Vol. 37 No. 7 情報処理学会論文誌 July 1996

バンバン粒度制御：高並列汎用処理における最適粒度制御

日高康雄[†,☆] 小池汎平[†,☆☆] 田中英彦[†]

高並列計算機においては，計算速度が飽和領域に達する高負荷状態と，非飽和領域にとどまっている低負荷状態とでは，最適な実行方式はまったく異なる．飽和領域では，従来の逐次処理に基づいた粗粒度並列処理により，あらゆる余分なオーバヘッドを可能な限り削減することが重要であるが，一方，非飽和領域では，オーバヘッドと速度向上のトレードオフをとり，クリティカルパスを短縮することが重要である．本稿で提案する「バンバン粒度制御（BGC: Bang-Bang Granularity Control）」では，粗粒度（高負荷時用）と細粒度（低負荷時用）の2つのバージョンのコードを生成し，実行時の負荷状態に応じてそれらを切り替えて粒度を制御する．最適な中間粒度という従来の考えを捨て去り，両極端の2つの粒度を使う．コンパイル時の最適化を高負荷状態向けと低負荷状態向けに明確に分離することにより，長く議論されてきた粒度に関する問題の多くが，解決，もしくは，容易な問題に還元される．本方式を，高並列推論エンジンPIE64と細粒度非定型処理向きのコミッティッドチョイス型言語Flengを用いて，実装，評価したところ，1）単体プロセッサでは，ワークステーション上のC言語に匹敵する高い実行効率が得られ，2）問題の並列性が十分に高い場合は，逐次処理と比べても高い台数効果が得られ，3）問題の並列性が低い場合には，従来の細粒度並列処理と同じ限界までの速度向上が得られることが確認された．

Bang-Bang Granularity Control: Optimal Granularity Control in Highly-Parallel General-Purpose Processing

YASUO HIDAKA,[†,☆] HANPEI KOIKE[†,☆☆] and HIDEHIKO TANAKA[†]

In Bang-Bang Granularity Control, program execution is switched between two versions of code with extremely different levels of granularity according to runtime load level. The coarse-grained version is optimized for a heavily-loaded situation, and the fine-grained version is optimized for a lightly-loaded situation. By separating compilation for two heterogeneous situations and throwing away mixed concepts, such as optimal intermediate granularity, really many long-standing difficult problems concerned with granularity are successfully resolved or reduced to much easier problems. The scheme is implemented on a highly-parallel inference engine PIE64 using a committed-choice language Fleng. A quantitative evaluation of the actual system shows (1) single-processor performance comparable with the C language, (2) almost-linear parallel speedup under high concurrency, comparing even with sequential processing, and (3) the same parallel speedup under low concurrency as the maximum speedup attainable by a fine-grained parallel scheme.

1. はじめに

効率の良い並列処理を実現するには，負荷分散やスケジューリングの単位となる粒度の最適化が鍵となるが，最適な粒度を求めるのは容易ではない．端的にいえば，並列性の高いプログラムは過度の並列性を抑制して粒度を粗くした方がよく，並列性の低いプログラムは，並列性の低下を避けて粒度を細かくした方がよい[4),7),9),10)]．しかし，プログラムに内在する並列性は，実行時に与えられる入力データによって変化するうえ，実行の経過に従って時間変化する．

一方，真の高並列汎用処理を実現するには，数値計算などの定型処理だけでなく，記号処理や整数演算などの非定型処理をも満遍なくこなすことが要求されるが，非定型処理は一般に動的で複雑な計算パタンを持ち，コンパイラによる並列性の予測が困難である．

本稿では，このような高並列汎用処理において最適な粒度を得るために，あらかじめ粗粒度（高負荷時用）と細粒度（低負荷時用）の2種類のコードを用意し，実行時の負荷状態に応じてコードを切り替える，

† 東京大学工学系研究科
Faculty of Engineering, The University of Tokyo
☆ 現在，株式会社富士通研究所
Presently with Fujitsu Laboratories Ltd.
☆☆ 現在，電子技術総合研究所
Presently with Electrotechnical Laboratory

表1 逐次処理と各種並列処理方式の比較
Table 1 A comparison between various-granularity schemes.

	逐次処理	粗粒度並列処理	細粒度並列処理	バンバン粒度制御方式
計算順序	全順序	全順序（＋半順序） （部分的に半順序）	（全順序＋）半順序 （部分的に全順序）	全（＋半）⇄（全＋）半 （実行時に切り替え）
出発点	逐次言語	逐次言語	細粒度並列言語	細粒度並列言語
稼働率	100 %	×（特に並列性が低いとき）	○	○
実行効率	○	○	×（特に並列性が高いとき）	○
問題点	性能向上の限界	非定型処理の分割困難	オーバヘッド大	コードサイズが2倍
有効領域	————	並列性† ＞ N	並列性† ≲ N	全領域

N：PE 数 ×（レイテンシの隠蔽等のために，PE 1 台あたりに必要な並列性）．
†：プログラムに内在する本質的な並列性．小規模な並列処理では，十分に高い場合が多いが，高並列・超並列処理になるにつれて，PE 数に較べて相対的に低い場合が増えてくると考えられる．

「バンバン粒度制御：BGC（Bang-Bang Granularity Control）」を提案する．これは，「並列計算機の飽和領域における動作と非飽和領域における動作は，本質的に世界が異なる」という考えに基づいている．

もちろん，最適な実行のためには，負荷状態のほかにも考慮すべき要素は多い．しかし，それらの中で，まず第一に負荷状態を取りあげ，飽和領域における最適化と，非飽和領域における最適化を明確に分離することにより，実に様々な問題に対する答が自ずと明らかになる．たとえば，並列化によるオーバヘッドと速度向上のトレードオフを考える必要があるのは，非飽和領域だけである．飽和領域では並列化など考えず，逐次的な処理によりひたすらオーバヘッドの削減に専念すればよい．すると，大域スケジューリングでは非飽和領域のみを扱えばよく，fork した処理がただちに休止プロセッサで並列に処理されると仮定して，リモート実行のコストを考慮しつつクリティカルパスを短縮すればよい．

なお，バンバン粒度制御の名称は，最適制御理論におけるバンバン制御からとったものである．バンバン制御とは，リレーや弁の開閉のように制御入力が2値しかない制御のことをいう．最適制御理論によると，線形制御で制御入力範囲が有限であるときに，初期状態から最終状態への移動時間を最短化する最短時間制御問題では，制御入力をその可変範囲の両端の2値だけで切り替えるバンバン制御が最適であることが知られている[6]．

