修 士 論 文

Parallel TRAM を基にした 超並列 TRAM の実装と評価

指導教官　二木厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

平田 寛道

1998 年 2 月 13 日

要旨

本研究では，Parallel TRAM で用いられている並列項書換えの実装に関する理論を基に，TRAM を分散メモリ型の超並列計算機上での並列書換えを可能とした超並列 TRAM の設計と，その実装・評価を行った．

項書換えシステムは，等式で記述された論理の世界を自然な形で計算の世界に結びつける事ができる計算モデルである．その計算機上への実装という面では，その相性は非常に良く，ナイーブな実装は比較的用意に行う事ができる．しかし，そのような性質とは対照的に，一般にその実行効率は悪い．

項書換えシステムの書換え効率を改善するための研究は数多くなされているが，そのなかで TRAM は項書換えシステムの実行効率に重点を置いて実装された抽象機械で，弁別ネットに代表される多くの要素技術が盛り込まれている．また，代数仕様言語”CafeOBJ”の実行エンジンとして採用されている．

項書換えシステムには潜在的に並列性が内在し，それを引き出しマルチプロセッサや超並列計算機に実装して，並列書換えを行えば，逐次書換えに比べ大幅な効率向上が期待できる．Parallel TRAM は，TRAM を並列書換えが可能なように，逐次簡約の指定しかできなかった E-戦略を拡張し，並列書換えの指定を可能とした並列 E-戦略や，並列書換えのための抽象機械命令の追加などを行い，4 プロセスを使用し TRAM の２倍程度の性能向上を達成している．

そこで本研究では，計算の実行モデルを明確に表現する事が可能な (項書換えの) 抽象機械を，分散メモリ型の超並列計算機上で並列書換えが可能となるよう拡張し，その設計を行ってゆく．また，これを超並列計算機 Cray T3E システム上に実装し，その評価を行う．

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 本研究の目的

項書換えシステム [7] は，代数仕様の直接実行など等式論理を基礎としたさまざまな分野に応用可能な計算モデルである．この項書換えシステムには潜在的に並列性が内在し，それを引き出しマルチプロセッサや超並列計算機に実装して並列書換えを行えば，逐次書換えに比べ大幅な効率向上が期待できる．

そこで本研究では，計算の実行モデルを明確に表現する事が可能な (項書換えの) 抽象機械を，分散メモリ型の超並列計算機上で並列書換えが可能となるよう拡張し，その設計を行ってゆく．また，これを超並列計算機上に実装し，その評価を行う．

本研究で拡張する抽象機械は，簡約化戦略に E-戦略を採用した TRAM[8] とし，また並列化に関する理論を，この TRAM を共有メモリ型のマルチプロセッサ上で並列書換えが行えるよう拡張した，Parallel TRAM[9][10] から得る事にする．

1.2 本研究の背景・特徴

項書換えシステムは，等式で記述された論理の世界を自然な形で計算の世界に結びつける事ができる計算モデルであり，等式論理の定理証明，代数仕様記述や関数プログラムの直接実行等，さまざまな分野に導入されている．また，その実際の計算機上への実装という面から見ても，その相性は非常に良く，ナイーブな実装は比較的用意に行う事ができる．しかし，そのような性質とは対照的に，一般にその実行効率は悪い．特に計算原理と

なる書換え一つ一つに対し，数多くの書換え規則の中から入力項と一致するパターンを見つけ出す作業が必要となるため非常に効率が悪く，またこの部分の効率は項書換えシステム全体の効率にも大きく影響する．

本研究で拡張の対象としている TRAM は，本学 言語設計学講座の緒方助手により設計，実装された項書換えシステムで，代数仕様言語”CafeOBJ” の実行エンジンとして採用されている．TRAM は項書換えシステムの実行効率に重点を置いて実装された抽象機械で，弁別ネットに代表される多くの要素技術が盛り込まれている．また，並列書換えの理論の土台としている Parallel TRAM は，この TRAM を並列書換えが可能なように，逐次簡約の指定しかできなかった E-戦略を拡張し，並列書換えの指定を可能とした並列 E-戦略や，並列書換えのための抽象機械命令の追加などを行なっている．

本研究で設計・実装を行う”超並列 TRAM”は，TRAM に盛り込まれた多くの要素技術を受け継ぎ，また Parallel TRAM での TRAM の並列化技術を元に，分散メモリ型の超並列計算機上で並列書換えを可能とした並列項書換えシステムである．その設計を，メッセージパシングを用いて使用計算機環境に依存しない形で行うことで，将来，より利用されるであろう分散計算機環境上での並列項書換えにも対応可能であると考えられる．

1.3 本論文の構成

本論文の構成は以下のようになっている．第 2章では，本研究の基盤となる項書換えシステムについての解説，およびいくつかの定義を行った後，その関連研究についての解説を行う．第 3章では，今回並列拡張の対象とした TRAM について，TRAM が採用している E-戦略と，TRAM の基本的な動作について説明する．第 4章では，TRAM を分散メモリ型計算機に並列拡張するにあたり，並列拡張に関する基盤とした Parallel TRAM について，並列拡張のための並列 E- 戦略と，TRAM の並列拡張についての概念を説明する．そして第 5章で，分散メモリ型の超並列計算機上で並列書換えを行う超並列 TRAM の設計を行う．まず，分散メモリ型の計算機で効率的なデータのやりとりを行うための基本概念を述べ，それに基づいて超並列化のための基本構造を述べる．続いて，分散メモリ型計算機で他プロセスのメモリを参照するために必要な，参照テーブルとして「**INFO**」という領域を戦略リスト中に追加する．続いて，Parallel TRAM の理論をもとに，並列書換えのための抽象命令を定義して行く．そして最後に，プロセスを管理するための機構について説明する．第 6章では，設計した超並列 TRAM の実装を行い，その評価を行う．第

7章を結論，および今後の課題の章として，また本実装に用いた超並列プログラミング用ツールについてと，第6章で用いたベンチマークプログラムを付録としている．

第 2 章

基礎概念

2.1 項書換えシステム

項書換えシステム (項書換え系)[7] は項の書換えを計算の基本とした計算モデルである．与えられた項を書換え規則に基づいて書換える事で，それ以上書換えられない項を元の項に対する計算結果として得る事ができる．項書換えシステムは項の集合と，項を書換えるための書換え規則の集合の対で定義される．項書換えシステムは，等式論理の定理証明，代数仕様記述や関数プログラムの直接実行などへ応用されている．以下で，項書換えシステムの基本的な事項に関しての定義を行う．

階数 (アリティ) 階数とは演算子の引数の個数である．

項 項とは，次の条件で定義される記号の並びである．

1. 定数記号，および変数記号は項である．
2. $t_1, t_2, \cdots, t_n$ が項で，fが階数を n とする関数記号なら，$f(t_1, t_2, \cdots, t_n)$ も項である．

定数 定数は，階数が 0 である関数記号とする．

部分項 部分項は，t が変数記号か定数記号の場合はそれ自身であり，t が $f(t_1, t_2, \cdots, t_n)$ の形をしている時は $t_1, t_2, \cdots, t_n$，およびそれらの部分項，そして t 自身が部分項である．

書換え規則 書換え規則 $s \to t$(ただし，s,t は項) は以下の条件を満たすものである．

1. s は変数記号ではない．
2. t に出現している変数記号は s にも現れなければならない．

この s を通常**左辺**と，また t を**右辺**と呼ぶ．また，書換え規則の左辺に同じ変数が高々１回しか現れない時，その書換え規則は**左線形**であるという．

変数置換 変数置換とは変数から定数項への写像である．変数置換σを項 t に適用した項 σt は，項 t に出現する変数でありかつ変数置換σ上で値を持つ変数全てを，その値に置き換えた項である．

項の照合 (パタンマッチ) 変数を持つ項 t に，ある適当な変数置換σを適用した結果が t' と合致する時，t は t'に照合 (マッチ) するという．

書換え 定数項 t に対し，書換え規則 $s \to u$ の左辺 s とパタンマッチを行い，合致した場合に定数項 t とパターンとを等しくする変数置換σを右辺 u に適用した結果σu を得る事を書換えという．

リデックス 項書換えシステム Rの書換え規則の左辺に対し，照合するような項を Rのリデックスという．

正規形 Rにおいて，それ以上書換えられない項，つまり部分項として Rのリデックスを含まない項を Rの**正規形**という．

簡約化戦略 (書換え戦略) 項の書換えを行う際に，どのような順序で書換えを行うリデックスを選択するか，そのその順序を決定する手続きを**簡約化戦略 (書換え戦略)** と呼ぶ．代表的な簡約化戦略として，最も左で最も内側に出現するリデックスを選択する最左最内戦略，最も左で最も外側に出現するリデックスを選択する最左最外戦略などがある．

項書換えシステムの基本的性質として，停止性，合流性がある．

停止性 項書換えシステム Rにおいて，無限の書換えが存在しないならば，Rは停止性を満たすという．このような時，Rではどのような順序で書換えを行っても必ず正規形を求める事ができる．

合流性 合流性とは，与えられた項から異なる書換えの列が存在しても必ず同じ項に到達するという性質である．

2.2 関連研究

項書換えシステム (項書換え系) は，様々な分野への応用が可能であるという性格上，理論・実装両面にわたり盛んに研究が行われている分野の一つである．特に，項書換えを並列に動作させ，その実行効率をあげる研究は近年盛んに行われている．

Gouguen らは，これまでの von Neumann の計算モデルに代わる新しい計算モデルとして，Concurrent Term Rewriting[13] を提唱し，これを RRM(Rewrite Rule Machine) の計算モデルとして用いている．Concurrent Term Rewrinting とは，項書換えシステムには本質的に並列性が内在している点に着目し，そのような項を並列に書換えるというもので，これを計算モデルとして用いる事で，von Newmann bottleneck が解消できるとされる．RRM はその名からも分かるように，項書換えシステムの直接実行を行う計算機である．RRM は非常に多くのプロセッサで構成されており，特別な技術を用いずに通信コストを抑えるため，プロセスの構成をプロセス数で 4 つの粒度に分け，粗粒度の要素は細粒度の要素を複数合わせる構造となっている．また，そのレベルによって書換えのモードを MIMD モードと SIMD モードで使い分けている．これにより，非常に効率の良いアプリケーションの記述・実行が可能であるとしている．

また，項書換えシステムをあらゆるレベルで並列に行わせるような Parallel Rewriting の実装についての研究が [12]，Kirchner らによって行われている．書換え規則はその構造が木構造，あるいは DAG(Directed Acyclic Graphs) 構造となるが，そのあらゆる深度で同時に書換えを行える部分を探しだし，その書換えを行わせるものである．そのため，ボトムアップでパターンマッチをおこない，マッチしてさらに書換え時に他のレベルに影響を与えないものを捜し出すようにしている．そして，書換えではこれらの項を全て同時に書換えを行わせている．なお，この書換えプロセスでは，項は DAG として表現されている．

以上，本研究と関連する研究をいくつか紹介した．なお，本研究と最も深く関係し，今回の超並列化の対象となる TRAM と，その理論を応用する Parallel TRAM については，次章以降で記述してゆく．

第 3 章

項書換え抽象機械：TRAM

TRAM(Term Rewriting Abstruct Machine)[8] は項書換えを対象とした抽象機械で，項書換えシステムの実行効率に重点をおいて設計，実装されている．そのために，E-戦略 (3.1節) の使用による書換え順序の制御，弁別ネット (3.2.1節) を用いたパターンマッチの高速化，戦略リスト (3.2.3節) を用いたリデックス検索の高速化などを行っている．

3.1 E-戦略 (Evaluation Strategy)

E-戦略は，演算子ごとに書換えの順番を指定する事ができる戦略である．書換え順序の指定は数列を用いる．この数列の各要素は，

0: 全体項簡約

n: n 番目の引数項簡約 ($0 < n \leq$ 引数の個数)

となる．この E-戦略を用いることの利点は，次のような状況で起こる．
次のような書換え規則があるとする．

```
if( TRUE, X, Y ) -> X.
if( FALSE, X, Y ) -> Y.
```

