ICOT Technical Memorandum: TM-0399

TM-0399

ICOT-WG Workshop on Parallel Inference Machines and Multi-PSI Systems

田中秀彦(東大)，内田俊一，後藤厚宏
他

October, 1987

ICOT
Mita Kokusai Bldg. 21F
4 28 Mita 1 Chome
Minato ku Tokyo 108 Japan
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

ICOT-WG Workshop
on
Parallel Inference Machines
and
Multi-PSI Systems

July 17-18, 1987

ICOT

ＰＩＭ／ＭＳＩＭ－ＷＳの概要

ＩＣＯＴ　第四研究室

１．開催の経緯

ＰＩＭ／Multi-ＰＳＩ　ＷＧでは，国内，海外の並列処理システムの研究動向の紹介と討論を行うとともに，中期ＰＩＭ，マルチＰＳＩ，ＰＩＭＯＳ等，ＩＣＯＴにおける並列推論マシンの研究開発について議論を重ねてきた。

このような活動を通して，ＷＧの委員，オブザーバの相互理解が深まるとともに，並列推論システムに関する問題意識も具体化してきた。

そこで，ＷＧのメンバを中心にワークショップを開催し，ＩＣＯＴのＰＩＭの研究開発のこれまでの経緯を中心に，並列アーキテクチャの最近の研究動向を整理し，将来の研究方向を探ることとした。討論においては，並列論理型言語，および並列ソフトウェアも含めて並列推論マシンに関していろいろな角度から議論することとした。

２．開催場所，日時，参加人数

１９８７年７月１７日（金）１３：００から７月１８日（土）１３：３０まで

ＮＴＴ 葉山 憩の家　　〒240 - 01 神奈川県 三浦郡 葉山町 一色 2258

総参加者数　　５２名

内訳　大学関係　　１２名
電総研　　１名
関連会社　　１２名
ＩＣＯＴ　　２７名

発表件数　　２２件

総 括

田中　英彦（東京大学）

２日間にわったて行なわれたＰＩＭ－ワークショップ。並列処理を研究している第一線の研究者が集まり、ことの本質をついた非常に活発な楽しい議論が行なわれ、１日目は夜１２時に迄及んだ。

話され、議論が闘わされた内容としては、個人的には次のような点が印象的であった。

1　並列処理の応用問題

大きなプログラムをどんどん書き、ベンチマークプログラムを充実させることが大切。

2　並列プログラミング

プログラミング手法は、逐次の場合と異なる、従って、その手法確立が必要である。また、並列アルゴリズムを次々とため込んでゆくことも大切である。

3　ＧＨＣ

ＯＲ並列に比して、記述力は確かにあるが、未だプログラミング言語としては改良の余地があり、自己記述力をもたせることが必要なのではないか。また、多くのプログラマに使ってもらう為にその環境整備が必要である。ハードウェアからみると、すべてがコミットであるというのは厳しい。

4　並列アーキテクチャ

設計のポイントは、データ共有方式、負荷分方散式、プロセッサ間の結合トポロジである。結合網は、一様性・拡張性を重視するか、実装性・実用規模を考えるかで結論が異なる。制御方式としては、如何に処理の局所性をうまく引き出すかである。

5　将来に向けて

コネクションマシンに見られる超並列マシン、メッセージ通信の利用、プログラマにとって柔軟性・信頼性のあるアーキテクチャ、三次元・光素子・超電導等新しい素子技術の積極利用等を考えたい。

第五世代プロジェクトも、まる５年が過ぎ、その中心テーマの１つである並列処理の研究もベースが出来上って来た感がある、前期の始めのころと異なって、並列処理を一通り研究してきた経験のある研究者ばかりである。並列処理の難しいポイント、問題点が見えてきた段階にあって、今、過去を振返り将来を議論するのは実に時機を得ていた。

並列処理を討論する場合の用語が研究者間に常識化し、議論がかみ合うようになって来ている。しかし、問題点が見えて来たとは言っても、それらに対する解決策がすべて出揃った訳ではない。逆に、ますますその広大な研究領域が明らかになり、目の前に拡がって来たと言った方が正確であろう。そして、ややもすればその問題点の多さの足が竦む感がある。並列処理は実に奥が深いのである。しかし、問題点をもう一度吟味してみると、「完全に理想的な並列処理」には程遠いかも知れぬが、現在の技術で実現可能な並列処理が、思ったより近い所にあることに気ずく。

今、為すべきことは、ともかくも絶対性能の高いマシンを作ってみせることであろう。そして、それを補うものとして、並列度が非常の大きい場合への適用が可能な処理モデル、マシンモデルに対する足がかりをつけることであろう。

１０年前迄の研究は，先達の歩んだ後を追えばよく、安易にその成果が得られた。しかし、これからはそうではない、完全なオリジナル研究は、１００に１が成功であればしめたものである。研究コスト成果比等を云々するのは、技術後進国の研究である、失敗を恐れず多くの試みを繰り返し、真にすばらしい、世界をリードしてゆくオリジナリティを生成したいものだと思う。並列処理は、それに値する研究分野であり、現に、我国に於いて、そのような兆しが幾つか見え始めている。ＩＣＯＴの研究も確かにその流れの中にあるものである。並列処理研究者層の拡大を望みたい。

Program & Organizing Committee

General Chairman	田中　英彦	（東京大学）
	Prof. Hidehiko TANAKA	
Program Committee	内田　俊一	（ＩＣＯＴ）
	Shunichi UCHIDA	
	後藤　厚宏	（ＩＣＯＴ）
	Atsuhiro GOTO	
Organizing Committee	松本　晃	（ＩＣＯＴ）
	Akira MATSUMOTO	
	稲村　雄	（ＩＣＯＴ）
	Yuu INAMURA	

Participants

田中 英彦（東京大学）	相田　仁（東京大学）	小池 汎平（東京大学）
和田 耕一（筑波大学）	柴山　潔（京都大学）	松田 秀雄（神戸大学）
小畑 正貴（岡山理科大学）	富田 眞治（九州大学）	吉田 紀彦（九州大学）
平木　敬（電総研）	伊藤 總義（沖電気工業）	川上　桂（松下技研）
久門 耕一（富士通研）	杉江　衛（日立製作所）	中島　浩（三菱電機）
横田　実（日本電気）		

喜連川 優（東京大学 生研）	M.Nilson （東京大学）	山内　宗（東京大学）
雨宮 真人（ＮＴＴ基礎研）	小倉　武（ＮＴＴ厚木通研）	長沼 次郎（ＮＴＴ厚木通研）
平田 圭二（ＮＴＴ基礎研）	西川　宏（松下技研）	幅田 伸一（日本電気）

伊藤 英則	内田 俊一	市古 喬男	二宮 敏彦	生駒 憲治
上田 和紀	後藤 厚宏	松本　明	中川 貴之	佐藤 正俊
木村 康則	西田 健次	六沢 一昭	清水　肇	今井　明
瀧　和男	近山　隆	中島 克人	市吉 伸行	宮崎 敏彦
佐藤 裕幸	杉野 栄二	越村 三幸	金枝上 敦史	稲村　雄
吉田 かおる	寿崎 かすみ			

PROGRAM

July 17(Friday)

・ Registration　　13:30 ～

1. Opening Address　　田中 英彦（東京大学）　　14:00 ～ 14:10

2. ＰＩＭの将来に向けての私の経験？　　14:10 ～ 15:30

・Chairman　　田中 英彦

2.1　並列推論マシン：前期の結果から
久門 耕一（富士通研）

2.2　中期ＰＩＭに対する私見
杉江　衛（日立製作所）

2.3　データフローマシンとＬＳＩ技術
伊藤 徳義（沖電気工業）

2.4　並列処理アーキテクチャを考えるとき気になる二つの問題
雨宮 真人（ＮＴＴ基礎研）

3. 中期プロジェクト　　16:00 ～ 17:00

・Chairman　　内田 俊一

3.1　ＰＩＭの中期計画を作るに当たって考えたこと
内田 俊一（ＩＣＯＴ）

3.2　並列処理文化づくりに向けて
瀧　和男（ＩＯＣＴ）

3.3　並列マシンはかくありたい
中島 克人（ＩＣＯＴ）

4．並列アーキテクチャに対する私の主張　　19:00 ～ 21:00

・Coordinator　富田 眞治

4.1　並列アーキテクチャに対する私の主張
富田 眞治（九州大学）

4.2　並列アーキテクチャに対する私の主張
相田　仁（東京大学）

4.3　放送機能を持つプロセッサアレイによる並列ユニファイアの構想
和田 耕一（筑波大学）

4.4　マルチプロセッサシステムＰＡＲＫ上の並列Ｐｒｏｌｏｇ処理系について
松田 秀雄（神戸大学）

4.5　並列アーキテクチャに対する私の主張
小畑 正貴（岡山理科大学）

4.6　並列アーキテクチャに対する私の主張
吉田 紀彦（九州大学）

4.7　メモリ割当の並列性について
川上　桂（松下技研）

4.8　P^3でほんとにいいの？
中島　浩（三菱電機）

4.9　並列アーキテクチャ研究に対する私の主張
小池 汎平（東京大学）

4.10　並列アーキテクチャに対する私の主張
横田　実（日本電気）

July 18(Saturday)

5．これから何をすべきか，どうありたいか　　09:30 ～ 12:00

・Chairman　　内田 俊一

5.1　論理型言語で記述された応用問題のための並列計算機アーキテクチャについて
平木　敬（電総研）

5.2　アーキテクチャ研究におけるＰＩＭ開発プロジェクトの役割
柴山　潔（京都大学）

5.3　デバイス技術の発展／限界と並列マシンアーキテクチャ
― 並列マシンアーキテクチャに新たな枠組を！ ―
小倉　武（ＮＴＴ厚木通研）

5.4　並列計算機研究の方向はこうあるべきである
近山　隆（ＩＣＯＴ）

5.5　宇宙服は要らない！
吉田 かおる（ＩＣＯＴ）

6．Closing Address　　田中 英彦
"本ＷＳを総括して‥‥将来の並列処理の方向は‥‥に違いない"

ＰＩＭの将来に向けての私の経験？

並列推論マシン　前期の結果から

富士通株式会社　久門耕一

１．はじめに

ICOTの並列推論マシンに関する方針は、前期と中期ではかなり異なってきたと思う。特に、OR並列実行からAND並列実行及び、PrologからGHCへの変更は大きく、その理由は重要な意味を持つ。

そこで、この方針の変化について、私の経験を元にいくつか考えてみたいと思う。

２．並列推論マシンとは－OR並列とは

元々、私はこの並列推論マシンという言葉があまり好きではない。なぜならば、「推論マシン」という言葉の持つ雰囲気が、ちょうど人工知能という言葉と同様に、この分野の専門家が言っている意味と、分野外の人が受ける印象が余りに異なっているからである。

私は、会社内で並列推論マシンのデモンストレーションを行なうたびに、「ここで言う並列推論マシンとは、Prologを並列に実行する並列計算機の事です」という注釈を必ず付ける事にしている。

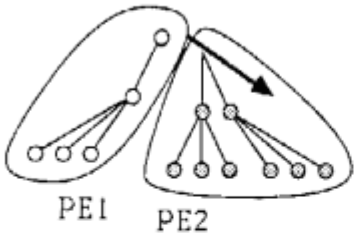
大体、推論という言葉が、三段論法を指すなどという事は不明朗である。

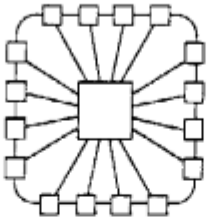
それは、それで置いておくとして、前期で言う並列推論マシンは、三段論法を行なうと言われるPrologを実行するものであったが、なぜ中期に入ってPrologではなく、GHCに変わったのかを考えてみる。

Prologの実行を並列に行なう場合、すぐに思い付くのはOR並列実行である。このOR並列実行の特徴としては、並列に実行するプロセス間の相互作用がない為、制御が簡単であり、プロセス間の通信がいらないという特徴を持つ。

このような特徴は、並列処理においては大変好ましい性質だから、並列推論マシンの処理方式として採用するのは当然であると思われる。

事実、私達が提案した株分け処理方式では、並列処理の指標である「台数効果」が大変に良く、10台で処理して9.9倍早いなどという事をいともたやすく実現した。また、株分け方式では単体性能もかなり良く、現在の実験機上でワレンコードをエミュレートする方式を取っているが 5KLIPSの速度を持っている。この株分け方式の単体性能が良いのは、単体の処理が普通のProlog処理系と全く同じ処理を行なっており、通信のオーバヘッドが小さいから当然である。





株分け実験機の構造

株分け方式の特徴

要求による新プロセス生成

バックトラックによる
環境の再利用

専用ハードによる暇ＰＥ検出

３．何故OR並列がだめか－株分けをもってしても

ここまで、株分け方式のコマーシャルを述べてきたが、何故こんなに良い方式を諦める気になったのか？

それは、一言でいえば遊べないからである。ベンチマークプログラムとして、用いる

プログラムは、皆ツマラナイものだし、第一ゲームのプログラムを書く事がどうも本質的に出来ないようであった。
対局ゲームというものは、人間が次々に行なう指し手に対して、相手（この場合は計算機）が応戦するという事が基本となっている。ところが、OR並列処理は、プロセス間の通信が無いのが特徴だから、人間（これも、一つのプロセス）からの入力が実行中には出来ない。必然的に、バッチ処理的に動かす事になる（どうもバックエンドプロセッサにしかならないなあ）。
これだけでも気が乗らないが、さらに、OR並列とは全解探索が基本となっているので、探索空間がほぼ無限に広い場合には、たとえ株分け16台PEをもってしても、なんの足しにもならない。
さらに悪いことには、OR並列の場合、一般的に探索時間と解の個数が比例しそうであるということである。
最後のとどめとして、大体、OR並列では、フォークしたプロセス間の独立性：－すなわち、複数の解の関係が粗であるので、解Ａと解Ｂとの関係を云々する事が出来ない。
通常の探索問題では、何らかの評価関数があって、その評価が一番高いものが答えであるという形が一般的である。これは、明らかに、複数の解の間の関係について評価を行なえと言っているわけだから、どうすればよいのだろうか？
結局、単純OR並列では、全解探索が高速化されたとしても、後の処理を行なう術がないのである。又、答えを全て集めて処理を行なう為に、Bagof機能を実現したとしても、先ほど述べたように解の数が莫大であり、解を蓄えるのが大変であるうえに、解集合の処理はOR並列には（多分）実行できないから、ボトルネックになるだろう。

４．逐次計算機よりも遅い並列計算機－算法で負ける

逐次計算手法の中には、探索問題で探索範囲を狭める手法が幾つも存在している。私が知っている範囲で、ｱﾙﾌｧ･ﾍﾞｰﾀｶｯﾄと言う手法がある。この手法は、特に対局ゲームなどで、min-max戦略を用いた時に、何手かの先読みを行ない、最も良い評価値を持つものを探す時に用いられる。
この手法は、うまく働くと探索範囲のオーダが、元の範囲の1/2乗に小さくなるのである。
もし、1手毎に10の可能性があり、10手先読みを行なうとすると、探索範囲は100億となる。
これがｱﾙﾌｧ･ﾍﾞｰﾀｶｯﾄにより、10万に狭められたとすると、10万倍の速度向上が望めるため、台数効果などは吹っ飛んでしまう。
並列計算手法とは、並列計算機においてもこのような効果的な手法を開発する事だと思うが、OR並列を基本とすると、どうも相性が悪いような気がする。
では、世の中に全くOR並列が不必要かと言われれば、それは嘘である。全解が欲しい事が存在するのは確かだ。でも、用途が限られるだろう。
株分け方式におけるOR並列探索法は、大変に効率が良い。推論動作には不必要な複写を極力減らし、並列実行を必要とする時のみ環境の複写を行なっている（要求駆動に近い）。
だから、単体速度や台数効果は満足できたのだが、アプリケーションが無い事には、どうしようもない。

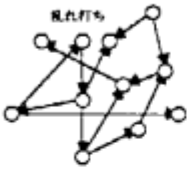
５．何故Prologではなくghcか－答えにならない答え

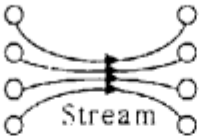
ここら辺は、余り明確な答えは持ち合わせていない。
何故なら、GHCというのは大変に書きにくい言語であり、とても人に向かって「便利ですから是非お使いください」と言えたものではない。
純粋Prologが持っていた論理的健全性などは持ち合わせていないGHCであり、宣言的プ

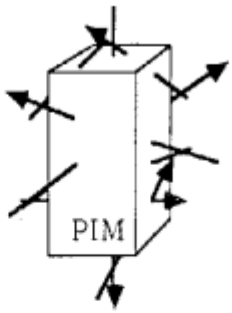
ログラミングなど全く無縁の言語であり、完全に手続き的な言語の癖に、デバッグなどは、宣言的な意味合いをくみ取らないと出来ない言語と言ったら言い過ぎか。
しかし、従来の言語に無い特徴もあるので、使いたいという人に向かって「辞めたほうがいいよ」とも言えないのである。

6．ANDストリーム並列は良い方式かーハードウェアがネを上げる

「何々を行なうのに」という対象のドメインがはっきりしないと答えられないが、何でも出来ますと言うのが優等生的答えで、毒にも薬にもならない。
ただ、OR並列のシミュレーションが出来る事は確かであり、その時に完全複写法によるOR並列実行となるという事だ。（最近、レイヤードストリーム法というのがあって、完全複写を行なわないらしいが、とても説明できるほど知らないし、間違って説明すると恥をかくので止めます）。
これは、株分けの効率の良さにはかなわないが、多くのOR並列実行法と同程度の効率を持つと言える。
ただ、論理型言語の特徴である単一化が、GHCにおいてはかなり制限されているために、未束縛変数を扱う事が難しいというかなり致命的な欠点を持っている。
具体的には、GHCによりPrologのインタプリタを記述しようとすると、GHCが持つ単一化機能を用いる事が出来ず、自力で単一化ルーチンを書かなければならず、GHCは本当に論理型言語だったのか？と疑いたくなる。
単一化の問題で言えば、GHCのインタプリタもGHC自身で記述する事が出来ない。これは悲しい事である。Prologの場合には、インタプリタを実現するのに必要な最小限の機能を追加すれば、兎も角動くインタプリタが作れたのだから。
又、ハードウェアを考えると、プロセス間の通信としてデータのみならず、制御情報などが、怒涛というよりも、ゲリラ戦のごとく全方位から飛んでくるので、とにかく厳しいのも事実である。
だから、ハードウェアでは通信はストリームではなく、散弾だと思うしかない。
だから、こんな言語を効率良く実行出来るハードウェアが実現可能かどうかについては・・・
OR並列の場合は、簡単な問題を簡単な制御機構で簡単にこなすと言えるが、ストリーム並列では、どんな問題でも複雑に全力で取り組まないと他の全てに負けてしまうという気がする。







7．一体推論マシンは何を行なう計算機かー用途が無いのは汎用計算機だ

第五世代計算機計画の大きな目的はハードウェアを作ることではなく、高度な知識情報処理を行なう事だと思っていた。そして、その目的の為には従来の計算機では速度的に問題があるので、並列計算機を用いる必要があり、その手段としてＰＩＭがあるのだろうと思ってきた。
最近、こう思っているのは私の思い過ごしではないかと思う事もある。

高度知識情報処理とは何を指すのかが分からなくなった為である。
現在まで、将来作るであろうＰＩＭ上で走る高度知識情報処理を行なうプログラムの開発を行なっているという話がないのは、一体どう言う事だろう。ハードウェアの開発工数が大きい事は認めるが、知識処理を行なうソフトウェアの開発が簡単であるなどとは思えない。
昔から、並列計算機が実験室の外に出なかったのは、並列計算機を活かすアプリケーションプログラムが書かれなかったからであり、GHCは記述言語に過ぎないのだから、応用を用意しておかなければならない。

８．終わりに

並列推論マシンについて、私なりに勝手に意見を述べてきた。今まで述べてきた事は、OR並列と言っても株分けに依存した部分が多い事は確かであるし、純粋なOR並列のみを実行するマシンを作ろうと考える人はいないだろう。
しかし、色々な機能を付ける事により、本来のOR並列の単純さや効率の良さが失われたのでは、余り意味がない。特に、対局ゲームが出来るかどうかという事は、実際の問題においてもかなり重要な意味を持つと思っている。
並列推論マシンは、作成者が遊べるようなマシンでなければならないというのが私の本心である。

中期ＰＩＭに対する私見

日立中研　　杉江　衛

１．はじめに

通常の議論は、「マクロな方向既にありき」として進められている感が強い。本ワークショプを機会に、我々の中期ＰＩＭ研究のマクロな位置づけを自分なりに整理してみたい。

２．ＰＩＭ研究に対する視点

ＰＩＭ研究では、一言でいえば、知識処理向き並列アーキテクチャの確立を目指しているが、そこには、２つの側面がある。すなわち、

（１）逐次マシンを上回る性能の達成

と

（２）ユーザへの超並列環境の提供

である。次章以降にて、こうした側面から見た場合の、中期ＰＩＭ研究の位置づけについてのべる。

３．性能面

性能の向上が研究の motive force となっている側面である。高度な知識処理を行うには、逐次型計算機では達成できない程の性能が必要であると言われて研究が開始された。しからば、何時、何台規模のプロセシング・エレメントからなる並列計算機によって、逐次型計算機の性能を上回るのであろうか。高度並列システムを想定した場合、高集積デバイスを使用する必要性から、プロセシング・エレメントの単体性能が逐次型計算機に対してハンディを負うことは免れない。10台規模で上回るのか、100台規模か、はたまた、100万台規模なのか？　この臨界台数によって、研究開発で攻めるべきポイントが異なってくると考える。

図１に推論マシンの性能トレンドを示す。ＰＩＭの直線は、［プロセシング・エレメントの性能向上度］×［集積度の向上度］を全体性能の向上度として引いてある。さて、我々のＰＩＭ研究は、どの線に沿って進めようとしているのであろうか？

ＰＩＭ性能は次式で表すことができる。

$$\text{ＰＩＭ性能} = \text{プロセシング・エレメント単体性能} \times \text{台数} \times \text{稼働率} \div [1 + \text{並列化オーバーヘッド}] \quad -(1)$$

稼働率＝稼働率（粒度，負荷分散方式，ネットワーク）

並列化オーバーヘッド＝並列化オーバーヘッド（粒度，負荷分散方式，言語）

いま、プロセシング・エレメント単体性能を、逐次型最高速マシン性能で正規化して

１／１０とし、稼働率を１／２、並列化オーバーヘッドを４とすると、（１）式は次式となる。

ＰＩＭ性能　＝　１／１００　×　台数　　－（２）

（２）式は、図１において、逐次型性能を表す直線とＰＩＭ性能を表す直線が、100台の点で交差することを示しており、ＰＩＭ性能の目標を逐次型マシンの10倍とすると、1000台規模システムによって、初めてＰＩＭの存在価値が生じることになる。コマーシャルベースのマルチプロセッサ・システムが高々数台規模である現状からすると、1000台システムは次々世代のシステムと見なせ、並列化オーバーヘッドよりも、高度並列時の稼働率に焦点を合せたアプローチが重要になる。これが前期のＰＩＭ－Ｒのアプローチであり、上述の仮定を検証した。

（２）式導出の際の仮定において、並列化オーバーヘッドを～０とすると、次式が得られる。

ＰＩＭ性能　＝　１／２０　×　台数　　－（３）

（３）式は、100台規模システムでも、逐次型最高速マシンの５倍の性能が達成できることを示している。これが中期ＰＩＭの現アプローチであり並列オーバーヘッド～０の条件の達成が研究の焦点となっている。並列化オーバーヘッドは粒度の関数であり、一方、稼働率も粒度の関数である。現在、粒度の最適化によって、並列化オーバーヘッド～０と稼働率～１／２の条件の成立を図っているところである。

４．超並列環境

計算機システムを輸送システムと対比させて考察してみよう。図２に両者の関係を示す。逐次型計算機は‘大きな’トラック‘１’台で輸送するシステムに当り、パイプライン型スーパーコンピュータは、固定軌道上を走る高速コンテナ列車システムに当る。また、並列型計算機は‘比較的小さな’トラック‘複数’台で輸送するシステムに当る。

日本において歴史的にながめてみると、輸送システムは、飛脚⇒鉄道輸送⇒大型トラック輸送と形態を変化させてきた。つまり、逐次⇒スーパーコン⇒並列と進んできた感がある。そして、これは、ニーズと使用可能手段との整合の結果と見ることができる。この整合を誤ると、Los Angeles の交通システムのように、大スループットの高速道路網を持ちながらも渋滞を招く結果となる。

輸送システムの例に見るように、対象すなわち応用プログラムの性質と手段との整合が重要である。しかるに、現状では、対象の性質が明らかにされていない。

道路がないところでは世界が閉じているので宅配の発想は生まれない。応用世界の広さを認識し、手段を生かす発想を生ませるための道路造り、これが中期ＰＩＭ試作のもうひとつ位置づけである。この視点からすると、中期ＰＩＭは、後期ＰＩＭに通じるネットワークを具備しているべきであり、１～２階層のネットワークによる100台の結合が望ましいと思われる。クラスタ内の共有メモリ結合は、制約を強めすぎているとは言えまいか。また、64台のMulti-ＰＳＩと比較した場合の存在意義も自分には不明である。

5．まとめ

現在の中期ＰＩＭ研究は100台規模での性能の追求と位置づけるべきと思う。

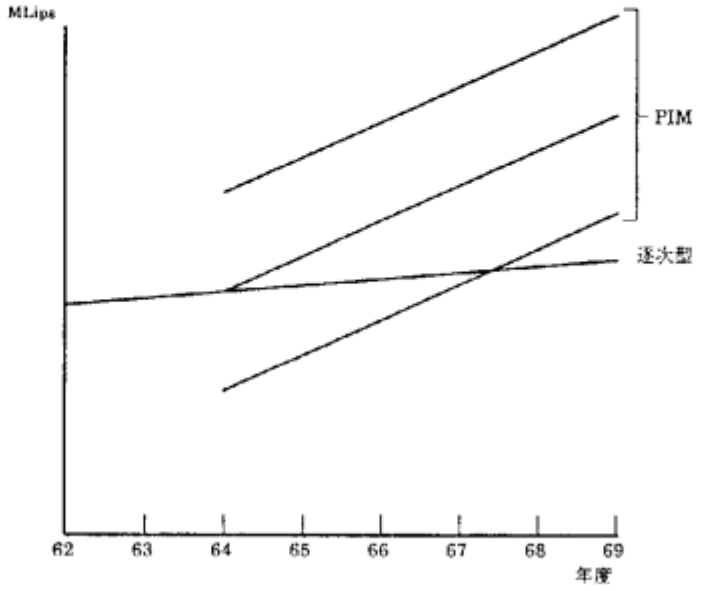


図1. 推論性能トレンド

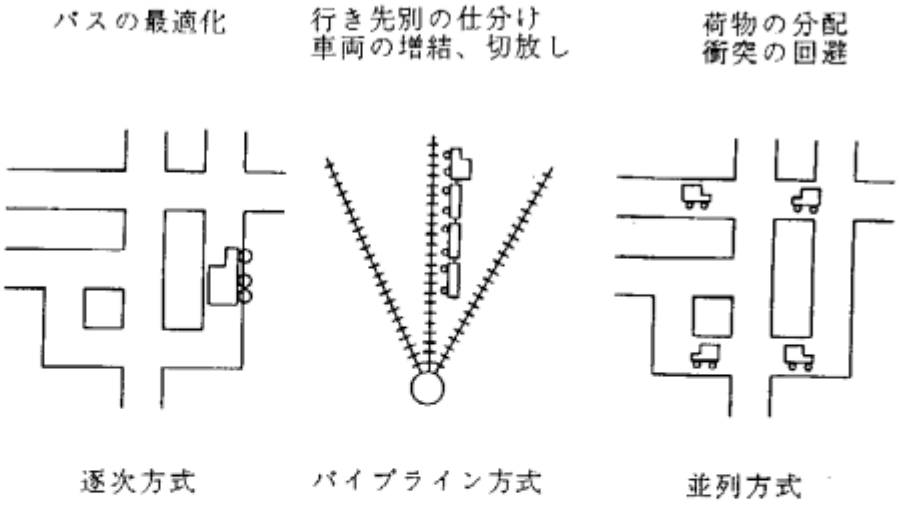


図2. 計算機システム vs. 輸送システム

データフロー・マシンとＬＳＩ技術

伊藤徳義　（沖電気）

１．はじめに

データフローモデルをベースとした並列推論マシンの処理方式を検討し，その実験機PIM-D(Parallel Inference Machine based on the Dataflow model)の試作・評価を行ってきた。試作実験機のハードウェアは，ビットスライスマイクロプロセッサや市販TTLチップを主体に構成されておりその物理的サイズは大きい（例えば，処理要素モジュールのハードウェア規模はA3版大の基板 11枚で構成されている）。このことが基板間やモジュール間の配線負担を大きくしており，結果的にネットワークバンド幅の低下をもたらしている。従って，最終目標である大規模並列推論マシンを構築するためには，その構成要素をＬＳＩ化することによって小型化し，高性能化を計ることは必須である。現在この方針に沿って中期ＰＩＭの処理要素のＬＳＩ化の検討を行っている。

現状のＬＳＩ技術では，ＬＳＩチップに収容可能なゲート数の制限，パケット（トークン）転送のための配線領域が大きくなることによる制約，及び，ＬＳＩチップの入出力ピン数の制限により，ＬＳＩ化データフロー・プロセッサを短期間で開発するのは難しい。本稿ではこのような問題について述べる。

２．データフローマシンの特徴

データフロー・プロセッサはノイマン型プロセッサに比して，以下のような特徴を持つ。

・context switchingに対して＂強い＂

データフロー・プロセッサはcontext switchingが頻繁に発生するようなアプリケーションに対して強い。例えば，GHC版N-queensプログラムのように平均で数reduction位実行した所でcontext switchingを発生するようなアプリケーションはノイマン型プロセッサにとっては＂最悪＂である。即ち，このようなアプリケーションをノイマン型プロセッサで並列に実行するにはオーバヘッドが大きく，必然的にcontext switchingができるだけ発生しないように処理を逐次化する必要がある。これに対して，データフローマシンでは命令毎の独立実行（いわば，命令毎にcontext switchingを行うこと）を保証しているのでこのような問題は特に気にしなくてよい。

・パイプライン制御の容易さ

データフロー・プロセッサはハードウェア的にみれば汎用パイプラインプロセッサであると考えることができる。逐次型（パイプライン）プロセッサでパイプを淀みなく詰めるのは一般に困難である。特に，論理型言語のように実行時タグチェックを必要とするような場合は，分岐制御が頻繁に発生し，これによってしばしばパイプの乱れを生じ実効的な処理性能の低下をもたらす。これに対して，データフロー・プロセッサではこのような配慮は不要であり，並列に実行可能な命令が存在する限りパイプは自然に埋まる。

このように，データフロー・プロセッサは本質的に並列処理を容易に実現できる可能性を持っているが，後述するようにこのようなハードウェアを現状のＬＳＩ技術で実現するにはいくつかの解決すべき問題が残されている。以下，その理由について述べる。

３．データフロー・プロセッサの構成

データフロー・プロセッサを用いた場合の処理要素の一構成例を図１に示す。同図に示すMS(Matching Store)は命令の発火制御を行うための連想検索機能を持つメモリである。このMSは命令のオペランドを格納するメモリであり，発火した命令のオペランドは，CC(Code Cache) からフェッチした命令コードと共に実行可能命令を形成し，PU(Processing Unit)に送られる。 PUは実行可能命令を解釈・実行するユニットである。このPUをマイクロプログラム制御で実現するとすれば，マイクロプログラム格納のため

