ICOT Technical Memorandum: TM-0835

TM-0835

Proceedings of ICOT-WG Workshop on ここまでわかったPIM (Parallel Inference Machines) そして今後

内田俊一，後藤厚宏，石川　篤

December, 1989

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

Proceedings of

ＩＣＯＴ－ＷＧ Workshop on

ここまでわかったＰＩＭ (Parallel Inference Machines)

― そして、今後.

― 鎌倉にて ―

平成元年 10 月 6 日 (金) ～ 7 日 (土)

ＩＣＯＴ　第四研究室

ＰＩＭ－ＷＧ（Ｗorking　Ｇroup）ワークショップの概要

ＩＣＯＴ　第四研究室

開催経緯

ＰＩＭ－ＷＧでは、国内外の大学や研究機関での並列処理についての関連研究を互いに紹介しあい討議を行ってきた。そして、ＩＣＯＴのプロトタイプ・ハードウェア（ＰＩＭ：Ｐarallel　Ｉnference　Ｍachines）とそれに関連するＫＬ１（Ｋernel　Ｌanguage　１）やＰＩＭＯＳ（ＰＩＭ　Ｏperating　Ｓystem）の研究開発について討議し、助言を得てきた。

この活動を通じて、ＷＧの各委員、各オブザーバと出席者の相互理解が深まると共に、並列推論システムに関する問題意識がさらに具体的なものとなってきたと思われる。

そこで、ＷＧの構成メンバを中心にワークショップを企画・開催し、ＰＩＭの研究開発の中期計画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、後期計画の進むべき研究方向を探ることとした。

また、本年は後期計画の最初の年に当たり中期計画までの経験を踏まえて、討論では並列処理について、ハードウェア試作の現場サイドからの現状報告と、ハードウェア／ソフトウェアの研究サイドからの研究状況報告というふたつのサイドからの発表を基に、議論をすることとした。

開催日時と期間

１９８９年１０月６日（金）１３：３０～１０月７日（土）１２：００まで

開催地

ＮＴＴ　ゆかり荘　鎌倉市材木座５－７－５

参加人数

総参加者数　　48名

内訳:

大学関係	6名
電総研	3名
関連会社	13名
ＩＣＯＴ	26名

発表件数　　18件

総　括

田中　英彦（東京大学）

第１回のＰＩＭ－ＷＧワークショップを行ったのが１９８７年７月であるから、今回は２年後のワークショップになる。第１回は、ＦＧＣＳプロジェクトの真中に当たり、その目的は前期を評価し中期以降に資する所にあった。今回は、微か２年後とは言え、後期の初年度であり、第五世代コンピュータのプロトタイプ構築が現実の仕事として形成され、設計が進展し、それに関わっている人々の目の色が変わり始めた時期である。この時期に一堂に会して、今迄に明らかになって来たことを確認し、今後為すべきことを議論しようというのがその目的であった。

ワークショップは２日間にわたり、１日目は例によって深夜に及び白熱した議論が行われた。そのキーワードを取りあげてみれば以下のようになるかと思う。

1　各ＰＩＭマシンの設計が固まり実装に入りつつある。

2　ＫＬ１処理系とツールが揃って来た。しかし、並列プログラミング手法は未成熟でプログラマも少ない。

3　並列機制御技術として、負荷分散、ローカル制御・グローバル制御、動的制御・静的制御、プライオリティ制御等、具体的な方策の検討が進んでいる。

4　応用については、幾つかの小規模問題を用いての評価が為され、問題の特性理解が進んでいる。また、新たな応用として遺伝子情報処理が検討され始めた。

5　並列推論以外の並列処理モデルとの比較検討、並びに並列処理の今後目指すべき方向についての議論があった。

これらの状況を眺めてみるうちに次のような感想が出て来る。ハードウェアアーキテクチャはかなり解って来たし、設計も固まって来た。今後は並列処理の経験を積み重ねるとき

る。各種応用をプログラム化することから、それらプログラムを並列アクティビティに展開すること、各アクティビティをスケジュールし制御して並列処理効果を上げることに至る迄、質的にも量的にも様々な経験を積み、並列処理に対する感覚を養い、種々のツールや小さな工夫を貯め込むことである。

今回のワークショップの参加者をみて気付くことは、２年前の人々に比して新人が増していることと、以前新人であった人々の著しい成長であった。これは、この分野研究者の質的・量的拡大を意味し、今後が大いに期待される。第五世代プロジェクトのもう一つの大きな成果はこの辺にあるのであろう。

しかし、この種の研究グループの世界が、我国内をみるだけでもまだまだ狭いように思える。ＰＩＭ－ＷＧの枠を越えて、広く研究成果の流布と交流をまき起こし、「文化創造」を目指して協力し合ってゆきたいものである。

開催委員

委員長	田中　英彦 Prof. Hidehiko TANAKA	(東京大学)
プログラム委員	内田　俊一 Shunichi UCHIDA	(ＩＣＯＴ)
	後藤　厚宏 Atsuhiro GOTO	(ＩＣＯＴ)
	石川　篤 Atsushi ISHIKAWA	(ＩＣＯＴ)
運営委員	赤尾杉　隆 Takashi AKAOSUGI	(ＩＣＯＴ)
	高木　常好 Tsuneyoshi TAKAGI	(ＩＣＯＴ)

出席者一覧

田中　英彦 (東京大学)	雨宮　真人 (九州大学)	久野　英治 (沖電気)
久門　耕一 (富士通)	小池　汎平 (東京大学)	小長谷明彦 (日本電気)
坂井　修一 (電総研)	柴山　潔 (京都大学)	富山　眞治 (九州大学)
中島　浩 (三菱電機)	西田　健次 (電総研)	樋口　哲也 (電総研)
松田　秀雄 (神戸大学)	松本　明 (三菱電機)	
長沼　次郎 (ＮＴＴ)	平田　圭二 (ＮＴＴ)	佐藤　正俊 (沖電気)
仲瀬　明彦 (東芝)	横田　治夫 (富士通)	中川　貴之 (日立製作所)
中尾　浩一 (シャープ)	中島　克人 (三菱電機)	

内山　俊一	吉岡　勉	近山　隆	後藤　厚宏
石川　篤	市吉　伸行	山本　礼己	木村　宏一
新田　克己	稲村　雄	六沢　一昭	杉野　栄二
越村　三幸	寿崎かすみ	今井　明	川合　英夫
星田　昌紀	赤尾杉　隆	大西　諭	松本　幸則
伊藤　文英	大嶽　能久	清原　良三	高木　常好
屋代　寛	近藤　誠一		

プログラム

10月6日(金)

1. 開会の挨拶　　田中 英彦 (東京大学)

2. ここまで分かった... と思う。　　14:00 ～ 15:30

座長　山中 英彦

2.1 PIMのアーキテクチャとクラスタ内処理系
後藤 厚宏 (ICOT)

2.2 KL1分散処理系の実装経験
中島 克人 (三菱電機)

2.3 ユーザと並列推論マシンの間、そして日頃気になっていたこと
市吉 伸行 (ICOT)

3. 私はここまで分かり、ここが分からないと思う。
第1部：現在の問題点、[理想と現実]　　16:00 ～ 17:30

座長　後藤 厚宏

3.1 PIM/pのネットワークに就いて
久門 耕一 (富士通)

3.2 PIM/cのご紹介
中川 貴之 (日立製作所)

3.3 私はこんなことしか分かっていないしこんなことも分からない
---どうすりゃいいの負荷分散---
中島 浩 (三菱電機)

4. 私はここまで分かり、ここが分からないと思う。
第2部：並列処理の問題点、[私の興味、実感、そして夢]　　19:00 ～ 21:30

座長　雨宮 真人

4.1 並列処理の問題点 ― 私の興味, 実感, そして夢
富田 眞治 (九州大学)

4.2 並列処理の問題点
柴山 潔 (京都大学)

4.3 並列処理の問題点、[私の興味、実感、そして夢]
小池 汎平 (東京大学)

4.4　PIM のここがわからない

小長谷明彦（日本電気）

4.5　並列計算機における静的と動的

坂井　修一（電総研）

4.6　並列処理の問題点 － 私の興味、実感、夢

樋口　哲也（電総研）

4.7　超 (OR) 並列推論のための基本アーキテクチャと負荷分散アルゴリズム

長沼　次郎（ＮＴＴ）

4.8　昨日 (きのう) まで、そして今日 (きょう) から

六沢　一昭（ＩＣＯＴ）

10 月 7 日 (土)

5. これまでと今後　　9:30 ～ 12:00

座長　　富田　眞治

5.1　2 年前 → これまで → これから

近山　隆（ＩＣＯＴ）

5.2　これからの知識プログラミングと並列記号処理

雨宮　真人（九州大学）

5.3　並列処理のこれまでと今後

田中　英彦（東京大学）

6. 閉会の挨拶　　田中　英彦（東京大学）

ここまで分かった．．．と思う。

PIMのアーキテクチャとクラスタ内処理系 と題してはみたが...

後藤 厚宏

1 はじめに: 年寄りっぽく昔を振り返って

夜明け前...

8月の暑い日の午後に, ICOTに足を踏み入れて早4年と2カ月. ICOTにおいては, ずっと「PIM」, 「PIM」, ...とばかり言い続けているうちに[1], 「ここまで分かった ...と思う。」と言うべき時期がきてしまった。

すこし, 昔を振り返ってみる。ICOTのPIMに関わりをもったのは, 学生時代(ICOTができる前の年: 1981年頃), ICOTの準備委員会のひとつをお手伝いすることになった時からで, 「第5世代のコンピュータ, データフローマシン / データベースマシンの調査研究」[2]という報告書をまとめたころからはじまる。その委員会の主査は田中先生で, 委員の方には, 雨宮さん(先生), 長谷川さん(室長), 喜連川先生, 北大の田中先生ほか, 沖の伊藤さん, 三菱の宮地さんと, 今で言うところの「いつもの人」がいらっしゃった。

当時の図式は,

(広義) データフローマシン — 並列推論マシン
関係データベースマシン — 知識ベースマシン

であった。私としては, 構造メモリや発火機構等, 要素となるようなメカニズムそのものが面白かったように思う。ICOT前期からのPIM研究の流れをみると, 確かに, 上記の図式が, 当時の研究者にとって研究の取り掛かりの場所を示していたことがわかる。

これと並行して, 私自身, 並列推論エンジンPIEなるものの研究を始めていた。実のところ, 修論の仕事をやりながら, 「早々に就職をして 金稼ぎに走ろう」と考えていたところ, 元岡先生から「もう3年残っていけよ。電総研の内田君というのが推論マシンというのを作ろうとしているんだけど, 君も対抗してやってみないか」と進められたのが事の始まり。(今になって, こんなにも, 特に内田さんと関わりを持つようになるとは予想していなかったけど)

流石に, 8年前と比べれば, 「ここまで分かった」と言えるような気がするが, 当時は, Pure Prologのインタプリティブな並列実行が主流であったため, 当り前かも知れない。ふと自分の論文をめくってみると, その結論の冒頭には,

> The logic has a long history. The logic programming is now in the second decade from the first Prolog implementation was built in Marseille. In constrast the research for high performance inference machines is a newcomer in the broad area of computer systems. Our first concern was "Will the high performance inference machines be realizable in near future?". The author firmly believes, "Yes", at present.

という青い英文があった。今も気持ちは変わらないが....

---

[1] ただし, 一つもPIMのハードが組み上がらないうちに, PIMの接頭語や接尾語が「中期」から「PIM/x」, いつも間にか「後期」と変わってしまったが。

[2] 報告書の最後の節にある, 1000 PUのPIMを作るために必要な研究者数を前期, 中期, ...の時期毎に線表的に書いてあるところは, 私がいい加減に書いたものである。必要な研究者1000人, 技術者1000人, とでも書いておけばよかった, と今になって思う。

創世期?

ICOTの中期の1年目は，前期PIM[3]のまとめと称して，どんなPIMを作ろうか? とみんなで悩んだ。何人かの人は，時には三田国際ビルの向かいのマンションの一室[4]で，濃コーヒーやコーラを片手に何時間も議論したことを覚えているだろう。それらの議論の中で，まず最初に掲げたスローガンとは:

1. 自分が使うためのPIM

2. 言い訳しないPIM

という，一見学術的でない，しかし，今でももっとも重要(かつ難しい)と信じているものである。

自分で使うためのPIM，または，自分で使いたくなるPIMは，計算機を使って計算機の技術を開拓しようとする我々にとって当り前でありながら，現実的には(アーキテクチャの研究をしているつもりの人間には)非常に難しい目標である。なぜなら，アーキテクチャ[5]をソフトウェアとハードウェアのインタフェースと考えた時，新しいアーキテクチャを示すには，新しいソフトウェアと新しいハードウェアの両方を何らかの意味で示さねばならないためである。当然，全てを一度の提示することは現実的に不可能に近い。多くのアーキテクチャ研究者は，新たなアーキテクチャを提示してみたいという夢をそっと胸の奥に隠しつつ，恥を忍んで目前の林檎ならぬ(シバシバ論文の冒頭に見掛ける)従来の計算機に磨きをかけるか，または，無意識の内にいろいろな言い訳を用意しつつ，新たなアーキテクチャを探す旅に出歩いてしまう。

ますます学術的でない第3のスローガンというより認識は，

3. 並列推論マシンは，数人でチョコッと出来上がる程ど簡単ではない，

であった。つまり，"人手が足りなくてできなかった" と後になって言い訳しなくても良いように，集められるマンパワーは集めるべきである[6]，という考えである。

2 議論: いろいろ

クラスタ構造

今作られつつあるPIMの共有メモリクラスタによる階層的構成のイメージは，1985年から1986年にかけて，沖の伊藤さんや日立の杉江さんとのディスカッションから固まってきたと思う。その考えは，"PIMのハードウェアは，そのプロセッサ台数についてのピークが数100台程度でも良いから，数年後の技術によって高い性能が得られるべきもの" というものであった。(私としては)プロセッサ台数に関して普遍的に適応できるハードウェアの構成方式を求めるという大それた考えを捨てようと考えた。

今日，10-20個のプロセッサを持つ並列計算機が身近に使えるようになり，直にでも，数100台，数1000台のプロセッサを持つ並列計算機も実現できそうに思い込み易く，僅かに，ハードウェアとしてはそれに近いものも既に存在する。しかし，現在の並列処理の使い方がほんの狭いところに制限されていることも明らかである。本音を言うと，プロセッサ数が10程度でも，並列推論マシンは十分に作り甲斐があると思っている。10個のプロセッサだけとしても，それを全体として自由に使いこなす術を見つけようとすることは，1000プロセッサのハードウェアを組み上げること以上に難しいと思うためである。

一方，ほんの限られた使い方しかできないとしても，並列処理の便利さが実感できていることも事実である。例えば，Symmetry上の並列makeを使っていると，いつまにか(大きめの?)プログラム開発の手順が変わり，気軽にmakeをするようになる。これは，今30才以上の人が，UNIXのviを，さらにはワークス

[3] PIM-D, PIM-R, 縦分けマシンというのがあった事さえ知らない人も増えてきたでしょうね

[4] 当時は，"サニーの会議室" と称してしばしば利用していた。

[5] そもそも，近年，アーキテクチャの定義を一言で言い難い，少し前なら，"IBMxxx の命令セット" と言えば良かった?

[6] 今でも人手は足りないが，悲観をこぼしたり，言い訳する，という状況ではなく，"みんなで頑張ったのだから胸を張ろう" と前向できると思っている。

テーションを使い出したときの喜びに通じるものと思う。何かに秀でた，並列推論マシンの場合は，ある局面でも良いから高い性能が得られれば，それは計算機技術における一つのインパクトになり得る，のである。

要素プロセッサ: CISC か RISC か

最近は「CISC か RISC か」の論争も次第に熱が冷め始め，「それぞれに適合する分野があるね」とか，「今後は双方が次第に歩み寄るようになる」という（海外においてさえ日本人が得意なような）結論にまとまることが多くなってきた。PIM のアーキテクチャ設計でも，その要素プロセッサは，CISC 的（実質上はマイクロプログラマブル）であるべきか RISC 的であるべきか，を議論した。結局，それぞれの PIM/x は，直接担当する人間の意思（と，今利用できるハードウェア技術，という現実的な理由）により，RISC 的，CISC 的，または両者を融合したプロセッサハードウェアを作ろうとしている。

KL1 の特性を考えた場合，私は PIM/p の:

RISC+CISC のアプローチ

が面白いと思っている。ただ，これも性能面では些細な話かもしれないし，本人がどこで楽しもうとするかという趣味の問題かもしれない。研究として意義のあることをやろうとした場合，それ以上に重要なことがあると思う。

CISC, RISC より DISC を

研究としてのアーキテクチャ設計では，思いきりの良さ が最も重要である。何を中心に置き，何を切り捨てたか，が明確であってこそ，その研究によって何が分かったのかが言えるようになる。狙いがはずれるとネガティブな結論となってしまうが，それでも 十分に次の研究に役立つ。これは，色々な人との話で聞いてきたことであるが，最初の内は同意しつつも実感が伴わなかった。今は，もっと思い切れるところがあったのではないか，と反省している。RISC の考え方にしても，1 チップ化とかコンパイラ主導，といったところへの思いきりがベースであろう。

F 社の久門氏との議論でできた話であるが，並列推論マシンのプロセッサアーキテクチャ研究が目指すべきは，RISC ではなく（といって CISC でもなく），DISC (Dedicated Instruction Set Computer) なのではないか，と思う。これは，KL1 の処理の一部に Dedicated な，という狭い意味ではなく，何かに明確に的を絞ったプロセッサというものである。

今の PIM の場合，狙った的の一つはガーベジコレクションと言えるだろう。国際会議や論文誌を中心に，これまでの PIM 関連の発表論文を見直してみると，その多くがガーベジコレクションに関するものであることからわかる。

- MRB scheme: Chikayama et al. Inamura et al.
- LRC scheme: Goto et al.
- Piling scheme: Nakajima
- コピー GC の並列実行: Sato et al.
- 分散 GC: Ichiyoshi et al.

この理由は，前世代において LISP 文化に染まった過去を持つ人や，近山教に感化された人が多かったから，と見ることもできる。勿論，並列推論マシンの原点はメモリ操作にあり，という当然の認識に基づいているためでもある。

Trivia Pursuit? ガーベジコレクション

確かに，ガーベジコレクションは楽しめる。ガーベジコレクタの設計は，言わば，鏡に写しながら言語処理系を設計することに近いためである。特に，他の研究グループと議論をしたりしたとき，ガーベジコレクションの立場から言語処理系を見ていることが，自分達の議論の支えになっていることに気付く。

ユーザとは，自分も含めて極めて保守的であり，これまで得られていた，これまで当り前であったものを失うことに大きな抵抗を示す。当然，配列の更新の手間が配列の大きさに比例するような言語処理系を使おうとするはずがない。この意味で，MRB 方式のように，狭い意味でのガーベジコレクションの役割だけでなく，(MRB が凸，つまり，単一参照であれば) 定 (低) コストで配列の要素の更新やストリームの併合を可能とするご利益をもたらす方式は，論理型言語のように効率面でハンディを背負う言語を，普通の言語として使うことを可能とする。

もっとコンパイラを

KL1 マシンとして素直に的を絞ろうとした時，コンパイラの役割が重要である。コンパイラと処理系の役割分担も，開発者の頑張りによって 大部わかってきた。インデキシング，MRB 管理，等，コンパイラの頑張りによって実行時処理系は軽くなり，処理系としての焦点を絞り易くなってきた。これからも，コンパイラが頑張れる部分はたっぷりあると思う。また，実行時処理系に比べて，開発する上でのパフォーマンス / コストも良いであろう。ただ，残念ながら，ICOT の開発グループのコンパイラ担当者はまだまだ少ない。少なくとも，“これはコンパイラで出来るのではないか?” と考える人がもっと増えるべきである[7]。RISC や VLIW の研究者 (特に米国) が “幾つもコンパイラを作って実験した” と自慢しているのが，羨ましく思う。

言語処理系の開発

前回の PIM ワークショップで中島 (克) 氏が「マイクロプログラム方式は本当に便利である」と主張していた。その柔軟性は研究用マシンとして捨て難いと言う理由である。現在，PIM/x のための処理系を開発しているものにとって，言語処理系の開発を柔軟に進めることのできる枠組が並列推論マシンの研究開発において非常に重要であることをヒシヒシと感じる。ただし，マイクロプログラムと KL1 という 2 階層の枠組では，マイクロプログラム開発の規模が大きくなり過ぎてしまい，現状の，師弟関係による無形文化財の伝授に近い形態[8]では，既に限界に近いとことまで来ているように思える。

実際，KL1 の並列処理系を作ろうとする時，色々と悩ましい局面にぶつかる。例えば，ある機能を，ハードウェア，マイクロプログラム，KL1 によるライブラリ的なプログラムの間でどのように役割分担して実現するか，という悩みである。まずは KL1 で，というのが常套手段であるが，ユーザの要求によって，より高速に実現しようとすると，次はマイクロプログラムで，ということになる。簡単な組み込み述語程度であれば，マイクロプログラマの頑張りで対処できる (実際，されていた) が，例えば，処理系全体の改造に相当するようなデバグ機能の追加やゴールの優先度制御機能の変更になると無理に近い。

最近，実際に悩んでいる優先度制御について言うと次のようになる。現 KL1 仕様では，ゴールのスケジューリングにおける優先度制御の手段をプログラマに提供している。このような優先度制御により，プログラマは，時として，処理最自体を節約できるようなプログラムが書けるようになったりすることも分かってきた。処理系作成者は，全てのプログラマがそのような優先度制御の機能を使いたがるのであれば多少のコストがかかっても優先度を忠実に守るような処理系を提供しようと努める気になってくる。特に，メモリを共有する PIM/x のクラスタの中では，相当忠実に優先度を守った制御も可能である。しかし，優先度制御を利用しないようなプログラムにおいてもそれなりの実行時コストが増えてしまう。できれば，優先度制御

[7] 白状すると，私はコンパイラを作ったこともないし，基本的な作りも最近になって聞き知った程度である。

[8] 中 一 稲 一 次 を言っているつもりはない。

方式の異なる言語処理系を作ってみたい、ということになる。さらに、優先度制御のように、アプリケーションによって要求が異なるものについては、将来においてもユーザが選択できることが望ましい。しかし、巨大なマイクロプログラムを入れ替えるとなると、人の資源、特に「元気さ」という資源に限りがある以上、"出来ない相談"となってしまう。

一方、これまでに、ハードウェアから積み上げてきた並列処理研究が期待はずれのまま終わっていったことも多い。その第1の原因は、処理系、ソフトウェアを積み上げていく途中で息切れしてしまったことにあると思っている。たとえ、否定的な結論であっても、しっかりとした積み上げた実験において得られたものであれば十分有益である筈である。この意味で、今後の並列推論マシン研究において重要なのことは、多分たっぷりあるであろうPIMの欠点を十分に吐き出せるまで使い熟すことにある。

言語処理系を作るための言語処理系: PSL/VPIM

PIM/xに向けて作ろうとしているPSL/VPIMの枠組は、KL1とマイクロプログラムの距離を埋めるための、言わば、言語処理系を作るための言語処理系であり、実行時処理系を作り替える元気さを補うことを目的としている[9]。現在のものは、"高機能マクロマイクロアセンブラ"のようなレベルであり、まだしっかりとした第3の層と自慢できるようなものではないが、KL1のような高級な論理型言語の並列処理を進める上では、その重要性は今後益々高まると思っている。

言語処理系は軽く作られるべき: PSL/VPIM不要論

ここまで、書いてみたところで、反対の主張も言ってみたい。即ち、"PSL/VPIM"のようなものは不要になるべきである"、とも言える筈だ。PSL/VPIMのような言語処理系開発ツールを色々と作らねば開発できないような複雑な言語処理系は"本物"にはならない。言語処理系は"軽く"あるべきで、もし、軽い処理系ができないようであれば、言語(この場合はKL1)、または並列処理そのものに問題があるはずだ、という主張である。

"KL1にジェネラルユニフィケーションは必要か?"とうような議論もあるし、また、(もともと論理型言語ではないが)ユニフィケーションや多重参照がないA'UMAはこのような処理系としての軽さを目標の一つとしているのかも知れない。ただし、メタ機能、スケジューリング制御、メモリ管理あたりになると、その複雑さの原因が、言語仕様によるものなのか、並列処理によるものなのか、はっきりしなくなる。

将来の方向としては、上位のプログラミングからの要求を(ほぼ)満たしつつ、第3の階層を必要としない、軽い言語処理系が作れるような言語とあくまで軽い処理系、を目指したいと思う。しかし、これは今の枠組を十分いじくり回した結果として分かるもののはずである。

