再帰的データ構造を対象としたループの並列投機実行方式

本河 俊樹[†], 山村 周史[†], 布目 淳[†], 平田 博章[†], 新實 治男[‡], 柴山 潔[†]

[†] 京都工芸繊維大学 工芸学部 電子情報工学科
〒 606-8585 京都市左京区松ヶ崎御所街道町
[‡] 京都産業大学 工学部 情報通信工学科
〒 603-8555 京都市北区上賀茂本山

本稿では，非数値計算分野のプログラムにしばしば現われる線形リストや２分木などの再帰的データ構造を処理するループに対して，これを投機的に並列実行する方式を提案する．本方式の特徴は，ループという制御構造に基づいて並列化を行うのではなく，より本質的に，データ構造そのものに着目して並列化を行う点にある．このため，よりセマンティックレベルの高い観点からデータの並列性そのものを反映した並列化が可能となる．

Parallel Execution of Loop Iterations by Speculative Traversal on Recursive Data Structures

Toshiki Motokawa[†], Shuji Yamamura[†], Atsushi Nunome[†], Hiroaki Hirata[†], Haruo Niimi[‡], Kiyoshi Shibayama[†]

[†] Dept. of Electronics and Information Science, Kyoto Institute of Technology
Matsugasaki, Sakyo-ku, Kyoto 606-8585 JAPAN
[‡] Dept. of Information and Communication Sciences, Kyoto Sangyo University
Motoyama, Kamigamo, Kita-ku, Kyoto, 603-8555 JAPAN

In this paper, we present a novel technique to parallelize a loop which operates on recursive data structures, such as linked list and binary tree. In our scheme, parallel threads are not exactly derived from control structure of loop, but they are created by speculatively traversing on data structure. Consequently, we can parallelize loops efficiently in the fashion which reflects intrinsic data parallelism.

1 はじめに

プログラムに内在する並列性の抽出において，従来よりループが有力な対象として取り上げられ，ループの並列化に関する多くの研究が行われてきた．もちろん，その成否はループという制御構造そのものによるのではなく，ループの各イタレーションが処理する対象のデータに並列性が存在するか否かによる．科学技術計算の分野ではデータ構造として配列が多用され，このため，ループの並列化の研究のほとんどは配列に関連するものである．

これに対して，非数値計算分野のプログラムでは，配列も利用されるものの，線形リストや２分木などの再帰的データ構造[1]を扱うことも多く，そのようなデータ構造の各要素をループで処理するプログラム記述がしばしば現われる．ランダムアクセス可能でかつ添字によってデータ要素の識別が可能な配列とは異なり，ポインタを用いて実現するそれらのデータ構造はランダムアクセスが不可能で，並列化のための静的な解析も困難である．しかし，非数

[1]配列を用いて線形リストや２分木などの抽象データ構造を実現することは可能であるが，ここでは，Ｃ言語のプログラムで一般に記述されるように，アドレスをポインタとして使用する場合を対象とする．

値計算分野のプログラムの高速化のためにはこの種のループを並列化することは非常に重要であり，本稿はその手法を提案するものである．

文献 [1] には，線形リストを処理するループを投機的に並列実行する試みがマルチスレッドプロセッサアーキテクチャの可能性の１つとして示されているが，方式上完全でなく，課題が残されていた．一方，最近の Hydra プロジェクトの研究 [2] では，配列のみならず線形リストを処理するループの並列化も視野に含めている．

本稿では，紙面の都合上，実装に関する詳細な議論は省略し，再帰的データ構造を扱うループの並列化手法とそれを実現する際に必要となるアーキテクチャ上の機能に絞って説明する．プロセッサ全体のアーキテクチャは SMT(Simultaneous Multithreading)[3] や文献 [1] で述べられたマルチスレッドプロセッサをベースに，これを逐次プログラムの並列実行が可能なように改良する方向で議論を進める．しかし，本稿で提案するループの投機的並列実行方式は必ずしもこの種のアーキテクチャに限定されるものではない．また，データに関する投機実行[2]を行うため，レジスタやメモリアクセスにおいてデータの依存関係をチェックする機構や依存関係が犯されたときの回復機構について記述すべきであるが，既に提案されている方式 [2, 4, 5, 6] を流用または応用可能であるため，そのような機構を設けていることを前提として議論を進める．