これは，書換え戦略によっては効率の悪い書換えとなり得る．例えば最内戦略で書換えを行ったとすると，引数項を 3 つとも書換えた後に全体項`if` の書換えを行う．もしここで第 1 引数の書換えの結果が`TRUE` だとすると，`Y` の書換えは無駄になってしまう．ここで，

if の戦略を (1 0) と指定する事で，第 1 引数の書換えを行った後，全体項if の書換えを行うので，無駄な書換えを生じさせないようにできるのである．

3.2 TRAM の概要

TRAM の構成は図 3.1のようになる．

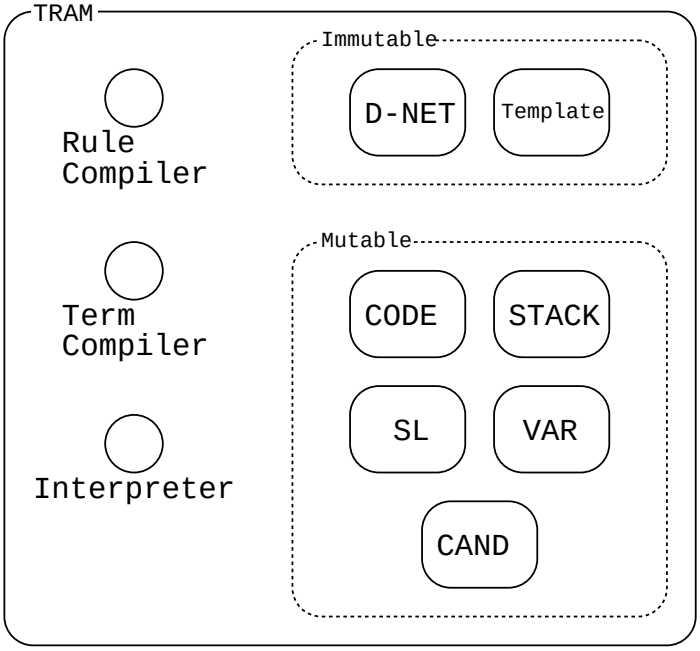

図 3.1: TRAM の構成

図 3.1から分かるように，TRAM は 3 つの処理ユニット (書換え規則のコンパイラ (*Rule-Compiler*)，入力項のコンパイラ (*TermCompiler*)，抽象命令のインタプリタ (*Interpreter*)) と，5 つの Mutable な領域 (コード (*CODE*)，戦略リスト (*SL*)，スタック (*STACK*)，変数 (*VAR*)，書換え候補リスト (*CAND*))，2 つの Immutable な領域 (弁別ネット (*D-NET*)，右辺のテンプレート (*Termplate*)) から構成されている．これらの領域について説明してゆく．

3.2.1 弁別ネット

弁別ネットとは，最外項シンボルをキーとして分岐したパタンマッチ用の木構造の事である．パタンマッチ時にこれを用いる事で，マッチする規則を効率良く検索する事が可能となる．弁別ネットは書換え規則の集合が与えられた時，書換え規則のコンパイラで生成され，D-NET 領域に格納される．

3.2.2 マッチングプログラム

マッチングプログラムは，適用可能な書換え規則の検索と，変数束縛を弁別ネットを用いて実行する抽象命令の事である．マッチングプログラムは構造的に項と等価なものであり，TRAM に入力された項は入力項のコンパイラでコンパイルされ，マッチングプログラムの形で CODE 領域に格納される．また，マッチングプログラムは項を表現しているのみならず，これを抽象命令のインタプリタで実行する事で，規則の検索と変数束縛を行える．また，マッチングプログラムは項と等価である事から，書換えが行われるに従い自身の構造を動的に変化させて行く．このことから，TRAM は「自己改変的」なシステムであるといえる．

3.2.3 戦略リスト

戦略リストは，E-戦略で指定された書換えの順序を制御するために用いられるリスト構造で，マッチングプログラムのラベル列となっており，入力項のコンパイラで，マッチングプログラムと E-戦略から構築され，SL 領域に格納される．抽象命令のインタプリタは，この戦略リストの順番に従ってマッチングプログラムを実行し，書換えを行ってゆく．

3.2.4 右辺の雛型

3.2.2節で述べたように，TRAM は項をマッチングプログラムの形で格納している．そのため，書換えが行われ入力項の形が変わると，それに伴いマッチングプログラムの構造も変化させなければならない．書換えとは書換え規則の左辺を右辺で置換える事であるので，つまりマッチングプログラムの左辺に当たる構造を右辺を表す構造で置換える事になる．従って置き換える元となる，右辺の構造の雛型が必要になる．また，同様に戦略リストの再構築も行わなければならないため，戦略リストの雛型も必要になる．これらは，

書換え規則の集合が与えられた時，書換え規則のコンパイラで生成され，Template 領域に格納される．

3.3 抽象命令インタプリタ

TRAM の書換え処理は，次に示す抽象命令列をインタプリタで実行する事で進められる．ここではその抽象命令の動作を説明する．なお，命令列中の番号は説明文の番号と一致している．

```
  1:        init
  2:  Loop: next
            jump L
  3:        go_ahead
  4:        select
  5:        rewrite
  6:        jump Loop
----------------------
  7: BINGO: bingo
```

1. `init` 命令で，TRAM の大域変数やレジスタを初期化する．
2. `next` 命令で，戦略リストの先頭からマッチングプログラムのアドレスを一つ取り出し，そのマッチングプログラムの実行に処理を移す．これが`jump L` 命令にあたる．また，取り出したアドレスが `BINGO` の場合には，`bingo` 命令に処理を移す．
3. 2でマッチングプログラムに実行が移された後，適用可能な規則の右辺を探し出し `go_ahead` 命令へ処理が戻って来る．適用可能な規則を発見した場合，この`go_ahead` 命令でバックトラックを起こし，他の適用可能な規則を全て探し出す．
4. `select` 命令は，適用可能な規則から一つを選択する．
5. `rewrite` 命令は，4で選択した規則で 2のマッチングプログラムを置換え，戦略リストを再構成する．
6. `jump Loop` 命令で，Loop に処理を移す．
7. `Bingo` 命令が実行されると，その時点でのマッチングプログラムが書換え結果として返される．

3.4 処理の流れ

最後に，TRAM の処理の流れについて説明する．TRAM の処理は次のような 3 つのフェーズに分かれている．

1. 書換え規則のコンパイル
   入力された書換え規則をコンパイルし，その左辺から弁別ネットを，また右辺から右辺の雛型を構築する．

2. 入力項のコンパイル
   入力項をコンパイルし，マッチングプログラムと戦略リストを構築する．

3. 書換え
   戦略リストの先頭から順にマッチングプログラムを取り出し，それを実行してゆく．Bingo 命令が実行された時，その時点のマッチングプログラムを入力項の書換え結果として出力する．その動作は 3.3節で記述した通りである．

第 4 章

並列項書換え抽象機械：Parallel TRAM

Parallel TRAM[9][10] は TRAM をマルチプロセッサ上で実行できるように拡張したもので，E-戦略をユーザが並列性を明示できるように拡張した，並列 E-戦略 (4.2節) を用いている．

4.1 項書換えシステムの並列性

項書換えシステムには潜在的に並列性が内在している．例えば，項

$$add(fib(s(s(0))), fib(s(0))) \tag{4.1}$$

は，図4.1のような木構造となる．ここで，add の第1引数 $fib(s(s(0)))$ と第2引数 $fib(s(0))$ はそれぞれ独立した項であり，お互い書換えに影響を及ぼす事はない．したがって，add は第 1 引数と第 2 引数を同時に書換えを行う事ができる．

4.2 並列 E-戦略

並列 E-戦略は，E-戦略に並列書換えの指定を追加したもので， アリティ n の演算子 f に対して，表 4.1のように定義されている．

$< ParallelElements >$ が並列書換えのために追加した定義で，その要素を並列に書換えを行わせることを表す．また，$< SerialElements >$ は，通常の E-戦略と同様に，その要素を左から順に書換えを行う．

4.3 Parallel TRAM の概要

Parallel TRAM はマルチプロセッサを対象としており，その構成は図 4.2のようになる．

それぞれのプロセッサには，抽象命令のインタプリタと，書換えの際動的に内容が変化する 4 つの領域 (コード，スタック，戦略リスト，変数バインディング) を複製し，抽象機械のレベルで仮想的に割り付ける (この 1 ブロックを「プロセスユニット」とする)．また，内容が動的には変化しない領域 (弁別ネット，右辺の雛型) は，グローバル領域に格納される．また，各ユニットの動作状態を保持するテーブルとして「プロセス状態」というテーブルをグローバル領域には位置している．

また，メインとなるプロセスユニットを 1 つ決め，このユニットに書換え規則のコンパイラ，入力項のコンパイラを配置し，書換えの前処理をこのユニットで逐次に行っている．

4.3.1 戦略リスト

Parallel TRAM の戦略リストは，並列 E-戦略で指定された並列書換えを反映させるために，TRAM の戦略リストに対し次のような拡張が行われている．

1. 引数項書換えは，その区切りを識別できるようにブロック化されている．
2. 並列指定された引数項書換えは，その最後の引数項書換えを除いて全て先頭に「**FORK**」が付加される．
3. 並列書換えの終りには,「**JOIN**」が付加される．

FORK，**JOIN** は，並列書換えを実現するために新たに付け加えられた抽象命令であり，並列書換えの指定時にはこれらの命令に処理を移せるよう，戦略リストの適切な位置にこれらの命令へのラベルが挿入される．そして，これらのラベルの挿入は上記 2，3の作業を機械的に行うことで可能となる．また，戦略リストの雛型も同様の作業で作成できる．

戦略リストの構造

FORK，**JOIN** などの命令は，それを実行するためにいくつかの情報を必要とする．これらの情報を戦略リストの空き領域に格納し，それを効率良く管理している．それらの情報を含めた **FORK**，**JOIN** の戦略リストの構造と，情報の意味は，次のようになる．

FORK : < LabelFORK , BlockEND , JoinAddress >
JOIN : < LabelJOIN , NumOfFORK , — >
EXIT : < LabelEXIT , JoinAddress , ParentUnit >

LabelFORK FORK 命令へのラベル.

BlockEND 引数項書換えの区切りを表し，戦略リスト中のある要素のアドレスを指す. FORK 命令はidle 状態のプロセスユニットに，LabelFORK の次の戦略リストから，この BlockEND までの戦略リストの該当する書換えを割り当てる.

JoinAddress FORK した書換えが戻るべき JOIN のアドレス.

LabelJOIN JOIN 命令へのラベル.

NumOfFORK FORK 命令で作られた子プロセス数．JOIN は，この値が 0 になるまで同期を取って待機する.

LabelEXIT EXIT 命令へのラベル.

JoinAddress 割り付けられた書換えの終了時に戻るべき JOIN のアドレス.

ParentUnit 書換えを割り付けた親ユニット.

4.3.2 抽象命令

Parallel TRAM では，並列書換えを実現するために，TRAM にいくつかの抽象命令を追加している．それらのうち，**FORK**，**JOIN**，**EXIT**，**SLEEP** の各命令について簡単にその動作を説明する.

FORK 他ユニットに引数項の書換えを割り当て，実際に並列書換えを行わせる.

JOIN FORK した書換えが全て終了するまで待機する.

EXIT FORK によって割り当てられた書換えが終了することを，親ユニットに伝える.

SLEEP この命令はサブプロセス上で書き返すべき処理がなくなった場合に実行される. この命令を実行すると，そのユニットは自らの状態を idle 状態にし，他の書換えが割り当てられるのを待つ.

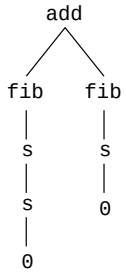

図 4.1: $add(fib(s(s(0))), fib(s(0)))$ の解析木

$$
\begin{aligned}
< ParallelE-strategyList > &::= \text{`('}\ \text{`)'} \mid \text{`('}\ \{(SerialElements)\}^{*}\ \text{`0'}\ \text{`)'} \\
< SerialElements > &::= \text{`0'} \mid < ArgNum > \mid < ParallelElements > \\
< ArgNum > &::= \text{`1'} \mid \text{`2'} \mid ... \mid \text{`}n\text{'} \\
< ParallelElements > &::= \text{`\{'}< ArgNum >^{+}\ \text{`\}'}
\end{aligned}
$$