3 おわりに

今、並列推論マシンに求められているのは、並列処理を如何にして、普通に、当り前のように提供するかにある。並列推論マシンを作るということは特殊な計算機を作ることではない。逆に、これまで"汎用"と呼ばれていた計算機は真に汎用なものではなく、実際にはその応用範囲が限られてきたと考えるべきである。つまり、計算機の適用範囲を広げるという意味において、並列推論マシンの開発は、より汎用な、または、より強力な計算機を作ることと考えてよい。並列推論マシンとは、単にゲーム木の探索を行うだけの計算機ではなく、これまでの計算機が持っている多くの機能も兼ね備えていなければならないからである。このように、並列推論マシンの研究開発は"計算機システム全体の機能と性能を強化する努力"と考えるべきであろう。

まとまりの悪いことばかり書いているうちに締切時間となったしまった。

---

[9] 当然、最初の処理系を作り上げねば話にならない。

1989年9月26日
三菱電機　中島克人

KL1 分散処理系の実装経験

1　はじめに

KL1 分散処理系はスケーラブル大規模並列マシンを効率良く動かすことを目標に設計され、マルチ PSI 上に実装された実用（商用ではない）並列言語処理系である。想定マシン規模は、疎結合ノード（PIM でいうクラスタ）の数で例えば 1000 かそれ以上というのが、研究開発メンバの暗黙の了解であったと思う。

2　はじめに（Part 2）

マルチ PSI 上に KL 分散処理系を実装するため、方式設計に 1 年、実装に 1.5 年余り[1]の歳月を費やした (今も改良を続けている。マイクロ容量は約 12K 語／53 ビット）。方式設計の期間中に、簡単なベンチマークプログラムが走る程度のプロセッサ（PE）内実験処理系を試作したが、これには 2 か月程度しか掛からなかったことと比べても分散処理系が如何に労作かが分かっていただけるであろう[2]。

D.H.D. Warren 博士がある場所で言ったそうである。「マルチ PSI の処理系？ああ、あの複雑なヤツか。(もちろん英語で)」 確かに複雑である。取りまとめをしていた筆者自身、常に頭に入れていた仕様の量は恐らく全体の 40% ぐらい（いや 20% かな）であったろう (筆者の頭の悪さは割り引いて下さい)。複雑になった理由は幾つかある。

(1) 疎結合マシンへの本格的分散処理系であるので、効果が定かではなくとも色々なアイデアを実装し、それを評価するという立場をとった。(動かす自信があった。シミュレータでは動かせない大規模かつ実際的な状況でしか評価出来ないと予想されるものが多かった。)

(2) OS の機能を実現するに際し、KL1 では記述が難しいもの（メモリ管理）やパフォーマンスに大きな影響を与えるもの（実行資源と実行優先度管理）は処理系でサポートすることにした。

アイデアにしろ、システムにしろ、「シンプルなものが生き残る」等とよく言われるが、筆者自身も実際、この処理系は「生き残るには複雑すぎる」のか、「実用的な分散処理系と言うものはこの程度の複雑さが最低限必要」なのかの判断をつけかねている。これは、並列言語の仕様や疎結合 (非メモリ共有) 並列マシンとしてのサポートハードなどへの見直しも含め、じっくり見極めたいものである。

本資料では、KL1 分散処理系における特徴と処理系の評価結果の一部を示し、複雑かどうかの判断も含め、皆さんの評価をいただくための材料の一つにしたいと思う。

---

[1] 両者はオーバラップしてるが。

[2] 国家プロジェクトだから、それだけの人とお金を掛けられたのだというご批判もあろうかと思うが、ともかくも我々はやり遂げたのである。

3　KL1 分散処理系の特長

色々自慢したいことがある。研究室レベル（大学関係者の人には失礼！）のマシンではなく、相当真面目に (1)「実用性」 を考えてきた。しかし、まだまだ不十分なところがある。(2) マシンをスケーラブルにする ことにも相当留意している。最後に（これは自慢ではないが）、前述のように、(3) 本来 OS の機能の一部であるものを処理系で直接サポートしている こともPIMOS のプログラムを綺麗にするとともに、まともなパフォーマンスを達成することに貢献している[3]。OS サポートの件についてはここでは省略する。

3.1　実用性

「実用性」には色々あるが、KL1 分散処理系（および PIMOS）の現状としては以下のようなところであろうか[4]？

(1) 並列プログラムが書き易い？
KL1 言語自身の話を別にし、KL1 分散処理系として見るなら、プラグマと荘園がユーザの思い通りに使えるかというと言う点であろう。とくに優先度プラグマが気になる所である。32 ビットで表わされる論理優先度を標準では 12 ビットの物理優先度に変換して実現している。また、PE 毎にローカルに優先度を管理しているので、厳密なスケジューリングになっていない。これらの評価はまだ行なっていない。

(2) デバッグがしやすい？
DEC-10 Prolog システムのデバッガやスパイ機能に相当するものはある。並列なゴールをトレースするため若干の工夫はある。トレース中でも実行速度はそれほど低下しない。ビジュアルなデバッガはまだない。パフォーマンスバグをとるための仕掛けもこれからの課題である。

(3) 操作性が良い？
KL1 分散言語処理系や CSP に関しては「???」である。PIMOS は今後もドンドン良くなるであろう。

(4) 保守性が良い？
「???」である。マシンの物理構成を変えるのには、ハード保守員の動員が必要。論理構成の変更は初期化ファイルの変更だけで可能。

(5) システムとして健全である（ダウンしにくい）？
例えばある PE でメモリが足らなくなれば、その PE だけで局所 GC が行なわれる。それでも駄目な場合は大域 GC が必要であるが、現在未実装。また、現状では外部参照ポインタ（PE 間を渡る参照ポインタ）の数がある固定数を越えるとダウンするが、これは近々改良予定。計算途中での PE のハードエラーは回復不能。（論理構成を縮退して再立ち上げ。）

---

[3]「ファームウェアで OS を書くなんて！」という方法論としての反省・批判もあるだろうが、ファームウェアで 1 年少々で出来る程度のものだから、という見方もあろう。

[4]あまり自慢になっていないが....

3.2 スケーラビリティ

処理系の方式検討を行なうに当たり、最も配慮したことはPE間通信を減らすことであった。特にPE数に比例してメッセージが増加してしまうような仕組みを排除することに留意した。システム全体を止めてしまう大域GCも極力避けることを考えた。また、非メモリ共有であることから、各PEが無駄なデータを持たないようにも留意した。工夫の数々を以下に示す[5]。

(1) 里親を用いた荘園管理:
荘園の管理情報を各PE上の里親で分散管理することにより、荘園の存在するPEへの通信の集中を防止した。

(2) WTCによるゴールの終了判定:
ある荘園内のゴールの終了判定を効率良く行なうため、重み付き参照カウント（WRC）の応用である重み付きThrowカウント（WTC）を導入した。荘園から借り出されたWTC[6]はPE間を飛び交うゴールによって運ばれ、各PEの里親に蓄えられる。里親は自PE内にゴールがなくなると終了し、蓄えられていたWTCを荘園に返却する。この方式により、PE間の通信にディレイがあるマルチPSIのようなシステムにおいても安全かつ効率良く、すべてのゴールが消滅したことを検出することができる。

(3) 局所GCを可能にする外部参照ポインタの管理:
「輸出表」を用いてPE外部からの参照ポインタを管理し、単一PEによるGC（局所GC）を可能にした。

(4) WECによるPE間即時GC:
大域GC頻度の抑制のため、外部参照ポインタに関する参照カウントGCを必須と考えたが、カウントのためのPE間メッセージの頻度が大きいことや、カウントのレーシングの問題があった。やはり重み付き参照カウントの応用である重み付き輸出カウント（WEC）方式により、これらの問題を解決した。WECを貯えるため、参照側には「輸入表」を設けた。また、これにより複数回受け取った同じ外部参照ポインタを本にまとめることも可能にしている。

(5) 構造体グローバル管理:
プログラムコードに関しては、必要なPEに必要な時にロードするという方針（オンデマンド・ロード）とした。そこで、色々な経路（PE）から同じプログラムコードのコピーを重複して受け取らないように、そのような構造体データにはシステムに一意なIDを付与して、それを管理することにより、コピーの重複を防止した。

4 評価

PE間の通信量低減のため、色々な工夫を凝らしたわけであるが、果たしてそれらの元がとれるかというのは是非知りたいところである。しかしながら、計算機動特性評価における多くの場合と同様、PE間の通信量はプログラムに大きく依存するため、「このようなプログラムではこうなる」という言い方しかできない。つまり、

[5] こちらは少し自慢になるかな？

[6] 「WTC」を「おかね」と読み変えていただくとわかり易い

コスト (Overhead) = 単価 × 回数

であるが、回数の方はプログラム依存であり、負荷分散依存であり、実行依存（実行毎の非決定性）である。

本章ではこのPE間通信の単価の計測結果と、中規模実験プログラムにおける通信コストやネットワークのトラフィックの測定結果を示す。

4.1 PE間通信処理の単価

3引数のゴールの投げ出し（throwメッセージと称する。この場合は65バイトのメッセージ）に約85マイクロ秒、外部参照ポインタの値を問い合わせるメッセージ（readメッセージと称する。この場合は14バイトのメッセージ）の送出に約25マイクロ秒かかることがわかった。それぞれのメッセージの受け取りにも同じ程度掛かっている。それぞれの所要時間のうち約40～50％はネットワーク制御ハードを操作するためであり、残りがメッセージの内容に依存する処理、すなわちメッセージの種類を判断したり、KL1のデータとしてどのようにパケットにエンコードすれば良いかを判断したり、輸出入表を操作したりするコストであった。

ネットワーク制御ハードの能力としては例えば65バイトのメッセージ送出・転送には13マイクロ秒しか掛からないことから、ネックはファームにあることは確かである。

4.2 PE稼働率とPE間通信処理のコスト

OR並列全解探索問題である詰め込みパズル（Pentomino）では、16PE上での実行においては、ソフトウェアによる自動負荷分散の工夫により、平均稼働率は95%に達した。95%のうちPE間通信処理に要したステップ数は4%程度であった。それに対し、レイヤードストリームを用いた並列構文解析プログラムPAXでは、（静的な負荷分散しかしていないせいもあるが）平均稼働率が35％程度と相当に低く、35%のうちPE間通信処理は40%程度も掛かっていた。つまり、本来の計算のための稼働率は35%×60%=20%ということになる。

4.3 PE間通信によるネットワーク・トラフィック

Pentominoでは、16PE上での実行におけるメッセージの頻度が平均7.5Kメッセージ／秒であった。平均メッセージ長の24バイトと、平均メッセージ移動距離[7]を掛け合わせると、システム全体で450Kバイト／秒のメッセージが飛び交っていることになる。これはネットワークチャネル1本当たり9.3Kバイト／秒、即ち、5Mバイト／秒のチャネルバンド幅の0.19％しか使用していないことになる。

PAXにおいてさえ、ネットワーク・トラフィックはPentominoの3倍の0.6%であった。PAXはマシンの平均稼働率が35％程度と低いせいもあるが、たとえ100％だとしても2%程度である。

[7]メッセージの出先・行き先をランダムと仮定。正方形メッシュの場合、性能PE数の平方根に比例する。16PEの場合は2.5。

つまり、今まで試したマルチPSI上でのプログラムでは、PE接続のトポロジの議論など全然関係ないトラフィック量であった。もっとも、マシンが10000台規模になると平均メッセージ移動距離の関係で[8]、もはや安心できないレベルになる[9]。

4.4 考察（疎結合マシンは生き残る？）

PAXでのように例えアイドル時間とPE間メッセージ処理のコストの合計が全体の8割を占めようとも、スケーラブルであることは捨て難い。PEの台数に関わらず2割の計算時間（計算稼働率）が確保できれば十分であろう。1000PEで200倍のスピードアップなのだから。しかし、負荷分散がうまくいかないと、通信量が増えるばかりでなく、メッセージ待ちによる稼働率の低下もあるため、計算稼働率はPE台数の増加に伴い低下することを忘れてはいけない。これは特に疎結合並列マシンの弱みである。

個人的な感想としては、台数効果がPE数の平方根以上なら、満足しようと思っている[10]。

5 その他の感想

5.1 デバッグのためのサポートはエンジニアリングとして軽視出来ない

高速な実行と柔軟なデバッグサポートには必ずトレードオフが存在する。処理系の上のソフトだけでインタープリタを書いたりすると、デバッグモード時には通常に比べ100倍以上も遅くなったりするため、今時のユーザは許してくれない。そこで、デバッグのための仕組みは処理系の内部に組み込むことになる。KL1分散処理系ではシステムの完成に至るまでの各レベルに応じ、種々の仕組みを実装した。そのなかでも、PIMOSのリスナ／デバッガで用いる機構としてサポートした、トレース例外とスパイ例外は重要である。これは、指定したゴールおよびそのサブゴールに対し、リダクション単位のステップ実行とスパイ述語のチェックを行なうためのものである。スパイ機能は現在実装中であるが、例えばこれに代わり、PIMOS側のソフトで行なうと[11]、通常実行の10000分の1程度に速度低下するのに対し、処理系サポートによると50％以下のオーバヘッドで済むことがわかっている。

5.2 擬似 (Pseudo) マルチPSIはヒットであった

身近にマシンがあると言うことが重要であることを再認識させられた。実機のマルチPSIは擬似マルチPSI上でデバッグが終わり、台数効果などを測定するためだけに使われている。実機マルチPSIでしか動作しないような大きなプログラムがまだ開発されていないのが大きな理由の一つかも知れないが、例え大きなプログラムを開発するとしても、そのサブモジュールの開発はやはり身近にあるマシンを使用したくなるであろう[12]。

---

[8]例えば、4×4=16PEが100×100=10000PEとなるとトラフィックが25倍となる

[9]もちろん、通信の空間的または時間的ホットスポットがあると状況はもっと悪くなるし、通信の局所性が出れば状況は良くなるので、単純に計算するのは危険であるが。

[10]贅沢かな？

[11]トレース例外を用いたステップ実行によりリダクション毎に呼び出し述語をソフトでチェックする

[12]各人のPSIからマルチPSIをリモートアクセスするための機能は現在開発中であるが、立ち上げや障害時のことを考えると.....

擬似 (Pseudo) マルチ PSI のための機能追加やそのデバッグは KL1 分散処理系の開発の 50 %程度のウェートを占めた[13]が、擬似マルチ PSI を先行開発したおかげで、実並列マシンで予想されるハードバグと処理系以上のバグの切り分けの苦労の殆どを回避出来たこともヒットだと思う。

[13]担当者によって大幅に偏りがあるが、取りまとめをした私の苦労感から見た値

ユーザと並列推論マシンの間，そして日頃気になっていたこと

市吉 伸行 (ICOT 4 研)

概要

筆者はFGCS'88までマルチPSI上KL1処理系開発が主な仕事であったが，現在は並列プログラムをマルチPSIのような並列マシンにどのようにマッピングしたらよいかが筆者の問題意識の中心になっている．本稿は2部からなり，第1部では，ユーザ(応用問題を持っている人)が並列推論マシンで問題を高速に解くプログラムを書くに当たって，どんな問題があり，どんな助けがあったらいいか，について述べたい．第2部では，日頃ぼんやりと気になっていることを言葉にしてみたい．

第1部: ユーザと並列推論マシンの間

1 並列推論マシンの平均的ユーザ像

並列推論マシンの“平均的”ユーザ[1]像として思い描くのは，次のような人である．

(a) 解くべき中-大規模の問題を持っている．

(b) 問題の解き方が分かっている．[2]

(c) より大きな問題を解くため，またより速く解くために並列化することに吝かでない．

(d) 必要に迫られればKL1でプログラムを書いてもいいと思っている．

(e) 並列推論マシン(マルチPSI, PIM)を使える環境にある．

囲碁対局システムはこれに当てはまっていると言えよう．

2 平均的ユーザにとって何が厭か

KL1で問題から高い並列度を抽出した，アルゴリズムとしても良い，「よしこれは速くなるぞ」と張り切るが，並列推論マシンで並列に動かすには負荷分散を考えなくてはならない．そして，それが簡単でなくて，プロセッサを沢山使っても思うように速くならない．クサル．

というのが，悪いシナリオである．根本的な原因は，ユーザの思い描く並列性が実際の並列マシンに期待通りにマッピングされ難いことにある．多くの場合，ユーザの思い描く並列性は，(1) プロセッサが無限にあって，(2) それらが全てのデータに小さな時間でアクセスできるという理想的モデルである．(1)については無理なことは承知しており，プロセッサ数が増えて行くに従って台数分に近い高速化が得られればいいと考えている．小規模密結合並列マシンにおいては(2)は大体成り立っているので期待は余り裏切られないが，大規模並列マシンではデータアクセスが一様に小さい時間でできるという近似は全くできなくなるので，理想と現実とのギャップが大きい．

逐次マシン上で効率の良いレイヤードストリーム手法や自然言語パーサPAXがマルチPSIで高速化し難いのは典型的な例である．

3 並列推論マシンとユーザの仲介者の仕事

ハード屋さんと処理系屋さんの血と汗と涙の結晶である並列推論マシンと善意の[3]ユーザとが，このような不幸な出会いをしてしまうことを極力避けようというのが，仲介者の務めである．

具体的には，ユーザが悪いアルゴリズムを使っていないことを確認した後，KL1の良いプログラミングスタイルに従っているか調べ，並列推論マシンへの良いマッピングの仕方を考えてあげることである．「プログラムのマッピングの仕方によっては結構速くなるよ．ちょっと面倒なのは我慢して頂戴．最初のCray-1だって，メモリにECCがなかったり，ベクトル演算のチェイニングがなかったって話ですよ[4]」と言えれば仲介者冥利に尽きようというものである．

---

[1] 実は理想的ユーザかも知れぬ:-)

[2] 意外と難しい．例えば，統計理解．

[3] 法律用語で，悪意を持たずに — すなわち，無知のために — 何らかの不幸を招く行為についていう．

[4] 日立中研・長島氏による．

4 事態は深刻か

では、どれ位の割合のプログラムが並列化により劇的に速くなるのだろうか. 最適な逐次アルゴリズムの設計を一般的にカバーする理論がないのと同様、並列アルゴリズム, 並列プログラムのマッピングの一般的理論が生まれるとは考え難い. そこで逐次アルゴリズムの時と同様, 色々な種類の問題について良い並列アルゴリズムとそれの並列マシンへのマッピングとを研究開発し、それができるだけ広い問題領域をカバーするようにしていくより他にないだろう.

手始めとして, 4 つの実験的並列プログラム (表 1) が開発されて台数効果が測定された (表 2, 測定条件等省略). なお, プログラムやマッピングが最適に近いという保証はないので, 台数効果については台数効果を上げる易しさの目安くらいに理解して欲しい.

表 1: 実験的並列プログラムの類型

プログラム	プログラムの類型	通信量 / パターン	見込み計算量
詰込みパズル (ペントミノ)	OR 並列全解探索	僅小	ゼロ
最短経路問題	局所性のある分散計算	大 / 局所的	中
自然言語パーサ (PAX)	ボトムアップ全解探索	大 / 不規則	ゼロ
詰碁	ゲーム木探索	小	大

表 2: 実験的並列プログラムの台数効果

プログラム	台数効果	
	16 台	64 台
詰込みパズル (ペントミノ)	15	50
最短経路問題	8	20
自然言語パーサ (PAX)	3	-
詰碁	3~ 10	-

見込み計算 (speculative computation) とは, 後で無駄になるかも知れない計算のことである. 例えば, 詰碁プログラムにおいて, 逐次アルファベータ探索であれば左の枝の探索結果により枝刈りされていたであろう右の枝を左の枝と並列に探索してしまうのは見込み計算である. 並列アルファベータ探索については, プロセッサ数百台の疎結合型並列マシンで 100 倍程度の台数効果を上げた例が報告されている.

4 つの中でいちばん台数効果が上げにくかったのは PAX である. これは, プロセッサ間通信が多いだけでなく, プロセッサ稼働率も悪く, 速度向上を妨げている. PAX (あるいは動的計画法一般) はマルチ PSI にとっては最も苦手な部類のプログラムであろう.

高い並列度のあるプログラムの大半は台数効果の点で, 詰込みパズルと PAX の間に来よう.[5] それらの台数効果を詰込みパズルに近く持ってくるのが今後の研究課題である.

第 2 部: 日頃気になっていたこと

5 スケーラビリティ (台数拡張性)

マルチ PSI のアーキテクチャを説明 (または正当化) する時に, 「スケーラブル (台数拡張性がある)」という言い方をよく使っている. 例えば, 「共有メモリ型並列マシンはスケーラブルでないが, 疎結合マシンはスケーラブルである」などのように. スケーラブルという言葉を, 無意味でなくかつ安心して使えるために, スケーラブルって何? スケーラブルって必要なの? について考えてみたい.

[5] PAX よりも更に困難な問題, すなわち計算負荷が不均一, 通信量が多くパターンが不規則, しかも見込み計算の多い「三重苦」の問題もあり得るが...

5.1 スケーラブルの定義

共有メモリ型並列マシン (より正確には, 共有バス結合並列マシン) がスケーラブルでないと言う場合, 大規模な共有メモリ型並列マシンが物理的に作れないということではなく, 作っても共有バスネックで性能が上がらないということを意味している. つまり, スケーラブルという言葉は何らかの評価基準を仮定している. そこで, スケーラブルと言う概念を次のように定義したい.

定義 1 *Scalable* は *2* 引数の *2* 階の述語であり, 第 *1* 引数として自然数全体の集合 N からあるドメイン D への部分関数 ϕ で無限個の n に対して $\phi(n)$ が定義されているもの ($\phi(n)$ を「サイズ n の ϕ」と言う) を取り, 第 *2* 引数として D から実数全体の集合 R への関数 (*逆評価関数*) を取る. $Scalable(\phi, \rho)$ は次のように定義される.

$$Scalable(\phi, \rho) \stackrel{def}{\equiv} \rho(\phi(n)) = O(1)$$

一般に, n が大きくなるに従って $\rho(\phi(n))$ が大きくなるような ρ を暗黙の逆評価関数 (関数の値が小さいのが好ましいので "逆" 評価関数という) としてスケーラブル云々が語られる.

例 1

$$Symmetry : n \mapsto \text{プロセッサ } n \text{ 台の } Symmetry$$

$latency$ を, 並列プロセッサを定義域とし "平均" メモリアクセス時間[6]を返すような関数とする. この時,

$$\neg Scalable(Symmetry, latency)$$

である.

例 2 $mesh, hyper, butterfly$ を以下のような部分関数とする.

$$mesh : n \mapsto \text{サイズ}n\text{の } 2 \text{次元正方メッシュネットワーク } (n = m^2)$$

$$hyper : n \mapsto \text{サイズ}n\text{のハイパキューブネットワーク } (n = 2^m)$$

$$butterfly : n \mapsto \text{サイズ}n\text{のバタフライネットワーク } (n = 2^m)$$

また, $neighbor$, $netDensity$ を, ネットワークトポロジを定義域とし, ある固定したハードウェア技術で実装した時の, それぞれ, ノード間最短アクセス時間, ネットワークハード量密度を返すような関数とする. この時,

$$Scalable(mesh, neighbor),\ Scalable(mesh, netDensity),\ Scalable(hyper, neighbor)$$

などが成り立つが,

$$Scalable(hyper, netDensity),\ Scalable(butterfly, neighbor),\ Scalable(butterfly, netDensity)$$

などは成り立たない ($netDensity(hyper(n)) = netDensity(butterfly(n)) = \Theta(\log n)$).

上記のような評価基準によれば, メッシュネットワークのマルチ PSI のアーキテクチャをスケーラブルと言うのは間違いでないことになろう.

5.2 何故スケーラブルである必要があるか

第五世代で「大規模並列処理」と言った場合, プロセッサ数が 1000 台規模のものを指しているが, これは 10 台規模の並列マシンの技術 (ディスクリート・アーキテクチャとでも呼ぼうか) を外挿できないことを意味し, 自然数でパラメタライズされるようなアーキテクチャ・スキームを考えなくてはならない. そして, どんなにサイズ n が大きくなっても有効なアーキテクチャはスケーラブルである必要がある.

実際には大規模並列マシンと言えど大きさに限りがある訳で, その範囲内で逆評価関数の値があまり大きくならなければ実用的には困らない. バタフライネットワークやハイパキューブネットワークもかなり大規模なものが実現されている.[7]もっとも, 超大規模となると純粋なバタフライネットワークやハイパキューブネットワークは駄目であろう.[8]

---

[6] これは計る基準が難しいので, 次のように定義される computation-communication-ratio (ccr) を使った方が良いかも知れない.

$$ccr = (\text{Total CPU power})/(\text{Bus bandwidth}).$$

[7] BBN Butterfly (バタフライ結合) は最大構成 512 プロセッサ, Connection Machine (ハイパキューブ結合) は 64K プロセッサ.

[8] 「超大規模」の定義は, 純粋なバタフライネットワークやハイパキューブネットワークが駄目になる位の規模の意!

