共有メモリ型並列計算機上への Fleng 処理系の実装及び評価

馬場 恒彦, 荒木 拓也, 田中 英彦
東京大学 工学系研究科
東京都文京区本郷 7–3–1

Fleng は Comitted-Choice 型言語の一つで, 細粒度高並列記号処理向けの言語である. プログラム中に内在する全ての並列性を抽出可能である特徴を持つ. 本稿では, 共有メモリ型並列計算機上に並列ランタイムシステムを実装し, 従来の逐次ランタイムシステムと比較した. その結果, queen で 4 スレッド時に 89% の高いスレッド利用率を実現し, 2.77 倍の速度向上が得られた. 一方で, primes では負荷分散が行なわれない問題点が明らかになった. この問題点の解決方法の一つとして, 負荷分散方式を改善した手法を実装し, 負荷分散方式改善のための予備実験を行ない, 検討を行なった.

Implementation and Evaluation of Fleng Run-time System on Shared Memory Parallel Computers

Tsunehiko BABA, Takuya ARAKI, Hidehiko TANAKA
School of Engineering, the University of Tokyo
7–3–1 Hongo, Bunkyo-Ku, Tokyo

Fleng is one of committed-choice languages and a fine-grained highly parallel symbolic processing language, which has a high ability to extract potential parallelism from any programs. In this paper, we have implemented Fleng run-time system on shared memory parallel computers and have evaluated its performance by comparing it with sequential run-time system. These evaluations make it clear that this run-time system has a high efficiency rate in N-Queen and Gauss, but does not work well in Primes.We proposed an improved load balancing method in order to solve this problem and we preliminarily examined this method.

1　はじめに

並列プログラミング言語は設計方針から二種類に分類される．一つは, 逐次処理向けに設計された言語をベースとし，並列計算機用に仕様の拡張と並列化を行なった言語であり，もう一つは最初から並列計算機での動作を考慮して，最初から設計・開発された言語である． 現在, 従来の言語との互換性などの点から実用ベースで主流となっている言語は前者であり, 代表例として HPF (High Performance Fortran) 等がある．しかし, 将来的により高並列な計算機への要求が高まることは十分に考えられることであり，前者に分類される言語は元来逐次処理向けの言語であったために，並列計算機向けに仕様拡張しても十分な並列度を抽出することが難しいため，後者に分類される言語に関する研究の必要性は高い．

こうした背景から, 本研究室では並列計算機に関するハードウェアとソフトウェア技術の研究を行なってきた．ソフトウェア技術として, Committed-Choice 型言語 Fleng を開発し [2], 様々な研究を行なってきた [6] [3]．これらの研究の中で Fleng の評価が行なわれてきているが, Fleng を高速実行する目的で設計された並列推論マシン PIE64 [1] 上で評価を行なっているものが多い．

そこで，本研究では共有メモリ型並列計算機 (マルチプロセッサワークステーション) 上に Fleng 処理系を実装し，いくつかのベンチマークによりその評価を行なった．また, その評価結果から実装したランタイムシステムの問題点について検討した． さらに，負荷分散方式の改善に関する予備実験を行ない, 考察を行なった．

以下, 第 2章では, Fleng と本研究で実装を行なった共有メモリ型並列計算機上の処理系の実装方針について説明する．続いて, 第 3章では実装処理系の評価及び検討を行なう．第4章では前章で明らかになった問題点を解決する改善手法に関する予備実験と評価について述べる．最後の第 5章はまとめである．

2　Fleng と Fleng 処理系

2.1　Committed-Choice 型言語 Fleng

Fleng は本研究室で設計された細粒度高並列記号処理向けのプログラミング言語であり, Committed-Choice 型言語，並列論理型言語の一つである． シンタックスは Prolog のものと良く似ているが，バックトラックを行なわないという点で，セマンティクスは大きく異なる．Committed-Choice 型言語には, 他に GHC, KL1 などがある．

Committed-Choice 型言語では, 単一変数代入とデータフロー同期の機構を用いることにより, ゴール (関数) 間に渡る並列性を含め，プログラムに内在するすべての並列性を抽出することが可能である． したがって，容易に大量の並列性を抽出することができる．

Fleng のプログラム例として,

foo(A,R) :- add(A,1,B), mul(B,2,R).

