TM-1260

Report on PIM WG Grand Meeting

by
K. Hirata (NTT)

May, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5

Institute for New Generation Computer Technology

Report
on
PIM WG Grand Meeting

1993 年 3 月 22, 23 日

ICOT Annex A-1 会議室

編集: 平田 圭二 (ICOT)

目次

開催の経緯

PIM Working Group では，並列推論システムの構築技術，利用技術に関する議論を重ねてきた．そして平成 4 年度 (1992 年度) は，第五世代プロジェクト 11 年間の最終年度である．そこで，

- これまでの ICOT の PIM 関連研究の成果総括と評価，
- 世界的に見た並列マシン研究開発の状況とその中での PIM の位置付け，
- 並列 (推論) マシンの将来展望

について，次ページに示すようなスケジュールで集中討議の場を持つこととした．

1993 年 1 月に入り 久門，畑澤，平田で会議の企画を開始し，PIM WG の委員，オブザーバの方々より参考意見を聞きながら内容を詰めた．パネル討論会のテーマを公募したり，委員，オブザーバ各位からはポジションペーパを提出して頂いた．会議全体のテーマは，

「今思えばこうするべきだった」そして「後世に残すとしたらこの技術」

とした．また PIM 関連で成されてきた仕事を，一歩下がって客観的に見る，できるだけ定量的に見ることも心がけた．従って参加者も，いつもの PIM WG 委員，オブザーバに加えて，広く PIM 現場担当者，PIM 関係 OB，ICOT 研究員，ソフトウェアハウス外注者，調査国際部委員会メンバなどにも声をかけ参加をお願いした．

折しも PIM 共通一次といって，5 台の PIM 上で同じベンチマークプログラムを走らせ比較検討するというプロジェクトが進行中であり，再委託各社の担当者には PIM WG に向けてデータ収集等の面で多大なるご協力を得た．

紙面の都合でここに全員のお名前を挙げることはできないが，本会議に参加して下さった方々，企画，運営にご協力頂いた方々にこの場を借りてお礼を申し上げたいと思う．(文中敬称略)

本レポートの構成

まず前半は，参加者や OB の方々から頂いたポジションペーパをまとめた．jlatex ソースで頂いたのを編集したので，著者の思い通りのレイアウトになっていないものもあるかも知れないが御容赦願いたい．

後半は，発表順にその議事録と発表で使われた OHP シートをまとめた．特にパネル討論は 2 回とも，テープ起こしを行ないそれを議事録とした．所々発言者不明の箇所があるが御容赦願いたい．

PIM WG プログラム

「今思えばこうするべきだった」そして「後世に残すとしたらこの技術」

1993 年 3 月 22, 23 日

3/22 (月)

(発表者 (敬称略, 五十音順))

1	PIM/i	武田, 六沢	(2:00–2:45)
2	PIM/c	朝家, 今西, 垂井	(2:45–3:30)
3	PIM/m	近藤, 中島 (克)	(3:30–4:15)

<< 休憩 4:15–4:30>>

4	KL1 は裸の王様か	久門	(4:30 5:00)
5	パネル 1: 並列処理のための言語とはどうあるべきか		(5:00–6:20)

司会: 平田　　パネリスト: 雨宮, 上田, 木村, 関口, 村上.

<< 懇親会, 中国飯店にて 6:30–>>

3/23 (火)

6	VPIM とは何だったのか?	後藤	(10:00–10:30)
7	プロセス制御, メモリ管理に限らず, 設計からシステム完成に至るまでを振り返り, 私が持つ VPIM ノウハウのすべてをここに捧げます	今井	(10:30–11:00)
8	VPIM の反省としての KLIC	近山	(11:00–11:30)
9	ハードとソフトの役割分担	近藤, 平野	(11:30–12:00)

<< 昼食 (出席者全員にお弁当が出ます) 12:00-1:30>>

10	PIM/k	仲瀬	(1:30–2:15)
11	PIM/p	新井, 小沢	(2:15–3:00)

<< 休憩 3:00–3:15>>

12	(仮想) 共有メモリと分散メモリ	中島 (浩)	(3:15–3:45)
13	PIM 共通一次問題による評価 ～ 途中経過の報告	久門	(3:45–4:15)

<< 休憩 4:15–4:30>>

14 パネル 2: 並列 (推論) マシン: 未来への展望 (4:30–5:50)

司会: 後藤　　パネリスト: 天野, 関口, 中島 (浩), 松岡, 村上.

サブテーマ: (a) PIM を作って何が分かったか? 何が分からなかったか?
(b) PIM のアーキテクチャ, ハードウェア
(c) 並列アーキテクチャ研究はどう進めるべきか
(d) 推論機 / 並列機として残すべきアーキテクチャ / ハード
(e) 汎用並列マシンのベースとしての PIM
(f) 汎用並列マシンへの期待

15	結びの言葉	田中英彦 主査	(5:50–6:00)

ポジションペーパ

並列処理から見た並列論理型言語への期待

東京大学工学部 田中 英彦

1. まえがき

並列処理記述言語は、並列処理で非常に重要な位置を占める。その重要性を思い起こし、並列論理型言語の今後を考えたい。

2. 注目したい言語のポイント

- 並列処理単位の作成: 自動作成可能、細粒度、再粒を集めて並列処理単位の作成
- 単位間同期機能: データフローによる自動同期
- 記述能力: プログラム行数の少なさ

3. 改良したい言語のポイント

- 明示的並列制御機能: 必要な場合に利用可能とすること。プラグマからの脱却
- 明示的同期機能: プログラマや利用者からの意思の反映
- 可読性の向上

4. 今後の更なる研究の必要なポイント

- 細粒を集めて一つの処理単位とする手法: 組合せ、最適化、変数の配置、コンパイラ・スケジューラ
- Garbage Collection
- 利用者向け言語: 並列論理型言語はどのレベル用言語として位置付けるべきか。

5. 結論

並列処理から見れば、論理型言語は素晴らしい特質を備えている。しかし、未だ未熟な点もあり、広く使うには足りない所もある。この素晴らしい特質を残し、なお且つ並列処理の中核言語として使えるようにならないものだろうか。C 言語を中核に用いる方向は、実用的で容易であるが、そのままでは「いつか来た道」でありその先に明るい未来は見え難い。並列論理型言語のこの素晴らしい特質こそ、成果として残し、新しい並列処理研究に付加すべきポイントではなかろうか。

ポジションペーパ

PIM WG ポジションペーパ

東京大学　相田　仁

10年前に立てられた第5世代コンピュータ研究プロジェクトの目標は、（正確な表現は忘れたが）

論理型言語の並列処理を行なうことにより各種の知識情報処理アプリケーションを高速に処理する

というようなことであったと思う。10年経ってこの目標が達成できたかどうか考えてみたとき、前半の論理型言語の並列処理を行なうことに関しては、成功したといえると思う。これに対して、後半の、各種の知識情報処理アプリケーションに十分な性能が得られたかということに関しては、否定的に思う。

この10年間にICOTをとりまく周囲におけるコンピュータ科学の状況は大きく変化した。早いものからいえば、注目されるプログラミングパラダイムが論理型からオブジェクト指向へと移行し、知識処理アプリケーションのための高レベル記述パラダイムも論理型から制約充足へとシフトした。プロセッサアーキテクチャはCISCからRISCに移行し、並列マシンのプロセッサ数に対する考え方が10^2のオーダから10^5のオーダのいわゆる超並列に移行しつつある。

このような潮流の中、ICOTにおける研究もこれらをフォローして、決して世界の最先端のレベルに後れることのない研究が行なわれていたように思う。例えばPIMのアーキテクチャを見てみれば、要素プロセッサはRISCであり、確かにKL1用にオプティマイズされてはいるが、論理型言語に特化したアーキテクチャというほどではなく、HPFでも載せれば科学技術計算に用いることができそうである。

では第5世代コンピュータ研究プロジェクトの成果として世界に誇れる／貢献できるものとしては何があるだろうか。プロジェクトの前半で、並列処理に適したセマンティクスを持つ言語としてGHCあるいはKL1を提案できたことは確かにそれに値すると思う。これに対して後半の研究、例えばPIMは、確かに世界最高レベルとは思うのだが、世界に貢献できるといえる内容はあまりないように思う。

これはなぜかと考えてみるに、前半の研究では並列処理という目標に向かってどのような道を通ったらよいかトップダウン的あるいはゴール指向的な研究メカニズムがうまく機能していたのに対して、後半の研究ではKL1をいかに速く実行するか、KL1でいかにOSを書くかというように、KL1ありきの研究になってしまい、肝心の（？）何をやりたいのか（どんなアプリケーションを動かしたいのか）ということがなおざりにされていた感がある。

このようなことは実は第5世代プロジェクトに限ったことではなく、プロジェクト研究においてしばしば見受けられることである。私自身も今後十分気を付けて行かなくてはならないと考えている。

なぜこのようなことが起こるのかさらに考えてみると、プロジェクト研究の成果として、「実際に動くものを作ってみせる」ことが必要以上に重要視されている点が挙げられる。もちろん何も成果の上がらない研究というのも困るが、どんな研究でもデモンストレーションにつながらなくてはいけないというのでは、基礎研究は育たないであろう。

ポジションペーパ

PIM は KL1 の城塞から出られるか?

慶応大学 天野英晴

1 PIM は計算機アーキテクトから正当に評価されていないのではないか。

PIM は研究用の並列計算機プロトタイプとしては、世界に類例を見ない程、成功したプロジェクトといってよい。PIM/m, PIM/p, PIM/c, PIM/i, PIM/k それぞれ特徴を持ったマシンが、最大 512 プロセッサで稼働し、その上で実用的な OS に基づき様々なアプリケーションプログラムが開発されている。しかし、一般の並列計算機アーキテクチャ屋からは、推論専用の「特殊用途マシン」として扱われ、関心も低く、正当な評価も受けていないような気がする。これは、今までの PIM に関する発表が (フォローしている文献はそんなに多くはないが)、以下の点で一般のアーキテクチャ屋にとって読み難いものであったことが考えられる。

- 並列推論マシンとして、目的充足型のストーリであること。
  KL1 の実行にとって、アーキテクチャがうまくできているという議論が多く、一般の細粒度大規模並列マシンにとって PIM の中の構成技術が有効かどうかという議論に結びつかない。
- いままでの論文が KL1 とその処理系への理解を前提としていたこと。
  この部分を全く受け付けない人も居る。
- PIM のアプリケーションの中には並列 CAD 等、「推論」との関係が薄いものも多い。しかし、それらのアプリケーションの論文は、アプリケーションの載せ方の範囲に興味が制限されており、アーキテクチャとの関連性に関する検討が少ない。

2 結合網周辺で知りたい事

PIM は、確かに KL1 を実行するためのアーキテクチャではあるが、大規模な並列計算機を研究開発するアーキテクチャ屋にとって、並列推論の「推論」の部分を取ってしまっても、数多くの興味深い並列計算機構成技術が用いられている。プロセッサのアーキテクチャに関しては、KL1 に依存する部分が多いので、特に結合網周辺で、一般的な細粒度並列計算機として議論できる部分が多い。

以下、既に発表されている話もあるかと思うが、特に以下の点が注目される。

- 各種の結合網の使用
  PIM はマシンによって、メッシュ、マルチハイパーキューブ、階層メッシュ、クロスバ等多くの種類の結合網を用いており、これらに対し、交信の分布の分析、混雑状況の比較が可能である。

  そこで、様々なアプリケーションについて、まず、そのトラフィックを調べる。非数値計算の細粒度プロセスにとって、交信の局所性はどの程度あるのか。ランダム交信を仮定した理論的な評価と比べてどの程度実測値は異なるのか。結合網はどこが混雑するか。マルチハイパーキューブにおける交信負荷の分散はうまくいくのか。メッシュの場合、レイテンシィはどの程度か。これらのデータは今後の大規模細粒度並列マシンの結合網の開発にとって非常に重要である。
- クラスタ構造を持った初めての大規模マシン。
  クラスタ構造は、大規模マシンの構築のために最も有望な方法であり、Stanford DASH, Illinois Cedar 等で試みられているが、プロセッサ数は 32 前後である。これに対し、PIM は桁はずれのスケールを持つ。

  そこでクラスタ内のプロセッサ数を 1-8 に変化させ、クラスタ内外のトラフィックを解析する。クラスタ内はプロセッサ数が多すぎないか。クラスタ構造は効果があるのか。クラスタ構造はトラフィックにどのような変化を与えるか。
- 各種のスヌープキャッシュプロトコルの使用。
  スヌープキャッシュプロトコルに関しては、様々な提案がなされている割には、実際のマシン上での評価が乏しい。PIM は、その動作が一般的なプロセッサと異なるが、PIM/i は放送型で他は無効化型であり、この両者についての比較が可能である。

実際に効率が上がるのはどちらか。ピンポン現象や、不要な放送による効率の低下は実際には起きているのか。シミュレーションと実測値はどの程度異なるのか。また、PIM/p はキャッシュ間にメッセージ転送機能を持つが、この効果に関する評価は大変興味深い。

3 KL1 の城塞を出られるか

1. 現在の手続き型記述中心のプログラミング言語はいずれ行き詰まる。その時はプログラムの記述力の点でも、並列処理の記述の点でも論理型プログラミングは大変有望で、将来は一般的になるにちがいない。

2. KL1 は並列処理を記述する論理型言語の中で優れた解の一つである。

3. PIM は、KL1 を高速に効率良く実行でき、この点に関しては世界一の性能であり、素晴らしい。

以上の議論の展開は、1. を認めれば、多くの研究者にとって文句のつけようがない。他に有望な競争者がいない現在、PIM は RPS 値で評価すれば、世界最高速である。1 を認めるかどうかは人によると思うが、一寸先は闇のこの世界では、誰も頭ごなしに否定はできないのではないか。このストーリ展開で「目的充足」型論文を書けば、文句が出ることはないだろう。つまり、PIM は KL1 の城塞に立てこもる以上無敵である。

しかし、KL1 の城塞に立てこもりっぱなしだと、外部から孤立するだけではなく、城塞の限界が PIM の限界になってしまう。つまり、PIM の研究全体が、プログラミング言語としての KL1 と心中せざるを得ないことになる。専用マシンである以上、この心中はやむを得ないと考えるには、PIM は成功しすぎたマシンであろう。

心理的に、城を出るのが恐い、ということはあるかもしれない。最近の RISC の性能の向上はあまりにも急で、一般用途とした場合、PIM に勝ち目がないのではないか、という点である。しかし、並列計算機の研究用プロトタイプでは、プロセッサ本体の性能は、極端に低くなく、結合網の能力とバランスが取れていれば、問題はなく、この点で PIM は十分以上の性能を持つ。少なくとも、結合網周辺だけでも、評価、比較、検討するテーマは山ほどあり、これらについて KL1 となるべく切り離した見方から、詳細な評価と綿密な検討を行い、一般的な並列アーキテクチャの分野で積極的に発表活動を行えば、PIM は将来の大規模細粒度並列マシンの元祖としての地位を確保する機会が残されている。

ポジションペーパ

超並列コンピュータの研究課題

雨宮 真人 （九州大学 大学院総合理工学研究科）

1 はじめに

最近、計算機科学の分野では並列処理が重要な研究テーマとなっている。アーキテクチャや言語の研究に限らず、基礎理論から応用まで必ずいってよいほど並列というキーワードが現れる。これは、計算機科学に関する各分野の研究が並列を抜きにして新しいテーマを設定することがむつかしくなってきているという現状もあるが、ＶＬＳＩ素子技術の急速な進歩によってプロセッサやメモリがますます高速、高容量、小型になり、多数のプロセッサチップやメモリチップを用いた超並列コンピュータのハードウェア構成が比較的容易に実現できるようになったことによると思われる。

特に米国において、超並列コンピュータの研究・開発が活発である。多数のプロセッサを結合してTeraflopsオーダの性能を出す超並列コンピュータの研究開発を、多少エキセントリックになっているきらいはあるが、進めようとしている。これには、米国の日本に対する技術優位性が脅かされているという今日の状況にあって、いま米国に残された技術優位性をもつ牙城としての威信を賭けているという印象がする。一方、翻って我が国における超並列コンピュータの研究開発の現状を見るに、大学・研究所での研究はいろいろ行われているものの、なかなか実用化・商用化に結び付かない。この原因がどこにあるのか分析はなかなか困難であるが、企業サイドに立って考えてみると、その大きな理由はつぎのようなことではないかと思われる。

「わざわざ超並列コンピュータを開発したところで、どれだけのメリットがあるのかはっきりしない。ＶＬＳＩ技術の進歩により単体性能でも速度はどんどんあがっているではないか（しかもソフトウェアの互換性を保ちながら）。並列コンピュータを開発したとしても今までのソフトウェアがそのまま使えないのではユーザは使ってくれないだろう。ましてや超並列コンピュータのソフトなんて難しくてとても一般には使えないだろう。そんなもので、大きな市場性は期待できない。」

しかしながら、性能に飛躍的な向上が見込まれ、ソフトウェアの作成も危惧するほど困難なものでないとわかれば、状況は変わるだろうと思われる。ただし、本当にそのようになることを示すにはそれ相当の規模のシステムを作り実証することが必要であるが、そのためにはまたかなりの研究開発費が要ることも確かである。しかし、そこを冒険することから新しい技術と市場が開拓されるのではないだろうか。米国においては過去このような冒険が多く試行されてきており（ＬＩＳＰマシンのように失敗してしまったものも数多くあるが）、技術を先導してきた。超並列コンピュータの研究もこの例にもれないだろう。

本稿では、このような状況認識を踏まえて、超並列コンピュータ研究の意義と研究開発の問題点・課題、研究動向を議論し、超並列コンピュータ開発の重要性を提言することにする。

2 超並列コンピュータ研究の意義と目標

超並列という概念は、コンピュータ科学の研究者にとっては魅力あるものではあっても、コンピュータの利用者一般にとっては、実はどうでもよいことであろう。利用者にとっては、コンピュータの中の構造がどうであろうと、つぎの３点を満たしてくれればそれでよいのである。

- 自分が解きたい（あるいは処理したい）問題を速く解いて（処理して）くれる。
- 自分の解きたい問題を難しい言葉を使わずに、素直に単純に表現してコンピュータに与えることができる。
- 安く手に入り、安く使える。

つまり、高速性、プログラム容易性、経済性の３点である。

高速性の追求については、超並列によって多数のプロセッサを同時に動かして台数分の高速性を得るということでわかりやすい。しかし、これはあくまでもコンピュータの内部構造の問題であって、利用者にとっては間接的な効果としか見えない。超並列によって得られる高速性と引き替えに、プログラム記述が非常に困難なものになってしまったのでは、利用者はたまったものではない。もし別の手段で高速性が得らるならばその方がよいということになる。　経済性の追求については、素子技術、ハードウェア技術に依存するところが大きく、これは日進月歩であることは衆目の一致するところである。これに加えて、ソフトウェアコストの低減化が重要であるが、これは記述の経済性、つまり、言語・インタフェースの問題となる。

言語・インタフェースの追求は非常に重要な問題である。コンピュータの構造を超並列化して高速性を得るとしても、利用者には超並列を特別に意識させないようなインタフェース、つまりプログラミング言語を提供することが非常に重要なことである。この点については現状では研究が遅れているように思える。言語の問題は、利用者が問題の定義とプログラム記述をどのような視点にたって行うかという基本問題に関わる。これは、計算のモデルに関わってくる。コンピュータの内部構造とは関係なく、利用者にどのようなイメージでプログラム記述を行わせるのが自然であるかというところから考える必要がある。

処理の記述、プログラミングという問題について考えてみると、従来の逐次型手続きによる処理は、単一プロセッサを前提としたものであった。プロセッサ内ではあらかじめスケジュールされた操作手順にそって処理を行っている。しかし、単一プロセッサシステムにおいてもこの逐次手順処理ではきれいに処理できない問題が現実にはたくさんある。実時間システムなどでは外界の事象発生が予測できないような処理がほとんどであり、これには割り込み概念や排他制御などの技術を駆使して対処しているが、完全にエラーのない処理を記述することは非常に難しく、これがソフトウェア生産性の問題となって現れている。このように、あらかじめ計画されスケジュールされた逐次手続きで処理できる問題は一般にまれであり、大半は非同期的に発生する事象を事象駆動で処理することが要求される。このような処理の記述には、これまでの逐次型手続き処理ではなく、全く異なった概念に基づく問題の定義や処理の記述法を開発する必要がある。超並列コンピュータの研究開発の意義は、これまでの、単一プロセッサを意識した問題記述、プログラム手法とは異なったまったく別の問題記述、プログラム手法を開発することにあるといえる。

3　超並列コンピュータのイメージと研究開発の課題

超並列コンピュータのイメージと研究開発課題を、以下、記述系・言語、ソフトウェア、ハードウェア、適用領域の各項目について考える。

3.1　並列処理の構造

並列処理構造は大きく並列展開（空間並列）構造とパイプライン（時間並列）構造の二つで捉えることができる。また、並列処理が同期的に行われるか非同期で行われるかによって、ＳＩＭＤとＭＩＭＤとに分けられる。さらに、並列化の対象がなんであるかによって、データ並列とコントロール並列という分け方もできる。

超並列コンピュータはまずＳＩＭＤ型によって商用システムが作られた。Thinking Machine 社の CM1,CM2 がその例である。ＳＩＭＤ型は限られた応用に対してはその効果を発揮した。しかし、汎用性に乏しく適用領域が狭いという欠点が次第に明白になってきており、現状ではＭＩＭＤ型や、ＳＩＭＤ型とＭＩＭＤ型の中間をねらうＳＰＭＤ型が商用機開発の目標となっている。Thinking Machine 社の CM5 や Intel 社の Paragon などがその例である。

データ並列は構造データの要素にたいして処理を並列に行うということであり、データ要素の数に比例した並列化効果（スケーラビリティ）が期待できる。ベクトル、行列、集合、等の処理においてデータ並列が可能となる。データ並列はＳＩＭＤ型実行が直観的であるが、種々のアルゴリズムの実現、トータルな処理での並列化効果ということになると必ずしもＳＩＭＤ型が適当だとはいえない。コントロール並列は処理体間やルーチン間の並列実行をさす。コントロール並列で期待できる並列効果は分割統治法、再帰構造（ループ構造）、並列フォークなどであるが、これらによって大規模並列化効果（スケーラビリティ）がどの程度得られるかはまだ明確ではない。エージェント配列（関数配列）など言語レベルでコントロール並列を定義できる記述法を与えることができれば効果が顕著となるであろう。

並列性の抽出はコンパイラによって行う。最も重要な研究課題は並列化コンパイラの開発であるように思われる。データ並列のように規則性をもった並列処理構造を抽出するのは比較的簡単であるが、不規則な構造を持つ並列処理構造の抽出をいかに行うかはこれからの研究課題である。特に、記号処理、知識処理、ＡＩの分野では並列処理対象が動的に変化するのでデータ並列の効果が得にくい（処理アルゴリズムが整構造でなく、データの構造に依存する。そしてデータの構造は動的に変化する）。したがって、処理系でいかに並列性の予測、抽出を行うかが問題となる。エージェント配列、プロセスネットワーク、再帰処理などによって実行中の超並列構造が得られるかどうかを明かにすることが今後の研究の重要課題である。

いずれにしろ、並列処理構造の抽出にはデータ依存解析、関数依存解析が基礎となる。並列化コンパイラの設計は従来の手法を利用するとともに新たな依存解析アルゴリズムの開拓が必要である。

- 処理体の大きさ（粒度）

  処理体の大きさとはある処理体が他の処理体との通信を行わずに処理対象の操作を行うことのできる大きさをいう。処理の途中で他の処理体へ処理要求を発行したり、他のプロセッサ上にあるデータへのアクセスが必要

なときなどには、その結果を待つあいだ処理が中断する。このようなときは処理要求を発行するまでの処理ステップが処理体の大きさとなる。一般に実行の開始（或は続行）からつぎの処理要求による中断（あるいは処理の終了）までは他の処置体との干渉なしで一気に実行できる。このような処理単位をスレッドという。粒度とはスレッドの大きさ（ステップ数）である。

粒度はスレッドの大きさと処理体の間でやり取りするデータの大きさ（つまり通信オーバヘッド）とのトレードオフで定まる。スレッドを小さくしすぎると通信頻度が増加し、通信のオーバーヘッドが顕著となってプロセッサの稼働率を低下させてしまう。ハードウェア（プロセッサ）の稼働率を高く保つためには粒度はできるだけ大きくするのがよい。しかし、粒度を大きく保とうとすると、他プロセッサへの処理要求やデータアクセスなどに関してハードウェア構造を考慮したプログラム記述が必要となり、ユーザの負担は非常に厳しいものとなってしまい、問題を素直に記述することができなくなる。

ユーザにとっては問題の構造を素直に記述でき、またプロセッサの稼働率を下げない程度に粒度を定めるようにすることが必要である。これをいかに可能とするかが超並列コンピュータのためのプログラム言語の設計と言語処理系開発の重要な課題である。

・ マッピング

処理体をどのようにプロセッサに割り付けるか、またデータ要素をどのようにメモリに割り付けるかという問題である。割り付けの指針はプロセッサの負荷を均等になるようにすることとプロセッサ間の通信を局所的なものにすることであるが、これはなかなか困難な問題であり、まだ十分な解法は見いだされていない。問題構造を意識して記述したソフトウェアのトポロジー（処理体間通信構造）とハードウェア・トポロジー（プロセッサ結合構造）との間には当然ギャップがありこれをどう解消するかはなかなか難しい。データ並列の場合は、データのマッピングと処理体のマッピングとは一致するような方策がとられる。エージェント並列では処理体のマッピングはエージェント間のデータ／メッセージ通信が局所的になるようにマッピングされる。データ配列やエージェント配列のように比較的規則的なトポロジーが静的に定義されているような構造では言語処理系でマッピング規則を定めることができるが、一般のプロセスネットワーク構造ではエージェント間の通信は整構造ではなく、しかもエージェントの生成・消滅が動的であるため言語処理系で最適マッピングをあらかじめ定めることができない。

3.2 記述系

特に並列処理を意識させない記述法、言語方式を確立し、新しい情報処理パラダイムを開拓する。その目標とするところは、並列・逐次を超越してプログラムを記述することにある！

プログラム言語の設計に当たっては、プログラム記述・読解の容易性と構成されたプログラムの実行効率という相反する二つの要求条件を満たさなければならない。

従来のプログラム記述は逐次処理（ノイマンモデル）を基本概念としており、現状の並列コンピュータ用言語はこの逐次型プログラム言語をベースにしてこれに並列処理記述のための拡張機能を与えることによって対処してきている。しかし、これでは超並列処理のための記述性の要求には答えられない。本来、逐次処理と並列処理を区別する必要はなく、逐次・並列の概念を超越したプログラムの記述法を開発することが重要である。

我々が直観的に考えるのは処理を受ける対象と処理を行う処理体の区別であろう。（前者をオブジェクト、後者をエージェントとよぶ。）これまでのプログラミングでは入力されたデータをどのような手順によって操作するかという視点で全ての処理を記述していた。しかし、このような視点は単一プロセッサ内での逐次的な処理に対しては有効であるが、多数のプロセッサの間にまたがって処理が行われるようなシステムの記述には適さない。なぜならば、複数の処理体の間で処理が行われる場合には（逐次的な）手順による操作の順序では記述が困難でありどうしても処理対象のやり取りを含む処理体間の相互作用、つまり通信の概念が基礎になければならないからである。このような通信をベースとする処理の概念では処理対象がどのような処理体の間をどのように流れ、その内容がどのように変化するかを意識して、ネットワーク概念によってプログラムを記述するのが自然であろう。

ソフトウェアの設計においてもこのネットワーク概念が重要である。つまり、処理体をノードとし、処理体間のデータの流れ、メッセージの流れをリンクとしたネットワーク（プロセスネットワーク、データフロー、メッセージフロー、等）の構造で処理を考える。このネットワーク概念に基づく並列・分散指向のマルチエージェントシステム、データやメッセージによる同期処理など通信をベースとした処理概念の具体化が重要な研究課題となる。

3.3 ハードウェア

超並列コンピュータのイメージは 10^4 個以上のプロセッサで構成されるコンピュータシステムである。プロセッサは演算機能に加えてメモリ機能、通信機能の比重が高くなり、メモリ機能と通信機能の充実が重要である。演算処理部と通信処理部、メモリ部は独立な構成とし、演算、メモリアクセス、通信処理の並列／並行化が必要である。演算プロセッサは従来のＲＩＳＣアーキテクチャを踏襲することになるであろうが、その場合でも特に高速なスレッド／プロセス切り替えの機構を実現することが不可欠である。他のプロセッサへの処理要求や遠隔メモリへのアクセスが頻繁に起こりまたその結果が得られるまでには時間を要すため、その間プロセッサは待ち状態となる。これはレイテンシー問題といわれる。プロセッサの設計ではレイテンシーによって処理速度が低下しないような機構を実現することが重要であり、その解決のためには高速なプロセス／スレッド切り替え機構が不可欠である。

メモリは 10^4 個以上のプロセッサからなる超並列コンピュータでは共有メモリシステムはアクセスネックが生じるため適切ではない。メモリがプロセッサ毎に分散配置される分散メモリ構造にすることは避けられない。この場合アドレスの仮想化、メモリアクセスのレイテンシー解消が必要であり、これらの制御機構のメモリ内実現などが課題となる。通信機構では単純な通信プロトコルを実現し通信のオーバーヘッドできるだけ小さくすることが重要である。

プロセッサの結合トポロジーは単純なものほどよい。２次元あるいは３次元メッシュ構造やツリー構造が現実的であろう。

ハードウェア設計の原則はできるだけ単純な構成とすること。アドレス割り付けやメッセージ処理など概念的には複雑高度な機能を必要とする場合でもその処理をなるべくソフトウェア（言語処理系、ＯＳ）で解決することに努め、ハードウェアで直接サポートすることは避ける方が賢明である。

3.4 適用領域

並列・分散概念を原理とする新しい情報処理体系の適用領域は特定領域に限るものではない。並列処理によって高速実行が期待できる分野は以下のように広範にわたる。

- 信号処理、音声処理、画像処理
  （ＦＦＴ、ディジタルフィルターリング、ニューラルネット、エッジ検出、領域検出、ラベリング、等）
  - データ並列：　画像処理（画素数分の並列性、近傍通信）
  - コントロール並列：　信号処理、音声処理（パイプライン、ストリーム）
  - コントロール並列：　ニューラルネット（エージェント並列）

- 数値シミュレーション（科学技術計算）
  （行列・ベクトル計算、差分方程式の数値解法、モンテカルロ法、有限要素法、流体シミュレーション、粒子シミュレーション、等）

  - データ並列：　行列・ベクトル計算、差分方程式解法
    整構造、静的割り付け
  - コントロール並列：　モンテカルロ法、有限要素法、粒子シミュレーション
    非整構造、動的割り付け

- 非数値シミュレーション
  （事象シミュレーション、トラフィックシミュレーション、等）
  - コントロール並列：　エージェント並列、メッセージフロー
    非整構造、半動的／動的割り付け

- 実時間システム
  （トランザクション処理、データ交換システム、データベースシステム、等）
  - コントロール並列：　超多重並行処理、パイプライン
    非整構造、半動的／動的割り付け

- 記号シミュレーション（知識処理、ＡＩ）
  （記号推論、意味ネットワーク、自然言語処理、画像理解、等）
  - データ並列：　テキスト、文字・記号データ
  - コントロール並列：　再帰構造
    非整構造、動的割り付け

4　おわりに

以上超並列コンピュータのイメージと研究開発の諸問題について考察した。そして、特に超並列コンピュータのための言語・処理系の開発が重要であることを指摘した。超並列コンピュータを実用化し商用化に結び付けるためには、超並列コンピュータのメリットを明確にし、既存のコンピュータを凌駕できることを実証することが必要であるが、そのためにはより多くの応用問題を実際にとりあげ、アルゴリズムの開発、記述、実行性能の評価を意欲的に進めていくことが必要である。米国ではこの作業が日々進められ超並列コンピュータのノウハウが蓄積されている。しかし、我が国は、この点で遅れを取っているように思えてならない。国産の超並列コンピュータはまだ本格的に商用化されていない。このため言語処理系、並列化コンパイラ、マッピング方式など、超並列コンピュータに関するノウハウがなかなか蓄積できないでいる。国産の商用超並列コンピュータが早く開発され種々の応用が実験される日が来ることを願う次第である。

本稿では特に人工知能との関連については直接議論しなかった。人工知能研究でも超並列ＡＩ、分散協調など並列処理の観点からいくつか研究が行われているが、ここでは特にこれらについて議論しなかった。本稿では個々の応用について取り上げることをさけ、より一般的な観点から超並列コンピュータの構造と研究課題を議論したかったからである。人工知能の研究においても結局はヒューリスティクスをアルゴリズム化するプロセスが重要であり、曖昧な点をアルゴリズムという形で明確にしその可能性と限界を明らかにすることが人工知能研究の目的である。そしてその研究によってアルゴリズムとして切り出され効率的な処理が可能となった部分が実用化されいくと、筆者は考えている。このような観点から、人工知能の各問題が超並列処理パラダイムによってアルゴリズム化でき、自然な記述と高速実行が可能な部分を切り出していくことが超並列ＡＩ研究のキーポイントであると考えている。

このためには、まず超並列コンピュータが人工知能への適用に限らず、各種の適用分野において、性能・記述性の両面で既存のコンピュータを凌駕できることを示すことが先決であると考える。

ポジションペーパ

PIM開発プロジェクトと大学

柴山 潔 (京都工芸繊維大学・工芸学部・電子情報工学科)

[自己批判] — PIM開発プロジェクトと私 —

- (PIM-WGと関わりになる前(1983年頃まで))
  マイクロプログラム制御, VLIW(低レベル並列処理), リスト処理, 高級言語マシン ……
  ⇒ アーキテクチャ + 記号処理システム + 並列処理
- ICOTの設立と同じ頃(1984年)に「論理型言語向き並列計算機KPR」の開発プロジェクトを始める. (1990年まで約6年間)
  ⇒ 論理型言語(特に, Prolog)を知る.
  ⇒ KPRプロジェクトはプロトタイプの一部の実装に留まり未完.
- 論文数編と著書1冊をものにする.
  ⇒ まだ, 「教育」と「研究」が噛みあっていた.
- (PIM開発プロジェクトが蒔いた種)
  AIシステムアーキテクチャに関する研究のとっかかり.

[総括-1] — 大学における関連研究の動向 (ICOT初期) —

- システムプログラム(特に, 言語プロセッサ)
  ⇒ software + firmware
- アーキテクチャ
  ⇒ マイクロプログラム制御(firmware)による問題適応性を利用.
- 記号処理, 並列処理
  ⇒ 応用よりもシステムに近い立場が多数.

⟹ PIMの応用分野の開拓という面からは, そろそろ「教育」の難しさが顕在化してきた.
(特に, アーキテクチャ教育)

[総括-2] — 産学におけるアーキテクチャ研究形態の変遷 —

「学」主導型から「産」主導型へ

- ディスクリート回路構成からマイクロプロセッサを用いた組み合わせシステム構成へ(プロセッサのブラックボックス化).
  (例) 並列アーキテクチャ ……
- 人力から設備 / 道具の時代へ.
- 「産」: PCMアーキテクチャからシステム実装の多様性追求へ(カスタムチップから出来合いチップへ).
- 「学」: 実機による評価(実証)が困難に.

大学における「教育」の積み残し (大学人の生き残る道は?)

- 理論派へのモデルチェンジ(「研究」と「教育」を両立可能)
  ⇒ モデル構築, ペーパマシン + シミュレーション
- software(特に, システムプログラム)研究分野へのちょっかい.

- 「教育」への埋没 (教科書執筆に専念).
  ⇒ 「産」と「学」の意図的切り離し.
- プロジェクトへの積極的参加, 顧問 / 評論家への転身.
  ⇒ 「産学協同」の推進, 「官」の壁.

[提言というより自己主張] — まとめ —

大学における教育 (環境の構築) が必要

- FPGA などのハードウェア (アーキテクチャ) 教育への積極的な利用.
- 教育 (及び研究) のテストベッドとしての「仮想マシンアーキテクチャ」の構築.

ポジションペーパ

並列推論マシン PIM とは何者であったか

瀧　和男 (神戸大学工学部情報知能工学科)

1　はじめに

ICOT を離れてまだ半年ほどにしかならず、最後の PIM WG に向けて何か書こうとすると、書きたいことは留処もなく溢れて来るけれども、全てしたためるわけにもいかず、「PIM とは結局どういうシステムだったか」という問いに対する自分なりの答を手短かに示して私のポジションとしたい。

私が後に示すような考えに至った背景には、他の多くの ICOT 研究員とは異なる次のような役回りがあったためと考えている。一つは、ハード作りから言語処理系を経て応用プログラム作りまで担当したために、システム全体を眺める視点と、応用側からシステムのかなり細部まで見る目を持ち得たことである。もう一つは、ICOT の外に対して並列推論システムを宣伝する役目を度々仰せつかったため、外の人の言葉で言うと PIM はどう説明出来るのか、外の人の目で見ると PIM はどこが優れあるいは他とどこが異なりどこが同じなのか、あるいはどの部分は外の人には注目されないのか、といったことをしばしば考える必要があったためである。

2　私のポジション

私の主張の 7 ～ 8 割は、FGCS'92 の「並列推論マシン PIM」または「Parallel Inference Machine PIM」の報告の中に凝縮したつもりである。以下に概要部分を抜き出して私のポジションとする。読まれて何か感じるところがおありなら、ぜひもう一度、FGCS'92 の会議録に目を通していただければと思う。

「並列推論マシン PIM は、分散メモリ構造を持つ大規模 MIMD 型並列計算機であり、並列論理型言語 KL1 (核言語) を効率良く実行する機構を持つ。KL1 の言語仕様、その実現方式、マシンアーキテクチャはいずれも、知識処理に限らず、より広い問題領域として、動的で均質さの低い大量計算問題を効率良く実行するのに極めて優れた性質を持つ。このような問題領域は、ベクタープロセッサや SIMD 型並列計算機が得意としない分野であるとともに、市販されはじめた数値計算用の大規模 MIMD 型計算機システムも、満足にカバーできていない分野である。すなわち PIM システムは、これまでにほとんど開拓されていない新しい領域の大量計算問題を効率良く実行するための、ソフトウェアとハードウェアの技術を指向しているのである。

本論文では、2 つの重要な事項を述べる。第一は、並列推論マシン PIM と、その上の KL1 言語処理系について、研究開発の概要を報告し、技術内容を要約することである。第二は、開発したシステムが、特に動的で均質さの低い大量計算問題の高効率実行に優れているという観点から、言語、その実現方式、マシンアーキテクチャのそれぞれについて、特徴を列挙し他の並列計算機システムにない優れた点を明らかにすることである。これらにより、PIM システムの全体イメージを鮮明に描写することを試みる。」

3　補足説明

上に示したような並列推論マシンシステムの捉え方を提唱したのは、ICOT の中で当初私一人だったのだけれども、その後、近山氏他幾人かの方にはある程度の部分について賛同をいただけたのではないかと思う。以下では、そのような考えに至った背景としてのいろいろな思いについて触れておきたい。

1. KL1 は汎用の並列言語、あるいは汎用の並列記号処理言語として強力であった。けれども、知識処理の記述言語として特別に強力だったわけではない。それでも、従来あまり取り組まれなかったある広範な問題タイプの並列処理に対して、たいへん強かったように思われる。一体その問題タイプというのは、どのような言葉で言い表せるのだろうか。

2. PIM は、プロセッサとキャッシュの一部に KL1 実行に適した機構を持っているが、それ以外のところは、当たり前の並列処理研究用マシンである。一般的な並列処理研究者に PIM の特徴を主張するとしたら、何を主張するべきなのか。

3. OS よりも下層に言語系が来る KL1 言語の載せ方は特徴的である。大きな WCS や内部メモリは、KL1 言語の載せ方を支援しているといえよう。けれどもその載せ方は、何にとって必然だったのか。

4. ある問題タイプの並列処理に強かったのは、言語仕様のせいだけではなくて、言語実装方式にも因っているのではないか。一般的な言葉で言えばOS核に当たる機能の、かなりの部分を言語実装に取り込んでいる。だからこそ、通信や同期やスケジューリングなどのOS核機能をきわめて安く利用できたのではないか。従来にない並列プログラミングの取り組みが成立した理由がそのへんにもありそうである。

5. 結局のところ、並列推論マシンシステムの優位性はどこにあるのか。KL1言語が高速に走りソフトの開発環境が整っている、というのはあまりに自閉的な説明ではないか。そのことは詰まるところ、よそには出来ないどんなことを可能にすると主張していることに当たるのか。

なかなか全てを言い尽くせてはいないけれども、これらの思いから、並列推論システムの特徴を主張する一つの切口として、前節のような主張をするに至ったのである。この主張に対する諸兄の評価はまちまちかも知れないが、長い目で見たとき、あのときICOTがした仕事のうちで並列処理の世界に最も貢献したものは何だったろうか、とあとで思い起こすならば、前節に提唱したような新しい技術の方向性を世に問い、実績を作った、ということになるのではないかと私は信じている。一つ気がかりと言えば、実はまだこの主張は、十分には世に問うことが実行出来ていないかもしれないということである。

ポジションペーパ

並列プログラミングの展望
―論理型に未来はあるか―

中島 浩（京都大学工学部）

計算機の世界において，10年あるいはそれ以上の未来を予測するのは無謀であるが，ハードウェアに関しては「こんなものかな?」と思ったレベルの一桁上ぐらいが実現されるというのが経験則である。従って，巷間伝えられる100万プロセッサを「こんなものか」と考えれば，実は1000万プロセッサが実現されているという「恐れ」もなしとはできない。

このような超超並列マシンの世界では，計算の様々な側面で「厳密さ」が失われ，あるプログラム（という概念が生き残っていたとして）をn回実行するとn通りの違った結果が得られるなどということが，日常茶飯事になっているものと「期待」している。このような「非厳密性」は論理型の持つ「非決定性」と一脈通じるところがあり（本当?），「硬い」感じのする他の言語に比べて有望なのではないかと夢想したくなる。この他にも言語として「柔らかい」部分が沢山あって，「いいかげんな」プログラミングに向いているような気がする。

一方，言語の実装に目を転じると，この「柔らかさ」が癌であり，「普通はinstantiateされてるんだけど稀に...」とか，「絶対unboundとは言えないんだよね，ほらこんな時は...」などと，実装屋の気力が抜けるような話が極めて多い。それがために，例えば普通は方向が決まっている分散ユニフィケーションを正直に双方向で実装するなど，超超並列に向かない「硬い／堅い」実装になりがちである。

この原因の主なものとして，論理型言語の記述レベルの低さ（誤植に非ず）が挙げられる。極端に言えばGHC/KL1はアセンブリ言語みたいなもので，例えばオブジェクト指向プログラミングができると言っても，アセンブリ言語だってやろうと思えば当然できるのである。「リスト処理のループなのか，プロセス／オブジェクトなのか判らない」とか，「プロセス生成なのかサブルーチン・コールなのか判らない」とか言われると，やはり実装屋は気力が抜けてしまう。

実行効率が悪い言語は結局生き残れないだろうし，GHC/KL1を普通に実装したのでは逆立ちしても効率は上がらないだろうから，言語の記述レベルを上げて，不要な「柔らかさ」を切り捨てることが生き残りの道であろう。筆者の趣味では，残して欲しい「柔らかさ」はプロセス結合の柔軟性が第一であり，ユニフィケーション方向の固定とか，効率重視のための制限はいくらでも受け入れられるのではないかと思っている。

とにかく頑張って，効率が上がるような高レベル言語（そう言えばKL2というのはどうなったのだろう）が設計できれば，21世紀が論理型の時代ということも，かならずしも幻想ではないかもしれない（歯切れが悪い）[1]。

[1]但しハードウェアとは違って，「こんな風になればいいな」と思っても，10年立っても全然そうならなくってFORTRANがはびこってたりするのが，経験則である。

ポジションペーパ

並列言語って何?

西田健次 (電子技術総合研究所)

1 KL1 は並列言語

KL1 (荘園などの機能を使わなければ FGHC と言っても良い) は、並列言語だと言われている。その理由として、

- 各クローズは並列に実行されることを前提としている。
- クローズの実行順序は、(基本的に) 変数の依存性によってのみ制御される。
- クローズ間の同期は言語処理系に組み込まれており、ユーザはそれを意識することなくプログラムできる。

等が挙げられてれ来た。実際、KL1 によって書かれた PIMOS では、同期を書き損なうことによるバグはなかったと言われている。しかし、このことをして、「KL1 は並列言語として優れている」と言ってしまって良いものなのか[2]?

2 同期: 誰がそれを書くの?

KL1 においてユーザが同期を意識する必要がなかったということは、他方で KL1 では同期そのもの (同期に関わるハードウェアの動きなど) を記述することができないということを意味する。では、KL1 では実際に同期を記述したのは誰か (あるいは、どのレベルであるのか)?

Multi-PSI、PIM-m においては、それはファームウェアであり、PIM-p 等においては、VPIM 処理系であったと考えられる[3]。これらは、世間一般のアセンブラ程度、下手をするとそれよりも低いレベルの環境であると考えることができる。KL1 は、その基本コンセプトとしては正しく並列言語としての機能を持っている。しかし、実際に並列処理での重要な機能である同期操作を記述したのは、アセンブラよりも低いレベルの言語 (?) であったという、一種の矛盾を抱えてしまったように思う。

3 同期を書くための道具は並列言語とは言わない?

同期操作を記述するための言語は並列言語とは言わないのか、いや、そもそも、そういう言語が存在すると言えるのか。今までのところ、言語と呼べるもので同期操作を記述できるものはなかったのではないだろうか。その理由としては、同期操作の記述は、実際問題としてマシン依存度が高く、結局アセンブラ (機械語) で書くしかったということが考えられる。そして、アセンブラが一つ一つのプロセッシングエレメントの動作を記述するものであると考えるならば、並列言語としてのアセンブラなどが存在しようはずもない。

4 開発者のための並列言語

並列マシン上に OS なり言語処理系を実装するための枠組は、並列言語によって与えられては来なかったと考えられる。しかし、マシン依存度の低い (汎用性のある) 同期操作までも記述できる、開発者のための並列処理記述言語が必要なのではないだろうか。幾つもの異なる機械語を持ちながら、共通な処理を行なう PIM が揃った今こそ、この種のことを考えてみるチャンスなのではないかと思う。

---

[2] 並列アプリケーションのための並列言語として優れていることに異論はない

[3] KL1b でさえ、同期処理を書くには不足 (高過ぎ?) であった

ポジションペーパ

並列処理の将来 －計算機資源の湯水化に向けて－

吉田 紀彦 (九州大学)

一言でいえば，並列処理マシンをプロセッサの離散的な集合体ではなく，連続的な一種の「液体」として捉えるようなモデルが，将来の「こうなって欲しい」というか「こうなると思う」という姿の一つではないかと考えている．すでに提案されている竹内（ＮＴＴ）のコネクティクスモデルや所（慶大）の計算場モデルに通じるものでもあるが，残念ながらそれらはイメージだけが先行していて具体像はまだ明確ではない．

並列処理マシンが莫大な数のプロセッサから構成されるようになると，もはやプロセッサやプロセスを個々に意識してシステムを構築するのは不可能になり，全体を一つの連続体として意識しなければならなくなる．そのようなハードウェアおよびソフトウェアの基盤となるべきモデルはどのようなものであろうか．

粘菌という生物がいる．これは個々の個体は一種の原生動物であるが，捕食活動などにおいては無数の個体の集団が連続的な組織を構成して，全体が一つの生体であるかのような挙動を示す．しかし，組織がどのように形成されるのか，特にその際に個体間でどのような相互作用がなされるのかは，未だに解明されていない．粘菌のような挙動を模倣するシステム，すなわち analytic ではなく synthetic なアプローチも，特に自律分散システムの分野で研究されているが，今のところは粘菌にも劣る能力のシステムしか実現されていない．

一方，人工知能の分野で，発現的計算（または創発的計算．emergent computation の訳）というテーマが現れてきている．これは，非平衡熱力学における秩序形成やカオスなど，これまで equational に定式化されてきた対象を symbolic に定式化し直して研究しようとするものである．しかし，今はまだ「秩序とは何か」，「計算とは何か」という根源的な問題のところに留まっている．

並列論理型プログラミング言語はプロセスの粒度を非常に細かくでき，上のようなモデルに向いているようにも思える半面，ある意味で「いい加減さ」を内包したモデルな訳であるから適合性に疑問が残るようにも思える．

以上，最近とりとめもなく思っていることを書き連ねてきた．上のようなアプローチなども踏まえて，未来の「マス・コンピュータ」に向けてのモデルを構築できるようだと面白いのだが，正直なところ，先は遠そうである．
(以上)

ポジションペーパ

分散共有メモリマシンと KL1 分散処理系

中島 克人 (三菱電機 情報電子研究所)

1 はじめに

KL1 分散処理系においては、メモリ管理が要である。局所 GC を可能にするための「輸出入表によるノード間ポインタ管理」は KL1 の細粒度負荷分散には重過ぎるのではないかという印象を持つ。白輸出や構造体 ID 等の最適化をやめ、出来るだけ簡単化したとしてもなお、輸出入表エントリの動的確保、参照カウント (WEC) 管理、可変長データの送受など、ハードウェアサポートするには複雑な処理が必要でだからである。

そこで、分散共有メモリマシンにおけるコヒーレントキャッシュをノード間通信に積極的に利用できないだろうかと考える。

2 設計方針

(1) 少なくとも各ノードに HP(ヒープポインタ) が必要。従って、アドレス空間を分割
(2) システム全体の停止を避けるため、ある仕事 (ユーザ) に関わるノード領域だけによる GC
  - そのノード領域外部 (共通 I/O プロセス等) からのポインタは輸出入表による管理
  - その内部は「複数ノードによるコピーイング GC」(今井他、JSPP'91,pp277–284)

3 メモリセル管理上の課題

ゴミセルをどうするかという課題がある。使い捨てでは構造体要素の破壊的更新もできず、キャッシュのヒット率が余りにも悪い。従って reuse (含む、f 構造体要素の破壊的更新) だけを目的とした参照カウントは採用したい (KLIC と同様？)(フリーリストを用いたセル管理はうまく行く気がしない)。

そこで、参照カウントをコヒーレントキャッシュの機能としてサポート出来ないかを検討するべきである。コヒーレントキャッシュは通常、ブロック (ラインとも称する) 単位かつノード間での exclusive/shared を管理しているが、参照カウントの場合は、

- セル単位かつ、同じノードでも参照数の増減が必要
- 参照数 2 の未定義セルへの write により、参照数を 1 に減らす機能が必要
- reuse しようとしたセルが参照数 2 以上の場合に trap 等が必要

となる。更に、分散共有メモリの場合は常に strict consistency を保つと性能が出ないため、

- strict consistency : 未定義変数への書き込み
- weak consistency : 変数セルの参照、ヒープへの書き込み (含む HP 管理)

のように使い分けられる機能が必要となろう。

4 まとめ

まだまだ詳しく検討しないと良く分からない。本方式でも使用セルの分散が大きいので、Cache Only Memory Architecture の方が良いかも知れない。スケーラビリティも良く分からない。

負荷分散は容易だ。@node と 動的負荷分散 (LLS[-G] 方式 (佐藤 (令) 他、SWOPP'92,JSPP'93 など) を併用してはどうか。@node では貰うノードが enqueue 命令を実行。動的負荷分散の場合はゴールレコードへのポインタだけを渡すことにし、コピーは行なわないことにしよう。

以上、今後真面目に取り組む価値のあるテーマである。

ポジションペーパ

PIM/m KL1 処理系の開発と評価 「今思えばこうするべきだった」

近藤 誠一 (三菱電機)

概要

PIM/m の kl1 処理系の開発と評価について述べる。開発の経緯、性能 (絶対速度, 台数効果) に関するコメント、システム性能改良の方法論、実装法に関する改良点の提案、について示す。なお、前回の PIM-WG (1992/10/29) の内容も含める。

1 PSI, PIM/m の開発

1.1 PSI, PIM/m の歴史

三菱電機は ICOT 前期より、推論マシンの共同研究に参画し、PIM/m まで、3 世代にわたって、一貫して水平型マイクロによるインタプリタ方式で推論マシンの開発を行ってきた。

```
  kl0                  kl1

PSI-I  (1983)
  ↓           Multi-PSI/v1 (1985)
PSI-II (1986)       ↓
  ↓           Multi-PSI/v2 (1988)
PSI/UX (1989)       ↓
                  PIM/m    (1991)
```

それぞれの世代では、次のように開発を進めた。

1. 逐次処理系 (kl0) の開発

   逐次型推論マシンとしてオブジェクト指向論理型言語 ESP をサポートし、SIMPOS (Sequential Inference Machine Programming and Operating System) を登載した。

2. 逐次マシン上での擬似並列処理系

   PSI 上に kl1 のマイクロインタプリタを開発し、プロセス単位で、kl0 との入れ替えを行い両者を共存させた。

3. 並列マシン上での並列処理系

   2 次元メッシュ型で共有メモリを持たない分散メモリ構成の並列マシンを開発した。マイクロインタプリタはネットワーク制御部、FEP 通信部をのぞいて、擬似並列処理系と共通である。

1.2 処理系開発 / 性能評価のためのツール

3 世代にわたる開発によりノウハウが蓄積され種々のデバッグ用ツールや評価用ツールが装備されていった。デバッグ用ツールを以下に示す。

- ソフトウェアによるシミュレータ

  PSI 上に PSI/UX, PIM/m のシミュレータを開発した。マルチプロセッサのシミュレーションも可能である。シミュレータ上にて multi-PSI で使用されたテストプログラムを通すことにより、マイクロインタプリタのテストをひととおり行うことができた。

- マイクロトレース

  実行中のマイクロのアドレスを保存する。PSI/UX では、1 K 個、PIM/m では、256 個のアドレスを常に記録することができる。

- マイクロブレーク

  マイクロフィールドの最上位ビットがデバッグ / 評価用であり、そのビットを立てたステップで停止させることができる。

- 命令語レベル条件ブレーク

  機械語命令レベルでの条件ブレークブレークをかけることができる。条件として、命令アドレス、命令コード、メモリタグ / データ、レジスタタグ / データを指定することができる。また、それぞれの命令をパイプラインキャンセルをしながら独立して実行するので、パイプライン関係のデバッグにも有用である。

評価用に次の機構が装備されている。

- GEVC

  マイクロフィールドの最上位ビットがデバッグ / 評価用であり、スイッチの切り替えにより、ビットを立てたステップが実行されると GEVC というレジスタがインクリメントされる。特定の部分の実行状況を実時間で記録することができる。

- processor profiler

  プロセッサレベルの情報を収集することができる。ファームウェアにより実現されている。次の情報のログをとることができる。

  - アイドル時間
  - GEVC
  - エンコードされたメッセージの出現頻度
  - デコードされたメッセージの出現頻度
  - 局所 GC の直前直後の時刻とヒープトップ
  - ユーザ定義イベントログ
    組込述語 log_event(EventID) 実行の時刻と EventID を記録する。
  - メッセージの組み立て分解に要した時間

  これらの情報は、PIMOS のサポートするモニタで視覚的に得ることができる。

- shoen profiler

  荘園内で実行された述語の頻度およびサスペンドの頻度を記録することができる。また、サスペンドした組込述語の情報も得ることができる。

2 逐次処理系との性能比較

まず、PSI/UX との性能比較を行う。以下の点が大きく異なる。

- PIM/m では、MRB による即時ガーベジコレクションを採用しており、回収されたセルのフリーリスト管理などがオーバヘッドとなる。

- 述語呼びだしの際、PSI/UX ではスタックを用いて、環境を保存しているのに対して、PIM/m ではゴールレコードを用いているため、保存データが重複するなどのオーバヘッドがかかる。

表 1 に PSI/UX と PIM/m との実行速度の比較を示す。(1) は KL1 のプログラムを単純に逐次 Prolog に変換したもので、(2) はそれをさらに逐次 Prolog 用に最適化をかけたものである。

append, nrev の性能比は、MRB によるフリーリスト管理に起因しているものと考えられる。逐次 Prolog では 1 reduction の間にメモリ参照回数が 6 回であるのに対して、kl1 では 10 回となっている。また、逐次 prolog にお

表 1: 逐次 Prolog との性能比較

プログラム		逐次 Prolog(msec) (1)	(2)	KL1(msec)	比
append	1000 * 1000	1,497	858	1,641	1.91
	100 * 10000	1,237	854	1,618	1.89
	30 * 100000	3,585	2,582	4,927	1.91
nrev	1000	611		827	1.35
	100 * 100	660	651	875	1.34
	30 * 1000	646	483	876	1.81
qsort	2 ** 11	158	99	207	2.09
	2 ** 13	824	519	1,488	2.87
	2 ** 15	4,398	2,704	8,758	3.24
lisp interpreter	tarai3	210	173	292	1.69
	fib10	20	14	35	2.50
	reverse30	11	9	27	3.00
nqueen	8	97		163	1.68
	10	2,104		3,337	1.59
	12	68,091		101,453	1.49
pentomino 8*5		45,930	37,605	74,442	1.98

ける (1) と (2) の速度差は、測定のためのループの回し方による。(2) は再帰呼びだし、(3) はいったん fail した後、call する。fail により、スタックがたたまれ、キャッシュのヒット率が向上する。kl1 ではフリーリスト管理をしているので、再帰呼びだしの場合でも、キャッシュはヒットする。

qsort, lisp interpreter は Prolog コンテストの課題では、本来行うべきカットが欠落しているため、逐次 Prolog の最適化がうまく働かない。kl1 では必ずガードからボディへの移行するときにカットと同じ機構が働くので、公平のためにその部分にカットを加えた。

これらの結果から、逐次型 Prolog と kl1 の速度比は約 1.5 〜 2.0 倍であることがわかる。

3 Multi-PSI との性能比較

3.1 H/W

Multi-PSI と PIM/m のハードウェアの違いを表 2にまとめた。これらが、性能にどのように影響しているかについて考察する。

- マシンサイクル

  Multi-PSI が 5 MHz (200 ns) に対して、PIM/m は 15.4 MHz (65 ns) と、約 3.08 倍 PIM/m の方が速い。

- パイプライン

  Multi-PSI では命令フェッチ程度であったものが、PIM/m では 5 段という深いパイプラインとなっている。また、デレファレンス機能がパイプラインの上流で行われる。半面、レジスタがロックされた場合および、条件分岐により、パイプラインをキャンセルする場合はオーバヘッドとなる。

- アドレス変換方式

  Multi-PSI では、固定テーブルを用いてハードウェアによって行われるが、PIM/m ではマイクロによるテーブルの入れ替え方式をとっている。したがって、いったんテーブルがはずれると大きなオーバヘッドとなる。

- キャッシュメモリ

  Multi-PSI では、データ / 命令共通で 4 K 語であったが、PIM/m では命令 1K 語、データ 4K 語 と分離された。

表 2: Multi-PSI と PIM/m の H/W 比較

	Multi-PSI	PIM/m
制御方式	53 ビット水平型マイクロ	64 ビット水平型マイクロ
制御記憶容量	16 K 語	32 K 語
使用 LSI	CMOS ゲートアレイ 8K/20K ゲート 9 種	CMOS セルベース 0.8/1.0 μ m
マシンサイクル	5 MHz (200ns)	15.4 MHz (65 ns)
パイプライン	命令フェッチのみ	5 段パイプライン
アドレス変換方式	固定テーブル	マイクロによるテーブル入れ替え
キャッシュ	4K 語	命令 1K 語 + データ 4K 語
主記憶アクセス	800 ns	1430 ns
PE 間ネットワーク 転送速度 バッファサイズ	2 次元メッシュ (max 8*8) 同期 10 ビット / チャネル 5 Mbps 4K バイト	2 次元メッシュ (max 16*16) 非同期 10 ビット / チャネル 3.84 Mbps 1K バイト
メモリ	80M バイト (16M 語)	80M バイト (16M 語)
FEP との通信	PE 間ネットワーク	SCSI
内蔵ディスク	なし	8 PE ごとに 670 M バイト
Append 性能	128 KRPS	615 KRPS

```
dummy(N,X,Y,End) :- N > 0, N1 := N - 1,
    _ := X + Y |
    dummy(N1,X,Y,End).
dummy(0,_,_,End) :- true | End = end.
```

(a)

```
add(N,X,Y,End) :- N > 0, N1 := N - 1,
    _ := X + Y, _ := X + Y,
    _ := X + Y, _ := X + Y |
    add(N1,X,Y,End).
add(0,_,_,End) :- true | End = end.
```

(b)

図 1: ガード組込述語評価用 KL1 プログラム

- 主記憶アクセス

  Multi-PSI ではキャッシュがはずれて主記憶アクセスとなる場合は 4 クロック分、800 ns であったが、PIM/m では 22 クロック分、1430 ns と遅くなった。

- ネットワーク

  Multi-PSI では、転送速度 5 Mbps だったものが PIM/m では、3.84 Mbps、バッファサイズも 4 K バイトだったものが、1 K バイトといったように、仕様が縮小された。これは、Multi-PSI の評価により十分であると判断されたことによる。

3.2 組込述語

KL1 言語で用意されている算術演算、構造体操作などの組込述語を Multi-PSI と PIM/m において処理速度の性能評価を行なった。評価の方法としては、まず各組込述語の処理速度の計測のために計測用のプログラムを用意し、同じプログラムを Multi-PSI と PIM/m で実行し、処理速度を計測し比較するという方法を取った。Multi-PSI と PIM/m は機械語レベルで互換性があり、同じコードで測定を行うことができる。計測に用いたプログラムでは、計測対象の組込述語の実行のために前後で実行される命令列の処理時間を前もって計測しておき、その命令列に計測対象の命令を追加するという方法で計測を行なった、図 1にガード、図 2にボディーの具体例を示す。

上記の方法を取って行なった評価結果を表 3に示した。

これらの表から PIM/m は Multi-PSI と比較して以下のことがわかる。

- 整数加減算はクロック比 3.08 よりも速くなっている。これは、パイプラインの上流で、デレファレンスおよびタイプチェックができるためである。

```
dummy(N,X,Y,End) :- N > 0, N1 := N - 1 |
    _ := X + Y, dummy(N1,X,Y,End).
dummy(0,_,_,End) :- true | End = end.
```

(a)

```
b_add(N,X,Y,End) :- N > 0, N1 := N - 1 |
    _ := X + Y, _ := X + Y,
    _ := X + Y, _ := X + Y,
    b_add(N1,X,Y,End).
b_add(0,_,_,End) :- true | End = end.
```

(b)

図 2: ボディ組込述語評価用 KL1 プログラム

表 3: 組込述語

組込述語	Multi-PSI	PIM/m	比
b_add	6,009	1,949	3.08
b_multiply(2*1)	36,058	14,293	2.52
b_multiply(32767*65535)	50,079	29,886	1.68
b_divide(2/1)	42,067	23,388	1.80
b_divide(2147385345/65535)	94,149	93,555	1.01
b_single_float_add	48,077	45,478	1.06
b_single_float_multiply	79,928	94,724	0.84
b_single_float_divide	111,979	133,706	0.84
g_vector_element(6 要素 MRB-off)	37,258	12,721	2.93
b_vector_element(6 要素 MRB-off)	62,699	19,101	3.28
set_vector_element(6 要素 MRB-off)	67,105	19,491	3.44
g_string_element(16bit MRB-off)	78,726	11,486	6.85
b_string_element(16bit MRB-off)	86,739	17,800	4.87
set_string_element(16bit MRB-off)	100,761	26,246	3.84

- 整数乗除算は、Multi-PSI では、ビット操作用の ALU が CPU に接続されていたが、PIM/m ではすべてファームウェアで実現している。そのため、たとえば、複雑な除算では同程度の性能しか出ない。
- 浮動小数点数演算も整数乗除算と同様に、ALU の影響が出ている。
- 構造体操作系組込述語では全体にクロック比 3.08 よりもよい値となっている。これらはパイプラインの影響も多少あるが、主に、コーディングテクニックによるところが大きい。Multi-PSI についてはさらに最適化の余地が残されている。ただし、制御記憶容量が PIM/m では Multi-PSI の 2 倍となっており、PIM/m ではデータタイプによって専用の手続きを記述することができることが多少影響していると思われる。

3.3 基本制御機能

KL1 の特徴的な動きとして、述語の書き換え (述語呼びだし) がある。次の 3 つの基本制御機能について性能評価を行った。

1. 1 クローズ 1 呼びだし

   1 クローズのみを呼び出す 100 クローズをならべ、それを 10,000 回繰り返す。

```
p0(N) :- true | p1(N).
p1(N) :- true | p2(N).
p2(N) :- ....
    ....
p99(N) :- true | true.
```

   Multi-PSI, PIM/m のファームウェアでは、最初の呼びだしは、通常ゴールレコードを作ってゴールスタックに積むという手順を最適化して、直接実行するため高速に呼び出すことができる。

表 4: 基本制御機能性能

プログラム	Multi-PSI		PIM/m		比
	msec	Krps	msec	Krps	
1 呼びだし	874	1,144	259	3,861	3.37
100 呼びだし (中断なし)	12,731	78	3,948	253	3.22
100 呼びだし (中断付き)	31,631	32	10,063	99	3.14

2. 1 クローズ 100 呼びだし

1 クローズに 100 個の述語呼びだしをならべ、それを 10,000 回繰り返す。呼び出した述語はユニフィケーションを用いて、順に引数を送っていくもので、中断はしない。

```
enq :- true | enq(0,X1),
       enq(X1,X2), ....,
       enq(X99,_).
enq(X,Y) :- true |
            Y = X.
```

Multi-PSI, PIM/m のファームウェアでは、まず 100 個のゴールレコードを作り、それをすべて、ゴールスタックに積み、上から順に実行していく。中断はしない。

3. 1 クローズ 100 呼びだし (中断付き)

1 クローズに 100 個の述語呼びだしをならべ、それを 10,000 回繰り返す。値を wait するので、すべてのゴールはいったん中断する。

```
susp :- true |
     susp(X99,_), ...,
     susp(X1,X2),
     susp(0,X1).
susp(X,Y) :- wait(X) |
                Y = X.
```

中断なしと異なる点は、ゴールスタックから取り出して、実行したとき、値が確定していないので、中断する。すなわち、すべてのゴールは、" ゴールレコードを作る → ゴールスタックに積む → ゴールスタックから取り出す → 中断する → 再開してゴールスタックに積む →ゴールスタックから取り出して実行する " という操作を行う。

表 4 にそれぞれの Multi-PSI と PIM/m の比較データを示す。同じ方式をとっているのでクロックの差 (3.08 倍) に近い数字となっている。若干 PIM/m 高速なのは、パイプラインでの処理が影響している。

次に命令キャッシュの影響について考察する。上記の 1 クローズ 1 呼びだしをさらに拡張し、100, 200, ..., 1000 クローズまで可変とし、それぞれのリダクション速度を測定した。表 5 に Multi-PSI と PIM/m を比較したデータを示した。

命令コードは 1 クローズあたり 5 語で、Multi-PSI のキャッシュが 5 K 語、PIM/m の命令キャッシュが 1 K 語から計算すると、クローズ数が Multi-PSI で 800 (4000/5) あたりに、PIM/m では、200 (1000/5) あたりに速度の境界があることになり、実測値と一致する。また、キャッシュがミスしたときのオーバヘッドが multi-PSI が 1.7 倍程度であるが、PIM/m では 6.4 倍と非常に高いこともこのデータから読みとることができる。

3.4 単体性能

表 6 に PIM 共通のベンチマークプログラムの測定結果を示す。ここでは、multi-PSI と比較した PIM/m の特徴について示す。

表 5: 命令キャッシュの影響

クローズ数	Multi-PSI		PIM/m		比
	msec	Krps	msec	Krps	
100	874	1,144	259	3,861	3.37
200	1,752	1,142	612	3,268	2.86
300	2,754	1,089	3,685	814	0.75
400	3,531	1,133	6,680	599	0.53
500	4,424	1,130	8,317	601	0.53
600	5,447	1,102	10,016	599	0.54
700	6,263	1,118	11,692	599	0.54
800	7,341	1,090	13,342	600	0.55
900	11,018	817	14,990	600	0.74
1000	14,807	675	16,638	601	0.89

Multi-PSI および PIM/m では、MRB による単一参照セルの回収とフリーリストの管理による実時間 GC を採用している。PIM/m では、1 ～ 8, 16, 32, 64, 128, 256 の 13 種類のセルが用意されており、フリーリストが空になったとき、および、GC 後に 2 ～ 16 個確保する。表 6の上下はこれを積極的に利用しているものとそうでないものとをわけたものである。たとえば、bestpath100*100 実行の前後では約 983 K 語のヒープが使用され、実行後のフリーリストには、そのその約 67 % にあたる 656 K 語のセルが回収されて残る。一方 pentomino では、3,300 K 語のヒープが使用され、わずか、5.5 K 語のセルのみが残る。実行前には、4.7 K 語がつながれており、回収がごくわずかであることがわかる。また、ゴールレコードとなる 16 語のセルに注目すると、bestpath100*100 では、11 K 個がフリーリストにつながれて残るが、pentomino ではわずか、23 個残るのみである。

表から、両者グループでは処理時間比がきれいに二分されることがわかる。

表 6では、GC 直後で、フリーリストがほとんど空の状態から開始したときの実行速度である。表 7 に、連続 10 回 bestpath を起動したときの実行速度を示す。ヒープ領域の限界を設定して、GC によってワークエリアを小さくした場合を付した。(0) は設定なし (約 6 M 語)、(1) は 1 M 語、(2) は 2 M 語である。(3) はプログラムの開始時に故意に GC を発生させたときの GC 込みの時間である。上段は bestpath 100 * 100, 下段は 200 * 200 のデータである。PIM/m では、メモリアクセスに関しては、Multi-PSI と比較して、クロックと主記憶アクセス速度との比が、約 5 倍ほど悪く、さらに、ファームウェアによるアドレス変換テーブルの入れ替えという手間がある。このデータより、MRB によるセルの回収は、メモリの局所性を悪化させる逆効果があることがわかり、メモリアクセスが悪い PIM/m では、その差が顕著となる。また、kl1 の特徴であるプロセスネットワークをつくって suspend/resume を繰り返すプログラミングは並列性をあげる半面、メモリの局所性を低下させるという諸刃の剣であり、PIM/m の処理系はそれに対して十分対応できているとはいえない。

MRB 無についてもいくつか PIM/m の弱点があらわれている。

- 15 パズルでは、multi-PSI 比 が 2.40 で、RPS も 89 K と低い。解析の結果、組込述語のサスペンドによるオーバヘッドであることがわかった。PIM/m の場合、

```
p(X,L) :- true | q(L,Y),
        Z := X + Y, r(Z).
```

というコードで、Y が q/2 の出力の場合、q/2, r/1 のエンキューの前に 加算が実行され、サスペンドしてしまう。サスペンドした組込述語では例外処理が発生したときその位置を報告するための情報を得るために大きなオーバヘッドがかかる。PIM/m, multi-PSI では全く同様の方式を用いているがこのサスペンド処理のオーバヘッドのかかり方が異なるものと思われる。

```
p(X,L) :- true | q(L,Y),
  call_add(X,Y,Z), r(Z).

call_add(X,Y,Z) :- true |
              Z := X + Y.
```

表 6: 単体性能

プログラム	Multi-PSI		PIM/m		比
	msec	Krps	msec	Krps	
life game	18,583	19	8,020	45	2.32
qlay11	35,459	49	15,057	116	2.35
semi	19,579	37	8,192	89	2.39
bestpath100	45,530	62	19,263	129	2.36
tsumego5	6,477	39	2,479	101	2.61
mastermind	23,683	73	7,837	222	3.02
nqueen12	280,690	92	92,483	278	3.04
pascal	5,373	61	2,051	161	2.62
puz15-6	216,075	37	89,173	89	2.42
puz15-6(no susp)	161,368	49	57,994	137	2.78
puzzle	32,702	39	9,901	128	3.30
waltz	15,540	77	4,734	255	3.28
zebra	16,482	25	5,450	74	3.02
pentomino1(8*5)	214,229	40	89,366	95	2.40
pentomino2(8*5)	181,108	45	69,792	121	2.59
pentomino3(8*5)	164,026	62	54,883	135	2.99

のようにサスペンドしないようにプログラムを書き換えると表の puz15-6(no susp) のように大幅に性能が向上する。

- pentomino (詰め込みパズル) では、その盤面をコピーしながら、枝別れしていく。 pentomino1, pentomino2 ではそのテーブルをベクタでもち、8*5 の盤面の情報を 1 語を 4 ビットづつ区切って、5 語を使用している。pentomino1 ではその情報の参照 / 変更のため乗除算を用い、pentomino2 では、ビット演算 (シフトなど) を用いている。PIM/m は、組込述語の節で述べたように乗除算が遅く、multi-PSI 比が悪くなっている。

- さらに、pentomino2 ではガード組込述語にて、ビット演算によって、参照 / 変更を行っている。しかし、PIM/m では、5 段という深いパイプラインを採用しているため、連続するガードでのタイプチェック / ビット演算によって、インターロックがかかり、multi-PSI と比べてクロック比ほど性能が出ない。pentomino3 では、8 ビットストリングを用いて、処理系が位置決めを行っている。テーブルの大きさは倍になるが、インターロックが解消され、速度が向上する。multi-PSI で PIM/m ほど速くなっていないのは単にストリング系の組込述語が遅いためである。

3.5 複数プロセッサ性能

Multi-PSI の性能評価からネットワークの稼働率はそれほど高くないという結果を得た。そこで、 PIM/m では Multi-PSI に比して、次のように仕様を落とした。ここではこれらの影響を中心に複数プロセッサの性能について示す。

- ネットワークと CPU のバッファとして、読み込み用と書き込み用がある。 Multi-PSI では、両者とも 4 K バイトであったが、PIM/m では 1 K バイトと小さくなった。

- ネットワーク転送時間が、 Multi-PSI では 5 Mbps であったものが PIM/m では 3.84 Mbps となった。クロック比ではさらに大きなものとなっている。

逆に、PIM/m では、1 語、半語単位でのバッファの読み書き用の命令を加え、処理の高速化をはかっている。

表 8 に 1 PE と 16 PE の Multi-PSI, PIM/m の性能比を示した。この表から以下のような性質を読みとることができる。

- puz15 や pentomino といったネットワークアクセスの低いプログラムに関しては、単体性能の影響のみで 16 PE あたりではプロセッサ数による差異はほとんどない。

表 7: bestpath の連続実行

Multi-PSI msec		(0) msec		(1) msec		(2) msec		(3) msec	
46,559	1.00	19,711	1.00	19,288	1.00	18,940	1.00	19,860	1.00
47,030		25,858		21,247		25,132		21,028	1.06
47,186	1.01	28,161		21,150		27,454		21,008	
47,158		28,805		25,760		27,954		21,027	
47,129		29,188		25,435		21,211		21,008	
47,016		29,608		27,648		25,464		21,028	
47,178		29,537		25,442		27,811		21,009	
47,062		29,807		28,927	1.50	28,304	1.49	21,028	
47,181		29,940	1.52	22,265		20,614		21,008	
47,173		29,755		26,955		24,642		21,027	
47,067	(ave)	28,037	(ave)	24,411	(ave)	24,752	(ave)	20,903	(ave)
188,296	1.00	81,717	1.00	103,272		92,268		81,203	1.00
198,852		111,599	1.37	107,717	1.07	93,520		83,042	1.02
189,772		99,484		106,160		97,006		83,278	
200,070		109,876		105,514		89,360		83,042	
196,203		106,441		104,612		96,372		83,279	
203,892	1.08	85,745		104,394		88,827		83,041	
193,159		110,486		107,079		97,349		83,279	
199,368		100,574		100,844	1.00	87,811	1.00	83,041	
194,261		109,947		107,491		97,494	1.11	83,279	
193,364		106,999		105,645		90,831		83,041	
195,723	(ave)	102,286	(ave)	105,272	(ave)	93,083	(ave)	82,952	(ave)

- 1 PE より 16 PE のほうが遅いプログラム (master mind, puzzle, waltz, zebra) はプロセッサ間の同期をとりながら実行されるので、ネットワークに負荷がかかるものと考えられる。これらについては、1 PE より、16 PE の方が、PIM/m, Multi-PSI 比が大きくネットワーク性能の影響が出ている。

- bestpath100 は単体性能の項で述べたようにワーキングセットが大きく、メモリ性能が大きく影響するものと思われる。16 PE に仕事を分散すると、1 プロセッサあたりのワーキングセットが小さくなるので、1 PE よりも 16 PE のほうが、性能比がよくなっている。

表 9に pentomino をさらに多くのプロセッサで実行した結果を示す。16 PE あたりではさほど差は見られなかったが、64 PE では 2.85 倍と若干ネットワークの影響が見られる。

さらに、実用的な応用プログラムでの評価結果を示す。表 10 は並列自動証明システム MGTP の実行時間の測定結果である。MGTP は PIM/m の応用プログラムの中でも最も利用率が高いもので、pentomino よりもネットワーク稼働率が高い。'問題' の項はは証明しようとしている問題で、実際には数分で解けるものから、256 プロセッサで数時間かけても解けないようなものまである。ここでは、64 PE で数分程度といった比較的小さな問題で評価を行った。

ここでも、pentomino と同様に 64 PE あたりでネットワークの影響がでている。さらに、考察を加えるために、表 11に 64 PE, 256 PE での処理の内訳を示す。ガーベジコレクションやメッセージ処理である encode/decode は実行時間に含まれる。リダクション数や、表 12 のメッセージの種類とその数から、Multi-PSI と PIM/m ではほぼ同じ動きをしていると類推されるにもかかわらず、PIM/m では、idle 時間や encode/decode の割合が高いことがわかる。その理由として、不要になったと判断された PE 間にまたがる構造体のリリースの集中とメモリアクセス速度の相対的な低下による GC 時間の割合の増加があげられる。PIM/m の処理系は GC 中に不要になったと判断された他の PE の構造体を開放するためにメッセージが送られる。このメッセージは GC 時に集中し、ネットワークが混雑する原因となる。このため、GC 時間が長くなり、他の PE の処理との同期が遅れ、次々と suspend していき、稼働率が低下する。

このような現象をふせぐため、MGTP のユーザは、ある PE で メモリの使用率が 99 % を越えた時、すべての PE に対して、一斉に GC を起動するという方式をとっている。そのときのふるまいを 表 12 (2) に示した。この結

表 8: 複数プロセッサ性能

プログラム	1 PE			16 PE		
	Multi-PSI	PIM/m	比	Multi-PSI	PIM/m	比
tsumego5	6,534	2,615	2.50	2,502	1,083	2.31
pascal	5,459	2,177	2.51	843	377	2.24
puz15-6	216,075	89,721	2.41	15,405	6,328	2.43
puz15-6-new	161,368	57,994	2.78	11,651	4,225	2.76
pentomino1	223,703	91,777	2.44	14,431	6,003	2.40
pentomino2	187,297	70,330	2.66	12,293	4,671	2.63
pentomino3	172,702	58,710	2.94	11,124	3,810	2.92
master mind	24,287	7,974	3.05	56,206	32,270	1.74
puzzle	32,702	10,495	3.12	57,129	23,111	2.47
waltz	15,658	4,795	3.27	20,202	7,302	2.77
zebra	16,482	5,624	2.93	53,437	21,805	2.45
life	18,583	8,431	2.20	4,522	1,977	2.29
bestpath100	10,659	5,735	1.86	1,391	675	2.06

表 9: 複数プロセッサ性能 (Pentomino 8*5)

プロセッサ数	Multi-PSI		PIM/m		比
	msec	台数効果	msec	台数効果	
1 PE	172,702	1.00	58,710	1.00	2.94
2 PE	86,563	2.00	28,779	2.04	3.01
4 PE	42,958	4.02	14,492	4.05	2.96
8 PE	21,712	7.95	7,423	7.91	2.92
16 PE	11,124	15.53	3,810	15.41	2.92
32 PE	5,968	28.94	2,085	28.16	2.86
64 PE	3,379	51.11	1,185	49.54	2.85
128 PE			804	73.02	
256 PE			788	74.51	

果、CPU の稼働率は 97 % を越え、実行時間も約 10 % 向上する。半面、GC が集中するため、構造体リリースのためのメッセージが集中し、ネットワークの待ちが多くなる。

表 12K MGTP を実行した際のメッセージの種類を示した。メッセージ数が 0 である荘園関係のメッセージなど (start, stop, abort, ask_statistics, answer_statics, shoen_profile, shoen_profile_request) は省略した。この表から以下のことがわかる。

- 資源関係のメッセージが (request_resource, supply_resources) の数が PIM/m の方が少ない。これは、単体性能、ネットワーク性能を考慮して、PIM/m では一度に供給する資源の量を Multi/PSI より多めに設定しているからである。この値はシステム立ち上げときにパラメータとして与えることができる。
- メッセージが大きいときはバッファに入りきらないので、マルチパケットとして分割して送信する。マルチパケットは、いったん、バッファからメモリ上にコピーするためその処理時間が大きくなる。バッファの大きさが Multi-PSI 4 K バイトに対して、PIM/m 1 K バイトのため、メッセージの大きさの見積もりが、Multi-PSI では 979 バイト (バッファの 1/4 弱)、PIM/m では 466 バイト (バッファの 1/2 弱) よりも大きい場合にマルチパケットとなる。マルチパケットになったメッセージの総数はマルチパケットの最後を示す multi-packet_last の総数に一致する。表では殆んど差異はなく、ここでは、バッファを縮小した影響は出ていない。

4 感想

ここでは、個人的な感想と心残りについて触れたい。

表 10: 複数プロセッサ性能 (MGTP)

問題		Multi-PSI		PIM/m		比
		msec	Krps	msec	Krps	
pro20Om2	16 PE	1,477,791	551	691,007	1178	2.14
	32 PE	776,256	1076	373,667	2234	2.08
	64 PE	433,167	2007	227,692	3830	1.90
	256 PE			127,622	8448	
pro60Om2	16 PE	918,008	605	438,122	1274	2.10
	32 PE	481,721	1176	224,991	2532	2.14
	64 PE	267,106	2191	138,525	4148	1.93
	256 PE			92,546	6900	
pro98Om2	16 PE	1,043,910	641	465,758	1434	2.24
	32 PE	549,054	1266	239,643	2798	2.29
	64 PE	293,808	2383	150,720	4638	1.95
	256 PE			95,047	8335	

表 11: 実行時間の内訳

処理	Multi PSI	PIM/m		
	64 PE	64 PE (1)	64 PE (2)	256 PE
実行時間	93.8 %	88.0 %	97.6 %	47.9 %
GC	0.7 %	3.2 %	5.0 %	0.0 %
Encode	3.0 %	4.8 %	6.5 %	3.8 %
Decode	5.5 %	8.3 %	8.7 %	11.0 %
Idle	6.2 %	12.0 %	2.4 %	52.1 %
パケット総数	17,534,676	17,229,922	17,147,908	60,424,122
リダクション数	700,015,407	699,008,990	712,840,024	792,217,641

- PIM/m チップは kl0 チップ?
  純粋に kl1 マシンを作っていたら、ちがったものができていたかもしれない。ただし、 PIM/m は Pseudo PIM/m を提供するという役割もあったので、一概には言えない。事実、 Pseudo PIM/m に多くのユーザがつき、バグの早期発見による安定化、性能向上に果たした役割は大きい。

- kl1 では、メモリの割付けを意識しないでプログラミングができるというメリットがあるが、実際に、プログラムのチューニングを行うときには無視することができない。にもかかわらず、それを制御する方法としては GC しかなく、ユーザは不必要な試行錯誤によって性能向上をめざしている。

- ネットワークの評価はできていない。
  ICOT にある kl1 program は multi-PSI, PIM/m 用にチューニングされたものばかりで、それなりの使い方しかしていない。それをもって判断するのは危険。今のところ、プログラマが承知して十分にチューニングしたおかげで、ネットワークが負担にならない範囲内に納めることができたということくらいしか言えない。

- kl0(逐次 Prolog) の特徴はユニフィケーションとバックトラック、 kl1 の特徴はプロセス間 (ゴール間) の通信 (suspend/resume) による自然な AND 並列の記述といわれているが、実際はこれらの処理は遅いので、極力使用しない方が速いプログラムを書くことができるという矛盾をかかえている。ユニフィケーションは代入のみ、バックトラックを起こさないように、手続き的にプログラム書く。 MRB を黒くしてどんどんコピーをつくりながら枝別れしていく OR 並列で書き、プロセス間の通信はしないようにする。処理系の都合で本来の姿とは違ったプログラミングスタイルを押し付けてはいないか?