6 Constant-time Embedding (定数倍埋込み)

論理型言語を手続き型言語と比較する時にしばしば, constant-time embedding (定数倍埋込み) という言葉が出てくる. 手続き型言語処理系で $O(1)$ でできる処理と同値なことを論理型言語処理系でも $O(1)$ でできれば, 定数倍埋込みできたと言う (もちろん, $O(1)$ といっても小さい定数であってほしい訳である). 例えば, 手続き型言語処理系では長さ n のベクタの 1 要素を $O(1)$ 時間で更新できる. もし, 論理型言語に正直に翻訳すればもとのベクタと 1 要素だけ異なるベクタを作ることになろう. その場合, 更新処理に $O(n)$ 時間かかり, 定数倍埋込みできていない. ミュータブル・ベクタを実装すれば, 更新処理は $O(1)$ 時間になるが, 読出し時間が最悪ケースで n によらずいくらでも悪くなってしまう. KL1 処理系の MRB 機構は, MRB 白ベクタの更新処理を, 他に副作用を伴うことなく, 定数倍埋込み可能にしている. 同様に, ストリームマージも MRB 機構によって定数倍埋め込みされる.

手続き型言語の単位処理が定数倍埋込みできないと, 同じアルゴリズムを用いても, 計算量オーダが手続き型言語処理系と論理型言語処理系とでは違ってきてしまう. だから, 定数倍埋込み (時間, 空間とも) は是非とも必要である.

では, KL1 処理系[9]では手続き型言語処理系の単位処理を全て定数倍埋め込みできているだろうか. MRB 黒ベクタの更新に $O(n)$ かかるから単純には No である. また, KL1 処理系ではプロセッサを渡ってベクタ要素にアクセスするとベクタ全体がコピーされるが, これには $O(n)$ のコストがかかる (プロセッサ間の距離を定数として). しかしこれらは, 最新のベクタを複数プロセスが読み書きすることが分かっていれば, ベクタオブジェクト (白ベクタを抱いたプロセス) として実現することにより, 定数は大きくなるが $O(1)$ オーダで埋込める. だから, 定数倍埋込みを議論する時には, データ構造 + 操作としてのデータ型に参照数やプロセッサへのマッピングの仕方まで含めて考える必要がある.

予想 (Conjecture) 1 手続き型言語処理系の単位処理は, データ型の実現の仕方によって何時でも *KL1* 処理系に定数倍埋込みできる. *(必要ならば処理系側の小さな変更を許すとする.)*

予想の系 1 同じアルゴリズムを, *KL1* 処理系では手続き型言語処理系と同じ計算量オーダで実現できる.

7 台数効果

台数効果とは, プロセッサ 1 台での実行時間とプロセッサ n 台でのそれとの比であるが, 並列プログラムをプロセッサ 1 台で実行するのは, 逐次プログラムをプロセッサ 1 台で実行するのと比べてオーバヘッド (スケジューリング, 同期, プロセス切り替えなど) がある. 普通は, 逐次アルゴリズム計算量の定数倍になっていると暗黙の内に仮定しており, それは正しいと思われるが, プログラムによって定数倍の定数がかなり違うかも知れない. 台数効果で見落としてしまいがちな, この並列化オーバヘッドも忘れないようにしたいものである.

8 データフロー並列とデータ並列

並列プログラムのマッピングとは, 並列プログラムの規定する論理的並列性を並列マシンにおける物理的並列性として実現する仕方のこと, と説明しているが, では, プラグマ (マッピング指定) を除いた KL1 プログラムは, どんな論理的並列性も表現できるのだろうか. 平たく言えば, どんな並列言語で書かれたプログラムも同一の並列性を持つ KL1 プログラムに落とせるだろうか. データフロー並列は素直に KL1 に落とせる筈だが, ではデータ並列はどうであろうか. グローバルな時刻を表わすストリームを全プロセスが共有することで, データ並列をシミュレートできそうではあるが, それができても *translation* というより *implementation* という感じである. データ並列でデータフロー並列を模倣するのも同様であろう.

さらに, データフロー (MIMD) とデータ並列 (SIMD) とが本質的に違っているとした場合, それ以外の型の並列性はあるのだろうか.

以上, 乱筆にて.

[9]断って KL1 処理系と言えば, マルチ PSI/PIM 上の処理系を指すものとする.

私はここまで分かり、ここが分からないと思う。

第１部：現在の問題点、［理想と現実］

PIM/pのネットワークに就いて

久門 耕一
（富士通）

PIM/cのご紹介

(株) 日立製作所中央研究所　中川

0　PIM/cとは

PIM/cとはリダクション型パイロットモジュールの別名である。

１００台規模の並列推論マシンを試作する。

中期においては２クラスタ、２PE/クラスタの４PEのモデルを試作する。

1　アーキテクチャ

1）KL1/Bインタフェース

命令セットはICOTが開発したMRB-GC対応のKL1/Bとする。

2）水平型マイクロ

上記を実現するためICOTが開発したKL1/Bの仕様記述(VPIM)を104bit/wordのマイクロプログラムとして格納する

2　制御ソフトウェア

1）VPIMのマシン依存部分の変換／デバッグ用ツール群

VPIMからPIM/cのマイクロプログラムを生成するツール群を開発中である

3　ハードウェア

1）TCMP（密結合）/LCMP（疎結合）の２階層のマルチプロセッサ

a) Snooping Cache 結合TCMP

TCMPのターンアラウンド短縮のために、ICOT推奨の、以下の５状態をとるキャッシュを実装する。

Invalid:	無効なキャッシュブロック
ExclusiveClean:	共有が無く、未変更のキャッシュブロック
ExclusiveModified:	共有が無く、変更されたキャッシュブロック
SharedClean:	共有が有り、未変更のキャッシュブロック
SharedModified:	共有が有り、変更されたキャッシュブロック

b) クロスバーネットワーク結合LCMP

メッセージ通信LCMPのパケットヘッダーのpack/unpackをハードウェアで実行する。

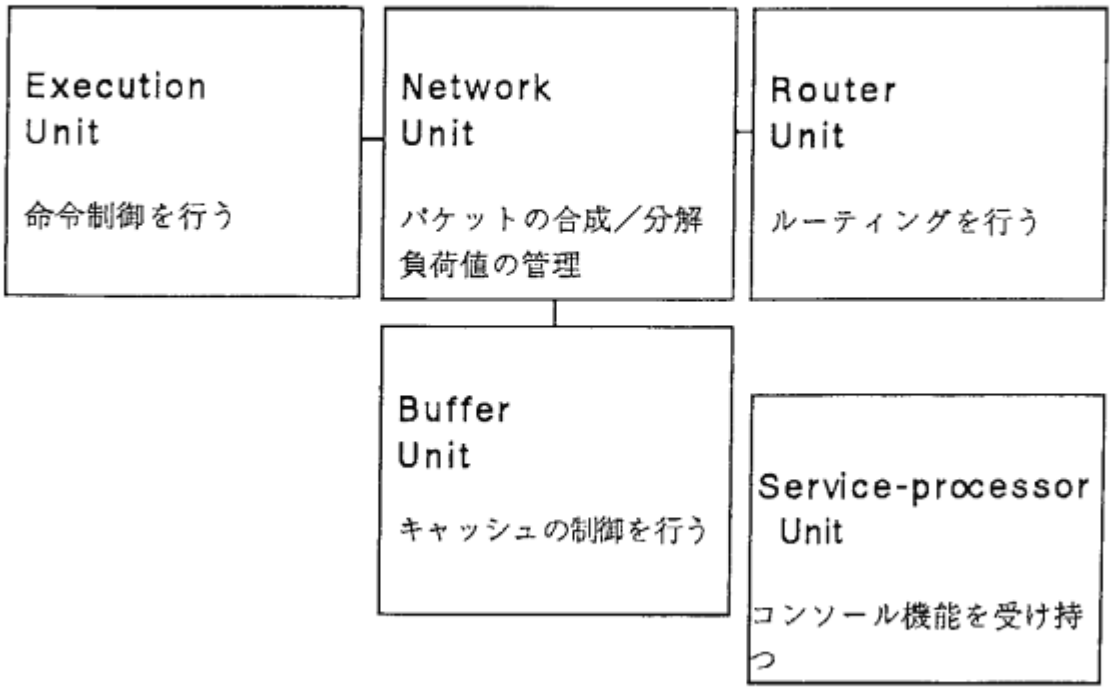
負荷分散の実験用にクラスタ負荷を管理する以下のコマンドパケット機能を備える。

負荷値インクリメント／デクリメント

負荷値セット／リード

負荷要求セット／リセット

PIM/cの論理ブロック構成



PIM/cの諸元表

	実装技術	0.8 μ
	マシンサイクルタイム	５０ns
Eu	命令形式	水平型マイクロ
	word構成	8bittag+32 bitdata/word
	コントロールストレジ	３２kword×１０４bit
	レジスタファイル	２５６word
	汎用レジスタ	１６word
Bu	キャッシュプロトコル	５状態
	キャッシュメモリ	2kcol,2set,2bank,4word
	共有メモリ	16Mword
	共有バス	2wayインタリーブ
Nu	ネットワークバッファ	64word(I),64word(O)
Ru	ネットワークトポロジ	クロスバー

私はこんなことしか分かっていないし こんなことも分からない — どうすりゃいいの負荷分散 —

中島 浩
(三菱電機)

1 はじめに

PIM の重要な研究課題の一つが，単位プロセッサにいかに仕事を割付けるかという，所謂「負荷分散」の問題であることはいまさらいうまでもないことである。この問題に関しては，Multi-PSI 上でプログラムに依存した静的／動的な大粒度のプロセス割付の研究が成されている。また，メモリ共有型のマルチプロセッサで，ゴール単位での割付手法の研究も行われている。しかしながら，傍で見ている印象としては（傍で見ていて良いのかという話もあるが），なんとなく地味な感じがして，もう一つぱっとしないなあ，というのが「現実」である。

一方，「理想」的な話，即ち数百から数千のオーダのプロセッサに見合う大きな問題に関して，「モデル」とか「一般的な戦略」とか「プラグマ」とかいった話は，あの P^3 以外には（少なくとも筆者には）聞こえて来ない[1]。その P^3 にしたところが近山氏から提案された頃には（もう 3~4 年は前のことである）いろいろな議論が成されていたが，ここ 1~2 年は全く棚ざらしの状態になっているように見える（誰が棚にさらしたかという話もあるが）。実際，P^3 に直接関係する問題で筆者に分かったことといえば;

- 特定の P^3 の座標点を担当するプロセッサへのゴール転送を，転送経路上のプロセッサの局所的な情報のみで行うアルゴリズムがある。
- このアルゴリズムでは転送経路上の各プロセッサが 1 辺あたり最大 19 回の乗算と 27 回の加減算（まあとりあえずこのくらい）を行わなければならないので，結構大変である。
- プロセッサの数が多くなければ 1~2 回の RAM アクセスで代用できるが，プロセッサの数が多くなると結構大変である。

くらいであり，その結果 FGCS'88 の論文が 1 つと [1]，誰もアクセスしない SRAM が 4M ビットほど残ったに過ぎない（まあそれだけ残れば良いではないかという話もあるが）。

結局，筆者が中期 4 年間で分かったことと言えば「負荷分散は重要だけれど P^3 はしんどい」ということだけであり，分かってないのは「じゃあどうすりゃいいの」ということなのである。そこで，2 年ぶりに話を蒸し返し（覚えているあなたは偉い!），「どうすりゃいいか」を議論するための種を蒔いてみたいと思うのである。

2 P^3 の美学的評価

「どうすりゃいいか」を考えるにあたって，まず P^3 の得失を評価してみたい。とはいえ，P^3 は実際にインプリメントされた訳ではないので，話は当然定性的かつ主観的なものとなる。

まず，P^3 が「私は美しい」と言っている色々な性質について，それらが本当に必要な美点であるのか，また本当に美しいのかどうかを考えてみる。

[1] PPL (Processing Power *Line*) というよれた話はあるが。

(1) 通信の局所性：
P^3 はプロセッサ間の通信を局所化するような負荷分散戦略をプログラマが記述できるようなモデルである。プロセッサ間通信の局所化の必要性については，改めて言うまでもなかろう。第一，この話がなければ暇なプロセッサに仕事を割付けるだけのことになり，考えるのがつまらなくなってしまう（それだけでも簡単ではないとは思うが）。従って，この美点は心理的に必要であり，かつ P^3 は美しい。

(2) モデルの抽象性：
特定のマシンに依存したモデルについて，それ以外のマシンの関係者の助力当てにするのはそもそも虫が良すぎる。従って，マシンの物理的構造からは独立したモデルが美しいことは，政治／経済的観点からも支持される。さて，P^3 は計算資源がぎっしり分布した平面であり，「ぎっしり」というところが現実的ではないので抽象性はある（現実的でなければ良いと言うものでもないが）。しかし，「平面」と言っているために，P^3 を適用可能な（少なくとも相性がいい）アーキテクチャは限定されてしまう。例えば，Tree はダメであり，共有メモリとハイパー・キューブの2階層などということをやっている人は「わしゃ知らん」と言いそうである。「3次元以下のメッシュで実現できない並列アルゴリズムはダメ」という過激な意見もあるが [2]，ここは八方美人的な態度で臨むのが政治／経済的に得策である。従って，この美点は政治／経済的にも必要であるが，P^3 はさほど美しくない。

(3) 情報の局所性：
プロセッサ間通信を局所化することが目的である以上，大域的な情報を用いて負荷分散を行うのは邪道である。第一，プログラマに通信の局所化を説きながら，裏でこっそりブロードキャストや集中管理を行うという行為は，人倫にもとると言わざるを得ない。P^3 は動的負荷分散という裏作業を，近隣のプロセッサの負荷という局所情報のみで行うことができる。従って，この美点は倫理的にも必要であり，かつ P^3 は美しい。

これらの他に，P^3 が口を濁しているが，こういうモデルに必要な美点が幾つか考えられる。

(4) 記述の容易性：
プログラマに何かさせる以上，その手続は簡単なものでなければならない。また，多くのプログラマが「プログラムが正しく動作する」ことを第一義的に考えるであろうから，負荷分散のための記述の付加や修正がプログラムの論理に与える影響を，できるだけ小さくすることが人道的であろう。さて，P^3 の上で負荷分散戦略を立てるのは容易であるように見えるが，次のような事態は考えられないだろうか。

(a) プログラマが方眼紙を前にして，定規を片手にああでもないこうでもないと悩んでいる。色々やってみるものの，ａとｂとをくっつけると，ｃとｄとがくっつかなくなってしまい，だんだんいやになってくる。結局疲れてしまって「まあいいか」と適当に手をうってしまう。

(b) プログラマがヒイヒイ言いながらデバッグをしている。プログラムを直している間に最初くっついていたｅとｆがくっつかなくなることに気が付くが，「面倒臭いなあ」といってあきらめてしまう。また，ｇとｈが泣き別れしたことには気付きもしない。

(c) やっとプログラムが動くようになるが，案の定c/d，e/f，g/h の泣き別れのおかげでたいして速くない。「せっかく動いているのに」とぶつぶつ言いながらプログラムを見直してみるが，もう最初の絵がどこかに行ってしまっており，年度末も迫っていることでもあるので知らなかったことにしてしまう。

P^3 は，「兄弟」は比較的良くくっつくが，「従兄弟」はくっつきにくく，24親等ぐらい離れるとくっつけるのは絶望的になると言う性質がある。負荷分散戦略の変更，即ち P^3 上の配置の変更が，プログラムの構造の変更を要求するのは問題ではなかろうか。従って，この美点は人道的に必要であるが，P^3 はそれほど美しくないように見える。

(5) 実現の容易性：
いくら良いモデルであっても，インプリメントできなければしょうがないのは当たり前である。別にハードウェアで直接インプリメントしなければならないとは言わないが（全くハードウェアに関係ないのは寂しい

が），楕円積分やテンソルを持ちだされても困るのである（こんなもの何なのかも知らない）。P^3 は平面幾何学と言う一見簡単そうな数学を基礎においているが，小学校4年以上の算数は裏でちょこちょこやるには難しすぎるのである。従って，この美点は工学的に必要であるが，P^3 は余り美しくない。

以上のように，P^3 は心理的，倫理的には美しいものであるが，政治，経済，人道，工学の面で，美しさに欠ける部分があるように思われる。

3 美意識の追及

3.1 抽象性の美学

皆で仲良く研究するためには，並列マシンのアーキテクチャに依存しないモデルが望ましい。従って，あらゆる並列マシンに普遍的に存在する性質だけを相手にする必要がある。即ち;

- 沢山のプロセッサがあって，プロセッサ p_i の「忙しさ」l_i が定義される。
- 2つのプロセッサ p_i と p_j との間で行われる通信に要する手間に基づき，2つのプロセッサの間の距離 d_{ij} が定義される。
- 特定のゴール g に関して基準となるプロセッサの集合 R^g がある。
- ゴール g は p_i と R^g の各要素との距離の集合 D_i^g と l_i とに，ゴール g の性質を加味して定まるエネルギー関数 $E_g(D_i^g, l_i)$ が最小又は極小となるプロセッサに割付けられる。

程度のことを基本にし，R^g や E_g の「決め方」をモデルとして定めれば，どんな並列マシンにも適用できるだろう。残る問題は，R^g と E_g の「決め方」である。

3.2 記述容易性の美学

プログラマにとって，2つのプロセスの間の通信量の多寡を考えるよりも，プロセス間に介在するデータのアクセス頻度を考える方が容易ではなかろうか [3]。即ち，「私はこのデータをよく触る」とか，「このストリームにはよくデータが流れる」とか言うのは難しくないのではないか。また，プロセスとデータの関係はソース・プログラム上で局所的なものであり，他の部分をどういじってもくっついているものはくっついたままにできる。更に，ストリームの端につながっているプロセスが何百親等離れていたって，くっつけることができるのである。

従って，ゴールを割付けるときに基準となるプロセッサの集合 R^g には，少なくともゴールが参照しているデータや変数を保持しているプロセッサが含まれなければならない。単純に言えば，「この変数を持っているプロセッサのできるだけ近くに割付けなさい」と言うのである。

このような方法をとった場合，データも動いてくれなければ面白くないが，実は今でも動くのである。例えば，Multi-PSI では，リストを使ったストリームの場合はリスト・セルは generator 側に動く傾向があり，マージのために最適化されたストリームの場合は consumer 側に動く傾向がある。従って，例えば後者の場合；

- generator が乗り込む。
- generator が consumer に「こっちへおいで」と誘う。

などができるだろう。また，ストリームのインプリメントをうまく考えれば，generator と consumer が各々の置かれている環境（他のストリームとの兼ね合いや自分がいるプロセッサの忙しさなど）についての情報交換をして，どちらが動けば良いか（またはどちらも動かない方が良いか）を決めることもできるだろう。更に，ストリームの流量を測定して（データの大きさやストリームの両端の「距離」などを加味して），プログラマが言ったことを修正することもできるだろう。

3.3 実現容易性の美学

局所的な情報だけで裏作業を行う以上，その結果の大域的良否にこだわるのは危険であろう。例えば P^3 の場合，「特定の座標を含むプロセッサ」というのは大域的な性質を持っており，それを局所情報のみから捜すのはやはり大変なのである。もし，「自分及び隣接する4つのプロセッサとの中で，与えられた座標に最も近いプロセッサにゴールを転送する」という方法を用いれば，本当のプロセッサには届かないことがあるかも知れないが，インプリメントは遥かに容易になるだろう。

同様に先に述べたエネルギー関数 E_g の評価にしても，「最小」となるプロセッサを見つけるのは大域的な作業である。従って，「極小」で我慢してインプリメントを容易にすることが必要であろう。

4 おわりに（青年の物語）

青年は故郷の有様に絶望して，新天地を求めて旅に出た。適当な町が見つかりアパートに落ち着いた青年は，故郷に残してきた恋人のことを思い出す。ここから，物語は始まる。

物語1 青年には他に好きな娘ができたので，恋人のことは忘れてしまう。

物語2 やはり恋人のことが忘れられず手紙を書き倒す。

物語21 恋人には故郷に他に好きな男がいるので，青年は無視される。

物語22 恋人から「こっちも近頃は住みやすいわよ」という返事が来たので，軟弱にも青年はUターンしてしまう。

物語23 恋人は青年の情熱にほだされて故郷を離れ汽車に乗る。

物語231 とはいえ故郷は離れがたいので，恋人は途中の駅で降りてしまう。

物語232 途中に「すっごく素敵な」町を見つけたので，恋人は途中の駅で降りてしまう。

物語233 恋人は青年の町に着き，

物語2331 青年と幸せに暮らす（めでたしめでたし）。

物語2332 青年が自分を尋ねて故郷に返ってしまったことを聞いてがくぜんとする（ひっどい，バグよ!）。

参考文献

[1] Y. Takeda, et al., A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation, In *Proc. of FGCS'88*, 1987.

[2] 近山 隆, 「並列計算機研究の方向はこうあるべきである」, In *ICOT-WG Workshop on Parallel Inference Machines and Multi-PSI Systems*, 1987.

[3] 吉田 かおる, 「宇宙服は要らない!」, In *ICOT-WG Workshop on Parallel Inference Machines and Multi-PSI Systems*, 1987.

私はここまで分かり、ここが分からないと思う。

第２部：並列処理の問題点、「私の興味、実感、そして夢」

並列処理の問題点――私の興味，実感，そして夢

富田眞治（九州大学）

1　分かったような気になっていること

(i)汎用構造の重要性

専用などは別にして一般的には，メンテナンス，性能価格比の点で汎用構造が不可欠

スーパコンピュータの成功：高速化と汎用化＋＋特殊化

データ量の変動に対する許容性：シストリックアレイ

ＲＩＳＣの発展：ゼイ肉落し

(ii)要素プロセッサ設計の重要性

ＶＬＳＩとの関連

コンパイル時ｖｓ実行時並列性検出：よいバランスが必要

ＳＩＭＤ対ＭＩＭＤ

ＶＬＩＷ対データフロー

(iii)相互結合網がキー

クロスバ網：数１００ｘ数１００が現技術で可能

アナログ（多値），光の利用：１ビット／１ラインはけしからん?

(iv)メモリが結局いつもボトルネック

汎用機：キャッシュメモリ大変，データフロー：もっと大変

スーパコンピュータの巨大データ供給系

同時リード／ライト可能メモリ

2　興味（研究中のもの）

(i)可変構造型マルチプロセッサ

相互結合網とメモリ構造の問題適応化

静的可変性：問題の定式化のベースとして、ユーザによる仮想トポロジの設定．並列プログラムの生産性向上

動的可変性：密結合，疎結合の動的クラスタリングなど，「類は類を呼ぶ」方式

(ii)ＶＬＩＷとスーパスカラー計算機

均質型多重命令パイプライン方式の『新風』の研究開発

複数命令／１クロックの計算機

ポストＲＩＳＣ計算機

３　実感

(i)ノイマンは偉かった！！

(ii)概念思想と合致した用語をつくる能力が必要

(iii)社会組織、生物組織と並列計算機のアナロジの重要性

存在するものは歴史的にその存在理由がある.

会社組織などにおける人の生かし方や政党の派閥力学などの

研究をやった方が近道?

(iv)パイプラインがやはり主流なのかもしれない?

４　ここが分からないと思うことと夢

(i)超並列と並並列の質的な差異

アルゴリズムの考え方の根本的変革が必要では

ニューロコンピューター：１つの考え方

厳密でない弱い同期

(ii)脳における意識と記号の発生メカニズムの解明

はじめに記号ありきから出発する人工知能はあまり面白くない

並列処理の問題点

柴山　潔

（京都大学）

1．現在の並列処理技術の問題点

1.1 並列処理技術の動向

科学技術計算分野への適用を主要な目標としているスーパーコンピュータやMIMD型並列計算機では，コンパイラやオペレーティング・システムといったシステム・プログラムが優れていれば，遺産としてあるソフトウェアを比較的簡単に利用，高速実行できるので，将来もその地位を保ち続けるものと考えられる．

逐次計算機の限界を打破することによって計算機の応用範囲を拡大するためには，10^5以上規模の要素プロセッサから構成される超並列計算機の出現が待望されている．そのためには計算機アーキテクチャの構築原理におけるブレークスルーが必要とされている．コネクション・マシンなどの極細粒度の専用高並列計算機，シストリック・アレイなどのVLSI指向型並列計算機，コネクショニズムを計算原理とするニューロ・コンピュータなどに，その兆候はうかがえる．しかし，超並列計算機については，まだシステム構築の原理すら解明できておらず，ましてやシステムにおけるハードウェアとソフトウェアのトレードオフの議論などはまだ端緒に付いたばかりである．計算機システムの目標が計算の高速化にある限り，並列処理方式は常に古くて新しい問題として注視され続けるであろう．

1.2 並列処理計算機の課題

計算機の並列化(対象言語や処理機能への並列性の導入)に伴う解決すべき課題は，応用に依らず，並列計算機全般に関連する問題と言えるものが多い．例えば，高並列化に伴いシステム全体の制御が難しくなるが，計算機の初期化や処理結果の取り出し方法などについての研究は，まだ緒についたばかりである．

細かい粒度のシステムにおいてはもちろん，粗い粒度の並列計算機を構成する要素プロセッサにもチップ化の波が押し寄せてくることが予想され，計算機の高並列化は容易になるとともに，単位処理機能間の通信方式のブレーク・スルーが要求されてくるであろう．

ハードウェア資源が有限であるシステム上で細かい論理的粒度を実現する場合には，要素プロセッサ自体のハードウェア/ファームウェア/ソフトウェアのトレードオフが重要な問題となってくる．例えば，最近ではRISCアーキテクチャを採用した逐次型計算機も見られるので，これらを用いて並列計算機を構築する場合などにもシステム全体のトレードオフに対する慎重な考察が必要である．

1.3 並列計算機の並列度

［定義］

（並列度）＝（PE台数）×（PE粒度）　（単位：台・ビット）

（例）

- CM：64K　（64K台×1ビット）
- SIGMA-1：4K　（128台×32ビット）
- PIM/?：16K　（512台×32ビット）
- 超並列計算機：10^5以上

1.4 論理的粒度と物理的粒度

［定義］ 粒度(granularity) ＝ 並列処理単位の機能の大きさ

・論理的粒度
→応用や問題が本質的に含んでいる並列性の単位機能.