並列処理の粒度は線形制御ではなく，最適制御理論を単純には応用できないが，粒度の範囲にも制限（無限台のプロセッサで最適な細粒度限界と逐次処理の粒度）がある．また，粒度の最適化も，粒度自身の最適化を目標とすべきではない．プログラム実行開始時のメモリ等の内容を初期状態，終了時の内容を最終状態と考え，その間の移動時間，つまり，プログラムの実行時間を最短化することを目標とすべきである．本研究では，理論面の検討は将来の課題として残されているが，何らかの近似的な線形制御モデルを導入すれば，その近似範囲における最適性を示せるであろうと，我々は考えている☆．

なお，本稿では「粒度」という用語は，コンパイラで仮定している「コード粒度」（コンパイラの計算モデルで想定している抽象的な粒度）の意味で使っており，一般的な「計算量粒度」（1つのプロセスやタスクなどがになう処理量の粒度）とは異なるので注意されたい．実際，5章の最後に述べるように，バンバン粒度制御においても，計算量粒度には無数の中間レベルが生じる．バンバン粒度制御では，2つの両極のコード粒度と実行時の機構によって，計算量粒度を適正レベルに制御するのである．

本稿の構成は以下のとおりである．2章では，粗粒度並列処理と細粒度並列処理の得失を述べ，提案方式と比較する．3章では，低負荷時と高負荷時で，仮定される状況や最適実行形態，最適化方針がどのように異なるかを論じる．4章では，本方式を実装した PIE64 と Fleng について，5章では PIE64 におけるバンバン粒度制御の実装について，6章では性能評価について，それぞれ述べる．また，7章では関連研究について述べる．

2. 粗粒度並列 vs. 細粒度並列

表1に，逐次処理と各種並列処理方式の比較を示す．

粗粒度並列処理は，逐次処理に半順序関係を部分的に導入したもので，逐次処理と同等の高い実行効率という利点を持つ反面，逐次プログラムの分割という問題がある．特に，非定型処理における自動分割は困難

☆ 厳密にいえば，最適性がまだ証明されていないため，現時点では，バンバン粒度制御は単なる2値制御の一種でしかない．最適性が理論的に証明されると，最適制御の一種となる．

で，現状ではプログラマが明示的に分割や同期の指示を行っているが，同期のバグは発見が難しく，それを避けようと多数の同期命令を挿入すると，並列性が不足して高い稼働率を得られない．

一方，すべての計算を半順序関係として表現する細粒度並列処理は，プログラムに内在するすべての並列性を容易に抽出できる，同期のバグがあまり起こらない，という利点を持つ．しかし，頻繁に行われる同期処理等のオーバヘッドが大きく，逐次処理や粗粒度並列処理と較べて効率が悪いという問題点がある．

バンバン粒度制御方式は，上記両方式の利点を組み合わせた方式である．すなわち，細粒度言語を出発点として最大限の並列性を抽出する．十分に高い並列性が得られているときは，細粒度言語の fork を手続き呼び出しやループに置き換えて全面的に逐次化し，逐次型言語で発達してきた全順序に基づく最適化手法（たとえば，手続き間にまたがるレジスタ割当て，インライン化，ループアンローリング，etc.）を用いて，逐次処理なみの高い実行効率を達成し，逐次処理と比べても十分に高い台数効果を目指す．逆に，並列性が低いときは，きわめて頻繁に同期をとって処理間の半順序性を保つ細粒度並列処理により，並列性および稼働率の向上をはかり，可能な限りの速度向上を目指す．

3. 負荷状態により異なる2つの世界

要素プロセッサ（PE）数に比べて十分に高い並列性が得られている「高負荷状態」と，相対的に低い並列性しか得られていない「低負荷状態」の，最も本質的な違いは，休止中の PE が存在するか，全 PE が稼働中かであり，そこから，様々な状況の違いが生じ，最適実行形態や最適化方針がまったく異なってくる（**表2**）．

低負荷状態においては，処理を fork すれば休止中の PE によってただちに並列に実行される．ゆえに，実行待ちの ready queue の長さはほぼつねに 0 であり，処理は本質的に FIFO 順序，priority をつけても効果はない．

低負荷状態における最適実行形態は，細粒度並列処理が基本であり，FIFO 順序により自ずと幅優先実行となる．コンパイル時の最適化では，ループを木状に分割し，局所性より並列性を重視し，逐次化は依存関係のある部分にとどめるべきである．ただし，リモート実行コストの考慮は重要で，fork した処理がただちに並列実行されると仮定した大域スケジューリングによって，クリティカルパスの最短化をはかり，実行時間を直接短縮すべきである．また，PE 内の局所スケジューリングに意味はなく，依存関係に沿った負荷分散が重要である．休止中の PE を遊ばせることはないため，投機的計算が有効であり，サスペンド☆を回避する意義は少ない．

表2 低負荷状態と高負荷状態の比較
Table 2 A comparison between a lightly-loaded situation and a heavily-loaded situation.

低負荷状態	高負荷状態
状況の違い	
並列性が低いとき	並列性が高いとき
休止中の PE が存在	全 PE が稼働中
fork すればただちに実行	fork しても後回し
|ready queue|=0	|ready queue| ≥0
本質的に FIFO 順序	FIFO/LIFO ともに可
priority は無効	priority が有効
最適実行形態，最適化方針の違い	
細粒度並列処理が基本	逐次処理が基本
幅優先実行	深さ優先実行
ループの木状分割	ループアンローリング
並列性重視	ローカリティ重視
クリティカルパスの短縮	オーバヘッドの削減
実行時間を直接短縮	実行時間を間接短縮
負荷分散重視	スケジューリング重視
（大域スケジューリング）	（局所スケジューリング）
投機的計算が有効	サスペンド予測が有効

一方高負荷状態では全 PE が稼働中で，処理を fork しても ready queue に入り後回しになる．queue には実行待ちの処理があり，FIFO や LIFO，priority の導入など，様々な局所スケジューリングが可能である．

高負荷状態における最適実行形態は，逐次処理が基本であり，スタックを用いて過剰な並列性を自動抑制可能な，深さ優先実行である．コンパイル時の最適化では，ループアンローリングや fork の逐次化が重要である．オーバヘッドを削減し，ローカリティを向上させることで，実行時間は間接的に短縮される．負荷分散より PE 内の局所スケジューリングが重要であり，サスペンドはできるだけ回避すべきで投機的計算に価値はない．

