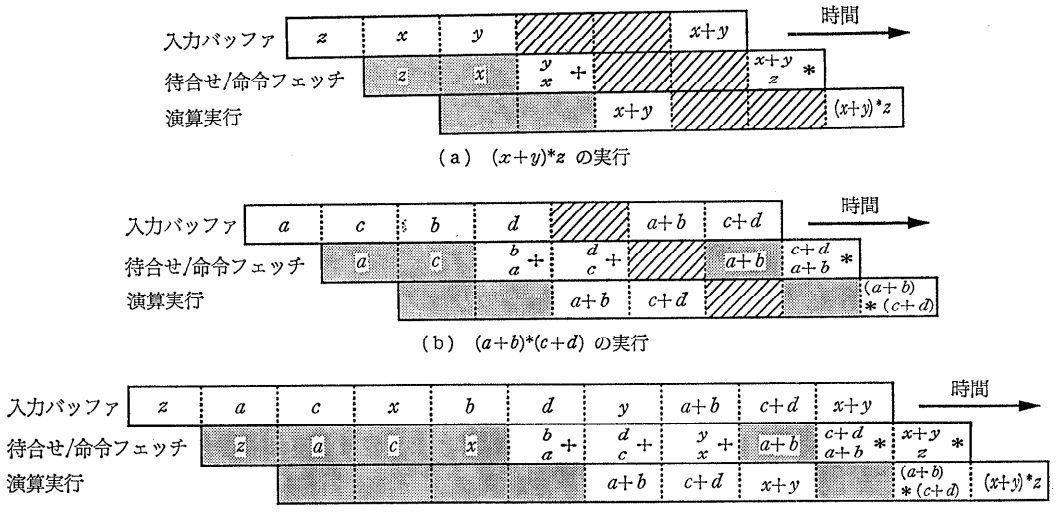


図-3 データ駆動計算機におけるパイプライン実行

働きをする．2個の入力引数データ（トークン）を必要とする2入力命令においては，一方の引数データが他方のトークンの到着を待ち合わせ，待合せが完了すると命令フェッチ部に送られる．待合せ部は，このような待合せ処理を行うユニットであり，通常ハッシングハードウェアなどの連想記憶によって実現される．待合せ部には待合せ記憶があり，待合せを行っているトークンはここに格納される．演算実行部は与えられた演算を次々と実行するユニットである．各ユニットはパイプライン的に独立した動作を行い，要素プロセッサ内部で閉路（循環パイプライン）を構成する（待合せ部と命令フェッチ部を，並列的に処理する方式もある）．

2.3 データ駆動によるパイプライン動作

細粒度の並列計算機は，実行可能な処理単位に対して次々と実行の制御を移すことが可能である必要がある．データ駆動計算機の基本的な制御構造はこのような性質をもっており，しかも命令単位で処理を切り替えられるという特長がある．しかしながらその反面，実行効率の点でいくつかの問題点も有している．このことを，簡単な例で示そう．**図-3**は，データ駆動方式のパイプライン実行の様子を示したものである．このアーキテクチャは簡単化のため，待合せ処理と命令のフェッチが並行して行われると仮定している．図-3において横方向は時間軸上のスロットを表し，縦方向は図-2に示したパイプラインの各段の実行状況を表している．入力バッファの各項目に示された値は，その値が新たにバッファに入力されたことを示している．また，待合せ／命令フェッチに示された値で，x や y のように一つの値しかないところは，待合せが不成功であったことを示し，それ以外は待合せが成功して，命令が読み出されたことを示している．演算実行の項目は，演算が行われるスロットだけ実行結果が表示されている．図-3において，図-3(a)は，$(x+y)*z$ を実行したときのものである．この例では，演算部は6クロック中2クロックしか，働いていない．また，図-3(b)に示すように $(a+b)*(c+d)$ を実行したときには，演算部は7クロック中3クロックしか働いていない．これは，基本的には次の二つの原因による．

1. 演算処理の結果（トークン）によって次の命令を発火させるために，パイプライン実行による遅延がある．

2. 待合せが不成功に終わった後は，命令を実行できない．

図-3において，斜線をほどこした部分は 1. に対応し，網かけをほどこした部分は 2. に対応している．このうち，1. に関しては，独立した複数の計算を同時に実行すれば，次々に演算可能な処理を行うことによって，演算実行の空きをなくすことができる．実際に，図-3(c)に示すように，$(x+y)*z$ と $(a+b)*(c+d)$ の計算を同時にデータ駆動計算機で実行した場合には，1. による演算部の空き時間は生じない．しかしながら，このよう

にしても 2. の原因による演算部の空きは解消されない．また，(c)は理想的な状況でトークンが到着し，パイプライン実行による遅延を隠蔽することができた場合の例であり，このような状況にならない場合には，やはりパイプラインに空きが生じる．このような点に加えて，純粋なデータ駆動計算機は待合せ処理の効率化の限界，パケット出力速度の限界などにより，演算実行部の空き時間が長くなり，実質的に演算実行部の稼働率が低く抑えられる可能性が高いことが指摘できる．