表 4.1: 並列 E-戦略の定義

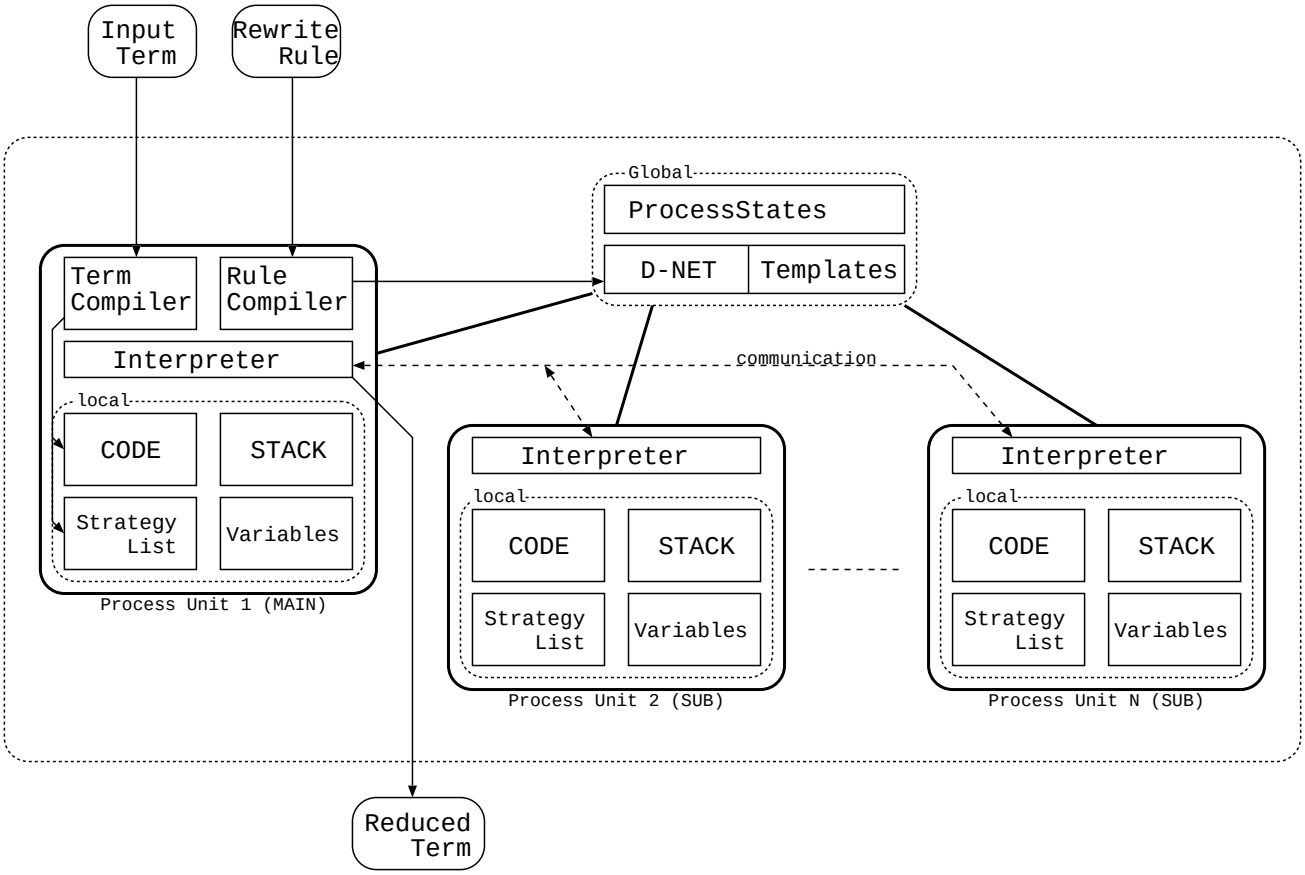

図 4.2: Parallel TRAM の構成

第 5 章

超並列 TRAM の設計

本章では，Parallel TRAM の設計理論を基に，超並列 TRAM を設計する [11].

4章で説明したように，Parallel TRAM はマルチプロセッサを対象としている．しかし，本研究では対象とする計算機を分散メモリ型の計算機としているため，Parallel TRAM の構造・動作をそのまま用いる事はできない．したがって，Parallel TRAM の理論を基に，分散メモリ型の超並列計算機で動作する超並列 TRAM を設計する．

5.1 基本構成

5.1.1 データ授受

Parallel TRAM では FORK 時には，プロセス間のデータの移動はできるだけ最小限になるように，可能な限り実データでなくデータへのポインタなどを渡している (図 5.1). これは，共有メモリ型の計算機ではどのプロセスからでも全てのメモリを直接参照する事が可能であることから，これを利用しているのである．しかし，分散メモリ型の計算機では，プロセスごとに独立したメモリを持っており，プロセス間の直接メモリ参照は不可能であるか (図 5.2 (a))，または複雑な処理を用いなければならない (図 5.2 (b))．本研究で使用する計算機，Cray T3E[15](6.1.1節) では，分散メモリを共有メモリとして利用する機構 (図 5.2(c)) も用意されているが，システムに大きく依存するため，それを用いた場合プログラムの移植性が落ちる．本研究は分散計算機環境での TRAM の高速化に応用が可能であると考えられるため，システムに依存しない形での設計を行う．よって，超並列

TRAM では分散計算機環境で良く利用されているメッセージパシングを利用して必要なデータを全て送受信する事で，プロセス間の直接メモリ参照を無くし，プロセス間のデータ授受の処理をできるだけ単純にする (図 5.3)．これにより，各プロセスを書換えにできるだけ専念させ，処理効率の向上を図ることにする．

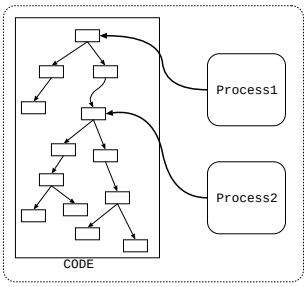

図 5.1: 共有メモリを使用したデータのやりとり

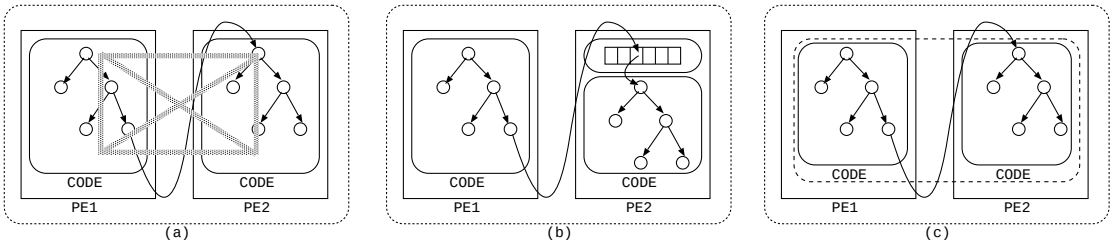

図 5.2: 分散メモリ上での，プロセス間メモリ参照の例

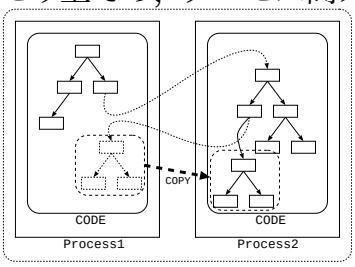

図 5.3: メッセージパシングを使用したデータのやりとり

5.1.2 プロセス管理

次にプロセス管理だが，Parallel TRAM ではプロセス管理領域を共有メモリ上に置き，全てのプロセスからそれを操作していた．しかし，分散メモリではやはりこの方法は使えない．そこで，メッセージパシングシステムで良く利用されるプロセス管理手法である，

Master-Slave(あるいは Master-Worker) モデル [17] を利用したプロセス管理を行う．これは図 5.4に示すようなモデルであり，Slave に仕事を割り当てる Master と，割り当てられた仕事を行う Slave とで構成される．超並列 TRAM では Master に，Slave のプロセス状態を管理するための機構 (5.4節) を備え，これを用いて各 Slave に仕事を割り当てる．

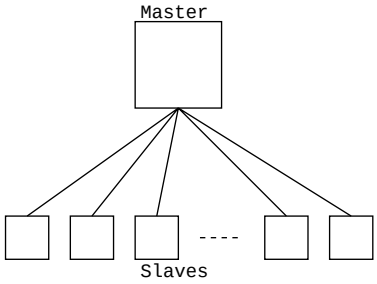

図 5.4: Master-Slave Model

5.1.3 システム構成

超並列 TRAM を，前節で述べた Master-Slave モデルを用いて構成したものが，図 5.5 となる．Master には，入力項のコンパイラと，それに必要な書換え規則のコンパイラ，および 4 つの領域 (コード，戦略リスト，弁別ネット，右辺の雛型) を用意している．さらに，Slave の状態を管理するための機構を持ち，Master はこれを利用して，各 Slave の状態の管理，仕事の割当てを行う．これは 5.4節で詳しく説明する．また各 Slave には，書換え規則のコンパイラ，抽象命令を解釈・実行するインタプリタと，書換えに必要な 7 つの領域 (コード，スタック，戦略リスト，変数，弁別ネット，右辺の雛型，書換え候補リスト) を用意している．

5.1.4 処理の流れ

ここで，超並列 TRAM の処理の流れについて簡単に説明しておく (図 5.6)．

超並列 TRAM では，ユーザーインタフェースは Master が受け持つ．入力された書換え規則は Master と各 Slave に渡され，各々でコンパイルされる (a)．入力項は Master でコンパイルされた後，Slave の一つ (Slave$_a$とする) に渡され，そこで書換えが始まる (b)．書換

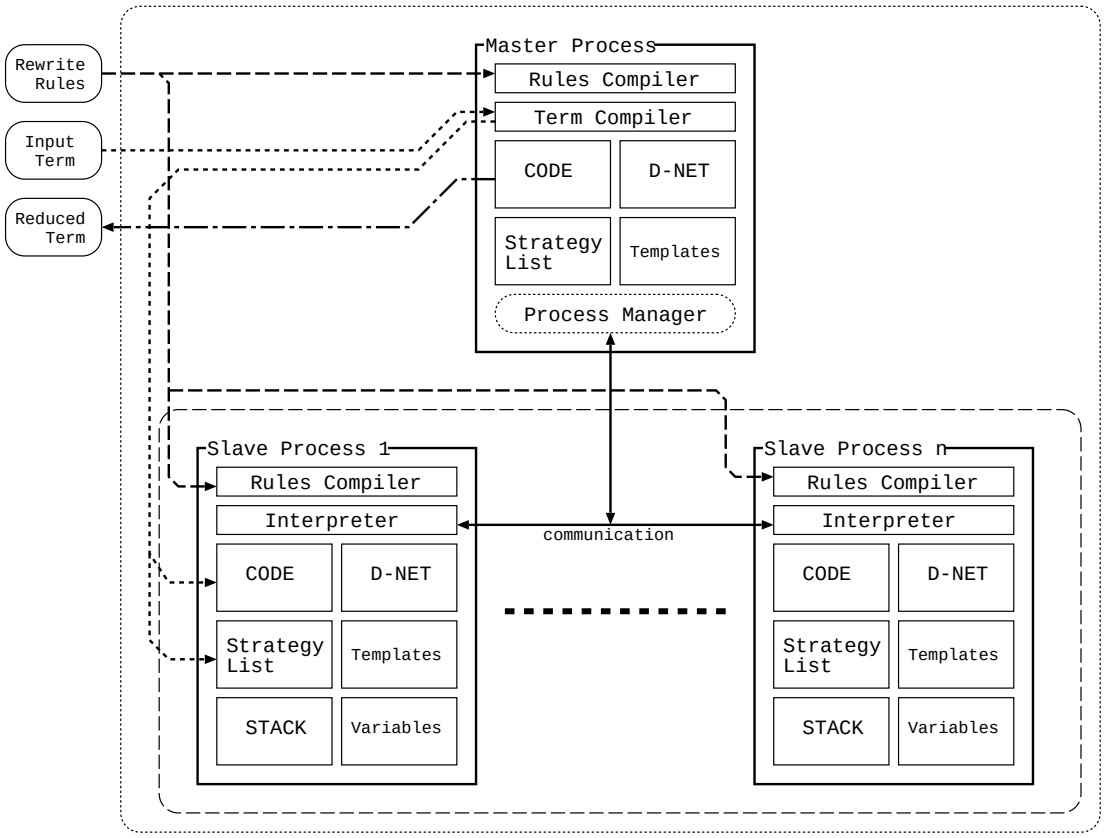

図 5.5: 超並列 TRAM の構成

え中に並列に書換えができる部分項に当たると，Slave_a は Master に対し，並列の書換えを割り当てるための Slave を要求する (c (i))．Master は処理をしていない Slave(Slave_bとする) を返し (c (ii))，Slave_aは Slave_bに部分項を渡し，並列に書換えを行う (c (iii))．Master で処理をしていない Slave が見つからなかった場合，Slave_aはその部分項の書換えを自分で行う．書換えが終った場合，それが部分項の書換えであったらそれを FORK した Slave に返し (d (i))，入力項の書換えであったら Master に返し，それを結果として出力する (d (ii))．

図 5.6: 超並列 TRAM の処理の流れ

5.2 戦略リストの変更

4.3.1節で述べたように，Parallel TRAM では並列 E-戦略を反映し，Fork，Join，Exit の各抽象命令を実行するために，戦略リスト中にそれらの抽象命令を組み込んでいる．超並列 TRAM でも並列 E-戦略を用いて書換え順序を指定するため，やはり戦略リスト中に同様の抽象命令を組み込む.

