

計算機複合体用接続装置における Concurrent Pascal の適用

— 抽象データ型に基づく並列処理について —

樋 口 哲 野
(慶応大学理工学部)

内 田 俊 一
(電子技術総合研究所)

1. まえがき

近年では LSI 技術の進歩によって安価に CPU やメモリを手に入れるようになってきた。そしてこれに伴い、複数個の CPU を組み合わせて、計算負荷の増大に対処しようとする多くの試みがなされるようになった。しかしこのマルチプロセッサに基づく処理能力の拡大の方法は、専用用途に対しては好結果を示すものが見出せるものの、汎用的なものになるとその実現は中々容易でないのが現状である。ハードウェアの設計は比較的容易にできても、その上で並列プロセスの効率的運用やマンマシンインタフェースの高度化をどう図るかといったソフトウェアの問題まで含めると、トータルなシステムとしての完成例は少ない。この理由の一つは、並列処理に際し、解くべき問題を「あくまでもプログラマビリティを損うことなく」、「できる限り自然な形で」、並列処理が可能な形態に分割することが一般に容易でないためである。例えば並列処理の実行単位を考えた場合、プログラム内の手続きや関数は、ごく自然に考えられる問題分割の単位である。しかし、副作用を意図的に利用するフォンノイマンモデルに基づく言語においてこれを行おうとすると、共有変数等の副作用を持つ存在に厳重な注意を払う必要が生ずる。この注意を怠ると、デバッグの困難な、時間に依存した再現性のないエラーを発生する源となる。このため必然的に並列プログラミングのコストが高くなり、使い易い並列処理システムの実現を阻害する一因となっている。従って、この並列プログラミングの面が改善されて、

そのコストが逐次型プログラムの場合と大差のない程度にまで低減できるようになれば、安価な LSI を生かした並列処理システムが真に実用価値を持つことになる。

このような背景を踏まえ、本論文では抽象データ型によって問題の自然な分割がもたらされる²⁾ 事実に着目し、並列性を持つ抽象データ型を一種のタスクと考えるマルチプロセッサ上で実行することを提案する。抽象データ型はソフトウェア方法論の立場で提案され、発展したものであるが、その計算の実行形態は、メッセージ交換型並列プロセスとの高い親和性を示す。そしてこの性質は、抽象データ型内の手続きや関数を単位とした並列処理が、プログラマビリティを極端に低下させることなく、ノイマン型 CPU を基にしたマルチプロセッサ上に実現できる可能性を示唆している。もちろん、抽象データ型に基づくプログラミングの過程で自然に生ずる分割単位が、すぐそのまま並列処理可能形態に対応するわけではなく、プログラムはある程度並列性の制御を明示しなくてはならない(この点ではデータフロー言語のように行かない)。しかしながら、その同期の手段の一つとなるモニタ自体も抽象データ型であり、従って並列性記述によってプログラマビリティが損われて並列プログラミングのコストが高くなることは少ないと考えられる。本論文ではこのような方式の有効性を、画像処理や行列計算向きの計算機複合体 PDP-11 における高機能接続装置 ICPL¹⁾ を対象にして具体的に示す。ICPL は複合体の各構成プロセッサと共通バスとを接続するが、その特徴はマ

マイクロコンピュータを内蔵し、接続しているプロセッサの処理を機能分散できる点にある。この機能により、ICPLはその構成プロセッサと、協調した並列処理を行える。ここでは、FFT実行時の1台の構成プロセッサとICPLの挙動を、拡張した Concurrent Pascal(CP)を用いて並列に実行可能な抽象データ型として捉える。そしてプロセッサとICPLの間でパイプライン的並列処理が行えることを示し、並列処理において抽象データ型を利用する本方式のメリットを明らかにする。

