計算機アーキテクチャ　68－4
（1987・11・12）

ＮＣモデルに基づくＭＩＭＤ型並列計算機のための高性能プロセッサ

A high performance processor for MIMD machines based on NC model

朴　泰佑、野村　茂弘、天野　英晴

Taisuke BOKU, Shigehiro NOMURA, Hideharu AMANO

慶應義塾大学　理工学部　電気工学科

Department of Electrical Engineering, Fuculty of Science and Texhnology, Keio University

１．はじめに

近年の科学技術計算はその応用範囲の広がりと問題領域の拡大によりますます大量の計算を必要としている。このようなニーズに対応するためにスーパーコンピュータは必要不可欠となっている。しかし、既存のスーパーコンピュータのほとんどはパイプライン方式のベクトル計算機であり、問題の系が不規則であったり、計算中に多くの条件分岐が存在するような場合、問題が持つ本来の並列性を十分に生かして効率の高い並列処理を行うことが極めて難しい。

そこで、この様な問題に対処するために最近ではＭＩＭＤ方式の科学技術計算用並列計算機の開発が活発化している。筆者らの研究室で開発中の一般疎行列専用並列計算機（ＳＭ）^{2}-IIもその一つである。（ＳＭ）^{2}-IIではユーザはＮＣモデルと呼ばれる並行プロセス・モデルに基づいてプログラミングを行い、計算は任意数の協調プロセスによって実行される。現在のプロトタイプではシステム内のプロセスの実行管理はＤＩＰＲＯＳと呼ばれる分散処理系によって行われている。

ＤＩＰＲＯＳはＮＣモデルに基づき、プロセス間の交信や切り替えを行うが、現状ではこれら全てをソフトウェアによって行っているため、システムの効率が低い。特に、プロセスの粒度の小さい問題を解く際に効率の低下は著しくなる。

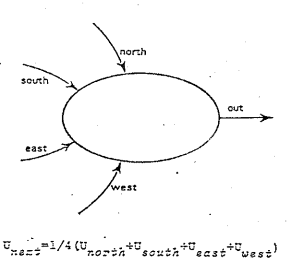
我々が現在開発中の高機能プロセッシング・ユニットは、このようなプロセス間交信（ＩＰＣ：InterProcess Communication）管理をハードウェアにより行い、また高性能な浮動小数点演算装置（ＦＰＰ：Floating Point Processor）を効率的に利用するコントローラを備えている。本報告ではこのプロセッシング・ユニットのアーキテクチャを提案する。

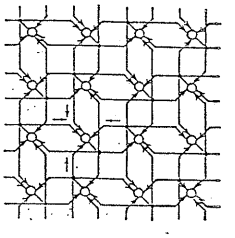
２．ＮＣモデル

科学技術計算の分野において、問題の対象となる系が存在し、その形状が終始変化しないような問題は数多い。例えば、有限要素法、偏微分方程式によるポテンシャルの計算、電子回路の過渡解析等がこれにあたる。このような問題では処理を行う複数の節点と、各節点の状態を他の節点に伝える交信網によって問題空間を表現することができる。

我々はこのような系で表わされる問題を効率的に並列処理するための問題記述モデルとしてＮＣモデルを提案した[1]。ＮＣモデルでは能動的な処理を行うノード（節点）と、ノード間のデータを転送するネットワーク（交信網）を使って問題を表現する。例えば、２次元のポアソン方程式を差分法によって計算する場合、空間分割された各領域をノードとみなし、隣接するノード間を結ぶ正方格子状の結合がネットワークの形状となる。各ノードにおける処理とネットワークを＜図１＞に示す。

ＮＣモデルにおけるノードとネットワークには次のような特徴がある。





＜図１＞ＮＣモデル

（１）ノード及びネットワークは処理の実行開始から終了まで存在し、その機能及び形状は変化しない（静的である）。
（２）各ノードには複数の入力または出力ポートがあり、ノードはこれらのポートを介してネットワークに結合される。
（３）ネットワークは各ポート間を結ぶ結合線により構成される。結合線は１つの出力ポートと１つ以上の入力ポートを結合したものである。
（４）ノード間のデータ転送はポートを介して行われる。データを出力するノードは出力ポートからデータを送る。そのデータは結合線を通して、接続されている全ての入力ポートに送られ、相手側のノードは対応する入力ポートからデータを受け取る。
（５）ノード間の同期は入力ポート側にある単一のバッファによってとられる。データの送信は、結合されている全ての入力ポートのバッファが空いていれば行うことができ、そうでなければ待たされる。

以上の特徴のうち、（４）のデータ交信方法をマルチキャストと呼ぶ。我々が対象としている問題においては一般的に一つの節点の状態は他の複数の節点に伝えられる必要がある場合が多い。従って、このような問題において、マルチキャストは効率の良い交信方法である。