のCS(Control Store) が必要となる。命令実行の際，必要に応じてメモリアクセスが行われる。DC(Data Cache)はこのメモリアクセスを高速化するためのキャッシュである。PUにおける演算結果は結果パケットとして次の命令に送られる。結果パケットは一時的にPQ(Packet Queue)に格納され，再びMSに転送され次の命令の発火制御を行う。なお，MS，CC，及び DCに関しては，これらがあふれた際に退避する大容量記憶BS(Backing Store) に接続されているものとし，このBSアクセス時間はこれらのキャッシュアクセス時間に比して大きいものとする。

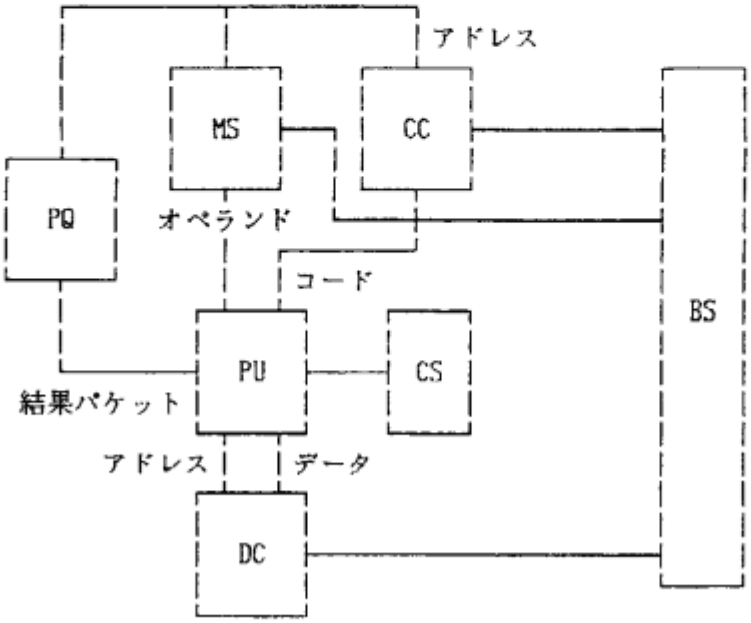


図1．データフロー・プロセッサを用いた処理要素の構成

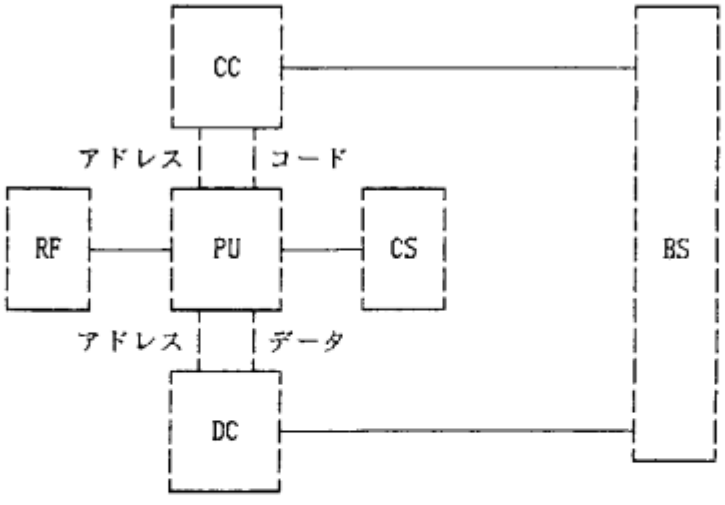


図2．ノイマン型プロセッサを用いた処理要素の構成

この構成をノイマン型プロセッサと比較してみよう。図2はこのようなプロセッサの構成例である。同図に示すように，RF(Register File)，PU(Processing Element)，CS(Control Store)，CC (Code Cache)，及びDC(Data Cache)から構成される。

図１と図２を比較して明らかであるのは，データフローマシンでは，RFの代わりにMS及びPQが余分に必要なことである。逐次型プロセッサでは，命令の間のオペランドの受け渡しはRF内のレジスタを介して行われるのに対し，データフロー・プロセッサではMSを介して行われる。一般に，このMSの記憶容量はRFのそれに比して少なくとも仮想的にははるかに多く必要である。従って，MSは大容量仮想レジスタファイルであるとみなすこともできる。以下，これらのハードウェア構成について簡単に説明する。

(1) MS(Matching Store)

MSは，命令発火機能を持つオペランド格納用メモリである。命令の入力オペランド数がたかだか２であるとすれば，命令発火制御のためにはパートナーオペランドがすでにMSに格納されているか否かを調べる連想検索機能があればよい。このような機能は通常のキャッシュと同様，N-way set-associativeメモリやfull-assiciativeメモリで実現できる。従って，ハードウェア的には従来のキャッシュとそれほど大きな違いがないと思われる。検索は，プロセス識別子と命令アドレスのペアをタグとしてタグメモリをアクセスし，その一致検出結果を判定することによって行われる。一致するオペランドがメモリに存在しない（ミスヒットした）とき，通常のキャッシュと同様BSからのオペランドロードが行われる。

(2) PQ(Packet Queue)

PUで生成される結果パケットを一時的に格納するメモリであり，通常FIFO(First-in First-out)メモリにより実現されよう。各パケットに必要な情報は，プロセス識別子，結果の宛先命令アドレス，及びオペランドであり，所要ビット幅は大きい。従って，このパケット転送を高速化しようとする場合，転送路の負担が大きくなる。

4．ＬＳＩ化における問題点

現状のＬＳＩ技術で最も問題となるのは，以下の３点である。

- 収容ゲート数
- 配線の複雑度
- 入出力ピン数

以下，これらの問題点について述べる。

(1) 収容ゲート数

現在利用可能な汎用ゲートアレイやスタンダードセル等のＬＳＩでは，数十Ｋゲート位の収容ゲート規模が限度である。この規模で上記のようなプロセッサを短期間で開発する場合，CS(Control Storage)の不要なRISC(Reduced Instruction Set Computer)のようなプロセッサを１チップ中に収容するのが限度であり，キャッシュ容量が大きいとキャッシュもこのチップの外付けとなる。さらに収容ゲート数を拡大しようとすればある程度のカスタム化は避けられない。従って，図１のような処理要素を１チップに収容するとなるとかなりの開発工数と期間を必要とすると考えられる。

(2) 配線の複雑度

一般に，ＬＳＩチップでは配線領域はチップ面積のうちで大きな比重を占めており，比較的単純なバス構成のアーキテクチャであってもしばしば配線領域の面積は素子の占有する面積に比して大きくなる。前述のように，データフロー・プロセッサでは通常のメモリアクセスバスの他にパケット転送のバスも必要であり，前述の収容ゲート数による制約よりもこの配線のためにチップサイズが許容限度よりも大きくなる可能性が大きい。この問題は，例えばＬＳＩ内部の３金属配線層以上のプロセス技術やサブミクロン技術の具体化で改善されようが，現状では大きな問題となる。

(3) 入出力ピン数

上記のように，データフロー・プロセッサの処理要素を１チップ化するのは現状では難しい。従って，複数のＬＳＩチップを用いて処理要素を構成せざるを得ない。このとき，問題となるのはチップの入出力ピン数である。現状ではゲートアレイのようなＬＳ

Ｉでは，実装可能な最大ピン数は２００程度であり，図１に示したような機能ブロック単位でＬＳＩ化を行うと，パケット転送やキャッシュアクセスのためのパスをＬＳＩ外部との間で接続する必要があり，これらのパスをそのまま入出力ピンに割り当てると容易にこの制限を越えてしまう。従って，ＬＳＩ外部と接続する必要のあるパスを多重化してＬＳＩ入出力ピンに割り付けることになり，入出力ピンによる伝搬遅延の増加や多重化することによる性能の低下は避けられない。

５．おわりに

データフロー・プロセッサをＬＳＩ化する場合のハードウェア的な制約について述べた。現在，このような制約を踏まえた上でデータフロー・プロセッサをＬＳＩ化する方式を検討中である。基本的な考え方は，入出力ピンの制約を満足し，しかもパケット転送パスのバンド幅が低下しないチップ分割法を提供することにある。この検討結果については別の機会で改めて述べることにする。

並列処理アーキテクチャを考えるとき気になる二つの問題

ＮＴＴ基礎研究所　　雨宮 真人

１．まえがき

並列マシン開発のアプローチは大きく２通りが考えられる。一つは専用処理マシンであり、もう一つは汎用処理を指向するものである。専用処理とは、アルゴリズムが明確で、しかも単純な処理の規則的な繰り返しで処理できるような問題に適用を限定しその適用範囲内で並列処理を最大に行おうとするものである。すなわち単能の高性能エンジン開発である。これは、今のハードウェア技術をもってすればそれほど困難なことではないだろう。むしろポイントはそのような単能の専用マシンがどれだけ市場の広がりを持ちペイ出来るかという商品化戦略にあるといえる。

我々にとって興味があるのは、どれだけ汎用的な並列処理が可能かという点であり、汎用の並列処理技術の開発である。世の中の現象は全て本来並列的であり、分散的である。つまり個々のコンポーネントがある制約条件の下である法則に沿って協調的に動いている。なぜ、コンピュータだけが逐次的な動作という枠に閉じ込められなければならないのだろうか。コンピュータの適用領域を一層拡大させて行くためには、コンピュータをこの逐次的動作の枠から解放しなければならない。現象の素直な記述のためには、逐次処理の"かせ"をはずして、並列、分散、協調という観点から処理を記述することが必要であり、これを効率的に実行できるコンピュータの開発が不可欠である。これが汎用の並列処理マシンを研究する由縁である。

ＰＩＭの研究も汎用の並列処理マシン研究の一環であると考える。特に、リストやストリームのように生成・消滅を繰り返し、その構造が動的に変化するデータを対象とする場合の並列処理構造を明らかにすることが研究の目的であると考える。したがってこれは、画像処理や信号処理といったような処理対象に依存した専用向きアーキテクチャの研究ではなく、より一般的（generic）な観点からの並列処理構造の抽出とアーキテクチャに関する研究である。

ここでは、ＰＩＭに限らずリストのような動的に変化する構造データを処理の対象とし、処理記述は関数型言語あるいは論理型言語を記述のベースとするソフトウェアパラダイムおよびマシンアーキテクチャを前提として考えることにする。

さて、前置きが長くなったが、本稿ではこのようなgenericな並列処理マシンのアーキテクチャを具体化する上で気になる問題について考えてみる。

２．集中か、分散か？

アーキテクチャ上の気になる問題とは、①動的に生成・消滅を繰り返す多数のプロセスをどう管理するか、②動的に変化する構造データをどう管理し処理するか、という問題である。

①の問題は粒度の低い（fine grain）多数のプロセスの並行処理をどう実現するかという問題である。関数型プログラムにしろ論理型プログラムにしろ、リスト処理を記述し、

そこに eager evaluation や lazy evaluation のメカニズムを実現しようとすれば関数あるいは節がプロセスとなり、小さな粒のプロセスが多数生成されることになる。この問題をどう解決するかである。コンパイラ技術によって幾つかの小さなプロセスをひとかたまりの大きなプロセスのまとめ上げることができるか、それとも超多重処理、すなわち10^3個程度のプロセスを並行動作させることができるようなプロセッサを開発してこの問題に対処するか？。

②の問題は多数プロセス間で交信され、処理される構造データを実際にはどの様に管理したら良いかという問題である。つまり、現実には多数のプロセッサの間で複雑な構造のデータがやりとりされることになるが、これらのデータのコピーを作って授受させるのか、共有メモリを用いてコピーを避けるかという問題である。

また、リストやストリームのような構造データの処理では構造データの生成・分解、ゴミ管理など高度なメモリオペレーションが必要となり、メモリアクセスの速度が性能に大きく響いてくる。いわゆるメモリアクセス遅延の問題が深刻となる。この問題をどう解決するかである。データのコピーを作るか、データ共有の方式をとるか？。ローカルメモリ方式でいくか、共有メモリ重視でいくか？。

①も②も、基本的には一つの問題に帰着される。それは、並列処理マシンのアーキテクチャを考える際にローカリティの問題を重視するか、それとも徹底して分散の概念を重視するかという問題である。①の多重処理の問題は、ダイナミックに動作する多数のプロセス間にデータ依存関係のようなある種の関係が抽出できればそれらのプロセスの処理をまとめて集中的に一気に実行させてしまうことができる。たとえば、Recursive な 処理を Tail Recursion に変換しさらにループ実行に置き換えるなどはその例であろう。（例えば Prolog の append が Tail Recursive な構造を持っていることに着目してループ実行させ高速化を図っていること。Accumulating Parameter 方式によって Tail recursion に変換すること。実は append のように最後の処理が cons のような場合には Accumulating Parameter 法によってループに置き換えることができる。これは Prolog における append と同じ原理である。） しかし、 eager/lazy evaluation のことを考えると一般にはそう簡単にはループ化できないだろう。

②のデータコピーか共有かの問題は、あるデータが何回も続けてアクセスされるかどうかにかかっている。いくつかの並列処理プロセスが一つのデータを共有するとき①の場合と同じようにプロセス間の関係が抽出できそれらの処理を集中的に行うことが出来ればデータアクセスのローカリティが得られるが、そうでなければむしろ積極的に分散化させてしまい共有メモリ方式の徹底を図ったほうが良いことになる。

本稿では、コンパイラによって大粒のプロセスを作り出すことが極めて困難であるとの認識で、高度並列処理の実現には、ローカリティを前提にしたアーキテクチャよりも、分散化を前提にしたアーキテクチャの方が有利であることを定性的に示す。

（出来ればコンパイラで大きなプロセスを抽出し、ローカリティ重視のノイマン方式プロセッサで実行できればそれに越したことはないが、それは無理であろう。一方、すべてハードウェアにおんぶというのも芸のない話である。現実的な解としては、コンパイラによる静的解析とハードウェアによるダイナミックな多重処理管理との旨い接点を見つけ出すことにあると思われるが、それにしても今のアーキテクチャとはかなり異なったアーキテクチャが望まれる。）

3．プロセッサとメモリの粗結合

ノイマン・ボトルネックの問題はプロセッサとメモリとの間に必要以上のデータが往復し、しかもハードウェアはこの頻繁なデータの往復に耐えるだけ十分密に結合されていないという点にあった。この問題に対してはこれまで、ハードウェア・アーキテクチャの観点から解決が図られ、プロセッサとメモリを接近させるという方法がとられた。階層化メモリがそれである。つまり、プログラムの持つローカリティを利用して頻繁に往復するデータはプロセッサと密に結合したメモリに置くということである（キャッシュメモリがその典型である）。逐次型プログラムではこのローカリティ抽出がうまく行き、キャッシュの効果も活かされた。しかし、さきに述べたような並列処理ソフトウェアではこの特性が活かせるだろうか。むしろ、プロセッサとメモリの結合を粗にしてしまい、プロセッサとメモリを互いに独立動作させる方が得ではないか。両者の間はパケットによってデータの授受を行わせる。両者の間には必要最小限のデータが往復する。メモリはかなり高いレベルの演算機能（たとえば、リスト演算、ユニフィケーション機能、共有変数によるプロセス間の同期制御など）を持ち、プロセッサはメモリ演算を動作させるコマンドのスケジュールとその発行、他のプロセッサとの通信、入出力などを行う。これはパケット通信を基礎とするアーキテクチャである。

両者はパケット通信によって非同期に動作する。したがって、問題に十分な並列処理性があり系内に十分なパケットが存在しているかぎり両者とも十分高い稼働率が得られる。プロセッサ、メモリともに多数のハードウェア装置を用いマルチ構成にしたとしても十分な並列動作が可能となる。

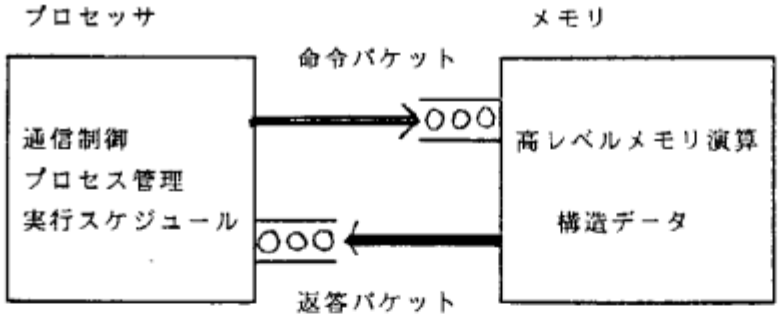


図1　プロセッサとメモリを粗結合したシステムの概念

4．プロセッサ内超多重処理

もし、コンパイラで十分大きな粒度のプロセスを作り出すことができないとしたら、並行に動作する多数の小さなプロセスの管理を実行時に効率良く行わなければならなくなる。個々のプロセスは少し走っては他のプロセスに要求を出す。そしてその結果が返ってくるまではその先の実行が出来なくなってサスペンド状態となる。その間プロセッサは他の実

行可能なプロセスを即座にとりあげ実行させなければならない。関数型プログラムや論理型プログラムではひとつのプロセスが走れるステップ数（実行される命令数）は１００以下であると思われる。つまり逐次実行マシンでは１００命令実行毎にプロセス切替が生ずることになる。

この観察が正しいとすれば、プロセス切り替えの度に生ずるレジスタの退避・回復、キャッシュデータの入れ換えに要する時間はプロセス本体の実行時間を越えることにもなりかねない。レジスタセットを数個設けてもあまり効果はないであろう。また、ローカリティを旨く抽出してキャッシュの入れ替えを極力抑えたりするのも困難である。

この問題に対する解決策も、ローカリティを思い切って諦め分散処理を徹底させることである。ローカリティがないのだから、各プロセスは皆一様に走っている。したがって、ノイマン・アーキテクチャで重要な役割を果たした作り付けのレジスタなるものは意味がなくなる。代わって、各プロセスに平等にレジスタを割り付ける。つまり、メモリをプロセッサ内で同時実行を許すプロセス数分だけブロックわけして与え、これをプロセスのワークメモリとして用いる（最近ではスタティックメモリでけっこう早いものがあるから例えばこれを用いる）。このようにすれば当然作り付けのレジスタに比べアクセス速度はかなり低下する。しかし、この速度低下分はプロセッサ内のパイプライン実行によって相殺できる。パイプライン化の効果が十分あがり、実質の処理時間が、メモリアクセス時間に等しくなれば、超多重処理によるオバーヘッドは殆ど無視でき、高い性能が得られることになる。このような方式は実にデータフロー・アーキテクチャに他ならないのである。

高速レジスタ集中方式による多重処理（モデルＩ）と低速一様分散レジスタ方式による多重処理（モデルII）の間の簡単な性能比較を行ったのでそれを図２に示す。図によれば、モデルＩの高速レジスタ数を３２とし、高速レジスタと低速レジスタ（メモリ）のアクセス速度比を１０とすると、１プロセスの平均命令実行数が１００以下ではモデルIIの方が有利であることが分かる。

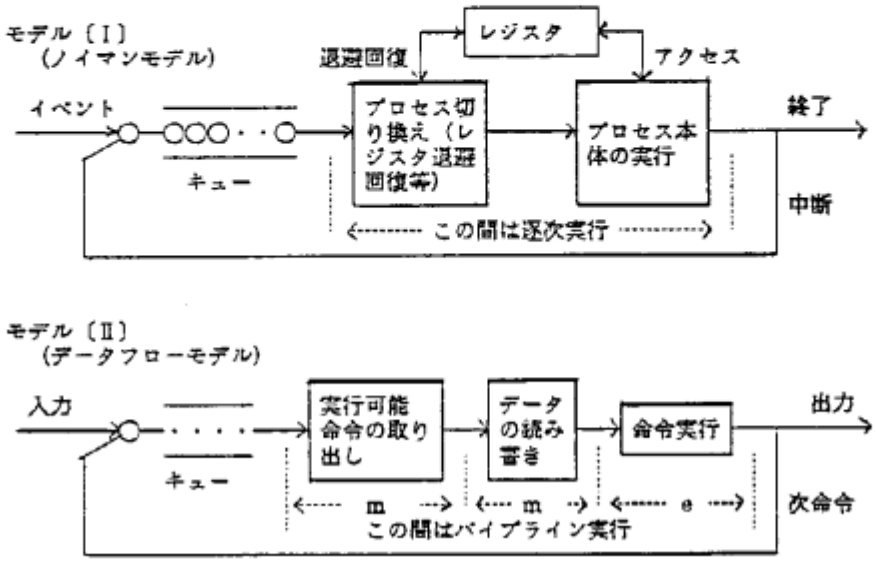
５．あとがき

汎用の並列処理マシンを考える際に日頃気になっている問題を述べた。そして、これまで一般に考えられてきた高速化の手法とは逆の発想が必要であることを述べた。これまでは、ローカリティ抽出が可能な場合に如何に高速なマシンを実現するかという観点で、メモリシステムやプロセッサ構成が考えられてきた。しかし、もし並列処理ソフトウェアに対して十分なローカリティが抽出できず、コンパイル段階で粒度の高いプロセスを抽出することが出来ないとすれば、むしろローカリティを前提としたアーキテクチャを考えるのでなく、ローカリティの逆すなわち分散化を徹底させるアーキテクチャを考えるべきだということを主張した。そして、プロセッサとメモリの粗結合、従来の高速レジスタ集中方式に代わって低速一様分散レジスタ用いる方式の方が有利であることを定性的に示した。

勿論、ここでの議論は非常に単純化した原理的な話であるから現実にどうなるかは方式の具体化によって定量評価してみなければ分からない。

やはり最後まで気になるのは、プログラム変換やデータ依存解析などによってコンパイラでローカリティを抽出し、大きなプロセスを作り出すことが可能なのかという問題である。このことが可能ならばそのほうが良いに決まっているからである。

モデル	方　　式	特　　徴
モデル〔Ⅰ〕	・プロセス実行単位　：大 ・プロセス実行　　：逐次制御 ・プロセス切り換え　：実行環境のスイッチ	マルチプロセス管理（ノイマン・プロセッサでのプロセス管理）
モデル〔Ⅱ〕	・プロセス実行単位　：小 ・多プロセス混合実行：データ駆動型制御 ・プロセス切り換え　：命令レベル	プロセス切り換えのオーバヘッドをパイプライン実行で吸収



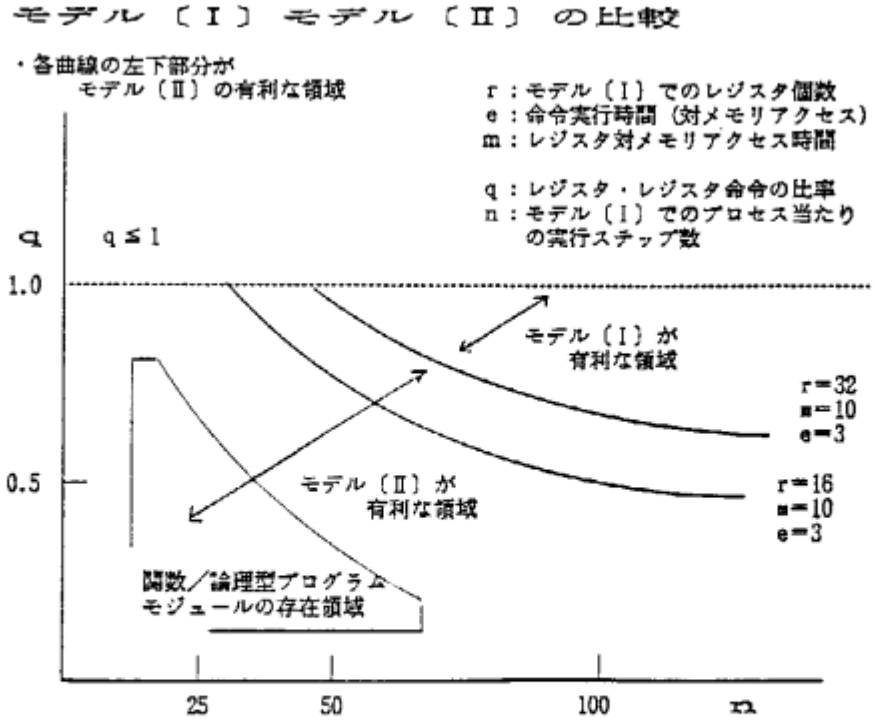


図 2　多重処理モデルの比較

中期プロジェクト

ＰＩＭの中期計画を作るに当たって考えたこと

内田　俊一　（ＩＣＯＴ）

１．はじめに

第５世代プロジェクトは，前期，中期，後期の３期に分かれており，研究目標が未踏分野であることから，各期の具体的計画はそれ以前の期の進捗を考慮して各期の初めに立案することになっている。中期のＰＩＭの計画は，前期の終りから作り始めた。計画を作るにあたっては，当然私の個人的な思い入れ（思い込み？）が反映しているが，これまではその思い入れを表面に出して書くことはなかった。フォーマルな文書ばかり書いていたので書くチャンスもなかったようである。そこで，ここではその辺を少し含めて書かせてもらうこととした。

ＩＣＯＴにおいて，ＰＩＭの研究開発の実質的な責任を負うようになったのは中期になってからである。大学やＥＴＬでＦＦＴプロセッサや図形処理用のマルチプロセッサの研究開発に携わってきたこともあり，そこでの反省がいろいろな形で計画作りに反映している。また，前期にＰＳＩ，ＣＨＩ およびＳＩＭＰＯＳの開発を担当し，そこでの経験もＰＩＭの計画作りに大きな影響を与えている。そこで少々さかのぼって話を始めさせてもらうこととする。

２．　前期の初めの頃，何を考えていたか

まず，ＩＣＯＴに来る以前には，次のようなことを考えていた。（「第５世代プロジェクトは，国家プロジェクトとしてやるのだし，予算も人も十分あるだろうから，人類の共有財産になるようなことをやれ。－（淵）」というのを真に受けて．．．．）

１）コンピュータ・アーキテクチァの研究は蓄積が重要。論文，仕様書，ソフトウェアやハードウェアのモジュールなど，ともかく他人がブラックボックス的に利用できる形でのこす。

２）中途半端にならないような研究目標の設定と，それをやれる体制をつくってやる。

３）開発ツールを十分に凝ってつくる。作る対象が不明確な分だけ環境をしっかり作る。

４）研究として面白くないものはやらない。（面白くなるように進める。）

（上の方針の始めの３つは，自分のやってきたことが，それまではそうでなかったことを示している。．．．と読む。）

並列マシンに関しては，次のようなことを考えていた（のだろう）と思われる。（ＩＣ

ＯＴに来てから変った（強化された）かもしれない。）

ａ）並列マシンにもＯＳを初めとする管理用のソフトウェアや言語処理系が必要。

－よくＤＦＭの論文等の硬い文句にでてきた「マシンがプログラムの中の並列性を半ば自動的に抽出し実行する。」といった表現はおかしい。この表現には，コンパイラやＯＳがあたかも不要であるかのような間違った印象をあたえる。

ｂ）並列ソフトウェアの話をきちんとやらないと並列ハードウェアの研究は進まない。

－ソフトウェアとハードウェアの何かうまい言語的切り口を見つけることが不可欠。言語はその意味を与えるフォーマルなモデルを持っていること。

－切り口となる言語は，プロセス間の通信制御や資源の割当て処理を記述するプリミティブを持っていること。（ＯＳが書けるように）

＊ 関数型言語とＤＦアーキテクチャは一つの理想型。
（ＧＨＣも気にいっている。）

ｃ）ハードウェアは所詮単純な処理をひたすら高速に実行する以上のことはできないのだという確信。（ソフトウェアとハードウェアの役割分担が設計のポイント）

－並列処理可能なタスクをバランスよく実行するのをハードウェア／ファームウェアだけでやるのは，たとえ図形処理のような対象が簡単な構造のものでも大変。

ｄ）並列マシンは，ある程度以上の規模のものを作らないと意味がない。とくに規模の大きなソフトウェアと組み合わせた実験ができるものをやろう。

－ＶＬＳＩデバイスとＣＡＤも進歩するからＤＦアーキテクチャも実装可能となるだろう。（この辺の読みは甘かった。）

３．そして前期の終りには...