・物理的粒度
→マシン･アーキテクチャ上で実現する場合に示されるハードウェアの処理要素(PE)としての単位機能. （1.3節の定義参照）

・論理的粒度と物理的粒度の対応関係(図1参照)
→粒度に応じたハードウェア／ソフトウェア・トレードオフを構築できる柔軟な並列処理機構が必要.

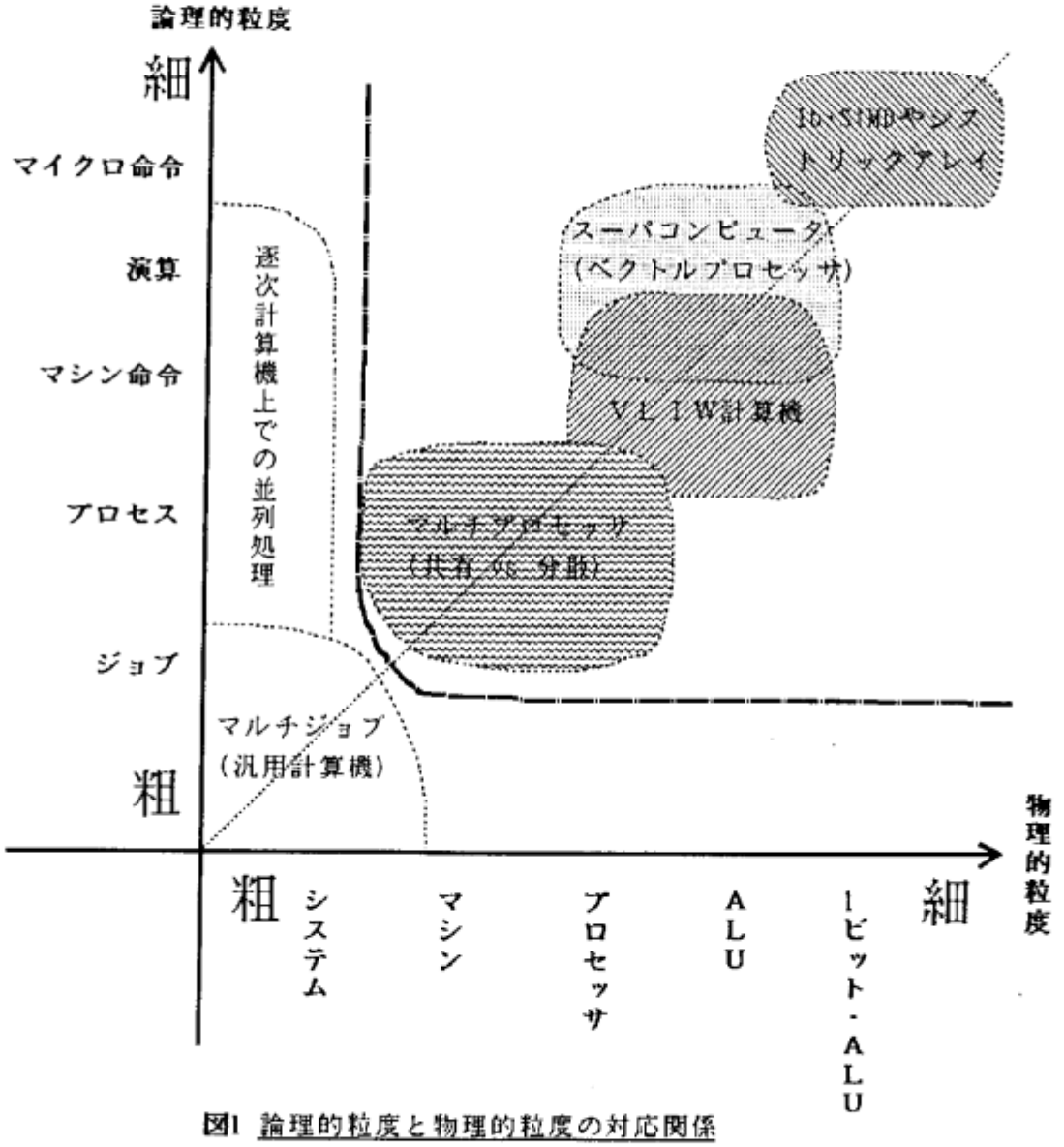


図1 論理的粒度と物理的粒度の対応関係

2. 超並列計算機システムの要素プロセッサ・アーキテクチャとしてのVLIW方式

2.1 定義

最近，SIMD型とMIMD型を混合したようなシステム構成方式が注目されている．これは，「VLIW(超長形式命令語)型」と呼ばれる並列処理方式の一種である．VLIW型では，非常に長い1語命令が複数のフィールドから構成されており，各フィールドが相異なる演算要素を制御する．超長形式命令語をマクロ的に捉えるとSIMD型であり，超長形式命令語を構成する各命令フィールドを別々の命令と捉えるとMIMD型である．

VLIW型並列処理方式を演算回路やマイクロ命令の機能レベルに適用するものを「低レベル並列処理方式」とも呼ぶ．低レベル並列処理計算機は，比較的長い語長の命令の相異なるフィールドで複数の演算器やメモリ・アクセス機構や順序制御機構などをきめ細かく同時に制御する方式である．従って，汎用性と高速性の両方を追求する必要のあるユニバーサル・ホスト計算機ではよく採用される方式である．

VLIW型並列処理全般に対するハードウェア上のサポート機能としては，命令語が水平型，メモリの分散(バンク)化，データ転送路幅の複数・拡大化，命令やデータ処理に対するパイプライン方式との併用などが必要となる．

また，逐次型言語の処理においても，その処理機能を洗い直してみると，並列処理可能な部分が発見できる．これらの並列性はあまり高くないが，大量データに対してパイプライン効果が利用できたり，対象とする問題に依存した並列あるいは命令パイプライン処理が可能である場合も多い．例えば，Prologマシンや Lispマシン，Smalltalkマシンなどの専用逐次計算機においても，低レベル並列処理方式を採用しているものがある．

また，逐次計算機内の処理構造だけではなく，並列計算機の単位処理機能がパイプライン構造を持ち，計算機全体のアーキテクチャによる高並列性を補完する(並列ーパイプライン)方式の計算機も数多い．これらについては，低レベル並列処理方式の計算機との区別が困難であるものが多い．

現在の並列処理技術と，将来の超並列処理技術の「つなぎ(橋渡し)」として有望である．

2.2 着眼点

・PEを並列化するときにもメタなVLIW方式(命令の機能レベルが異なるが，インタプリト・メカニズムは同一)を適用できないか?
- 超並列計算機(1000台PE以上規模)の要素プロセッサ(PE)がVLIWアーキテクチャ
- 階層内 & 階層間でのスケーラビリィティ(scalability)を確保するアーキテクチャとしてのVLIW方式?
- FUレベルとPE(プロセッサ)レベルの2階層で，10^5プロセッサ規模になる?

・プロセッサ(RISCなど軽くなる傾向) と メモリ(スヌープ・キャッシュとか重くなる傾向)の速度差を吸収する手だて?
- キャッシュ以外にないか? ← 「マルチFU(ALU)ー共有レジスタ」構造のように，「マルチプロセッサー共有メモリ without キャッシュ」が構成できないか?
- やはり共有メモリ・アーキテクチャか? FUレベルは共有，PEレベルは分散にして，階層間scalabilityは実現できるか?

・静的最適化(VLIWの性能を左右するのはコンパイラの最適化能力といわれている) vs 動的最適化(例えば，動的データフロー)
- いかにして VLIW 1語にオペレーションを詰め込むか?
- restricted データフロー方式?
- コンパイラの問題にしたくない. → アーキテクトの「楽しみ」

・パイプラインはどうするか?
- スカラー and/or ベクトル → 応用(問題)に依存する?

ワークショップ原稿

小池　汎平
koike@mtl.t.u-tokyo.ac.jp
東京大学 工学部

1989 年 10 月 6 日

1　近況報告

ようやく、SU チップの動作に納得することができました。今朝、2 種類目のネットワークボードも届いて、そこにおいてあります。もうすぐでき上がるネットワークテスタを待っているところです。

2　並列計算機の通信系について

並列計算機の最も重要な要素であるネットワークなどの通信系について、転送データ量とか遅延時間とかいうあまりに一般的な低いレベルでしか、議論がなされていないのは問題だと思います。まず第一に必要なのは、並列処理を行なったときに、その通信系を使って「やらなければならないこと」が何であるかを明確化することであり、どのようにすればそれを「どれだけはやく」「どれだけたくさん」できるか、を基準にして通信系の設計を行なうべきだと思います。また、「やらなければならないこと」は、ある程度高い水準に設定し (感じとしては浮動小数点演算程度)、専用の通信ハードウェアでメインの処理となるべく独立に高速実行できるようにすることが、性能の面からも、扱い易さの面からも重要です。枠組としては単方向メッセージ通信では不十分で、リモートプロシージャコールに基づくべきでしょう。こう考えると、回線交換ネットワークも捨てたものではありません。まあ、いずれにしても非常に多くの要因の間でバランスを取ると言う作業が必要なのは言うまでもありません。

あと、やっぱり、とりあえずは、最大限高い通信性能を得るべく努力することが、汎用並列計算機ハードウェアを研究するポイントだと思います。(勿論、無限を考えることは意味の無いことです。) 限られた空間で最大限の通信能力を得ることが大切だなあ、とあらためて思いました。ネットワークの高性能化の将来に関しては、極細光ファイバ (空間当たりの通信密度の向上は 2-3 桁程度でしょうか?) ファイバの編み方 (ビーチスキンを作ってしまう日本のハイテク繊維工業に期待するのはどうでしょうか?)、耐故障性、に興味があります。

3　どうやって、並列プログラムを作るかについて

それなりの環境を用意して、ハッカーを養成しなければ無理でしょう。

4　百万台とか、一兆台とかについて

これは、100 台とか 1000 台をちゃんと作ってみてからでないと良くわかりません。

ＰＩＭのここがわからない

ＰＩＭＷＧ合宿資料
小長谷　明彦
日本電気株式会社
Ｃ＆Ｃシステム研究所

捜すものは何ですか？

ＰＩＭの研究を見てて一番分からないことは、何を目指して言語、アーキテクチャ等のディシジョンを行っているのかである。スーパーコンの場合には、偏微分方程式を解くために必要な数値処理演算機能の強化が大前提であり、その結果、今日のベクトル計算機アーキテクチャがあるといってよい。

ＰＩＭの場合、理念的には、「並列」、「論理」、「知識処理」というお題目があるものの、ＫＬ１、ＰＩＭＯＳ、ＰＩＭの設計思想がどこまでそれらを反映しているかよくわからない。何が言いたいかというと、ＫＬ１、ＰＩＭＯＳ、ＰＩＭの設計がいいか悪いかではなく、その設計思想が何を対象として想定しているのかがよく見えないということである。。

見つけにくいものですか！

ここ、２年ほど遺伝情報処理の研究に足を踏み入れているせいもあるが、計算機を利用者の立場から見る癖がついてきた。そして、最近になって、「並列」、「論理」、「知識処理」という枠組みは、それほど悪くないのではないかと思い始めたのである。一例として、現在、私を悩ましている蛋白配列の特徴抽出の問題を考えよう。

蛋白配列はそれを構成するアミノ酸の列であり、その長さは数十から数百程度に及ぶ。ＮＢＲＦ蛋白データベースには、現在、約５０００種の蛋白配列が格納されており、それらは、配列の類似性の観点から「スーパーファミリ」と呼ばれるグループに分割されている。

特徴抽出問題とは、各々のスーパーファミリを特徴付けるようなパターン（通常モチーフと呼ばれる）を配列の中から捜し出すことである。モチーフの存在は蛋白質を機能づける部位は遺伝的に保存され易いことから来ている。このような特徴抽出問題は音声認識やパターン認識等と同根であり、知識処理の代表的な問題の一つと言えよう。

まだまだ、探す気ですか？

この特徴抽出問題を解くために、「確率論理に基づく帰納推論（付録Ａ）」を用い、Ｐｒｏｌｏｇで実現した。帰納推論を用いたのは、単に特徴抽出を行うメカニズムを作るだけでなく、特徴抽出を行ったときの根拠を充分説明できるようにするためである。また、確率論理を用いたのは、モチーフのような特徴抽出を行う際には、真か偽かという２値論理では不十分であり、設定した仮説の真偽値の確率的な解釈を行いたかったからである。

機能推論は以下の方式で行った。
（１）　ある仮説に基づいて、特定のスーパーファミリを代表するようなパターンを表すアサーションを作成する。
（２）　実際のデータを用いて、アサーションの真理度を検定する。
（３）　真理度の高いアサーションを用いて、別なアサーションを合成する。
（４）　（２）、（３）のフェーズを繰り返す。

より具体的に述べると、アミノ酸の種類は２０種類である。従って、各アミノ酸の分布が一様とすると（実際には一様ではないが）、長さ２のアミノ酸配列がランダムに表れる確率は４００分の１、長さ３ならば８０００分の１となる。蛋白配列の平均的な長さは２００から３００であるから、長さ３のアミノ酸配列に注目すれば充分モチーフの指標となる。

そこで、各スーパーファミリごとに長さ３のアミノ酸配列の出現頻度を調べたところ充分に偏りがあることが確認された。しかしながら、長さ３のアミノ酸配列だけでは、スーパーファミリを特定するには不十分であったため、次に、２つの長さ３のアミノ酸配列の組み合せについて調べてみた。得られた結果の一部を付録Ｂに示す。チトクロームＣ（全体は１１９配列）の例では、このような簡単なパターンでも７割程度の配列が識別できることがわかった。さらに、パターンにワイルドキャラクタを含めたり、既に特徴パターンが見つかったグループと元ののグループとの差分に着目して特徴パターンのチューニングを行えば、より正確な特徴検出が可能となろう。

それより僕と・・・？

上記の特徴検索で重要なことは、以下の３点である。
（１）確率論理を問題解決の基礎においたこと。
（２）帰納推論を問題解決の手法として用いたこと。
（３）データ量が非常に多く、計算に時間がかかったこと。
すなわち、この問題は、ＰＩＭが目指す「論理」「知識処理」「並列」という３要素を全て備え、かつ、実用的という、まことに好都合な問題である。次に、この問題を並列に解くことについて考察しよう。

並列度：

長さ３のアミノ酸配列という非常に簡単なパターンにおいても組み合せは８０００通り。この任意の２つの組み合せは、約３２百万通りにもなる。さらに、これを５０００個以上の全ての配列について検査するのであるから、潜在的な並列度に関しては全く問題にならない。むしろ、無駄な計算を行わないために、いかにして、並列処理を制御するかが問題の焦点となる。
例えば、ＣＨＩでは、一つのスーパーファミリに関するパターンの組合せを高々１０個に制限している。並列に処理する場合においても、同様な技法は有効であり、並列言語はこのような並列発生を抑制するような機能が必要であろう。

メモリ消費量：

ＣＨＩでの実現において、もっとも、注意を要したのが、メモリ消費量をいかにして減らすかである。ＣＨＩは３２ØＭバイトとという大容量主メモリを備えているとはいえ、全ての配列の頻度情報（８ØØØ＊５ØØØ語＝１６ØＭバイト）を蓄えている程の余裕はない（不可能ではないが、いづれ、限界がくる）。そこで、積極的に利用したのが、「破壊的なベクトル処理」である。

破壊的代入は、「論理的」でないという人がいるかも知れないが、それは、「理論的」な作成者の論理であり、利用者の論理ではない。私が、利用したかったのは方法論としての「論理」であり、「論理型言語」ではない。利用者の立場からは、言語はあくまでも目的実現のためのツールであり、方法論として正しければ、実現法はどうでもよいのである。

最近になって、ＫＬ１にも、メモリ利用効率を重視した構造体実現法が実装され、少し、良くなったが、利用者の立場からは、もっと、「非論理」的でも効率のよい機能があって欲しいと思う。例えば、ベクトルを再初期化するような機能があれば、応用プログラムは作り易くなろう。

通信処理：

並列プログラムのキーポイントの一つはいかにしてＰＥ間の通信コストを少なくするかにある。特徴抽出問題においても、特定のパターンが表れる頻度を調べるためには全ての配列に対して調べる必要があるため、ＰＥ間での通信を高速に行うための工夫が必要となる。

次に、どのような、通信処理が必要となるかについて考察する。特徴抽出問題は次の２つのフェーズからなる。

（１）特徴パターンの抽出
（２）特徴パターンの頻度測定

特徴パターンの抽出に関しては、スーパーファミリの中で閉じている処理なので、スーパーファミリ単位に、独立に処理することができる。一方、特徴パターンの頻度測定は、全ての蛋白配列について検査を行う必要があるため、蛋白配列か、特徴パターンのどちらかをＰＥ間で転送する必要がある。蛋白配列の個数（５ØØØ）に比べ、対象となる特徴パターン（数万～数十万）の方がはるかに多いため、蛋白配列を転送あるいはコピーする方式の方が有利である。

夢の中に

特徴抽出問題に限って言えば、このような問題を解くためだけならばＰＩＭを作るまでもないというかもしれない。蛋白配列データは高々２Ｍバイト程度であるから、これを、各ＰＥに初めからコピーして於て置けば、ＰＥ間の転送を全くなくすことすら可能である。しかしながら、現実的には、現在のＰＩＭのアーキテクチャ、ＫＬ１の機能では、この問題をＣＨＩ以上の性能で解けるという保証も無いのである。

このギャップは、恐らくは、物事を必要以上に難しく考えていることに起因しているように思える。並列で解きたいこと、並列で解けること、これらは自ずとして逐次処理のそれとは異なるはずである。並列処理の一側面に捕らわれて、「木を見て、森を見ず」というようなことにはなって欲しくはないと思う所存である。

付録　確率論理による帰納推論　（大野　１９８９年度人工知能学会全国大会）

原子論理式の観測に基づいて、論理式の真理値を決定する論理体系。人間の直観にあった帰納推論が実現できる。

定義　確率論理

Ａ：　原子論理式の集合、ａ　ε　Ａに対し、
観測度　μ（ａ）：　ａを観測する確率
ただし　$\sum \mu$（ａ）＝１
真理値　τ（ａ）：　ａが真である確率
ただし　$\emptyset \leqq \tau$（ａ）$\leqq$　１

μ（Ａ$\wedge$Ｂ）＝μ（Ａ）μ（Ｂ）
μ（Ａ$\vee$Ｂ）＝μ（Ａ）μ（Ｂ）
τ（Ａ$\wedge$Ｂ）＝τ（Ａ）$\wedge\tau$（Ｂ）
τ（Ａ$\vee$Ｂ）＝τ（Ａ）＋τ（Ｂ）－τ（Ａ$\wedge$Ｂ）

定義　モデルエントロピー　（仮説の確率的誤差を示す尺度）

仮定①　平均対数尤度（Shannonエントロピーと Kullback-Leibler情報量の和）の妥当性
仮定②　真理値の事前確率分布の均等性

このとき、モデルエントロピーＭｈを次式で定義する。このモデルエントロピーが小さいほど、妥当な仮説として解釈できる。一般にＭｈは真理値と推定真理値との誤差が小さいほど、また、仮説の数が少ないほど小さい値をとる。

$$Mh = \sum_{L_i} \mu_i \int_{\emptyset}^{1} (H(\tau_i) + D(\tau_i ; \hat{\tau}_i)) f(\tau_i) d\tau i$$

ただし、L_iは仮説Ｈの論理式
μ_iは論理式L_iの観測度
τ_iは論理式L_iの真理値
$\hat{\tau}_i$は論理式L_iの推定真理値
Ｈ（τ）はShannonエントロピー
Ｄ（τ；$\hat{\tau}$）はKullback-Leibler情報量
ｆ（τ）は真理値τの事後分布

定義　推定真理値と観測度

仮定　真理値の事前分布を均等とする。
ベイズ推定により、推定真理値および観測度を下記のように定義できる。

$\hat{\tau}_i$　＝　（Ｌｉを真とする実例数＋１）／（Ｌｉの前提部を満たす実例数＋２）
μi　＝　（Ｌｉの前提部を満たす実例数＋１）／（全実例数＋全論理式数）

付録Ｂ　　チトクロームＣの推定に用いた仮説と推定真理値の一部

```
contain(['TKM','CHT'],'cytochrome c',69,68,0.971830985915493)
contain(['ANK','CHT'],'cytochrome c',61,59,0.9523809523809524)
contain(['GTK','CHT'],'cytochrome c',71,68,0.9452054794520548)
contain(['GPN','CHT'],'cytochrome c',70,67,0.9444444444444444)
contain(['TKM','ANK'],'cytochrome c',67,64,0.9420289855072464)
contain(['TKM','NPK'],'cytochrome c',73,69,0.9333333333333333)
contain(['PNL','CHT'],'cytochrome c',71,67,0.9315068493150685)
contain(['AYL','CHT'],'cytochrome c',61,57,0.9206349206349207)
contain(['NPK','GPN'],'cytochrome c',73,67,0.9066666666666667)
contain(['TKM','PGT'],'cytochrome c',83,76,0.9058823529411765)
contain(['PGT','CHT'],'cytochrome c',75,68,0.8961038961038961)
contain(['GPN','ANK'],'cytochrome c',67,60,0.8840579710144927)
contain(['TKM','GTK'],'cytochrome c',85,76,0.8850574712643678)
contain(['NPK','CHT'],'cytochrome c',75,66,0.8701298701298701)
contain(['IPG','CHT'],'cytochrome c',75,65,0.8571428571428571)
contain(['GTK','NPK'],'cytochrome c',81,70,0.855421686746988)
contain(['TKM','PNL'],'cytochrome c',80,69,0.8536585365853659)
contain(['TKM','GPN'],'cytochrome c',80,69,0.8536585365853659)
contain(['TKM','IPG'],'cytochrome c',79,68,0.8518518518518518)
contain(['PNL','ANK'],'cytochrome c',73,61,0.8266666666666667)
contain(['NPK','AYL'],'cytochrome c',70,58,0.8194444444444444)
contain(['GTK','ANK'],'cytochrome c',78,64,0.8125)
contain(['TKM','AYL'],'cytochrome c',77,63,0.8101265822784811)
contain(['GTK','GPN'],'cytochrome c',84,68,0.8023255813953488)
contain(['PNL','IPG'],'cytochrome c',82,66,0.7976190476190476)
contain(['IPG','GPN'],'cytochrome c',82,66,0.7976190476190476)
contain(['PGT','NPK'],'cytochrome c',87,69,0.7865168539325843)
contain(['AYL','GPN'],'cytochrome c',77,60,0.7721518987341772)
contain(['NPK','PNL'],'cytochrome c',86,67,0.7727272727272727)
contain(['GTK','IPG'],'cytochrome c',88,68,0.7666666666666667)
contain(['PGT','ANK'],'cytochrome c',83,64,0.7647058823529412)
contain(['AYL','ANK'],'cytochrome c',72,55,0.7567567567567568)
contain(['GTK','AYL'],'cytochrome c',80,61,0.7560975609756098)
contain(['PGT','GPN'],'cytochrome c',91,68,0.7419354838709677)
contain(['NPK','ANK'],'cytochrome c',82,61,0.7380952380952381)
contain(['AYL','IPG'],'cytochrome c',79,58,0.7283950617283951)
```

並列計算機における静的と動的

坂井修一　（電子技術総合研究所）

1．はじめに

「これは速い！」「これなら使いたい！」と人を説得できるアーキテクチャを提示するためには、要素技術の開発と同時にバランスの良いシンセシスの研究が重要である。

アーキテクトの頭の中では、ひとつひとつの要素技術の開発というシーズから全体のシステムを考えるというボトムアップ的な思考と、全体のシステムを構築するために必要な（あると嬉しい）技術をニーズとして提出し解決法を統合するというトップダウン的な思考がいつも渦巻いている。この部分←→全体の上下動の中で、一見して魅力的ないくつものアイディアが捨てられ、別の一見地味そうで実は効果的な手法が採用されることがしばしばある。

例えば、計算機のパイプライン設計を行なう際、パイプの総段数と各段の時間長を決めなくてはならないが、これは計算機全体の細部を総合的に見渡すことを必要とする作業であろう。素子技術の進歩、プログラムの動特性解析の進歩、過去の設計の反省などがその根拠となるだろうが、可能な内部動作をはじめからすべて考慮してパイプライン設計を行なうことは不可能と考えられる。いくつかの典型的な場合からパイプラインを決定し、これにあわせて他の場合を調整したり、生じた例外に対して全体を補修したりするのではないだろうか。

ここで、「あるパイプラインレジスタから別のパイプラインレジスタへのデータ転送の間に小さなデータ整形を入れることで、全体の効率が１０％向上する」ことを、末端の設計者の一人が発見したとする。彼がこの修正に固執するのは自然なことだ。しかし、前段のレジスタのファンアウトの問題、データ整形回路の遅延の問題、本回路の後に入るマルチプレクサの遅延の問題、チップ内のゲート数の問題、ボード上の石の総数の問題、などなどがすべてクリアされるには相当の手続きが必要となるだろう。下手をするとこの修正を入れたばかりに１クロックの長さが２０％増しになったり、新しい大きなＬＳＩが必要になったりしないとも限らない。

*

データフローの問題がこういう問題と本質的に異なるのは、単なる要素技術としてデータフローを取り込む、という立場が立てにくいところにある。データフロー計算機は、データフローモデルに基づく計算機のことだが、それには、

(1)全命令における動的同期（マッチング）
(2)パケットアーキテクチャ＋循環パイプライン
(3)単一代入言語

など、計算システムの全体を規定する概念がからみついている。

したがって、計算・実行モデルとしてＩＣＯＴ－ＰＩＭのように並列フォンノイマン型モデルを考えるか、ＥＴＬ－ＥＭ４のように改良型データフローモデルを考えるか、は、すでに前提条件から異なっているのかもしれない。

ＩＣＯＴの研究者とＥＴＬの研究者の間で討論する場合、モデルや言語構造をいかにハードウェアに写像したか、という視点は、前提が違いすぎて有効ではない。例えば、ＧＨＣのサスペンション機構をどう高速化するか、という問題はＥＴＬ側にはあまり興味がない（全然ないわけではない）ことだし、データフローの残留トークン問題をＥＭ－４でいかに解決したか、は、ＰＩＭの研究者にはどうでもよい問題だろう。

そこで、ここでは別の視点から並列アーキテクチャ最適化の方法論のようなものを考察したい。その際もっとも有効な視点として、「静的・動的の切り分け」ということを考える。すなわち、ある仕事の実行ないし実行制御を、

(1)プログラマの指示に従って静的に実現する
(2)コンパイラの自動検出に従って静的に実現する

(3)ソフトウェア的に動的に実現する
(4)ハードウェア的に動的に実現する

ということの切りわけをどのようになすべきか、そのためにはどのような技術が必要か、ということを以下の各節で（ごく簡単に）考察して、「並列処理の問題点」という課題のとりあえずの回答としたい。ここで、静的とはある事柄がプログラムの実行以前に完全に決定済みであることを言い、動的とはこれが実行時に決められることを言う。筆者は、シンセシスを行なうという立場でのアーキテクトの最も重要な仕事が、この「静的・動的の切り分け」にあると考えている。

2. 静的・動的の選択

2. 1 選択の対象

静的・動的を切り分ける際のポイントは、次の2点に集約される。

(1)マシンで対象とする応用プログラムの挙動はどこまで静的に解析可能であるか?
(2)(1)の解析はどのぐらいのオーバヘッドをもたらすか?