2. 抽象データ型の概念と並列処理との関連性について

2.1 抽象データ型の概念

Pascalにおけるレコード型等のデータ構造化機能は、整数型や実数型といった基本データ型を要素として、段階的に新しいデータ型を構成することを許す。このようにプログラマが問題に応じて任意に新しいデータ型を定義していける機能は、高い抽象化の手段であり、構造的なプログラム作成に欠かせない。一方、あるデータに対してある演算が施されるときに正しい結果を保証するためには、その演算が、そのデータの型に対して許されているものでなくてはならない。基本的なデータ型に対してはコンパイラがその検査を行い、誤動作を避けている。従って、データ構造化機能を用いてプログラマが任意に定義したデータ型に対しても、それに許される演算を予めプログラマがそのデータ型とひとまとめにして定義しておけば、プログラム中でその新しいデータ型のデータに対して正しい演算が施されているかどうかをコンパイラが検査することができ、高い信頼性と抽象性が得られる。このような考え方と Simula のクラスを融合させたものが抽象データ型の概念である。抽象

データ型言語によるプログラミングでは、プログラマはまず新たなデータ型とそれに関する手続きを定義し、次にそのテンプレートを用いて new 等の構文を用いて具体化 (instantiate) し、オブジェクトとしてプログラム内に存在させる。プログラマはそれらを用い、目的の計算を記述、遂行する。抽象データ型言語においては、プログラマは抽象データ型の内部構造を考慮する必要はなく、その外部的な挙動だけに注目すればよい。従ってモジュール化が可能になり、問題を抽象データ型に基づき、自然な形で分割できるようになる。

2.2 抽象データ型と並列処理の関連

抽象データ型におけるオブジェクトの概念は Simula のクラスに由来する。ブロック構造言語においては、ブロックで宣言された変数等がそのブロックから制御が離れた段階で消滅するのに対し、クラスは制御が離れた後も存在するものを表わす手段となる。抽象データ型の具体化であるオブジェクトも同様であり、従ってプログラム実行環境に複数のオブジェクトが並行して存在できる。一方、抽象データ型内の手続きの実行結果がほしい場合、プログラムの記述としては、その手続き名を指定するだけでよく、実行方法についての指示はいらない。これは how よりも what の記述に近く、従って抽象データ型に対してメッセージが伝達される形態といえる。大まかに言えば、抽象データ型によるプログラミングは、問題に応じ、並行して存在するオブジェクトを定義、具体化し、その間でのメッセージの交換を記述するものであるという捉え方ができる。従って、モニタ等の同期手段を用いてオブジェクト間の並列性を生かすことができれば、それは分散処理アーキテクチャ上のメッセージ交換型並列プロセスの実行にきわめて近い形態となる。以下では

この抽象データ型を利用した並列処理の利点を挙げる。第1は並列処理単位を選択である。並列処理の単位は、抽象データに基づくプログラミングの結果、自然に派生するものの中から利用するため、潜在するアーキテクチャを意識して問題分割を行ったりする必要がない。又、具体的な処理単位は抽象データ型内の手続きや関数に対応するが、その計算負荷はノイマン型CPUによる並列処理の単位に適している。第2は、並列性記述のし易さである。抽象データ型のプログラムは並列性制御情報を含まないため、プログラムは利用しうる並列性を認識して、その制御を記述しなければならない。しかしその同期手段となるモニタ自体も抽象データ型であり、全体のプログラミングスタイルに融和する。従って並列性記述によって著しくプログラマビリティが損われることは少ない。第3は、副作用による誤動作の危険の少なさである。抽象データ型内の手続きによるデータアクセスは、対応するオブジェクトのアドレス空間内だけに限られる。又、副作用を持つ共有変数をアクセスする場合には必ずモニタを用いなければならない。従ってノイマン型言語の場合のような、副作用の危険性はより小さい。第4に、メッセージ交換型プロセスの利点として、各プロセスが互いにどう影響するのかをプログラマに明瞭になることが挙げられる。以上に示した利点は、抽象データ型を利用した並列プログラミングのコストが逐次型のレベルまで低減できる可能性を示していると考えられる。

2.3 抽象データ型に関連する実用例

抽象データ型の概念に関連する言語やシステムの例を表1のように分類してみた。本論文の立場は表1の(3)に属するものである。(3)に挙げたもののうち、特に興味深いのはCm*とActorで

表1. 抽象データ型の概念に関連した言語、システム

(1)	抽象データ型、及びそれに近い効果を有するはん用、又はシステム記述言語 Clu, Smalltalk, Alphard, Euclid, Iota, Ada, Simula, etc.
(2)	OS等のシステム記述のために、抽象データ型に加えて並列性記述機能を持つ言語 Concurrent Pascal, Mesa, Modula, (Ada), etc.
(3)	分散処理アーキテクチャ上でオブジェクト指向の挙動を示すシステムや計算モデル Cm*/StarOSとTASK言語、Distributed Process, C. mmp/Hydra, Cluに対する拡張 ⁵⁾ Actor, etc.