3. データ駆動に基づく細粒度マルチスレッド処理

一般的に，細粒度の効率的な並列処理を考えた場合には，循環パイプラインに基づく純粋なデータ駆動計算機においては，(1)先行制御を行うことが不可能，(2)待合せ処理のオーバヘッドが大，(3)レジスタの利用が困難，(4)細かいピッチのパイプラインを作るのが困難，などの欠点をもつ．このような欠点を克服し，データ駆動方式を発展させるものとして研究されているアーキテクチャは，次のような考え方に基づいている．

1. データ駆動の原理と実行の機構を必ずしも一致させず，データ駆動の原理を踏襲しつつ，最適な実行を目指す．

2. 計算の局所性を有効に活用し，不必要な同期を排除する．

このようなアーキテクチャの特徴を表すために，実行環境を有するプログラムの実行単位としてスレッドというものを考える．本稿では，スレッドとはある要素プロセッサ内において局所的な実行環境をもつ一連の命令集合を指すものとする．スレッドは，ある要素プロセッサに割り付けられる実行の単位であり，途中で実行が中断される場合には，中断された計算の実行を再開することが可能な実行環境を保持し，必要な時点でその実行を再開できる性質を備えている．このような性質をもつ複数のスレッドを複数の要素プロセッサによって効率良く実行する並列計算機のことをマルチスレッド計算機と呼ぶ．この場合に一つの要素プロセッサ内でも複数のスレッドの実行を切り替えながら処理が行われることはもちろんである．ここで，実行環境をもつスレッドをどのような大きさの形態として与えるかという問題がある．たとえば，手続きや関数の実行などをも包含し，仮想的に独立したプロセッサで処理されるプロセス的なものを一つのスレッドとして考える場合もあるが，ここでは関数や手続きを単位とするもっと粒度の小さい実行形態をスレッドの単位とし，細粒度スレッドと呼ぶ．このような細粒度スレッドを複数の要素プロセッサによって効率良く並列処理することを可能とするアーキテクチャが細粒度マルチスレッド計算機である．データ駆動原理から派生した細粒度マルチスレッド計算機は，データ駆動的な並列処理の実行機構を拡張し，局所的なスレッドの処理を効率良く実行することを目指した方式として捉えられる．

マルチスレッドによって効率良くプロセッサを働かすためには，たとえば大域的なメモリ参照によって計算が中断されても，プロセッサがアイドル状態になることを防止する必要がある．このためには，現在実行中とは異なるスレッドに制御を移すことが可能であればよい．このような条件を満足するためには，(1)大域的なメモリ参照や他のプロセッサからのメッセージ応答がスレッドの実行を再開するための実行環境 (continuation) を表し，(2)他のスレッドへの切替えのコストが十分小さい必要がある．スレッドの生成およびスレッド間の同期のコストが小さければ小さいほど，より小さな粒度でマルチスレッドによるスレッドの切替えを行うことができるようになる．

以上の点は，データ駆動的な視点から次のようにまとめることができる．すなわち，データ駆動による実行制御の本質は，計算実行の同期とデータの受渡しを同一視することにある．データ駆動制御においては，これらの処理は各命令に埋め込まれているが，この二つを，次のように分離して考えることによって，より柔軟な制御機構と計算の局所性を生かすことが可能となる．

1. スレッドの中のある局所的な計算領域では，データの受渡しによる同期を行わない．すなわち，静的にスケジューリングされた同期やデータの受渡しを行う．

2. スレッド間においては，動的な同期によってデータの受渡しを行う．

これらの細粒度マルチスレッド計算機における，スレッド間の同期とスレッドの実行の切替えの方式は，**図-4**に示すように，おおよそ次の三つ

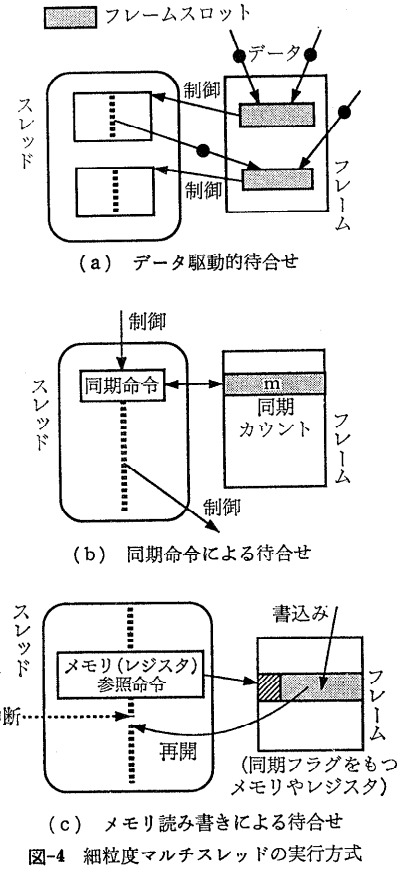


図-4 細粒度マルチスレッドの実行方式

に分類できよう．この中で，関数，手続き，ループなどの一つのまとまった処理の実行環境を保持するための領域をフレーム (frame) と呼ぶ．関数や手続きなどの実行にともなって起動されるスレッドはその起動されたスレッドに対応する同一のフレーム内にアクセスすることになる．さらに，フレーム内で個々のデータを保持する領域（レジスタやメモリ）のことを，フレームスロット (frame slot) と呼ぶことにする．

