1995 年並列処理シンポジウム (JSPP'95) にて発表

PIE64 における複合粒度並列処理を用いた最適粒度制御
– 細粒度並列と粗粒度並列; 二つの異質な世界の分離と融合 –

日高 康雄　小池 汎平　田中 英彦
東京大学 工学部

概要

低負荷時と高負荷時の最適化を陽に区別し, 細粒度と粗粒度の二種類のコードを実行時に切り替える「複合粒度並列処理」を提案する. 本手法を並列推論エンジン PIE64 上で評価した結果, 1) 単体プロセッサでは, C 言語に匹敵する性能が得られ, 2) 並列性が十分に高い場合は, 逐次処理と較べても高い台数効果が得られ, 3) 並列性が低い場合は, 細粒度並列処理による限界と同じ速度向上が得られることを確認した.

Abstract

We propose hybrid-grain parallel processing where heavily-loaded situation and lightly-loaded situation are explicitly distinguished in static optimization, and two-version codes are switched dynamically. An evaluation of the method in PIE64 shows (1) comparable single-processer performance with the C language, (2) near-linear parallel speedup under high concurrency, comparing even with sequential processing, and (3) the same parallel speedup under low concurrency as the maximum speedup by fine-grain parallel processing.

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良く, 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良い [7, 9, 12, 13]. 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上, 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, あらかじめ細粒度 (低負荷時用) と粗粒度 (高負荷時用) の二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, 「複合粒度並列処理」を提案する. これは, 「並列性が十分に高い時と不足している時とでは, 本質的に世界が異なる」という基本的な考えに基づいている.

並列性が十分に高く, 全プロセッサが稼働している時は, 逐次処理に基づく粗粒度並列処理が最善の実行方式である. 処理を fork して並列性を高めても, それを実行するプロセッサはなく, オーバヘッドが増加するだけであり, 逐次処理に近い実行方式の方が効率が良い. 従来の逐次計算機はその特殊な例であり, 常に十分な並列性が確保されていると見なせる.

一方, 並列性が不足している時は, 細粒度並列処理が最善の実行方式となる. 休止中のプロセッサが存在するため, 処理を fork すれば直ちに実行に移され, 並列に処理される. コンパイラにおける最適化では, 本質的な依存関係のある箇所を逐次化しつつ, 並列性の向上を優先させてクリティカルパスを短縮することが重要になる.

本稿で提案する複合粒度並列処理による最適粒度制御方式は, 並列推論エンジン PIE64[2] 上で, 細粒度並列処理用のコミッティッドチョイス型言語 Fleng[11] を用いて実装し, 評価を行なった.

本稿の構成は以下の通りである: 2 節では, 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 節では, 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定性的に論じる. 4 節では, PIE64 と Fleng について説明し, 5 節では PIE64 における最適粒度制御, 6 節では性能評価について述べる. また, 7 節では関連研究について述べる.

Optimal granularity control by hybrid-grain parallel execution in PIE64 – fine grain vs. coarse grain; distinction and fusion of two heterogeneous worlds – by Yasuo HIDAKA, Hanpei KOIKE, and Hidehiko TANAKA (Faculty of Engineering, The University of Tokyo).
E-mail: {hidaka,koike,tanaka}@mtl.t.u-tokyo.ac.jp

表 1: 逐次処理と各種並列処理方式の比較.

	逐次処理	粗粒度並列処理	細粒度並列処理	複合粒度並列処理
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
実行効率	○	○	×	○
並列性抽出	————————	×	○	○
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	コードサイズ大
有効領域	————————	並列性† > PE数 × n	並列性† ≲ PE数 × n	全領域

n · · · レイテンシの隠蔽等のために, PE1 台当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. PE 数の少ない小規模な並列処理では, PE 数に較べて十分に高い場合が多く, 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.