前期は，ＰＳＩ，ＣＨＩ およびＳＩＭＰＯＳの研究開発をやったが，研究らしいことができたのは，ＰＳＩの設計のはじめの方だけで，あとは研究の方向づけ，研究開発環境と体制作り，予算獲得作業，その他，諸々の雑事。

しかし，非常に多くのことを勉強させてもらった。とくに言語処理系やＯＳ作りについては，経験がなかっただけに新鮮で面白かった。（ＳＩＭＰＯＳはマルチウィンドウに一文字でるのに３秒かかる時代もあったり，ＦＧＣＳ’８４のデモにまにあうかとやきもきしたり，十二分にスリルも味わったが．．．）
おかげで，並列ソフトウェアの研究開発をどう進めるかの具体的アイデアが固まった。

また，前期のＰＩＭの研究状況を見させてもらい，技術的にも，研究開発体制的にもなかなか大変そうだとの認識を深めた。（純技術的問題に行き着くまでの問題がこれほど大変とは思わなかった。）

前期のうちには，いろいろな研究開発成果の蓄積があったが，それらのうちの主なものを拾ってみるとつぎのようになろう。

－Ｐｕｒｅ－Ｐｒｏｌｏｇをベースとする並列実行方式のアーキテクチァ的研究蓄積
－ＰＩＭ－Ｄの試作における実行方式の評価と実装における経験
－ＧＨＣを生み出した並列論理型言語の研究と処理系の実装方式
－ＰＳＩやＣＨＩの研究開発によるハードウェア構成方式と実行方式
－ＷＡＭの出現とコンパイラによる最適化の効果
－ＥＳＰの開発とそれによるソフトウェア生産性の高さ
－ＳＩＭＰＯＳの研究開発による大規模ソフトウェアの開発経験
－ＰＳＩリリースによる多くの応用プログラムの試作と処理能力の要求

これらの蓄積を総動員して，中期のＰＩＭの計画をたてることとしたが，そのときの基本とした考えとしては，次のようなものがあった。

1）ＧＨＣをベースとする核言語第１版をＰＩＭの言語とする。ＯＳの制御も記述できなによりもシンプルなのがハードウェア的には魅力。

2）ＰＩＭ用の並列ＯＳ（ＰＩＭＯＳ）を本格的につくる。マルチＰＳＩを作って，並列ソフトウェアを本格的に作る環境とする。その接続は並列ソフトウェアとアーキテクチァのマッピングが考えやすいものを選ぶ。

3）ＰＩＭとしては，まずコンパイラによる最適化技術も取り入れた十分な処理能力をもつＰＥを考える。　ＰＥはソフトウェアとの整合性を重視して逐次型でいき，ＧＨＣのためのハードウェア・サポートに工夫を凝らす。

4）接続方式としては，ＰＥ間の通信の遅れが，ソフトウェアのオーバヘッドも含め最小になるような方式をとる。（クラスタと並列キャッシュ）

5）並列処理の本格的なアプリケーションをその専門家がやり，メリットがあるような環境を作り出す。これにより，並列アルゴリズム等の研究が始まるようにする。

この方針では，すでに並列ソフトウェア重視の選択をしている。これと異なる選択としては，アーキテクチァ的に面白い並列マシンをやるという方法がある。（例えば，意味記憶マシンの類。）ＤＦＭなども，もっと追及してみたい希望があった。いろいろ考えたが，やはり，ＯＳなしの専用マシンはこのプロジェクトの大きな枠組みを生かすことにならないと思いやめた。

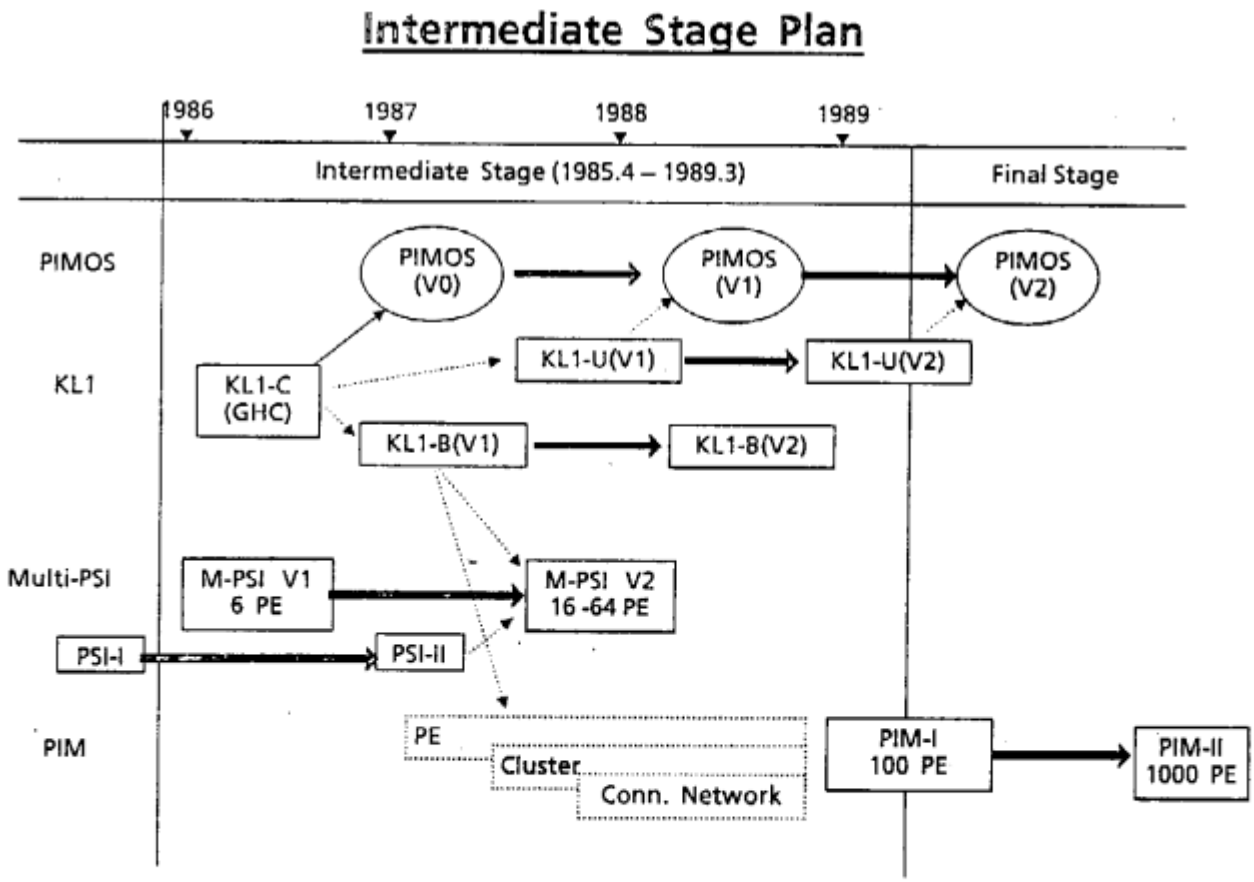
それに，ＰＩＭに限らず，並列マシンの研究開発は一般にアーキテクチァ優先でやられてきた経緯があり，並列言語や並列ソフトウェアを実際に作る研究についてはあまり蓄積がない。この問題は規模の大きな組織的研究を行わない限り，効果的な実行は不可能であり，アーキテクチァとソフトウェアの研究者を同時に動員する必要がある。こちらの方がこのプロジェクトの主旨にあうし，世の中に対する貢献度も大きい。という訳で，このような選択をした。（．．．と後から整理してみた。）

このほか，ＣＡＤを縦横無尽に使って多種多様なＬＳＩを作りこれによりいろいろ面白いアーキテクチァを実装してみるというのは，かなり夢みたいな話であるということがわかったことも，大きく影響している。 どうも推論マシンのもつ機構はマイコンに比べやはりハードウェア量が多いということも効いているようである。

４．おわりに

この後の話は，昭和６０年９月からスタートしたＰＩＭ／ＭＳＩＭ－ＷＧで報告しているとうりである。 現状については，いろいろ苦労も多いが，そして，関係する皆さんにも苦労させているが，かなりうまくやれているのではないかと思っている。（最近だいぶ気が長くなった。）
アーキテクチァの新しさという面では少々ゆずるものの，並列ソフトウェアとの一体化という点では他の追随を許していないと自負している。（これも皆さんのおかげです。今後もずうーっとそうでありたい。）
並列ソフトウェアの挙動が，ＰＩＭＯＳや応用ソフトウェアができてはっきりしてきたら，そして，其のときにはＣＡＤもＶＬＳＩもずっとよくなっているから，また極め付けに斬新なアーキテクチァをもつＰＩＭをやってみたいところである。

最後に，中期計画の縁表をのせておきます。



並列処理文化づくりに向けて

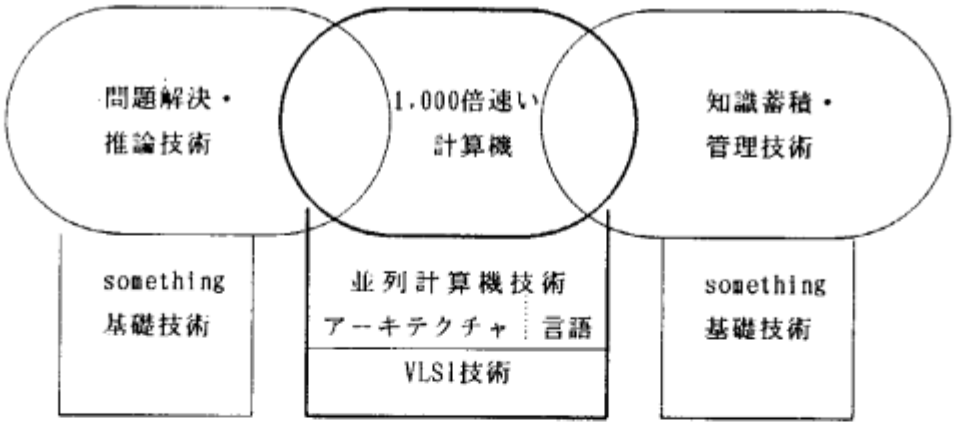
瀧　和男
ICOT　第４研究室
1987. 7.17

１．流　れ

初期の構図　　プロジェクトの始まりの頃は、こんな構図を描いていたように思う。

我々は、高度な知識情報処理システムを作る。膨大な知識と高度な推論能力を備え、自然言語で人と対話し、プログラミングというめんどうな手続きなしに、多くの問題を解決してくれる。

このようなシステムは、その一部を実現しようとするにも、当時の大型計算機の 1,000 倍以上の計算能力を必要とするだろうといわれた。一部には、 100万倍という声もあったようにも思う。 1,000倍の性能を持つ計算機と、問題解決・推論の技術、知識蓄積・管理の技術があれば、目指す高度な知識情報処理システムは、その一部分にしろ実現できるだろうと期待されたのであった（図１）。



注：知的インタフェースは省略

図１　高度な知識情報処理システム実現の為の
初期の構図

並列計算機研究から並列ソフトウェア研究へ　そこで、計算機アーキテクチャ屋ないし計算機システム屋は、1,000倍の能力を持つ計算機を作れば、第五世代コンピュータ技術の一角は固まるのだと信じて、勇んで研究に着手したのであった。

逐次実行型計算機の性能向上率は、10年で約10倍である。10年のプロジェクトで1,000倍の性能を実現するには、並列計算機しかないというのが大方の意見の一致するところであった。さあ、みんなで知識処理向きの並列計算機を作ろうではないか。

多くのアーキテクチャ屋がいろんなところでいろんなアーキテクチャを提案し、その中からいくつもが試作され、・・・・・そして展示室に静態保存されていった。

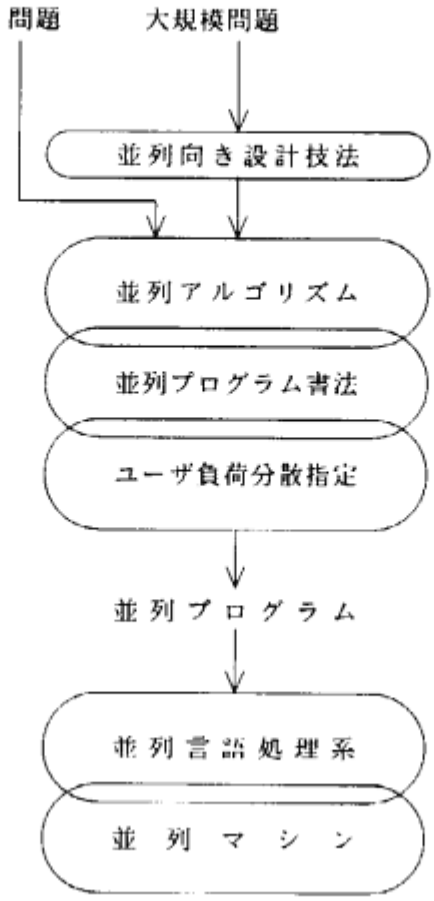
すべてに共通することは、それなりの応用プログラムが作られて利用されるところまで、辿り就けなかったことである。さる指導者の言葉によれば、「ソフトウェアレスのマシンは生き残れない」のである。すなわち、言語、処理系、プログラム開発環境まで含めて、手堅く作られたシステムでない限り、十分な評価はできず、また子孫を栄えさせるだけのエネルギーを生み出せないのであろう。

ICOTでは、この考えに基づいて、知識処理向き並列計算機のための、ソフトウェアサイドの固めに着手した。マルチPSI 、PIMOS をとりまく一連のアクティビティがそれである。その中で、並列処理言語KL1 、KL1 言語処理系、オペレーティング・システムPIMOS 、プログラム開発環境の準備が着々と進められている。

並列処理はNew Culture である　マルチPSI にかかわる仕事の中で、先頭を切って製作されたのがマルチPSI 第1版とそのシミュレータ、pseudoマルチPSI である。主に第1版の評価の目的から、約10種の問題が取上げられ、約20個の異なるKL1 プログラムとして実現された。第1版の評価作業はまだ続いているが、その中であらためて確認された重要な問題点がある。それは、今更という声もあるが、「並列マシンを満足のゆくように並列に動かすことは、極めて難しい」ということであった。

マシンのレベルで、良く設計された並列言語や負荷分散のための基本メカニズムがサポートされていることが基本的な要件ではあるが、それに加えて図2に示すようなソフトウェアに関する要件が存在する。まずアルゴリズムに並列性がなければ並列実行は出来ない。とはいえ並列性が高くても計算量が不必要に多いものは使えない。また問題をプログラム化する際のプログラム書法或いはプログラミングスタイルの例が十分に蓄積されていない。書き易さや効率の観点からの整理も勿論されていない。さらに並列実行の効率向上のために負荷の均一分散、通信の局所性の維持、部分問題間のスケジューリングなどに関するユーザ指定を追加することが必要であるが、期待通りには動いてくれない。またより大規模な問題では、並列実行向きのシステム設計技法なども重要となりそうである。

このように考えると、並列実行を行なわせるには、マシンが並列マシンであること以外に、言語もアルゴリズムもプログラム書法も、ことごとく並列実行向きであることが必要とされるように思われ、そこに逐次実行マシンの世界で育まれてきた諸技術がどのくらい



問題を並列プログラムとして記述し並列実行する過程で、逐次実行マシンの世界で培われた文化が役立つ曲面は極めて少ない。

図２　問題を並列プログラム化し実行する過程

適用できるか大いに疑問なところである。すなわち、並列処理の分野、とりわけソフトウェアの分野は、計算機の世界の中で、ほとんど未開の地であり、今後開拓されるにしても逐次マシンの世界で育ってきた文化とはずいぶん異なる計算機文化、すなわち並列処理文化が展開されるような分野というふうに考えられるのである。

２つの計算機文化を起こす構図　我々は、第五世代コンピュータの文化、すなわち高度な知識情報処理文化を形成するために、その要素技術の１つとして 1,000倍速い計算機の研究に着手した。その当初は、図１に示したように、並列マシンアーキテクチャと並列言語あたりが中心技術であるかに見えた。しかしながら、いくつかの試作マシン作りを経て、並列プログラムの開発環境や並列マシン用OSの必要性を認識するに至り、マルチPSI

を題材としてそれらの研究開発に着手した。ところがマルチPSI で並列プログラミングを始めてみると、並列処理の問題点の中心は、むしろ並列言語、並列アルゴリズム、並列プログラミング手法、負荷分散やスケジューリング指定といったソフトウェアサイドに多く存在し、これらは計算機分野の中でほとんど未開拓の地であることが分かってきた。すなわち並列処理をまともに行なうためには、逐次計算のための諸技術つまり逐次処理文化とは、ずいぶん異質な並列処理のための文化、とりわけ並列ソフトウェア文化を形成しなければならないらしいことが明らかとなってきたのである。

かくして我々のプロジェクトは、図3に示すように高度知識情報処理文化を形成するために、それを支える並列処理文化をまず打ち立てねばならないという重責を負ったのである。この構図はなかなかに容易ならざる構図に違いなかろう。プロジェクトの目的を達成するためには、視点の異なる、２つの新しい計算機文化を形成しなければならないのである。

この構図のはなしを、H社のさる管理職に聞かせたところ、たちどころに「難しそうだね」という言葉が返ってきたわけで、単純明解を旨とするプロジェクトのストーリー展開上も、この重層構造の研究開発構図はあまりうれしいものではない（いかにも実現が困難そうに見えてしまう）。

そういってみてもなお、高度な知識情報処理システムを実現するために、並列処理文化の育成が不可欠であることは、疑いようのない現実のように思われる。

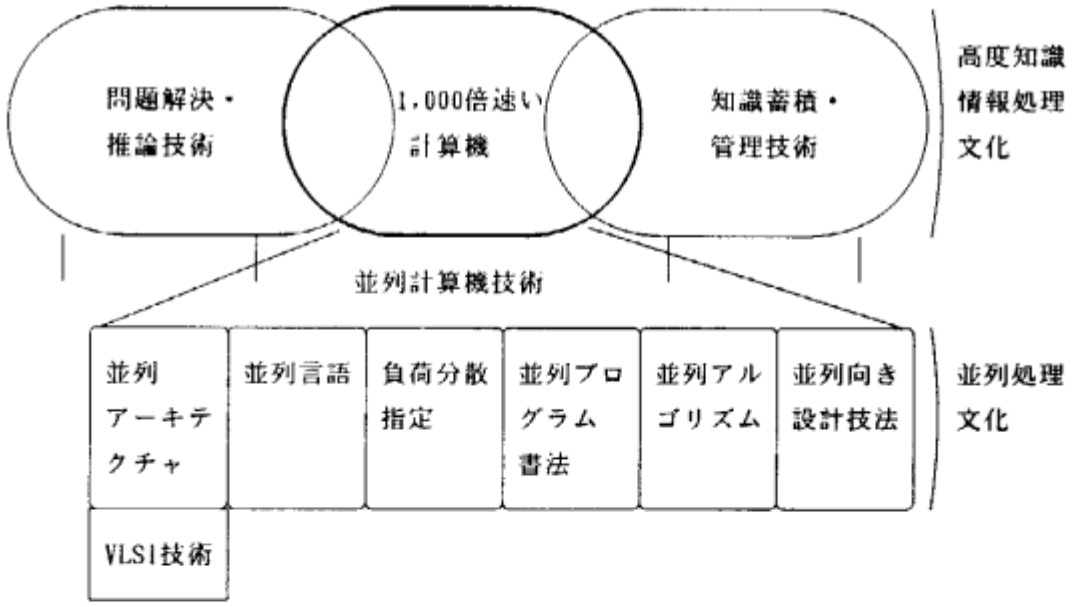


図３　高度知識情報処理文化形成の為の
最近の構図

2. 標　的

やりたいこと　　我々のやりたいことは、

「大規模MIMDマシンによる高度な知識処理」

といってよかろう。しかしながら、直接そこに至ることは容易ではない。なぜならば、図3を用いて言うと、上層の知識処理にかかわる諸技術も未成熟であれば、下層の並列処理そのものにかかわる諸技術も同様に未成熟だからである。直接的アプローチが難しいのであれば、目前の標的として何を考えるべきだろうか。

ベースは並列記号処理　　前節で述べたように、高度知識情報処理文化の形成を支えるものとして並列処理の技術が不可欠なものとすると、我々が今なすべきことは大規模MIMDマシンに関する並列処理技術を積み上げることであろう。並列処理といっても数値計算中心のものではなく、将来の高度な知識処理のベースとなる並列記号処理の技術を蓄積することである。

文化の幅と厚み　　図3に示したように、高度知識情報処理文化の重量を支えてゆくためには、並列記号処理の文化が、十分な幅と厚みを持つまでに育ってゆかねばらない。厚みを増すには、多くの研究者により、十分に幅の広い研究が続けられねばならず、その方法論が問題である。

並列記号処理研究の指向性　　並列記号処理技術と並列知識処理技術は、図4に示すような関係にあるものと考えよう。すなわち、前者は並列の非数値処理に関する基本的かつ一般的で幅の広い技術であり、後者はよりspecificで機能的には高いレベルを狙った技術と考えることにする。

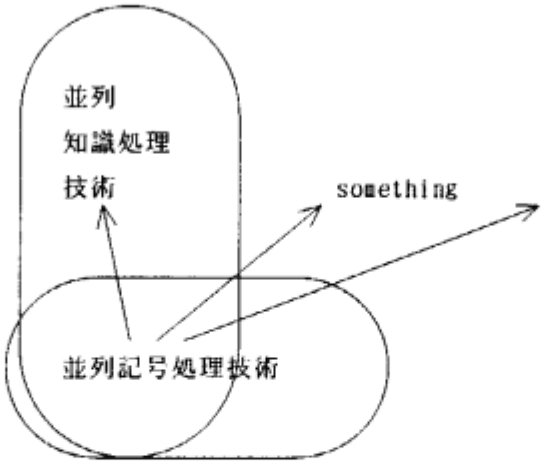


図4　並列記号処理技術と並列知識処理技術の関係

大規模MIMDマシンにおける並列記号処理研究の方向性として、ICOT正統派と考えられるものは、図4でいうと並列知識処理技術を強く指向した並列記号処理の研究であろう。しかしながら、並列記号処理技術の幅と厚みを増し、並列処理文化と呼べるところまで成長させるためには、それだけで十分であろうか。間口を広げ、より一般的な並列非数値処理の研究として多くの研究者を巻き込むことも重要ではなかろうか。そして並列記号処理ブームを巻き起こすところまで層の厚みを増すことが必要と考えるのである。知識処理を強く指向した並列記号処理研究の他に、知識処理的ではないが非常に計算量の多い非数値計算を並列に行なうような指向性の研究も、並列処理文化の幅と厚みを増すのに極めて友好と考えるのである。

知識処理に限定しない並列記号処理も標的に　前述のように考えるとき、我々の最終目標へ至るための目前の標的は、大規模MIMDマシンにおける並列記号処理の研究として、

(1) 知識処理指向の並列記号処理

(2) 知識処理に限定しない大規模並列非数値計算

の両方の研究を進めてゆくことではないだろうか。

ここで(2) に当る問題はいかなる問題かについて少し補足をしよう。まず a. 計算量が十分大きく、今の計算機ではほとんど能力不足となる（またはなりそうな）問題でかつ、b. 数値計算主体の問題ではなく、さらに c. アルゴリズムが明確・単純でデータ構造が固定化しているというのではない問題である。特にc.の条件というのは、ベクタプロセッサなり、画像処理マシンやシミュレーションマシンなりの専用プロセッサで効率良く実行できにくいような問題であり、逆にいうと、アルゴリズムが単純かつ明確でデータ構造が固定化している問題については、効率の良いspecificな専用マシンが作り易いともいえる。(2) で対象とする問題は、そのような専用マシンで扱えない種類の問題ということにしておきたい（専用マシンが得意とする分野は専用マシンに任せてあげよう）。

並列処理研究の幅と厚みを増すには(1) のみでなく、(2) に関しても、十分に魅力的で実用に供したくなるような事例をみつけ出し、プログラム化して動かして見せることが非常に重要なことであると考えるのである。

3．今　日

汎用並列記号処理の文化を育てるために今行うべき具体的なとりくみ方について、ICOTではこのようなあたりから始めたらどうかと思うところを述べる。

（1）どの様な問題を扱うか

a．指向性による分類

i）知識処理指向の並列記号処理

自然言語処理－－－－－－－－－ICOTにエキスパートがいる　○
基礎研究にまだ時間を要する　×

エキスパートシステム

棋士システム

ii）知識処理に限定しない大規模並列非数値計算

半導体CAD －－－－－－－－－－－－－－ICOTにエキスパートはいない　×
結線問題　　がやりたいと思う人がいる　○
テストパターン発生問題
etc.

？？？

b．こんな観点からの分類も有る

少入力大量計算型問題　　最終的なAI問題は、システムが多くの知識を抱えており、少しの入力で非常に多くの計算量を発生し、そして意味のある少しの出力を行う。これを、少入力大量計算型の問題と呼ぶことにしよう。これまで、知識処理指向の問題として我々が探していたのは、少入力大量計算型の問題であったように思う。

多入力少量計算型　　しかしながら、最終的な知識処理マシンが非常に大規模なシステムであると思えば、我々が近い将来に作る並列マシンは、そのごく一部の規模しか持ち得ないであろう。複雑な処理を扱う大規模なシステムの一部を切り出してきたとき、全体システムは少入力大量計算型であっても、切り出してきた部分システムは、計算量に比べて入出力処理の比率が高くなることが十分考えられる。すなわち、多入力少量計算型の傾向が出てくる可能性がある。

後者の問題も扱ってみよう　　マシンの性能・規模が小さいとき、それに見合った部分システム問題を載せようとすると、多入力少量計算型の傾向が出てくるかもしれない。部分システムの例にはならないが、例えば構文解析を例にとる。解釈の仕方が非常にたくさんある一つの文の処理を少入力大量計算型とすると、

解釈の数の少ない簡単な文を次から次から処理するのは、多入力少量計算型の傾向を持つといえよう。この様な問題は、I/O のバンド幅をを必要とすると共に、データのパイプライン的な処理、そして計算に入る前に各PEにデータを分配したり結果を回収したりといった処理を発生しそうである。この様な傾向の問題はけっこうあるように思われ、現実的な応用と結び付きやすい可能性がある。またマシン屋にとってこの種の問題は、PEやネットワークへの負荷のかかり方が異なるという点で注意を要する。

（2）並列ソフトウェア文化の構成要素

図２では問題を並列プログラム化し実行する過程として、設計技法、アルゴリズム、プログラム書法、負荷分散指定の各項目を階層的に表現した。これらは、プログラミングの際にはむしろ、混然一体のものとして扱われることも多かろうが、ここではそれらを並列ソフトウェア文化の構成要素と考え、ICOTで研究を進めるうえでの研究項目を整理してみたい。

a．並列言語

現ＫＬ１－ｕ　　オブジェクト指向機能に重点が置かれた並列言語で、主にPIMOS記述用に設計されている。論理型言語らしさはかなり薄くなった。汎用言語としてどのくらい成長するかがポイント。いろいろなタイプのプログラムを書いてみることがとにかく重要。

探索問題記述用言語，ｅｔｃ　　現KL1-u が得意としないような分野向けの言語を別に設計する必要があるかもしれない。

b．並列アルゴリズム

逐次計算の文化の中でもxxアルゴリズムと名前のついているものはそんなにたくさんは存在していない。例えばソーティングなどは有名。ここでは並列計算のためのそういった問題、すなわち部品として使えるようなまとまった問題で計算効率が非常に重要となるような問題のことを考える。

例えば探索問題などは、類型パターンに分類しておき、良いアルゴリズムをみつけておきたい例であろう。このようなspecificな問題についてアルゴリズムの研究を進める意義は、並列プログラミング用の部品を増やすという点や、問題が明確であるためにあとでのべる d．などの研究を促進させるという点にあろう。また試作マシンのベンチマークプログラムとしても有用であろう。

並列アルゴリズムを考える場合、次の点が重要である。

ｉ．計算量が小さいこと－－－計算量のオーダが低いこと

ii. 並列度が高いこと

iii.部分問題間の通信量がなるべく小さいこと

iv. 特定プロセスへの通信の集中がないこと

v. グローバルな通信より局所的な通信の比率を高められる可能性があること

これらのうちiii.～v.は、あとでのべる d. と係わりが深い。

c. 並列プログラム書法

問題をプログラム化するときのプログラミングスタイルのことである。 b. の様に特定の問題に限らず、並列問題一般のプログラミングを対象と考える。並列問題を素直に表現し、実行効率も良いような、プログラミングスタイルの類型パターンが整理できると都合が良い。

GHC の様な言語ではどの様にでも書けてしまうために、この種の努力が重要である。それに対して、ユーザ言語がプログラミングスタイルをある程度規定するように、言語設計してしまう行き方もある。現KL1-u がその方向であるが、言語の汎用性は減少する傾向である。

類型パターンとその実行時の特徴が、b.のi.～v.の様な観点から整理できると良い。

パターンとしては、プロセス型、ジェネレータ、フィルタ、コンシューマ、バウンデッドバッファ、・・・といった低レベルのものから出発して、それらの組み合わせの類型が重要となろう。

d. 負荷分散とスケジューリング

並列プログラムを実際の並列マシンにマッピングして実行しようとするとき、負荷分散やスケジューリングに関するユーザ指定が重要となる。ユーザにより、PE数や物理的なPE配置まで考慮した指定を行うのではなく、ユーザプログラムの動特性をなるべく詳しくシステムに伝えるというセンスで指定を行う。重視すべき項目は次の通り。

i. 通信の局所性の維持

ii. 負荷の均等化

iii.好ましい実行順、枝刈り効率など

これらはプログラムが出来上がってから考えるのではなく、設計段階からb.,c. と合わせて考えられるべきであろう。

e. 並列向き設計技法

大規模な複合問題の設計技法は、要求仕様定義の問題などとも合わせ、逐次型マシンの世界においても、十分に確立されてはいない。設計技法の分野は、前述のb.～d.に比べると、並列と逐次の差が小さいのではないかとも思われる。ともあれ1000PEマシンを使い尽くす様な大規模複合問題を設計する局面を早く迎えることが先決である。

４．明　日

私はハード屋であるから、新しいマシンが世に出ることには非常なる喜びを感じる。特に実用に供されるようであれば嬉しい。

2001年宇宙の旅のＨＡＬは、確か1992年だか1994年にプロトタイプが作られたことになっていたと思う。

1992年だと、1024PE. メモリ1Gw のPIM は、標準筐体4 本、頑張れば2 本に実装可能である。が、まだ大きい。２～３年たてば筐体１本になる。

知識処理を強烈に指向した並列処理は、基礎研究の進捗の関係から、ゆっくり進展するであろう。一方、よりgeneral な並列記号処理は、良い例が示されれば、もう少し急速に進歩するかもしれない。各PEに数値演算プロセッサを抱かせることは、応用を拡げる面で役立つであろう。

PIM が筐体１本になれば、super PSI と接続して、パーソナルLSI CAD システムに使いたい。specを記述すると、論理設計、シミュレーション、マスク設計、テストパターン生成まで個人環境上でやってくれて、ネットワークでそのまま工場へ発注する。一か月後LSI チップが届けられる。作りたい回路機能はほとんど、プリント板を使わなくても一チップに収まってしまい、はんだ付けの苦労はずいぶん減る。

さらに数年立つと、1000PIM はデスクサイドに収まる。この頃になれば、自然言語対話機能付き汎用ワークステーションがかなり使い物になっているかもしれない。KBM 機能とspeech recognition processorも統合されているだろうか。

これに使いたいのだという夢があると、苦労して作る側から楽しく使う側へ、いつか移りたいと思うだろう。マシンが並列に、まともに動き出すと、早く移りたいと思うかもしれない。並列マシンを前にしてそう思う人が増えるほど、並列処理文化は厚みと広がりを増して行くにちがいない。

１９８７．７．１７
ＩＣＯＴ ４研　中島克人

「並列マシンはかくありたい」

０．はじめに

白状するが，私はハードウェア屋ではない。 大学では少々のハードウェア設計とラッピング，入社後しばらくＬＳＩの論理設計などをおこなったが，大学でも，会社でも，そのほかの殆どの時間を「水平型マイクロ命令計算機のマイクロ・アーキテクチャ設計」とそのマイクロ命令でのコーディングで過ごしてきたといって良い。私は日本でも有数の「マイクロ・アーキテクト」であると自負している（他に自慢できるものがない）。
私のこの専門領域に携わる人の数は，マイクロプロセッサの発達やＲＩＳＣなどというラッタッタまがいのものの流行に押されているせいもあり，年々減少している。 私の専門技術は今や，大企業の中のほんの一部のセクション（例えば，何かの専用ハードウェアの設計部門）かＩＣＯＴ（でのほんの一部のセクション）でしか役に立たないのではないかと心配している。 そこでマルチＰＳＩ-Ｖ２の研究・開発にも参画する事にした。（ただし，こちらのほうも，当分は企業の商売に役立たないと思われるが．．．）

１．要素プロセッサの内と外

さて，御存じのように，
我々はＩＣＯＴにおいて，疎結合マルチプロセッサ・システム「マルチＰＳＩ-Ｖ２」を研究・開発中である。 マルチＰＳＩ-Ｖ２においては，早期開発という使命のおかげで，ＣＰＵやネットワーク・ハードウェア（両方を合わせて，「要素プロセッサ」または「ＰＥ」と呼ぶ事にする）の設計時には，それほど細かい最適化には気を付けずに済ましてきた。 ただし，マルチＰＳＩ-Ｖ２の実現に関して効果のありそうな一般的（汎用的）な機能（例えば，ＣＰＵ内では命令コードを１０ビットにして，機械語命令を豊富に持たし得るようにした事や，色々なタグ部判定のためのマイクロ・オーダを用意した事，ネットワーク経由のパケット到着は割込みでＣＰＵに知らせる事等）は予め用意した。 従って，この要素プロセッサは，汎用の論理型言語マシンというのが正しい。

「要素プロセッサ（ＰＥ）の中での処理に比べ，他のプロセッサへ仕事を投げ出した場合は，かなりのオーバヘッドを覚悟しなければならない。」という言葉は，並列マシン研究者の間では古い古されている。
ネットワーク・ハードウェアやその上を走るメッセージの組み立て・分解に特別に趣向を凝らしていないマルチＰＳＩ-Ｖ２では，この言葉はなお重くのしかかる。 しかし，このＰＥ内外のギャップは，どんな並列マシンにも存在するものである。その差が小さいといわれるマシンにおいても，ＰＥ数を大きくすると，物理的な実装面の制約から，大きなギャップが必ず，どこかに生ずるはずである（例えば，中期ＰＩＭのクラスタ間）。 従って，どのＰＥ間も（ＰＥ内に比べれば）等しく遠いマルチＰＳＩ-Ｖ２では，我々研究者にとって，「何とかしなくては」という半ば開き直りに近い勇気（研究の動機）を与えてくれる。