ＮＣモデルにおけるノードは協調型計算システムにおけるプロセスとみなすことができ、ネットワークはプロセス間の交信経路である。つまり、ＮＣモデルは静的な系を対象とする並行プロセス・モデルである。

３．並列計算機（ＳＭ）^{2}-II

ＮＣモデルに基づいて記述されたプログラムを効率的に実行するためには、柔軟な交信形態を持つＭＩＭＤ型並列計算機が好ましい。本研究室ではＮＣモデルに基づくアーキテクチャ及びプロセッサ間交信機能を持つ一般疎行列専用並列計算機（ＳＭ）^{2}-II[2]を提案した。

（ＳＭ）^{2}-IIはＲＳＭ(Receiver Selectable Multicast)と呼ばれるＰＵ(Processing Unit)間交信機構を持ち、１ＰＵから複数のＰＵへのマルチキャストを効率よく行う。ＲＳＭは一種の仮想共有メモリ・システムであり、全てのＰＵは１つの仮想アドレス空間を共有するが、全てのアドレスに対して実際のデータ・メモリが存在するわけではない。データを送信するＰＵはＲＳＭ上の１つのアドレスにデータを書き込む。各ＰＵ上にはＲＳＭマネージャと呼ばれる選択機構が備わっており、書き込まれたアドレスによってそのデータがそのＰＵにとって必要であるかどうかを判断し、もし必要であればＲＳＭマネージャ内の受信データ・メモリに書き込み、必要でなければ交信に関係しない。これによって、データの必要性が各ＰＵ毎に独立に判断され、結果としてマルチキャストが実現される。このように、ＮＣモデルにおけるマルチキャストはＲＳＭを利用して効率的に行われる。

（ＳＭ）^{2}-IIにおけるプログラミング言語として、ユーザは並行プロセス記述言語，ＮＣＣを利用できる。ＮＣＣはＮＣモデルに基づく言語であり、静的なプロセス及びプロセス間を結合する結合線を通したマルチキャストをＩＰＣとして提供する。ユーザはＮＣモデルにおける基本概念をそのまま生かしてプログラミングを行うことが出来る。ＮＣＣに関しては次節で詳述する。

ＮＣモデルにおけるノード（プロセス）及び結合線のネットワークが静的であるため、計算実行中のプロセスの生成・消滅や、プロセス間結合線のトポロジーの変更は行われない。

４．ＮＣＣ(Node oriented Concurrent C)[3]

ＮＣＣはＮＣモデルに基づく並行プロセス記述言語である。ＮＣＣにおいては、ＮＣモデルにおけるノードをプロセスと呼ぶ。プロセスはプロセス・タイプと呼ばれるテンプレートから作り出すことができ、そのプロセスはそのプロセス・タイプの定義部に記述された処理を行う。また、一つのプロセス・タイプから複数個のプロセスを生成することができる。

１つのＮＣＣプログラムはプロセス宣言部、プロセス記述部、ネットワーク記述部の３つの部分から成る。プロセス宣言部では必要な全てのプロセスと、そのプロセスのプロセス・タイプを宣言する。プロセス記述部では各プロセス・タイプの処理を記述する。ネットワーク記述部では各プロセスが持つポート（出力ポート、入力ポート）の結合を手続き的に記述し、システム全体のネットワークを構成する。

ＮＣＣには一般的なＣ言語の機能に、ＩＰＣを実行する関数と、マルチ受信エントリをサポートするエントリ構文が追加されている。ＩＰＣ関数及びエントリ構文は以下の通りである。

・ＩＰＣ（プロセス間交信）関数

send(port,data)

出力ポートからデータを送信する。

receive(port,var)

入力ポートからデータを受信し、変数に格納する。

probe(port)

入力ポートにデータが到着しているかどうかを調べる。

・マルチ受信エントリ構文

```
entry{
  (port1,var1):
    statement1;
    continue または break;
  (port2,var2):
        .
        .
        .
  }
```

各入力ポートにデータが到着したら対応する変数に格納し、対応する文を実行する。各文の最後がcontinueの場合はさらに別のエントリの受信を待ち、breakの場合はその文を実行した後、まだ未受信のエントリがあっても構文を抜ける。

このように、ＮＣＣにおけるＩＰＣはポートを介したデータの送受信によって行われ、各ポートはネットワーク記述部において結合線によって結合される。この時、１つの出力ポートは複数の入力ポートと結合することができ、この結果、データのマルチキャストが実現される。

ＮＣＣはＮＣモデルの基本概念をすべて取り入れた言語で、分布系のシミュレーションや行列計算などの科学技術計算を並列アルゴリズムを使って容易に記述することができる。

５．ＤＩＰＲＯＳ

ＮＣＣにおいてユーザは任意個のプロセスを作成することができる。実際の計算はこれらのプロセスを並列実行させることによって行われる。この時、プロセスの数が(ＳＭ)^2-IIシステム上のＰＵ台数を上回った場合、１つのＰＵ内で複数のプロセスを並行実行させる必要が生じる。また、このような場合、ＩＰＣは、ＰＵ内ですむ場合もあれば、ＰＵ間交信となる場合もある。

