プログラミング 10−3
(1996. 11. 1)

動的負荷分散機構を備えた分散lisp処理系の実装

[1]嶋田 一郎 [2]松井 俊浩 [3]中西正和

[1,3]慶應義塾大学 理工学研究科 計算機科学専攻
[2]電子技術総合研究所 通信知能研究室

本研究では，分散メモリ型高並列計算機 AP1000 上に，実行時に自動的に負荷分散を行なう分散 lisp システムを実装した．副作用のない関数型言語は，その局所性から分散環境に適応させやすい．本研究では関数型言語の代表である lisp を取り上げ，その分散化を試みた．知能ロボット研究などの人工知能の分野では，lisp によりアプリケーションを記述することによる利点が多い．近年，機能的に分散させることで効率の良い実行が実現できるアプリケーションや，大きなアプリケーションを実行させることの必要性などにより，ノード数が制限されない分散メモリ型並列計算機上の lisp システムが求められてきている．

本稿では，その中でも特に負荷分散法に着目し，分散 lisp における動的負荷分散手法に関する評価及び考察を行なった．

A Dynamic load balancing method on a Distributed lisp system.

[1]Ichiro SHIMADA [2]Toshihiro MATSUI [3]Masakazu NAKANISHI

[1,3]Department of Computer Science, Keio University
[2]Electrotechnical Laboratory

We implemented a distributed lisp system which support an automatic load balancing faculty on AP1000, loosely coupled parallel machine. A functional language without side effects can be easily distributed because of locality. Lisp are used for symbolic computations by a wide variety of large application in AI. On the other hand, dynamic load balancing techniques have proved to be the most critical part of an efficient implementation of various algorithms on large distributed computing systems.

In this paper, we discuss a dynamic load balancing method for distributed lisp system.

1　はじめに

分散システムにおいて，処理性能をあげるためには，効率の良い負荷分散手法が不可欠である．共有メモリ型並列計算機の場合ならばスケジューリングはタスクプールを設ける手法などの集中管理方式で構わないが，分散メモリ型の場合，拡張性を保つためには協調方式の負荷分散手法を考える必要がある．実行前に静的に負荷分散方針を決定してしまう方法は，再帰的定義を多く用いるlispの場合には実行前の静的解析による粒度決定が困難であるという理由から，汎用性に欠ける．また，負荷分散を記述するための構文を用意し，ユーザに負荷分散を任せるという方法も考えられるが，ユーザへの負担が大きくなるだけでなく，プログラムの汎用性を低下させる原因となるため，好ましくない．

本研究では，システムが実行時に自動的に負荷分散を行なうため，ユーザに負担がかからず，かつ実行時の負荷の偏りを最小限に抑えることが可能となる．本稿は，2章で分散lisp，3章で負荷分散について述べ，4章，5章で結果及びその評価を行ない，6章で結論および今後の予定を述べる．

2　分散lisp

lispは代表的な関数型言語として，古くから様々な分野で注目されてきた．知能ロボット研究の分野においては，lispによりアプリケーションを記述することで処理効率の向上が得られる場合が多い[8]．これは，lispのポインタやリスト操作機能は，面や稜線などの要素間のトポロジーの操作に都合がよいためである．現在，そのようなアプリケーションにはより強力な計算パワーが必要なため，高並列処理が不可欠となりつつあるが，実現されている並列lispのほとんどは，密結合型並列計算機上でのものであり，更なる高並列処理の実現には，拡張性に富んだ疎結合型並列計算機上で動作する分散lispが求められている．

2.1　分散関数型言語の並列性抽出

これまでいくつかの分散関数型言語が提案・実装されており，並列性の抽出方法として様々な方法がとられている．代表的な方法としては，データ並列によるものや，Schemeの継続点を利用したもの[4]などがある．

2.2　問題点

分散メモリ上での関数型言語処理系の実装例が少ないことは先に述べたが，その要因としては，関数型言語の特徴であるリスト構造は，システム全体で共通の大域アドレスをもたない分散環境において実現が難しいことや，lispなど副作用のある言語において，オブジェクトの同一性(identity)を保証することが困難であることなどがあげられる．