2 ループの並列実行方式

線形リストや２分木などのデータ構造を扱う場合，例えば，複数のリストを１つにマージしたり，また，１つのリストを複数に分割するなど，リストの構造自体に大きな変更をもたらす場合もあるが，これらの場合を除くと，そのほとんどは構造自体の変更は僅かである．検索のみならリストの構造に変化は生じない．また，要素の挿入や削除なども構造上は部分的な変更であり，実際にポインタのつなぎ替えを行う前に，挿入・削除位置の検索を行うことも珍しくない．そこで，本方式では，データ構造に変更が加えられないものと仮定してループの各イタレーションを動的に展開し，そのそれぞれをスレッドとして投機的に実行する．

本方式の特徴の１つは，ループの展開が，正確にはループという制御構造上の区切りに対応せずに行われる点にある．すなわち，スレッドの fork 点は

[2]ここでは，データの依存関係の有無についての予測（常に依存関係が無いと予測する場合も含む）を基にした投機実行を想定している．

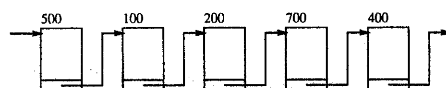

図 1: 線形リストのデータ構造

```
p = header;
while( p ) {............ C
    A
    p = p -> next;  I
    B
}
```

図 2: コード例（線形リストの走査）

ループ本体の開始点ではなく，データ構造の走査に用いるポインタの更新が行われる時点である．これにより，プログラム記述した結果として現われる制御構造に制約されることなく，より本質的なデータの並列性を抽出・活用することが可能となる．

2.1 線形リスト走査の例

まず，簡単な例として，連結リストを用いて実現された線形リストを走査する場合の並列化について述べる．

線形リストのデータ構造の例を図 1に，これを処理する典型的なコード例（C 言語）を図 2に，また，これを並列化したときの実行の様子を図 3にそれぞれ示す．図 1中の数字は各データ要素の先頭番地を表し，図 3においても図 2の文 Iにおいて変数p がどのような値に更新されるかを付記している．なお，図 3の I^nにおいて，n はループ開始時点から数えて論理的に何回目の I の実行かを表している．

まず，当初はスレッド 0が基幹スレッド[3]としてプログラムを逐次実行しているものと仮定する．スレッド 0が図 2のループの実行を開始するとき，まず，終了条件判定の C とループ本体の A の部分を実行し，ポインタ値を更新する I へと到達する．このとき，I を必要回数実行することによって新た

[3]投機実行状態でない正規のスレッドをこう呼ぶことにする．システム中に基幹スレッドは必ず１つ存在する．

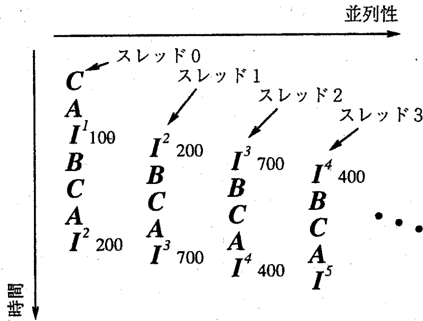

図 3: 並列実行の様子（線形リストの走査）

なスレッドを生成する．図3の例では，スレッド0はそのまま線形リストの2番目の要素の処理を行い[4]，スレッド1，2，3が線形リストのそれぞれ3，4，5番目の要素の処理を投機的に開始する．