ある。Cm*では、TASK言語によって、互いに関連する並列プロセスの集合であるタスクフォースを定義し、メッセージ交換型プロセスとして実行する。TASK言語は、データ抽象を実現するモジュールと呼ぶ単位を定義したり、モジュール間の関係を記述することによってタスクフォースを定義する。又、文献3では、メッセージ交換の概念に基づく計算モデルであるActorと、並列性を持つ抽象データ型との関連が指摘されている。

3. ICPLへの機能分散と抽象データ型との関連

3.1 ICPLのアーキテクチャ

ICPLは図1に示す共通バス方式の密結合計算機複合体POPSのための、高機能接続装置として開発された。POPSは、画像処理に関する実験や大型行列演算における、密結合計算機複合体の適用性を考察するための実験的システムである。現在はその一部が試作されており、図2のように、1MBの共有記憶、2台のミニコン、及びICPLから構成されている。ICPLの構成は図3に示すように、24ビット幅のALUを中心に構成され、350

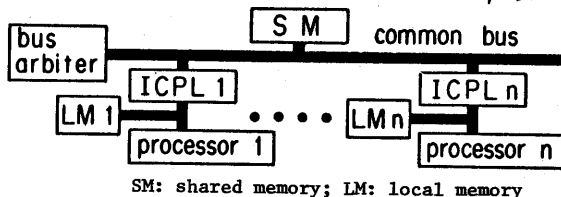


図1. POPSの構成

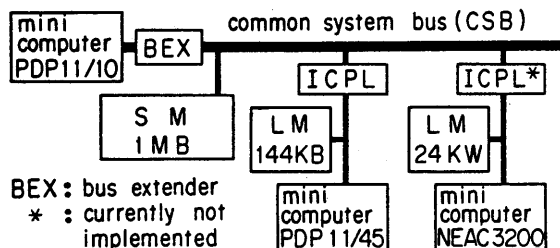
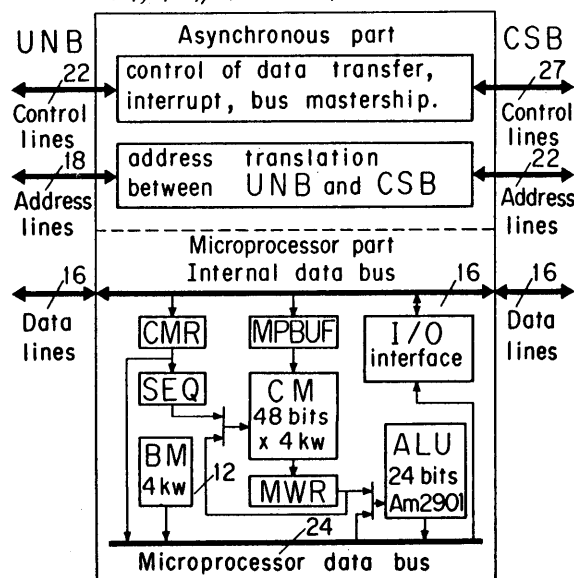


図2. 実装部分の構成

nsの実行速度を持つマイクロ命令を用いてデータ処理が行えるほか、共通バス上の共有記憶や接続しているPDP11/45の主記憶をアクセスできるので任意のデータ転送を行える。この機能を生かし、ICPLは次の2つの導入目的を果たしている。第1は、共通バス上の競合を低減することにより、複合体の構成プロセッサ数が増加した際の稼働率低下を防ぐことである。ICPLの接続しているミニコンが必要とする共有データを、ICPLがミニコンの代わりに高速に番地計算を行って主記憶にデータ転送することにより、共通バスの使用頻度を下げている。第2



UNB: PDP unibus; CM: control memory;
CSB: common system bus; BM: buffer memory;
MPBUF: microprogram buffer; SEQ: sequencer;
MWR: micro instruction register;

