計算機アーキテクチャ　63－19
（1986・11・21）

データ駆動計算機による マクロ・データフロー処理

Macro Dataflow Processing by A Dataflow Processor

平木　敬　　戸田賢二
Kei Hiraki　　Kenji Toda
電子技術総合研究所
Electrotechnical Laboratory

1．はじめに

命令レベルデータ駆動計算機の性能を広い範囲の応用分野で活用するためには、計算機自身の構成や性能のみならず、適用する問題を記述する言語が非常に大きな重要性を持つ。データ駆動計算機で実行することを目的とした言語として、これまで所謂データフロー言語が数多く提案され、試験的に使用されてきた【1、2、3、4】。これらのデータフロー言語を、すべての実行が四則演算や分岐操作に代表される基本操作レベルで、データフローグラフに対応する意味付けに基づいて行われることから、ミクロ・データフロー言語と呼ぶ。これまでに提案された多くのミクロ・データフロー言語は、データフローグラフに対応する意味付けにたいして簡易に問題を写像するために①　関数型言語的であること、②　単一代入を基本的規則とすること、③　副作用を表面的に消去するために繰り返しを基本構造に取り込むこと等を特徴とした。

歴史的に概観すると、命令レベルデータ駆動アーキテクチャはミクロ・データフロー言語に直接対応する解釈機構のアーキテクチャとして発展し、具体化されてきた【5、6】。しかしながら、このような歴史的経緯を離れると、命令レベルデータ駆動アーキテクチャはミクロ・データフロー言語とは独立した存在と考えることが自然である。すなわち、命令レベルデータ駆動アーキテクチャを、命令実行がデータの到着を契機として開始されるという命令実行順序制御に基づくアーキテクチャとして捉えることである。このような見地に立つと、命令レベル・データ駆動計算機を目的計算機とした言語をより広い範囲から求めることが可能となる。

直列的な意味構造に基づく言語記述は各基本操作が各々変数またはメモリに対する代入という形で状態の変更を積み重ねてプログラムを進行させる点において、先に述べたミクロ・データフロー言語に基づく記述と対極をなす。プログラムの内包する並列性の叙述においても、直列言語では明示されず、並列性解析（同時実行可能性解析）により初めて結果を不変に保つ並列実行部分を抽出できることに対し、ミクロ・データフロー言語では記述により並列性を全て明示するため、並列性解析無しで並列実行が可能である。しかしながら、その反作用として、順序性の検出が必要である条件分岐構造、ループ構造、関数の生起終了の処理を行うために、順序解析が必要となる。順序解析は名前こそ異なるが、内容的には並列性解析とほぼ同一の内容をもつ。従って、言語から並列処理形態への変換という視点からはミクロ・データフロー言語の優位性は殆ど主張できない。むしろ並列性を全て直接明示して記述するために引き起こされる記述性の低下、局所的な並列性と大局的な並列性が無差別に明示されることから引き起こされる並列性制御の困難などがミクロ・データフロー言語の問題点として認識される。第2節ではミクロ・データフロー言語の特性に関する諸問題点を述べる。

マクロ・データフロー言語は上記議論の妥協点として位置付けられるものである。従来、マクロ・データフロー処理方式は並列フォンノイマン計算機またはその変形上で並列処理を関数またはコードブロック単位で行う方式として提案されてきた【7、8、9、10】。すなわち、基本的には直列的な意味付けが行われているプログラムに対して、コンパイラ等の手段で解析を行い、並列環境で実行する実行方式である。しかし

ながら、本報告では処理すべきことがらを記述する方式として、マクロ・データフロー言語を捉える。すなわち、大局的並列性を明示し、局所的並列性は明示しないことである。マクロ・データフロー言語に関しては第3節で詳述する。

マクロ・データフロー言語で記述されたプログラムをどのようなアーキテクチャで解釈・実行するかが次の問題である。第4節では命令レベル・データ駆動計算機によるマクロ・データフロー言語の実行方式を述べる。最後に付録で例題を通じてミクロ・データフロー言語とマクロ・データフロー言語の記述および実行について、比較を行う。