このように，高負荷状態と低負荷状態では，プログラムの最適化方針が 180° 異なる．このため本研究では，粗粒度モードと細粒度モードの2種類のコードを用意し，それぞれ異なる方針で最適化し，実行時の負荷状態によってそれらを切り替える方式をとる．実装上のポイントは，粗粒度モードから細粒度モードへの切り替えである．粗粒度モードで負荷状態が低下しても逐次実行を続けると稼働率が低下するため，いったん逐次実行を解いて，並列性を高める必要がある．

☆ 未計算のデータが参照されたとき，現在の処理を後で再実行するように，中断・延期すること．理想的なスケジューリングでは生じない．細粒度並列処理では顕著なオーバヘッド要因のひとつ．

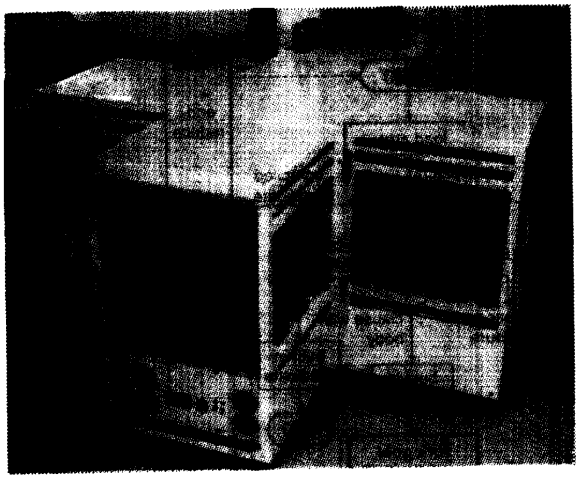


図1 並列推論エンジン PIE64．64台の推論ユニットと2系統の相互結合網を持つ．自動負荷分散機能を使って，マシン全体の負荷状態を容易に判断できる．

Fig. 1 PIE64 consists of 64 inference units and two interconnection networks. Global load level is easily observed with an automatic load balancing facility.

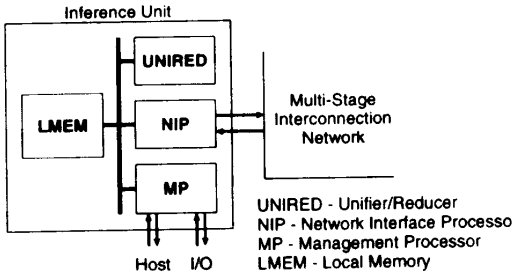


図2 推論ユニットの構成．計算をになうUNIREDは，NUMA型共有メモリを提供し，サイクルごとのコンテクスト切り替えにより通信レイテンシを隠蔽する．

Fig. 2 Organization of an Inference Unit. Computation is performed by UNIRED. UNIRED offers a global address space of NUMA-type distributed shared memory, and hides communication latency by interleaved multiple contexts with pipeline interlocking.

4. 並列推論エンジン PIE64

本方式は，並列推論エンジン PIE64[1]とコミッティッドチョイス型言語 Fleng[8]を用いて実装した．PIE64は，推論ユニット（Inference Unit: IU）と呼ばれる要素プロセッサを64台と相互結合網を2系統持つ，ノイマン型のMIMD並列計算機である（図1）．

4.1 PIE64のアーキテクチャ

相互結合網[13]は，負荷最小IUを自動的に選択する自動負荷分散機能を持つ．この機能を利用すると，マシン全体の負荷状態を容易に判断することもできる．すなわち，各IUが申告した負荷値は結合網内で比較され，最小値が各IUにフィードバックされるため，この最小負荷値を各IUがローカルに閾値と比較すればよい．

1台のIUは，計算をになうUNIRED（Unifier/Reducer），管理をになうMP（Management Processor），通信をになうNIP（Network Interface Processor）の3種類のプロセッサ等から構成される（図2）．

UNIRED[12]は，プログラムを実行するメインプロセッサである．一般的なRISC型命令セットとFlengに適した若干の命令を持ち，リモートメモリへの参照命令をNIPへのコマンドに自動変換するNUMAアーキテクチャを持つ．また，4セットのレジスタファイルとプログラムカウンタをサイクルごとに切り替えるマルチコンテクスト処理で，通信レイテンシを隠蔽する．一方，実行可能なコンテクストのみを先行制御パイプライン☆に連続投入し，並列性不足時の効率低下を防いでいる．

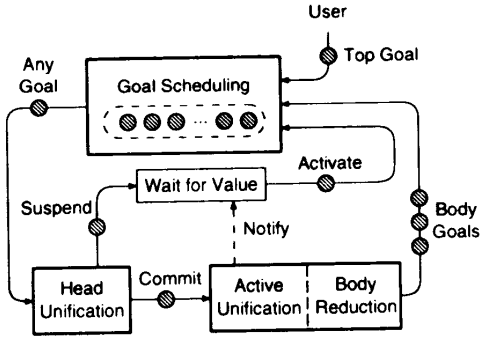


図3 コミッティッドチョイス型言語 Fleng の実行モデル

Fig. 3 Execution model of Committed-Choice Language Fleng.

MPは，負荷分散，スケジューリングを始めとする管理処理をにない，汎用RISCであるSPARCと，そのファームウェアである並列処理管理カーネル[4]によって構成される．なお，MPの処理がボトルネックとなったため，現在の実装では，管理処理の一部をUNIREDに移し，MPの負荷を減らしている．

4.2 コミッティッドチョイス型言語 Fleng

Fleng[8]は細粒度並列処理向きの言語で，PARLOG，Concurrent Prolog，GHCなどのコミッティッドチョイス型言語，並列論理型言語[11]のひとつである．

Flengでは処理の単位をゴールと呼び，プログラムは定義節の集合として与える．プログラムの実行（図3）は，1）キューからゴールを取り出し，2）そのゴールと左辺（ヘッド部）がユニファイ可能な定義節を選び，3）その節の右辺（ボディー部）の組み込み述語（ア

☆ ステージ間のデータ依存関係の整合性を保つインタロック機構を備えたパイプライン．

クティブユニファイ等）を実行し，4）ボディー部のゴールに書き換えてキューに戻す，という操作の繰返しである．

ゴールの実行順序に制約はなく，すべて並列に実行しても構わないが，ヘッド部の実行で変数値が未定義であると，そのゴールはサスペンドする．サスペンドしたゴールは，他のゴールの実行によって変数値が具体化されると，アクティベイトされキューに戻される．なお，変数は単一代入であり，バックトラックはいっさい行わない．