本方式の特徴の1つは，図2のループに対して $CAIB$ または $AIBC$ を各スレッドで実行するのではなく，$IBCAI$ を実行の単位とする点にある．つまり，ループの制御に基づいて並列化を行うのではなく，ポインタのたどりを先行的に実行することによって並列化を行う．ループ本体を並列実行の単位とする場合には，手作業またはコンパイラによって文 I をループ本体の冒頭部分に移動しなければ，並列化しても実際の並列処理効果は得られない．また，そのような移動を行った上でかつデータ投機を許したとしても，変数p の値はスレッド間の通信によって受け渡されることになるため，ほとんどの場合，待ちが生じるかあるいはデータ依存関係を犯したという理由で再実行を余儀なくされる．本方式は，ループのイタレーションごとにほぼ確実に値の更新が行われる性質をもつp のような帰納変数に類する変数の更新は，スレッドの fork 時に処理してしまうことを基本的な考え方としている．それ以外のデータについては，データ依存関係が犯された場合は再実行することにより，データ投機を可能とする．

スレッド0が I^2 に達したとき，スレッド1の I^2 の実行結果をチェックし，不整合が生じていればスレッド1以降を無効化するとともに，正しい値を用いて線形リストの3番目の要素の処理を開始する．I^2 の実行結果に不整合が生じていなければ，スレッド1を投機スレッドから基幹スレッドへと昇格させ，スレッド0は終了する．基幹スレッドがループから抜けるときには，もちろん，投機スレッドをすべて無効化しなければならない．同時にアクティブとなるスレッドの数はスレッドスロット（要素プロセッサ）数に依存し，もちろん，スレッドスロットが空けば新たなスレッドを割り付ける．

[4]通常は Bの部分は空か，またはp とは無関係な処理である．

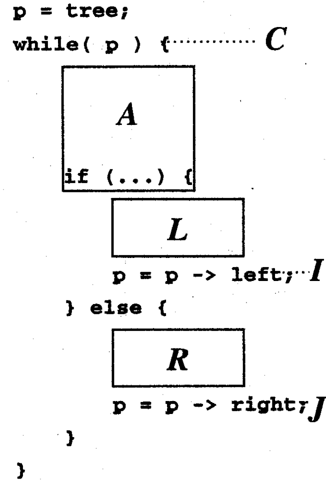

図 4: コード例（2分木探索）

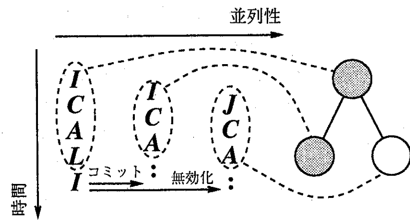

図 5: 並列実行の様子とデータ構造との対応（2分木探索）

2.2 2分木探索の例

2分木探索の場合の典型的なコード例を図4に，これを並列化したときの実行の様子を図5にそれぞれ示す．

線形リストの場合と同様に並列処理の単位をポインタ変数の更新を始終点として設定するが，この場合は I と J の2つが存在するため，合計4種類[5]の処理単位を処理するスレッドが生成され得る．

[5]$ICALI$，$ICARJ$，$JCALI$，$JCARJ$．

図5に示すように，ループのあるイタレーションにおいて I に制御が到達したとすると，その次のイタレーションにおいて I または J のいずれかに到達するものと仮定して，それ以降の処理を2つの投機スレッドを生成して実行する．従って，データ構造の観点からは，2分木のあるノードについての処理と並列にその子ノードに対する処理も行われる．親ノードの処理を終了するときに左右のどちらの子ノードをたどるかが確定するので，生成された一方の投機スレッドは無効化される．一般に，2分木探索の応用例では ALI, ARJ の部分が逐次性の高い処理となる傾向があり，基本ブロック単位の投機実行や単純なループイタレーションの並列実行では実質的に高速化を実現することは難しい．これに対して，本方式では，図4中の if 文に影響されず，早期にポインタ変数の更新を行うことができる点に注目されたい．このため，データのもつ並列性を直接的に抽出・活用することができる．