図3. ICPLの構成

の目的は、データ転送だけでなく、更にミニコンの処理をICPLへ機能分散することによって、複合体全体の処理能力を高めることである。

3.2 ICPLに抽象データ型を

適用する目的

前節で述べた機能分散の具体的な内容は次のようなものである。まず第1は、接続プロセッサへの演算データの転送である。その対象となるのは、共有記憶内の大型の2次元配列やその部分図形である。第2はデータ転送以外の機能分散であるが、これは問題に依存する。画像処理実験を例にとれば、種々の前処理や統計データの収集等が挙げられる。

上記の機能分散の第1の特徴は、いずれもICPLが大容量の共有記憶を対象としている点である。大型の構造化されたデータの中から必要部分だけを選択するためにはしばしば煩雑な番地計算を伴う。このため、ICPLの扱うデータとそれに関するアクセス手続きを一体化して抽象データ型として扱い、接続しているプロセッサの側から抽象化できれば、そのメリットは大きい。それは、ICPLがミニコンの下位プロセッサとして位置し、両者の実行するプログラムの関係が階層的になっているためである。例えば画像処理実験では、ICPLが原データを前処理して抽出した情報をミニコン側に送り、ミニコンではそれをもとにより高度な認識処理を行うといった形態が少なくない。従ってICPLの処理の抽象化は、ミニコン側のプログラム作成にとって有用である。

機能分散の第2の特徴はSIMD的な処理が利用できることである。つまり、SIMD的処理の場合には、ミニコンがICPLから供給されたi番目のデータセットに対して演算を行っている間に、ICPLは次のi+1番目のデー

データを共有記憶からアクセスするというパイプライン的並列処理を行い易い。その場合、ミニコン側の処理についても抽象データ型を基に記述できるならば、ICPLを含む全体のプログラムが抽象データ型を基に自然に分割されることになる。しかもその中の並列性を持つ抽象データ型が実際の並列処理ハードウェア上で利用可能となる。後述のFFTはこのような処理例である。ところで、要素プロセッサ(ミニコンとICPLのペア)間での共有記憶のアクセスについては次の前提をおく。すなわち、PとPSのOSが要素プロセッサごとにアクセスすべき共有記憶内の特定データ領域を定めることとし、複数の要素プロセッサが同一のデータ領域を同時にアクセスしないように排除するものとする。

4. Concurrent Pascal (CP) について

4.1 CPの選択理由

3.2で述べたように、PとPSの要素プロセッサ(ミニコンとICPLのペア)の上で機能分散を行うためには、その記述言語が並列性を持った抽象データ型を扱えることが条件である。CPやMesaはこれに合う(表1)が、単一プロセッサの多重化を実現しているだけであって、実際の並列マシン上での実行を支援しているわけではない。しかしCPでは、プロセッサの多重化を行っているカーネルを書き直すだけで、マルチプロセッサ上でCPを実行できることが示されている。文献4はマルチマイクロプロセッサ上でCPマシンを実現した例である。又、CPはコンパイルされるとPコードのレベルに対応する中間言語となるため、そのインタプリタとカーネルを、対象とするマシンの機械語で書くだけで移植ができる。このため本論文では、ミニコン、及びICPLの2つの処理装置を対象にし

てCPカーネルを再設計することにより、並列性を持つ抽象データ型の実行を支援する。CPは元来OS等のシステム記述用言語であり、CPに対する評価も、高級言語で初めて同期、交信を実際に書けるようにしたという点に集まっている。従って本論文のようにユーザ用の問題解決用言語としてCPを取り扱う試みはやや意外な感を与えるかもしれない。しかしCPはユーザ用抽象データ型言語としての使用も可能であり、少なくとも、並列性を有する抽象データ型を並列処理に利用しようという我々の主張を実証するには十分な機能を持つ。又、CPを選んだもう1つの理由は、CPのインタプリタをICPL上に実現することによって、マイクロプログラム計算機であるICPLがCPマシンになることである。つまり従来までは、機能分散させたい処理をマシンのマイクロ命令で書き、ミニコン側からICPLへ送っていた。しかしインタプリタ形式にすれば、ICPLの挙動をCPで書くだけで済み、プログラミングコストが大きく改善される。