思い起こせば，フォン・ノイマン計算機のハードウェア・アーキテクチャの改良の歴史は，メモリとＣＰＵの距離を如何に縮めるかにあった。 もちろん，デバイスの進化を抜きにしては語れない部分ではあるが，アーキテクチャ面では，キャッシュ・メモリ／バッファ／ハードウェア・スタックの発明，パイプラインの発明等がある。 いま，非フォン・ノイマン計算機の草創期において，マルチＰＳＩ-Ｖ２はまさにこれらのアーキテクチャ上の発明がなされる前の姿を連想させる。 各ＰＥにとって，自ＰＥ内の処理は昔でいうレジスタ・オペレーションで，自ＰＥ以外の世界は昔でいうメモリ（恐らくコア・メモリ。筆者はそれ以上古いものは知らない）である。 このメモリは旧来のものと違い，インテリジェンスがあり，自分と同種・同等の仕事を自律的にやる能力を有している。

今，我々はこの遅い「メモリ」を用いて何かを行おうとしている。 マルチＰＳＩ-Ｖ２上では，ハードウェアはもう与えられているのであり，如何にして「レジスタ」内での処理を多くし，「メモリ」上の処理を「レジスタ」のそれの１００分の１（以上...とＩＣＯＴでは言っている）少なくしようかと検討している所である。 そしてＰＩＭのハードウェアを考えている人達（一応「他人事」と言うことにしておく）は，如何なるハードウェア・サポートが効果的であるかを検討しているわけである。 中期ＰＩＭでは共有メモリ型マルチプロセッサをクラスタとする階層構造を検討中であるが，あるＰＥからクラスタ内の他のＰＥを見ると，それはまさに「キャッシュ・メモリ」という位置づけになる。

さてそう考えると，我々並列マシンの研究に携わっているものにとって，大変明るい未来，あるいは別の言葉でいうと大きな可能性を感じるわけで，また，設計をするにあたり，「これはＰＥ数を無限に増やして行った場合には通用しないアーキテクチャだから」などと卑下したり諦めたりする必要は無く，「現在または近未来のデバイス技術と実装技術に適したもの」と胸をはる事ができるわけである。

しかしながら，アーキテクチャ屋の一人である筆者個人の好みとしては，（今度は？）遠未来のアーキテクチャ，つまり，大規模並列（ＰＥ１０００以上は実装できる）アーキテクチャを目指したい。 筆者は，つい最近になって並列マシンの研究に参画したばかりの新米であり，また，「素人（現実を知らない人）ほど大きな夢を抱く傾向にある」という事も承知の上で，敢えて言っているのである。

２．私のＰＩＭ

２．１ トポロジ

「大規模並列＝階層の無い一様なＰＥ接続」ではない。 どんな階層構造を持つマシンも，ＰＥ数をどんどん増やしていけば，最後には，一番上の階層では，なにか一様な構造を持ち込まざるを得ないと考える（ツリー状のアーキテクチャのリーフ側へ延ばしていけば... 等という考えは，特定のＰＥ間の距離を極端に悪くする方向なので，やめたほうがいいと思う）。 従って，筆者の着目点が，一番上の階層にのみあり，二階層目以下は１つのＰＥとみなすと言うことである。

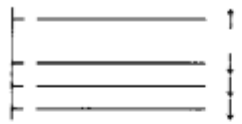
さて，マシンの実装面から考えると，その最上位階層においては，如何なるトポロジがふさわしいであろうか？

ここまでいえば，答えは殆ど絞られてくる。 筆者の解は「メッシュ」である。 恐らく２次元メッシュを折り畳んで実装するか，３次元のメッシュであろう。 例えば立方格子の場合の各「ＰＥ」からの足は６本であり，極めて実現性の高い数である。 他にも「六方細密充填格子（？）」等の中学（高校？）時代の理科の教科書にでてくるようなものもあるが，大差は無かろう。 むしろ立方格子は，その格子点にアドレスをふりやすい事や，絵を書きやすいなどの，「ソフトウェアを書く人に説明し易い」という素晴らし長所を持っている。

２．２ ハードウェア実装方式

ＰＥの三次元配列を提案したからには，その実装についても何かコメントするのが良心的かと思い，若干考えてみた。 しかし，あまり良い考えには至っていない。

現在の「プリント基板」の上に２次元に並べる方式は，確かに調整に便利である。 従って，この基板を重ねて行く方式を踏襲するとしたら，上下の基板間を直接ケーブルで繋ぎたい。調整時のためにこのケーブルは取り外し可能となっている。取り外しの時には基板間の隙間は拡げたい。従って，バックボードへのコネクタ部分が蝶番のようになっており，何処か一箇所ぐらいは拡げられるようになっているのが良かろう。
蝶番ではなく，スライドするコネクタでも良かろう。



（こういう事を書くと，「やっぱりハードウェア屋だなぁ」と言われそうであるが，こんな事が現実的かどうか等は，何も考えていないのであり，会社の人に言うと「バカヤロゥ，そんな簡単に行かへんで」（出向元の会社には大阪の人が多い）といわれるかも知れないのである。 そういう意味で，やっぱり私はハードウェア屋ではありません。）

１９９２年頃（５Ｇプロジェクト終了時点）には，基板１枚に１６ＰＥぐらいは期待できそうである（？？。 ただし，信号数の面では，一つのＰＥからＸＹＺの６方向に２０本ずつの手が出ているとした場合，基板外への信号数はＸＹ方向のみの３２０本（２０Ｘ４Ｘ４）となり，現実感のある数字。）であるので，１筐体６４ＰＥとして計 １０２４，１６筐体で合計

１６Ｋ ＰＥ !!

ぐらいまでは可能かも知れない。

なお，システムの増設は任意の次元方向というわけにはならないが，もともと繊長であるので，たとえば正立方体にするなら，１６Ｘ１６＝２５６筐体を並べて，２５６ＫＰＥとなる。

２．３ ＰＥ間通信をサポートするハードウェア

さて，アーキテクチャの話に戻ろう。 ＰＥ外の世界が「メモリ」であると思った瞬間に思い付くのは，メッセージに乗って来るデータのキャッシュである。 これは当然行えばよいのであるが，残念ながら，読み出しデータのみのキャッシュにしかならないであろう。つまり，ライト・スルー・キャッシュである。しかも，読み出しデータが未定義の場合にはキャッシュしておくわけにはいかない。 しかしこれでも，多環境が必要な場合に比べ，バージョン管理などが必要無いのであるから，まだましだと思わなくてはならない。

この読み出しデータのキャッシュはマルチＰＳＩ-Ｖ２でも行う予定であるが，ハードウェア・サポートまでする必要はなさそうである。 ただし，メッセージの分解・組み立てをおこなう部分（キャッシュ・コントローラ）はあった方が絶対に良い。

さて，インテリジェンスのあるメモリを持つプロセッサとしては，メモリ内での自動ＧＣを期待するわけである。 メモリ内（あるＰＥ内）でのみのローカルなＧＣを行えるように，マルチＰＳＩ-Ｖ２では沢山の種類のタグを設けたり，ＰＥ間を渡るデータは輸出表や輸入表等を用いて管理している。また，そもそも一括ＧＣが頻繁に走らないようにＭＲＢを全面採用している。 いわば，多額の税金が支払われているわけである。
これらの処理に対するハードウェア・サポートは大変効果がありそうである。

これらの努力をかさねていけば，ＰＥ内処理と外部ＰＥで行う場合の格差は（ネットワーク・トラフィックでの障害を除けば）一桁ぐらいは縮められると思われる。

２．４ ＣＰＵ

ＷＡＭアーキテクチャは，従来の機械語処理用ハードウェアにタグに関するものを付加するだけでかなり高速なＰｒｏｌｏｇ実行を可能にする事がＰＳＩ－Ⅱ等によって証明された。 ＰＳＩ－Ⅱ ではさらに幾つかの特殊なハードウェアを付加してはいるが，どれも小さなものである。 最近このＣＰＵにファームウェアでＫＬＢをインプリメントする事を検討しているわけであるが，幾つかのハードウェアはかなり有効に使われそうである（ＫＬＢはＷＡＭベースとしているのであたりまえであるが）。 特に感じている事は，以下の２点である。

- マイクロプログラム方式は本当に便利である。
- ＭＲＢサポートのハードウェアがほしかった。

前者について，一言述べさせていただく。
並列推論マシンは確立した技術ではない。従って，この時期にマシンを作るとしたら柔軟性は大事である。 効果の予想される幾つかのハードウェアは実装しておくべきであり，また，それをしなくてはアーキテクチャの進歩はありえない。しかし，ガチガチにハードウェア化してしまうと，処理系の実装方式の改良などについていけない。従って，１～３割の速度を犠牲にしても，マイクロプログラマブルとしておくのが良い。

ハードウェアはＲＩＳＣマシンとしておき，上記の柔軟性はコンパイラで吸収するというやりかたは，グローバルな最適化がうまく行くとマイクロプログラムよりも勝れる可能性はあるが，逆に下手をすると５割以上の速度損失となる可能性もある。また，ＰＥへのコードのダイナミック・ローディングを考えた場合，ネットワーク・トラフィックの上でも，ＰＥごとのメモリ量の面でもかなり不利である。 またそもそも，ＲＩＳＣばかりではアーキテクチャの進歩はありえないし，特殊なハードウェアを付加する話は，ＲＩＳＣには馴染まない面が多いと思う。 特にＭＲＢ処理のように低レベルで動的な判定を必要とするものに関し，ＲＩＳＣでは色々の問題がある。

ところで，将来（１０年ぐらい先）におけるＣＰＵであるが，当面は，ＣＰＵ内では逐次実行に頼る事になるであろう。そして，次第にＰＩＭの経験を積む事により，所謂ＣＰＵとは非同期な特殊なハードウェアが常識化され，ＣＰＵ全体として，機能的に分化した複合プロセッサとなるであろう。 しかし，個々のプロセッサはやはり逐次ベースであろうし，また，それで悪い理由は何もない。

２．５ 言語

ＦＧＨＣはインプリメントを考えれば考えるほど，並列マシンにとって楽な言語である。多環境は必要無いし，ストリーム等という（効率はあまり良く無さそうであるが）同期の手段もそなえている。 こんな楽な言語で当分済ますならば，その上位言語は，ユーザにかなりのサービスをすべきであり，ハードウェア（処理系）は，上位言語使用による効率低下を補うだけの性能を達成すべきである。
将来はその上位言語を直接サポートする方向であるが，それによる性能向上はせいぜい１桁前後であろう。

２．６ 性能予測

１９９２年（５Ｇプロジェクトの終了時）頃の私のＰＩＭの性能をざっくり見積もってみる。

ボディ・ゴールが合計３つあり，１つは直接実行（ｅｘｅｃｕｔｅ），もう１つは自ＰＥ内にエンキュー（ｅｎｑｕｅｕｅ），もう１つは他ＰＥへの投げ出し（ｅｎｑｕｅｕｅ＿ｗｉｔｈ＿ｐｒａｇｍａ）という，負荷分散がうまく行っていないかなりひどいケースで考える。

現在ＰＳＩ－Ⅱ でのａｐｐｅｎｄの１リダクションあたりのステップ数は，ＭＲＢフルサポートで約３０ステップであり，ＭＲＢなしで約２５ステップである（と思う）。将来ＭＲＢサポートのハードウェアを付加するとして，この５ステップの短縮は可能という感じにしておく。 また，ネットワーク・メッセージのサポート・ハードウェアは付加されていると考える。

以下，何の根拠も無い数字を，フィーリングだけで積み上げていく。

３つのボディの呼出しで約７０ステップ（２５＋１２＋３３），１リダクションごとに他ＰＥから投げられたゴールのエンキューに約２５ステップ，リターン処理に３０ステップ（１５＋１５），投げ出し元ＰＥへの終了メッセージ送出に約２５ステップとして，合計１５０ステップとなる。 このケースでは，これでも楽観的な数字かも知れない。何故なら，他ＰＥへの１つのアクションのために，同期などの関係で複数のメッセージが必要になる場合があるためである。

さて，サイクルが２０ｎｓｅｃとすると，上記３リダクションで3micro ｓｅｃとなり，ＰＥのアイドルが無いとすると，これは１ＰＥ当たり，

１ＭＲＰＳ

となる。 １６，０００のＰＥで

１６ＧＲＰＳ

となる。 まあまあの線ではないかと思う。

３．おわりに

あれこれと考えを巡らした結果，あまり良い考えは思いつかなかった。 そんなに簡単に思い付くようだと「やりがい」がないともいえる（という言い分けも十分に平凡である）。

私の将来のＰＩＭの研究での楽しみは，個々の新しいハードウェアの発明もさることながら，ひどい負荷分散ソフトのためにさんざんな目に会ながら，徐々に（といっても１桁ずつか？）向上する効率に一喜一憂する事になろう。また，今度こそはオバケみたいに大きなハードウェアを作って（自分はマイクロ・アーキテクチャなどを検討するだけとし，「ハードウェア調整」という楽しみは人に譲ろうと思っている），世間の人の嘲笑を受けて見たい気は多いにある。

以上

並列アーキテクチャに対する私の主張

並列アーキテクチャに対する私の主張

富田眞治
（九州大学総合理工学研究科）

大規模並列計算機の商用化の段階として、定型的処理に対して演算パイプライン方式による高速化がスーパコンピュータによって達成されてきた段階といえる。演算パイプライン方式が今のところ主流になっているが、ベクトル化コンパイラの開発、ユーザのプログラミング経験を集積して、今後、並列処理はより非定型処理指向、高並列化へと発展し、これまで商用システムとしては成功例が少なかったSIMD方式、マルチプロセッサ方式も商用レベルに大きく発展するものと思われる。各種方式についての私見を述べる。

1　演算パイプライン方式

（1）スーパコンピュータ

演算パイプライン方式の利点は、

・逐次的メモリ、入出力機構と合致し、ハードウェアが作り易い

・ベクトルや行列に対する理解しやすい一次元的処理が可能である

・最内ループに対するベクトル化は比較的容易である

などである。スーパコンピュータはこの方式を利用した極めて汎用性の高い並列計算機である。デバイス技術の発展によって一層の高速化が図られる一方、方式的には一層の汎用化と特化がなされよう。しばらくの間、主流の座を占めることは間違いなく、また汎用化と特化でこの方式がどこまで広い応用分野をカバーできるか興味のあるところである。演算パイプライン方式は本質的にはSIMDである点に注意。

・汎用化

リストベクトル機構が汎用化に果たす影響は極めて大きい。データ供給系の高速化と柔軟化が必要である。BSPやStaranなどのSIMD計算機が定型的なデータ構造に対するアクセスを集中制御していたのに対し、リストベクトルを含むデータ供給系では分散制御が必要となり、知的機能を入れると一層柔軟性がでよう。

・特化

簡単な装置を付加して特殊応用向きに特殊化して、高速化を達成する方式である。日立ではデータベース演算、論理シミュレーション処理への特化が報告されている。

（２）VLSIハードウェアアルゴリズム

各種ソータ、シストリックアレイなどのVLSIハードウェアアルゴリズムが多数提案されている。データ量やデータ形式についての柔軟性が必要である。

２　SIMD方式

Illiac IV、BSP、MPP、Connection Machineなどがある。前３者は通信路の制御もSIMD方式でなされる。従って処理できるデータ構造は定型的なものに限られるが、通信オーバヘッドは小さく低粒度の演算に向いている。後者は通信は非同期でなされ、不規則なデータ構造も扱える。MIMDとの性能比較も興味がある。

３　VLIW方式

超長形式命令型計算機では、並列操作可能なものをコンパイル時に１つの命令に多数埋め込むことによって高速実行が可能になる。ハードウェア構成は単純であり、また演算器間の通信は命令で直接制御されるので、低粒度の演算に有効である。汎用計算機の機械命令を多数同時に起動し実行させる多重命令パイプラインの演算装置としても用いられよう。

４　マルチプロセッサ

マルチプロセッサは長い歴史があるにもかかわらず、余り多くのことが明らかになっていない。密結合あるいは疎結合がいいのか？トーラスあるいはトリーがいいのか？共通なフェアな土俵の上で各種の方式が評価されていない。ハードウェアを作ってしまうと多様なパラメータでの評価ができないし、物理的なシステムパラメータ（たとえば、PE数）に引きずられた評価しかできないことが多い。したがって、作成者の自己満足で終っている場合も多々あるようである。ICOTのマルチPSIでトーラスがアプリオリにでてくるようでは困るのである。そこで、遠回りのようであるが、まずフェアな土俵として大規模なテストベッドが必要と考える。テストベッドをマルチプロセッサで構成し、相互結合網を可変構造にし、評価データをほぼリアルタイムで得られるようにしたい。相互結合網は３ステージClos網（$m > 2n - 1$）位が妥当であろう。マルチプロセッサのマルチプロセッサによるマルチプロセッサのためのシステムが必要である。応用に最適なトポ

ロジを動的に設定できる。プロセススイッチの高速化や負荷分散も重要であるが、並列プログラムのデバッグ手法の確立がもっとも重要であろう。この意味では共有メモリ型よりメッセージ交換型の疎結合マルチプロセッサがいいかも知れない。

5　データフロー方式

現状のデータフロー計算機は、演算の機能レベルに較べて通信オーバヘッドが大きいこと、必要なメモリバンド幅が大きいこと、線形メモリや参照の局所性を利用しにくいこと、マッチング記憶が複雑であること、デバッグが困難であることなどの理由で実用化には至っていない。プログラムの局所的領域に対するデータフロー方式としてすでにIBM360/91、CDC6600、スーパコンピュータのチェイニング機構などがある。汎用計算機の命令パイプライン（演算部）の高速化、VLIW方式などの複数演算器の制御に積極的に使用できよう。循環パイプラインを使用し、プログラム格納にメモリを使用し、パケット通信というオーバヘッドの多い方式ではパフォーマンス向上は得られないように思う。アナログ計算機のような素朴なデータフロー方式を有効に演算装置に動的に埋め込める方式がよいのではと思う。

ＰＩＭ－ＷＧワークショップ資料

東京大学工学部　相田　仁

ＰＩＭはマルチユーザ，マルチタスクで動くとの仮定の下に，ふだんから疑問に思っている次のようなことに関して，議論してみたい。

１．”全てのＰＥが同一のコードのコピーを有している”でよいのか

タテ割り分業がよいヨコ割り分業がよいか －
ＰＥ内での機能分業は別として，ハードとしては同一（或いは少種類）のＰＥで構成したほうが作りやすい（ネットワークを考えやすい／管理しやすい）。
しかし，なかにはスーパーコンピュータのように，機能分割がうまくいった例もある。
全てのＰＥに同一のコードが載っている必要があるか？
異なるユーザの間でのコードの競合はどう解決するか。逆にライブラリのようにユーザ間で共用したいものはどうするか。

２．ＰＩＭは定常的に動くのか

1000台規模，ＩＧＬＩＰＳのＰＩＭにおいて
タスク（ジョブ）の平均 elapsed time は？
平均 cpu time は？
平均メモリ使用量は？
マルチユーザ／マルチタスクは時分割？／空間分割？

cf．ガーベジコレクション：
瞬間的に動いて一旦おさまるならば，それまでの間メモリがもてばよい。途切れなく動く必要があるのなら on the fly が必須となる。

放送機能を持つプロセッサアレイによる並列ユニファイアの構想

筑波大学　電子情報工学系
和田耕一

1．はじめに

現在まで，Prolog 言語の単一化機能や後戻り機能など，言語の特徴的な機能に着目して，これを高速化する研究を行ってきた．ここで，言語面の特徴のみでなく，実際の応用プログラムの特性を考えると，データベースのようにルール節の数に比較して膨大なファクト節を有するものが多い．このような応用に関しては，事実の取り扱いや高速な検索法など，新たに解決すべき問題がある．

本稿では，上記の問題に対応すべく現在検討中のマルチプロセッサシステムの構想について述べる．本システムは，高並列計算の導入により多数のファクト節との単一化処理を高速化することを目的としている．

2．設計方針

設計にあたっては，以下のような点について考慮する．

①プロセッサ間結合方式は，ＰＥ（要素プロセッサ）間の距離が均等なバス結合とするが，バス競合のオーバヘッドを軽減するため，ローカルなＰＥ間接続バスを設ける．

②データベースや全解探索問題への応用を考慮し，ＰＥ間での放送機能，各ＰＥからホストプロセッサへの解の収集機能を強化する．

③データベースとなるファクト節は，プロセッサアレイ上の各ＰＥに分散配置する．

④規模としては，ＰＥ百台程度のシステムを目標とする．

3．システムの構成

本システムの全体構成を図１に示す．ＰＥは 68000 をＣＰＵとし，512Kbyteのメモリを持つ．各ＰＥは，隣接するＰＥと通信用メモリを介して接続されている．また，ホストコンピュータを含めて各ＰＥは，共通バスに接続されており，ホストコンピュータおよび各ＰＥはこのバスを用いて他ＰＥへデータを一斉に書き込むこと（放送）が可能である．

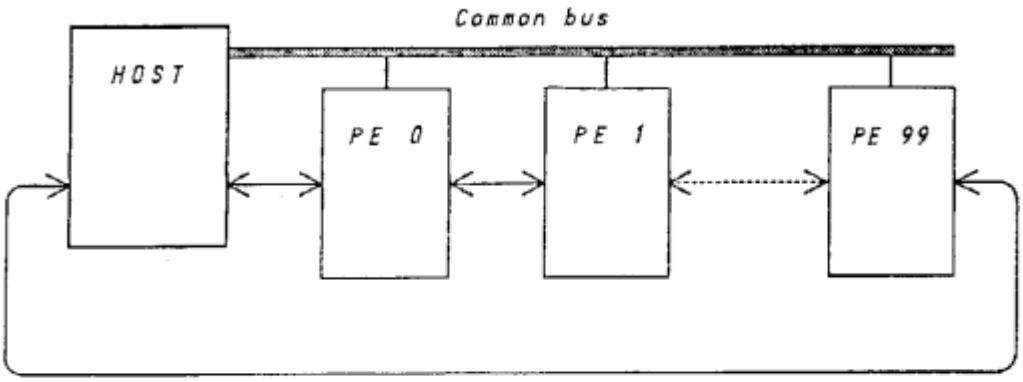


図1．全体構成

4．おわりに

プロセッサアレイによる並列ユニファイアの構想について述べた．当面は，ホストコンピュータからの問い合わせに対して，プロセッサアレイ上に分散されたファクト節との単一化を行い，解をホストへ回収するシステムを考えている．その後，セットオペレーションへの適用や，OR並列 Prolog の実現を予定している．

ＰＩＭ－ＷＧワークショップ資料

マルチプロセッサシステムＰＡＲＫ上の並列Ｐｒｏｌｏｇ処理系について

神戸大学工学部システム工学科　松田秀雄

１．ＰＡＲＫ

ＰＡＲＫは要素プロセッサとしてマイクロプロセッサ68000（モトローラ社製）を用いたマルチプロセッサシステムである．プロセッサの結合形態はバス型であり，バス競合を抑えるためにバスの階層化，メモリの分散化を行っている（図１）．

１．１　バス結合について

（１）実装が容易である．
（２）プロセッサ間距離がO(1)である．
（３）全てのプロセッサが対等である．
（４）プロセッサ台数の増加など構成の変更が容易である．
（５）台数がある程度以上増加するとバス競合のため性能向上が頭打ちとなる．
（６）台数が増えるとバスの配線距離が増大する．

１．２　ＰＡＲＫの構成

全体の構成：LS-TTLとMOS(ﾒﾓﾘ,CPU)，280IC／台
ＣＰＵ：MC68000（クロック8MHz）
バス：ローカル（ｱﾄﾞﾚｽ23,ﾃﾞｰﾀ16,ｺﾝﾄﾛｰﾙ27），共通（ｱﾄﾞﾚｽ23,ﾃﾞｰﾀ16,ｺﾝﾄﾛｰﾙ19）
メモリ：ローカル（128KB,ｻｲｸﾙ500ns），共有（512KB,非放送1.6μs,放送2.7μs）

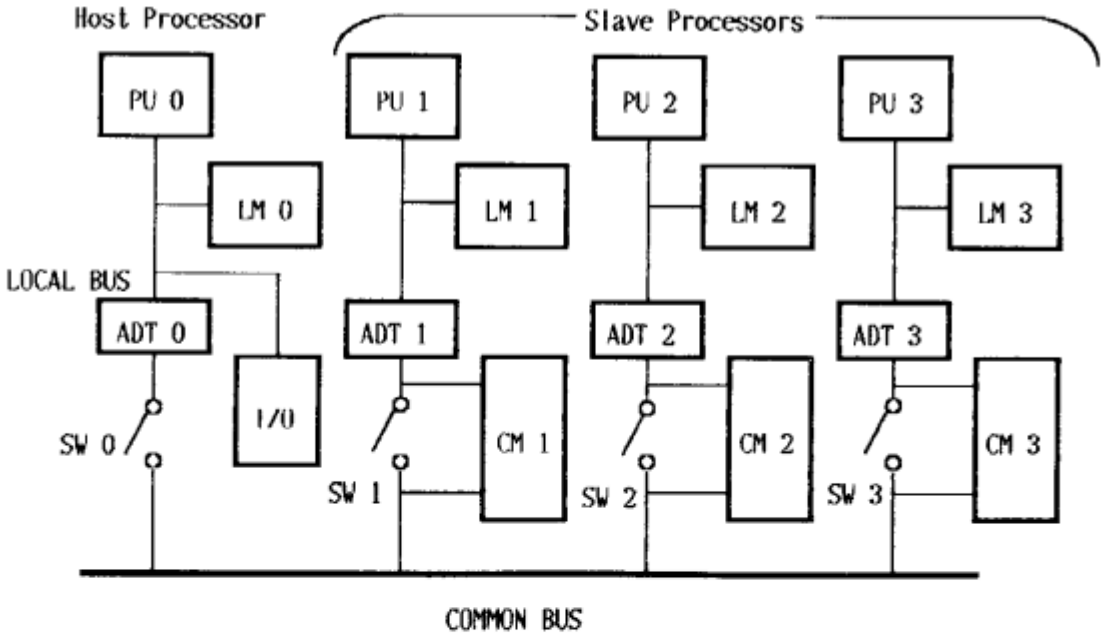


図１　ＰＡＲＫの構成

2．ＰＡＲＫ－Ｐｒｏｌｏｇ

（１）ゴールの逐次的な実行をプロセスとしてとらえる．明示的な並列実行の指定
（並列実行の粒度の調節）

```
p :- q,r,s.  （１プロセス）   p :- q // r // s.  （４プロセス）
```

（２）チャネルを介した多対１のプロセス間の同期・通信

```
p :- mkchan([C]), q(C), r(C).
q(C) :- C!a.  （送信，同期型）      q(C) :- C!!a.  （送信，非同期型）
r(C) :- C?X.  （受信，同期・非同期共通）
```

3．処理系の実現

（１）ＷＡＭコードに似た中間言語に変換後68000機械語に展開
（２）状態によりプロセスを管理　　free → scheduled → running → free
↑　↓
ready ← suspend
（３）キュー（scheduledとready）によりスケジューリング（readyキュー優先）
（４）プロセスのプロセッサ割当ては負荷の最小のものを選択（割当て後は変えない）
（５）動的なメモリ管理（共有メモリのページング）

プロセスを構成する領域
Areas to construct process

	割当メモリ	内容
ＰＣＢ	放送領域	プロセスの状態，PWBのアドレス
ＰＷＢ	ローカル メモリ	レジスタ退避・一時変数領域 他の領域へのポインタ，スタック
ＥＮＶ	共有メモリ	ローカル変数，グローバル変数 選択点，トレイル
ＣＯＤＥ	ローカル メモリ	オブジェクトコード
ＣＨＡＮ	放送領域	チャネル
ＭＳＢ	放送領域	メッセージバッファ

PCB: Process Control Block, PWB: Process Working Block
ENV: Environment, CHAN: Channel, MSB: Message Buffer

4．性能　８クィーン全解探索（７クィーンのプロセス８ケにより実行）

	プロセッサ台数（カッコ内は速度向上比） １台	２台	３台
同期通信	3009	1645 (1.8)	1298 (2.3)
非同期通信	2989	1501 (2.0)	1137 (2.6)

（単位はmilliseconds）

「並列アーキテクチャに対する私の主張」　　岡山理科大学　小畑正貴

ハードウェア

非同期並列と同期並列（小粒度非同期並列の限界）

並列計算機は計算機のパワーアップが目的であるが、計算機の高速化への要求として次の２点がある。

１）大規模な問題をそこそこの時間で処理したい場合。

２）それほど規模は大きくないが実時間処理が要求される場合。

１）に対しては並列処理の粒度を大きくとり、非同期並列で処理する方法（ＰＩＭでとられている方針）が有効である。問題がさらに大きくなっても部分問題間のつながりはそれほど広がらず、局所性は維持できると考えられる。階層的に分解して処理して行けば、ハード的にはプロセッサ台数を単純に（あるいは階層的に）増やすことで対応できるだろう。

このようなシステムで一つ考えなければならないのは「小規模問題に対しても同じだけの並列性が必要とされるか？」という点である。小規模問題は無理に並列処理しなくてもよく、大規模になっても（並列処理により）処理時間が大きくならなければ十分ありがたいと思うのだが。

２）の要求に対しては小粒度並列処理が必要となるが、その場合パイプライン処理やアレイ処理といったようなＭＩＳＤ、ＳＩＭＤ型の同期並列処理（クロックレベルではなく、静的負荷分散・通信路固定といった意味で）が入り込む余地があるのではないか。

同期並列処理では負荷の大きさがある程度そろっているならば、簡単な制御機構で同期のオーバヘッドを小さくできる（ただし純粋な同期待ち時間は大きい）。また同期や動的負荷分散のための通信は不用であり、通信路はデータのみの転送に使用できる。バスと共有メモリによる非同期並列処理ではプロセッサ台数は１０台程度が限界であるが、同期処理ではもっと多くのプロセッサが接続できる。

論理ゲートを導線でつないだノイマン計算機をベースにして小粒度の非同期並列処理を行なおうとしても、並列度の限界はすぐに見えてくるのではないか。どれかひとつをはずさなければ高並列計算機はできないと思う。

ソフトウェア

並列処理と並列プログラミング環境（並列処理はパワーアップのみに必要か？）

「従来の計算機でも処理速度さえ早くなれば並列計算機はいらないのではないか」「一般的にはプログラミングが余計にややこしくなるだけではないか」といった議

論をよく耳にする。現行の計算機のパワーを考えると並列計算機不用論は成り立たないのであるが、これだけだと応用レベルでの並列プログラミングは不用になる（もちろん並列度を引き出すための並列アルゴリズム開発には必要であるが、これはまた別に考えられるべきものである）。並列プログラミングは制御分野では当然のように使われているし、通常でも問題が大きくなると有用であろう。

「高速化のために並列計算機が必要でこれを動かすのに並列プログラムが必要」であることとは別に、「並列プログラミング環境が便利であり、それが並列計算機の上でうまく動く」という議論ももっとすべきである。それには並列環境全体について考える必要がある。

直列プログラム	並列プログラム	（応用レベル）
直列処理	並列処理	（ＯＳレベル）
単一プロセッサ	複数プロセッサ	（物理レベル）

・ハード的には物理レベルで複数プロセッサがあれば並列計算機である。
・１台のプロセッサでも複数プロセスが協調並行動作する機構があれば並列計算機と変わらない。（リアルタイムモニタ）
・複数プロセッサを仮想的に１台に見せ直列プログラムを処理した場合は、処理速度が上がったと感じるだけである（非常に重要で難しいことではあるが）。

並列プログラミングの良さは各部分では他のところを考えなくてよく、その中では直列的に仕事が進められる点にある。数時間程度の範囲での人間の意識的な動作や思考は直列的であると思う。並列計算機のためにすべてを並列に考えなければならなくなると思われるところに誤解があるのではないか。同期や通信を全部書かなければならないような並列プログラムはできれば書きたくない。「並列にやりたいときに、並列動作のための諸々の事柄から解放してもらえる」そういったのもが並列プログラミング環境といえないだろうか。