更に，もう1つの大きな要因として，静的に粒度設定を行なうことが困難であることがあげられる[1]．このため，粒度を考慮していない場合，負荷の偏りが生じシステム全体の処理効率を著しく低下させる可能性がある．このような場合，実行時に動的に負荷を分散させ，プロセッサ間の負荷の偏りを軽減させる必要がある．

2.3　本システムの構成

本システムは，実行時に自動的に負荷分散を行なう機能をもった分散lispシステムであり，実装は富士通研究所の高並列計算機AP1000上で行なった．lispの仕様はCommon Lispに準じており，規模は必要最小限のものに抑えた．

2.3.1　オブジェクトの一貫性の保証

lispの分散化の方針として非常に重要なオブジェクトの一貫性に関しては，シンボルの一貫性のみ保証した．シンボル以外のオブジェクトには一貫性がないため，逐次lispとのsemanticsのずれが生じてしまうが，実用上，シンボルの一貫性を保証すれば問題はないと考えられる．シンボルの一貫性の保証は，シンボルテーブルの管理を一括して担当するマスターを設け，他のノード(スレーブ)はシンボルの属性値が必要なときにマスターに問い合わせるようにすることで実現した．

マスターには輸出表を設け(図1)，シンボルの属性値と，現在どのスレーブに対して参照を輸出しているかという情報を管理している．一方，スレーブには輸入表を設け，一度マスターから得た情報をキャッシュしておくことで通信量を最小限に抑えた．またキャッシュアクセスの時間短縮のため，全てのシンボルについてマスターでのエントリとスレーブでのエントリが一致するようにした．この方法だと無駄な領域が存在してしまう可能性があるが，それに伴うキャッシュアクセス時間，及び通信データの増大を考えれば，妥当であると言える．

組み込み関数や，特別なシンボル(`NIL`など)に関しては，実行前にあらかじめマスターとスレーブの両方に登録しておくことにした．

2.3.2　並列性抽出

本システムでは，`make-task`構文を提供することにより，タスクを次々に生成していく，という方針で並列プロ

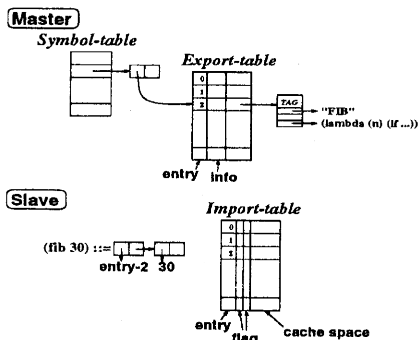

図 1: シンボルの管理

グラムを記述することになる．ユーザは make-task 構文を用いてタスクの生成を指定するだけで，負荷分散に関する指定をする必要はない．

本システムでは，各ノード毎にローカルな 2 種類のキューを持たせた (図 2)．実行可能キュー (以下 A-Queue) と，待機キュー (以下 W-Queue) であり，前者は実行可能なタスクのため，後者は生成したタスクの終了を待つタスクのためのキューである．

make-task 構文は引数に与えられた S 式を評価するようなタスクを生成し，A-Queue の末尾に加える (このときその時点での環境も一緒に持たせるため，本システムでのタスクは全て closure ということになる)．

```
(make-task <S-exp>)
  ⇒(<S-exp> <env>)
```

親のタスクは，自分の生成したタスクが確定するまで W-Queue に移される．そして親のタスクは，答えが全て揃った時点で W-Queue から取り出され，後処理 (fib の場合ならば+) をする．