さて，上記の説明では，プログラム中におけるポインタ変数の更新場所があらかじめわかっているものと仮定していた．しかし，実際にはプログラム実行の過程でこの情報を収集する．つまり，図4のループの実行を開始する時点では，プロセッサは I や J についての情報を持っていない．例えば，最初のイタレーションにおいて制御が I に到達したとすると，この時点でプロセッサは初めて I の命令アドレス等の情報を得る．それ以降，制御が J に到達するまでの間は，あたかも線形リスト構造であるかのごとく左ノードを処理する投機スレッドのみを生成することになる．従って，図5のように処理を行うのは，I と J の両方の情報を得た後である．

2.3 その他のデータ構造について

2.1や**2.2**では典型的なデータ構造とその処理例を用いて本方式の説明を行ったが，実際の応用プログラムでは多種多様なデータ構造が使用される．それらにどのように対応すべきか，ここでは次の2点について簡単にふれておく．

ノードのもつポインタの数　データ要素のノードがもつポインタの数に応じて，ポインタ変数更新位置を記憶するための場所をプロセッサ内に用意しておかなければならない．しかし，本方式では，ノードの定義におけるポインタの数ではなく，ループ中にポインタ変数を更新する箇所がいくつ現われるかが実際上の問題となる．例えば，双方向リストの場合は2つのポインタを有するが，実際に検索を行うループでは一方のポインタのみである．ノードの静的な定義において多くのポインタを持っていても，その処理まで含めて考えると線形リストとみなせる場合も少なくない．応用によっては3分木や4分木などを利用する場合もあろうが[6]，この場合には2/3や3/4の確率で投機スレッドの実行結果を破棄しなければならない．利用される頻度から考えて，ポインタ変数更新位置の記憶数は2が妥当と判断する．

複合データ構造　実際の応用プログラムでは線形リストや2分木が単独で用いられるとは限らず，より複雑なデータ構造が利用されることも少なくない．しかし，そのような複雑なデータ構造であっても，通常は，線形リストなどの典型的なデータ構造を組み合わせて実現することが多い．このような場合，データ構造全体に対する処理は多重ループで記述されることになり，その個々のループは典型的なデータ構造の処理に帰着する．従って，それぞれのループで本方式の適用を検討することができるが，実際には，(i) ポインタ変数の更新がどのレベルのループのものかを判別しなければならない，(ii) どのレベルのループで並列化するべきかの指針を得るのが困難，などの点から，最内ループに対して並列化を行うものとする．

3 アーキテクチャによる支援機能

3.1 管理テーブル

本方式を実現するのにアーキテクチャにおいて用意すべき管理テーブルは SPT(Start Point Table), TCT(Thread Control Table), STQ(Speculative Thread Queue) の3種類である．

SPT は投機スレッドの開始アドレスを保持するためのテーブルである．**2.3**での検討により，エントリ数は2とする．

TCT は実行状態のスレッドを管理するためのテーブルである．スレッドには一意に定まる0以外の識別子 (THID) を与え，これをインデックスとして TCT のエントリを参照する．TCT のエントリは以下のフィールドで構成される．

- 履歴情報：祖先のスレッドの THID
- 開始アドレス：スレッドの開始アドレス
- 再実行開始アドレス：データ依存関係を犯していることが検出された際に，スレッドを再実行するときの開始アドレス
- 再実行コンテクスト：スレッドを再実行する場合の汎用レジスタの値など[7]

[6]ある種の応用では B 木のように多数の子ノードへのポインタをもつデータ構造を利用するものもあるが，この場合のポインタは通常は配列として定義され，従って次項で述べる複合データ構造とみなせる．

[7]すべてのレジスタの値を保持するのかあるいは一部のみを

```
p = &tree;
while( *p ) {
    .....
    if( ... )
        p = &((*p)->left);
    else
        p = &((*p)->right);
}
```