1. フレームスロットが待合せフラグをもち，スレッドの実行の起動やスレッド間の同期がデータ駆動の待合せと同じように，フレームスロットでの待合せによって行われるもの．電子技術総合研究所の EM-4[16),23),24)]，や MIT の Monsoon[15)]はこの方式に分類される．

2. 同期用の命令を実行するもの．この場合に，フレームスロットは同期をとるための領域として用いられる．この方式には，リアルワールドコンピューティング研究計画で開発中の RWC-1[25)]，MIT の P-RISC[13)]や *T[14)]，カリフォルニア大学バークレー校で研究されている抽象マシン TAM[4)]などがある．

3. スレッド内の命令としてフレームスロットへのアクセスがあったときに，そのスロットが空であったときには，そのスレッドの実行が中断され，別の実行可能なスレッドに実行を切り替えるもの．中断していたスレッドはフレームスロットへの書込みがあって，そのスロットが空でなくなると，再開可能となる．この方式は，Dataflow/von Neuman Hybrid[11)]と CODA[21)]に見られる．

スレッドを実行をどのように行っていくかという観点からみたときには，前2者は静的に与えられた単位で積極的にスレッドの実行を切り替えるものであるが，第3番目の方式は，あるスレッドの実行をブロックする要因が発生するまで，そのスレッドを続けて実行するものである．より具体的には，レジスタやメモリキャッシュなどへの読出し時に，その内容がすでに書き込まれているかどうかを判断して，値が書き込まれていなければ，そのスレッドの実行を続けることができないため，実行を別の実行可能なスレッドにハードウェア的に切り替える実行機構を有する．

第3の方式におけるデータ駆動的な同期の機構は，アーキテクチャ的観点からは後述する非同期的なメモリアクセス構造によってインプリメントされるので，以下では 1. と 2. についてのみ述べる．第1の例として EM-4 を，第2の例として P-RISC をとりあげる．

3.1 データ駆動待合せに基づくマルチスレッド実行

データ駆動モデルにおいては，すべての命令が他のすべての命令と対等の立場で並列的に実行されるという前提がある．すなわち，命令の実行順序はデータの依存関係によってのみ拘束され，プログラマがある意図をもって実行順序を制御しようとしてもできないという問題がある．この点を解決するためには，データ駆動的な待合せ機構をもちながら，局所的な最適実行を目指すアーキテクチャが必要である．そのために，計算の局所性を実行モデルとして表現する必要がある．本稿では，データ駆動モデルを拡張することによって，細粒度マルチスレッド処理をサポートする強連結枝モデルを紹介する．

強連結枝モデルは，データ駆動の基本的なモデ

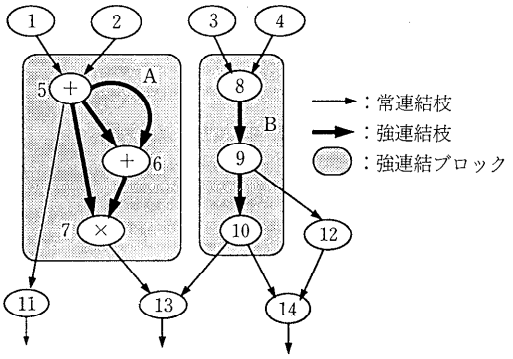


図-5　データ駆動モデルの拡張（強連結枝モデル）

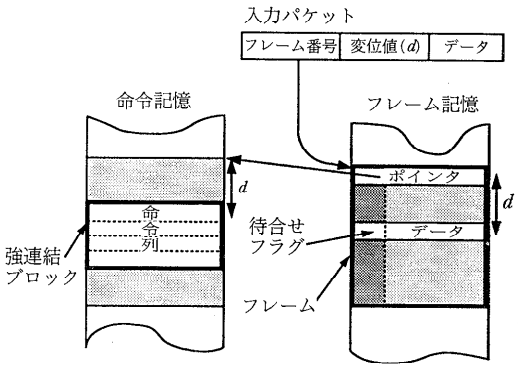


図-6　直接待合せ方式（EM-4）

ルにおけるノード間のアークの性質を拡張し，ノードの連結の強さに二つのモードを設けることによって計算の局所性を表現することが可能なモデルである[16),23)]．強連結枝モデルでは**図-5**に示すように，データ駆動図式においてノードの出力アーク(枝)に2種類の区別を設け，一方を常連結枝，他方を強連結枝と呼ぶ．常連結枝の働きは通常のデータ駆動モデルと同一である．強連結枝によって連結されたデータ駆動図式内のある領域（強連結ブロックと呼ぶ）が定義される．強連結ブロック内のあるノードが実行されると，同じ強連結ブロックの中の発火可能なノードだけが次に実行される．そのブロックへの外部からのデータがすべて到着し，かつブロック内に実行可能なノードがすべてなくなると，強連結モードは解除され通常のデータ駆動制御にもどる．