表 12: MGTP 実行時のメッセージの種類

メッセージ	Multi-PSI	PIM/m	
	64 PE	64 PE	256 PE
throw_goal	466	462	1,046
ready	266	273	682
terminated	837	715	1,158
read	5,528,806	5,550,009	22,065,384
answer_value	5,387,881	5,404,115	18,994,792
release	5,590,619	5,803,751	18,823,868
unify	259,554	261,233	270,490
exception	209	206	212
request_wtc	0	0	0
supply_wtc	0	0	0
return_wtc	1,471	358	425
request_resource	378,384	96,862	115,097
supply_resource	378,568	97,122	115,779
return_resource	18	18	72
syncronized_GC	0	0	0
multi-packet_continued	6,903	14,129	33,428
multi-packet_last	694	669	1,689
TOTAL	17,534,676	17,229,922	60,424,122

ポジションペーパ

ハードとソフトの役割分担
～ ハードとソフトにはさまれて複雑化していった処理系 ～

近藤 誠一 (三菱電機)

PSI, Multi-PSI, PIM/m 系では、伝統的にファームウェアによるインタプリタ方式をとってきた。特に、PSI-II → PSI/UX, Multi-PSI → PIM/m では、オブジェクト互換である。Multi-PSI は、53 ビット水平型マイクロで、制御記憶容量は 16 K 語である。PIM/m は、64 ビット水平型マイクロで、制御記憶容量は 32 K 語である。PIM/m では、さらに、パイプライン上流の制御のため、54 ビット * 512 語が用意されている。

以下に PIM/m, Multi-PSI, PSI/UX のマイクロ総量を示した。

multi-PSI	kl1	15,672	(3D38)	95.7 %
PIM/m	kl1	23,754	(5AD6)	71.0 %
PSI/UX	kl0	25,859	(6503)	78.9 %

PIM/m のうち、11.2 % は、SCSI のルーチンである。PIM/m のファームウェアの execute 命令の一部を図 3に示す。

```
:: /'*************************************************************************'.
:: /'*  +------+-----------+-----------+-----------+-----------+     *'.
:: /'*  | KBLT ! Opcode 361 |   arity   |           label       |     *'.
:: /'*  +------+-----------+-----------+-----------+-----------+     *'.
:: /'*************************************************************************'.
:: /'$ KLB_execute:                                                    $'.
:: /'$  ADR                                                            $'.
:: /'$    1_word , jump , mar := PC + ir/s16 ;                         $'.
:: /'$  DRD                                                            $'.
:: /'$    mdr := mar ;                                                 $'.
:: /'$  DPT                                                            $'.
:: /'$    (mar2 , mdr2) := mdr , mdr1 := CRR ; .                       $'.
:: /'*************************************************************************'.
KLB_execute:
        CRR := mdr1 - 2 , yr <- 3 , cond(Slit_Check_exsw04) .
                %% CRR = CRR_mdr1 - 2
        := mdr1 - yr , cond(sign) , xr <- PSTR , reset_SUSPF_sw4 ,
            if (cond) call Sub8_kl1_spy_check_in_exec .
                %% check CRR =< 2 , Suspension Stack Flag OFF
                %% return : saved mdr2 , := CRR - 1 , cond(sign)
        PC := COD! mdr2 , xr <- PC , cond(interrupt_all) ,
            if (cond) goto $enqueue_and_proceed .
                %% PC = mdr2 (PC + IR/S16)
        if not (cond) eop , r04 := xr , reset_CGCF_sw3 .
                %% Current Goal Content Flag OFF
                %% i_fetch and eop are suppressed if ZERO
```

図 3: マイクロソース

PIM/m 系は 3 世代の間にファームウェアの開発 / デバッグ環境が向上し、多くのノウハウが蓄積されてきた。そして、kl0/kl1 で書き切れない部分や、性能が出ない部分に関しては、高速化の名のもとに、ファームウェア化されていき、膨大かつ複雑化していった。その原因として、

- 機械語のレベルが高い。(というよりも高機能組込述語がどんどんできていってしまった。)
- 選択肢は kl1/kl0 とファームウェアしかない。(論理型言語のみに固執してしまった？)

- 多少複雑な処理でも充実したファームウェアのスタッフによりこなしてきてしまった。

などがあげられる。処理系に負担をかけるという伝統は、VPIM に対して悪い影響を与えてしまったかもしれない。

PIM/m に関しては、オブジェクト互換のため共有部分が多く新規開発が少ないので、確実性を重視して、Multi-PSI をそのままシフトしてしまった。処理系で負担を吸収する方式は開発途上のシステムに対しては、非常に強力で、Multi-PSI の開発では一応の成功を見たが幅を広げるという面では、発展性が悪い (ハードの新規開発のコストは膨大)、保守性が悪いという点で、将来性はあまりない。

また、機械語レベルが高いのでコンパイラ開発者の楽しみがない。kl0 とちがって、ガードとボディが明確に区別されているので、最適なコンパイルコードに対する工夫の余地がほとんどなく、そのようなコードを出すのはさほど難しくない。

ポジションペーパ

KL1 分散処理系の設計 / 実装 / 移植の経験から

六沢 一昭 (沖電気工業 (株) 情報通信システム開発センタ)

Multi-PSI 上の KL1 分散処理系の処理方式設計に初期の頃から参画し、そして VPIM/KL1 処理系の実装に加わり、さらに PIM/i への VPIM/KL1 処理系の移植にもかかわりました。これらの経験からいくつかコメントします。

1. (PIM/m を除く) PIM/x への KL1 処理系の実装は、「基本的に ICOT が処理系 (VPIM 処理系) の設計を行ない各メーカは処理系の移植を行なう」という方式であった。

   この方式は、やはり実際的であったと思う。KL1 の言語仕様だけが与えられて各社ばらばらに処理系を設計 / 実装したのでは「すべての PIM/x で KL1 処理系が動く」ということは決してなかったに違いない。

   一方この方式には「メーカのソフト担当者のオリジナリティを発揮する部分が少なくなり研究的な作業がわずかしかない」という面もあった。「メーカのソフト担当者のやる気をそいでしまった」というマイナス面はやはり無視できないだろう。

2. 計算機ハードウェアを新しく作る場合、最新のハードウェア設計 / 実装技術が使えないと、たとえ実装する言語が新しいものであっても、高性能なシステムを作るのは困難である。「新しい言語であるのでその言語処理系の実装方式は我々が一番よく知っている」ということだけに頼るのには限界がある。

3. 共有メモリ型並列計算機上に、共有メモリであることを活かして並列言語処理系を実装するのは、はっきり言ってしんどい。

   「今書いたデータが次の瞬間に他のプロセッサによって書き換えられてしまう」可能性をたえず考えなければならないため、処理系の設計時にとても神経を使う。また当然バグも入りやすい。しかもこの手のバグは「検出されにくく除去しにくい」という悪質な性質を持っている。

(以上)

ポジションペーパ

FGCS Project: Personal Perspectives

上田 和紀 (NEC C&C 研)

CACM Vol.36 No.3 pp.65–76 (1993) の記事より抜粋.

An outstanding feature of the Fifth Generation Computer Systems (FGCS) project is its middle-out approach. Logic programming was chosen as the central notion with which to link highly parallel hardware and application software, and three versions of so-called *kernel language* were planned, all of which were assumed to be based on logic programming. The three versions corresponded to the three-stage structure of the project: initial, intermediate, and final stages.

The first kernel language, KL0, was based on Prolog and designed in 1982 as the machine language of the Sequential Inference Machine. Initial study of the second kernel language, KL1, for the Parallel Inference Machine started in 1982 as well. The main purpose of KL1 was to support parallel computation. The third kernel language, KL2, was planned to address high-level knowledge information processing. Although the Institute for New Generation Computer Technology (ICOT) conducted research on languages for knowledge information processing throughout the project and finally proposed the language Quixote [11], it was not called a "kernel" language (which meant a language in which to write everything). This article will focus on the design and the evolution of KL1, with which I have been involved since 1983.

What are the implications of the middle-out approach to language design? In a bottom-up or top-down approach, language design could be justified by external criteria, such as amenability to efficient implementation on parallel hardware and expressive power for knowledge information processing. In the middle-out approach, however, language design must have strong justifications in its own right.

The design of KL0 could be based on Prolog, which was quite stable when the FGCS project started. In contrast, design of KL1 had to start with finding a counterpart of Prolog, namely a stable parallel programming language based on logic programming. Such a language was supposed to provide a common platform for people working on parallel computer architecture, parallel programming and applications, foundations, and so on.

It is well known that ICOT chose logic programming as its central principle, but it is less well known that the shift to concurrent logic programming started very early in the research and development of KL1. Many discussions were made during the shift, and many criticisms and frustrations arose even inside ICOT. In these struggles, I proposed Guarded Horn Clauses (GHC) as the basis of KL1 in December 1984. GHC was recognized as

a stable platform with a number of justifications, and the basic design of KL1 started to converge. Thus it should be meaningful to describe how the research and development of KL1 was conducted and what happened inside ICOT before KL1 became stable in 1987. This article also presents my own principles behind the language design and perspectives on the future of GHC and KL1.

Joining the FGCS Project

When ICOT started in 1982, I was a graduate student at the University of Tokyo. My general interest at that time was in programming languages and text processing, and I was spending most of my time on the thorough examination of the Ada reference manual as a member of the Tokyo Study Group of Ada. (Takashi Chikayama, author of an article included in this issue, was also a member of the Tokyo Study Group.) A colleague, Hideyuki Nakashima, one of the earliest proponents of Prolog in Japan, was designing Prolog/KR [13]. We and another colleague, Satoru Tomura, started to write a joint paper on input and output (without side effects) and string manipulation facilities in sequential Prolog, with a view to using Prolog as a text processing language instead of languages such as SNOBOL.

The work was partly motivated by our concern about declarative languages: We had been very concerned about the gap between the clean, "pure" version of a language for theoretical study and its "impure" version for practical use. I was wondering if we could design a clean, practical and efficient declarative language.

I had been disappointed with language constructs for concurrent programming because of their complexity. However, Hoare's enlightening paper on CSP (Communicating Sequential Processes) [10] convinced me that concurrent programming could be much simpler than I had thought.

I joined the NEC Corporation in April 1983 and soon started to work on the FGCS project. I was very interested in joining the project because it was going to design new programming languages, called kernel languages, for knowledge information processing (KIP). The kernel languages were assumed to be based on logic programming. It was not clear whether logic programming could be a base of the kernel language that could support the mapping of KIP to parallel computer architecture. However, it seemed worthwhile and challenging to explore the potential of logic programming in that direction.

KL1 Design Task Group

Prehistory

The study of KL1 had already been started at the time I joined the project. ICOT research on early days was conducted according to the "scenario" of the FGCS project established before the commencement of ICOT. Basic research on the Parallel Inference Machine (PIM) and its kernel language, KL1, started in 1982 in parallel with the development of the Sequential Inference Machine and its kernel language, KL0. Koichi Furukawa's laboratory was responsible for the research into KL1.

Although KL1 was supposed to be a machine language for PIM, the research into KL1 was initially concerned with higher-level issues, namely expressive power for the description of parallel knowledge information processing (e.g., knowledge representation, knowledge-base management and cooperative problem solving). The key requirements to KL1 included the description of a parallel operating system as well, but this again had to be considered from higher-level features (such as concurrency) downward, because *ad hoc* extension of OR-parallel Prolog with low-level primitives was clearly inappropriate. The project was concerned also with how to reconcile logic programming and object-oriented programming, which was rapidly gaining popularity in Japan.

Research into PIM, at this stage, focused on parallel execution of Prolog. Concurrent logic programming was not yet a viable alternative to Prolog, though, as an initial study, Akikazu Takeuchi was implementing Relational Language in Maclisp in 1982. It was the first language to exclusively use guarded clauses, namely clauses with guards in the sense of Dijkstra's guarded commands. Ehud Shapiro proposed Concurrent Prolog that year, which was a more flexible alternative to Relational Language that featured read-only unification. He visited ICOT from October to November 1982 and worked on the language and programming aspects of Concurrent Prolog mainly with Takeuchi. They jointly wrote a paper on object-oriented programming in Concurrent Prolog [16]. The visit clearly influenced Furukawa's commitment to Concurrent Prolog as the basis of KL1.

The Task Group

After joining the project in April 1983, I learned it was targeted toward more general-purpose computing than I had expected. Furukawa often said what we were going to build was a "truly" general-purpose computer for the 1990s. He meant the emphasis must be on symbolic (rather than numeric) computation, knowledge (rather than data) processing, and parallel (rather than sequential) architecture.

As an ICOT activity, the KL1 Design Task Group started in May 1983.* Members included Koichi Furukawa, Susumu Kunifuji, Akikazu Takeuchi and me. The deadline of the initial proposal was set for August 1983 and intensive discussions began.

By the time the Task Group started, Furukawa and Takeuchi were quite confident of the following guidelines:

- (Concurrent Prolog) The core part of KL1 should be based on Concurrent Prolog, but should support search problems and metaprogramming as well.
- (Set/stream interface) KL1 should have a set of language constructs that allows a Concurrent Prolog program to handle sets of solutions from a Prolog engine and/or a database engine and to convert them to streams.
- (Metaprogramming) KL1 should have metaprogram-

*Fortunately, I found a number of old files of the Task Group in storage at ICOT, which enabled me to present the precise record of the design process in this article.

ming features that support the creation and the (controlled) execution of program codes.

Apparently, the set/stream interface was inspired by Clark et al.'s work on IC-PROLOG [5], and metaprogramming was inspired by Bowen and Kowalski's work on metaprogramming [5]. The idea of sets as first-class objects may possibly have been inspired by the functional language KRC[18].

I knew little about Relational Language and Concurrent Prolog prior to joining the project. I was rather surprised by the decision to abandon Prolog's features to search solutions, but soon accepted the decision and liked the language design because of the simplicity.

Various issues related to the preceding three guidelines were discussed in nine meetings and a three-day workshop, until we finally agreed on those guidelines and finished the initial proposal. We assumed that KL1 predicates (or relations) be divided into two categories, namely *AND relations* for stream-AND-parallel execution of concurrent processes based on don't-care nondeterminism, and *OR relations* for OR-parallel search based on don't-know nondeterminism. The clear separation of AND and OR relations reflected that the OR relations were assumed to be supported by a separate OR-parallel Prolog machine and/or a knowledge-base machine. (Years later, however, we decided not to create machines other than PIM; we became confident search and database applications could be supported by software with reasonable performance). Set/stream interface was to connect these two worlds of computation. We discussed various possible operations on sets as first-class objects.

Metaprogramming was being considered as a framework for

- the observation and control of stream-AND-parallel computation by stream-AND-parallel computation, and
- the observation and control of OR-parallel computation by stream-AND-parallel computation.

The former aspect was closely related to operating systems and the latter aspect was closely related to the set/stream interface. Metaprogramming was supposed to provide a protection mechanism also. The management of program codes and databases was another important concern. Starting with the "demo" predicate of Bowen and Kowalski, we were considering various execution strategies and the representation of programs to be provided to "demo."

Other aspects of KL1 considered in the Task Group included data types and object-oriented programming. It was argued that program codes and character strings must be supported as built-in data types.

The initial report, "Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1)," which was published as an ICOT Technical Memorandum in September 1983, comprised six sections:

1. Introduction
2. Parallelism
3. Set Abstraction
4. Meta Inference
5. String Manipulation
6. Module Structures

In retrospect, the report presented many good ideas and very effectively covered the features that were realized in some form by the end of the project, though of course, the considerations were immature. The report did not yet consider how to integrate those features in a coherent setting. The report did not yet clearly distinguish between features requiring hardware support and those realizable by software.

ICOT invited Ehud Shapiro, Keith Clark and Steve Gregory in October 1983 to discuss and improve our proposal. Clark and Gregory had proposed the successor of the Relational Language, PARLOG [3]. Many meetings were held and many ICOT people outside the Task Group attended as well.

In the discussions, Shapiro criticized the report as introducing too many good features, and insisted that the kernel language should be as simple as possible. He tried to show how a small number of Concurrent Prolog primitives could express a variety of useful notions, including metaprogramming. While Shapiro was exploring a metainterpreter approach to metaprogramming, Clark and Gregory were pursuing a more practical approach in PARLOG, which used the built-in "metacall" primitive with various features.

From the implementation point of view, most of us thought the guard mechanism and the synchronization primitive of PARLOG were easier to implement than those of Concurrent Prolog. However, the KL1 Design Task Group stuck to Concurrent Prolog for the basis of KL1; PARLOG as of 1983 had many more features than Concurrent Prolog and seemed less appropriate for the starting point. Some people were afraid that PARLOG imposed dataflow concepts that were too static, making programming less flexible.

The discussions with the three invited researchers were enlightening. The most important feedback, I believe, was that they reminded us of the scope of KL1 as the *kernel* language and led us to establish the following principles:

- Amenability to efficient implementation
- Minimal number of primitive constructs (cf. Occam's razor)
- Logical interpretation of program execution

Meanwhile, Furukawa started to design a user-level language on top of KL1. The language was first called Popper (for Parallel Object-oriented Prolog Programming EnviRonment), and then Mandala. On the other hand, the implementation aspect of KL1 was left behind for a long time, until Shapiro started discussions of sequential, but serious, implementation of Concurrent Prolog. The only implementation of Concurrent Prolog available was an interpreter on top of Prolog, which was not fast—a few hundred reductions per second (RPS) on DECsystem-20.

After the three invited researchers left, the Task Group had many discussions on the language specification of KL1 and the sequential implementation of Concurrent Prolog. Although we started to notice that the informal specification of Concurrent Prolog left some aspects (including the semantics of read-only unifica-

tion) not fully specified, we became convinced that Concurrent Prolog was basically the right choice, and in January 1984 started to convince the ICOT members and the members of relevant Working Groups.

Three large meetings on Concurrent Prolog were held in February 1984, which many people working on the FGCS project attended. The Task Group argued for Concurrent Prolog (or concurrent logic programming in general) as the basis of KL1 on the following grounds:

1. It is a general-purpose language in which concurrent algorithms can be described.
2. It has added only two syntactic constructs to the logic programming framework.
3. It retains the soundness property of the logic programming framework.
4. Elegant approaches to programming environments taken in logic programming could be adapted to concurrent logic programs.

People gave implicit consent to the choice of the Task Group in that nobody proposed an alternative basis of KL1 in response to our solicitation. However, as a matter of fact, people were quite uneasy about adopting Concurrent Prolog as the basis of KL1. The arguments being made by the Task Group seemed based on belief rather than evidence. Many people, particularly those working on PIM, were rather upset (and possibly offended) that don't-know nondeterminism of Prolog was going to be excluded from the core part of KL1 and moved to a back-end Prolog engine. Unlike logic programming, the direction of computation was more or less fixed, which was considered inflexible and unduly restrictive. However, Furukawa maintained that the parallel symbolic processing was a more important aspect of KL1 than exhaustive search.

Implementation people had another concern: whether reasonable performance could be obtained. Some of them even expressed the thought that it could be too dangerous to have parallel processing as the main objective of the FGCS project.

Nevertheless, through the series of meetings, people agreed that a user language must be higher level than Concurrent Prolog and that various knowledge representation languages should be developed on top of the user language. We also agreed that programming environments for Concurrent Prolog (and a KL1 prototype) must be developed quickly in order to accumulate experiences with concurrent logic programming. We decided to develop a Concurrent Prolog implementation in a general-purpose language (C was considered first; Maclisp was chosen finally) to study implementation techniques.

In March 1984, the Task Group finalized the report on the Conceptual Specification of KL1 and published it as an ICOT Technical Report [7]. The report now concentrated on the primitive features to be supported directly in KL1 for flexible and efficient KIP.

Implementing Concurrent Prolog

A good way to understand and examine a language definition is by trying to implement it; this forces us to consider every detail of the language. In April 1984, the Task Group acquired some new members, including Toshihiko Miyazaki, Nobuyuki Ichiyoshi and Jiro Tanaka, and started a project on the sequential implementation of Concurrent Prolog under Takeuchi's coordination. We decided to build three interpreters in Maclisp, which differed in the multiple environment mechanisms necessary to evaluate the guard parts of program clauses. The principal member, Miyazaki, was quick in designing and Lisp programming. We also started to build a Mandala implementation in Concurrent Prolog.

As an independent project, Chikayama started to improve Shapiro's Concurrent Prolog interpreter on top of Prolog. By compiling program clauses to some extent, he improved the performance to 4kRPS, a much better number for practical use. I further improved the performance by compiling clauses to a greater degree, and obtained 11kRPS by November 1984, a number better than most interpretive Prolog systems of that time.

We had a general question on the implementation of concurrent logic languages as well, which had been mentioned repeatedly in our discussions on systems programming. The question was whether basic operations such as many-to-one interprocess communication and array updates could be implemented as efficiently as could be done in procedural languages in terms of time complexity. For systems programming without side effects to be practical, it seemed essential to show that the complexity of these operations is not greater than that of procedural languages. I devised an implementation technique for these operations with Chikayama in the beginning of 1984, and presented it in the FGCS'84 conference. These two pieces of work on implementation convinced me of the viability of concurrent logic programming languages as the basis of KL1.

Meanwhile, Clark visited us again in spring 1984, and introduced a revised version of PARLOG [4]. The language had been greatly simplified. Although we were too committed to Concurrent Prolog at that time, the new version influenced the research on KL1 later in various ways.

The three Concurrent Prolog interpreters were almost complete by August 1984, and an interim report comparing the three methods was written. Two successor projects started in September, one on an abstract KL1 machine architecture, and the other on a KL1 compiler. I started to design an abstract machine instruction set with Miyazaki, but was not very excited about it. One reason is that we had found several unclarified points in the definition of Concurrent Prolog, most of which were related to read-only unification and the execution of the guards of program clauses. I started to feel that we should reexamine the language specification of Concurrent Prolog in detail before we went any further. The other reason is that full implementation of the guard construct seemed to be too complicated to be included in a KL1 implementation. The idea of Flat Concurrent Prolog (FCP), which avoided the nesting of guards by allowing only simple test predicates in guards, was conveyed to us by Shapiro in June 1984, but few of us, including me, were interested.

In retrospect, it is rather curious that we stuck to the full version of Concurrent Prolog, which was hard to implement. However, we were not confident of moving to any subset. The guard construct, if successfully implemented, was supposed to be used for OR-parallel problem solving and for the protected execution of user programs in an operating system.

People working on PIM, who were supposed to implement KL1 in the future, were getting impatient in mid-1984. As architects, they needed a concrete specification of KL1 as early as possible and wanted to know what kinds of operations should be particularly optimized, but the design of KL1 had not reached such a phase. On the other hand, members of the KL1 Design Task Group were unhappy that they received few constructive comments from outside. A kind of mutual disbelief was exposed in three meetings of the PIM Working Group held from June to August, in which the Task Group conferred with the PIM people.

Proposal of GHC, a New Basis of KL1

After the FGCS'84 conference in November 1984, I started to reexamine the language specification of Concurrent Prolog in detail, the main concerns being the atomicity (or granularity) of various operations, including read-only unification and commitment, and the semantics of the multiple environment mechanism [19]. Many subtle points and difficulties were found and discussed. I had to conclude that although the language rules could be made rigorous and coherent, the resultant set of rules would be more complex and require more sequentiality than we had expected.

The result of that work was not very encouraging, but I continued to seek a coherent set of language rules. In mid-December, I came up with an alternative to Concurrent Prolog, which solved the problems with read-only unification and the problems with the multiple environment mechanism simultaneously. The idea was to suspend the computation of the guard of a clause if it would require a multiple environment mechanism, that is, if the computation would instantiate variables in the caller of the clause. The semantics of guard now served as the synchronization mechanism as well, making read-only unification necessary.

On December 17, I proposed the new language to the KL1 Design Task Group as KL0.7. The name KL0.7 meant the core part of KL1 that left:

- the decision on whether to include pure Prolog to support exhaustive search directly
- machine-dependent constructs and
- the set of predefined predicates

The handout (in Japanese) included the following claims:

1. Read-only annotation is dispensed with because it does not fit well in fine-grained parallel computation models.
2. Multiple environments are unnecessary. It is not yet clear whether multiple environments must be implemented, while it certainly adds to implementation cost.

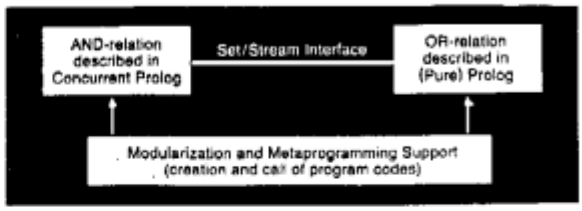


Figure 1.
Conceptual configuration of KL1 (1984) [7]

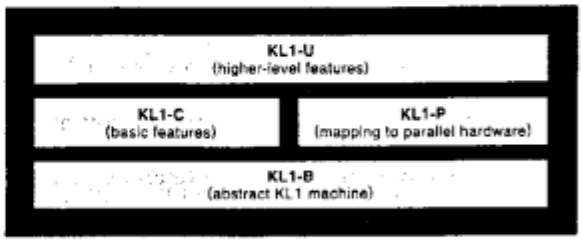


Figure 2.
Structure of KL1 (1985)

Multiple environments make the visibility rule (of the values of variables) and the commitment rule less clear.
3. Mode declaration is dispensed with; it can be excluded from the kernel language level.
4. One kind of unification is enough at the kernel language level, though a set of specialized unification primitives could exist at a lower level.
5. Implementation will be as easy as PARLOG.
6. Implementation will be as efficient as PARLOG.
7. A metainterpreter of itself can be written.
8. Sequentiality between primitive operations is minimized, which will lead to high intrinsic parallelism and clearer semantics.

Interestingly, the resultant language turned out to be semantically closer to PARLOG than to Concurrent and FCP in the sense that it was a single-environment language. Unlike PARLOG, however, it did not assume static analysis or compilation; PARLOG assumed compilation into Kernel PARLOG, a language with lower-level primitives. The handout claimed also that pure Prolog need not be included in KL1 if we made sure that exhaustive search could be done efficiently in KL1 without special support.

The only new aspect to be considered in the implementation of KL0.7 was the management of nested guards. I found it could be done anyway and expected that static analysis would help in many cases. It was not clear whether complex nested guards were really necessary, but they were free of semantical problems and thus could be retained for the time being. In addition, the new language was undoubtedly more amenable to compilation than Concurrent Prolog.

I quickly finished two internal reports, "Concurrent Prolog Re-Examined" and "A Draft Proposal of CPII" and brought them to the Task Group meeting on December 20. The name CPII (for Concurrent Prolog II) was selected tentatively and was used for a while. The Task Group seemed to welcome my proposal and appreciated the simplification.

In January 1985, the Task Group continued designing KL1 with a new basis. Takeuchi proposed that KL1 be CPII with the metacall construct à la PARLOG and primitives for the allocation and scheduling of goals. The proposal well reflected the final structure of the core part of KL1. Set/stream interface and modularization (as a user-level feature) were still considered to be part of KL1, but were put aside for the moment. By January 1985, the Task Group reached an agreement to base KL1 on CPII. The agreement was quick and without too much discussion, because we had agreed to base KL1 on some concurrent logic language, and CPII seemed to have solved most of the problems we had experienced with Concurrent Prolog. CPII did exclude some of the programming techniques allowed in Concurrent Prolog, as Shapiro's group at the Weizmann Institute pointed out later. However, we preferred a language that was simpler and easier to implement.

People outside the Task Group also welcomed the proposal of CPII, though most of them were not yet convinced of the approach based on concurrent logic programming in general. It was not very clear, even to us in the Task Group, how expressive this conceptual language was in a practical sense, much less how to build large parallel software in it. However, there seemed to be no alternative to CPII as long as we were to go with concurrent logic programming, since the language seemed to address "something essential."

In early January 1985, Masahiro Hirata at Tsukuba University, who was independently working on the formal operational semantics of Concurrent Prolog, notified me that the functional language, Qute, designed by Masahiko Sato and Takafumi Sakurai [15] had incorporated essentially the same synchronization mechanism. The news made me wonder if the essence of CPII was simply the rediscovery of a known idea. After learning that Qute introduced the mechanism to retain the Church-Rosser property in the evaluation of expressions, however, I found it very interesting that the same mechanism was independently introduced in different languages from different motivations. This suggested that the mechanism introduced in these languages was more universal and stable than we had thought at first. Apparently, Hirata was independently considering an alternative to the synchronization mechanism of Concurrent Prolog, and later proposed the language Oc [9], which was essentially CPII without any guard goals.

By January 21, I modified my Concurrent Prolog compiler on top of Prolog and obtained a CPII compiler. The modification took less than two days, and demonstrated the suitability of Prolog for rapid prototyping. Miyazaki also made a GHC compiler with more features by modifying Chikayama's Concurrent Prolog compiler on top of Prolog.

In the meantime, I considered the name of the language by putting down a number of keywords in my notebook. The name was changed to Guarded Horn Clauses (GHC) by February 1985.

In March 1985, the project on Multi-SIM (renamed to Multi-PSI later) started under Kazuo Taki's coordination. Its purpose was to provide an environment for the development of parallel software. Thus, by the end of the initial stage, we could barely establish a starting point of research in the intermediate stage.

From GHC to KL1

In June 1985, the intermediate stage of the FGCS project started, and I joined ICOT while maintaining a position at NEC.

Shortly before that, the KL1 Design Task Group (the members being Furukawa, Takeuchi, Miyazaki, Ueda and Tanaka at that time) prepared a revised internal report on KL1. The two main aspects of the revision were (i) the adoption of GHC in view of parallel execution and (ii) the reallocation of proposed features to three sublanguages, KL1-C (core), KL1-P (pragma), and KL1-U (user). KL1-C, the core part of KL1, was supposed to be GHC augmented with some metacall construct to support metainference and modularization. KL1-P was supposed to deal with the mapping between KL1-C programs and physical resources of the underlying implementation. The proposed components of KL1-P were an abstract machine model, a mechanism for allocating goals to processors, and a mechanism for scheduling goals allocated in the same processor. KL1-U was considered as a set of higher-level language constructs to be compiled into KL1-C and KL1-P, which included the support of pure Prolog (with a set/stream interface) and a module construct.

Another sublanguage, KL1-B, was added to KL1 after a while. Although KL1-C and KL1-P were supposed to be the lowest-level sublanguages for programmers, they were still too high-level to be executed directly by hardware. We decided to have a layer corresponding to the Warren Abstract Machine for Prolog. Initial study of the operating system for PIM, called PIMOS, started as well in June 1985.

We had assumed that KL1-C had all the features of GHC, including nested guards, until Miyazaki and I visited Shapiro's group at the Weizmann Institute for two weeks from July to August 1985. During the visit, we had intensive discussion on the differences between GHC and Concurrent Prolog/FCP. We had discussions also on the subsetting of GHC to Flat GHC, an analogue of FCP obtained from GHC.

Colleagues at the Weizmann Institute (Stephen Taylor in particular, who later codesigned Strand and PCN) were greatly interested in Flat GHC as an alternative to FCP. However, they were concerned that the smaller atomic operations of Flat GHC made the language less robust for describing their Logix operating system. In Concurrent Prolog and FCP, a goal publishes binding information to outside *on* the reduction of a goal to others, while in GHC, the publication is done *after* reduction using an independent unification goal in a clause body. The separation made implementation much easier, but caused a problem in their metainterpreter approach to operating systems: the failure of a unification body goal might lead to the failure of the whole system.

Our visit provoked many discussions in the FCP group, but they finally decided not to move to Flat GHC on the ground that Flat GHC was too fragile for the

metainterpreter approach [17]. On the other hand, we chose the metacall approach because we thought the metainterpreter approach would require very careful initial design in order to get everything to work well, which could be too time-consuming for us. The metacall approach was less systematic, but this meant it would be easier to make extensions if required in the development of the PIMOS operating system.

Back in ICOT, a meeting was held to discuss whether we should move from GHC to Flat GHC. Since Flat GHC was clearly preferable from an implementation point of view, the question was whether the OR-parallel execution of different nested guards was really necessary, or if it could be efficiently compiled into the AND-parallel execution of different body goals. We did not have a definite answer, but decided to start with Flat GHC since nobody claimed the necessity of nested guards. A week later, Miyazaki submitted the detailed design of a Flat GHC implementation for Multi-SIM, and Taki submitted the design of interconnection hardware for Multi-SIM. Miyazaki also submitted a draft specification of KL1-C as a starting point for discussions. The points to be discussed included the detailed execution rule of guards, distinction between failure and suspension, the detail of metacall predicates, the treatment of extralogical predicates, requirement for systems programming, and the handling of various abnormal situations (i.e., exception handling).

However, the detail of KL1-C was left unfinalized until summer 1987. We had a number of things to do before that. From an implementation point of view, we first had to develop basic technologies for the parallel implementation of Flat GHC, such as memory management and distributed unification. From a programming point of view, we had to accumulate experiences with Flat GHC programming. Although the focus of the R&D of parallel implementation was converged on (Flat) GHC by the end of 1985, it was still very important to accumulate evidence, particularly from the programming point of view, that the concurrent logic programming approach was really feasible. One of the greatest obstacles to be cleared in that respect was to establish how to program search problems in Flat GHC.

I started to work on compilation from pure Prolog to Flat GHC in the spring of 1985. Since Hideki Hirakawa had developed a pure Prolog interpreter in Concurrent Prolog[8], the initial idea was to build its compiler version. However, the interpreter used an extralogical feature—the copying of nonground terms—which turned out not to make any sense in the semantics of GHC. After some trial and error, in September 1985, I found a new method of compiling a subset of pure Prolog into Flat GHC programs that enumerated the solutions of the original programs. While the technique was not as general as people wanted it to be in that it required the mode analysis of the original programs, the avoidance of the extralogical feature led to higher performance as well as clearer semantics.

Although the technique itself was not widely used later, people started to agree that an appropriate compilation technique could generate efficient concurrent logic programs for search problems. An important outcome along this line was Model Generation Theorem Prover (MGTP) for a class of non-Horn clause sets [6].

My main work in 1985 and 1986 was to examine and justify the language design from various respects and thus to make the language more robust. I had a number of opportunities to give presentations and have discussions on GHC. These were very useful for improving the way in which the language was explained. The paper on GHC was first presented at the Japanese Logic Program Conference in June 1985 [20]. A two-day tutorial on GHC programming, with a textbook and programming assignments, was held in May 1986 and 110 people attended. All these activities were quite important, because people had not had sufficient exposure to actual GHC programs and had little idea about how to express things in GHC.

At first, I was introducing GHC to people by comparing it with the framework of logic programming. However, I started to feel it was better to introduce GHC as a model of concurrent computation. GHC looked like a concurrent assembly language as well, which featured process spawning, message sending/receiving, and dynamic memory management. I revised my presentation transparencies describing the syntax and the semantics of GHC several times. The current version uses only one transparency, where I refer to the syntactic constructs of logic programming only for conciseness.

KL1-related R&D activities in the intermediate stage started as collaboration between the first research laboratory for basic research (to which the KL1 Design Task

From an implementation point of view, **we first had to develop basic technologies for the parallel implementation** of Flat GHC, such as memory management and distributed unification. From a programming point of view, **we had to accumulate experiences** with Flat GHC programming.

Group belonged) and the fourth research laboratory for implementation. As the Multi-SIM project progressed, however, the interaction between the two laboratories decreased. The fourth research laboratory had to design implementation details, while the first research laboratory was concerned with various topics related to concurrent and parallel programming. In November 1986, all the development efforts related to KL1, including the design of KL1, were gathered in the fourth research laboratory.

The detail of KL1 had to be determined with many practical considerations in implementation. GHC was to concurrent logic programming what pure Prolog was to logic programming; there was still a gap between GHC and a kernel language for real parallel hardware. I was of course interested in the design of KL1, but thought there would be no other choice than to leave it to the implementation team. During 1986 and 1987 I spent much of my time in giving tutorials on GHC and writing tutorial articles. I did have another implementation project with Masao Morita, but it was rather a research project with the purpose of studying the relationship between language specification and sophisticated optimization techniques.

In the summer of 1987 Chikayama and his team finally fixed the design of KL1-C and KL1-P. The design of KL1-C reflected many discussions we had since Miyazaki's draft specification, and took Chikayama's memory management scheme based on 1-bit reference counting (MRB scheme) [2] into account. KL1-C turned out not to be a genuine extension of Flat GHC but had several *ad hoc* restrictions which were mainly for implementation reasons. I did not like the discrepancy between pure and practical versions of a language, but I felt that if some discrepancy was unavoidable, the two versions should be designed by different people. In our project, both GHC and KL1 are important in their own rights and had different, rather orthogonal design rationales, which were not to be confused. Fortunately, the discrepancy is far smaller than the discrepancy between pure Prolog and Prolog, and can be negligible when discussing the fundamental differences between GHC and KL1 (see the following subsection "GHC and KL1").

Research in the Final Stage

Since 1987, the activities related to the kernel language in the first research laboratory were focused on basic research on Flat GHC and GHC programming. The additional features of KL1 (by KL1 we mean KL1-C and KL1-P henceforth, ignoring the upper and lower layers) were too practical for theoretical study, and Flat GHC itself still had many aspects to be explored, the most important of which were formal semantics and program analysis.

I had long thought that in order to maintain its own "healthiness," in addition to reconciling parallel architecture and knowledge information processing, a kernel language must reconcile theory and practice. A programming language, particularly a "declarative" one, can easily split into a version for theoretical study and another version for practice, between which no substantial relationship remains. I wanted to avoid such a situation. Unfortunately, the interests of most ICOT theoreticians were not in concurrent logic programming (with a few exceptions, including Masaki Murakami, who worked on the semantics of Flat GHC, and Kenji Horiuchi, who worked on abstract interpretation). Since January 1988, I thought about how the set of unfold/fold transformation rules for Flat GHC, initially proposed by Furukawa, should be justified. I finally developed what could be viewed as an asynchronous version of theoretical CSP, in which each event was a unit transaction between an observee process and its observer, and presented this idea at the FGCS'88 conference.

In the FGCS'88 conference, I was invited to the final panel discussion on theory and practice of concurrent systems chaired by Shapiro, and presented my position on the role and the future direction of kernel languages [24]. The panel was exceptionally well organized and received favorable responses, which was unusual for panel discussions.

I suggested two research directions of KL1 in the panel. The first was the reconstruction of metalevel features in KL1. By metalevel features I meant the operations that referred to and/or modified the "current" status of computation. Jiro Tanaka was interested in the concept of reflection since 1986 and was designing reflective features for Flat GHC with his colleagues. I liked the approach, but felt that a lot of work was necessary before we could build a full-fledged concurrent system with reflective operations.

The second research direction was the simplification of KL1 and the development of sophisticated optimization techniques, the motivation being to promote KL1 programming with many small concurrent processes. The ultimate goal was to implement (a certain class of) processes and streams as efficiently as records and pointers in procedural languages. I became interested in optimization techniques for processes that were almost always suspending, and began studying with Masao Morita in September 1988. The work was intended to complement the R&D of Multi-PSI and PIM and to explore the future specification of KL1 to be used beyond the FGCS project.

We soon devised the basic idea of what we later called the message-oriented implementation technique [25], though it took a long time to generalize it. We found it interesting that Flat GHC programs allowed an implementation technique totally different from the one adopted by all the other implementations.

Sophisticated optimization clearly involved sophisticated compile-time analysis of programs, particularly the global analysis of information flow (mode analysis). Concurrent logic languages employed unification as the basic means of communication. Although mathematically elegant, the bidirectionality of unification made its distributed implementation rather complicated. From the language point of view, the bidirectionality might cause unification failure, the failure of unification body goals. Unification failure was considered an exceptional phenomenon very similar to division-by-zero in procedural languages (not as a superficial analogy, as ex-

plained in [21]), and hence it was much more desirable to have a syntactic means to avoid it than to have it processed by an exception handler.

On the other hand, people working on development were skeptical about global program analysis, suspecting that it was not practical for very large programs. The skepticism, however, led me to develop an efficient mode analysis technique that was efficient and amenable to separate analysis of (very) large programs [25]. The technique was based on a mode system which turned Flat GHC into a strongly moded subset called *Moded Flat GHC*. I presented the technique at ICOT's 1990 new-year plenary meeting. Very interestingly, two other talks at the meeting argued against general unification in KL1 as well. The group implementing distributed unification complained of its complexity. The group working on natural languages and knowledge representation pointed out that unification in KL1 did not help in implementing unification over richer structures such as feature graphs. These arguments convinced me that general unification was not necessary or useful *at the kernel language level*, though the progress made with KL1 implementations on PIM had been too great for us to stop implementing general distributed unification. KL1 implementations on PIM would have been considerably simpler if the mode analysis technique had been proposed earlier.

Reflections and Future Prospects

GHC and KL1

How close is the current status of KL1 to my vision?

In many senses, KL1 was designed from very practical considerations, while the main concern of GHC was the basic framework of concurrent computation. As a positive aspect of KL1's design policy, its performance is no worse than procedural languages in terms of computational complexity, and its absolute performance is also good for a novel symbolic processing language.

On the other hand, the constructs for metaprogramming have stayed rather conservative. I expected that practical metaprogramming constructs with some theoretical background could be designed finally, but it turned out to be very difficult. Also, the precise semantics of guards seems to have somewhat *ad hoc* aspects. For instance, the *otherwise* construct for specifying "default" clauses could have been introduced in a more controlled way that allowed better formal interpretation.

From a methodological point of view, the separation of the two languages, GHC and KL1, turned out to be successful [25]. In designing these two languages, it turned out that we were trying to separate two different, though closely related, notions: concurrency and parallelism. Concurrency has to do with correctness, while parallelism has to do with efficiency. GHC is a concurrent language, but its semantics is completely independent from the underlying model of implementation. Before GHC was designed, Shunichi Uchida, who led the implementation team, maintained that the basic computational model of KL1 should not assume any particular granularity of underlying parallel hardware.

To make effective use of parallel computers, we should be able to specify how a program should most desirably be executed on them—at least when we wish. However, the specification tends to be implementation-dependent and is best given separately. This is an important role of KL1, or more precisely, KL1-P. The clear separation of concurrency and parallelism made it easier to tune programs without affecting their meaning.

On GHC, the main point of success is that it simplified the semantics of guards by unifying two previously distinct notions: synchronization and the management of binding environments. When Gérard Huet visited ICOT in 1988, he finished a CAML implementation of Flat GHC in a few days. I was impressed with the quick, constructive way of understanding a programming language he presented, but this was possible because GHC was so small.

Another point of success is that GHC turned out to be very stable—now for eight years. I always emphasized the design principles and basic concepts of GHC whenever I introduced it, and stubbornly kept the language unchanged. This may have caused frustration to GHC/KL1 programmers. Indeed, the design of GHC has not been considered deeply from a software engineering point of view. However, the essence of GHC is in its semantics; the syntax could be redesigned as long as a program in a new syntax can be translated to a program in the current syntax in a well-defined manner. I found the design of user languages much more difficult to justify, though they should be useful for the development of large software. Many candidates for KL1-U were considered in ICOT, but the current one turned out to be a rather conservative set of additional syntactic conveniences.

Although I have kept GHC unchanged, I have continued to study the language. It added much to the stability of the language and improved the way the language was explained. Many ideas that were implicit when GHC was designed materialized later from the research inside and outside ICOT, and contributed to the justification of the language design. Important theoretical results from outside ICOT include the logical account of the communication mechanism by Maher [12] and Saraswat's work on concurrent constraint programming [14] that subsumes concurrent logic programming and Flat GHC in particular. On a personal side, I have always been interested in clarifying the relationship between concurrent logic programming and other formalisms of computation, including (ordinary) logic programming and models of concurrency. I have also been interested in subsetting and have come up with a strongly moded subset called Moded Flat GHC.

Many people in the project worked on the implementation of KL1 and KL1 programming, and produced innovative outcomes [11]. They were all important in demonstrating the viability of the concurrent logic programming approach and provided useful information for future language design and implementation. I believe our R&D went quite well. A new paradigm of parallel symbolic programming based on a new programming language has gone in a promising direction, though of course, much remains to be done.

Did logic programming have anything to do with the design of KL1? The objective of concurrent logic programming is quite different from the objective of logic programming [22]; nevertheless logic programming played an important role in the design of GHC by giving it strong guidelines. Without such strong guidelines, we might have relied too much on existing concurrency constructs and designed a clumsier language. It is not easy to incorporate many good ideas coherently in a single language.

Consequently, GHC programs still allow nonvacuous logical reading. Instead of featuring don't-know nondeterminism, GHC and other concurrent logic languages tried to give better alternatives to operations that had to be done using side effects in Prolog. Logic programming provided a nice framework for reasoning and search and, at the same time, a nice framework for computing with partial information. Concurrent logic programming exploited and extended the latter aspect of logic programming to build a versatile framework of concurrent computation.

Of course, the current status of concurrent logic programming is not without problems. First of all, the term "concurrent logic programming" itself and the fact that it was born from logic programming were—ironically enough—a source of confusion. Many people considered GHC as an unduly restrictive logic programming language rather than a flexible concurrent language at first. I tried to avoid unfruitful controversy on whether concurrent logic programming languages are "logic" programming languages. Also, largely due to the confusion, the interaction of the concurrent logic programming community with the community of concurrency theory and the community of object-oriented concurrent programming has been surprisingly small. We should have paid more attention to concurrency theory much earlier, and should have talked much more with people working on object-oriented concurrent programming. The only basic difference between object-oriented concurrent programming and concurrent logic programming seems to be whether sequences of messages are hidden or exposed as first-class objects.

ICOT as a Research Environment

ICOT provided an excellent research environment. I could continue to work on language issues for years discussing them with many people inside and outside Japan, which would have been much more difficult elsewhere. Electronic mail communication to and from overseas was not available until 1985. Of the three stages of the project, the initial stage (fiscal 1982 to 1984) was rather different in the sense that it gave us who worked on KL1 much freedom as well as much responsibility for the R&D of subsequent stages.

I have never felt that ICOT's adherence to logic programming acted as an obstacle to kernel language design; the details were largely up to us researchers, and it was really interesting to try to build a system of new concepts based on logic programming.

The project's commitment to logic programming was liable to be considered extremely political and may have come as an obstacle to some of the researchers who had their own fields of interest outside (concurrent) logic programming. However, in retrospect, ICOT's basic research activities, particularly those not directly related to concurrency and parallelism, could focus more on connecting logic programming and their primary fields of interest.

Parallelism, too, was not a primary concern for most people working on applications. Parallel programming in KL1 was probably not an easy and pleasant task for them. However, it was clear that somebody had to do that pioneering work and contribute to the accumulation of good programming methodologies.

Position and Beliefs

Fortunately, I have been able to maintain a consistent position regarding my research subject—at least since 1984 when I became acquainted with the project. I was consistently interested in clarifying the relationship and interplay among different concepts rather than amalgamating them. The position, for instance, reflected in the research on search problems in concurrent logic languages. Although the Andorra principle was proposed later as a nice amalgamation of logic programming and concurrent logic programming, our research on search problems, including the MGTP project, focused on the compilation approach throughout. An interesting finding obtained independently from my work on exhaustive search and the MGTP work is that a class of logic programs, which the specialists call *range-restricted*, is fundamentally easier to handle than others. Thus the compilation approach led us to recognize the importance of this concept.

The separation of a concurrent language, GHC, and a parallel language, KL1, is another example. The panel discussion of the FGCS'88 Conference included a heated debate on whether to expose parallelism to programmers or to hide it. My position was to expose parallelism, but in a tractable form. This was exactly what KL1 tried to address by separating concurrency and parallelism. It is often claimed that GHC is a language suitable for systems programming, but the fact is that GHC itself lacks some important features for systems programming, which are included in KL1.

In language design, there has been a long controversy within the concurrent logic language community on whether reduction (of a goal) and unification (for the publication of information) should be done atomically or separately. Here again, we continued to use the separation approach.

One reason I stuck to the separation of concepts is that the gap between parallel hardware and applications software seemed to be widening and was unlikely to be bridged by a single universal paradigm. Logic programming was a good initial approximation to the paradigm, but it turned out that we had to devise a system of good concepts and notations. The system of concepts and notations was supposed to form a new *methodology*, which the FGCS project was going to establish as its principal objective. GHC and KL1 were to form the substratum of the system. (This is why the performance of KL1 imple-

mentations is very important.) Later on, languages such as GDCC [11] and Quixote provided higher-level concepts and notations. First-order logic itself can be regarded as one of such higher-level constructs, in the sense that MGTP compiles it to KL1 programs. These languages will play the role of Mandala and KL2 we once planned.

I was always interested in the interaction between theory and practice and tried to put myself in between. Now I am quite confident that a language designer should try to pay attention to various aspects including its definition, implementation, programming and foundations simultaneously. Language design requires the reconciliation of constraints from all these aspects. (In this sense, our approach to the project was basically, but not strictly, middle-out.)

Mode analysis and the message-oriented implementation technique were the recent examples of simultaneity working well. It would have been very difficult to come up with these ideas if we had pursued theory and practice separately. In the combination of high-level languages and recent computer architectures, sophisticated program analysis plays an important role. It is highly desirable that such analysis can be done systematically rather than in an *ad hoc* manner, and further that a theory behind the systematic approach is expressed naturally in the form of a language construct. By stipulating the theory as a language construct, it becomes a concept sharable among a wider range of people.

Language designers need feedbacks from specialists in related fields. In semantics research, for instance, one position would be to give precise meanings to given programming languages. However, it would be much more productive if the mathematical formulation gives constructive feedbacks back to language design.

The Future

What will the future of GHC/KL1 and concurrent logic programming in general be? Let us look back to the past to predict the future.

The history of the kernel language design was the history of simplification. We moved from Concurrent Prolog to GHC, and from GHC to Flat GHC. Most people seemed to believe we should implement distributed unification for Flat GHC at first. My present inclination, however, is not to do so. The simplification needed a lot of discussions and experiences, but the performance requirement has always been a strong thrust to this direction. It is not yet clear whether we can completely move to a kernel language based on Moded Flat GHC in the near future, but if successful in moving, I expect the performance can be approximately half of the performance of comparable programs written in procedural languages. The challenge is to achieve the performance in a non-*ad hoc* manner:

> For applications in which efficiency is the primary issue but little flexibility is needed, we could design a restricted version of GHC which allows only a subclass of GHC and/or introduces declarations which help optimization. Such a variant should have the properties that additional constructs such as declarations are used only for efficiency purposes and that a program in that variant is readable as a GHC program once the additional constructs are removed from the source text. [20, Section 5.3]
>
> We hope the simplicity of GHC will make it suitable for a parallel computation model as well as a programming language. The flexibility of GHC makes its efficient implementation difficult compared with CSP-like languages. However, a flexible language could be appropriately restricted in order to make simple programs run efficiently. On the other hand, it would be very difficult to extend a fast but inflexible language naturally. [20, Section 9]

Review of the design of KL1 and its implementation is now very important. The design of different models of PIMs may not be optimal as KL1 machines, because they had to be designed when we did not have enough knowledge about KL1 implementation and KL1 programming. Also, as experimental machines, they included various ideas we wanted to try. Now the machines have been built and almost a million lines of KL1 programs have been written. Based on the experience, we should try to simplify the language and the implementation with minimum loss of compatibility and expressive power.

Another problem facing KL1 is the huge economical and social inertia on the choice of programming languages. Fortunately, the fact that KL1 and other concurrent logic languages address the field of parallel computing makes things more advantageous. For example, PCN [1], a descendant of concurrent logic languages, addresses an important issue: parallelization of proce-

I have never felt that ICOT's adherence to logic programming acted as an obstacle to kernel language design; the details were largely up to us researchers, and it was really interesting to try to build a system of new concepts based on logic programming.

dural programs. I am pleased to see that a new application area of concurrent logic programming is developed this way, but at the same time, I feel we should study whether parallel applications can be made to run very efficiently without interfacing to procedural codes.

Formal techniques, such as verification, are the area in which the progress of our research has been very slow so far. However, we believe that GHC/KL1 is quite amenable to formal techniques compared with other concurrent languages. The accumulation of technologies and experiences should be done steadily, as the history of Petri nets has shown.

In his invited lecture of the final day of the FGCS'92 conference, C. A. R. Hoare concluded his talk, "Programs Are Predicates" [11], with comments on the similarities between his and our approaches to programming languages and formalisms, listing a number of keywords—simplicity, efficiency, abstraction, predicates, algebra, concurrency, and nondeterminism.

Acknowledgments

The author is indebted to Akikazu Takeuchi for his comments on the early design activities of KL1. ■

References

1. Chandy, M. and Taylor, S. *An Introduction to Parallel Programming*. Jones and Bartlett Inc., Boston, 1992.
2. Chikayama, T. and Kimura, T. Multiple reference management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press, 1987, pp. 276–293.
3. Clark, K.L. and Gregory, S. PARLOG: A parallel logic programming language. Res. Rep. DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, London, 1983.
4. Clark, K.L. and Gregory, S. PARLOG: Parallel programming in logic. Res. Rep. DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London, 1984. Also in *ACM. Trans. Prog. Lang. Syst. 8*, 1 (1986), 1–49.
5. Clark, K. and Tärnlund, S.-Å., Eds. *Logic Programming*, Academic Press, London, 1982, 153–172.
6. Fujita, H. and Hasegawa, R. A model generation theorem prover in KL1 using a ramified-stack algorithm. In *Proceedings of the Eighth International Conference on Logic Programming*, MIT Press, 1987, pp. 535–548.
7. Furukawa, K., Kunifuji, S., Takeuchi, A. and Ueda, K. The conceptual specification of the kernel language version 1. ICOT Tech. Rep. TR-054, ICOT, Tokyo, 1984.
8. Hirakawa, H., Chikayama, T. and Furukawa, K. Eager and lazy enumerations in Concurrent Prolog. In *Proceedings of the Second International Logic Programming Conference* (Uppsala Univ., Sweden, 1984), pp. 89–100.
9. Hirata, M. Letter to the editor. *SIGPLAN Not. 21*, 5 (1986), 16–17.
10. Hoare, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (1978), 666–677.
11. ICOT, Ed. In *Proceedings of the Fifth Generation Computer Systems* (Ohm-sha, Tokyo, 1992).
12. Maher, M.J. Logic semantics for a class of committed-choice programs. In *Proceedings of the Fourth International Conference on Logic Programming*. MIT Press, Cambridge, Mass., 1987, pp. 858–876.
13. Nakashima, H. Knowledge representation in Prolog/KR. In *Proceedings of the 1984 Symposium on Logic Programming*, IEEE Computer Society, 1984, pp. 126–130.
14. Saraswat, V.A. and Rinard, M. Concurrent constraint programming (Extended Abstract). In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, N.Y., 1990, pp. 232–245.
15. Sato, M. and Sakurai, T. Qute: A functional language based on unification. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, ICOT (Tokyo, 1984), pp. 157–165.
16. Shapiro, E. and Takeuchi, A. Object oriented programming in Concurrent Prolog. *New Generation Computing 1*, 1 (1983), 25–48.
17. Shapiro, E.Y. Concurrent Prolog: A progress report. *Computer 19*, 8 (1986), 44–58.
18. Turner, D.A. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, ACM, New York, N.Y., 1981, pp. 85–92.
19. Ueda, K. Concurrent Prolog re-examined. ICOT Tech. Rep. TR-102, ICOT, Tokyo, 1985.
20. Ueda, K. Guarded Horn Clauses. ICOT Tech. Rep. TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E., Ed., *Lecture Notes in Computer Science 221*, Springer-Verlag, Berlin Heidelberg, 1986, 168–179.
21. Ueda, K. Designing a concurrent programming language. In *Proceedings of the InfoJapan'90, Information Processing Society of Japan*, Tokyo, 1990, pp. 87–94.
22. Ueda, K. Parallelism in logic programming. In *Inf. Process. 89*, G.X. Ritter, Ed., North-Holland, 1989, pp. 957–964.
23. Ueda, K. and Chikayama, T. Design of the kernel language for the Parallel Inference Machine. *Comput. J. 33*, 6 (Dec. 1990), 494–500.
24. Ueda, K. and Furukawa, K. Transformation rules for GHC programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, ICOT (Tokyo, 1988), pp. 582–591.
25. Ueda, K. and Morita, M. A new implementation technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press, 1990, pp. 3–17. A revised, extended version submitted to *New Generation Computing*.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.6 [**Software**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages, Data-flow languages, Nondeterministic languages, Nonprocedural languages*; K.2 [**Computing Milieux**]: History of Computing

General Terms: Design, Experimentation

Additional Key Words and Phrases: Concurrent logic programming, Fifth Generation Computer Systems project, Guarded Horn Clauses, Prolog

About the Author:

KAZUNORI UEDA is assistant manager of the Computer System Research Laboratory at NEC C&C Systems Research Laboratories. Current research interests include design and implementation of programming languages, logic programming, concurrency and parallelism, and knowledge information processing. **Author's Present Address:** NEC C&C Systems Research Laboratories, Computer Systems Research Laboratory, 1-1 Miyazaki 4-chome, Miyamae-ku, Kawasaki 216, Japan; email: ueda@csl.cl.nec.co.jp

ポジションペーパ

後世に残すとしたらこの技術

— プロセス制御, メモリ管理 に限らず、システム完成に設計から至るまでを振り返り 私の持つノウハウすべてを捧げます —

今井 明 (シャープ (株) 技術本部 情報技術研究所)

1 はじめに

平田さんから、「後世に残す技術」について話をして欲しいという依頼を受けてから、果たして何が残すべき技術なのかについて随分考えた。ところが、思いつくのは、「今度同じようなシステムを作るとしたら、これはやらない」とか「頑張ったつもりだったけれど、あまり見返りがなかったな」とか、「こちら方面を頑張っておけば良かったな」という方が多く、残念ながら「今度も是非これを採り入れたい」と思えるようなことが少ない。そこで、「後世に残せること」だけでなく「これは後世に残せない」と思えることをも指摘し、同じ間違いをしないように示唆することも重要ではないかと考えた。

まず、私はICOTで何をしてきたかだが、私がICOTに着任した昭和62年(1987年)7月は、FGCS'88を一年数カ月後に控えていた頃である。ハードウェアの設計に関しては、詳細は各メーカにお任せして、ICOT研究員はKL1の実行方式を考えようという頃だった。当時、分散KL1処理システムを構築してゆく上での基本設計がすべて終っていた訳ではなかった。しかし、「自分はKL1システムの基本設計の段階から関わって来たのだ」とはとても言えない。つまり、学部を出て入社して3カ月で会社を放り出され、ICOTに着地した人間にとって、ICOTの研究開発のレベルへの到達は、地球から月へ着陸することのように感じ、実際にも最初の1年間は、貢献できる仕事は何一つしていない。実際に、自分で仕事ができるようになったなと感じた時には、既に「分散KL1処理システムはこう作るべし」のアウトラインはできていた。自分はそれを元に、基本処理方式をほとんど変更することなく、共有メモリでの実現のための拡張、VPIM[4]の設計・開発、またFGCS前半年くらいからは、PIM/p[10]への移植、デバッグ、チューニングをやってきた。

以上のような経緯を踏まえると、「これは後世に残せない」と指摘することは、先達の設計したことへの批判にも繋がりかねないのだが、それを今まで無批判に鵜飲みにした自分の方が非難されるべきであり、自己批判の意味だけを持った意見であると解釈して頂きたい。かなり批判的な書き方もするが、我々は「どうすれば実現できるかわからないこと」に対して「とりあえずこうすれば実現できる」ことを見つけるのに精一杯であったのも事実である。しかし、「ペーパーマシンに終らせないで、とりあえず動作するものを作る」ことの意味の重要性は認識して頂けると信じているし、今振り返って、批判的な意見が多くなるのも、「ひと通り動くものができて、やっと善悪を判断できる余裕が出てたんだ」ということだと思う。なお、生まれつき頭が悪いので、「何が悪かった」だけで「こうした方がいい」まで言えない点が多い点もどうか了承して頂きたい。

だんだん言い訳がましくなってきたので、本題に移る。

2 システムの設計を振り返って

2.1 どんな状況にも対応できる Interpreter の限界

我々の構築したPIM上のKL1システムはやはり、やはりinterpreterの域を越えていなかった[4]。KL1コンパイラが頑張ったとはいえ、やはりまだ動的なチェックが多過ぎる。これを如実に物語る極端な例として、ベクタをある整数で初期化するようなプログラム、

```
init_vector(Vect,Idx,Elem,NewVect) :-
  Idx > 0 |
    Idx2 := Idx - 1,
    set_vector_element(Vect,Idx2,_,Elem,Vect2),
    init_vector(Vect2,Idx2,Elem,NewVect).
init_vector(Vect,0,_,NewVect) :- true |
    NewVect = Vect.
```

[4]PIM/p上のKL1システムをinterpreterと呼べるかどうかの議論は置いておくとして

の毎 set_vector_element の度に

- Vect へのデレファレンス有無の判定
- Vect がベクタであるかどうかのタイプチェック
- Vect のサイズを得る
- Idx2 へのデレファレンス有無の判定
- Idx2 が整数であるかどうかのタイプチェック
- Idx2 がベクタサイズより小さいかどうかの判定
- Idx2 が正かどうかの判定

さらに、ループの中には

- Idx へのデレファレンス有無の判定 (ガード部 1 回とと、ボディ部の decrement 組込述語の中で一度)
- Idx が整数であるかどうかのタイプチェック
- Idx2 の MRB[2] を on にする処理 (整数値の場合意味がない)

という過剰なチェックが入っている。MRB によりベクタのオーバーライトができるようになった点は大きな進歩 (後世に残すべき技術) であるが、所詮、配列の初期化なのだから、同じことを行なう C プログラム、

```
foo(array,size,init_value)
int array[],size,init_value;
{
  int i;
  for(i=0;i<size;i++)
    array[i] = init_value;
}
```

との隔たりが大き過ぎる。つまり、この C プログラムのループ部分は、SPARC では 4 命令であるのに対し、上述の KL1 プログラムは、PIM/p で、53 命令[5](13.3 倍) も必要になっている。これでは勝ち目がない。

極言すれば、この一桁以上のコンスタントオーバヘッドの差は、本質的に並列動作できない部分と並列動作可能な部分が同じコードで動いていることに起因する。(簡単に可能なところからでもいいから) データフロー解析で、並列動作できない部分を抽出し、その部分は並列化のためのコンスタントオーバヘッドを取り除いたようなコードを生成することは、現状の処理速度を飛躍的に高めるためには必須といえよう。

2.2 RISC? CISC?

前節で述べた最適化は、PIM/m のようなマイクロプログラムによる KL1-B の interpret ではやりにくい[6]し、だからこそ PIM/p のようなコンパイラの頑張る余地のあるマシンでトライすべきであると思う。

そういう意味で、私が ICOT 研究員になって 3 日目から葉山で行なわれた PIM-WG の合宿[7]におけるポジションペーパ [18] を読むと、

中島 克人 「.... 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方は、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があるが、逆に下手をすると 5 割以上の速度損失になる可能性がある。」

[6] ver09.23 現在。なお、53 命令中には lock/unlock は一切入っていない。

[5] つまり、現状の Multi-PSI, PIM/m のように、一つのボディ組込述語を一つの KL1-B とし、それに opecode を割り振るような方式では、set_vector_element_1st_arg_noneed_deref とか、set_vector_element_all_arg_noneed_deref_no_range_check とか、全部用意するのは不可能。

[7] 当時は何を議論しているかすらわからなかった

既に5年半前に、こういう先達の指摘がありながら、グローバルな最適化についてあまり考慮しなかったのは責められてもやむを得ない気がする。ただし、現状での PIM/p と PIM/m の単体での平均速度差 1.89 倍は、クロック 23%, ロックなどの共有メモリアクセス競合のための対策コストを 10% を差し引いて考えると、速度差が3割程度となり、実装上「下手をした」のでもないだろう。

なお、現状の PIM/p 処理系は、PSL レベルでのデバッグを考慮し、命令入れ換えによるインタロックの回避、delayed 分岐などによる最適化などを行なっていない[8]。これによる最適化の効果が2～3% 見込めることもわかっている。

吉田 かおる 「ソフトウェアから望むことは、至極簡単である。特別なアーキテクチャでなくても良い..... 要素プロセッサは、マイクロプログラマブルであること、WCS はたっぷり、....」

やはり、マイクロプログラムによるプロセッサを望んでいる。

ひと通り動作する KL1 システムを作るまでは、確かに、適切な中間言語 (KL1-B[9]) を定め、「マイクロプログラム」で interpret する方式の方が作りやすかったかも知れない。しかしながら、これから、高速化を考え、グローバルな最適化に手を染めるなら、PIM/p のような RISC マシンの方が頑張り甲斐があると思うし、作りやすいと思う。このような最適化を考えるフェーズでは、むしろ柔軟性が高いのはマイクロマシンではなく、RISC マシンではないか。

2.3 ヒープベースメモリ管理の限界

ヒープベースのメモリ管理はやはり重い。「メモリ管理 = ヒープ管理」を前提としてしまったのは、やはり間違いではなかったか。

スタックベースのメモリ管理に比べ、フリーリストを使ったヒープベースのメモリ管理は、

1. 割付・回収が複雑でコスト高

   スタックなら、ポインタの加減算だけでメモリ割付・回収ができるが、ヒープの場合、(特に) 回収・再利用のコストが高いだけでなく、システムが複雑になることを実感した。

2. フラグメンテーションの問題、メモリアクセスの物理的ローカリティの問題がある

   特に、仮想記憶マシンだと後者が大いに問題になるであろう。

3. 管理すべきポインタの数が多い

   これは、フリーリスト管理をするという前提がある時のみ言えることで、KL1 のように任意のサイズの構造を割り付ける言語の場合、フリーリストポインタの種類がやたら多くなる。ゆえに、すべてをレジスタに置くことができなくなる。

という問題がある。とにかく、ヒープベースを前提に、安易に「ヒープ中の構造をリンクでつなげば、とりあえず実現できる」に傾き過ぎた。

このあたりに関しては、最近平田さんから紹介して頂いた "Continuation Passing Style" というコンパイル技法の宣伝本 [1] をみると、従来ヒープ管理だったものが全てスタックで管理できるなどという宣伝がなされているが (詳細不明)、そういう考えも必要ではないか。

また、先ほどの話と関連するが、KL1 においても、電総研坂井さんのいうところの「強連結枝」[15] を見つけ、それらはスタックを用いて逐次的に実行し、並列に動くべきプロセス間の共有データのみをヒープにおくという方法で、(MRB[2] が狙ったように) ガーベジコレクションの発生を押える工夫が必要であったかも知れない。プロセッサ数以上の並列度は、プログラムの書きやすさには貢献しても、効率には貢献しないのであるから。

2.4 稼働率優先の動的負荷分散

共有メモリマルチプロセッサでは、(佐藤正俊氏の主張 [16] であるところの) 「動的負荷分散は稼働率優先で」は (今のところ) 間違いではないと思う。メモリアクセスローカリティとかをあまり考えない現方式でも、主なベンチマークを 8PE で動かした時の Speedup は、

[8] cc -O で dbx が動かないようなもの。

	Speedup
15Puzzle(6)	1.23
BestPath	1.78
Life	3.54
Pascal	5.07
SemiGroup	5.57
LayerdQueen12	6.03
Pentomino	7.16
KumonQueen12	7.72
Puzzle	7.72

となっている。PIM/p の場合、結構いい加減な負荷分散でも性能が出ているところからして[9]、

- 共有メモリによるプロセッサ間通信のコストはやはり安い
- 遊ばせているくらいなら、働かせろ

という風に安易に結論づけたい。もちろん、単体性能が上がり、共有バスが込んできたら必ずしもこの方針が言える保証はないが、今すぐに、基本方針を考え直して、改良に着手すべき点ではないと思う。

2.5 Copying GC の並列実行は必要か?

Copying GC の並列実行 [7] については一所懸命設計もし、コードも書き、デバッグもし、シミュレータで評価もしたし、それ自体は研究として非常に面白かったし、おまけに海外旅行にも行くことができた。しかし、それを PIM/p 上で動くようにしたのに、その上で評価しようとする気が起きなかった。理由は、

- ほとんどのプログラムにおいて、GC 処理は一瞬で終ってしまうので、とてもボトルネックになっているとは思えない。これは、自分の設計・実装した方式の効率が良かったわけでも何でもなく、Copying GC さえ採用すれば、もともとボトルネックとなりそうな処理ではなかったのではないか。それにしてもそんなに頑張って設計すべきところではなかったのかなと思えた。
- (半分冗談だが) 本方式に関する論文をあまりにもたくさん書き過ぎて、これ以上 GC の評価に関する論文を書くのが後ろめたくなった。

などである。なお、並列化するためのアイディア (構造を 2 の冪乗に丸めて割付け、一つのページには同じ大きさの構造しか置かないことで、ページ境界まで使い切る技法) 自体は、他に使えると思う。

2.6 「動的」に頼り過ぎた

「interpreter の限界」でも少し述べたが、何もかも「動的」では性能が出ない。「静的」でできることはもっとなかったか? ヒントとして、特に構造体定数というものは成功例だったと思う。構造体定数とは、全て値の具体化された構造の場合、予めコンパイル時に構造を割り付けておき、実行時にはその構造へのポインタを渡す処理だけで済ますものである。例えば、

```
p(X) :- true | X=[a,b,c,d,e].
```

の場合、動的に 5 つのコンスセルとを割り付けるのではなく、予め割り付けられた構造へのポインタを渡すだけで済ませている。無駄なコピーが避けられるだけでなく、(PIM は違うが) 仮想記憶マシンでは、同一ページに置くことができるのも嬉しい点である。

なぜこのようなことができるか考えてみると、KL1 の性質上、具体値が変更されることが起こり得ないため、構造体の要素へは read のみで、write がないからである。Mach のメモリ管理 [14] で、copy-on-write という遅延評価による無駄なコピー防止テクニックがあるが、これは、論理的に別の領域を用意しなければいけない場合でも、read だけである限りコピーをしないという方法である。KL1 の場合にも、

```
p(X) :- true | X=[a,b,c,d,Y], q(Y).
```

[9] BestPath は、マルチプロセッサで動かすと荘園切替が頻発するためと思われる

のような場合は、bがコミットする度に動的に5つのコンスセルを割り付けている。ある意味で、[a,b,c,d|..] という構造を割り付けることは、コード領域からヒープ領域へのコピーと見なすことができ、読み出しだけの領域をeagerにコピーしていたといえる。

ページ単位のプロテクションが効くようなハードウェアでは、[Y] へのポインタをプロテクトされたページのアドレスとし、そこへのアクセスが発生した時に、トラップ処理ルーチンで本当にアクセスすべきアドレスをするというような方法も考えられなくはない。いろいろ考えてみたが、

```
p([read(X)|Stream]) :- true |
        read(Item,Size),
        X=normal(Item,Size),
        p(Stream).
```

のようなよくある処理で果たして(うまくいくか/得するか)は不明。

2.7 使用するメモリ量削減のための最適化は有効だったのか?

