並列計算機 PIE64 における Committed-choice 型言語 Fleng の負荷分散手法

中田秀基[†] 村上 聡[††] 荒木拓也[†††]
小池汎平[†] 田中英彦[†††]

本論文では，並列推論エンジン PIE64 上での Committed-choice 型言語 Fleng の実装に関して，負荷分散手法に重点をおいて論じる．負荷分散の要点は，データ参照の局所性とプログラムの並列性の抽出である．この両者はトレードオフの関係にあり，しかもトレードオフの最適点がプログラムの実行状況に応じて動的に変動する．このため最適な実行を行うためには，動的に両者の関係を調整するような制御を行わなければならない．我々は，すでにコンパイラによる静的な負荷分割と実行時の動的な制御を組み合わせる手法を提案している[1]．この手法は静的な負荷分割の情報を動的に用いることでつねに最適な実行を行うものである．本稿ではこの手法の実装と，静的な負荷分割の具体的な手法について報告する．静的な負荷分割は，プログラム中のデータとプロセスの間の依存関係をグラフとして表現し，グラフ分割することで行う．実行時には，計算機の動的状態を参照し，静的な負荷分割に基づいてプロセスの配置を行う．実機にこの手法を実装し，実行時の環境を変化させて評価した．この結果，環境によらず実行が高速化され，本手法の有効性が確認された．

A Load Distribution Method for Committed-choice Language Fleng on Parallel Inference Engine PIE64

HIDEMOTO NAKADA,[†] SATOSHI MURAKAMI,[††] TAKUYA ARAKI,[†††]
HANPEI KOIKE[†] and HIDEHIKO TANAKA[†††]

In this paper, we show an implementation of committed-choice language Fleng system on Parallel Inference Engine PIE64 focusing on a load distribution method. The points of load distribution are data-reference locality and execution parallelism. It is impossible to achieve both of them simultaneously, therefore keeping balance between them is the key issue. It is difficult because the optimal balance point varies depending on the execution environment. We already proposed a method which handles this issue. In our method, static load partitioning and dynamic load distribution compensate each other[1]. This paper gives an implementation of the method and a new method of static load partitioning. Our compiler partitions loads statically by dividing data-dependency graph, and run-time system locates the partitioned loads according to the runtime situation. We implemented our method on the actual machine and it proved to be effective.

1. はじめに

負荷分散問題，すなわちプロセスやデータをどのプロセッサに配置するかは，並列計算のメイントピックの1つである．負荷分散には以下の3つが要求される．

- メモリアクセスの局所化
- 並列性の抽出
- 負荷の均等化

この問題に関しては，従来さまざまなアプローチが試みられてきた．古くは粗粒度並列に対して，タスクグラフの分割埋込み問題として解決がはかられてきた．また，データ並列性に基づく定型的なアプリケーションでは，プロセスをデータの配置に従って配置する方法が一般的である．データ配置の手法としては，コンパイル時に自動的に配置を行う方法や，プログラマが与えたプラグマに従って配置する方法などが行われている．

しかし，知識処理に代表される非定型的アプリケー

† 電子技術総合研究所
Electrotechnical Laboratory
†† NTT データ通信株式会社
NTT Data Communications Systems Corporation
††† 東京大学工学部
Faculty of Engineering, The University of Tokyo

ションを細粒度で実行するためには，これらの方法を採用することはできない．これは，細粒度であるため，タスクグラフが巨大になり扱えないこと，プロセスが動的に生成されるため，プログラムの挙動を静的に推定することが難しく，コンパイル時にプロセス，データを静的に配置することができないためである．

代表的な Committed-choice 型言語の 1 つである KL1 の並列推論マシン PIM 上での実装では，負荷分散はプログラマが pragma で指定することで行われる．これは，ユーザに大きな負担となっている．

我々は並列推論エンジン PIE64[2] 上に自動的な負荷分散を実現する Committed-choice 型言語 Fleng の処理系を実装した．PIE64 は，負荷の均等化を支援するハードウェアを持つ．処理系はメモリアクセスの局所化と並列性の抽出を両立させるために，コンパイラによる静的な負荷分割と実行時カーネルによる動的な制御を併用する[1]．

Fleng のコンパイル時の静的負荷分割に関しては，プロファイリングを用いた手法も提案されている[3]．この手法は，プロファイリングを行った結果を ATMS (Assumption based Truth Maintenance System) に似た機構で多重世界的に管理する．この機構は，ある負荷分割を仮説として与えるとその負荷分割を行った場合のプロセス，データの配置を返す．この機構にさまざまな負荷分割を仮説として与えることで，全解探索的に最適な負荷分割を探索する．この手法では，良好な負荷分割が得られるが，プロファイルの実行に時間がかかるうえ，プロファイラの出力の解析にも膨大な時間がかかるため，負荷分割の手法として実用には適していない．