そこで、このような実行時におけるプロセスの管理やＩＰＣのサービスを行うために、システム内にオペレーティング・システムが必要となる。(ＳＭ)^2-IIは、ＮＣＣの処理系及び(ＳＭ)^2-II本体のオペレーティング・システムとしてＤＩＰＲＯＳ[4]と呼ばれるシステム・ソフトウェアを持つ。

各ＰＵ上にはLocal Processing System（ＬＰＳ）が常駐し、それらがＲＳＭを介して交信し合うことによりＤＩＰＲＯＳを構成している。

ＬＰＳの機能を以下に挙げる。

（１）ＩＰＣのサポート

プロセスが出すＩＰＣ要求を受け、交信の方法（ＰＵ内交信かＰＵ間交信か）を判別してデータの受信バッファへのマルチキャスト（送信の場合）あるいはバッファからのデータの読出し（受信の場合）を行う。

（２）プロセスのスイッチング及びスケジューリング

実行中のプロセスがＩＰＣ待ちになった時、プロセスをサスペンドし、別の実行可能プロセスを走らせる。また、サスペンドされているプロセスのＩＰＣが終了後、そのプロセスが再起動されるようにスケジューリングする。

（３）マルチ受信エントリのサポート

マルチ受信エントリ構文を実行するプロセスのエントリを管理し、データの到着順に対応する文が実行されるようにスケジューリングする。

（４）ホスト計算機へのファイル・アクセス及び入出力のサービス

（ＳＭ）^{2}-IIはスタンド・アロン型の計算機ではない。そのため、ファイル・アクセスやユーザとの交信は結合されているホスト・マシンを通じて行う。プロセスから出された入出力要求はＬＰＳを介してホスト・マシンに伝えられる。

ホスト計算機上でコンパイルされたＮＣＣプログラムはＤＩＰＲＯＳによって(ＳＭ)^2-IIの各ＰＵにダウン・ロードされ，ＤＩＰＲＯＳの管理の下で実行される。計算中に発生するＩＰＣ要求やファイル・アクセス要求などはすべてＬＰＳによって処理され、他のＬＰＳとの交信またはホスト計算機との交信を行うことによって実行される。

ＤＩＰＲＯＳは現在、(ＳＭ)^2-IIのプロトタイプ２号機上に実装され、稼働中である。

６．高機能プロセッシング・ユニット

ＮＣＣによって記述できる問題領域は広く、これまでに種々のアプリケーションが作成されている。我々は様々なアプリケーション・プログラムを実行し、実際にプロセスを実行している時間、即ち有効稼働時間が全計算時間中に占める割合を実測によって求めた。この割合を有効稼働率と呼ぶ。

その結果、有効稼働率は４５％から２５％と、かなり低い値であることがわかった。ＤＩＰＲＯＳはプロセス及びＩＰＣに関する全ての処理をソフトウェアによって行っているが、この結果からそのオーバヘッド

が無視できない大きさであることがわかる。

頻繁にＩＰＣを行うようなプロセスはＩＰＣ処理時間だけでなく、プロセスのスイッチング回数及びスケジューリングに要する時間を増大させ、結果としてシステムのオーバヘッドを増加させる。従って、計算時間に対してＩＰＣの頻度が高いプロセス、即ち粒度の小さいプロセスが多いほど、有効稼働率は低くなる。

粒度の小さいプロセスから成る問題を効率的に解くためには、現在のソフトウェアによるプロセス及びＩＰＣの管理ではパフォーマンス不足である。このためには、ＤＩＰＲＯＳの全部あるいは一部の機能をハードウェア化し、計算（プロセス）の実行とＤＩＰＲＯＳの処理を並列に行う必要がある。即ち、１ＰＵ内にプロセスを実行するプロセッサと、プロセス・スケジューリングやデータのマルチキャスト、他のＰＵとの交信などを行うプロセッサの２つを設け、両者を並列実行させる。

メッセージ・パッシングをベースとする計算機ではこのようなオーバヘッドは一般的にかなり大きい。そこで、このようなメッセージ処理を専用のプロセッサを用いて負荷分散する研究が行われている[5][6]。また、外部から到着したメッセージにより自動的にプロセッサ内の処理が行われるようなアーキテクチャも提案されている[7]。しかし、我々のシステムではシステム全体のメッセージ転送だけではなくＰＵ内におけるメッセージ転送とそれに付随するプロセス・スイッチングも高速に実行する必要がある。このようなハードウェアを持ったプロセッサとしてはTransputer[8]が存在する。TransputerのＩＰＣはＣＳＰ[9]に基づくランデブーである。しかし、我々が対象としているような問題においては、ランデブーによる一対一交信よりも一枚バッファによるマルチキャストの方がより効率的な処理が行えると考えられる（ランデブーに基づく交信ではＩＰＣのブロック率が高くなってしまう）。従って、ＮＣモデルに基づくハードウェアＩＰＣサーバが必要である。