より簡単にいうと、

☆（広義の）スケジューリングは実行以前にどこまで決定するべきか?

ということになる。

スケジューリングを実行以前に完全に決定するという方式がもっとも極端なものとして想定される。しかし、静的な最適スケジューリングが完全に可能であるとしても、一般にこれはＮＰ完全の問題となり、現実的な解が得られない。静的な方式で最適スケジューリングの近似解を求め、これに従って実行する方式が次に考えられる。しかし、このことは次のような理由により完全には困難である。

①相互結合網をデータが通過するのに要する時間が予測困難であり、したがって必要なデータが揃うタイミングを知ることがむずかしい。
②条件分岐・（終了条件が動的に決まる）ループ・再帰など、実行時にならなくては挙動がわからない部分がプログラム中に存在する。

では、スケジューリングのうち静的に行なう部分、動的に行なう部分は何であろうか?

これを考える前に、ここでいう「広義のスケジューリング」の諸要素を分析してみる。すなわち、静的・動的の選択対象となる主要な事柄は、次のようなことである。

(1)同期
(2)負荷分散（計算負荷の分散）
(3)（構造）データの分配
(4)デッドロック防止／回復

これらそれぞれについて、静的・動的の切り分けを考えなくてはならない。もちろん、現実にはこれらは単純に切り離して論じられる問題ではない。例えば、計算負荷は対象となるデータの分配状況によって変動するから(2)と(3)は同時に考えていかなくてはならないし、数の爆発による資源の枯渇、という問題は、(2)(4)にまたがる問題として認識される。ただし、ここでは(1)(2)に話を絞り、要素技術と動的・静的の選択がどのように関係しているのかを個々に見てみたい。参考までにＥＭ－４で採用した方式を付記することにする。

2. 2 同期

同期のオーバヘッドがターゲットとする並列計算機全体の性能とどういう関係にあるか、は、重要な問題である。前節で述べた①②の理由から、動的同期が必要なのは言うまでもないが、これを実現する機構、動的・静的を切り分ける機構、各方式の同期の高速化などは十分に検討されなくてはならない。これらは、要素プロセッサ（ＰＥ）のパイプライン設計と深くかかわる問題である。

いま、シングルスレッドのパイプラインをもつフォンノイマン型の並列計算機を考えよう。動的同期は、プロセス間でメッセージをやりとりすることで行なうとし、ハードウェア機構としては割り込みとコンテクストスイッチがあるとする。この場合、コンテクストスイッチのオーバヘッドが全システムの性能に支配的となる危険があり、ある同期のタイミングから次の同期のタイミングまでを平均してできるだけ長くとるようなスケジューリ

ングが必要となる。それでも、この点において、データフロー方式のような、待ち合せをマルチスレッドパイプラインに埋め込む方法には劣ると言わざるを得ない。

次に従来のデータフロー方式を考える。動的な同期は高速に実現されるが、データ待ち合せのオーバヘッドがすべての命令に付随するのはマイナスである。通常待ち合せ処理は、循環パイプラインの1フェーズとして埋め込まれるが、並列度の低い問題では循環パイプラインの長いことがネックになってしまう（これはプリフェッチなどの先行制御ができないことが大きくかかわる）。この問題を解決するために、擬結果方式のようなプログラムレベルの先行制御によって、本来直列的な部分にも多数のトークンを生成してパイプラインを埋める工夫をしなくてはならないが、これにもオーバヘッドがある。

以上を考えると、

(1)シングルスレッドのパイプライン＋静的シーケンシング

(2)マルチスレッドのパイプライン＋動的同期

の共存する構造が必要であることがわかる。すなわち、静的に同期がとれるところは、通常のシングルスレッドのシーケンシングを行い先行制御をするなどして高速化をはかり、動的に同期をとるのが有利なところは、データ駆動的な手段でこれを実現し、そのオーバヘッドは循環パイプラインに埋め込む、という二重構造がもっとも有利であると結論される。

これを支える技術として、

①シングルスレッドパイプライン（静的な部分の実行パイプライン）の最適化

②待ち合せの高速化とマルチスレッドパイプラインの最適化

③2つのパイプラインのハードウェア上の融合

④プログラム上で静的に同期をとる部分と動的にとる部分の切り分け

が必要である。ここでは①②は要素技術であり、③④は融合化技術であると位置づけられる。①に関しては、レジスタやキャッシュ、ＲＩＳＣなどのフォンノイマン計算機の最適化手法が基本的には応用できる。②に関しては、従来のデータフローの技術の上に、ＥＴＳ（Ｍｏｎｓｏｏｎ）のような工夫がなされ、Ｉストラクチャのキャッシュ化やパイプライン長の最適化に関する検討がなされている。②は①と組み合わせることで、パイプラインピッチをより細かなものにすることができる、などの新しい展望がひらけてくる。③に関しては、レジスタや演算器を2つのパイプラインの間でどのように共有するか、2つのパイプラインの接続をどのように行なうか、などが検討されなくてはならない。また、④の作業はコンパイラまたはプログラマが決定するのが通常の方法と考えられるが、その最適化手法が問題となる。

ＥＭ－４は、①レジスタファイルを用いた2段のＲＩＳＣ型シングルスレッドパイプライン（先行制御あり）、②直接マッチングと4段のマルチスレッドパイプライン、③フェッチ・デコード・演算の各フェーズを共有化した自然なパイプラインの融合、という方式によって2レベルの同期を高速に実現した。④に関しては、最適化コンパイラが切り分けを行なうが、そのアルゴリズムは現在検討中である。

2．3　負荷分散

負荷分散を行なうにあたって、主要な選択点は次のものである。

①何を静的に分散し、何を動的に分散するべきか。

②動的負荷分散の制御はソフトウェアが行なうか、ハードウェアが行なうか

③動的負荷分散の制御は集中的に行なうか、分散的に行なうか

④静的負荷分散の手法はどのようなものを用いるべきか

①②は「切り分け」の問題であり、③④は実現のための要素技術にかかわる問題である。

①から考える。ここでは、負荷の粒度を3つのレベルにわけて考えることにする。すなわち、最小のレベルは命令であり、中間のレベルはブロックであり、最大のレベルは関数（プロシージャ）とする。ブロックとは、関数内の連続するいくつかの命令を含むもので、ループ一周分の命令を1ブロックとする方式などが考えられるだろう。

命令レベルでの動的負荷分散は、実行時間に比して動的負荷分散のオーバヘッドが大きく、実用的ではない。ブロックレベルの負荷分散では、(1)ブロックが十分に大きく、(2)ブロックの生成が動的に決まる、という2条件が満足される場合に、動的負荷分散が有利

になる可能性がある。関数レベルの負荷分散は、数値計算のように関数呼び出しのパターンが実行前にある程度予想される場合には静的に行なうことが有利な場合もあるが、探索問題のような場合には、動的に行なわざるをえないだろう。

次に先の②の問題を考える。負荷分散は、原理的にはＰＥの演算処理とは独立な場所で高速に実現することが可能である。また、負荷分散の制御は単純な作業である。これらの点からソフトウェア的な実現よりハードウェア的な実現が望ましいといえる。ハードウェアで負荷分散を実現する場合に問題となるのが、そのコストである。

③の問題に移る。負荷分散制御の集中化は全体性能を低下させる危険があるために、分散化が必須だが、分散化した場合にシステム全体にわたる負荷の管理ができなくなる危険がある。ＰＩＥ、ＳＩＧＭＡ－１などで実装されている結合網負荷分散方式は、分散的にシステム全体の負荷分散管理を行うことができ、さらにハードウェアコストが安い優れた方式であると考えられる。

④に関しては、コンパイラで自動割付をすることが望ましい。これは、並列フォンノイマンマシンのいくつかの試作機で試みられている手法がある。

ＥＭ－４では、ブロックレベル、命令レベルはコンパイラによる静的負荷分散を採用し、関数レベルのみで動的負荷分散を行なうこととした。もちろん、関数レベルでも静的負荷分散を行なうことが可能である。動的負荷分散の実現法として、結合網による自動負荷分散の発展形である「裏循環路負荷分散」と呼ばれる低コストで効率的な手法を確立した。また、④としてデータフローグラフのノードにラベルづけをする最適化手法を開発中である。

3. おわりに

マイヤーズはアーキテクトの仕事を、「ハードウェアで何をやり、ソフトウェアで何をやるか、そのインタフェースを決定すること」だとした。並列計算機を考える場合、同期・通信・負荷分散・データ分散・デッドロック回避など、確立しなくてはならない要素技術が多く、「いかに」の問題に話が終始しがちである。たしかに実現形態の多様さにおいて並列計算機は直列計算機をはるかにしのぐ選択枝をもっているように見えるが、並列計算機においても、マイヤーズのいう「インタフェース」の問題は「何」「いかに」と同等に重要であろう。

本資料では、「ソフトウェアの仕事とハードウェアの仕事とのインタフェース」というだけでなく、「動的と静的のインタフェース」ということを考えなくてはならないということ、それには抽象的な計算モデル・実行モデルばかりでなく、マシンの物理的なイメージまでも含めた具体的な世界と、インタフェース設定という意志決定の間のフィードバック関係が重要であることを述べたつもりである。

我々は、こんどこそ本当に高並列計算機を作ることができるようになった。こうなると、アーキテクトは、要素技術に徹底してこだわるミクロスコーピックでマニアックな性格と、バランスよいインタフェース設定と取捨選択を行なうマクロスコーピックで”大人”の性格を、同時に強力にもたなくてはならない。アーキテクト本人（達）の中での両者のせめぎあいにどれだけ耐えられるか、が、いいアーキテクチャを生むための大切な条件のように思われる。

参考文献

[1]Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yyba, T: An Architecture of a Dataflow Single Chip Processor, Proc. of the 16th Annu. Int'l. Symp. of Comp. Arch. (ISCA89), pp.46-53 (June 1989).

[2]Yamaguchi, Y., Sakai, S., Hiraki, K., Kodama, Y. and Yuba, T: An Architectural Design of a Highly Parallel Dataflow Machine, Proc. of IFIP World Comp. Cong. (IFIP89), pp.1155-1160 (Aug. 1989).

[3]坂井、平木、山口、児玉、弓場：データ駆動計算機のアーキテクチャ最適化に関する考察、情処論、Vol.30, No.12掲載予定 (Dec. 1989).

並列処理の問題点 — 私の興味、実感、夢

電子技術総合研究所
樋口哲也

1 興味

私は意味ネットワークに基づく知識情報処理システムの開発に従事しており、主に意味ネットワークマシン(IXM)のハードウェアとアプリケーションの開発を行なっている。現在は73台のCPUと256KWの大容量連想記憶からなるマシンのデバッグを行なっている最中であり、6万4千リンクまでの意味ネットの超並列処理を実現しようとしている。

並列処理に対する興味としては、データレベルの並列性を生かす超並列処理が第一にあり、これが知識ベースマシン（PSではない）に対し、どういう形で活かされるのかについて大きな関心をもっている。アプリケーションとしては、MCCのCYCプロジェクトのようなアプローチが好みというか、非常に気になるところである。また最近では意味のあい昧性除去を行なえるような、辞書の語義のマシンによる実現（裏現）に興味を持っている。

2 実感

意味ネットワークマシンを作っていて意識するのは、絶対速度、ユーザにとっての使いやすさ、の二点である。このあたりについて日頃実感していることを述べてみたい。

2.1 RISCの進歩

並列処理マシンの目標性能を設定するには、たとえば実時間処理応用のように現場からの具体的な性能の要求があるのが一番自然である。しかしAI応用でそのようなケースはまだ稀である。我々のスタート時点、つまり数年前に考えていたことは、「きっと数年以内には、意味ネットやフレームをはじめとするネットワーク型知識の応用が実用段階に達するはずだから、とりあえずある程度速い意味ネットワークマシンを作ってみて、評価はその時点できっと動いているはずの各種応用プログラムを対象に行なおう」ということであった。しかし数年を経てみて、あまり状況は変わっていない。おかげで今は、なるべく一般性のあるベンチマークの開発に労力をさかれている。

並列処理マシンをとりまく状況として、ここ数年で大きく変わったことはRISCの成功だと思う。これは、専用マシンのアーキテクチャを考える人々にとっては、かなりの精神的圧迫になっているものと想像する。ひと昔まえは、ミニコンと比較して何倍速くなったとかいう論法が通る時代だったが、いまのワークステーションは実に速く、新しく作るマシン自体の存在そのものがゆるぎかねない状況とさえ言える。

応用を意味ネットに絞り、Cでプログラミングした場合を考えると、これまでの我々の実験データからみて、たとえば最近出たECL SPARCチップなどは、大型機との処理時間の差が1桁を十分に切っていると推測する。

スーパーコンはどうかというと、筑波のCRAY XMPを例にとると、意味ネットではベクトル化が困難なため、大型機の富士通M780に比べて5割程度遅いと考えられる。（いまクレイのCライブラリが変でデータがとれない）。つまりスーパーコンでさえ、最近は安閑としていられないのではないかとおもう。（もっとも筑波のクレイは安めなためメモリが遅いし、最近ではSX3が出てしまった）。これまで不動と（私には）思われたスーパーコンや大型機の優位性をゆるがしかねないところまでRISCが進歩するとは全く予想もしなかった。

このような結果は、意味ネットで書いた知識ベース処理の特徴（演算処理よりもメモリアクセスの多さが処理時間を支配する）によるものであり、もちろんスーパーコンの意義を疑問視しているわけではない。しかし、いずれにしても最近の情勢はアーキテクトには相当シビアであって、並列処理マシンを作るに当たっては

相当覚悟がいるものと思う（億の金と労力をつぎこむだけの価値の表明）。RISCがこれだけ速くなったのだから、目標性能としてスーパーコンあるいは大型機に伍する性能を掲げるつもりでとりかかる必要がある。ただ台数をたくさん並べてみて並列処理効果の実験をしてみようかといった時代はもう済んで、真に並列処理マシンの価値を問われる時代になってきたと思う。

2.2 知識ベース

話題をかえて、Prologで知識ベースを構築した場合を考えてみる。ここで想定する知識ベースは、PSではなく、フレームや意味ネットなどのネットワーク型のものとする。我々のグループは、当初意味ネットのインタープリータを書くのにPrologを使っていた。意味ネット自体は、宣言的知識の集合であるが、これだけでは十分な推論が行なえないため、手続き知識の付加（フレームでの手続き付加と同じイメージ）を必要とする。その場合、Prologは実に書きやすくて、インタープリータの作成も短期間で行なえた。現在も、知識獲得システム用の知識ベースの記述には、Prologで実装した意味ネット言語(IXL)を用いている。

しかし、大きな意味ネットの記述には用いていない。これは処理時間のためである。おそらく1万リンクを超えるシステムの構築には、Prologでの実装は適さないと考える。ユニフィケーションは、意味ネットにおける単純なリンクたどりに頻繁にもちいるにはあまりに高級で、コスト（時間）がかさむからである。

しかし、Prologの書きやすさ、とくにプロトタイピングに使った場合の効率の良さは他言語にくらべても相当なものなので捨て難い。そこで我々は次のようなアプローチをとった。つまり、意味ネット処理用の述語を用意し、これらを通常のProlog(Quintus)の中から呼べるようにする。そして、それら意味ネット処理用述語の実行は専用マシンにまかせ、解を全解探索で求める。

仮にPrologのAND、OR並列を活かすことを横方向の並列処理にたとえるならば、我々のアプローチは縦方向の並列処理であり、いわゆる機能分散である。ちょうど画像処理プロセッサが、CPUから原データに近いレベルに並列演算機能をシフトしてスループットをあげているように、低レベルのネットワークデータ構造のレベルで可能な並列処理はすべてすませ、Prolog側にはより抽象度の高いデータのみ供給するようにすれば、処理時間の向上に加えて、Prologプログラムの構造も見通しがよくなると考えた。

このような考え方が実際にうまくいくのかどうか、我々は今実際のマシンの作成を通して検証しようとしているが、そこで早速出てくるのがベンチマークの問題である。一般性があって偏らず、かつ意義のある例を用いた評価が重要である。このことを考える時、コネクションマシンのいくつかの論文で、その道のエキスパートがCM用に専用アルゴリズムを考えて評価していることを思い出す。

蛇足ながら、IXMは現在Prologのバックエンドとなっているが、Lispのバックエンドとしても使うことが可能である。

2.3 ユーザにとっての使いやすさ

並列処理ではMIMDマシンの構築が一つの理想ではあるが、その際、超、とはいかなくても並列性の制御をユーザにどこまで委ねるかは昔から議論されてきたところである。しかしメッセージ交換にしろ共有記憶タイプの同期機構を用いるにしろ、MIMD型の並列性指示を一般のユーザに多く期待することには無理があるように思う。トランスピュータは、ユーザがMIMD型の並列性指示を行なえる数少ない商用マシンの例であるが、いま自分でトランスピュータ73台のシステム用にプログラムを開発していて、ベアなマシン形態のままではとても普通の人には解放できないなというのが実感である。やはり並列処理、とくに粒度の比較的大きいプロセスレベルの並列プログラミングは、ある種のプログラミングスタイルというか、その習熟にある程度の経験を要する。加えて、デッドロックの危険は常につきまとうので、デバッグ技術のノウハウもいる。一般ユーザの場合、バグが出ても、まず自分のソフトのエラーだけを気にかければよいはずだが、マシンを作る側はさらにプログラム以外にハードも疑わねばならない。（これは当り前のこととはいえ、システムの規模が大きくなってくると実につらい。）

やはり現状ではユーザのつかいやすさと処理能力のバランスという点でSIMDが一番御しやすいと思う。コネクションマシン、DAP、AAP2（そしてIXM）といったSIMDマシンが商用化あるいは実働化されてい

るのはその意味で当然かもしれない。これらはいずれもデータレベルの並列性に着目したSIMD処理であり、その処理がサブルーチン化でき、かつデータ数分の超並列処理の効果があげやすい。

3 夢

知識表現、知識ベースの観点から、最近辞書が面白いと思っている。もともと意味ネットの提案は、キリアンがBasic English Dictionaryの語義文をノードとリンクで表現し、英単語の意味の交差を求めようとしたことから始まった。この辞書は基本850語だけを使って語義文を書いている。つまり、850の基本語の意味だけで、辞書に収録する20000語の意味を表現しようというものである。現在でも語義の意味ネット化の話しは複数の場所で進行しているようである。MCCのCYCも、こちらは辞書ではなく、百科辞典から基本400記事を選び、それをベースとして約40000記事の知識記述を行なおうとしている点でキリアンと共通性がある。また、文の意味のあいまい性の除去を目的として従来から意味梁性の研究がおこなわれているが、これも広義にみて知識表現といえる。

辞書だか百科辞典だかわからないような巨大な知識ベース（常識?）がまず欲しい。（本当はこれが一番先にくるべきであって、並列処理を考えるのはそのあとのことである。）そして、それを計算機の集合体だか能動的なメモリだかわからないような超並列ハードウェアの上に載せてみたい気がする。いま超並列というと、すぐ1ビットプロセッサの集合体になってしまうが、もっと他の形態があってもいいはずである。

たとえば連想メモリである。我々のグループでは、意味ネット処理だけでなく、連想メモリによる算術論理演算についての実験も行なっているが、これがなかなか面白い。かつて60年代に連想メモリを使った種々の数値演算アルゴリズムが盛んに提案されたが、大容量の連想メモリがないため、実現されることはなかった。どのアルゴリズムも基本的にはビットシリアルで時間はかかるものの、連想メモリ内の全ワードに対して並列に行なえる利点がある。仮に足し算に50マイクロかかったにしても、連想メモリに1000語（のデータ）あれば、1データあたり50ナノでできることになる。我々はNTTの20KbitCAMを用いて256KWのシステムを作っており、16ビットの比較演算では18マイクロ（実測）かかるが、1データあたりに換算すると、0.068ナノとなって、かなり魅力的である（約14400 MOPS）。しかも、この種の演算ではデータを取り出さずにその格納場所で演算できるので、いわゆるフォンノイマンボトルネックと称される、CPUとのデータトラフィックが極めて少なく保てる。従って複雑な構造の知識ベースに対する、算術論理演算を含むqueryの処理などにおいてかなり使えると考えている。

参考文献

1. D. Lenat, et al., CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks, AI magazine, p65-85.

2. 樋口「意味ネットマシンIXM第2版の概要」、計算機アーキテクチャ研、情報処理学会、9月、1989.