トレードオフのある選択に迫られた時、メモリを節約するのか、処理速度を速めるのかに、(多くの場合は前者を採ったが)一貫性がなかった。

メモリを節約した例として、

- ロングベクタと、ショートベクタの区別

  VECT0...VECT8 などというタグによるベクタの区別は必要なの? なら、なぜ、STRG1, STRG8, STRG16, STRG32 がないの?

- コードのon-demandによる配布と、構造体表による二重配布防止

  どうして予めコードを全クラスタに配っちゃいけないの? 信頼性のないネットワークで繋がった超並列(分散)システムならまだしも、信頼性のある高速ネットワークで繋がっているわけでしょ? このために、クラスタを跨るゴールの送受信のメカニズムが随分複雑になっている。

- 黒輪出入表のハッシュによる一本化

  そのために、処理(特に排他制御)の複雑化を招いたが、同じものを何度も輸出入する頻度は、コードや構造体変数を予め配っておけば、非常に少ない。

処理時間を節約した例として、

- collect_valueの可能な限りの回収を諦めたこと

  ```
  a(X) :- true | true.
  ```

  において、Xが指す構造は、単一参照である限り再帰的に構造全てを回収することができるが、現実にはリストでCDR部3段で留めている。

2.8 なんでもガリゴリ書き過ぎた

肥大化したUNIXカーネルの反省に基づいて設計された、Mach, Amoeba [12] などのマイクロカーネルによる分散OSの設計では、必要な機能をkernelに取り込むべきものと、サーバとして実現すべきものを区別している。そういう意味で現在の(OSをも記述できる)KL1言語処理システムは、肥大化したUNIXカーネルと同じで、反省対象なのかもしれない。もっと、本質でない部分をカーネルに取り込むのではなく、KL1で書き下す(サーバ化する)べきであった。

例えば、荘園の操作とか、SCSI操作の組込述語のインプリメントを、全て「マイクロプログラム/PSLによるガリゴリコーディング」でやったのが決して良かったとは思えない。こういう特殊なプログラムだけが使える組込述語は、更に一段下の論理の枠組を壊した組込述語を用意し、カーネル(VPIMという名のランタイムシステム)の容量を縮小できなかったか? 一時話が出て、立ち消えになった点を再び反省している。

2.9 では、今後生き残りそうな技術とは?

基本的な技術で、使えそうなことはたくさんあると思う。あまりにいろんなことをしたから、全て網羅できている自信はないが、

1. 「分散環境における、有・無の判定に weight 用いる方式」

   参照の消滅判定に用いた WEC[5] (Weighted Export Count) と、荘園 (タスク) の終了判定に用いた WTC[13] (Weighted Throw Count) があった。

2. 「共有リンク・共有ハッシュチェーンの操作における排他制御方式」

   コンスタント merger を、共有メモリ環境でも、Compare and Swap だけで実現できた [8]。

3. 「論理的な意味と物理的実装はうまく切り分ければ、意味的な美しさを保ったまま、効率的な実装ができるということ」

   つまり、MRB[2] のようなものを持ち込み、破壊的上書きを可能としたこと。

4. 「構造の 2 の冪乗による領域丸め」

   計算機が 2 進数で動く限り、2 の冪乗で丸めることは (速度の点で) 効率良く動くことに寄与する。また、VPIM では、同一ページ (256 語) 内には、同じ大きさの構造のみを置くという方式をとったが、意外にもデバッグに有効であった。つまり、メモリをダンプした時、構造の切れ目がどこかがわかりやすかった。

5. 「ソフトウェアによるコントロール可能な並列キャッシュ機構」

   (処理系の話ではないが)、キャッシュ操作の命令[10]によりキャッシュメモリをトランスペアレントに見せないこと [3] も、システムプログラムレベルでは、効率のためには重要であると思う。

3 システムをデバッグしてみて

デバッグの体験から得られたことなどになると、ちっとも技術的ではないし、ましてはアカデミックなんぞ程遠いということになる。しかし、やはり、そのシステムを設計した人間とあまり面識のないようなユーザが使うことのできるシステムを構築するためには、早期にバグを取り除くための工夫は重要であるし、そのためのノウハウも後世に残すべきと考える。

3.1 とにかく「急がば回れ」

回り道 1: 他に先行した KL1 システム

とにかく、PDSS, Multi-PSI, PIM/m のおかげで、PIMOS を含めた KL1 プログラムに問題があるケースが少なかった (皆無ではなかった) ので、KL1 プログラムを疑うことが少なかった。なお、これは、別に、「急ごうとして回った」わけではない。結果的に、(私の都合良く解釈すれば) PIM/p システムの完成だけを目標にすれば、そのようなシステムを先行して開発して頂いたことは「回った」ことになり、大いに恩恵を受けた。つまり、KL1 プログラムが正しく動作しない場合は、「疑うべきは自分達が作った部分のみ」だと思うしかなかった。人間、困った時には他人を疑いやすい傾向があるが[11]、その点で無駄な努力が少なかったと思う。

とにかく、先行して開発して頂いたおかげもあるが、PIMOS のバグの少なさには驚嘆した。PIMOS で出たバグも、

- KL1 の言語仕様で、そのインプリメンテーションを定めていない部分に依存するもの。つまり、KL1 では、複数クローズ間の上下関係は意味を持たないが、たまたま Multi-PSI, PIM/m では、上のクローズ優先というインプリメンテーションになっていたため、記述の曖昧性に気づかなかった例。
- @processor の解釈の違い

[10] direct-write とか、read-invalidate など

[11] 今井だけか?

のような、並行性と関係のない、小手先で直るバグがほとんどであった。

KL1で書いているのだから当たり前といえば当たり前なのだが、実際に共有メモリマルチプロセッサ上でPIMOSが並列に動作したが故のバグは、私の知る限り、mergerに対する複数のストリームから、ある順序の入力を期待していた場合だけである。PIMOSが、良くKL1を理解している人によって書かれたプログラムであるということを指し引いても、驚くべき短期間でPIMOSが動いたと思う。

(本末転倒かも知れないが)システムの動作開始時期を急いで、暗黙の同期機構を言語に持たせたという意味では「急がば回った」のかも知れない。ただし、このために性能面で失ったものも大きいような気がする。

回り道2: Symmetry上のシミュレータでデバッグ

VPIMのデバッグは、PSLをCに変換し、Sequent Symmetry上のpdbxを用いて行なったが、この上でも再現性のない並列動作ならではのバグがかなり発生した。しかし、強力な[12]pdbxのおかげで、比較的短期間にバグをとることができた。もし、これがなく、PIM/pの実機上でしかデバッグできなかったかと仮定すると、いまだにとれていないバグが多々あるのではないか。

一例であるが、Multi-PSI, PIM/mで採用しているKL1-Bは、共有メモリマルチプロセッサでは「正しく動作しない」。これは、設計段階で気づいたわけではなく、実際にSymmetry上のシミュレータでデバッグ中に、その原因追求の中ではじめて露見したことである。

もちろん、シミュレータだけでバグがすべてとれたわけではなく、PIM/p上のデバッグでも、再現性のないバグもあったが[13]、発生した件数では、Symmetry上で起きたケースの方がはるかに多かった。

並列性の問題だけでなく、実機でSCSIで繋がるFEPをTCP/IPでシミュレータに接続したことも、PIMOS動作時の早期デバッグには役に立った。もっとも、走らせ始めてからログイン開始まで、1時間以上かかったからあまり効率は良くなかったが。

回り道3: 開発支援環境の整備

UNIX-C + GnuEmacsで`M-x find-tags`に相当するような、見たいソースを即座に参照できるような環境[17]を整備しておいたことも、巨大化したVPIMの中をtraverseするのに非常に便利であった。ただし、巨大化しても簡単にtraverseできた点が、VPIMを巨大化たらしめた犯人なのかも知れない。

回り道4: PIM/pの命令レベルシミュレータでデバッグ

実機上の初期デバッグの頃は、ハードウェアが仕様通りに動いていないことなどでかなり戸惑ったりしたこともあったが、とにかく「命令レベルシミュレータ[11]で動くのに、なぜ実機で動かないの?」という疑問があったからこそ、ハードウェアの不具合を早期に発見することができたのだと思う。また、VPIMシミュレータで動くものが、命令レベルシミュレータで動かない場合、VPIMをPIM/pの機械語に変換するPSLコンパイラに問題があるというように、原因を特定できたことも早期デバッグに貢献した。

回り道5: PIM/p実機上のデバッグシステムの充実

かなり時間をかけ、真面目にPIM/p実機デバッグのためのコンソールシステム(CSP)[6]を設計/構築して頂いたおかげで、スムーズにデバッグができた。また、自分がデバッグする上で欲しい機能をあれこれリクエストしたが、それにすぐ応えてもらえるような体制であったことも、今考えれば重要であったと思う。

今反省するとしたら、命令レベルシミュレータのコマンドと、実機上のコンソールシステムのコマンドが違う点で、これは早期に統一しておくべきだったと思う。こういう何気ないことが、実はデバッグに重要な集中力を失う原因になる。

また、「システム記述はPSL、しかしデバッグ時に追うのは絶対番地のPIM/pアセンブルソース」という当初のデバッグ状況を打破すべく設計した**Appendシステム**[14]は実に便利であった。ソースで数Mbyte、コンパイルされたオブジェクトで500K byteもあるシステムのデバッグで、コンソールシステムが上げてくる「XXX番地でaddress boundary exception」と言われて、そこから関連する情報を持ったファイルを目で追いかけて、PSLでのソース位置を特定するのは至難の技である。そこで、システム再構築の度に、「PSLのソース位置、アセンブラのソース位置、それが置かれた絶対番地」などの情報をすべて網羅した"Append

[12] というと言い過ぎの感があるが、「PIM/p上のデバッグシステムに比べると」という注釈をつければ言い過ぎではないだろう。

[13] コードネーム「Tsumegoの謎」とか「Zebraの謎」という名前で、FGCS前にはかなり緊迫したが.....。

[14] "A Pim/P ENvironment for Debug"の略。

サーバ" を立ち上げておき、デバッグ中に、Gnu-Emacs から "Append サーバ" に問い合わせをすることで、PSL でのソース位置などを瞬時に見つけることができるようにした。開発環境として、別に新しいコンセプトではないが、「世間並」さえない環境でのデバッグはやはり辛いのである。

今振り返ると、いろいろ回り道をしたことが、結果的に現状の PIM/p システムの完成にうまく作用したと思うが、逆に言えば、これだけ回り道をしてやっと出来上がるようなシステムは、複雑過ぎるのではないかという疑問も、少し残る。

3.2 予想もしなかったものが役に立つことがある

確かに、「予想もしなかったものが役に立つこと」があったが、だからと言って「じゃあ、今後どうすればいいのだ」に答えられないので、事例を紹介するに留める。

PIM/p の LED

システムの稼働状況が、簡単に、直観的に、低コストでわかってしまう PIM/p の LED パネルは有り難かった。つまり、期待すべき時間で応答がない場合、もう少し待てば答が返ってくるのか、それとも今すぐ Ctrl-C をたたくべきなのかが、LED パネルのおかげで良くわかった。LED パネルはデモ効果だけだと思っていたが、デバッグにも役に立った[15]。

標準品は重宝する

遠隔地からの PIM/p の立ちあげは CSP が UNIX 上で動いていたため、remote login さえすれば何の問題もなく実行できた。Multi-PSI, PIM/m では、これが PSI であったため、remote CSP の設計とかに時間をとられていた。

人間関係

やはり、共同作業には人間関係が重要。日頃から仕事に関係のないつまらない話でも交流があれば、いざ仕事で話をする必要が生じた時にもスムーズである (ちっとも技術的じゃない話だ)。

謝辞

多くの人に支えられ、育てられ、無事に ICOT を卒業することができました。また、この WG でも、良い経験をさせて頂きました。この場をお借りして、お礼申し上げます。

参考文献

[1] A. Appel, "Compiling with Continuations", Cambridge University Press, 1991.

[2] T. Chikayama and Y. Kimura, "Multiple Reference Management in Flat GHC", In *Proc. of Fourth International Conference on Logic Programming*, pp. 276–293, 1987.

[3] A. Goto *et al.*, "Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures", In *Proc. of 16th Annual International Symposium on Computer Architecture*, pp. 25–33, 1989.

[4] K. Hirata *et al.*, "Parallel and Distributed Implementation of Concurrent Logic Programming Language KL1", In *Proc. of International Conference on Fifth Generation Computer Systems 1992*, pp. 436–459, 1992.

[5] N. Ichiyoshi *et al.*, "A New External Reference Management and Distributed Unification for KL1", In *Proc. of International Conference on Fifth Generation Computer Systems 1988*, pp. 904–913, 1988.

[6] ICOT 第 1 研究室 PIM/p 開発グループ, "PIM/p CSP マニュアル (Version 0.93)", ICOT, 1993.

[7] A. Imai and E. Tick, "Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor", IEEE Transactions on Parallel and Distributed Systems, (To appear in 1993).

[15] 今度つける時は、弊社の液晶パネルにして下さい :-)

[8] 今井 明他, "共有メモリマルチプロセッサにおける効率的な KL1 ストリームマージ処理方式", 情報処理学会第 41 回全国大会, 2E-10, 1990.

[9] Y. Kimura and T. Chikayama, "An Abstruct KL1 Machine and its Instruction Set", In *Proc. of Symposium on Logic Programming*, pp. 468–477, 1987.

[10] K. Kumon *et al.*, "Architecture and Implementation of PIM/p", In *Proc. of International Conference on Fifth Generation Computer Systems 1992*, pp. 414–424, 1992.

[11] 三露 哲司他, "並列推論マシン PIM/p のシミュレータ", 情報処理学会第 43 回全国大会, 1Q-4, 1991.

[12] S. J. Mullender *et al.*, "Amoeba – A Distributed Operating System for the 1990s", IEEE Computer, May, 1990.

[13] K. Rokusawa *et al.*, "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems", In *Proc. of the 1988 International Conference on Parallel Processing*, Vol. 1 Architecture, pp. 18-22, 1988.

[14] R. Rashid *et al.*, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", Technical Report CMU-CS-87-140, 1987.

[15] 坂井 修一他, "データ駆動型シングルチッププロセッサ EMC-R における強連結枝モデルの導入", 電子情報通信学会データフローワークショップ 1987 予稿集, pp. 231–238, 1987.

[16] M. Sato and A. Goto, "Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor", Parallel Processing, M. Cosnard, M.H. Barton and M.Vanneschi (Editors), Elsevier Science Publishers B.V. (North Holland), pp. 305–318, 1988.

[17] 高木 常好他, "並列マシンにおける言語処理系の開発環境と実装手法", 情報処理学会研究報告, 90-ARC-83-23, 1990.

[18] 田中 英彦他, "ICOT-WG Workshop on Parallel Inference Machines and Multi-PSI Systems", ICOT-TM 399, 1987.

ポジションペーパ

ポジションペーパに代えて

長沼 次郎 (NTT LSI 研)

超 OR 並列推論マシン (Multi-ASCA) の 自己再構成アルゴリズムとその評価

長沼 次郎　小倉 武

NTT LSI 研究所
〒243-01 神奈川県厚木市森の里若宮 3-1
Phone : 0462 40 2139
E-mail : jiro@nttica.ntt.jp

あらまし

超並列処理、WSI(Wafer Scale Integration) アーキテクチャの研究が盛んである。本稿では先に提案したフォルトトレランスを有する超 OR 並列推論マシン (Multi-ASCA) の自己再構成アルゴリズムとその評価結果を示す。欠陥 PE への要求機能と実現される自己再構成アルゴリズムの関係を明らかにし、再構成された推論プレーンの実行時の動的性能を計算機シミュレーション (Trace-Driven Simulation) により評価する。本自己再構成アルゴリズムは局所的な欠陥 PE の情報のみを用いるが、先に提案した大域的な欠陥 PE の情報を用いた再構成アルゴリズムと同様の高い性能が得られる。

Self-reconfigurable Algorithms and their Evaluations for a Highly OR-Parallel Inference Machine (Multi-ASCA)

Jiro NAGANUMA　Takeshi OGURA

NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi, Kanagawa, 243-01 JAPAN
Phone : +81 462 40 2139
E-mail : jiro@nttica.ntt.jp

Abstract

Highly parallel processing and WSI(Wafer Scale Integration) architectures have been intensively studied. In this paper, self-reconfigurable algorithms and their evaluations for a Highly OR-Parallel Inference Machine (Multi-ASCA) are described. Self-reconfigurable algorithms and required functions for a defect PE are analyzed. The dynamic characteristics of a self-reconfigured inference plane (1024 PEs : 32 × 32 array) are evaluated through the performance study based on a trace-driven simulation. The self-reconfigurable algorithms only use a local information of defect distribution but they achieve high performance level as same as the previous reconfigurable algorithm using a global one.

1 はじめに

推論等の知識情報処理の高速化を目的とした並列推論マシンの研究が活発化している [1][2]。現在、最先端のデバイス技術を用いた WAM[3] ベースの推論プロセッサ (マシンサイクル：50ns 以下) 単体で、1～2 MLIPS (Mega Logical Inference Per Second) が達成されようとしている [4]。ところが、将来的に必要とされる実用的な知識情報処理を実現するためには、さらにこれらの推論プロセッサを 1000 台規模以上で並列処理する技術の確立が不可欠である。このような大規模な並列処理を実現するためには、要素プロセッサ、ネットワークの構成といったアーキテクチャの研究 [5][6]、多数の要素プロセッサに空きなく負荷 (仕事) を割り付ける負荷分散の研究 [7]-[11] とともに、これらをシリコン上で実現するためのフォルトトレランスを有する WSI (Wafer Scale Integration) アーキテクチャの研究 [12][13] が重要である。

我々はこれまで、連想メモリを用いた高速逐次推論マシン (ASCA)[14] を要素プロセッサとし、要素プロセッサ間を簡単なネットワークで多数結合して OR 並列処理を実現する VLSI 向きの超並列推論マシンのアーキテクチャとその負荷分散アルゴリズムの検討を行なってきた。特に、超並列実現の鍵となる通信のオーバーヘッド (通信コスト／推論コスト比の増加傾向 [15]) に着目し、通信量が少なく、局所制御が可能な新たなアーキテクチャ、負荷分散アルゴリズムを提案した [16]。また、このような局所制御を追求したアーキテクチャや負荷分散アルゴリズムは、その結果として、PE 数、プレーンサイズ、ネットワークトポロジー等に依存しない任意個数で任意規模の並列処理の実現を可能とし、本質的にフォルトトレランスな概念が容易に融合できる。物理スイッチを用いて欠陥 PE を正常 PE へマッピングするような従来の冗長構成 (物理的な再構成) ではなく、欠陥 PE の識別子を正常 PE にソフトウェア的にマッピングするだけで再構成を行う (論理的な再構成) 新しいフォルトトレランスの実現が可能となる。このようなフォルトトレランスを実現するため、先に大域的な欠陥 PE の分布情報を用いて欠陥 PE を除去し、正常 PE だけの識別子を再構要するアルゴリズムを提案した [17]。しかし、大域的な欠陥 PE の分布情報を用いているため、ウェーハ内部で自己再構成するアーキテクチャにはあまり向いておらず、製造時の欠陥救済に適していた。

本稿では、先に提案したフォルトトレランスを有する超 OR 並列推論マシン (Multi-ASCA) のための、局所的な欠陥 PE の分布情報のみを用いる新たな自己再構成アルゴリズムとその評価結果を示す。欠陥 PE への要求機能と実現される自己再構成アルゴリズムの関係を明らかにし、再構成された推論プレーンの実行時の動的性能を計算機シミュレーション (Trace-Driven Simulation) により評価する。本自己再構成アルゴリズムは局所的な欠陥 PE の情報のみを用いるが、先に提案した大域的な欠陥 PE の情報を用いた再構成アルゴリズムと同様の高い性能が得られる。このような自己再構成アルゴリズムは、運用時の定期的な欠陥救済にも適用できる。

2 フォルトトレランスを有する超 OR 並列推論マシンの概要

2.1 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。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。

以下の検討では、1024 PEs : 32×32 として、要素プロセッサ当りのリンク数は 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 としたものを採用する (分散メッシュ)。また、通信 / 推論コスト比 $\alpha = 20$ (1 MLIPS / PE, 20 Mbit/s / リンク)、遅延係数 = 0.7、仮想係数 = 16 を用いている。

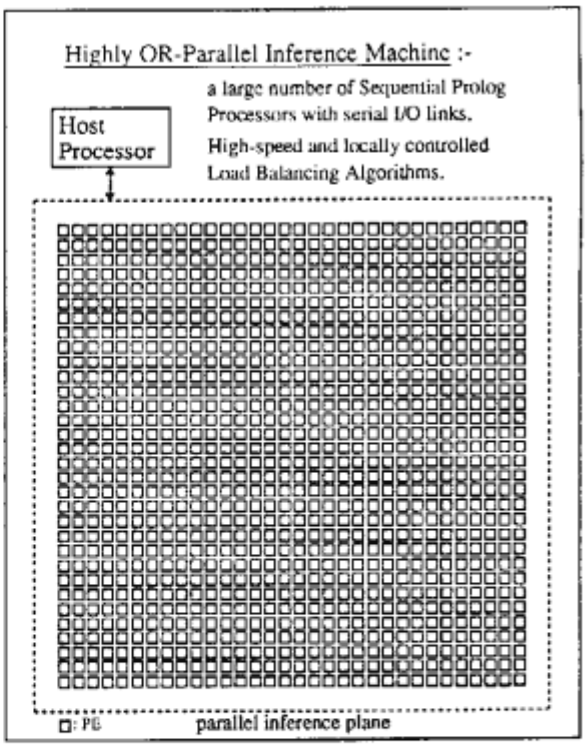


図 1: 超並列推論マシンの基本アーキテクチャ

2.2 負荷分散アルゴリズム

本超並列推論マシンでは、以下の4つの負荷分散アルゴリズムを採用している。

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。

環境再生負荷分散 (RLB)

初期負荷分散の後は、稼動状態に基づく要求駆動により、負荷分散を行う。負荷分散に際しては、従来のように実行環境自体 (バインド情報、コントロール情報等) を転送するのでなく、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、その実行環境を再生する。α を通信 / 推論コスト比とすると、$\alpha > 1$ では、実行環境を転送するより再生した方が有利であり、少ない通信量で高速な負荷分散が実現できる。

遅延負荷分散 (LLB)

負荷分散によるオーバーヘッド (通信処理、OR分割処理) 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控えるべきである。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。

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、初期負荷がより均等化される。

2.3 フォルトトレランスのためのアルゴリズム

本超OR並列推論マシンの欠陥PEを仮定した場合の影響を表1に示す。

表 1: 各負荷分散アルゴリズムの欠陥の影響

アルゴリズム	使用する情報	欠陥の影響
ILB	PE識別子、全体PE数	有
RLB	稼働状況、推論深さ	無
LLB	稼働状況、推論深さ	無
VLB	PE識別子、全体PE数	有

大域的な欠陥PEの分布情報を用いた再構成アルゴリズムでは、欠陥PEの機能として、PEの故障状態は外部から観測できる、PEの識別子は外部から認識できる、の仮定を用いている。大域的な欠陥PEの分布情報を用いた再構成アルゴリズムは、(1) 孤立正常PEの除去、(2) 欠陥PE識別子のマッピングから構成される。

孤立正常PEの除去

自らは正常であるが、結合されているリンク先 (8個) のPEが全て故障している場合、このようなPEに与えられた仕事は他のPEに負荷分散できない。このため、与えられた仕事を最初から最後まで自らの中だけ処理しなければならないため、負荷が重く、全体の並列台数効果を大きく低下する場合がある。孤立正常PEの除去は、このようなPEに仕事を与えないようにする。

欠陥PE識別子のマッピング

大域的な欠陥PEの分布情報を用いた再構成アルゴリズムは、欠陥PEの分布情報を用いて欠陥PEを除去し、正常PEだけの識別子、全体PE数を再構築し、その新たな識別子、全体PE数を用いて、推論処理を開始するものである。

欠陥PE識別子のマッピングのアルゴリズムは以下のステップで実現される。

- 欠陥PEの識別子を収集する
- 全ての欠陥PEの識別子を基に、欠陥PEを除去した識別子、全体PE数を再構築する (図2参照)

欠陥PE識別子のマッピング処理の後、正常なPEは、新たなPE識別子、全体PE数を用いて、通常の推論処理を開始する。このような大域的な欠陥PEの分布情報を用いた再構成は、製造時の欠陥救済に適していた。

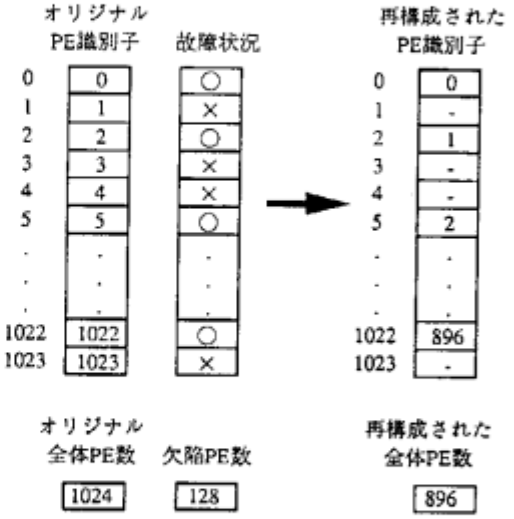


図 2: PE識別子の再構築の概念図

3 自己再構成アルゴリズム

ウェーハ内部で自己再構成するアーキテクチャに適した局所的な欠陥 PE の分布情報のみを用いる新たな自己再構成アルゴリズムを示す。また欠陥 PE への要求機能と実現される自己再構成アルゴリズムの関係を明らかにする。

自己再構成アルゴリズムを実現するため、欠陥 PE の機能として以下のような仮定を用いている。

- PE の故障状態は自己診断できる
- 自己再構成アルゴリズムは実行できる

3.1 自己再構成アルゴリズムの概要

従来、欠陥 PE 識別子の除去および正常 PE の個数の算出に、大域的な欠陥 PE の分布情報を用いていた。これに対し、新たに提案する自己再構成アルゴリズムでは、仮想 PE の概念 [16] を用いて、問題を簡単化し、単なる欠陥 PE 識別子の正常 PE への重複、歯抜けのないマッピング (転送) 処理に置き換えている。このため、本自己再構成アルゴリズムは、従来必要であった欠陥 PE 識別子を除去した正常 PE のみ識別子の再構成を不要化し、局所的な欠陥 PE の分布情報のみを用いて実現できる。

仮想 PE の概念の意義は、論理的な PE 識別子・個数と物理的な PE 識別子・個数との対応を、完全にソフトウェア的に実現することである。これにより、実際の物理的な PE 数を越えた仮想的な PE 数を用いた負荷分散等が実現できる。

このような自己再構成アルゴリズムは、ソフトウェア的な PE 識別子のマッピングであるため、如何なる欠陥分布であっても 100% の再構成が可能であり、すなわち欠陥分布による歩留まり (再構成できるか否か) の議論は不要になる。また、"k-out-of-n" 冗長構成の議論は、単に実現したい欠陥の割合に対し、予備に準備するスペア PE 個数をいくらにするかといった議論となる。

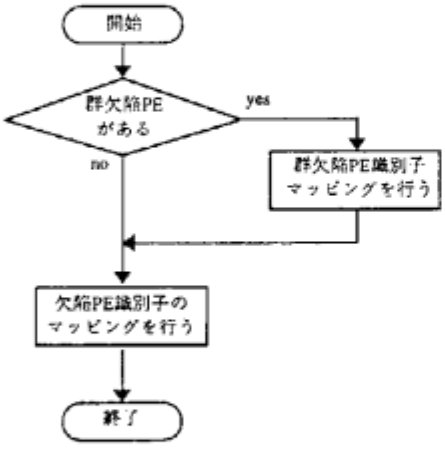


図 3: 自己再構成アルゴリズムの構成

自己再構成アルゴリズムの構成を図 3に示す。本アルゴリズムは図 3に示すように、(1) 群欠陥 PE 識別子のマッピング、(2) 欠陥 PE 識別子のマッピングから構成される。

3.2 群欠陥 PE 識別子のマッピング

群欠陥 PE とは、自ら故障しており、かつ、結合されているリンク先 (8 個) の PE が全て故障している場合を示している。群欠陥 PE は、単純には自らの PE 識別子を正常 PE にマッピングできない。このような状況を回避するため、以下に示す例外処理を行う。

群欠陥 PE 識別子のマッピングは以下のステップで実現される。

- 欠陥 PE は結合されているリンク先 (8 個) の PE の動作状態を調べる
- リンク先の PE が群欠陥 PE でなければ、その PE に自らの識別子を転送する、すべてが群欠陥 PE であれば、任意の PE に自らの識別子を転送する (再帰的に行う)

具体的なアルゴリズムは、以下に示す欠陥 PE 識別子のマッピングが適用できる。この場合、群欠陥 PE 識別子の普通 (群欠陥でない) 欠陥 PE への重複、歯抜けのない転送処理と考えられる。

このような例外処理は、故障率が多い推論プレーンの自己再構成では極めて重要である。25% 以内の実際的な欠陥においては、群欠陥 PE が発生する確率はあまり高くない。

3.3 欠陥 PE 識別子のマッピング

基本的な動作は、欠陥 PE 識別子をどこかの正常な PE に転送する方法である。欠陥 PE 識別子が送られた正常 PE では、その PE 識別子を仮想 PE 識別子として蓄積する。蓄積された仮想 PE 識別子は、仮想負荷分散の中で使用 (消費) される [16]。実現される欠陥 PE 識別子のマッピングの結果は、以下に示す自己再構成のアルゴリズムにより異なる。

3.3.1 転送機能のみ用いた自己再構成法 (A)

この方法は、推論プレーンを一次元のリングとして捉え、欠陥 PE 識別子を識別子の大きい正常 PE の中に蓄えるものである。その概念図を図 4に示す。この場合、欠陥 PE に要求される機能は、隣接転送である。このアルゴリズムは以下のステップで実現される。

このステップは逐次的にリング長分 (PE 個数分) 繰り返される。

- 欠陥 PE は一次元的な PE 識別子の方向に、自らの PE および仮想 PE の識別子を 順次転送する
- 正常 PE に出合うとその中に仮想 PE 識別子として蓄積する

この方法は、推論プレーンを二次元のアレイとして捉え、欠陥 PE 識別子を X 方向、Y 方向、X-Y 方向、Y-X 方向のそれぞれの識別子の大きい正常 PE の中に蓄えることも可能である。その概念は、一次元の場合のリング長が短い場合と等価である。

3.3.2 転送機能と評価機能を用いた自己再構成法 (B)

この方法は、欠陥 PE 識別子を隣接する 8 個の正常 PE の中に、より均一になるように蓄えるものである。その概念図を図 5に示す。この場合、欠陥 PE に要求される機能は、隣接転送と隣接する PE の動作状態を収集し、簡単な論理演算により、ひとつの PE を選択することである。

このアルゴリズムは以下のステップで実現される。このステップは逐次的に PE 個数分繰り返される。

- 欠陥 PE は結合されているリンク先 (8 個) の PE の動作状態を調べる
- 正常な PE のうち、仮想 PE 数の最も小さい PE を選択する
- 選択された正常な PE に自らの識別子を転送し、仮想 PE 数をカウントアップ する

3.3.3 負荷分散機能を用いた自己再構成法 (C)

この方法は、欠陥 PE を稼働状態の PE、他の正常 PE を空き状態の PE と捉え、負荷分散の機能を利用して、欠陥 PE 識別子を正常 PE に転送し蓄えるものである。その概念図を図 6に示す。この場合、欠陥 PE に要求される機能は、負荷分散機能 (隣接転送と隣接する PE の動作状態に応じてひとつの PE を選択する) である。方法 (B) と比べ、PE の選択方法が異なる。また通常の負荷分散と同様に並列分散処理可能である。

このアルゴリズムは以下のステップで実現される。このステップは並列に各 PE で独立に実行される。

- 欠陥 PE は結合されているリンク先 (8 個) の PE の動作状態を調べる
- 正常な PE のうち、ある特定の (負荷分散と同様)PE を選択する
- 選択された正常な PE に自らの識別子を転送し、仮想 PE 数をカウントアップ する

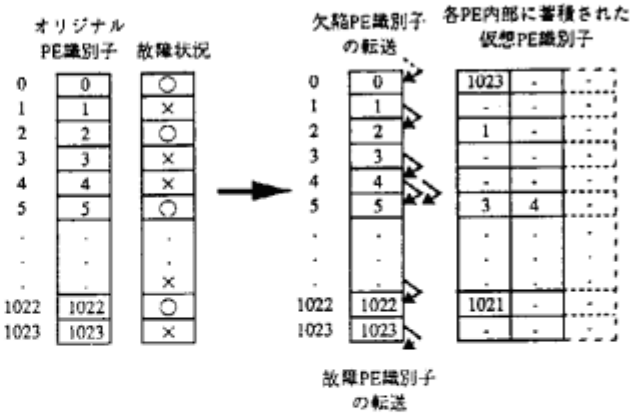


図 4: 転送機能のみ用いた自己再構成の概念図

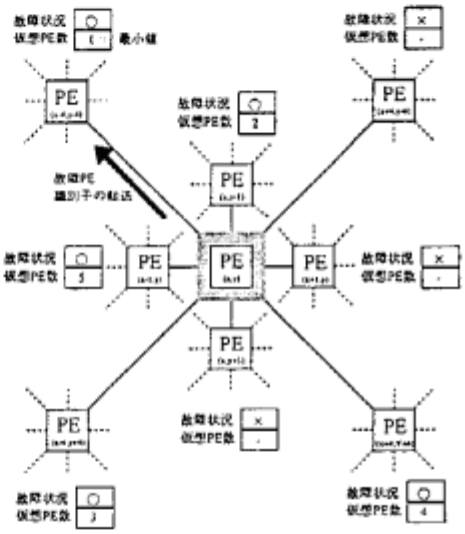


図 5: 転送機能と評価機能を用いた自己再構成の概念図

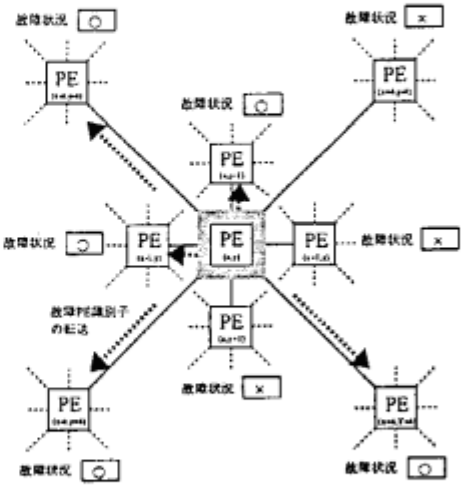


図 6: 負荷分散機能を用いた自己再構成の概念図

3.4 各処理アルゴリズムの特徴

各処理アルゴリズムの特徴を表 2に示す。(A) の方法では、欠陥 PE の機能は、PE 間転送の繰返しであるが、通常の推論処理にはないため、新たに追加する必要がある。本処理は逐次的であり、計算量は最悪値は、PE 個数を "N" とすると "$O(N^2)$" となる。次に、(B) の方法では、欠陥 PE の機能は、PE 間転送 (直接結合のみ) と仮想 PE 数の大小演算である。仮想 PE 数の大小演算処理は、新たに追加する必要がある。本処理は逐次的であり、計算量は PE 個数を "N" とすると "$O(N)$" となる。

これに対し、(C) の方法では、欠陥 PE の機能は、通常の推論処理と同様の負荷分散処理であるため、新たに PE に追加する処理はない。本処理は並列的であり、計算量は PE 個数に依存せず "$O(1)$" となる。実質的には数ステップ程度であり、超並列処理には、最も適した方法と思われる。これらの各処理アルゴリズムは推論実行前に行われ、自己再構成 (欠陥 PE の救済) を行った推論プレーンが実現される。この欠陥 PE の救済を行った推論プレーンの実行時の動的性能は、計算機シミュレーションを用いて次章で詳細に評価する。

表 2: 各処理アルゴリズムの特徴

方法	欠陥 PE の機能	追加機能	計算量
A	転送のみ	PE 間転送繰返し	$O(N^2)$
B	転送+論理演算	仮想 PE 数の演算	$O(N)$
C	負荷分散	なし	$O(1)$

4 性能評価

4.1 シミュレーション方法

新しく導入した自己再構成アルゴリズムにより救済される推論プレーンの実行時の動的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改良した [17]。Trace Driven Simulation [18] 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。本シミュレータの概要を図 7に示す。

本シミュレータは、まず、推論木抽出プログラムにより、評価の対象となる 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、フォルトの種類、自己再構成アルゴリズムの設定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。

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 (perm8)、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 8に示す。

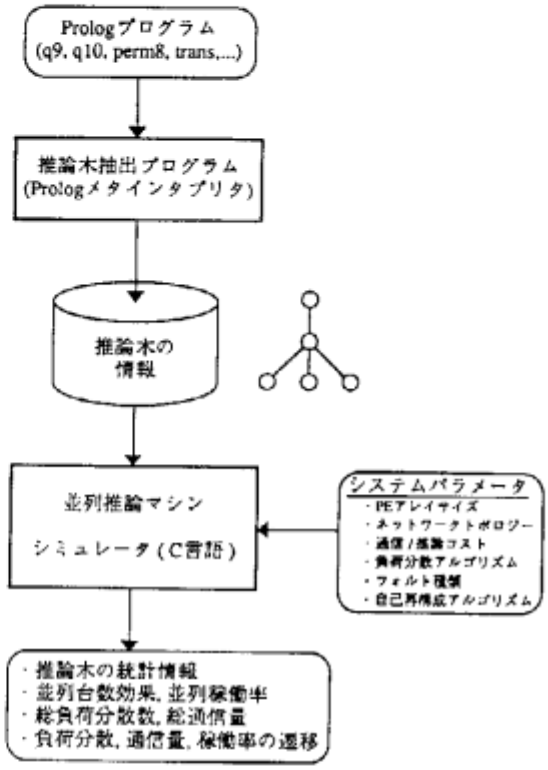


図 7: ソフトウェアシミュレータの概要

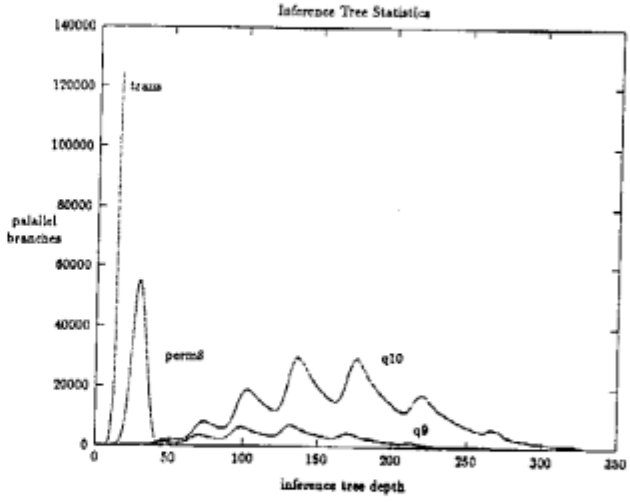


図 8: ベンチマークプログラムの特性

4.3 シミュレーション結果

4.3.1 欠陥 PE が存在しない時の並列台数効果

本超並列推論マシンのフォルトトレランスな性質を把握するため、まず、欠陥 PE が存在しない時の並列台数効果を図 9に示す。1024 PEs : 32 × 32 で、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。

4.3.2 欠陥 PE の割合と並列台数効果

自己再構成された推論プレーンの性質を明らかにするため、本超並列推論マシン全体における欠陥 PE の割合と並列台数効果の関係を調べた。欠陥 PE の発生は乱数を用い、その割合を 0% ～ 25% と変化させた。12.5% の欠陥 PE を仮定した時の欠陥 PE マップを図 10に示す。今回用いた欠陥の割合では、孤立正常 PE、群欠陥 PE は発生しなかった。各自己再構成の方法 (A,B,C) による仮想 PE 個数の分布を付録に示す。

欠陥 PE の割合と並列台数効果の関係を図 11に示す。局所的な欠陥 PE の分布情報を用いて自己再構成する方法 (A, B,C) の場合とともに、比較のため、従来の大域的な欠陥 PE の分布情報を用いて再構成する方法 (global) の場合を示している。

方法 (A)(B)(C) いずれの場合も、欠陥の増加の割合と並列台数効果の低下の割合はほぼ比例している。詳細には、並列台数効果の低下の方が欠陥の増加の割合よりも少ない場合が発生している。これは、並列台数効果そのものが数百台規模以上で飽和傾向にあるためである。

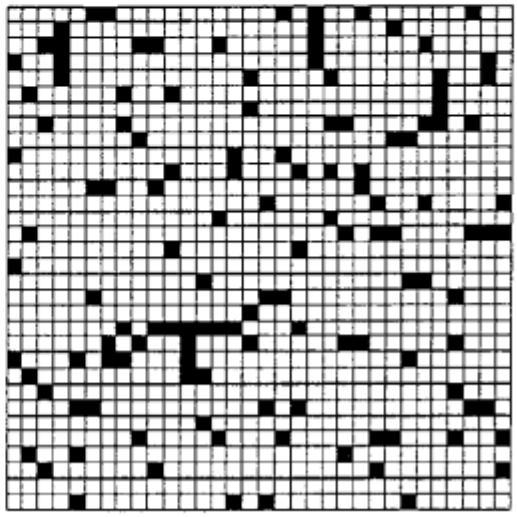


図 10: 欠陥 PE12.5% の時の欠陥 PE マップの例

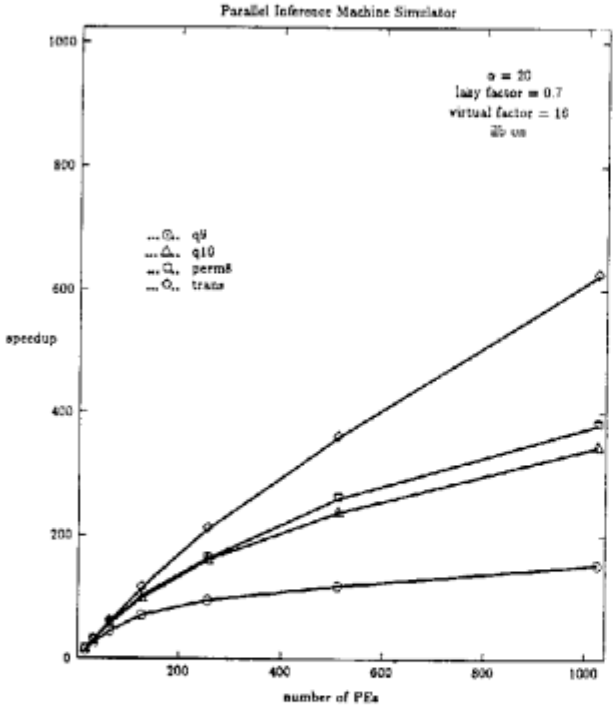


図 9: 欠陥 PE が存在しない時の並列台数効果

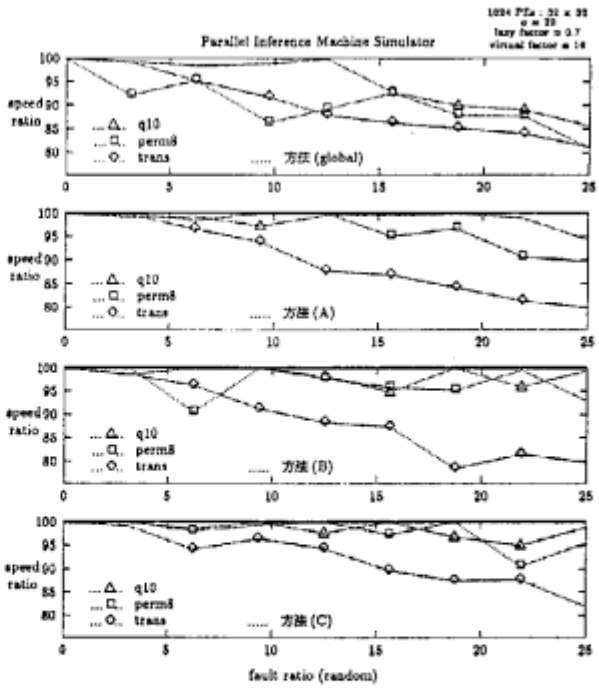


図 11: 欠陥 PE の割合と並列台数効果

4.3.3 各処理アルゴリズムと稼働率の遷移

方法 (global) と方法 (A)(B)(C) の各処理アルゴリズムと稼働率の遷移を図 12に示す (perm8、12.5% 故障)。比較のため、実質有効な PE 数が同一で、故障がない推論プレーン (32×28) の場合 (方法 (ideal))、また ILB を用いない場合 (方法 (ILB なし)) の稼働率の遷移も合わせて示している。

図 12において、図中より明らかなように、稼働率の遷移は、欠陥のない方法 (ideal) の場合、大域的な再構成を行った方法 (global) と、自己再構成を行った方法 (A)(B)(C) の場合でほとんど差がない。すなわち、実質有効 PE 数が同じであれば、欠陥救済した推論プレーンとすべて正常な推論プレーンで実行上の差 (並列台数効果、稼働率等) がないことを示している。また、方法 (B)(C) を比較すると、仮想 PE 数の評価を行いより均一なマッピングを行った方法 (B) と単に評価せず負荷分散に従ってマッピングを行った方法 (C) の方法であまり差がない。このことは、方法 (C) が超並列処理に適した自己再構成の方法であることを示している。方法 (C) は、製造時の欠陥救済のみならず、運用時の定期的な欠陥救済にも適用できる方法である。

なお、図中、自己再構成を行った場合 (方法 (A,B,C)) が、故障がない場合 (方法 (ideal)) より高い性能を示している。これは、自己再構成を行った場合、仮想 PE 数が増加するため、仮想負荷分散の効果により [16]、稼働直後の負荷分散による通信が低減化され、推論処理の稼働率が高められるからである。シミュレーション時間において、0～200 ステップ付近で、方法 (A,B,C) が方法 (ideal) より高い稼働率を示している。仮想負荷分散の効果はプログラムに依存する。

4.4 考察

本自己再構成アルゴリズムのフォルトトレランスな性質は、以下のように整理することができる。

- 欠陥の増加の割合と並列台数効果の低下の割合はほぼ比例している
- 実質有効 PE 数が同じであれば、欠陥救済した推論プレーンとすべて正常な推論プレーンで実行上の差 (並列台数効果、稼働率等) はほとんどない
- 仮想 PE 数の評価を行いより均一なマッピングを行った方法 (B) と単に評価せず負荷分散に従ってマッピングを行った方法 (C) で差が小さい
- 本自己再構成アルゴリズムは局所的な欠陥 PE の情報のみを用いるが、大域的な欠陥 PE の情報を用いた再構成アルゴリズムと同様の高い性能が得られる

このことは、本アーキテクチャが有するフォルトトレランスの性質において、欠陥の割合の増加、欠陥の有無、自己再構成の方法によるオーバヘッドがほとんどないことを示している。

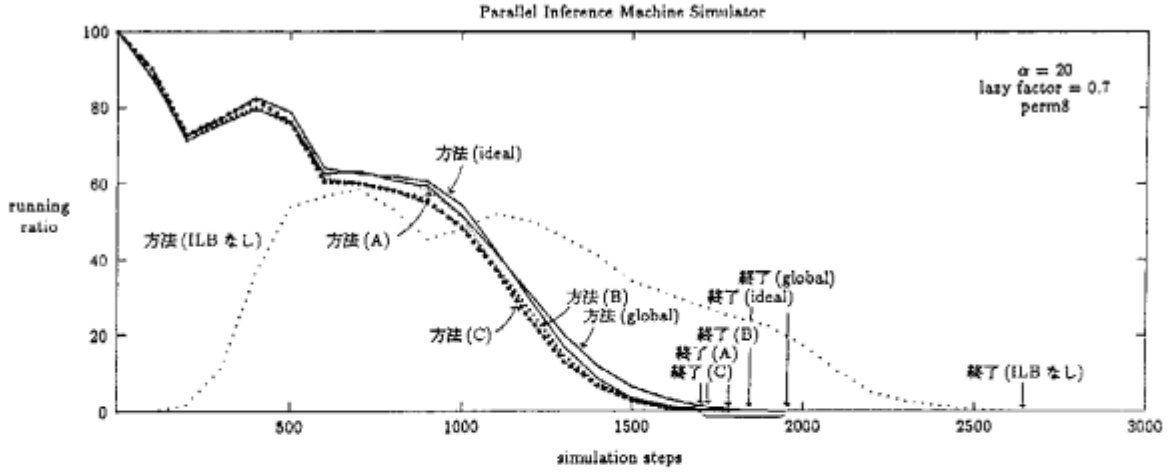


図 12: 各処理アルゴリズムと稼働率の遷移

5 おわりに

物理スィッチを用いて欠陥PEを正常PEへマッピングするような従来の冗長構成(物理的な再構成)ではなく、故障PEの識別子を正常PEにソフトウェア的にマッピングするだけで再構成を行う(論理的な再構成)新しいフォルトトレランスの実現を可能とする超OR並列推論マシンのアーキテクチャのための、局所的な欠陥PEの分布情報のみを用いる自己再構成アルゴリズムを提案した。また、欠陥PEへの要求機能と実現される自己再構成アルゴリズムの関係を明らかにし、再構成された推論プレーンの実行時の動的性能を計算機シミュレーション(Trace-Driven Simulation)により評価した。

本自己再構成アルゴリズムのフォルトトレランスな性質は、欠陥の増加の割合と並列台数効果の低下の割合はほぼ比例しており、また、実質有効PE数が同じであれば、欠陥救済した推論プレーンとすべて正常な推論プレーンで実行上の差(並列台数効果、稼働率等)はほとんどない。本アルゴリズムは、局所的な欠陥PEの情報のみを用いるが、大域的な欠陥PEの情報を用いた再構成アルゴリズムと同様の高い性能が得られる。また、仮想PE数の評価を行いより均一なマッピングを行った方法(B)と単に評価せず負荷分散に従ってマッピングを行った方法(C)で差が小さい。このように、本自己再構成アルゴリズムは欠陥の割合の増加や欠陥の有無、また自己再構成によるオーバヘッドがほとんどないことを示している。

本アーキテクチャでは、欠陥PEに必要な機能として、(1) PEの故障状態は自己診断できる、(2) 自己再構成アルゴリズムは実行できるを仮定している。これらは欠陥PEへの要求される機能としてはあまり高くない。このような機能を仮定することにより、局所的な欠陥PEの情報のみを用いた自己再構成アルゴリズムが実現できる。特に、方法(C)は実質的には数ステップ程度で実現でき、超並列処理に適した自己再構成の方法である。これにより、製造時の欠陥救済のみならず、運用時の定期的な欠陥救済にも適用できる。

一方、これまで提案した方法は、あくまで製造後あるいは運用時の推論実行前に適用できるフォルト(ディフェクト)トレランスである。すなわち、実行中の故障の発生等は考慮していない。しかし、WSI等で実現された超並列処理システムにおいては、その動作の信頼性の観点からも、実行時の故障にも対処すべきである。超並列処理システムにおいては、多重計算方式による多数決法等が有望である。今後、実行時の故障も扱えるフォルトトレランスの実現の可能性を探っていく。

謝辞

本研究を進めるにあたり、終始御指導、御助言を頂いた酒井保良設計システム研究部長に感謝いたします。また、種々の有益な御討論を頂いた安達徹リーダ並びに研究グループの方々に感謝いたします。

参考文献

[1] 柴山, "記号処理マシン," 情報処理, Vol. 28, No. 1, pp. 27-46, Jan. 1987.

[2] 田中, "論理型言語指向の推論マシンの位置付けと開発の現状," 情報処理, Vol. 32, No. 4, pp. 415-420, Apr. 1991.

[3] D. H. D. Warren, "An Abstract Prolog Instruction Set," Tech. Note 309, Artif. Intell. Center, SRI International, Oct. 1983.

[4] 横田, "論理型言語の逐次実行処理方式," 情報処理, Vol. 32, No. 4, pp. 421-434, Apr. 1991. 金田, 松田, "逐次型推論マシンのアーキテクチャ," 情報処理, Vol. 32, No. 4, pp. 450-457, Apr. 1991.

[5] 後藤, "並列型推論マシンのアーキテクチャ," 情報処理, Vol. 32, No. 4, pp. 458-467, Apr. 1991.

[6] 馬場, "超並列マシンへの道," 情報処理, Vol. 32, No. 4, pp. 348-364, Apr. 1991.

[7] 市吉, "論理型言語の並列処理方式," 情報処理, Vol. 32, No. 4, pp. 435-449, Apr. 1991.

[8] J. S. Conery, "The AND/OR Process Model for Parallel Interpretation of Logic Programs," Ph. D. Thesis, University of California, Irvine, 1983.

[9] 久門他, "株分け並列推論方式とその評価," Proc. the Logic Programming Conference '86 (ICOT), pp. 193-200, June 1986.

[10] E. Y. Shapiro, "An Or-Parallel Execution Algorithm for Prolog and its FCP Imple-mentation," 4th Int. Conf. Logic Programming, pp.311-337, May 1987.

[11] K. A. M Ali, "OR-Parallel Ececution of Prolog on BC-Machine," in Proc. 1988 Int. Conf. Logic Programming, pp. 1531-1545, July. 1988.

[12] 南谷, "並列処理におけるフォールトトレランス技術," 情報処理, Vol. 27, No. 9, pp. 1039-1048, Sep. 1986.

[13] 堀口, "WSIデバイスの研究開発動向," 電子材料, Vol. 30, No. 5, pp. 16-23, May 1991.

[14] J. Naganuma et al., "High-Speed CAM Based Architecture for a Prolog Machine (ASCA)," IEEE Trans. Comput., Vol. 37, No. 11, pp. 1375-1383, Nov. 1988. 長沼, 小倉, "連想メモリを用いたPrologマシンの実現とその評価," 信学論 (D-I), Vol. J73-D-I, No. 11, pp. 856-863, Nov. 1990.

[15] 小倉, "デバイス技術の発展 / 限界と並列マシンアーキテクチャ," ICOT PIM WG Workshop, July 1987.

[16] 長沼, 小倉, "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," 信学技法, CPSY 91-45, pp. 23-30, Oct. 1991. J. Naganuma and T. Ogura, "A Highly OR-Parallel Inference Machine Architecture and its Load Balancing Algorithms," IEEE Trans. Comput. (submitted)

[17] 長沼, 小倉, "フォルトトレランスを有する超OR並列推論マシン (Multi-ASCA) の基本アーキテクチャ," 電子情報通信学会, ウェーハスケール集積システム研究会, WSI92-13, Dec. 1992.

[18] P.S. Cheng, "Trace-driven System Modeling," IBM Systems Journal, vol. 8, no. 4, pp. 280-289, 1969.

付録: 自己再構成法と仮想 PE の分布 (欠陥 12.5%)

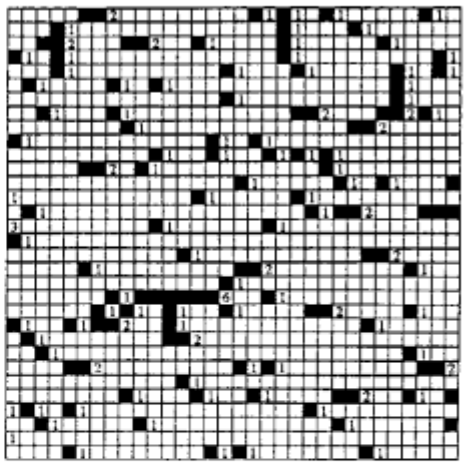


図 13: 転送機能のみを用いた自己再構成法 (A)

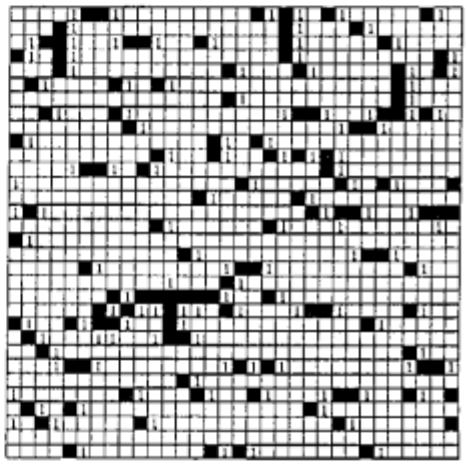


図 14: 転送機能と評価機能を用いた自己再構成法 (B)

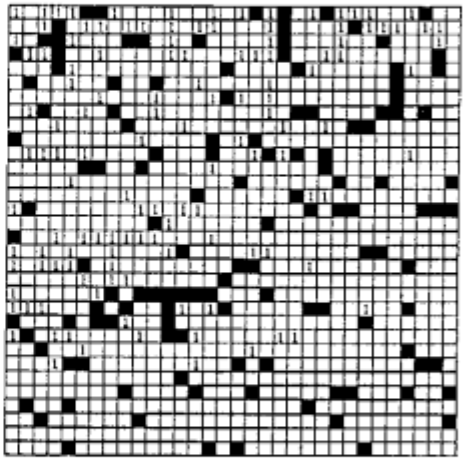


図 15: 負荷分散機能を用いた自己再構成 (C)

ポジションペーパ

ポジション・ペーパー「PIE64・UNIREDとPIM」

島田 健太郎 (東京大学 工学部 電気工学科 田中英彦研究室)

1 初めに

我々の研究室では，PIMに遅れること数年にして，同じような目的を持った並列推論マシンPIE64を開発してきた．私はその中でもプロセッサ要素の中核をなす部品として，UNIREDとよぶ専用プロセッサを設計した．ここでは同じような目的を持ちながら，大学の研究室で作るとPIMとはどのように異なったものになったかという点について，考えてみたい．

2 PIE64

PIE64の各PIMとの違いで，まず目につくのはネットワークであろう．PIE64のネットワークは回線交換で3段オメガ網であり，64台の推論ユニット(IU)と呼ぶプロセッサ要素を均一に結合する．ネットワークはまったく同じものが2系統あり，それぞれDAAN(Data Access/Allocation Network)及びPAN(Process Allocation Network)と呼んでいる．回線交換であることを利用して，使用中でない回線路を用いて各IUの負荷情報を流しており，ネットワーク中のスイッチング・ユニット(SU)はこの情報を使って自動負荷分散接続を行なう機能を持つ．この回線交換ネットワークによる自動負荷分散機能は，以前よりPIEの大きな特徴であった．

各IUには，推論処理を行なう専用プロセッサUNIRED，ネットワークと接続して高度な通信/同期機能を提供するNetwork Interface Processor (NIP)，各種管理を行なうManagement Processor (MP)，及び1Mbytex4バンク構成で計4Mbyteのローカル・メモリがある．PIE64では言語としては，ガード部及びゴール間の論理的関係がない，FGHC / KL1より更に簡略化されたFlengと呼ぶCommitted Choice型言語を用いている．UNIRED，NIPはFlengのために専用化されているが，機能的にはKL1を用いていたとしても，大差無いものであろう．この二種のプロセッサはネットワークのSUと共にCMOSゲートアレイを用いて特に設計したものである．これに対しMPはUNIRED，NIPで扱えなかった処理をすべて吸収するため，汎用RISCプロセッサであるSPARCを用いて柔軟性を大きくとっている．ローカル・メモリはUNIRED，NIP，MPの三種のプロセッサの間で3ウェイ構成のバスにより共有されている．またこれらのプロセッサの間での効率的な協調動作を可能とするために，ローカル・メモリ・バスとは別個にコプロセッサ・コマンド・バスが設けられている．

以上からわかるように，PIE64では各PIMに較べて野心的にアーキテクチャ上さまざまな取り組みがなされており，メモリがすべてSRAMでキャッシュが設けられていないものの，一台のIU当たりのハードウェア規模は大きくなった．これには実装技術の差も大きいと考えられる(PIE64ではSU，UNIRED，NIPの他は専用チップはなく，基本的にPALや標準的なTTLロジックで回路が組まれている)．ネットワークの実装状態は，大学の研究室ならではの高効率実装となっている．

3 UNIRED

次に私が直接設計したUNIREDについて述べる．

UNIREDはPIE64では，推論処理を行なう専用プロセッサとして位置付けられてきた．当初はFlengのプログラムを内部形式に変換してインタープリティブに処理を行なうハードウェア・インタープリタとなる予定であったが，後に変更されてRISC型のパイプライン構成の上に専用の命令セットを実現したプロセッサとなった．この辺りの柔軟性は大学の研究室の賜であろう．

UNIREDでもいくつかのアーキテクチャ上の取り組みがなされているが，最も中心的なものは，多重コンテクスト処理と呼ぶサイクル毎の多重スレッド機構(Cycle by Cycle Multithreading)であろう．これは予めプロセッサ内に複数の(UNIREDでは最大四つの)コンテクスト(プロセス)を用意しておき，サイクル毎に実行可能なコンテクストをパイプラインに流すものである．これによって二つの利点を得ることが出来る．一つは依存関係のある同一コンテクスト内の命令がパイプライン上で離されるためハザードの可能性が減少しインターロックが減る．これはプロセッサ内効果である．二つ目はあるコンテクストが他IUのリモートなメモリをアクセスして待ちの状態になった時，残りのコンテクストの命令で速やかにパイプラインを充足してプロセッサの稼働率を保つ効果である．これはプロセッサ間効果と言える．UNIRED内のコンテクスト(スレッド)へのプロセスの割り付けはMPが行なう．

UNIREDの二番目の特徴は，Flengのための専用命令セットである．この命令セットについては，RISC型パイプラインの上に可能な限りの専用化を行なってみた．即ち性能の上がりそうなものは何でも出来る限り実現する方向で当時のコンパイラ担当の人と議論しながら，設計を行なった．このため命令セットは見た目にはスマートとは言い難く，後々まで議論の対象となったが，このような設計ができるのも大学の研究室の良いところ(悪いところ)であろう．

4 PIMとの比較

PIE64・UNIREDと各PIMを比較して，初めに思うことはやはり柔軟性の違いであろうか．PIE64は大学の一つの研究室で開発しているマシンであり，VPIM / PIMOSという共通事項を抱えた各PIMに較べて遥かに小回りが効いていた．これは逆にともすると行き当たりばったりになりがちとなり，またソフト開発には障害となることもある．我々の研究室では優秀なデバッガが開発されたこともあり，Flengの言語としての研究 / アプリケーションの開発も活発に行なわれてきた．現在PIE64の上に最終的な処理系を構築中であるが，近い内にアプリケーションの開発を中心に移行できるであろう．

各PIMとの比較で，次に目につくことはやはりハードウェア規模であろう．PIE64は64台のプロセッサ要素を持つと言うことで，一台のハードウェア規模は大きいものの，全体としては小さくまとまったような印象を受ける．これは初めから64台という台数が決まっていてネットワークなども設計されたのであるが，もし256台あるいは1024台と言うような台数を実現できたとすれば，どのようになっていたであろうか．PIE64の設計が始まった当初，256台あるいは1024台の構成法について模型を作成して議論もされていたが，実際に見てみたかったと思う．

5 終りに

PIE64・UNIREDとPIMとの違いについて，おもにPIE64側のアーキテクチャの特徴について考えながら議論した．思えばPIE64の設計が始まって6年近く，UNIREDの設計は本格化してから4年の月日が経過した．その間，私としては専門課程の学生として，あるいは以前からのアマチュアとしてのハードウェアの設計の経験はあったものの，プロセッサの命令セットの設計，パイプライン回路の設計など，見より見まねでやってきたことが大半である．そのように思う時，各PIMを見るとやはりプロの開発したマシンという印象を強く感じる次第である．

ポジションペーパ

今思えば、そして今思う

市吉 伸行 ((株) 三菱総合研究所 応用技術部)

1 今思えば

小生は中期以降、マルチPSI上KL1処理系と並列応用に関わってきた。「今思えばこうするべきだった」と言い出すとキリがないが、反省すべき点を敢えて2つ挙げてみる。

- 細部の方式検討に熱中しすぎて、全体的バランスを余り考慮できなかった

  例えば、分散処理方式の検討に当たって、「プロセッサ間処理はプロセッサ内処理の１００倍重い」という仮定をし、だから、如何にしてプロセッサ間メッセージ数を減らすか、に知恵を出しあった。ところが、マルチPSIでの計測結果によると、プロセッサ間通信の処理時間の大半はプロセッサ内におけるメッセージ処理にかかっており、ネットワークは空いていた。

  全体の処理がこれこれで、そのうちこれにどれくらいかかって、というトータルな見方ができなかったことを反省している。[16]

- 応用プログラマのサポートが余りできなかった

  ICOT内外のKL1応用プログラマがKL1プログラミングを習得する労力には大変なものがあった。この点、KL1講習会とその資料は大きな意義があったが、並列プログラム作法には余り立ち入れなかった。[17] 1プロセッサ向けのKL1プログラミングとnプロセッサ向けのKL1プログラミングとの隔たりが大きく、応用プログラマはその間をほとんど独力で渡るしかなかった。

2 今思う（やろう）

- よりよい並列処理方式を目指して

  Dallyの言うような、「1リットル当たり（あるいは1ワット当たり）何MIPS」というような発想[18]には感銘を受ける。超並列をgivenと思って出発するのでなく、ある制約の下での最適解を探すという意識を常に持って、ハードから基本ソフト、引いてはプログラミング生産性まで視界に入れたシステム作りをして行きたいものである（その実力のある技術者はごく僅かだが）。

- 負荷分散ライブラリ

  その反省として、この半年余り、**KL1負荷分散ライブラリ**なるものを開発してきた。負荷分散の典型的パターンをテンプレートとして用意し、ユーザは問題依存部のみを書けばいい（ユーザは@nodeプラグマを書かない！）、というものである。最低レベルのネットワーク生成ユーティリティから、プロセスマッピング・ユーティリティ、動的負荷バランサ、その上の木探索ユーティリティ（枝刈りなし、分枝限定による枝刈りあり、など）、を取り敢えず書いてみた。[19]

  これらは言わば押し着せライブラリであり、ユーザ側に負荷分散に関するの自由度はない。しかし、初心プログラマは初めは、ある型にはまったプログラムで経験を積むのがいいのではないだろうか。習字にしても、初めは教科書や先生の字の丸写しから始める。芸術やスポーツも同様。KL1のような（少なくとも負荷分散に関しての）押し着せがほとんどない[20]言語では、このような押し着せライブラリは意義があると考えている。また、並列プログラムを問題記述部分と負荷分散部分とに分けて考える、という、大袈裟に言えば、並列プログラム方法論を身に付けることも期待している。

  押し着せで並列プログラミングに慣れたプログラマが自分なりの負荷分散方式を工夫したくなったら、例えば、ネットワーク生成ユーティリティをインフラとして用いることもできる。ルーチンの部分を考えなくて済むので、より大事なところに労力を集中できるであろう。

---

[16] 制約の下での最適化問題の一種をシミュレーテッド・アニーリングで解いたようなものだが、温度スケジュールでアニーリング温度を早く低く下げ過ぎた…。

[17] お手本となるプログラムが少なかったし、一般論については誰もよく分かっていなかった（今も？）、というのが実情か。

[18] 究極の定量的アプローチ？

[19] 近くユーザリリースの予定。

[20] データ並列が書きにくいくらいか。

5年たったら、ポスト・グランド・PIM-WG でお会いしたいですね。また同じテーマで。

ポジションペーパ

PIM WG ポジションペーパ
「今思えばこうするべきだった」

杉野 栄二 (北陸先端科学技術大学院大学 情報科学研究科)

今更どうこう言うのも、年寄りの繰り言のようで、なんか気が引けますが、....。

思うのは、『KL1 は、やっぱり 中途半端だったなぁ。』です。KL1 自身は、並列言語として有望な言語だと今でも信じますが、実装に際しては、最初から、ポジションを「システム記述言語」と、はっきり置くべきだったと思っています。