という $R = (A+1)*2$ を計算するプログラムを考える．

このプログラムでは，$A = 1$ として foo(1,R) を実行すると，add(1,1,B), mul(B,2,R) という 2 つのゴールに書き換えられ, それぞれを並列実行する．しかし，この場合, mul(B,2,R) は add(1,1,B) の結果によって B が決定されないと実行できないため，決定されるまで処理を中断 (サスペンド) し，決定後処理を再開する． このようにデータフロー同期を用いて全てのゴールを並列に実行することにより，本質的に逐次な部分を除いて，全ての部分を並列に実行することが可能である． この機構を実現するため，変数は単一代入であり, 書き換えることはできない．変数は値が決まっていないか決まっているかの 2 つの状態をもち, 一度決まってしまえば, その値が変わることは無い．

図 1に Fleng の実行モデルを示す．この図のように, Fleng の実行では,

1. ヘッドユニフィケーション
2. ボディーゴールのフォーク

を実行すべきゴールが無くなるまで繰り返す．

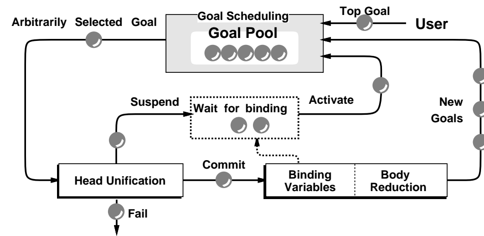

図 1: Fleng の実行モデル

2.2　Fleng 処理系

2.2.1　全体構成

Fleng プログラムが実行に至る迄の処理の流れは次の各段階から構成される．

1. マクロ展開プリプロセッサ:
   Fleng の記述性を向上させるために用いられているマクロを展開し, 純粋な Fleng の言語仕様に則ったプログラムへ変換を行なう.

2. 粒度制御プリプロセッサ:
   このプリプロセッサは効率向上を目的としており, マクロを展開した Fleng プログラムを入力とし, 粒度を変えた Fleng プログラムを出力する.

3. **Fleng** コンパイラ:
   マクロを展開 / 粒度制御を行なった後の Fleng プログラムを入力とし, C 言語プログラムにコンパイル, 出力する.

4. ランタイムシステム:
   Fleng プログラム もしくは, Fleng コンパイラにより出力された C 言語プログラムをコンパイルしたオブジェクトコードを読み込み, 各計算の実行順スケジューリングとガーベジコレクション (GC, Garbage Collection) を含む処理を行なう.

本稿では, 4 番目のランタイムシステムの共有メモリ型並列計算機上への実装を行なった. 以下に実装した処理系の実装方針について述べる.

2.2.2 ランタイムシステムの実装方針

共有メモリ並列計算機として, SunOS 5.X が動作可能なマルチプロセッサの WS(Work Station) を対象とした実装を行なった. これは, マルチプロセッサ WS が並列環境として比較的安価であり, 広く用いられていること, また他の言語との比較が容易にできるためである.

本実装理系の実装方針の概要について従来逐次型処理系との違いから以下に示す.

Multithread 化 対象 OS とした SunOS 5.X はスレッドライブラリを採用している. このライブラリを用い, スレッドの生成・切替を行ない, マルチスレッドによる並列実行を実現する. また, 本処理系では, 各プロセッサーに LWP (Light Weight Process) を割り付け, 各 LWP 毎に各スレッドをバインドすることにより, 各スレッドが異なるプロセッサに割り付けるようにし, スレッドの効率的実行を行なう.

メモリ空間は全てのスレッドで共有している. このため, GC は一括して全スレッド同時に行なわれる. また, マルチスレッド化により, ゴールスタックから複数のスレッドが同時にゴールをプッシュ・ポップするようになった. しかし, 上述のようにメモリ空間を共有してるため, ゴールスタックも共有されており, 正当性を保つためには, アクセス時に排他制御が必要となる. さらに, ゴールスタックのアクセス以外にも, 変数の高速, メモリアロケーション等にも排他制御が必要となる. こうした排他制御も P スレッドライブラリを用いて, 実現する.

負荷分散方式 共有メモリ型並列計算機向けの負荷分散の手法として, 各スレッド毎のゴールキューと負荷分散用のゴールキューの二種類を用いる手法を採用した [5].