強連結枝モデルのようなデータ駆動を拡張したモデルに基づいたデータ駆動計算機では，実行される命令の範囲を局在させ，実行される命令の範囲を限定することができる．具体的には，強連結ブロック内で待ち合わせるトークンの格納場所として，レジスタなどの一時記憶を利用することによって，連続して実行される命令の範囲を局所化し，処理の高速化を図ることができる．また，強連結ブロック内におけるトークン（パケット）を同じ要素プロセッサ内に閉じ込めることによって，相互結合網の通信量を削減できる．

電子技術総合研究所の EM-4 は，上に述べた強連結枝モデルに基づき強連結ブロック内の実行をレジスタを用いた RISC 的な先行制御パイプラインによって制御し，データ駆動的な待合せ機構との融合型のアーキテクチャを構成している．

EM-4 におけるフレームは，後述する待合せ機構において説明されているように，関数や手続きの実行形態に対応してスレッドの環境を保存するための記憶領域にとられる．細粒度マルチスレッドという観点からは，強連結ブロックの実行形態は一つのスレッドの断片であると捉えられる．一つの強連結ブロック内にある命令群は一続きのスレッド断片の命令列の処理として逐次的に実行され，強連結ブロックの終了にともなって，スレッドの実行が中断され，他の実行可能なスレッド内の強連結ブロックに実行が切り替えられる．中断されたスレッドの実行は，同一のフレームを有する他の強連結ブロックの起動によって再開される．これによって，無駄なパイプラインの空きが生じず，高効率なマルチスレッド実行を支えている[16)]．

このようなデータ駆動的な待合せに基づく細粒度処理においては，データの待合せ機構とそれによって起動されるスレッドとの対応にアーキテクチャ的な工夫がなされている．例として，EM-4 における直接待合せ方式を**図-6**に示す．データの待合せは，手続きあるいは関数を単位として，それが起動されたときに動的に確保されるフレーム記憶を入力パケットの待合せ記憶領域として用いることにより行われており，スレッドの実行機構との相性が良い．EM-4 では関数が起動されたときに待合せ用のメモリの領域（フレーム）を割り当て，待合せはその領域の先頭番地と変位 (d) を指定して，直接フレーム内のメモリを参照することによって行っている．フレーム内の記憶領域の一部は待合せフラグとして使用され，これを専用のハードウェアで高速に解釈する機構が備わって

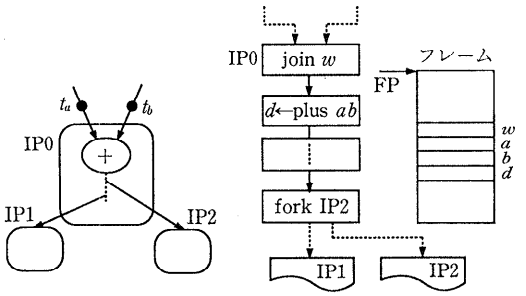


図-7 同期命令によるマルチスレッド処理

いる．また，プログラムはフレーム記憶とは別の命令記憶内に格納され，待合せ記憶領域と命令記憶領域内におけるパケットが置かれる待合せ番地と命令番地の間には対応関係がつけられている．

3.2 同期命令によるマルチスレッド実行

この方式は，マルチスレッドの処理を並列プロセッサで処理するために必要な，スレッドの生成と複数スレッドの同期をオペレーティングシステムのシステムコールなどのレベルではなく命令レベルで実行することによって，細粒度の並列処理をサポートするものである．図-7に，データ駆動図式とこれと同等の処理を行うマルチスレッド実行の例を示す．これは，P-RISC の例であるが，TAM や *T においても，基本的な方式は同一である．図-7の左に示したデータ駆動図式においては，t_a および t_b なる二つのデータの到着によって加算命令が起動されるが，右側のマルチスレッド実行においては同様の処理を行うために，データの受渡しと同期処理を切り離して実行する．

図-7 における join 命令は，データ駆動と同様の処理を行い，加算命令以下のスレッド実行を起動するための命令である．join 命令は，スレッド間の同期をとるために，二つのスレッドからのアクセスによって2回実行される．まず，あるスレッドがフレーム内の a 番地にデータを書き込んでおき，次にこの join 命令を実行する．(join w) を実行したときの処理は次のようになる．まず，フレーム内の w で表される番地の内容が調べられる．この値が0であるならば，この番地の値を1にセットするとともに，このスレッドの実行は中断される．次に，別のスレッドがフレーム内の b 番地にデータを書き込んでから，この join 命令を再び起動すると，w 番地の内容はすでに1であるので，このスレッドの実行が開始される．具体的には，次に実行される (d←plus ab) 命令によって，すでに値が書き込まれている a，b 番地からデータが読み出され，加算された結果が d 番地に書き込まれる．P-RISC では，join 命令は，二つのスレッドの同期のみをサポートするが，*T や TAM ではカウンタを設けることによって任意の数のスレッドの同期をサポートすることを可能としている．また，P-RISC のモデルではスレッド内の演算実行の命令と同期命令とが同一のプロセッサで処理されるが，*T のアーキテクチャでは演算処理と同期処理とを異なった処理装置で実行する方式が提案されており，同期命令の処理と演算処理が独立に実行される．