図 6: コード例（ポインタのポインタを用いた2分木探索）

- 生成コンテクスト：子の投機スレッドを生成するときに初期値とする汎用レジスタの値など
- 子の投機スレッドの THID（最大 2 個）

STQ は未実行の投機スレッドを登録しておくためのテーブルで，以下のフィールドで構成される．

- 履歴情報：祖先のスレッドの THID
- 開始アドレス：スレッドの開始アドレス

TCT や STQ に含まれる履歴情報は，祖先のスレッドの THID を順に右詰めで並べたものである．4 代前までの履歴を保持するものとして，自分の THID が 6，親の THID が 2，そのまた親の THID が 3，そのまた親（基幹スレッド）の THID が 1 とすると，履歴情報は 0-0-3-2 となる．基幹スレッドには THID は与えられているが 0 として表現する．新たに子の投機スレッドを生成した場合，その子スレッドから見た履歴情報は 0-3-2-6 となる．ここで THID が 3 の投機スレッドが基幹スレッドに昇格した場合は，TCT と STQ のすべての履歴情報において 3 の部分を 0 に置き換える．また，THID が 3 の投機スレッドを無効化する場合には，履歴情報に 3 を含むエントリを消去することで，その子孫のスレッドも一斉に無効化することが可能となる．

3.2 追加命令

図2や図4の I や J は，一般的な RISC プロセッサでは以下のような 1 個のロード命令にコンパイルされる（r_i は i 番の汎用レジスタ，$offset$ は定数とする）．

$$\text{Load} \quad r_i \leftarrow \text{Mem}(r_i + offset)$$

保持するのか，また，TCT 自体にレジスタの値を埋め込むのかあるいはレジスタセットへのポインタとするのか，などの実装の詳細についてはここでは議論せず，アーキテクチャ上の論理的な仕様についてのみ述べることにする．

しかし，ポインタをたどる場合以外の目的でもこのような命令は生成されるし，また，図 6 のように，ポインタをたどるのに複数の命令を要する場合もある．従って，このような命令領域をスレッド制御領域 (TCS; Thread Control Section) と呼ぶことにし，専用命令で囲むことにする．

結局，本方式を実現するのに命令セットに追加すべき命令は次の 3 つである．

- SLI(Speculative Loop Initialize) 命令：**3.1**の各テーブルを初期化するとともに，実行中の投機スレッドをすべて無効化する．ループの入口および出口に配置する．
- TCSIN(TCS In) 命令：TCS の入口を示す．自 TSCIN 命令のアドレスを SPT 中で検索し，無ければ SPT に登録する．
- TCSOUT(TCS Out) 命令：TCS の出口を示す．

TCS 実行時の動作は，基幹スレッドか投機スレッドかで異なる．まず，基幹スレッドの場合，TCSIN 命令実行において，その時点で TCT を参照して子スレッドが存在しない場合[8]は STQ をすべて初期化する．TCS 内の命令を実行するとき，その結果を TCT 内の生成コンテクストにもコピーし，TCSOUT 命令の実行において SPT に記憶されている命令アドレスを用いて子スレッドを STQ に生成・追加する．基幹スレッドはそのまま実行を続行する．

TCSIN 命令実行時に子スレッドが存在する場合には，まず，TCT 内の子スレッドのエントリを参照して，その開始アドレスと自 TCSIN 命令のアドレスが一致することを確認する．TCS 内の命令を実行するときにはその結果を子スレッドの再実行コンテクストと比較する．TCS 内の命令が同じレジスタに複数回書き込みを行う可能性もあるので，その過程で不一致が検出されてもこれを放置する．TCSOUT 命令実行時に不一致が検出されたままになっていなければその子スレッドを基幹スレッドに昇格させ，自らは終了する．なお，不一致が検出されたままの場合には，その子スレッドを無効化し，その結果，子スレッドが存在しない場合はそのまま実行を続行する．

STQ に未実行のスレッドが存在し，かつ空きのスレッドスロットが存在する場合には，プロセッサはそのスレッドに新たに THID を付与し，TCT エントリの設定を行う．まず，STQ 内の履歴情報および開始アドレスを TCT にコピーし，親スレッド[9]の子スレッドとして新たに付与した THID を登

[8]ループに入った最初の TCS の実行，全ての投機スレッドが無効化された後の実行，あるいは子スレッドがまだ実行されずに STQ 内に存在する場合のいずれかである．