5.2.1 INFO

5.1.1節で述べたように，超並列 TRAM ではメッセージパシングを利用してデータの授受をおこない，プロセス間の直接メモリ参照をなくしている．したがって，並列項書換えに際し，部分項の書換えに必要なデータは全て相手プロセスに渡される．Parallel TRAM ではこの部分項は，それを持っている全体項のアドレスを持っており，部分項の書換えが終了すると結果はそのアドレスに返される[1]．しかし，分散メモリの場合この方法では，全体項と部分項が異なるプロセス上に存在する事になり，異なるプロセスのアドレスを部分項が持つ事になる．この場合，全体項を持つプロセスで GC が起き項のアドレスが変わる時，部分項が保持しているアドレスも変更しなければならないため，GC 時にプロセス間で同期をとらなければならず，またアドレスの変更と言う作業も必要となる．このため，GC のオーバーヘッドが非常に大きくなり，書換え効率が非常に落ちる．このようなオーバーヘッドを回避するために参照テーブルを導入し，部分項ではアドレスでなくこのテーブルの参照を行わせる (図 5.7)．これにより，プロセスごとに非同期に GC が行えるようになり，また部分項が保持する全体項の参照先を変更せずに済む.

超並列 TRAM ではこの領域を戦略リスト中に「INFO」として埋め込む事にする (図 5.8)．これは戦略リストの構築時に FORK の数だけ戦略リスト中に埋め込まれる．この領域を独立して確保する場合，必要な領域のサイズが不明であり，最大 FORK 可能数だけ確保しなければならないが，このようにする事で，不必要な参照テーブルの領域を確保せずに済む.

[1] 実際には，全体項が保持する部分項へのポインタを張り換える

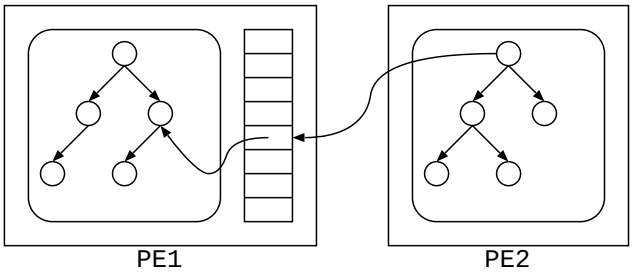

図 5.7: ReferenceTable を用いた親項の参照

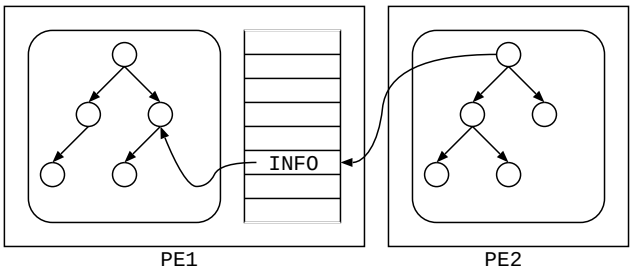

図 5.8: INFO

5.2.2 戦略リストの構造

4.3.1節で説明したように，Parallel TRAM では書換え順序に並列 E- 戦略を反映するために，戦略リストに Fork， Wait， Exit の各命令へのラベルが追加されている．これらの命令ではさまざまな情報が必要となるが，それを戦略リストの 3 つ組の空き領域に格納する[2]．また，参照テーブルとして Info が追加されている．ここで，それぞれの命令が必要とする情報と，その説明をしてゆく．

各領域に含まれる情報は以下のようになる．

Fork	:	<	LabelFork	,	BlockEnd	,	WaitAddr >
Wait	:	<	LabelWait	,	WaitNum	,	NumOfFork >
Exit	:	<	LabelExit	,	WaitAddr	,	WaitTag >
		<	ParentPE	,	ResultTerm	,	InfoNum >
Info	:	<	LabelInfo	,	ParentAddr	,	— >

[2]Exit 命令は使用する情報が多いため，2 ブロックを使用している．

これらの各項目の定義は次のようになる．

LabelFork Fork 命令へのラベル．

BlockEnd Fork 命令で他のプロセスに部分項の書換えを割り付ける際，LabelFork の次の戦略リストから，この BlockEnd の戦略リストまでに対応する書換えを割り付ける．つまり BlockEnd は，引数項書換えの区切りを示し，その値は戦略リスト中のある要素のアドレスを指している．

WaitAddr Fork して他のプロセスに割り当てた書換えが戻るべき Wait のアドレスを指す．

LabelWait Wait 命令へのラベル．

WaitNum そのプロセスの戦略リスト中で，戦略リストの Bottom から何番目にある Wait かを表す．Exit では，WaitAddr の Wait 中の WaitNum を保持している．

NumOfFork この Wait を戻るべき場所としている Fork の割り当てた書換えで，まだ結果の戻らない書換えがいくつあるかを表す．これが 0 の場合，全ての書換えの結果が戻っている事を表す．

LabelExit Exit 命令へのラベル．

ParentPE 書換えの結果を返すプロセスの番号．

ResultTerm 親プロセスへ返す項のアドレス．

InfoNum Fork された書換えの全体項に関する情報が納められている Info 領域の番号．戦略リスト中の WaitAddr からのブロック数となっている．親プロセスは，WaitAddr の何番目の Info に含まれる情報を利用するかをこれで知る事ができる．

LabelInfo Info 命令へのラベル．

ParentAddr Fork した部分項の親アドレス．

5.3 抽象機械命令の定義

超並列 TRAM では Parallel TRAM と同様に，並列書換えを行うために幾つかの抽象命令を追加している．これらは Fork，Wait，Exit，Sleep の各命令である．これらの各命令について，その動作を定義してゆく．なお，アルゴリズムの定義には，C 言語調のシンタックスを用い，関数名はその動作を直接表すことにする．

5.3.1 Fork 命令

Fork 命令は，BlockEnd までの項を他の PE(Child PE) に割当て，並列書換えを開始させる．Child PE の書換えの結果は，WaitAddr にある Wait に返される． その定義は図 5.9のようになる．

```
Fork(){
    childPE ← getChildPE()
    if( childPE == -1 ) return;
    packSendData();
    SendData( childPE );
}
```

図 5.9: Fork 命令のアルゴリズム

getChildPE() で Master に子プロセスを要求し，送られて来たプロセス番号を childPE に納める．この値が−1 なら空きプロセスがない事を表すので，Fork 命令を抜ける．そうでない場合は，packSendData() で子プロセスに送るべきデータ (マッチングプログラム，戦略リスト) をまとめ，SendData(childPE) で子プロセスにまとめたデータを送信する．

5.3.2 Exit 命令

Exit 命令は，Fork によって割り付けられた部分項の書換えが終了した事を示し，その結果を親プロセスに返す．その定義は図 5.10のようになる．

sendReducedDataToParentPE() で，書換え結果のマッチングプログラムを ParentPE の WaitAddr へ送り返す．その後，Master に EXIT 命令を送る．

```
Exit(){
    sendReducedDataToParentPE( ParentPE, WaitAddr );
    sendCommandToMaster( EXIT );
}
```

図 5.10: Exit 命令のアルゴリズム

5.3.3 Wait 命令

Wait 命令は，Fork した部分項の書換えの結果が全て戻って来るのを待つ．その定義は図 5.11のようになる．

```
Wait(){
    if( proveMessageFromSlave() == TRUE ) getReducedTerm();
    if( NumOfFORK == 0 ) return;
    sendCommandToMaster( WAIT );
    while( TRUE ){
        if( proveMessageFromMaster() == TRUE ){
            command ← getMessageFromMaster()
            break;
        }
        if( proveMessageFromSlave() == TRUE ) getReducedTerm();
    }
    switch( command ){
        case Awake:
            return;
        case Reduce:
            getData();
            break;
    }
}
```

図 5.11: Wait 命令のアルゴリズム

まず，proveMessageFromSlave で FORK した書換えの結果が戻って来ているかを調べ，戻って来ていれば getReducedTerm でそれを受け取る．そして，この Wait の持つ NumOfFork が 0 なら FORK した書換えが全て戻って来ているので，Wait 命令の処理を抜ける．そうでないときは，Master に WAIT コマンドを送り，Wait 状態に入った事を知らせる．そして，Master から Awake コマンドか Reduce コマンドが送られて来るのを待つが，この時同時に FORK した書換えが戻って来ているかのチェックと，その受け取りの処理を行

う．これは，5.3.3で説明したように，子プロセスでは書換え結果を親プロセスに返してからIdle 状態へと移るため，Wait 命令のこの待ち状態の時に受け取りの処理を行わないと，デッドロックを起こしてしまうからである．Master からコマンドを受け取ると，Awake コマンドならこのまま Wait 命令を抜け，また Reduce コマンドなら他のプロセスから部分項を受け取り，Wait 命令を抜け，それぞれ書換えを再開する．

5.3.4 Sleep 命令

Sleep 命令は，Master から次に行うべき命令が送られて来るのを待つ．Master では，Sleep 状態の Slave を Idle Process として扱う．その定義は図 5.12のようになる．

```
Sleep(){
    command ← getMessageFromMaster();
    switch( command ){
        case Initialize:
            programFile ← getFileName();
            CompileProgram( programFile );
            break;
        case ReduceInputTerm:
            changeSleepToBingo();
            getData();
            break;
        case ReduceSubTerm:
            getData();
            break;
        case Quit:
            return;
    }
}
```

図 5.12: Sleep 命令のアルゴリズム

Sleep 命令では Master からのコマンドを受け付ける．コマンドは Initialize，ReduceInputTerm，ReduceSubTerm，Quit の 4 種類がある．各コマンドの動作は以下のようになる．

Initialize 書換え規則のコンパイルを行う．Master から入力された書換え規則のファイル名を受け取り，そのファイルを読み込み，コンパイルを行う．

ReduceInputTerm 入力項の書換えを行う．まず，全ての Slave の戦略リストの Bottom には Sleep 命令があり，それを実行してこの命令に処理が移っている．このコマン

ドを受け取った Slave は，changeSleepToBingo で戦略リストの Bottom を入力項の書換えの終了を表す Bingo に変更し，getData で入力項をコンパイルして得られたコードと戦略リストを受け取り，CODE，SL の各領域に配置した後，Sleep 命令の処理を抜ける事で，入力項の書換えが始まる．

ReduceSubTerm 部分項の書換えを行う．部分項を割り付けて来る親 Slave からデータを受け取り，Sleep 命令を抜ける事で，部分項の書換えが始まる．

Quit 超並列 TRAM の処理を終了する．

5.4 プロセス管理機構

超並列 TRAM では Master が Slave の状態の管理と仕事の割当を行う．本節では，そのためのプロセス管理機構について説明する．

5.4.1 Slave 状態

超並列 TRAM では，Master が各 Slave の状態を管理し，その状態に応じて次の動作を決定し，それを Slave に知らせる．Slave 状態には *Idle*，*Wait*，*Busy* があり，その定義は次のようになり，その状態は，図 5.13のように変化する．

Idle 何も処理をせず，他プロセスからの書換えの割当を待っている状態．Slave から FORK の要求があった場合，部分項の書換えを行わせる子プロセスとして優先的に割り当てられる．Idle Stack で管理される．

Wait FORK した部分項の書換えの結果が戻って来るのを待っている状態．Slave から FORK の要求があった場合，Idle Stack に Idle 状態のプロセスがない時，部分項の書換えを行わせる子プロセスとして割り当てられる[3]．Wait 状態はいつ Busy 状態に遷移するか不定であるため，Idle 状態のように Stack でなく，List で管理する．これを Wait List と呼ぶ．Master から Awake コマンドを受け取る事で，Busy 状態となる．

[3] 現在の実装ではこのような処理は行われず，待ち状態のまま待機させている．

Busy 書換えを行っている状態．全ての書換えが終ればIdle状態になり，Wait 命令が実行後されるとWait 状態となる．

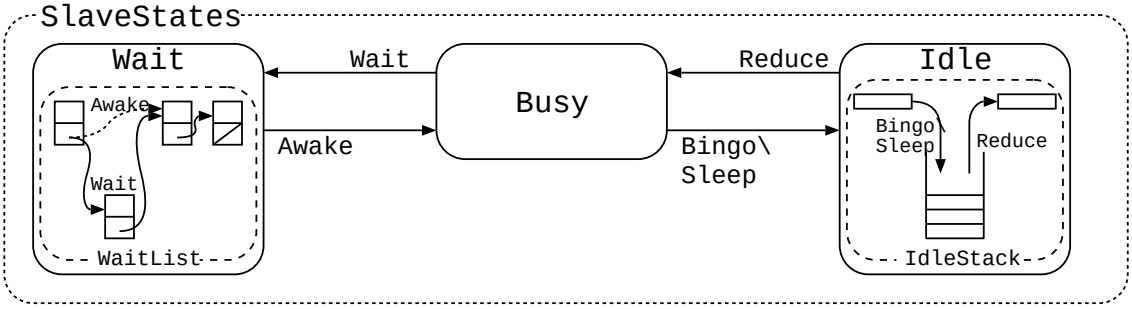

図 5.13: Slave の状態遷移

5.4.2 ForkTable

Wait 状態のプロセスはMaster から Awake メッセージを受け取って次の書換えを実行するのだが，そのためにはMaster が各 Slave の Wait の状態 (Fork の結果をいくつ待っているか) を知る必要がある．そのために，ForkTable(図 5.14) を用いる．これはプロセス番号と WaitNum からなるテーブルで，セルの値が -1 なら Fork した全ての書換えが終了していて WaitNum の Wait の次の書換えが実行可能である事を，また 0 なら Fork した全ての書換えが終っているが，その WaitNum の Wait の前に再開すべき Wait が残っている事を示す．Master は Slave から WAIT，または EXIT 命令を受け取った時，次の操作を行った後，そのセルを参照し，Slave の状態を決定する