2 粗粒度並列 vs 細粒度並列

粗粒度並列処理は, 逐次処理に部分的な半順序関係を導入したもので, 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が容易でないという問題点がある. 特に, 非定型処理における自動分割は困難で, 現状ではプログラマが同期命令等を挿入して分割を行なっているが, 同期命令の挿入ミスは発見が難しく, 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, 細粒度並列処理の利点は, 全ての計算を半順序関係として表現するところから出発しているため, プログラムに内在する全ての並列性を容易に抽出できること, 同期ミスによるバグがあまり起こらないことである. しかし, 頻繁に行なわれる同期処理等のオーバヘッド大きく, 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.

複合粒度並列処理は, 上記両方式の利点を組み合わせた方式である. すなわち, 細粒度並列言語を出発点として最大限の並列性を抽出する. そして, 十分に高い並列性が得られている時には, 全順序関係による最適化を全面的に採り入れて逐次処理なみの高い実行効率を達成し, 逐次処理と較べても十分に高い台数効果を得ることを目指す. また, 並列性が低い時には, 半順序に基づく細粒度並列処理を行ない, 稼働率を向上させて可能な限りの速度向上を目指す.

なお, 各処理方式と提案方式の比較を表 1に示す.

3 負荷状態により異なる二つの世界

本節では, 要素プロセッサ (PE) 数に較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」で, どのように状況が異なり, その結果, 最適実行形態や最適化方針がどのように異なるかを論じる (表 2).

最も本質的な違いは, 休止中の PE が存在するか全

表 2: 低負荷状態と高負荷状態の比較

低負荷状態	高負荷状態
状況の違い	
並列性が低い時	並列性が高い時
休止中の PE が存在	全 PE が稼働中
fork すれば直ちに実行	fork しても後回し
|wait queue|=0	|wait queue| ≥0
本質的に FIFO 順序	FIFO/LIFO 共に可
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
並列性重視	ローカリティ重視
Critical path の短縮	オーバヘッドの低減
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
(Global scheduling)	(Local scheduling)
投機的計算が有効	Suspend 予測が有効

PE が稼働中かであり, そこから全ての違いが生じる.

低負荷状態においては, 処理を fork すれば休止中の PE によって直ちに並列に実行される. 故に, 実行待ちの wait queue の長さはほぼ常に 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, ローカリティより並列性を重視し, 逐次化は依存関係のある部分に留めるべきである. 但し, リモート実行のコストを考慮しつつ, fork した処理が直ちに並列実行されると仮定して, critical path 長が最短になるような global scheduling によって実行時間を直接短縮すべきである. また, PE 内の local scheduling 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,

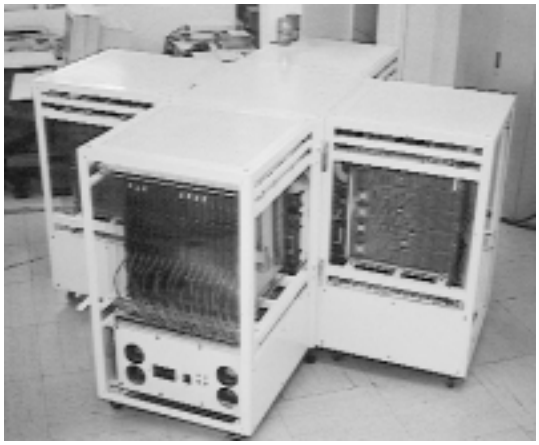


図 1: 並列推論エンジン PIE64. 64 台の推論ユニットと 2 系統の相互結合網を持つ. 自動負荷分散機能を使って, マシン全体の負荷状態判断が容易に可能.

サスペンド覚悟の実行でもあまり問題はない.

一方, 高負荷状態では全 PE が稼働中で, 処理を fork しても wait queue に入って後回しになる. 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 local scheduling 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, ローカリティを重視すべきである. fork を減らして逐次化することにより, ローカリティが向上すると共に, fork や同期のオーバヘッドが削減され, 実行時間は間接的に短縮される. 負荷分散よりも各 PE 内の local scheduling が重要で, サスペンドはできるだけ回避すべきである. また, 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二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, 高負荷状態から負荷が低下した時の切り替え方法である. 粗粒度モードで並列性をスタックに残して逐次実行する場合, 低負荷状態になっても逐次実行が続くと稼働率が低下するため, このスタックから並列性を抽出しなければならない.