また，図-7 における fork 命令 (*T では start という名前の命令) は，データ駆動の例にあるような同時実行可能な複数の処理を起動するために，あらたに並列実行可能なスレッドを生成するための命令である．

4. データ構造に対するアクセス

4.1 細粒度並列処理とデータ構造アクセス

データ駆動やマルチスレッドなどの細粒度並列計算機においても，ノイマン型計算機と同様にデータ構造の処理方式は重要な問題である[1),3),32)]．データ駆動計算機や細粒度マルチスレッド計算機においても，データを構造化することによって計算やデータへのアクセスの構造化が容易となり，高水準言語による効率の良いプログラミングが可能となるが，その実現は簡単ではない．ここでは極端な例として，データ駆動計算機にデータ構造を導入する場合を想定してみる．データ駆動計算機においては，データを表現するための基本的な形式はノード間でやりとりされるトークンに付加されるデータ型である．純粋なデータ駆動計算モデルにおいては，データを記憶に格納し，それを逐次的に更新することによって処理を進めるという考え方は存在しない．このモデルにおいては，データがある命令によって実行されたとき，それらのデータはすべて消滅し，かわりに実行された結果のみが新たなデータとして生成されると考える．このことは，構造をもったデータ，すなわち，構造体に対しても成り立つ．しかしながら，このような考え方でデータ構造を

実現した場合には，非常に非効率なシステムにしかならないことは明白である．このため，データ構造処理を効率化するための手段がいくつか考えられた．それには，アレイのようなデータ構造の各要素をデータ駆動の通常のトークンとして処理するとともに，おのおののデータトークンのタグにインデックスを付加することによって，構造体の個々の要素に関して非同期的なアクセスを可能にする直接的な方式と，特別のメモリモジュール(通常はこれを構造体記憶と呼ぶ)を仮定して，その個々の要素に対して非同期的なアクセスを行う間接的な方式がある．前者の例としては，一次元的に並んだ値の列として定義されるストリームと呼ぶデータ構造をインデックスによって表現する方法やトークンリラベリング (Token Relabeling)[8] があげられる．ストリームは，その要素の値の生成と消費が逐次的に規定されてはいるが，すべての値が同時に存在しなくてもよいという性質をもっているため，データ駆動待合せとの相性が良い．また，トークンリラベリングは，より一般的なデータ構造に対して適応可能で，ループ処理などのときにデータ駆動待合せを行うトークンの待合せタグをある定まった関数で変換することによって，待合せ記憶上でデータの生成者とその消費者との間で非同期的なデータの受渡しを可能にするものである．後者の非同期的アクセスを有するデータ構造の代表的なものとしては，I-構造 (I-Structure)[2] があげられる．

以上は，データ駆動計算機に関するデータ構造操作の問題点であったが，細粒度マルチスレッド計算機においても，要素プロセッサ間でやりとりされるデータの表現方法に関して同様の問題が存在する．細粒度マルチスレッドにおけるデータ構造のアクセスでは，上に述べた間接的な方式が一般的である．以下では，間接的な方式の代表である I-構造を例にして，データ駆動計算機や細粒度のマルチスレッド計算機における非同期のメモリアクセスについて解説する．

4.2 非同期メモリアクセス

細粒度の並列処理を実現するためのデータ構造に対するアクセスについて考えてみる．ここでは，データ構造操作を専用に処理する特殊なメモリの存在を仮定することにする．このような仮定のもとで並列処理を行っている要素プロセッサからのメモリへの読み書きを効率良く処理するためには，プロセッサおよびメモリは次のような機能をもつ必要がある．

1. メモリに対する読出し要求と読み出されたデータに対する処理は切り離され，非同期的に実行可能である．

2. データ構造の各要素に対する読出し要求が書込み処理よりも時間的に先に到着することを許す．

前者は，スプリットフェーズ (sprit-phase) 読出しまたはアクセスと呼ばれており，メモリに対する読出し要求から読み出されたデータが戻ってくるまでの遅延 (latency) が，並列処理においては比較的大きくなると同時に予測できないことによる．このため，並列処理の各処理ユニットは，読出し要求を発すると，別の実行可能な処理を行い，読み出されたデータが戻ってきた時点で，そのデータを必要とする処理を実行することになる．また，データ駆動計算機や細粒度のマルチスレッド計算機においては，必ずしも書込み要求が読出し要求よりも先に処理されるというわけではない．したがって後者は，メモリユニットにおいて，データの書込みと読出しの間の同期をとる機能が備わっており，読出しが書込みより先にくる (deferred read) を許す必要があることを示している．

I-構造[2]は，MIT の Arvind らによって提案された，細粒度並列処理のためのデータ構造の表現モデルであり，上に述べた機能を備えたデータ構造モデルの一種である．このモデルにおいては，生成および消費されるデータ構造の性質が限定されている．このことによって，単純なアーキテクチャを用いることによりアレイなどのデータ構造を用いたプログラムを細粒度並列計算機上で効率的に実行できるようになる．