- Slave から FORK 命令を受け取った場合，Slave のプロセス番号と Fork の戻るべき Wait の WaitNum から，対応するセルの値をインクリメントする．
- Slave から EXIT 命令を受け取った場合，親プロセスの番号と WaitNum から，対応するセルの値をデクリメントする．もし後0 になった場合，このセルの WaitNum+1 のセルが -1 なら，このセルの値も -1 とする．

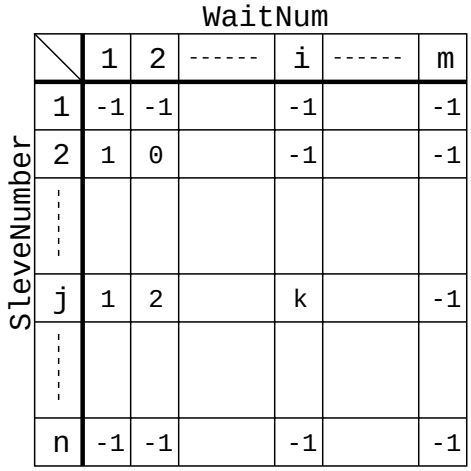

図 5.14: ForkTable

5.4.3 SlaveSchedule

Master での Slave 管理は図 5.15のようなループで処理される．Slave の一つからコマンドを受け付け，そのコマンドに対応した処理を行う．またコマンド受け取り時に，そのコマンドを送ったプロセスを requestPE として確保しておく．受け付けるコマンドは FORK(Slave が Fork 命令を実行した時に送られる)，EXIT(Exit 命令の実行時に送られる)，WAIT(Wait 命令の実行時に送られる)，BINGO(Bingo 命令の実行時に送られる) である．この各コマンドの処理について，詳しく見てゆく．

FORK

FORK コマンドの処理は図 5.16のようになる．getIdlePE で空きプロセス (childPE) を得て，コマンドを送ってきたプロセス，requestPE にその値を送る．

getIdlePE(図 5.17) では，IdleStack から空きプロセスを一つ得る．ここで空きプロセスがなかった場合には，NoIdlePE を返す．得られたプロセスに対し，requestPE から部分項の書換えが割当られる事を知らせる．また，IdlePE を要求したのが Master でない場合には，ForkTable の操作を行う．

```
slaveSchedule(){
    while( TRUE ){
        command ← ReceveMessage( ANY_SLAVES );
        requestPE ← SenderSlaveNumber;
        switch( command ){
            case FORK:
                childPE <- getIdlePE;
                SendMessage( requestPE, childPE );
                break;
            case EXIT:
                parentPE <- ReceveMessage( requestPE );
                waitNum <- ReceveMessage( requestPE );
                putIdleStack( requestPE );
                decrementForkTable( parentPE, waitNum );
                state <- checkForkTable( parentPE, waitNum );
                if( state = TopWait ){
                    SendMessage( parentPE, AwakeCommand );
                }
                break;
            case WAIT:
                waitNum <- ReceveMessage( requestPE );
                state <- checkForkTable( requestPE, waitNum );
                if( state = TopWait ){
                    SendMessage( requestPE, AwakeCommand );
                } else {
                    putWaitList( requestPE );
                }
                break;
            case BINGO:
                resultTerm <- ReceveMessage( requestPE );
                printTerm( resultTerm );
                return;
        }
    }
}
```

図 5.15: Slave 管理のアルゴリズム

```
    case FORK:
        childPE <- getIdlePE;
        SendMessage( requestPE, childPE );
        break;
```

図 5.16: FORK コマンド

```
getIdlePE( requestPE:空きプロセスを要求したプロセス,
           waitTAG:空きプロセスを要求したFORKに対応するWAITの番号 ){
    if( IdleStack != Empty ){
        IdlePE <- GetIdleStack
    } else {
        return NoIdlePE;
    }
    StartReduceMessage( IdlePE );
    if( waitTAG != Initialize ){
        incForkTable( requestPE, waitTAG );
    }
    return IdlePE;
}
```

図 5.17: getIdlePE

EXIT

EXIT コマンドの定義は図5.18のようになる．まず requestPE に書換えを割当てた親プロセス番号 (parentPE) と，合流する Wait の waitNum を requestPE から受け取り，ForkTable の対応するセルをデクリメントする (decrementForkTable)．次に，checkForkTable で parentPE の waitNum の状態を調べ， Awake 可能な状態なら parentPE に Awake コマンドを送る (SendMessage)．最後に，putIdleStack 命令で requestPE を IdleStack に積み Idle 状態に変更する．

```
case EXIT:
    parentPE <- ReceveMessage( requestPE );
    waitNum <- ReceveMessage( requestPE );
    decrementForkTable( parentPE, waitNum );
    state <- checkForkTable( parentPE, waitNum );
    if( state = TopWait ){
        SendMessage( parentPE, AwakeCommand );
    }
    putIdleStack( requestPE );
    break;
```

図 5.18: EXIT コマンド

WAIT

WAIT コマンドの定義は図5.19のようになる．まず，requestPE から WAIT コマンドを送った Wait の waitNum を受け取り，checkForkTable でその状態を調べる．その結果，Awake 可能な状態なら requestPE に Awake コマンドを送り (SendMessage)，そうでなければ requestPE を WaitList に入れ，Wait 状態にする (putWaitList)．

BINGO

BINGO コマンドの定義は図5.20のようになる．requestPE から書換え結果を受け取り，それを出力する．そして，slaveSchedule のループを抜ける．

```
case WAIT:
    waitNum <- ReceveMessage( requestPE );
    state <- checkForkTable( requestPE, waitNum );
    if( state = TopWait ){
        SendMessage( requestPE, AwakeCommand );
    } else {
        putWaitList( requestPE );
    }
    break;
```

図 5.19: WAIT コマンド

```
case BINGO:
    resultTerm <- ReceveMessage( requestPE );
    printTerm( resultTerm );
    return;
```

図 5.20: BINGO コマンド

5.5 スケジューリングの詳細

本節では，これまで説明してきた事項をまとめ，超並列化 Parallel TRAM の動作の詳細を説明する．説明のための例として用いる書換え規則を次のように定める．

```
add(X, s(Y)) -> s(add(X, Y))
padd(X, Y) -> add(X, Y) {strat: ({1 2} 0)}
```

1. 書換え規則のコンパイル (図 5.21 (1))
   書換え規則が与えられると，それは全てのプロセスに渡され，それぞれのプロセスでコンパイルされる．

2. 入力項のコンパイル (図 5.21 (2))
   入力項が与えられると，それはまず Master に渡され，そこでコンパイルされる．コンパイルされたコードと戦略リストは，Slave の一つ (Slave$_a$ とする) に渡され，Slave$_a$ でコンパイルが始まる．

3. 書換え Master からコードを受け取った Slave_a は，戦略リストからコードのラベルを取り出し，そのコードの実行を行ってゆく．

(I) Fork(図 5.21 (3))

書換え中に Fork 命令が実行されると，次のような動作が起こる．

i. Slave_a で Fork 命令が実行される．

ii. Master に FORK コマンドを送り，部分項の書換えを割り当てるための，子プロセスを要求する．

iii. Master は FORK コマンドを実行し，その結果空きプロセスがあれば (Slave_b) その番号を Slave_a に返す．また，Slave_b に，Slave_a から部分項の受け取りを指示する．空きプロセスが無い場合，-1 を Slave_a に返す．

iv. Slave_a は Slave_b に，部分項の書換えを割り当てる．Slave_b は部分項を受け取り，その書換えを始める．

(II) Exit(図 5.21 (4))

書換え中に Exit 命令が実行されると，次のような動作が起こる．

i. Slave_b で Exit 命令が実行される．

ii. 部分項の書換えを割り当てた Slave_a に，書換えの結果を返す．

iii. Master に EXIT コマンドを送り，部分項の書換えが終わり，Idle 状態に移った事を知らせる．

iv. Master は EXIT コマンドを実行し，Slave_a が Wait 状態なら Slave_a に Awake コマンドを送り，Slave_a に書換えを続けさせる．

(III) Wait(図 5.21 (5))

書換え中に Wait 命令が実行されると，次のような動作が起こる．

i. Slave_a で Wait 命令が実行される．

ii. Fork した書換えを割り付けた子プロセス (この場合 Slave_b) から，書換えの結果が送られているか確認する．送られてきている場合，それを受け取る．

iii. NumOfFORK が 0 ならば，次の書換えに移る．そうでなければ，Master に WAIT コマンドを送り，Wait 状態に移った事を知らせる．

iv. Master は WAIT コマンドを実行し，ForkTable の参照結果が TRUE なら Slave_aに Awake コマンドを送る．また，参照結果が FALSE なら Slave_aを WaitList に入れる．

v. Slave_aでは，Master から Awake コマンドが送られて来るまで，Fork した書換えの結果を受け付ける．

(IV) Bingo (図 5.21 (6))

書換え中に Bingo 命令が実行されると，その時点でのマッチングプログラムを Master に返し，Master はそれを入力項の書換え結果として出力する．

5.6 GC

通常分散メモリ型の計算機では，プロセス間の直接メモリ参照が起こる場合，GC(ガベージコレクション) はグローバルな同期を必要とする．これは，参照先のデータのアドレスが GC によって変更されることから，GC の動作中はそのデータへの参照を禁止するのと，GC 後にアドレスが変更されたデータを参照している参照元の参照アドレスを変更する必要があるためである．

しかし，超並列 TRAM では 5.2.1節で述べたように,「**INFO**」という参照テーブルを利用して他プロセスへの直接メモリ参照は行わないようになっている．このため，GC の際にプロセス間でグローバルな同期を取る必要は無く，プロセスごとに必要に応じて個別に GC をおこなう事が可能である．したがって，超並列 TRAM では GC は，それぞれのプロセスで必要に応じてローカルに GC を行わせることにする．また，その方法は TRAM と同じ，コピー方式の GC を採用する．

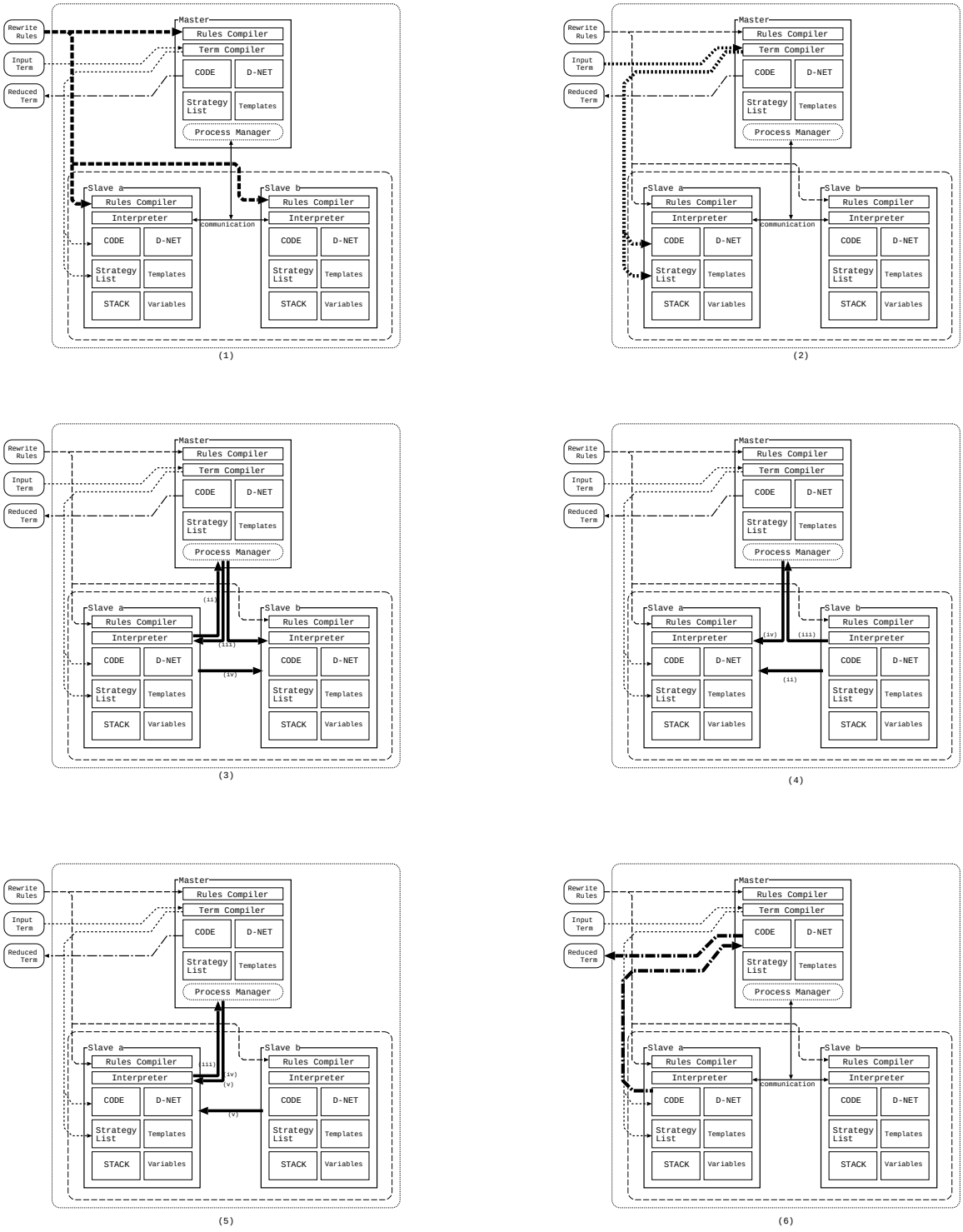

図 5.21: 超並列 TRAM の処理の詳細な流れ

第 6 章

実装と評価

第 5章では，Parallel TRAM の理論を基に TRAM を分散メモリ型の超並列計算機上で並列書換えを行わせる超並列 TRAM の設計を行った．これにより，どの程度の書換え効率の改善が行えたかを見極めるために，この超並列 TRAM を実際の計算機上に実装し，その評価を行う必要がある．そこで本章では，超並列 TRAM を共有分散メモリ型の超並列計算機である Cray T3E の上に，分散計算機環境で良く用いられるメッセージパシングライブラリ，MPI を用いて実装を行う．また，その性能についてベンチマークを実行し，TRAM との比較を行う．