PSI では、ターゲットに ESP という高級言語があって、その下にシステム記述言語として KL0 があったから、KL0 は論理型言語とはいえ、かなりキタナイ機能も持っていました。バスメモリの read/write や、"有名な"(^ ^;) 『バスリセット』なんて組み込み述語は、どう考えたって、キタナイ機能ですが、だから KL0 レベルでいろいろ書けたわけでしょう。 速度は犠牲にしていますが、実現の容易さは間違いなかったはずです。だから、ESP、SIM-POS の実現が可能だったと考えています。

それに比べると、KL1 は、高級言語たらんとするあまり、キタナイことはファームで書き、それでいてシステム記述言語たらんとするあまり、ファームで機能追加するといった風に、しわよせが全部ファーム、言語処理系開発 に行ったように思います。最初から高速化を狙いすぎていたせいかもしれません。

逐次処理系よりも並列処理系の実現に困難があることを考えれば、PSI の時のようなアプローチがあるべきではなかったでしょうか。つまり、初めから KL1 を「システム記述言語」と位置付けて、もっと低レベルでキタナイ機能も持たせるべきではなかったかと感じています。何につけても、その上にターゲット足り得る高級言語がなかったのが悔やまれるところです。

それでもマルチ PSI は、優秀な職人ともいうべきファームグループによって、手作りされたから、あれだけのものになったのですが、... かえって、それが PIM に災いだったかもしれません。(後で速度が比較されるという意味だけでなく (^ ^;))

マルチ PSI のときに、もっと人的パワーがなかったら、KL1 はあれほど太らなかったかもしれません。 いや、太れなかったでしょう。 処理系提供機能も厳選されたかもしれません。 もっとキタナイ機能も組み込み述語で提供されたでしょう。そして、もっと多くの部分が KL1 で記述されることになったかもしれません。そして PIM の開発部分も少なくなったんじゃあないかという気もします。PIM の「システム記述言語」ともいうべき、PSL なんかも存在しえなかったでしょう。

「もし、... たら」ということばかりですが、今から見直すとしたら、KL1 の提供機能の厳選でしょう。 KL1 は、もっとシェイプアップされるべきです。その名の通り、『核』となる機能だけに絞って、システム記述言語として再構成されるべきだと考えます。

(おわり)

ポジションペーパ

ハードとソフトの役割分担 (PIM/p 編)

平野 喜芳 (ICOT)

「ハードとソフトの役割分担」と言われても何を意味するのか良く分からない。そこで、ＰＩＭのハードウェアには並列論理型言語ＫＬ１を実行する際に「ソフトウェアで処理するより高速に実行できる」ように幾つかの特殊な(一般のプロセッサには無い)命令/機構が用意されているので、その命令/機構の効果についてソフトウェアで実現した場合と比較することにより考察する。

ＫＬ１のための「特殊」な命令/機構はＰＩＭの機種によって提供されるものが異なるが、ここではＰＩＭ/ｐを対象にする。ＰＩＭ/ｐのプロセッサは通常のプロセッサが持っているような機械命令はひと通り持っているが、ＫＬ１を実行するために以下のような特殊な命令/機構が追加されている。

- タグアーキテクチャ
- タグ部にあるデータ型の情報を判定する条件分岐命令
- タグ部の情報を判定する条件サブルーチンコール命令
- ＭＲＢをＯＲしながらメモリを読む命令
- リスト構造を操作する命令

これらの評価をするにあたって、ＰＩＭ/ｐシミュレータ(１ＰＥ)上で機械命令毎の実行頻度等を調べる久門さんのツールを、今井さんが使って集計した情報を利用させていただきました。この情報から、上記の項目に関連した機械命令の実行頻度を求めたのが付録にある表です。なお、この評価は機械命令レベルのシミュレータで求めたものでありキャッシュに関しては考慮していない。評価の対象外とします。

タグアーキテクチャ

ＰＩＭ/ｐではプロセッサが直接扱うことのできるデータの種類として「タグ付きワード」(タグ部8bit＋値部32bit)が追加されており、汎用レジスタはタグ付きワードを入れることが可能な幅を持っている。そして、タグ部と値部を一緒または別々に操作する機械命令が用意されている。タグ部と値部を一緒に操作する命令には Read,Write, Move 等が、タグ部だけを操作する命令には Move,And,Or,Xor がある。

付録の表を見てみると、タグ部と値部を一緒に操作する命令が非常に高い頻度(機械命令実行総数比で４０～５５%)で使われていることがわかる。これは、ＫＬ１を実行する際には、構造体の分解生成やポインタの繋ぎ変えのためにメモリの Read/Write が多く行なわれ、そして、基本データがタグ付きワードだけでメモリを他の単位(バイトやワード)でアクセスできないＶＰＩＭをベースに処理系を開発したために、その仕様をひきずっていて、本来必要でない箇所(単一の型しか格納しない変数等、割合は不明)にもタグが付いているのも大きく影響していると考えられる。

タグアーキテクチャをサポートしない(レジスタが32bit幅、タグ部と値部を一緒に操作する命令が無い)場合にどうなるかを考えてみる。そのようなマシン上でタグ付きワードを表現するためには以下の２通りの実現方式が考えられる。

(a) タグ部と値部を別々に扱う方式。

タグ部と値部を別々のレジスタに置くので各種操作に２命令必要になる。また、タグ付きワードのメモリアクセスがアトミックな操作でなくなるので、共有メモリアクセスの排他制御が必要な場面が増える(どの程度かは不明)と考えられる。さらに、(今回のベンチマークレベルではあまり関係無いが)レジスタの必要量が多くなるので、レジスタ上のデータをメモリに退避する場面が多くなるのでその分の速度低下も考慮する必要がある。しかし、処理系の見直しや、コンパイル時の解析を行なうことで、本来必要でない箇所のタグ操作を削除できれば速度低下はかなり(半分以下に？)少なくできる。

(b) タグ部と値部を１ワードにパックする方式。

タグ部と値部が１ワードにパックされていれば通常の命令で操作できるので、タグ付きワードの操作に特別な時間は必要ないが、値部だけを使いたい時(演算やアドレス参照)には値部だけを取り出す処理が必要になる。

このための処理時間はタグ部に割り振るビットの位置や値を工夫する(整数をタグ値0で表す→加減算がそのままできる、タグ部を下位数bitにする→アドレス参照時のオフセットの調整でうち消すことができる)ことで減らせる。ただし、ビット数が少なく制限されるので、タグ部で表現できるデータ型の種類が少なくなり、ショートベクタのようにタグ部に要素数を持たせることは難しくなる。値部のビット数も32bitでなくなるので(ハード依存とわかっていてもやばっり使っている箇所は多いと思う)、言語のコンパチビリティが低下する。

というわけで、現在の(Multi- ＰＳＩコンパチの)ＫＬ１の仕様になるべく忠実にしたい、また、ＶＰＩＭから大幅な変更をしたくないのなら、(a)の方式を取ることになり、ソフトで工夫をしても、排他制御とレジスタの問題を考慮すれば(この見積りは非常に難しいが)５割ほど速度が低下するのではないかと考えられる。(６４ bitプロセッサを持ってくると(b)方式でよくなって速度低下はほとんど無し？)

タグ部にあるデータ型の情報を判定する条件分岐命令

ＰＩＭ/ｐでは単一の命令でデータ型を判定する条件分岐ができる。この条件判定は、タグ部にマスクを掛けた結果が指定の値になるか否かを調べるもので、データ型を表す値を工夫することで、特定の型であるか否かの判定だけでなく、アトミックであるとかベクタグループであるかも判定できる。

付録の表から、この種の条件分岐命令は機械命令実行総数のだいたい１０～１５％の割合になっている。一般的なプロセッサでは、このような処理には、２～３命令(マスク＋比較＋分岐、マスク＋比較&分岐)掛かるので、この種の条件分岐命令が無い場合には、１０～３０％ぐらい総命令実行数が多くなると推定できる。

しかし、マスクした結果を次のデータ型判定の時まで保存できる場面、型チェックが連続する箇所では速度低下は少なくできる。また、データの流れを解析すれば型チェックを省略できる場所もあり(コンパイラの頑張り方にもよるが、かなりの部分が省略可能になると思う)、速度低下はさらに少なくできる。結局、この部分は、コンパイラが頑張れば、ハードサポートが無くても同じ程度の性能が得られると期待できる。

タグを判定する条件サブルーチンコール命令

ＰＩＭ/ｐでは単一の命令でタグ部の値(データ型とＭＲＢを表現する)を判定して条件サブルーチンコールができる。これにより、例えば、reuse_listというKL1-B命令は、ＭＲＢがＯＮならばリストを割り付けるサブルーチンを呼び出すという単一の機械命令に変換できる。

この命令は、現在の処理系では、ＫＬ１をコンパイルしたコードだけで使われており、機械命令実行総数のだいたい２～５％となっている。この命令は例外的な事項が起こった時にサブルーチンコールする場面で使用しており、タグ判定条件分岐命令と無条件サブルーチンコール命令を使うことにより、実行速度をほとんど低下させずに(低下するのはサブルーチンコールする場合＝例外的な場面で実行頻度が少ないと期待される)代替可能である。ただしコードサイズは大きくなる。

なお、現在のＰＩＭ/ｐ処理系では、この命令を十分に活用しているとは言えない。従って、将来、処理系をチューニングすると使用頻度が増えると予想される。

ＭＲＢをＯＲしながらメモリを読む命令

ＰＩＭ/ｐはMRB-GCをサポートするために特殊なRead命令を持っている。ポインタとオブジェクトのＭＲＢ(タグ部の1bit)をＯＲしながらオブジェクトをレジスタ上に読む命令で、デレファレンスや構造体要素の読み出しのために使用される。

この命令の実行数は機械命令実行総数のだいたい３～６％となっている。普通のプロセッサではこのような処理に、３～４命令(Read+MRB判定＋分岐＋MRBセット)掛かるので、この命令が無ければ、６～１８％ぐらい総命令実行数が多くなると推定できる。

しかし、データの流れを解析すればＭＲＢのＯＮ/ＯＦＦをコンパイル時に確定できる場所もあり、その場合には速度低下は少なくできる。やはり、これも、コンパイル時にどこまで頑張るかにかかっているわけで、完全に解析できれば(完全でなくても解析が難しいところは回収をあきらめるというのでも良いと思う)、ＭＲＢそのものが不要になる。

リスト操作命令

これはリスト構造(リスト型データ,フリーリスト等)を操作する命令である。一般的なプロセッサでは２～３命令で実現できる内容であり、実行数は機械命令実行総数のだいたい３～１０％なので、この命令が無い場合には、３～２０％ぐらい総命令実行数が多くなると推定できる。

まとめ

というわけで、非常に大雑把な評価ですが、ＫＬ１のための「特殊」な命令/機構が無く、ソフト側が現状のままの場合には、命令実行数がだいたい２倍ぐらいになってしまうと推定できる。しかし、その分をソフトウェアで頑張ると、頑張った程度により、１．５倍から同等の命令実行数に押えられると期待できる。

当然、これらの命令/機構があって、ソフトウェアも頑張ればより高速実行が可能なわけであるが、その場合には「特殊」命令/機構の利益は相対的に少なくなるはずで、わざわざ専用のプロセッサを作るだけの利益は無いように思うが... それから、現在のソフトはハードに甘えて手を抜いている部分が多いと思う(反省)。

ポジションペーパ

PIM 研究開発の 3 年間を通じて感じたこと

平田 圭二 (ICOT)

私事で恐縮ですが, 平田が ICOT に出向したのは 1990 年 2 月でした. それから約 3 年間, PIM の研究, 開発現場の担当者というより, そういう第一線で仕事をされている再委託 5 社の方々, ICOT で VPIM や PIM/p 上の処理系を研究開発されている方々をバックアップするような気持ちで, いわば中間管理職的マネージャ (?) として 3 年間を過ごさせて頂きました. その間, 色々なことがあり色々思う所がありました. こうして PIM の研究開発を振り返ってみると, 若手の優秀な人材がこんなに沢山揃っているのに, それに見合った成果は出たのかという問いに 100 % YES とは答えられません. 平田のマネージメントに至らない点があったのも一因でしょう. これからの自分に再びこのような機会が巡って来るのかどうか分かりませんが, 将来再びマネージメントをするような立場になった時の自分に宛てる積もりで, 自戒, 自省の意味を込めて, PIM 研究開発マネージメントの「今思えばこうするべきだった」です.

良好な人間関係: PIM のような大規模なシステム作りは, 基本的にチームプレーですから, 適材適所で縦と横の信頼関係を保つ努力は大切だと思いました. 良い関係にないグループや人の間では, 技術も流れないようです. 結局, 作った物や書いたものの信頼度はその人の信頼度に比例するのだと思います.

計画の立て方: 作ること自体の計画だけでなく, 作った物を評価する計画についても事前に良く考えておかねばならないと思いました. また, 計画が予定通り進まない時, どう予定変更するのかということも事前に十分シミュレーションしておく必要があると思います.

中途半端はダメ: 一旦「やる」と決めたら, 何が何でも (不況が来ても, 少々時代遅れだと思っても, 上司が変わっても) とにかく最後までやらなくてはいけないと思います. FGCS'92 後, 1992 年後半, 折角動き始めた PIM システムの評価をせずに, 社内の次プロジェクトに PIM 担当者が散って行った会社があったのは残念でした. 逆に「止める」ならキッパリと止めるべきだとも思います.

Playing Manager: PIM のような未知の分からない物を作るのですから, マネージャ自らハード設計, コーディングをしなければならないと思います. そうでないと, 現場でどういう問題が起きているのか, 問題の本質は何か分からなくなってしまうと思います.

既存技術, 最新技術: 常に基礎体力を鍛えて, 先行研究をしっかり押えなければならないと思います. 新しい物を作っているからと言っても, その過程で出て来る問題のほとんどは未知の問題ではない場合が多かったです. できあがったシステムが良くない理由は, 単に不勉強で, 基礎体力が身に付いていないだけという例もありました. 基本を押えておかないと, やはり本当に解決すべき問題が見えてこないようです.

目的意識, 問題意識: グループ全体が日々の作業に埋没しないように, マネージャは人一倍強力な目的意識, 問題意識を持たなくてはいけないと思いました. playing manager としてシステムと関わり, より多くの人々の共感を呼ぶような本質的に困った体験をし続けるのがその原動力となるのではと思います.

年 1 本の論文: 自分のした仕事を整理する, グループの中での自分の位置を確認する, 回りをサーベイする, という意味でちゃんとした論文を年間最低 1 本は書かなくてはいけないと思いました (reject されても書くことが大切).

海外の動向を見ていると,

技術的に魅力がある = お金が儲かりそう

という図式が成立しているように見えます. 経営者を説得できる研究者技術者, 研究者技術者を信頼できる経営者が居るということなのかとも思います. そういう意味では, 日本にはスーパースター的研究者が居ないのだと思います.

いろいろ偉そうなことを書いてしまいました. このような場で, このようなことを書くのは釈迦に説法で大変恐縮してしまいます. これは恥ずかしながら飽く迄自分の至らなかった点の反省ですので, 誤解のないようにお願い致します.

言語のことを少し

FGHC のプログラム変換に関して最近少し研究をしていますので少しコメントを述べたいと思います.

いざ何か書こうと思ったら, 平田の場合 C か Prolog か何かそこら辺を使ってしまいます. 手元に PDSS があって KL1 が走る環境にあっても, やっぱり C か Prolog です. 何がそうさせるのだろうかと考えています. 何かが足りないのか; メタ機能か? 効率か? プログラミング環境か? 信頼度? あるいは単なる習性か? 何かが余分なのか; ……???

しかし, 将来並列マシンを制御する言語としては, logic に基づく FGHC は有望だと平田は直観的に思います (と平田が書いても何の説得力もありませんが). FGHC を並列言語の milestone として今後の研究を進めて行ければと思います.

PIM/i

発表者: 武田 浩一、六沢 一昭 (沖電気)
日時: 2:00 ～ 2:45, 3/22
議事録担当: 高木 常好 (ICOT)

1 研究開発の歴史

86.1 PIM/i 開発スタート
87.6 ハードウェア具体設計スタート
88.3 PE ボード以外のハードウェア完成
89 IP89 実験版処理系
90.3 IP90
90.12 VPIM 実装開始
92.5 VPIM 動作
92.6 FGCS
92.12 処理系の改良により 1.5 倍

2 ハードウェア / アーキテクチャ

2.1 PIM/i 概要

ハードウェア構成の説明

2.2 ここはよかった

(1) LIW 命令レベル並列性の利用
水平型マイクロ命令 → (V)LIW
複合的な専用命令を導入しないアプローチ

(2) ハードウェアの増加分と性能利得 もとはとれた
レジスタファイルのポート数 Tr 数 4/3
命令メモリ 5/4
タグ操作ユニット Tr 数 2 %
タグ判定ユニット Tr 数 15 %

命令あたりの並列度 1.45
サイクルあたりの並列度 1.26

(3) 並列度に貢献したもの
タグ操作、タグ判定分岐

(4) 特別なメモリ空間を持たない
心理的な圧迫感がない
オーバーフローの苦労がない

(5) パストレースメモリ
デバッグ用
パストラフィックのモニタに使えた

2.3 動作クロックの言い訳、こうすればよかった

100ns → 240ns
プロセッサは 90 ～ 110ns で動くはずなのだが...

(1) 原因不明のパリティエラー

(2) 共有メモリの遅延見積りミスにより 30～40ns 増

(3) クリティカルパスのチェックが不十分

(4) アドレスデコード
ローカルメモリの実装でクロックが延びた
デコードは以外と時間がかかる

(5) 診断用ハードウェア、パリティチェック

(6) 全体が同一クロック
デバッグしやすいが、低速な部分(メモリコントロール)で同期が決まる
シミュレーション、クリティカルパス解析をしておけばよかった
時間がなかった(「急げ」と言われたので...)

(7) プロセッサのクリティカルな部分
条件コード

ゼロ条件が最もクリティカル
条件を生成する命令を限定すればよかった

割り込み

複数の割り込みが重なると... などといろいろ考えると、ロジックが重くなってしまった

(8) キャッシュメモリ
スヌープ方式

2 ポートメモリはコストが大きい
PIM/i プロトコルはコストパフォーマンスがわるい

サイクル毎の連続書き込み制御を入れたかった
書き放し制御を入れたかった

(9) プロセッサ
命令長 40 ビットでは制限がきつい

64 ビットにすればバイトアドレスマシンとしてすっきり

ピン数の制約(当時の CAD で 208 ピンしかなかった)

ピンを共有する方法が後に分かったが既に遅かった

2.4 もう一度作るとしたら

on the CAD マシン
十分なシミュレーションをしてから作る

3 議論

後世に残すとしたらこの技術に関するコメントは?

パイプライン制御に力を入れて作ったので、その設計方式を残したい

「今思えばこうするべきだった」

そして

「後世に残すとしたらこの技術」

PIM/i

六沢 一昭　武田 浩一

・研究開発の歴史

・ハードウェア/アーキテクチャ

PIM/iハードウェア編

沖電気工業株式会社

武田　浩一

2

・PIM/i　の概要

・ここはよかった

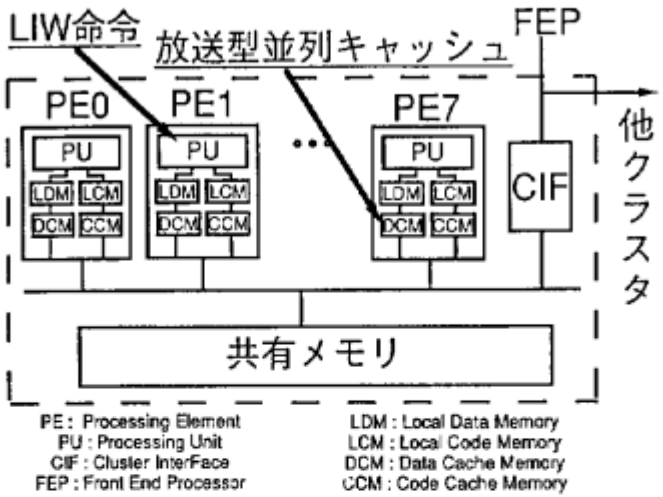
・動作クロックの言い訳

・こうすればよかった

・もう一回作るとしたら

3

PIM/iのハードウェア構成



4

ここはよかった

LIW　命令レベル並列性の利用

垂直型マイクロ命令 → RISC
水平型マイクロ命令 → (V)LIW

命令クラス(分岐，メモリアクセス，演算，タグ操作)毎の並列処理

複合的な専用命令を導入しないアプローチ

ハードウェアの増加分と性能利得	もとはとれた
レジスタファイルのポート数(3ポート → 5ポート)	Tr数で4/3倍
命令メモリ(32ビット → 40ビット)	5/4倍
タグ操作ユニット	Tr数で2%
タグ判定ユニット	Tr数で15%
命令あたりの並列度	1.45操作
サイクルあたりの並列度	1.26操作

並列度に貢献したもの

タグ操作，タグ判定分岐

5

ここはよかった

特別なメモリ空間をもたない

制御記憶やそれに相当する別の世界をもたない

ランタイムルーチンのコードをサブルーチン分岐命令で呼び出す

コンパイルコードもランタイムコードもどこにおいてもよい

心理的な圧迫感がない

メモリオーバフローの苦労を味わなくてすんだ

6

ここはよかった

バストレースメモリ

デバッグ用

バスの内容を2Kサイクル分トレースし，記録する機能を用意した

キャッシュ間転送のデバッグ等

予定外の使い方

バストラヒックのモニタ

7

動作クロック

100nsのはずが

240ns

プロセッサは90から110nsで動作可能なのだが…

8

プロセッサのクリティカルな部分

条件コード
　ゼロ条件が最もクリティカル
　条件を生成する命令を限定したほうがよかった

割り込み
　基本動作はふたつの戻り番地と幾つかの状態を退避して，
　割り込みベクタを出力する

　割り込みマスクのチェック，
　複数の割り込みが同時に起きた場合，
　キャッシュミスが同時に起きた場合等を考えると
　ロジックが割と重くなって！

いいわけ

実装，回路設計上

原因不明のパリティエラー

共有メモリの遅延見積りミス

クリティカルパスのチェックが不十分

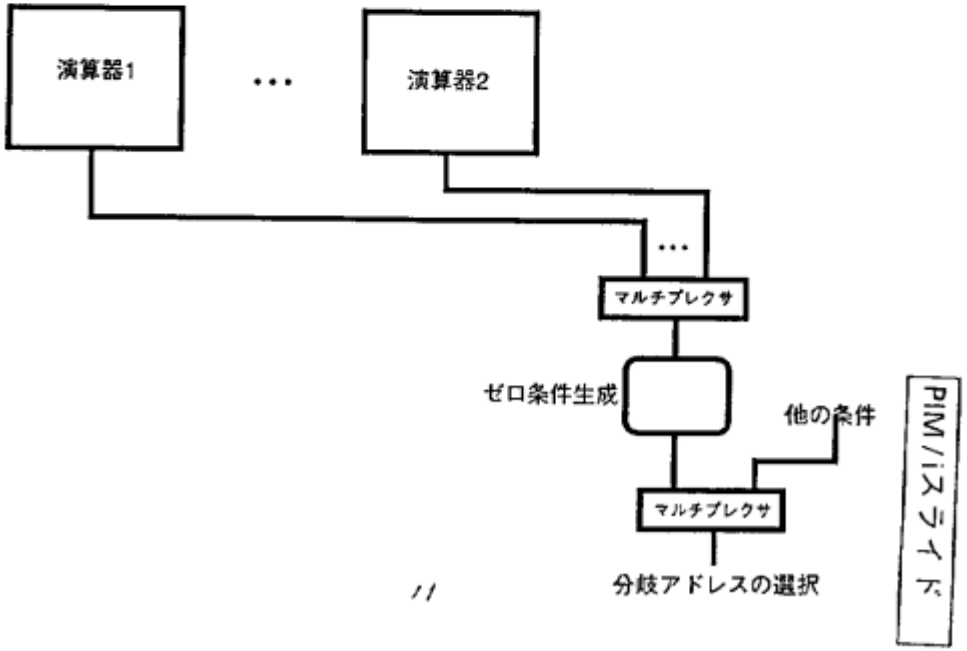
アドレスデコード
　ローカルメモリの実装でクロック周期が延びた
　デコードは以外と時間のかかる処理

診断用ハードウェア，パリティチェックなど

全体が同一クロックで動作
　デバッグはしやすいが
　システム内の最も低速な部分でクロック周期が決まってしまう

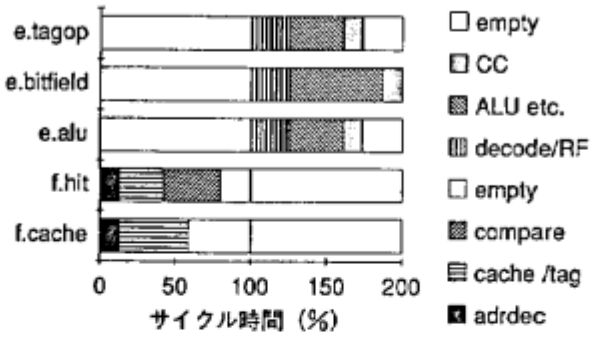
シミュレーション，クリティカルパス解析をしておけばよかった

9



11

PIM/iスライド



12

こうしたかった
こうすればよかった

キャッシュメモリ

スヌープ方式

キャッシュ間転送，更新型プロトコルのために使用した2ポートメモリはコストがかかる

PIMプロトコルはコストパフォーマンスがよくない

サイクル毎の連続書き込み制御

書き放し制御

13

こうしたかった
こうすればよかった

プロセッサ

命令長が40ビットでは命令フィールドの制約から
組み合わせられる操作の制約が強い
- 分岐とタグ操作
- 条件付きサブルーチン分岐
- 条件分岐とメモリアクセス

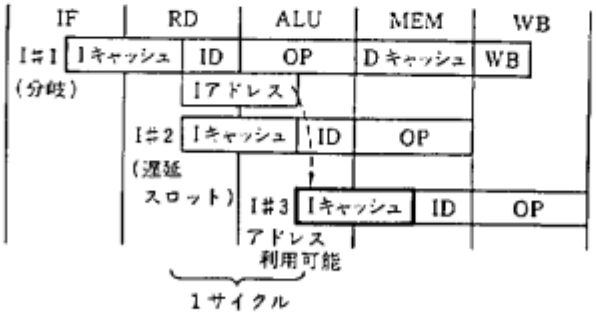
命令もデータも64ビットにするとバイトアドレスマシンとしてすっきりする

ピン数の制約　(当時のCADで208ピン)

code address	30
code	64
data address	30
data	64
合計	188

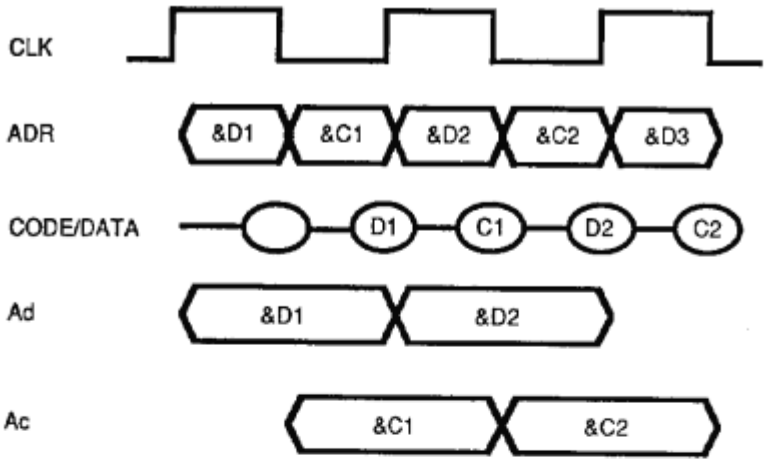
制御信号線を出す余裕がない

14

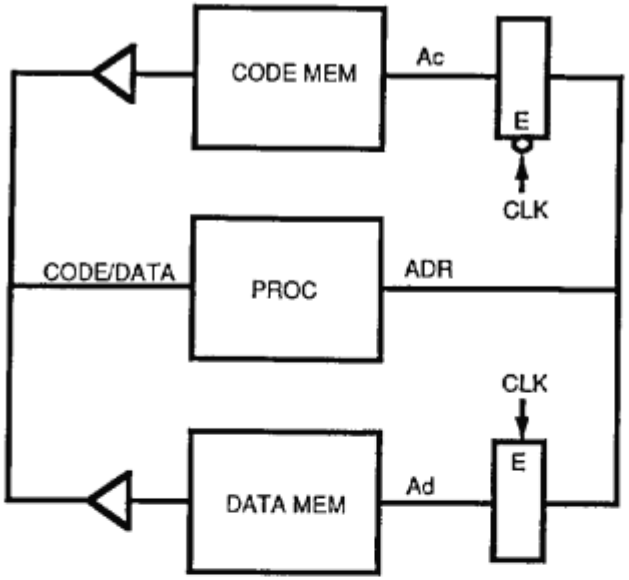


MIPS パイプライン

15



16 クロックの前半後半でそれぞれコードとデータをアクセスする



17

もう一回作るとしたら

on-the-CADマシン

CAD上でシステム全体を構築して十分
シミュレーションをしてからつくる

18

PIM/iスライド

PIM/c

発表者: 朝家 真知子, 今西 祐之, 垂井 俊明 (日立)

日時: 2:45 ~ 3:30, 3/22　　議事録担当: 仲瀬 明彦 (東芝)

1　ハードウェア (垂井)

1. ひとまわり大きなきょう体を使い、両面からボードを入れれば 1 筐体に 256PE 入ったかも知れない。
2. クラスタに 1 つのクラスタコントローラを廃止し、全 PE からクラスタ間ネットワークを出せば、ネットワークスループットが 8 倍になり、クラスタコントローラボトルネックもなかったかも知れない。
3. ネットワークはクラスタ間用に 3 ポート持っているのだから、ハイパーキューブにすれば 8 クラスタつながった (512PE になる) かもしれない。
4. CS を両面実装すれば 64K 語になったかもしれない。

2　処理系 (今西)

1. 条件分岐の時、分岐 jump を行なう命令内に分岐先の処理を入れてしまえば、30% 程度の無駄を防げた。
2. 自動生成された冗長なマイクロプログラムを KL1 で最適化しても面白いかも知れない。

3　アプリケーション (朝家)

1. KL1 のアプリケーションには、1 レベルの負荷分散だけに対応しているものもあり、クラスタ内で負荷分散しないものも多い。→ 2 レベル以上の負荷分散対応したプログラミングを研究したい。

4　質疑応答

Q. VPIM を手で書くオプティマイズは行なっていないか？

A. やってない。泥臭い仕事は全部機械にやらせたかった。その結果処理系が遅くなってしまったが、自動変換でバグが入りにくく PIMOS の立ちあげは早かった。

Q. シミュレータがなかったことなど、何か反省点はあるか？

A. シミュレータは工数的に無理だった。一番の反省点は、CS の容量など種々のサイズをもっと念入りに予測すべきだった。

Q. もし CS が 64kw あれば、20 倍の性能になったか？

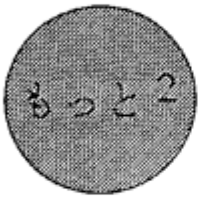
A. 5 ~ 6 倍程度であろう。もっと性能をあげるには、VPIM のサブルーチンの切りわけを見直さねばならない。

初日, リラックスした雰囲気の会場

PIM/cスライド

ファームウェア：こうすればよかった

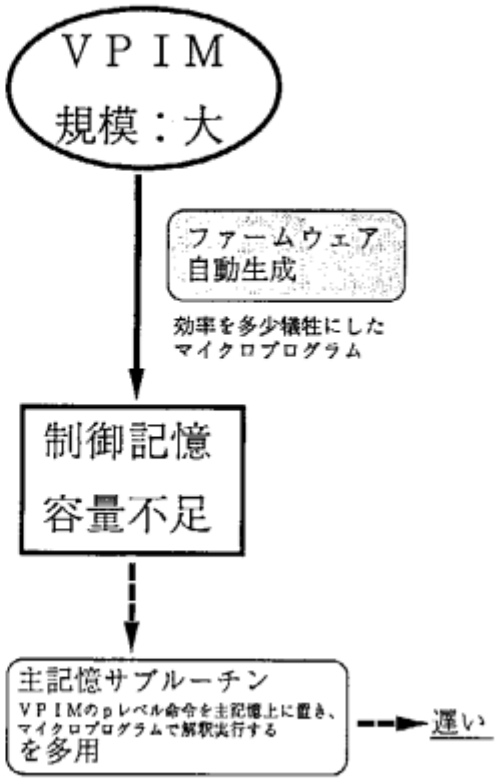
こうすれば もう少し速く なった



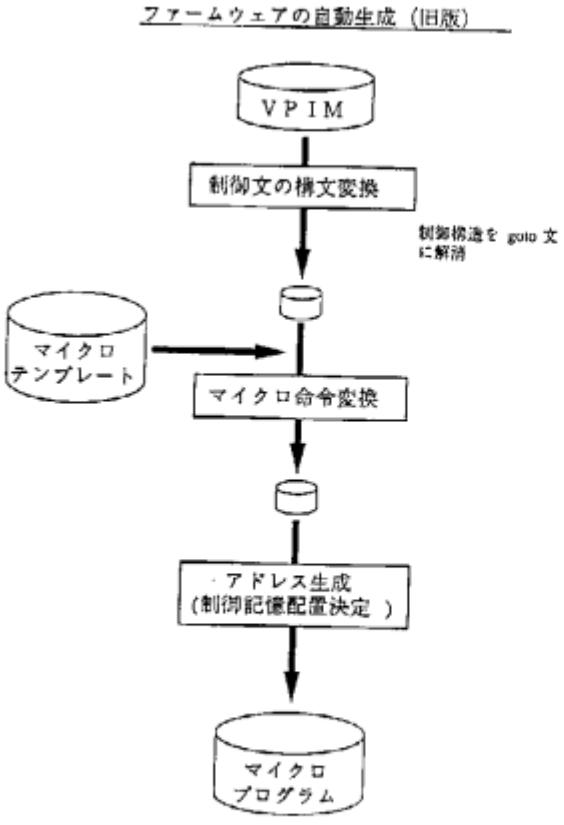
制御記憶を無駄なく使う

マイクロプログラム
自動生成方式

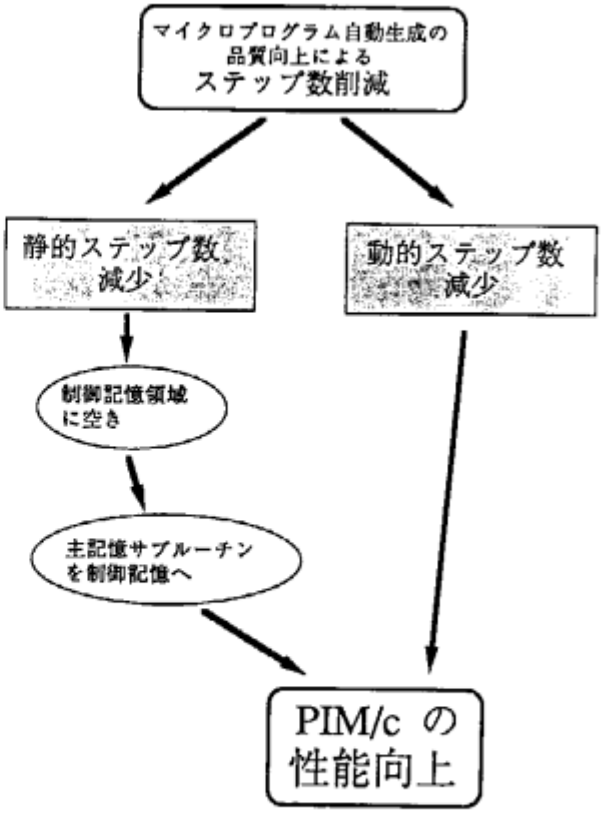
1



2



3



4

PIM/cスライド

非効率的な分岐方式

分岐のテンプレート

IF (条件判定)	
THEN 部へのジャンプだけ（2ステップ）	ELSE 部へのジャンプだけ（2ステップ）
THEN: ＿＿＿＿＿＿ ＿＿＿＿＿＿ ＿＿＿＿＿＿ goto ENDIF	
ELSE: ＿＿＿＿＿＿ ＿＿＿＿＿＿ ＿＿＿＿＿＿ goto ENDIF	
ENDIF: ＿＿＿＿＿＿ ＿＿＿＿＿＿ ＿＿＿＿＿＿	

テンプレートで置き換える

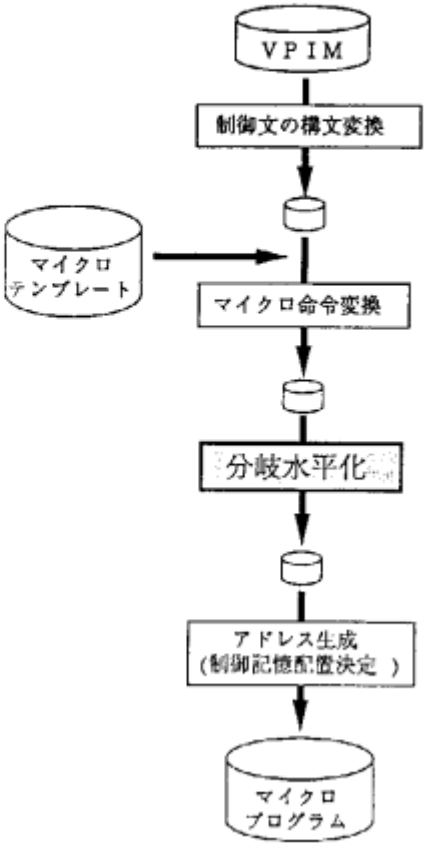
→ 次に実行する命令のアドレスを決めるだけのステップが発生

全体の３割弱

このうち90% 以上はなくせるはず

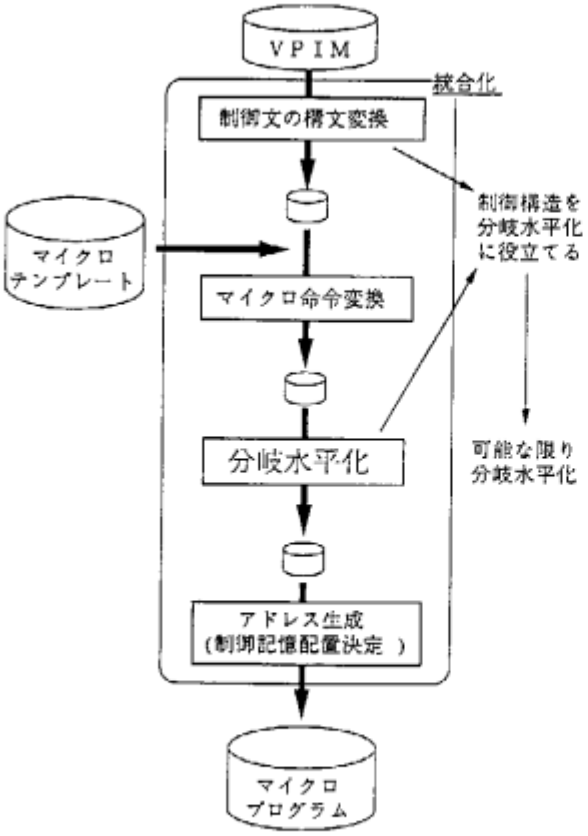
5

ファームウェアの自動生成（現状）



6

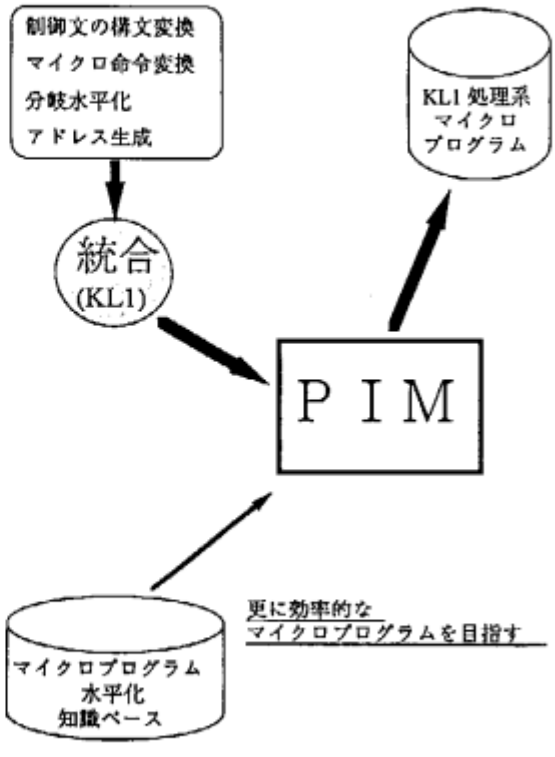
ファームウェアの自動生成（こうすれば…）



7

こうすれば……

分岐水平化率向上を目指す



8

PIM/cスライド

PIMOS: ~~苦労した話~~ 楽をした話

PIMOS を動かし始めて 1週間程度 で
シェル、リスナが立ち上がった

↑

ＫＬ１の移植性の良さ

後悔

ただし、PIMOS動作中にＧＣなど
PIMOS以前の基本的な処理のバグ
で落ちて苦労した
先に処理系のデバッグを充分に行
っておくべきでした

9

今思えばこうするべきだった
- 応用プログラム実装の経験から

○クラスタ間ネットワークの強化

→台数効果アップに最後まで修正必要だった

○クラスタ構成を意識したプログラム手法の提案

→クラスタ内負荷分散

○タイマ機能の具備

→パフォーマンス測定に欲しい

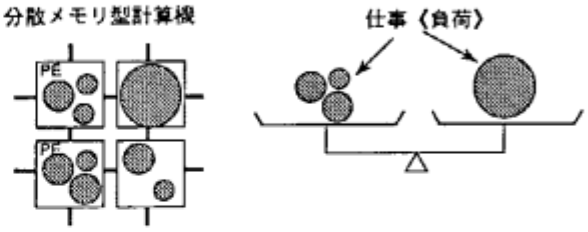
○応用プログラム、基本ソフトウエア、処理系のバージョン合わせ

→移植時の確認事項が大

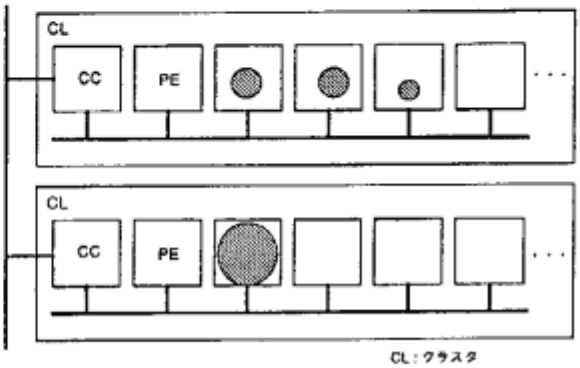
1

階層構造の問題点

再帰ゴールはクラスタ内負荷分散できない。



階層型計算機



CL：クラスタ
PE：プロセッサエレメント
CC：クラスタコントローラ

2

（1）クラスタ内で負荷分散されない例（再帰的ゴール）

```
throw(List,Node,Result):- integer(Node) |
        foo(List,Result)@node(Node).    ... (a)

foo([A|Tail],Result):- integer(A) | foo(Tail,Result).
foo([],Result):- true | Result=done.
otherwise.
foo([_|Tail],Result):- true | foo(Tail,Result).
```

（2）クラスタ内で負荷分散される例

```
throw(V,Node,Result):- integer(Node) |
        foo(V,Result)@node(Node).    ... (b)

foo({A,B,C},Result) :- wait(A),wait(B),wait(C) |    ... (c)
 foo_1(A,A1),
 foo_2(B,A1,B1),
 foo_3(C,C1),
 result(A1,B1,C1,Result).

result(A1,B1,C1,Result):- wait(A1),wait(B1),wait(C1) | Result = done.
```

図　プログラム例

3

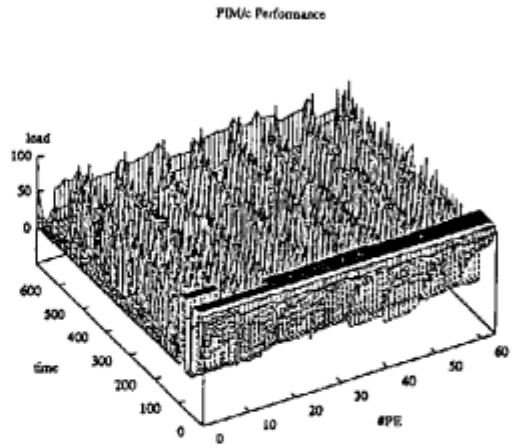


図　クラスタ内で負荷分散されない例
（稼働率グラフ）

4

PIM/cスライド

ＰＩＭ／ｃハードウェアこうすれば良かった

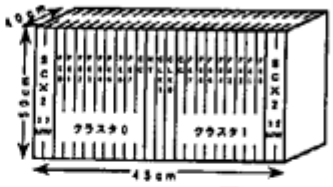
日立　　中央研究所

垂井　　俊明

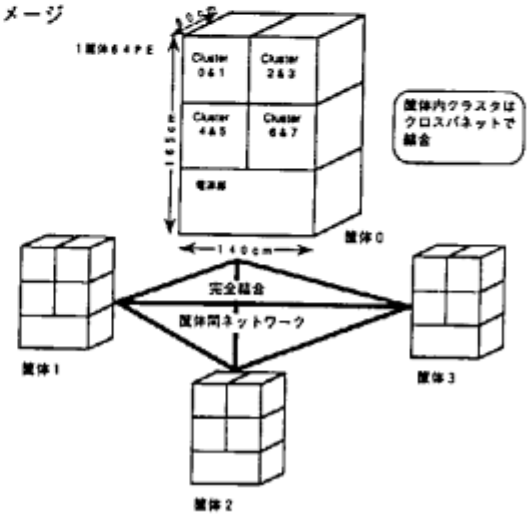
目次

/

マザーボード実装イメージ

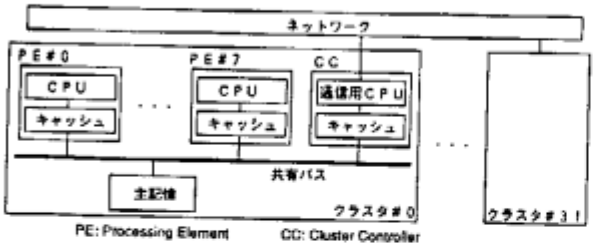


筐体イメージ



2

０．現在のＰＩＭ／ｃのハードウェア

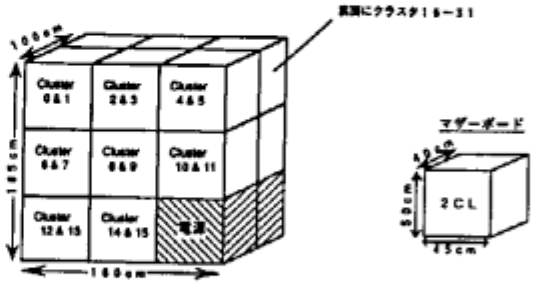


１クラスタ：　８ＰＥ＋１ＣＣ
１筐体　　：　８ＣＬ
システム　：　４筐体（＝２５６ＰＥ）

３

１．こうしておけば１筐体に２５６ＰＥ入ったかもしれない

・一回り大きい筐体を使用する
（１４０×１６５×８０　→　１６０×１８５×１００）
・マザーボードを筐体の両面に配置（現状は片面）
・マザーボードを３×３に配置（現状は２×２）
・電源の実装体積を削減

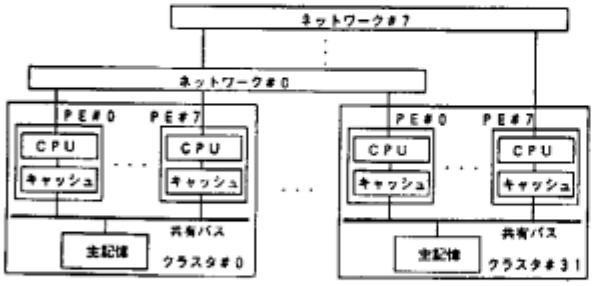


問題点：放熱をどうするか？

4

2．こうしておけばネットワークスループット
8倍になったかもしれない

・クラスタ内の全てのＰＥからネットワークに口を出す
・クラスタ間では同じＰＥ同志をネットワークでつなぐ
（他のクラスタのＰＥ番号が同じＰＥにのみ通信可）
・ＣＣは廃止する



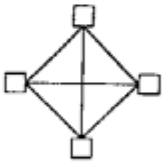
問題点：ＳＣＳＩをどこにつなぐか？
（ＰＩＭ／ｃではＳＣＳＩとネットワーク共存できない）
→ ＳＣＳＩ専用のＰＥを新設する！？

5

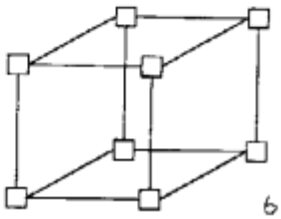
3．こうしておけば５１２ＰＥつなげたかも知れない

ネットワーク用ルータの構成

4筐体完全結合
＝ ２５６ＰＥ



8筐体キューブ状結合
＝ ５１２ＰＥ



6

4．こうしておけばＣＳ容量２倍になったかも知れない

現在のＣＳ容量：３２ｋｗ
（６４ｋ－ＳＲＡＭ６０個片面実装）

容量大幅に不足！！

ＣＳから溢れたコード→主記憶サブルーチンで実行

２０倍おそい！！

ＣＳのみ両面実装していれば：
６４ｋｗ

問題点：基板の配線入ったか？
アドレス／データをドライブできたか？

7

PIM/m

発表者: 近藤 誠一, 中島 克人 (三菱)
日時: 3:30 ~ 4:15, 3/22
議事録担当: 山本 礼己 (日立)

1 ネットワークアーキテクチャの話し (中島 (克))

1. MultiPSI 5 → PIM/m 3.85 何の数字でしょう？
   ネットワークスループットです (MByte/s)。

   PIM/m のネットワークは MultiPSI 程度でよいとして設計したが、クロックの方が、予定 50nsec、実際 65nsec となり遅くなった。

   ネットワークハードおよび KL1 ファームが処理する時間に比べて、ネットワーク上を飛んでいる時間は、MultiPSI では小さかった。PIM/m ではコンパラになってしまった。3.85 は危なかった。

   瀧：コンパラなら良い設計では？
   → 一斉局所ＧＣでリリースメッセージが出まくると本当に危ない。

   中島 (浩)：実際のサイクルあたり送出バイト数は？
   → 忘れました。

2. PSI-II で人がガチガチに書いたマイクロは、2.5 ~ 4 程度の水平度が出ています。静的か動的かは忘れました。

3. KL1 処理系は動的処理が多く、ハードかが大変 (例えば KL1 データの輸出入処理)。
   → 分散共有メモリ (スタンフォードの真似) がいいかも。

   近山：何が嬉しいの？
   → KL1 データ交換が速くなる。

   近山：メッセージ数は増えるでしょう。一括ＧＣは？
   → 私 (中島 (克)) のポジションペーパを見て下さい。

   今井：分散共有メモリでも @node プラグマは必要 (ないと悲惨)。
   データ分散が適切ならば、一括ＧＣもそこそこに走りそう。

2 KL1 処理系の話 (近藤)

1. PIM/m は三菱の３代目です。VPIM を使ってません。経験 (人、ツール、ノウハウ) がものをいってます。現時点での比較評価は、この経験の差が大きいのかも知れません。

2. 先ずは良かったところ：GEVC は良かった。

3. KL0 と KL1
   同じベンチマークについて、KL1 は KL0 の２倍遅かった。MRB、エンキュウ／デキュウなどありますが、もっと頑張れるのではと思っている。

4. MultiPSI と PIM/m
   PIM/m は CPU は速いがメモリアクセスは遅い。ベンチマークの結果を見ると、すぐ原因がわかってしまう。

   たとえば、白いプログラムが速くなっていない。再利用率の高さが、メモリの非連続アクセスを招き、(みかけの？) ワーキングセットを大きくしている。bp100x100 を例にとると、ＧＣ直後と、何度も走らせた後とでは、後者が 50% も遅い。

5. ボディ組込のサスペンド
   プログラマの作法には任せておけない気がしてきた。(年をとったのかな？)

6. ベンチマークプログラム
ICOTにあるKL1プログラムは、めいっぱいMultiPSIチューンニングされている。複数ＰＥの正当な比較評価が出来ない。

MGTPでもPIM/mはMultiPSIほどの台数効果は出なかった。しかしユーザは工夫してしまうのも事実MGTPでは、一斉局所ＧＣを組み込んで性能を出した。

7. PIM/m CPU
PIM/mはKL0チップ。始めからKL1を意識していたらもっと違ったかも知れない。

8. GCの見せ方
アプリケーションレベルのユーザにＧＣをどのように見せるべきか？データの局所性を予測する処理系／ハードを提供する？

9. 特徴を生かすと遅くなる
KL0の特徴であるバックトラック、KL1の特徴であるMRB白、使うと遅くなる。処理系とアーキテクチャのミスマッチングか？

近山：KL1の特徴を知らずにＣ調でコーディングしていた人たちが、KL1の特徴を使うようになって特徴に特有の性能が見えてきただけ。ＣでKL1の特徴を真似ればもっと遅くなる。

瀧：メモリが遅い、キャッシュが小さいからでは？

中島(浩)：KL1ではキャッシュは増やしてもしょうがない。
→ 少なくとも回収再利用はやらない方が良さそう。破壊代入のためにだけMRB管理をするのかな。

中島(浩)：キャッシュサイズを越えたらＧＣすれば良い。要はワーキングセットを小さくすること。

近山：早めのＧＣの嬉しさについては明日話します。

真剣に講演に耳を傾ける聴衆

PIM/mスライド

PIM/m kl1 処理系の開発と評価
「今思えばこうするべきだった」

近藤 誠一 (三菱電機)

/

PSI, PIM/m の歴史

```
kl0              kl1

  PSI-I  (1983)
    ↓          Multi-PSI/v1 (1985)
  PSI-II (1986)      ↓
    ↓          Multi-PSI/v2 (1988)
  PSI/UX (1989)      ↓
                   PIM/m    (1991)
```

開発フェーズ

1. 逐次処理系 (kl0)
2. 逐次マシン上での擬似並列処理系
3. 並列マシン上での並列処理系

2

デバッグ用ツール

- ソフトウェアによるシミュレータ
- マイクロトレース
- マイクロブレーク
- 命令語レベル条件ブレーク

評価用ツール

- GEVC
- processor profiler
  - アイドル時間
  - GEVC
  - エンコードされたメッセージの出現頻度
  - デコードされたメッセージの出現頻度
  - 局所 GC の直前直後の時刻とヒープトップ
  - ユーザ定義イベントログ
  - メッセージの組み立て分解に要した時間
- shoen profiler

3

逐次処理系との性能比較

プログラム		逐次 Prolog(msec) (1)	(2)	KL1(msec)	比
append	1000 * 1000	1,497	858	1,641	1.91
	100 * 10000	1,237	854	1,618	1.89
	30 * 100000	3,585	2,582	4,927	1.91
nrev	1000	611		827	1.35
	100 * 100	660	651	875	1.34
	30 * 1000	646	483	876	1.81
qsort	2 ** 11	158	99	207	2.09
	2 ** 13	824	519	1,488	2.87
	2 ** 15	4,396	2,704	8,758	3.24
lisp interpreter	tarai3	210	173	292	1.69
	fib10	20	14	35	2.50
	reverse30	11	9	27	3.00
nqueen	8	97		163	1.68
	10	2,104		3,337	1.59
	12	68,091		101,453	1.49
pentomino 8*5		45,930	37,605	74,442	1.98

4

Multi-PSI と PIM/m の H/W 比較

	Multi-PSI	PIM/m
制御方式	53 ビット水平型マイクロ	64 ビット水平型マイクロ
制御記憶容量	16 K 語	32 K 語
使用 LSI	CMOS ゲートアレイ 8K/20K ゲート 9 種	CMOS セルベース 0.8/1.0 μ m
マシンサイクル	5 MHz (200ns)	15.4 MHz (65 ns)
パイプライン	命令フェッチのみ	5 段パイプライン
アドレス変換方式	固定テーブル	マイクロによるテーブル入れ替え
キャッシュ	4K 語	命令 1K 語 + データ 4K 語
主記憶アクセス	800 ns	1430 ns
PE 間ネットワーク 転送速度 バッファサイズ	2 次元メッシュ (max 8*8) 同期 10 ビット / チャネル 5 Mbps 4K バイト	2 次元メッシュ (max 16*16) 非同期 10 ビット / チャネル 3.84 Mbps 1K バイト
メモリ	80M バイト (16M 語)	80M バイト (16M 語)
FEP との通信	PE 間ネットワーク	SCSI
内蔵ディスク	なし	8 PE ごとに 670 M バイト
Append 性能	128 KRPS	615 KRPS

5

単体性能

- 組込述語
- 基本制御機能

PIM/x 共通プログラム

プログラム	Multi-PSI msec	Krps	PIM/m msec	Krps	比
life game	18,583	19	8,020	45	2.32
qlay11	35,459	49	15,057	116	2.35
semi	19,579	37	8,192	89	2.39
bestpath100	45,530	62	19,263	129	2.36
tsumego5	6,477	39	2,479	101	2.61
mastermind	23,683	73	7,837	222	3.02
nqueen12	280,690	92	92,483	278	3.04
pascal	5,373	61	2,051	161	2.62
puz15-6	216,075	37	89,173	89	2.42
puz15-6(no susp)	161,368	49	57,994	137	2.78
puzzle	32,702	39	9,901	128	3.30
waltz	15,540	77	4,734	255	3.28
zebra	16,482	25	5,450	74	3.02
pentomino1(8*5)	214,229	40	89,366	95	2.40
pentomino2(8*5)	181,108	45	69,792	121	2.59
pentomino3(8*5)	164,026	62	54,883	135	2.99

6

フリーリストの状況

```
bestpath 100*100
    983 K 語中 656 K 語 (67 %) 回収
    16 語セル 11 K 個

pentomino 8*5
    3,300 K 語中 5.5 K 語回収
    16 語セル 23 個
```

bestpath の連続実行

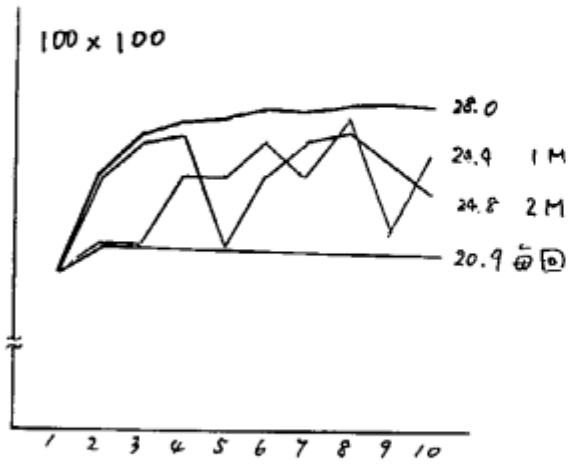
(0) そのまま

(1) ヒープトップ + 1M 語

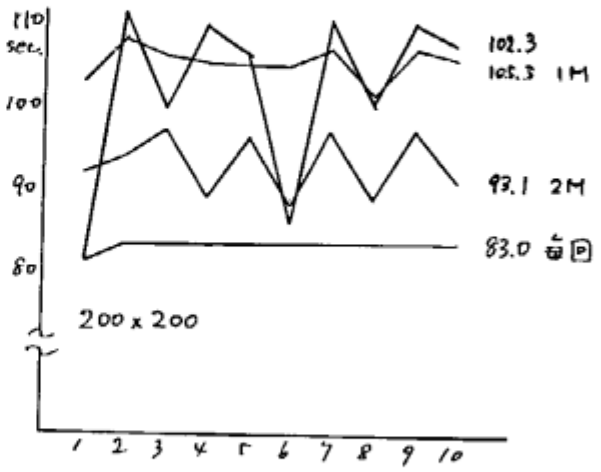
(2) ヒープトップ + 2M 語

(3) 毎回

7



8



9

PIM/m の弱点

- ボディ組込述語のサスペンド
- 乗除演算、浮動小数点数演算
- ガード組込述語

10

複数プロセッサ性能

実は、公正な評価はされていない。

- 手元に multi-PSI, PIM/m にチューニングした kl1 プログラムしかない。
- 当社比

11

PIM/mスライド

PIM/m：今思えば.... (その1)

三菱電機　情報電子研究所　中島 克人

1. はじめに

マルチ PSI　→　PIM/m

5 MB/s　→　3.85 MB/s

項目	マルチ PSI/V2	PIM/m
サイクル時間	200 ns	65 ns
PE間ネットワーク	5 MHz (200ns) 全 PE 同期クロック	3.85 MHz (65ns×4) ↑ (5 MHz (50ns×4)) 全 PE 独立クロック
パケット操作サポートハード	なし	若干 (4 bytes ⇔ 1 bytes 等)

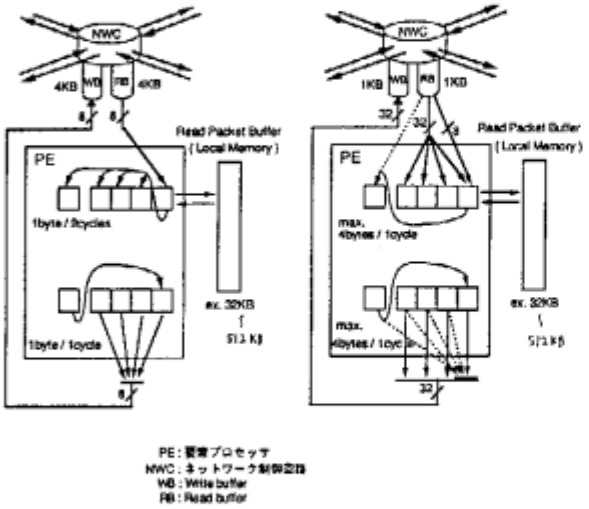
1/12

2. ネットワーク性能について

- アーキテクチャ概観
- パケット・フォーマット
- パケット内データ表現 (共通)
- マルチ PSI の通信 (メッセージ処理) コスト
- マルチ PSI vs. PIM/m 通信コスト比較

2/12

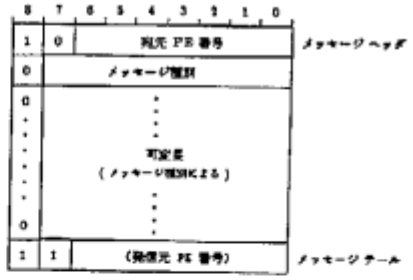
アーキテクチャ概観
（ネットワーク～PEインタフェース部）



PE：要素プロセッサ
NWC：ネットワーク制御回路
WB：Write buffer
RB：Read buffer

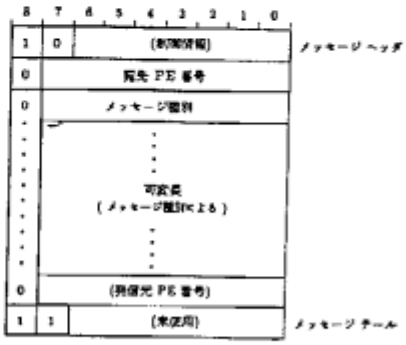
3/12

パケット・フォーマット



□ の部分のみがネットワーク制御回路で解釈され、
他は CPU 側のファームウェアで処理される

マルチ PSI のメッセージパケット・フォーマット



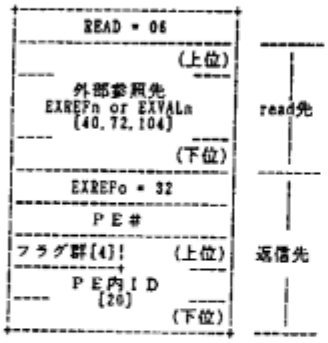
□ の部分のみがネットワーク制御回路で解釈され、
他は CPU 側のファームウェアで処理される

PIM/m のメッセージパケット・フォーマット

4/12

パケット内データ表現(共通)

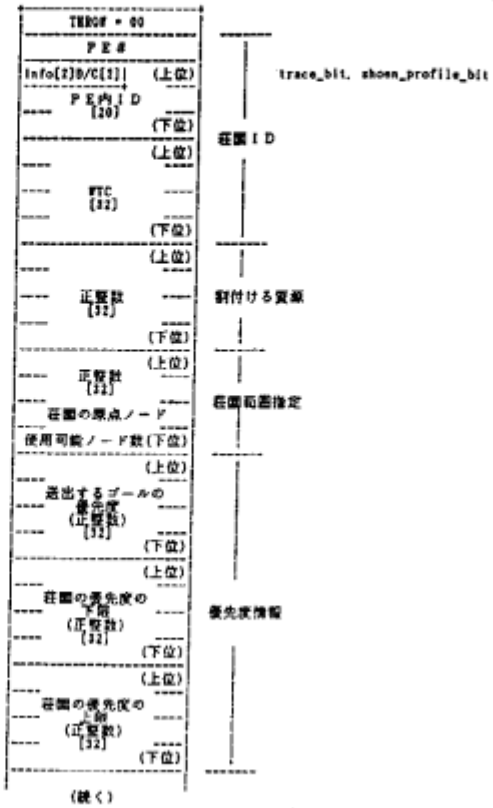
$read(外部参照先, 返信先)



フラグ群[4] = 001x (WECゼロフラグ = OFF
構造体idフラグ = OFF
Safeフラグ = ON)

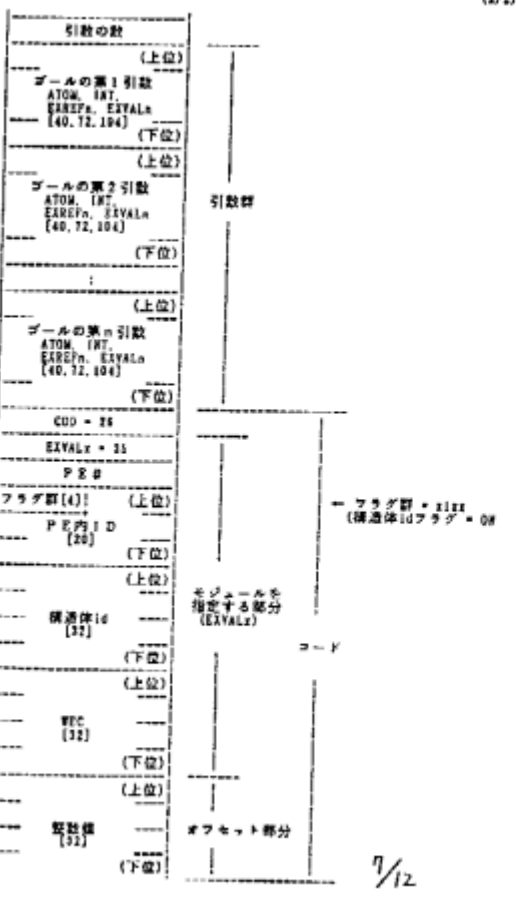
5/12

$throw(荘園ＩＤ, 割付ける資源, 優先度情報, 引数の数, 引数群, コード)
(1/2)



4/12

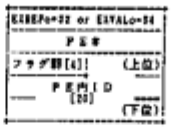
throw(荘園ＩＤ, 割付ける資源, 優先度情報, 引数の数, 引数群, コード)
(2/2)



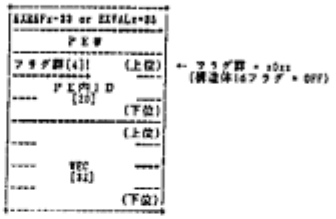
7/12

データの形式

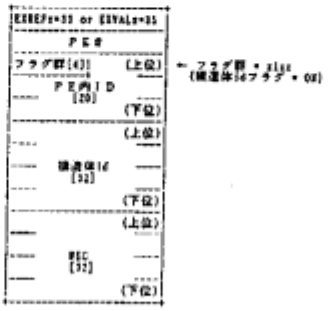
○外部参照ＩＤ



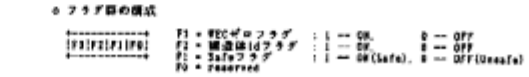
●外部参照ＩＤ (構造体idが付きでない場合)



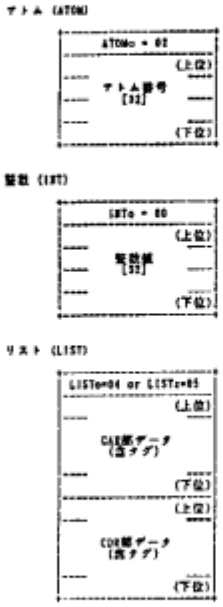
●外部参照ＩＤ (構造体idが付きの場合)



○ フラグ群の構成

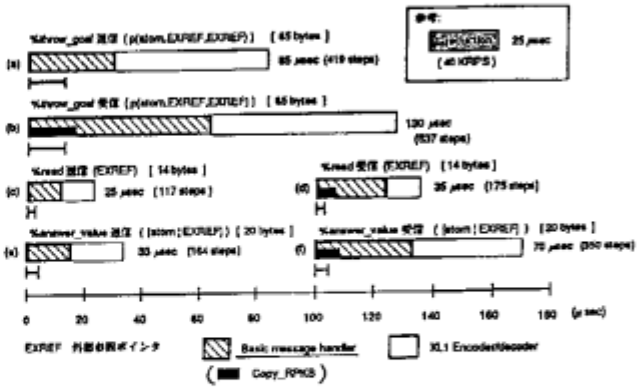


8/12



9/12

マルチPSIの通信(メッセージ処理)コスト



Basic message handler (ハードウェア操作)

- メッセージ・ヘッダの付加
- 翻訳されたデータの 32 ビット→5 ビット変換
- メッセージ・テールの付加

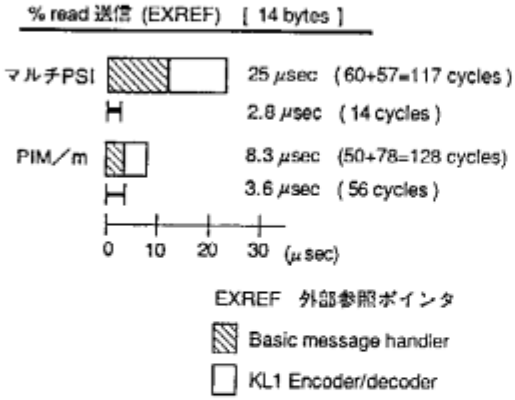
(- Copy_RPKB(受信メッセージのメモリへの一時退避))

KL1 Encoder / decoder (KL1 処理)

- ゴール引数の翻訳(含む、外部参照アドレスへの翻訳)
- 述語属性、実行情報の読出し・翻訳
- コードの翻訳

12/12

マルチPSI vs. PIM／m 通信コスト比較



11/12

3. まとめ

- CPU のメッセージ処理が 1/3 に短縮され、3.85MB/s は結構ギリギリ!!
- それでもメッセージ処理はまだ重い
  - → PIM/m のネットワーク部は タグ・アーキテクチャになっていない。 (現状の分散共有メモリマシンも同様)
  - → 現行処理方式の徹底ハードサポートはかなり困難。
- 分散共有メモリマシンであれば、局所 GC(転出入方式) を諦められるか?

12/12

議事録

KL1 は裸の王様か? ハードウエアとユーザとの間で

発表者: 久門 耕一 (ICOT)
日時: 4:30 ~ 5:00, 3/22
議事録担当: 清原 良三 (ICOT)

1 計算機言語は何故存在するか?

1. 計算機任意の処理を方法を示す。 → assembler?
2. 典型的な記述を簡潔に表現する。 → Fortran の do 文など
3. 操作、記述法に制約をはめスタイルを保つ。 → Pascal など
4. 面倒な管理をユーザから解放する。 → Lisp など

2 KL1 は何向きの言語?

1. 大規模知識処理用言語?
   - 不定形のデータを扱う。不定形のデータを表すと、
     - プロセスグラフで記述すると遅く, デバッグも困難
     - 配列を自分で管理し、ポインタを扱うと並列度はなくなってしまう。
2. 並列処理記述用のアセンブラか?
   - 現状の KL1 には書けないことがある。
   - 破壊書き替えが明に行なえないと共有メモリマシンの努力が現れない。

3 言語仕様か実装方式なのか?

MRB の存在によりこの区別を難しくしている。
現状では、実行時細粒度並列はやらない方がいい。
コンパイラによる実行順序の並べ替えや、逐次化は必須。
通信量の小さい AND 並列ならよい。
ストリーム並列は同期の負担が大きい。

4 KL1 の記述力とは?

VPIM は実行時ルーチンが巨大
論理型言語に囚われず記述プリミティブを入れた方がよかったのでは?