本稿では，プログラムのデータ依存関係解析に基づいてデータとプロセスを配置することで，適切な静的負荷分割を行う手法を述べる．また，文献 1) で述べた静的負荷分割と動的負荷分散の組合せによる手法を実機に実装し，評価を行う．

本稿の構成は以下のとおりである．2 章では対象となる Committed-choice 型言語 Fleng と並列推論エンジン PIE64 について簡単に述べ，PIE64 上の Fleng 処理系の概要を説明する．3 章では PIE64 上での Fleng の負荷分散手法について述べ，4 章では静的負荷分割の手法を詳説する．5 章で実機での評価を行う．

2. Committed-choice 型言語 Fleng と並列推論エンジン PIE64

2.1 Committed-choice 型言語 Fleng

Fleng[4] は並列論理型言語の子孫である Committed-choice 型言語の 1 つである．Fleng の実行は，ゴールと呼ばれる単位をクローズと呼ばれる書換えルールに従って書き換えることで行われる．同名のクローズの集合を述語と呼ぶ．クローズはヘッド部とボディー部からなり，ヘッド部にマッチしたゴールをボディー部のゴールへと書き換えるルールである．代表的な Committed-choice 型言語である KL1 と Fleng との最大の相違点は，Fleng にはガードゴールが存在せず，ヘッドのパターンのみでガードを構成する点である．

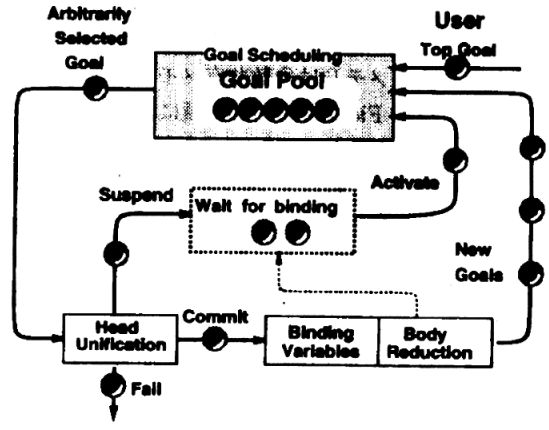


図 1 Fleng の実行モデル
Fig. 1 Fleng execution model.

Fleng の実行モデルを図 1 に示す．Fleng の実行は，ゴールプールにある複数のゴールから，書換えの対象となるゴールを任意に選択し，それを書き換えることで行われる．すべてのゴール実行は最終的には，システム述語と呼ばれる言語システムで用意した述語の実行に帰着する．システム述語にはユニフィケーションや，基本的な算術演算などがある．

1 つのゴールをリダクションした結果生じるゴールをサブゴールと呼ぶ．サブゴールがシステム述語である場合には，ただちに実行することが可能なので，性能の観点から，現在の Fleng 処理系ではその場で実行を行っている．

2.2 並列推論エンジン PIE64

PIE64 は，Fleng の実行を念頭において開発された並列計算機である[5]．PIE64 は 64 個の IU (Inference Unit) と呼ぶ要素プロセッサを持つ．各要素プロセッサは 2 つの 3 段のクロスバネットワークで相互に結合している．

要素プロセッサは，3 種類のプロセッサを持つ．Fleng の実行を行う専用のプロセッサ UNIRED (Unifier/Reducer)，通信/同期を専門に行う NIP (Network Interface Processor)，並列計算にともな

う雑多な仕事を請け負うMP（Management Processor）である．これらのプロセッサは，専用バスで密に結合している．メモリは分散共有メモリである．

UNIREDは，Flengの実行を念頭においたタグアーキテクチャと，命令セットを持つ．メモリアクセスのレイテンシ隠蔽のために1つのパイプラインを4つのコンテクストが共有しており，1クロックごとのマルチコンテクスト実行を行っている．MPには，汎用プロセッサであるSPARCを使用している．MPが計算用のプロセッサ以外に独立して存在することで，PIE64では計算速度を低下させることなく動的な制御を行うことができる．

PIE64のネットワークは自動負荷分散機能を持っている．各要素プロセッサがネットワークに自分の負荷情報を流し，各クロスバスイッチが最小の負荷を選択してフォワードすることによって，各要素プロセッサは最も軽い負荷を持つ要素プロセッサとその負荷値を知ることができる．自動負荷分散機能によって負荷が均等に分配されるので，最小負荷の要素プロセッサの負荷値からPIE64全体の負荷が類推できる．

3. PIE64の負荷分散

3.1 負荷分散の方針

本節では文献1)に従って，負荷分散の方針を概説する．

負荷分散の要件は以下の3つである．

- メモリアクセスの局所化
  プロセスとそれが必要とする変数とデータはなるべく同じプロセッサに配置する．
- 並列性の抽出
  なるべく多くのプロセッサにプロセスを割り当てる．
- 負荷の均等化
  負荷をなるべく均等に分配する．