4.3 従来の細粒度方式による実装

プログラムとして与えられた定義節の集合は，述語ごとにまとめてコンパイルされる．述語は，逐次型言語の手続きや関数の定義に相当し，ゴールは，手続き呼び出しや関数呼び出しに相当する．

従来の細粒度方式実装における処理の流れを，**図4**に示す．UNIREDはディスパッチルーチンで引数をレジスタにロードし，該当述語のエントリポイント（EP）にジャンプする．ヘッドユニファイに成功して定義節が選択されると，ボディー部組み込み述語を実行，ボディーゴールを1つを除いてすべてforkし，残る1つのゴールを末尾呼び出し（tail call：再帰に限らない）により実行する．この処理を，ボディゴールのない定義節が選択されるか，サスペンドが生じるまで繰り返す．

ゴールのforkでは，ヒープメモリ上に述語記号や引数を書き込んだゴールフレーム（GF）を作成して，NIP（任意IU/指定IUで実行するとき），またはMP（ローカルIUで実行するとき）に投げる．このforkはかなり軽いとはいえ，逐次型言語の手続き呼び出しに比べると，以下のようなオーバヘッドがある．

- 手続き呼び出しの引数はレジスタ渡しであるのに対し，ゴールforkではメモリ渡しになる．
- 手続き呼び出しでは，親手続き内のcall命令で直接呼び出すのに対し，ゴールforkでは，GF上の述語記号の格納/読み出し，間接ジャンプになる．
- 手続き呼び出しでは必要のない，ゴールキューのenqueue/dequeue処理（MP上）が必要になる．

一方，tail callは，引数をレジスタ上に用意してEPに直接ジャンプするため，逐次型言語の手続き呼び出しとほぼ同じコストである．また，ヘッドユニフィケーションの処理は，引数の型や値による分岐であり，逐次型言語の条件分岐に相当する．そのコストは，サスペンドを生じなければ条件分岐と同程度であり，forkやtail callよりも一般にかなり小さい．

なお，図4では省略されているが，サスペンド時に

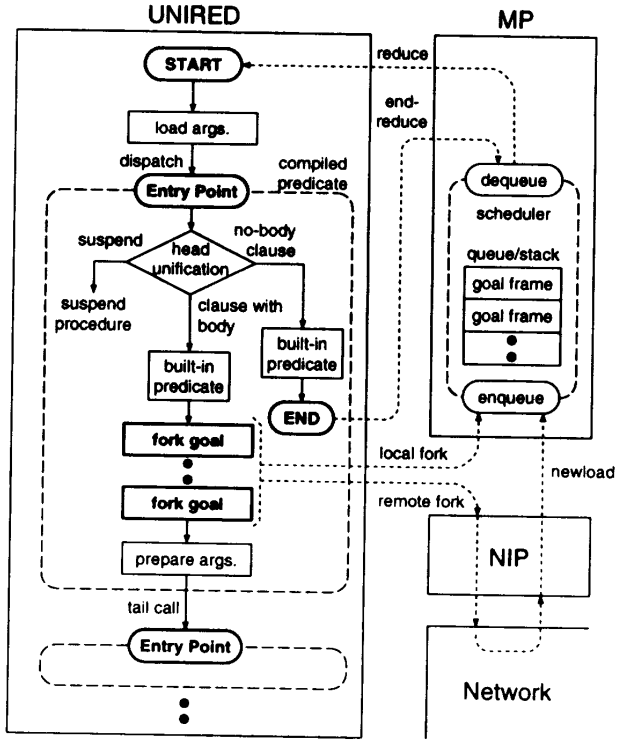


図4 細粒度方式における処理の流れ
Fig. 4 Control flow in the fine-grained scheme.

は，レジスタ上の引数をGFに退避し，サスペンド登録コマンドをNIPに発行し，MPにendreduceを発行して，当該コンテクストを停止させる．

5. PIE64におけるバンバン粒度制御の実装

5.1 バンバン粒度制御方式の実装

バンバン粒度制御方式における処理の流れを**図5**に示す．各述語は細粒度モードと粗粒度モードの2種類のコードを持ち，それぞれEPを持つ．細粒度モードでは，従来同様のforkとtail callによる実行を行い，粗粒度モードでは，forkの代わりにcallによる逐次的実行を行う．また，全体的な実行の様子は**図6**に示すようになり，過渡的には，IU間やUNIRED内のコンテクスト間で，2つのモードが混在する．

負荷状態判定のオーバヘッドを減らすために，判定はディスパッチルーチンのみで行い，以後，そのコンテクストは停止するまで同じモードで実行する．ただし，粗粒度モードで長時間の逐次実行を行っている間に低負荷状態に戻ると，MPによりbreak（一種の割り込み）がかかる（後述）．