ここで問題となる項目は、命令レベルデータ駆動アーキテクチャと人間の思考形態とをどのように橋渡しするかという漠然かつ包括的なものである。本報告では、その端緒として直列的な意味構造を持つ言語と命令レベルデータ駆動計算機の関係、特にミクロ・データフロー言語で命令レベル・データ駆動計算機を使用する際の諸問題点を考察し、それを踏まえてマクロ・データフロー言語と命令レベル・データ駆動計算機との適合性について論じる。

2．ミクロ・データフロー言語

ミクロ・データフロー言語はデータフローグラフに対応する意味付けを行う言語である。説明を簡単にするため、現在当所で開発を進めているミクロ・データフロー言語ＤＦＣを代表例とする[1]。プログラムが完全に平面的である、すなわち条件分岐、ループ、再帰的関数呼び出しを含まない場合にはプログラムは変数に相当する識別子をアークとするデータフローグラフに対応し、解釈は完全に並列的である。演算は対応するデータフローグラフの構造に従ってのみ進行し、プログラム内の順序は無視される。記述が更に単一代入則に則っている場合には、プログラムに記述した順序が任意であることから陰に並列性を記述したとも、また各演算はデータ依存関係から全て並列に実行される可能性を持つため並列性を明示したとも解釈される。

しかしながら、このような単純な構文だけで記述可能なプログラムは大幅に限定される。プログラムで表現したい内容自身に状態概念をふくむ場合や対象物が繰りかえし構造を持つ場合には、単一代入則を満たさないデータフローグラフの記述能力、条件分岐構文、ループ等繰り返し構造、または再帰的関数呼び出しのうち一部または全部が必要である。何れの場合も、陽に呈示されるか構文から指定されるかして、アークの合流（マージ）が出現することが特徴的である。その結果もはやデータ依存関係だけに準拠した意味付けが困難となり、何等かの順序関係規定が必要となる。

ミクロ・データフロー言語では表記上単一代入則を遵守するため、アークの合流は構文から自動生成されるものに限定される。何れにせよ、順序関係が部分的に規定される記述形式においては、もはやデータ依存関係だけで全体の動作を理解することが困難であり、陰な並列性記述とはいえない。寧ろ構文上暗に順序制御されている場合を除き全ての並列性を明示した記述形式と見ることが妥当であろう。

ミクロ・データフロー言語の命令レベル・データ駆動計算機に対する適合性を下記の2項目から考察する。

（1）ミクロ・データフロー言語は命令レベルデータ駆動計算機にとって処理効率の良いものであるか。

（2）ミクロ・データフロー言語は問題を記述することに適した言語であるか。

第一の項目は、コンパイラ作成に際する問題点および計算機上での実行効率に関するものである。先に述べたように、ミクロ・データフロー言語は並列性を完全に明示することを目標とした言語である。しかしながら、記述性を上げる目的で条件分岐構文、ループ等繰りかえし構造、再帰的関数呼び出し等順序関係を基本とした要素を取り込まざるを得ない。この結果並列性を基本とした記述の中から順序関係を抽出する順序解析が不可欠なものとなる。

ＤＦＣコンパイラにおける処理を例とると、順序解析は① 分枝したグラフを完全に合流するため、② 繰り返し構造等順序化された構文で、外部からのアーク、内部で閉じているアーク、繰り返し構造中を伝播するアークを区別し、各々に必要な処理を行うため、③ 関数の終了、並列処理部分終了の判定等で内容的には時間的順序関係を与えたい場合に行われる。これらの解析は、各データフローグラフ上のノードに対してコントロール情報の伝播を全ての順序について調べ、並列に記述された内容と一致する動作を常に保証するようにコードを生成する。

上記の順序解析を必要とする構文は、残念ながら特殊なものでなく、通常のプログラム中に普遍的にあら

われる。従ってコンパイラはプログラムの殆どの部分について順序解析すなわちコントロールフロー解析を行う必要がある。このコントロールフロー解析は順序制御関係が複雑に錯綜する場合、例えば多重の条件分岐とループが組合わさる場合には複雑かつ大規模なものとなる。

このような、プログラムを並列に実行するためのフロー解析は、直列的言語から並列に実行できる要素を抽出するために行われるデータフロー解析【11、12】と内容的には殆ど一致している。従って、並列に実行可能なコードを生成するという観点からは、記述自身に並列性が明示してある特徴は殆ど生きてこない。