4 並列推論エンジン PIE64

本方式は, 並列推論エンジン PIE64[2] 上で, コミッティッドチョイス型言語 Fleng[11] を用いて実装した. PIE64 は, 64 台の推論ユニット (IU: Inference Unit) と呼ばれる要素プロセッサと, 2 系統の相互結合網から構成される MIMD 型並列計算機である (図 1).

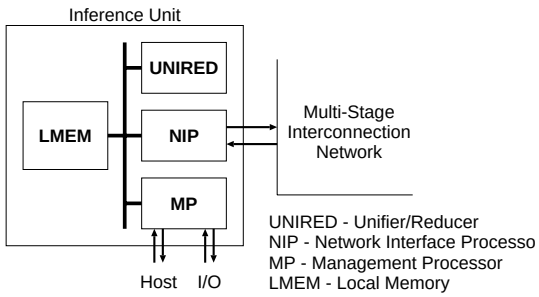


図 2: 推論ユニットの構成. 計算を担う UNIRED は, NUMA 型共有メモリを提供し, サイクル毎のコンテクスト切り替えにより通信レイテンシを隠蔽する.

4.1 PIE64 のアーキテクチャ

相互結合網 [16] は, 負荷最小 IU を自動的に選択して負荷分散を行なう, 自動負荷分散機能を備えている. またこの機能を利用して, 各 IU はマシン全体の負荷状態を容易に知ることも出来る. すなわち, 各 IU が申告した負荷値は相互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良い.

IU[6] は, 計算を担う UNIRED (Unifier/Reducer), 管理を担う MP (Management Processor), 通信・同期を担う NIP (Network Interface Processor) の三種類のプロセッサ等から構成される (図 2).

UNIRED[15] 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, 他の IU に対するメモリ参照命令を 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 Program Counter(PC) をサイクル毎に切り替えるマルチコンテクスト処理により, 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.

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担い, 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 [7] によって構成される. なお, 当初は UNIRED を計算に専念させる予定であったが, 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 細粒度並列処理言語 Fleng

Fleng[11] は細粒度並列処理向きの言語で, コミッティッドチョイス型言語, 並列論理型言語の一種である. 同種の言語には, GHC[18], KL1[19] などがある.

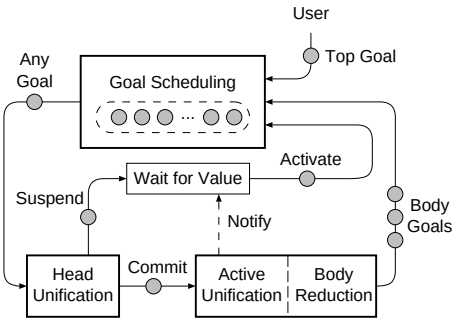


図 3: Fleng の実行モデル.

Fleng における計算処理の単位はゴールと呼ばれ, プログラムは定義節の集合として与えられる. プログラムの実行は, (1) まずキューからゴールを取り出し, (2) そのゴールと左辺 (head 部) が unify 可能な定義節を選択し, (3) その定義節の右辺 (ボディー部) の組み込み述語 (active unify 等) を実行し, (4) ボディー部のゴールに reduce してキューに戻す, という操作の繰り返しである (図 3).

ゴールの実行順序に制約はなく, 全て並列に実行しても構わないが, ヘッドユニファイ時に未定義変数の値が必要であると, そのゴールはサスペンドする. サスペンドしたゴールは, 他のゴールのボディー部において変数が具体化されると, アクティベイトされて再びキューに戻される. なお, 変数は単一代入であり, バックトラックは一切行なわない.