粗粒度モードにおけるcallは，引数（レジスタ上）と戻りアドレス（スタック上☆）を用意してEPに飛

☆ tail callのオーバヘッドを増やさずに，同じEPを使うため．

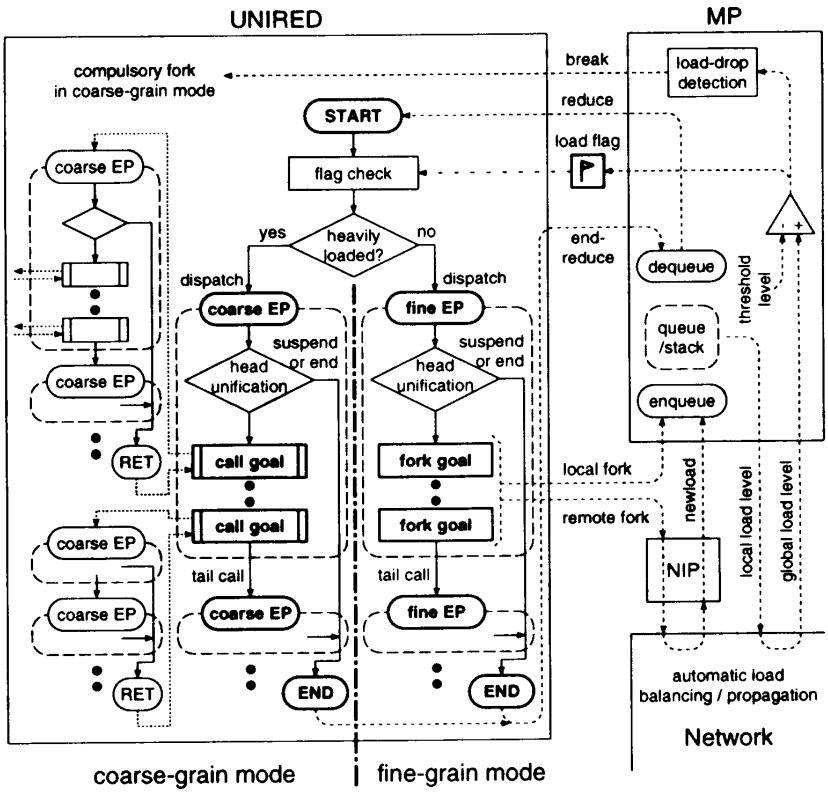


図5 バンバン粒度制御方式における処理の流れ
Fig. 5 Control flow in the BGC scheme.

ぶだけで，fork 処理は部分的にもいっさい行わない．呼び出し側は逐次型言語の手続き呼び出しとほぼ同じで，手続き型言語と同様のスタックフレーム（SF）に環境を保存する．SF は，ヘッド部の実行後（定義節が確定したとき）に確保し，末尾呼び出しの直前に開放する．手続き呼び出しとの違いは，呼ばれた側で入力引数をつねに検査することであるが，そのコストは小さい☆．

5.2 負荷状態に応じた粒度切り替えの実現

MP 上のスケジューラは**表 3** の 6 つの状態を持ち，キューの状態などに応じて，**図 7** に示す状態遷移を行い，HIGH 状態においてメモリ上のフラグを高負荷状態にする．UNIRED はこのフラグを見て，細粒度/粗粒度モードを選択する．なお，図 7 中の H，L は，MP の負荷状態判断を表す．これは，稼働コンテクスト数+実行待ちゴール数を自分の負荷値として結合網に流し，結合網から返される最小負荷値を，最大使用コンテクスト数+オフセットと比較したもので，一致，または，上回るときに H，下回るときに L と判断している．

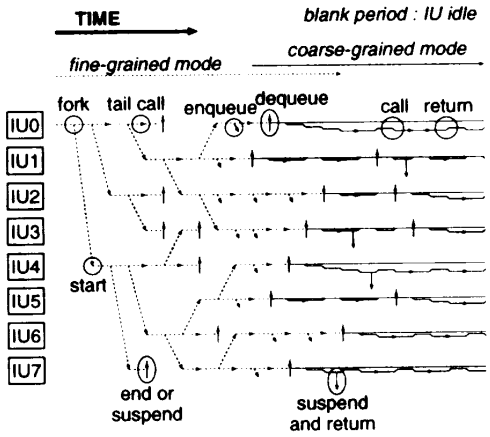


図6 バンバン粒度制御方式の全体実行イメージ
Fig. 6 A global execution image of the BGC scheme.

HIGH 状態で L を検出すると，メモリ上のフラグを低負荷状態に戻し，SUSTAIN に遷移するが，粗粒度モードをただちに停止させることはしない．SUSTAIN では，MP は queue（通常は LIFO だが，このときは FIFO）内のゴールを定期的に取り出して負荷分散を

☆ 逐次化に依存して入力引数の検査を省くと，負荷低下時の並列性抽出が制限される．一方，出力引数は，未定義変数渡しと値の代入を，戻り値のレジスタ渡しに置き換えることで，参照側の変数読み出しも含め，オーバヘッドを大きく削減できる．もし負荷の低下で強制 fork されたら，未定義変数を確保して戻せばよく，その値を渡された後の述語は，入力引数を検査するので問題ない．この方式は現在評価中で，本稿の評価には含まれていない．

表3 スケジューラの状態
Table 3 States in the scheduler.

状態	UNIRED context	実行待ちゴール	負荷フラグ	実行モード
FREE	空きあり	なし	低	細
JUST	満杯	なし	低	細
LOW	満杯	あり	低	細
HIGH	満杯	あり	高	細/粗
SUSTAIN	満杯	あり	低	細/粗
RELEASE	満杯	なし	低	細/粗

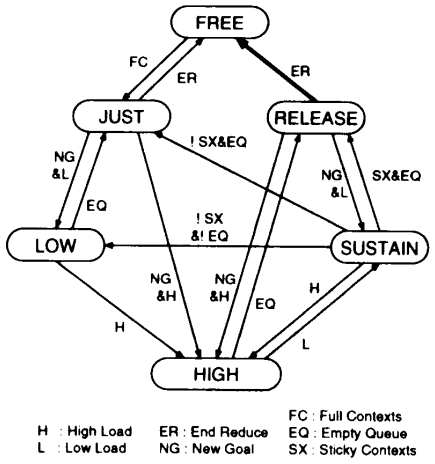


図7 スケジューラの状態遷移．太矢印の遷移のときに粗粒度モードを続けているコンテクストに break をかけ，強制 fork させる．

Fig. 7 State transition in the scheduler. On a transition of the thick arrow, MP sends a *break* command, which causes compulsory forks, to the sticky contexts: the contexts that are still running in CGM.

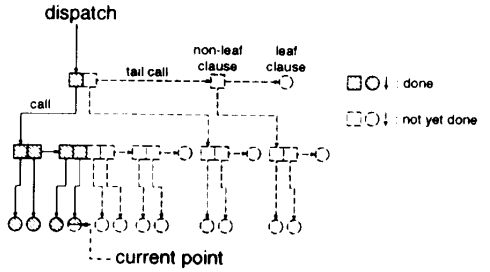
行い，H を検出すれば再び HIGH 状態に遷移する．

一方，FREE 状態に戻るときに，まだ粗粒度モードを続けているコンテクスト（sticky context）があれば，そのコンテクストに break をかけて強制 fork させる．