ミクロ・データフロー言語の持つ、全ての並列性が原則的に明示されている性質は、並列性の制御という観点からも問題を含んでいる。有限個の処理装置上で実行することを前提とする並列処理においてはプログラムから抽出される並列度には適正な個数があり、多ければ多い程良いわけではない。極端な場合として、並列性が無制限に膨張すると資源が枯渇し、実行が続行できなくなる場合すら存在する。従って何等かの並列性制御が必要であることは明白である。ミクロ・データフロー言語による記述は全ての並列性を明示することを原則としているため、局所的な並列性、例えばループ内の並列動作と大局的な並列性、例えば関数間並列性が区別なく、並列性が明示されない場合と事実上差がない、言い換えると、並列性が明示されることの優位性が主張できない。

一方、第二の観点すなわち問題の記述性からもいくつかの問題点が指摘される。問題をプログラムとして表記する立場に立つと、ミクロ・データフロー言語は

（1）単一代入則に従う。すなわち、識別子はプログラム全体を通じて一回しか値を定めてはいけない。

（2）副作用がない。上の項目で示されるように、同一識別子に複数回値を与えることにより状態を作らない。

（3）実行順序が表記した順序ではなくデータ駆動原理により定まる。

（4）ループ構造で世代間を跨がり使用する値を表記するために、ｏｌｄまたはｎｅｗといった特殊構文を使用する。

これらの項目はプログラムを書く場面でプログラマに種々の負担を課する。

まず（1）の制限の結果、明らかに局所的に使用される中間変数に対しても新たな識別子を作成せざるを得なくなるため、プログラム中で使用する総識別子数が増加する。また順序は関係なく多数回値を入れたい場合を表記することが大袈裟な表記になってしまう。また（2）の制限の結果、状態を直接的に作る方法が閉ざされ、ループ世代間を渡る値に関するｎｅｗ等の構文で間接的に作らざるを得ない。特にシミュレーション等でステップを進めながら繰りかえし演算を、しかも不均一に行う問題を記述する際に間接的で煩雑な表記となる。

（3）と（4）は深く関係している項目である。実行順序がデータ依存関係で決定する結果、ループの世代間で受け渡す値はｎｅｗ等明示しない限り認識できない。このことから、プログラマはループ内の識別子に入る値の時間的関係を常に意識して書かないと、プログラマの意図の通りに動作するプログラムを作成できない【13】。これは表記の順序関係というプログラマが最も認識し易い情報を捨て去るために発生すると考えられる。

これらミクロ・データフロー言語特有の問題点は、それを解釈実行する命令レベル・データ駆動アーキテクチャにあるわけではなく、むしろミクロ・データフロー言語自身に存する。なぜなら第3節で述べるように命令レベル・データ駆動アーキテクチャ上に上記（1）～（4）の制限を撤廃した言語を実装することが可能だからである。

3．マクロ・データフロー言語

過去多くのマクロ・データフロー処理に関する提案がなされてきた。その内容は、あるものは純直列的意味付けが行われているプログラムを、コンパイラが並列性解析により並列フォンノイマン計算機上で実行する方式であり、またあるものは関数またはコードブロック単位で並列性をプログラム上に明示し、並列フォンノイマン計算機上で大粒度データフロー演算をシミュレートして実行する方式である。何れの場合もマクロ・データフロー処理は並列フォンノイマン計算機上で疑似的にデータフローを実現する実行方式として扱

注1）現在のＤＦＣには単一代入則を破る記述を許す仕様が含まれているため、厳密な意味でのミクロ・データフロー言語とは異なっている。

われてきた。しかしながら、ここでは処理を記述する方式としてのマクロ・データフロー言語を提案する。

マクロ・データフロー言語の目標は、並列性を含む問題を最も自然に記述する方式を求めることである。本報告で提案するマクロ・データフロー言語は状態を持った系を記述することを目標に含む。従って、単一代入および副作用無しという概念は制限を受ける。しかしながらこれらを野放しにすると並列性の抽出、特に大局的並列性の抽出において困難をきたす。このため、単一代入則や副作用を禁止することに替わる新たな規範が必要である。ここでは新たな規範として、多重代入と副作用を直列的な意味付けを与える部分に封じ込めるアプローチを取る。直列的な意味付けかつ副作用を含む要素を直列部として切り出し、他の部分では原則的に並列な意味付けで解釈するものとする。

