C-012

# 排他的マルチスレッド実行モデルにおける多段なループ処理へのスレッド間パイプライン並列実行方式の適用とその評価

# Evaluation of the Thread Pipelining for the Cascade Loop Processing in the Exclusive Multithread Execution Model

泉 雅昭 † 雨宮 聡史 † 松崎 隆哲 ‡ 雨宮 真人 ‡
Masaaki Izumi Satoshi Amamiya Takanori Matsuzaki Makoto Amamiya

## 1. はじめに

近年まで命令レベルの並列性を利用するプロセッサ技術やコンパイラ技術により性能改善を行ってきた. しかし, 単一プロセスまたは単一スレッド実行における命令レベル並列性の利用には限界があり [3], 性能改善が困難になってきている. 一方, スレッドレベル並列性の利用によって性能改善をする研究が広く行われている. プログラムを複数スレッドへ分割することにより I/O 処理やメモリアクセスの遅延隠蔽を図る. しかし, 複数スレッドの実行は同期処理や複雑なスレッドスケジューリングのオーバヘッドが性能改善を妨げている問題がある.

我々は排他的マルチスレッド実行モデルに基づいた Fuce プロセッサ [2, 4] を提案してきた. 並列処理と親和性の高いデータフロー計算モデルを基盤とした継続概念 [1] を排他的マルチスレッド実行モデルに採用することで, 複数スレッドの並列実行性能を徹底的に追求する. スレッドは排他的に実行するため, スレッドスケジューリングを単純化できる. 排他的マルチスレッド実行モデルにおいてバッファレスなプログラムを実現し, パイプライン並列性を抽出するスレッド間パイプライン並列実行方式 [5] を提案した.

本稿では, 多段なループ処理へのスレッド間パイプライン並列実行方式の適用について議論する.

## 2. Fuce アーキテクチャ

### 2.1 排他的マルチスレッド実行モデル

Fuce アーキテクチャの排他的マルチスレッド実行モデルは, データフロー計算モデルに基づいた継続概念を核とする.

Fuce アーキテクチャは関数とスレッドを中心にしてプログラミングモデルを定義する. スレッドは他からの干渉を受けずに排他的に実行する命令列として定義され, 先行スレッドからデータ依存関係と制御依存関係を解決したという通知を全て受け取ると実行可能になる. この通知を継続と定義する. スレッド間に依存関係が存在しない場合は, 複数スレッドが並列実行できる. 関数は複数のスレッドで構成され, その関数の実行環境 (命令列とデータ領域) として関数インスタンスを持つ.

†九州大学 大学院システム情報科学府, Graduate School of Information Science and Electrical Engineering, Kyushu University
‡九州大学 大学院システム情報科学研究院, Graduate School of Information Science and Electrical Engineering, Kyushu University

#### 2.1.1 スレッド間のデータ授受

継続は依存関係解決の通知であるため, スレッド間のデータ授受はメモリを介して行う. 図 1 にスレッド間のデータ授受の概略を示す. データ領域の先頭アドレスを Base-address で表す. Thread 1, 2, 3, は同一の関数に属したスレッドであり, データ領域を共有する.

最初に, Thread 1 と Thread 2 はデータをメモリに書き込み, Thread 3 に継続する. Thread 3 は最初に実行を開始し Thread 1 と Thread 2 が書き込んだ値をロードする. Fuce アーキテクチャでは, スレッド間のデータ授受にスタックを使用しない.

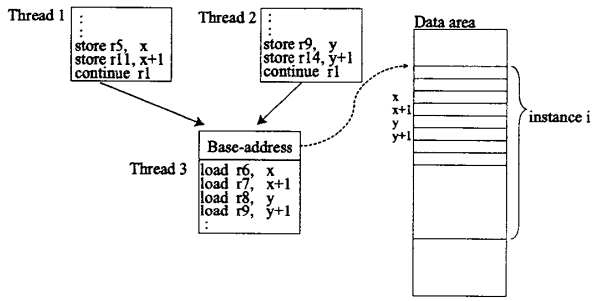


図 1: スレッド間のデータ授受

#### 2.1.2 排他制御

Fuce アーキテクチャは資源を保持するスレッドに対してロック操作を試みる手法を用いて, 排他制御を行う. 二つの先行スレッドが, ある一つの継続スレッドに継続する場合を考える. Fuce プロセッサはスレッド管理情報の一つとしてロック操作のための lock-bit を保持し, それを利用することで排他制御を実現する. 図 2 に排他制御の動作モデル例を示す.