一方、科学技術計算では大量の浮動小数点演算を行うため、高性能なＦＰＰが利用される。そこで、このようなＦＰＰを有効利用するために、専用のＦＰＰコントローラを設ける。これにより、ＦＰＰはある程度独立に演算を行うことができるようになる（この必要性は第８節で述べる）。従って、１つのＰＵはこれら３つのモジュールから構成される。これらのモジュールをタスク・エンジン、ＩＰＣエンジン、ＦＰＰエンジンと呼ぶことにする。各モジュールの機能は次の通りである。

タスク・エンジン：ＮＣＣプログラムのプロセスを逐次実行する。プロセスがＩＰＣを行う際にコンテキストをＩＰＣエンジンに渡し、次に実行可能となった別のコンテキストをＩＰＣエンジンから受け取ってそのプロセスを続行する。

ＩＰＣエンジン：タスク・エンジンからのＩＰＣ要求を受け、ＰＵ内あるいはＰＵ間のマルチキャストを実行する。同時にＩＰＣの完了により実行可能となったプロセスをタスク・エンジンが実行できるようにスケジューリングする。

ＦＰＰエンジン：タスク・エンジンがプロセスを実行中に発生する浮動小数点演算要求を受け、一連の計算を行う。また、浮動小数点演算が他のプロセスからのデータ待ちである場合、データが到着し次第、タスク・エンジンを介さずＩＰＣエンジンによって直接起動されることもある。

これら３つのモジュールが並列に動作できるようになれば、システムのパフォーマンスは飛躍的に増大すると考えられる。しかし、これらの内、ＩＰＣエンジンはその機能の複雑さからハードウェア化するのは非常に難しい。特に、対象が一般的な並行記述言語であればＩＰＣを完全にハードウェア化するのは極めて困難である。しかし、$(SM)^2$-IIで使われているＮＣＣはモデルの単純さと、静的であるという点をうまく生かせば比較的簡単な構造でハードウェア処理できると考えられる。次節ではＩＰＣエンジンの機能と、それらをどのような方法でハードウェア化するかについて述べる。

7．ＩＰＣエンジンのアーキテクチャ

ＩＰＣエンジンの主な機能は（マルチ受信エントリを含む）ＩＰＣ要求の処理、プロセスのスケジューリングであるが、これらは密接に関係し合っており、切り離すことはできない。本節ではこれらの機能を総合的に処理するアルゴリズムとその実現に必要なアーキテクチャについて述べる。

ＩＰＣエンジン中には２つの流れ、即ちデータの流れとコンテキスト（プロセス）の流れがある。データの流れをバッファの状態遷移でとらえると、バッファは空の状態とデータが入った状態を往復する。また、コンテキストの流れは、送受信を待っている状態と送受信が完了して再実行可能になった状態からなる。これらの状態を管理する機構として次の５つの要素を考える。

・ＳＲＴ（Send Request Table）
送信要求を出したコンテキストを登録するテーブル

・ＲＲＴ（Receive Request Table）
受信要求を出したコンテキストを登録するテーブル

・ＦＰＱ（Free Port Queue）
対応するバッファが空である出力ポートを保持するキュー

・ＰＭＱ（Private Message Queue）
マルチキャストされたデータを受信プロセス別に保持するキュー

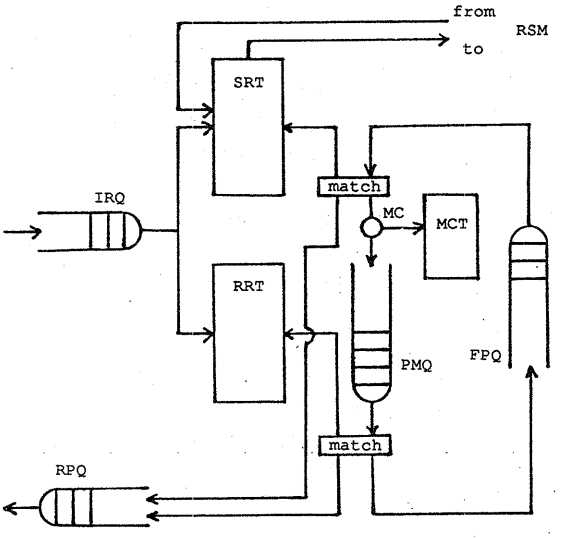
・ＲＰＱ（Ready Process Queue）
送受信が完了して再実行可能になったプロセスを保持するキュー