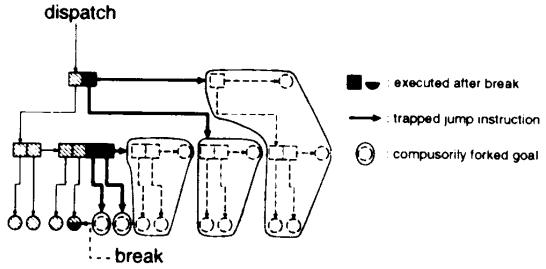
この強制 fork のために，EP の呼び出しには，スリットチェック機能☆を持ったジャンプ命令（break がかけられていると，trap ルーチンに飛ぶ）を使う．trap ルーチンでは，サスペンド時と同様にレジスタ上の引数を GF に退避して fork し，呼び出し元に戻る．これをスタックが空になるまで続ける．つまり，図8 に示すように，実行途中のボディー部の残りをすべて実行しつつ，未実行の EP へのジャンプを trap してゴールを fork させる．

本手法ではスタックを完全に崩してしまうが，一般にはスタックの一部を fork させる手法もある．その

☆ めったに成立しないが定期的に検査すべき条件を，ループジャンプなどの命令に隠れて検査する機能．命令実行時に条件が成立していると，通常とは別の動作を行う．



(a) 強制 fork しない場合の計算過程



(b) break により，強制 fork されるゴール

図8 粗粒度モードにおける負荷低下時の並列性抽出．

Fig. 8 Concurrency extraction in the coarse-grain mode on a load-level drop.

場合，計算量の多いスタックの底側の処理を fork させるべきである[7]が，そうするには単純な環境スタックよりも複雑な構造が必要で，call, return 時のオーバヘッドが大きくなる．本研究では，高負荷状態での効率向上を重視して call, return の処理をできる限り単純化した．負荷低下時の速やかな並列性の向上と，SUSTAIN 状態での MP からの負荷分散により，臨界飽和状態にヒステリシスを作り，強制 fork の回数をおさえている．

強制 fork の結果，再び高負荷状態に戻れば，fork されたゴールは粗粒度モードで実行されるが，各ゴールの計算量は fork 前の残存計算量より少なく，飽和状態を保つのに適正な計算量に制御される．すなわち，「コード粒度」は同じでも，「計算量粒度」は細かくなっており，序論で述べたように「計算量粒度」には無数の中間レベルが存在する．fork 前の残存計算量が少なければ，そのまま細粒度モードに移行する．

6. 性能評価

提案したバンバン粒度制御に基づく実行時処理系を実装し，ハンドコンパイルによって評価した．プログラムには，N-Queens の全解探索問題を使い，N の値は 6，8，11（それぞれ Q6，Q8，Q11 と表記）として並列性を変化させた．N=6，8 のときは実行時間が短か過ぎるため，N=6 のときは 1000 回，8 のときは 100 回，全解探索を繰り返させた．使用する UNIRED

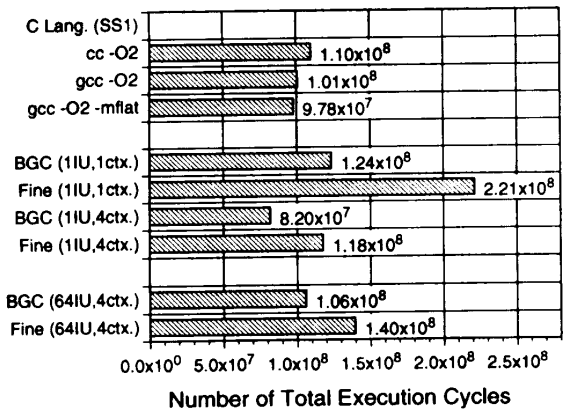


図9 各方式によるQ11の実行効率の比較．総実行サイクル数は，実行サイクル数を単純に台数倍したもので，稼働率低下にともなうアイドルサイクル数を含む．バンバン粒度制御方式（BGC）は，C言語に匹敵する逐次実行効率を示す．

Fig. 9 A comparison of efficiency of the schemes for Q11. The number of total execution cycles includes idle cycles due to low processor utilization. BGC shows excellent efficiency comparable with the C language.

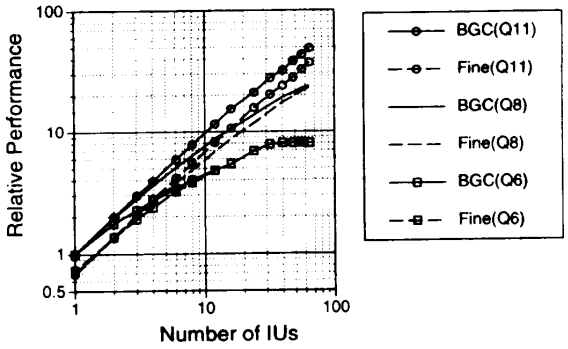


図10 並列処理による性能向上．基準は逐次処理に近いBGCの単体性能．BGCは，並列性が高いときは逐次処理と比べても高い台数効果を，並列性が低いときはFineと同じ性能を示す．

Fig. 10 Speedup achieved by parallel processing against BGC single-IU performance, which is close to the performance of sequential processing. BGC shows almost-linear speedup in Q11, and the same speedup in Q6 as the fine scheme.

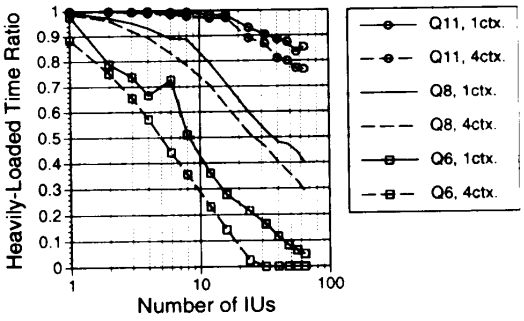


図11 高負荷状態の時間割合の変化．IU台数，コンテクスト数が増えると必要な並列性も高くなり，高負荷状態の割合は下がる．

Fig. 11 The ratio of heavily-loaded time to total execution time. The heavily-loaded time decreases with increasing the number of IUs or contexts.