1. 先行スレッドは継続スレッドへロックを試みる.
2. **(a)** ロック成功
   継続スレッドへ継続する.
   **(b)** ロック失敗
   自スレッドを再起動し, 再度ロックを試みる.
3. 処理終了の後に継続スレッドはロックを解除する.

これにより複数の先行スレッドが存在する場合の排他制御を行う.

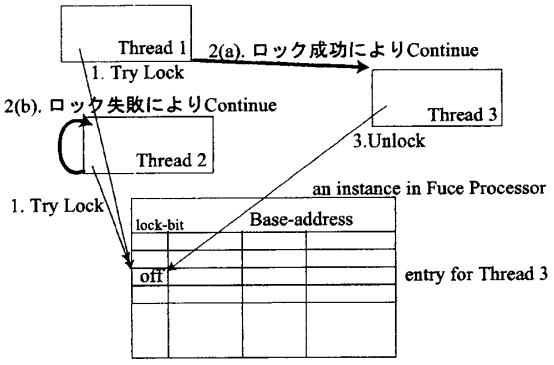


図 2: 複数スレッドでの排他制御モデル

### 2.2 Fuce プロセッサの基本構成

Fuce プロセッサは，スレッドの実行管理を行う Thread Activation Controller (TAC), 複数のスレッド実行ユニット, メモリを持つ. 図 3 は Fuce プロセッサの概要図である.

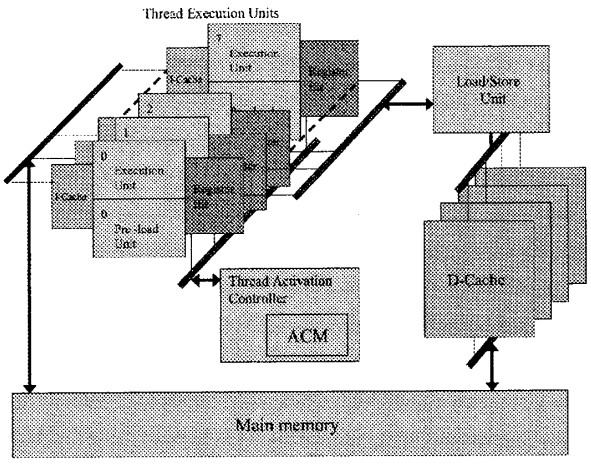


図 3: Fuce プロセッサの概要

**Thread Activation Controller**

Fuce プロセッサは, スレッドの同期管理と起動を制御する機能ユニット TAC を利用して, スレッドレベルの並列実行を実現する. TAC は Activation Control Mmemory (ACM) を用いて, スレッドの同期管理と起動を実現する. ACM はスレッド管理情報と関数インスタンスの情報を保持する. 図 4 に ACM の概要を示す. ACM は複数のページで構成される. 関数インスタンス毎にページを割り当て, 一つのページには関数インスタンスに含まれる Base-address と複数のスレッド管理情報を保持する. スレッド管理情報として fan-in, sync-count, lock-bit, code-entry を持つ. fan-in はそのスレッドが先行スレッドから継続される数である. sync-count の初期値は fan-in であり, そのスレッドが継続されるたびに sync-count をディクリメントする. lock-bit は排他制御に利用し, 初期値は 0 である. 仮に, lock-bit が 1 ならばそのスレッドはロックされている. code-entry はそのスレッドの命令コードの先頭アドレスである.

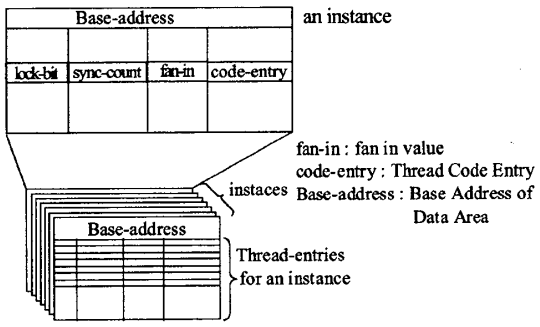


図 4: ACM の概要

**スレッド実行ユニット**

スレッド実行ユニットはスレッドを排他的に実行するユニットである. Fuce プロセッサはスレッド実行ユニットを 8 個持ち, 8 個のスレッドを同時に実行できる. 複数スレッドの効率的な実行のために, スレッド実行ユニットは演算ユニットとプリロードユニットの二つで構成する. 演算ユニットは単純な RISC コアであり, プリロードユニットは演算ユニットのサブセットとしてデータのロード命令のみサポートする.