5 まとめ

- 並列計算機で実行するメリットを明確に
- 並列に実行できることはみな記述可能に
- グラフ構造の簡便な表現と高速な実行
- 逐次に実行するメリットを追求
- 論理性にこだわらない記述力

KL1は裸の王様か？

ハードウェアとユーザとの間で

この後のパネルのために

ICOT　第一研究室

久門　耕一

1

言語の持つ意味

・その言語による自然な書き方が望ましい書き方になる

・人間の意図を表わしやすい言語

ところで

・並列計算機での望ましい動きとは
→プログラムの正しさ
→高速実行の保証

・並列計算機上での人間の意図とは
→人間の意図には逐次とか並列とか
あるのか？

3

計算機言語はなぜ存在する

・計算機に任意の処理方法を示す力
→assembler（言語じゃないかも知れない）

・典型的な記述を簡潔に表現する力
→FortranなどのDO文

・操作、記述法に制約をはめスタイルを保つ力
→Pascalなど

・面倒な管理などをユーザから解放する力
→LISPなどのメモリ管理

2

KL1は何向きの言語か

(1) 大規模知識処理用言語である
→不定形のデータを扱う大量の計算を行う
→データは配列などで表わしにくくグラフ構造等を持つ

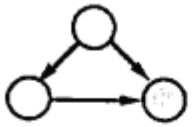
(2) 並列処理記述用のアセンブラである
→基本的な操作もユーザが書く必要がある
→上位言語を落とすために必要な記述力があるか

4

KL1は裸の王様かスライド

不定形のデータを表わすと

・グラフ構造は多重参照データを内包する
→更新することが出来なくなる



多重参照され更新不可

(a) プロセスグラフにより記述
→遅い、デバッグが困難

(b) 配列を自分で管理してポインタを扱う
並列度が全くなくなる
まるで Fortran によるリスト処理のよう

5

細粒度処理と大粒度処理

・現状では実行時細粒度並列はやらない方が良い
→コンパイラなどによる実行順序の並べ換えや逐次化が高速実行に必須

```
p(X):- true | r(Y), X := Y+1.
r(X):- true | X := 0.
```

これは近々逐次実行化されるはず

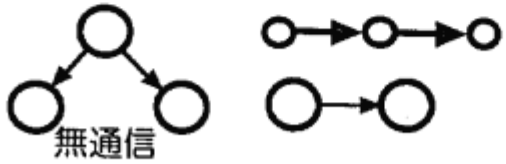
・通信量の少ないAND並列なら良いと思う
→内部が逐次動作の大きなモジュール間の並列実行

6

OR並列とAND並列実行

・OR並列を行うなら多重環境管理をシステムで行うのかユーザが行うのかの違い
でもバックトラック実行とコピーによる実行では計算のオーダが異なる。コピーする構造の大きさと書き換える構造の大きさの違い

・ストリーム並列は同期の負担が大



・今のプログラムで台数効果の出ているのは探索型問題？ユニファイヤは自作する

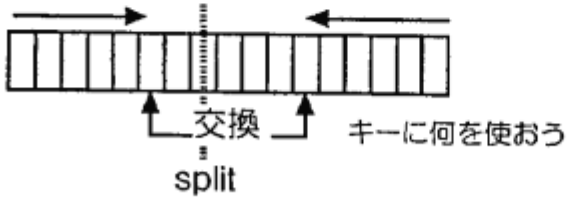
7

KL1は並列処理用アセンブラか

アセンブラにしては複雑な仕様だと思う

・現状のKL1には書けないことがある
→書きにくくてもよいが書けないのは困る

なぜKL1で書くqsortは配列を使わないか？

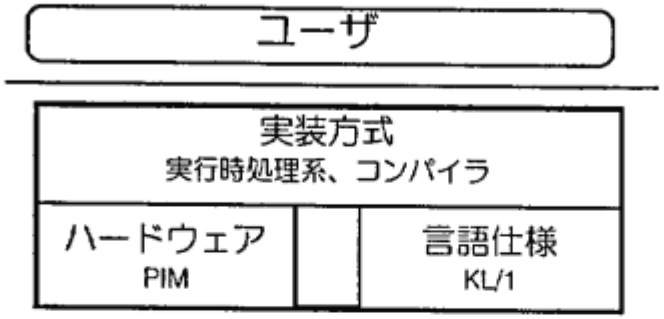


・破壊書換を明に行えないと、共有メモリマシンの努力が現れない（隔靴掻痒の思い）

8

KL1は裸の王様かスライド

言語仕様か実装方式なのか



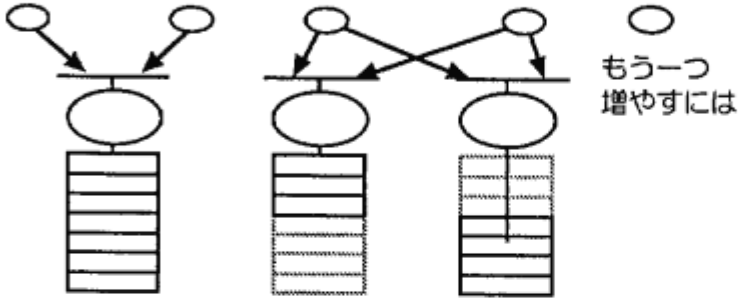
・言語仕様による問題点と実装方式による問題点は
分離して議論するべきだが
MRBの存在はこれらの区別を難しくしている

9

破壊書き換えの手段
プロセス内に白参照配列

・管理プロセスの入り口で逐次化される

並列度を稼ぐためには管理プロセスを複数用意



10

KL1の記述力とは

・VPIMは実行時ルーチンが巨大
→PIMOSサポート用の組み込み述語
KL1では速度、記述力の点で問題

論理型言語に囚われず記述プリミティブを
入れたほうが良かったのではないか？

・FGHCの枠内でメタインタプリタを書く

サスペンドと受動部単一化をどう記述するか

・OR待ちと共有データの更新とは同程度危険

11

まとめにならないですが

・並列計算機で実行するメリットを明確に

・並列に実行できる事はみな記述可能に

・グラフ構造の簡便な表現と高速な実行

・逐次に実行するメリットを追及

・論理性にこだわらない記述力

12

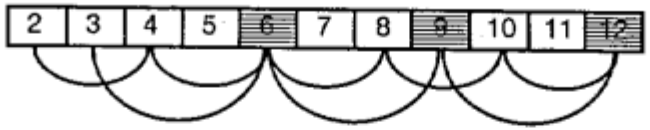
KL1は裸の王様かスライド

多重参照配列とプロセス

- 多重参照される配列をプロセスに閉じ込め
  破壊書き換えのように見せる
- 配列が単一参照になるがプロセスの入り口が
  複数から参照したいのでマージャを入れる
- n個独立に参照するときには参照数を増やす
  度にn個のマージャ口を増やす必要がある
- 配列の各要素を独立に更新する様な
  プロセスが頻繁に生成される場合には無理

13

エラトステネスの篩の場合



この様に複数のプロセスが１つのデータ構造に
対してアクセスを行う形の並列実行は書けない

14

KL1は裸の王様がスライド

パネル討論 1: 並列処理のための言語とはどうあるべきか

司会: 平田
パネリスト: 雨宮, 上田, 木村, 関口, 村上
日時: 5:00 ～ 6:20

【司会】
……パネリストの方々は皆さん良くご存知だと思いますので、今さらご紹介しなくても大体いいと思います。どんな感じでポジショニングを期待しているかというと、上田さんはもちろんGHCというのと、雨宮先生は関数型言語、データフロー、関口さんはHPFというか、ハイ・パフォーマンス・フォートラン関係、木村さんは元KL1教、現在、改宗、アンチKL1ということで、村上先生から並列計算機をコントロールする並列言語という感じで行こうかと思います。並んでいる順に、上田さんからお願いします。

【上田】
3月に出たばっかりのCACMの記事をポジション・ペーパーにそのまま使わせていただきます。ちゃんとまとまった資料ではないんですが、むしろ私が持ってきたOHPでしゃべるよりは、今の久門さんのOHPをそのまま使ってやったほうがいいような気もするので、後でお借りします。

結局、何を10年間かけてやってきたんだろうかということをまとめてみたんですけれども、大体、こんなことだろうと思います。いろいろ、論理型に無いものを入れろとかいう話もありましたけれども、それはやらなかったんですね。あくまでも今までにない単純な並列・並行言語を作ろうということにこだわったわけです。そういう、いろいろつまらないこだわりを続けたことによって、例えば並列と並行を分離するとか、幾つかのこの分野における概念整理に貢献できたということがあります。

それから、多分、このWGではあんまり注意されないことかもしれませんけれども、もうちょっと理論のWGにいくと、並行プログラミングにおける新しい理論研究の触発ということが実際にあります。それは、例えばコンカレント・コンストレイント・プログラミングなんかを指しているわけですね。

ということで、単にプログラムを変えて速く走らせるというだけじゃなくて、新しい計算モデルとしての価値が生まれてきたとか、それからこういう単純な言語にしておくことによって、例えばプログラム解析などの方法、形式的な技法の開発が促進されたとかいうメリットが実際に生まれていると思います。

それから、単純であることによって、単にこの言語でいろいろものが書かれたという以外に、それがほかの例えば並列・並行言語とどういう関係があるかということが、きちんとわかってきているというメリットが出てきたと思います。

今、このパネルは並列言語としてどういうものがいいかという話なんで、あとは処理系作成とか、もうちょっと技業の話になりますけれども、同時にこの関連でいくと、並列記号処理系の作成技法としては、ほかのコミュニティと比べても、おもしろい技術がいろいろ出てきていると思います。例えば、それがメモリ管理とか、分散実行でありますし、コンパイル技術でありますし、それから負荷分散のことを随分考えてきたということもあると思います。

それから、プログラミングという観点から見ても、いろいろパラダイムは生まれているし、並列探索というのがまさに知識処理言語としてではなく、非常にローレベルの言語の上で探索をどうやるかということがいろいろ考えられてきた。それはそれで役に立ったと思います。それから、負荷分散もいろいろ考えられてきた。

それから、一番大きいのは、たくさんのお金と人が注ぎ込まれて、これは正しいかどうかわかりませんけれども、およそ100万行の並列ソフトや、KL1で書かれたものが蓄積されたということで、全体としては随分遅々として進まないようにも見えるし、まだ不満もあるように見えるんですけれども、一応、計算機科学における並行・並列記号処理における新しい一つの分野がようやく生まれてきて、うまくいけばもっと発展するし、それでなくても一応、ある程度の蓄積ができたという気がします。

私が、ほかのパラダイムと比べてどのぐらいのところまで行って、どこが至らないかをどう見ているかというのがこの表なんですけれども、ちょっとずうずうしいかもしれません。並列Lisp・データフローの世の中のコミュニティのレベルには十分達したと思っています。例えば、並列処理系作成技法とか、ソフトへの蓄積とか、言語設計に対する考察とかいう点では、このプロジェクトは随分頑張ったと思います。

ですが、さっき久門氏も言われたように、ただ、言語全体としては、例えば並列Lispはより汚いものを残していて、それで何とかあるものは書けるとかいうメリットがあるわけですね。そういう言語全体として、つまり一番きれいな部分じゃなくて、全体としての記述能力というのはもうちょっと、こういう言語族とも比べてみる必要があろうと思います。

それから、Lindaというのがありますけれども、これも例えば形式理論との結びつきとか、計算モデル的考察とかいう点では、このプロジェクトでやったことは非常に進んだと思います。だけど、Lindaで何としても勉強になったのは、あの宣伝普及の激しさですね。それから、並列オブジェクト指向、これも似たようなことが言えると思います。並列オブジェクト指向のほうが少し、ソフト・エンジニアリングを気にしていたんですね。我々は、もうちょっと下のレベルの計算モデルを気にしていたという差が出ているような気がします。ですが、そのこともあって、記述能力という点からいくと、まだ、いろいろ勉強する点があるし、例えばリフレクションの研究なんていうのも随分進んでいるように見えます。それから、あと、オブジェクト指向の人々を見ていて勉強になるのは、あそこは文化を築くことが上手いということがあります。

それから、並列手続型言語、HPFとか、あるいは最近は他のいろんな手続型言語にメッセージ・パッシングのインタフェースを入れるとかありますけれども、そういうことから見ると、言語的には初めから並列をベースにして設計したということもあって、並行と並列の分離というのは、我々がやったことの一番大きなメリットだと思います。HPFの中でも、完全にそれがすっきりいっているようには見えない。

ですが、一方、こういう言語は効率を非常に気にしてつくられていますから、例えばHPFに入っているようなマッピングの指定の仕方なんていうのは、非常に勉強になるところがあるような気もするし、特にKL1はベクタ周りが弱いですから、今後、そのベクタ周りを並列化するときには非常に参考になると思います。それから、言語としての実行効率という点でも参考にすべきかと思います。

この辺が割と実用にもなりそうな言語でして、一番下の一段は並行処理の計算モデルとの比較という点があるわけです。それはなぜかというと、GHCやKL1というのは、単にプログラムを書いて走らせるというだけじゃなくて、一応、計算モデルとしての位置づけも重要だろうと思うからなんですけれども、そうやってほかのパラダイムと比較してみると、動的な処理、つまりプロセスをつくるとか、データをダイナミックにつくるとかいうことの処理が気持ちよく書けるという点では、一歩出たと思います。

それから、ほかの言語は一応、計算モデルであって、あるものはそこから言語にもなろうとしているんですけれども、言語として見たときの実用性は、KL1は、ほかのこういう計算モデルベースの言語に比べて少し進んでいるかと思います。

ですが、一方、学ぶべき点というのは、理論的成果を蓄積することとか、仕様記述言語としてGHC、KL1自体が使えるかということは、これからも少し検討しなければいけないと思います。

ということでOHPはあと1枚。でも、久門さんの問いかけに答えるのがちょっと書いてあるので、これをやっておきましょう。そのCACMの記事にも書いておいた部分というのが、何とその記事の中では大々的にワンセンテンスが抜き書きされて、大きな活字で組まれてしまったんですけれども、ロジック・プログラミングへのこだわりというのは、私がこだわったというのはICOTが最初からこだわったという歴史があるわけですけれども、それなしに、どんな言語でもいいから並列言語をつくれと言われて、この2つが出てきたと私は思っていません。

というのは、まず、そのこだわりがある程度強い設計指針を与えたということもありますし、有用な概念を受け継ぐことができたということもありますし、一番大事だと思うのは、その論理プログラミングと並行論理プログラミングはどう違うかということを前期どろにやった、いわゆる、言ってみれば路線闘争ですね。そういうものを通じて随分、みんな考えたと思います。そういう考えたことによって、どういう違いがあるか。互いに互いを認め合うという関係にあるんだと思いますけれども、そういうことが概念整理に有効であった。それから、既存の並行・並列処理の方法論に頼らなかったため、つまり何かあるものに頼らないで、全部、結局作ってしまったということで、作った人は少なくともよく考えたわけです。プログラマの皆が考えた結果がうまく移管できているかというと、それはまだ疑問かもしれませんが、ともかく、いろんなことを実はこの中で考えたわけです。

パネリストの一人, 上田氏の発表

今後の課題というのはこの辺に書いてある、割と当たり前なことですけれども、もうちょっとブレークダウンします。言語処理系じゃなくて言語だけの話としては、あと例えば3年か4年の間に何をやりたいかと思っているかといいますと、きちんとスタティックにモードづけされたFLAT GHCやKL1というもので、実際に大規模な、ある

いはシステム・プログラムが書けるか。それから、これはさっきのMRBの話ですね。白黒をきちんと意識してプログラムを書けるような言語にして、しかもそれを処理系なり環境なりが支援して、しかも意識したものは、処理系が非常に速くインプリメントをするということですね。

それから、もう一つは、どうしても並列に速く走らせるためにプログラムのロジックを書きかえるということが、まだまだ行われているようなのですが、それを初期の趣旨に合ったように完全に分離することができるほど、マッピングのほうの機能自体を充実させたい。それから、いろいろ書いてみると、確かにベクタ周りは弱いですね。私も、今年度の後期は遺伝子のプログラムを書いていたんですけれども、数値計算をやろうと思っても全然だめです。そこはどうにかしたいと思います。それから、書きやすいという点からいうと、例えばマクロ機能なんていうのは、もうちょっとうまく入りそうな気もするので、そういうこともやれば、エンジニアリング的にも結構使える言語に持っていくことは、うまくやればできるような気がします。

【雨宮】

私は、関数型データフローという立場で話をしますけれども、実はあんまり具体的な話をするよりも、極めて抽象的な感じで話をしたほうが、まずはいいのではないかと思います。

今、私の問題意識は実証あるのみと思っています。特に関数型というのはラムダ計算とか、項書き換えとか、いろいろ理論レベルの研究をかなりやっているんですけれども、その一方の批判は、それでどれだけ実用性があるのかどうかという、その一点に尽きると思うんですね。関数型でというのは、私はもう一方でデータフローとの相性がいいということは、多分、皆さん、漠然と認めるようになっているんですが、実際にちゃんと証明をしていく。証明というのは実証ですね。そのためには、実証しかないと思っているんです。言語を作って、それを実際のこういうプロセッサのマシン上で、こういう環境を作って、例えばデータ並列で数値計算、科学技術計算、FORTRANでやるのと遜色ないものを示す。実際、MITのグループ、Arvindのグループとかは、そういう風なことをやって、絶対値じゃないですけれども、可能性もかなり示していると思うんですね。それに比べて我々は、まだヂンタラヂンタラしていて、非常にいらいらしているんですけれども、環境が整わないというもどかしさがあります。

最近、並列だとか超並列だとかというキーワードがやたらと出てくるんですが、私はこう思うんですね。並列とか、超並列とかというのは、我々コンピュータ・アーキテクチャをやっている情報処理屋にとってありがたい言葉であって、その外から見ると、並列だろうが何だろうがそんなことはどうでもいいんだというふうに思うんです。並列、並列と偉そうに言っているのは、極端に言えば要するに専門家寄りの立場じゃないかと思うわけです。つまり、ユーザ側から見れば、解きたい問題なり処理をいかに速くやってくれるかということと、それを解くためのプログラムというか問題の記述をいかに易しく直感的に書けるかということと、もう一つは、そういう装置、システムが安く手に入るか、この3つだと思うんですね。この3つを満たせば、並列だろうが超並列だろうが、何でもいいとは思うんです。

でも、多分、速くするためには逐次的にやっているよりも、今の素子テクノロジ、ハードウエア・テクノロジの環境からすれば、たくさんのプロセッサをパラレルに動かしたほうが速くなるというのは、直感的にどなたもわかるわけですね。それが多分、並列といって、専門外の他の分野の人も非常に期待を寄せるところだろうと思うんです。安くなるというのは、多分、デバイス・テクノロジとか、その辺の進歩のおかげだと思います。

残るは、この真ん中の易しいということですね。五世代のプロジェクトの大きな柱、目的の一つには、易しい、だれにもできるということで、知識処理というのがそのキーワードとしてあったと思うんですが、いかにわかりやすくプログラムできるか、処理を記述できるかということにあると思うんです。速くして並列にするために易しさを犠牲にしたのでは、何にもならないということだと思います。

それで、いろいろやってきて、これは関数型とかのコミュニティでもどこまでそれが実際に認識されてきているかどうかは知らないんですが、少なくともICOTのPIMグループでは、ある時期からこのようなことが言われ出したと思います。「並、逐、超越」です。要するに、並列処理、逐次処理というのを、超越したところでプログラミングを考えたらいい。さっき久門さんが、人間は並列的に考えるか、逐次に考えるのがいいのかという、別にこれは対立概念として区別する必要はない。そういうところから、プログラミング言語を考える。その一つのあらわれが関数型、つまり方程式を記述する。関係を記述するというところまでは行っていないんですが、少なくとも関数。だから、入力パラメータに対して、それのバリューが出てくるという関係。入力と出力の間の関係を書くという意味では、そこには処理の構造が並列であるか逐次であるかということは、何も考えなくていいということで、まずこれが重要だろう。

それから、粒度無依存という話です。これは実際の処理系というか、マシンの方に関わってくると思うんですが、やっぱり速くして、通信のオーバーヘッドなどを無くそうとすれば、なるべく大きな粒にして高速レーンにしろと言うんですが、ユーザ側から見れば、問題を自然に書くためには、それがプロセスあるいは並列の単位として処理されるときに、それがどのぐらいの大きさになるかということはあんまりというか、全然気にさせてはいけないということ

とになる。

そうして、このアーキテクチャと横に書いていますが、そこに関わってくると思うんです。結局、これをずっと突き詰めていくと、並、逐、超越と粒度無依存。粒度無依存ということは、細粒度でもプロセッサやアーキテクチャがカバーできるポテンシャリティを持っていないといけないということになるわけです。そうすると出てくるところは、多分これはプロセスというより、もっと細かいレベルのスレッドで、逐次処理実行部分の長さという問題。最近、マルチスレッディングという言葉がよく使われるようになりましたけれども、ウェイトをするのではなくて、一回走ると、そこですぱんとスレッドが切れて、次に駆動されるまでそのスレッドはそこで処理は終わる。別名スプリットフェーズと言われていますが、その機構を速くできるか。

それから、資源が別々になればどこかに要求を出してそれが返ってくるまで当然時間がかかるわけですから、その間、他の仕事がやれるような仕組みにしないといけない。これはレイテンシ・トレラブル。レイテンシがかかっても、それに対して一向に性能が低下しないような方式ということです。

それから、並列で資源が分散化すると、当然通信というのはつきものですから、この通信を中心に据えた計算というか、通信ベースの計算というのが非常に重要になると思うわけです。だから、結局、突き詰めていくと、この3つがキーポイントではないかと思います。

さらに、これが一番プリミティブなところの要素だとすると、それらを組み合わせてシステムを組み上げる。これはレイテンシ。返ってくるのがどうしても遅くなるから、その間、ほかの仕事ができなければならない。だから、これはスループットを上げるような話。こっちはターンアラウンドを短くする話。負荷制御は当然ですね。これはなかなか難しいんですが、この負荷制御というのをやらないと負荷がアンバランスになったりしたらだめです。その時、プロセス、あるいはデータの資源へのマッピングをどうするかという話。 それで、ここが静か動かという問題があります。負荷制御にしても、静的負荷制御と動的という話があります。それから、マッピングにしても静にするか動にするか。データフローというのはある時期、今から思うとある意味で間違った方向というか、かなりダイナミック性をアーキテクチャレベルで気にしていた。ところが、ダイナミックな構造というものは、実行中のところにはなるべく持ち込まないで、できるだけコンパイル中にスケジュールしてしまうということが実は最も大事な話であって、アーキテクチャレベルでは、そのダイナミック性をサポートするようなことはしないほうがいい、何もしないほうがいい。これは一時期、ファームや、あるいはOS360から進んできた、OSを指向したアーキテクチャというのが、ダウンサイジングなどによって立場を失ってしまったのと同じように、超並列においてもあんまりそういうことを考えない非常にシンプルな構成。ネットワーク構成にしてもそうですし、プロセッサにしてもそうですし、メモリにしてもそうです。ここのところの知恵ではないかという風に思います。

関数型云々というより、もっとベースのレベルで話をしました。

【関口】

雨宮先生と同じく、本職としてはデータフローの方をやっているんですけれども、今日頂いたテーマは、ここに来ておられる方全員を敵に回すような立場で、並列処理のための言語とはどうあるべきかというタイトルに対しまして、FORTRAN 擁護派宣言をここでやってみたいと考えております。FORTRANという言葉といいますか、FORTRANというキーワードが、今日の発表の中でも何回か出てきていますけれども、皆さん、これだけはご存じでしょう。「FORTRANはこんなに素敵」ということで、実際、事実上の標準言語となっています。現在、商用の並列マシンというのは幾つか出ておりますけれども、そういう所では、FORTRANがないような物は売れないということで、事実上、標準言語になっている。それから、従来型のプログラムといいますか、これまでずっと培って来たプログラミングスタイルが、そのまま連続してFORTRANの上に実現できます。FORTRANで今まで書いてきたから連続性があるのは当然です。これはもちろん人に寄りますが、非常に親しみのある記述ができる。このような特典がある。それから、最後は、上田さんの方からもご指摘がありましたように、非常に効率のよいプログラミングがこれまで蓄積できていたということで、非常に高品質なライブラリを継承することができる。このように非常に良い事があるわけです。

このような従来のFORTRANに対しまして、さらに最近ではFORTRAN最前線ということで、High Performance FORTRAN (HPF) というものが出てきていますが、これでなぜ悪いのでしょう。非常にデータパラレル的な記述によって何でもかんでも並列にできる。とにかく実行したいものは簡単に並列にできるというのが重要でして、このことを特にMPP、モンキー・パラレル・プロセッシング、サルでも書ける並列処理という風にHPFのほうでは呼んではいませんけれども、そのような並列処理ができる。

さらに、forallというような記述によってここでも並列化ができます。このHPFのスタンダードではないんですけれども、work sharingという方法によりまして、実際にデータの存在する場所でちゃんとプロセスが起動できるような記述も可能になる方向に、今進んでおります。

それでも皆さんの方からFORTRANのこういう所が悪いんじゃないかということで、ご指摘があるのは、非常

に regular な形状に関してはFORTRAN の array syntax というものはいいんですけれども、irregular なものに対しては対応していないんじゃないか。それから、今、言いましたように、forall やメッセージ・パッシング・ライブラリなんかを使いまして、並列性を明示しないといけないのではないか。しかし、こういう書き方をしているとバグが入ってしまうのではないか。ちゃんと正しいプログラムが動かないのではないかというようなご指摘とか、これは感性の問題ですけれども、汚いというような話があると思います。

ところが、皆さんのご指摘に対しまして、FORTRAN はそんなことないよと。実は超並列の世界になってきますと、これは私が断言するようなことじゃないんですが、結合網というのはだんだん単純になってきます。ついこの間まではハイパーキューブ、ハイパーキューブと言っていましたけれども、最近、ハイパーキューブというものは出て来ない。せいぜい2次元か3次元の mesh になっています。それから、自動並列化コンパイラというものも全くの夢ではない。というのは、できることだけやってやれというふうな制限を与えることによって実現できます。最後に、通信と同期のバグに関して、本質というか、これらのメカニズムを実際どういう風に実装するのかということは、KL1 やデータフローのこれまでのアイデアに学んでやりましょうと。

さらに、こんな風な守りに入っていてはいけないので、FORTRAN の方から逆襲しますと「いろんなことが書ける」他の言語でも確かに書けるわけですけれども、例えばループというものを書きたいといった時に、例えば tail recursion で書けるじゃないかという反論もありましょうが、こんなのじゃ嫌だ、ちゃんとしたループが書きたい。それから、並列処理をした場合には、結果の回収といいますか、例えば reduction 等のオペレーションがありますけれども、そのようなものを書かざるを得ない。こういうのはいろんなスレッドで走っているものをバッーと集めてくるわけですけれども、そういうところではなかなか関数性がない。いや、これは関数性があるんだというふうな反論ももちろんありまして、例えば Id なんかでは特別扱いをしています。KL1 はどうなんでしょうか。僕は、その辺は勉強していないんです。それから、Valid はどうなんでしょうか。このような reduction のオペレーションに関して、ちゃんと関数性を保ったままできているんでしょうか。 このようなことを考えていきますと、例えば内積なんかの演算を考えまして、配列をそれぞれ幾つかの部分に分けておきます。内積じゃなくて、普通の単純な足し算でもいいんですけれども、それぞれのスレッドで部分和を計算します。その計算した部分和を最後に全体で回収して一個の値を出すわけです。そういう処理は一般にリダクションと呼んでいまして、回収というのは一つの例として結果を全体から集めてくるという意味で回収という言葉を使っていますけれども、リダクションというのが普通じゃないかなと思っています。

ヒストグラムの方に関しまして、これも似たようなものです。先ほど久門さんの方からありました配列の多重参照を許さないという話ですが、それと似ていまして、とにかくヒストグラムですから、ある場所の要素、データが幾つかに散らばっているわけです。データをある領域のところにポイと落としていくわけです。その時、どこの領域に何個来るかというのは実行時にならないとわからない。そういうような演算を並列にやろうと思ったときに、一番単純な話は全部逐次化してしまう。逐次化して、そこへポロポロ一個ずつ順番に落としていくという方法もありますし、もう一つは、それぞれのリージョンに対してそれぞれで並列化していく。とにかく、update のところで全部 critical region になりますので、そこの部分をうまくやる方法があります。例えば Id というような言語ですと、アカウントという特別な関数を使って、関数性を失っているような所を字面上、隠してしまうというような特別扱いをやっております。 このように、いろいろきれいな言語だと言っているような時でも、実際、こういうことをやろうと思ったときには、全部化けの皮がはげていまして、結局、アセンブラと同じようなレベルに落ちているのではないでしょうか。

ただ私は High Performance FORTRAN だけではあまりいいとは思ってなくて、これでどうだということで、並列性の規律というものは、あまりコーザに開放しない。開放するのは逐次化だけ、つまりここととこの同期をとってやりなさい、ここととこの順番をつけてやりなさいというような規律だけを許す。それ以外のところは、システムの方で分かる限りの並列性を出してやろう。もちろん、それはターゲットのマシンによりますけども、fine grain のものが出せるのであれば、そこまで出して、そうじゃないものは分かる限りのことをやってやる。

それから、バグの入り込み易いメッセージパッシングというものは、ライブラリとしても禁止する。こういうことはやっちゃいかん。あまり explicit に同期や通信を書かないようにする。その代わりに、これは Linda の言葉かもしれませんけれども、distributed shared object みたいなのを考えまして、そこにすべての通信と同期というものを封じ込めてしまう。そこで、データフローの言葉で言いますと、I-structure に近いような構造体メモリを使うことで、先ほどの配列の多重参照などを解決している。

以上のような FORTRAN の今後の拡張みたいなのを希望しまして、これは久門さんのチャートですけども、この2つに合わせまして、並列に実行できるところは、要素レベルの並列性まで全部、記述可能にしてやる。当然、これは高効率なので速くなります。先ほどの配列の話に関しましては、I structure みたいなものを拡張して導入してやる。逐次記述というものを導入することによって、このようなメリットを追及してやる。当然、論理性にはこだわっていないという、私のフォートラン擁護ということを終わらせてもらいたい。

【木村】
富士通研の木村です。

さっき平田さんのほうから、かつてICOTでKL1の仕事をやっていまして、現在は別なことをやっていて、改宗したということで紹介がありましたが、ということは、私に期待されているのは、なぜ改宗したかということをここで赤裸々にしゃべるのが、皆さんご希望だと思うんですが、ちょっとはぐらかしまして、ちょっと違う話をしてみたいなと思っています。

私のヒストリーですが、まず、KL1コンパイラは85年から89年までICOTでやっていました。先程話のあったKL1-Bとか、さっきさんざんやっつけられていた、MRBGCとか、インデキシングとかを作っていました。89年、FGCSが終わって、会社に帰って、何をしようか。失業したわけですね。何しようかなと思って、さっきから出ています Linda というのがあるから Linda を作ってみようか。私、基本的には作る人間で、仕様を考えるほど頭良くないので、とにかく作ってみましょうと。要するに、tuple space、仮想的共有メモリと5～6のオペレーションがありまして、それでメッセージパッシングをやりながら通信を行う。それはCとかFORTRANの上に載せられるので、こういうこと言うと会社の偉い人は喜ぶわけです。これは、逐次言語の並列化という意味で。もう1つ、僕、一番おもしろいなと思ったのは、仮想共有メモリというのがあって、tuple space というのは、共有メモリ計算機で作るには何も問題ないですね。すぐできます。だけど、分散メモリマシンでやる時、これをどうユーザに見せるかということは、今、いろいろ議論されていると思います。そういったものをプログラムレベルから一段、抽象化したレベルでユーザに見せれば、分散メモリ計算機がうまく使えるんじゃないか。そのハードウエアとユーザのプログラマとの橋渡しとして、例えばこういうものがうまくいけば、おもしろいかなと思ったのが、きっかけでした。

やったといっても、実際は、共有メモリ上の処理系を、2週間ばかりで作って - 2週間もかかりましたか、村上さんがやっていた、SWoPP の沖縄に行ったという、それだけなんです。行ったのはいいんですが、帰りに、今NTTですが、元ICOTの尾内さんから、おまえ、何やっとるんだといって、さんざん飛行機の中で叱られまして、それで止めてしまった。(笑) いい先輩持ったなと思いました。

今でもやはり思っているのは、仮想共有メモリみたいなのを90年どろ、もうちょっと本気でやっていたら、もう少しおもしろい結果が分かったのかなというのを、後悔しています。その後、だんだんと会社も傾いてきまして、もっと飯の種になるものをやれという話になって、命令レベル並列コンパイラというので、VLIW とかスーパースカラとかありますが、それのコンパイラをやっています。

具体的には、トレースとかバーコレーションとか - トレースというのは、Josh Fisher ですか、Yale大学にいて今HPの西海岸の研究所に居ると思いますけど、後、その下にいた Alex Nicolau とかがやっているバーコレーションでは、コードを見てそれを並列化するという、basic block を越えて並列化するというスケジューリングです。これもちょっと少しずつ遊んでみようかなと思ってやっているんですが、これもまだお遊びの段階です。この3つを見て何を思ったかというと、ここに書いてありますけど、並列にはいろんな階層があって、ユーザに見える並列、ユーザに見えない並列がたくさんあります。その見え方も多様性があって、いろんなレベルの並列があって、それをユーザに見せるか見せないか、コンパイラまで見せるか、ユーザ用に見せるかというようなことがいろいろあって、やはり一言で並列といっても、なかなか一筋縄ではいかないなというのがありました。

先ほど「改宗した」と言ったんですが、これはほとんどさっきと同じスライドなんですが、一言、言っておきたい。「転向したわけではない。」私は今日これを言うために来ました。なぜかというと、KL1のコンパイラをここICOTでやらせていただきまして、何年かやって帰ってみて、Lindaをやったり、いろんな命令レベル並列をやったりしていましても、やはり考えるべきところは同じである。並列性をどのレベルで扱うか、命令で扱うか、プロセスで扱うか、ゴールで扱うかだけの違いであって、本質的に何も変わってない。つまりこんな辺を区別するほど、まだ我々は進歩してない。そういうレベルまで行ってないというのが、僕の主張です。

どうあるべきかというので、1つは作り易さ、これは開発期間、処理系の大きさ、性能、つまり10年かかって良いものを作っても遅いということ。それからいろんなことをすぐやれるということ。そういう言語であること。1回直すのに、何年もかかったらしようがない。それから作る方から見まして先ほどの通信同期というプロセス、並列実行系とありましたが、これをどうつくるか、これに尽きる。つまり、KL1で言うと、ストリームの通信のあたり、suspension/resume のあたり、ゴールをどう配置するか、ゴールをどう作るか、どう読むかというあたり、Lindaであればタプルスペースをどう扱うか。そのスペース間の通信をどうするか。eval というのでプロセスを作るんですが、それ以前の問題として、どのように作らせれば良いか。

それから、メッセージベースであれば、やはり通信同期をコンパイラでどのくらいチェックするか。あるいは副作用がありますから、その辺をどうコンパイラがチェックするかという力、基本的にそれだけなんですね。そこをどうするかというので勝負は決まってしまう。使う方から言えば、これをユーザにどう見せるか。つまり全然見せない。命令レベルであれば、通信同期や、並列実行系は基本的にはユーザには、今はそういうようには見せない。

HPF なんか多少見せるようにしているかも知れませんが、今の FORTRAN は見せてない。

それから Linda にしても、Linda はこれを５つのプリミティブとタプルスペースというものを持ってきてユーザに見せている。これでいいでしょうと言っているわけです。良いかどうかはユーザが判断するわけです。

KL1 というのは、ゴールを持ってきてストリームの通信を持ってきて、これで良いでしょうと言っているんですが、それが良いかどうかはまだ分からない。それから、プログラム開発と、変更の容易さをどうするかというのも問題です。

最後に感想もどきなんですが、一言で言うと、ユーザは保守的である。ここにいらっしゃるような人というのは、基本的に技術者研究者ですから、新しい物を出したいかもしれませんが、我々の会社などでいろんな人とつき合ってみると、我々の予想以上に保守的である。だからそういう人を、どういうふうに説得してやって行くかというと、やはりユーザから見た特徴をはっきり見せる。つまり言語として、通信同期、論理がありますよ、ガード付きの節で、データフローですよと言っても、ユーザは、あ、そうですかとしか言わない。それだから何がいいのか。何がうまく表せるのか。どう使われるかというのを、はっきり言わないと、ユーザは多分納得しない。そのための１つの方法としては、アプリケーションをはっきり、これは使えるというのを見せる。やるほうとしては、辛抱強く気長にやる。それから、WG が今日こういう所で２日間開かれているというのは、評価、PIM の WG の集大成として、進歩をわかりやすく、皆さんに知らしめるという意味でいいと思う。非常に賛成で、良いと思います。そういうことを頻繁にやるべきである。最後は、仲間を増やす、これに尽きる。これでおしまいです。

【村上】

並列性のための言語というのはどうあるべきかという、非常にこれは大きなタイトルなんですが、基本的に私は言語は何でもいい。要するに、日本人は日本語を得意として、アメリカ人は米語といいますかアメリカ語を得意とするように、得意な言語があればいい。要は、やりたいことができる。先ほど久門さんのところでありましたが、やりたいことができない言えないような、言いたいことが伝わらない言語ではダメである。それから、覚えやすい、文例がある。先ほど Pascal の例ですが、要するに、スタイルがある。それから、やはり効率よく喋りたい。つまり、高速実行したいとか、効率的に書きたい。書けさえすれば何でもいいと思うんですが、ここであえて幾つか半田さんからのリクエストに沿って、システムをつくる側から望ましい言語というので、話をしていきたいと思います。

皆さんご存じのように、これは Wirth の、「アルゴリズム＋データ構造＝プログラム」という図式です。これが逐次処理昔々ですが、昔はメモリ容量とか、メモリ管理というと、プログラムする側が気をつけなければ書けなかった。ところが仮想記憶という技術が開発されまして、現在はアルゴリズムとデータ構造だけ気をつけていれば一応プログラムできる。その一方、並列処理の現状はどういう風になっているかというと、これ以外に考えなくてはいけないことがいっぱいあるわけです。例えば、マシン構造。先ほども言いました、ネットワークのトポロジとか、メモリ容量、まだもう一遍メモリ容量を考えなくちゃいけないかなと。それから、通信容量、通信バンド幅とか、プロセッサ数とか、それからアクティビティの制御、要するに粒度とかそういったものをいろいろ考えて、しかもそれを陽に記述しないと、自分の思ったとおりにプログラムは書けない。理想としては、最終的にやはりここに持ってくるべきだと思います。

そのためには、どういった言語が望まれるか。

処理系にとって PMS が容易な、ユーザにとって、PMS な言語。ポップでスマートでマニアックでといろいろありますが、ここで PMS と言っているのは、語呂合わせもあるんですが、処理系にとって PMS というのは、パーティショニング、マッピング、スケジューリングです。私の好きな３つのキーワードなんですが、１つはパーティショニング。要するに粒度というのは、これは処理系によって定まるもので、言語自身が粒度を最初から定めていてはいけないと思います。これは何故かというと、しばしば陥っている迷信といいますか、「粒度×並列度は問題サイズに等しい」となって、並列度を上げるためには粒度を下げる必要がある。言語で想定しているような粒度も下げてしまっている。最近、細粒度という言葉が１つのはやり文句になっていますが、細粒度というのは何を言っているかというと、処理量と通信量がほぼ等しい、もしくは通信量のほうが大きいわけです。そういった言語で問題を書いてしまうと、そもそも並列に実行してしまう。効率悪いわけです。粒度が小さくなれば、これは１台のマシン上で実行した方が良いわけです。最初から言っているようにそういう細粒度というものを、言語では考えるわけです。やはり粒度というのは言語から見ると、トランスペアレントで処理系に任せてほしい。

もう１つはマッピング。ユーザが書くのはデータ構造だけであって、物理的なマシン構造というのは、ユーザは意識しない方が良いと思います。そういう意味で、データ構造とマシン構造の間のマッピングが容易な言語。先ほど上田さんから並行と並列というのがありましたが、これも似たようなマッピングの話だと思います。

最後に、スケジューリング、これはタスクスケジューリングというか、負荷分散とかいろいろとあります。

もう１つ、ユーザにとって、ポップで、マニアックで、スマートというのは、PMS、これをポータブルでモジュラでスケーラブル、もしくはシームレスということです。ポータブルというのは、アーキテクチャであって欲しいと

いうことです。もう1つはユーザ独立、アーキテクチャ独立はわかりますが、ユーザ独立というのは、ある特定のユーザしか使ってないような言語では困る。あえて言いませんけども……。(笑) それから、モジュラである。要するに、大規模ソフトウエアが書けなければ、いつまでたっても言語としてはやはり一人前にはならない。最後のスケーラブル、シームレスというのは、逐次から大規模並列までの連続性といいますか、例えば P 台のプロセッサで動くようなプログラムを書くのに、$O(P)$ の手間がかかっていては困るわけです。やはりこれは $logP$ ぐらいで、もしくはコンスタントな手間でスケーラブルに書けなくちゃいけない。シームレスというのは、やはり逐次から並列まで継ぎ目のない記述ができるという意味でまとめますと、処理系にとって、いろいろなユーザにとってPMSな言語というのが望まれると思います。

【司会】

どうもありがとうございました。

一応これで5人のパネラにポジショニングして頂きまして、その前に久門さんにお話をしてもらいました。何か、パネリストの方にご質問があればどんどん……、久門さんに対する質問でもいいです。

【中島】

村上さんが最後に、P 台のプロセッサのプログラミングに、せめて $logP$ ぐらいの手間でとかというのを聞いたんですが、今まで大規模というか、スケーラブルなプログラミングというのは、コンスタントの手間で一週苦労したら、並列言語を用いたらもうあとは 1000 台、1 万台に適応可能なプログラムが可能かなんて思っていたんですけども、村上さんに聞いているんじゃなくて、例えば瀧さんに、$logP$ でいい? と聞いてみたいですね。

【瀧】

256 台しか私は試してないですが、私が出てから 512 台が動き出したんですが、256 台に関しては、1 台で動かすときよりはやや手間がかかっているけど、10 台で動かすのも 256 台で動かすのも同じような手間で書けていると思いますがね。今までは。

【 ?? 】

これから、$logP$ となると……

【瀧】

やっぱり変じゃないですか、増えるって。私はコンスタントオーバヘッドで、つまり、あなたはコンスタントオーバーヘッドで並列化できると言っているわけですよね、多分。幻想だと思います、私は絶対。なぜかと聞かれるとちょっと困りますが、マシンだってコンスタントオーバヘッドでできないんだから、その上でプログラム書くこと自体、その作業自体全くコンスタントオーバヘッドでは多分できないと思います。

こういう経験があります。64 台の Multi PSI から 256 台の PIM/m にスケールアップした時に何を変えなければいけなかったかというと、同じデータを持って来たんじゃ効率がすごく悪かった。別に、速くなってんだからいいんですが。マシンを目いっぱい使いたかったので結局、データに大きいのを持って来たんです。データに大きいのを持ってきたら、メモリ消費の問題などがありましたが、そこをちょっと直すということはありましたが、そういう手間だったな。

【 ?? 】

私が関口さんに聞きたいのは、今まで我々、逐次的なもので始めるのではなくて、最初から陽に並列なもんだと言っておいて、それで進んできたわけです。そういう意味で、例えば、ここに居る ?? 君のやっているシミュレータなどはそもそも全然やっていることが逐次的じゃないプログラムなんですね。ただ、ある意味ではデータパラレル的なことをやっているんだけど、全然逐次的じゃないことをやっている。

それに対して、HPF の進んでいる道というのは、基本的には逐次なのに、それがたまたま並列にやってうまく行くというような発想で来ているのかなと思うんですね。これからその方向で、どこまで進めるんだろうというのが、私にはイマイチつかめないんです。その辺、皆さん、どうでしょうか。

【関口】

世の中、本当にグジャグジャな物がいっぱいある。どうしてもそれを書かざるを得ないというような立場の方、もちろんいらっしゃると思うんですが、その反対の極端に非常にきれいな世界もある。この中でうまく行けばいいんだという、まず極端を考えます。HPF の目指しているのは、とにかくベクトル的な、僕で言う一番右で、そっちのほうからずっと始まって来ているわけですけども、そういう風に、データの束を全部一つの太い柱にして、その中の細いデータパラレル的な要素を抽出するというようなやり方だと、ちょっと移っただけで、その辺のマージンがあまり無いわけです。非常に書きにくくなってしまう。HPF はどこに行くか知りませんが、もう1つの言語の方向としては、そういうふうな、束を束として扱うのではなくて、もうちょっと緩い結合として、だからもう少し束の間に自由度があるような、かといって、勝手に動き回るんじゃない。何となく皆で緩い結合を保ったままの言語ということが考えられないかと思っています。別に HPF のフォーラムに入っているわけではないので、どっちに向かうかということ

はお答えできませんけども、そのあたりの言語というのは、少なくとも、私なんかの考えているようなアプリケーションの立場からすると、非常に reasonable なポジションかなと思います。

【司会】

そこで、一番最初に並列性というものを、陽に書くか、それとも書かないのか、実行モデルの中にそれが入っているほうがいいのか、それとも要らないのか……。

【関口】

僕の立場としては、頭の中では、多分、逐次的に物事を考えているから、逐次に書くのだろうと。これをやって、次これをやって、これをやってというふうに、多分頭の中で考えているんだろうと思うんです。少なくともぼくはその程度しか考えられないので、そういう風に考えている。その中で、元々のユーザなりプログラマが意図したような結果と同じ結果を保証するような範囲で、並列に実行できる所というのは、割にコンパイラも探しやすい。それはある意味ではデータフロー的な細粒度のパラレリズムということは抽出できると思います。

だから、言語の字面としては逐次的なセマンティクスが入っているにしても、実行モデルとしては並列ということは可能だと思うんですね。というか実際我々の立場というのはそういうふうに今までやってきている。本職のほうでやってきているというふうに思っていただいて……。

【司会】

そこら辺の実行モデルとか、人間の意図がちゃんと計算機に入るかというところに関して、パネラ全員にお伺いできたらと思うんですが。

【上田】

……実行のモデルというか、計算のモデルというのは、FORTRAN が好きな人は FORTRAN で考えればいいと思うし、そうでなくて、どうしても論理的な仕様から自動合成したい人はそうすればいいと思うんですね。だから多分ここにいる人の中で合意が取れるのは、ユーザがマッピングを書くことができる、少なくとも書きたい場合にはね、その位では合意が取れちゃうんじゃないかと思うんですね。それを書きたくない時にも自動的にやってくれるかどうか、という点では議論が分かれると思います。私としては、たまたま並列論理型言語でモデルが並行なので、それが何とか並列計算にマップができたということはありますけど、でもかなり独立の話ですね。そういう意味で元になる計算モデルの話とそれをマッピングするという話は。

【雨宮】

個人的にイメージしているところは、やっぱり、計算モデルという意味ではデータフローというのはそんなに捨てたものではないと思っているんですね、それをマシンにそのまま実装するかどうかは別にして。多分、私は関数型を考えているんですが、GHC みたいなコミットチョイス型の論理型言語でも、枝葉を落としていったら、モデルとしてはほとんど等価なものであろうと思っています。ただ、言っている所にちょっと食い違いがあるぐらいで、本質に変わりはないと私は思っています、そういう立場ですね。それから、逐次型言語で書かれたものからどれだけ並列性を抽出するかというのは、多分、現実問題としては利口な方法ではないでしょうか。つまりそれはソフトウエア資産の問題ですね、いかに継承していくかというところが重要で、そのためには何かしないといけない。だから、FORTRAN のコンパイラを作るとか、GHC みたいな話ですね。私は一時期にそんなことをやってごらんなさいって非常に言ったんですが、そうも言ってられなくて。

ただ、うちでは Pascal をやっています。何ゆえ Pascal かという話があるわけですが、それは言語の一つの規範を与えるという教育用言語ですが、ユーザは頭の中では逐次的に意識して書いたんだけれどもその中にいかに並列構造があるか。それは実はよく解析してみると、多くの並列性が埋もれているんではないかと。そういう1つの conjecture の元に今やっています。どれだけ並列性が抽出できるかどうかというのは、今はちょっと、まだ結論は見えてないんですが。いずれにしろこういう立場です。並列抽出という問題に関しては、実際にマシンがあってハードウエアがあったら、超並列マシンを目いっぱい使わなくても、まあ、その 10% でも稼働するならそれで良いではないかという、その位のつもりでコンパイラは考えてもいいんじゃないかと。

【木村】

私もモデルという意味だとやっぱり、雨宮先生ではないんですけど、データフロー的なものをやっぱり考えておくべきだと思うんですね。それをどう実現するかというのは、そのマシンに合ったように作ればいいわけで、それを逐次にやってもいいし、本当にデータフロー的にやっても、それで効率が出るんだったらそれでもいいんじゃないかなと私は思っています。逆は多分できないと思います。

FORTRAN の並列化について、最近少し思っているのは、例えば今雨宮先生からソフトウェア資産という話が出ましたけれども、今までの逐次型のコンパイラというのは、例えばレジスタの数は、なるべく数が少なくなるようにコンパイルするわけですね。そうすると何が起こるかというと、いわゆるレジスタのぶつかりで、output dependency とかがいっぱい出てくるわけです。これは村上さんの世界かもしれませんけど、それをどうハード的にほどい

て並列に実行させるかというのが、今、例えばスーパースカラとか、その辺で一番トピックと言いますか、それをどう解決するかというのがあったわけです。それをもう少し、コンパイル技術を上げて、例えばレジスタの推定も、例えば 36 なら 36 個あるように、それをできるだけ目いっぱい使うようなコンパイル技術が出ればもう少し余裕が出て、うまく並列度が稼げるようなコンパイラになるかもしれません。現在その辺がまだ解決されていなくて、そういう逐次の頃から逐次に最適化されたプログラムをどう並列に持って行くかという時には、もっともっと考えなければいけない。今、KL1、GHC というのは、あまりにも並列度がありすぎて、多分我々は持て余していて、流れとしてそれをどう制限するかという方向に行っているような気がするんですが、それの中和点みたいなものが見つかればいいんじゃないかなというのを私は考えています。

【 ?? 】

ちょっと今、FORTRAN のコンパイラの話が出たので、コメントさせていただきたいと思います。それはオブジェクトのコンパチビリティーの話で、バイナリを別のマシンでも並列に実行しろという話ではないので、今みたいに仮想的にレジスタが山のようにある場合ですと、従来で言うオプティマイズのフェーズを取払えば良いと。そこでやっているレジスタのアロケーションみたいなことを止めてしまうわけです。じゃあ、それを実マシンの上でどういうふうに実行するのがいいかというと、これはデータフローでやってしまうのがいい。というのはレジスタというものをそれぞれのトークンとして表わせばいいので、もうこれは何も考えずにできるのではなかろうかと。

【 ?? 】

それでもだから、あまりにも効率が悪いというので、もうちょっと制限したい。つまり、無限に持ってきてデータフローでやろうとすると、それはそれできれいだと思うんですね。それでもいいとは思うんですが、それをもうちょっと粒度を大きくして、もうちょっと効率良くやりたいっていう要望がある時には、そのターゲットのマシンアーキテクチャのハードウエアリソースをかんがみてコンパイラを考えなければいけない。そうすると、コードのコンパチビリティのようなものはひょっとすると崩れるかもしれないですね。それは仕方ないと思う。それはマシンによって、レジスタの数なんていうのはマシンが出てくるたびに違いますから、そこまで言っていると、実際問題、何もできなくなってしまうんじゃないかなと私は思っています。

【西田】

今みたいな話は、私よりもうちの坂井とか、その辺が聞くと非常に泣いて喜びそうな話です。というのは、どの方では無節操に全部並列性を抽出してしまって、結局、並列度が出すぎて手に負えなくなったわけですね。そういう意味でのコントロールができなくなった。その点 EM-4 のマシンでは、それを適当に一つの強連結ブロックと彼らは表してますけれども、そういうことで逐次化してやったということによるコントロールで性能が今非常に注目されているのではなかろうかということは私は坂井に伝えておきます。

【村上】

先程どもマッピングという話をしたんですけれども、実行モデルに並列性が無い時、先ほどの雨宮先生の自動並列化というのは、要するに、落葉拾い的なアプローチではないかと。さらに関口さんの話でいくと、データフロー的に爆発的に広がった並列性を今度は逆にふろしきを縮めるというか、とにかく爆発的に広がった並列性を少なくすると。そこでやりたいことの基本的な技術というのは同じなんです。データ依存の解析をやって、フローの解析をやって、さらに中心の局所性とか、参照局所性を活かすような形で並列化する、もしくは逐次化するということで、そこでやっている技術というのは基本的には一緒だと思います。そういう意味では逐次が先にあろうが並列性が先にあろうが、マッピングする対象が逐次か並列かはどちらでもいいと思います。ただ問題なのは先ほども言ったようにマッピングしやすいような言語であって欲しい。要するに昔の変な歴史的な経緯でマッピングしづらいような言語では困る。そういう意味では、依存関係、データ依存とか、コントロール依存とかそういったものを抽出しやすいような実行モデルであれば何でもいいんじゃないですか。

【雨宮】

結局、今ちょっと言い忘れたのですが、Pascal で逐次化する話で、結局データディペンデンシを追っていって、そこから並列性を抽出するというコンパイルをするわけです。普通に、例えば Pascal で関数を実行して、そこで持ち込むパラメータが call by value 、variable じゃなくてとか、それからプロシージャを呼ばないとか、つまり副作用の無いプログラムならそこからかなりの並列性を抽出できるんですね。問題は何かというと副作用があるものですね。rewrite とか、メモリの R/W とかですね。そういう mutation を起こすようなところになると、途端に非常に難しい問題になるんです。多分、これから研究で本当にやるべきことは、そのあたりをもうちょっとどこまでできるかどういう手法にすれば副作用で書き換えが起こったとしても、それを避けるような、あるいはそこをうまく同期を入れて逐次化をするかとか、あるいはバリアをどうするとか、その辺に非常にクリティカルポイントがあるような気がしています。

【木村】

関数とか、mutation、書き換えですね、そこにはやはり同じ問題があると思いますね。今の雨宮先生と僕は全く同感です。先ほどから、KL1でベクタとか、MRBのGCという話がありましたけど、あれは要するにKL1が単一代入であるからどんどんデータを使っちゃうわけですね。アロケートするために変数を1個使ってしまう。これがKL1で問題になっていて、今度FORTRANとかの逐次言語だと副作用が問題になる。だから極端走っているわけですね。KL1というのはある意味ではメモリのスピードを考えなければ、並列化に関しては、論理の部分だけ考えればよかったですね。しかし、一旦書き換えを許してしまうと、メモリ上での振る舞いというのを考えなくてはいけなくなってきて、一段難しくなるわけです。普通の逐次のプログラミング言語です。だから、先ほど久門さんからKL1に逐次的な破壊代入でやることを入れるか入れないかという話が出ましたけど、そういうのをKL1で安易に入れてしまうと、それはもう難しさというのは普通の今の逐次言語と同じになってしまって、もう何もKL1であることのメリットが無くなってしまうんですね。だから、雨宮先生が言われたように、やっぱりあくまでも単一代入っていうのは死守すべきであって、その世界でユーザにどう見せて、それをどうコンパイラなりハードウエアがうまくインプリメントするかというアプローチを取らないと、我々の存在価値はなくなってしまう。つまり、KL1という統一教会は崩壊してしまうと私は思います。

【司会】

その話に関して、上田さん。久門さんの最後の方で論理性は捨ててもいいんじゃないかという話があったと思うんですけれども、いかがですか。

【上田】

その話は多分、今の木村君の話と同じでして、やっぱりそれを捨てると、結局何をやって来たんだろうということになるし、実際捨てなくて済むと思うんですね。一つ、久門さんが非常に心配していたのはグラフ構造が扱えるかという話で、プロセスグラフで表現すると、今のKL1では遅くしか走らないしデバックがしにくいという話でした。私の経験だと、まずデバックがしにくいかどうかについては、結構難しい木構造で、ノードを上下入れかえるというイヤなことをするようなプログラムをプロセスとストリームを使って書きましたが、ほとんどパッと動きました。そんな感じで、もちろん手で入れてますから、構文的な誤りもありますがそれはほとんどKL1で行くとvarcheck一発で取れるようなもので、そのぐらいのものです。遅いかどうかという点については、久門さん自身と平田君も例のプログラム変換の仕事をしていますし、私もメッセージ指向の処理系で、そういうものを速く動かすという研究をしていますから、本当にPascalのポインタ構造と同じになるかどうかわかりませんけれども、結構いいところまでは行くと思います。面白かったのは、久門さんが節という例題を出してきて、関口さんがヒストグラムという例題を出して、ほとんど本質的に同じ問題なんですね。何が同じかというと、配列をシェアしたい。シェアしている間にやっていることは、割と単調な書き換えなんですね。配列のエレメントを1増やすとか、ゼロだった所を1にするとか、その間に読んで取っておくとかいうことをやらないんですね。確かに、おもしろい例であると思います。配列をプロセスだと思って、何かを書くんだと思いますが、そういうものが上手く書けるような、上手く処理できるような方式というのを考えればどうにかなるんじゃないかと思います。エラトステネスのプログラムはほとんど書くばかりであって、問題が少ない方ですが、それに対するサポートは何かあってもいいような気はします。

【久門】

エラトステネスのはいいんですが、quick sortをarrayでやるというやつはやっぱり大変です。あのアルゴリズムだと並列性が本質的に殺されるんですね。

【上田】

それは全然問題無いと思います、今のKL1ではダメなんですけどね。まず、配列の要素を入れかえるというのは今でもGHCではできるわけですね。もちろんMRB内だと思ってください。それからこの次、どういうオペレーションを用意すればいいかというと、配列をある場所で二つに切るというオペレーションを用意すればいいわけです。切った配列を二人のプロセスに渡してあげればいいんで、後、切った配列をくっつけるというのも、くっつけるべき配列がたまたま実は物理的に隣の場所にあったらタダでくっつくというようなオペレーションを用意してあげれば済む話です。私が配列のオペレーションが弱いねとOHPを使って申し上げたのは、一つにはその辺のことを指しているんです。だから、二次元配列でも、行毎にL本に切ってバラまくとか、それから各列、各行の間で表現の変換をするとか、OHPでもそういうことを言いましたけど。例えば、そんなサポートがあれば、例えば2次元配列の各行や各列に対して、並列にしかもプログラマがinterferenceの心配をしないで、アクセスするなんていうプログラミングが将来的にはできるんじゃないかと。今その辺ができますということではないです。だけど、ちょうどいい研究課題じゃないかという気がします。

【??】

配列の持ち込み方の最大の問題点は、配列がプロセスじゃなかったことです。配列がKL1のファーストクラスのデータオブジェクトで、あれがプロセスじゃなかったのが最大の敗因じゃないかと思うんですね。プロセスにして、プロ

セスがKL1のアレイを抱いていて、そこにマージャーが来て云々というモデルにすると、確かにシリアライズされるとか、それをアクセスするモデル自体がアーキテクチャの世界でいうと、ストロングコンシステンシになったりしてまずいんですけれども、何かの意味で、やっぱりデータ構造というのはプロセスに見せちゃう。オブジェクトに見せちゃうという世界で、裏は後でごまかすと。例えばそれを実はメモリで処理したって、セマンティクスが合っていればいいわけで、そういうような世界で本当は考えたほうが良かったんじゃないかなと。

【上田】

配列の場合は、データとしての配列を用意していれば、プロセスの配列は作れるんですね。逆は難しいです。だから、もっとプリミティブな方を提供したということじゃないかと私は思うんですけど。データとしての配列はそれなりに結構楽しいこともありますね。私は実は、最近割と好きになったんですけれども、ちゃんと自分で白黒を意識していると非常に気持ちよく使えるし、Prologで同じことを書いたのに比べて、破壊的なrenewalのおかげで速く動いてくれますし、楽しいこともあるような気がします。

【司会】

その配列に関する新しい操作とかを導入すると、久門氏の話にもありましたし、木村さんの話にもあったと思うんですが、その処理系の作り易さとか仕様の簡単さという面から、そういうのをどんどん加えて行っていいのかという話もあるかと思うんですが。

【上田】

私の個人的な見解ですけれども、今のKL1に加えていくと、やっぱり大変だから、例えばモード解析なんかができていればそれほど難しい話にはならないでしょう。モード解析だけではなくて、参照数の解析なんていうのも実は非常に重要だと思いますけれども、そういうものが解析されていて、例えばこのデータ、白いか黒いかということをちゃんと解析するようなツールなり、言語サポートなりがあるような状況だったらやってもいいと思います。確かに今のまま入れるのは、多分混沌の度を増すだけであって、VPIMって既にとても1人で理解できるような大きさではないという話もございますし、そこはだから、1回整理した上でやるべきことだと思います。

【司会】

では何かフロアのほうからご意見があれば。

今までの所で、皆さんおっしゃっていたのは、コンパイラが大切であろう、それと、マッピングを分離してちゃんと書けるということがどんなプログラミング言語でも大切であると。そんなお話であったと思います。

じゃあ、パネリスト皆さんにちょっと順番に聞いていきたいと思うんですけれども、並列の言語にlogicというのは必須だったと思うかどうか。論理型プログラミング言語の論理ですね。

【 ?? 】

何にでも論理があるという話はありますよね。

【上田】

例えば、関口さんの話でFORTRANが汚いという話がありましたけれども、それはある意味で何か論理的でない、ちょっと抽象的な言い方で申し訳無いんですけれども。

【司会】

昔から、例えばlogicというのは宣言的であると、実行順序によらないと、で、きれいでしょうと言われていたと思うんですけれども。それで、並列処理が簡単に書けるとか、きれいにかけるという風に言われていたと思います。いまだにそれはやっぱり必須かと。いろいろ並列処理を記述するパラダイムがありますけれども、やはり、その中でも論理というのは特に優れていると思える所であるんでしょうかと。

【上田】

なかなか、いいご質問だと思いますけれど、答えるのが難しいんですね。どうなんでしょう。他の言語とちゃんと比較しなきゃいけないですね、こういうのって。どんな言語でも必ずその言語できちんとプログラムを書くための論理体系があるわけです。だから、それを使ってきちんと書いて、その論理体系がどのぐらい美しいかとか、いかにきれいな定理がいっぱいあるかとか、そういう話はFGCSの一番最後の日に多分Hoareさんが言っていたことと同じになってくるんですね。並列の見方に関しては、そういう意味での並列論理プログラミングのための論理って、まだ我々でそれを使って書いている人はそんなにいないんじゃないかと思いますね。じゃあ、元々ロジックプログラミングをベースにしたことがいいかというと、そのレベルではlogicとは無関係だと思います。ロジックプログラミングが、プログラミング上のいろいろな不完全データ構造とか、unificationとかいうおいしそうなところを取ってきたという所にはメリットはあったと思いますけど、ロジックプログラミングにおけるlogicが我々のKL1プログラミングに役に立っているかというと、そんな風に使っている人って居ないんじゃないですか。だから、それはKL1プログラミングのためのきちんとした論理体系を作るべきという、これもまた今後の課題になっちゃいますけれども、そういうことだと思います。

【雨宮】
ロジックプログラミングという意味のlogicですね、質問の意味は。最初、自分の話をして恐縮なんですが、最初の頃は何とかPrologというか、ロジックプログラミングの並列アーキテクチャとデータフローとうまく結びつけてということを一生懸命考えてました。これは私の結論ですから、とても正しいかどうか分からないですけど、ある時期から、すごくハードウエアアーキテクチャにとってコスト高になって、言語仕様のどこかを削らないとこれは無理だという風に自分なりに判断します。その1つは多分unificationで、余りにもgeneralなunificationをサポートするということに無理があると。それの1つの現われがコミットチョイス型言語になってきたんだと思いますね。そこまで行っちゃうと、私は、これだったら別に論理型でなくてもいいのではないか。この辺ちょっとデータフローと違う。単に関数ベースでやってきたラムダ計算モデルならいいんですが、コミットチョイスにすると実際にインプリメントしようとするとできないところがあります。やっぱり変数のバインディングという概念をうまくアーキテクチャ上で実装に結びつけて行かないといけないという問題があって、データフローそのままだとそれが抜けちゃいますので、そこにはやっぱりちょっとパワーが足りないということは確かにあったんですね。だけども、そこは今のMulti-thread Advanced Dataflowで何とかカバーできるのではないか。そうすると、関数ベースにパターンマッチングというか、制限したunificationを入れて、そのunificationというのは結局コンパイル段階でコンパイルできちゃうんですね。そうすると論理型って何だろうというのが私の正直な感想です。それはある時期からずっとそうで、だんだん、やっぱりそうじゃないかという気がしてますが。

【関口】
ここにいらっしゃる皆さんと私の1つ大きな違いはと言いますと、私は論理型プログラミングというのはやったことがない、書いたことがないというのが正直な所でありまして、はっきり言って役に立つとかどうかというのはあんまり私の立場からはコメントできる立場ではないと思うんですけど、1つはっきり言えることは、FORTRANというものはロジックプログラミングができる前から存在していたということだけで、私のコメントにさせてもらいたいと思います。

【木村】
古いというのがどういう価値を持つかどうか良く分からないんですが、ロジックプログラミングというか、KL1、GHCなどのレベルの言語から始まったということで、僕はああいう新しいことを始める時に、そういう1つのよりどころを求めるというのは非常にいいと思っています。しかしやっぱりここは効率が悪いなとか、例えばさっき言われた一般的なunificationとか、そういう話はやはり非常に効率が悪くて、そこがあるが故に体の効率を落としかねないと思っています。だから、その辺はもう少し割り切ってもいいのかなと。何かによりどころを求める時に、今ある何か別のいわゆる数学的なものに頼る必要はなくて、もっと計算機に合ったような、何かそういうものを求めても私はいいと思っています。別に数学屋さんばっかりがこういうゲームを使う訳でもないし、だからもっと新しいのを求めて行ってもいいんじゃないかなという気がします。だから、具体的には、やはりもう少し関数型的になるんですかね、パターマッチングとストリームのようなものとか、incompleteなデータ構造あたりは残したいなという気はしますけれども。その辺を残しておけば、かなりの効率の良い、私は作る方しかいつも頭にないんですが、良いものができるような気はします。

【村上】
平田さんのご質問は、並列処理プログラミング言語にロジックプログラミングが必須かという風に理解します。まず、プログラミング言語としてlogicが必須かというとそれはちょっと分かりません。多分logicでないと書けないような問題が存在すれば必須なんでしょうけれども。並列という方において、並列言語にロジックが必要かというと、並列に実行したい、要するに並列性をどこから引き出そうかと考えた時に、ロジックがよりどころにしているのはgoal by goal、要するにゴールという観点から並列性を引き出す。関数型言語というのは関数をよりどころにして並列性を引き出す。例えば、HPFみたいなものはデータをよりどころにして並列性を引き出す。ですからデータパラレリズムという風に呼ばれているわけです。ですから、何かよりどころにするという意味であれば、並列言語としてロジックは必須であると言えると思います。

【鬼塚】
私はあまりロジックプログラミングで書いたことがないとか、いろいろ問題があるんですが、実は最近1つ気がついていることがあります。非常に一般的な話になるんですが、コンピュータ言語の中では例えばLispとかPrologとかというのは比較的数学的なものが最初にベースにあって、それを元にして実用的なものを作っていこうという形でつくられたものだと思うんです。それに対して、FORTRANとか、C言語とかもそうですが、アセンブラもそうなんでしょうが、非常に現場から出てきた言語というのがありまして、アセンブラなんかも最初に現場ありきだと思うんですね。その中で最近非常におもしろいと思っているのはオブジェクト指向の概念なんです。これは実は現場から出てきまして、構造体の中に関数ポインタを突っ込んだことが最初の初まりで、そこからSimulaとかが出てきたらし

初日，パネラーの面々，右から雨宮，関口，木村，村上の各氏

いんですけれども、そのほかForthなんていうのも現場のプログラマが作った言語だと思うんです。そういうような現場から出てきたものの中に非常におもしろい概念があると最近非常に思っていまして、むしろ、最初に数学ありきでつくった言語と違った、そこからしか考えられないような妙な哲学とか思想というものがありまして、私はよく知らないんですが、そういう風にHPFも現場から出てきた並列言語であると。それに対してKL1は元はおそらく数学ベースだと思うんですが、そういう所から出てきたと。現場ベースで出てきた言語と数学ベースで出てきた言語が、今後どういうふうに行くのかなというのが非常におもしろくて、現場から出てきた並列言語がどういう並列性を狙っているかとか、どういうふうに便利に並列を使えるようにしているかというのを、やはりつぶさに観察していくことが、今後の並列システムをつくる上で、非常にいいのではないかと思います。それはオブジェクト指向もやはり現場から出てきたものがこれだけ発展したという、それで、数学的な根拠も少しずつ作られつつあるという意味で、非常におもしろいんじゃないかなと思っています。

元にあったGHCとか、さらにその元のPrologが非常に論理的にきちっとしたものだったと、私はそう解釈しているんですが、それからだんだん現場に移っていくという形式ですね。ところがFORTRANはおそらく最初から現場だったんじゃないかなという気がしているので、それの方向性が、むしろだんだんくっついていく方向に行くとしたら、何が一番いいのかなというのを考えてみるのが一番おもしろいと思います。

【雨宮】

私はそれほど知っているというわけではないですが、FORTRANがほんとに現場かなと思うんですね。フォーミュラトランスレータ、あれはやはりBNFバッカスで、あの連中がやはりアセンブラとか機械語で書いていたんでは、とてもたまらんと。だけど、あのころは科学技術計算、数値計算だから、そこでやられる数式処理というものはこんなものだという風にして、かなりフォーミュラ、名前の通り出てきたと思うんです。そういう意味では、やはりあの当時ではモデルが先にあったと私は思います。

それから、C++の、オブジェクト指向を考えるようになったのは、オブジェクト指向という概念がその以前にあったわけです。Simula、シミュレーションランゲージ、これもそうは言わなかったけども、そこにやはりプロセスペースの概念というのを、あの言語を通して消化していたと思うんです。シミュレーション・ランゲージとして提案はされたものの、あの裏にあるフィロソフィーがかなり当時注目されたと思うんです。現場から実体は出てくるけども、やはりそういう抽象的なところで、解釈できるモデルというか、そういうのは絶対必要だと思います。それでこの機能は、実はこういうところに位置づけられるんだという風なことが必要だと思いますけど。

【鬼塚】

そうですね。おそらくそういうことで、やはり最初に現場から妙なプログラミングスタイルが生まれてくると思うんですが、その生まれてきたものを誰かがそれを論理化というか、一種、理論化して、そこから新しい言語とか、思想とかが生まれてくるということだと思います。現場からという意味で、僕はむしろPrologが生まれてきた時はやはり、昔からあった論理学が最初にあったんじゃないかなと思ったんで、その意味で議論かなというふうに考えたわけです。

【酒井】

今まで、細々とKL1処理系なんか作ってきた訳ですけど、やはりスピードということを考えちゃうわけです。例えば、10年前といいますか振り返ると、インタープリタは、WAMにしたということで、あれは多分、コードをすごく小さくしたということがあると思うんです。

ただ、実際に普通のマシンでやろうとすると、KL1-Bというのが、ここでVPIMで言う最小単位であったりするんだけど、実際のマシンでやるときは、それが何命令かにまた展開されちゃうわけです。そうすると私の立場ですと、KL1というのはある目的を持った仕様記述的な所があるんだけど、KL1-Bでも、例えば処理系が見てやっているというのは本当は良くなくて、もっと各マシンが本来持っている命令による、例えば展開しなきゃいけないんじゃないかななんて思うわけです。そのときFORTRANなんかだとマシン命令というのはすごく連想されるんだけども、

どうもKL1というのはなかなかそこから、マシン命令にうまく落ちないんじゃないかと思って。そういうプログラミング言語をつくる立場から言って、KL1というのを、皆さんどう思っていらっしゃるのかというのを、お伺いしたいんですが。

【上田】

確かに、FORTRANほど簡単じゃないかもしれないけど、ちゃんと真面目にやれば、ちゃんとFORTRANと同じスピードが出ると私は思っています。だからもしニーズさえあれば、HPFに対抗して、数値計算用KL1のサブセットを作って、処理系も作って記述したいという気がします。

実際、何がうれしいかというと、もし配列あたりも同じスピードが出るのだったら、他にちょこっとリストをつくるとか、構造体をつくるとか、バラすという機能があると、プログラムはぐっと書き易いんですね。だからそういうメリットもありますし、世の中ついてきてくれるかどうか非常に疑問ですけれど、そういう方向に行くのも、面白いんじゃないかという気がしています。

【司会】

その発言の自信を裏づける技術というのは、例のメッセージ指向……

【上田】

別に関係ないです。それも入ってくるかもしれませんけど、それは配列以外の……。

【??】

ちょっとそれでお聞きしたいんですが、そういう処理系を作るとなった時には、なんか処理系の仕様といいますか、KL1-Bだと、処理系として大き過ぎるという風に……

【上田】

それはそうだと思います。だからもう最初からこういうKL1プログラムを実行したらこういうRISCのコードが出るだろうなと思って、コンパイラの機能推論するというような態度で考えないとダメかも知れません。というか、逆に、こういうコードを出したいんだったら言語はこうしなきゃいけないということが、私がここ数年考えてきたことでもあるわけです。

【司会】

上田さんはKL1のコードを見るとオブジェクトが連想できるかもしれないんですが、できない人も多いわけですね。このギャップは何故なんでしょうか。

【上田】

それはPrologでも同じじゃないんでしょうか。10年ぐらい前、近山さんがやはり「僕はPrologのプログラムを見るとオブジェクトコードが連想できる」といっていたような気がしますけど。そのころ私は全然分からなかったですね。いまだにPROLOGだとあまりきちんと連想できないけど、KL1だと、よりやく連想できるようになりました。やはり長年処理系をつくっていると、要らなそうなコードがいっぱい出ていて、嫌だと思う、そういう動機があればいいんじゃないでしょうか。

【??】

上田さんの自信の裏には、スタティックアナリシスに対する、ここ2年ぐらいの自信がすごくというのか、あるんだろうなと思うんですが、私もスタティックアナリシス、大賛成なんですけども、必ずそれがすべてのことを解決するかどうか、特にKL1みたいな言語で、嬉しさというのがどこにつながっているか分からないんです。例えば、スタティックアナリシスと言った時に、字面では基本的に分からないようなプロセス間のチューンなども、実はスタティックアナリシスしちゃうんだというぐらいの、根性が多分必要になるんじゃないかなと思って、私はそっちの方で行こうかなと思っているんですけど。

これはただのコメントです。

【中島】

処理系の重さに関して言うと、例えば、PIM/m、Multi-PSIなどでKL1-Bを解釈している部分というのは、全体の3〜4割という気分なんです。残りはランタイムルーチンなんですね。特に分散処理というか、プロセッサ間、ノード間、群間、里親、MRBの導入で複雑になっている部分とか、ハード化した部分というのはもちろんあるんですけども、並列ゆえに、たくさんやらなきゃいけない部分があるということを言いたいです。それで、上田さんはそこら辺をどう考えていられるのかなと。他の言語にしたら、KL1でなくて関数型にしたら、そこら辺簡単なのかなと。そこら辺、皆さん、いかがですか。

【上田】

すいません。確かに今、中島さんがおっしゃったところは、私があまり考えていないところではあります。

【??】

上田さんにちょっとお聞きしたいのですが、KL1の場合、スタティックアナリシスというのはどの位、例えば、白

分は10できたら満足できるなと思った時に、実際どのくらいできるものなんでしょうか。どうしてもわからないという場合もあると思うんです。さっき言われましたけど、とことんゴールを探し回らないといけないとか、そういうこともあるんじゃないか。だから、実用レベルでどこまでできるかという所で勝負は決まりそうな気がするんですが、その辺ちょっと、僕はあまり最近フォローしてないんですけども、どのくらいの見通しをお持ちなんでしょうか。

【上田】

どうお答えしたらいいでしょうか。これだけの解析……、やはり自分が満足な解析ができる位言語を書けなくしちゃうという、悪い方向に私は次々と考えてしまうんですけどね。それは書きたいことのバラエティがそんなにないんだったら、それは1つの見識だと思います。もちろんそれによって、排除される人は非常に気の毒というか、そういうものに本当にニーズがあるんだったら、サポートしなきゃいけないですけど。一方で、速くて柔軟性があるというのは無理なわけですから、もしかしたら上手く行くかも知れませんけど、そうなら、速い方のニーズがとりあえずすごくあるのですから、それにつき合って柔軟性の無い言語を作っていけないこともないような気がしますね。