PIE64ではハードウェアで負荷の均等化をサポートしているため，メモリアクセスの局所化と並列性の抽出がコンパイラシステムと実行時システムの役割となる．

一般にメモリアクセスの局所化と並列性の抽出はトレードオフの関係にある．このトレードオフのポイントはその時点での計算の並列性に応じて動的に変化する．負荷が大きいとき，すなわち計算の並列度が十分大きい場合には，あらたに並列性を抽出する必要はなく，メモリアクセスの局所化に専念できる．負荷が小さいとき，すなわち計算の並列度が十分にない場合には，多少メモリアクセスの局所性が失われても，並列度を抽出する必要がある．高速な実行を行うには，このように実行時の負荷に応じて動的にトレードオフのポイントを変化させる必要がある．

これを実現するために，PIE64では負荷分散を以下の3段階によって行う．

(1) **コンパイラによる静的負荷分割**
並列性を最大に抽出し，並列性を抑制しない限りにおいてメモリアクセスの局所性を向上させる．具体的には，データ依存関係があるために実際には並列に実行できないゴール，データを1つのグループとし，並列実行できる可能性のあるゴールは異なるグループとする．

(2) **ランタイム・カーネルによる動的負荷分割**
実行時の負荷状態に応じて，(1)で分割したグループを他のIUに投げるかどうかを決定する．低負荷時には，静的負荷分割の結果に従ってそれぞれのグループを別のIUに割り当てる．高負荷時には負荷を分割する必要はないので，すべてのグループを自分のIUに割り当てる．

(3) **相互結合網による自動負荷平滑**
(2)で他のIUに割り当てると決定したゴールのグループを，実際にどのIUに割り当てるかを決定する．このときPIE64の相互結合網の機能を用い，最小負荷のIUに割り当てる．

3.2 負荷分散手法の実現

前節で述べた負荷分散手法を実現するために，Flengの処理系を実装した．PIE64のFleng処理系は，実行時処理系とコンパイラ系との2つからなる．

3.2.1 複数のコードによる制御

ランタイム・カーネルによる動的負荷分割を実現するための手法として複数のコードによる制御を導入した．これは，低負荷時用に静的負荷分割を行ったコードと，高負荷時用に負荷分散を抑制するコードとを用意し，これらを実行時に動的に切り替えることで制御を行うものである．切替えの基準となる負荷の目安としては，各プロセッサのゴールプールに存在するゴールの数を用いる．PIE64では動的負荷平滑機能により，各プロセッサのゴールの数は平滑化されるため，これをPIE64全体の負荷の指標として用いることができる．この手法では比較的低いオーバヘッドで，動的な制御を実現できる．

3.2.2 実行時処理系 piexf

PIE64上の実行時処理系piexfは，大きく分けて次の2種類のプログラムからなる．

- **MP上で動くランタイム・カーネル**
  ゴールの管理を行う．Flengの実行単位であるゴー

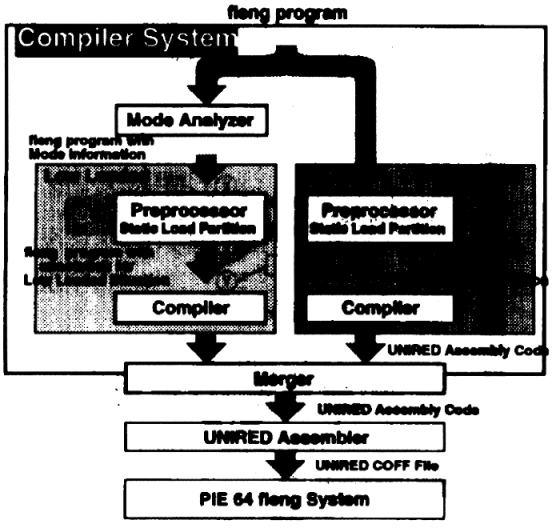


図 2 Fleng コンパイラシステム
Fig. 2 Fleng compiler system.