## 3. スレッド間パイプライン並列実行方式

ループ処理の例を用いて, スレッド間パイプライン並列実行方式 (以下, パイプライン方式と呼ぶ) の概要を述べる. 図 5 はスレッド内における通常のループ処理を示す. Task A, B, C はそれぞれある処理単位であり, Task B は Task A に, Task C は Task B に依存する. ループ処理の例ではループ変数 i を初期値から, max まで更新しながら処理する.

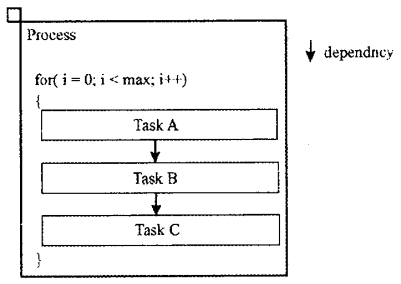


図 5: 通常のループ処理

次に, パイプライン方式によるループ処理の概要を述べる. 図 6 にパイプライン方式を示す. 図 6 の Thread A, B, C は, 図 5 に対応する処理を行うスレッドである. Task 間の依存関係と同様にスレッド間の依存関係が存在する. 最初に, 先行スレッドである Thread A はループ変数 i=t に関する処理を完了すると, 継続スレッドである Thread B にデータを転送し, 継続する. ただちに Thread A はループ変数 i=t+1 の処理を開始する. 同時に, 継続された Thread B はループ変数 i=t に関する処理を開始する. この時, Thread A と Thread B は並列に実行できる. このように, パイ

プライン方式は, データの流れに沿って記述することで, 先行スレッドと継続スレッドが並列実行できるため容易にスレッドの並列性を抽出できる.

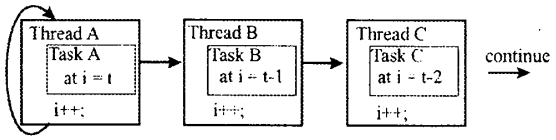


図 6: スレッド間パイプライン並列実行方式の概要

### 3.1 スレッド間パイプライン並列実行方式の制御

パイプライン方式による並列性の抽出を概念的に示した. しかし, 継続スレッドが処理を開始する前に, 先行スレッドがデータを転送し上書きしてしまう可能性がある. この問題を排他制御によって回避する.

まず, 先行スレッドは継続スレッドのロックを試みる. 継続スレッドのロックに成功した時はデータを転送し, 継続する. 次に, 自スレッドのロックを解除して次のデータが転送されるのを待つ. もし, 継続スレッドのロックに失敗した時は, 自スレッドを一旦終了する. そして, 自スレッドを最初から実行することで, 継続スレッドに再度ロックを試みる.

### 3.2 多段なループ処理への適用

複数のループ処理の間に依存関係が存在しない場合はそれぞれのループ処理を並列に実行できる. 一方, 依存関係が存在する場合はそれぞれのループ処理を並列に実行できないため, パイプライン並列性を抽出するパイプライン方式を試みる. ループ処理内のあるデータが次のループ処理のデータに依存するときに, パイプライン方式を適用する.

図 7 に多段なループ処理へのパイプライン方式の適用を示す. ループ処理の 1 回のイタレーションをスレッドとし, ループ処理間の依存関係を継続としてパイプライン方式を構成する. 始めのスレッドのみ自スレッドへ継続することで, 全体としてループ処理と同等の処理内容を行う.

## 4. 評価

パイプライン方式によるパイプライン並列性の抽出効果について評価した. 評価は VHDL にて記述した Fuce プロセッサを ModelSim 上で動作させて実験した. 今回利用した Fuce プロセッサのシミュレーション環境は, TAC のアクセスレイテンシを 1 サイクルとし, スレッド実行ユニットの本数は可変で, メモリアクセスレイテンシは可変である. ただし, データキャッシュは実装されていない. アセンブラで全てのプログラムを作成した.

### 4.1 ベンチマークプログラム

多段なループ処理で構成されるクイックソートプログラムを用いて, パイプライン方式によるパイプライン並列性の抽出効果を測定する. 評価は一般的によく知られているアルゴリズムをマルチスレッド実行する従来方式 (Standard) とパイプライン方式 (Thread Pipelining) で行った.