以上のような基本構成から、プログラムは図1に示すように直列部分をコードブロックとする要素をデータフローグラフで結合した形態を取る。副作用はある直列コードブロック内部に封じ込められる。直列コードブロック内部で呼び出す関数は再び並列に意味付けられる。その結果直列コードブロックと並列コードブロックが任意に組合わさるプログラム構成が可能である。

第2節でも述べたように、制御構造が平板である限り記述に対する意味付けは表記法以上の深い意味を持たない。問題となるのは表記の順序関係が意味を持つ、条件分岐構文、繰り返し構文、関数呼び出し等の場合である。

条件分岐構文（所謂条件代入文を除く）は与えられる条件によって真または偽のコードブロックを選択して実行するため、どちらかのコードブロックは表記されていても実行されない。これは明白な順序づけであり、直列コードブロック内だけで使用する。ただし真または偽の節は直列または並列の何れでも差し支えない。

繰り返し構造は表記の節約のためのものと、イテレーションが基本となる2種を区別する必要がある。ここでは前者を forall 型、後者を iter 型と呼ぶ。forall型の繰り返し構造では展開した形でも完全並列に実行可能なため、並列コードブロック、直列コードブロックの何れの中でも使用可能である。また forall で記述される繰り返し内部でも並列コードブロック、直列コードブロックの何れの使用も可能である。一方、iter 型ループは繰り返し間の干渉が基本構造をなすため、本質的に直列的である。また、 iter 節は繰り返し後の結果を得た状態は明確に時間的順序づけが繰り返し前の状態との間になされるため、意味構造が直列である。従って直列コードブロック内部でのみ使用可能である。

大域変数（複数のブロック間で参照関係のある値）の扱いは並列動作環境下では微妙な問題を内包する。もしプログラム中の全変数に単一代入則が厳密に適用されていれば、大域変数にはなんら問題が存在しない。しかしながら、状態概念を含んだ問題の記述に iter 型繰り返し構造を必要とし、記述力が強く制限された。また純直列的な意味付けをする記述においては、いかなる放漫な大域変数に対する扱いをしても、プログラムの決定性は損なわれなかった。しかしながら、放漫な参照関係はプログラムからの並列性の抽出に対して大きな障害となる。並列性を明示する言語において、大域変数に対する多重代入と副作用の記述は並列環境下で値の決定性を失なってはならないという制約を受ける。このことを最小限の制限で実現するため、

（1）並列コードブロックでは単一代入則を適用する。

（2）直列コードブロックでは多重書き込みを許す。

（3）直列コードブロック内での同一変数への多重書き込みはそれ自身または全て直列コードブロックで構成される内部関数に限定する。

このような制限は、全体に渡る単一代入則の適用より軽微な制限であるが、動作の決定性を損なわないことは明らかであろう。

以上、非形式的にマクロ・データフロー言語の満たすべき要件を検討した。ここでは並列と直列はコードブロックという概念を使用したが、勿論並列ということは並列に実行されるという意味付けのことであり、直列ということは、直列に逐次解釈した場合に正しい動作を与える意味付けのであり、実際の計算機上での実行形態を示している訳ではないことに注意したい。

4．データ駆動計算機によるマクロデータフロー処理

本節では、命令レベルデータ駆動計算機によるマクロ・データフロー言語の実行方式について述べる。マクロ・データフロー言語のミクロ・データフロー的部分の解釈、実行は既に広く実現されているためここでは述べない。だたし、従来ミクロ・データフロー言語から命令レベルデータ計算機に翻訳する際に問題となる順序解析を含む部分は全て直列コードブロック部分に移したため、並列コードブロックの翻訳には順序解析は必要なく単なる構文の解析とコード生成になることを付記する。