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, 述語毎にまとめてコンパイルされる. この述語は, 逐次型手続き言語における手続きや関数型言語における関数に相当し, 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, コンパイルされた該当述語のエントリポイント (EP) にジャンプする. Head unification 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, ボディーゴールを一つを除いて全て fork し, 最後に, 残る一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 で実行する時), または MP (ローカル IU で実行する時) に投げる. この fork は

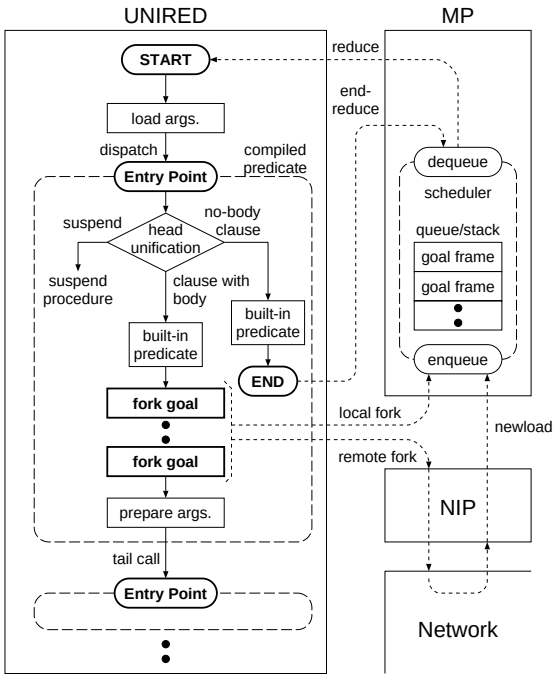


図 4: 細粒度方式における処理の流れ.

かなり軽いとはいえ, 逐次型言語の手続き呼び出しに較べると, 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, 親手続き内の分岐命令で直接呼ぶのに対し, 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, 末尾呼び出しの時は, 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のコストとほぼ同じである. また, head unification 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, サスペンドが生じなければ条件分岐と同程度であり, fork や末尾呼び出し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, 該当コンテクストを停止させる.

5 PIE64 における最適粒度制御の実装

5.1 複合粒度方式の実装

複合粒度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二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,

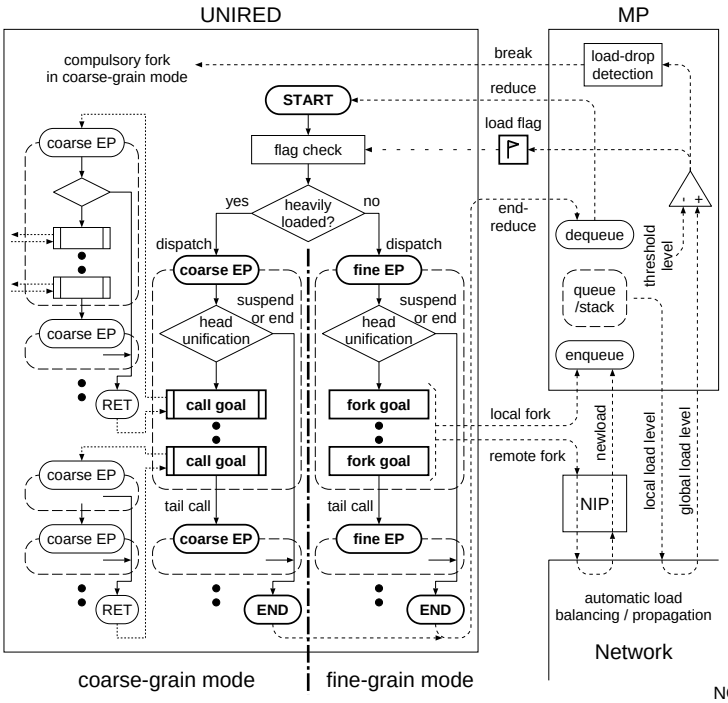


図 5: 複合粒度方式における処理の流れ.

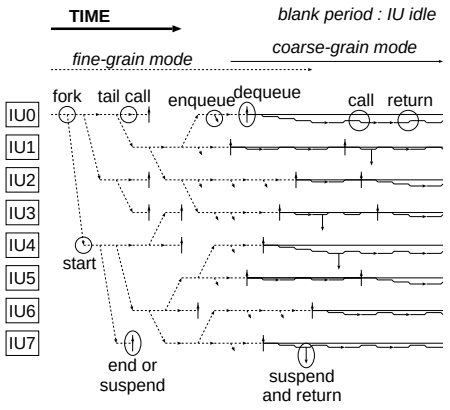


図 6: 複合粒度方式の全体実行イメージ.