このような考えの実現として、関数型プログラムのデータフロー的処理が近いような気がする。すべての関数は常に動作状態にあり、関数を呼ぶときには処理するデータと結果の送り先を与えるようにする（従来の関数は呼ばれたら起動し、かならず呼んだところに戻る）。たとえば関数名（引数｜出力先関数名）という形で関数を呼ぶ。引数は一度にそろう必要はなく、出力先を省略すれば自分に戻る。また呼んだところで待っている必要はない。

最後に、私の考える並列計算機環境：

ほとんど従来の関数型言語＋並列化コンパイルＥＳ＋実行時に同期をとるマシン

並列処理ソフトウェアをどう作れば良いか？

87/07/17
吉田紀彦
九州大学工学部

□□　高並列処理へのプログラム変換の応用　□□

プロセッサ・アレイなどによる規則的細粒度並列処理は，高い並列性を追求し易い反面，その高並列性を活用するアルゴリズムの設計が非常に困難である．

データ駆動／要求駆動マシンのような命令レベル並列処理では，その並列性はアルゴリズムの関与するところではない．一方，荒粒度並列処理では，プロセスの内部動作は従来通りの逐次的な思考法で設計することができる．ところが，微少なプロセスが多数並行に動作する細粒度（高）並列処理では，プロセス群全体の動作を設計しなければならず，設計に際して大きな発想の転換が要求される．しかも，計算モデルが確立されていないので，どのように転換すべきかが明確でない．

現在考えていることの１つに，以前研究に携わっていたプログラム変換の手法を高並列プログラムの開発に応用する試みがある．変換によるプログラムの並列性向上，高並列処理の導出が目標であり，単純なシストリック・アルゴリズムの導出などに（一応）成功している．自動変換システムの作成も進めている．

変換の対象は，いわゆるストリーム並列の論理型言語（のサブセット）で表現された恒久プロセスである．ごく簡単な変換例を以下に示す（とりあえずＧＨＣを用いて表わす）．

パイプライン化（図１）

```
pp([X | XX], ZZ0) :- true | ZZ0 = [Z | ZZ], p(X, Z), pp(XX, ZZ).
p(X, Z) :- true | f(X, Y), g(Y, Z).
    ⇩
pp(XX, ZZ) :- true | p'(XX, YY), q'(YY, ZZ).
p'([X | XX], YY0) :- true | YY0 = [Y | YY], f(X, Y), p'(XX, YY).
q'([Y | YY], ZZ0) :- true | ZZ0 = [Z | ZZ], g(Y, Z), q'(YY, ZZ).
```

（狭義の）並列化（図２）

```
pp([Xs | XX], YY0) :- true | YY0 = [Ys | YY], p(Xs, Ys), pp(XX, YY).
p([X1, X2], Ys) :- true | Ys = [Y1, Y2], f(X1, Y1), g(X2, Y2).
    ⇩
pp'([XX1, XX2], YYs) :- true | YYs = [YY1, YY2], p'(XX1, YY1), q'(XX2, YY2).
p'([X | XX], YY1) :- true | YY1 = [Y | YY], f(X, Y), p'(XX, YY).
q'([X | XX], YY1) :- true | YY1 = [Y | YY], g(X, Y), q'(XX, YY).
```

□□　プロセスのネスティング　□□

プロセスの内部動作が逐次処理である必然性は全くない．プロセスの内部がより小さなプロセスの協調処理であるような階層的計算モデルについて，少し検討を進めている．なお，この計算モデルがどの様なマシン・アーキテクチャに対応するかは，今のところ，まだ明確でない．

並列処理の将来形としては，マクロに見るとプロセス・レベルの荒粒度並列処理で，ミクロに見るとデータ駆動／要求駆動，ないしは（応用を特定化した場合には）規則的高並列処理，という構成を想像している．

□□　雑感　□□

プロセッサ内でのプロセス切り替えのオーヴァヘッドは，かなり大きい．これを低減するために，Mach におけるthreadのような機構や，さらにはプロセス切り替え支援ハードウェアなどを考えることも，有用であろう．

今後，並列アプリケーションが充実することで（８クイーンや素数生成からはそろそろ卒業したい），これまで数多く提案されている並列アーキテクチャの特質（応用に対する向き不向きなど）も明確化してくるはずである．この分野が本当に面白くなるのはこれから，という気もする．

以上

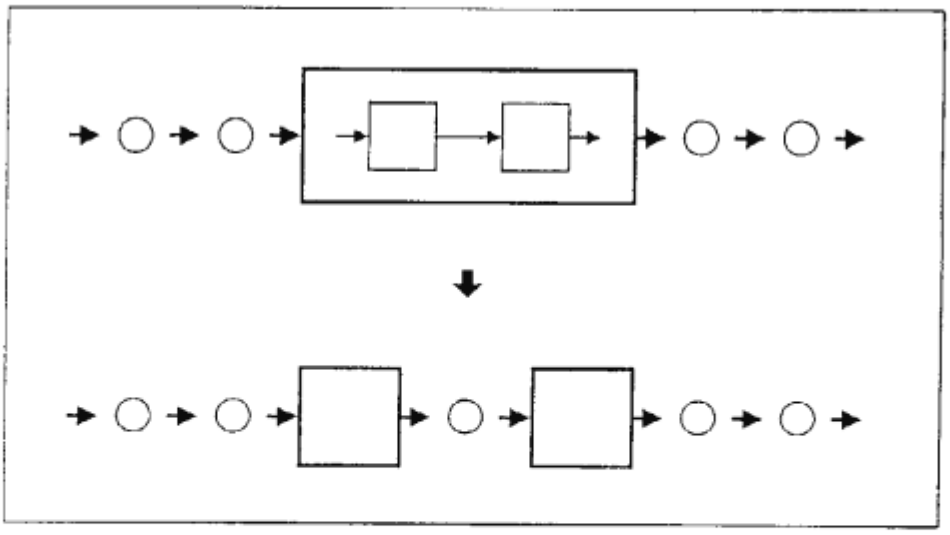


図1

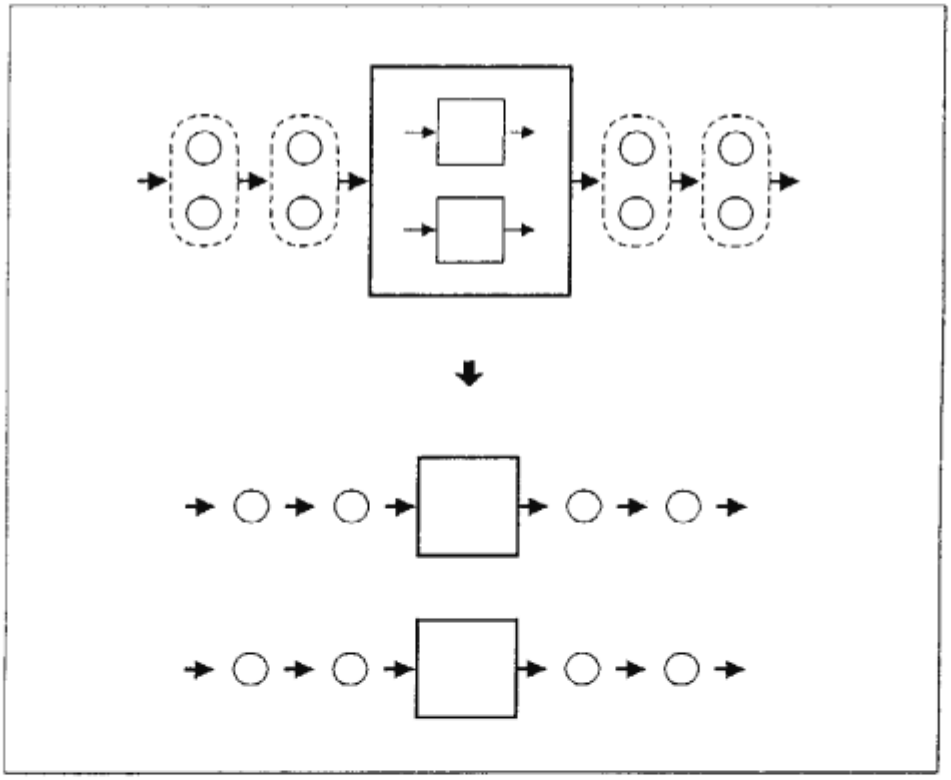


図2

メモリ割当の並列性について

松下技研　　川上　桂

.要旨

並列処理システムにおいては、演算処理の並列性とともにメモリアロケーションアルゴリズムにも新しい評価基準が必要である。従来の計算機於ては、（１）アロケーション速度、（２）メモリ使用効率　の２つが大きな評価基準であったが、並列処理システムにおいては、第３の基準として　（３）アロケーションの並列性が必要となる。　この観点からすると　Buddy　システムはよい性質をそなえている。

１記憶割当の頻度

図１は、データフローマシン上で典型的プログラムを実行した場合の、構造メモリにおける記憶割当および解放の頻度についてのシミュレーションの結果である。　図１－(a)は、単一のプロセッサで実行した場合、　図１－(b)は、無限個（十分大きな数）のプロセッサで実行した場合のものである。図１－(c)　は、この場合に、並列に実行された命令数である。プロセッサ一個の場合には、記憶割当は平均して発生しているのに対して、解放は集中して発生している。これは、新しい演算を実行し中間結果を生成する度に新しい記憶領域を必要とするのに対して、その中間結果が不要であることが明らかになるのは最終結果が出る頃になるということに起因しているものと推測される。即ち、単一代入規則による影響が作用しているものと考えられる。

図１-(b)　即ち、同一のプログラムを可能な限り並列に実行した場合には、記憶割当のほうが急速に立ち上がり、実行の並列度のグラフに追随した形になっている。これも、演算を実行する度に新しい記憶領域を必要とすると言う単一代入規則の影響がうかがえる。

以上の傾向はいくつかのプログラムにおいて共通に見られるもので、一般的傾向と考えられる。従って、並列処理装置上で、特に単一代入規則によりプログラムが実行される場合には、演算の並列性に応じて記憶割当も並列に処理される必要がある。

2　記憶割当の並列性

図２はデーターフローマシンの構造メモリの為の並列記憶割当システムの一例である。演算装置からの記憶領域要求(Space Request) および記憶装置からの解放要求(Liberate Request) に対して複数個の allocator が処理を行っている。このような記憶割当システムを実現する方式として、Buddy システムが便利な性質を備えている。即ち、Buddy システムにおいては、記憶ブロックの先頭アドレス（SA）とブロックの大きさ（Z）の２つの値のペアだけから次の事が決定される。

（１）更に細かく分割する場合、それぞれの分割された小ブロックの先頭アドレスと大きさ。
（２）他のブロックと結合される場合、結合相手の先頭アドレスと大きさ。

以上の事から、２変数 <SA, Z> を一つのトークン内に収めることにより、このトークンはどの Allocator でも処理することが出来るようになる。

おわりに

Buddy システムには Binary buddy, Fibonnacci buddy その他いくつかのバリエーションがあるが、それぞれ特長を持っているので、システム固有の問題に合わせて選択されるべきであろう。

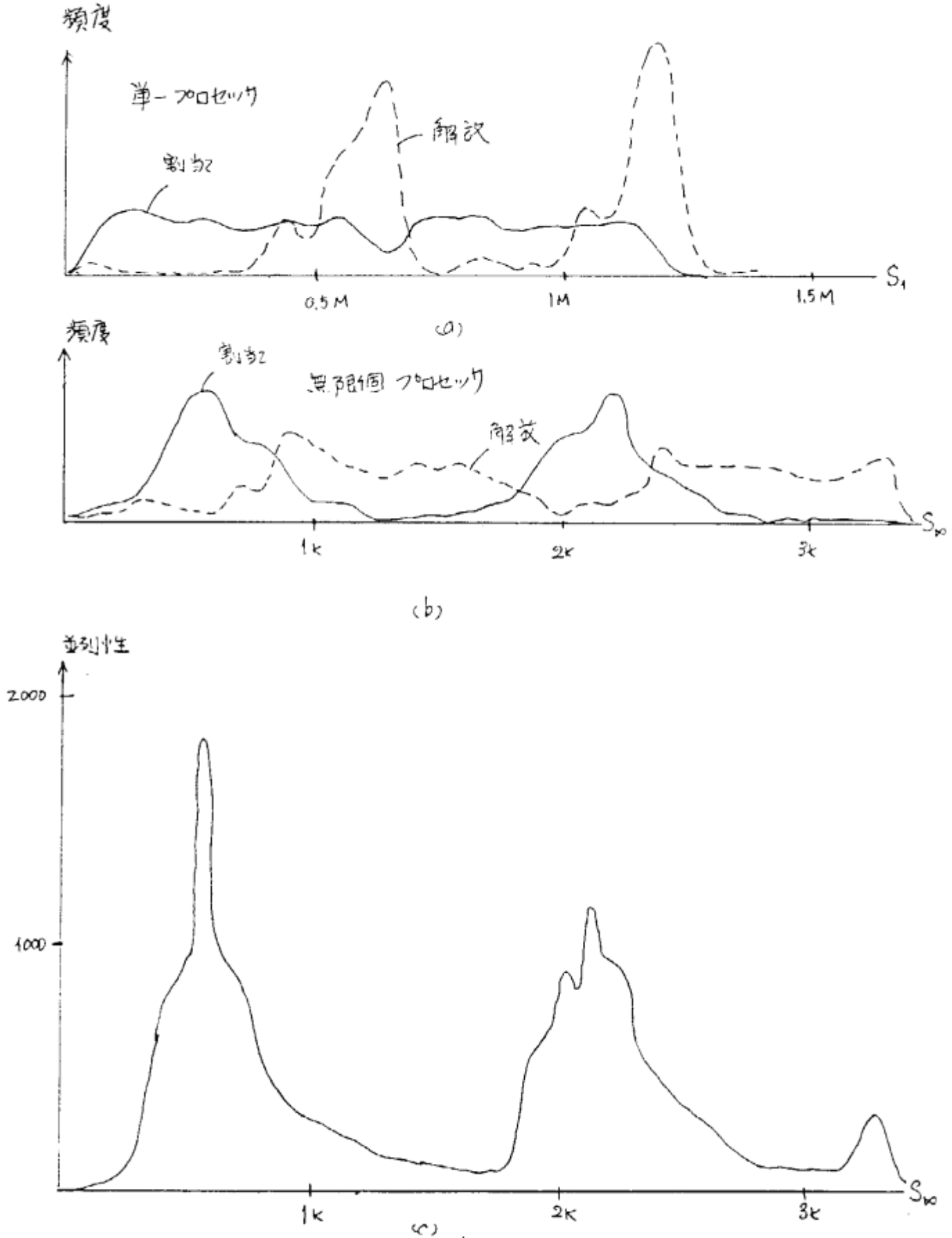


図1 静的割当てと解放の頻度

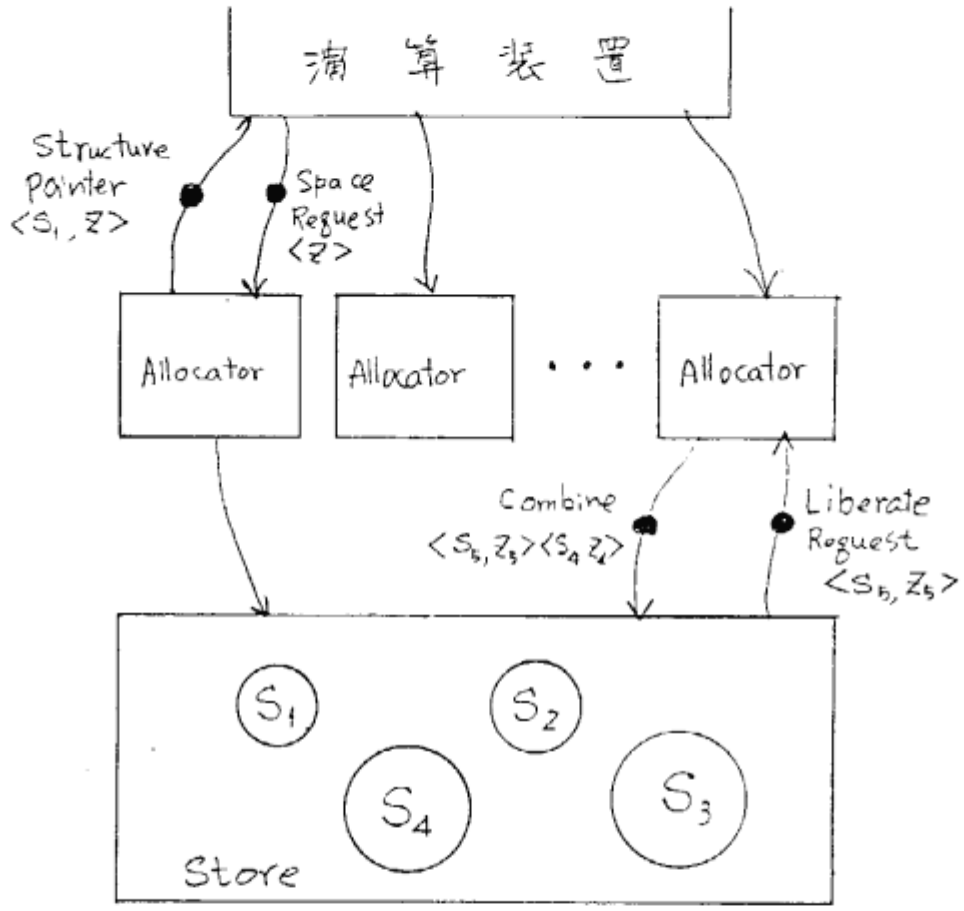


図2　並列記憶割当てシステムの一例

P^3 でほんとにいいの？

中島　浩
三菱電機（株）

1．期待されるプロセッサ像

ＰＩＭ／Ｍ－ＰＳＩの負荷分散のモデルとして，P^3 (Processing Power Plane) が提唱されている。P^3 が対象とする並列プロセッサは，以下の性質を持っていることが期待される。

(1) 2次元メッシュのように，2次元平面にマッピングできるような結合形態である。
(2) 充分多くの単位プロセッサから成っている。
(3) 全ての隣接プロセッサ間結合が均質である。

(2) 及び(3) は，プログラマにプロセッサ境界を意識させないために，必要な（期待される）性質である。

2．現実のプロセッサ像

(a) Ｍｕｌｔｉ－ＰＳＩ（64ＰＥ）
性質(1) と(3) は満たしているが，(2) は満たしていない。
(b) 中期ＰＩＭ（10^2 ＰＥ～）
クラスタ間結合が2次元メッシュであれば，性質(1) は満たすが，(2) と(3) は満たさない。
(c) 後期ＰＩＭ（10^3 ＰＥ～）
(d) 晩期ＰＩＭ，末期ＰＩＭ（10^4 ＰＥ～）
プロセッサの数を増やすためには，小さな空間に多くのプロセッサを詰め込まなければならない。しかし，いくつかのプロセッサを詰め込んだ空間（チップ，ボード）内部での結合と，空間相互の結合を均質にするのは，プロセッサを詰め込めば詰め込むほど困難になってくる（とりあえずつなぐだけでも大変になる）。従って，プロセッサ間の結合形態は，ハードウェア実装の階層に応じた，階層構造をとらざるを得ないであろう。よって，性質(1) ～(3) をすべて満たすのは極めて難しい。

3．期待されるモデル像

現実のハードウェアが，階層構造を持たざるを得ないとすると，それを無視したモデルでは，何かと不都合が多い。例えば，Ｍ－ＰＳＩではプロセッサの数があまりに少ないため，距離空間の不均質性（プロセッサ内外のギャップ）目立ってしまう。その結果，P^3 での「距離」と現実の「通信コスト」とのミスマッチが負荷分散に悪影響を及ぼすこととなろう。

さて，P^3 のバックグラウンドとして，以下のような動機があった。

①プロセッサが多くても適用可能
②プログラマはハードウェアを意識しない
③通信コストがプログラマに見える
④局所情報による動的負荷分散
⑤現実のプロセッサにマップできる

これらは全て正しくかつ好ましい動機であることに変わりはない。ただ，P^3 が動機⑤を満たしていない（またはP^3 をマップできるプロセッサを作れない）のが問題なのである。即ち，負荷分散モデルは，

(a) 階層構造のプロセッサにも適用できる
(b) プロセッサが少くてもうまく動く
(c) 容易に実現できる（P3 はあまり容易でない）

ことが強く望まれる。

4．とりあえずこんなモデル

P3 の問題は，均質でない空間を均質にみせようとしたことである。そこで，不均質な空間にプロセッサが離散的に存在する「重力場モデル」（または「宇宙モデル」）を提案する。

①最上位のネットワークを「宇宙」という。ネットワークが2次元メッシュであれば，宇宙は2次元であり，閉じている。

②最上位のノードを「銀河系」という。銀河系は均質の「エーテル」の中に浮かんでいる。

③ゴールはある「初速（方向と速さ）」を持って，銀河系内の天体より発射される。

④初速が銀河系の「脱出速度」に満たないときは，銀河系にとどまる。

⑤初速が脱出速度を越えているゴールは，銀河系間空間を飛行する。飛行中は，エーテルによって減速される。

⑥飛行中のゴールはその速度が立ち寄った銀河系の脱出速度に満たなくなったとき，銀河系に捕らえられる。

⑦負荷が重くなった銀河系は，その脱出速度を減らして（重力を減らして）ゴールの脱出を容易にし，またゴールの捕獲頻度を軽減する。

⑧銀河系の中には，いくつかの「太陽系」があって，固有の脱出速度を持っている。また，太陽系間空間には，「星間物質」があって，ゴールを減速する。

⑨太陽系の中には，「惑星系」があって，その中には，

・・・

（ここに宇宙の絵を書いてください）

並列アーキテクチャ研究に対する私の主張

東京大学　小池　汎平

実用的な計算機を作るには、実行するプログラムの動特性の把握が重要である。並列マシンの研究を進めていきながら常にこのことを痛感している。そこで、プログラムの動特性をアーキテクチャに反映させるために、我々が通常行なうのは、ベンチマークプログラムによるアーキテクチャのシミュレーション評価である。しかし、シミュレーション規模の制限、実用規模のプログラムがまだない等の理由で、ベンチマークプログラムとして通常用いられるのは、トイプログラム程度の単純な問題（要するに、エイトクイーン、リバース、クイックソート、良くてBUP）であることがほとんどであり、このことを嘆く声を聞くことが多い。では、実用的な規模のベンチマークプログラムが手に入ったとして、我々は、何をすればよいのだろうか？実行時間を測定して、一番速いアーキテクチャを見付ければよいのだろうか？これで、広範囲のプログラムに対して、実用となる並列マシンアーキテクチャが得られるだろうか？むしろ、あらゆるトイプログラムの動特性に熟知することが、現在、我々が採らなければならない道ではないだろうか？このことなくして、実用規模のプログラムの解析は、できないのではないか？

ここで、私が提案したいのは、実用規模のプログラムの動特性を、我々が通常ベンチマークプログラムとして用いている程度の単純なプログラムの動特性と、これらのプログラムの相互の干渉の特性とに分離して、把握する方法である。もし、実用規模プログラムの動特性が全くのランダムでないならば、このような分離が可能なはずである。このやり方が、並列アーキテクチャを研究していくにあたって、我々がとれる唯一の道ではないだろうか？

これから何をすべきか，どうありたいか

論理型言語で記述された応用問題のための並列計算機アーキテクチャについて

平木　敬　　（電子技術総合研究所）

１．はじめに

並列計算機システムを用いて応用問題を計算する動機は、並列計算機上での振る舞いが興味の対象である場合を除き計算速度の向上にある。従って、並列計算機システムの構造、動作は全て計算速度の向上に焦点を合わせたものでなければならない。並列計算機システムが明らかに逐次処理計算機より制限された汎用性を持つことを鑑みると、並列計算機システム設計のためにはより明確な処理形態の特定が必要である。

論理型言語は問題を記述する道具であり、それ自身の並列性や並列性出現の形態を論じることは不可能である。記述された応用問題こそ、並列アーキテクチャの設計の前提となる。しかし、論理型言語で記述された応用問題の並列処理形態は、何を解くかという根本問題を含め明確になっていない。このような事態は、実は新しいアーキテクチャ開発に常に付きまとう問題である。というのも、新しいアーキテクチャ上に適した問題は、それが実際に利用可能になって初めて開発可能となるからである。数値計算におけるパイプラインアルゴリズム、シストリックアルゴリズム、メッシュ結合アルゴリズムなどは、その好例である。

このように困難な状況での並列推論計算機アーキテクチャ開発は、逐次型推論計算機のアーキテクチャとともに、既に実用化されている数値計算用並列計算機のアーキテクチャが基礎となろう。本発表では数値計算用並列計算機との比較を軸として、並列推論計算機アーキテクチャの持つべき条件を考察する。　まず、議論の出発点として次の項目を前提とする：

前提１：並列計算機上での数値処理の並列実行は、推論処理の並列実行より容易である。すなわち、

① 数値処理はより大きい平均並列度を持ち、
② 並列度の時間的変化が少なく、
③ 並列度を維持するために必要な横方向のデータ通信が少なく、
④ 構造データ内並列性等、利用が容易な並列性を持つ。

ただし、別サンプルのデータを処理することに代表される自明の並列性は、ここでは問題にしない。

前提２：アーキテクチャ並列化の目的は絶対的な実行速度の追求であり、コストパーフォーマンスの向上でない。つまり、問題とする実行性能の範囲は、高速の逐次計算機では実現不可能な領域である。

以上の前提に基づき、現在数値計算向け並列計算機の設計上大きな問題となる (1)メモリの共有と構造体処理の問題、 (2)データ転送結合網の構成と能力の問題、 (3)プログラム要素の同期実現の問題について定性的考察を行ない、最後に並列推論計算機アーキテクチャの一例を検討する。

２．共有メモリと構造体処理

数値計算における並列性の根源が、構造体に内在する処理の並列性にあることは広く認知されている。従って、演算の対象となる構造体の要素を如何に高速に演算器に送り込み、結果を高速に構造体として送り込むかが高速化の鍵となる。これを実現するために、構造体からの要素の取り出し、要素の転送、演算処理、新しい構造体の生成が一連のハードウェア動作として実現する必要がある。多くの数値計算用並列計算機システムでは、各処理装置間での円滑な構造体の転送、複写をアーキテクチャとして実現している（CRAY-XMP,CRAY-1,ETA-10,SIGMA-1などは各々そのメカニズムを備えている）。

一方、推論処理、特に記号的演算処理においても、スタック処理を含めた構造体処理が中心的役割を果たしている。しかしながら、①構造体の構造が複雑である、②数字による単純な要素指定が不可能である、③ポインタ構造で実現した場合には処理が本質的に逐次的となる、④構造体の平均的大きさが小さい、など数値処理の場合と比較して高速化が困難となる要因が多い。特に並列環境でポインタ構造の命令による逐次アクセスを行なった場合には、アクセス遅延による処理速度の低下が顕著であり、並列化のメリットが失われる。従って、(1)高バンド幅の構造体転送能力、(2)ポインタ構造を介在しない構造体転送方式、(3)サイズに対する立ち上がりの速い構造体処理、(4)記号的連想アクセスによる要素指定能力、が強く要請される。

(1)については、数値演算では演算能力以上の高バンド幅が必要と推定される。一般に数値処理計算機では計算能力の３倍以上のバンド幅が必要であると考えられている 。このことは、次節で検討する結合網の能力に対する強い要求である。一方(2)、(3)は互いに関連し、複雑な構造体の合理的転送方式が要求されている。しかし、その研究はその端緒についたばかりであり、更に多くの研究を要する［１］。(4)についても、インデクスの内容により様々な処理方式が検討される。記号列その物をインデクスとする場合、構造も含めた構造体の内容全体をインデクスとする場合とも、ハッシング、特に仮想鍵を用いたハードウェアハッシング方式は非常に有効な方式である［２］。ハードウェアハッシングを採用することによるハードウェア量の増加は僅かであり、アーキテクチャへの組み込みを阻害する要因は、設計が面倒臭いこと以外には考えられない。

３．データ転送と結合網

結合網に関しては、その方式、トポロジーとともに、結合網を挿入する位置、データ転送方式が問題となる。トポロジーは実現の難易と絡み多くの方式が提案されている。更に、処理すべき応用問題の性質が最も強く反映する所でもある。数値計算においては、メッシュ結合（FEM,PAX），多段結合網（GF-11,RP-3,PASM），クロスバースイッチ（SIGMA-1,CEDER）、ハイパーキューブ（NCUBE,FPS-T series,COSMIC CUBE）が市民権を得、各々対応するアルゴリズムが研究されている。メッシュ結合の場合、メッシュ結合アルゴリズムを実行することが強く要請される。メッシュ結合推論アルゴリズムに対する見通しが無い限り、検討の対象とはなり得ない。多段結合網では、データ転送が共有メモリモデルに従う限り、データ転送の衝突によるホットスポットの危険性を回避することが困難であり、危険な選択肢である。クロスバースイッチも、結合数が多く、何らかの階層構造が必要な場合は、同様の危険にさらされる。一方、ハイパーキューブはホットスポットの危険は少ないが、回線結合網としない限りデッドロックを回避するメカニズムを備える必要があり、構成要素が低速かつ複雑化する。回線結合網では、アクセス時間の損失が堪え難い。

このように、決定的結合網が無い状況では、設計者の嗜好により決定せざるを得ないが、すくなくとも数値計算の場合より、バンド幅、結合網遅延の両面でより強力なものが必要であることは疑いない。また、共有データに対するアクセスも、並列性出現の根源から数値計算の場合より激しいと考えられる。従ってホットスポットの発生に対する対策は不可欠であろう。私見では、共有メモリによるデータ転送を行なう限り、多少複雑でも、デッドロック対策を施したハイパーキューブが好ましく、すくなくとも処理装置間データ転送がパケット転送方式（データフローアーキテクチャに基づく）にする場合は、均一でない階層ネットワーク（ＳＩＧＭＡ－１［3］など）が好ましい。

結合網を挿入する位置は、各処理装置が階層記憶を採用するという前提のもとで、図１（ａ）～（ｄ）の４種類が可能である。（ａ）では処理装置間で直接転送する方式であり、処理装置がデータフロー方式でない限り現実的でない。（ｂ）はアクセス時間の増大という点が解決すれば、自然にコヒーレントとなる階層記憶であり、主記憶との転送バンド幅にも問題がない。（ｃ）ではキャッシュメモリのコヒーレンスが問題となり、バス以外のネットワークで実現することが困難である。しかしながら、バス結合で、必要なバンド幅を得ることは、非常に困難である（数値計算用並列計算機においても、２台以上で用いられた成功例を知らない）。推論計算機ではバンド幅が演算能力の４倍必要であると仮定すると、８台で２４倍、要素処理装置が１０Ｍ１ＰＳ程度として９６０ＭＢ／ｓの能力が必要であり、１０ＭＩＰＳを実現する技術ではまったく不可能である。（ｄ）では更にコヒーレンスの問題が深刻であり、プログラムが直接支配する入出力的転送だけが可能であろう。

コヒーレンス問題の重要性を鑑みると、（ａ）または（ｂ）、またはその双方が望ましい。

４．同期と分散

前節までの考察では、処理装置のアーキテクチャがフォンノイマン型であるか、データフローであるかについて、基本的な差異はなかった。しかしながら、同期と分散のコスト

、特に並列環境下におけるコストを考慮すると要素処理装置をフォンノイマン型で構成するメリットは殆どない。同期に関するデータフローアーキテクチャの利点はすでに多くの文献で明らかであるため、ここでは要点を列挙することに止める。