3. T.Higuchi, et al., The prototype of a semantic network machine IXM, ICPP 1989.

超(OR)並列推論のための基本アーキテクチャと負荷分散アルゴリズム

長沼 次郎　小倉 武

(NTT LSI研究所)

1．はじめに

並列推論マシンの研究が活発化している。本稿では，高速逐次推論マシンを簡単な結合網で多数結合してOR並列を実現するVLSI向き超並列推論マシンの基本アーキテクチャと，通信量の少ない，局所制御による高速負荷分散アルゴリズムの概要を示す。計算機シミュレーションによる性能評価を通して，本超並列推論マシンの実現性を明らかにする。

2．基本アーキテクチャ

2．1　基本思想

（1）通信量の少ない並列処理

VLSI技術の進展を考慮しても，プロセッサ単体に比べ，ネットワークの性能向上は小さい。500KLIPSの推論プロセッサの隣接間を10Mbit/sのシリアルリンクで接続した場合，通信コスト／推論コスト比は20倍以上となる。この傾向は今後より大きくなる。

（2）局所制御による全体の統一動作

クラスタを用いて階層化した場合，10^3台以上の並列処理の実現には，クラスタ間を10^2台規模で並列処理しなければならず，オーバーヘッドが大きくなる。均一な局所制御による全体の統一動作を行う。

（3）均一かつ単純な構成の追求

要素プロセッサと簡単なネットワークを一体化したような均一な構造を目指し，メモリLSIのように単純な拡張を可能とする。

2．2　アーキテクチャ

本超並列推論マシンの基本アーキテクチャを図1に示す。論理型言語のOR並列処理を前提に，高速逐次推論マシン(ASCA)を要素プロセッサとし，各要素プロセッサ当り数本のシリアルリンクで結合したVLSI向きアーキテクチャを採る。

各要素プロセッサ内で実現する基本並列処理単位は，論理型言語のゴールのリダクション過程における選択枝を含むORプロセスである。要素プロセッサ間は，通信量の少ない，局所制御による高速負荷分散アルゴリズムにより並列に動作させる。局所制御による全体の統一動作を実現することにより，要素プロセッサの個数に依存しない構成とし，均一な任意個数の要素プロセッサで任意規模の並列推論プレーンを実現する。

3．負荷分散アルゴリズム

本超並列推論マシンでは，通信量の少ない，分散制御が可能な4つの新たな負荷分散アルゴリズムを採用する。

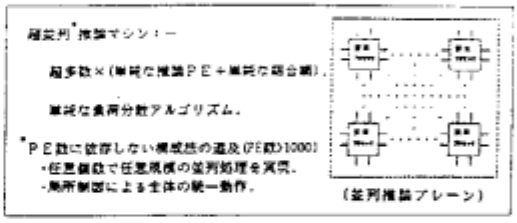


図1．超並列推論マシンの基本アーキテクチャ

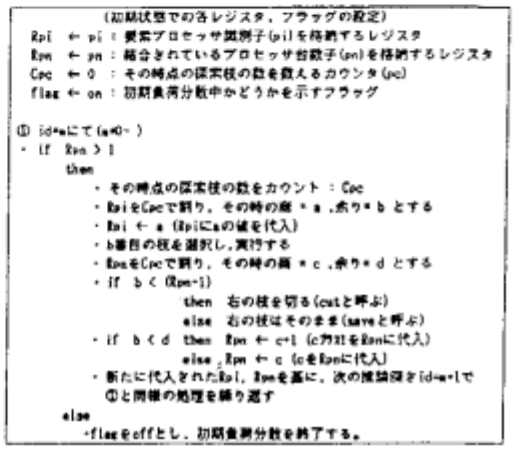


図2．初期負荷分散アルゴリズム

推論段さ	各要素プロセッサ PE0	PE1	PE2	PE3
id=0	Rpi = 0 Rpn = 4	Rpi = 1 Rpn = 4	Rpi = 2 Rpn = 4	Rpi = 3 Rpn = 4
	Cpc = 2 a = 0 b = 0 (cut) c = 2 d = 0	Cpc = 2 a = 0 b = 1 (cut) c = 2 d = 0	Cpc = 2 a = 1 b = 0 (cut) c = 2 d = 0	Cpc = 2 a = 1 b = 1 (cut) c = 2 d = 0
id=1	Rpi = 0 Rpn = 2	Rpi = 0 Rpn = 2	Rpi = 1 Rpn = 2	Rpi = 1 Rpn = 2
	Cpc = 3 a = 0 b = 0 (cut) c = 0 d = 2	Cpc = 4 a = 0 b = 0 (cut) c = 0 d = 2	Cpc = 3 a = 0 b = 1 (save) c = 0 d = 2	Cpc = 4 a = 0 b = 1 (save) c = 0 d = 2
id=2	Rpi = 0 Rpn = 1 flag off	Rpi = 0 Rpn = 1 flag off	Rpi = 0 Rpn = 1 flag off	Rpi = 0 Rpn = 1 flag off

：各プロセッサの各推論段さでの選択する探索枝とその選び方(右を切る，切らない)を示している

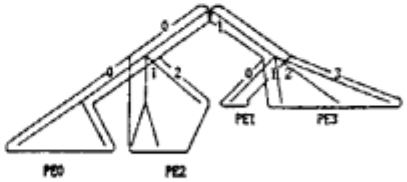


図3．各要素プロセッサの内部状態の遷移と選択するORプロセス

３．１　初期負荷分散（ＩＬＢ）

大規模な並列推論マシンにおいては，稼働直後の稼働率を高めるため，初期ゴールの高速な負荷分散が必要である。初期ゴールが与えられると同時に各要素プロセッサが異なるORプロセスを選択して稼動状態になる新たな初期ゴールの負荷分散アルゴリズムを採用する。本アルゴリズムの概要を図２に示す。

この本アルゴリズムを適用した場合の各要素プロセッサの識別子，台数子等の内部状態の遷移と選択したORプロセスの概念を図３に示す。各要素プロセッサは独立に，各推論深さで，推論木の分岐数，自らのプロセッサ識別子，割り付け可能な要素プロセッサの台数等を用いて簡単な演算を行い，節を起動する。要素プロセッサ間の通信は必要ない。

３．２　環境再生負荷分散（ＲＬＢ）

負荷分散に際しては，実行環境自体を転送せず，その実行環境の生成に不可欠な少量の情報だけを転送し再生する。概念図を図４に示す。α（通信／推論コスト比）＞１では，実行環境を転送するより再生した方が高速な負荷分散が実現できる。その高速化は以下のように定式化できる。βを通信量の低減化率（再生/転送），環境転送の負荷分散時間を１とすると環境再生の負荷分散時間は（$\beta + 1/\alpha$）となる。単位負荷分散の高速化率（再生/転送）：γは，

$$\gamma = 1 / (\beta + 1/\alpha)$$

と表される。α>1，β<1で本アルゴリズムの効果が現れる。例えばα=20/1，β=1/50とした場合γ=14となり，高速な負荷分散が可能となる。

負荷分散に際して転送すべき情報は，稼働中の要素プロセッサの識別子，および過去の負荷分散の履歴情報（負荷分散した推論深さ，実行中の枝）である。これらの情報は実行中，負荷分散の発生毎にその履歴情報として要素プロセッサ内に蓄積する。負荷分散された要素プロセッサ(Pj)では，要素プロセッサ(Pi)の初期ゴールから負荷分散までの実行環境を，転送された情報から決定的に生成する。その実行環境の生成過程を図５に示す。要素プロセッサ(Pj)では，まず要素プロセッサ識別子(Pi)で初期負荷分散を行なう。その後，転送された情報に従い，実行すべき枝を選択しながら実行を進める。

３．３　遅延負荷分散（ＬＬＢ）

負荷分散によるオーバーヘッド（通信，ORプロセスの分割）がなければ，負荷分散が多いほど高い並列台数効果を得ることができる。しかし，負荷分散によるオーバーヘッドが存在する場合（現実の世界），バックトラックで直ぐ到達できるような非効率な負荷分散は抑えるべきである。

このため，自らの中でバックトラックすべきか，負荷分散すべきかをある時点で評価し，効率的な負荷分散が可能な時点まで負荷分散を遅延させる。その評価は，現在の推論深さ(idc)と分割できる推論深さ(ids)を比較し判定する。遅延係数をk(0<k<1)として，k=idc < idsの間は負荷分散を抑制し，それを越えると負荷分散する。このような単純な評価によって，非効率な負荷分散の発生を抑制できる。

３．４　仮想負荷分散（ＶＬＢ）

実際の要素プロセッサの台数の仮想係数倍の要素プロセッサ台数を仮定して，初期負荷分散を行う。要素プロセッサ内でひとつの仮想プロセッサ識別子で処理が終了した場合，次の仮想プロセッサ識別子を発生して処理を継続する。また，負荷分散に際しては，仮想プロセッサ識別子のみを転送する。

これにより，各要素プロセッサで発生できる仮想プロセッサ識別子と初期ゴール近くの負荷の偏りが相殺され，初期負荷がより均等化される。また，初期負荷分散中に負荷分散が可能となり，また負荷分散の通信量をさらに低減化される。

これらの新たな負荷分散アルゴリズムの適用により，簡単なネットワークでも高速な負荷分散が実現できる。

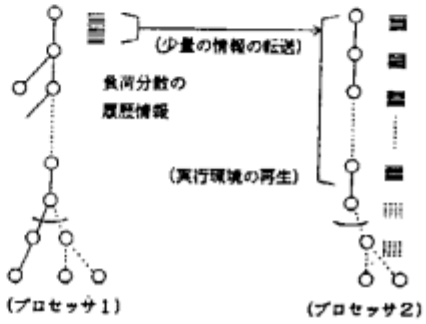


図４．環境再生負荷分散の概念図

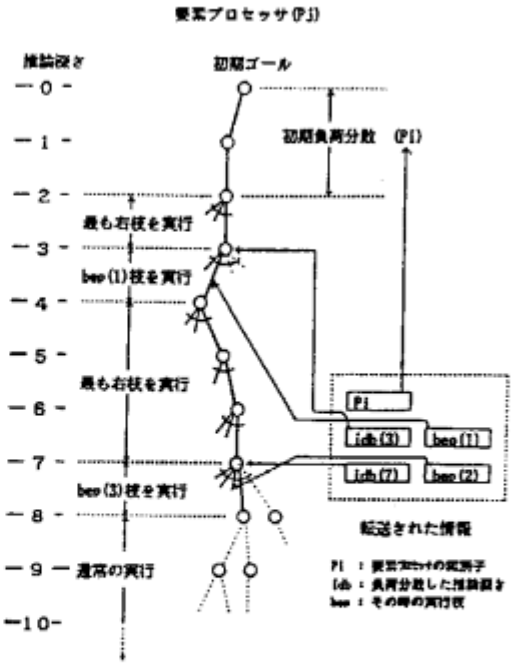


図５．実行環境の再生過程

4．性能評価

4．1　シミュレーション方法

本超並列推論マシンおよび各種負荷分散アルゴリズムの性能評価を行うため，本超並列推論マシンのシミュレータを作成した。本シミュレータでは，(1)応用レベルのより大きな問題を扱えること，(2)10^3台を越える並列推論プレーンを扱えることを目標にモデル化した。Trace Driven Simulationの手法により，問題の推論木をベースとして，round-robinによる並列シミュレーションを行う。また，シミュレーション深度はリダクションレベル，要素プロセッサの状態としては，推論処理（再生を含む），通信処理，空き処理の３状態とした。本シミュレータの概要を図６に示す。

本シミュレータは，まず，通常のProlog処理システム上にメタインタプリタで実現されている推論木抽出プログラムにより，評価の対象Prologプログラムの推論木の情報を抽出する。次にＣ言語で記述されたシミュレータに並列推論プレーンサイズ，ネットワークトポロジー等のシステムパラメータを与え，与えられた推論木の情報を並列にトレースする。これにより，要素プロセッサ内の稼働状態の遷移，並列台数効果，総負荷分散数，総通信量等を評価する。なお，本シミュレータはSUN3上に構築されており，現在，推論木:3Kノード，PE数:16Kまで扱える。

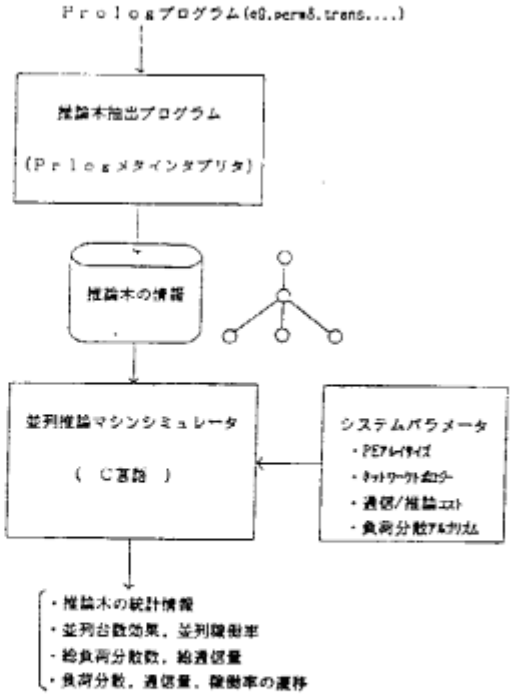


図６．シミュレータの概要

4．2　ベンチークプログラム

ベンチマークとしては典型的なＯＲ並列の問題として，Queens(q9,q10)，リストの並び換え(pers8)，自然言語の生成(trans)を取り上げた。これらのプログラムの推論木の並列実行可能なノード数等の動的な特性を図７に示す。この他にもデータベース，論理回路のテストパタン生成等の問題も取り上げたが，基本的な特性はこれら４つに類似している。

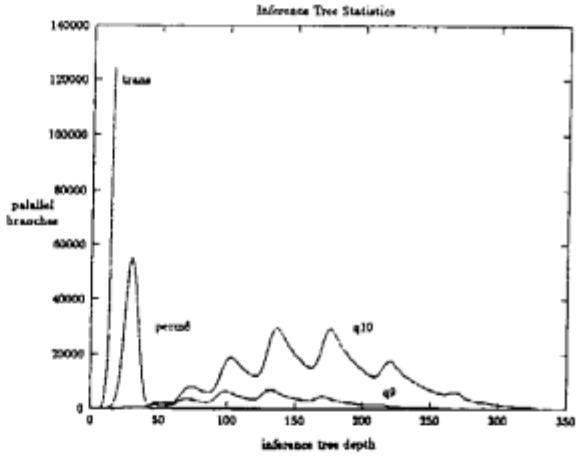


図７．ベンチマークプログラムの特性

4．3　シミュレーション結果

（１）システムパラメータ

並列推論プレーンは1024台：32x32として，要素プロセッサ当りのリンク数，結合トポロジーはシミュレーションにより決定した。リンク数は８本，トポロジは８隣接メッシュの対角位置の結合距離を４としたものを採用した。リンク数，結合距離と並列台数効果の関係を調べた結果，リンク数：８，結合距離：４でほぼ完全結合の場合に迫る並列台数効果を得ることができた。一方，1024台をハイパーキューブ接続しても完全結合に迫る並列台数効果を得た。このことは論理型言語のOR並列処理の場合，局所的なネットワークでも十分な並列台数効果が得られることを示している。

（２）稼働率の遷移

本超並列推論マシン全体における各要素プロセッサの稼働状態の遷移を図８に示す(pers8)。環境を転送するモデル(copy)と本モデル(RLB+ILB+LLB+VLB)の場合の違いを示している。図中の各線は，下から

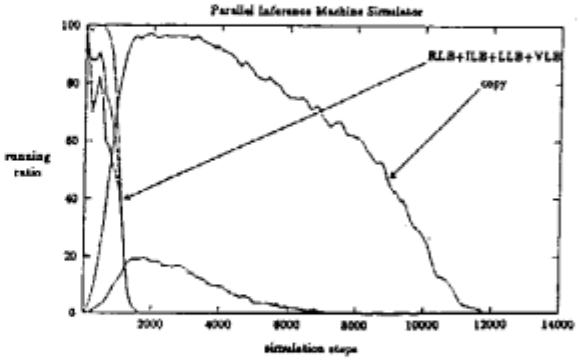


図８．稼働状態の遷移(copyモデルと本モデル)

推論処理，再生処理(本モデルの場合のみ)，通信処理，空き処理のそれぞれの割合を示している。copyモデルでは，推論処理の稼働率は高々20%程度であり，80%にもおよぶ大部分は通信処理に費やされる。これに対し本モデルでは，再生処理を除いても推論処理は80%程度を占め，通信処理は僅か10%程度である。総通信量は約1/80に低減化されている。

（3）並列台数効果

本超並列推論マシンの並列台数効果を図9に示す。q9の場合150倍，q10の場合350倍，perm8の場合380倍，transの場合620倍程度の並列台数効果が得られている。q10，perm8のように十分な並列度を有するプログラムにおいて，300倍を越える並列台数効果が得られている。transのようにより並列密度の濃いプログラムでは500倍を越える並列台数効果を得られる。

一般化は難しいが，10^3台規模の並列推論マシンでは，要素プロセッサ台数の3割強の並列台数効果を得ることがひとつの目安かもしれない（10^2台規模では5割強）。一方，いずれのプログラムも，要素プロセッサの台数が16台の時，15.5倍を越える並列台数効果を得ている。本超並列推論マシンアーキテクチャは，小規模な並列推論マシンの実現にも有効な手段である。

4．4　考察

超並列推論マシン実現の鍵は通信量の低減化である。α(通信／推論コスト比)に依存しにくい並列システム設計をいかに実現するかである。本超並列推論マシンの並列台数効果のαに対する依存性を図10に示す(perm8)。図中，実線は環境を転送するcopyモデルの場合を示している。copyモデルの場合αの依存性が極めて大きく，αの増加に対し並列台数効果は大きく低下する。これに対し，4つの負荷分散アルゴリズムを用いる本モデルでは，図中点線に示すように，各負荷分散アルゴリズムがαが1より大きい領域で効力を発揮し，αの依存性をそれぞれ減少させる。

このように，通信量の低減化する4つの負荷分散アルゴリズムによりにαに依存しにくいシステム設計が可能となる。これによりVLSIアーキテクチャ(αの高いシステム)による超並列推論マシンの実現が可能となる。

5．おわりに

将来的な10^3を越える超並列処理のための基本アーキテクチャと負荷分散アルゴリズムを中心に示した。超並列推論マシン実現の鍵は通信量の低減化である。通信量を低減化する4つの負荷分散アルゴリズムがα(通信／推論コスト比)に依存しにくいシステム設計を可能にし，VLSIアーキテクチャによる超並列推論マシンの実現を可能にする。

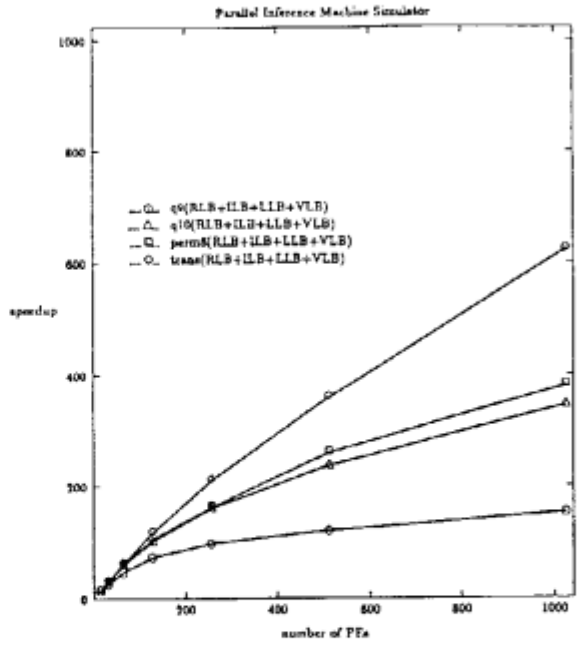


図9．本並列推論マシンの並列台数効果

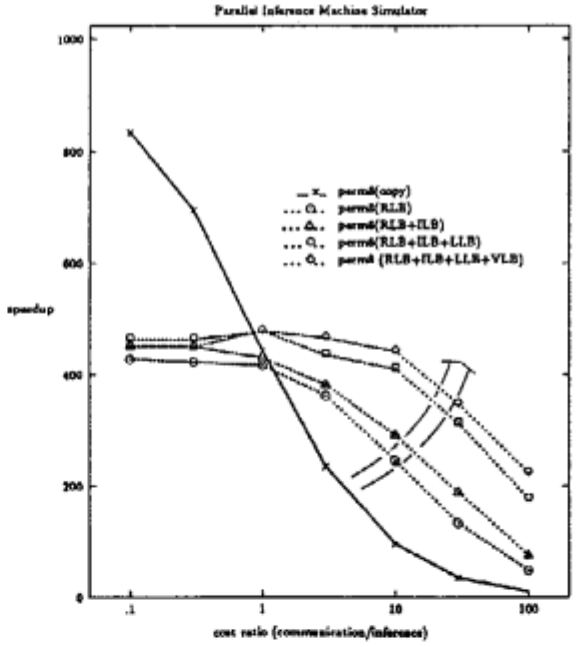


図10．α(通信コスト／推論コスト比)に対する依存性

［謝辞］　本研究を進めるに当たり，終始御指導頂いた前須藤カスタム化技術研究部長，武谷主幹研究員，木村主幹研究員，並びに中島主幹研究員に感謝いたします。また，種々の有益な御討論を頂いた研究室の方々に感謝いたします。

昨日まで，そして今日から

ＩＣＯＴ４研
六沢　一昭

昨日まで

マルチＰＳＩ周辺での出来事

1. マルチＰＳＩへＫＬ１処理系をインプリメントした。
   とにかくＧＣに本気で取り組んだ。
   ローカルに，そしてインクリメンタルに処理が行なえる方式を考えた。
   また"ひどい"状況を起こさないことに注意を払った。
   「並列環境では起こりうることは必ず起こる」ということをいつも頭においた。

2. ＦＧＣＳでマルチＰＳＩのデモができた。
   (今だから言えることですが，７月頃はできる実感が全くありませんでした)。

3. 行動的な並列プログラミング研究が始まった。

できたこと／実感

1. 並列論理型言語は並列処理を記述するのに向いていることを確信した。
   "並列プログラム"から連想される
   「(バグのために) 実行するたびに結果が異なる」
   といった恐ろしい状況はデモプログラムの開発において起こらなかった。

2. 使い物になるＫＬ１処理系を並列マシンに実装できた。
   実装の基礎技術は一通り開発したと思う。

3. プライオリティを利用した枝刈りは非常に効果がある。
   並列には絶対できないと思われた逐次アルゴリズムをプライオリティを使うことによって並列に実行できたこともあった。

わからないこと

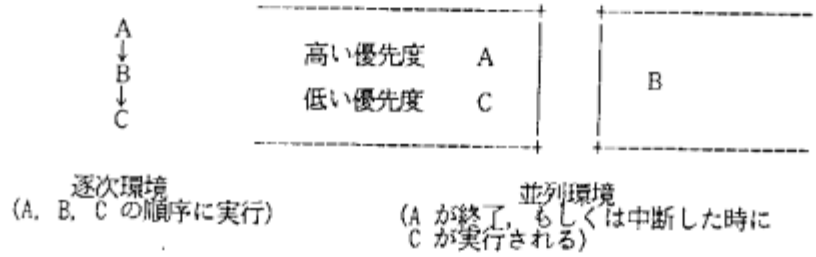
1. 外部参照(他のプロセッサへの参照) の振る舞いがわからない。
   「平均的にはこうだ」とか，「こうあってほしい」という予想や希望はある。
   しかしその予想や希望がどの程度正しいのか本当のところわからない。
   また質のわるい外部参照の振る舞い(例えばチェインが延々と延びてしまうもの)
   については全く見当がつかない。

2. 起こりそうな"ひどい"状況とはどの程度のものだろうか。
   プロセッサが１台ならば"ひどい"といってもまだ限度がある気がする。
   ところが並列環境では信じられないほどひどいことが簡単に起こるに違いない。
   「９９％の確率で１０分で実行が終わります。
   でも１％の確率で"ひどい"状況が起こって１週間かかります。」
   ということは絶対に避けなくてはならない。

今日から

1. 本気で並列プログラミングができる環境が整ってきた。
   これからは並列プログラミングをおおいに行なう時代だと思う。
   並列プログラミングでは逐次ではあまり体験できなかった
   　プロセッサのアイドル
   　デッドロック
   を体験することができる。
   また信じられないくらい実行の遅いプログラムを書くこともできるだろう。
   人間ひとりだと混乱しても時間がたてば何とか直る。
   ところが人間がたくさんいるとパニック状態になってしまう。
   並列プログラミングは，時にはこのような状況との戦いになるかもしれない。

2. プライオリティを利用したアルゴリズムを考えよう。
   プライオリティは並列プログラミングにおいて主要な要素になるに違いない。
   "条件分岐" や "待ち合わせ" といったことと同列に扱われることを僕は予想する。
   将来「プライオリティ制御のない並列計算機は使いモノにならない」ということが言われるのではないだろうか(ここでいう並列計算機はマルチＰＳＩ的なＭＩＭＤ型計算機を指す)。



   ３つのプロセス(A,B,C) がある。
   B の結果によっては C は実行する必要がなくなる。
   このような場合，並列環境では例えば上図のようにして実行する。

3. プログラミング・レクリエーション on the Multi-PSI
   - o パフォーマンスメータを使ったアニメーション
   - o 最も遅いプログラム(ただし稼働率が低くてはいけない)
     → "リダクション数／稼働率" が最低のプログラム
   - o 最も短い時間で大域ＧＣを起こさせるプログラム
   - o メッセージが最多のプログラム
   - o 一括ＧＣが最多のプログラム
   - o マルチＰＳＩウイルス