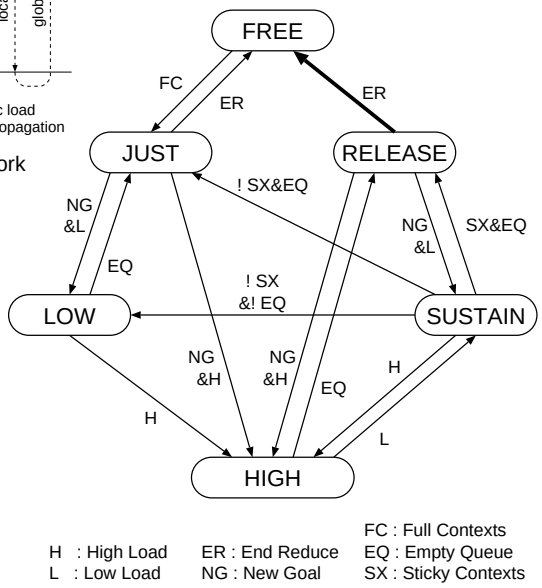


図 7: スケジューラの状態遷移. 太矢印の遷移で粗粒度モードを続けているコンテクスト (sticky context) に break(一種の割り込み) をかけ, 強制 fork させる.

従来同様の fork と末尾呼び出しによる実行を行ない, 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なう. 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 間や同じ UNIRED のコンテクスト間で, 二つのモードが混在する.

負荷状態判定のオーバヘッドを抑えるために, 判定はディスパッチルーチンのみで行ない, 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. 但し, 粗粒度モードでは並列性をスタックに残して長時間の逐次実行を行なうため, 低負荷状態になっても粗粒度モード実行を続けると稼働率が低下する. このため, 粗粒度モードで実行中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 に飛ぶだけで, fork 処理は部分的にも一切行なわない. 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あり, 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 (定義節が確定した時) に確保し, 末尾呼び出しの直前に開放する. 呼び出された側では, ヘッド部で引数をチェックするが, サスペンドしない限りそのコストは小さい.

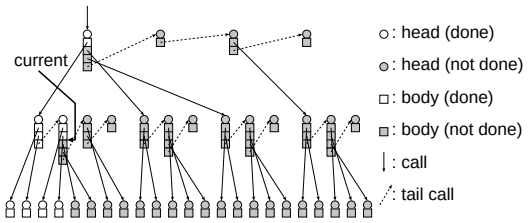
5.2 最適粒度制御の実装

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ない[1], HIGH 状態においてメモリ上の負荷フラグを高負荷状態にする. UNIRED はこの負荷フラグを見

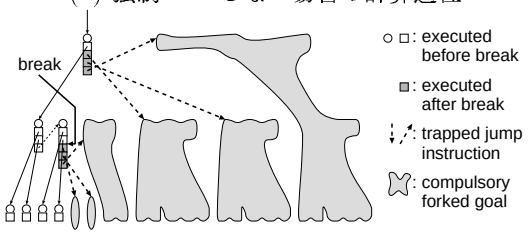
表 3: スケジューラの状態.

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

[1]図 7中の H, L は, 稼働コンテクスト数 + 実行待ちゴール数を自 IU の負荷値として相互結合網に流し, フィードバックされる最小負荷値を使用コンテクスト数と比較した結果である.



(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.

て, 細粒度 / 粗粒度モードを選択する.

HIGH 状態で負荷が低下すると, メモリ上の負荷フラグを低負荷状態に戻し, SUSTAIN, RELEASE 状態に遷移するが, 粗粒度モードを直ちに停止させることはしない. SUSTAIN では, MP は queue 内のゴールを定期的に取り出して負荷分散を行ない, 高負荷状態に戻れば再び HIGH 状態に遷移する.

一方, FREE 状態に戻る時に, 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, スリットチェック付きのジャンプ命令 (break がかかると 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全て実行しつつ, 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, 他にはスタックの一部を fork させる手法もある. その場合, スタックの底側の処理 (一般に計算量が多い) を fork させるべきである [9] が, そうするとスタックの構造が複雑になり, 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 の処理を出来る限り単純化し, 負荷の低下時には, 若干の時間的猶予の後, 全てを fork させて並列性を一気に向上させる方法を採った.