① データ依存関係に基づく同期関係が時間コストなしで実現する。
② データ依存関係に基づかない同期関係、全順序関係は、同期命令列で実現する。
③ 遠隔のメモリアクセスに付随する待ち時間が無駄時間とならない。
④ ネットワークにおける衝突に起因する遅延も、並列性がある限り無駄時間とならない。
⑤ 関数レベル等高レベルの同期と、演算に内在する低レベルの同期が同一メカニズムで実現する。

数値計算プログラムの特徴として、一つのプログラム単位が大きく、実行時の同期を必要とする場面は、遠隔の構造体アクセスを除き、プログラム単位のレベルか、条件判定と代入関係が実行時にしか決定しない場合（所謂ベクトル化の不可能な場合）に限定されている。それにもかかわらず、同期の問題は性能を決定する重要な要素となっている。従って、プログラム単位の大きさが小さく、条件判定の比重が高く、実行時に同期関係の決定する要素が多いと推測される推論処理または記号処理における同期コストの重要性は、数値計算プログラムより大きいことは疑いの無いことであろう。

１００台規模以上のフォンノイマンアーキテクチャに基づく数値処理用並列計算機システムが、隣接結合アルゴリズムを用いたもの以外成功していないことから、大規模並列推論計算機システムにデータフローアーキテクチャを導入することは必須であろう。

５．データフロー・オーバーヘッドについて

とはいうものの、データフローアーキテクチャを採用すると、データフローオーバーヘッドが恐くて、逐次性能が低下するのではないかと懸念する向きもある。本節では、データフローオーバーヘッドの原因を列挙し、逐次性能との関連を考察する。

（１）データのコピーに基づくオーバーヘッド
（２）サーキュラーパイプラインに基づく逐次性能の低下
（３）レジスタによる最適化をしないことによる逐次性能の低下
（４）実行順序がデータ依存関係の半順序だけに基づくことによるオーバーヘッド
（５）ごみトークンの回収に関係したオーバーヘッド

ＳＩＧＭＡ－１における経験によると、（１）、（４）、（５）は単に杞憂である。すなわち、フォンノイマン計算機におけるフォンノイマン・オーバーヘッドと同程度と考えられる。（２）はパイプライン設計の単純化の結果引き起こされるものである。すなわち、既存のパイプライン方式フォンノイマン計算機と同程度のパイプライン段間干渉を取り入れることによって解消できる。レジスタとデータフロー計算機との相性は、ハードウェア

量と速度のバランスで決定する。データフロー計算機でも、同時に存在することが許される全ての並列アクティビティに１組のレジスタを付加することにより、プログラムからレジスタを使用することが可能となる。ただし、レジスタ値を更新する操作は、全順序が保証されている状況下でしか使用できない。しかし、全レジスタを備えることはハードウェア量から許されない。

レジスタを最も高速な記憶階層と解釈すると、アクセスの局所性は非常に良いと考えられる。従って、キャッシュメモリに代表される仮想化を行なうことにより（FLATS のレジスタが好例である）データフローアーキテクチャとレジスタの両立は可能である。

このように、ハードウェア量が若干増加する点を除けば、逐次性能に於いても何ら劣る点がないと推定される。

６．私ならこうする－－終わりにかえて

アーキテクチャを構成する諸要素を、数値計算の場合と比較対照して検討を行なった。本発表のテーマは、数年以内に完成する１００台規模の並列推論計算機のアーキテクチャに関する私見を述べることである。アーキテクチャの開発を短期的展望で考察する場合には、全く新しい要素の導入は困難である。従って、見通しの立っているアーキテクチャ技術を総合するにより設計を行なうことが必要である。具体的には、

（１）データフローアーキテクチャの採用
（２）共有メモリを用いた構造体の実現
（３）仮想化したレジスタの導入
（４）ハードウェアハッシングによる記号的インデクシングのサポート
（５）クロスバースイッチを中心とした高バンド幅結合網

を基本的方針とする。全体ブロック図を図２に示す。図中結合網１、２はデータ幅８０ビットの回線制御クロスバースイッチで、ポートあたり２００ＭＢ／ｓ程度のの転送能力を持つもの、結合網２はデータ幅４０ビットのパケット交換結合網で５０ＭＢ／ｓ程度の転送能力を持つものである。これは先に述べたバンド幅に対する条件を満たす。共有キャッシュメモリとＰＥを結合網を介して結合する結果、コヒーレンス問題を忘れることが可能である。結合網がキャッシュメモリとの間に入るために引き起こされる遅延時間の増大は、キャッシュのアドレステーブルをＰＥ側に持たせることにより回避可能であろう。

要素処理装置（ＰＥ）は図３に示すように命令、命令発火データ、レジスタ、局所データを保持するローカルなキャッシュメモリ（ＬＣＭ）、命令列生成ユニット（ＩＧＵ）、レジスタ・インデクスユニット（ＲＩＵ）、構造体処理ユニット（ＳＵ）、演算処理ユニット（ＥＵ）で構成される。

命令列生成ユニットは、データフロー命令発火条件を検出し、実行可能命令を、他のユニットに送付する。データの内容が命令発火条件を決定する条件分岐等非パイプライン命令を除いては、命令列生成ユニットだけで多重環境下における実行可能命令列を生成できることは、フォンノイマン型パイプライン計算機と同様である。ローカル・キャッシュメ

モリは、２入力以上の命令発火を司るハッシングメモリ、命令を格納するメモリ、レジスタおよび局所環境実現のために使用する。この機能を実現するために、４バンクの並列連鎖ハッシングメモリ［４］を使用する。このキャッシュメモリは局所的に使用されることを想定して、バンド幅の狭い結合網で主記憶と結合し、コヒーレンス保持は行なわない。

レジスタ・インデクスユニットは局所的なデータのアクセス管理を行なう。演算処理ユニットは構造体操作を伴わない命令を処理する。構造体処理ユニットは結合網を会して共有キャッシュメモリをアクセスし、構造体に対する操作を実現する。共有キャッシュメモリでは、並列オープンハッシングメモリ操作および仮想鍵方式ハッシングメモリ操作を実現し、アトムおよび構造体に対する連想操作を実現する。

共有キャッシュメモリはクロスバースイッチで結合した通常のキャッシュである。だだし、アドレステーブルを処理装置側に分散配置し、高速化を図っている。主記憶装置は、単一処理汎用計算機のものと同一である。

このシステムの性能、特に並列動作性能、シミュレーションを行なわない限り推定不可能である。ただし、要素処理装置単体の性能を非常に大まかに推定すると、共有キャッシュメモリのアクセス時間が１２０ナノセカンド程度であり、専用アーキテクチャを採用することによる加速率を５倍として１ＭＬＩＳＰ程度の性能も不可能ではないと思われる（ＦＬＡＴＳにおけるＬｉｓｐの加速率は、８倍程度である）。従って、並列推論システム全体で１２８ＭＬＩＰＳのピーク性能を持つことになり、平均稼動率が多少悪くとも、最高速の逐次推論計算機より高速の性能を得て並列化することの本来の意義を発揮できるであろう。

参考文献

［１］平木、関口、島田：科学技術計算用データ駆動計算機ＳＩＧＭＡ－１における構造体処理の高速化、第２回データフローワークショップ発表予定、１９８７。

［２］平木、後藤：数式処理計算機ＦＬＡＴＳのアーキテクチャ、情報処理学会論文誌、第２７巻、第１号、pp.81-89、１９８６。

［３］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, IEEE, pp.524-531, 1984.

［４］Hiraki,K. Nishida,K. and Shimada,T.,"Evaluation of Associative Memory Usinf Parallel Chained Hashing," IEEE Trans. on Comput., Vol.C33, No.9, pp.851-855, 1984.

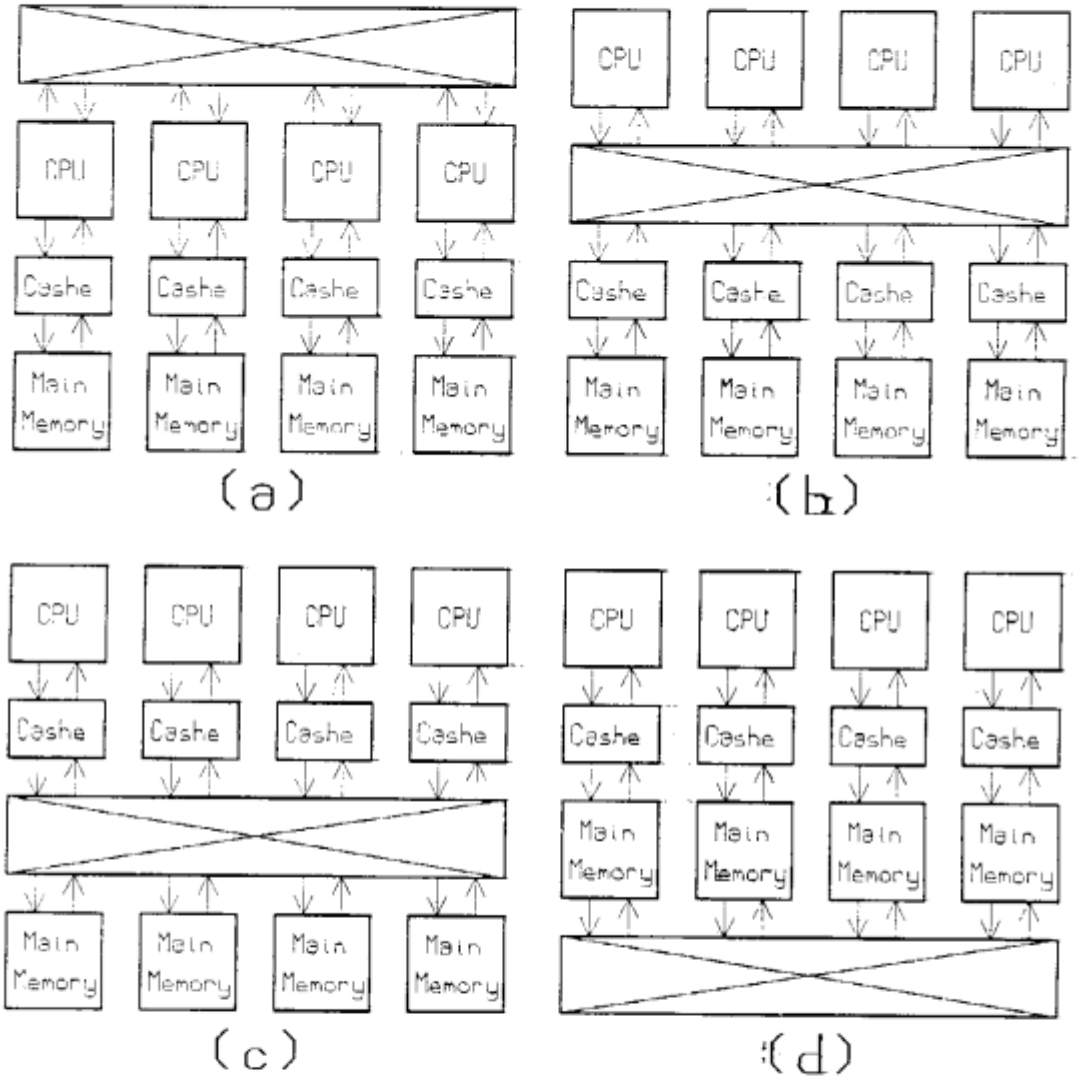


図１．　結合網の接続形態

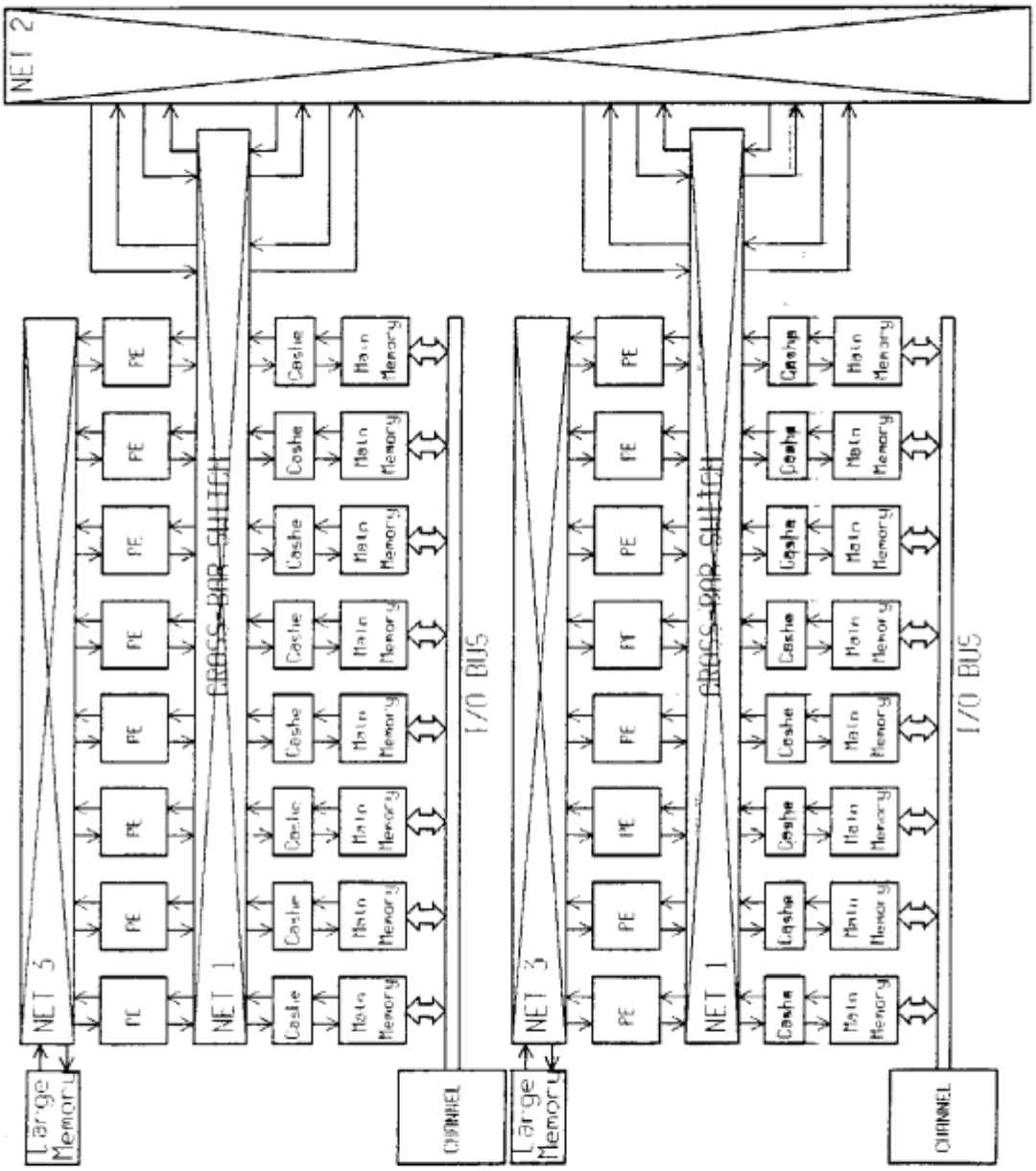


図2．　並列推論計算機の全体構成

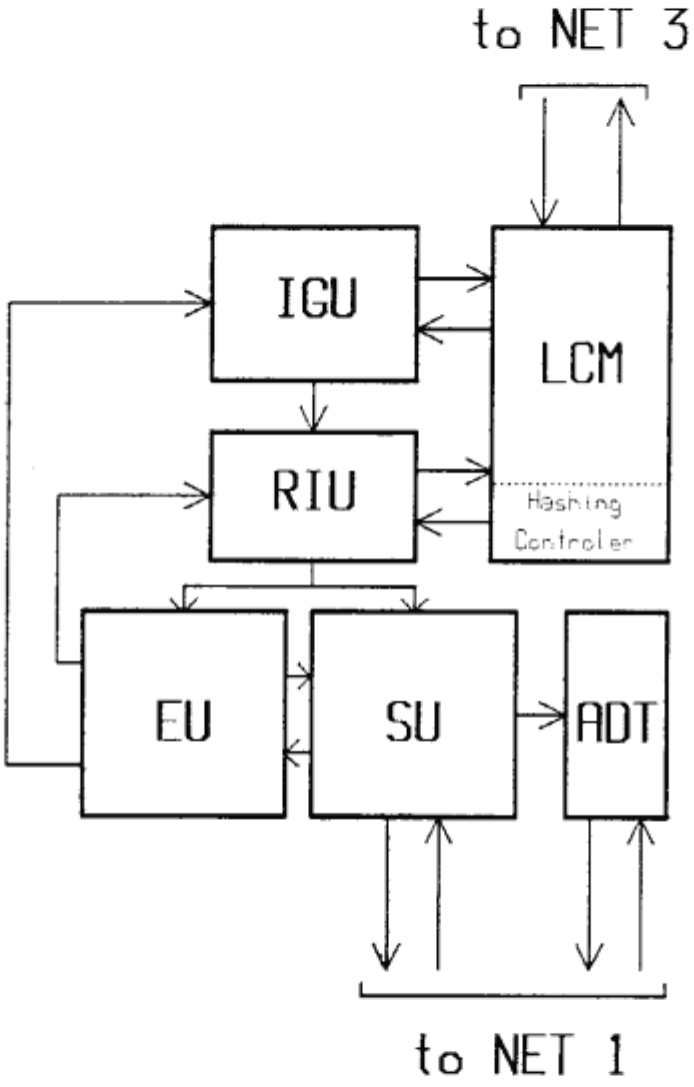
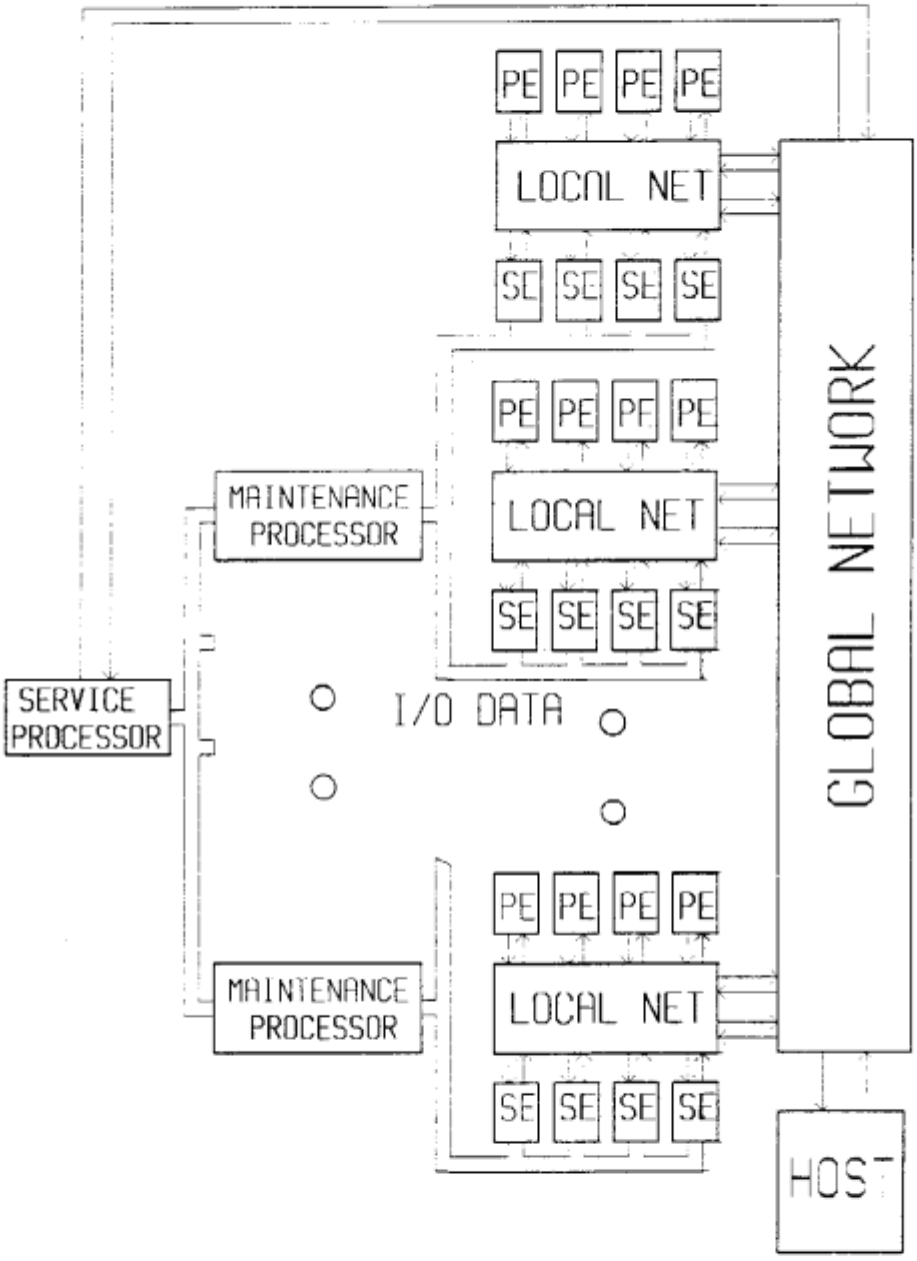
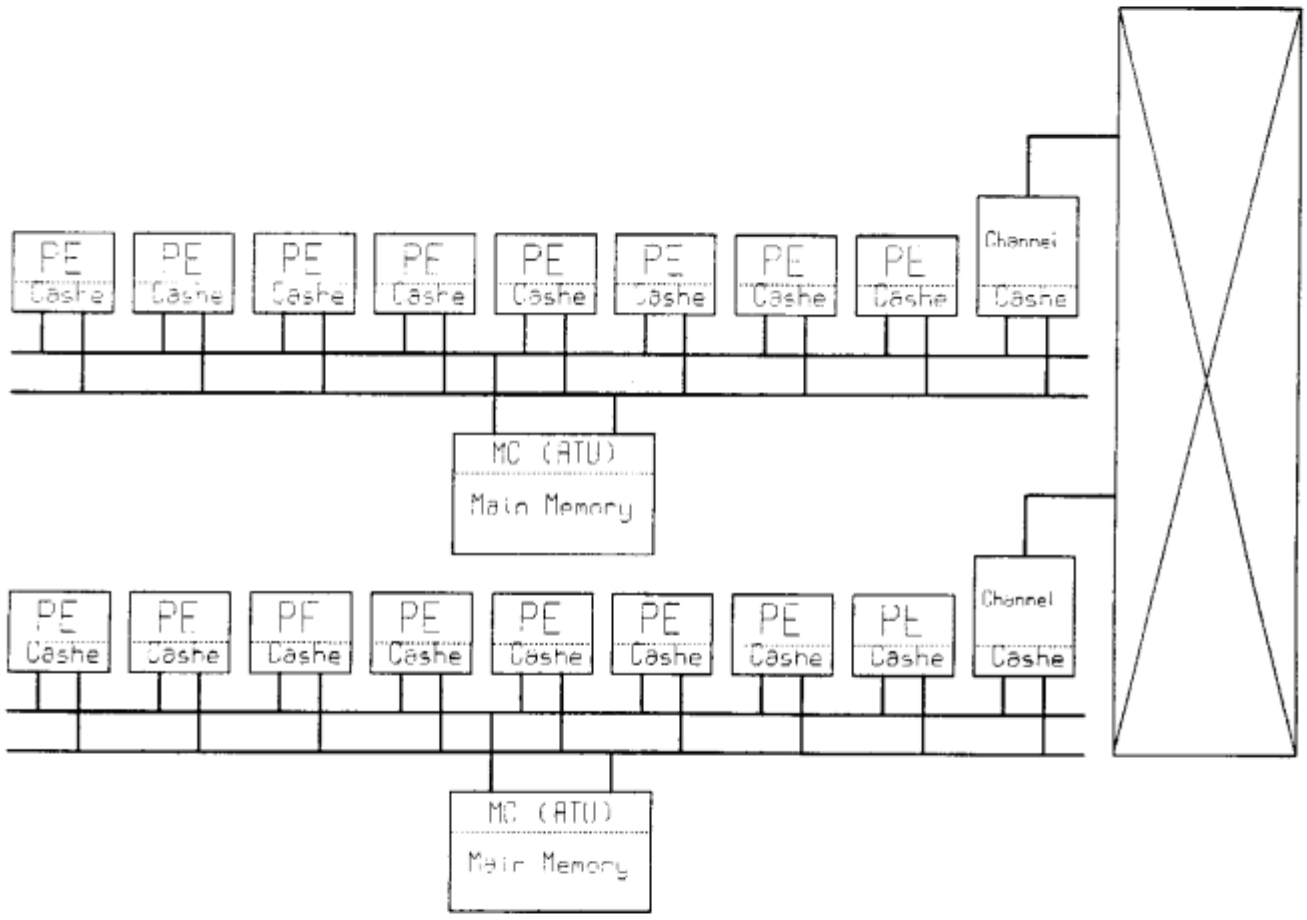


図3．　要素処理装置の内部構成



参考図　ＳＩＧＭＡ－１の全体構成



参考図　中期ＰＩＭの全体構成

アーキテクチャ研究におけるPIM開発プロジェクトの役割

柴山 潔 (京都大学)

1. ICOTの開発しているマシンと京都大学-KPR[2]との比較

- アーキテクチャ(表1参照)[1]
- 実行モデルにおける粒度

  株分け ≧ 中期PIM(マルチPSI) ≧ KPR ≧ PIM-D、PIM-R

  ただし、中期PIMはプログラマの指示による粒度の変更も可能。原則として、1個のPE内でゴールの逐次的処理も行う。
- プロジェクト規模

  ICOT……通産省、予算規模-大、人員-多数、しがらみ-種々有り、商用?

  KPR……文部省、予算規模-ICOTに比して小、人員-少数、自由度-大、教育的効果-要

  いずれも「夢」がある(あった)はず? 同じ夢だろうか?
- 物理的粒度

  ICOTの中期PIMのPEのハードウェア構成(ソフトウェアとハードウェアのトレードオフ)が現在でも未定?

2. アーキテクチャ研究に対する私見

- 一人で地道に(あるいは華やかに)やるのか、チームで意見を闘わすのか?
- (花形分野の)専用機を目指して厳しく競争するのか、(流通する必要のない)汎用機で独りよがりの悦にいるのか? アーキテクチャのブレーク-スルーは汎用機の開発から生まれる?!
- アーキテクトの学問的基盤は何? アーキテクトの素性は、ソフトウェア屋さん(応用屋さんを含む)?、ハードウェア屋さん(デバイス屋さんを含む)?、やっぱりトレードオフ(システム)屋さん?!
- 今は道具や手法の重要な時代、すなわちASICの時代であり、(高速)シミュレーションやテストベッドの価値を見直し、これらを充分に利用しなければ、時流に乗り遅れる?! 例えば、「マイクロプログラム技術」は生き残るのか?
- デバイスやソフトウェア技術、そして応用分野に刺激を与えるアーキテクチャのパラダイムを提示しなければアーキテクトでない?! 単なるハードウェアとソフトウェアのインタフェースとしてのアーキテクチャではなく、応用と計算機システムとのインタフェースとしてのシステム-アーキテクチャの構築を目指そう?!

表1　論理型言語向き並列マシン(ICOTマシンとKPRのアーキテクチャ上の比較)

マシン名	株分け方式	PIM-D	PIM-R	マルチPSI	中期PIM	KPR
現況	・16台PE構成のハードウェア・シミュレータ完成	・ハードウェア・シミュレータ(4台PE,4台構造メモリ)稼動中 ・16台PE,15台メモリまで拡張予定	・ハードウェア・シミュレータ(MC68000・2MBメモリ付き16台)稼動中 ・各メモリ増設(8MB)中 ・100台PE構成想定	・PE(PSI)は稼動中 ・1986・6台PE構成稼動予定(最大64台PE想定)	・シミュレーション中 ・1987稼動予定(100台PE想定)	・プロトタイプ(5台PE)設計中 ・1000台PE構成を想定
対象記号処理用言語	Prolog(純粋Prolog+cut機能)	KL1(GHC),OR並列型言語	Prolog,CP,GHC	KL1(GHC)	KL1(GHC)	並列論理型言語(純粋Prolog+並列型Prologの基本機能)
応用分野			自然言語処理,レイアウトCAD,論理DA	並列推論ソフトウェア(言語,OS,応用)の開発	後期PIMハードウェアやソフトウェアの開発	論理プログラミング
論理的粒度	インタプリタ	データフロー・グラフ(単一化や非決定性を制御する基本ノード)のインタプリタ	リテラル(ゴール)	ゴール	ゴール	ゴール(プロセス),デマンドやイベント,PE内の単一化
物理的粒度	68010(10MHz,2MBメモリ)	PE(基本演算ユニット2個,パケット・キュー,命令制御,パイプライン構造),構造メモリ	推論モジュール(プロセス・プール,単一化ユニット),共有構造メモリ(ground instance格納用)	PSI(逐次型Prologマシン)	30~40ビットCPU(専用?)	ヘテロジニアスPE(ストリーム並列処理用,OR並列処理用,データベース処理用)
ネットワーク	リング(12ビット並列,要求/ステータス用),多段網(8ビット並列DMA,仕事のブロードキャスト機能)	階層型バス	格子	格子(10ビット並列)	階層型(クラスタ内:共有バス,クラスタ間:疎結合?)	多重2進木(40/40ビット並列)
並列処理方式	OR	OR,AND,引数間	OR(Prolog) AND(CP)	AND	AND	OR, ストリーム
その外のアーキテクチャ上の特徴	・分散メモリ型マルチPrologマシン・システム	・データフロー・グラフにコンパイル後実行 ・最小クラスタ:4台PE,4台構造メモリ	・探索戦略の変更可 ・ネットワーク・ノードが負荷分散を制御可 ・逆順コンパクションと呼ぶリテラル格納方式 ・プロセス・プール内にメッセージ・ボード(並行プロセス間チャネル用分散化共有メモリ)	・分散メモリ型マルチPrologマシン・システム ・KL-1Bにコンパイル後実行 ・可変長パケット転送	・共有メモリ型マルチPrologマシン・システム ・KL-1Bにコンパイル後実行 ・並列キャッシュ機構 ・ハードウェア・ロック機構(分散制御) ・アドレス変換機構付き共有メモリ ・クラスタ:10PE程度	・PE内の単一化では最大4個の単一化を並列実行可 ・PE対は共有メモリ(レジスタ)型密結合
共同開発機関	富士通	沖電気工業	日立製作所	(少数?)	(多数)	(なし)

3.「AIマシン」って何? (AIマシン = parallelism + α?)