4.2 CPの特徴

CPではプロセス、クラス、モニタという3つの抽象データ型を実現するシステムタイプが用意されている。ユーザはまずプログラムの始めでそれらを定義する。次に初期プロセスの中でそれらの具体化を行う。その後、具体化されたオブジェクトの間で制御が行き来し、目的とする処理が遂行される。

抽象データ型言語の性質に忠実に従うと、各オブジェクトのアドレス空間は互いに独立性の高いものでなくてはならない。このメモリ管理は、プログラムの実行中に動的にオブジェクトの生成を許す場合、少なからぬ手間を伴う。この問題は抽象データ型言語一般に言えることであって、抽象データ型

マシン実現の際のポイントの一つでもある。この点についてCPでは、PDP 11/45の、マッピングレジスタによる仮想記憶機構を用いて、各プロセスごとにマッピングレジスタを書き換え、別々の物理メモリを指すように工夫している。加えて、オブジェクトの動的生成を禁ずることによってメモリ管理の負担を軽減している。このためプログラム実行の最後の方で必要となるようなオブジェクトでも、予め初期プロセス内で起動しておかねばならない。この制約によりコンパイラはプログラムの実行前に、メモリ管理に関する情報をすべて決定できる。

5. CPの実現方法について

CPを要素プロセッサの記述言語として実装するに当り、次の方針をとる。すなわち、モニタ呼び出し、スケジューリング、プロセス切替え等を行うCPカーネルの機能をICPLのファームウェアで実現することにより、並列処理のオーバーヘッドを小さくする。これは計算時間の短い抽象データ型内手続き性に対しても並列処理の導入効果を得るためである。この方針に伴い、PDP 11/45、及びICPLのメモリ配置は次のようになる。PDP 11/45の主記憶には、PDP 11/45によって実行されるインタプリタ、ICPLとPDP 11

/45で共有されるCPコード、モニタ、プロセス内データが置かれる。一方、ICPLの中の制御記憶(CM)にはICPLによって実行されるインタプリタと、PDP 11/45とICPLの双方のインタプリタ内から実行要求の出されるカーネルが置かれる。現在、この方針に基づき、ICPL用インタプリタ、及びカーネルのファームウェア化を行っている。又、共有記憶内のデータ参照については、そのデータと同じ大きさのものをCPプログラム内に宣言し、それ(PDP 11/45の主記憶上にある)を二次元番地変換機構⁶⁾によって、共有記憶内の該当データに写像する方式を検討中である。

6. FFTにおける応用例

1次元FFTを例に、抽象データ型を利用した並列処理の実例を示す。プログラムは付録に、オブジェクト間の参照、実行関係は図4に示す。FFTの施される1行分のデータは、POPSの共有記憶に置く(このデータをPDPの主記憶に写像するため、プロセスgetvecshfle内に変数sharmemvecを宣言)。モニタ、プロセスはすべてPDPの主記憶に置く。又、オブジェクト名とその抽象データ型の対応は初期プロセスで付けられる。まずICPLはプロセスgetvecshfleを実行し、共有記憶上のデータ