ルをスタック状のゴールプールに蓄え，UNIRED に送って実行させる．また，MP どうしでの通信を実現する．

- **UNIRED 上で動く Dispatcher ルーチン**
  Dispatcher は，ゴールの名前と引数の数から，対応する述語を特定し，その述語の定義であるコンパイルド・コードを呼び出す．このとき，前述した相互結合網の機能を用いて PIE64 全体の負荷状態を参照し，負荷状態に応じて適切なコードを実行する．これによって，複数コードによる制御が実現される．

3.2.3 Fleng コンパイラシステム

複数のコードを用いる制御を行うため，コンパイラシステムは複数のコードを内蔵するオブジェクトファイルを生成する．図 2 にコンパイラシステムの概要を示す．このコンパイラシステムの特徴は，2 種類のコードを生成するためにパスが 2 つに分かれている点である．もう 1 つの特徴は，静的負荷分割などの最適化がプリプロセッサで行われることである．プリプロセッサからコンパイラへの最適化情報の伝達は，プログラム中のアノテーションによって行われる．これによって，最適化のポリシーと実装とを明確に分離している．

低負荷用のコードに対しては，モード解析を行い，モードに基づいた負荷分割を行う．これに関しては次章で詳しく述べる．高負荷用のコードに対しては，すべてのデータ，ゴールを局所的に配置するように負荷分割を行う．このようにして作成された 2 種類のコードはアセンブリコードのレベルでマージされる．

4. 静的負荷分割

4.1 静的負荷分割戦略

Fleng での負荷分割の単位は，ゴール，変数，構造データの 3 つである．これらのそれぞれに対して，以下の 3 つ負荷分割戦略が考えられる．

- 実行中の IU に配置する．
- 最小負荷の IU に配置する．
- 指定した特定のゴール/変数/データと同じ IU に配置する．

本システムでは，これらの戦略をアノテーションで表現する．プリプロセッサは戦略を決定し，プログラムにアノテーションとして付加する．コンパイラはアノテーションを解釈しその戦略を実現するコードを出力する．

4.2 静的解析の概要

静的負荷分割の目的は，データ依存関係を持つため実際には並列に実行することのできないゴールを 1 つのグループにまとめることである．このときに，ゴール間のデータ依存関係の原因となるデータも同じグループに分類する．

このために，本システムは，まずプログラムのモードの解析を行う．このモードに基づいて，データ依存グラフを作成する．このグラフを分割することが負荷の静的分割を行うことになる．

4.3 モード解析

解析は，モード解析とデータ依存関係の解析の 2 つのフェイズからなる．モード解析では，述語の入出力モードを解析する．データ依存関係の解析では，モードの情報に基づいて，それぞれのクローズ間のデータ依存関係を解析する．

Committed-choice 型言語の入出力モード解析に関しては，すでに多くの研究がなされている[6]．しかし我々の目的には，従来の手法をそのまま用いることはできない．本解析で必要としているのはデータとゴールの間の依存関係であり，入出力そのものでなくそのデータがいつ必要になるか，いつ作成されるか，という情報が必要である．従来のモード解析で扱うモードは入力と出力のみで，本解析には十分ではない．また，従来の手法では述語のモードを大域的に解析する．本解析で必要となる情報は，局所的な依存関係のみであるから，コストの高い大域解析をする必要はない．

我々は本解析に必要十分な情報を得るために，新たなモードシステムを採用した．我々のモードシステムは入力，出力をそれぞれ強弱に分類する．述語のモードは各引数のモードのリストで表される（表 1 参照）．

表1 モードシステム
Table 1 Mode system.

記号	モード名	意味
++	強入力	そのゴールのリダクションに必要
+	弱入力	そのゴールのサブゴールのどこかで必要
−−	強出力	そのゴールのリダクションの結果生じたシステム述語であるサブゴールによって束縛される
−	弱出力	そのゴールのサブゴールのどこかで束縛される
?	unknown	

たとえば，下のプログラムを考えてみよう

```
foo(a, B):- B = b.
```

この述語は，実行のためには第一引数を必要とし，実行後には必ず第二引数を束縛する．したがってこの述語のモードは[++, --]となる．

弱入力モード，弱出力モードを完全に求めるためには，プログラムの全域解析が必要になるが，ここで必要なのは，強入力，強出力だけであるため，モード解析は局所的に，したがって高速に行うことができる．

強入出力のモード解析は以下のルールで行う．

(1) ヘッドでその引数が変数でなければ，その引数は強入力．
(2) ヘッドでその引数が変数で，その引数に対して値を束縛するシステム述語が存在すれば，その引数は強出力．
(3) 上記以外であれば unknown.

4.4 データ依存グラフの導出

次に，述語のモードからデータ依存関係を示す有向グラフを求める．あるゴールが強入力モードの変数を持つ場合，そのゴールがそのデータに依存することを意味し，あるゴールが強出力モードの変数を持つ場合，そのデータがそのゴールに依存することを意味する．

データ依存グラフは以下のステップで導出できる．

(1) 変数をデータノードとする．
(2) ボディゴールをゴールノードとする．
(3) ボディゴールが強入力のモードの変数を持つ場合にはその変数のデータノードから，ボディゴールのゴールノードへアークを作る．
(4) ボディゴールが強出力のモードの変数を持つ場合にはボディゴールのゴールノードから，その変数のデータノードへアークを作る．