命令レベル・データ駆動計算機の命令コードは原則的にはデータの到着が命令動作を開始させ、命令動作終了後は入力データが消費される特性を持つため、直列コードブロックの翻訳には、副作用を軸とする記述から、演算相互間の構造を軸とする実行形態への変換が必要となる。しかしながら、多重代入が行われない限り、ミクロ・データフロー言語のコンパイラにおける処理と同一である。以下多重代入を含む代表的な場合について、処理形態を示す。

4－1．単純な多重代入

単純な多重代入とは、同一識別子に複数回値を入れるが、個々の値を入れる文は1回ずつしか実行されない場合である。従って値の参照が多重代入中のどの値かは容易に解析可能である。逐次実行部分では、識別子でトポロジカル・ソートを行い対応するデータフローグラフを得る。多重に値が定められる識別子は最後の物だけ残し、他は適当に変名する。

4－2．　局所変数に対する多重代入

局所変数とは、変数に対する多重代入が全て同一または直列に接続するコードブロックで行われる識別子に対する値のことである。多重代入が繰り返し構造以外で行われる場合は4－1節に示した通りである。

forall 型繰り返し構造における多重代入は、繰り返し間の干渉がないため繰り返し毎に独立な多重代入と等しい。従って繰り返し毎に別個の識別子をあたえることにより実現する。iter 型繰り返し構造では、実行機構としてアンフォールディング方式【6】を用いる場合にはミクロ・データフロー言語における処理と同様、最後に出現する識別子への値の割り付けに対し繰り返しの次回へ渡す値の操作を行う（new の操作）。繰り返しを順次解く方式を用いる場合には、アンフォールディングを用いる場合にループ識別子を変える操作に換えて、図3に示すような同期操作を行う。

4－3．大局変数に対する多重代入

大局変数とは複数のコードブロックに跨がって多重代入が行われる変数である。このような多重代入形態は一般にプログラム中の並列性を低下させるため、好ましいとはいえないが、アルゴリズム上やむをえない場合が多々ある。勿論各直列部分毎に終了の同期をとることにより実現も可能であるが、引き出せる並列性も大幅に制限をうける。命令レベル・データ駆動アーキテクチャでは各コードブロックは並列に実行する多重処理として実現し、通信コストが無視できる特徴を利用すると、図2に示すように大局変数を局所化しより並列性を引き出し易い形で実行することが可能である。この方式は同期および通信のコストが非常に低い命令レベル・データ駆動アーキテクチャの特徴を生かした方式といえる。

4－4．並列フォンノイマン計算機との比較

マクロ・データフロー処理は元来並列フォンノイマン計算機上で実行することを想定した処理であった。しかしながら、実行効率を無視すれば並列フォンノイマン計算機、命令レベルデータ駆動計算機ともほぼ同等な計算能力を持つため、比較は実行効率や速度を以って行う必要がある。

並列フォンノイマン計算機を構成する要素プロセッサは逐次処理を主目的とするアーキテクチャであるため、要素プロセッサ内で実行するプログラム部分が並列性を持たなくても性能は殆ど低下しない。しかしながら、実行するコードブロックの切り換え、実行時に発生する同期処理には演算と比較してかなり多くの時間を消費する。特に他の要素プロセッサとの通信とそれに伴う同期操作には多大（1桁以上）の時間を消費する。一方、命令レベルデータ駆動計算機は純直列プログラムでの性能の低下、および実行する命令数が増加する所謂データフロー・オーバーヘッド【14】が問題となる。しかしながら命令レベルで同期をとりながら実行するという性質上、局所的な並列性をも利用可能なため、もしアーキテクチャが適切であれば、前者は問題とならない程度まで低下可能である。

さて、実際の性能比較は個々のインプリメンテーションに依存する部分が多いため、現段階では判断が困難であるため、ここではミクロ・データフロー言語とマクロ・データフロー言語の何れがより命令レベル・データ駆動計算機に向いているかという観点から考察しよう。