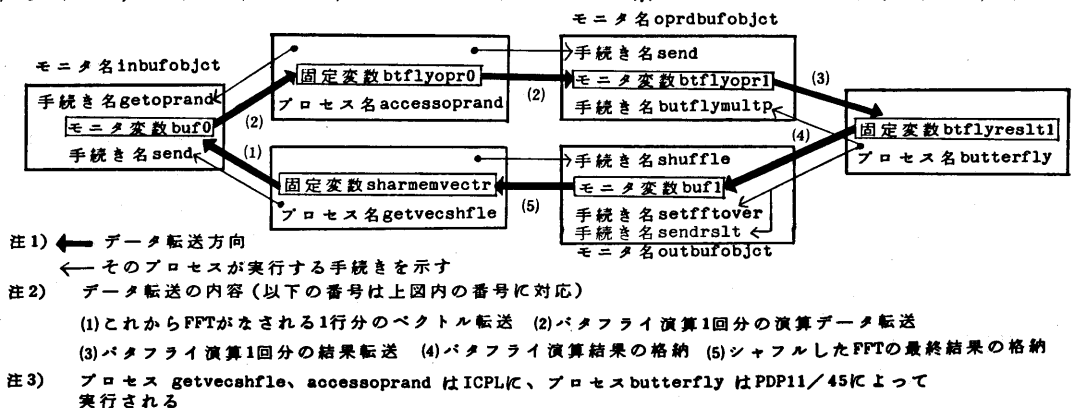


図4. オブジェクト間の参照関係

をPDPの主記憶(モニタinbufobject)に転送する。その後FFTが次の形態で進行する。FFTの演算は大別して、バタフライ演算データのアクセスと、それに基づくバタフライ演算の2つから成る。そこで演算データのアクセスをICPL(プロセスaccessoperand)に、バタフライ演算をPDP 11/45(プロセスbutterfly)に行わせる。この際、i番目のデータに対してバタフライ演算が行われているときに、i+1番目のデータをアクセスするというパイプライン型並列処理が図5のように達成される。これを支援するためにモニタoprdbufobjectを置き、同期をとっている。バタフライ演算の結果はモニタoutbufobjectに置かれる。すべてのバタフライ演算が終わると、プロセスbutterflyがフラグfftoverをセットする。それによってプロセスgetvecshfleがシャフルを開始し、シャフルを行いながら、そのFFT結果を共有記憶内へ格納する。以上の説明と、図4のオブジェクト間の実行関係から明らかなように、ここでの並列処理は抽象データ型を単位としており、プロセスの同期、交信もモニタを用いて明確に表現されている。又、プログラムの構造も整っており、並列性記述のために特にプログラマビリティが阻害されたということはない。但し、敢えて難を言えば、CPでは組込み関数が限られているために、自分で定義しなければならない(ex. odd)ことが挙げられる。

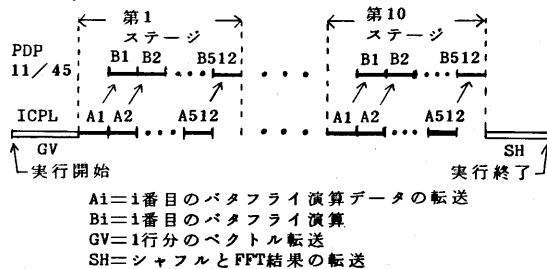


図5. FFT 実行タイムチャート

7. おわりに

本論文では、抽象データ型に基づく自然な問題の分割単位を並列処理に利用することを提案した。FFTの例で明らかにしたように、並列プログラミングのコストを改善する有効な方策の一つと考えられる。今後の課題としては動的なオブジェクトの生成の禁止が問題解決の上でどのような制約を与えるのかを考察することが挙げられる。

末筆ながら、日頃御指導頂く電総研石井治ソフトウェア部長、情報システム研究室棟上昭男室長と室員の皆様、方式研古谷立美技官、慶応大学相磯秀夫教授に感謝する。

参考文献

- (1) 樋口、内田："高機能接続装置による計算機複合体の接続方式とその評価", 信学論, Vol. -J65D, No. 2, 1982.
- (2) 佐渡、米ざわ："抽象データ型言語", Vol. 22, No. 6, 情報処理, 1981.
- (3) 米ざわ："Actor理論について", Vol. 20, No. 7, 情報処理, 1979.
- (4) 古谷："マルチプロセッサにおけるConcurrent Pascalマシン", Vol. 21, No. 2, 情報処理, 1980.
- (5) Liskov："Primitives for Distributed Computing", 7th ACM OS Conf., 1979.
- (6) 野島、天野、内田："複合計算機システムにおける計算機接続装置と番地変換機構", 情報17回全国大会.