4.5 データ依存関係グラフの分割

静的負荷分割はデータ依存グラフの分割に帰着する．分割したグラフを，それぞれプロセッサに割り当てる．データ依存グラフにおいて，ある2つのゴールノードを通るパスが存在する場合，それらのゴール間にはデータ依存関係があるため同時に実行できない．したがって負荷分割は，それぞれのパスをひとまとめにするように，グラフを分割することで実現できる．

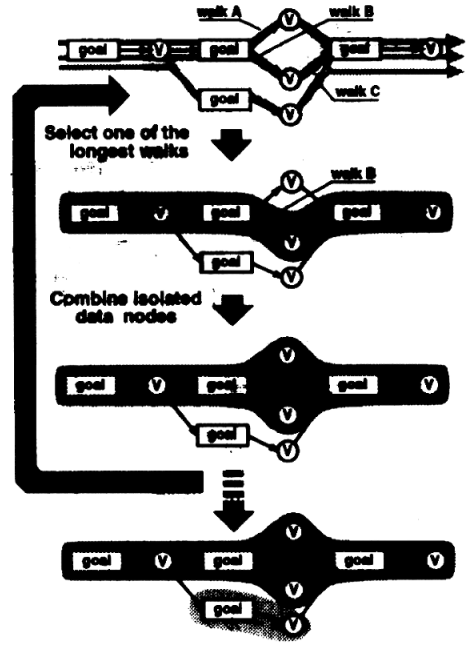


図3 データ依存グラフの分割
Fig. 3 Partitioning of a data-dependency graph.

このために，グラフの分割は以下のように行う：

(1) 任意の最長のパスを選ぶ．
(2) (1)で選択したパス上のゴール/データを取り除き1つのプロセッサに配置する．
(3) (2)によって孤立したデータノードがあれば，そのノードも取り除き(2)と同じプロセッサに配置する．
(4) (1)〜(3)をすべてのデータ/ゴールノードがなくなるまで繰り返す．

図3にこの様子を示す．このグラフは，3つの最長のパスを持っているが，ここではそのうちのBを選んでいる．次に孤立したデータノードをBと同じグループに入れ，これらをグラフから取り除く．この作業を繰り返すことで，最後の分割したグラフを得る．分割した結果，上のグループではデータのパスが複数になっているが，ここで並列になっているのはデータノードのみでゴールノードではないので問題はない．

4.6 アノテーション

4.1節でのべた戦略はアノテーションで表現される．

アノテーションは，変数，構造データ，ボディゴールに付加される．アノテーションは以下の書式で記述

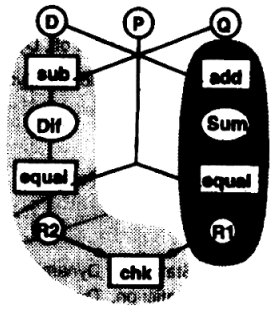


図4 データ依存関係と負荷の分割例
Fig. 4 An example of data-dependency graph partitioning.

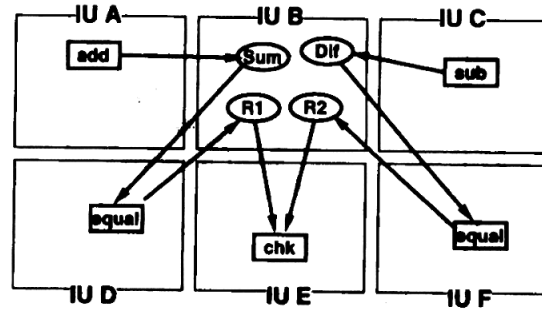


図5 データ/ゴールのナイーブな配置
Fig. 5 Naive allocation of data and goals.

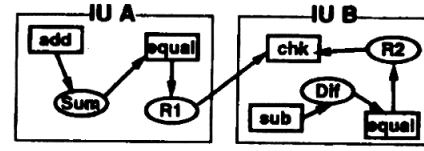


図6 データ/ゴールの最適配置
Fig. 6 Optimum allocation of data and goals.

される.

Item @[*annotation1, annotation2,....*]

負荷分割に用いるアノテーションは下の4つである.

- local: データ/ゴールをローカル IU に配置
- on(*label*): データがある IU を *label* に代入
- to(*label*): データ/ゴールを *label* が指す IU に配置
- any(*label*): データ/ゴールを最小負荷の IU に配置, その IU を *label* に代入

このうち on は, ヘッド部の変数に付加するものである.

4.7 負荷分割の例

上述の方法を例をあげて示す. 下のプログラムは, n-queen のプログラムの一部である.