マクロ・データフロー言語とミクロ・データフロー言語の相違点は① 直列的解釈か、② 多重代入を許すか、③ 副作用を許すかという点に集約される。ミクロ・データフロー言語におけるように単一代入かつ副作用を許さない制限下では直列的解釈とデータ依存関係に基づく解釈の間には、トポロジカルにソートしてある以上の相違は無く、命令レベルデータ駆動アーキテクチャの長所は関数またはプロセス間通信に限定された。一方、多重代入かつ副作用を許す場合には、代入と参照の依存関係は実行時にしか定まらないものを含むため、代入ごとに参照と同期をとる必要がある。このような内部変数レベルの同期こそ、命令レベル・データ駆動計算機の能力を活かせる。言い換えると、代入と参照の前後関係が実行時にならないと定まらない記述が行われている場面が命令レベル・データ駆動計算機を活用できる場所である。

従来、ミクロ・データフロー言語は命令レベル・データ駆動計算機に適した言語であると広く信じられていたが、上記の考察からは逆の対応、すなわちマクロ・データフロー言語等副作用を利用する言語こそ命令レベル・データ駆動計算機に適していると判断される。

5．おわりに

非常に低コストな同期操作、通信を実現する命令レベル・データ駆動計算機は実用的な並列処理形態を求める際に一つの理想の極と考えられる。しかしながら、その潜在能力を発揮する条件として、人間との橋渡しをする言語からのアプローチが不可欠である。本報告ではマクロ・データフロー言語とその実現という観点から命令レベル・データ駆動計算機の可能性を探った。その結果、定性的ではあるがミクロ・データフロー言語と比較するとマクロ・データフロー言語に命令レベル・データ駆動計算機との親和性がある、という結果を得た。また、ミクロ・データフロー言語の持つ諸問題点、特に問題の記述性について知見を得た。しかしながら、命令レベルデータ駆動計算機による実現という観点を離れると、ミクロ・データフロー言語には構造的であるという重要な性質があり、幅広い研究が期待される。今後具体的な実装を含め更に人間との親和性の良い記述方式について検討を行う予定である。

なお本研究の遂行にあたり御討論いただいた弓場計算機方式研究室長ならびに同僚諸氏に感謝いたします。

参考文献

【1】Arvind,Gostelow,K.P. and Plouffe,W.:An Asynchronous Programming Language and Computing Machine,Tech.Rep.TR114a,Dept.Information and Computer Science,Univ.California,Irvine(1978).

【2】Ackerman,W.B. and Dennis,J.B.:VAL-A Value-Oriented Algorithmic Language:Preliminary Reference Manual,Tech.Rep. TR-218,LCS,MIT(1979).

【3】McGraw,J.R.:The VAL Language:Description and Analysis, ACM Trans.Prog.Lang.Syst.,Vol.4, No.1,pp.44-82(1984).

【4】島田、平木、関口: 科学技術計算用データ駆動計算機ＳＩＧＭＡ－１の高級言語ＤＦＣ，信学会データフローワークショップ１９８６予稿集、ｐｐ．１２７－１３４（１９８６）．

【5】Dennis,J.B. and Misnas,D.P.:A Preliminary Architecture for A Basic Dataflow Processor, Proc.2nd Ann.Int.Symp.Computer Architecture, IEEE,pp.126-132(1975).

【6】Arvind,Kathail,V. and Pingali,K.:A Dataflow Architecture with Tagged Tokens,Tech.Rep. TR-174,LCS,MIT(1980).

【7】Kuck,D.J. et al.:Dependence Graphs and Compiler Optimization,Proc.8th ACM Symp. Principles of Programming Languages,pp.207-218 (1981).

【8】Gajski,D. et al.:Ceder-A Large Scale Multiprocessor,Proc.Int.Conf.Parallel Process.pp.524-529(1983).

【9】Babb,R.,Storc,L. and Ragsdale,W.:A Large Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS,Proc.Int.Conf.Parallel Process., pp.845-848(1986).

【10】戸田、内堀、弓場: 関数レベルデータ駆動計算機の検討、第３３会情処全大、５Ｃ－７（１９８６）.

【11】Yasumura,M.,Tanaka,Y. and Kanada,Y.: Compiling Algorithms and Techniques for the S-810 Vector Processor,Proc.Int.Conf.Parallel Process.pp.285-290(1984).