のコンテクスト数（ctx）は，1または4とした．負荷判断の閾値のオフセットは，図13を除いて0である．すべての測定は，10回測定して平均をとった．なお，ここで示す実行時間は，純粋な計算時間のみで，ガーベジコレクション（GC）を含まない．これは，GCの速度向上がスーパーリニア（GC回数削減とGC1回あたりの時間短縮による）であり，比較の妨げになるからである．

プログラムの実行方式には，バンバン粒度制御方式（BGC）と，従来同様の細粒度方式（Fine）を用意した．ただし，従来方式では実行と逆順にforkを行っていたが，BGCの細粒度モードとフェアに比較するために，実行と同じ順序でforkを行うようにした．また，同じアルゴリズムをC言語で記述し，SPARCstation 1（SS1）で実行して比較した．コンパイラには，cc（SunOS4.1.1）とgcc（version 2.6.3）を使い，**-O2**で最適化した．gccでは，**-mflat**（レジスタウィンドウを使わない）を付けた場合も測定した．

図9に，Q11の総実行サイクル数（実行サイクル数の台数倍）の比較を示す．命令セットの違いがあるため，一概には優劣を判断できないが，UNIREDも基本的にはRISCであることから，BGCでは，C言語に匹敵する逐次実行効率が達成できると考えられる．また，Fineよりも効率が良く，並列実行の効率も高い．

図10に，並列処理による性能向上の様子を示す．逐次処理に近いBGCでの単体性能を基準とした．コンテクスト数は4．並列性が高い場合，BGCは逐次処理と比べても高い台数効果を示す．Fineも高い線形性を示すが，台数が増えてもBGCの性能には追い付かない．並列性が低いときは，BGCはFineと同じ性能を示す．

図11は，BGCで高負荷状態の時間割合が，IU台数でどのように変わるかを示したものである．縦軸は，各IU内での高負荷状態時間の合計を，総実行時間で割った値．IU台数やコンテクスト数が増えるとより多くの並列性が要求され，高負荷状態の割合は低下する．

図12に，強制forkされたゴール数の総ゴール数に対する比率の変化を示す．図11と同様に，台数，コンテクスト数の増加に従って，強制forkされるゴールは増える．Q6, 4ctxでは，32台で，高負荷状態時間，強制forkゴール数ともに，ほぼ0となるが，図10から，ちょうどその台数まで性能向上が見られることが分かる．

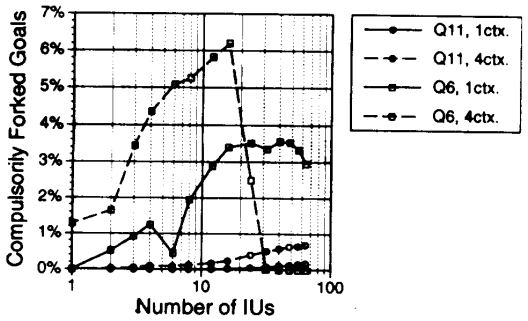


図12 強制 fork ゴール率の変化．台数，コンテクスト数が増えると強制 fork も増加し，台数効果が限界に達すると 0 になる

Fig. 12 The number of compulsory forks. After increasing with the number of IUs, it decreases to zero when the speedup is saturated.

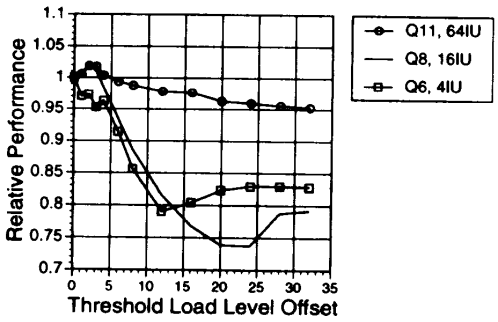


図13 負荷判断閾値のオフセットの効果．小さなオフセットには若干の意味があるが，大きなオフセットは，性能の低下を招く

Fig. 13 Influence of the offset to load threshold level on performance by BGC with 4 ctx. The offset is not so beneficial.

図13 に，負荷判断閾値のオフセットの効果を示す．縦軸は，オフセット 0 のときの性能に対する相対性能である．コンテクスト数は 4．Q11（64 台），Q8（16 台）で，小さなオフセットで若干性能が向上するのは，SUSTAIN での MP からの負荷分散が増え，臨界飽和状態のヒステリシスが大きくなり，強制 fork が減るからである．しかし，大きなオフセットは，必要な並列性を増加させ，高負荷状態時間の減少，局所性の低下により性能を低下させる．Q6（4 台）の性能が向上しないのは，必要な並列性が少し増えただけで，高負荷状態時間が減少し，強制 fork が増えるためである．Q6，Q8 で，大きなオフセットで若干性能が上がるのは，ヒステリシスが大き過ぎて，MP が LOW から HIGH に入らなくなり，SUSTAIN での負荷分散処理がなくなって MP の処理が軽くなるためである．

7. 関連研究

Lazy Task Creation (LTC)[3),7)]は，タスク生成点で fork の準備のみを行い，休止プロセッサが生じたときに，スタックの底側から fork させる手法である．LTC は，幅優先/深さ優先の切り替え，高負荷状態での効率向上と負荷低下時の稼働率向上の両立など，本研究と共通点が多いが，飽和状態を保つことに特化しており，低負荷状態の細粒度実行がない点や，スタックの底から fork させるオーバヘッドを持つ点などが異なる．

StackThreads[15)]は，並列オブジェクト指向言語において，block が起きるまではヒープ上にフレームを確保せず，通常のスタックを使うことで，同一プロセッサ内の非同期メソッド呼び出しを高速化する手法である．本研究とは機構的に類似しているが，StackThreads では同一プロセッサ内処理の高速化を主眼としており，プロセッサ間の負荷分散，低負荷状態での並列性抽出などの視点は入っていない．

命令レベルデータフロー計算機の分野では，並列性の爆発によるメモリ資源の枯渇を回避するために，Throttle Mechanism[9)]，DFM-II の実行制御機構[14)]，遺伝子の概念[16)]，k-bounded loops[2)]などの研究が行われた．特に，9)，14) では，LTC と同様に幅優先/深さ優先を実行時の負荷状態で切り替える方式が検討されており，本研究と関連が深いが，これらはデータフロー計算機に大きく依存している点が，von Neumann 型計算機に基づく本研究と異なる．

近年のデータフロー計算機は，命令よりも粒度の粗い thread 単位で同期をとり，この thread を von Neumann 型プロセッサで逐次実行する Multithreaded 計算機へと変化してきた[5)]．それらの研究における thread 分割は，本研究における低負荷状態での細粒度実行の最適化と類似点を持つが，高負荷状態でも細粒度実行を続けるという点が異なる．10) では，EM4 上の負荷状態に応じた粒度制御の研究が示されているが，負荷低下時の逐次実行の解除が検討されていない．

本研究で利用した Fleng の特徴は，ゴール単位で細粒度実行できること，述語の入口で強制 fork できることなどで，他の細粒度言語にも本方式を応用できるだろう．また，ハードウェア的には，グローバルな共有アドレス空間は，積極的な細粒度実行や動的負荷分散のために不可欠と考えられるが，自動負荷分散や負荷状態検出は，ソフトウェアで代用可能と考えられる．