なお，実装には C 言語 [16] を使用している．

6.1 実装

6.1.1 Cray T3E

今回実装を行った計算機は，SGI Cray Research 社の T3E システムで，これは分散共有メモリ型の超並列計算機である．プロセス数は 128 個で，各プロセスごとに 64MByte のローカルメモリを実装している．プロセス間結合方式は３次元トーラストポロジで，プロセス間のデータ転送実効速度は 450MByte/s となっている．

6.1.2 MPI

MPI(Message Passing Interface)[1][2][3][4][5][6] は，異機種環境における統一的な分散処理環境を提供する事を目的として提唱されたメッセージパシング型ライブラリである．PVM のようなさまざまな種類のメッセージパシング・ライブラリの機能を取り入れ，統一的なコーリング・シーケンスによって，ハードウェアや OS に依存することなくアプリケーションの開発を行える環境を提供している．MPI はライブラリとして FORTRAN，および C 言語から使用する事が可能で，タスク間のメッセージの送受信，バリア同期など，メッセージパシング形式の並列化分散アプリケーションの構築を支援する．なお，プログラムの構築に利用したユーティリティについて，AppendexAに記載する．

本研究で利用した MPI の機能について，記述してゆく．

コミュニケータ

コミュニケータとは，プロセスのグループと通信領域を結びつけるもので，メッセージ通信を行う領域を定義したものである．システムやユーザによって定義される．プロセス間通信はこのコミュニケータの領域内で行われ，またこれにより特定のプロセスグループ内でのコレクティブなメッセージの配信を行う事ができる．

コミュニケータにはIntraCommunicator と InterCommunicator とがあり，前者はグループ間通信に，後者はグループ内通信に用いられる．また，InterCommunicator はコレクティブなメッセージ通信に用いる事ができ，これによりグループ内の全てのプロセスにメッセージを同時配信する事が可能である．

プロセスにはそれぞれ番号が与えられ，この番号はプロセスが属する各グループ内で固有の番号が与えられる．複数のグループに属している場合，それぞれのグループごとに固有の番号を与えられ，したがって複数のプロセス番号をそのプロセスは保有することになる．

超並列 TRAM では，使用するプロセスを Master グループと Slave グループに分け，それぞれの IntraCommunicator と，Master グループと Slave グループ間での InterCommunicator の 3 種類のコミュニケータを使用している．

メッセージ通信

MPI でのメッセージ通信は基本的に ”point to point communication” であり，メッセージの送信側と受信側でそれぞれ送信・受信命令を呼び出すことでメッセージのやりとりが行われる．メッセージの送信命令にはいくつかの通信モードがあり，これらを適時選択する事で効率的なメッセージ通信が行える．その通信モードは以下のようになる．

Synchronous Mode メッセージの受信側で受信命令が実行されていない場合には，それが実行されるまでメッセージの送信を待つ．

Bufferd Mode メッセージの送信側でユーザが送信バッファを確保し，送信命令の実行時にその送信バッファにメッセージを送る事で，受信側で受信命令が実行されていなくてもメッセージの送信を終える事が出来る．

Standard Mode 送信バッファの確保が可能なだけのメモリがある場合，送信バッファをシステムが自動的に確保し，Bufferd Mode として機能する．バッファの確保が行えない場合，Synchronous Mode として機能する．

Ready Mode メッセージの受信側で受信命令が既に実行されている事が前提であり，受信命令が実行されていない場合，メッセージの送信は失敗となり，エラーを返す．

また，通常の命令はその処理が終了するまで次の処理に移れないが，*nonblocking* な命令を用いると，複数の命令を同時に処理する事が可能となる．これにより，複数のメッセージを同時に送受信する事が可能となる．ちなみに，通常の命令は *blocking* な命令と呼ばれる．

データのパック，アンパック

プロセス間でデータを送受信する場合，サイズの小さなデータを，送受信命令を繰り返し呼び出してやりとりするより，それらのデータを一つのバッファにまとめ，1 回の送受信命令でまとめてやりとりした方が，命令の呼び出し時にかかるオーバーヘッドが少なく，効率的なデータ通信が行える．そのため，メモリのあちこちに散らばったデータをパックするための命令が MPI には用意されている．また，パックされたデータをアンパックするための命令も当然用意されている．超並列 TRAM では，部分項の送信時に CODE 領

域内に散らばったコードをまとめ，受信側でそのコードを CODE 領域内に再配置するために，これらの命令を用いている．

6.2 性能評価、および考察

本節では，TRAM の超並列化により，どの程度の効率改善が実現できたのかを，いくつかのベンチマークを実行させて評価を行う．基本的な評価方法は，T3E 上で動作する超並列 TRAM と，同じく T3E 上に移植した TRAM の実行速度を比較する事で行うものとする．なお，ベンチマークに使用したプログラムを，Appendex Bに記載する．

6.2.1 逐次書換え性能の比較

まず，TRAM と超並列 TRAM における逐次書換えの性能の比較を行う．そのために，次のような計算を行わせた．

1. TRAM におけるフィボナッチ数列 fib(34) の計算．

2. 超並列 TRAM，逐次書換えにおける フィボナッチ数列 fib(34) の計算．

その結果は表 6.1のようになる．

	書換え時間 (s)	r/s	speed up
TRAM	223.3	247911	1
超並列 TRAM	235.7	234908	0.95

表 6.1: 逐次書換え性能の評価

この結果から，逐次書換えに関して，超並列 TRAM は TRAM とほぼ遜色ない処理能力を有している事が確認できる．

6.2.2 基本性能の評価

基本性能の評価として，次の計算を行わせた．

評価１

1. TRAM におけるフィボナッチ数列 fib(34) の計算．

2. 超並列 TRAM における並列フィボナッチ数列 pfib(34) の計算．Slave 数を 1〜127 個の間で 10 個おきに計測.

その結果を，表 6.2に，また速度向上比のグラフを図 6.1に示す.

	TRAM	超並列 TRAM						
Slave 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	223	232.3	38.9	22.4	18.2	17.4	14.3	11.5
r/s (x10,000)	24.7	23.83	142.5	247.6	304.0	317.5	386.5	480.2
speed up	1	0.96	5.77	10.0	12.3	12.9	15.6	17.4
FORK 成功回数	-	0	23	44	44	50	57	60
	TRAM	超並列 TRAM						
Slave 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	223	12.0	9.1	3.72	3.50	3.71	3.45	3.80
r/s (x10,000)	24.7	460.6	603.0	1489	1583	1493	1603	1656
speed up	1	18.6	24.4	60.3	64.1	60.4	64.9	67.0
FORK 成功回数	-	70	79	88	88	88	88	88

最大 FORK 数：88

表 6.2: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(34) の演算

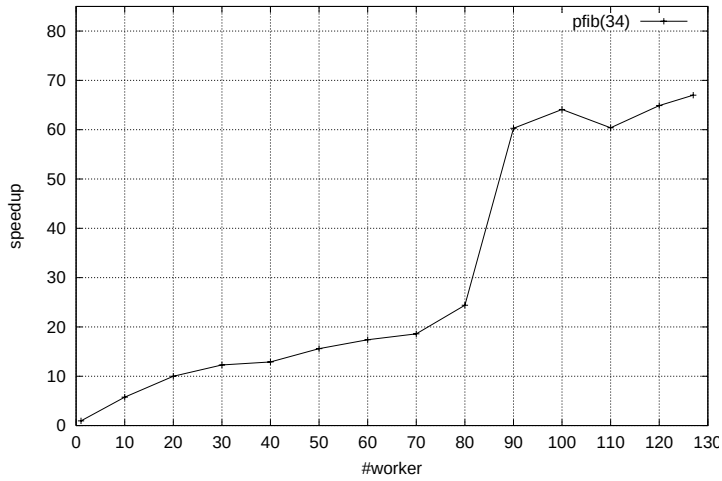

図 6.1: pfib(34) の速度向上比

pfib(34) の演算は，最大で 88 回 FORK が行われる．従って，Slave を 89 個[1]以上使用した時全ての FORK が成功し，理論上最大の並列度が得られる事になるが，上記結果からその推論が正しい事が分かる．しかし，この速度向上比は，理論的には FORK 数だけ，つまり 88 倍は向上するはずである．この結果では，約 30%ほど効率が低下している．これは，Fork，あるいは Wait など，逐次書換えでは行われない命令が実行され，さらにこの時必ず何らかのメッセージ通信が行われるため，そのオーバーヘッドによって能力の低下が生じているものと考えられる．

この結果を観察すると，Slave 数が 80 個と 90 個の間で Speed up にかなりの差がある事が分かる．最大 FORK 数は 88 回であることから，Slave 数が 80 個の場合いくつか FORK を失敗しており，90 個の場合全ての FORK が成功している．FORK が失敗した場合，その部分の書換えをそのプロセスで行わなければならないため，その書換え時間は FORK が成功した場合に比べかなり遅くなるものと考えられる．

評価２

評価１で使用したプログラムは，過度に FORK を行うのを抑えるために，書換え規則中にしきい値を設け，その項の引数がしきい値に満たない場合には逐次に書換えを行うようにしている．評価１ではこのしきい値は 25 であり，pfib(25) が与えられると，それは逐次に処理するようになっている．入力項として同じ項 (この場合 pfib(34)) を与えた場合，このしきい値をを変更する事で最大 FORK 回数が変わるため，このしきい値の違いにより書換え効率に何らかの違いが現れると考えられる．そこで評価２として，このしきい値を 24〜27 と変化させ，その様子を観察した．

この結果は表 6.3〜6.5のようになる (しきい値：25 は表 6.2)．また，この Speed up のグラフは，図 6.2のようになる．この結果から，全ての FORK が成功した場合，その入力項が持つ並列度を最大限に生かす事ができ，その FORK 数に近い性能向上を得る事ができる，といえる．そのため，入力項の持つ並列度が，使用する Worker 数以内，かつ最大となるよう，書換え規則を定義する事で，かなりの書換え効率の向上が見込めると考えられる．

[1] 最初に Master から入力項を受け取る Slave を加えるため，88+1 個になる．

	TRAM	超並列 TRAM						
使用 PE 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	223	232.3	37.1	18.5	15.8	12.8	10.8	11.1
r/s (x10,000)	24.7	23.8	149.3	299.8	350.2	433.5	510.3	499.4
speed up	1	0.96	6.04	12.1	14.2	17.6	20.7	20.2
FORK 成功回数	-	0	23	58	75	75	78	77
	TRAM	超並列 TRAM						
使用 PE 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	223	11.5	12.1	9.86	9.15	7.78	7.28	8.00
r/s (x10,000)	24.7	480.4	458.4	561.7	604.8	711.8	760.8	692.1
speed up	1	19.4	18.6	22.7	24.5	28.8	30.8	28.0
FORK 成功回数	-	83	91	98	106	110	120	127

最大 FORK 数：143

表 6.3: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(34) の演算 (しきい値:24)