(付録) C PによるFFTプログラム

```

**** FFT program for 1024 complex data by Concurrent Pascal
type datapoint=record realpart:real; imgnrypart:real end;
type btflyresult=record resltxreal, resltximgnry, resltreal,
  resltymgnry:real; rindex1, rindex2: integer end;
type btflyoprands=record xreal, ximgnry, yreal, yimgnry, twid1,
  twid2:real; indx1, indx2:integer end;
type line=array(1..1024.) of datapoint;

*****
input buffer --- will hold the whole vector on which
***** FFT is performed.
type inputbuffer=
monitor
var buf0:line; sintable, costable:array(1..256.) of real;
  index1, index2:integer; sender, receiver:queue;
function odd(y:integer):boolean;
begin if (y mod 2)=1 then odd:=true else odd:=false;
end --- of boolean function odd;
function exponent(a,b:integer):integer; --- gets a to power b
var x,y,z:integer;
begin x:=a; y:=b; z:=1;
  while y>0 do begin if odd(y) then z:=z*x; y:=y div 2;
    x:=x*x; end;
  exponent:=z;
end --- of integer function exponent;
function bitreverse(reversedpart, noofrevbit:integer):integer;
var i, sumbitrev, tempk, tempm:integer;
begin sumbitrev:=0; tempk:=reversedpart;
  for i:=1 to noofrevbit do
    begin tempm:=tempk div 2;
      sumbitrev:=sumbitrev*2+(tempk-tempm*2); tempk:=tempm;
    end;
  bitreverse:=sumbitrev;
end --- of integer function bitreverse;
function reversecount(counter, noofrevbit:integer):integer;
var reversedpart, exp, temp:integer;
begin exp:=exponent(2, noofrevbit);
  temp:=(counter div exp)*exp; reversedpart:=counter-temp;
  reversecount:=temp+bitreverse(reversedpart, noofrevbit);
end --- of integer function reversecount;