【中島】

話にケチつけているわけではありませんが、やはりGHCじゃないと嫌だという話と、HPFでいいという話というのは、多分あると思うんですね。そのちょこっとしたリスト処理とかいうのが、書けたりした途端に、木村さんの怖がっているような話が出てくるんじゃないかなと、私は思ったりします。とりあえず。

あと、例えば、C++というのがオブジェクト指向言語だということになっているらしいけど、あのオブジェクト指向という話が残る程度に、GHCを並列論理型と言える位に縮小化して何かいいものができるのかどうか、ちゃんと考えておかないと。いい所だけ取って、それが本当に良いものかというのを考えなきゃいけないかなと思います。

【久門】

先ほど、上田さんから、コンパイラがスタティックアナリシスをやっていいコードを出すというお話があったんですが、これには1つすごく気になることがあります。それは上田さんはコンパイラやスタティックアナリシスをやられているわけで、こういう風に書けばスタティックアナリシスがされていいコードが出るって、知っているんですね。ところがそれが自明かどうかというのは全く別の話で、よく似た別のコードを書くと、全然思ったようにアナリシスできないということは往々にしてあると思うんです。例えば、MRBの白黒の話というのは比較的簡単ですから、変数が2カ所以上で参照されてはいけないというのを守ると白くなるというのは結構簡単なんですが、それでもこのことが書いている最中に自明かというと、あまり自明ではないと僕は思っています。

同じような話で、もっと複雑な条件のもとで、ここでこう使われて、こういう風に引用されて、だからこれは参照できないんだけど、こういう風に持ってくればできるでしょうというのを、自明な形であらわせないならば、極めてトリッキーに書いたプログラムが、スタティックアナリシスでFORTRANに匹敵する速度で走るといっても、説得力が無いと僕は思うんです。つまり大方の人が、期待するような書き方を、自分がやりたいと思う書き方を書いて、それが期待するようなコードに落ちてくれる。少なくとも今のCコンパイラなんかはそうで、僕もある程度Cコンパイラのソースを読みますから、こう書けばこう落ちるというのが、人以上には分かっているかも知れないんですが、そういう多くの人に分かるような制約、あるいはさっき上田さんがおっしゃったみたいに、言語そのものを規定しちゃうというのも、1つの案だと思います。そうしないと、トリッキーなプログラムを書いて、ここの所のユニフィケーションの左右をひっくり返しておいてとか、本来だったら別々にしておくべき所をくっつけると速いんだというコードではダメなんではないかなと思うんですが、そこら辺、どうお考えでしょうか。

【??】

やはり速く動かしたいんだったら、速く走りそうもないものは、コンパイル時に何か警告を出すというのが、正しい方向だと思います。

【近山】

Cコンパイラは、普通の人がこの順番に実行すると思う順番に実行してないんですね、最近。RISCのせいで。実際何しているかというと、Cコンパイラは、書いてあることの意味を一生懸命データフローグラフに直して、そのデータフローをうまく実現するにはどうするか、というコードに再編成して出している。だからもう今やC言語ってすごく計算機に密着しているように見えるけど、実は全然密着してなくて、1回、データフローの世界に行って、それからまた戻って来るという感じですね。だからそういう意味で、もう世の中は、言語を素直にコンパイルするといいコードが出るという時代ではないと私は思います。

【司会】

それは別に逐次型言語とか並列言語とか関係ない話ですね。

【??】

そこで問題になるのが、さっきから言っているベクタとか、そういう話だと思うんです。結局、何でみんなベクタを使いたがるかというと、それが今の普通の計算機のメモリモデルによく合っているからなんです。PIMにしても、

下を見てみると、プロセッサがあって、そのプロセッサからはアドレスピンというのとデータバス用のピンというのが出ていて、結局メモリの読み書きをしている。そういう、CPUチップがやることとベクタのモデルというのは、よく合っているというのがありますが、それに対して、論理型というのは、結局、それとミスマッチしているというか、もっときれいな言葉で言えば、抽象度が高いと言うか。だから、先ほど近山さんが言っておられましたけれども、論理型言語を使うのに対し、論理型言語風にやるようなことを逐次型のCとかで書いたのを比べれば速く行く。それは確かに間違いないと思うんですが、これ、良く分かりません。世の中の人が普通にやりたいような問題というのをやらせた時に、論理型で書くのと、いわゆるCみたいなもので書くのと、とにかくやはり計算機との整合性から言うと、素朴には逐次型でやったほうがいいから、だから平たく言ってしまえば、HPFで書いたほうが速くできることが多そうな気がするわけですね。だからまとめにならないですが、結局、論理型で書くことのメリットというのは何かというのが、やはり明確にならないといけないんじゃないか。スタティックアナリシスで同じくらいの効率のコードが出るよといっても、論理型で行われている抽象というのが、いろんな人にとって非常に便利なものであって、そのメリットをかなり多数のプログラマの人が、よく認識するようにならないといけないんじゃないか。教育の問題なのかもしれないですけど。何かICOTの研究が、途中からKL1ありきみたいになってしまって、KL1使えばこう書けるよというような格好で進んでしまって、本当に書きたいものが何かとか、本当にKL1で書くことが書き易いことなのかとかいうような議論が抜けてしまったような気がして、そこのところは残念な気がします。そういう意味で、論理型で書くメリットが何なのかというのがもう少し明確にならないと、HPFとかそういうものには対抗できないのではないかという気がするんですけれども、コメントをいただければと思います。

【上田】

多分、そのメリットを明らかにすることは必要だったと思うんですが、限られたリソースで10年間でやるとなると、やはりどうしても処理系を作る方に人を割かなければいけませんし、そういうプログラム言語の理論をやる人が日本には非常に少ないという状況を考えた時に、ICOTのその面が多少遅れたということじゃないかと思います。直観的には、何か非常にいいものがそこから出て来るだろうという気はします。でもそれは直観でしか支えられてないんですね。それは多分、HPFを使うよりは、プログラミングのためのlogicを作った時に、なんか美しいものになるであろうという直観だけで、それを信じてくださいということじゃないかな。それは多分誰かが頑張るとできるようなものだと思います。だからまだ立証されているものじゃないのではないでしょうか。

【瀧】

論理型のメリットじゃないけど、KL1で書くことのメリットというのは、いろいろアプリを書いて、やはり幾つかはっきりしてきたんじゃないかと僕は思っています。KL1の中には、論理型言語から生まれたことによるメリットと、KL1を実用言語にする時にくっついてきたものに関するメリットとあります。ざっとまとめると、データフロー同期なんていうのは、これはどっちかというと、論理型の方からきていると思います。それから、比較的細かい－私の言ったのは、関数型とそんなに区別しないぐらいの論理型です。それから、比較的小粒度の並列プロセスというのがあまり違和感なく扱えた。これはメリットとデメリットと両方あるかも知れないんですが、並列で引き出しやすいという意味でメリットがあったと思うんです。

それから非決定性というのがあります。これはやはりguarded commandから受け継がれているもので、これは論理型というよりは、GHCのというかコミッティドチョイスのということでしょうか。

論理型じゃないところでプラグマというのがあります。これはやはりコンカレンシとパラレリズムを独立に扱うということを、現実のものとして扱えるようにした、アロケーションプラグマ。それから、まだよく分かってないんだけど、いろいろ役に立ったのがプライオリティのプラグマ、それからあと、組み込みの機能がいろいろあったり、壮園があったりしますが、ちょっとその辺はまだあまり整理できてないです。

今言ったようなことで、結局どう書いてみたときにありがたかったかというのを、私なりにまとめた言葉は、もう何遍も言っていますが、結局、irregularな問題、非常にデータが不均質であったり、処理がダイナミックに変化するような処理であったり、そういうものを扱って、かつ性能に関するチューニングができると。まずそういう風なものがプログラムとして表現できて、それからそれを実際のマシンにマッピングして、ちゃんと性能のチューニングができるというふうな、そういう並列の実験用の言語として、非常にすぐれていたんじゃないかということは言えると思うんです。

ただそれが、どのくらいどこが本質で、いつまで続かなきゃいけないものかというのはよく分からないんですけどね。とにかく今の時点での、実験用の言語としては非常に優れていると。性能に関してはもうちょっと頑張りたいという感じはありましたね。

【司会】

では、最後にまとめるという意味で、パネラの皆さん、一言ずつ、ICOTのこれからの並列言語研究に期待するところ、期待しないところというのを、一言ずつお願いできますか。

【上田】
みんな喋ってしまったので、もう何も思いつかないんですが、まあ、並列論理型とかやって、8年ぐらい何とかもったんです。あと8年もつと21世紀なんで、私がそれまで、その分野で論文が書けるということを期待しています。
【雨宮】
ちょっと文化が違う。でも大方同じ文化圏内にいると思います。多分、21世紀というか、これから先、その2つあるとすればロジックとファンクションですね。どっちがどうすり寄ってきているのか、私は今、言えませんが、いわゆるDeclarative-base++というか、その裏に並列というのがあって、新しい、それこそパラダイムというものをこれから大いに作って行くために頑張ってください、さらに。ということです。
【関口】
今まで黙っていましたけども、私は実は、今日お話しするまで、HPFに対して非常に懐疑的で批判していたんですけども、今日お話しさせていただいて、皆さん思った以上に、HPFに対して好意を持っていただいているようなので、私自身も少し考えを改めなければいけないなと思っています。

そういう意味で、いわゆる数値計算の世界と、こちらのような論理型といいますか、知識情報処理の世界というのは、徐々に歩み寄りつつあるのではなかろうかなということを期待しております。
【木村】
徐々に歩み寄るかどうか、私分かりませんが、論理ということで、ICOTを10年やって、これから何年もやると思うんですが、今までやってこられたんで、これからもどんどんと実証することによって、いろんな世界ですか、日本ですか、リードしていっていただきたいなと思っております。
【村上】
並列の論理でもよろしいんですが、ICOTで抱えている問題というのをきちんと定式化して、一般化して欲しいなと思います。要するに、KL1とかPIMとかそういう文脈に依存しないような形で問題を一般化して、もしそれに対する解が見つからないのであれば、open problemということで、世の中にちゃんと提示してくれれば、いろんな人が興味を持って研究をやってくれると思います。

いろいろ実装することで時間が非常にかかったんでしょうけれども、コンテキストに依存した問題の提示の仕方が非常に多いと思いますので、今後の希望としてはそういう一般的な問題として、問題をきちんと定式化して欲しいなと思います。希望です。
【司会】
最後だけパネルらしくなって、良かったなと思うんですが、いたらない司会で、すみませんでした。どうもきょうは遅くまでありがとうございました。

(おわり)

パネル1スライド 上田

核言語設計にとっての論理プログラミングの役割

論理プログラミングなしでは、GHC/KL1 の枠組は生まれなかった。

— 強い設計指針を与えた。

— 有用な概念を受け継ぐことができた (例：不完全データ構造)

— 差異・関連の明確化が概念整理に有効であった。

— 既存の並行・並列処理の方法論に頼らなかったため、並行・並列処理の本質の追求に深みを与えることができた。

今後の課題

言語的成熟

— 静的解析とコンパイル技法

— 大規模並列ソフトウェア構築の支援

並行・並列処理の概念基盤の提供から、文化の構築へ

1

他の言語族／計算モデルとの比較

	優れている点	学ぶべき／考察すべき点
並列 Lisp (Multilisp, Qlisp, ...)	処理系作成技法の検討 ソフトウェア蓄積 言語設計	言語全体としての記述能力の比較
データフロー (Id, ...)	同上	同上
Linda	計算モデルの考察 形式意味論	宣伝普及
並列オブジェクト指向	モデルの単純さ 形式意味論	記述能力 (reflection 等) 文化構築
並列手続き型言語 (HPF, ...)	並行と並列の分離	マッピング指定方式 実行効率
並行処理モデル (CCS, CSP, Petri Net)	動的処理の記述 プログラミング言語としての実用性	仕様記述言語としての検討 理論的成果の蓄積

2

核言語関連研究の成果

- 単純な並行・並列言語の提案
  - → 概念整理への貢献 (例：並行と並列の分離)
  - → 理論研究の触発
    - → 計算モデルとしての価値
    - → プログラム解析等の形式的技法の開発促進
  - → 他の多くの分野との関連の発生
- 並列記号処理系作成技法の開発
  - — メモリ管理、分散実装
  - — コンパイル技術の進化
  - — 負荷分散支援
- 並列記号処理プログラミング技術の蓄積
  - — プログラミングパラダイム
  - — 並列探索技法
  - — 負荷分散技法
  - — 並列ソフトウェアの蓄積 ($\sim 10^6$ 行)

→ 計算機科学 (並行・並列記号処理) における新分野の創立

3

Challenges in the Near Future

(1) Strongly Moded FGHC/KL1 による大規模/システム プログラミング

(2) 値の resource としての側面を意識したプログラミングとその支援

(3) 「並行」と「並列」の完全分離は可能か？
— マッピング機能の充実

(4) ベクタまわりの大改良
— マッピング
— 並列アクセス
— 多次元配列とその操作

(5) 理論的基礎をもったマクロ機能

4

並列処理のための言語とはどうあるべきか

関口智嗣
電子技術総合研究所
sekiguchi@etl.go.jp

1993年3月22日

並列処理のための言語とはどうあるべきか

— Fortran 擁護派 宣言 —

Fortran こんなに素敵

- 事実上の標準言語
- プログラミングの連続性
- 親しみのある（！）記述
- 高品質なライブラリの継承

Fortran 最前線 (HPF) でなぜ悪い

- Data Parallel 記述による容易な並列化

  (**M**onkey **P**arallel **P**rocessing!?)
- Forall 記述による容易な並列化
- Work Sharing による負荷分散

Fortran のここが悪い！

× Irregular な形状に対応していない?

× 並列性を明示しないといけない?

× バグが入りこみ易い?

× 美しくない！

Fortran の逆襲: 化粧落せば皆同じ

- プログラムの再利用 (Loop)
  → Tail Recursion じゃやだ
- 結果の回収 (Reduction(op,var))
  → アセンブラでは関数性がない
- 結果の回収 (Histogram)
  → Id だって特別扱い， KL1 は? Valid は?

Fortran そんなことないよ！

- 超並列の世界では (結合網とて) 単純
  → HyperCube の衰退を見よ
- 夢でない自動並列化コンパイラ
  → できることだけやってやれ
- 本質は「通信と同期」の実装
  → KL1，データフローに学ぶ

5

Fortran これでどうだ！

- 並列記述をユーザに開放しない
  (逐次化記述のみ)
- Message Passing は禁止し，
- Distributed Shared Object による
  通信と同期 (高機能構造体メモリ)

6

パネル1スライド雨宮

超並列コンピュータ

-- 高速化への挑戦 --

雨宮真人
(九州大学)

静 対 動

- 宣言的言語
  コンパイラ
- 単純アーキテクチャ

実践

$10^3 \sim 10^4$ 環境

- 速い
- 易しい
- 安い

・並逐超越　言語

・粒度無依存　アーキテクチャ
  - スレッドスイッチ
  - レイテンシー
  - 通信

・負荷制御

「並列処理のための言語とはどうあるべきか」

（1）作り易さ

開発期間、処理系の大きさ、性能、．．．

実験のやりやすさ

通信、同期、並列実行体、の実現

（2）使い易さ

通信、同期、並列実行体、の見せ方

プログラム開発／変更の容易さ

ユーザは保守的

ユーザからみての特徴の明確化（アプリを示す）

長く辛抱強く、気長にやる

評価など進歩をわかりやすく示す

味方（仲間）を増やす

（1）ＫＬ１コンパイラ（’８５～’８９）

ＫＬ１Ｂ

ＭＲＢ－ＧＣ／インデキシング

（2）Ｌｉｎｄａ

逐次言語の並列化

仮想共有メモリ（タプルスペース）

（3）命令レベル並列コンパイラ

ＶＬＩＷ／スーパスカラ

Trace / Percolation scheduling

並列化の階層、並列の多様性

ユーザへの「並列」の見せ方

パネル１スライド木村

パネル1スライド村上

処理系にとってPMSが容易なPMSな言語（ユーザにとって）

- Partitioning — 粒度
- Mapping — データ構造 ⇔ マシン構造
- Scheduling

- Portable < Architecture Independent / User Independent
- Modular — 大規模ソフトウェア
- Scalable / Seamless

粒度 × 並列度 = 問題サイズ
スループット上げるには 単くなる

アルゴリズム + データ構造 = プログラム

逐次処理の昔

+ メモリ容量，管理
⋮

並列処理の今日

+ マシン構造
+ マシン容量
（メモリ容量，通信容量，プロセッサ数，…）
+ アクティビティ制御
⋮

大懇親会（中国飯店にて）

初日夜の懇親会には大勢の方々が参加し大変盛り上がった

懇親会の席上，奥に談笑する田中英彦主査と渕所長の姿が見える

そして懇親会終了後，議論の場は ICOT に移った．

VPIM とはなんだったのか?

発表者: 後藤 厚宏 (NTT)

H時: 10:00 ~ 10:30, 3/23

議事録担当: 屋代 寛 (ICOT)

1 KL1 の並列実行処理系としての側面

- マルチ PSI 分散処理系, 佐藤氏の共有メモリ処理系 (Crammond, Foster, Tick) などを反映.

2 PIM/s の開発環境 (VPIM/PSL) としての側面

- コンパイラ屋, マイクロ屋不足を補う.
- 多すぎるハードをサポートするため.
- 何回も作り直したい (実際にはできなかったが).

3 並列マシンの OS (カーネル) プロトタイプとしての側面

- KL1 を利用した再粒度並列 OS
- ハードウェア資源の仮想化
  - マルチプロセッサ
  - メモリ (→ 各種のガーベジコレクタ)
  - ネットワーク

4 VPIM を作った反省点 (多数)

- 一人で見渡せることが必要だが, 現在の VPIM は大き過ぎる.
- やはりコンパイラにしわ寄せがいってしまった.
- VPIM の機能を評価するための枠組が書けていた.
- 並列アーキテクチャの評価手法
- 「並列アーキテクチャ動向委員会」をもっと早くやるべきだった.

5 得られたもの

- 多くの要素技術
  - メモリ管理, GC, 各種のキャッシュプロトコルなど
- KL1 で, 技術目標を大きく振ったことによって, 他の技術に縛られなかったこと.

6 議論

- VPIM が肥大化した原因は, Multi-PSI 処理系の影響も大きかった.
- 肥大化した原因 ← PIM/m 以外の PIM に共通の問題点.
  - 見通しの効く大きさの閾値を越えてしまった.
  - 切り捨てる作業をすべきだった.
  - VPIM が大きくなることを恐れずに書いた. 当初は, オブジェクトサイズに関しては, 予測していなかった.

- 大きくなるのは分かっていたが，それが問題となることを認識していなかったのが問題．

- 名前の付け方などに工夫をしているため，可読性はよい．

- 仕様記述と実装の 2 つを追ったためのオーバヘッドがあったのではないか?

- 各社に VPIM を提供する方法として別の選択肢はなかったか?
  → 提供の方法としては，それほど悪くなかった．

- 各社でやったメリットは?

  - 相互に助け合うことができた．(例: PIM/i で確認されたバグを，開発が先行していた PIM/p にも反映することができた．)
  - 関係者の人間関係．

- より抽象的なところに落したのはメリットではないか?

  - 結果としては利用できなかった．

- ICOT 外の人間から見ると汎用的な部分に返す技術が少ない．VPIM はハードウェアをひきずり過ぎている．

- PSL が仮定しているハードウェアは世の中の並列マシンを反映しているか?

  - 直観的に，面倒なことはやっていない．スケジューラ，メモリ管理のメカニズムを汎用として提供するのは不十分．

- 作り直すとして VPIM という枠組をまた使うのか?
  → 大きな見直しが必要そう．(例: KLIC 処理系など)

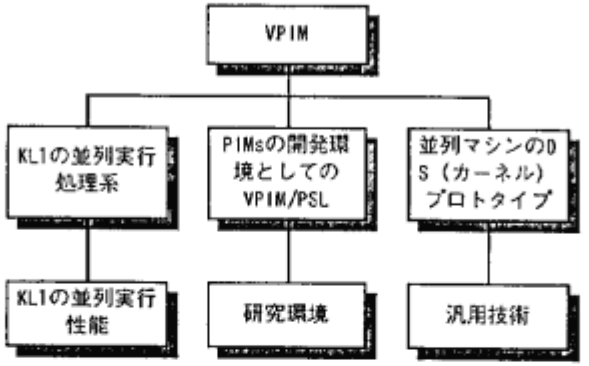
VPIMとは何だったのか？

NTTソフトウェア研究所

後藤　厚宏

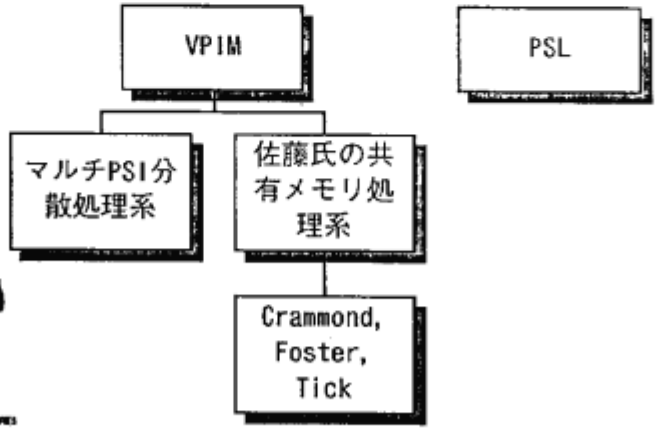
1

VPIMとは何だったのか？



2

KL1並列実行処理系として



3

並列マシンOS

- （KL１を利用した）細粒度並列処理OS
- ハードウェア資源の仮想化
  - マルチプロセッサ
  - メモリ（→各種のガーベジコレクタ）
  - ネットワーク

4

VPIMとは何？スライド

開発環境として

コンパイラ屋不足

マイクロ屋不足

多すぎるハード

何回も作り直したい

5

反省多数

- 足し合わされれば大きくなる
  - 恐竜の滅亡（総身に知恵が...）
- 一人で見渡せることが必要
  - 現在のVPIMは「カーネル」と呼べないくらい大きい
- やはりコンパイラ
- VPIMの機能を評価する枠組みが欠けていた
- などなど

6

反省の続き

- VPIMアーキテクチャの評価尺度
  - ハードウェアの性能限界との関係
  - それぞれの機能要素の性能目標は？
  - 目標のどこまで達成した？
- 並列アーキテクチャの評価手法
- 「並列アーキテクチャ動向委員会」をもっと早くやるべきだった？

7

得たものも大きいはず、と思いたい

- 多くの要素技術
  - メモリ管理とGCs
  - 各種のプロトコル
- 大きく振ったからこそ体験できた
  - 「KL1」であったこと
- 汎用的技術としての見直しが重要

8

プロセス制御, メモリ管理に限らず, システム完成に設計から至るまでを振り返り私の持つノウハウすべてを捧げます

発表者: 今井 明 (シャープ)
日時: 10:30 ~ 11:00, 3/23　　議事録担当: 寿崎 かすみ (ICOT)

1 感想

動くものを作るのは大切である. 批判ができるのは動いたからである.
PIM の研究開発において他流試合が少なかった.
KL1 の評価を現在のシステムのみを考えて行なうのは問題がある.

2 反省

どんな状況にも対応できるインタプリタには限界があった. これは, 動的判定が多過ぎることによる.
最適化コンパイラが現在存在しない.
RISC と CISC の問題については, 始めの処理系が動くところまでは KL1-B を設定した現在の方式は良かったと思う. しかしその後最適化を行なうにあたっては問題があった.
メモリ管理の方式としてヒープベースのものを用いたのは, 開発の工数など考えると割高になった. 逐次性の高い部分を抽出し, 基本処理はスタックベースとするなどの方式も検討するのが良かったのではないか.
コピー GC の所要時間は, 問題になるほど大きくなかった.
動的な処理に頼りすぎた. 構造体定数は良かった.
メモリ容量と時間の節約の優先度に一貫性がなかった.
ゴリゴリプログラムを書き過ぎた.

3 今後残るのは

基本的なアイデア:

- weight を使う分散管理方式
- 共有リンク, ハッシュなどのテクニック
- 論理的意味と実装は別であるという発想

などだと思う.

4 デバッグの経験から得たノウハウ

教訓 : 急がば回れ

1. PIM には PDSS, Multi-PSI, PIM/m など先行するシステムがあった.
   このため PIMOS あるいは KL1 を移植してこれらに問題があるケースが非常に少なかった.
2. VPIM on Symmetry を作ったため, 再現性のないバグが事前に発見された.
   大きなバグの 1 つとしては, 共有メモリマシンの上で実行するには KL1-B レベルに問題があったことがある.
3. 開発支援環境を充実させることは大切である.
4. PIM/p には命令レベルシュミレータがあり, H/W バグ, PSL コンパイラのバグを発見できた.
5. 実機上のデバッグシステムの充実
   デバッグする人と環境を作る人は異なるので, デバッグする側からのフィードバックは大切だった.
   また, 書いた言語でデバッグできることは大切である.

5 予想もしなかったことが役立つこともある.

1. PIM/p の LED がデバッグに有効だった.

2. 標準品を使うことは有効なこともある.
   PIM/p ではコンソールを UNIX で動かしていたのでどこからでもログインできた.

3. 円満な人間関係は大切である.

6 質疑

Q. 回り道をするとしたら, VPIM の 0.5 版あるいは 1.0 版で性能評価をすべきだったのではないか
A. そのとうりです
Q. PIM/p と PIM/m の性能評価について, 5 割以上の損は免れたというはなしをもう少し詳しくしてほしい.
A. PIM/p が 1 に対して PIM/m が 1.89.
マシンサイクルで 23%, 共有メモリ対策で 10% ロスしているとすると 1 : 1.3 で 30% 以内で収まっている.
Q. もっと解析してはどうか
A. 開発者のスキルなどが違う, また実装のレベルが違うなどの要因があるので比較することは容易ではない.
Q. KL1-B は共有メモリでは動かないということは前もってわからなかったのか
A. メモリアクセスの際のプロセッサの競合に関するクリティカルな問題だったのでわからなかった.

ノウハウ捧げますスライド

Final Grand PIM-WG

「今思えばこうするべきだった」

そして

「後世に残すとしたらこの技術」

1993年 3月 23日

シャープ(株) 技術本部 情報技術研究所

今井 明 imai@shpcsl.sharp.co.jp

プロセス制御, Xモリ管理に限らず, システム設計から完成に至るまでを振り返り, 私の持つノウハウすべてを捧げます

1987	7. ICOT着任	↓ 仕事していない空白の1年
1988	夏. VPIM構想着手.	
	12. FGCS'88 (Multi-PSI Debut)	
1989	VPIMの開発.	
1990		
1991	3. VPIM ver. 1.0 Release	
1992	6. FGCS'92	↓ PIM/pついての1年.
1993	12. ICOT クビ	KL1 history は終ったわけではない

1

システムがひととおり動くようになり少しばかりのTuningを終えた今思う正直な感想

- ◎ 動いてよかった …… ホッ
- ◎ ひととおり動くものを作ることは大事
  - ・批判ばかりで先に進まなければ無意味
  - ・ひととおり作ったからこそ言えることがある
- ⊙ 動くものを作ることと, 効率良く動かすことは別物
- ⊙ 今のKL1システムで「KL1の限界」は語れない
  - ・もっともっと評価を
- ⊙ 他の事例は直接役立たないが, それにしても思想の輸入, 他流試合が少なかった。
  村上先生の言葉は重かった.
- ⊙ 今後,「KL1システム」を他の事例として見ることは役立つと信じる.
- ◎ ICOTから離れて冷静に見つめていたい. もう少しわかることができそう.

2

★ どんな状況にも対応できる interpreter の限界

動的な判定が多すぎる

KL1

```
init_vector(Vect, Idx, Elem, NewVect) :-
    Idx > 0 |
      Idx2 := Idx - 1,
      set_vector_element(Vect, Idx2, _, Elem, Vect2),
      init_vector(Vect2, Idx2, Elem, NewVect).
init_vector(Vect, 0, _, NewVect) :- true |
      NewVect = Vect.
```

C

```
foo(array, size, init_value)
int array[], size, init_value;
{
  int i;
  for(i=0; i<size; i++)
      array[i] = init_value;
}
```

KL1 on PIM/p …… 53 instructions

C on SPARC …… 4 instructions

並列実行可能なコード
並列実行不可能なコード
} を切り分けたコードが必要!!

3

どんな無駄?

Idx, Idx2, Vect に対する
- ・デレファレンス必要性の判定
- ・望ましいタイプへの具体化判定
- ・index の overflow 検査

並列言語としての同期
未定着なら中断

高級言語としての親切

```
f()
{
  int a[10];
  a[20] = 100;
}
```

cc -O　　kc -O
cc -g　　kc -g

4

★ RISC ? CISC ?

・ひととおり動作するものを作るまで

・Optimize (特に Global Optimization) の Phase.

1987年夏. 中島克人先生の予言（預言??）

「... 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方は, 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があるが, 逆に下手をすると 5割以上の速度損出になる可能性がある。」

PIM/p (RISC風)	PIM/m (CISC)
1 : 1.89	
-) マシンサイクル 23%	
-) 共有メモリ対策 10%	
1 : 1.3	

Global Optimization に手を染める Phase では RISC の方が柔軟性が高い.

ノウハウ捧げますスライド

★ ヒープベース メモリ管理の限界

1. 割付・回収が複雑でコスト高
2. フラグメンテーション・メモリアクセスの物理的ローカリティの問題
3. 管理すべきポインタの数が多い (freelist management).

逐次性の高い部分を抽出し.
基本処理は Stack Base.
他を Heap Base.

★ 稼働率優先の動的負荷分散

そんなに悪くない.
今すぐ改良すべき点ではない.

★ Copy GC の 並列実行.

頑張ったけど, むくわれていない.
一括GC処理は bottle neck ではない.

★「動的」に頼り過ぎた.

「構造体定数」は成功例.
他に適用できないか??

・構造体定数はなぜ正解か?

p(X) :- true | X = [a,b,c,d,e].

・動的に構造を割り付けるのでない.

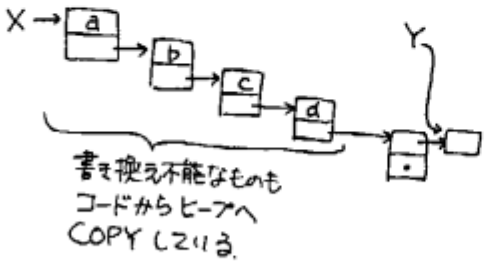
{ メモリ容量・局所性
  手間

・なぜうまくいったのか?

・値の決ったものへの write は起きない.
  ↓
  share で充分

・他に適用できないか?

p(X) :- true | X = [a,b,c,d,Y], g(Y).



★ メモリ容量を節約?
処理時間を節約?

柔軟に対応してきた.
模範回答
一貫性がなかった.

★ なんでもガリゴリ書き過ぎた.

Flat GHC 処理系 (ガリゴリ)

荘園によるメタコントロール (ガリゴリ)

共有メモリ 並列実装 (ガリゴリ)

非共有メモリ 分散実装 (ガリゴリ)

論理の枠組に留まった SCSI 操作組込言語 (ガリゴリ)

UNIX (ガリゴリ)

TCP/IP
NFS (ガリゴリ)
pty (ガリゴリ)

[反省に基づく micro-kernel]
Mach
Amoeba

[反省に基づく]
▽PIM light

ノウハウ捧げますスライド

★では今後生き残る技術は？

・のではないかと思える
・って欲しい
・んじゃないかな．まちょっと曖昧はしろ

基本的なアイディア
（基本的であればあるほど適用範囲は広い）

1. 分散環境における 有・悪 の判定に weight を用いる方法．
2. 共有リンク・共有ハッシュの操作法．
3. 論理的意味 …… 物理的実装．
4. 構造を2の冪乗で丸めて割付
5. ソフトウエアコントロール Cache．

システムをデバッグしてみて

得られた 真理 / 知見 / 技術 / ノウハウ → は何か？

急がば回れ！

PIM/p の完成まで．
多いに回り道をした。

決して無駄ではなかった。

「あの頃の僕達は 回り道をしようと
思っていたんじゃなかったよね」

回り道1． 他に先行したKL1システム

PDSS（UNIX上の byte code interpreter による逐次処理系）
Multi-PSI，PIM/m

で動くKL1プログラムが 自分達のシステム で動かなければ，疑うべきは自分達．

PIMOSを含めて，KL1に問題のあるケースが非常に少なかった．

・実装依存部をKL1の言語仕様と思い込む
・@processor の解釈の違い
・mergeの使い方

回り道2． Symmetry上のシミュレータ

PSL → C on DYNIX

・再現性のないバグが頻発．
・共有メモリで正しく動作しない KL1-B
・FEPを繋いで PIMOSデバッグ

回り道3：開発支援環境 for VPIM

M-x find-tags 相当

回り道4：PIM/p命令レベル Simulator

・ハードウエアの不具合発見
・PSLコンパイラの不具合発見．

回り道5：実機上のデバッグシステム充実

・時間，人をかけて正解．
・必要な機能はデバッグする人のみわかる
→ 即座に feed-backできる態勢
・書いた言語でデバッグする．
Append システム

世間並の環境はないとやはり辛い!!

これだけ回り道をして，3年以上かけないといけないシステムって何なの？

ノウハウ捧げますスライド

予想もしなかったことが役立つ
ことだってある!!

- PIM/pのLED
  デモ効果のみならずデバッグに有効
- 標準品
- 人間関係

皆様 大変お世話になりました。

VPIM の反省としての KLIC

発表者: 近山 隆 (ICOT)

日時: 11:00 ～ 11:30, 3/23

議事録担当: 平野 喜芳 (ICOT)

1 5G は「小さな」プロジェクト

- 全部作っていられない
  → いままでは全部つくってきたので世の中に遅れてしまった。
- あるもの を使う
  「汎用」計算機システム (WS, 並列 UNIX システム)
  既存の OS (UNIX)
  既存の開発環境 (gcc, gdb, prof, ···)

```
KL1 —[コンパイル]→ C —[コンパイル&リンク]→ a.out
                                libkl1.a →┘
```

そして 5G の成果 (言語, 言語実装技術, PIMOS, 応用ソフト)

2 C にコンパイルする方式の利点

- 移植が簡単 (パソコンでも 64bit マシンでも OK)。
- 低レベルの最適化はCコンパイラに任せられる。
  → 最近のCのオプティマイザは非常に賢い。人間がアセンブラを書くより良いコードを出す。
- 既存のプログラムとリンクして使える。→ KL1 の特性を生かした他言語インタフェース (状態を持てるようにする)。

3 試験的実装の性能評価

- Naive Reverse の実行速度は以下の通り。

System	Speed	Code Size	Data Size
Symmetry S81	118 KLIPS	608 bytes	16 bytes
Sun-3/260	205 KLIPS	532 bytes	16 bytes
SparcStation 1+	580 KLIPS	640 bytes	16 bytes
SparcStation 2	985 KLIPS	640 bytes	16 bytes
SparcStation 10/30	2,000 KLIPS	640 bytes	16 bytes
DEC alpha 7000	3,701 KLIPS	?	?
SICStus 0.7 (byte code)		272 bytes	

- 実行サイズはヒープサイズによって変わる。
  ヒープサイズ大 → ワーキングセット大 → キャッシュに乗らない → 遅い!
- ワーキングセットをキャッシュに乗せることが重要。
  … 早いプロセッサほど相対的にメモリが遅いのでキャッシュは重要。
  → Generation GC が欲しくなる。
- マルチプロセッサでは PE 間通信 (特にレスポンス) の遅さを考慮する必要あり。
  … 早いプロセッサほど相対的に PE 間通信が遅い。
  → バッファリングによる通信頻度低減。
  → 多レベル一括転送 (静的解析により必要な部分を無駄なく)。

4 PIM 上の処理系から受け継ぎたい技術

- Pure な言語の特徴を生かした実現手法
  - データのコピーは自由に持てる。
  - 実行順序は原則自由。
- 分散実装の諸方式
  - Weighted Reference Counting.
  - 分散メモリ管理。

5 PIM 上の処理系から受け継がない技術

- MRB
  - ハードサポート無しでは管理コストが高い。
  - フリーリスト管理 (ワーキングセットが大きくなる) より GC のが安い。
- 荘園間でのデータの単純な共有
  - 資源 (CPU 時間, メモリ量) 管理を UNIX に頼り難い。
  - プロセッサ間通信が複雑化する。
    (荘園自体は必要 …プログラミング上最低限必要な機能は残す)
- 共有メモリモデル
  - 単に面倒だから … もあるが、局所性を重視するため。
- KL1 言語仕様の墨守
  - なるべく同じプログラムで動くようにするが、性能に対する影響が大きければ仕様変更する部分もある。

6 開発時期

'93 年 3 月	単一プロセッサ版の評価用処理系
'93 年 9 月	単一プロセッサ版の試験リリース
	複数プロセッサ版の評価用処理系
'94 年 3 月	複数プロセッサ版の試験リリース
～	ソフトウェア開発環境の充実
	最適化 (システム依存の最適化を含む)

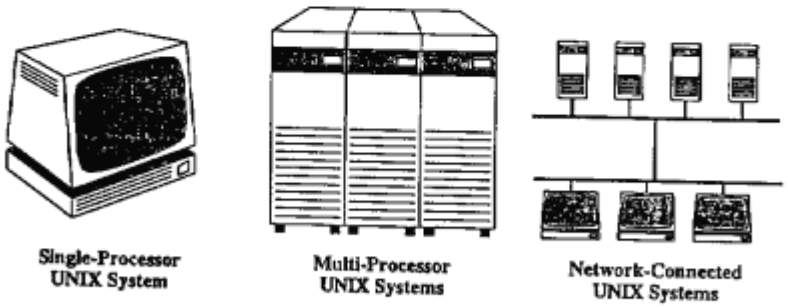
7 議論

- 再帰呼び出しのプログラムで無駄なデレフを削るような最適化はするのか？
  → あまり姑息な最適化はしない。無理に入れるとバグが入り失敗する。
- '95 年以降の予定は？
  → Klic コンソーシアムのようなものを作りたい。
  問題は事務上の雑事をどうするか。
- ベクタ処理を入れるか？
  → 現在の KL1 はひ弱。できるだけ入れるようにしたい。
  ああ無情, レ・ミゼラブルも言っている「ベクトルゆうごう！」

UNIX 上の KL1 処理系

近山 隆
ICOT 第一研究室

対象とするハードウェア



3

5G は 小さな プロジェクト

→ 全部作っていられない (作っていると世の中に遅れる)

→ あるものを使う

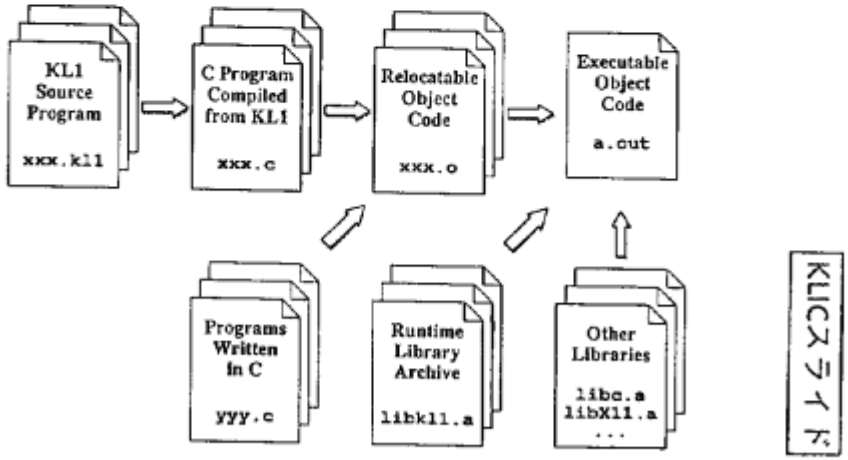
「あるもの」

- 「汎用」計算機システム (WS, 並列 UNIX システム)
- 既存の OS (UNIX)
- 既存の開発環境 (gcc, gdb, prof, ...)

- 5G で作った言語, 言語実装技術, PIMOS, 応用ソフト

2

実装の基本方式: KL1 から C にコンパイル



KLICスライド

4

C にコンパイルする方式の利点

- 簡単に移植できる
- 低レベルのの最適化は C コンパイラに任せられる
- 既存プログラムとリンクして使える
  KL1 の特性を生かした他言語インタフェース

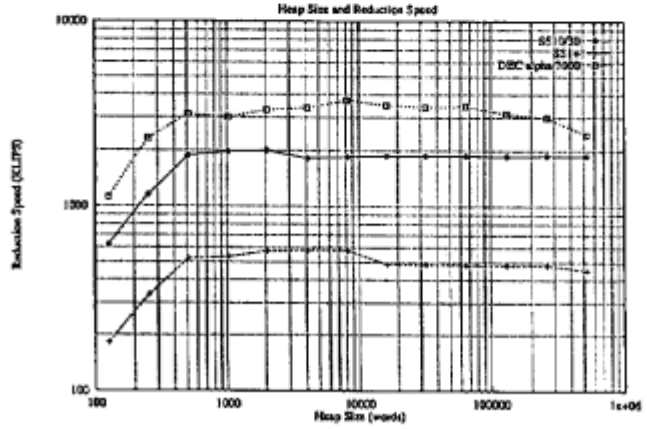
5

試験的実装の性能評価

System	Speed	Code Size	Data Size
Symmetry S81	118 KLIPS	608 bytes	16 bytes
Sun-3/260	205 KLIPS	532 bytes	16 bytes
SparcStation 1+	580 KLIPS	640 bytes	16 bytes
SparcStation 2	985 KLIPS	640 bytes	16 bytes
SparcStation 10/30	2,000 KLIPS	640 bytes	16 bytes
DEC alpha 7000	3,701 KLIPS	?	?
SICStus 0.7 (byte code)		272 bytes	

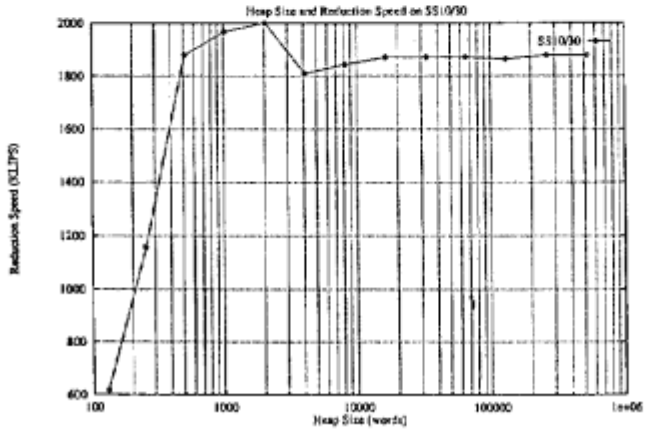
6

ヒープサイズと naive reverse の性能の関係



7

ヒープサイズと naive reverse の性能の関係 (SS10/30)



8

KLICスライド

プロセサが速くなった

→ (相対的に) メモリが遅くなった
  → Generation GC (オーバヘッド 10 ～ 15% か)

→ (相対的に) 通信が遅くなった (特にレスポンス)
  → バッファリングによる通信頻度低減
  → 多レベル一括転送 (静的解析で無駄なく)

9

開発時期 (見込み)

'93 年 3 月	単一プロセサの評価用処理系
'93 年 9 月	単一プロセサ版の試験リリース
	複数プロセサ版の評価用処理系
'94 年 3 月	複数プロセサ版の試験リリース
～	ソフトウェア開発環境の充実
	最適化 (システム依存の最適化を含む)

10

従来のPIM上の処理系から受けつがない技術

- MRB　- ハードサポートなしでは管理コスト高
  - フリーリスト管理よりGCが安い.
- 「荘園」間でのデータの単純な共有
  - 資源管理をUNIXに頼りたい (CPU時間, メモリ量)
  - プロセサ間通信が複雑化している
- 共有メモリモデル　- 単に面倒だから. もあるが
  - 局所性の重視 (たとえばcache sizeを意識した世代GC)
- KL1言語仕様の墨守

11

PIM上の処理系から受けつぎたい技術

- Pureな言語の特徴を生かした実現手法
  - データのコピーは自由に作れる
  - 実行順は原則自由
- 分散実装の諸方式
  - Weighted Reference Counting
  - 分散メモリ管理

12

KLICスライド

ハードとソフトの役割分担

発表者: 近藤 誠一 (三菱), 平野 喜芳 (ICOT)
日時: 11:30 ~ 12:00, 3/23　　議事録担当: 高橋 勝巳 (三菱)

1 PIM/m ～ハードとソフトにはさまれ複雑化していった処理系～ (近藤)

1. KL1 はきれいに見えても
   - 優雅に湖面をゆく白鳥のごとく
     水面下では、ファームウェアがドタバタしながら実行している。
2. PIM/m がサポートしている言語
   - PIM/m には、KL1 とマイクロアセンブラの 2 つの言語がのっている。
   - マイクロアセンブラは、結構書き易い。
     レジスタの代入も、ニモニック記述で変数代入に見えてしまう。
3. なんでもファームウェアでやってしまった
   - KL0/KL1 で{書けない | 性能が出ない} 部分は、どんどんファームウェアに入れられていった。ファームウェアの制御記憶容量に対する比率
     Multi PSI (kl1)　95.7% (3D38/4000 語)
     PIM/m (kl1)　71.0% (5AD6/8000 語)
     PSI/UX (kl0)　78.9% (6503/8000 語)
   - 処理系に負担をかける悪い伝統が、VPIM にも…
     → VPIM 巨大化の悲劇。原因はここにある？
4. 巨大化した処理系への反省
   - 論理にこだわり過ぎたのでは？ KL1 は、もっと低レベルのものの方が、よかったのではないだろうか？
   - ファームにどんどん機能が入ってくる時にストップがかけられれば…。
   - ファームにもっとマンパワーがなければ。→ 機能は厳選された？
5. 先行ゆえの悲劇
   - PIM/m の場合、すでにユーザがいて、確実に動くことが要求されていた。新しいセマンティクスを入れるためには、無理をしなければならなかった。
   - Multi-PSI, PIM/m はオブジェクト互換である。ハードの違いを処理系が吸収するというのは、開発途上のシステムでは、(確実に動作させるという意味で) 強力であった。しかし、処理系の発展性も殺してしまった。
6. 討論から
   - Multi-PSI の時には RISC 型命令を入れたことがあった。その結果は、LOAD でも 3clock かかるというものだった。

2 PIM/p ～ PIM/p のプロセッサに用意された特殊命令の効果 ～ (平野)

1. PIM/p チップは普通のチップ。ちょっと手を加えて早くした。
   - PIM/p では、普通のプロセッサに、タグアーキテクチャ等 5 つの機能を追加した。
2. 5 つの機能の評価評価はシミュレータ (1PE, キャッシュ無し)
   - タグアーキテクチャ
     ハードウェアがなければ、性能は 50 使用率: 40 ～ 50%, 一般命令換算: 2 命令

- タグ部にあるデータ型の情報を判定する条件分岐命令
  コンパイラの頑張りで判定は減らせる。ハードがなくても性能は落ちない?
  使用率: 10～15%, 一般命令換算: 2～3 命令
- タグを判定する条件サブルーチンコール命令
  あまり使われていない。今後のチューニングによって使用頻度が増えるかも知れない。
  使用率: 2～5%
- MRB を OR しながらメモリを読む命令
  MRB-GC をサポートするための機能。これもコンパイラの頑張りしだい。
  使用率: 6～18%, 一般命令換算: 3～4 命令
- リスト操作命令
  ソフトで工夫するのは無理。3～20% の性能減
  使用率: 3～10%, 一般命令換算: 2 命令

3. KL1 機械として

- 現状のソフトでは、2 倍の命令を実行しなければならない。
- ソフト (コンパイラ等) の頑張りでは、1.5 倍から同じ数の命令で実行できるようになる？
  → とすれば、専用プロセッサのご利益ってなんだろう？

4. 討論から

- あまり考えてないのだが、タグ付きというのは必要だったのか？
- 処理系にしわ寄せが来たのは、OS や言語がはっきりしていなかったからだろうか？
- いろいろな悪影響を及ぼしている MRB-GC は、なぜ処理系に入ったか？
  - なぜ入ったか？
    PIM/p: VPIM に入ってたからしょうがなかった。
    VPIM: PIM/m に入ってたからしょうがなかった。
    PIM/m: Multi-PSI に入ってたからしょうがなかった。
    Multi-PSI: 入れても遅くならないし、みためが面白かったから入れてしまった。
    → PIM/m では、キャッシュに当たらず悲劇に
  - いいところはないのか？
    確かに、キャッシュやページングとは相性が悪い概念だった。しかし、ダイナミックにメモリをアイデントする面白い概念である。

役割分担スライド

ハードとソフトの役割分担
～ ハードとソフトにはさまれて複雑化して いった処理系 ～

近藤 誠一 (三菱電機)

1

PIM/m, Multi-PSI の処理系

Multi-PSI　53 ビット水平型マイクロ
　　制御記憶容量は 16 K 語

PIM/m　64 ビット水平型マイクロ
　　制御記憶容量は 32 K 語

マイクロ総量

multi-PSI	(kl1)	15,672	(3D38/4000)	95.7 %
PIM/m	(kl1)	23,754	(5AD6/8000)	71.0 %
PSI/UX	(kl0)	25,859	(6503/8000)	78.9 %

2

マイクロソース

```
:: /'*************************************************************'.
:: /'*  +------+-----------+-----------+----------+----------+ *'.
:: /'*  | KBLT | Opcode 361 |   arity    |        label        | *'.
:: /'*  +------+-----------+-----------+----------+----------+ *'.
:: /'*************************************************************'.
:: /'$ KLB_execute:                                             $'.
:: /'$  ADR                                                     $'.
:: /'$    1_word , jump , mar := PC + ir/s16 ;                  $'.
:: /'$  DRD                                                     $'.
:: /'$    mdr := mar ;                                          $'.
:: /'$  DPT                                                     $'.
:: /'$    (mar2 , mdr2) := mdr , mdr1 := CRR ; .                $'.
:: /'*************************************************************'.
KLB_execute:
        CRR := mdr1 - 2 , yr <- 3 , cond(Slit_Check_exsw04) .
               %% CRR = CRR_mdr1 - 2
        := mdr1 - yr , cond(sign) , xr <- PSTR , reset_SUSPF_sw4 ,
            if (cond) call Sub8_kl1_spy_check_in_exec .
                %% check CRR =< 2 , Suspension Stack Flag OFF
                %% return : saved mdr2 , := CRR - 1 , cond(sign)
        PC := COD! mdr2 , xr <- PC , cond(interrupt_all) ,
            if (cond) goto $enqueue_and_proceed .
                %% PC = mdr2 (PC + IR/S16)
        if not (cond) eop , r04 := xr , reset_CGCF_sw3 .
               %% Current Goal Content Flag OFF
               %% i_fetch and eop are suppressed if ZERO
```

3

巨大化した処理系

- 機械語のレベルが高い。
- 選択肢は kl1/kl0 とファームウェアしかない。
  (論理型言語のみに固執してしまった？)
- 多少複雑な処理でも充実したファームウェアのスタッフによりこなしてきてしまった。

VPIM に対して悪い影響を与えてしまった？

4

PIM/k

発表者: 酒井 浩, 仲瀬 明彦 (東芝)
日時: 1:30 ～ 2:15, 3/23
議事録担当: 畑澤 善宏 (富士通 SSL)

1 全体の進め方として良かったこと・悪かったこと (酒井)

1. 良かった点

- スペックをほとんど変えずに済んだ
- 少人数で出来た
- 評価に値するレベルに達した

2. 悪かった点

- ハードを最初に作り過ぎた
  TAXI によるリングネットで半年ロスした
- VPIM から離れられなかった
  性能評価に基づくチューニングが後回しになった

2 性能評価について (仲瀬)

- VPIM ベースの処理系を最適化しつつ手動でコーディングした
  - VPIM が PIMOS 対応で大きくふくれた時期には、機能追加についていけなかった。
- PIM/p との比較
  - クロック周波数、バス幅、オブチマイザがないことといったハンデを考慮すれば、PIM/p の 45 %程度という性能はまずまず。
  - チューニングで、あと 10% 程度は頑張れるか。
- プロファイル収集ツール
  - 複数 PE では、メモリリードのレイテンシが大きいことが分かる。
  - 複数 PE での処理別クロック数を見ると、ゴール送信などが大きな割合を占める。アイドル PE を示すためのビットマップにアクセスが集中していた。
    → この部分の改良によって、ゴール送信以外の処理もクロック数が減った。
- 階層構造 (ミニクラスタ) と負荷分散
  - pentomino などでは 12PE1CL とするより、1PE12CL とした方が速い。→ 自動負荷分散のチューニングが必要
- まとめ

  処理系のチューニング
  - アーキテクチャの評価にはもう少し使える
- 質疑
  - 階層キャッシュ経由の通信は PIM/m のメッシュ経由より速いか？
    → キャッシュ間では 1 バイトでも 40 クロック程度かかる。PIM/m で隣の PE までは 8-10 クロック。パケットの生成まで含めれば速くなるのではないか。
  - データのパーティショニングを考えれば速くできる。
    → 現状では自動負荷分散よりも、プラグマで負荷分散を指定した方が速い。

- 共有メモリを使ったマシンの難しさがある。台数によって難しさが違う。
- メモリ全体はコヒーレントに見える。
- 階層キャッシュの実装コストは？
  → インプリメンテーション自体は割と簡単だった。上下のバスから同時に要求の来る場合の処理が問題になった。

休憩時間，ICOT を取材した数年前のテレビ番組のビデオを観る

PIM/k ここが良かった、ここが悪かった

（株）東芝　　酒井　浩

1

良かったこと（その1）

スペックをほとんど変えずに済んだ

単一のシステムクロック（100ナノ秒）

16（32）プロセッサ

2階層並列キャッシュメモリ

性能評価機構

良かったこと（その2）

少人数でやれた

他社の後追い

プロセッサの流用

ハードとソフトの境目のデバッグ環境

2

良かったこと（その3）

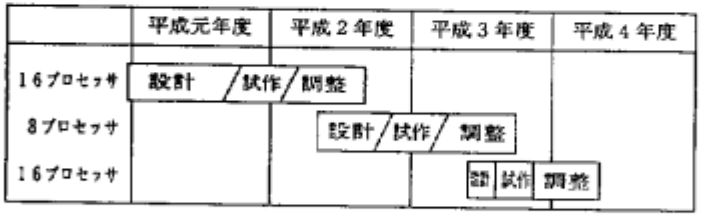
評価に値するレベルに達した

まともに動くハード

チューニングされた処理系

充実した性能評価ツール

悪かったこと（その1）

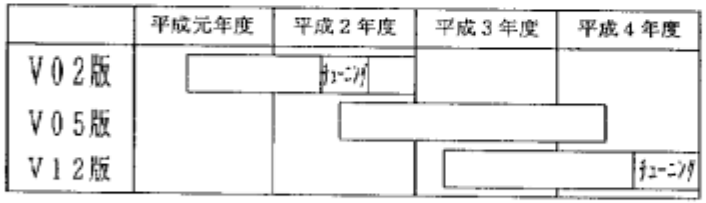


最初に作り過ぎた

ボード調整用の TAXI によるリングネットワーク

3

悪かったこと（その2）



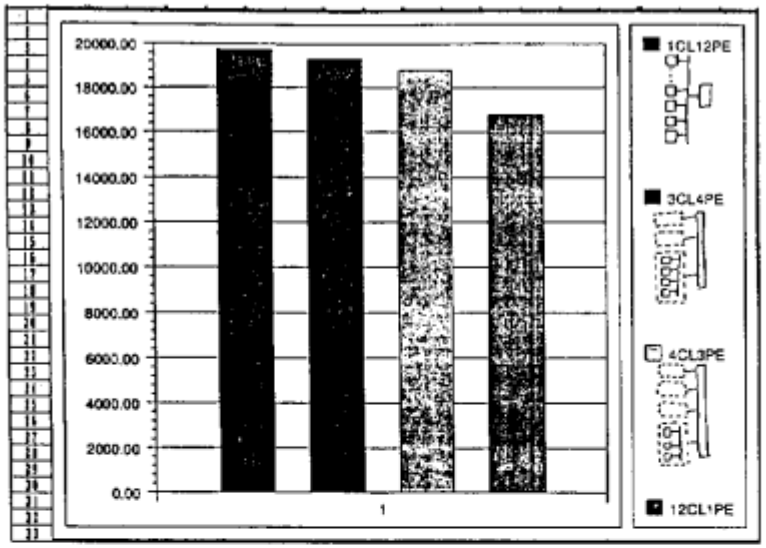
VPIM から離れられなかった

性能評価に基づくチューニングが後回し

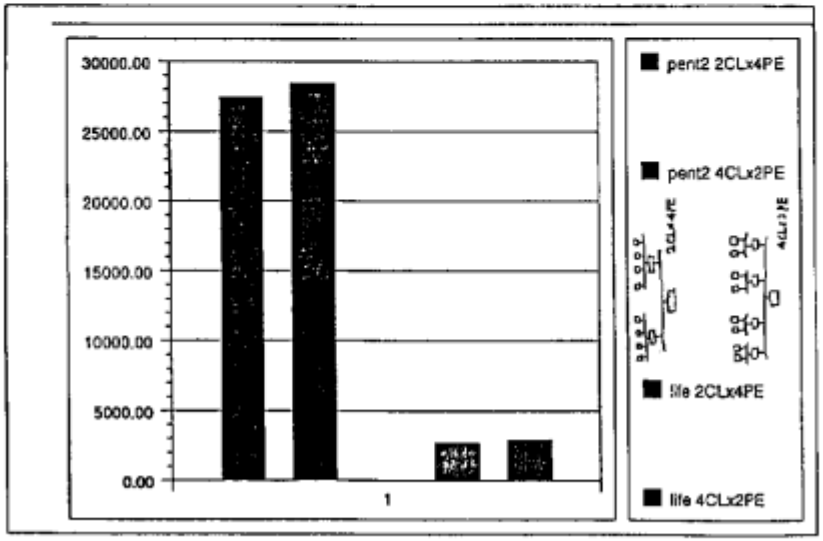
設計者の勘に頼る改良

処理系の改良に結びつかなかったシミュレーション

4



PIM/kを複数クラスタ構成と見た時の実行時間 (マンデル8x5) 5

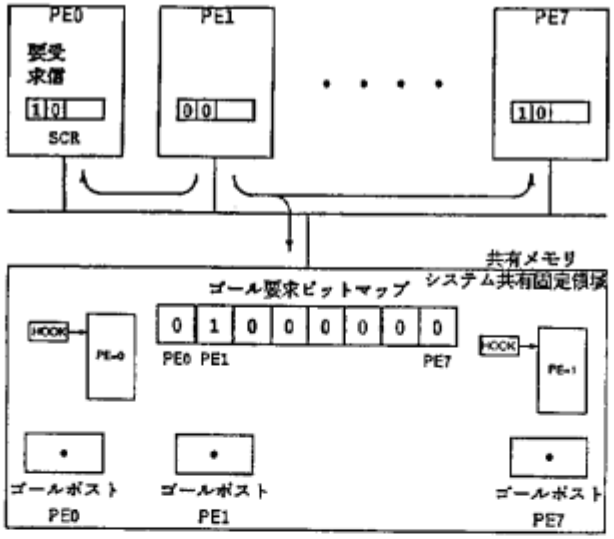


階層構造を変化させた時の実行時間 (ペルリ8x5, ライフゲーム) 7

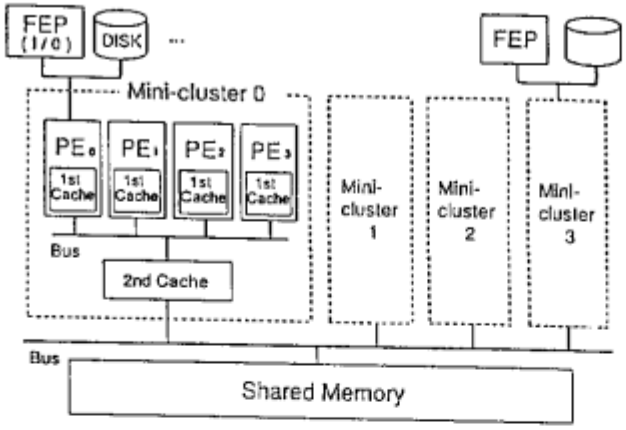
まとめ

- 処理系チューニング
  - 実際の計算と関係ない処理を削減
  - メモリバスを混ませる原因を削減
- アーキテクチャの評価を含めもう少し使える

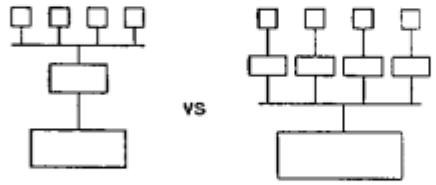
6


PE0
PE1
PE7
要求
SCR
共有メモリ
システム共有固定領域
ゴール要求ビットマップ
0 1 0 0 0 0 0 0
PE0 PE1
PE7
ゴールポスト
PE0
ゴールポスト
PE1
ゴールポスト
PE7

10


FEP
(I/O)
DISK
FEP
Mini-cluster 0
PE0
PE1
PE2
PE3
1st Cache
Bus
2nd Cache
Mini-cluster 1
Mini-cluster 2
Mini-cluster 3
Bus
Shared Memory

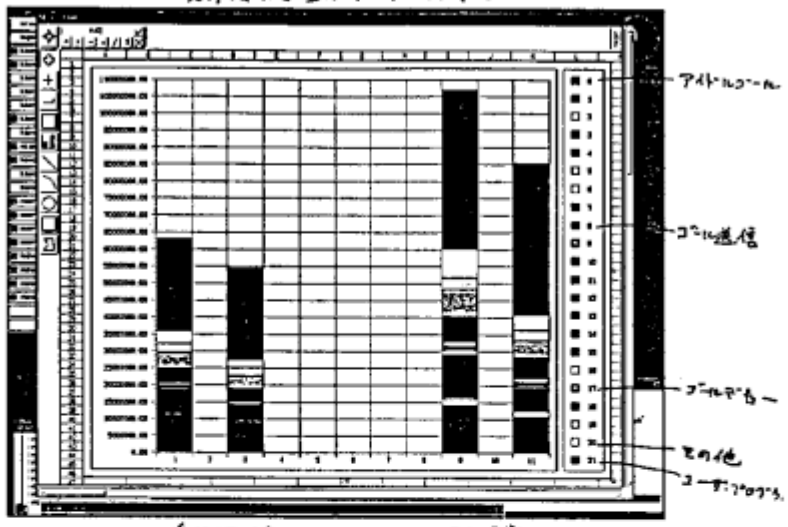
11


VS


VS

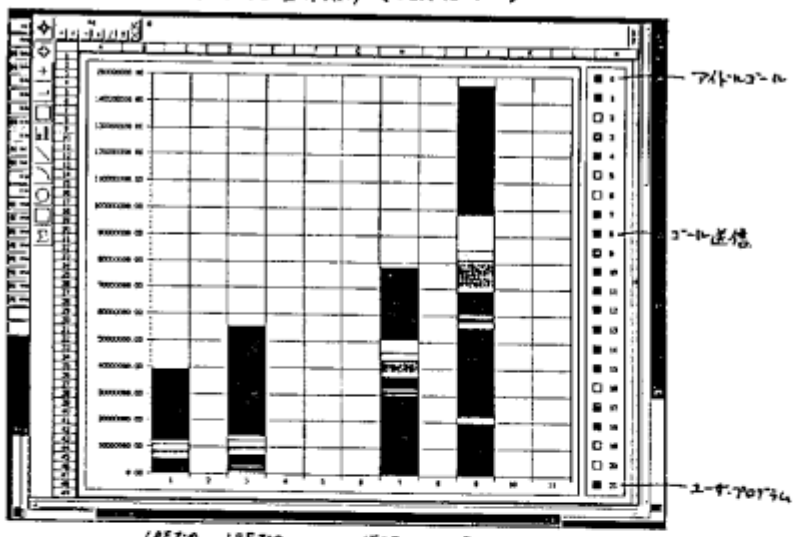
12

ペントミノ5×5実行時の処理の内訳（15PE）

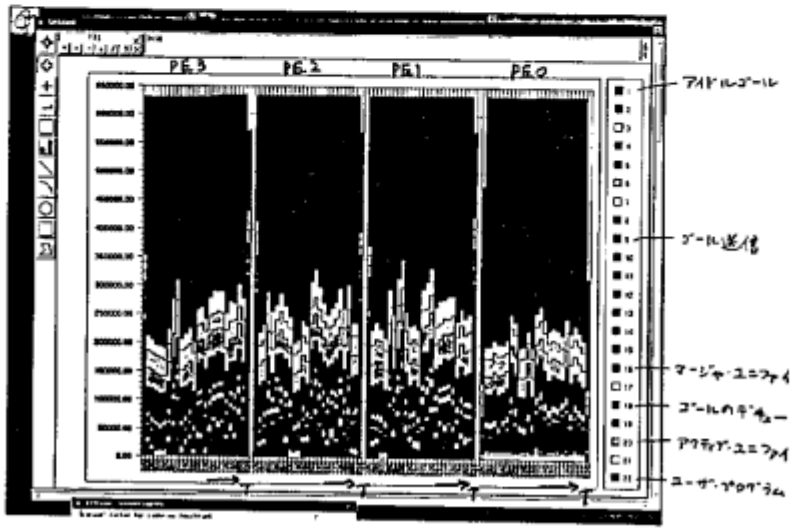


13

ペントミノ5×5実行時の処理の内訳（Ver.930121）

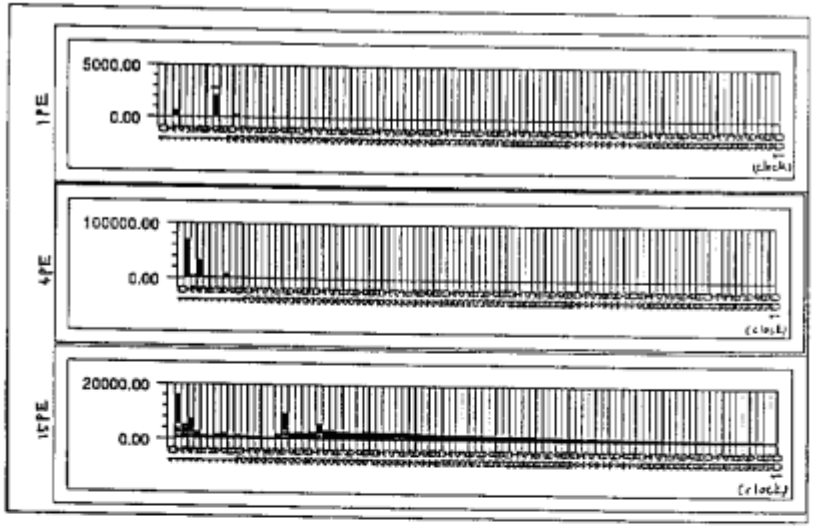


14



ペントミノ5×5での処理のクロック数（4PE）

15



PE台数を変化させた時の共有メモリへのREADのレーテンシ(サイクル)

16

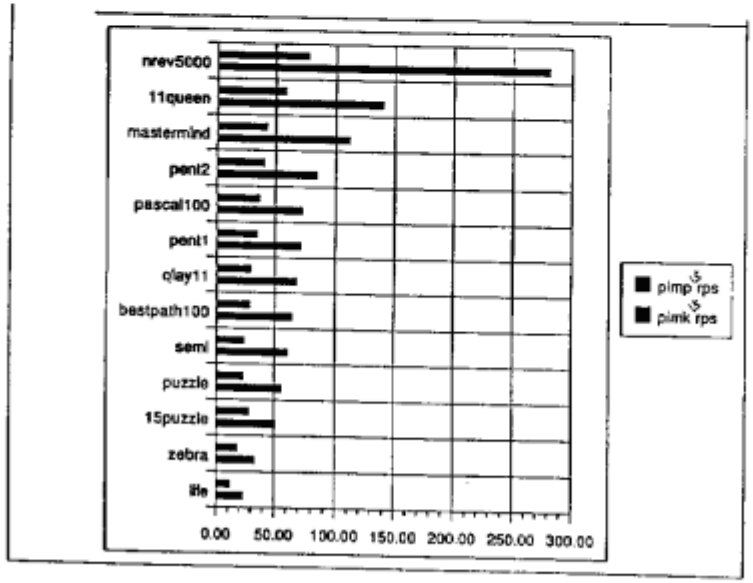
並列実行プロファイル収集ツール

- 各処理別 dynamic step 数
- 各処理別所要 clock 数
- メモリアクセス時のレーテンシ解析

ベンチマークプログラムを用いて並列実行時のボトルネック解析

Ver.930121 初期バージョン
Ver.930205 idle プロセッサビットマップへのアクセスがネック
idle プロセッサビットマップへのロック方式を変更
Ver.930224 ゴール通信処理がネック
HOOK ゴールのリジューム処理方式を変更

17



単一PEによるPIM/pとPIM/kの速度の比較

18

• 処理系作成時の最適化

- 分岐命令における分岐方向の最適化
- サブルーチン間の呼びだし関係の解析とレジスタ割りつけの最適化
- ピープホール・オプティマイゼーション

• 処理系作成後の最適化

- 実行トレース解析による不用命令の削除
- 他機とのメモリ消費量の比較にパフォーマンスバグの修正

19

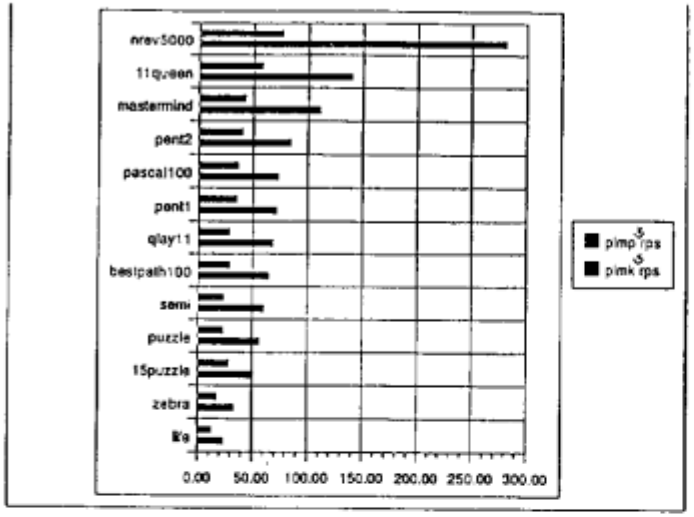
PIM/kスライド

PIM/k における KL1 処理系の評価と改良

(株)東芝 神誠 明彦

単一PE性能最適化

- 処理系作成時の最適化
  - 分岐命令における分岐方向の最適化
  - サブルーチン前の呼びだし関係の解析とレジスタ割りつけの最適化
  - ピープホール・オプティマイゼーション
- 処理系作成後の最適化
  - 実行トレース解析による不用命令の削除
  - 他機とのメモリ消費量の比較にバフォーマンスバグの修正



単一PEによるPIM/pとPIM/kの速度の比較

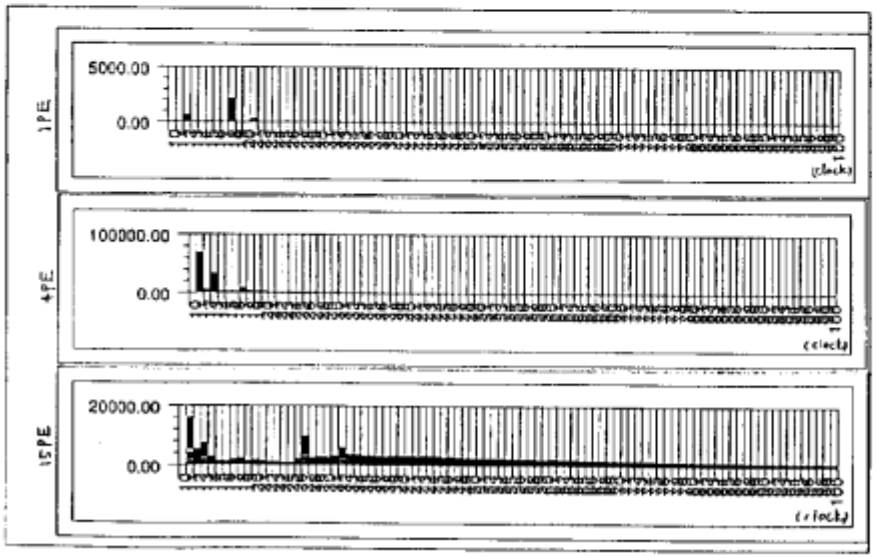
並列実行時最適化

並列実行プロファイル収集ツール

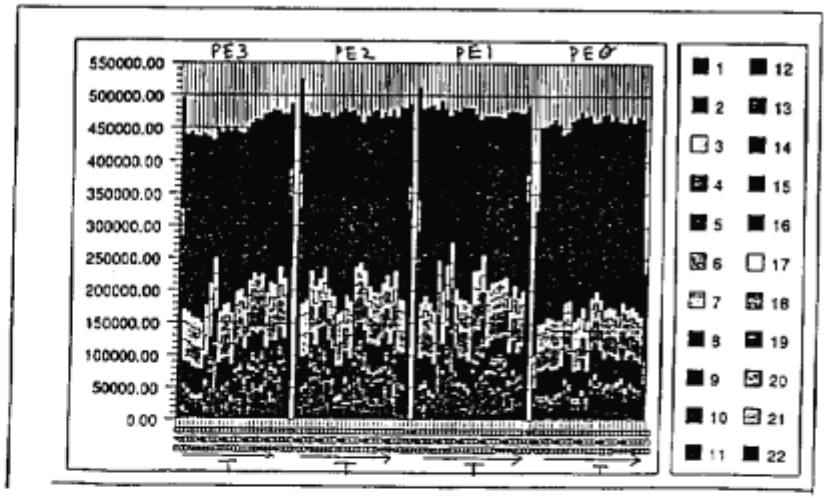
- 各処理別 dynamic step 数
- 各処理別所要 clock 数
- メモリアクセス時のレーテンシ解析