I-構造の基本構造は，データ構造の各要素に対して1回だけデータの書込みを許し，また書かれたデータに対して1回だけ読み出すことを許す点である．したがって，基本的な I-構造 は，データ構造の各メモリ要素ごとに1ビットのタグをもたせることによって実現することができる．このメモリのタグはデータが書き込まれると1にセットされ，データが読み出されると0にリセットされる．もし，データを書き込もうとしたときにこ

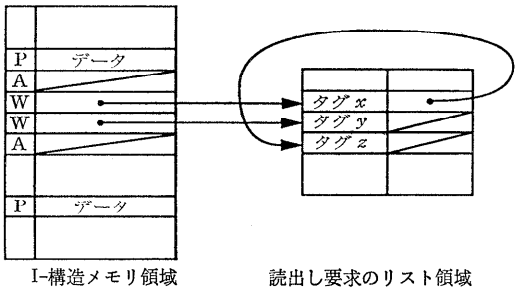


図-8 複数の読出し要求の処理

のタグがセットされていたならば，その書込み要求は待たされこのデータが読み込まれるまで実行されない．同様にこのタグが0のときに，データを読み込もうとしてもデータが書き込まれるまで待たされる．

この基本的なI-構造を拡張して，一つの書込みに対して複数の読出しを許すことが可能である[3),20)]．このような複数の読出しを実現するためには，書込みより先に到着した読出し要求をなんらかの形で待ち行列に構成する必要がある．これを実現するには，図-8に示すように，各メモリ番地に対して三つの状態が必要である．状態Aは，その番地に対して読出し要求も書込み要求もないことを示す．状態Wは，読出し要求があったことを示す．読出し要求が一つの場合には，読み出した値を送りつけるスレッドのアドレスがこの番地のデータ部に書かれる．読出し要求が複数あった場合には図-8に示すように，データ部は待合せ行列エリアへのポインタとなり，このエリア内において待ち合わせている読出し要求のリストが作られる．書込み要求があると，このリストをたぐりながら読出し要求に対して書き込まれたデータを送りつけた後に，データ部に値を書き込む．状態Pは書込み要求のみがあった場合を示している．この場合には，データ部には書込み要求のあったデータが置かれ，後に読出し要求があればこの値が返される．

具体的に，$x=A[i]+2$ および $A[i]=x*x$ をI-構造を用いてデータ駆動方式で実現した場合を図-9に示す．細粒度マルチスレッドとして実現する場合にも，プロセッサとメモリの間でのインタフェースとしての動作は同一になるはずである．図において，パケット p1 はメモリ読出しの要求パケットであり，パケット p2 はメモリへの書込みパケットである．I-read 命令を実行して，

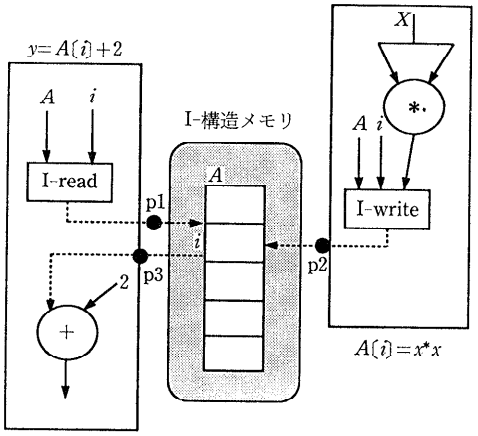


図-9 I-構造に対するアクセスの様子

p1 パケットを送出したプロセッサは，その読み出したデータの到着を待たずに別の実行可能な命令を実行する．I-構造メモリでは，p1 および p2 パケットのどちらが先に到着してもよい．仮に p1 パケットが先に到着して，その後に p2 パケットが到着した場合を仮定すると，p2 の到着によってそのデータは $A[i]$ に書き込まれ，それと同時に p3 パケットが送り出される．p3 パケットには，書き込まれたデータとそのデータによって起動される命令（細粒度マルチスレッドの場合には，起動されるスレッドの識別子）が指定されている．このようにして，I-構造のような非同期メモリを導入することにより，メモリアクセスの遅延に関わらずに効率的な細粒度の並列処理が可能であることが理解されよう．

5. 最適な細粒度実行のためのソフトウェア技術

高級言語で記述されたプログラムをコンパイルしてデータ駆動に基づく細粒度並列計算機上で実行する場合には，コンパイラの最適化技術およびハードウェアおよびソフトウェアによる負荷分散や負荷制御の技術などが必要になる．後者の実行時における負荷制御に関する技術は，並列処理一般におけるより広い研究対象であり，データ駆動に関する個別的な問題は少ないので，本稿では，前者のコンパイラ技術に絞って述べる．たとえば関数性をもったデータ駆動型の高級言語をデータ駆動計算機の機械語であるデータ駆動図式にコンパイルする方法は次のようになる．高級言語自体

は並列処理のセマンティックスをもっているため，まず記述されたプログラムの構造をグラフ的な表現に変換し，さらに再帰的にそのサブグラフを求め，最終的にデータ駆動図式になるまで変換を行えばよい．このような，変換過程において，効率的なデータ駆動図式を得るための手法およびその最適化については，文献28)に解説がある．