- 種々の定義
  - (0) = 汎用機 + Prolog
    - →(まじめに主張する人もいるが.....)
  - (1) = 汎用機(せいぜいRISC) + マルチ言語パラダイム(FP,LP,OOP)
    - →良心的な商用機(逐次型)
  - (2) = 並列マシン(マルチ汎用機) + マルチ言語パラダイム(FP,LP,OOP,並列言語)
    - →PIM一般
  - (3) = 並列マシン(専用機) + マルチ言語パラダイム(FP,LP,OOP,並列言語)
    - →ICOTの中期PIM?
  - (4) = 並列マシン + PS/意味ネットワーク/知識ベース
    - →ルール指向マシン、KBMなど
- 第5世代計算機(≈ 並列推論マシン) ≈ AIマシン だろうか?
- AI向き言語 = 探索(推論)Parallelismの実現 + control(制限)機能
- AI向き言語パラダイム
  - (a) FP(関数型言語) → Lisp??
  - (b) LP(論理型言語) → Prolog
  - (c) OOP(オブジェクト指向型言語) → Smalltalk-80
- マルチ言語パラダイムの実現方法には、ソフトウェアのみで対処する場合と、ファームウェアも利用する場合とがある。
- アーキテクチャ構築技術としての"+α"
  - i) マイクロプログラム技術
  - ii) スタック
  - iii) 高機能メモリ(アドレス変換器)
  - iv) 通信(同期)制御

4. 論理型言語向きアーキテクチャ

- プログラミング言語としての機能差が応用分野に違いを生む?! (図1参照)
- アーキテクチャと対象言語との関係 (図1参照)
  - ◇逐次マシン上に並列型言語の処理をインプリメントできる。一旦コンパイルする方法もある。
  - ◇並列マシン上で逐次型言語の処理をすることも考えられる。
- AND/OR並列型(論理型)言語って何?
  - [A] AND並列型言語 → AND並列、OR選択(コミット選択)
  - [B] OR並列型言語 → AND逐次、OR並列(このような言語が実用的に意味を持つのか?)
- アーキテクチャとしてサポートが必要な機能
  - [A] AND並列型言語 → (AND並列ゴール間の)共有変数の無矛盾性検査、OR並列性(parallelism)を制限(control)する「コミット」に伴うプロセス間通信と同期
  - [B] OR並列型言語 → データベース処理
  - [C] 純粋Prolog → AND/OR論理の保持

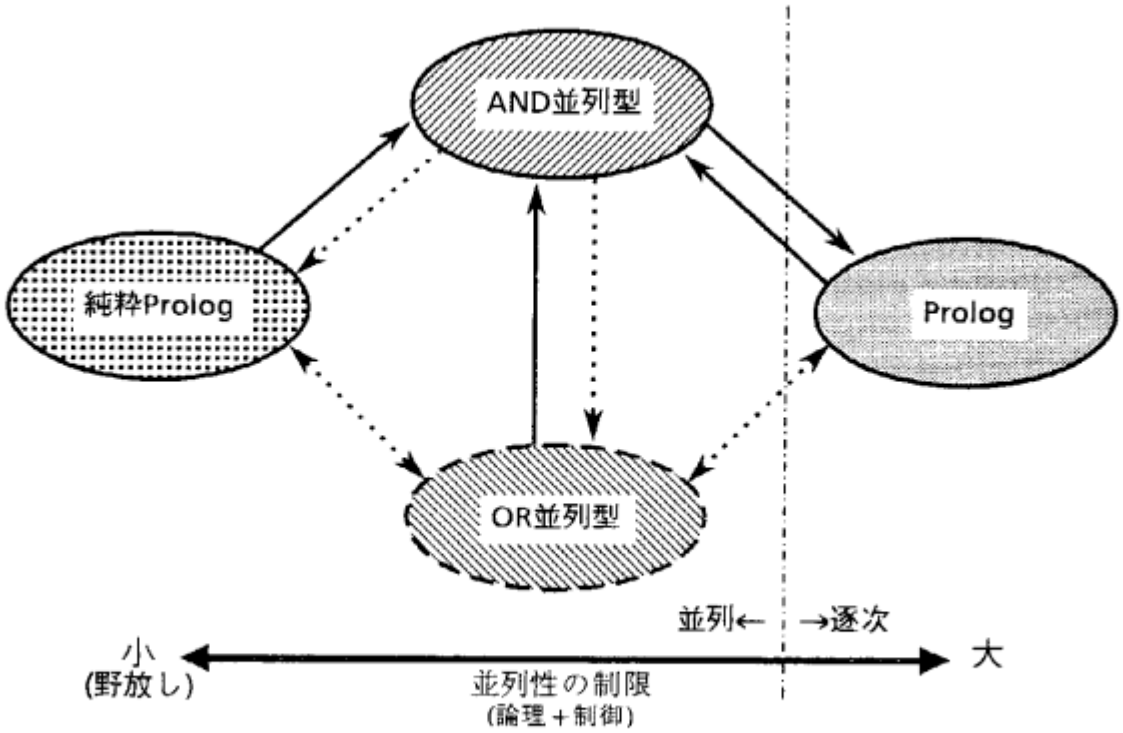


図1　論理型言語パラダイムの関係

●論理的並列性を制限することの意味

◇AIの本質的意味の実現

◇ハードウェア資源(有限)による制約

参考文献

[1] 柴山潔: "記号処理マシン", 情報処理, Vol.28, No.1, 情報処理学会, pp.27-46 (1987).

[2] 柴山ほか: "論理型言語向き並列計算機KPRのアーキテクチャ", *Proc. of the Logic Programming Conf. '87*, 9.1, ICOT, pp.183-192 (1987).

デバイス技術の発展／限界と並列マシンアーキテクチャ
－並列マシンアーキテクチャに新たな枠組を！－

NTT 厚木研究所 小倉 武

1．はじめに

『①並列マシンアーキテクチャからの要請によるデバイス技術の発展　及び
②デバイス技術の発展と限界を見越した並列マシンアーキテクチャ研究があるべきだ』
という考えを出発点として私見を述べる．

2．私見－デバイス技術の発展と限界－

①CMOS，バイポーラともにVLSIは，発展途上．しかし，その頂上は近い．
②汎用超大型マシンと異なり，並列推論マシンの場合，10年先を考えたとしても，主役のデバイス技術は，より発展したCMOS技術．
③CMOS技術以外にもシステムとしてのボトルネックを解消する特定領域向きの夢候補はいっぱいある．
④しかし，これらは，あくまでも候補．デバイス技術の発展と限界を見越した革新的並列マシンアーキテクチャ（新たな枠組）の構築が必要不可欠．

表1　デバイス技術の発展と限界のスコープ

デバイス技術	コ　メ　ン　ト	参考図
CMOS スケーリング	・論理素子，メモリの高集積化，高速化は進展． ・VLSIとしては， 伝搬遅延時間 100pS，数百万素子程度が限界． ・ゲートアレイ，コアマクロ等の利用により， VLSI開発の容易化が一層進展．	図1 図2
3-D VLSI	・近未来－連想メモリ等の高機能メモリVLSI． ・中未来－光技術との結合． 縦方向配線によるネットワーク実現． ・遠未来－システムの1チップ化．	図3 図4
光技術	・近未来－VLSI間のファイバ接続． ・中未来－VLSI間の導波路接続． 3-D VLSIとの結合． ・遠未来－VLSI間の空間伝搬による超並列接続． neural networksとの結合．	図5
高温超電導	・VLSI上配線－低抵抗化，多層化． ・新原理デバイス(?) ・常温超電導は魅力的．	
neural networks	・集中制御（アルゴリズム）が不要な システム制御のモデル(?)	

3．考察 － デバイス技術の発展/限界と並列マシンアーキテクチャ －

3.1 前提－並列マシンアーキテクチャー

①並列マシンアーキテクチャとは，複数のプロセス担体が，統一目的に向かって，プロセス担体内での情報処理とプロセス担体間の情報授受を行なうマシンアーキテクチャ．

②並列マシンアーキテクチャの性能規定要因

＝プロセス担体の情報生成能力

＆プロセス担体間の情報転送能力

＆制御オーバーヘッド．

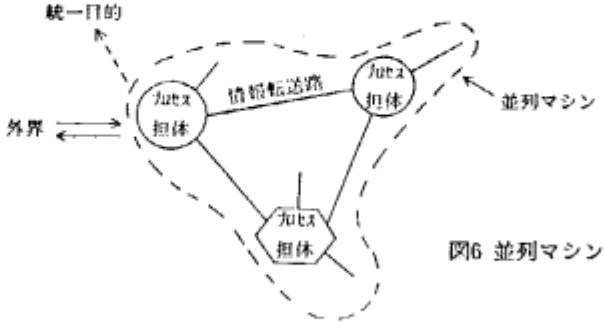


図6 並列マシンアーキテクチャ

3.2 考察

①プロセス担体の情報生成能力は，CMOSの微細化に伴ない，それなりに向上する．

理由：CMOSのスケーリングパラメータをK（K＞1）としたとき，

論理ゲートの伝搬遅延時間は，1／Kに向上する．(図1)

②しかし，プロセス担体間の情報転送能力は，CMOSを微細化しても向上しない．

理由：情報転送路のRC時定数は，一定．

⇒ 情報転送路がボトルネックとなる可能性大 ⇒ デバイス／アーキテクチャのブレークスルーが必要

ⓐデバイスのブレークスルー候補：

●情報転送能力向上に寄与するデバイス技術

⇒・3-D VLSI → 多機能/高機能VLSI(図3,図4)

縦方向配線の有効利用によるネットワーク実現

・光技術 → ファイバ/導波路/空間接続による接続数/速度制限の排除(図5)

・高温超電導 → 低抵抗配線，薄膜化による多層配線

⇒ VLSI技術と同様な汎用技術になりうるか？

●neural networksを実現するデバイス技術

⇒ ？

ⓑアーキテクチャのブレークスルー候補：

●情報転送が少ないアーキテクチャ

⇒value passingレベルの軽負荷情報転送アーキテクチャ

●集中制御が不要なアーキテクチャ

⇒局所制御による全体の統一動作が理想

←neural networks/神経情報処理様式[3]からのアナロジ

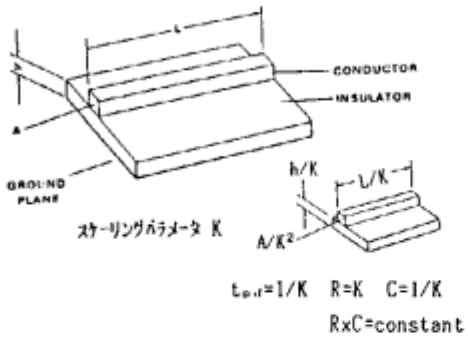


図1 スケーリング則

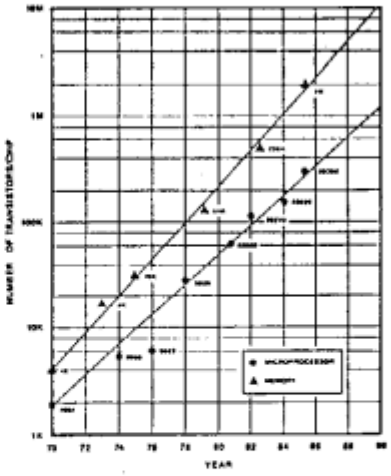


図2 VLSI上の素子数の年次変化[1]

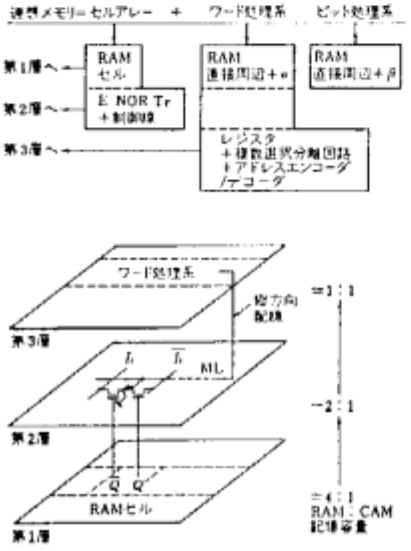


図3 連想メモリの3-D VLSIでの実現

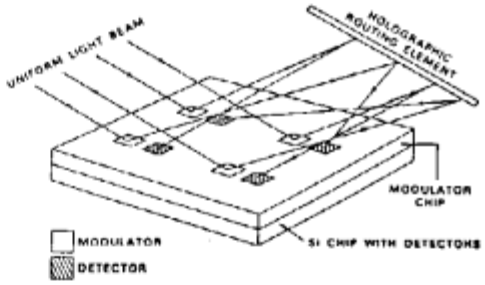
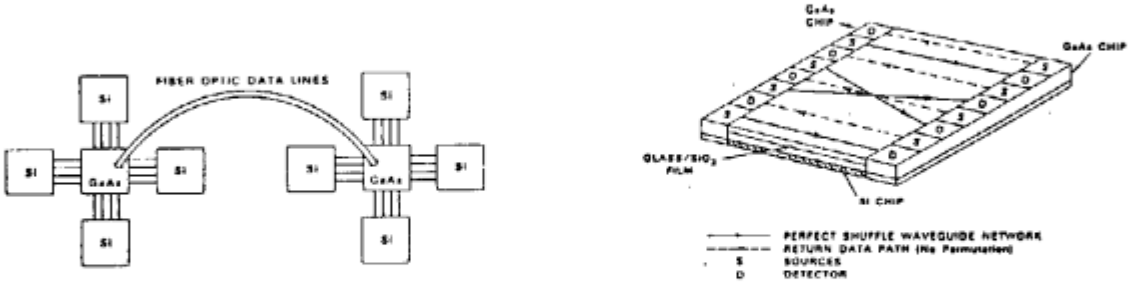


図4 3-D VLSIと光技術との結合[2]

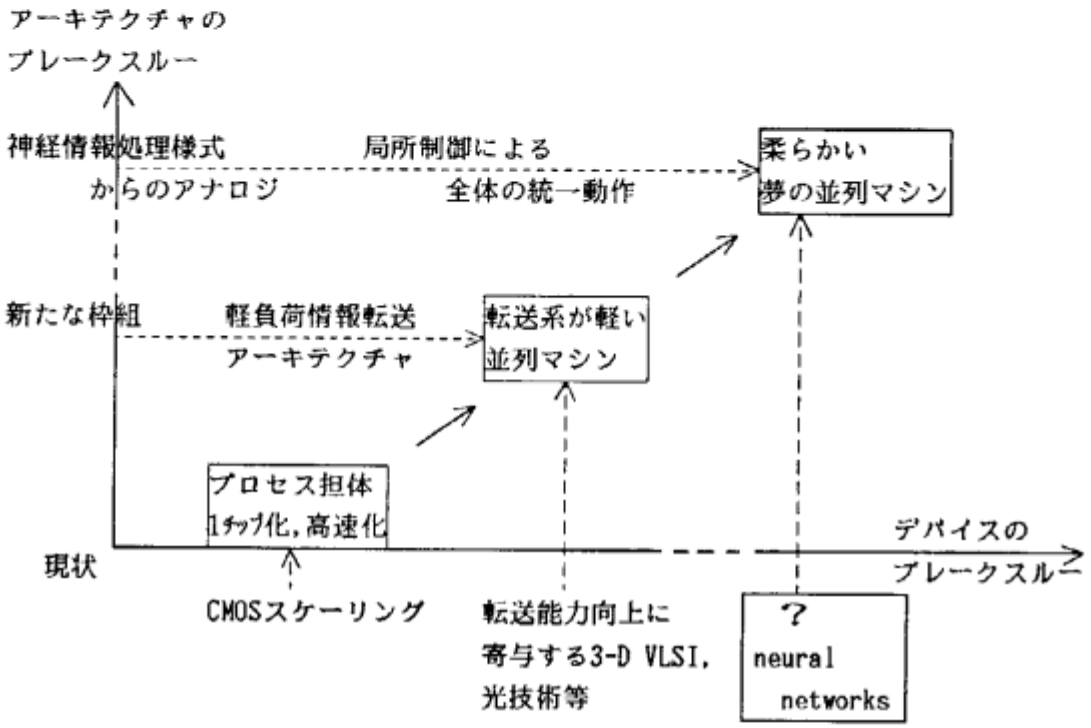


(a)ファイバ結合　　(b)導波路結合

図5 光技術による情報転送[2]

4．まとめ

①デバイス技術は，発展する．
②しかし，その発展には，限界がある．
⇓
デバイス技術の限界を打破する革新的並列マシンアーキテクチャを切望する！



柔らかいとは？

①集中制御のためのガチガチの枠組が不要で，
必要なプロセス担体が必要な時点で勝手に動きだす．
②均一構造で，どのような規模のマシンを実現しても，
それなりに動く．
③冗長性，耐故障性．

図7　デバイス技術と並列マシンアーキテクチャ

[参考文献]
(1)G.J.Myers etal.:"Microprocessor Technology Trends," Proc. IEEE, vol.74, No.12, pp.1605-1622 (1986).
(2)J.W.Goodman etal.:"Optical Interconnections for VLSI Systems," Proc. IEEE, vol.72, No.7, pp.850-866 (1984).
(3)「ニューラル・ネット」，数理科学，No.289 (1987).

「並列計算機研究の方向はこうあるべきである」

‥‥と思っているのは　近山 隆 （ＩＣＯＴ 第４研究室）

【１】何を考えないか

並列マシンの研究は，

— 非常に多数の，しかし有限個のプロセサを用いて，

— 非常に大きな，しかし有限の問題を解く

にはどうすればよいかだけを考えるべきである。ここで：

プロセサの数〈〈 並列度〈 問題の大きさ

となる算法が存在しない場合は並列ハードウェアの生かしようがないので，問題外である。

結局『都合の良い場合しか考えない，それ以外は考えても利益がない』ということ。

１．小さな問題

小さな問題については並列処理をしても結局逐次的にやるほうがよい。Fortran は数値処理に向くといっても，足し算をひとつしたいだけなら手計算のほうがよい。このような問題を相手にしていてもはじまらない。

ときたま並列処理計算機に対して『そんなに大きな問題があるのだろうか』という疑問を聞くことがある。こういう想像力の乏しい人々は相手にしないことにしよう。

２．小さな並列計算機

プロセサ数が少ない並列計算機にプロセサ数が多い計算機と同様の方式を用いてもメリットはない。とりあえず作れる規模の並列計算機では，小さいが故の性格があろうが，それにこだわって将来を見失わないようにしたい。

３．開始時／終了時の逐次性

並列処理を論じる場合によく話題になるのは，計算の開始時／終了時に逐次性があって，そこが結局計算処理時間全体のボトルネックになるのではないか，という疑問である。これは，図１に示すような並列度を示す計算にかかる時間の大部分がほとんど並列性がないことを問題としているのである。

しかしながら，計算に用いるプロセサ数よりもはるかに大きな並列度のある問題を解くのなら，アルゴリズム上の最大並列度は図１のようになりえても，実際の並列度は図２のようになり，実際の計算時間の大部分は並列性が生かせることになる。このように，アルゴリズム設計上気にしなくてはならないのは，無限個のプロセサを仮定した場合の理想的な環境での総計算時間に占める並列計算可能な時間ではなく，単純な並列度の積分のようなものでよいのである。

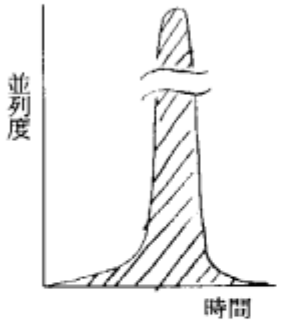


図１ 理想的並列度

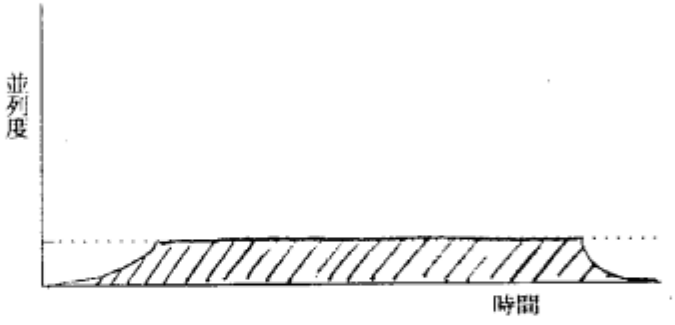


図２ 現実に得られる並列度

【２】何を考えるか

１．局所性をもったハードウェアでなければ話にならない

（１）空間的局所性 — 大規模並列計算機はメッシュしかない！

我々は4次元ミンコフスキー空間に住んでいる。どんな計算機も3次元空間内に実現するしかないし，通信には最低でも距離／光速の時間がかかる。したがって，物理的局所性を考慮しない計算機システムは，どうしても体積の3乗根に比例して遅くなる。これはＬＳＩ技術がどう進歩しようと（空間を曲げる技術が進歩しない限り）変わりようがない。一方ＬＳＩ技術はまだ進歩する余地が十分あるので，通信に要するコンスタントオーバヘッドはまだ小さくなってくれそうである。

空間的局所性を考慮すると，本当に大規模な計算機の構成は3次元以下のメッシュ構成しかない。

Ｎ台のプロセサからなるハイパーキューブ構成の計算機をある軸に沿って二分すると，その片方に属するプロセサはすべてもう一方の対応するプロセサと電線でつながっている。この電線の総延長は計算機システムの一辺の長さの半分×ＮｌｏｇＮぐらいになるはずで，次元数が大きく（ｌｏｇＮ）〉3と）なると電線の体積は：

電線の体積 ～ 一辺の長さ × ＮｌｏｇＮ

となる。電線の体積はプロセサ数より速く大きくなるから，仮にプロセサの体積を無視すると，一辺の長さは電線の体積の3乗根になり：

全体の体積 ～ ＮｌｏｇＮ の ３／２乗

となる。平均の電線長はは一辺の長さに比例するから：

電線長の平均 ～ ＮｌｏｇＮ の平方根

となる。ハイパーキューブのモデル上の隣のプロセサまでの距離がこれだけあり，これが通信遅れを支配するとしたら，Ｎ台のプロセサに同じ情報を流すのにかかる時間は：

放送遅延時間 ～ Ｎの平方根 × ｌｏｇＮの３／２乗

となり，決してｌｏｇＮには比例しない。メッシュ構成なら，隣との距離はコンスタントであるから：

放送遅延時間 ～ Ｎの立方根

となり，これはオーダとしては光がシステム全体に行き渡るのに必要な最低限である。

ツリー状の構成も事情は似ている。ツリーを実際の3次元空間に配置するには，根元近くの枝は計算機システム全体の体積の立方根程度の長さが必要であるから，ツリー構成Ｎプロセサ計算機システムでのブロードキャストの遅延もＮの立方根を下回ることはできない。

このように，メッシュ以外の構成法は，規模が小さくて配線コストがプロセサコストより極端に安い場合にしか意味がない。現在の計算機技術がそのような状態にあるのは事実であるから，今ハイパーキューブを作るのはそれなりの意味があるが，拡張性がないことがわかっているという点で，ハイパーキューブ構成は逐次型計算機と本質的に変わらない。

『メッシュではうまくいかないがハイパーキューブならうまくいく算法がある』という声も聞く。それはその通りである。そもそも逐次計算機でないとうまくいかない算法が山ほどあるのと同じことである。3次元以下のメッシュでは実現できないような算法は，本質的に並列処理向きではない算法なのである。

（2）だれが局所性を意識するか ― ハード／ソフトの分業

時間的に局所性のある計算機を生かすには，局所的なデータにより多くアクセスするような処理方式が必要である。処理対象が時間とともに変化する場合，近くに置くデータをそれに応じて入れ替える必要がある。

単一プロセサの設計には，空間的局所性の重要性は十分認識されている。レジスタをＣＰＵの中に作り，キャッシュメモリを隣に作り，メインメモリはもう少し遠く，ディスクをさらに遠くに作るのは常識である。空間的局所性を生かせるためにはあらゆる工夫がなされている。レジスタ上に良く使うものを置き必要に応じて入れ替えるのはコンパイラ，キャッシュのＬＲＵ管理はハードウェア，ディスクとのページングはハードウェアのサポートをもとにしているが，オペレーティングシステムの責任である。

同様の配慮は並列計算機でもなされるべきである。どこをハード，どこをソフトの責任とするかは，実現の効果と難易のトレードオフで決めることである。たとえば，あるＲＩＳＣプロセサでは，キャッシュの管理をハードで行なう代りに，レジスタファイルを大きく取って管理をコンパイラに任せて

しまった。こうした実現手法の問題はどちらかというと小さな問題であるが，ハード／ソフトの単独では処理しかね，総合的なシステム設計能力の問われるところである。

2．局所性を生かす処理方式とソフトウェア

（1）局所性を生かせる算法

大規模並列計算機のための算法は，並列度が充分にあるだけではだめで，局所性が良くなくては話にならない。

逐次計算機のための算法にも，局所性の良いものと悪いものがある。巨大な仮想論理空間をもった計算機上でソートティングする時に，1ギガエントリに対するヒープソートプログラムを書くのは簡単である。計算量のオーダはＮｌｏｇＮである。しかし，ＮｌｏｇＮという時の1は，1回のディスクアクセスのことになってしまいかねない。このような大規模な問題には，マージソートを用いたほうが良いのは周知の事実である。これは，マージソートのほうがデータに対するアクセスの局所性が断然優っているからである。

（2）集中管理しないこと

物理的にはなにひとつ集中管理するべきではない。ひとつでも集中管理すると，そこにアクセスが集り，全体のボトルネックになるし，それ以前に，そこにアクセスするための通信コストは莫大である。

金庫を管理したいと思っても，1円単位の出納を監視していたら必ずボトルネックになる。財布に小銭を持っている人間を大勢作って，大金の収支と財布と金庫の間の出し入れだけ監視しよう。そのために財布を持ち逃げされることもあるが，その被害よりも金庫のところまでいちいち行かなくてはならないオーバヘッドによる損の方がずっと大きい。

（3）現地調達主義 ― 産地と消費地を近付けよ

遠くのデータにアクセスする必要性はなるべく減らすべきである。ジェネレート&テスト型の問題では，いったんジェネレートしたデータは『その場で』テストすべきである。いったんどこかに集めてからテストするのでは，局所性が失われてしまう。これは「プログラムはどのデータにも共通に使われ，量的に小さい」という性質を利用したものである。いわゆるブラックボードモデルは並列実行を考えると最初から破産している。データはあくまで局所的に作り，局所的に消費すべきなのである。

（4）局所性を殺さない負荷分散

負荷分散はもちろん大切であるが，プロセサ稼働率を上げることよりも局所性を崩さないことが重要である。

札幌支社に暇になった人がいても，那覇支社の仕事を頼むのは得策ではない。必要な書類をいちいち那覇からファックスで送り，処理した結果をまたいちいち送り返すのは，明らかにばかげている。特別の技術を必要とする仕事でないのなら，札幌支社で何人か解雇し那覇支社で何人か新規採用するのが得策である。いろいろな会社が同じことをすると，長期的には札幌には就職口が少なくなるので，北海道の人口が減り，沖縄の人口が増えることになろう。個々の会社は人口の移動を計画しているのではない。また，札幌から直接那覇に移住した人も少ないだろう。しかし，全体としては必要に応じた分散がなされている。このような方式は，分散メカニズムが仕事の内容や関連性をまったく理解していなくてもかなわない，という大きな利点がある。『自動的』負荷分散はこのようなものであるのが望ましい。

まったく違う方針も考えられる。那覇支社の担当になっていた仕事のうち，他の仕事と関係の薄いひとまとまりの仕事を，まとめて札幌支社に移してしまうのである。移した仕事のまとまりが良ければ，いちいち札幌と那覇で連絡を取る必要は少なくて済む。いっぺんに『最適』に近い状態に到達できるのが利点である。このような手段をとるには，どの仕事とどの仕事が関連が深くて切り離せず，どの仕事とどの仕事はほとんど無関係にできるか，という内容に立ち入った知識が必要である。したがって，こうした負荷分散方式はアプリケーションレベルでなんらかの指示をしないと難しいであろう。

前者を『自由経済方式』後者を『計画経済方式』と呼ぶこともできよう。計画経済方式がオブジェ

クトレベルではるかに効率が良さそうに見えるのに，メタレベルまで考えると自由経済方式の利点が大きいことは現代史の教えるところである。

（５）『良い』並列算法とは

上記のことから，大規模並列計算機に向いた算法は，以下のような条件を満たしていなくてはならないことがわかる。

① 充分な並列性を持つこと
② 充分な局所性を持つこと
③ 局所性を壊さずに負荷分散しやすいこと

このうちのどれが欠けていても『良い』算法とは言えまい。

３．故障の診断と回復

超大規模計算機システムを考えるとき大きな問題のひとつは，故障の診断と回復手段である。当然のことだがＭＴＢＦが１０００年のプロセサでも，１００万台集めれば毎日３回故障する。故障率を下げるにはなんらかの冗長性を持たせるしかあるまい。

メモリシステムでは冗長性を持たせるのは常識である。ただ，扱う情報が単純なので，冗長度は非常に低くて済んでいる（６４ビットに対して８ビット，など）。もっと複雑なデータを扱う場合，このような単純な冗長性で済むわけはない。

（１）多数決方式

多くのプロセサに同じ仕事をさせて多数決で決める方法がもっとも普通であろう。ＥＣＣも基本的には多数決方式であって，３ビットも誤りがあると『無理が通れば道理が引っ込む』ということになる。人間社会でも似たようなもので，大部分の人間がこうだと思っていることは，本当の正誤にかかわらず正しいものとされる。基本的には多数決以外に良い方法はあるまい。

しかし，多数決をとるための回路の故障が問題になる。選挙は行なっても，選挙管理委員会が投票数を故意に数え間違ったりすると，多数決が実際には行われないことになる。そのような場合，人間社会では革命が起こる（フィリピンで起きたように）。選挙管理は政府機関が行なうことになっていても，圧倒的に多数の人々がその権威を認めなくなると，管理方式自体をメタレベルでひっくりかえしてしまうのである。このような機構を計算機システムに導入するのは容易ではない。

冗長度が充分にあれば，多数決回路をアナログ回路として実現してしまうことも考えられる。アナログ回路はＭＳＢ１ビットの誤りでとんでもない判断をするようなことはない。同じ仕事を１００プロセサでさせておけば，３０プロセサぐらいまでがこわれても正しく動いてくれそうである。実はこわれたプロセサがみな同じこわれ方をするとは思えないので，半分以上のプロセサぐらいがランダムにこわれても，正しい総合判断を下せるかもしれない。

（２）故障したプロセサの切離し／回復

人間社会の場合，普段から正しい判断をしていないとみなされる人間の判断は最初から多数決の数のうちに入れない，ということにより，信頼性の維持を容易にしている。これは，故障したプロセサは切り離す，ということにあたる。冗長度をあまり上げずに済ませるために，このような処理はぜひ必要であろう。

人間社会なら正しい判断をできないとみなされる人物は『更正』させようと努力する。これは，故障したプロセサを初期化から立ち上げなおすことに相当しよう。これも，一時的な故障から回復させるのには有用である。ＥＣＣ付きのメモリのソフトエラーの回復のために，読み出した訂正済データを書き戻すのはこれにあたる。

（３）ソフトウェアの故障

ソフトウェアについてももちろん同様の方式がとれる。同じ問題について複数の解法を用意し，多数の同じ結果が得られたものを解と考える。ソフトについては切り離しは容易だが，回復 ― 算法を修正することは，メタレベルの充分な知識無しには不可能であり，当分望みはなさそうである。

【4】おわりに ― 人間社会から学ぶこと

人間社会のいろいろな仕組はじつに良くできている。これは神の与えたもうたものや，いにしえの大賢が考えたものではなく，試行錯誤の結果であろう。できの悪いシステムを持った社会は数多くできたであろうが，自滅していったり，より良い社会構造をもった隣国に滅ぼされていったのだろう。

自然科学は自然界から多くを学んだ。計算機科学は人間社会から多くを学べそうに思うのである。

宇宙服は要らない！

吉田かおる

（財）新世代コンピュータ技術開発機構

1．はじめに

まず私は大変困惑している。
”並列アーキテクチャ”がどの範囲を意味するのかわからないが，もしハードウエアという意味での”アーキテクチャ”を指すならば，ほとんど興味を持っていないからである。アーキテクチャなどと言うものから程遠い私には，一般の並列問題を想定してアーキテクチャを考えることなどできない。