僕が初めて計算機を使ったのは今から１３年前，高校の選択科目の実習でした。
WATFIV(実習用 FORTRAN) で図形の面積を求めるプログラムを動かした記憶があります。
当時計算機を使うということは全く新しい経験で，すべてが楽しかった。
並列プログラミングもあの時と同じ感動を与えてくれるものと期待しています。

ＩＣＯＴ－ＷＧ Workshop

これまでと今後。

2年前 → これまで → これから

近山 隆

概要

2年とちょっと前の PIM-WG 合宿のときに私自身が書いたものを読返しながら，当時私が考えていたことが最近までにどのように展開されてきたかを振り返り，今後の方向を考えてみる．

1 対象とする問題を限定すること

2年前の合宿の資料に私は，PIM を考える際に対象とする問題を以下のように限定することを提案した．

- 大きな問題だけを相手にすること．小さな問題は並列に実行してもつまらない．
- 大きな並列計算機だけを考えること．小さな並列計算機を考えてもつまらない．
- 開始/終了時の逐次性は気にしないこと．処理の本体に十分な並列性がある大きな問題については，前後の逐次性が高い部分など気にする必要はない．

これらはとりあえず並列処理の研究にはいるにあたって，面倒なことは考えなくても十分実りある成果の得られる問題がたくさんある，と考えて提案したものである．

現在までの Multi-PSI, PIM の研究開発は，かなりプロセサの数が多い場合に最適化する方針をとってきた．既に世の中では，ずっと少ないプロセサ数の並列計算機が商業的成功を収め始めている．それらと競合する研究領域を狙わない方針としたのは，当面の戦略としても適当であったろうし，国家プロジェクトとしての性格上，本来とるべき道をとったと思っている（それにしても日本の計算機業界は，こういうちょっと新しい分野となると，また遅れを取っている）．

小さな問題を考えないことは，今はもう常識になっているだろうと思う．「並列にすることの意義は，同じ問題をより高速に解くことではなく，より大きな問題を同じ時間で解けるようにすることにある」と Ehud Shapiro がどこかでいっていたそうだが，それは私と同じ方向の主張である．

開始/終了時の逐次性が問題にならないという主張については，まだ実証したとはいいきれない．64個の比較的遅いプロセサしか持たない Multi-PSI で解ける程度の問題では，サイズが小さ過ぎて，あらかじめプログラムコードやデータを分配する手間などが無視できない．しかし，既に実験されているシステムでも，少なくともコードやデータの初期分配が主な処理ではないようなサイズの問題を解いており，開始/終了時の逐次性など問題にならないという主張が実証できる（というか，常識になる）日は遠くなかろう．

いずれにせよ，上述のような性質を持った問題だけを考えても，まだ十分にやり残した仕事はあり，余計な雑音に惑わされずに，このような限定された問題（といっても，十分広い範囲の問題）に対象を絞った研究開発を進めていくべきだと思う．

2　局所性について

2.1　ハードウェアの局所性

私はハードウェアはアルゴリズムの局所性を生かすものでなければならない（完全結合などとんでもない），大規模並列処理で本当に局所性が生かせるのはメッシュしかない，と主張した．そして，三次元以下のメッシュでうまく解けないような算法は，そもそも並列処理向きの算法ではない，と書いた．

現在までに作ってきたハードウェアの範囲内では，まだプロセサ内外に関する以外の局所性が問題になる規模になっていない．したがって，まだこれは実証できるところまできていない．これはだいぶ先のことになりそうだが，どのくらい先なのだろう．

3 年 4 倍の集積度向上が続くとすると，15 年で千倍，30 年で百万倍であるから，Multi-PSI のような並列計算機の最大規模のものは，15 年後には十万並列，30 年後にはだいたい一億並列程度のものになるはずである（コネクションマシン的なものは百億並列程度になるのだろう）．これは，バスはもちろん，（コネクションマシンのようなものではない）本当のハイパーキューブ接続など不可能な台数である．ツリーはまだ 15 年くらいなら生き残れるかも知れないが，用途が限られるだろう．やはり 30 年たてばメッシュしかない，と今でも思っている（私は，そしてワークショップに出席される皆さんも，後30 年くらいは生きながらえそうであるから，このぐらい先のことまでは心配しておいた方が良さそうである）．

2.2　だれが局所性を考えるか

私は局所性を生かす機構はハードウェアとソフトウェアで適当に分担すべきである，と主張した．そして，その分担の設計は難しい仕事だ，と書いた．

現在まで作ってきた処理系では，従来の逐次型計算機で既に確立された局所性を生かす機構（たとえばキャッシュのブロック単位管理）以外は，すべてソフトウェアないしファームウェアの責任になっている．基本的にはこの方針は適当であろうと考える．ハードウェアを投入するのは，ソフトやファームを用いた経験を積み重ねた上で，どのような局所性を意識するのが適当であるかがはっきりわかってからで十分であろう．

今後の研究方向としては，やはりまずソフトウェアによる実験を通してどのように局所性を生かしていけるのかを考え，十分確立した方式を得た後に初めて言語や処理系の設計に反映するのがよかろう．ハードウェアを考えるのはさらにその後で良い．現在の KL1, Multi-PSI, PIM は，こうした研究を進めるために十分な汎用性を持っていると思う．

3　局所性の生かし方

私は以下のことを主張した．

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，並列度が十分にあるだけでは不十分で，局所性が良くなくては話にならない．

集中管理の回避: 情報の集中管理は必ずボトルネックになるので，決してすべきでない．

現地調達主義: 情報の生産地と消費地は近づけなくてはならない．

局所性を殺さない負荷分散: 負荷分散は大切だが，プロセサ稼働率を上げることより局所性を崩さないことの方が重要である．

これらの主張は，Multi-PSI 上のソフトウェア開発の指針として役だっていると思う．

PIMOS の設計にあたっては，集中管理を回避することを大きな課題のひとつとして，かなり徹底した分散管理を行ったが，できあがった木構造による資源管理はごく自然なものになった．OS のように管理を主たる役割とするようなプログラムでさえ，管理の分散化は自然にできることが示せたわけである．

局所性を殺さない負荷分散が重要であることは，Multi-PSI 上の実験的な応用プログラムの開発・評価過程で鮮明になった．たとえば並列自然言語パーサ PAX の例では (PAX 方式による並列処理は全体としては成功したとはいえないのだが) むやみな負荷分散よりは局所性の維持の方が重要であることが明らかになった．同様の結果は KL1 のセルフコンパイラの負荷分散実験からも得られている．

現地調達主義は，本来はデータベース的な性格を持つプログラムに重要になるはずのものである．まだこれに対応するような実験は十分に行われていないが，今後のより大規模な応用を考えていく過程で，この指針の正当性も明らかになっていくだろうと考えている．

今後も，個別の問題に対しての局所性を利用した通信の小さい算法の研究開発を続行すべきであろう．そうした具体的な事例研究の中から，いくつかの局所性維持パラダイムが次第に明らかになり，汎用的なアルゴリズムも育ってくるものと考えている．

4 信頼性

私は信頼性向上のためにはハードウェアに冗長性を持たせるのが良い，と主張した．現在までに開発したシステムは，まだこのような方式が必要になるほどのハードウェア規模にはなっていない．

現在，PIMOS のファイルシステムの設計を行っているが (現在は FEP のファイルに頼っている)，その中では各所に冗長性を持たせることによって信頼性を上げることを考えている．たとえば，まったく同じ内容を持つディスクを複数持つミラーリング方式，空き領域やファイル構造に関する種々の管理情報を冗長に保持することによって，障害回復を行う方式がそれである．これは，比較的 MTBF の短いディスクを多数結合するであろうことに対応したものである．

こうした冗長性を利用した信頼性向上策は，将来より大規模なシステムになる場合，ディスクまわりに限らずシステム全体に適用すべきものであろう．今までファイルシステムに使われてきた技法，今回 PIMOS のファイルシステムのために設計している手法は，こうした信頼性向上策の雛形になりうると思う．

5　優先度概念

2 年前のワークショップでは主張しなかったことで，その後に Multi-PSI 上の応用プログラム開発を通じて明らかになったことのひとつが，優先度概念の重要性である．

本来の GHC では，ふたつのゴールの実行順序については:

どうでもよい: どっちが先でも，あるいは同時並列でもよい．

決まっている: 片方が作るデータを見ないと，もう片方は動けない．

の二種類しかなかった．この二種類だけでは，非常に重要な概念である「優先度」を表すことができない．つまり:

優先関係がある: どっちが先でも，同時並列でもよいのだが，効率を考えると一方の実行を他方に優先した方がよい．

という制御ができないのである．

たとえば，アルファベータ探索時には深さ『優先』探索を行う．このとき，『下方』と『右方』の探索の関係は単純な並列でもなければ厳格な順序でもない『優先』という関係であって，先に左の枝を深く潜らないと右の枝の探索ができない，ということはない．左に深く潜っていってもまだ暇にしているプロセサがあれば，右の方もどんどん探索して構わないのである．逐次処理の場合，厳格な順序と優先関係の区別はなかった．ひとつのことをしている最中に他のことをすることは，どうせできなかったからである．優先関係は並列処理の導入に伴って，その重要性が現れてきた概念なのである．

こうした優先度概念が必要になるであろうとはあらかじめ予想していて，簡単な優先度機構 (単一プロセサ内でのみ有効な 4096 段階の数値による優先度) は KL1 の仕様に含めた．しかし，応用プログラムの実験を通じて種々の優先度機構の利用形態が明らかになってくるに従い，現在の単純な優先度機構では機能不足 (応用ソフトに負担をかけ過ぎる) らしいことが徐々に明らかになってきている．

今後は，負荷分散とリンクしたより高度な優先度機構を導入を考えて研究を進めるべきであろう．この研究方向は並列ソフトウェア研究の重要な一分野に発展しそうな気がする．

6　おわりに: 目標をしっかりと持って

大きな問題を大規模並列で解く，焦点はここに絞るべきである．これを忘れて，小さな問題を少し速く解けるようにするための妥協や，規模の小さな並列システムでしか生きないような方式の採用は，極力控えるようにすべきである．

プロジェクトが一応の収束に向かう時，目標が何だったのかを常に心していかないと，小さくまとまった (ように見える) 研究成果を出して満足してしまう結果になりかねない．そういう小市民的精神は，こと研究に関しては，決して利益にならない (人類の，あるいはプロジェクト全体の利益にならないだけではなく，長期的に見た研究担当者個々人の利益にもならない) と思う．

（ＰＩＭ－ＷＧ Workshop　資料）

これからの知識プログラミングと並列記号処理

雨宮 真人 （九州大学）

１．宣言型記述と並列処理

記号処理のような複雑なデータ構造の処理には宣言的な記述を用いる方が分かりやすい。データの変換を記述したい場合にはその変換をどのように行なうか（how）ではなく、入力と出力の間の関係（what）を関数を用いて記述する。データ構造をパターンで表わし、入力パターンから出力パターンへの変換を関数によって記述する。

データベースの蓄積・検索においてもデータ構造をパターンで表現し、データベース中のデータ構造と照合をとることによって検索する。一般的には、検索したいデータがベータベース中のデータと直接パターン照合できるとは限らない。この場合には演繹推論によってパターンが自動的に変換されパターン照合が試みられる。このような演繹推論の記述には、パターン照合を一般化した単一化（unification）の概念を基礎とする述語論理表現が用いられる。

演繹推論によるデータベース検索の場合、検索条件を満たすデータは一般に複数個存在するからこれらを処理するためには集合データの記述が必要である。したがって、また、データの写像変換を記述する場合にも集合データの記述が必要となる。

２．宣言的記述　－関数型記述と論理型記述－

データの写像変換については、数値計算の場合は直観的である。また、リストやパターンなどのデータ構造の処理にも写像変換の概念による記述がわかりやすい。これら変換に記述には関数による記述が適している。

データの変換操作の記述では、入力と出力が予め定められている。つまりデータの流れ（データフロー）は入力から出力へ向けて単一の方向性を持っている。このような方向性をもった処理は関数的に記述すると分かりやすい。

しかし、一般には必ずしもデータの流れが単方向の処理ばかりとは限らない。述語論理の記述では述語で示される命題を真とするようなデータがデータベース検索あるいは写像変換によって求められるが、この場合どの引数が入力でどの引数が出力かは実行するまで判らない。すなわち、データフローは双方向性となる。

論理型記述ではこのようにデータフローの方向を固定せずにデータ間の関係を記述できる。これは見方を変えれば、述語によってデータ間の制約関係を記述する方程式であると捉えることができる。

3．宣言型プログラムの並列性

宣言型プログラムにはかなりの並列性が内在している。このことは先に示したプログラムの実行過程を図で表してみると分かりやすい。
宣言型プログラムでは、関数や述語はそれぞれ起動されると独立の計算実行体（プロセス）として活性化され他のプロセスと並行して動作する。集合の各要素について処理する場合には、その各々の要素毎にプロセスが活性化し並列に処理が進む。

再帰展開による並列実行の過程ではまた集合やベクトルの要素を時系列のストリームに分解して流し、個々の要素に対する処理をパイプライン的に行っていると見ることもできる。この場合、集合やベクトルから要素への分解、処理、組立はループ実行によって行われる。ストリーム処理の概念は宣言型プログラムを実際化する上で（処理効率上）重要なものである。

また、履歴依存の処理については、末尾再帰実行ごとに新しいプロセスが活性化され直前のプロセスからその状態情報を引き継いで行くことにより実行される。各プロセス活性体の間で履歴に依存せずに処理できる部分は先行して実行させることが出来るので、ループの unfolding によって平行に実行させることができる。

手続き的記述では逐次的実行を前提として処理操作手順を記述する。このため、並列プロセスの間の同期制御を明示しなければならないなど並列処理の記述が困難となる。これに比べ宣言型記述では処理の操作手順を直接意識する必要がなく、特に並列実行のための制御を特に意識する必要がない。宣言型プログラムでは特に逐次、並列を意識する必要がなく、プログラムに並列性が内在していれば自然とその並列性が実現される。この意味で宣言的記述は逐次、並列を超越したものである。

4．宣言型プログラム実現上の問題点と解決法

4.1 宣言型記述の問題点

さきに示したように、宣言型記述は、特に集合やベクトルなどのデータ構造の処理をパターン表現を用いて分解、変換、組立という自然な形で記述することができる。しかもその記述から高い並列性が抽出されるので、陽に並列処理制御を意識することなく並列処理の記述を行うことができる。

宣言型プログラムはこのような長所を持っているが、処理効率の点で問題がある。一般に、プログラム言語への要求を考えるときその記述性と効率性との間には相矛盾する要素をはらんでいる。宣言型プログラムは記述性に優れるが効率上の問題を解決しなければ実用にならない。処理効率の問題は、コンパイラとハードウェアによって解決を図るということになる。

4.2 コンパイラの課題

処理効率については、実行速度と資源使用量の２面で考えなければならない。まず、実行速度について考えてみよう。

宣言型記述では実行制御が表に現れない。コンパイラは宣言型プログラムを解析して実行制御情報を抽出して命令の実行順序を定め機械語に変換することになる。関数型記述に対してはデータ依存関係を解析して実行制御情報を抽出する。論理型記述や制約記述に対してはまず述語依存関係や制約依存関係を解析してデータフローを抽出し、そこから制御情報を抽出ことになる。このようにプログラム解析はデータ依存関係や関数（述語）依存関係などの解析を行うが、プログラム解析によって生成された機械語コード［注5］は手続き型言語のコンパイルコードに比肩できるものでなければならない。このため、コンパイラには精密でインテリジェントなプログラム解析が要求される。

また、2章の例でわかるように、一般に宣言型記述では関数や述語は小さな単位で定義されることが多い。したがって、これらの関数、述語の活性体であるプロセスは小粒なものとなる。このような小粒のプロセスが多数起動され並行して動作するため、プロセス切替えなどのオーバヘッドが増大し実行速度が著しく低下することになる。プロセス切替えのオーバヘッドを低減することは主にハードウェア構成上の問題となるが、コンパイル段階で小粒の関数（述語）をいくつかまとめ大粒の関数、述語に組み立てていくことも必要となろう。

資源使用の問題に関しては、まずメモリの問題がある。宣言型プログラムをその意味の通りに実行すると構造データの scrap and build が頻繁に起こる。しかし実際には scrap される構造データの多くはその一部を変更して再使用可能なものである。コンパイラはデータ構造の生成・消費関係を解析し、無駄なデータコピーをしないような機械語コードを生成することが要求される。

無限個の資源を仮定して並列プロセスの展開・活性化を行えば理論上は実行速度のオーダが上がるようなプログラムでも実際は有限資源の個数分以上の速度向上は望めない。むしろ、むやみに並列プロセスの展開・活性化を行うとプロセス数の爆発を来たし、資源の枯渇やデッドロックを招くことになる。このためプロセス活性化のスケジュールが大きな問題となる。また、資源の負荷に偏りが生ずるようなプロセス発生を行うとハードウェア性能を十分に引き出せなくなる。負荷を資源に均等化させるための負荷制御も重要な問題である。

4.3 アーキテクチャ上の問題

以上の問題は、静的なプログラム解析によってコンパイラで解決できる場合もあるが、動的に変化するような問題はハードウェアで解決しなければならない。コンパイラで解決すべきかハードウェアで解決すべきかはトレードオフに絡む問題であり一概に結論は下せない。ここでは、このような問題を考慮し、並列記号処理マシンのアーキテクチャを設計する際に検討すべき項目を列挙するにとどめる。

① プロセッサ結合 -- プロセス間の通信はプロセスの起動、引数と結果の授受が主となる。したがって、プロセッサ間結合のトポロジーは木構造が好ましい。

② プロセッサ構成 -- 多数のプロセスの並行実行を効率的に制御できること、特にプロセス切り替えのオーバヘッド最小限に抑えることが必要である。一般に構造メモリへのアクセスには時間がかかる。このアクセス時間の伸びがプログラムの実行速度に影

響を与えないような実行制御が必要である。コンパイラによって局所性が抽出できるならキャッシュメモリなどの導入も効果的である。

③ 構造メモリ -- 特に構造データを記憶するメモリはすべてのプロセッサから均等にアクセスできるような共有メモリが好ましい。使用済み領域の回収と自由領域管理、割付機構等のメモリ領域管理機構を内蔵するなど高機能化を図ることが必要である。

④ 結合ネットワーク -- プロセッサやメモリの独立性を高め非同期実行を行わせるために、ネットワークはパケットスイッチ方式が望ましい。ネットワーク内に負荷制御機能を持たせることも必要である。

5．おわりに

知識情報処理記述の新しい枠組みとして、宣言型プログラムの特徴と問題点、コンパイラと並列マシン設計上の要求条件について考察した。ここで述べたような宣言型プログラムを実用的なものにして行くには、多くの実際的応用への適用を試み、また実用化のための多くの問題を解決していかなければならない。

第五世代コンピュータ開発プロジェクトもいよいよ後期段階に入り、並列推論マシンＰＩＭの開発、論理型言語をベースにした並列知識プログラミング言語の開発、制約プログラミングやプログラミング解析・変換等の基礎ソフトウェアの研究などが進んでいる。これらの研究開発を通して、宣言型記述を意識した新しい情報処理システムの枠組みが次第に整えられつつある。第五世代のプロジェクトはこのような新しい枠組みに向けての研究開発環境を整えることに大きな意義があるといえる。この意味で並列論理型言語や高並列マシンＰＩＭの開発によって得られた知見や経験をもとに、宣言型記述を基礎とする知識情報処理システムの研究開発がさらに活発になるものと期待する。

並列処理のこれまでと今後

東京大学　田中英彦

１．これまで

コンカレントプログラミング

マルチプロセサ：主記憶共有、バス共有

用語：キャッシュ、ネットワーク、処理単位、負荷分散、データ割付け

言語：Ｆｏｒｋ、ｌｏｏｐ，述語

述語：OR、R-AND 、Committed-Choice、AND-OR

応用：偏微分、画像、Ｔｏｙ

ＯＳ：ＩＰＣ支援、グループ管理

並列機：１０＊＊２／PE単体、１０＊＊５／クラスタチップ

２．今後

意識並列／無意識並列の分化

専用：ＰＤＥ／画像に特化、多量使用

汎用：メイン（10**2 、信頼、連続）、新フレーム（10**3 ，オブジェクトのベースマシン）

プログラミング：オブジェクト指向＋専用サブルーチン

シミュレーション指向プログラミング、ＯＯＰ＋ＯＯＤＢ

主論理逐次＋並列探索（自然モデル）＋特殊並列（画像）＋分散並列（協調）

システム

オンチップキャッシュ（４ＭＢ）、６４Ｍｂチップ

10**２～３　　：新フレーム

10**５～６　　：データ並列

10**８～９　　：偏微分専用機（1000*1000*1000）10**4 TFLOPS

10**１１～１２：neuronet（10**8 neuron＊4000結合）

信頼性

ハードMTBI～チップ数、冗長並列ラン、シミュレーション指向デバグ

値感度（幅by Fuzzy、不十分データby neuro）

MINUTE

議事録

1　開会の挨拶 ： 田中 英彦

- 「それぞれの発表は会社・ICOT を背負っている訳ではないので、楽にやりましょう。」
- 「時間いっぱいしゃべるのはやめて、discussion の種を提供しましょう。」
- 「並列処理はなかなか現実のものになっていないが、面白いテーマです。」
- 「自由な討論を期待します。」

2　ここまで分かった . . . と思う。

2.1　PIM のアーキテクチャとクラスタ内処理系 ： 後藤 厚宏

(1) PIM の歴史

1985. 8	Pure Prolog から GHC へ → 言い訳をしない PIM、自分で使いたい PIM
1986. 4	現在の PIM が見えてきた時期 ― 階層構造アーキテクチャ、並列キャッシュ
1986.10	KL1 処理系開発開始
1987. 4	各社ハードウェア設計開始 命令セットの検討
1988.10	VPIM PSL 開発開始

中期の成果　次期 PIM の LSI 開発がスタートした。
50 - 100 ns マシンサイクル, KL1 用の命令セット, 一貫性キャッシュ, 高速ネットワーク

(2) クラスタ構造

- 現実路線を採っていった。
  PE 台数のピークが数百でいいから数年後の技術で高い性能を得るものを作ろうと考えた。
- 将来にわたって PE 台数に普遍的に適用できる構成は一回の研究でできるとは思っていない。
- PE 台数が 10 個でも自由に使いこなすのは難しいだろう。
- どこかに特徴があれば、計算機処理にインパクトを与えることができる。

(3) 要素プロセッサ

- 各社 RISC、CISC のそれぞれのアーキテクチャで作っているが、どれがよいのかよくわからない。
  処理系の作り易さがポイントになる。
- アーキテクチャ設計は、思いきりの良さが大切である。
  何かに明確に的を絞ったプロセッサ (DISC) を目指すべきである。

(4) GC

- 現在、PIM の研究は GC に Dedicate している。近山教に感化されている。
- MRB、クラスタ内コピー GC、WEC が主になっている。MRB はハードウェアサポートしている。

(5) もっとコンパイラを

- WAM → KL1-B → New KL1-B の研究の流れ
- もっとコンパイラが頑張れる余地がある。

(6) 言語処理系開発

- マイクロプログラム → 便利であるが、規模が大き過ぎ、限界に近い。
- PSL/VPIM → 言語処理系を作るための言語処理系

・ 言語処理系は軽くあるべき → PSL/VPIM 不要論

・ 並列処理のモデルをもっと単純な形に置き換えたい。

(7) 討論

・ 現在、KL1 の下はマイクロ命令しかない。その間に PSL がある方が良い。

・ KL0 は 16K ワードでフラット、KL1 は 12K ワードで複雑である。

2.2 KL1 分散処理系の実装経験 : 中島 克人

(1) 処理系の紹介

・ 目標 → 大規模でも動かしたい → 疎結合並列マシン

・ ネットワーク機能 → 双方向チャネル 9 ビット 200 ns オートマチックルーティング

・ CPU → PSI-II とほぼ同じ。

・ キャッシュ → 4K ワード direct map

(2) KL1 分散処理系は複雑か ?

・ 実際、複雑だと思う。なぜか ?
色々なアイディアを入れるため、OS の機能の一部をサポートするため
〔シンプルなものが生き残る〕(疎結合マシンはだめか ?)

(3) KL1 の特徴

・ 実用性 荘園、優先度指定、GC、デバッグ機能をサポートしている。

・ スケーラブルマシン用処理系

メッセージ増加を抑えている。データ重複コピーを防止している。

・ OS サポート

性能に効く部分、KL1 で書き難い部分をサポートしている。
メモリ管理、実行資源管理、優先度管理など

(4) スケーラビリティ

・ 荘園、単親を用いて分散管理する。

・ WTC によるゴールの終了判定を行っている。

・ 局所 GC を可能にする外部参照ポインタの管理

輸出表を用いて管理している。

・ WEC による PE 間即時 GC を行っている。

・ 構造体グローバル管理を行っている。

プログラムコードはオンデマンド・ロードしている。

(5) PE 間処理の評価

・ 実験プログラム (ベントミノ、PAX) による評価 (16PE 版)

・ 通信は、ファームウェアネックで、ネットワークは遊んでいる。

・ ベントミノは稼働率が 95% で、かなりよい。

・ PAX は、稼働率が低く、動いている時も 40% がネットワーク通信に使われている。

・ ネットワーク稼働率は、0.19 ～ 0.6% で、がらがらである。

(6) 考察

・ 疎結合マシンは生き残るか ?