8. 終わりに

本稿では，細粒度実行における効率の改善と，スタックを用いた強力な逐次化を組み合わせ，実行時の負荷状態に応じて切り替える，バンバン粒度制御方式の提案を行った．PIE64 と Fleng を用いて実装，評価した結果，1）単体プロセッサでは，細粒度並列言語にお

いても，同じアルゴリズムでC言語で書かれたプログラムに匹敵する効率を達成できること，2）並列性が十分に高い場合には，逐次処理と比べても高い台数効果が得られること，3）並列性が低い場合には，細粒度並列処理と同じ限界まで性能向上が得られることを確認した．

今後の課題としては，本方式に対応したコンパイラの開発，それを用いた本格的な応用プログラムによる評価のほか，序論でも述べたように，最適制御理論面からの検討として，線形制御近似モデルの構築と最適性の証明，などがあげられる．

謝辞 日頃からご討論いただく，田中英彦研究室のOB諸氏，学生の皆さんに感謝します．また，本研究の一部は，文部省科研費（課題番号62065002，03555071，03003891）の補助を受けて行われました．

参考文献

1) Araki, T., Hidaka, Y., Nakada, H., Koike, H. and Tanaka, H.: System Integration of the Parallel Inference Engine PIE64, *Proc. Workshop on Parallel Logic Programming, FGCS'94*, pp.64–76 (1994).
2) Culler, D.E.: Managing Parallelism and Resources in Scientific Dataflow Programs, Ph.D. Thesis, Dept. of EECS, MIT (1989).
3) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. Parallel Symbolic Computing: Languages, Systems, and Applications, LNCS*, Vol.748, pp.94–107, Springer-Verlag (1992).
4) Hidaka, Y., Koike, H. and Tanaka, H.: Architecture of Parallel Management Kernel for PIE64, *Future Generations Computer Systems, Special Issue on PARLE'92*, Vol.10, No.1, pp.29–43, Elsevier Science Publishers (1994).
5) Iannucci, R.A., Gao, G.R., Halstead, Jr., R.H. and Smith, B. (Eds.): *Multithreaded Computer Architecture: A Summary of the State of the Art*, Kluwer Academic Publishers (1994).
6) 伊藤正美：自動制御概論（下），昭晃堂 (1985).
7) Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *Proc. ACM Symp. on Lisp and Functional Programming*, pp.185–197 (1990).
8) Nilsson, M. and Tanaka, H.: Massively Parallel Implementation of Flat GHC on the Connection Machine, *Proc. FGCS'88*, pp.1031–1040 (1988).
9) Ruggiero, C.A.: Throttle Mechanisms for the Manchester Dataflow Machine, Ph.D. Thesis, Dept. of CS, University of Manchester, UMCS-87-8-1 (1987).
10) Sakai, S., Okamoto, K., Matsuoka, H., Hirono, H., Kodama, Y. and Sato, M.: Super-Threading: Architectural and Software Mechanisms for Optimizing Parallel Computation, *Proc. Int. Conf. on Supercomputing 93*, pp.251–260 (1993).
11) Shapiro, E. (Ed.): *Concurrent Prolog: Collected Papers*, Vol.1-2, The MIT Press (1987).
12) Shimada, K., Koike, H. and Tanaka, H.: UNIREDII: The High Performance Inference Processor for the Parallel Inference Machine PIE64, *New Generation Computing, Special Issue for FGCS'92*, Vol.11, No.3,4, pp.251–269 (1993).
13) 高橋栄一，小池汎平，田中英彦：並列推論マシンPIE64の相互結合網の作製及び評価，情報処理学会論文誌，Vol.32, No.7, pp.867–876 (1991).
14) Takesue, M.: A Unified Resource Management and Execution Control Mechanism for Data Flow Machines, *Proc. 14th ISCA*, pp.90–97 (1987).
15) Taura, K., Matsuoka, S. and Yonezawa, A.: StackThreads: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs, *Proc. JSPP'94*, pp.25–32 (1994).
16) Toda, K., Uchibori, Y. and Yuba, T.: The Gene Concept and Its Implementation for a Dataflow Schemed Parallel Computer, *Proc. PARLE'89, Vol.1, LNCS*, Vol.365, pp.306–322, Springer-Verlag (1989).

（平成7年9月4日受付）
（平成8年5月10日採録）

日高 康雄（正会員）

昭和38年生．平成元年東京大学工学部精密機械工学科卒業．平成3年同大学院情報工学専攻修士課程修了．平成6年同博士課程単位取得退学．工学博士．昭和59年（株）創夢設立に参画．同年取締役．平成4年日本学術振興会特別研究員．平成6年東京大学工学部電子情報工学科助手．平成8年（株）富士通研究所入社．平成7年度丹羽記念賞受賞．計算機アーキテクチャ，並列処理，CAD等に興味を持つ．最近は特に，高性能計算機の設計，実装に興味がある．電子情報通信学会，精密工学会，IEEE-CS，ACM各会員．

小池　汎平（正会員）

昭和36年生．昭和59年東京大学工学部電子工学科卒業．昭和61年同大学院工学系研究科情報工学専門課程修士課程修了．平成元年同博士課程単位取得退学．同年同大学工学部電気工学科助手，平成3年同講師，平成7年同助教授．平成8年通産省工業技術院電子技術総合研究所入所．この間平成6年より平成8年までマサチューセッツ工科大学コンピュータサイエンスラボラトリ客員研究員．工学博士．並列処理計算機の研究に従事．平成4年度元岡賞受賞．

田中　英彦（正会員）

昭和18年生．昭和40年東京大学工学部電子工学科卒業．昭和45年同大学院博士課程修了．工学博士．同年東京大学工学部講師．昭和46年助教授，昭和53～54年ニューヨーク市立大学客員教授，昭和62年東京大学教授，現在に至る．計算機アーキテクチャ，並列処理，計算モデル，人工知能，自然言語処理，分散処理，CAD等に興味を持っている．「非ノイマンコンピュータ」，「情報通信システム」著，「計算機アーキテクチャ」，「VLSIコンピュータI，II」，「ソフトウエア指向アーキテクチャ」共著，New Generation Computing編集長，情報処理学会，電子情報通信学会，人工知能学会，ソフトウエア科学会，IEEE，ACM，各会員．