ここで, 本研究の一般性について考察する. 本研究で実装上利用した言語的特徴は, ゴール単位で細粒度並列実行できること, 述語の入口で強制 fork が可能なことなどで, 関数型言語など他の細粒度並列言語にも本手法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の存在は, 積極的な細粒度並列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, ソフトウェアでも代用可能であろう.

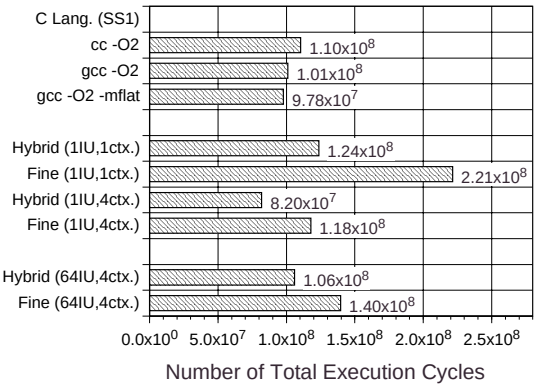


図 9: 各方式による総実行サイクル数の比較. 複合粒度方式は, C 言語に匹敵する逐次実行効率を示し, 従来の細粒度方式よりも効率が良い.

6 性能評価

提案した複合粒度方式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 の時は実行時間が短過ぎるため, N=6 の時は 1000 回, 8 の時は 100 回, 全解探索を繰り返させた. 使用する UNIRED のコンテクスト数 (ctx) は, 1 または 4 とした. 全ての測定は, 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 回当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, 複合粒度方式 (Hybrid) と, 従来同様の細粒度方式 (Fine) を用意した. 但し, 従来方式では実行と逆順に fork を行なっていたが, 複合粒度方式の低負荷モードとフェアに比較するために, 実行と同じ順序で fork を行なうようにした. また, 同じアルゴリズムを C 言語でも記述し, 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, オプション -O2 でコンパイルした. gcc では, オプション -mflat(レジスタウィンドウを用いない) を付けた場合も測定した.

図 9に, 各方式による Q11 の総実行サイクル数 (実

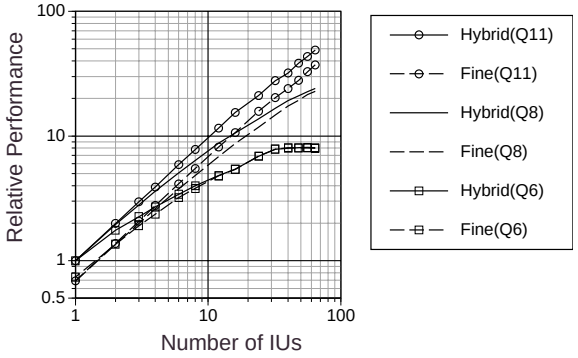


図 10: 並列処理による性能向上. 基準は, ほぼ逐次性能に等しい複合粒度方式での単体性能. 複合粒度方式は, 並列性が高い時は逐次性能と較べても高い台数効果を, 並列性が低い時は細粒度方式と同じ性能を示す.

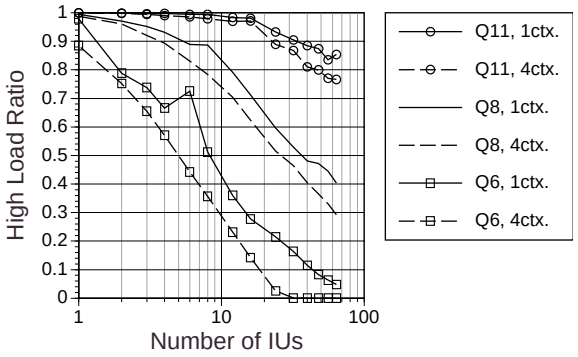


図 11: 高負荷状態の時間割合の変化. IU 台数, コンテクスト数が増えるとより多くの並列性が要求され, 高負荷状態の割合は下がる.