LGS (Local Goal Stack) は各スレッドごとに設けられた有限の大きさのゴールスタックで, GGS (Grobal Goal Stack) は全スレッドによって共有する負荷分散用のゴールスタックである. LGS と GGS は以下のような動作をする. (図 2)

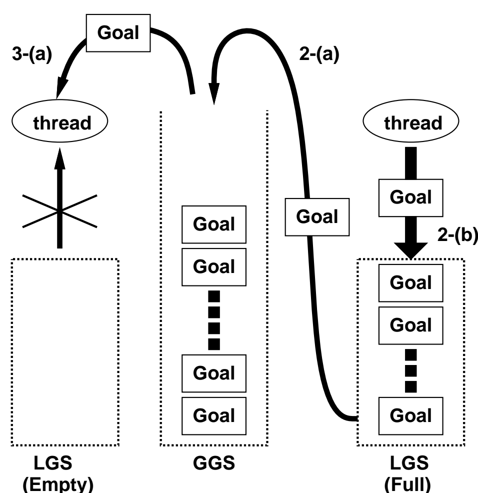

図 2: 負荷分散方式 (LGS の動作)

1. アクティベイトしたゴールのプッシュ
   (a) GGS にゴールをプッシュする.

2. リダクションしたゴールのプッシュ
   (a) LGS に空きがある場合はそのまま新たなゴールをプッシュする.
   (b) LGS が一杯の時は, LGS の底のゴールを一つ GGS にプッシュし, 空きを作ってから, LGS に新たなゴールをプッシュする.

3. ゴールのポップ

(a) LGS にゴールが入っている場合は LGS からポップする.

(b) LGS が空の場合は GGS からポップする.

他のスレッドに並列実行させるのではなく, 自分で逐次実行する場合には, 自分がリダクションしたゴールを LGS にプッシュすれば良い. これにより, この負荷分散方式は次のような利点を持つ.

- GGS はスレッド間で共有されているため, アクセスに対する排他制御が必要だが, LGS はスレッド固有のものなので, 他のスレッドによるアクセスが起こらず, 排他制御の必要がない. よって, GGS へのアクセスが減れば, 排他制御によるオーバーヘッドが減少する.
- LGS は有限の大きさなので GGS に比べて, 構造が簡単でアクセスが高速である.

3 実装処理系の性能評価

3.1 評価条件

実装したランタイムシステムを用いて速度測定を行なった. また, 実装した処理系の性能評価を行なうため, 従来の逐次型 Fleng ランタイムシステムでの実行速度との比較を行なった.

測定環境は以下の通りである.

Architecture :
Sun Enterprise 3000(4 CPU 構成).

Software :
SunOS 5.5.1 (OS),
Pthread 1.13 (Library),
gcc 2.7.2 (C Compiler)

また, 評価プログラムとしては記号処理と数値演算の二種類から選び, 前者として primes, queen を後者として gauss の計 3 つのプログラムを用いた. それぞれのプログラムは以下の通りである.

primes: 「エラトステネスのふるい」の手法を用いて素数列を用いるプログラムである. N = 10000 までの素数列を求める. プログラムの並列性は低い.

queen: N-Queen 問題の解を全解探索するプログラムである. 11 × 11 マスの時の解を求める. プログラムの並列性が高い.

gauss: Gauss 法による連立 1 次方程式の解法を行なうプログラムである. ピボット (pivot) 選択を行なわない方法での解法を用いる. 単一代入変数によるオーバーヘッドを削減するために配列要素の書換えは破壊代入により実現している [4]. 80 × 80 の行列での解を求める. プログラムの並列性は中程度である.

表 1: primes の実行速度 (N = 10000)

スレッド数	1	2	4
逐次	0.67(s) (1.82)	–	–
本実装 (並列)	1.22(s) (1.00)	1.23(s) (1.01)	1.24(s) (1.02)

表 2: queen (N = 11) の実行速度

スレッド数	1	2	4
逐次	3.43(s) (1.38)	–	–
本実装 (並列)	4.74(s) (1.00)	2.82(s) (1.68)	1.72(s) (2.76)

表 3: gauss (N = 80) の実行速度

スレッド数	1	2	4
逐次	2.59(s) (1.30)	–	–
本実装 (並列)	3.36(s) (1.00)	2.06(s) (1.63)	1.25(s) (2.69)

3.2 結果及び考察

それぞれのプログラムについて測定した結果を表 1, 表 2, 表 3に示す. 各測定は 5 回ずつ実行した場合の平均の値である. また, 括弧内の数値は実装ランタイムシステムにおける 1 スレッド時の速度を 1.00 として正規化した速度比である.