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum),
    equal(Sum, P, R1),
    sub(Q, D, Dif),
    equal(Dif, P, R2),
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A).
```

ここで add, sub, equal, chk のモードはそれぞれ [++, ++, --], [++, ++, --], [++, ++, --], [++, ++, ?, ?, ?, ?, ?, ?, ?] とする.

図4はこのプログラム中のデータ依存グラフと, その分割を示している. この分割に従ってアノテーションを付加したのが下のプログラムである.

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum @ [any(1)]) @ [to(1)],
    equal(Sum, P, R1 @ [to(1)]) @ [to(1)],
    sub(Q, D, Dif @ [any(2)]) @ [to(2)],
    equal(Dif, P, R2 @ [to(2)]) @ [to(2)],
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A)
                @ [to(2)].
```

図5, 6 にゴール/データの配置を示す. 大きい四角は, IU を示し, ゴールとデータの間のアークはデータの参照を意味する. 図5 はこのアノテーションがない場合の配置を, 図6 はこのアノテーションの結果の最適な配置をそれぞれ示している. アノテーションなしの場合は, ゴールはなるべく分散し変数はローカルにとる戦術が選択されるため, ほとんどすべての参照がリモートになってしまう. これに対して最適配置では, リモートメモリ参照は1箇所のみとなる.

5. 負荷分散手法の評価

5.1 実験環境

実験は PIE64 のシステムを用いて行った.

測定条件は以下のとおりである.

- 使用 IU 台数: 2, 4, 8, 12, 16, 24, 32, 48, 64
- 動的負荷分割: ON, OFF
- 静的負荷分割: ON, OFF

使用 IU 台数を変化させることで, 相対的にプログラムの負荷量とプロセッサ台数の比率が変化する. すなわち, 台数が少ない場合には高負荷に, 多い場合には低負荷になる.

静的負荷分割が OFF の場合には, すべての負荷が別々のものとして分割される状態になる. 動的負荷分割が OFF の場合には, 実行時の負荷に対する動的追従を行わず, つねに低負荷時のコードで実行を行う. 静的負荷分割を行わず動的負荷分割のみを行うという場合は, 低負荷時にはすべての負荷が別のものとして

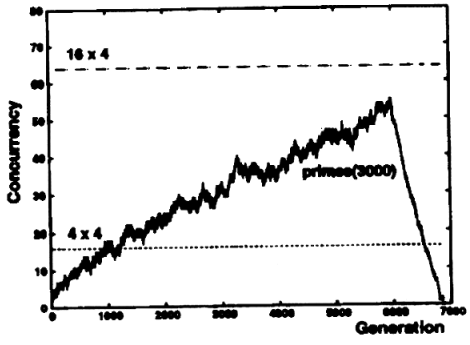


図7 primes の並列度
Fig. 7 Parallelism of 'primes'.

扱われるが，高負荷時には高負荷時用のコードが実行され，負荷分散は行われないことになる．

5.2 対象プログラム

対象としては，エラトステネスの篩を用いて3000までの素数を求めるプログラム "primes" を用いた．このプログラムは，それぞれの素数に対する篩を Fleng のパーペチュアルプロセスとして実現し，このプロセス間に自然数のストリームを通して，素数の数列を得るものである．

このプログラムは，一般に想像されるほどには並列性がない．パーペチュアルプロセス自体は素数の数だけ生成されるが，実際に稼働するのは小さい値の素数に対応する篩ばかりで，大きい値の素数に対応する篩はほとんど動かないからである．図7 に primes を3000まで求めた際の世代ごとの並列度を示す．ある世代のリダクションによって生じた実行可能なゴールは，次の世代でリダクションされると仮定している．また，サスペンドしたゴールは，ある世代のリダクションでサスペンドが解けると，次の世代で実行される．これは，1リダクションにかかる時間がすべて同じで計算機の台数が無限大であることを仮定することに相当する．

このグラフから分かるように，primes の並列度は4台の IU の16のコンテクストを埋めるには十分であるが，16台の64のコンテクストには届かず，64台の IU に対してはまったく不足している．したがって，4台近辺を高負荷時，16台近辺を中負荷時，それ以上を低負荷時と考えることができる．

5.3 実験結果

それぞれの場合について NIP 実行時間率，実行時間，サスペンド回数，相対的速度を測定した．

図8 に NIP の実行時間率を示す．これは，総実行時間に対する NIP が実行している時間の割合で，NIP は通信を司るためこの数値が通信の頻度を示すと考え

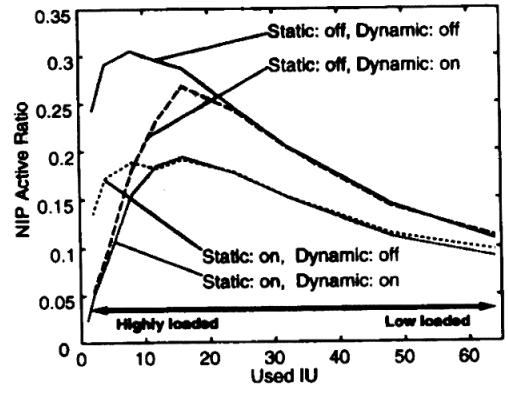


図8 NIP 実行時間率
Fig. 8 NIP active time ratio.