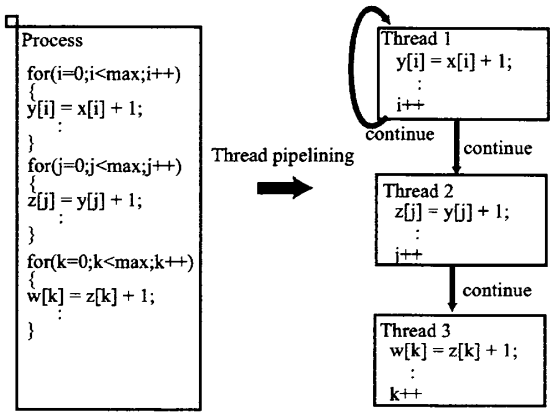


図 7: 多段なループ処理へのパイプライン方式の適用

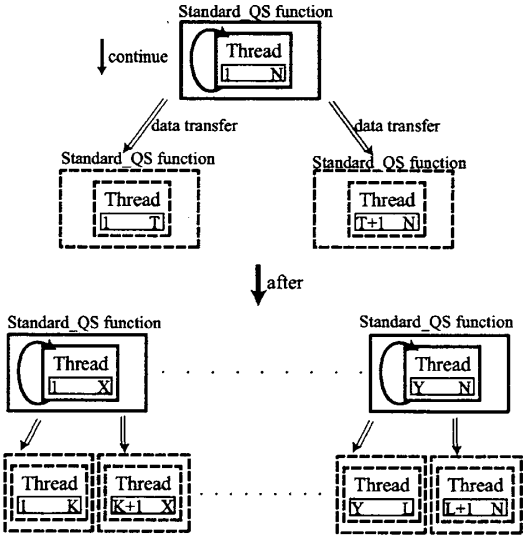


図 8: 従来方式による実行の様子

従来方式はクイックソートアルゴリズムに存在するデータ並列性を利用するプログラムである. 図 8 に従来方式の実行の様子を示す. 最初に従来方式は 1 つのクイックソート関数しか実行しない. その関数の実行が終了すると, 新たに 2 つの関数を生成し処理を進める. さらに時間が経つと互いに依存関係が存在しない関数が増加し, 同時に複数の関数を実行できる.

一方, パイプライン方式はデータ並列性に加え, パイプライン並列性を抽出するプログラムである. 図 9 にパイプライン方式の実行の様子を示す. 最初にパイプライン方式はデータを処理すると, そのデータをピボットとする新たな関数を生成する. 元々の関数と生成された関数はパイプライン方式によって実行し, パイプライン並列性を抽出しながら処理を進める. また, 生成された関数間はデータ並列性を利用して, 同時にその関数を実行できる.

### 4.2 性能評価

スレッド実行ユニット本数を 1〜8 まで変化した台数効果を調べることで, パイプライン方式によるパイプライン並列性の抽出効果について評価した.

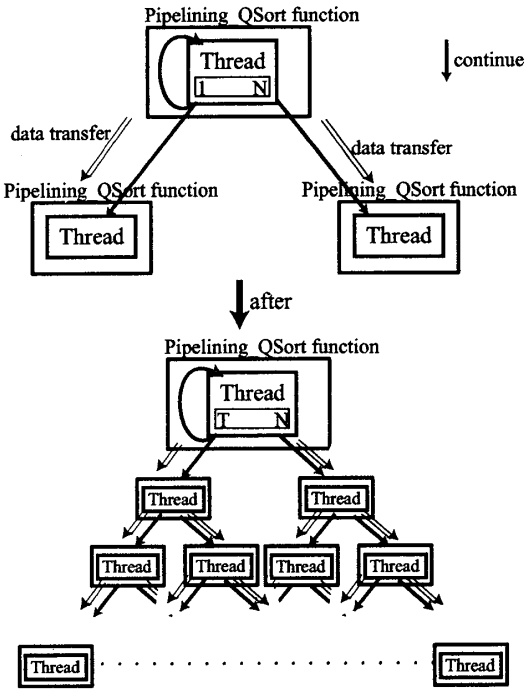


図 9: パイプライン方式による実行の様子

図 10 に各プログラムのユニット本数 1 を基準とした性能向上比を示す．メモリアクセスレイテンシは 20, 60, 100 サイクルで実験した．

ユニットが 8 本の時に従来方式は 3.89 倍の性能向上を示し，その時パイプライン方式は 5.89 倍の性能向上を示した．これは，パイプライン方式がデータ並列性に加えてパイプライン並列性を抽出したためである．