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出来ないにせよ, UNIRED も基本的には RISC であることから, 複合粒度方式では, 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, 従来の細粒度方式よりも効率が良く, 並列実行の効率も高い.

図 10では, 並列処理による性能向上の様子を示す. ほぼ逐次性能と見なせる複合粒度方式での単体性能を基準に用いた. コンテクスト数は 4. 並列性が高い場合, 複合粒度方式は, 逐次処理と較べても高い台数効果を示す. 細粒度方式も高い線形性を示すが, 細粒度処理のオーバヘッドが大きい. 一方, 並列性が低い場合, 複合粒度方式は細粒度方式と同じ性能を示す.

図 11では, 高負荷状態の時間割合が, IU 台数でどのように変化するかを示す. 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. また, 使用するコンテクスト数が増えると, 高負荷状態の割合はさらに低下する.

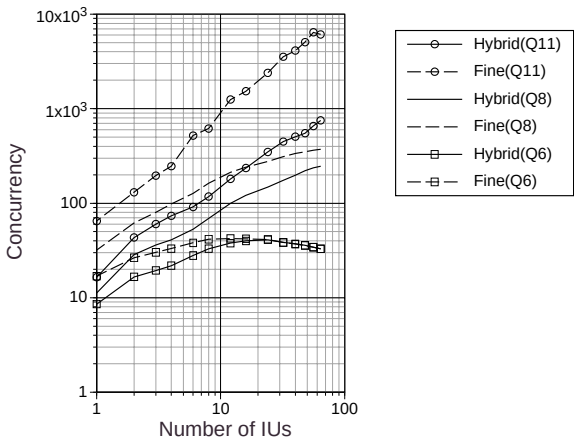


図 12: 実行時に得られた並列性の時間平均. 細粒度方式の実行では, 並列性が高い時 (Q11) に, 過剰に並列性が抽出されている.

図 12は, 実行時に得られた並列性の時間平均値を示している. 並列性は, MP の用いる負荷値, すなわち, 稼働中のコンテクスト数 + 実行待ちゴール数で測定した. 使用したコンテクスト数は 4. 細粒度方式では, 並列性が高い時に過剰に並列性が抽出されている. fork の順序を実行と逆順にすればこれを抑えることができるが, その場合低負荷時の性能が悪化する.

7 関連研究

Lazy Task Creation(LTC)[9, 5, 20] では, タスク生成点では fork の準備のみを行ない, 実際に休止プロセッサが生じた時に, スタックの底側から fork させる手法が提案された. LTC は, マシンが飽和状態になるまで幅優先, その後は深さ優先で実行し, 非飽和状態に戻ると要求駆動的に fork するという点や, 高負荷状態での逐次化による効率向上と負荷低下時の稼働率向上の両立など, 本研究と共通点が非常に多い. しかし, 低負荷状態の細粒度実行が検討されていない点や, スタックの底から fork させるためのオーバヘッドが大きい点が本研究と異なる.

StackThreads[17] では, block が起きない限りヒープ上のフレームを確保せず, 通常の stack で実行することにより, 同一ノード内の非同期手続き呼び出しの高速化をはかった. StackThreads も LTC と同様にスタックを用いた逐次化による効率向上という点が本研究と共通しているが, 低負荷状態での並列性抽出など, 負荷状態に関する検討が行なわれていない.

究極の細粒度実行を目指した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[12], k-bounded loops[3] などの並列性制御の研究が行な

われてきた．特に，[12] 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，本研究と非常に関連が深いが，データフロー計算機に大きく依存している点が，von Neumann 型計算機をベースとする本研究と異なる．

一方，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 Architecture へと変化してきた [1, 8, 14]．また，Threaded Abstract Machine[4] では，データフロー計算機向きの関数型言語 Id を，通常のプロセッサで効率良く実行する手法を示した．こ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っている．

8 終わりに

本稿では，細粒度実行における効率の改善と，スタックを用いたより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，複合粒度実行方式の提案を行なった．PIE64 と Fleng を用いて実装，評価を行なった結果，(1) 単体プロセッサでの実行では，細粒度並列処理言語においても，同じアルゴリズムで C 言語で書かれたプログラムと較べて遜色のない性能が達成できること，(2) 並列性が十分に高い場合には，粗粒度の並列実行によって高い台数効果が得られること，(3) 並列性が低い場合には，単体性能に対する台数効果は低いものの，細粒度の並列実行による限界と同じ速度向上が得られることを確認した．

参考文献

[1] B.S. Ang, Arvind, and D. Chiou: StarT the Next Generation: Integrating global caches and dataflow architecture, CSG Memo 354, Lab. for Computer Science, MIT, 1994.