[9]STQ 内の履歴情報から特定可能．

録する．その後，開始アドレスから実行を開始するが，いきなりTCSに入る．親のスレッドの生成コンテクストを用いてTCS中の命令を実行し，その結果は自分の再実行コンテクストにもコピーする．TCSOUT命令の実行において，TCSOUT命令の次の命令アドレスを再実行開始アドレスに登録し，STQ内に子スレッドを生成してTCSを抜ける．

投機スレッドが2回目のTCSIN命令を実行しようとしたとき，実行は中断される．これは，まだデータ依存関係に問題がないことを確認していないからである．つまり，基幹スレッドに昇格するまでは，再実行に関する責任を持たなければならない．基幹スレッドになった時点でTCSに入ることが可能となるが，この場合の動作は前述のとおりである．

SLI命令についても，同様に，投機スレッドは基幹スレッドとなるまで実行を待たされる．

4 コンパイラおよびOSによるサポート機能

本方式の実現に当たってコンパイラが行うべき処理は，ループの入口と出口にSLI命令を，また，TCSを認識してTCSIN/TCSOUT命令ではさむことである．SLI命令をループの入口にも配置する理由は，これによって自動的に最内ループが暗黙のうちに動的に検出され，そのループが並列に実行されるようにするためである．従って，コンパイラはループのネスト構造をチェックすることなく，それぞれのループのレベルで**3.2**の命令の挿入を行えばよい．

また，本方式は，ループの入口で並列化のための処理を行うのではないため，例えば割り込み復帰後でも並列実行を開始することができる．すなわち，並列実行中に割り込みが発生した場合，投機スレッドを無効化して基幹スレッドのコンテクストのみを退避する．割り込み復帰時に基幹スレッドの実行を再開すると，基幹スレッドがTCSを実行する時点から並列実行が開始される．このとき，OSが本方式に関連して行わなければならない処理は，割り込み復帰時にSLI命令を実行したのと同様の状態で基幹スレッドの実行を再開することのみである．

5 おわりに

本稿では，非数値計算分野のプログラムの高速化を目的として，線形リストや2分木などの再帰的データ構造を処理するループの投機的並列実行方式を提案した．ここで示した方式は，我々が現在開発中の次世代マイクロプロセッサアーキテクチャの主要技術の1つであり，再帰的データ構造のみならず，配列を用いたループに対しても適用可能である．現在は，本方式の評価および実装の詳細について検討中であるが，2分木のような再帰的データ構造を扱う場合には，ループだけでなく関数の再帰呼び出しを用いる場合も少なくないので，この点についての高速化の検討も行っている．

謝辞

本研究の一部は，文部省科学研究費補助金(09558031, 10480062, 11480069, 10780188, 1187-8052) および並列・分散処理研究推進機構 (PDC) の補助による．

参考文献

[1] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa: "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads." *Proc. of the 19th Annual Intl. Symp. on Computer Architecture*, pp. 136-145, (1992).

[2] L. Hammond, M. Willey and K. Olukotun: "Data Speculation Support for a Chip Multiprocessor." *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 58-69, (1998).

[3] D.M. Tullsen, S.J. Eggers and H.M. Levy: "Simultaneous Multithreading: Maximizing On-Chip Parallelism." *Proc. of the 22th Annual Intl. Symp. on Computer Architecture*, pp. 392-403, (1995).

[4] G.S. Sohi, S.E. Breach and T.N. Vijaykumar: "Multiscalar Processors." *Proc. of the 22th Annual Intl. Symp. on Computer Architecture*, pp. 414-425, (1995).

[5] A. Moshovos, S.E. Breach, T.N. Vijaykumar and G.S. Sohi: "Dynamic Speculation and Synchronization of Data Dependences." *Proc. of the 24th Annual Intl. Symp. on Computer Architecture*, pp. 181-193, (1997).

[6] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. Smith: "Trace Processors." *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, pp. 138-148, (1997).