ベンチマークプログラムを用いて並列実行時のボトルネック解析

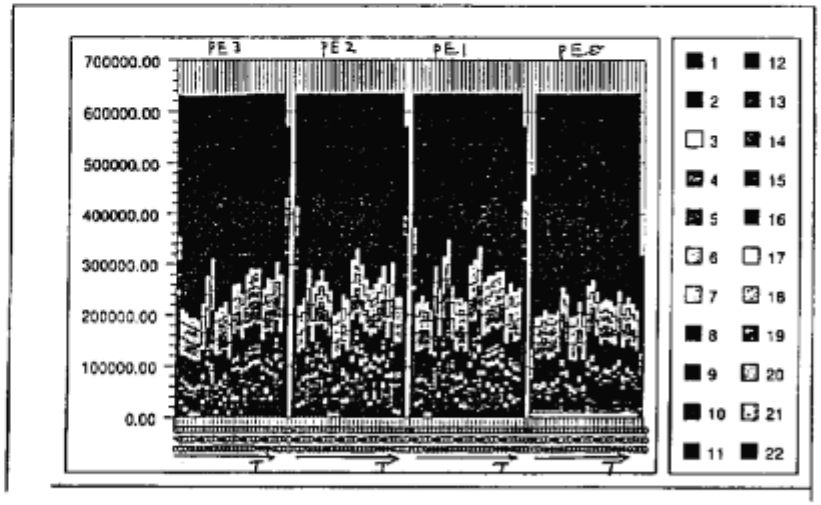
Ver.930121 初版バージョン
Ver.930205 idle プロセッサビットマップへのアクセスがネック
idle プロセッサビットマップへのロック方式を変更
Ver.930224 ゴール送信処理がネック
ROOTゴールの ノジュール処理方式を変更



PE台数を変化させた時の共有メモリへのREADのレーテンシ(PIM/k 5x5)



4PEで実行した時の各処理の動的ステップ数 (ベントミ 5x5)



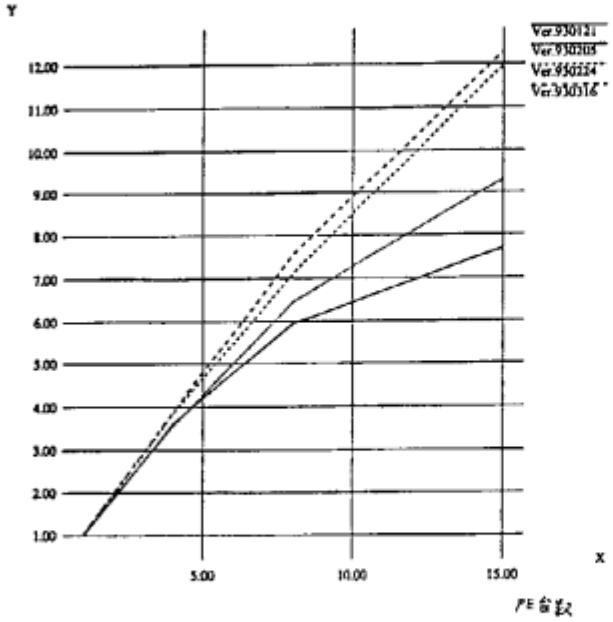
4PEで実行した時の処理のクロック数 (ベントミ 5x5)

各処理の内容

1	IDLE	12	MRB を用いたデータ回収処理
2	DCODE 処理	13	デレファレンス処理
3	1ワードフリーリスト作成	14	サスペンド処理
4	2ワードフリーリスト作成	15	ゴールのエンキュー処理
5	4ワードフリーリスト作成	16	マージャのユニフィケーション処理
6	8ワードフリーリスト作成	17	サスペンド処理
7	16ワード超フリーリスト作成	18	ゴールのデキュー処理
8	スリットチェック処理	19	ゴールのエンキュー処理
9	PE間ゴール送信処理	20	ボディ部のアクティブユニファイ処理
10	PE間ゴール受信処理	21	その他の処理
11	DCODE用ゴール作成処理	22	ユーザプログラム

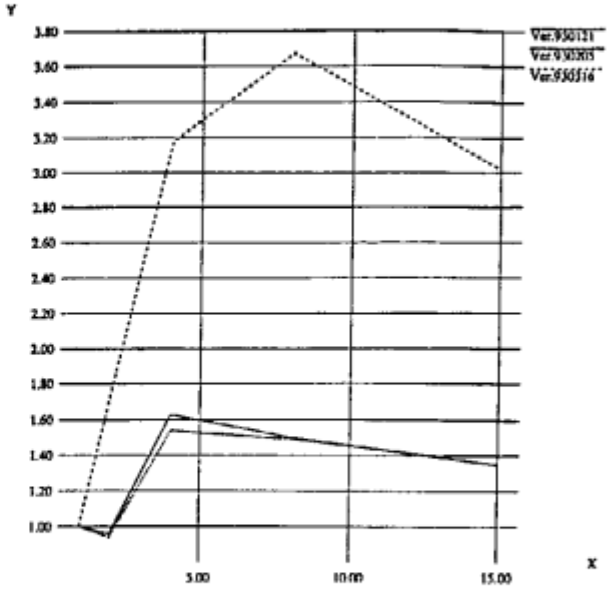
PIM/kスライド

Pent2.8x5 SpeedUp



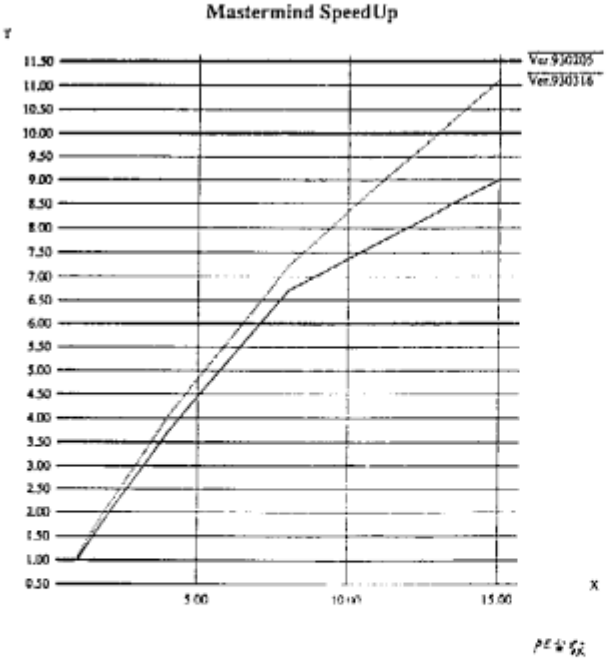
PE台数

LifeGame SpeedUp



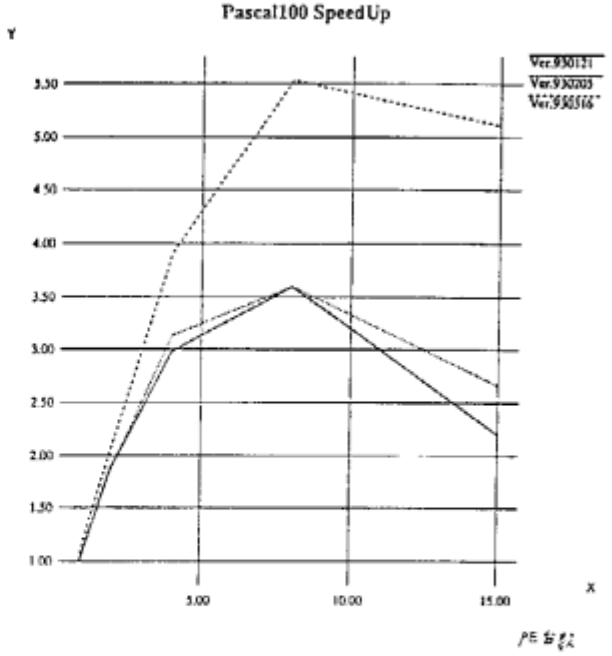
PE台数

Mastermind SpeedUp



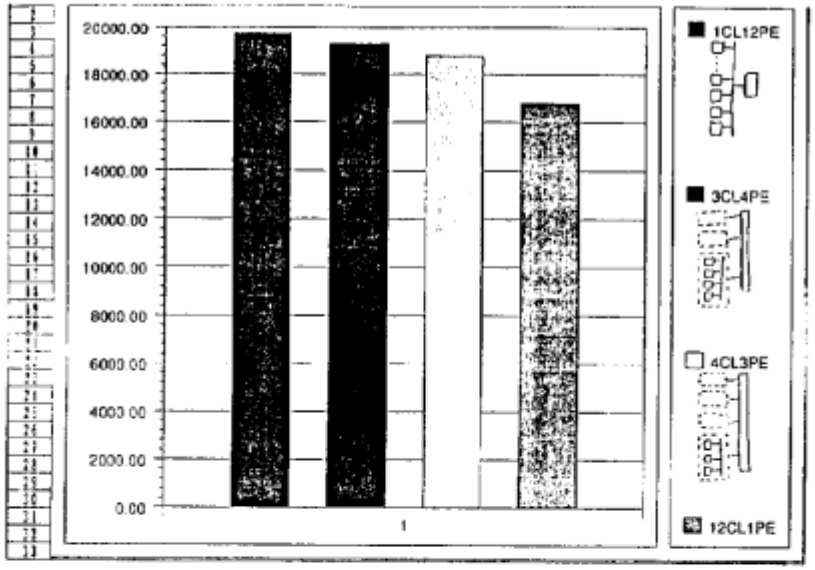
PE台数

Pascal100 SpeedUp



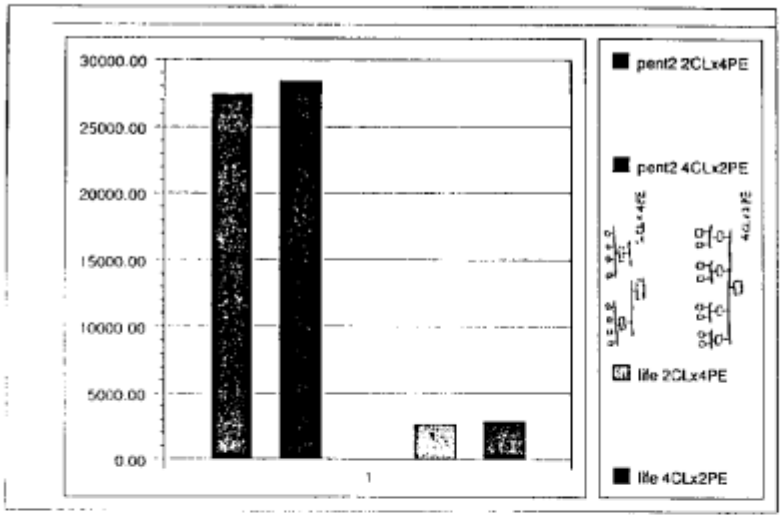
PE台数

4



PIM/kを独立クラスタ構成と見た時の実行時間（マトリクス8x5）

5



静的負荷を変化させた時の実行時間（マトリクス8x5, 5ステップ）

5

まとめ

- 処理系チューニング
  - 実際の計算と関係ない処理を削減
  - メモリパスを混ませる原因を削減
- アーキテクチャの評価を含めもう少し使える

6

PIM/kスライド

PIM/p

発表者: 小沢 年弘, 新井 進 (富士通)
日時: 2:15 ～ 3:00, 3/23
議事録担当: 武井 則夫 (富士通 SSL)

1 アーキテクチャまわりの評価 (小澤)

1. PIM/p アーキテクチャの評価
   PIM/p 独自のアーキテクチャを主に評価した結果を報告する。

2. 評価方法
   評価は、

   - PIM/p 命令レベルシミュレータを用いたベンチマークテスト
   - ライブラリプログラムの静的解析

   のふたつを行なった。

3. シミュレータによるベンチマーク評価
   命令別の動的実行回数を求めたところ、以下のようなことがわかった。

   - ブランチ系命令が全体の 20%。
   - read/write 系命令が全体の 30 ～ 40%。
   - パイプラインブレークのコストが 20% 以上。

   このうち、read/write 命令については、処理系で用いるスタック領域とメモリレジスタへのアクセスが 45% を占めている。これは、処理系側で軽減できないものだろうか。

4. PIM/p 特有命令について
   PIM/p 特有の命令には、平野氏から報告があったように、skip 命令やマイクロメモリを使用したマクロ命令がある。これらの効果を、もしなかった場合と比べてみた。

   - skip 命令はあまり使われていない。
   - MRB の OR つき deref はプログラムにより、使用頻度が異なる。
   - マクロ命令は、2 ～ 10% くらいの効果が見られた。

5. タグ・アーキテクチャ
   read/write 命令、演算命令にタグ操作をともなう命令を装備している。これらの命令について、PSL コンパイラ出力コードのデータの依存関係の静的解析から評価を行なった。

   - read 系命令の出現率は 16% で、うち 80% が読み出したデータとタグの両方を使用している。
   - 演算命令の出現率は 21% で、うち 15% が結果のタグを使用している。
   - write 系命令では、10% がタグとデータの両方が使用されている。

   これより、これらの命令がなかった場合に比べると、read 操作で 12%、演算で 3%、write 操作で 11% の命令増が見込まれる。

2 キャッシュまわりの評価 (新井)

1. PIM/p クラスタの評価
   PIM/p のクラスタは、共有メモリを介して密結合した複数プロセッサから構成される。このため、共有バスの負荷を軽減する目的で装備した、マクロ命令、特殊メモリアクセス命令について評価を行なった。また、台数効果を限定する要因の分析を実験結果から行なった。

2. マクロ命令機構の評価

マクロ命令の評価は、シミュレーションを用いた。マクロ命令の効果については、マクロ命令によるアクセスをキャッシュ命令によるアクセスに変換することで、比較を行なった。評価に用いてベンチマークプログラムでは、1PEで2割程度の効果が見られた。

3. 特殊メモリアクセス命令の評価

A. Read Invalidate 命令 の評価

Read Invalidate命令は、Fetch&Invalidateという機能を実現している。いくつかのベンチマークプログラムをハンドコンパイルし、計測を行なった。"Life"プログラムで、比較的実行頻度の高い部分のみ書き換えると、11%の台数効果の向上が見られた。また、全無効化の3分の1が処理系内で行なわれていた。これは、ひとつのキャッシュブロックに複数のデータがのるため、不必要な共有・無効化が生じるためと考えられる。

B. 他の特殊メモリアクセス命令 について

Direct Write命令の問題点として、以下のことがあげられる。

- 処理系立ち上げ時のヒープ初期化など、使用できる箇所が限られてしまう。
- キャッシュの整合性の維持にコストがかかる。Read Purge命令の問題点としては、以下のことがあげられる。
- 同一ブロック内でのデータの共有により、処理系では使用できない。
- KL1プログラム、処理系双方ともキャッシュ操作をする機構がないため、並列キャッシュの特性を生かし切れないのではないか。

4. 台数効果の阻害要因

台数効果の阻害要因としては、1命令あたりのクロック数(CPI)、命令増、プロセッサアイドル時間、リダクション増による複数プロセッサでの増分が考えられる。これらを測定したところ、CPIが目立って多かった。CPI増加の要因は、共有バス負荷の増加が考えられる。これは、キャッシュの状態遷移とデータのキャッシュミスが大きい。これにより、共有バスの使用率が高くなり、バスにコマンドが出しにくい、キャッシュミスが多いなどの現象が生じる。

5. 質疑応答

- ふつうのRISCマシンでアプリケーションを動かした時のCPIはどれくらいか？
  → プログラムの作りによる。クラスタ内負荷分散により、ゴール間通信がCPU間通信になってしまうこともある。
- キャッシュブロックの大きさについては、現在の4ワードが妥当か？
  → ハードの物量の制限がある。ただし、4ワードは中途半端な大きさという印象がある。
  → 設計段階およびその後のシミュレーションから、2ワード¿4ワード¿8ワードの順で性能がよかったが、実装では処理系立ち上がりや全体性能を考えて4ワードとした。妥当な大きさといえるのではないか。
- サイクルタイムが期待値より低かったが、どの部分がネックになったか？
  → パイプライン・ブレークがもっとも大きかったが、これは回避できる。キャッシュのタグ引きとパイプラインとのやりとりの影響が大きいと考える。

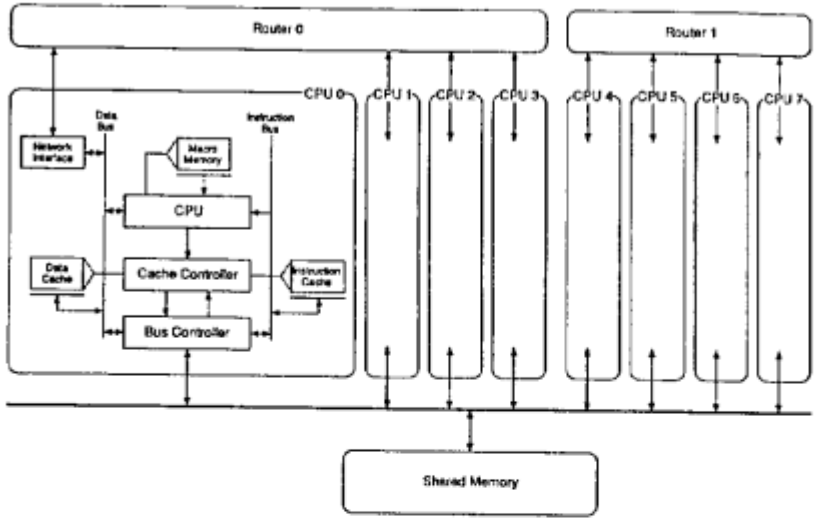
今思えばこうするべきだった

富士通

小沢，新井

PIM/pの tag／data 操作命令

```
rtw R1, offset(R2)        popt R1,offset(R2), TAG
        :
add  R1, R2, R3     /* R1.tag = R2.tag,  R1.data = R2.data + R3.data */
or     R1, R2, R3     /* R1.tag = R2.tag OR R3.tag */
        :
wtw R1, offset(R2)    w8t  R1, offset(R2)    pusht R1, offset(R2), TAG
```



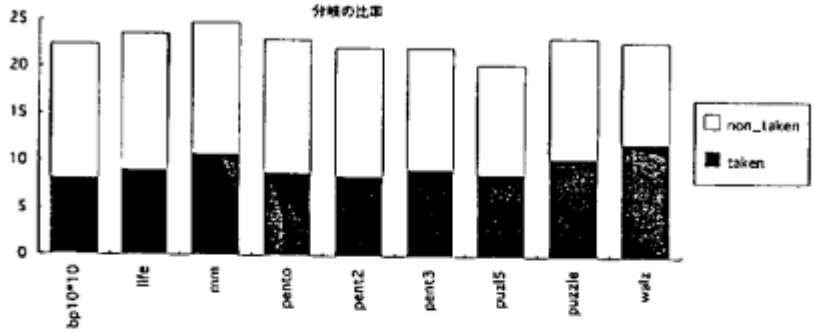
PIM/p クラスタの構成

命令の比率（静的）

命令の種類	出現頻度（%）	備考
branch	22.5	条件分岐の分岐条件の５６％は、レジスタのタグ部による。
read	16.0	１５％は、tag,data共に読みだす。うち８０％が両方のデータを使用。
write	11.1	１０％は、tag,data共に書き込む。
calc	31.2	２１％は、tag,data共に操作。うち１５％が両方のデータを使用。
move	8.3	
others	10.9	

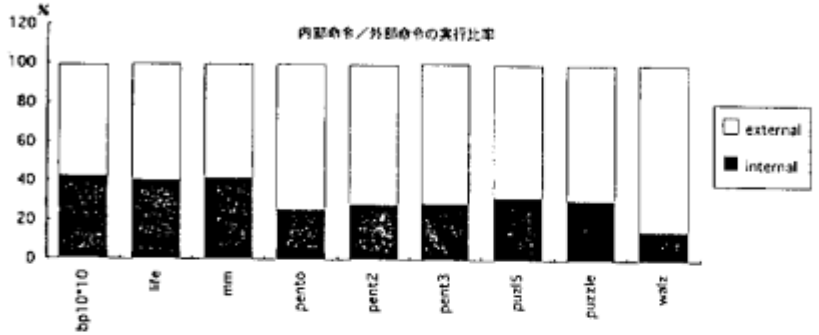
PIM/pスライド

グラフ分岐の比率



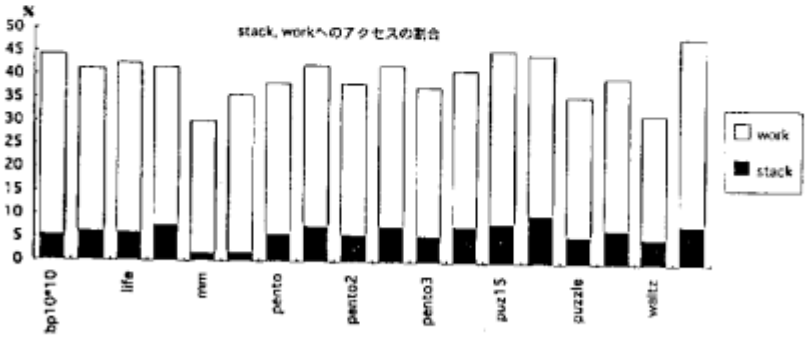
5

グラフ内部外部比率



6

グラフアクセスの割合



7

PIM/p特有命令

1. SKIP系
   条件を判定し、成立すれば続く1命令をNOPとして実行。
   通常の分岐命令に比べて、分岐が2クロック速い。

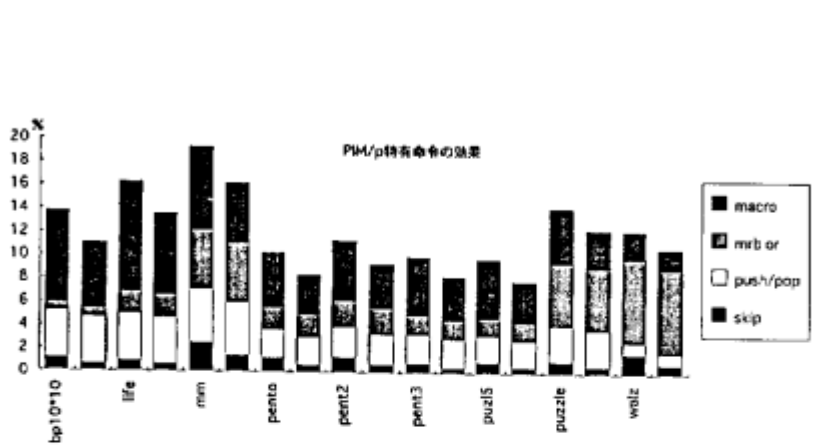
2. PUSH/POP系
   pushtw R1, offset(R2) = wtw R1, offset(R2) + mvtw R2, R1
   poptw R1, offset(R2) = mvtw R2, R + rtw R2, offset(R2)

3. MRB OR 系
   rtwm R1, offset(R2) = andi R0, R1, 0x80 + rtw R1, offset(R2) + ot R1, R1, R0

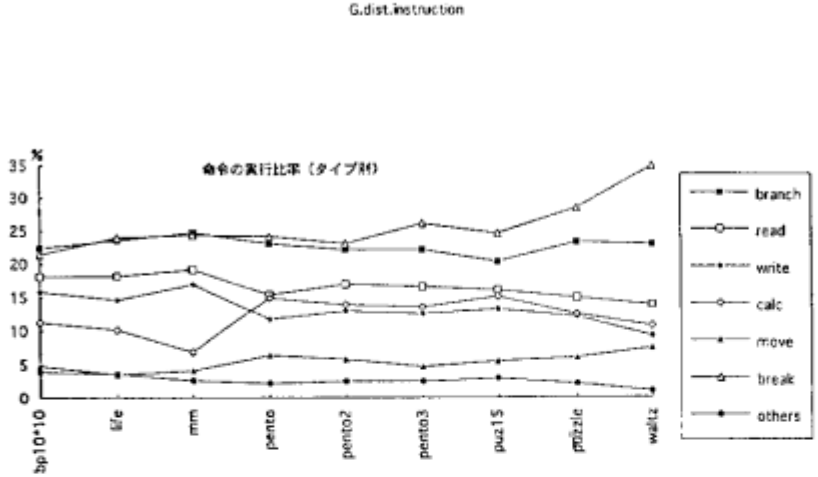
4. MACRO CALL
   外部命令から内部命令への分岐は、通常の分岐に比べて、2クロック速い。
   内部命令から外部命令への復帰は、通常の分岐に比べて、3クロック速い。
   内部命令から外部命令への分岐は、通常の分岐に比べて、1クロック遅い。
   内部命令から内部命令への分岐は、通常の分岐に比べて、1クロック遅い。

8

PIM/pスライド



9



10

PIM/p CPU アークテクチャの評価

1）ベンチマーク・プログラムの動的特性による評価

—— 命令レベル・シミュレータによる１PEでの実行結果より

・命令種別ごとの実行頻度

・PIM/p特有命令の効果

2）ライブラリ・プログラムの静的解析による評価

—— レジスタのtag部、data部の依存関係より

・タグアーキテクチャの評価

11

PIM/pスライド

PIM/p クラスタの評価

(株) 富士通　新井 進

・共有バス負荷を減らすための工夫
  ・マクロ命令
  ・特殊メモリアクセス命令

・台数効果を阻害する要因
  ・1命令当りの実行時間の増加
  ・並列キャッシュのペナルティの増加

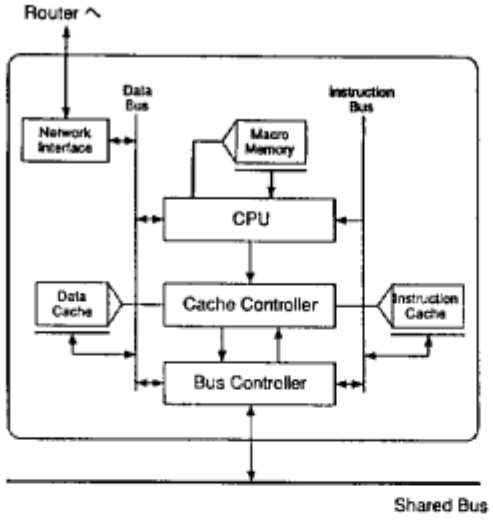
マクロ命令機構の評価

[評価基準]

マクロ命令機構を利用することでどれだけキャッシュミスの抑制できているか

[評価方法]

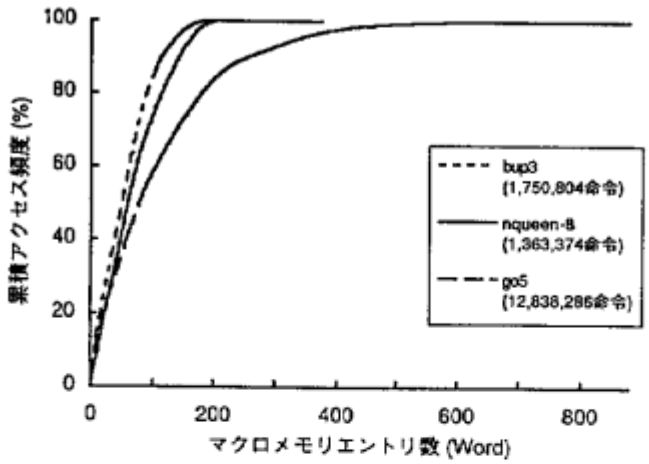
1) シミュレーションにより、アクセスパターンを採取
2) マクロ命令へのアクセスをキャッシュ命令へのアクセスに変換
3) 変換したアクセスパターンをキャッシュシミュレータに掛け、キャッシュミス率、共有バス使用クロック数を求める



PIM/p CPU の構成

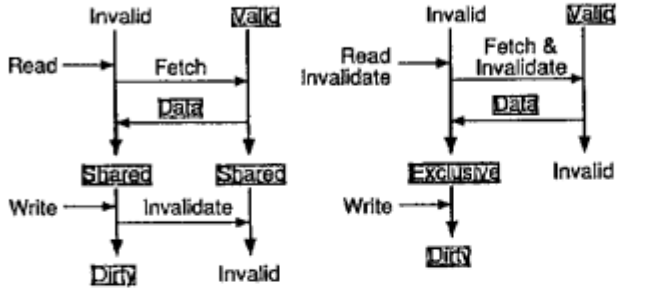
		キャッシュミス		共有バス使用	
		回数	増加率	クロック数	増加率
bup3	マクロ機構使用	12,424		101,066	
	マクロ機構使用せず	13,254	1.12	106,536	1.05
	マクロ機構使用せず＋キャッシュ増	7,748	0.35	61,956	0.61
go5	マクロ機構使用	563,955		3,662,154	
	マクロ機構使用せず	675,903	1.20	4,350,414	1.19
	マクロ機構使用せず＋キャッシュ増	133,238	0.24	968,182	0.26
nqueen	マクロ機構使用	6,706		61,198	
	マクロ機構使用せず	6,821	1.02	62,278	1.02
	マクロ機構使用せず＋キャッシュ増	6,370	0.95	47,524	0.78

マクロ命令機構の効果 (1PE)



マクロメモリの利用状況

5



通常の Read を使用　　Read Invalidate を使用

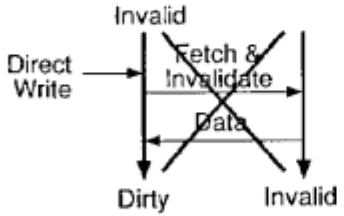
Read Invalidate 命令の動作

6

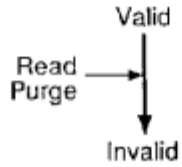
Read Invalidate 命令の効果

- Life で台数効果が 11% 向上
  (実行頻度の高いゴールのヘッド部と、ユニフィケーション部分に適用)
- それでも、多くの無効化が残る
  - 処理系の共有領域
    全無効化 の 1/3 がこの領域集中
  - 1つのキャッシュブロックに複数のデータが同居で不必要な共有・無効化が生じる

7



Direct Write 命令の動作



Read Purge 命令の動作

2

PIM/pスライド

Direct Write 命令の問題点

・適用できる箇所が限られる

(例) ヒープの初期化

実時間 GC を採用していることもあり、初期化の頻度は高くない

・整合性の維持にコストがかかる

ヒープの初期化に Direct Write を使用した場合、GC の後に、キャッシュをパージする必要がある

Read Purge 命令の問題点

・整合性の維持が困難

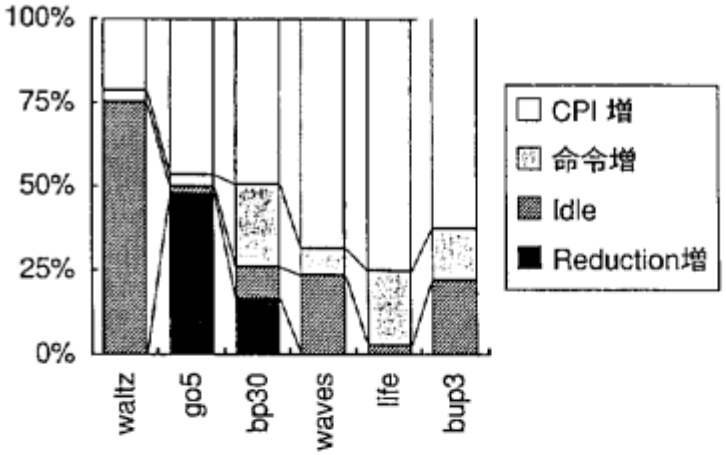
同一ブロック内に別のデータが有った場合に問題

9

Program	bp30		bup3		go5		life		waves	
	1pe	8pe	1pe	8pe	1pe	8pe	1pe	8pe	1pe	8pe
CPI	1.59	2.05	1.42	1.90	1.70	2.30	1.72	2.90	1.78	2.76
命令枯渇	0.03	0.15	0.01	0.11	0.10	0.32	0.02	0.06	0.06	0.17
分岐	0.27	0.35	0.20	0.31	0.33	0.31	0.29	0.36	0.33	0.44
状態遷移		0.08		0.07		0.03		0.27		0.16
データキャッシュミス	0.08	0.46	0.02	0.22	0.07	0.39	0.20	0.92	0.12	0.50
その他 / 誤差	0.10	0.01	0.06	0.19	0.12	0.24	0.11	0.29	0.17	0.49
バス使用率	0.03	0.62	0.01	0.55	0.02	0.62	0.06	0.68	0.05	0.64
ミスペナルティ	7.31	9.13	7.34	14.07	7.17	10.65	7.44	14.94	7.36	15.32
ミス率	0.06	0.09	0.03	0.05	0.06	0.08	0.05	0.08	0.06	0.08
共有率		0.44		0.72		0.51		0.73		0.67

CPI 悪化の要因

11



台数効果の低下要因

10

PIM / p スライド

共有メモリと分散メモリ (どっちが GHC/KL1 に適しているか)

発表者: 中島 浩 (京大)
日時: 3:15 ~ 3:45, 3/23　　議事録担当: 中越 靖行 (富士通 SSL)

1　GHC/KL1 分散メモリで十分インプリメントできる

例) Multi-PSI, PIM/m

2　共有メモリでは？

嬉しい　ロード、ストアで通信出来る。　キャッシュしてくれる
悲しい　ロード、ストアしか出来ない。　キャッシュされてしまう

3　GHC/KL1 に共有メモリは向かない？

producer のアドレスが沢山 consumer に見える
unbound が読めてしまう

4　要は car だけ送れば良い

car を貯める FIFO があればよいのに。

5　MBP (Memory Based Processer) では?

メモリ空間上に FIFO が沢山作れる。
FIFO の ID は アドレス
store = enqueue, load = dequeue

プロセッサ間のストリームに FIFO を割り付ける
FIFO は輸出表の一部と思う (アドレスを見せる)
message の大きさや作成タイミングは 頑張って解析する。

6　まとめ

ユニフィケーションの方向を解析する
consumer のメモリに producer が書く
FIFO を用意する

てなことをやれば共有メモリもなかなかいける。

7　議論

プロセスが移動する場合は？
共有メモリとの関係は？
MBP は CPU が代わりにやってもいいのじゃないか。

（なんとか）共有メモリと分散メモリ

— どっちが GHC/KL1 に適してるか —

中島 浩（京都大学）

◇ 共有メモリの嬉しいところ

- load/store で話ができる
- キャッシュしてくれる

◇ 共有メモリの悲しいところ

- load/store でしか話ができない
- キャッシュされてしまう

⇒ GHC/KL1 には共有メモリは向かない?

⇒ ということで今日のお話はおしまい?

⇒ というわけにも行かないので...

◇ プロセッサ間ストリーム通信に話を限って...

◇ まず普通にやると

- producer がリストを持つ
- consumer はリストの先頭のポインタを持つ
- consumer は先頭の cons cell を読む
- consumer は car を処理し，cdr を新しいポインタにする

★ producer のアドレスが沢山 consumer に見える

★ unbound が読めるとかなり悲惨

★ cons cell の false sharing も少し悲惨

★ load の latency の問題

◇ そこで（マージされた）ストリーム通信の最適化みたいなことをやると

- consumer がリストを持つ
- producer はリストの末尾のポインタを持つ
  ⇒ やっぱりアドレスが沢山見える
- producer は consumer 側に cons cell を作って...
  ⇒ これも結構悲惨

◇ 要は car だけ相手に送れれば良い
c.f. PIM/m の実装 (?)

⇒ car を貯める FIFO があればいいのに...

◇ 文部省重点領域研究「超並列システム」D マシン（仮称）

⇒ MBP (Memory Based Processor) という，おりこうさんがいて

- メモリ空間上に FIFO が沢山作れる
  ⇒ FIFO の ID はアドレス
  ⇒ store = enqueue, load = dequeue

◇ だから

- （マージされた）プロセッサ間ストリームに FIFO を一つ割付ける
- FIFO は輸出表の一部と思う（アドレスを見せる）
- producer は FIFO に car を store すればよい
- producer 側では原則としてキャッシュしない
- キャッシュ・ライン程度のまとめはしたりする

PIM-WG 5

◇ FIFO が空になったらどうするの?

⇒ MBP はおりこうさんで

- 「空だよ」と教えてくれる
- 空の FIFO に書くと
  - ★別のアドレス（FIFO かも知れない）に FIFO のアドレスが書かれたり
  - ★割込を起こしたりする

◇ FIFO がいっぱいになったらどうするの?

⇒ MBP はおりこうさんで

- どうしようかなぁ，と悩んでいる

PIM-WG 6

◇ メッセージは 1 ワードじゃないんだけど

⇒ キャッシュ・ライン単位 (32B) というのもあるよ

◇ もっと大きいんだけど

⇒ ではこうしましょう (c.f. ABCL/onEM4)

- consumer は multiple reader の FIFO を用意する
- FIFO にはメッセージ・バッファのアドレスを入れておく
- メッセージ・バッファはいろんなサイズを用意し，FIFO もそれに応じて用意する
- producer は必要な大きさのメッセージ・バッファに対応する FIFO を読んで，バッファのアドレスを得る
- FIFO は適当にプリフェッチして latency を隠す
- メッセージはキャッシュせずに consumer のメモリに書込む
- メッセージを書いたら，そのアドレスをストリームの FIFO に書く

PIM-WG 7

◇ メッセージができてるかとか，メッセージの大きさとかが，ストリームに突っ込む時には良く判らないんだけど

⇒ そんなこと言うからいけない

- 頑張って解析して判るようにする
- プログラマーに教えてもらう

◇ メッセージの中に返信用の unbound があるんだけど

⇒ そんなこと言うからいけない
とは言わない

- write mode の unbound ってことが判るようにしよう
- そうするとストリーム通信と同じノリで行くんじゃない?

PIM-WG 8

共有、分散メモリスライド

◇ まとめ（でもないけど）
- ユニフィケーションの方向を解析する
- consumer のメモリに producer が書く
- （できれば n 個の）FIFO を用意する

てなことをやれば，共有メモリもなかなかいける
（といいなぁ）

PIM 共通ベンチマーク中間結果

発表者: 久門 耕一 (ICOT)
日時: 3:45 ～ 4:15, 3/23　　議事録担当: 西崎 慎一郎 (富士通 SSL)

1　ベンチマークテストの目的

	ハードウェアを知る	ソフトウェアを知る
ハードウェア開発者	設計の善し悪し	ユーザのコーディングスタイル
ソフトウェア開発者	ハードウェアの性能	コーディングの善し悪し

1. 測定条件

- PIM/p 用のベンチマークを使用。
  - 全ての機能をテストできれば良いが、 VPIM は大き過ぎる。
  - 少なくとも append よりはまともである。
  - より良いベンチマークとはどのようなものかを探る手がかりに。
- 1PE の単体で実行。
  - 処理系の実装方式、アプリケーション、アーキテクチャなど、各 PIM の基礎データを得る。
  - リダクション数、サスペンド回数などの条件を整えやすい。
  - 複数 PE で実行する時間がなかった。
- Multi-PSI を基準にする。
  - 性能が安定している。
  - RPS 値よりは意味がある。

2　測定結果

- append(nrev) の問題の大きさを変えると、キャッシュのミスヒットがあると遅くなるのがわかった。
- ベンチマークの性質の違いがわかった (いくつかのベンチマークは性質に違いがないことも)。
- リダクションの重みがベンチマークによって違い過ぎる。 append では性能の指標にならない。
- Multi-PSI と比較したグラフを PIM ごとに作成。
  - VPIM をベースにした処理系は、似たような形になる。
  - PIM によっては append を速くするためのチューニングをしている。
  - VPIM をベースにした処理系は、 bestpath が遅い (priority queue が遅い?)。
  - クロックを Multi-PSI と同じと仮定しても、 VPIM には Multi-PSI に及んでない (インプリメンテーションに問題あり)。

3　討論

- もっと大きいベンチマークで測定できないか。
  全部の PIM 共通では使えないので、難しい。
- このような小さなプログラム (Toy Program) で何がわかるか。
  - append で比較するよりはましである。
  - 結果の解析の仕方によっては、いろいろなことがわかる。
- チューニングの目標を定めにくい。
  ある一つだけの目標を定めると、全体としてバランスが悪くなるかも知れない。

- 書き方の悪いプログラムを変更すべきか。
  - 結果を見てプログラムを書き直して測定し、より良いベンチマークにする。
  - あえて書き直さずに、どんな書き方をしても速くなることを考えるべき。

PIM共通ベンチ中間結果

ICOT　第一研究室

久門　耕一

1

基準となる物差しは？

- アプリケーション毎に大きく変わらない指標が便利
- 異論はあるでしょうがマルチΨを１とする
  →RPSよりはましだろう
  →システム変更が少なく時間的に安定
- 少し述べますがappendによる評価は殆ど意味がありません

3

各マシンのアペンドＲＰＳ

		nrev1500	nrev5000
Mpsi	200ns	125	114
pim/m	65ns	600	411
pim/p	80ns	305	281
pim/c	66ns	55	56
pim/i	240ns	65	65
pim/k	100ns	76	76
SS10	28ns	128	128

2

なぜ単体性能に関してなのか

- 各PIMを比較するための基礎データ
  →実装方式なのか
  →アプリケーションコーディングなのか
  →アーキテクチャなのか
- 条件を整えやすい
  →サスペンド数
  →リダクション数
- 時間がなかった

4

何のためのベンチマークか

- 設計者が設計の善し悪しを調べる
- ユーザがハードの性能を知る
- 設計者がユーザのコーディングを知る
- ユーザがコーディングによる性能差を知る

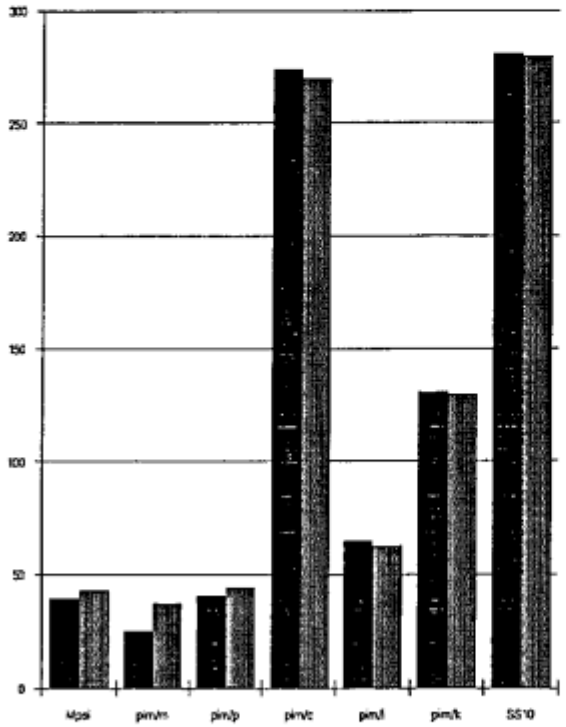
5

共通ベンチマークの意味

- アペンドよりも多少まともなプログラムによる速度の測定
- アプリケーション実行に必要な機能の目安
  何を早くすることが重要なの？
- より良いベンチマーク（指標）を設定する
  appendが速くないといけないとは言うが、
- 元々はPIM/pのチューンアップ用のベンチマーク群だった

6

アペンド実行に掛かるクロック数



7

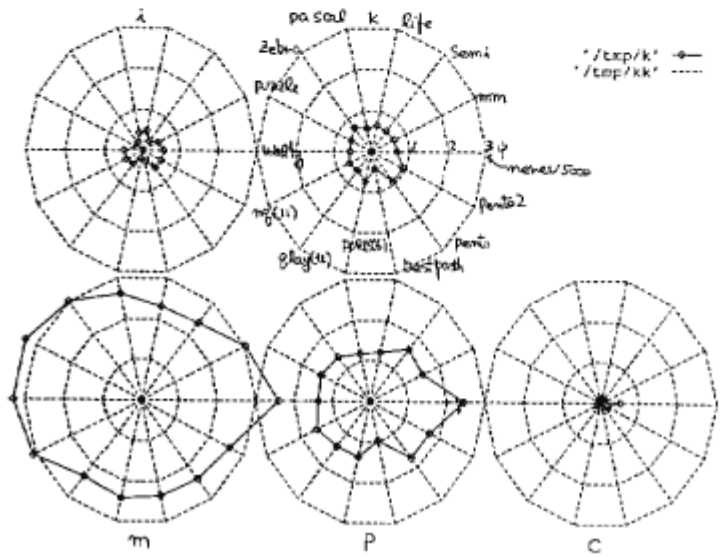
共通ベンチスライド

ベンチマークの種類

nrev1500,nrev5000
Master Mind
Semi Group
Life
Pascal
Zebra
Puzzle
Walts
Nqueen(11)
Qlay(11)
puz15(6)
Bestpath(100x100)
pento(8x5)
pento2(8x5)

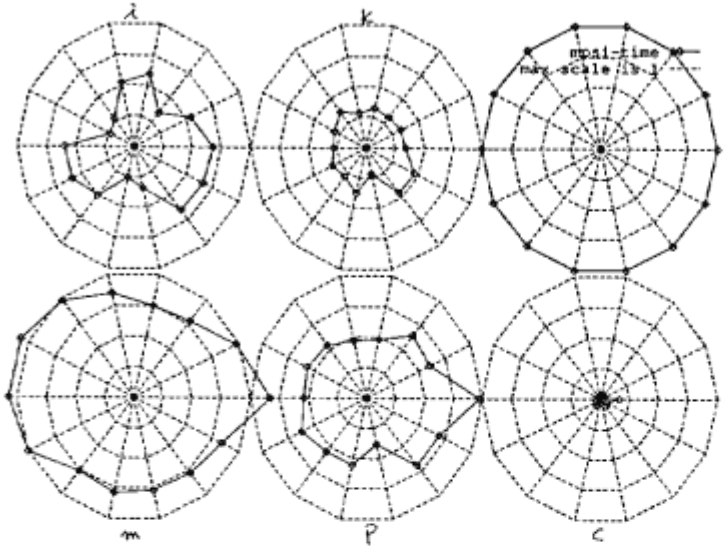
8

Multi-ψ を基準にした各PIMの速度

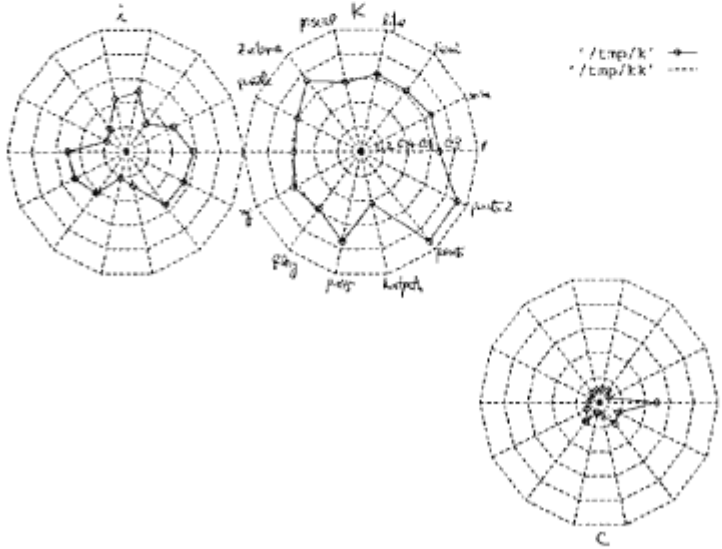


10

ワロクで正規化したもの



9



11

共通ベンチスライド

ベンチマーク間の非相関(Spearman法)

	nrev5000	nrev1500	waltz	mm	nqueen(1	pascal	life	bp(100x1	semi	puz15(6	zebra	puzzle	qlay(11)	pento(8x	pento2(8x
nrev5000	*0*	0.14	0.91	0.87	0.73	0.74	0.72	0.59	0.62	0.68	0.82	0.92	0.35	0.41	0.50
nrev1500	0.16	*0*	0.82	0.78	0.64	0.66	0.65	0.52	0.55	0.79	0.73	0.84	0.35	0.43	0.47
waltz	0.42	0.43	*0*	0.10	0.12	0.13	0.20	0.27	0.35	0.45	0.43	0.51	0.31	0.39	0.33
mm	0.41	0.43	0.10	*0*	0.09	0.12	0.16	0.24	0.25	0.35	0.33	0.41	0.27	0.32	0.26
nqueen(1	0.38	0.39	0.13	0.09	*0*	0.09	0.13	0.21	0.26	0.39	0.37	0.45	0.23	0.28	0.23
pascal	0.43	0.45	0.14	0.12	0.09	*0*	0.09	0.19	0.27	0.40	0.38	0.46	0.23	0.30	0.25
life	0.44	0.48	0.18	0.16	0.13	0.09	*0*	0.26	0.32	0.46	0.44	0.53	0.23	0.24	0.20
bp(100x1	0.48	0.47	0.28	0.26	0.22	0.18	0.23	*0*	0.20	0.32	0.28	0.37	0.25	0.38	0.34
semi	0.42	0.44	0.30	0.23	0.20	0.24	0.24	0.23	*0*	0.23	0.21	0.29	0.24	0.28	0.25
puz15(6)	0.57	0.61	0.33	0.26	0.32	0.31	0.31	0.36	0.22	*0*	0.10	0.16	0.39	0.40	0.37
zebra	0.53	0.55	0.28	0.23	0.27	0.27	0.30	0.30	0.22	0.10	*0*	0.13	0.38	0.42	0.38
puzzle	0.54	0.53	0.30	0.28	0.32	0.33	0.39	0.33	0.30	0.18	0.13	*0*	0.44	0.52	0.48
qlay(11)	0.31	0.34	0.47	0.43	0.31	0.32	0.31	0.24	0.28	0.51	0.47	0.57	*0*	0.17	0.18
pento(8x5	0.38	0.45	0.44	0.39	0.30	0.31	0.28	0.31	0.29	0.50	0.46	0.55	0.18	*0*	0.08
pento2(8x	0.39	0.46	0.36	0.30	0.23	0.24	0.21	0.29	0.27	0.45	0.42	0.52	0.19	0.07	*0*

12

ベンチマーク間の非相関(Spearman法)の上位

	nrev5000	nrev1500	waltz	mm	nqueen(1	pascal	life	bp(100x1	semi	puz15(6	zebra	puzzle	qlay(11)	pento(8x	pento2(8x
nrev5000	*0*	0.14													
nrev1500	0.16	*0*													
waltz			*0*	0.10	0.12	0.13	0.20								
mm			0.10	*0*	0.09	0.12	0.16								
nqueen(1			0.13	0.09	*0*	0.09	0.13								
pascal			0.14	0.12	0.09	*0*	0.09	0.19							
life			0.18	0.16	0.13	0.09	*0*								0.20
bp(100x1						0.18		*0*	0.20						
semi					0.20				*0*						
puz15(6)										*0*	0.10	0.16			
zebra										0.10	*0*	0.13			
puzzle										0.18	0.13	*0*			
qlay(11)													*0*	0.17	0.18
pento(8x5													0.18	*0*	0.08
pento2(8x													0.19	0.07	*0*

13

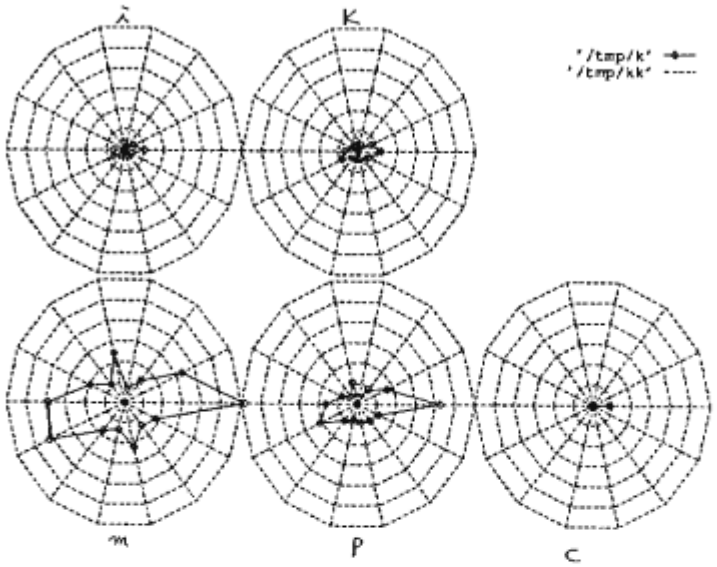
ベンチマーク間の非相関(0.4以上)

	nrev5000	nrev1500	waltz	mm	nqueen(1	pascal	life	bp(100x1	semi	puz15(6	zebra	puzzle	qlay(11	pento(8x	pento2(8x
nrev5000	*0*		0.91	0.87	0.73	0.74	0.72	0.59	0.62	0.68	0.82	0.92		0.41	0.50
nrev1500		*0*	0.82	0.78	0.64	0.66	0.65	0.52	0.55	0.79	0.73	0.84		0.43	0.47
waltz	0.42	0.43	*0*							0.45	0.43	0.51			
mm	0.41	0.43		*0*								0.41			
nqueen(1					*0*							0.45			
pascal	0.43	0.45				*0*				0.40		0.46			
life	0.44	0.48					*0*			0.46	0.44	0.53			
bp(100x1	0.48	0.47						*0*							
semi	0.42	0.44							*0*						
puz15(6)	0.57	0.61								*0*				0.40	
zebra	0.53	0.55									*0*			0.42	
puzzle	0.54	0.53										*0*	0.44	0.52	0.48
qlay(11)			0.47	0.43						0.51	0.47	0.57	*0*		
pento(8x5		0.45	0.44							0.50	0.46	0.55		*0*	
pento2(8x		0.46								0.45	0.42	0.52			*0*

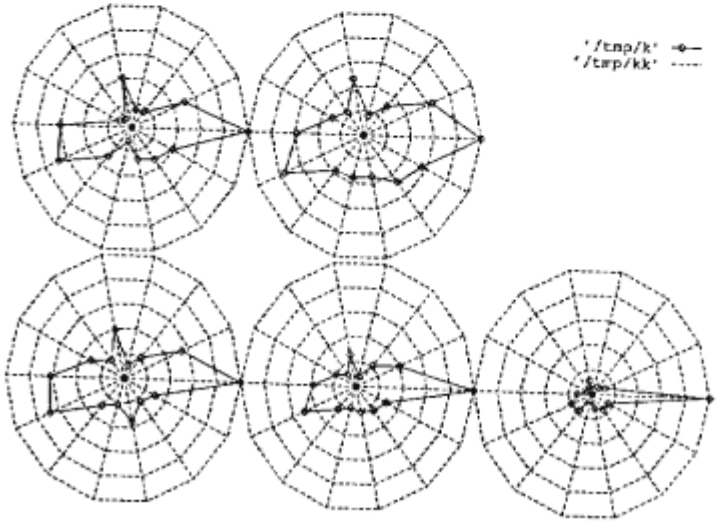
14

共通ベンチスライド

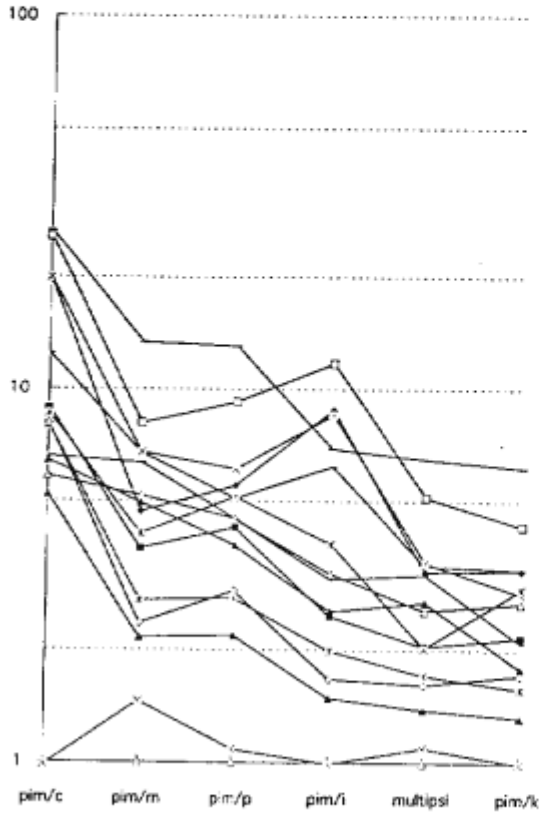
RPS を plot したもの



Append の速度で正規化



nrev1500を基準とした各種ベンチマークリダクション時間



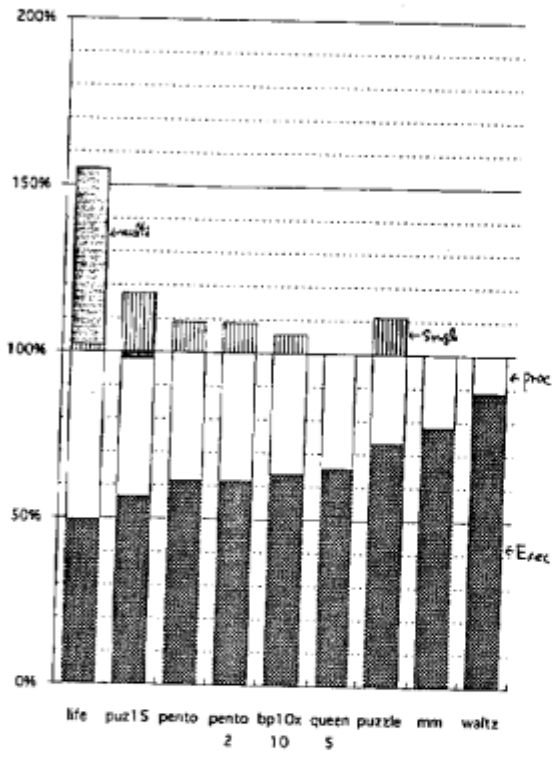
共通ベンチスライド

リダクションと時間

	0 red	1 pim/c	2 pim/m	3 pim/p	4 pim/i	5 multipsi	6 pim/k	pdss.d
nrev1500	1128756	20452	1880	3694	17220	8986	14783	8800
nrev1500	1128756	1.00	1.00	1.00	1.00	1.00	1.00	1.00
nrev5000	12512506	0.98	1.46	1.09	0.99	1.10	0.99	1.00
mm	1736863	7.98	2.71	2.75	1.99	1.71	1.58	1.86
semi	731869	12.44	6.72	5.08	6.20	3.36	3.28	2.66
life	359984	26.64	13.38	13.04	6.91	6.48	6.16	5.02
pascal	330416	8.88	3.73	4.26	2.46	2.04	2.16	1.70
zebra	404253	25.72	8.09	9.24	11.72	5.12	4.29	4.73
puzzle	1268027	19.77	4.69	5.51	8.79	3.24	3.27	3.06
waltz	1206930	8.04	2.35	2.88	1.67	1.62	1.72	1.36
nqueen(11)	4571143	5.20	2.16	2.19	1.49	1.39	1.32	1.66
qlay(11)	1747255	5.83	5.17	4.50	3.20	2.55	2.67	2.68
puz15(6)	7926506	19.90	6.75	6.10	8.52	3.42	2.82	1.72
bp(100x100)	2825153	8.57	4.09	5.15	3.84	2.02	2.91	2.08
pento(8x5)	8467175	6.59	6.34	4.48	3.10	3.18	2.09	2.60
pento2(8x5)	8467175	6.42	4.95	3.81	2.54	2.69	1.79	2.43

18

各ベンチマークの時間(exec,proc,singl,mult)



	life	puz15	pento 2	pento 10	bp10x 5	queen	puzzle	mm	waltz
nrev5000	1.00	1.00	-----	-----	-----	-----	-----	-----	-----
nrev1500	1.00	1.00	-----	-----	-----	-----	-----	-----	-----
mm	-----	-----	1.00	-----	1.00	1.00	-----	-----	-----
semi	-----	-----	-----	1.00	-----	-----	1.00	1.00	-----
life	-----	-----	1.00	-----	1.00	1.00	-----	-----	-----
pascal	-----	-----	1.00	-----	1.00	1.00	-----	-----	-----
zebra	-----	-----	-----	1.00	-----	-----	1.00	1.00	-----
puzzle	-----	-----	-----	1.00	-----	-----	1.00	1.00	-----
waltz	-----	-----	1.00	-----	1.00	1.00	-----	-----	-----
nqueen(11)	-----	-----	1.00	-----	1.00	1.00	-----	-----	-----
qlay(11)	-----	-----	-----	-----	-----	-----	-----	-----	-----
puz15(6)	-----	-----	0.99	-----	-----	1.00	1.00	-----	-----
bp(100x100)	-----	-----	1.00	-----	0.99	0.99	0.99	-----	-----
pento(8x5)	-----	-----	0.99	-----	-----	-----	-----	-----	-----
pento2(8x5)	-----	-----	0.99	0.99	0.99	0.99	-----	-----	-----

	0.99	1.00	0.99	0.99
	-----	-----	-----	0.99
	-----	0.99	-----	0.99
	-----	1.00	0.99	-----
	-----	1.00	0.99	-----
	-----	-----	-----	-----
	0.99	-----	0.99	-----
	-----	1.00	0.99	-----
	0.99	0.99	1.00	1.00
	0.99	1.00	1.00	1.00

life 0.57
waltz 0.57
puz15 0.18
puzzle

19

共通ベンチスライド

パネル討論 2: 並列 (推論) マシン: 未来への展望

司会: 後藤
パネリスト: 天野, 関口, 中島 (浩), 松岡, 村上
日時: 4:30 ～ 5:50

【後藤】
私、平田氏に脅迫されて司会をやれと言われまして。最後の1時間ちょっとのパネルではこういうテーマが与えられています。「並列マシン、推論マシンの未来への展望」と称して、何が分かった、何が分からなかったか。それからアーキテクチャは、並列アーキテクチャ研究はどういう進めるべきか、何を残せるか。それから汎用並列マシンのベースとしてのPIMはどうであったか。また、今後の期待とあります。ただこの中で、もう既に幾つかここ2日間に渡っていろいろ議論されてきたことも多いと思うので、ある程度テーマを絞って行きたいと思っています。

その前に、パネリストのご紹介をしたいと思います。正式タイトルは忘れましたけれども、今、調国さんをスポンサに開いております並列アーキテクチャ技術動向調査委員会という名前だと思いますが、この委員会のメンバです。実はこれも平田氏から「誰をパネリストにするの」と言われて、たまたまその時にこの委員会があった日だったもので、「このメンバで行こう」と答えただけなんです。実は小畑先生と中田さんがいるんですが、お二人とも昨日今日いらっしゃれなかったので、先ほどお話になった久門さんと、元々のこの会の幹事である平田さんは前には出てきていませんが、それ以外の5人に方に並んでいただいています。皆さんもうご存じの方ばっかりです。今後の日本の並列マシンを支えるメンバが揃っているんじゃないかと思って、そうおだてておいて、特に松岡先生なんかはPIM・WGメンバーじゃなかったんですが、無理やり引っ張ってきたというところです。また、この委員会で今レポートを書いておりますので、4月早々出ると思います。また立派なのを書いて頂いていますのでお楽しみにしてください。

さっきたくさんテーマがあったのですが、このうちPIMの細かい所がどうのこうのというのは既に大分議論されていたと思うので、幾つかのテーマに絞ります。気分としては、これは勝手な私の想像なんですけど、11年間やってきて、並列マシン研究にちょっと疲れ気味の方が多いのではないかと勝手に解釈しまして、そういう方を元気づけるメッセージを5人の方にお願いしたいと思っています。

1つは、これは自分は分かったつもりだという点、だから明日からはこれをやろう。これをやろうという部分がどう変わるか分からないんですが。それから、これは私の前からの課題だったんですけれども、汎用並列マシンというはどこまで主役になれるのか。これについてまず簡単に、時間がないので5分ずつぐらいでざっとポジションをお話しいただきたいと思っています。まずは中島先生から。

【中島】
後藤さんはさもああいうタイトルで喋れと事前に言っていたかのようですが、それは全然ウソでして、みんな「エッ!?」とか言って。さっきのタイトルの中で、PIMを作って何が分かって何がどうしたとかいう話があって、良く見るとPIMを作った人って、ここには私しかいないので、これは私が言わないといけないかなと。ところが、村上大先生は、非常にコンテキストがローカル過ぎる、もう少しgeneralなview pointから喋れとかいうようなことを昨日言っていまして、generalなview pointから喋ると何か良く分からない話になるんですけれども、例によって分からない話をします。

まずハードはたくさん適当につなげると何とかなる、というのが私は分かったと思います。基本的にハードなんていろいろなことを凝ってもしようがない。悪口はなるべく言いませんけれども、先ほど久門君が何か発表していましたけれども、あの中で一番手を抜いているのは、多分PIM/mだと思います。特にプロセッサ間通信、手を抜きっ放しと。それでも何とかなる。もうちょっと頑張れば、もっと何とかなる訳なんですけれども、とにかくある程度たくさんつなんでやるということがまず基本的な特性になる。どうしてもアーキテクチャ屋さんというのは頑張りたがるんですけれども、あまり頑張っても多分しようがないんじゃないかというのが私は分かったような気がする。本当かな。

分散処理にはいろいろな捉え方があると思いますが、とにかくグローバルなことをやらない。とにかく近くの人とだけ通信するとか、相手は遠いからあんまりメッセージをたくさん投げないとか、そういうのが分散処理全部かどうかというのは分かりませんが、あまりガチッと皆に枠をはめるようなのではなくて、何かバラバラやっているような処理というのが、これは多分うまく行くと。こういう態度じゃないと、多分うまく行かないんじゃないかと私は思っています。

それから、速くできる処理とそれなりの処理、昔そんな話があったと思うんですけれども、この格差というのはどんどん開いていくのではなかろうかと。つまり基本的に、例えばプロセッサの$logP$だとか、プロセッサの数に比例するとか、そういう処理とそうじゃない処理、つまり自分の所で閉じて何かする処理、この格差がものすごく広

がって行く筈であると。従って、速くできる処理をとにかく相手にするというのが基本的に物事のやり方なので、逆に言うと、プログラミングというのもそっちの方に持って行かないとダメなんだろうなと私は思っています。

どんどん話はまとまらなくなりますが、自動負荷分散とか動的負荷分散とか、クラスタ内でやられた話がたくさんあるんですけれども、あんなことをやっても大部分うまく行かないというのが私は分かったような気がする。自動というのは多分夢でしょう。やらない方がいいと私は思います。

次に台数効果ですけど、大体、プロセッサ数 N で飽和しそうなら、プロセッサをちょっと増やすともっと飽和する。そうすると10倍になると必ず飽和していると。これがほとんど真実でありまして、N というのは、気分として例えば100だとしましょうか。100で飽和の影が見えたら、1000になると必ず飽和しているというのが何となく作って分かったような気がします。我々、1000も作っていませんけれども、64で傾きかけたプログラムというのは、必ず256では平らになっていますから、1000とか1万になると、逆に下がっているかも知れない。8でとにかく傾きかけたらもう絶対ダメ、逆に言うとね。こうなりそうな話というのは極力避けなければいけない。だから、これで例えば何かが飽和しそうになったら、これは必ず飽和するものだと思わないと多分いかんのではなかろうかと思います。これが疲れた後に元気なメッセージかどうか知りませんけれども、終わります。

【後藤】

天野先生、お願いします。

【天野】

疲れた後ますます疲れさせるような話なんですが、励ましになるかも知れません。僕は、後藤委員会に入って初めてPIMを勉強させてもらったわけなんですが、PIMというのは大変成功したマシンであると。ただ何か一般の並列計算機屋からは特殊マシンとして扱われているんじゃないかと。だから、まず最初に敬遠の外に置かれているんじゃないか。それによって正当な評価が受けられていないのではないか、ということがだんだん分かってきた。それは何故かというと、3つ原因があって、まず論文を大体見ると、並列推論マシンとして目的充足型のストーリー展開になっているんですね。どういうことかというと、並列推論マシンというのは、これこれこういう機能があって、KL1というのがあって、これ位のスピードでやればうまく行く筈で、現にうまく行ったろうとかいう感じの論文の展開になっていて、うん、そうか、だから何なんだということになってしまうというのがまず第1点です。

2番目は、KL1とその処理系への理解がまず前提になっているということで、これはなかなか難しくて、実は僕も良く分かってなくて、ここの所は、多分、人によっては全く受け付けないのではないかなという不安があります。僕は後藤さんによって強制的に理解させられたので、それでもちょっとは分かってきたような気がするんですが、こういうのは体質的にちょっとダメなんじゃないかという人も居るとは思います。

あと、アプリケーションにすごくいいのが動いているんで久門さんの話を聞いて、だんだんこれからアーキテクチャとの関連性について検討がなされていくんじゃないかなと思いました。でも、いいアプリケーションがいっぱい動いても、逆にそのアプリ屋が威張っちゃった、アーキテクトがマッピングとか一生懸命工夫しちゃったことによってアーキテクチャが果たして良くなっているのかどうか、という所が良く見えなくなっちゃう。この辺が問題だったのではないでしょうかという訳です。

そこで、結局、いろいろなPIMで計算機構成技術が用いられているんですが、大規模な並列計算機というのは、多分これから細粒度並列計算機なんていうのが流行になるのではないかという感じなんで、多分すごく興味深いんじゃないかと。ただ、プロセッサアーキテクチャ関連に関しては、KL1依存性がものすごく強いので、とにかく僕は全然分かってない。多分ここからもいろいろなアーキテクチャ技術が抽出できるんじゃないかと思います。でも、分かってないから、結合網周辺で何か一般的な細粒度並列計算機として議論できる部分が割とあるのではないかという話です。

それで、まずPIMというのはいろいろなマシンがあると。これがすごくいいのではないかと。まずメッシュはあるし、階層バスもあるし、マルチハイパーキューブもあるし、階層メッシュもあるし、クロスバもあるとかいうので、さまざまな結合網のオンパレードです。しかも、非数値計算の細粒度プロセスで、これのトラフィック解析というのはあまりまじめにやられてないんですよね。結局、今の大規模並列マシンのトラフィック、要するにネットワークの解析なんかの研究では、大体全部何でもトラフィックで振っちゃっているんだけれども、あんなもの本当かどうか誰も分かりはしない。数値計算だとトラフィックは読めちゃうんだけど、逆にこれは読み切れちゃうんですね。だから非数値計算の細粒度プロセスのトラフィック解析というのはもの凄くアカデミックに価値があると僕は思う。

まず交信の局所性というのはどれぐらいあるか。ランダム交信を仮定した理論的評価と比べてどれぐらい実測が異なるか。どこが混んで、例えばマルチハイパーキューブの交信負荷の分散がすごく上手く行ったとか、メッシュの場合はレイテンシが大丈夫かとか、階層メッシュあるいは階層バスでは階層間の混雑はないかとか、この辺に関しては検討するとすごく面白いんじゃないか。特にネットワークはたくさんあるから、さっき久門さんがやられたように、横断的な評価プロジェクトが可能なので、細粒度プロセス非数値のトラフィック分析みたいな感じの論文を書くとも

の凄く面白いんじゃないかなというのが1つあります。

あともう1つは、クラスタ構造を持った大規模マシンという点です。このクラスタ構造が良いかどうかは分からないんだけれども、従来のクラスタ構造を持つ並列マシンに比べて、桁外れのスケールを持っていると私は思います。クラスタ構造のマシンというのは、例えばスタンフォードのDASHとかイリノイのCedarとか提案はいろいろされているんですが、実は大したものが動いてない。32とか、そんなものですよね。例の文部省の重点領域マシンが動けば、あれがきっと凄いんだけれども、まだちょっと分からないので、今の段階で、クラスタ構造を持って512も動いているのは多分世界にありません。だからクラス内のプロセッサ数をいろいろ変化させてみて、クラスタ内外のトラフィック分析をやってみる。例えばクラスタ内のプロセッサ数は過剰かどうか。さっきPIM/pの方のトラフィック分析の話が面白かったのですが、最適値はどれぐらいか。クラスタという構造を大きくしていくと、トラフィックにどういう影響を与えて行くかとか、この辺も結局非数値細粒度プロセスのトラフィックという点で、クラスタが絡むからさらに面白くなる。

さらにスヌーブキャッシュの問題があります。これはちょっと問題で、さっきまでの話は大丈夫だと思うんですが、スヌーブキャッシュのプロトコルに関しては、多分KL1を実行させるのと、普通のプログラムではかなり違った傾向が出るのではないかなと思われます。それでもまだ実際のマシン上での評価は乏しいし、KL1を動かしたからというのをちゃんと押さえておく必要があると思います。例えばPIM/iは放送型で、これは問題があるという話もあるんですけれども、これとのプロトコルによる性能の違いはどうかとか、ピンポン現象とか、不要な放送による効率の低下というのはどれぐらい起きているかとか、シミュレーションをやってみた値と実測値というのは実際どれぐらい違ったのかとか、それからPIM/pのキャッシュ間メッセージ転送というのはどれぐらい効果を及ぼしているかとか、この辺の評価をとってみると、もの凄く面白いテーマだと思います。ひょっとしておしまいになるのではないかとかいう話がさっき久門さんから出て来たんですけれども、とてもとても勿体ない話だから、これからこういう点をどんどん頑張って横断的に、ぜひ論文をじゃんじゃん書いて欲しいと。できれば、KL1のイントロ無しで、普通のアーキテクトでも抵抗ない形で論文を書いてくれれば、凄く皆興味持つし、世界的にももっともっとPIMというのは注目されるのではないか。

OHPにはKL1の城砦と書いてあるんですが、結局、今までのPIMの研究というのは、こういうストーリー展開があった訳です。今の手続き型中心のプログラミング言語は行き詰まるだろう。次は論理型プログラミングが有望で一般的になるに違いない。これは本当かどうか知りませんが、でもこれを頭から徹底的に否定できる人というのは世の中に居ないんじゃないかと。だって一寸先は闇ですから、どうなるかわからない。もしかしたらそうなっているかも知れない。KL1は並列処理を記述する論理型言語の中で優れた解の一つで、多分これも頭から否定する人はなかなか居ないと思います。PIMはKL1を高速に効率良く実行でき、この点に関しては世界一のRPS値を誇っていて素晴らしいと。これをやっている限りは無敵で、だけどこのストーリーで論文書いても誰も読まないと。だってそれはそうなんだから、ふーんそうかという感じになってしまって、結局、城砦の中に立てこもって、それで何か朽ち果てて行くのではないかという不安がある訳なんですよね。それで城砦に立てこもると何が問題かというと、まず外部から孤立しちゃうと。つまりそういう論文を書いても、あまり人の関心を呼ばないと。城砦の限界がPIMの限界になると。つまりKL1がこれからどんどん発達して広がってくればいいんだけど、そうじゃないとKL1と心中ということになりかねない。