・ プロセッサの数の平方根で速くなれば満足しよう。

(7) 討論

・ 16PE だけでなく、他の PE 数でも評価するべきだ。
・ PSI の場合は、通信に専念しても 100% の稼動率にはなり得ない。
・ PIM の場合は、使い切れるだけの CPU パワーはある。
・ 通信処理のハードウェア化は難しい。

2.3 ユーザと並列推論マシンの間、そして日頃気になっていたこと : 市吉 伸行

(1) 並列化の問題点

・ オーダではなく、台数効果を良くするのが目標
・ ユーザが気持ち良くプログラムできるようにする。

一生懸命書いたプログラムが速く動くようにしてあげる。

(2) 平均的ユーザ像

・ 解くべき中大規模の問題を持っている。
・ 問題の解き方は分かっている。
・ より大きな問題を速く解くため、並列化するのはやぶさかではない。
・ KL1 でプログラムを書いてもいいと思っている。
・ Multi-PSI/PIM を使える環境にある。

これらは、理想的ユーザ像？

(3) ユーザにとって何が厭か？

・ 並列に書いても速くならない → かえって遅くなることさえある。
・ プログラムに並列性があるのに十分に並列性を活用できない。
・ 並列プログラムを並列マシンにマッピングする仲介者が、ユーザと並列マシンとの間に必要

(4) 台数効果

・ 台数効果を上げるには

プロセッサ数を増やす。平均稼働率を上げる 負荷バランスを良くする。
無駄な計算をなるべくしないようにする。通信オーバーヘッドをなるべく小さくする。

・ リニアスピードアップを目指すが、他の考え方もあり得る。

(5) 各プログラムの特性

プログラム	特徴	台数効果
queen	通信量は少ない	PE 台数に近い
詰め込みパズル	動的負荷分散	64PE で 50 倍程度
最短経路問題		平方根程度
詰め碁	alpha-beta 探索	平方根～ PE 台数程度
PAX	通信量が非常に多い	16PE で 3～4 倍

(6) PAX について

・ チャレンジする価値のある問題
・ Bottom Up Parsing 法 → Dynamic Programming → 計算を減らすために通信を増やしている。
・ PE の割当てを工夫して通信量は減らせたが、負荷バランスは非常に悪い。

(7) 考察

- 全自動負荷分散は夢のまた夢 → 問題ごとに負荷分散を考える必要がある。
- きれいにマシンを隠せないのなら、プログラマにマシンを意識させる方が良い。

(8) 気になっていること

- スケーラビリティ (台数拡張性の定義について: スケーラブルは 相対的・極限的概念
- KL1 が手続き型言語と比べてオーダが悪くなるか？ (contant time embedding の問題)

  MRB によって定数倍のコストで解けるようになっている。
  enqueue_with_priority は現在 O(n)
- 台数効果 → 並列プログラムと逐次プログラムの実行時間は違うので、考慮する必要がある。
- データフロー並列とデータ並列 → これは本質的に違っていると思われる。 → 他の並列の種類はないか？

(9) 討論

- オーダは、今後多くの PE をつなぐ時に問題になる点である。

3 私はここまで分かり、ここが分からないと思う。

3.1 PIM/c のご紹介 : 中川貴之

(1) 目標

- 中期 → 2PE × 2 クラスタ
- 後期 → 8PE × 32 クラスタ
- ネットワーク回りの負荷分散の実験主体に行う。

(2) 理想と現実

- 使い易いマイクロ命令仕様 (理想)
- AI32 をベースにした仕様 → 記憶手段の使い分けが問題点 → ファームウェアへのしわよせとなる (現実)

(3) 制御ソフトについて

- VPIM からマイクロ命令への自動変換ツール作成中
- キーポイント → メモリアクセス削減、自動生成の最適化

(4) ハードウェアについて

- ターンアラウンド重視の TCMP
- 5 状態キャッシュ、スリットチェック機構、2way インターリーブの共有バス

(5) わかったこと

- 並列システムといっても逐次システムの成果を流用することが有る。
- 逐次文化で作ったハードウェア、ソフトウェアを再構成／統合するツール群必要 → VPIM は重要である。

(6) ここをわかりたい

- 並列ソフトウェア開発環境として、まず並列処理システムの実験評価支援ツール
- 並列向きの問題とは何？
- 良い並列処理は、逐次処理のタブーを破って良いのか？
  → 安定性能は必要？、非決定性は妨げとならないか？

(7) 議論

- AI32 はファームウェアが苦労することでハードウェアが簡単になればそれで良いではないか。
  しかし、実際はそんなに簡単にはならず、せいぜい 10% 程度の軽減
- マシンサイクルは 50 ns、0.8 μ ルール
- 各ユニットゲート数は、最大のもので、100k ゲート
- マイクロ命令の幅が 104 bit といってもそれほど沢山のことができる訳ではない。

3.2 私はこんなことしかわかっていないしこんなこともわからない : 中島浩

(1) わかったこととわからないこと

- 負荷分散は大切 → P^3(Processing Power Plane) が提案された。 → 結構大変だ
- ではどうすれば良いのか ?

(2) P^3 ははたして美しいのか ?

- 通信の局所性 → P^3 は通信の局所性を考えたモデルである。
- モデルの抽象性 → メッシュ以外のトポロジとは相性が悪い。
- 情報の局所性 → P^3 は局所情報のみを用いる。
- 記述の容易性 → P^3 はプログラム構造と密着している。
- 実現の容易性 → P^3 の平面幾何は結構難しい。
- 結局 P^3 は思ったほど良くない。

(3) どうしたら良いのか ?

- マシンに依存しないためには → 距離はメッシュの専売特許ではない。
  ハイパーキューブでも定義が可能
- 楽に記述するためには → データとプロセスの関係を記述する。
  このデータは好き、このストリームは太いなど
- 実行時に太いストリームの両端をくっつける。
  どの辺でくっつくか話し合う。本当の太さを測る。
- 簡単に作るためには → 「最適なプロセッサで」と言うのは大域的で大変
  「当たりを見回して一番良さそうな所で」で我慢する。

(4) 議論

- データ、ストリームの好き嫌いでプロセスを引き合わせると、一個所に集まるのでは ?
  何らかの反発力も必要だろう (本来プロセス間には反発力が働くなど)。
  しかしこれは結構難しいかも知れない。
- 本当にローカルのコントロールだけで旨く行くとは思えない。
  人間社会にも管理職は存在し、その役割が有る。 → では階層的なのが良いのかも知れない。

3.3 PIM/p のネットワークについて : 久門耕一

(1) PIM/p について

- 256PE(32 クラスタ) のハイパーキューブ結合
- 1 クラスタから 6 本の足、8PE は共有バス結合
- 各クラスタは SCSI インターフェースを持つ。
  (ディスクが付けられ、将来データベースマシンとしても使える)

・ ハイパーキューブの特徴 → ネットワーク直径が小さい。

筐体間結線を減らすため、一つのノードに複数のPEを接続

(2) ネットワークの問題

・ デッドロックの問題 → 固定ルーティングにより解決 (最下位bit優先ルーティング)

(3) 自動ルーティング機構の導入

・ 各クラスタの負荷値による自動ルーティング
・ 最下位bit優先ルーティングの制限下で最小負荷クラスタの方向にパケットを送り出す。
・ あるクラスタの持つ負荷値と周辺のノードの負荷値とを元にそのノードの負荷値を設定する。
・ これは山からプロセスが転がり落ちるイメージ、もともとは宇宙モデルから発したもの

(4) 評価

・ シミュレーションではうまく行った。
→ 世の中で最小の負荷値のクラスタに送る方法と隣のみを参照する方法との比較
→ 前者はグローバル情報の収集に時間がかかるため最新情報を得られないのが欠点

(5) 議論

・ この様なマシンは保守まで含めて考えると大変
・ CADの良いものがなかなかない (公称10万ゲートのchipにはまず10万ゲートは入らない)
・ クロストークの対策としては、クラスタ間に同軸ケーブルをつかっている。
・ このPIMは単一クロックだが、表側と、裏側のもので位相はずれている。
・ 並列キャッシュは相当難しいだろう。
・ PIMに載せたい応用ソフトはやはりCADかな？

3.4 並列処理の問題点 − 私の興味、実感、そして夢 : 富田眞治

(1) 高級言語マシンってほんとに要るの？

・ 汎用構造の方が大切ではないか。

(2) 結合はどういうのがいいの？

・ クロスバで、いけるだろう。

(3) 「可変構造型マルチプロセッサ」っていうのはどうだろう？

・ これは手段としての「可変構造」なの？
・ それとも「可変構造」にするのが目的なの？
・ いずれは目的にしたい。

クロスバでエミュレートできるだろう。

(3) ボトルネックはメモリ

・ メモリネックって、メモリそのものよりメモリへの通信がネックだ。

(4) ネーミングが大切

・ 日本人はもっとセンスのいいネーミングを考えよう。

(5) その他

・ 並列の分野で、逐次の分野と全く互換性のない文化が発生し、生き残る可能性はあるだろうか？
→ ある。

3.5 並列処理の問題点 : 柴山 潔

(1) 「論理的並列度 = 物理的並列度」が理想なんやろか？

- 真の姿は「低レベル並列処理」
- でも、誰が低レベル並列性を抽出するの？ コンパイラでしょ。（フロア）

(2) プロセサ : RISC など 軽くなる傾向

- メモリ : スヌープキャッシュなど 重くなる傾向
- キャッシュはきらいや！ こいつがハードを複雑にしとる。

(3) コンパイラの問題にしたくない。

- アーキテクトの楽しみを奪う、くさった考えだ！

(4) コンパイラとアーキテクチャのトレードオフをどこに求めるか？

- 記号処理マシンのアーキテクチャはおもしろそうって言ったってタグとマイクロだけじゃないの。

(5) 「構造の可変性」はアーキテクトの夢

3.6 並列処理の問題点 : 小池 汎平

(1) PIE64 というマシンを作っている。

- それについての苦労話が語られた。

(2) ネットワークは見た目が安心できるものじゃないとダメ。

3.7 ＰＩＭのここがわからない : 小長谷 明彦

現在の ICOT？　　下図のようになれるとうれしい！

GIP : Genetic Information Processing (遺伝子情報処理)

(1) 蛋白系列データベースから、スーパファミリという集合に属する。

- パターンを速く検出したい。

(2) GIP を、AI と並列の架け橋に！

(3) その他

- バックトラックは非常に効率のよい GC だ。

3.8 並列計算機における静的と動的 : 坂井修一

(1) アーキテクチャの研究

- 要素技術の開発 → シンセシスの研究
- 部分 → 全体の上下動
  (例) 計算機のパイプライン設計
- パイプの総段数と各段の時間長の決定
- 計算機全体の細部を総合的に見渡すことを必要とする作業

[根拠] 要素技術の進歩、プログラムの動特性解析の進歩、過去の設計の反省

- 内部動作を初めから全て考慮することは不可能

  他の場合を調整、例外に対して全体を補修

(2) データフローの世界

- 単なる要素技術としてデータフローを取り込むという立場が立て難い。
- データフローモデルに基づく計算機
- 全命令における動的同期 (マッチング)
- パケットアーキテクチャ + 循環パイプライン
- 単一代入

  ETL-EM4 ：改良型データフローモデル
  ICOT-PIM：並列型フォンノイマン型モデル

(3) 静的・動的の切り分け

- プログラマの指示に従って静的に実現する。
- コンパイラの自動検出に従って静的に実現する。
- ソフトウェア的に動的に実現する。
- ハードウェア的に動的に実現する。

(4) 静的・動的の選択の仕方

- 静的に解析可能か？
- それはどのくらいのオーバーヘッドか？

  静的な最適スケジューリングを実行以前に完全に決定するのは NP 完全
  近似解を求める → 挙動が予測困難なものがある。

[準備] スケジューリングの構成要素

- 同期
- 負荷分散
- データの分配
- デッドロック防止／回避

(5) 同期

- シングルスレッドのパイプライン + 静的シーケンシング (静的な同期)
- マルチスレッドのパイプライン + 動的同期 (動的な同期)

  という二重構造が有利

(6) 負荷分散

- 何を静的に分散し、何を動的に分散するべきか？
  - * 命令レベル：静的
  - * ブロックレベル：ブロックが十分に大きく、ブロックの生成が動的に決まるのなら動的
  - * 関数レベル：呼出しのパターンが予測可能なら静的、そうでなければ動的
- 動的負荷分散の制御はソフトウェアが行なうか、ハードウェアが行なうか？

  ソフトウェアよりハードウェアが望ましい
  コストが問題

・ 動的負荷分散の制御は集中的に行なうか、分散的に行なうか？

分散化が必須
結合網負荷分散方式

・ 静的負荷分散の手法にはどのようなものを用いるべきか？

コンパイラによる自動割り付けが望ましい。

・ EM-4

ブロックレベル・命令レベル ― コンパイラによる静的負荷分散
関数レベル ― 動的負荷分散 (裏循環路負荷分散)

(7) その他

動的負荷分散戦略を動的に変えるという研究もある。
データフローモデルの負荷の測り方

3.9 並列処理の問題点 ～ 私の興味、実感、夢 ： 樋口哲也

(1) 興味

・ 超並列処理 → 知識ベースへの応用
・ 意味ネットワークマシンの開発
・ 辞書、CYC → 並列性の宝庫 (?)

IXM (現在 2 号機) → 73 CPU (トランスピューター) → 64PE : 完全結合

(2) 実感

・ RISC の進歩 → アーキテクチャ研究の困難化
  * AI マシンの目標性能の設定
  * AI マシンの評価方法
  * RISC の進歩 → 意味ネット処理の場合、WS とスーパーコン・大型機との性能差は一桁以下
  * 並列処理マシン研究の価値表明
・ 知識ベース
  * ネットワーク型知識
  * 大規模

    IXL → Prolog の中から意味ネットワークの全解探索が可能
    述語論理の代替としての意味ネットワーク
・ 使いやすさ → 並列性の指示
  * ユーザーには期待できない。
  * MIMD の難しさ (ex. OCCAM)
  * SIMD の扱い易さ
・ 連想メモリの面白さ
  * 並列書込みの威力
  * 数値演算への利用
  * データ数分の超並列性
・ 実装、実動化

  実装を含めてのアーキテクチャ提案 → 台数, テクノロジ, 予算, 検査, 下請け業者の信頼度

(3) 興味

・ CYC : 意味ネットワーク化
・ 辞書、百科辞典的知識のハードウェア支援

3.10 超 (OR) 並列推論のための基本アーキテクチャと負荷分散アルゴリズム : 長沼次郎

(1) 超並列推論マシン実現の課題

- 通信量の少ない並列処理

  通信コスト > 推論コスト
  500K LIPS, 10M bit/s → 1 : 20

- 局所制御による全体の統一動作

  大局的制御が困難

- 均一かつ単純な構成の追求

  メモリ LSI のように単純な拡張が可能

(2) 超並列推論マシンの概要

- 超並列マシン : ー 超多数 × (単純な PE + 単純な結合網), 単純な負荷分散アルゴリズム
- OR 並列処理
- 逐次推論プロセッサをベース

  数本のシリアルリンクで結合
  VLSI 向きアーキテクチャ

- 高速負荷分散

  通信量の少ない、局所制御による負荷分散

(3) 新たな負荷分散アルゴリズム

- 初期負荷分散 (ILB)
- 環境再生負荷分散 (RLB)
- 遅延負荷分散 (LLB)
- 仮想負荷分散 (VLB)

(4) シミュレーションによる評価

- 稼働率の遷移
- 並列台数効果 (1000 台で 300 倍以上)
- 通信 / 推論コスト比に依存しにくいシステム設計が可能

(5) その他

単純なメッシュではだめ!
たとえ 16 台で 15.5 倍の台数効果が得られても 1000 台規模での性能は予測できない。

(6) 今後の予定

- 超並列推論マシンの試作 ?

  VLSI 推論プロセッサ

- 既存の並列マシンへの実装実験

  R256, IXM ?

3.11 昨日まで、そして今日から : 六沢一昭

(1) 昨日まで／マルチ PSI 周辺での出来事

- KL1 処理系の完成

  とにかく GC
  ローカルに、そしてインクリメンタルに

- FGCS'88 でデモ
- 行動的なプログラミング

(2) できたこと／実感

- KL1 は並列処理を記述するのに向いている。
- 使い物になる処理系と基礎技術を一通り揃えた。
- プライオリティは非常に有効である。

  C にプラグマを付けてもだめ、KL1 にプラグマを付けたから何とかなっている。
  (しかし KL1 でもプラグマを無視して書いたのではやはりだめ)
  プライオリティは OS を書くのに必要だったし、並列応用には極めて有効な概念 (フロア)

(3) わからないこと

- 外部参照の振る舞いが分からない。

  典型的なものは予想できるが性悪なものは分からない。

- 起こりそうなひどい状況も分からない。

  並列環境では信じられないほどひどいことが簡単に起こるかもしれない。

(4) 今日から

本気で並列プログラミングのできる環境が整ってきた。
引用:「精神論ではだめ。まともな環境が必要。(内田俊一 1987)」
1 つのキャッチフレーズでどれだけ沢山の人が楽しめるかが大事 (フロア)

(5) 並列プログラミングでは

プロセッサのアイドル
デッドロック
ひどい状態
人間がたくさんいるとパニックになるかも知れない。

(6) 最後に

初めて計算機を使った時 → 全く新しい経験で、全てが楽しかった。
並列プログラミングもあの時と同じ感動を与えてくれるものと期待してます。

4 これまでと今後

4.1 2 年前 → これまで → これから : 近山 隆

(1) 対象とする問題の限定

- 大きな問題だけを相手にする。
- 大きな並列計算機だけを考える。
- 開始／終了時の逐次性は気にしない。
  "大きな並列計算機" よりもむしろ "大きな問題" のほうが重要

(2) ハードウェアの局所性

- 三次元空間で局所性のない構成はだめ。
- いつ頃本当に物理的空間局所性が重要になるか？

年	3	15(=5 × 3)	30(=10 × 3)
倍率	4	$4^5 \sim 10^3$	$4^{10} \sim 10^6$

15年先頃気になる？　商用機では？

(3) 局所性は誰が気にするのか？

	ハードウェア	記号処理系	OS	応用ソフトウェア
これまで (Multi-PSI)	×	データ格納場所	×	ゴール分散すべて
これから (PIM)	×	クラスタ内分散自動化	負荷分散ライブラリ	新方式研究
もっと先 (VLSPIM)	確立技術をサポート	より広範囲の分散自動化	ライブラリ充実強化	まだまだ新方式研究

(4) どうやって局所性を生かすのか？

- 局所性を生かせる算法：並列性だけの算法はダメ
- 集中管理の回避：PIMOSでは階層的な分散管理を実践
- 現地調達主義：まだ実践できる大規模ソフトなし
- 局所性 >> 負荷分散：PAXなどで証明

(5) 信頼性の維持

- 冗長性によるしかない。
- PIMOSのファイルシステム（設計中）→ ソフトウェアレベルでの実験のひとつ
  - * ミラーリング（同内容の複数ディスク保持）
  - * 空領域、ファイルなどの管理情報の冗長化 → 障害回復

  → いずれ全システムに適用すべき考え方の先駆

(6) 優先度概念

- GHCでのゴールの実行順序制御

  任意：どちらが先でも、並列でもよい。
  データフロー：データができないと動けない。

- KL1ではこれに加えて

  優先関係：どちらが先でも良いが優先
  逐次プログラム：実は優先関係

- Multi-PSIではプロセサ内の簡単な優先度機構
  → 負荷分散とリンクした高度な機構の研究が必要であろう。

(7) おわりに ～ 目標をしっかり持って

- 「大きな問題、大規模並列」を目標に絞ろう。

  × 小さい問題を少し速く解けるようにするための妥協
  × 小規模並列にしか適用しない手法

(8) 討論

- 複雑なものでも本当に必要ならば取り入れることに意味がある。
  実際には、使っていないものが複雑にしている傾向がある。
- プログラミング言語として、ユーザに対しては安心のためKL1を軸と言い切るが、その一方で他の言語を考慮する。
- ハードウェアとソフトウェアの新規の開発を考えた場合、ハードウェアそのものの変更は安いかもしれないが、普通OSまで込みであるので決して安くない。
- 負荷分散の難しさ。
  優先度が低い仕事をしているプロセッサはたとえアイドルでなくても暇なプロセッサとして何らかの手段を打たなければならない。P^3もそのひとつかもしれない。各PEにタイマを埋め込み、優先度により進み具合を変えることによってPEの忙しさをはかることを試みたが、はたして優先度が高くて少しだけ走るPEは本当に暇ではないのだろうかという疑問も残る。

4.2 これからの知識プログラミングと並列記号処理 : 雨宮 真人

(1) Architecture

- Why Inference Machine ?
- Why Parallel Machine ?
- Why Unification Machine ?
  c.f. Pattern Matching

  データの流れが双方向で、さらにパラレルとなると多重環境の機構が必要となり非常に大きな規模となる。それよりもむしろデータを一方通行とした素直な機構がsimpleでよいのではないか。双方向性が現実に役に立つことはそれほど多くはないであろう。
- Why GHC Machine ?
  c.f. Dataflow

  論理型言語というシンタクスにこだわっているように思える。GHCのパラレルの本質がデータフローならば、いっそデータフローに徹すればsimpleにすむのではないだろうか？
- Why not Dataflow ?

  caching/locality　あまり無理しなくてもよいのでは？
  fine grain
  latency
- 並列性と局所性

(2) 制御 − 負荷 − Tactics

- ソフトウェアは記述性重視である。
  ハードウェアを意識したプラグマはなるべくなくすようにすべきである。
- ハードウェアは“simple is best”である。やはり、速度、資源管理が重点である。

(3) 討論

- データフローマシンのネックはマッチング、ジョインをのぞくスケジューリングハードウェア機構の付加により解決できる？
- ワークエリアとしてのキャッシュは意味があるが、ここでいうlocalityとは意味が異なる。localityがどれくらい無視できるか？

4.3 並列処理のこれまでと今後 : 田中 英彦

(1) これまで

(2) 今後

・ 意識並列／無意識並列の分化

専用： PDE/ 画像に特化、多量使用

汎用： メイン (10^2、信頼、連続) トランザクション処理をメインとしたもの。
新フレーム (10^3、オブジェクトのベースマシン、ワークステーションの発展形) オブジェクトによる自然並列。

・ プログラミング：オブジェクト指向 + 専用サブルーチン
シミュレーション指向プログラミング、OOP + 分散並列 (協調)

・ PE: 32 bitALU + 128 W + 通信
20 万 gates, 100M Hz

・ システム: 16 PE/chip, 40 pin × 12 = 480 pin
8 × 8 chip/ キャリア
16 キャリア × 32 枚 = 2^{19} 個 PE
2 × 3 箱 = 2^{21} 個 PE/ 筐体
100M × 1/8(OH) × 2^{21} × 2 本 = 50 TFLOPS

・ 64M bit 技術 =10M gates 技術
64 PE/chip 40 × 28 = 1120 pin
4 × 2 × 50 = 400 TFLOPS/2 筐体
200M Hz 2^{23} = 8 M 個 PE/ 筐体

・ $10^3 \times 10^3 \times 10^3$ PDE = 2^{30}
64 PE/chip × 10 × 10 × 10 = 64000 PE/10^3 cm^3
20 × 10 × 10 × 64000 = 128M PE/ 筐体
8 本 × 128 M = 10^9 PE
100 MFLOPS × 1/2(OH) × 10^9 = 5 × 10^{16} FLOPS
→ 10^4 TFLOPS
128 M × 100 円 = 128 億円 /6400 円 chip/1 筐体
800 円 chip/8 筐体
十分 available ?

(3) 討論

・ ICOT-PIM — 汎用機 ?

・ object
KL1 — ロジックではなくオブジェクト指向言語として使っている。

5 閉会の挨拶 : 田中 英彦

(1) まとめ

アーキテクチャ	後藤	実用になるもの。
ハードウェア	久門、中川	
OS	近山、中島 (克)、六沢	セットはそろった。
言語	中島 (祐)	C ではできないが KL1 では何とかなる?
応用	市吉、小長谷	still we go.
並列処理	見はてぬ夢	富田
	永遠の夢	雨宮
	大きな目標	近山
	新しい感動	六沢

(2) キーワード

負荷分散、局所性、動的／静的、超並列、棄ての美学、大規模 KB、再生方式
数 100 万並列、並列言語 (KL1)、priority、ハードウェア屋の復権

Proceedings of
ICOT-WG Workshop on
ここまでわかったPIM(Parallel Inference Machines)
- そして、今後。

平成元年　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
Tel. 03-456-3193