	TRAM	超並列 TRAM						
使用 PE 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	223	232.4	38.7	24.7	24.1	18.2	13.2	5.34
r/s (x10,000)	24.7	23.8	143.5	225.6	229.7	304.7	420.4	1037
speed up	1	0.96	5.81	9.13	9.30	12.34	17.02	41.98
FORK 成功回数	-	0	17	27	32	40	49	54
	TRAM	超並列 TRAM						
使用 PE 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	223	5.51	5.80	5.75	5.39	5.73	5.33	5.89
r/s (x10,000)	24.7	1004	955.2	962.2	1028	966.3	1038	940.1
speed up	1	40.64	38.67	38.96	41.62	39.12	42.02	38.06
FORK 成功回数	-	54	54	54	54	54	54	54

最大 FORK 数：54

表 6.4: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(34) の演算 (しきい値:26)

	TRAM	超並列 TRAM						
使用 PE 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	223	232.3	43.0	29.2	22.1	8.31	8.27	8.41
r/s (x10,000)	24.7	23.8	128.6	189.6	250.7	666.5	669.9	658.7
speed up	1	0.96	5.21	7.68	10.15	26.98	27.12	26.66
FORK 成功回数	-	0	14	20	29	33	33	33
	TRAM	超並列 TRAM						
使用 PE 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	223	8.73	9.15	9.08	8.46	9.02	8.41	9.28
r/s (x10,000)	24.7	634.2	605.0	609.7	654.1	613.5	658.7	596.4
speed up	1	25.68	24.49	24.68	26.48	24.84	26.67	24.15
FORK 成功回数	-	33	33	33	33	33	33	33

最大 FORK 数：27

表 6.5: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(34) の演算 (しきい値:27)

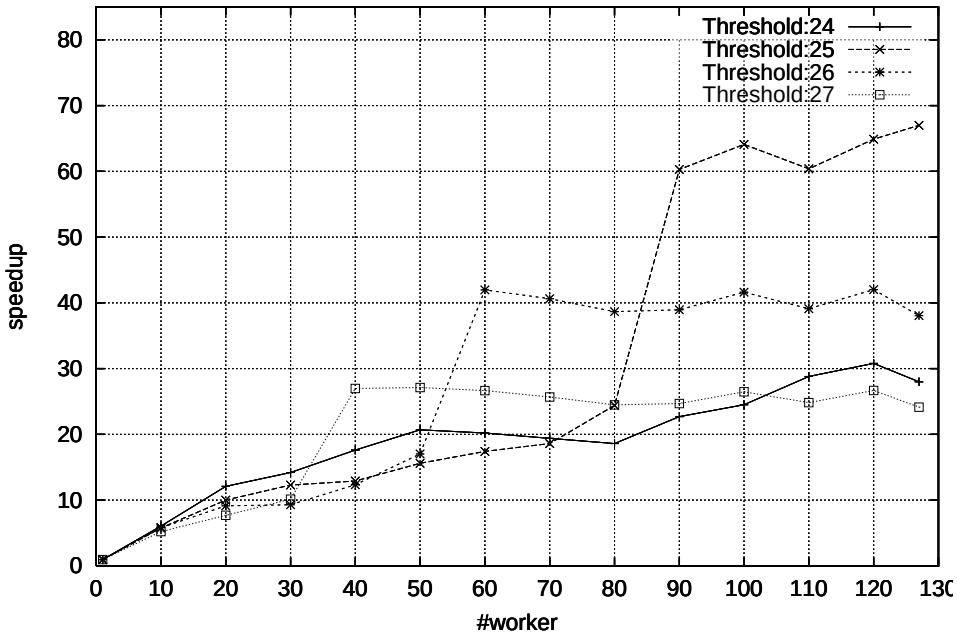

図 6.2: しきい値を変化させた時の Speed UP の変化

評価 3

評価１，評価２は入力項として，プリミティブを用いた項を与えていた．これは入力項に Successor 関数を用いた場合，項の表現が巨大となるため，メモリが足りず，大きな値の計算が不可能となるためである．また，項のデータサイズが大きくなるため，プロセス間のデータ送信などに時間がかかり，あまり高速化されないのではないかと考えたためである．この点が実際どのようになるかを確かめるために，評価３として次のような計算を行なわせた．フィボナッチ数列の計算で，Successor 関数で計算可能な最大の数である 25 を，プリミティブと Successor 関数を用いた表現で計算させる．その結果，どのような差が現れるかを観察した．なお，しきい値は 17 とし，これにより FORK は 88 起こる．

この結果は，表 6.6，6.7，図 6.3のようになる．

この結果から，Successor 関数を用いた時のように，項のサイズが非常に大きくなる場合，プリミティブを用いた場合に比べ，格段に Speed up が低くなることが分かる．これは，項のサイズが大きくなると，プロセス間を転送すべきデータの量も増大し，その結果データ転送にかかる時間が増すため，処理性能が低くなってしまうものと考えられる．このため，超並列 TRAM では項のサイズがそれほど大きくならない計算に対して有効であるといえる．

	TRAM	超並列 TRAM						
使用 Slave 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	5.15	5.36	3.27	3.05	2.97	2.97	2.94	2.75
r/s (x10,000)	16.6	16.0	26.2	28.1	28.9	28.9	29.2	31.2
speed up	1	0.96	1.58	1.69	1.74	1.74	1.75	1.88
FORK 成功回数	-	0	25	44	47	50	57	65
	TRAM	超並列 TRAM						
使用 Slave 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	235	2.78	2.73	2.76	2.77	-	-	-
r/s (x10,000)	23	30.9	31.4	31.1	31.0	-	-	-
speed up	1	1.86	1.89	1.87	1.87	-	-	-
FORK 成功回数	-	83	88	88	88	-	-	-

表 6.6: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(25) の演算 (Successor 関数版)

	TRAM	超並列 TRAM						
使用 Slave 数	1	1	10	20	30	40	50	60
書換え時間 (秒)	5.15	5.36	3.27	3.05	2.97	2.97	2.94	2.75
r/s (x10,000)	16.6	16.0	26.2	28.1	28.9	28.9	29.2	31.2
speed up	1	0.96	1.58	1.69	1.74	1.74	1.75	1.88
FORK 成功回数	-	0	25	44	47	50	57	65

	TRAM	超並列 TRAM						
使用 Slave 数	1	70	80	90	100	110	120	127
書換え時間 (秒)	235	2.78	2.73	2.76	2.77	-	-	-
r/s (x10,000)	23	30.9	31.4	31.1	31.0	-	-	-
speed up	1	1.86	1.89	1.87	1.87	-	-	-
FORK 成功回数	-	83	88	88	88	-	-	-

表 6.7: 逐次型 TRAM と超並列 TRAM の性能評価：Fib(25) の演算 (Primitive 版)

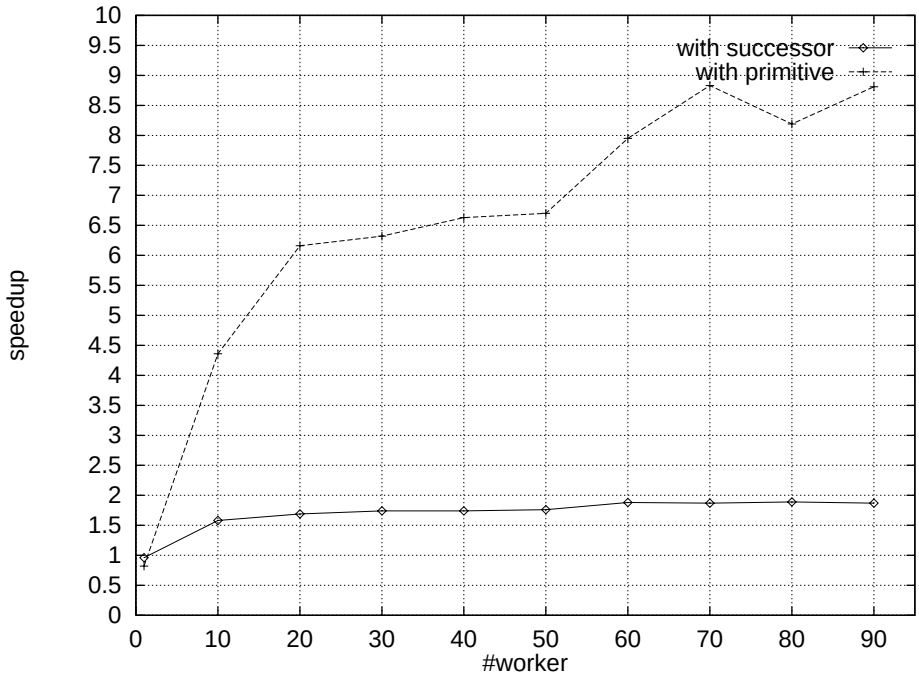

図 6.3: プリミティブと Successor 関数を用いた時の Speed UP の変化

第 7 章

おわりに

本研究では, 単一プロセス上での動作を想定して設計された項書換え抽象機械, "TRAM" を，マルチプロセスにおいて並列書換えを行えるよう拡張した"Parallel TRAM"の設計理論を基に，分散メモリ型の超並列計算機で並列書換えを行えるように拡張した"超並列 TRAM"の設計を完成させた．また，設計完了した超並列 TRAM を Cray T3E システム (プロセス数:128) 上に，メッセージパシングライブラリである"MPI"を用いて実装を行い，種々の評価を行った．

以上の事から次のような結論を得た．

7.1 Master-Slave Model

超並列 TRAM の計算モデルとして Master-Slave model を用いる事で，超並列化による並列項書換えの割当を効率的に行う事が出来るようになった．また，メッセージパシングを用いる事で，実装する環境に依存しない超並列 TRAM の設計が行えた．これにより，プログラムの移植性を高く保つ事が出来た．

7.2 INFO

分散メモリシステムではプロセス外，他プロセスへの直接メモリ参照を行った場合，GC を同期的に行わせる必要が生じるため，それにより書換え効率が非常に低下する可能性があった．超並列 TRAM では外部参照テーブルを用いる事により，プロセス間の直接メ

モリ参照を無くし，GC をローカルに行えるようにする事で書換え効率の向上が達成できた．また，参照テーブルを戦略リスト中に”INFO”領域として確保する事で，必要な参照テーブルだけを確実に確保することが出来るようになった．

7.3 実行速度

逐次書換え性能の評価，基本性能の評価という 2 つの項目について，実装した超並列 TRAM の評価を行った．評価 1 の結果から，プロセス数が 128 の場合にフィボナッチ数列の計算で TRAM の 67.1 倍の書換え速度の向上が確認できた．これにより，TRAM の超並列化により項書換えシステムの持つ並列性がかなり引き出され，かなりの書換え効率の改善が行える事を実証する事が出来た．また，評価 2 の結果から，プロセス数，FORK 数が増すほど，並列書換え時のオーバーヘッドが増す事が確認された．さらに，評価 3 から，超並列 TRAM は通信メッセージ処理時間に対し，書換え処理時間の割合が大きい計算ほど書換え効率が向上する事が分かった．

7.4 今後の課題

本研究で設計・実装した超並列 TRAM のより正確な評価と改良のために以下の事を今後の課題としたい．

7.4.1 メッセージ通信の効率化

現在の超並列 TRAM では，確実に動作する事を第一の目標としているため，メッセージ通信において必ずしも最適化が行われているとは言い難い．以下の点を考慮し，さらに改良を加える事で書換え効率の向上が見込めるだろう．

メッセージ通信モードの変更

現在の超並列 TRAM では，より確実な動作を保証するために，プロセス間のメッセージ通信は同期モードで行っている．しかし，このメッセージ通信モードを可能な限り非同期モードとする事で，メッセージ通信時のオーバーヘッドを減少させる事が出来，さらに書換え速度が向上すると考えられる．

転送データ量の削減

現在，FORK 時に親プロセスは戦略リストとマッチングプログラムを子プロセスに転送している．そのため，プロセス間で転送されるデータ量，また転送前の処理が多少のオーバーヘッドとなっている．そこで，FORK 時にはマッチングプログラムのみを子プロセスに転送し，子プロセス上でそのマッチングプログラムを基に戦略リストを構築する方法を取る事で，プロセス間で転送されるデータ量を減らし，転送時間を減らすことが出来ると考えられる．

7.4.2 設計仕様の形式化

本来これは設計段階で行わなければならないのだが，今回設計した超並列 TRAM を仕様記述言語を用いて形式化し，正確に動作するのかを確認する必要がある．特に並列計算を行う場合には，デッドロック等の不都合が起こる可能性があるため，動作の正当性を検証する事は，非常に重要である．

7.4.3 他の分散計算機環境への実装

今回の実装は，分散メモリ型超並列計算機である Cray T3E システム上で行ったが，このようなシステムは通常利用できるようなものではない．しかし，メッセージパシングは他の分散計算機環境，ワークステーションクラスタ等でも利用されている事から，メッセージパシングを利用し分散メモリ上での超並列化が有効である事が確認できた今，これらの分散計算機環境上に超並列 TRAM を実装する事で，効率の良い項書換えシステムをより一般的なものとする事が可能であろう．

7.4.4 他の並列項書換えシステムとの比較

今回の評価では，本研究のベースである TRAM との比較しか行う事が出来なかった．超並列 TRAM の能力を正確に評価するために，その設計理論とした Parallel TRAM やその他の並列項書換えシステムとの比較を客観的に行い，それぞれの性能を冷静に分析する事が必要である．

謝辞

本研究を終始御指導下さった二木厚吉先生に感謝致します．また，有益な助言をして下さった渡部卓雄先生，緒方和博先生，Răzvan Diaconescu 先生に感謝致します．様々な質問に快く答えて頂いた五百蔵重典氏に感謝致します．また，研究に関する議論につき合って頂いた言語設計学講座の皆様にお礼を申し上げます．

参考文献

[1] W. Gropp, E. Lusk, A. Skjellum. USING MPI : Portable Parallel Programming with the Message-Passing Interface. The MIT Press, 1996.