```

```

procedure getdatapair(counter, noofrevbit, offset: integer;
var btflyopr2: btflyoprnd);
begin
    var btflyopr2:=btflyoprnd;
    -- gets operands for butterfly multiplication from inputbuffer"
    begin index1:=reversecount(counter, noofrevbit);
    index2:=index1+offset;
    with btflyopr2 do begin xreal:=buf0(index1).realpart;
    ximgnry:=buf0(index1).imgnrypart;
    yreal:=buf0(index2).realpart; yindx1:=index1;
    yimgnry:=buf0(index2).imgnrypart; yindx2:=index2;end;
    end "---- of procedure getdatapair";
procedure gettwiddle(stagenumber, index: integer;
var btflyopr1: btflyoprnd); "---- gets
twiddle factors which are also operands for butterfly"
var a, b, ptpower, tableindx: integer;
begin a:=index*exponent(2, stagenumber-1);
b:=a div 512; ptpower:=a-b*512;
case ptpower div 128 of
0: begin tableindx:=ptpower+1;
btflyopr1.twid1:=costable(tableindx);
btflyopr1.twid2:=sintable(tableindx); end
" --- computation from case 1 to 6 are abbreviated here"
7: begin tableindx:=1024 div 4-ptpower*2+1;
btflyopr1.twid1:=--costable(tableindx);
btflyopr1.twid2:=sintable(tableindx); end;
end "---- of procedure gettwiddle";
procedure entry getoprnd(counter, noofrevbit, offset,
stagenumber: integer; var btflyopr1: btflyoprnd);
begin getdata(counter, noofrevbit, offset, btflyopr1);
gettwiddle(stagenumber, index1, btflyopr1);
end "---- of procedure getoprnd";
procedure entry send(sharmemvectr: line);
begin if full then delay(sender);
buf0:=sharmemvectr; full:=true; continue(receiver);
end "---- of procedure send";
begin full:=false; "initial statements"
end "---- of monitor inputbuffer";

*****
output buffer --- will hold results of FFT and be shuffled
***** by getvecshfle process."
type outputbuffer=
monitor
var buf1: line; fftover: boolean; shuffler: queue;
function odd(y: integer): boolean;
-- the same one as in 'monitor inputbuffer"
function exponent(a, b: integer): integer;
-- the same one as in 'monitor inputbuffer"
function bitreverse(reversedpart, noofrevbit: integer): integer;
-- the same one as in 'monitor inputbuffer"
function reversecount(counter, noofrevbit: integer): integer;
-- the same one as in 'monitor inputbuffer"
procedure entry setfftover; "----called by 'butterfly process"
begin fftover:=true; continue(shuffler);
end "---- of procedure setfftover";
procedure entry shuffle(var sharmemvectr: line);
"---- called by 'getvecshfle process'. The result of FFT
which is in outputbuffer is shuffled and stored into
permanent variable 'sharmemvectr' in 'getvecshfle."
var i, rev1: integer; treal, timgnry: real;
begin if not fftover then delay(shuffler);
i:=0;
while i<512 do begin
rev1:=reversecount(i, 10);
treal:=buf1(i+1).realpart;
timgnry:=buf1(i+1).imgnrypart;
sharmemvectr(i+1).realpart:=buf1(rev1+1).realpart;
sharmemvectr(i+1).imgnrypart:=buf1(rev1+1).imgnrypart;
sharmemvectr(rev1+1).realpart:=treal;
sharmemvectr(rev1+1).imgnrypart:=timgnry;
i:=i+2; end;
end "---- of procedure shuffle";
procedure entry sendrslt(btflyreslt1: btflyresult);
begin with btflyreslt1 do begin
buf1(rindex1).realpart:=resltxreal;
buf1(rindex1).imgnrypart:=resltximgnry;
buf1(rindex2).realpart:=resltyreal;
buf1(rindex2).imgnrypart:=resltyimgnry; end "---- of with";
end "---- of procedure sendrslt";
begin fftover:=false; "initial statements"
end "----of monitor outputbuffer";

*****
operand buffer --- holds operands for butterfly
***** multiplication"
type oprandbuffers=
monitor
var btflyopr1: btflyoprnd; full: boolean;
sender, receiver: queue;
procedure entry btflymultp(var btflyresult1: btflyresult);
var temp1, temp2: real;
begin if not full then delay(receiver);
with btflyopr1, btflyresult1 do begin
resltxreal:=xreal+yreal; resltximgnry:=ximgnry+yimgnry;
temp1:=xreal-yreal; temp2:=ximgnry-yimgnry;
resltyreal:=twid1*temp1-twid2*temp2; rindex1:=indx1;
resltyimgnry:=twid2*temp1+twid1*temp2; rindex2:=indx2;
end "of with statement";
"--- now permits 'accessdata process' to use this monitor"
full:=false; continue(sender);
end "---- of procedure btflymultp";
begin full:=false; "initial statements"
end "---- of monitor oprandbuffer";

*****
vector-fetch & shuffle process --- sends source vector,
***** on which FFT will be
performed, to inputbuf. After other processes halt, this
process performs shuffling."
type vecshufprocess=
process(inputbuf0: inputbuffer; outputbuf0: outputbuffer);
var sharmemvectr: line;
begin inputbuf0.send(sharmemvectr);
outputbuf0.shuffle;
end "---- of process vector-fetch & shuffle";

*****
access operand process -- activates access of butterfly
***** operand which is to be executed
by monitor inputbuffer. This
process is executed by ICPU."
type acsoprprocess=
process(inputbuf0: inputbuffer; oprandbuf0: oprandbuffer);
var btflyopr0: btflyoprnd;
stagenumber, noofrevbit, counter, offset: integer;
begin
offset:=512; noofrevbit:=10;
for stagenumber:=1 to 10 do
begin
counter:=0;
while counter < 512
begin
inputbuf0.getoprnd(counter, noofrevbit, offset,
stagenumber, btflyopr0);
oprandbuf0.send(btflyopr0);
counter:=counter+2;
noofrevbit:=noofrevbit-1;
offset:=offset div 2;
end;
end "---- of process access-operand";

*****
butterfly process -- gets results of butterfly multiplication
***** by activating monitor procedure 'btfly-
multp' in oprandbuffer."
type btflyprocess=
process(oprandbuf0: oprandbuffer; outputbuf0: outputbuffer);
var btflyreslt1: btflyresult; i, j: integer;
begin for i:=1 to 10 do
begin for j:=1 to 512 do
begin oprandbuf0.btflymultp(btflyreslt1);
outputbuf0.sendrslt(btflyreslt1);
end;
end;
outputbuf0.setfftover;
end "---- of process butterfly";

*****
initial process
*****
var inbufobjct: inputbuffer; outbufobjct: outputbuffer;
oprdbufobjct: oprandbuffer;
getvecshfle: vecshufprocess; butterfly: btflyprocess;
accessoprnd: acsoprprocess;
begin
init inbufobjct, outbufobjct, oprdbufobjct,
getvecshfle(inbufobjct, outbufobjct),
accessoprnd(inbufobjct, oprdbufobjct),
butterfly(oprdbufobjct, outbufobjct);
end "---- of initial process".

```