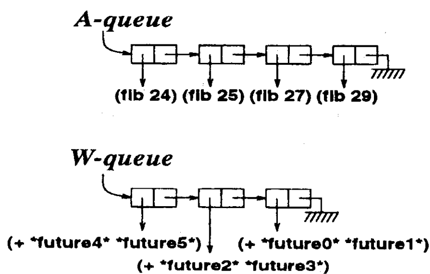

図 2: ローカルキュー

負荷分散を行なう際，タスク毎に持っている環境についても一緒に転送する，ということを前に述べたが，本システムでは alist によって環境を実現しているので，タスク転送の際に alist を全て転送している．この方法だと，セルの消費量が大きくなってしまうという欠点があるが，スレーブ間のメッセージ転送量を抑えるためにこのような方法をとった．

3 負荷分散

複数のプロセッサを並列に動作させ処理を行なう場合，あるプロセッサに負荷が集中するようでは，効率の良い処理は望めない．効率の良い並列処理に，**負荷分散**は不可欠である．

負荷分散には，静的な負荷分散と，動的な負荷分散がある [7]．前者は，規則的にタスクを生成するような処理においては威力を発揮するが，タスクの生成が事前に予期できない場合，後者の手法が望まれる．

3.1 分散環境における動的負荷分散

動的負荷分散では，実行時にシステムの負荷を均等にしようとするため，タスクの生成規則や粒度が事前に予測できない場合などにも対応できる．一方，実行時に負荷分散のための余分な処理が必要となるため，動的負荷分散を行なうためには，負荷分散にかかるオーバーヘッドをどれだけ抑えることができるかが重要である．

これまで，動的負荷分散について多くの手法が提案されている [2, 3, 5, 6]．本システムでは，その中の Gradient Model という負荷分散のモデルを利用し，それをいくつか改良した手法を用いている．Gradient Model を用いた理由としては，モデルが単純であるわりに効率の良い負荷分散ができることや，拡張性に優れていることがあげられる．Gradient Model は，1986 年頃提案されたモデルであり，その有用性はいくつもの論文で示されている．

動的負荷分散アルゴリズムは，どの状態のノードがシステム全体の負荷分散を促すかによって，次の 3 つに分類できる [7]．Gradient Model は，3 番目の相互駆動型である．

過大負荷駆動型 負荷がしきい値を上回ったノードが負荷を軽減しようとするものである．

過小負荷駆動型 負荷がしきい値を下回ったノードが負荷を要求するものである．

相互駆動型 過大，過小負荷駆動型の両方の性質を備えたものである [5, 6]．

上の 3 つの型に加え，更にアルゴリズムが適応型であるか否かで分類することもできる．適応型とは，システム全体の負荷の状態に適応し，動作を変えるものである．古い情報を保持しておく方法 [2, 6] や，しきい値を動的に

変える方法[5]などがある．

3.2 Gradient Model

Gradient Model(以下GM)[2]は，前述の適応型の性質を有し，相互結合網に依存しない，拡張性に優れているといった性質をもつ負荷分散のモデルである．GMは **Gradient Surface**(図3，以下GS)という概念でシステム全体をとらえ，この概念により負荷分散を実現するものである．

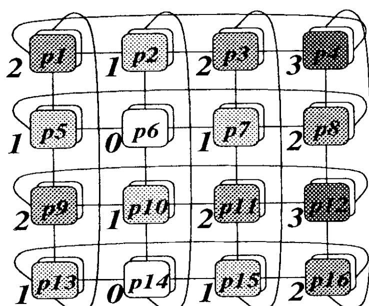

図 3: Gradient Surface

図3の各ノードの左下についている数字は，自分から最も近い過小負荷ノードまでの距離を表している．GMの実際のアルゴリズムでは，隣接ノードのみと情報交換を行ない，この数字の近似値を求めている．この数字はGSの高さを表しており，ちょうど水が高いところから低いところへと流れるように負荷が分散する．

3.3 負荷分散方針

Shivaratriらは，論文[7]において動的負荷分散の特徴を決定する4つの方針(転送方針，選択方針，配置方針，情報獲得方針)についてまとめている．以下，本システムで採用した負荷分散手法における4つの方針に関して述べる．

3.3.1 転送方針 (Transfer policy)