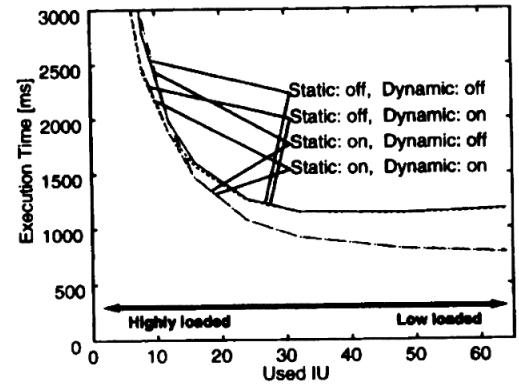


図9 実行時間
Fig. 9 Execution time.

てよい．一番上のラインが，動的負荷分割，静的負荷分割の双方とも行わない場合である．その下のラインが，動的負荷分割のみを行った場合である．高負荷時すなわち，台数が少ない状態では良い成績を示しているが，低負荷時すなわち，台数が多い状態では効果が見られない．

静的負荷分割のみを行った場合は，負荷値によらず NIP 実行時間率が減少している．高負荷時には動的負荷分割の方が効果的であるが，低負荷時には静的負荷分割の方が有効である．動的負荷分割，静的負荷分割の双方とも行った場合には，低負荷時には静的負荷分割のみの場合に準じ，高負荷時には動的負荷分割のみの場合に準じた特性を示している．

図9 にそれぞれの場合の実行時間を示す．静的負荷分割を行わない場合は，30台程度で飽和し，それ以上プロセッサ台数が増加しても実行時間が低下しない．これに対して静的負荷分割を行っている場合には，プロセッサ台数が増加し，プロセッサあたりの負荷が低

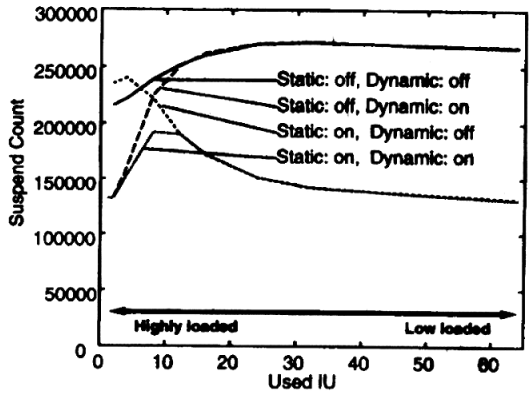


図 10 サスペンド回数
Fig. 10 Times of suspention.

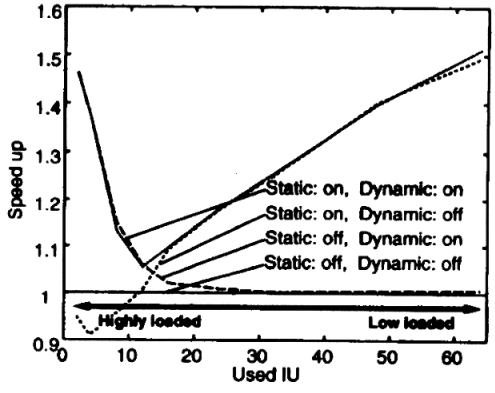


図 11 速度向上率
Fig. 11 Speedup by load partitioning.

下しても，実行時間が短縮されているのが分かる．

図 10 にサスペンド回数を示す．図 11 に動的負荷分割，静的負荷分割の双方とも行わない場合に対するそれぞれの場合の速度向上率を示す．動的，静的負荷分割を併用したものがすべての領域において高速化を実現しているのが分かる．サスペンドの回数も減少しているが，これは参照の局所化の副産物として，静的なスケジューリングをすることが可能になったことによる．

5.4 考 察

静的負荷分割は低負荷時の性能向上に貢献する．動的負荷分割は高負荷時の性能向上に貢献する．本実験によってこの両者を同時に用いることによって，すべての負荷領域において性能向上を図ることが可能であることが示された．

静的負荷分割を行わない場合には，実行時間の低下が 30 台程度で飽和する．これは，静的負荷分割を行わない場合は，データへのアクセスがリモートになる可能性が高いため，通信のオーバヘッドが生じ台数効果を打ち消しているためだと考えられる．NIP の実行時間率のデータがこれを裏づけている．