まず, 逐次ランタイムシステムとの速度を比較してみる. 1 スレッドの実行時を見てみると, 各プログラムとも逐次ランタイムシステムの実行速度より遅く, 最大で 55% の速度 (primes) しか実行速度が出ていない. これは, 1 スレッドでもあるにも関わらず, 排他制御のための機構を実行しているためである.

次に, 並列性が高い評価プログラムに関して考える. 記号処理向けの queen では, 2 スレッド時には 1.68 倍, 4 スレッド時には 2.76 倍 の速度向上が, 一方, 数値演算向けの gauss では, 2 スレッド時には

表 4: 実装処理系における N-Queen 問題 (N = 11) での実行時間

スレッド数	実行時間	$\sum ut_i$	$\sum st_i$	$\sum(ut_i + st_i)$	実効スレッド数
1	4.74(s)	4.69(s)	0.04(s)	4.73(s)	1.00
2	2.82(s)	5.29(s)	0.19(s)	5.48(s)	1.94
4	1.72(s)	5.57(s)	0.56(s)	6.13(s)	3.56

1.63 倍, 4 スレッド時には 2.67 倍 の速度向上が, 見られており, 複数 CPU を用いることでの速度向上を確認することができた.

また, スレッドが効率的に動作しているかについて, 評価プログラム queen の実行時間とスレッド i が使用した usertime(ut_i), systemtime(st_i) に着目して考える (表 4). 実行時間は実際に経過した時間であり, スレッド毎の usertime, systemtime は CPU が消費した時間である. 実効スレッド数は, 次の計算式で算出される.

$$\sum(ut_i + st_i)/\text{実行時間}$$

この式から, 2 スレッド時に 97%, 4 スレッド時に 89% のスレッドが稼働しており, 高いスレッド利用率を実現していることがわかる. 4 スレッド時に稼働率が減少している理由としては, Network 接続された WS を用い速度の計測を行なっているため, ランタイム以外のタスクが実行されている影響が考えられる. 測定時にはその他のタスクをできるだけ実行しないように注意したが, 完全に取り除くことはできないために各スレッドが全 CPU を占有することができていないためである.

一方, 評価プログラム primes においては, CPU 数が増加しても速度向上が見られていない. これは primes の並列性が低いために 1 つのスレッドの LGS の内部にしかゴールが存在せず, 結果として 1 スレッドしか動作していないためである. 並列性の低いプログラムを並列実行するために新たな負荷分散方式を検討が必要である.

4 負荷分散方式の改善に関する予備実験

4.1 改善方式の検討

並列度の小さなプログラムにおいても並列動作し高速化する負荷分散方式の改善手法について考える. 具体的には, 以下の方式が考えられる.

1. LGS のサイズを実行時に最適値に変更する方法

   LGS の大きさが最適であれば, 負荷分散の失敗による悪化は最小限に留めることが可能である. しかし, プログラムの種類の多様性やプログラムのどの段階を実行しているかを知ることは難しく, 実現手法は複雑なものとなる.

2. 自分自身の LGS と GGS が空の場合に, 他のスレッドの LGS から直接ゴールを取る方法.

   この手法では

   - ゴールを取り出す対象となるスレッドの決定方法.
   - そのスレッドの LGS 中から取り出すゴールの決定方法.

   といった点について, 考察する必要がある.
   また, 欠点としては同時に複数のスレッドが同一の LGS をアクセスする可能性があり, LGS のアクセスに対する排他制御が必要なことである. これにより自分自身の LGS に排他制御を行なうことになり, オーバーヘッドが生じてしまう.

3. ゴールをプッシュする歳に, GGS が空なために停止しているスレッドが存在した場合に GGS にプッシュする方法.