[2] T. Araki, Y. Hidaka, H. Nakada, H. Koike, and H. Tanaka: System integration of the parallel inference engine PIE64, Workshop on Parallel Logic Programming, FGCS94 pp.64–76, 1994.

[3] D.E. Culler: Managing parallelism and resources in scientific dataflow programs, PhD thesis, Dept. of EECS, MIT, 1989.

[4] D.E. Culler, A. Sah, K.E. Schnauser, T. von Eicken, and J. Wawrzynek: Fine-grain parallelism with minimal hardware support: A Compiler-Controlled Threaded Abstract Machine, ASPLOS-IV, pp.164–175, 1991.

[5] M. Feeley: A message passing implementation of Lazy Task Creation, Parallel Symbolic Computing : Languages, Systems, and Applications, US/Japan Workshop Proceedings, LNCS 748, pp. 94–107, Springer-Verlag, 1992.

[6] 日高, 小池, 田中: 並列推論エンジン PIE64 の推論ユニットのアーキテクチャ, 信学技報 CPSY90-44, pp. 37–42, 1990.

[7] Y. Hidaka, H. Koike, and H. Tanaka: Architecture of Parallel Management Kernel for PIE64, Future Generations Computer Systems, Special Issue on PARLE92, Vol.10,No.1,pp.29–43, 1994.

[8] R.A. Iannucci: Toward a dataflow / von Neumann hybrid architecture, ISCA88, pp.131–140, 1988.

[9] E. Mohr, D.A. Kranz, and R.H. Halstead, Jr: Lazy Task Creation: A technique for increasing the granularity of parallel programs, ACM Symposium on Lisp and Functional Programming, pp.185–197,1990.

[10] S. Murakami, H. Nakada, Y. Hidaka, H. Koike, and H. Tanaka: Load distribution system of PIE64, Workshop on Parallel Logic Programming, FGCS94 pp.77–90, 1994.

[11] M. Nilsson and H. Tanaka: Massively parallel implementation of Flat GHC on the Connection Machine, FGCS88, pp.1031–1040, 1988.

[12] C.A. Ruggiero: Throttle Mechanisms for the Manchester Dataflow Machine, PhD thesis, UMCS-87-8-1, Dept. of Computer Science, Univ. of Manchester, 1987.

[13] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato: Super-threading: Architectural and software mechanisms for optimizing parallel computation, International Conference on Supercomputing 93, pp.251–260, 1993.

[14] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba: An architecture of a dataflow single chip processor, ISCA89, pp.46–53, 1989.

[15] K. Shimada, H. Koike, and H. Tanaka: UNIREDII: The high performance inference processor for the parallel inference machine PIE64, New Generation Computing, Vol. 11, No.3,4, pp.251–269, 1993.

[16] 高橋, 小池, 田中: 並列推論マシン PIE64 の相互結合網の作製及び評価, 情処論, Vol. 32, No. 7, pp.867–876, 1991.

[17] K. Taura, S. Matsuoka, and A. Yonezawa: StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs, JSPP94, pp.25–32, 1994.

[18] K. Ueda: Guarded Horn Clauses, ICOT Technical Report TR-003, ICOT, 1985.

[19] K. Ueda and T. Chikayama: Design of the Kernel Language for the parallel inference machine, The Computer Journal, Vol. 33, No. 6, pp.494–500, 1990.

[20] I. Watson: SLAM - Lazy Task Creation and Dynamic Load Balancing with Active Messages, Draft Internal Report, Dept. of Computer Science, Univ. of Manchester, 1993.