これに対して，細粒度マルチスレッド計算機に対しては，同様のデータ駆動型言語を用いるものと，マルチスレッド向きの高級言語を設定するものがある．前者の場合には，並列処理のセマンティックスをもった構造からスレッドを抽出しスレッド間での同期を記述することによって，全体として最適な並列実行を可能にするための最適化を行う必要がある．一般的には，データ駆動図式のような制御フローのないプログラム形態を最適化コンパイラに用いられるプログラム依存グラフ[30]に類似した中間的な形態に変換する．このグラフは，基本的にはデータ依存と制御依存の2種類の区別をもった枝からなるグラフとして表現される☆．このような中間形態から，スレッドを抽出する手順は，次のようになる．すなわち，まずグラフの節をまとめてスレッドの基本となる単位(基本分割)を抽出する．そのような基本分割は，実行時にしか決定できない依存性，関数呼出し，データ構造へのアクセスなどによって，規定される[19]．次に，基本分割を組み合わせて，より大きなスレッドに融合する．このようなスレッドの抽出に加えて，スレッド内の不必要な制御の流れを削減したり，同一の関数スレッド内でのスレッド間のデータ転送を削減するなどの手法を用いた最適化手法も研究が行われている[22]．

一方，後者の例としては電総研のEM-C[18]やカリフォルニア大学バークレー校のSplit-C[7]などがある．後者の場合には，基本的にスレッドの生成などの並列処理記述はユーザに任されるためコンパイラで行う最適化は，逐次的なスレッド内部での最適化処理が大部分となり，RISCなどの最適化と同様の手法が用いられる．

☆一部のコンパイラ（たとえば，TAM[4]の中間言語を生成するためのコンパイラ）では，I-構造などのデータ構造に対するアクセスを明確に区別するために，さらに別の種類の枝が付加されることもある．

6. おわりに

本稿では，細粒度並列処理という観点からデータ駆動計算機およびその発展形としての細粒度マルチスレッド方式について，そのモデル，実行制御，非同期メモリなどに関して解説した．デバイス技術を中心としたハードウェアの進歩により，計算機の処理能力は飛躍的な進展を遂げてきているが，21世紀に向けてさらに増大するであろうと予想される計算の需要を満たすためには，並列処理の導入が欠かせないことが認識されてきている．特に大規模な科学技術計算や高速のシミュレーションなどに関する計算需要は増大しており，このような計算を実行するために，超並列計算機に対する期待が大きくなってきている．このような超並列計算機を構成するためには，非常に単純化した汎用超並列システムのための要素プロセッサを構築することや，並列処理の抽出を自動的に行うことなどが必要になる．データ駆動計算機の研究で開発されたアーキテクチャ技術や高級言語のコンパイラの技術は，このような要求に合致していると考えられる．特に，従来のデータ駆動計算機の欠点を克服しつつある細粒度マルチスレッド計算機は基本的にはデータ駆動モデルに示された柔軟で汎用性のある並列計算モデルを踏襲しつつ，効率的な実行性能を達成することを目指している．超並列システムにおける基本要素プロセッサの形態としては，なんらかの形態の同期機構と演算の実行機構が統一的に統合されたものになると予想される[12),17]．しかし，このような効率的な超並列計算機がハードウェア的に実現されたとしても，それを有効に利用するためのソフトウェアも同時に解決しなければならない．並列処理のためのプログラミング言語やコンパイラそしてプログラミング環境なども含んだオペレーティングシステムなどに関する広範囲な研究がさらに必要であると考えられる．

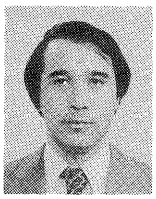
謝辞　本稿をまとめる上で，電気通信大学教授の弓場敏嗣氏，名古屋大学教授の島田俊夫氏，新情報処理開発機構つくば研究センタの坂井修一氏，ならびに計算機方式研究室の同僚諸氏との議論や知見が参考になった．ここに記して感謝する．