[2] M. Snir, S. W. Otto, S. H. Lederman, D. W. Walker, J. Dongarra. MPI : The Complete Reference. The MIT Press, 1996.

[3] P. S. Pacheco. Parallel Programming with MPI. Morgan Kaufmann Publishers, 1997.

[4] MPI:http://www.mcs.anl.gov/mpi/index.html

[5] M. A. R. Dantas, E. J. Zaluska. Improving Load Balancing in an MPI Environment with Resource Management. High-Performance Computing and Networking. LNCS 1067 Springer. (1996) 959-960

[6] R. Hempel. The MPI Standard for Message Passing. High-Performance Computing and Networking. LNCS 797 Springer-Verlag. (1994) 247-252

[7] 二木厚吉、外山芳人. 項書換え型計算モデルとその応用. 情報処理 Vol.24, No.2, 情報処理学会, 1983.

[8] K. Ogata, K. Ohhara and K. Futatsugi. TRAM: An Abstract Machine for Order-Sorted Conditional Term Rewriting Systems. Proc. of International Conference on Rewriting Techniques and Applications. LNCS 1232 Springer-Verlag. (1997) 335-338

[9] 近藤勝. 並列項書換え抽象機械：Parallel TRAM の設計と実装. 修士論文, JAIST, 1997.

[10] K. Ogata, M. Kondo, S. Ioroi and K. Futatsugi. Design and Implementation of Parallel TRAM. Proc. of International Conference on Euro-Par'97. Springer Verlag. (1997)

[11] 平田寛道、緒方和博、二木厚吉. 超並列計算機を用いた項書換えシステムの高速化. 日本ソフトウェア科学会, 第 14 回 (1997 年度) 大会論文集, pp.413-416

[12] C. Kirchner, P. Viry. Implementing Parallel Rewriting. International Workshop PLILP'90. Springer Verlag. (1990)

[13] J. Goguen, C. Kirchner, J. Meseguer. Concurrent Term Rewriting as a Model of Computation. Proc. of a Workshop. Springer Verlag. (1986)

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science.

[15] Cray T3E:http://www.cray.com/products/systems/crayt3e

[16] Cray C/C++ Reference Manual, SR2179 2.0. Cray Research, Inc. (1996)

[17] H. C. Chen, A. Lim, N. A. Warsi. Multilevel Master-Slave Prallel Programming Models. Concurrency and Parallelism, Programming, Networking, and Scecurity. LNCS 1179 Springer. (1996) 337-338

第 A 章

MPP Tools

A.1 Cray Totalview

Cray TotalView(図 A.1) は，Cray T3E システムの大規模なプロセス数を同時に表示し，個別にデバッグ可能な MIMD 対応デバッガである．

A.2 MPP Apprentice

MPP Apprentice(図 A.2) は，以下のような特徴を持つ性能解析ツールである．

- X Window System 対応のユーザ・インタフェース．
- ハードウェアの機能情報 (PE，メモリ，インターコネクト等) の表示．
- PVM ルーチン情報の表示．
- ソースコードブラウザとの連動機能．

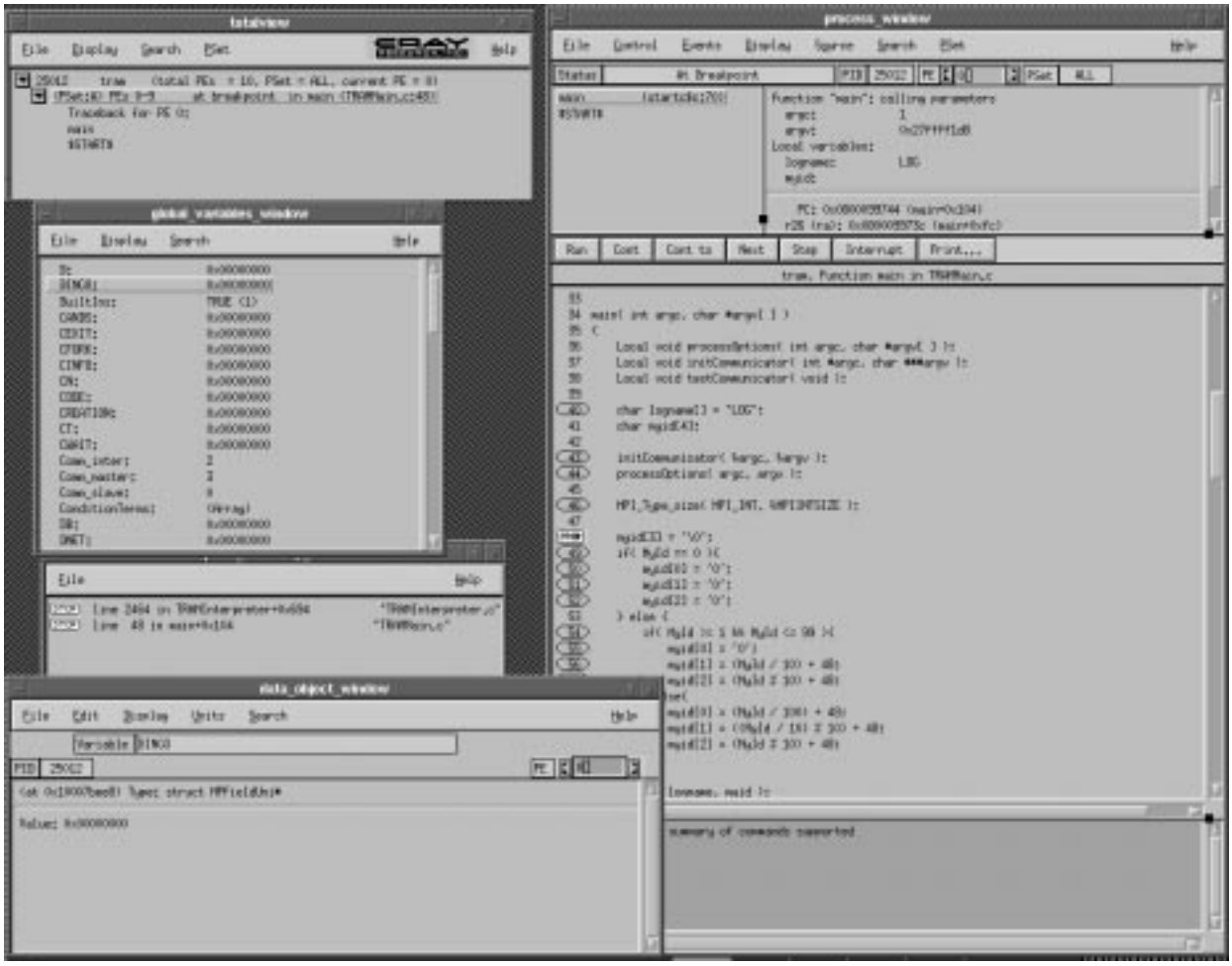

図 A.1: Cray TotalView

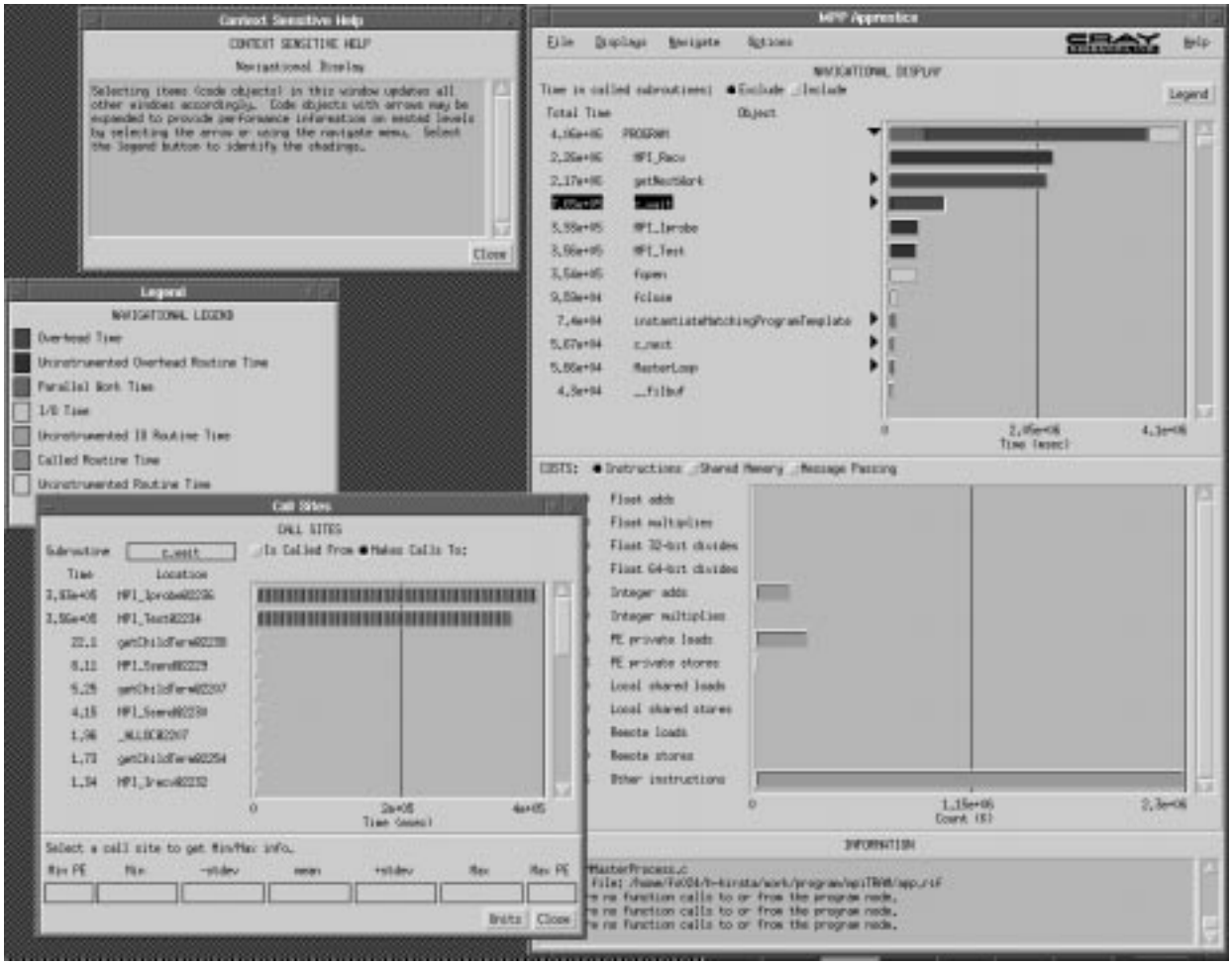

図 A.2: Apprentice

第 B 章

ベンチマークプログラム

B.1 Fibonacch 数列

B.1.1 逐次版

```
prim: on .
sorts: .
order: .
ops: fib : Nat -> NzNat .
vars: X Y : NzNat N : Nat .
rules: fib(0) -> 0
       fib(1) -> 1
       fib(N) -> add(fib(sub(N, 1)), fib(sub(N, 2)))
                 if gt(N, 1) = true.
```

B.1.2 並列版

```
prim: on .
sorts: .
order: .
ops: fib : Nat -> NzNat
     pfib : Nat -> NzNat
     padd : Nat Nat -> NzNat { strat:({1 2} 0) }  .
vars: X Y : NzNat N : Nat .
rules: fib(0) -> 0
       fib(1) -> 1
       fib(N) -> add(fib(sub(N, 1)), fib(sub(N, 2)))
                 if gt(N, 1) = true
       pfib(N) -> padd(pfib(sub(N, 1)), pfib(sub(N, 2)))
                 if gt(N, 24) = true
       pfib(N) -> add(fib(sub(N, 1)), fib(sub(N, 2)))
                 if gt(N, 1) = true & lt(N, 25) = true
```

```
padd(X, Y) -> add(X, Y).
```