また，静的負荷分割のみを行ったケースでは，高並列部分で実行速度がなにも行わないものよりも遅くなっている．この原因は，静的負荷分割のオーバヘッドによるものであると考えられる．動的負荷分割を行わない場合は静的に負荷分割したゴールのグループを，最低負荷の IU に投げようとする．このとき，どこにも負荷の軽い IU がないと，結局他の IU には投げずにローカルに実行する．この場合は参照の局所性は確保されるが，この投げようとする試行のコストがオーバヘッドとなって，速度の低下を引き起こしている．動的負荷分割を併用すると，高負荷時にはこの試行を行わなくなるので，このオーバヘッドは解消される．

6. 結 論

本論文では，並列推論エンジン PIE64 上での Committed-choice 型言語 Fleng の実装に関して，負荷分散手法に重点をおいて述べた．

負荷分散手法としてコンパイラ，ランタイムカーネル，ハードウェアの 3 段階の負荷分散を用い，複数のコードによる制御によりこれを実現する実行時処理系の実装を行った．静的負荷分割にはデータ依存グラフを分割する手法を用いた．コンパイル時に複数の負荷状態に対してそれぞれ静的負荷分割を行ったコードを作成し，それらのコードを実行時負荷に応じて切り替えることで並列性とメモリ参照の局所性のバランスを，負荷にかかわらず適切に保つことが可能になる．このような手法を実現するコンパイラ系を実装し実機でテストした結果，すべての負荷領域で性能の向上を確認した．

本論文では Fleng と PIE64 に関して議論したが，負荷分散は並列言語一般に共通して見られる問題点である．動的な管理に関しては PIE64 に依存した部分があるため，計算機によっては実装が難しいと思われるが，静的な解析および最適化に関しては他の細粒度高並列言語にも適用が可能である．

参 考 文 献

1) 日高康雄，小池汎平，田中英彦：PIE64 の並列処理管理カーネルのアーキテクチャ，情報処理学会論文誌，Vol.33, No.3, pp. 338–348 (1992).
2) Shimada, K., Hidaka, Y., Tatemura, J., Koike, H. and Tanaka, H.: Studies on the Parallel Inference Engine PIE64, Annual Report of the Engineering Research Institute, Faculty of En-

gineering, University of Tokyo, Vol.51, pp.73–78 (1992).
3) 日高康雄，小池汎平，舘村純一，田中英彦：実行プロファイルに基づくコミッティッドチョイス型言語の静的負荷分散，情報処理学会論文誌，Vol.32, No.7, pp. 807–816 (1991).
4) Nilsson, M. and Tanaka, H.: Fleng Prolog – The Language Which Turns Supercomputers into Prolog Machines, *Proc. Japanese Logic Programming Conferernce*, pp.209–216 (1986).
5) Araki, T., Hidaka, Y., Nakada, H., Koike, H. and Tanaka, H.: System Integration of the Parallel Inference Engine PIE64, *Proc. FGCS'94 Workshop on Parallel Logic Programming*, pp.64–76 (1994).
6) Tick, E.: Practical Static Mode Analyses of Concurrent Logic Languages, *Proc. Parallel Architectures and Compilation Techniques (PACT '94)*, Elsevire Science Publishers (North-Holland), pp.205–214 (1994).

(平成7年12月5日受付)
(平成8年12月5日採録)

中田 秀基（正会員）

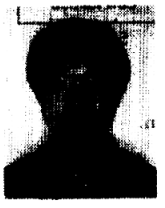
昭和42年生．平成2年年東京大学工学部精密機械工学科卒業．平成7年同大学院工学系研究科情報工学専攻博士課程修了．博士（工学）．同年電子技術総合研究所研究官，現在に至る．並列プログラミング言語，オブジェクト指向言語，分散計算システムに関する研究に従事．

村上 聡（正会員）

昭和45年生．平成5年東京大学工学部機械情報工学科卒業．平成7年同大学院情報工学専攻修士課程修了．同年NTTデータ通信（株）入社．情報共有システムの開発に従事．

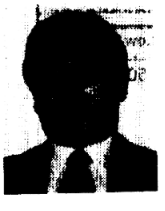
荒木 拓也（正会員）

昭和46年生．平成6年東京大学工学部電気工学科卒業．平成8年同大学院情報工学専攻修士課程修了．現在，同大学院博士課程に在学中．並列プログラミング言語の最適化の研究に従事．並列関数型言語，並列オブジェクト指向言語などに興味を持つ．



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6～8年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年東京大学工学部助教授，昭和62年教授．昭和53～54年ニューヨーク市立大学客員教授，現在に至る．並列処理，人工知能，分散処理，自然言語処理，メディア処理等の研究を行っている．'計算機アーキテクチャ' 'VLSIコンピュータI, II' 'ソフトウェア指向アーキテクチャ'（いずれも共著）'情報通信システム' 著．電子情報通信学会，人工知能学会，日本ソフトウェア科学会，IEEE，ACM各会員．