【１２】Huson,C. et al.:The KAP/205:An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer,Proc.Int.Conf.Parallel Process., pp.827-832(1986).
【１３】Gokhale,M.:Macro vs. Micro Dataflow:A Programming Example,Proc.Int.Conf.Parallel Process.,pp.849-852(1986).
【１４】平木、関口、島田：科学技術計算用データ駆動計算機ＳＩＧＭＡ－１命令の最適設計、第３２回情処全大、６Ｒ－３（１９８６）.

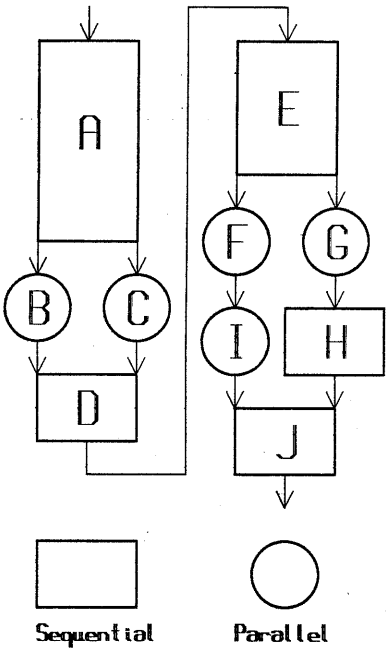

Fig.1 Sequential and Parallel Code Blocks

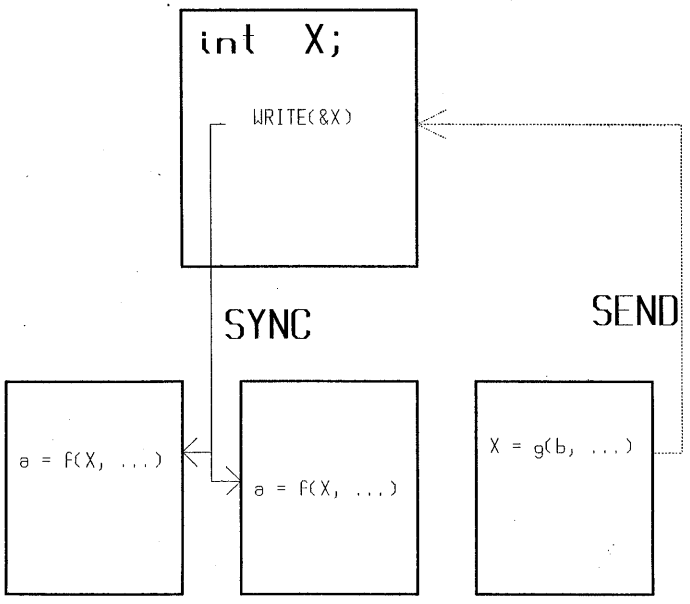

Fig.2 Writing Global Variable

付録．　例題によるマクロ言語とミクロ言語の比較

例題は、Ｎ－Ｑｕｅｅｎ（Ｎ＊Ｎの盤面にＮ個のチェスのＱｕｅｅｎを何通り置けるかを計算する問題）とした。これはバックトラックによる探索問題の代表的なものであり、並列性抽出についての言語の記述能力の評価に適する。バックトラックの操作はループで記述すると状態保存回復の部分で副作用を扱うため複雑になる。一方、関数の再帰呼び出しを用いるとその操作が自動的に行われる。従って、以下のプログラムでは再帰呼び出しを用いる。

マクロ言語

マクロ言語のモデルとして関数レベルの負荷分散を行うもの【１０】を想定する。このモデルでは、関数呼び出し以外は直列的に解釈され、関数の返値の待ち合せに関して幾つかの機能がある。付図.1にプログラムを示す。Ｎ－Ｑｕｅｅｎでは関数のインスタンス間に依存関係が無いため、成功した事例の数え上げはＣＡＬＬＥＲとＣＡＬＬＥＥの関係とは独立に行っている。このモデルではプロセッサ間の共有メモリを仮定しないのでｒｔｎａｄｄによって、両者のプロセッサが異なった場合、ｍａｉｎが実行されたプロセッサのｓｕｃｃｎｔにＣＡＬＬＥＥの値を加算してリターンする。