   この手法では, GGS の状態を監視することが必要となるが, 厳密な排他制御を行なうと大きなオーバーヘッドとなるため, 厳密なロックを行なわない flag を生成することによって回避を行うことができる.

上述の手法のうち比較的容易に実装可能な 3 番目の手法について, 実装を行なった. 新しく実装した負荷分散方式は具体的には以下の通りである.

- flag による LGS 状態の監視
  LGS の状態を表す flag を用い, スレッドの状態を監視する. この flag は全スレッドから参照される. LGS を 厳密に監視するためには, 排他制御が必要となるが, オーバーヘッドが大

きくなり, 問題となる. そこで, オーバーヘッドを削減するため flag は排他制御しない.

- リダクションしたゴールのプッシュ
プッシュする際に, flag の確認を行なう. 停止しているスレッドが存在するならば, ゴールを GGS にプッシュする.

- ゴールのポップ
LGS が空である場合には flag の値を書き換え, 各スレッドの LGS 状態を反映する.

4.2 予備実験と評価

前節で述べた負荷分散方式を実装し, 予備的な実験と評価を行なった. 評価条件は第3章と同様である. 結果を表 5に示す.

元のスケジューリング手法と比較すると, 4 スレッド時に primes で 1.13 倍の向上が見られ, queen においては変らない速度と速度向上が得られた. しかし, 一方で gauss では 2 スレッドでは 10%, 4 スレッドでは 40% の速度低下が見られ, 4 スレッド時には 2 スレッド時よりも実行時間が増大している.

その原因として system time の顕著な増大が見られ (表 6), system time が user time よりも支配的になっていることがあげられる. sytem time が増大する理由についてはいくつかの要因が考えられ, 現在, 各要因に関してデータを測定し検討中である.

表 5: 新負荷分散方式での実行時間

スレッド数	1	2	4
primes	1.23(s) (1.00)	1.25(s) (0.98)	1.09(s) (1.13)
queen	4.66(s) (1.00)	2.76(s) (1.69)	1.68(s) (2.77)
gauss	3.40(s) (1.00)	2.53(s) (1.34)	2.80(s) (1.21)

表 6: gauss(N=80) の system time

スレッド数	1	2	4
従来手法	0.00(s)	0.22(s)	0.43(s)
予備実験	0.00(s)	0.53(s)	3.35(s)

5 まとめ

本研究では, Committed-Choice 型言語 Fleng 処理系を共有メモリ型並列計算機に実装手法について述べ, 実装した処理系を用いて評価を行なった.

高い並列性を有する評価プログラムにおいては, 負荷分散が効率良く行なわれ, 4 スレッド時にスレッド利用率 89% を実現し, 2.88 倍の速度向上を得た. しかし, 一方で並列度が小さな評価プログラムに関しては, 負荷分散が行なわれず速度向上が見られない問題点が見られた.

この問題点を解決するための新たな負荷分散手法を現在検討中である. その予備実験を行ないその考察と問題点について述べた.

今後の課題として, 予備実験でので問題点の解決方法の検討及び, その他の負荷分散手法に関する検討と各手法の比較・検討を行なっていくことがあげられる.

参考文献

[1] Takuya Araki, Yasuo Hidaka, Hidemoto Nakada, Hanpei Koike, Hidehiko Tanaka. *System integration of the paralle inference engine PIE64.* In Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994, Dec, 1994.

[2] Nillson,M. and Tanaka,H. *Fleng Prolog - The language which turns Supercomputers into prolog Machines, In Wada, E.(Ed.).* Logic Programming'86, LNCS264, Spriger-verlag, 1986.

[3] 荒木 拓也, 田中英彦. *Committed-Choice* 型言語 *Fleng* における静的粒度最適化. プログラミング研究会 96-PRO-8, pp.109 - 114, Aug, 1996.

[4] 大野 一. *Committed-Choice* 型言語 *Fleng* における配列の静的コピー除去. 学士学位論文, March, 1997.

[5] 研究代表者 田中英彦. 共有メモリマルチプロセッサ上の *FLENG* 処理系 大規模知識処理システムの研究, pp.249 - 269, §11, Febrary, 1991.

[6] 日高康雄, 小池汎平, 田中英彦. *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. 並列処理シンポジウム JSPP '95, pp.115 - 122, May, 1995.