図 11 にパイプライン方式の性能向上を示す．メモリアクセスレイテンシ 100 サイクルの従来方式の実行時間を基準とした各プログラムの実行性能向上を示し，従来方式と比較したパイプライン方式の実行性能向上を調べた．

パイプライン方式は，同一メモリアクセスレイテンシの従来方式と比べてユニットが 8 本の時に 1.86 倍の性能向上を示した．パイプライン方式がユニットが 1 本の時に性能向上しないのは，従来方式に比べてスレッドの粒度が細かいため，よりスレッド切り替え回数が増加したためである．

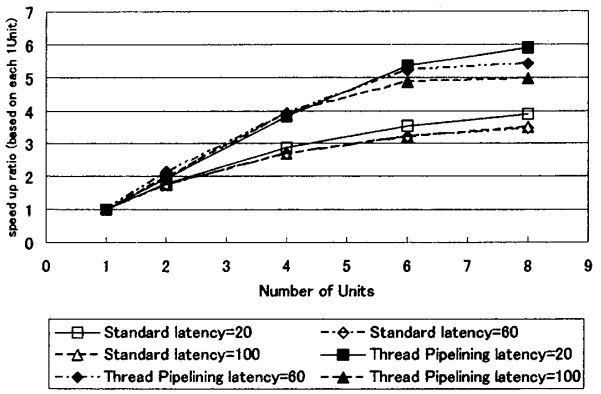


図 10: パイプライン並列性の抽出効果

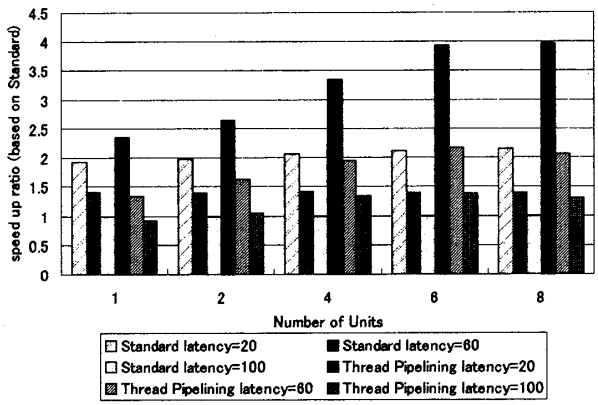


図 11: パイプライン方式の実行性能向上

## 5. おわりに

本稿では，スレッド間パイプライン並列実行方式を多段なループ処理への適用について議論した．多段なループ処理を持つクイックソートプログラムへスレッド間パイプライン並列実行方式を適用することで，本手法の評価を行った．評価から，スレッド間パイプライン並列実行方式によって高い並列性を抽出できることを確認した．

今後は，様々なベンチマークプログラムに適用することで，本手法の詳細な評価を行っていきたい．

**謝辞**

本研究は科研費基盤研究 (A)(2)「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号:15200002) と九州大学における 21 世紀 COE プログラム「システム情報科学で社会基盤システム形成」の若手研究者のための研究助成の一環として行ったものである．

## 参考文献

[1] Amamiya, M., "A New Parallel Graph Reduction Model and its Machine Architecture", Data Flow Computing: Theory and Practice, Ablex Publishing Corporation, pp.445- 467, 1991.

[2] Amamiya, M., Taniguchi, H. and Matsuzaki, T., "An Architecture of Fusing Communication and Execution for Global Distributed Processing", Parallel Processing Letters, Vol.11, No.1, pp.7-24, 2001.

[3] D.W. Wall, "Limits of Instruction-Level Parallelism," Proc. Fourth I nt'l Conf. Architectural Support for Programming Languages and Operating Systems, pp. 176-188, 8-11 Apr. 1991.

[4] 雨宮 真人 他, 通信・放送機構 (TAO) 研究成果報告書, 「情報通信網の基盤技術に関する研究」, 平成 15 年 3 月.

[5] 泉雅昭, 雨宮聡史, 松崎隆哲, 雨宮真人, "継続モデルに基づくスレッドプログラミング手法の提案," 情報処理学会研究報告, 2004-ARC-159, pp.79-84, 2004.