ミクロ言語

単一代入則にｆｏｒａｌｌの制御構造を付加したモデルで記述した（付図.2）。ｓｕｃｃｎｔについては、副作用を許さないためストリームを用いた。関数の引き数も値渡しである。プログラム中、内側のｆｏｒａｌｌで今のＱｕｅｅｎが置けるかどうかを判定している。ｆｒｅｅはこの処理が終了した後、初めて参照可能となる。これは置けなかった場合については直ちにｆｒｅｅ（０の場合）が参照可能となるべきであり、これを許す構文の導入が望まれる。これには、例えば、ｆｏｒａｌｌを関数としてとらえ条件による実行途中リターンを可能にする。実行途中リターンが発生しなかった場合は、実行終了時に評価される式（初期値に対する後期値と言うべきもの）の値を返す、等の方法が考えられる。また、関数呼び出しの引き数ａｒｇｖについて、毎回新しく作り直している。これは、配列においてはその要素の書き替え・追加は副作用となるため、順序関係を規定するためである。しかし、このプログラムのように書き替えられた要素について元のａｒｇｖの要素がそれ以上参照されない場合は、本質的に元の値を保持する必要はない。

両者の記述性における比較

直列的解釈では、現在分かっている結果から判断してそれ以上の余分な処理を中止することができるのに対して、並列化したために計算量の増加を招いている。このような性質を持つ並列性と、本質的に独立な部分の並列性は明確に区別されるべきであって、ミクロ言語での記述によってこれらが混然一体となってしまうのはむしろ有害である。また、並列処理部分の粒度についても、ハードウェア資源の効率的利用の観点から、問題の性質・計算量等によって適切な大きさが異なる。これらはプログラマが全体を意識しつつ適切なコーディングを行わない限り、また行える言語を用いない限り実現できない。勿論、プログラマに要求されるのはそれほど厳密なものではない。プログラマが取り出したい部分の並列性を簡潔に記述できることが肝心である。

```
/*      N-Queen (MACRO DATA FLOW LANGUAGE)      */

#define MAXQS   15
#define GRAINP  4       /* task grain parameter */

int  argv[MAXQS];
int  succnt = 0;        /* for other PEs */

main(){
    int n;

    succnt = 0;
    printf("\n++++ queen=");
    scanf("%d",&n);
    argv[0] = 2;        /* used array size */
    argv[1] = n;

    queen(argv);

    /* waittask();  check all task end */
    printf("\n++++ succnt=%d\n",succnt);
}

queen(argv)
int *argv;
{
    register i,j,jj,ix,n;

    ix = argv[0] - 2;
    n = argv[1];
    for(j=0; j<n; j++){
        for(i=0; i<ix; i++){
            jj = argv[i+2] - j;
            if(jj==0 || jj==i-ix || jj==ix-i)
                goto skip;
        }
        if(ix==n-1){
            succnt++;
            rtnadd(&succnt);
        }else{
            argv[ix+2] = j;
            argv[0] = ix + 3;
            /* distributable call */
            if(ix < GRAINP) queen<argv>;
            else queen(argv);
        }
    skip:
    }
}
```

付図.1 マクロ言語によるN－Queenプログラム

```
/*      N-Queen (MICRO DATA FLOW LANGUAGE)      */

#define MAXQS   15

int argv[MAXQS];
int succnt;
int stream suc;

main(){
    int n;

    printf("\n++++ queen=");
    scanf("%d",&n);
    argv[0] = 2;
    argv[1] = n;

    queen(argv);

    succnt = addall(suc);
    printf("\n++++ succnt=%d\n",succnt);
}

queen(argv)
int *argv;
{
    int i,j,jj,ix,n,free;

    ix = argv[0] - 2;
    n = argv[1];
    forall(j=0; j<n; 1){
        /* "free" is defined at
                the finish of the next forall */
        forall(free=1,i=0; i<ix; 1){
            int jj = argv[i+2] - j;
            if(jj==0 || jj==i-ix || jj==ix-i)
                new free = 0;
        }
        if(free)
            if(ix==n-1)
                suc = 1;
            else{
                forall(i=1; i<(ix+2); 1)
                    new argv[i] = argv[i];
                new argv[ix+2] = j;
                new argv[0] = ix + 3;
                queen(new argv);
            }
    }
}
```

付図.2 ミクロ言語によるN－Queenプログラム