<図２>にＩＰＣエンジンのブロック・ダイアグラムを示す。これらのデータ構造を用いて、ＩＰＣの実行とプロセス・スケジューリングを行う方法は次の通りである。

まず、初期状態として全ての出力ポートをＦＰＱに入れる（全てのバッファは空である）。到着したＩＰＣ要求は送受信の別によりＳＲＴもしくはＲＲＴに登録される。システム内では絶えずスケジューラが動作しており、実行可能なＩＰＣを見つけ出す。スケジューラはＦＰＱを順にスキャンし、対応する送信待ちのプロセスをＳＲＴより見つけ、見つかった場合はデータを全受信プロセス用にコピーしてＰＭＱに入れると同時にそのプロセスをＳＲＴから削除してＲＰＱに入れる。また、スケジューラはＰＭＱを順にスキャンし、対応する受信待ちのプロセスをＲＲＴより見つけ、見つかった場合はそのプロセスをＲＲＴから削除してＲＰＱに入れる。この時、もしそのプロセスがそのマルチキャストにおける最後の受信者であった場合は、そのポートをＦＰＱに入れる。

以上のようにしてデータとプロセスはこれらのデータ構造の中を巡回し、ＩＰＣ及びスケジューリングが実現される。ここで、送信と受信の処理を分けてあるのは、両者の処理に並列性があり同時実行が可能であるためである。従って、ＩＰＣエンジンの中はリクエスト受付部、送信側スケジューラ、受信側スケジューラの３つの部分から成り、並列実行される。

以上はＰＵ内のプロセスの間だけで交信が行われる場合である。交信相手がＰＵ外にいる場合はＲＳＭを介した交信が必要となる。この場合、送信に関しては



<図２>ＩＰＣエンジンのブロック・ダイアグラム

直接ＲＳＭに出力される。出力後のデータのマルチキャストはＲＳＭにより自動的に実行される。また、ＲＳＭから受信したデータはＰＵ内でさらにマルチキャストする必要があるので、この場合はデータがあたかもＰＵ内のプロセスから送信されたようにＳＲＴに登録する。その後の処理は送信側スケジューラが行ってくれる。このように、ＩＰＣエンジンではデータのマルチキャストを、相手プロセスがＰＵ内にある場合もない場合も一貫して処理することができる。

マルチ受信エントリ構文のサポートは、基本的に受信側スケジューラによって行われる。この場合は、まずプロセスがエントリ構文に入る際、全受信エントリに対応する受信要求をＲＲＴに登録する（この時、再実行開始番地も一緒に登録しておく）。後は受信側スケジューラに任せればよいが、問題なのはそのエントリがbreakにより終了している場合である。この解決法として、ＲＲＴにはそのエントリがブレークであるかどうかを一緒に登録しておき、スケジューラが受信可能と判断した際に、そのプロセスに関する他の受信要求を全てＲＲＴから削除する。こうすればその後の受信は成立せず、ブレークが実現される。

以上のように、ＩＰＣエンジンでは３つの独立した機構によりＮＣＣの処理系として必要な全機能を高速に実行することができる。

８．ＦＰＰエンジン

科学技術計算用の計算機に使われるＦＰＰ（Floating Point Processor）は一般的に数MFLOPSという高い

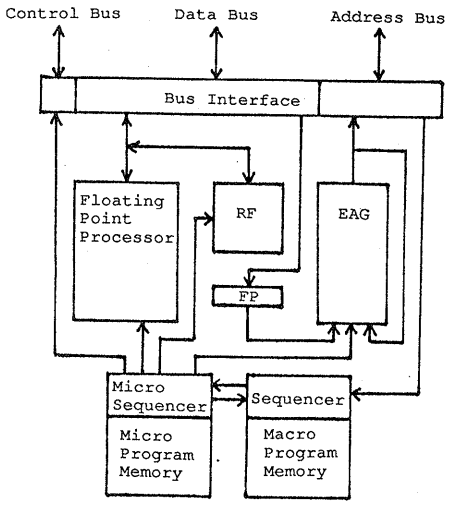
演算能力を持っている。しかし、この値はＦＰＰ単体の性能であって、実際の使用状況（例えばメイン・プロセッサによって起動される、データをメモリから読み取る等）においてはその能力を有効に発揮できるとは限らない。たとえ倍精度乗算が２００ナノ秒でできたとしても、そのデータを用意し、ＣＰＵから起動されて結果を返すまで数マイクロ秒かかっていては高性能のＦＰＰを使う意味がない。