転送方針は，ある状態における負荷分散の妥当性を規定するものであり，しきい値を用いる方法が一般的である．本システムにおいてもその方法を用いた．状態としては，過小負荷状態(Idle)，中間状態(Neutral)，過大負荷状態(Abundant)の3状態をもつことにした．GMでは，自分のノードの状態が変化したときに隣接ノードに情報を送る，という方針をとっているが，実験の結果，しきい値の付近でタスク量が変化するような場合，無駄な情報が大量に流れてしまい，本システムにおいては効率が得られなかった．このような状況を避けるため，本システムでは状態が変化してもすぐには情報を転送しないようにした．4に例を示す．横軸はタイムスライスを表しており，負荷情報を転送するタイミングである．縦軸は，ノードの負荷状態を表している．

本システムでは，前述の通り状態を3つ用意し，それぞれをしきい値(*threshold*)で区切った．そして，それぞれのしきい値よりも負荷の重い方に転送リミット(*limit*)を設け，負荷の重い側(グラフ縦軸の正方向)への変化を隣接ノードへ通知するのはこの転送リミットを越えたときのみに制限した．負荷の軽い側(グラフ縦軸の負方向)への変化の通知は，しきい値を下回ったときに行なうため，細かい状態変化に伴う無駄な通信量を抑えることができた．

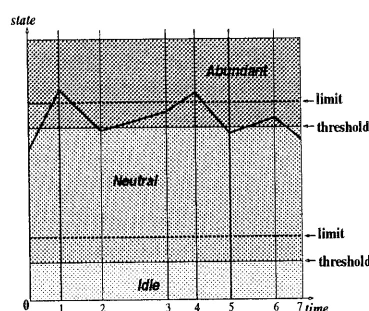

図 4: 転送方針

しかし，これだけではまだ通信量が多くなり過ぎてしまい，システムにかかるオーバーヘッドが大きくなってしまうため，効率の良い負荷分散は実現しない．そこで，しきい値を大きく上回って，しばらくその状態にいた場合にのみ，そのノードは過大負荷状態であるという判断をすることにした．

図4において，あるノードが太線で示したような状態変化をした場合，このノードは3度，負荷情報を回りに転送する．しかしこのノードは7度目のタイムスライスの後，最初の状態(中間状態)に戻っている．このため，3回の負荷情報の転送は無駄になってしまう．本システムではこのような現象を最小限に抑えるために，数回のタイムスライスの間，同じ状態にあり続けた場合にのみ負荷情報の転送を行なうことにした．

3.3.2 選択方針 (Selection policy)

負荷(タスク)を他のノードに転送する際，転送すべきタスクを，**選択方針**によって選出する．ローカルキューの先頭(末尾)からタスクを選ぶ方法や，最近生成されたタスクを選ぶ方法などがある．本システムでは，各ノードがローカルキューを持ち，その先頭から処理すべきタスクを選ぶというFIFO方式を用いた．

3.3.3 配置方針 (Location policy)

配置方針は，余剰タスクの転送先を決定する．順次全て，あるいは一部のノードに要求を出して選ぶ方法 (ポーリング) や，あらかじめ得られた情報を用いて選ぶ方法 [2] などがある．GM の場合は，得られた情報から判断して転送先を決定する．本システムでも，GM の方針に従い，得られた情報から判断するようにした．

3.3.4 情報獲得方針 (Information policy)

効率のよい負荷分散を行なうためには，まわり (自分以外のノード) の情報をできる限り正確に，かつ最小限のコストで得ることが重要である．正確さとコストのトレードオフを考慮し，いつ，どこから，どのような情報を集めるのかを規定するのが，**情報獲得方針**である．典型的なものとしては，必要になった時点で集める要求駆動型の方針をはじめ，周期的に集める方針 [7, 5] や，GM のように状態変化に応じて他のノードに知らせる方針 [2] などがある．

GM では，隣接ノード間での情報交換しか行なわないため，局所性に優れている．しかしながら，情報伝搬が遅れてしまう，タスク転送が集中してしまう，余分な負荷情報によるオーバーヘッドが大きくなってしまう，などといった問題点が論文 [3] で指摘されている．そこで，本システムでは，GM の情報獲得方針を改良して 2 つの工夫を行なった (図 5)．