参考文献

1) Amamiya, M., Takesue, M., Hasegawa, R. and Mikami, H.: Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine, Proc. 13th Annu. Int. Symp. Computer Architecture, pp. 10-19 (1986).
2) Arvind and Thomas, R.E.: I-structures: An Efficient Data Type for Functional Languages, Laboratory for Computer Science, MIT, TM, 178 (1980).
3) Arvind and Nikhil, R.S.: Executing a Program on the MIT Tagged-Token Dataflow Architecture, IEEE Trans. Comput., C-39, 3, pp. 300-318 (1990).
4) Culler, D.E. et al.: Fine Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine, Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Systems (ASPLOS), pp. 164-175 (1991).
5) Dennis, J.B.: First Version of a Data Flow Procedure Language, Lecture Notes in Computer Science, Springer-Verlag, 19, pp. 362-376 (1974).
6) Dennis, J.B. and Misnus, R.P.: A Preliminary Architecture for a Basic Data Flow Processor, Proc. 2nd Annu. Int. Symp. Computer Architecture, pp. 126-132 (1974).
7) Eicken, T. von, Culler, D.E., Goldstein, S.C. and Schauser, K.E.: Active Messages: A Mechanism for Integrated Communication and Computation, Proc. 19th Annu. Int. Symp. Computer Architecture, pp. 256-266 (1992).
8) Gaudiot, J.L. and Wei, Y.-H.: Token Relabeling in a Tagged Token Data-Flow Architecture, IEEE Trans. Comput., C-38, 9, pp. 1225-1239 (1989).
9) Gurd, J., Kirkham, C.C. and Watson, I.: The Manchester Prototype Dataflow Computer, Commun. ACM, 21, 1, pp. 34-52 (1985).
10) Hiraki, K., Shimada, T. and Nishida, K.: A Hardware Design of the SIGMA-1—A Data Flow Computer for Scientific Computations, Proc. Int. Conf. Parallel Processing, pp. 524-531 (1984).
11) Iannucci, R. A.: A Dataflow / von Neumann Hybrid Architecture, Laboratory for Computer Science, MIT, TR, 418 (1988).
12) Kodama, Y., Koumura, Y., Sato, M., Sakane, H., Sakai, S. and Yamaguchi, Y.: EMC-Y: Parallel Processing Element Optimizing Communication and Computation, Proc. Int. Conf. Supercomputing, pp. 167-174 (1993).
13) Nikhil, R.S. and Arvind: Can Dataflow Subsume von Neumann Computing?, Proc. 16th Annu. Int. Symp. Computer Architecture, pp. 262-272 (1989).
14) Nikhil, R.S., Papadopoulos, G.M. and Arvind: *T: A Multithreaded Massively Parallel Architecture, Laboratory for Computer Science, MIT, Tech. Rep. CSG Memo, 325-1 (1991).
15) Papadopoulos, G.M. and Culler, D.E.: Monsoon: An Explicit Token-Store Architecture, Proc. 17th Annu. Int. Symp. Computer Architecture, pp. 82-91 (1990).
16) Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Dataflow Single Chip Processor, Proc. 16th Annu. Int. Symp. Computer Architecture, pp. 46-53 (1989).
17) Sakai, S., Okamato, K., Matsuoka, H., Hirono, H., Kodama, Y. and Sato, M.: Super-Threading: Architectural and Software Mechanism for Optimizing Parallel Computation, Proc. Int. Conf. Supercomputing, pp. 251-260 (1993).
18) Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y. and Koumura, Y.: Thread-Based Programming for the EM-4 Hybrid Dataflow Machine, Proc. 19th Annu. Int. Symp. Computer Architecture, pp. 146-155 (1992).
19) Schauser, K.E., Culler, D.E. and Eicken, T. von: Compiler-Controlled Multithreading for Lenient Parallel Languages, Proc. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, Springer-Verlag, 523, pp. 73-101 (1991).
20) Shimada, T., Hiraki, K. and Sekiguchi, S.: Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computations, Proc. 13th Annu. Int. Symp. Computer Architecture, pp. 226-234 (1986).
21) Toda, K., Nishida, K., Uchibori, Y., Sakai, S. and Shimada, T.: Parallel Multi-Context Architecture with High-Speed Synchronizing Mechanism, Proc. 5th Int. Parallel Processing Symposium, pp. 336-343 (1991).
22) Traub, K.R.: Multi-Thread Code Generation for Dataflow Architectures from Non-Strict Programs, Proc. Functional Programming Languages and Computer Architecture, Springer Verlag, pp. 17-34 (1985).
23) Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y. and Yuba, T.: An Architectural Design of a Highly Parallel Dataflow Machine, Proc. IFIP Congress 89, pp. 1155-1160 (1989).
24) 児玉, 坂井, 山口：高並列計算機 EM-4 とその並列性能評価, 信学論 (D), J 75-D-I, 8, pp. 607-614 (1992).
25) 坂井, 石川：RWC における超並列システムの研究開発, 情報処理, Vol. 34, No. 12, pp. 1440-1444 (Dec. 1993).
26) 島田俊夫：数値計算向きデータフローマシン, 情報処理, Vol. 26, No. 7, pp. 780-786 (July 1985).
27) 島田俊夫：データフローマシン, 情報処理, Vol. 28, No. 1, pp. 85-93 (Jan. 1987).

28) 関口，山口：データ駆動計算機用の高級言語と処理系，情報処理，Vol. 31, No. 6, pp. 753-762 (June 1990).
29) 曽和將容：データフローマシンと言語，昭晃堂 (1986).
30) 本多弘樹：自動並列化コンパイラ，情報処理，Vol. 34, No. 9, pp. 1150-1157 (Sep. 1993).
31) 山口喜教：データフロー計算機，シミュレーション，Vol. 11, No. 1, pp. 29-37 (1992).
32) 弓場，山口：並列計算機の命令セットアーキテクチャ，情報処理，Vol. 29, No. 12, pp. 1446-1453 (Dec. 1988).
33) 弓場，山口：データ駆動型並列計算機，オーム社 (1993).

(平成5年10月21日受付)



山口 喜教（正会員）

1972年東京大学工学部電子工学科卒業．同年通商産業省工業技術院電子技術総合研究所入所．以来，高級言語計算機，データフロー計算機などの研究に従事．現在，情報アーキテクチャ部計算機方式研究室長．工学博士．1991年情報処理学会論文賞受賞．著書「データ駆動型並列計算機」（共著）等．電子情報通信学会会員．