いかにしてＦＰＰを有効利用するかは、ＦＰＰアクセスのためのハードウェア・アーキテクチャとコンパイラのインテリジェンスにかかっている。浮動小数点演算が必要な度に、ＣＰＵが命令を解釈して各命令毎にＦＰＰを起動する方式ではＦＰＰの能力を生かしきれない。そこで、まず、ある程度のマクロ命令をプログラムとして実行できるコントローラと、ユーザ・プログラム中の浮動小数点演算部分のみをコンパイルしたマクロ命令シーケンスを用意する。そして、ＣＰＵ（タスク・エンジン）はどのマクロ命令シーケンスを実行すべきかというトリガだけをコントローラに与えるようにする。ＦＰＰエンジンはこのようなＦＰＰとそのコントローラから成る。ＦＰＰエンジンのブロック・ダイアグラムを＜図３＞に示す。

ＦＰＰエンジンのコントローラは水平型マイクロ・プログラムに従って一連の命令（マクロ命令）セットを解読・実行する。また、単純なアドレッシング・モードを用意し、タスク・エンジンに頼らずにＦＰＰエンジンだけでメモリをアクセスできるようにすることで、ＣＰＵ－ＦＰＰ間のやりとりを最小限に抑える。コンパイラはプログラム中に浮動小数点演算があると、できるだけ長いマクロ命令シーケンスを生成し（これはＦＰＰエンジンを、できるだけ長期間タスク・エンジンの制御を受けずに動作させ続けるためである）、そのシーケンスの実行をトリガする命令だけをタスク・エンジンのプログラムとする。従って、コンパイラは高機能でなければならず、ある程度のインテリジェンスが要求される。

また、複数のプロセスを並行実行する場合、各プロセスのデータは局所変数に格納されている場合が多い。そこで、現在タスク・エンジンが実行中のプロセス（コンテキスト）のフレーム・ポインタの値をマクロ命令トリガの際に渡し、ＦＰＰエンジンではこのポインタを使ってオフセット計算を行うようにする。これにより、ＦＰＰエンジンはコンテキストを識別することなしに計算の実行が可能となる。

この方式ではタスク・エンジンとＦＰＰエンジンが並列に処理を行うことができる。但し、コンパイラはタスク・エンジンがＦＰＰエンジンの計算終了を待たなければならない点を検出して、その点で２つのエンジンの同期をとるようなコードを生成しなければならない。また、２つのエンジンは各々独立にメモリをアクセスするため、ＦＰＰエンジンがメモリをアクセスする場合は、タスク・エンジンにバス使用要求を出してバスを譲ってもらわなければならない。この調停作業はある程度の時間を浪費するため、両者のアクセスが頻繁に衝突するようなプログラムの場合、無理に並列実行をせずにタスク・エンジンを止めた方が良い場合も考えられる。この判断はやはりインテリジェンスを持ったコンパイラが行わなければならない。

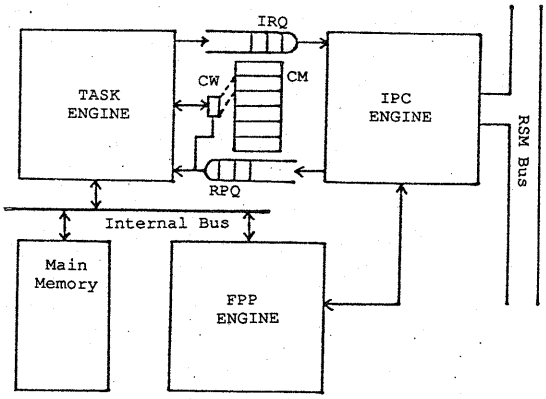


＜図３＞ＦＰＰエンジンのブロック・ダイアグラム

９．タスク・エンジンとＩＰＣエンジンの結合

以上３つのエンジン、タスク・エンジン、ＩＰＣエンジン、ＦＰＰエンジンについて述べてきたが、高機能プロセッサにとって重要なのはこれらのエンジンをどのように組合せ、交信させるかである。例えば前節の最後に触れたように、タスク・エンジンとＦＰＰエンジンがメモリを共有すると両者をうまく並列実行させることが難しくなる。この２つのエンジンの関係はどちらもユーザ・プロセスの実行の一部を行う点で共通なので、どちらかをメモリから切り離すわけにはいかない。しかし、例えばタスク・エンジンとＩＰＣエンジンのようにその機能が完全に分散されている場合はその関係に適した結合方式を考えなければならない。

高機能プロセッシング・ユニットの全体構成を＜図４＞に示す。タスク・エンジンとＩＰＣエンジンの関係は、一種のクライアント－サーバの関係である。即ち、タスク・エンジンがＩＰＣ要求を出し、ＩＰＣエンジンがその要求を受けると同時に既に要求が満たされたコンテキストをタスク・エンジンに返すことによ



＜図４＞プロセッシング・ユニットの全体構成

って両者の実行が継続される。この時、両者の動作が非同期に並列実行されるようにするためには、両者を結合するリクエスト及びリプライのキューが必要である。これらのキューをＩＰＣリクエスト・キュー及びレディ・プロセス・キューと呼ぶ。この２つのキューによって両者は互いに独立に動作できるようになる。