最近思っているのは、これはつまり今まで問題はPIMというのが商品になるためには、KL1がメジャーにならなきゃいけないし、KL1がメジャーになるためには、PIMの上で動かなければいけなかったから、というのがKL1とPIMが商品にならなかった一つの理由だと思うんですが、これ悪循環で互いを必要として困っているんですね。ところがKL1の方は離陸するんじゃないかと。さっき近山さんの話を聞いてそう思うのは、UNIX上でどんどんKL1が動いていって、例えばこれから大規模並列マシンが出て、重点マシンも動いたりして、その前にぼんぼんKL1は動いてどんどん使われる可能性があると。そうすると心中じゃなくて、KL1は生き残るけど、PIMだけ何か流れ去られていくという運命が待ち受けているのではないかという気が凄くします。

僕は、心理的に城砦から出るのが怖いというのは分かるような気がして、単体プロセッサの性能が最近のRISCより僅かに遅いということはあるんじゃないかと思うんですが、これは気にしなければいいと。だから何なんだと。つまり速くなくても僕はちっとも構わないと思うんです。というのは、並列計算機の研究用プロトタイプというのはプロセッサ本体の性能は極端に低いと困るんだけど、極端に低くなくて結合網などとバランスがとれていれば、例えば今言ったようなトラフィック解析とか全部できますよね。計測機能がしっかりしていればいいと。

ということで、これが元気づけになるかどうか分かりませんが、頑張ってこれから結合網周辺だけでも凄く面白いテーマがあってどんどん論文書けるので、これでおしまいにしないで、各社横断的に皆で評価をしていい論文をどんどん書いて行けば、何か将来、細粒度並列マシンというのが一般的になる時、その元祖としてPIMがあったということを皆が覚えてくれるようなプロジェクトになるのではないか、そういう風に思います。(拍手)

【後藤】
では次、村上先生。
【村上】
天野先生が非常に上手くまとめていただいたので、何も喋ることは無いんですが、OHPにサブテーマというのが幾つかあります。最初の2つは飛ばします。何もわかってないし、何もコメントすることはありません。

並列アーキテクチャの研究をどう進めるべきかという話です。昨日もちょっとそういう話をしたんですけど、次の3つの技術が重要であります。1つは、基幹技術といいますか基盤技術といいますか、そういった方向での検討、これはある意味で普遍性を持たせるような話ですね。もう1つは、実現技術、PIMを5つも作った、その5つ作ったということの善し悪しは別として、多様性というのは大切ですのでいろいろ実現してみることももちろん大切です。

最も大切なのは、作った物とか開発した物に対する評価技術を打ち立てていかなくちゃいけない。ここの委員会というのも、この評価技術をかなりメインにしています。並列アーキテクチャの評価といった場合何が問題になるかというと、評価指標、尺度というものがまだ確立されたものが無い。ここで言いたいのは、いろいろな評価指標、尺度があってもいいんですけれども、いわゆる線形速度向上というのを使うのはもうやめにしましょうよ、ということを一つ言いたいと思います。ここら辺にいらっしゃる方で線形速度向上を使われている方はまさかいらっしゃらないと思うんですけれども、何故かと言うと、元々このグジャグジャッとした線が実際に達成された性能なんですけれども、これに対して悪い悪いといって、この直線まで頑張りたいというような、そういうスタイルというのが非常に多いんですけれども、実際は問題の持っている性質から考えると、もともとスピードアップする筈が無いのに、元々は log の性能向上しかできないのに、わざわざリニアの場合を引っ張り出してここまで持って来たりというような話がある。最初からこれは log でしか性能向上しないのであれば、このゴジャゴジャというのも実はこれの定数倍分の1ぐらい出ている訳で、これは満足すべきであるという話になると思うんです。ですから要するに問題の質というのを見て、それから目標とする性能向上、速度向上比というのを求めましょう。

次に、ちょっと推論というのはわからないので、並列マシンとして残すべきアーキテクチャ、ハードということですね。2つ話をしたい思います。1つは、レイテンシ・トレラントマシンとしての細粒度並列マシンという観点。細粒度並列マシンというのが今も結構言われていますけれども、今後もいろいろとこれに関して議論が進んでいくと思います。細粒度並列マシンといった場合に、これは定義無しで使われることが非常に多いんです。粒度も普通定義無しで話されます。では細粒度並列マシンとは一体何なのかと。分からないけれども、いきなりそういう枕言葉が出てくる。考えられるのは3つあるんですね。まず1つは、細粒度並列の問題にしか適さないマシン、そういうマシンもあると思います。2つ目が、1つは粒度が細かくても耐えられるマシン、3つ目はいや応なしに粒度を細かくしてしまうマシンです。これは何かというのをぱっと見せるとこうなっちゃうんですよね。

まず粒度を細かくしてしまうデータフローマシン、これは命令レベルというのを付けないと電総研の坂井さんあたりに叱られるので、命令レベルデータフローマシンですね。粒度が細かくても耐えられるマシンというのが、いわゆるレイテンシ・トレラントマシンとしての細粒度マシンです。これも最近言われているんですけれども、両者、ぱっと見せちゃうと、僕が最近訳した本にも良く出て来るんですが、データフローマシンのfallacies and pitfallsで、スループットを上げれば速くなる、というのがデータフローマシンにおける一つの迷信といいますか、妄想みたいなものがあって、要は問題サイズ＝粒度×並列度なので、並列度を高くして粒度を落として、とにかくスループットを上げれば速くなるのではないか。これが問題の含んでいる自然の並列性を引き出して、性能を上げるというデータフローマシンの主張でした。

ところが、それをやっちゃうと、各々の処理自体のレイテンシが非常に長くなって、そこら辺のオーバーヘッドが無視できなくなって、これじゃいかんということで粒度を大きくする。これはEM-4とか、現在のデータフローマシンのアプローチです。レイテンシトレラントで細粒度マシンというのはどういうことかというと、レイテンシを何とかして低くするか、隠すということです。最近注目を浴びているのは隠す方ですね。hidingをしようするとマルチスレッドマシンになってきます。ところがマルチスレッドマシンの目指している所は、やはりスループットを上げようと、することです。ですから、これもただ闇雲にやっちゃうと、両方のアプローチが行き過ぎて、データフローマシンのfallaciesに辿り着いてしまう可能性がある。ここでやらなくてはいけないのは、レイテンシリダクションの方に対する基本的な技術というのを探すということです。これの代表的な技術というのはキャッシュですね。リダクションに関するキャッシュを超えるような技術はこれから非常に面白いと思います。ですからマルチスレッドマシンもいいんですけど、あまり突き進むとまたデータフローマシンの失敗を繰り返すことなると。

最後に、scalabilityの話をしたいんですけど、昨日もちょっと話をしたんですけど、それはプログラミングする上でのスケーラビリティだったんですけれども、今日はmassiveのスケーラビリティですね。これはどういうことかというと、超並列と言われているんですけれども、N というのはプロセッサ数なんですけど、プロセッサ数に対して、それの関数となるようなハードウエア資源というのはどんな物があるかというのをちゃんと見極めて、中規模でもい

いですからマシンを作らなくちゃいけない。例えばプロセッサ数 N に比例するようなハードウエア資源をノードプロセッサというのにつぎ込まなくてはいけないような機能というのは果たして良いのか。マシン全体で N^2 になってしまいます。その代表的な例としては、例えば完全結合網とか、コンシステンシをとるためのルックアップ・ディレクトリとか、あとOSなどに良く出てくる管理テーブルなどは、ここら辺に相当するような機能かも知れません。

ノードプロセッサあたり $logN$、プロセッサ数の log のハードウエア資源が必要となるようなもの、例えばハイパーキューブですね。これでもマシン全体で $NlogN$ のオーダ。ハイパーキューブ、昨日も最近衰退しているという話があったんですけれども、ハイパーキューブがひと頃良かったけれどもやはり今衰退しているというのは、こういうハードウエア資源、スケーラビリティの無さというのが効いていると思います。じゃあメッシュがいいのかというと、メッシュというのはプロセッサあたりのハードウエア資源がコンスタントで、マシン全体としても $O(1)$。本当はこの程度じゃないとダメです。もうちょっと先へ行くとこういうものがうれしい。要するにノードプロセッサあたりのハードウエア資源というのは、プロセッサ数に比例して小さくなる。マシン全体としてはコンスタントに、とにかくマシン全体としてコンスタントな資源である。そういった機能といいますか、こういうスケーラビリティを考えてマシンを作って行くということが必要ではないかと。以上、簡単ですけれども。

【後藤】

松岡先生、お願いします。

【松岡】

私はこの中では一番若造でありますし、非常にpureなソフト屋さんなので、アーキテクチャの方々のやってきたPIMの研究にケチをつけるというか、これはいかんと、そう意見をするというのは非常におこがましいというので、最初は後藤さんにお断り申し上げたのですが、ソフト屋さんというのは一体何を考えているんだということをどうしても知らせて欲しいということで、私もソフト屋の中でも極端な方かも知れませんが、今後こういう風な方面で研究をして行きたいと。ですから、必ずしもPIMに対するフィードバックというのが無いかも知れないんですが、それはお許しいただければ幸いです。 というわけで、これはメニューにあったのですが、並列推論マシンへの未来。赤い字を抜くと、「並列マシンの未来と展望」となるんですけれども、赤い字を読むと「並列推論マシンの未来展望はあるか」というふうになるわけです。赤い字の推論マシンの点はあるという風なことを実現するためには、ソフト屋さんにとって何が必要ないか、ないしは何を必要としているか。特に私とか、今世界でいろいろな並列マシンを使って細粒度言語などという無謀なものをインプリメントしよう、その中で世界最速のものを作ってやろう、などという大それたことを考えている人間、そういう変な人間は何を考えているか。ないしは、ソフト屋の衆迎に説法、ハードウエアをつくっている方々はこういう風なことを考えているんだよとおっしゃるかも知れません。しかし、ソフト屋から見ても、ソフト屋はこのくらい考えているんだよということを認識して頂ければ幸いです。最終的には、ワークステーションの二の舞にならないために、これはワークステーションで日本は非常に攻撃されてしまった訳ですけれども、並列マシンではやられないようにしたいなという私の願望が込められています。

それで、最近いろいろコンパイラの研究とか勉強とか、昔からABCL/1という言語を随分長くやっていますが、それを最近では本気で実装しようと。FORTRANに勝とうとは言いませんが、一応世界最速の並列オブジェクト指向言語を作ってやろうなどどいうことを考えていまして、そこから得られた知見、アーキテクチャに対してどういうものが要求されるのかということをまずお話ししたいと思います。

まず一番最初にすぐ分かったのは、gimmickは要らないということです。gimmickは要らないというのは、RISC思想というのは並列マシンでも適用するんだと。ソフトで最適化を行えば、もうgimmickは一切要らない。gimmickというのがどういうものかあえて言いませんが、要するに特定のものだけを速くするものを幾ら作ってもダメで、全体速度が速くなる所に殆どのハードウエアコストをかけていいと。ソフト屋は使わないから要らないんだと。

村上先生のお話にもありましたけれども、並列アーキテクチャの基本をきちんと守った機械を作ることが、ソフト屋にとっても結局一番大事です。その基本というのは計算と通信をオーバーラップさせて、レイテンシ・ハイディングができるということ、これが一番大事だと。我々にとっても一番うれしい。ただ、ポンとマシンが与えられるよりは、我々に与えられた時に、やはりトータルパッケージとしてうまく設計されることが何よりも大切だということ。当たり前のことが当たり前のようにやっぱりやってみて分かったということです。

それで、gimmickは要らないということです。これは昨日関口さんのお話で出てきたそうなんですが、細粒度並列言語というのを我々が良く勉強したり、作ってみると、やっぱり一皮むけばみんなになってしまうわけです。例えば関数型言語で、例えばId90というNikhilが作った言語がありますが、Id90を *Tというアーキテクチャに載っけるんですね。Id90をバークレーがやっているTAMというのに載っけるんですね。そうしたら殆ど変わらないわけです。実はこういう風な実行形式というものと、例えば我々が並列オブジェクト指向言語でやっているABCLをEM-4とか、AP-1000に載っけるというのは、細かいテクニカルな違いを述べればいろいろありますが、そんなに無い。むちゃくちゃある訳では無い。要するにプリンシプルとして、レイテンシを隠して、スレッド長をなるべく長

くとって、同期をコンパイラで減らし、どうしてもあるところはなるべく局所性を高めるなんていうプリンシプルはここら辺のどれをとっても同じです。

コミットチョイス言語でも、KL1、GHCで、例えば上田さんがFGCSで発表されたメッセージ指向の処理系とか、今、近山さんがやられている処理系なんかも、結局これもプリンシプルはほとんど一緒で、実は実装上もいろいろ似通った点があるんです。ですから、これを見れば明らかですけれども、実際gimmick は不要で、汎用マシンでもこういうものは動く。汎用マシンが速ければ速くなるほど、どんどんこういうものは速く動いて行くようなソフトウエア技術がまさに確立されようとしている訳です。

ですからまとめれば、結局そういう所でいかに速く実装するかの基本というのは、やはりいかに計算と通信をオーバーラップさせるか。例えばキャッシュとかメモリアクセスをきちんとしているとか、例えばプリフェッチはポストライトができるようなハードウエアになっているとか、スプリットフェーズの動作をきちんとサポートしているようなハードウエア、レイテンシ隠蔽のためには、ここでは詳しくは述べませんけれども、高速な同期というのが何といっても必要な訳です。

ただ単に特殊なハードウエアを作ったから使うというのではなくて、いろいろなコストとかを考えると、そういうものが汎用RISCで実現できれば、なるべくスレッド長を長くとれればこれは嬉しい訳ですね。そこら辺をうまくハードウエアとソフトウエアのバランスをとって、先ほど述べた並列計算の基本をうまく達成する技術というのはソフトウエアでかなりできている訳です。ですから、低消費電力の汎用RISCとせいぜいレイテンシ隠蔽の手助けになるような通信プロセッサ位のハードウエアで、あとはコンパイラやランタイムが頑張るんだと。ただ特定の言語パラダイムから独立した非常にいい汎用マシンを作ってくれれば、ソフトウエアは嬉しい。

じゃあそういうマシンが今あるかというと、そういうのは実は余りなくて、先ほど中島先生がプロセッサをつなげば何とかなるとおっしゃいましたが、でもやっぱりプロセッサをつなぐだけでは何とかならない。ソフトウエアから見て、ちょっとコストが高いなと思えるような言語が出て来たり、例えば大域同期とか、introspection というかモニタリングとか、OSを書くときに必要なプロテクションとか、producer/consumer 、これはアーキテクチャで上手くできますけれどもメモリを使ったいろいろな同期のサポートとか、こういう所が今の並列マシンでは弱いと。だからアーキテクチャ屋さんは今言ったようなものを使いながら、そういう所を非常に速くやるものを作ってくれるとうれしい。

じゃあ、ソフトウエアは何をやればいいかというと、ソフトウエアはひたすらコンパイラは頑張る訳なんですけれども、そこで疑問になるのが、PIMというのは、汎用ソフトウエア実装技術を研究するplatformになり得るかということがあって、私は申し訳ありませんが、不勉強でPIMを使ったことがないですし、コンパイラがどういう技術で実装されているかというのは正確には知りません。しかし、こういうコンパイラは、何か魅力的なplatformにしないと、ソフトウエア屋さんというのは飛びつかないわけです。コンパイラ屋さんは書いてくれない。

というわけで、最後になりますけど、結局、並列推論が成功するためには、ソフトウエア、コンパイラ実装、あと先ほど久門さんのお話にもありましたけど、一生懸命パフォーマンスを上げたり、ベンチマークのプログラムを書いてベンチマークをとるとか、そういうソフトウエア屋さんの育成が今後も鍵になるんじゃないか。この点で、最近いろいろ勉強していますと、わが国は、先ほどの細粒度言語に関してですが、大幅に遅れているという認識を強くして、非常に危機感を感じている訳です。私一人で頑張ってもしようがないので、うちの学生には一生懸命やらせていますし、もちろん日本でもいろいろな研究をやっていらっしゃる方がいるんですけれども、我々の感触でも、例えば並列オブジェクト指向で我々が考えていたことを例えば関数型では同じ問題をもう3年前にやられていて、彼らは彼らの解決をしている。彼らは一歩先を行っているのであぁ悔しいなという日々を過ごしているわけです。今、これからソフトウエアに10億円ぐらいくれれば、近山さんとか上田さんに10億円ぐらいあげれば、わが国も追いつくかなと、そういうようなことです。ソフトウエアのわがままと思って聞いてください。

【後藤】

どうもありがとうございました。じゃあ、関口さん、お願いします。

【関口】

電総研の関口です。昨日今日と色々このワーキンググループ、僕も大分サボってましたけれどもお話を伺ったり、今日既にここで4名の偉い先生方からお話しいただいて、だんだん私も喋ることが無くなって来ました。皆さんと大分重なってしまうんじゃないかなと思いながら前に出されて、後藤委員長に愚痴をまず述べて、拷問じゃないかなと思いますけど。そのあたりの愚痴からなんですけども、ICOTというのは電総研と非常に縁が深いということは私も前々から知っていまして、さりとて私も元々の営業分野であります数値アルゴリズム屋さんとしてはあまり関係がないと。ICOTというのは、うわさに聞いた三田の白い巨塔というイメージがありまして、私自身、ここに4年前まで足も踏み入れたこともなかった。それが何故こんな大事なパネルなんかに出せられるのかな、というのをまず恨みつらみとして、だんだん喋ることもなくなってきているなと思っているんですけれども。

1つは、こういう立場からして、ここに人工知能と書きましたけれども、知識情報処理についてどういう風に僕は前々から思っていたかというと、あるときのニュースで小学2年生並みの知能をついに計算機が持ったと。算数の問題だと思うんですけれども、文章題を読んで解けたというふうに大々的に新聞に発表になっているんです。その時最初に思ったのは、それは鶴亀算なんだから、こんなの簡単に解けるわなとポソッと思った訳です。私の方のイメージとしては、その鶴亀算だから、これは2×2のマトリクスだから、こんなもの、昨日のパネルじゃないですけど、サルでも解けると。小学2年生でも解ける。

ところが、その当時、私などが扱っていたのは、じゃあこれ鶴と亀じゃちょっと少ないんですけれども、1000×1000のマトリクスになったらどうするのと。そういう風にソフトウエアなりハードウエアのサポートがあって、たとえ鶴亀算みたいなのが解けたといって喜んでいても、そんなのは全然実世界と遊離しているんじゃなかろうかと。1000×1000といっても今でも大分小さいんですけれども、この辺が解けても、現実的な問題を解くことを考えてないんじゃないか。すなわち文章は理解しても、実際には解けてないという所で大分ギャップを感じているわけです。

また話が大分変わってしまうんですけれども、例えば数値解析、数値計算の方で使っているマシンというのは、だんだんこんな形になってくるであろうと。これは具体的に何かというのは申しませんが、ただ昨日も言いましたように、ハイパーキューブみたいなものはだんだん廃れてきまして、こういう割に単純な構造で頑張って行くんじゃなかろうか。そうなってくると、先ほどのギャップというのがますます大きくなりまして、リアルワールド、某プロジェクトとは関係ない一般名詞としてのリアルワールドなんですけれども、実際に解きたい世界というのがあると。ところが、世の中の解くべきマシンというのがこういう風に簡単な形になってきまして、そこへ何らかのプロジェクションなりマッピングという言葉を使っていますけれども、そういう風にこの世界のものをマシンにのっかりやすいように解釈してやると。そういうところで知識情報処理というのは非常に重要になってくるのではないかな。また逆に、ここのギャップがどんどん広がってきていますので、そのあたりをこれから埋めて欲しいなというのが1つの期待なのではありますが。

アーキテクチャの研究として、じゃあどうすればいいのと。もちろん先ほどのようにメッシュみたいなアーキテクチャを作ってしまうというのもあるんですけども、今後のアーキテクチャ研究として、とにかくしゃにむにマシンをつくればいいんでしょうかと。何を目指しているんだと。速度というものは、この委員会で先日多少米国の方に調査させていただいて、それのレポートを書いてないからまだここに居るのかなという愚痴もあるんですけど、速度というのはほとんど実装技術で決まってしまっていると。どうやって冷やすかというところが非常に大きな問題になってきて、ほとんどアーキテクチャというのは関係無くなってきているんじゃないかな。とにかくそういう状況において、とにかく次々、次々マシンを作ればいいのか。

今ちょうどICOTのプロジェクトは、ハードウエアというかマシンを作るプロジェクトというのは終わっていて、次を考えるときに何か忘れてはいないんでしょうかと。というので用意してきたOHPでは反省しなさいということを書いたわけですけれども、昨日今日のお話を良く伺っていますと、大分反省しておられるようなので、もう私の出番はないなと。

それでも最後にちょっと付け加えさせていただきますと、この反省にも多少数量的なアプローチが必要なのではなかろうか。実際、昨日今日のお話では、いろいろな反省点を数値として出しておられまして、特に言うことも無いんですけれども、ただ最近私興味持っているのは、ベンチマークというか性能評価技術ということに興味を持っています。先ほどの久門さんのお話にもあったのですが、ベンチマークというのは久門さんのお話だと、とにかくユーザからの視点であると。ユーザというのはベンチマークをやることによって幾つかシステムがあった場合に、それで何となくこっちを評価しておいて、実際自分が何か物を作ろうとした時に、それなりのことを推測するという風な、これはいわゆる従来までのベンチマークのアプローチな訳です。しかし実際これでは不十分でして、もちろんベンチマークにもいろいろなレベルがありまして実問題に近いようなベンチマークから、それからカーネルのループというのがこの世界にあるのかどうかわかりませんけれども、ライブラリのレベルとかループのレベルとかそれ以外のレベルがあると。数値計算の方は今までさんざんいろいろなユーザが居て、いろいろなマシンについてやってきていて、ICOTの世界は、今ちょうどなさっておられる所だと思うんですけれども、もう1つ今重要なのは、アーキテクチャというかシステムに対していかにフィードバックをかけるか。ICOTのシステム自体をいかに特徴付けるかというベンチマークをこれからやって行かなければいけないんだろうと。というのは、ベンチマークというのはユーザのためにあるのではなくて、久門氏が先ほどマトリクスを出して非常にクリアになったと思うんですけれども、ハードウエア・アーキテクトが次に何を改良しなきゃいけないか、次にどこを直さなきゃいけないかというのをクリアにするようなベンチマークというものを作らなければならないといいますか、ここをいかに考えて行くかということが一つの重要な点になってくるだろう。それをすることによって、何ができるかといいますと、新しいアーキテクチャを設計する時に、パフォーマンス予測、すなわち何をやった時にどうなるかという性能の予測をするという、このためにパフォーマンスをモデリングすると。先ほどで言いますと、アーキテクチャをいかにモデル化するか。1つのアプロー

チとして、先ほどの久門氏の共通一次というのは僕は非常に感銘を受けたんですけれども、あれだけのデータをとって、あれだけの相関解析をやって、あそこでまず不要なプログラムというのが分かる訳ですね。このベンチマークはやらんでいいと。4つぐらいあった中の1つだけ代表させてやればいい。その当たりをやって行きながら、それで、出てくる結論がappendに特化しているんじゃないかとか、それからVPIMが重いという風な、そういう定性的なものじゃなくて、もう少し何か定量的なものが言えないだろうか。そういうことを考えるのが重要になってくるのではなかろうかと。それによって、確立した言葉かどうか分かりませんけれども、Computational Computer Architecture Design。つまり数量化することによって、次のアーキテクチャというのをどんな風にして行くかという、デザイン、戦略、哲学などを確立して行って欲しいなということで、私の話を終わりにさせていただきたいと思います。

【後藤】

どうもありがとうございました。非常にもうズバズバと皆さんに話をして頂いたのですが、今全部のメモはとれなかったんですけど、こんな感じのお話でした。中島先生からは、少なくとも中島先生は分かったと話していますね。今後は多分いろいろ差が開いていく方向だろうと。それから天野先生はKL1から切り離した見方ができれば、結合網には楽しさいっぱいという非常に元気づけられる話をしていただきました。村上先生からは評価尺度が大事だと。今のは関口さんの話でもあるんでしょうけど、細粒度並列マシンというのをこれから見て行きたいという話と、スケーラビリティが大事だよと。松岡先生から、ソフトに任せないという小細工よりも、よく最近ノンノとかアンアンでベーシック・アイテムなんていう言葉を良く見かけるんですけれども、それを大事にして欲しい。後やっぱりコンパイラ屋が大切じゃないのという話が出ました。今、関口さんからは、実世界の問題をしっかりやりましょうと。それから今の数量的計算機アーキテクチャ設計ですか、数量的アプローチで反省しなさいと。非常にジーンとくる言葉ばっかりだったのですが、この辺でフロアの方で自由に討議を始めたいと思います。

司会，パネラーの面々，右から中島，後藤，松岡，天野，関口，村上の各氏

【??】

関口さんの数量的アプローチというのは全くそのとおりだと思います。そのお言葉を松岡さんにぶつけたいんですね。つまりもう余計な機能は要らないよと。もうベーシックな機能、計算と通信が速ければいいよと。全然ぜいたくを言ってないように聞こえますが実は凄いぜいたくを言っていまして、実際ハードを決める時に、どういうバランスでどこに重点を置き、ここは落とせないとか、いろいろなトレードオフを考えながらやるわけですね。ですからそこら辺をやっぱりsuggestionしてくれるようなことをソフト側から期待したいわけなんです。いかがですか。

【松岡】

これはちょっと細部に立ち入っちゃうんですけども、これはP-RISCというNikhilが91年に言ったアーキテクチャなんですけれども、これはよく見ると、ここで循環しているので、データフローの循環パイプラインみたいに見えますが、P-RISCの1番の基本というのは、ただ単にさっき村上先生が言われた通り、どんどん細粒度して行くんじゃなくて、ここの所でフレームを使ったRISC的動作をさせようというアイデアがあったわけです。ここいら辺のオペランドストアという所からコンティニュエーションが出て、ずっとある程度長いスレッドを確保して、実行した後でどうしても同期しなきゃいけなくなった時に、こちら側にハードウエアでぶち込もうと。そういうことをやって、ある程度の粒度を大きくするようなサポートをつけようという風に思ったわけです。

もう1つは、コミュニケーションのところで、例えばリモートの物を持ってくる時に、スプリットフェーズのトランザクションというのを可能にするために、ここからポンと出したら他のことをやって、そしてそのコンティニュエーションを持ったものが返ってきて、先ほど読んだ所から続けようなんていうことを考えた訳です。それでこれをソフトウエアで、先ほどスキップしちゃったんですけれども、実はこういう所というのは、ハードウエアじゃなくてソフトウエアでもできるということに気付いた人が居ました。それはactive messageと言うんですけれども、nCubeでアクティブメッセージをやると、例えば先ほどのパイプラインに出したり入れたりするのが、nCubeは実

は遅い遅いマシンなんですけれども、こういうハードウエア、nCube ではネットワークのインタフェースのアクセスが非常に遅いので、21 命令とか、34 命令位で、もうネットワークに手が届いてしまう。P-RISC ではこういうところをハードウエアがある程度自動化している訳ですが、こういう所も実はコンパイラが頑張ってやれば、スレッドのクラスタリングができるということを同じグループがやっていて、それが threaded abstract machine というものなんですね。じゃあそういう所で、彼らというのはどういう研究をやっているかというと、一生懸命色々なベンチマークを走らせて、スレッド長はどの位か、どの位のスレッドの融合ができるかとか、そういうことをいろいろ非常に数量的にやっているわけです。

その中で何が落とせなくなるかというと、やはり高速なレイテンシ・ハイディングが可能な高速なネットワーク経由の他ノードへのアクセスですね。そういう所は絶対落とせないな、となる訳です。実はそれさえあれば、逆に、こういう風にソフトが頑張れば結構速く行くということも分かってきた訳です。ですから、ぜいたくを言っているというか、速いマシンを作ってくれというのは確かにぜいたくなんですけれども、絶対我々として譲れないなというのは、ネットワークで他のプロセッサからいかにデータを持って来るか、他のプロセッサにいかにデータを送り込むか。どんどんプロセッサが速くなっていますから、局所動作をいかに速くするか。そのためにプリフェッチとか、ポストライトは必要だと。そういうところをアーキテクチャで押さえてくれれば、あとはコンパイラが何とかすると。ただ単に何とかするというのは、いい加減に何とかすると言っているんじゃなくて、そこで定量的にベンチマークを走らせて、先ほど関口さんがおっしゃったような定量的な評価によりこの位クラスタリングしたらこの位のスピードアップが得られるというのがだんだん分かってきますから、ある意味で言えば、そういうものの妨げにならないようなハードウエア、例えば自動的にスケジューリングするとか、そういうものは付けて欲しくないというところがあります。

もちろんこれは一番実行の下のところを実行しているわけで、その場合は、通常の細粒度並列でいわばブンブン回っている時の話です。実は、それ以外にもう１つ難しい問題があって、例えば今話が出た科研の超並列なんていうのがありますが、あそこにはOS を作るという話があります。OS を作るとなると、それとは全然別にプロテクションの話とかいうのが入ってくるわけです。そうすると、そこはパフォーマンス命でグルグル回しているのと、全然違う問題が入ってきて、そこは実は良く分からない。というのは先ほど申しました通り、こういうような速度が確保できるというのは、もう要するにハードウエアが直接アクセスしているからなんですね。こういうハードウエアで直接アクセスしているにもかかわらず、マルチユーザの環境であたかも今までの UNIX マシンを使っていたような OS、マルチユーザでマルチパーティションでなんていうのを、さっきの速度を犠牲しないでプロテクションを確保するにはどうするか、それにはどういうアーキテクチャがいいかというのは全然 trivial じゃないと思います。そういうことをきちんと実現したアーキテクチャというのはまだ無いです。私もどうやって作ったらいいのか分かりません。

【 ?? 】

何か思い付いたら喋ってしまうというので思い付いたんですが、今、メモリにアクセスするということは、直にメモリをアクセスしているような顔をしていて、実はページマップなどが入っていてプロテクションされていますよね。通信路というのもそういう機構が必要なんじゃないかという気がするんですが、そういうアーキテクチャの研究はすでにどこかにあるんでしょうか。

【松岡】

もうちょっと説明してください。

【 ?? 】

だからメモリはページマップなどを持っていて、ユーザは直接物理アドレスをアクセスできないんですね。論理アドレスをアクセスしている。それをハードウエアが TLB だとか何かいうのを介して、物理アドレスに変換してアクセスしている。だからあそこでユーザは直接ロード / ストアしているような気分で、実は間接的なロード / ストアをしている訳ですね。同じことが今の通信チャネルについても、直接通信チャネルをいじっているような気分になって、実はマッピングハードウエアかなんかが入っていて、直接扱っているんじゃなくて、例えば自分のプロセスに許された通信空間しかアクセスできない。そこをアクセスしている限りは、プロテクションができるというような仕組みというのは考えてもいいような気が、今松岡さんの話を聞いていてあるんですが、そんなことは考えていませんかね。

【松岡】

直接その答えになっているかどうか知りませんが分散型メモリの世界では、結局、メモリへのアクセスはすべて通信であるといった瞬間に、それはページマップがあって、それはちゃんとプロテクトがかかっていて、というような話にしようと。次に、それが十分かとかいう話は、この間、恐ろしい話があったんですけれども、あっちへ行ってはいけないとか、ある壁を越えて通信してはいけないとか、そんなマシンがありました。そういう恐ろしいハードウエアではとてもどう考えていいのか分からない訳です。そんな難しいことを言わなければ、その辺でプロテクトをかけるというのは、それしかできないとは思いますけれども、多分近山さんあたりはもうちょっと違ったイメージを持って

いると思いますが。
【??】
1つは、スタティックルーティングというやつがあって、プロセッサの番号は仮想でいいんですけど、それのルーティング情報を得る時に、それをあらかじめテーブルに入れておいて、そのテーブルをルックアップして、ルーティング情報を得るというアプローチがありますから、その中にプロテクションという概念を入れることは可能だと思います。

もう1つ、これは並列じゃないんですけれども、汎用機といいますか、IBMのOSなんかで、マルチプル・バーチャル・ストレージになっていて、要するにこれは、アドレス空間の間での通信を、この空間とこの空間では通信ができるかどうかというのをハードウエアでチェックする。ハードウエアといっても殆どマイクロコードですけれども、そういうのも昔からやられています。そういう概念というのは、要するにメモリの保護というのと同じように、通信についてもこれから並列マシンが汎用であるためには、これからプロテクション機構についてはいろいろと研究をやっていく必要があると。
【??】
できれば大学の先生に特許を取って頂いて、どんどん無償で公開して頂くのがいいんじゃないか。そうしないと、誰かが取っちゃいますから。
【後藤】
他にいかがでしょうか。松岡先生なんかからもソフトウエアから見て、明日から何々をやりなさいという話があったのですが、ハードウエア屋、アーキテクチャ屋、「元」が付いてもいいんですけど、明日からこうやりなさいというあたりの話はありませんか。
【近山】
ソフトのコンパイラ屋さんが足りないという話は、私、11年間ずっと感じて来たところでして、今月号のCACMの第5世代の特集で、その中で私が一つ書いた記事の中にたしか3回位コンパイラ屋が居なくて困ったと書きましたけど、このプロジェクトで何が分かったか。PIMを作って何が分かったか。何が良かったかの1つとして私が思うのは、今までハードウエアのことしか考えなかった人が、ソフトウエアもまとめて考えなきゃいけないなという気になったと。そういう部分が非常に大きかったんじゃないかと思います。

11年前の状況を考えると、日本でソフトウエアをやっている計算機専門家というのは非常に少なかったと思います。今でも少ない訳ですけど、今よりもっと少なかった。もっとずっとハードウエア志向でした。それは計算機科学の全体の流れをアメリカが先導している訳ですけど、それも最初はソフトウエアもハードウエアも全部1人の人が考えていた所から、まずハードウエアはどんどん進展して行って、それからソフトを考える人の人口が増えるというのが、後からついてきた感じがありました。日本もそれを追っているんですけれど、今、日本でソフトウエアのことを考える人口を増やすという点で、並列処理という部分において、このプロジェクトは何か少しは役に立ったんじゃないかという風に思います。評価の1つとしてこの点はあるんじゃないか。じゃあ、この先どうするのがいいか良く分からないんですが、とりあえずこの後継プロジェクトは14億円ばかりお金を頂けるそうで、方向としてはなるべくそういう方向を考えたいとは思っています。
【松岡】
一応誤解の無きよう申し上げておきます。私は別にこのプロジェクトはコンパイラのことを全然考えてないとか、そういうことを申し上げたのではなくて、実はICOTの中で、一番評価されるべき所というのは、幾つかもちろんありますけれども、他の所と比べて世間的評価が本当に得られるものより低いなと思っているのは、PIMとか、PIMOSとかそういう所を実装した技術だと思います。そこで育った人たち、私もそういう論文を読んで育ちましたから、そういうのを非常に尊敬しています。近山さんにこう言うのは何ですが、ぜひ並列マルチコンピュータ上で世界最速のKL1を作っていただきたいと。今のやり方でやれば、きっと可能なんじゃないかと思っています。
【後藤】
他にいかがでしょう。私も日頃感じていることなんですけど、今も松岡先生のところからアクティブメッセージの件ですか、ハードの工夫が実はコンパイラ屋さんがこうやればできるよという。僕もいわゆる小細工派だったのですが、小細工をしていろいろやっては、皆結構そういう経験があると思うんですが、それはコンパイラで何とかできるかも知れないと必ず後から言われるんですね。この後から言われるというのはどうしてなんだろう、もっと早く言ってくれれば良かったのにという思いが、多分、この辺にはたくさんそういう人が居るんじゃないかと思うんですけど、これはどうしてなんでしょうか。本質的にそうなんでしょうか。それともコンパイラ屋さんが少ないのか、冷たいのか。その辺、何かコメントございますか。
【??】
ハードウエアのほうがイメージが湧き易くて、いいアイデアが出るんじゃないですか。

【??】
結局ハードをつくるというのは簡単なんですよね。作っちゃえばいいから。作ってしまえば動くハードウエアというのは非常に簡単に作れまして、かなりイヤミが実は入っていたんですけど、実際に本当に簡単に作れまして、ハードウエアのロジックなんていうのは根が浅いものですから、ハードというのは作り易い。どうなるかというのが読み切り易い。パターンにハマれば早いと。何か楽なんですね。やっぱりコンパイラというのは、理屈で始まり、本当に作る、本当にやるという所とのギャップがある程度コンパイラというのは大きいから、例えばコンパイラに入れるバグの方がハードに入れるバグよりずっと多いとか、その辺の問題もあってなかなか難しいのかなというのが私の感触ですね。
【??】
それが普遍的事実だとすると、やっぱりハードが先行したほうが物事はうまく進むと。
【中島】
そんなことハードをやってもしようがないよと気が付くためには、ハードを一遍やらなきゃ、やってバカにされなきゃいけないという何かたまんない話があるんですけれども、多分そうだと思います。
【??】
コンパイラ屋さんの実情とか、たまに間近で見ていたりすると、やっぱり目先の仕事が忙しくて、そんなオプティマイズみたいなことまでわざわざ先回りして考えてやろうという程余裕がないという。先ほど人数が少ないというのと合わせて、そういう実情があるんじゃないかなと、そういう風に思いますけど。
【??】
ハードウエアではそういうのはないのでしょうか?
【??】
ハードウエアは何か思い付いてしまって、で、作らにゃいかんとか言ってどこかに発表してしまうと、そういう動機付けで作ってしまって。「使ってよ」とかいう風に言われると、「うーん、しようがない、使おう」という風になるんじゃないですかね。
【後藤】
そういう意味で、今出たように、引っ張るためにハードを作って行こうという人、何か話ないでしょうか。コンパイラ屋さんを目覚めさせるために俺はやるという、松岡先生のあれは違うんですか。そうは行かない?
【松岡】
難しい問題で、例えば我々のABCLの研究でもまさにそういうことがあって、例えば最初、EM-4の上でABCLを設計して、やっと最近N体問題とか動き出したんですが、そのAP-1000の上でも何とか同じことができないか、というのはやっぱり後から来たんですね。

どうやればEM-4みたいなある程度ハードウエア支援のスレッドがあるようなマシンに勝つのは難しいけれども、同じクロックで比較した時に同じレベルまで行くにはどうすればいいかというのをいろいろ考えて、今でもいろいろ考えているということな訳です。だからハードウエアで先に実現されたことが、ソフトウエアの実行効率のいい実装への指針となるというのは確かだと思います。ただ1つだけ言えるのは、汎用並列マシンというのはすでに世の中にいっぱい出てきていますから、今度はそうすると、最初から汎用マシンで速く動くというものが出てくるかもしれません。特にアメリカでは各計算機科学の学科にCMとかnCubeとかがいっぱい納入されだしていますから、そういうのに、例えば昔VAX750にBill Joyみたいな人がとりついてBSDを作ったように、そういう超並列マシンハッカーが張りついて、いきなり凄いものがソフトだけで飛び出すという可能性は否定できないと思います。
【天野】
やっぱり単純にコンパイラ屋とアーキテクチャ屋が没交渉であったというのが結構多いんじゃないかと思います。最近、僕はSNAILというマシンをやっていて、早稲田の笠原先生と密にミーティングをやったりしているんですが、アーキテクチャ屋さんがこれをやったらコンパイラ屋が喜ぶだろうなと思った機能を、コンパイラ屋さんはその半分位は全然喜ばないということがあります。半分位は当たっていて、こういう機能を入れればいいんじゃないと言うと、確かにそれは凄くいいなと言うんですが、半分位は全然嬉しくないとか言われます。あらかじめ話し合っただけでも結構違うんじゃないかなと。やっぱりかなり没交渉だったという点があるので、一人の人が何もかも理解しようというのはもう無理な時代なので、あらかじめ作る前に、平凡な話ですが、うんとコミュニケーションをすると。そういうのが大事なんじゃないかなという気が最近随分しております。
【藩】
短い答えでいいと思うんですけど、難しい答えかもしれない。今回のWGでは応用の話はほとんど出なかったんですよね。それで最後のパネルが「並列マシン未来への展望」ということだった訳だけど、やっぱりハードがあって、システムソフトがあって、それでもう1つ応用ソフトがあっての並列マシンだと思います。ここに並んでいる人は、

皆応用屋さんじゃないんだけれども、応用ソフトというのは、並列マシンが伸びていくためにどうでなきゃいけないか。あるいはアプリ屋さんがどうやってくれないと伸びていけないのか。あるいはそれと自分たちがやっている仕事との関係、何かその辺の話を一言、二言ずつ聞きたいんです。

【中島】

応用屋さんが並列プログラムを書いてくれなきゃ、とかツマンナイことを言ってもしようがないんですね。前に別のところでも言ったんですけど、多分応用屋さんが、褒くいい加減で、100回走らせると100回違う答えが出て来るような応用プログラムをたくさん生み出して、並列マシンでもあまり同期せずに動くというようなものを書いてくれるようになると、新たな展望が開けるのではないかと私は思っています。

【松岡】

正に仰る通りで、ICOTの応用プログラムの研究で、私は例えば並列推論のやつが褒く面白いと思っています。あとタイムワープをやった話は、非常にクオリティが高くて我々も同じようなことをやらきなゃいけないかなと思っています。ソフトで我々はアメリカなどに大きく負けてしまっているなと感じるのはいろいろあるんですけど、1つは応用の蓄積ですよね。ですから、彼らは一杯応用プログラムやベンチマークプログラムを持っていて、そこで例えばコンパイラで新しい最適化とかランタイムのアイデアがあると、パッとそれをかけられる。それで、ああこれは良かったとか、これはダメだったとか、そういうのが直ぐ分かる。そういう所の蓄積が我々の仕事だと全然薄くて、なかなか追いつけないという現状があります。ですから、コンパイラとかランタイムの実装の話でもアプリケーションはすごく大事で、我々も少しずつやっていますが、ちょっとまだまだ追いつけません。

【天野】

僕は瀧さんと一緒にワーキンググループでいろいろ並列CADをやったり並列アプリのほうにもかなり手を出しています。僕の経験から言うと、アプリ屋さんに書いてもらうためには、まず最初のギャップを小さくすることで、なるべくフレンドリな、例えば今までC言語で書いていた人がそのまま書いて何かUNIXに似たような環境で並列実行ができる、というのが多分取っかかりはいいと。

あと同期も最初はバリア同期などでボカッと書いて、またボカッと動かす。すぐパラレライズして、すぐ並列に動かすという環境があるほうがいいというのがまず第一です。

次に、もう1つ必要なのは、パフォーマンス・モニタです。先ほど松岡さんも言っていたんですけれども、パフォーマンス・モニタがあってすぐ状況が把握できると、プログラマにプレッシャーをかけられると。つまり、マシンが遅いんじゃないか、バスが遅いからこれ以上パフォーマンスは上がりませんとかプログラマが言ってきても、パフォーマンス・モニタで見て「ウソだろう、バスちっとも使ってないのに。遅いのはアルゴリズムが悪くて、インプリが悪いんだ、同期のとり方が悪いんだ」と言うと、割と直してくれて、かなり良くなっていくと。

だからこの2つがアプリ屋を呼び込む一番いい方法だと思うんです。そうすればアプリは結構たまって行きます。実は、いつか村上さんがJSPPのパネルで言っていたんですけれども、アプリ屋を増やさないとダメだと。アーキテクトの何割かはアプリ屋に転向した方がいいと言っていたんですけども、本当にそうだなと思うんです。だからなるべくそういうことをやって、つまりアプリ屋に受け入れられるような環境を作っていければなと思っています。

【関口】

例えばPAXというシステムなりプロジェクトというのがあって、それなりのユーザがついて来た理由は、やはり1つは、その性能自体が非常に高性能であった。ある特別なプログラムに対して高性能であったと。従来のスーパーコンで1カ月回して出る答えが、例えばPAXを専有して同じ時間をかけたとすれば1桁多く計算できて、例えば物理の方面ではその1桁多い数値というものが非常に価値があると。とにかくそういうアプリ屋さんがどうしてもこのマシンじゃなきゃダメだという風な機能をまず提供してもらえないと、なかなかそっちのほうには力を注いでくれない。ただ、一度もしそれが1桁、2桁、このマシンで出るんだよということが分かれば、その時には例えば物理屋さんなどはそれなりの労力をかけてくれる。言語がどうであれ何であれ、とにかくアセンブラでも何でも書いてくれるという位の力は持っていると思うんです。

もう少し普通のすそ野を広げるという意味でいうと、先ほどのアーキテクチャとアプリ屋さんの間のインタフェース、それは例えばライブラリであるとか、もちろん言語というのも1つのインタフェースですけれども、それがどこに行っても同じように使える。数学的なライブラリー、数学的なというのは、この分野ではどいう風に定義されているのか分かりませんが、そういう風なライブラリの標準化、例えば通信なんかの標準化ということが非常に重要になってくるんじゃないか。だからそのあたりで接点を見つけて、しかもそれは1つのマシンのためにというんじゃなくて、世の中のスタンダードというものもちゃんと動かすように設定してあれば、割にさっと移って来てくれるような気がします。

【村上】

さっき天野先生からちょうど2年ほど前のJSPPのパネルの時に言ったことを引き合いに出されて思い出したんです

けども、あの時何故それを言ったかというと、アプリ屋さんの数が限られていて、人口の自然増に比例した数でしか増えないとすると、アーキテクチャ屋がマシンを作れば作るほど、マシン当たりのアプリ屋さんというのは減って行く訳ですよね。そういう意味でアーキテクチャ屋はちょっとマシンを作るのを止めなさいよと。ちょっと熟成期間を置くというか。バランスのいいマシンであれば、少々性能が悪くてもそのマシンのライフサイクルが長ければアプリ屋さんはついてくる可能性はある。でもアーキテクチャ屋が次から次へとマシンを作って行くと、それもfancy なちょっとバランスの悪いようなマシンを作ると、とたんにそっぽを向かれてしまう可能性があると。ですから1つは、絶対性能が高ければ、先端的なアプリ屋さんはついてくると思います。しかし、それはごく一部の先端的なアプリ屋さんであって、そうでない平均的なユーザというのは、いかに絶対性能が高くてもやっぱりプログラミング言語あたりでジャンプしなくちゃいけない。きのうもシームレスという話をしたんですけど、要するにあまりにも飛び越える裂け目が大きいと、やっぱり尻込みしてconservativeな方に走ってしまう。ライフサイクルの長いマシンの方に走ってしまう可能性がある。ですから、アーキテクチャ屋の立場からアプリ屋さんを呼び込もうと思ったら、とことん自分たちの作ったマシンを評価していくべきだと思いますね。次から次へとfancy なマシンを作るよりは、ちょっと熟成させた方がいいと思います。

【久門】

今のベンチマークとかアプリケーションの話なんですけれども、僕はやっぱり並列マシンというのは、最近でこそ確かに商用のマシンが出ていますけれども、種類が多いと。種類が多いというのはどういうことかというと、種類の数だけ数値があって、その数値が互いに比較不可能であるという状況が現状じゃないかと思っているんです。例えばPIMでappend 600KRPSだからといって、それを何と比較するかというと、その比較のしようがないと。体重と身長を比べるような感じの比較の仕方しかできない。もしこれが、例えばこのマシンで走らせたらこの性能、このマシンで走らせたらこの性能というのを誰もが理解できるようなものさしで数字が出されていたとすれば、当然、みんなは大きな数字のものを注文する筈であると。それが例えばどんなベンチマークなのかというのはいろいろありますけれども、とにかく多くのマシンで走るようなベンチマークで、その数値の大小関係が一般のアプリケーション屋さんにとって理解できる、従って自分はこのマシンで走らせれば速く走るのではないか、と期待できるような共通の尺度が欠けている。各アプリケーションというのが、各マシンに特化している。あるマシンではこのアプリケーションは幾つだということができるけれども、それが他と比較不可能な状況だと、アプリケーション屋さんはどれを選んでいいのか分からないし、そもそも並列マシン上で努力するという気力が起きないんじゃないかと思う訳です。だからそういう意味でアプリケーションというかベンチマークプログラムを、ここでベンチマークと言っているのは僕がやっているようなものではなくて、もうちょっと理解できるベンチマークプログラムという意味です、それを並列マシン同上で共有するということが重要じゃないかと思いますけど、どう考えられますか。

【関口】

基本的にはおっしゃる通り、もちろん大筋の流れとしては、正に今久門さんが言われた通りだと思うんですけれども、ベンチマークについて言えば、ベンチマークプログラムそのものは理解されなくてもいいだろうと。ベンチマークで出てきた数値というものをそのままで理解して、何だかんだということは、多分必要ないんじゃないか。実際にやりたいことは何かというと、ユーザが自分で持っているプログラムを、いかにベンチマークの線型結合で表すか、もうちょっと違うコンビネーションになるかも知れませんけれども、何かそういう方向へ分解して、自分の持っているプログラムはベンチマークのこれとこれとこれがメジャーなパートであるから、これをこういう風に持ってくれば、これだけの性能が出るであろうと予測できる。そういうことが可能になればいいのであって、ベンチマークのプログラム自体が何か意味を持っている必要は無いと思います。

逆にそれはハードウエアの方からしても、例えばメモリのアクセスの速度だけを測っているベンチマークというのは実は余り意味が無くて、もうちょっと違う所で現れて来るようなものが何となく — 何となくと言うのは非常に歯切れが悪くて、というのは実際にそういうものを僕が提示している訳ではないので非常に歯切れが悪いんですけれども — その当たりでマシンのもう少し一段上でのレベルの性能が測れるようなベンチマークというものがあって、それを組み合わせれば、さらにもう一段上のライブラリみたいなのが評価ができて、さらにもう一段上の実アプリに近いものが評価できて、それを見ることによって実際にユーザが本当に自分の持っているプログラムが一番下で実行される時に、どれだけの性能で実行されるかという予測ができるような、上のレベルから下へのパス、下のレベルから上へまた戻っていくパスを実現すべきであろうと考えています。ですが具体的にどうかと言われると、今は答えがないというのが正直なところなんですが。

【後藤】

特にベンチマークの話では関口さんは専門家なので、幾らでも話が出ちゃうんですが、もうそろそろ時間なのでまとめたいと思います。PIM・WGを締めくくるパネルとしてどうだったかというのはフロアの皆さんに判断して頂くことにしまして、これはグランドミーティングですか、いずれにしろ、今後に対するいろいろなsuggestionも大分

はっきり出たんじゃないかと思っています。特にコンパイラ屋さんは自分がなるか育てるか、アプリ屋さんもそうだと思います。この部屋の中はハード屋さん、アーキテクチャ屋さんが多いと思うんですが、いろいろ反省はさせられたかも知れないけど、我いは今まで少なくとも並列処理のパイオニアではあったと。道を切り拓いてきたということに胸を張っていいんじゃないかと。胸を張りつつ反省して、コンパイラの勉強をしましょうというのが結論かも知れません。

これ以上、私もまとまらないんですが、最後に、田中英彦先生の総括が控えておりますので、パネルも含めて先生にバトンタッチしたいと思います。最後に、パネリストの方々に拍手をお願いします。(拍手)

(おわり)

パネル2スライド 後藤

並列マシン研究に多少疲れぎみのあなたに贈るメッセージ

- これまでに「....」はわかった.
- 明日から「...」をやろう
- 汎用並列マシンはどこまで主役になれる.

1

並列(推論)マシン：未来への展望

(a) PIMを作って何が分かったか？
何が分からなかったか？

(b) PIMのアーキテクチャ，ハードウェア

(c) 並列アーキテクチャ研究はどう進めるか（べき）

(d) 推論機／並列機として残すべき
アーキテクチャ／ハード

(e) 汎用並列マシンのベースとしてのPIM

(f) 汎用並列マシンへの期待

2

並列アーキテクチャ技術動向調査委員会

天野　関口
中島(委)　松岡
村上
久門　平田
(小畑)　(中田)

3

中島：差をひらく

天野：KL1のジュバク からの解放されれば：
結合網れが楽しさいっぱい

村上：評価尺度
細粒度並列マシン
H/Wスケーラビリティ

松岡：ソフトにまかせなさい.
小細工よりも「ベーシックアイテム」を！
コンパイラ屋を.

関口：実世界の問題を！
数量的アプローチで反省しなさい.

4

パネル2スライド 天野

PIM は KL1 の城塞から出られるか？

天野 英晴

慶應義塾大学 理工学部

PIM は大変成功したマシンである。

しかし、一般の並列計算機アーキテクチャ屋からは、推論専用の「特殊用途マシン」として扱われている。

- 並列推論マシンとしての目的充足型のストーリ展開。
- KL1 とその処理系への理解が前提。
- アプリケーションとアーキテクチャとの関連性に関する検討が少ない。

結合網周辺での一般化

PIM で用いられている多くの並列計算機構成技術は、大規模な並列計算機を研究開発するアーキテクチャ屋にとって、興味深い。

プロセッサのアーキテクチャに関しては、KL1 依存性が強い。
(と、いうより僕がわかってない。)

→ 特に結合網周辺で、一般的な細粒度並列計算機として議論できる部分が多い。

結合網とトラフィック

メッシュ、マルチハイパーキューブ、階層メッシュ、クロスバ等様々な結合網を用いたタイプがある。

非数値計算の細粒度プロセスのトラフィック解析

- 交信の局所性はどの程度あるのか。
- ランダム交信を仮定した理論的な評価と比べてどの程度実測値は異なるか。
- 結合網はどこが混雑するか。
- マルチハイパーキューブにおける交信負荷の分散はうまくいくのか。
- メッシュの場合、レイテンシィはどの程度か。
- 階層メッシュでは階層間の混雑はないか。

iv

クラスタ構造を持った大規模マシン。

従来のクラスタ構造を持つ並列マシンに比べ、桁はずれのスケールを持つ。

クラスタ内のプロセッサ数を 1-8 に変化させ、クラスタ内外のトラフィックを解析する。

- クラスタ内はプロセッサ数が多すぎないか。最適数は？
- クラスタ構造は効果があるのか。
- クラスタ構造とその大きさは、トラフィックにどのような影響を与えるか。

4

v

スヌープキャッシュ

スヌープキャッシュプロトコルに関しては、様ざまな提案がなされている割には、実際のマシン上での評価が乏しい。

- PIM/i(放送型) と他 (無効化型) のプロトコルによる性能の違い。
- ピンポン現象や、不要な放送による効率の低下は実際には起きているのか。
- シミュレーションと実測値はどの程度異なるのか。
- PIM/p のキャッシュ間メッセージ転送機能の効果は？

5

vi

KL1 の城塞

1. 現在の手続き型記述中心のプログラミング言語はいずれいきずまる。
2. その時はプログラムの記述力の点でも、並列処理の記述の点でも論理型プログラミングは大変有望で、将来は一般的になるにちがいない。
3. KL1 は並列処理を記述する論理型言語の中で優れた解の一つである。
4. PIM は KL1 を高速に効率良く実行でき、この点に関しては世界一の性能であり、素晴らしい。

PIM は KL1 の城塞に立てこもる以上無敵である。

6

vii

城塞から外へ

- 外部からの孤立。
- 城塞の限界が PIM の限界になる。
- PIM と KL1 が互いを必要とするが故にメジャーになれない。
  (しかし、KL1 は離陸可能か？
  と、すると心中もできないのでは？)

単体のプロセッサの性能が最近の RISC より遅くても気にしない。

並列計算機の研究用プロトタイプ

- プロセッサ本体の性能が、極端に低くないこと。
- 結合網の能力とバランスが取れていること。
- 計測機能

7

パネル2スライド 天野

結論

結合網周辺だけでも、評価、比較、検討するテーマは山ほどあり、これらについてKL1となるべく切り離した見方から、詳細な評価と綿密な検討を行ない、一般的な並列アーキテクチャの分野で積極的に発表活動を行なえば、PIMは将来の大規模細粒度並列マシンの元祖としての地位を確保する機会が残されている。

並列 (推論) マシン: 未来への展望

関口智嗣
電子技術総合研究所
sekiguchi@etl.go.jp

1993年3月23日

1

ICOT

- 電総研と縁が深いとはいえ
- 数値アルゴリズム屋には縁のない
- 噂に聞いた三田の白い巨塔
- 4年前まで足を踏み入れたこともなかった

2

人工知能について

小学2年生なみの知能：
＊文章題を読んで解けた＊

- そりゃあ，ツルカメ算だからやで (ボソッ)

3

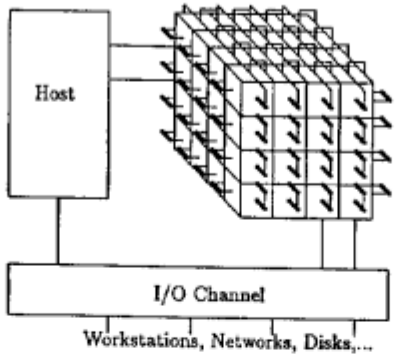
人工知能について (cont.)

- 2×2 は解けるわな
- 1000×1000 になったらどうするんやろ?
  → 実世界との遊離を感じた
- 文章は理解しても解けない

4

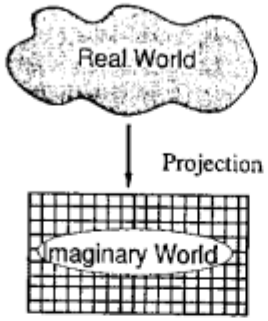
パネル2スライド関口

汎用の超並列マシンは今



5

超並列との Bridge



<< MPP>>

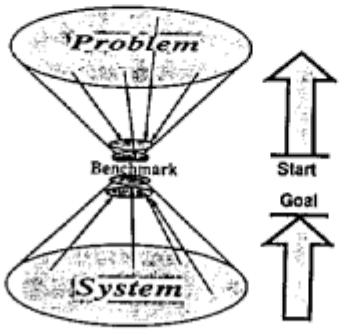
6

今後の並列アーキテクチャ研究

- 遮二無二マシンを作ればいいのか?
  → 何を目指す !?
  → 速度は実装技術 not アーキテクチャ
- 何か忘れていないか?
  → 反省

7

反省の数量的アプローチ



8

パネル2スライド開口

パネル2 スライド 関口

新しいアーキテクチャのために

- Performance Prediction をするための
- Performance Modeling の確立
- Computational Computer Architecture Design

パネル2スライド中島

PIMを作って何が分かったか

- ノードは適当に沢山つなげば何とかなる.
- 「分散処理」というのはうまくいくし, おそらくこれしかうまくいかない.
- 「速くできる処理」と「それなりの処理」の差が, どんどん大きくなる.

- 自動・動的負荷分散はたぶん夢.
- プロセス数＝nで飽和しそうなものは
  〃 ＝10nでは必ず飽和する.

ICOT PIM-WG パネル

並列推論マシンの未来への展望はあるか？

(or, ソフト屋は何が必要ないか。何が必要か？)

(or, ソフト屋の秋迎に説法)

(or, WSの二の舞にならないために)

松岡 聡
東大・理・情報

細粒度並列言語：一皮むけば皆ほぼ同じ

- 関数型言語 ⇒ 実装にギミックは不用
  Id90/*T, Id90/TAM [Nikhil][Culler]
- 並列オブジェクト指向言語
  ABCL/1 on EM-4, 1 on AP1000 [松岡,米澤]
  CST/1 on J-Machine [Dally, Horwat]
  CA/1 on CM-5 [Chien]
  !
- CCL ← PIMでは無く、汎用マシン上
  KL1・GHC [上田][近山]

3

得られた知見(アーキテクチャに対して)

- ギミックは(いらない)
  (ソフトで十分な最適化 ⇐ RISC思想)
- 並列アーキテクチャの基本が大事(演算と通信のオーバーラップ)
- 並列マシンはトータルパッケージが何よりも大切
  ↑
  ソフト・アーキテクチャ・実装技術

2

並列アーキテクチャの基本

(つかに計算と通信をオーバーラップ)

- 局所性の追求 ⇒ キャッシュ, メモリ階層
- プリフェッチ・ポストライト
- Split-Phase 動作 (レーテンシ隠ぺい)
  ⇒ 高速な通信・同期

これ等が実現されれば良い ⇒ 汎用
(コンパイラ + 多少のハード)

4

パネル2 スライド 松岡

現在の技術で「良い」パッケージ

汎用マシン（言語やパラダイムからは独立）
- PE：汎用RISC（ローパワー）＋通信コプロ
- ネットワーク：何でも良いから 高バンド幅・低レーテンシ
- ソフトでは高速化しにくい点のサポート
  - 大域同期
  - イントロスペクション
  - プロテクション
  - Producer-Consumer同期
  - メモリ同期
  - キャッシュへのネットワークからの書き込み

あとはコンパイラやランタイムの問題

並列マシンは パッケージだ

- ハードウェア実装技術＋コスト
- アーキテクチャ
- コンパイラ・ツール
- 言語

バランスし，先の「基本」が守られて限界のパフォーマンスが引き出せる．

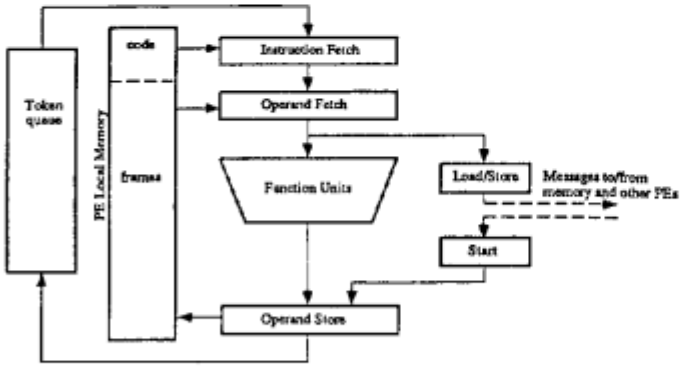
e.g. AP1000, nCube, Cray T3D

結局並列推論マシンの成功のためには，

ソフト屋（特にコンパイラ等の実装）の育生が力ギ

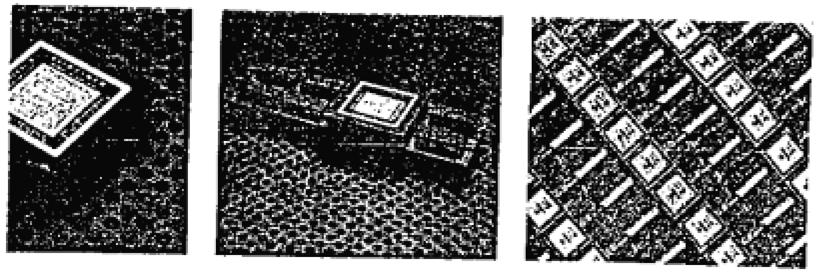
この点で，我が国はアメリカなどに大幅に遅れをとっている．

It's now or never!



⇒ Split-Phase操作の最適化

図 22: P-RISC の PE の構造（文献 [19] より）



物理パッケージングの例(nCube2)

9

◎PIMでは「汎用のソフトウェア実装技術」
として研究できるプラットフォームに
なり得るか?

・コンパイラ

・

・

10

(a) PIMを作って何が分かったか？
何が分からなかったか？
→ PIMを作ってないので何も分かっていない．

(b) PIMのアーキテクチャ，ハードウェア
→ ???

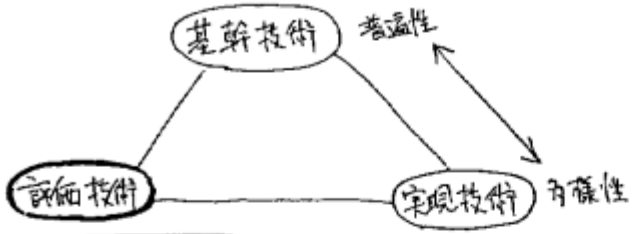
(c) 並列アーキテクチャ研究はどう進めるべきか？
→ 基幹技術，実現技術，評価技術

(d) ~~推論機~~/並列機として残すべきアーキテクチャ/ハード
→ レイテンシ・トレラント・マシンとしての細粒度並列マシン
→ スケーラビリティ

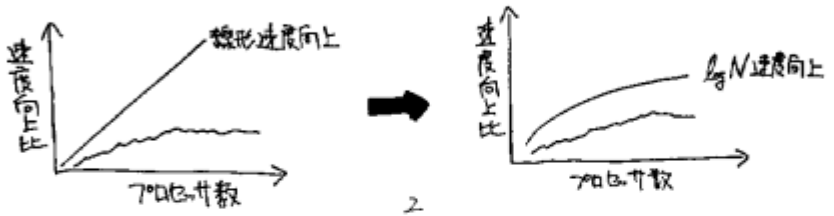
(e) 汎用並列マシンのベースとしてのPIM
→「コンパイルは任せきりで頑張ります」

(f) 汎用並列マシンへの期待
→ 何を置き換えるのだろう？

並列アーキテクチャ研究はどう進めるべきか？



評価指標/尺度



2

細粒度並列マシン

- 細粒度並列な問題にしか適さないマシン

- 粒度を細かくしてしまうマシン
  → (命令レベル)データフロー・マシン
  Fallay：スループットを上げりゃ速くなる
  粒度×並列度＝問題依存

粒度を大きくしよう

- 粒度が細かくても耐えられるマシン
  → レイテンシ・トレラント・マシン
  - レイテンシ・リダクション
  - レイテンシ・ハイディング

マルチスレッド・マシン

パネル2スライド村上

スケーラビリティ
– HW要求/コスト –

1ードあたり	マシン全体	
$O(N)$	$O(N^2)$	(交差線
$O(\log N)$	$O(N \log N)$	
$O(1)$	$O(N)$	メッシュ
$O(\frac{1}{N})$	$O(1)$	?

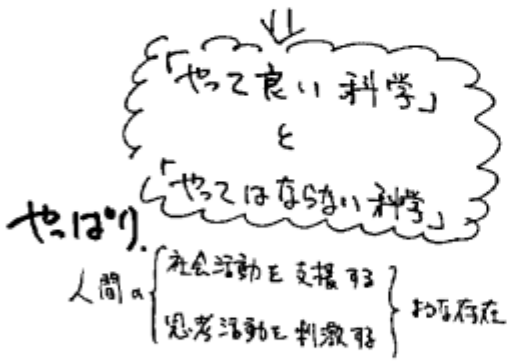
4

汎用並列マシンは
何を置き換えようと
しているのか？

→ メインフレーム？ I/O

→ スーパコンピュータ？ array

→ 人間（または，その脳）？



5

まとめに代えて:「膨大に技術・ノウハウのるつぼ」

発表者: 田中 英彦 主査
日時: 5:50 ～ 6:00, 3/23

【田中】
この１日半ですけども、一言も口を開くあたわざるで、くたびれました。ちょっと二、三まとめに替えまして喋ってみたいと思います。感想は、３日間フランス料理フルコース、朝昼晩という感じですが、結局の所、この１日半は、膨大ないろいろな技術のノウハウのるつぼであったと思います。

このワークショップの最初に、初めの言葉というのが無かったのでremindしておきます。11年間やったことを、今日明日で単なる終わりとするのではなく、それで止めるのではなく、意味をまとめようということ。それから、経験から出る実感を共有したい、これが重要だったろうと思います。とは言いましても、実際、真の意義というのが分かってくるのは数年後、いろいろな場を経験した上で初めていろいろ実感されて来るんだろうという風に思います。

プロジェクト自体の意義というのは、これは大きな話ですから、あまり言わない方がいいかも知れませんけれども、一応簡単に言うと、情報処理の基本モデル再考であった、11年。上から下まで全部やりました。それから知識処理パラダイムを作りました。それから、副作用として研究者層が厚くなって人が育った。人の輪が内外を含めてできた、これを大切にしたいなということじゃないかと思います。

この１日半、皆さんがもの凄くいろいろ沢山の言葉を出しました。それを全部並べると良かったのですが、私もさすがに眠ったりしましたので、ところどころ抜けてしまっています。１つずつ取り上げて行きたいと思いますけど、きっと必ずしも賛成じゃないというのがあるかと思いますので、その時はそう言って下さい。必要な時にはちょっと定量評価をやってみたいとも思います。「ハードは何回か作って始めて使える」、これはいいですね、PIMしかりという話です。それから「シミュレーションして作れ」、これもいいですね。それから、「メモリ容量は常に不足する」。それから「こうやると遅くなる」、"s"と書いてありますけれども、こういうことをたくさん聞きました。皆さんの中にたくさんノウハウがあると思います。速くなるというのももちろんあったんですけれども、遅くなるという方がたくさんありました。それから、「負荷分散からデータ分散の最適化へ」、これは余り言葉としては直接には出てこなかったかも知れませんけど、幾つかの方がチョコチョコと触れられていたんじゃないかと思います。これもいいですね。それから、「KL1とデータフロー、関数型とも言われたかも知れませんけれども、化粧落とせば皆同じ」。これはどうですか。イエスという方、手を挙げて頂けますか。半分ですね。化粧を落とせない、それも含めてダメということでしょう。50%。

「実証で世界をリードせよ」、これはいいですね。皆やって来たことですね。それから、「並列へのオーバーヘッドはかなりである」と言った人がいます。しかし、久門さんの絵の一部ではそうではなさそうな図も出ました。これはどうでしょう。「かなりである」というのは何だか分からないので、これは中島克人さんだったですかね、荘園などを含めて７割とおっしゃったんじゃないでしょうか。じゃあ２倍遅くなるというのでどうですか。オーバヘッドは倍ある。

「まとめに代えて」を発表される田中英彦主査

【フロア】
２倍以上ということはないんじゃないですか。
【田中】
では、それで行きましょう。２倍以下で十分できますという方は、どのぐらいおられますか。手を挙げて頂けますか。だから1.何倍のオーバヘッドで十分できると。何割かで済む。かなり少ないですね。今の感じは20%か25%。

それから「線の太さはスタイルを変える」、これはちょっと分かりにくいですね。これは「Bandwidthはプログラミングスタイルを変える」。何かちょっと女性週刊誌みたいですけど、これもいいですね。それから、次はちょっと問題があるかも知れません。「KL1にキャッシュは効かぬ」。

【フロア】
効かない処理があるということです。キャッシュは絶対万能ではないでしょうと。ソフト制御が必要だけどKL1ではできない。処理系の問題でしょう。
【田中】
では「反省に基づくVPIM light」。反省しろという人も居ましたので、反省というキーワードですね。これもいいですね。「デバグは急がば回れ」、これもいいですね。「5Gは小さなプロジェクト」、これは少し解説が要りますね。いろいろな成果を出したことに比較して投入したお金の額は小さかったという話ですね。
【フロア】
お金というか、マンパワーは必要でした。
【田中】
それから、「append RPSは無意味である:共通一次」。これはどうでしょうか。共通一次から来る警句という意味です。じゃあこれに賛成する方、手を挙げてください。微妙に7%です。少なくともappend RPSは押さえておくべきである。だがappend RPSはone of themであり過信は禁物というところでしょうか。

次、「PIMは超並列マシンの元祖である」。
【フロア】
ハードウエアでしょう。
【田中】
これはシステムではないでしょうか。これは努力次第でそうなり得ると仰ったと思います。これからの論文をどれほど頑張って書いて、一般化して世の中に見せるかがキーであると仰っていたと思います。今そうかどうかというのは、これはまだ議論があるかも知れません。「ソフト屋の育成が鍵である」と言った人もいます。これもいいですか。パーセントをとってもしようがないですね。あと幾つかありましたがこんな感じでした。

それで、今後の作業は、近山さんあたりがいろいろなさると思います。並列処理技術という意味では、やはりKL1からエキスを出すという意見が多かった思います。要するに技術を一般化して、汎用技術として易しく見せるという話。そうしないと良く分からないということですね。それから、並列言語に関してはこういう色々なものを、KL1から受け継ぎ、こんな機能も入れたり、こんなことを研究して下さいという注文もあったかと思います。要するにこれだと思います、「難しい言語は生き延びない」。万人にとって分かり易いキーとなるような概念にして、そして提示することが重要でしょうという意見が出たと思います。

それで、将来に向けてという話ですけれども、オリジナルな研究をやった実感を非常に大切にしたいと思います。人真似ではないオリジナルな研究を上から下までバッチリやったということ。それから、今後は好むと好まざるにかかわらず並列処理の時代で、何らかの意味で並列処理というのが入って来ると。そういう点から米国と日本を比べてみると、米国は実応用をやって規則的なプログラミングをたくさんやって蓄積に富んでいるというのが特徴でしょうか。日本は汎用並列というのをやって来たという実績があるのではないかと思います。並列は21世紀のあらゆる高度技術のベースになるというふうに言われていると思いますし、私もそう思います。そういう意味で、ぜひこの実感を大切にして、これから今後の未来のシステムを作り上げて行きたいと思います。後藤さんの「疲れ気味の」というパネルからも、いろいろなメッセージを頂いたことですし、ぜひ皆さん協力して未来のシステムを作り上げて行きたい、ということでまとめにさせて頂きたいと思います。どうも今日はご苦労さまでした。(拍手)

(おわり)

まとめに代えて.

「膨大に技術・ノウハウのるつぼ」

田中 英彦

3日間フランス料理フルコース
朝・昼・晩。

Remind しておくと

- 11年
- 単なる終りに止らず
  省みての意味をまとめる
- 経験から出る実感の
  共有の機会
- 数年後、真の意義が
  解って来る.

プロジェクトの評価

- 情報処理の基本モデル再考
  論理, 汎用並列, 上から下へ
- 知識情報処理パラダイムの確立
  エキスパートS. に止らず
- 副(主)作用
  研究者層の成育?
  人の輪(内・外)

2

警句集

- ハードは何回か作って，始めて使える
- きちんとシミュレーション後，作成すべし
- メモリ容量は常に不足する
- こうやると遅くなるs
- 負荷分散からデータ分散の最適化へ
- KL1とDF(関数型)化粧落せば皆同じ　50%
- 実証で世界をリードせよ
- 並列故のオーバヘッドは~~2倍~~ 2倍である　25%
- 線の太さはスタイルを変える
- KL1に キャッシュは効かぬ　分割系が悪い
- 反省に基づくVPIM light
- デバグは急がば回れ
- 5Gは小さなプロジェクト
- Append Lipsは無意味：共重一次　7%
- PIMは超並列マシンの元祖
- ソフト屋の育生が鍵

今後の作業

- 並列処理技術
  - 技術の一般化・汎用技術
  - KL1城からのEXODUS
- 言語並列
  - 単一代入，不完全データ
  - 配列の共有
  - ユーザへの見せ方
  - コンパイラ技術のリファイン

易しく解りやすいキー概念
万人のための技術

将来に向けて

- 全オリジナル研究をやった実感
- 今後は並列処理
  - 米：実応用，規則的の蓄積
  - 日：汎用並列の実績
- 21世紀のあらゆる高度技術のベース

⇒ 協力して作りあげよう！
つかれざれの！
風たさぬ，いざまきめやも．

まとめに代えてスライド