極端に言えば，並列実行の効率化に関してアーキテクチャに対する関心は無いに等しい。それ以前に，私はソフトウエアで解決できることをもっと考えてみたい。

2．ソフトウエア・レベルでの２ケタ

私は，この２年間ＳＩＭＰＯＳの虜であった。
ＩＣＯＴに入って１ヶ月，私はマルチ・プロセス遊びに興じていた。プロセスどうしがメッセージ交信しながら動く姿に何とも人間らしさを感じた。ストリームおよび資源管理に手を加えながら，メッセージ交信に基づいて，ファイルの共有を支援するファイル・システム，そしてネットワーク・システムにおけるリモート・オブジェクト機構を開発してきた。ひとえにメッセージ通信は美しいと思った。

ＳＩＭＰＯＳの開発から学んだことの一つは，ソフトウエアの如何により１ケタでも２ケタでも速くすることも遅くすることもできるということである。それに比べて，ハードウエアやファームウエアで桁上がりの高速化を望むのは難しい。だからと言ってアーキテクチャに凝る必要はないと言うのではない。アーキテクチャの高速化により，同じソフトウエアが黙って数倍速くなり，対象とする応用問題が広がることは嬉しい限りである。すなわち，アーキテクチャに凝らなくてよいという意味ではない。ただ，どんなに凝ったアーキテクチャでも，その上に幼稚なソフトウエアを載せてしまえば話にならないということである。

ソフトウエアの性能がその実装レベルの質すなわちコーディング技術によるのはもちろんであるが，やはり決定的な要因は処理方式およびアルゴリズムである。

例えば，ネットワーク・ハンドラおよびハードウエアでいくら頑張っても，メッセージ・プロトコルが貧弱で無闇やたらにメッセージを送出して送信待ちが多発したり，パケットがバッファ間で何度もコピーされたり，またパケット・サイズが小さ過ぎて割込みが小刻みに入るとか送達確認を上層で処理するといった原因でプロセス切替が頻発するなど，方式設計のミスにより簡単に２ケタ遅いシステムができあがる。ＳＩＭＰＯＳではプロセス切替えが重いこともあり，このような通信プロトコルの誤し悪しが全体の性能に大きく影響した。

3．通信量に基づく負荷分散

逐次型分散システムの負荷分散制御は，一般にプロセサ内処理量の均一化を第一制約条件にして行われている。それは，ジョブあるいはプロセスといった独立性が高く，しかも通信の結合関係が弱い大粒を並列の単位もしくは通信の対象としているからである。

ＳＩＭＰＯＳにおけるリモート・オブジェクト機構では，プロセスより小粒なオブジェクトを通信の対象にした。オブジェクトへのメソッド・コールのレベルでメッセージ通信を行ない，あるリモート・オブジェクトへのメソッド・コールが別のリモート・オブジェクトへのメソッド・コールを引き起こすといった入れ子通信の可能性をもっている。明らかに通信の結合関係は密になる。
ＫＬ０／ＥＳＰ言語が逐次型シングル・プロセサを仮定したものであったことから，

ソフトウエアで実現せざるをえなかった。各プロセサのライブラリ環境とシンボル表は独立であり，パケット生成時にライブラリへの問合せ，シンボル／アトム変換などのオーバーヘッドがあった。また仮想回線を使用したため回線接続からプロセス環境の整備までの初期化がかなり重かった。
その結果，メソッド・コールの内外のコスト比は１：２００００と極めて高くつき，パケット送信数を低減することが必須であった。

メッセージ・プロトコル，パケット・フォーマット，パケットのコピー，パケット・サイズなど処理方式に関わる要因から，１メッセージあたりの通信コストは計算できる。これを最小に抑えるように処理方式を決めることは必要条件である。むしろ問題は，いくら１メッセージあたりのコストが安くても，メッセージを頻発するような状況を起こしては元も子もないということである。あるメッセージ送信に起因して，数多くのメッセージが送信されているのかもしれない。つまり，何故メッセージ送信を起こすかについて解析する必要がある。

このリモート・オブジェクト機構の開発で，メッセージ送信量を低減するために最も検討したのは，オブジェクトの移動問題である。
オブジェクトには，移動しやすいものと移動しづらいものがある。例えば，大きな構造体を抱えるオブジェクトは移動しづらい。しかし，いくら移動しづらいオブジェクトでも，それをアクセス／参照する他のオブジェクトあるいはメッセージとの関係が密であれば移動せざるをえない。つまり，オブジェクトが実際に移動するか否かはオブジェクト自身の移動性だけでなく，他のオブジェクトやメッセージとの相互の結合関係によって決まるのである。

並列論理型のＭｕｌｔｉ－ＰＳＩ，ＰＩＭでは，並列の単位がさらに小粒なゴールである。同じ小粒でも，ＣＭのようにＳＩＭＤに基づく問題は，結合関係は規則的でありしかも疎であるが，通信量はやはり多い。ＫＬｌの場合，ゴール間が変数で結合され，結合関係は変則的でかつ密であり，通信量はより多いと予想される。もちろん，問題の性質や解法により結合関係の疎密が変わってくるだろう。layered stream のような複雑な構造体を複数のゴールが参照する場合は密となろうし，divide & conquer のようにデータが一方向に流れていく場合は疎となろう。（詳しくはＫＬ１－ＴＧメンバーまで）
サラサラの粉末洗剤でも摩擦熱を発生する。プロセサ内の通信抵抗は０であり，通信距離に従って通信抵抗が増加すると鑑みると，果たして通信量は摩擦熱か？

結論から述べる。

- 極小粒度で変則密結合のゴールを並列単位とするＰＩＭでは，プロセサ内処理量の均一化よりもプロセサ間通信量の最小化と均一化を第一制約条件にして，負荷分散を制御すべきである。

- ソフトウエア・レベルで通信量低減に役立つ最適化情報をできるだけ多く与えるべきである。

- 通信量は，分散されたゴールとデータの間の結合関係で決まる。

- 最適化情報はこの結合関係で与えたい。

- 結合関係をパターンするようなプログラミング・スタイルもしくは言語がほしい。

４．群れの移動と通信量

ゴールを一つ投げ出してみる。それに伴い，引数の外部参照・単一化のためのメッセージ通信が生じる。そしてゴールがさらに別のゴールを投げ出す。プロセサ空間にはゴール雲ができ，それらの間は変数による結合網が張られている。

通信量が増える原因を考えてみよう。
- ゴール１つでも，それが複数個の引数を外部参照すれば，メッセージ送信数は増える。
- 変数１つでも，プロセサを渡った間接参照が生ずれば，メッセージ送信数は増える。
- データー一つでも，複雑な構造体であれば，コピーに伴うパケット送信数は増える。
- 同じ構造体でも，複数回外部参照されれば，コピーよりも参照のためのメッセージ送

信数が増えるかもしれない。
－ パケット１つでも，遠いプロセサへ送信するのであれば，通信網の占有量は増える。

４．１ 通信状況を表わす量

通信網の容量は有限かつ均質であるため，通信網の占有量から通信状況を表わすのが適当であろう。

①区間占有率 H
H ＝ （単位時間あたりパケット通過数）／（区間容量）

②プロセサＰｉの送信負荷 Ｔｓ(ｉ)
Ｔｓ(ｉ) ＝
Ｓｕｍ {Ｔｓ(ｉ,ｊ)}　　Ｔｓ(ｉ,ｊ)：Ｐｊ への送信負荷
ｊ
Ｔｓ(ｉ,ｊ) ＝
（Ｐｉ－Ｐｊ 間通信距離）＊（単位時間あたりのＰｊ へのパケット送信数）

③プロセサＰｉの受信負荷 Ｔｒ(ｉ)
Ｔｒ(ｉ) ＝
Ｓｕｍ {Ｔｒ(ｉ,ｊ)}　　Ｔｒ(ｉ,ｊ)：Ｐｊ からの受信負荷
ｊ
Ｔｒ(Ｉ,ｊ) ＝
（Ｐｉ－Ｐｊ 間通信距離）＊（単位時間あたりのＰｊ からのパケット受信数）

ここで，全プロセサの送信負荷と受信負荷のそれぞれの総和は等しい。
Ｓｕｍ {Ｔｓ(ｉ)} ＝ Ｓｕｍ {Ｔｒ(ｉ)}
ｉ　　ｉ

まず区間占有率は通信先が埋もれてしまった情報であり，これからゴールおよびデータの分散具合を推測するのはむずかしい。
通信先が直接反映されるのは，送信負荷と受信負荷である。送信負荷は，プロセサ内の変数が複数個の外部ゴールから参照される時に増加する傾向にある。受信負荷は，プロセサ内ゴールが複数個の外部変数および構造体データを参照する時に増加する傾向にある。

４．２ ソフトウエアで与える最適化情報

－ プログラマにとって与えやすい（把握しやすい）最適化情報とは何か？
－ 最適化情報を抽出しやすくするには，どのようなプログラムであってほしいか？

私がプログラマであるならば，ゴールの投げ出しに関して，たとえそれが next などという相対位置であろうと論理プロセサ番号による絶対位置であろうと，投げ出し先という地理的情報に無関心でいたい。それより，プログラムを書く時にどの変数が構造体となるか，どのゴールからどのゴールへデータが流れるか（単一化の方向）がわかるわけである。このゴールとデータの結合関係で表現したい。後は，この結合関係を示す情報と通信状況を表わす上記の通信負荷に基づいて，どこに投げるかは動的に決めてほしい。（とにかく私は怠け者である。）

４．３ 移動度

①ゴールの移動度
生成されたプロセサに子々孫々にわたり定住する度合い。
例えば，ある構造体を親子代々に渡って参照するような永続的プロセスは移動性が低く，なかなか他プロセサへ投げ出されない。また，数多くの引数を参照するようなゴールも移動性が低い。

②データの移動度
他のプロセサからの外部参照時にコピーする度合い。
例えば，構造体データは移動性が低くなかなかコピーされない。基底データは移動性が高く参照時にコピーされる。

4.4 結合度

①ゴール間の結合度
他ゴールが投げ出されるならば，それに付いてあるいは近くに投げ出されたい度合い。
例えば，複雑な構造体データを参照しあう兄弟ゴールは結合度が強い。

②変数（データ）／ゴール間の結合度 （ゴールへの帰属性）
ゴールの生成されたプロセサあるいは近くに変数（データ）を置きたい度合い。
例えば，入力変数の場合ゴールへの結合度は強く，出力変数はゴールへの結合度は弱い。

4.5 変数とゴールの引合い

ゴールと変数（データ）間の結合度が強い場合，ゴールの投げ出し時に一緒に変数が移動するか，変数のあるプロセサにゴールが投げ出されるかは，移動度と結合度の関係で決まる。前者は，ゴールも変数も移動度が高い場合である。後者は，ゴールの移動度が変数の移動度に比べて高い場合である。（綱引きは身軽な方が負け！）

4.6 ＫＬ１－Ｃにおけるプラグマ指定

基本的には，問題を解法する時すなわちアルゴリズムを組み立てる時，すでにプログラマは上記の結合関係に関する大まかな情報を把握もしくは意識しているはずである。結合関係をパターン化するようなプログラミング・スタイルや言語が提供されれば，もはやプログラマが結合関係を意識する必要もなくなるだろう。

まず，移動度と結合度はＫＬ１－Ｃのレベルでゴールおよび引数に対して一種のプラグマとして指定できることが望ましい。肝心の指定方法と実現方法は思案中。

4.7 上位言語における結合関係のパターン化

現在，私はＰＩＭＯＳの記述を主目的としたＫＬ１－Ｃ上のユーザ言語群ＫＬ１－Ｕの一つを仕様検討中である。ＧＨＣにおける永続的プロセスへのストリーム通信を前面に押出した並列オブジェクト指向言語を想定している。

この言語では，プログラマが細かい並列性を意識せず，しかも潜在的な並列性を引き出すことを目指しているが，それに加えて，上で述べた移動度や結合度を内部表現においてパターン化することも目的の一つである。

メッセージの受信から他オブジェクトへのメッセージ送信を１周期とする永続的プロセスが実行の基本パターンである。オブジェクトは同一形式の引数列をもつ述語として内部表現し，連想表など移動性の低い構造体引数，受信ストリームなどゴールと結合度の強い入力引数，永続プロセスをなすゴール間結合が予めわかるので上記のような最適化情報は抽出しやすくなるものと思う。

5．メッセージの相乗り

ＫＬ１の場合，到着確認，領域解放，終了伝達などメッセージ名と１～２個の引数から成る数ワードの小さいメッセージが頻繁に送信されるであろう。メッセージを載せた物理的なパケット通信量を最小に抑えるには，メッセージを相乗りさせるべきであると考える。

街中を流れるタクシーの量を想像されたい。４人乗りのタクシーに一人しか乗らないで同一方面へ４台行くのと，１台に４人が相乗りしているのでは，流れるタクシーの量は明らかに１／４である。ここで，４人がそろうまでタクシー乗り場で待つとなると，最後に待つ１～３人は永久に乗れなくなる。タクシーが定期的に回ってきたら，その時，同一方面へ行く人を募ればよい。

このようにメッセージを送信待ちバッファに登録し，一定数のメッセージが集まるかもしくはタイマ割込みが入るかいずれかの原因でデーモン・プロセスを起動し，これが

同一のプロセサを宛先とするメッセージをまとめてパケットに詰めて送信する方式を得策と考える。

6．余裕ある柔軟なアーキテクチャを

将来の使い方を見越してシステムを設計するのは大変難しい。
大規模な応用問題を解くような実用化の時点では，もはやシステムを構成している技術は過去のものである。ＯＳ，言語，仮想マシン・レベル（ファームウエア）そしてハードウエアの順でその遅れは積算され，ソフトウエア側が一番つまらない思いをするのは，アーキテクチャの制限で仕様を決めざるをえない時である。

ソフトウエアから望むことは，至極簡単である。

特別なアーキテクチャでなくてよい。単に，将来の新しい要求を受け容れる余裕や柔軟性のあるアーキテクチャであってほしい。すなわち，手が加えられる術を残しておいてほしい。

具体的に言えば，

①要素プロセサ
- マイクロ・プログラマブルであること。
- ＷＣＳはたっぷり。
- レジスタはたっぷり。
  - ・General Unification の支援
    アドレス・レジスタ，データ・レジスタ，構造体ポインタは２組用意。
  - ・ＲＥＦx からの ＲＥＦo回収を支援するならば，
    書き戻しのためにアドレス・レジスタを追加。
- ＭＲＢ切出しおよび論理演算はハードウエアで支援。
- 負荷分散の計測支援
  - ・ゴールの enqueue 時にその優先度を加算するカウンタ。

②ネットワーク制御ハードウエア
- メッセージのブロード・キャスト機能の支援
  受信バッファから送信バッファへのコピーをハードウエアで支援。
- 負荷分散の計測支援
  - ・プロセサの通信負荷（送信負荷，受信負荷）

7．おわりに

見知らぬ国へ旅に出る。
最悪の事態に備えて生活に必要と思われる衣類，薬，電気器具に至る一切合切を鞄に詰め込んで出かける者と，体力と生きる術だけを身につけ軽装で出かける者がいるとしたら，おそらく私は後者である。重たい荷物で身動きできなくなるのが嫌いであり，極めて怠惰である。鞄は折畳み自由で中味に合わせて形のかわる枠のないものを愛用している。不便が生じても，心に余裕があれば自ずから知恵は浮かぶものであり，いつの間にか鞄はその国の香りでいっぱいになる。

ＰＩＭという桃源郷へ何を携えていこうか？
太陽と酸素と水のある同じ地球にあるのなら，宇宙服は要らないだろう。
金米糖に聞いてみよう。

MINUTE

議事録

1．Opening Address — 田中英彦先生（東大）

東京を離れて，個人のしがらみを忘れて議論をしよう。
ＩＣＯＴプロジェクトの中間点にあたり，並列マシンについてハードの人ばかりでなく，ソフトやアルゴリズムの人々とも議論しよう。

2．ＰＩＭの将来にむけての私の経験

2.1 並列推論マシンＰＩＭ　前期の結果から — 久門 耕一 氏（富士通）

(1) ＯＲ並列について
制御が機械まかせなので，条件をかけば答えが求まるのは簡単な場合だけ。
ＯＲ並列では，解の数と実行時間は比例する。→ 解の間の比較などがうまくできないため，解を集めた後の逐次の処理で処理の時間が決まってしまう。
アルゴリズムが重要であるにも拘らず，アルゴリズムの記述ができない。
↓
以上より，ＯＲ並列はだめだと思った。

(2) 株分けで分かったこと
クイーンはＯＲ並列にとって救いの神であった。
要求駆動の負荷分散で結構うまくいく。
処理の粒度が大きいときは，かなりいい加減な分散でよい。

(3) アルゴリズムの重要性
逐次の高速アルゴリズムに対抗する並列アルゴリズムの開発が必要。
負荷分散のアルゴリズムが疎結合計算機の最大の課題 → ハードウエアだけでは無理。

(4) ＧＨＣに対する疑問
自己記述ができないのが問題だ。 → インタプリタが書けない。
ＧＨＣの記述力の低さは足かせである。
新言語普及のための努力を行っているか。

(5) （中期）ＰＩＭについて
知識処理マシンを考えないとただの箱になる。
入出力が弱くて大丈夫か。

(6) 結論
パラレルプログラミングは奥が深いので全力で取り組もう。
動的負荷分散が大規模並列計算機の焦点。
頭だけでなく手足もほしい。

(7) 入出力とＰＩＭに関する討論
データをどんどん入れて，処理結果をどんどん出すのが並列計算機 — 平木
少しのデータで，じっくり考え結果を出すのが並列計算機 — 近山

2.2 中期ＰＩＭに対する私見　―― 杉江　衛 氏（日立）

(1) 前期から通算して約４年間ＰＩＭに従事している。
(2) ＰＩＭを作る意義として，性能と並列環境の提供があるだろう。
(3) メーカの立場からすると，高性能であることが非常に大切。性能が上がらなければ新しいマシンを作る意義は無い。
(4) 高速逐次マシンとＰＩＭの性能比較をしたとき，ＰＩＭが逐次マシンを上回るのは何台プロセッサを結合したときか？ ― 幾つか仮定を置き，式により求めて見ると，前期のＰＩＭ－Ｒのアプローチでは，１０００台規模となる。一方，並列オーバヘッドを０と仮定すると，１００台でも逐次マシンの５倍程度はだせることがわかる。これが中期ＰＩＭのアプローチである。
(5) 並列環境の提供においては，並列に動くべき対象をはっきりさせないと，どんな手段を持ってしてもうまくいかないのでは？ ― 交通システムのアナロジーから。
(6) この意味で，中期ＰＩＭの密結合クラスタの導入は並列環境の提供と言う視点から見れば後退である。― マルチＰＳＩでよい。
(7) 小粒度が遅いのは多環境がネックになっていたからである。― ＧＨＣに期待する。
(8) 中期の試作はまず性能である。

2.3 データフローマシンとＬＳＩ技術　―― 伊藤徳義氏（沖）

(1) 前期で検討，試作したデータフローモデルをベースとした並列推論マシンのＬＳＩ化を進めている。
(2) データフローマシンのメリットは，Context Switching に強いことと，パイプライン制御が容易に行えることの２つであろう。
(3) ＧＨＣには，数リダクション実行した後に Context Switching が発生する様なプログラムがある。この様なプログラムはノイマン型マシンで実行したのでは，高速実行は期待できない。しかし，データフローマシンは，context Switching に強いのでこの様なことは気にしなくてよい。
(4) データフローは，ハード的には汎用のパイプラインプロセッサと考えられる。汎用機では，パイプを隙間なく埋めるのは困難であるが，データフローでは実行可能な命令がある限り自然に埋まる。

2.4 並列処理アーキテクチャを考えるとき気になる二つの問題 ― 雨宮 真人 氏（ＮＴＴ）

(1) 二つの問題とは ① プロセスの粒度とプロセス管理 と ② 動的に変化する構造データの処理 のことである。
(2) ① の問題は，粒度の小さい多数のプロセス（論理型プログラムではゴール，関数型プログラムでは，関数に対応するだろう。）の並列実行をいかに実現するかと言うことである。
(3) ② の問題は，プログラム実行中に使われる構造体の扱いで，共有するのか，コピーするのか，すなわち，ローカルメモリでいくのか，共有メモリでいくのかと言うことである。
(4) これらの２つの問題は，結局，ローカリテイ重視で集中方式でいくか，徹底的に分散させるかの問題に帰着される。

(5) コンパイラでローカリティを抽出する方式として，プログラム変換（e.g.ＴＲＯ）や，データフロー解析などが考えられるが，非常に困難である。
(6) コンパイラによるローカリティ抽出が困難であるならば，並列処理アーキテクチャは分散化を志向したアーキテクチャの方が有利である。
(7) 例えば，従来のプロセッサとメモリの疎結合，高速レジスタに代わって，低速分散一様レジスタを用いる方式のほうが有利である。

3．中期プロジェクト

3.1 ＰＩＭの中期計画を作るに当って考えたこと ― 内田 俊一氏（ＩＣＯＴ）

(1) アプリケーションソフトは専門家が作らなくてはいけない。そのためには専門家をencourage しなくてはならないが，その際，精神論ではダメで，プログラミング環境を提供することが必要である。
(2) マシンを作るに当って，「ターゲットを絞って作る」というアプローチは，何をターゲットにしたらいいかわからないし，何かをターゲットにしても結局何もわからないのでとらない。
「ターゲットが定まるまで待つ」というアプローチは，いつまで待ったらいいかわからないのでやはりとらない。何かを待っていたら始まらない。

3.2 並列処理文化づくりに向けて ― 瀧 和男 氏（ＩＣＯＴ）

(1) 並列処理は New Culture である。
(2) 計算モデルまで含むことが必要であろう。
(3) 数値計算はアルゴリズムが存在するので単純に記述できる。数値計算自体は決して単純ではない。一方，記号計算はまだアルゴリズムが存在しないので単純には記述できない。

3.3 並列マシンはかくありたい ― 中島 克人 氏（ＩＣＯＴ）

(1) ＰＥ1000 台以上実装できる大規模並列アーキテクチャを目指したい。
(2) どんな階層構造を持つマシンでも，ＰＥ数を大幅に増やして行くと，最上位の層では一様な構造になると思われる。
(3) この最上位の一様な構造としてメッシュが最適だと思われる。２次元ではなく３次元メッシュにしたい。

4．並列アーキテクチャに対する私の主張

4.1 並列アーキテクチャに対する私の主張 ― 富田 眞治 先生（九州大学）

・富田氏の考えているものは、マクロなレベルでＳＩＭＤであるという意見が出て、ＳＩＭＤとＭＩＭＤの定義の違いにより、討論が発散した

4.2 並列アーキテクチャに対する私の主張 ― 相田　仁 氏（東京大学）

(1) ＰＩＭでは、おそらく１つの仕事はせいぜい数秒しか動かない。
(2) 従って、ＰＩＭはマルチ・ユーザ／マルチ・タスクで動く。
(3) 総てのＰＥにコードをコードがロードされている必要があるのかが疑問。
(4) まずは、１つの仕事を高速に実行することを考えるべきではないか。

4.3 放送機能を持つプロセッサアレイによる並列ユニファイアの構想 ― 和田 耕一 氏（筑波大学）

(1) メモリを高機能化した時に、どうなるかを考えている。
(2) fact中心のデータを考えるのなら、ベクトルよりパイプラインに向いているのではないか。
(3) おおまかに言えば、コネクション・マシンに近い

4.4 マルチプロセッサシステムＰＡＲＫ上の並列Ｐｒｏｌｏｇ処理系について ― 松田 秀雄 氏（神戸大学）

(1) マルチ・プロセッサを作成して、以下のことを感じた。
　・実行に再現性がなくデバッグしにくい。
　・ユーザ・レベルに逐次性を記述できる機構が欲しい。
　・並列計算機では、タイミングに関わるバグが非常に多く出る。
　・ＧＨＣは、メモリ・ロックが多すぎる --〉ＧＣをどうするのか？
(2) マルチＰＳＩでは、シミュレータを使って、再現性があるようにデバッグする。
(3) ＰＩＭとマルチＰＳ１では、最小の単位が異なるので、マルチＰＳ１はＰＩＭのデバッグにならないのでは
(4) ＧＨＣのコミットが非決定的なのがハード屋にとってこまる。

4.5 並列アーキテクチャに対する私の主張 ― 小畑 正貴 氏（岡山理科大学）

(1) 応用レベルでは並列性を意識したくない
(2) 逐次的に考えたものを並列に実行してくれるからよいのでは
(3) 逆に、defaultが並列で、逐次的なところだけ意識すればよいのがうれしい

4.6 並列アーキテクチャに対する私の主張　— 吉田 紀彦 氏（九州大学）

(1) ソフトウエア側から見た並列処理について考えている。
(2) （逐次でも同じだが．．．）並列プログラムのアルゴリズムの定石を貯えることが必要である。
(3) 現在考えているのは，
　① 高並列処理へのプログラム変換の応用
　　・負荷分散のサポートへの応用など考えている。
　② プロセスのネスティング
(4) フロアーからの意見
　（我々のように）逐次処理の考え方に毒されている人のプログラムを，並列に書き換えるには良いが，（将来現われるであろう）パラコン（パラレルコンピュータ）少年たちにとっては，変換後のプログラムは，あたりまえの書き方となっているのだろう。

4.7 メモリ割り当ての並列性について — 川上 桂 氏（松下技研）

(1) 問題に応じて，並列性を引き出す方法を考えるべきである。
　・「メモリ割り当て特有の並列性」を引き出す必要がある。
(2) 討論
　・メモリ頻度の集中は，スケジューリングによるのではないか。
　・マルチPSIでは，PEごとにフリーリストを持っているが，フリーリストのなくなったPEは，どうしたら良いか。
　→buddyでは，アロケータがメモリをなくして，リクエストを受け取ったら，他へリクエストを投げてしまう。
　・プログラムによっては，ジェネレータのPE，コンシューマのPEのように分業する場合がある。PIMでは，ジェネレータのPEで，フリーリストが尽きる可能性がある。尽きる前に知る方法はないか。

4.8 P^3 でほんとにいいの？　— 中島 浩 氏（三菱電機）

(1) P^3 は，進化していない。
　・大規模メッシュは作れない。なぜなら後期ＰＩＭのPE間線長は 5 mm - 100 cm。
　・線長の不均一さが増える。
　・大モジュール間の配線も，増える。
　このため，ハードウエアの階層性とP^3は，マッチしない。
(2) そこで，.... 「宇宙モデル」を考えた。
　「根性のあるやつだけが，他の銀河系へと飛び出せる。」

4.9 並列アーキテクチャ研究に対する私の主張 — 小池 汎平 氏（東京大学）

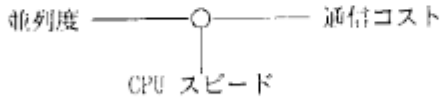
(1) プログラム動特性の把握にはベンチマークプログラムによるシミュレーションが必要である。

・（フロアーより）実用的でない???

(2) トイプログラムについての議論がされた。
・トイプログラムにも，まだ見るべきところがある。
・トイプログラムは，やめて大規模なプログラムをもっと書くべきだ。

4.10 並列アーキテクチャに対する私の主張 — 横田 実 氏（日本電気）

(1) 楽をしたい。 → レイジープログラミング
(2) 並列マシンの開発には，強い動機ずけとユーザーのニーズがなければならない。



(3) データ構造についても考慮する必要がある。
(4) ブロードキャスト も必要。

5．これから何をすべきか，どうありたいか

5.1 論理型言語で記述された応用問題のための並列計算機アーキテクチャについて — 平木 敬 氏(電総研)

(1) 並列計算機の目的は絶対的処理速度の向上にある。
(2) 論理型言語は問題を記述するものであり，それ自身並列性やその形態を論じるものにはならい。むしろ記述された問題こそが並列アーキテクチャの前提となる。
(3) 並列推論計算機アーキテクチャ開発においては，逐次型推論計算機アーキテクチャ及び数値計算用並列アーキテクチャがその基礎となる。
(4) 数値計算用並列計算機アーキテクチャとの比較を通して，並列推論計算機アーキテクチャの考察を次の観点から行った。
- 共有メモリと構造体処理
- データ転送と結合網
- 同期と分散
- データフローオーバーヘッド

(5) 以上の考察より数年先の並列推論計算機アーキテクチャに対する私見を述べた。

5.2 アーキテクチャ研究における ＰＩＭ開発プロジェクトの役割 — 柴山 潔 氏（京都大学）

(1) ＰＩＭとＫＰＲの様々な観点からの比較。
アーキテクチャ,粒度,しがらみ,夢,・・・
(2) アーキテクチャ研究の将来。
専用機か汎用機か？
ソフトに刺激を与えるようなアーキテクトを目指すべきである。
(3) (理想的な)ＡＩマシンとは？
５ＧマシンはＡＩマシンか。
アーキテクチャとしてはＬｉｓｐマシンで十分？
(4) 論理型言語
純粋Ｐｒｏｌｏｇの論理的並列性制限するすることによって，ハードウェアによる制約の下でＡＩの本質的実現ができる。

5.3 デバイス技術の発展／限界と並列マシンアーキテクチャ — 並列マシンアーキテクチャに新たな枠組を！— 小倉 武 氏(ＮＴＴ)

(1) VLSI技術の限界は近い。これを考慮した(限界を打破するような)並列マシンアーキテクチャの構築が必要である。
(2) プロセス担体の能力向上に比べプロセス担体間の情報交換能力向上は低い。したがってデバイス,アーキテクチャのブレークスルーが必要。
3-D VLSI，光技術，超電導，neural networks　局所制御
(3) 柔らかい(夢の)並列マシン
ガチガチではない，均一構造,冗長性,耐故障性

5.4 並列計算機研究の方向はこうあるべきである　— 近山 隆 氏（ＩＣＯＴ）

(1) 並列計算機研究において考えないこと
　　小さい問題，小さな並列度しかない得られない問題
　　プロセサ数 << 並列度 < 問題の大きさ が成り立たない場合は問題外。
(2) 大規模並列計算機はメッシュしかない。
　　ハイパーキューブには将来性がない。ツリーも同様である。
(3) ハード/ソフトの分業には総合的なシステム設計能力が必要である。
(4) 局所性が重要である。
　　算法，負荷分散も局所性を考慮に入れなければ話にならない。
　　良い並列算法は，充分な並列性(当然),充分な局所性(研究不足),局所性を保ったままでの負荷分散のしやすさ(未着手)がなければならない。
(5) 故障診断と回復
　　冗長性が必要。
(6) 淘汰され洗練されたシステムができる。

5.5 宇宙服はいらない！　— 吉田 かおる 氏（ＩＣＯＴ）

(1) ＩＣＯＴに入ってからの私，現在の私，これからの私
(2) メッセージ通信は美しい。
(3) 通信の結合関係に基づく負荷分散。
　　プロセサ間通信量の最小化と均一化が重要。
(4) ＫＬ１－Ｕ
　　並列性を意識しないで記述し，実行は並列。
(5) 柔軟性のあるアーキテクチャを望む。

6．総括　　田中 英彦先生（東京大学）

(1) Recomendation
　　1．本音の討論
　　2．並列研究　熱意とパワーはある。
　　3．ＩＣＯＴ ＰＩＭは成功する(に違いない)。(但し，中期マシンは)
　　4．将来の方向：議論５２出
　　5．幹事の方　ごくろうさま
(2) Addendum
　　将来の並列処理の方向は，
　　　　To be determined by each member.
　　に違いない。

Proceedings of
ICOT-WG Workshop on
Parallel Inference Machines
and Multi-PSI Systems

昭和６２年 発行
編集発行　（財）新世代コンピュータ技術開発機構
〒１０８　東京都港区三田１－４－２８
三田国際ビルヂング ２１Ｆ
０３－４５６－３１９３