タスク・エンジン内のプロセスがＩＰＣ要求を出した場合、タスク・エンジンはＩＰＣリクエスト・キューに必要な情報を入れる。ＩＰＣエンジンは要求をキューから取り出し、必要な処理を行った後、再実行可能なプロセスをスケジューリングしてレディ・プロセス・キューに入れる。

以上の点で特に問題となるのは次の３点である。

（１）サスペンドされているプロセスをどのように管理するか

コンテキストは基本的にタスク・エンジンが管理する。このために、メイン・メモリとは別にコンテキスト・メモリを用意し、プロセスがサスペンドされている間、全ての環境をこのコンテキスト・メモリに保存する。そして、ＩＰＣエンジンはそのプロセスのプロセスｉｄによりプロセスを管理し、レディ・プロセス・キューにプロセスｉｄを入れることによりタスク・エンジンにどのコンテキストを再起動すべきかを教える。

また、コンテキスト・メモリへのアクセスを簡単化するためにレディ・プロセス・キューと連動して動作するコンテキスト・ウィンドウを用意する。コンテキスト・ウィンドウはコンテキスト・メモリ中の現在のコンテキストだけをタスク・エンジンに見せ、タスク・エンジンはこのウィンドウを通して環境の保存・復帰を行う。これにより、タスク・エンジンは実質的にどのコンテキストを実行中かをほとんど意識せずに処理を進めることができる。

（２）送受信データの受渡しをどのような形で行うか

ＩＰＣエンジンはメモリに結合されていないため、プロセスの変数に直接アクセスすることができない。そこで、データを送信する場合はタスク・エンジンがデータそのものをＩＰＣリクエスト・キューに入れてＩＰＣエンジンに渡す。また、受信されたデータはＩＰＣエンジンがレディ・プロセス・キューに直接入れ、タスク・エンジンがそのデータを対応する変数領域に書き込むことによって受信が成立する。

この方法は比較的大きなデータをやりとりする場合、効率が落ちる可能性があるが、現行のＮＣＣではＩＰＣデータ・タイプとして倍精度浮動小数のみをサポートしているためこの方法でも問題は生じない。

（３）マルチ受信エントリの管理をどのように行うか

マルチ受信エントリの場合、複数のコンテキストが同一プロセスで生じる点が問題である。ＮＣＣの文法ではマルチ受信エントリの全てのコンテキストの実行開始環境はエントリ構文に入る直前の状態であると定められている。そこで、タスク・エンジンが各コンテキストを起動する際はコンテキスト・メモリ中の環境をそのまま用い、そのコンテキストが終了した段階で環境をコンテキスト・メモリに戻さなければよい。但し、ＣＰＵのレジスタ中でプログラム・カウンタだけは各コンテキストにより異なるため、ＩＰＣリクエストを出す際にそのコンテキストの開始番地を一緒にＩＰＣエンジンに渡すようにする。

以上の様にしてタスク・エンジンとＩＰＣエンジンは２つのキューを介して結合するだけで全ての処理を行うことができる。

１０．ＦＰＰエンジンとＩＰＣエンジンの結合

前節ではタスク・エンジンとＩＰＣエンジンの結合について論じた。ＦＰＰエンジンはタスク・エンジンによって起動されるため、ＩＰＣエンジンとは直接結合される必要はない。しかし、ＮＣＣで記述された各種アプリケーションを調べると、マルチ受信エントリにおいて、受信後に実行される文は多くの場合簡単な（短いという意味ではなく制御のほとんどない）浮動小数点演算のみであった。このような場合、タスク・エンジンが実行する命令はＦＰＰエンジンの対応するマクロ命令シーケンスのトリガという極めて短い処理だけである。しかし、前節の様な構成だと、タスク・エンジンはこれだけのために環境の復帰を行わなければならず、非常に大きなオーバヘッドとなる。そのコンテキストの環境の中で、実際にＦＰＰエンジンに使用されるのはフレーム・ポインタの値だけであり、その他のレジスタ等は復帰する必要がない。

そこで、このようなＦＰＰエンジンだけを使用するコンテキストに対してはＩＰＣエンジンから直接ＦＰＰエンジンを起動することを考える。このような場合、ＩＰＣエンジンはマルチ受信エントリ用のＩＰＣリクエストの際に、ＦＰＰのマクロ命令シーケンスの実行開始番地とフレーム・ポインタを受け取っておき、それらを使って直接（タスク・エンジンを介さずに）ＦＰＰエンジンを起動する。この際、タスク・エンジンとＩＰＣエンジンの間でＦＰＰエンジンへのアクセスの衝突が生じるが、これは簡単な調停装置により制御できる。

もし、コンテキストが浮動小数点演算と通常の処理の両方を含む場合はいずれにせよタスク・エンジンを使用しなければならい。この場合は通常の方法で、タスク・エンジンによりコンテキストを起動し、そこからＦＰＰエンジンを起動させる。このどちらの方法を採るかはコンパイラにより決定される。

以上のように３つのエンジンはそれぞれ適した方法で結合され、各々が並列に実行されることにより極めてパフォーマンスの高い高機能プロセッサが実現できる。

１１．まとめ

本報告では、科学技術計算用ＭＩＭＤ型並列計算機のための高機能プロセッシング・ユニット（ＰＵ）のアーキテクチャを提案した。このＰＵは基本的に３つの処理モジュール、タスク・エンジン、ＩＰＣエンジン、ＦＰＰエンジンから成る。そして、ＮＣモデルに基づくマルチプロセッシングをハードウェア的に処理することができる。

これら３つのモジュール間はキューあるいはバスによって結合され、並列に動作する。特にタスク・エンジンとＩＰＣエンジンは定常的に並列動作することが可能であると考えられる。

当初の我々の目的は現在我々が開発中の一般疎行列専用並列計算機（ＳＭ）^{2}-II の性能をより高めるためのＩＰＣ処理機構のハードウェア化であった。しかし、ここで提案したアーキテクチャは（ＳＭ）^{2}-IIに限らず、ＮＣモデルに基づくＭＩＭＤ型計算機であれば利用可能である。特にＩＰＣエンジンは、タスク・エンジンのＣＰＵを選ばず、ＦＰＰへのダイレクト・アクセス機能を使用しなければ特殊なＦＰＰエンジンも必要としないため、ＣＰＵやＦＰＰのバージョン・アップに容易に対応できるような構成になっている。

現在我々は、アーキテクチャの完全な設計を行い、各種パラメータ（内部テーブルの大きさやキューの最大長など）を決定するためにエミュレータを作成中である。このエミュレータは提案したアーキテクチャとほぼ同じ構成をしているが、ＩＰＣエンジンの部分が汎用のＣＰＵとメモリから構成されている。これは、ＩＰＣエンジンの完全なハードウェア化が難しいため、構造やアルゴリズムの詳細を決定し、実際の計算を行ってタスク・エンジンとの並列実行度を調べるためのものである。

エミュレータのＩＰＣエンジンはソフトウェアによって実行されるが、タスク・エンジンと切り離され並列に動作するため、この点だけでも従来の（ＳＭ）^{2}-IIより高速になる。しかし、この形ではプロセスの粒度が小さい場合にタスク・エンジンがプロセス待ちのアイドル状態となり、処理効率の低下は避けられない。やはり、ＩＰＣ処理をハードウェア化することによって広範囲な粒度に対応できる高機能ＰＵが実現できる。

現在、ＩＰＣエンジンの許容プロセス数として１００プロセス程度を見込んでいる。ＲＳＭは一般的な共有メモリ・システムに比べバス・コンフリクトによる性能低下が少ないため、クラスタ構成をとることによる数千台のＰＵからなるシステムを構築することができる。このような大規模システムでは数万点以上の節点の計算を行うことも可能である。

エミュレータによる検証が終了しＩＰＣエンジンを含む高機能ＰＵの機能が実証された後は、実際にＩＰＣエンジンのハードウェア化を行う。また、将来的にはＩＰＣエンジンをＶＬＳＩ化することにより、３チップ構成の高機能ＰＵによる並列計算機を作成することを目標としている。

［参考文献］

［１］T.Kudoh,H.Amano,T.Boku,"NDL:A language for solving scientific problems on MIMD machines," Proc. of 1st Super Computing Symp., Dec.1985

［２］H.Amano,T.Boku,T.Kudoh,"(SM)^{2}-II:The new version of Sparse Matrix Solving Machine," Proc. of the 12th Int. Symp. on Comp. Arch., Jun.1985

［３］高橋範朗、他「大規模ＭＩＭＤ型並列計算機のための並行プロセス記述言語ＮＣＣ」第３２回情報処理学会全国大会、１９８６年３月

［４］T.Boku,H.Amano,T.Kudoh,"DIPROS:A distributed processing system for NDL on (SM)^{2}-II," Proc. of the 20th Hawaii Int. Conf. of System Sciences

［５］U.Ramachandran,M.Solomon,M.Vernon,"Hardware Support for Interprocess Communication," Proc. of the 14th Int. Symp. on Comp. Arch., Jun.1987

［６］J.W.Wendorf,H.Tokuda,"An Interprocess Communication Processor: Exploiting OS/Application Concurrency," CMU Internal Report, Mar.1987

［７］W.J.Dally,et.al.,"Architecture of a Message-Driven Processor," Proc. of the 14th Int. Symp on Comp. Arch., Jun.1987

［８］D.May,R.Shepherd,"The Transputer Implementation of OCCAM," Proc. of Int. Conf. of FGCS,

［９］C.A.R.Hoare,"Communicating Sequential Processes," Comm. of ACM, Aug.1978