- 過小負荷状態のノード間で情報交換を行ない続けても負荷分散の役には立たないため，情報転送先を他のノードに変更する (左図)．これにより，負荷分散には全く関係しない負荷情報の転送回数を下げることが可能となる．
- 隣接ノードの中で自分の情報を持っていない過大負荷状態のノードが存在した場合，情報転送先をそのノードに変更する (右図)．これにより，より早く過大負荷ノードからタスクを受けとることが可能となる．

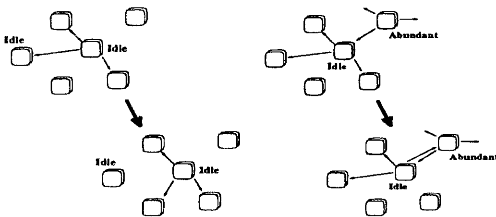

図 5: 本システムの情報獲得方針

4 実験結果

4.1 実験

実験は，fibonacci 関数を用いて行なった．図 6 に本システムでの fibonacci 関数の記述例を示す．

```
(defun fib (n)
  (if (< n 2)
      1
    (+ (make-task (fib (- n 1)))
       (make-task (fib (- n 2))))
  ))
```

図 6: `fibonacci` 関数

並列 lisp での fibonacci のプログラムを記述する場合，引数がある程度以下になったらタスクは生成せずにローカルに処理してしまう，という方法が普通である．しかし，本研究の目的は負荷分散にあり，fibonacci の実行速度をあげることにはないため，そのような方法はとらずに，最後までタスクを生成し続けている．

4.2 結果

図 7 に，`(fib 19)` の実験結果を示す．横軸が台数であり，縦軸が速度向上率を示している．`(fib 19)` では，タスクは 13529 個生成され，いずれの場合も各ノードでの処理量の差はそれほど大きくならず，均等に分散させることに成功した．

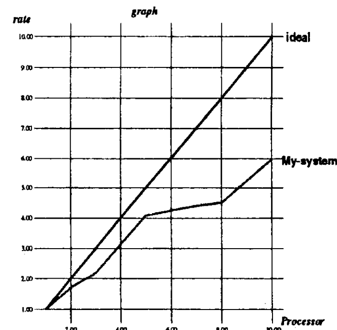

図 7: 台数効果

また，図 8 は，純粋な GM を用いた負荷分散方式と，本研究の負荷分散方式を用いた場合の，負荷情報の転送量の比較である．横軸に台数をとり，縦軸に負荷情報の転送量をとった．なおこのグラフは，`(fib 18)` の場合のデータである．これは，`(fib 19)` においては純粋な GM では通信量が多くなり過ぎて，AP1000 が用意しているメッセージバッファが溢れてしまい，実行不可能であったためである．

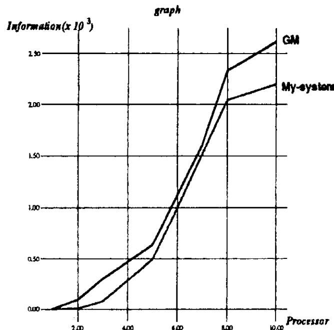

図 8: 情報通信量

5 性能評価及び考察

4 章の実験及びその結果から，純粋な GM に比べ，本研究での手法を用いた場合，負荷情報の通信量を減らすことができることがわかった．これは，3.3 で述べた転送方針の変更によるところが大きいと考えられる．それほど大きな差が見られないのは，情報獲得方針の変更により情報獲得先を変更することがあるため，純粋な GM の場合に比べて，積極的に情報を転送しようとする力が働くためであると考えられる．しかし，情報獲得方針の変更により，各ノードが実行中過小負荷である時間，即ちアイドル時間が減少することが確認された．

転送するタスクの選択方針に関しては，今回はキューの先頭から取り出すという FIFO 方式を採用したが，fibonacci 関数のようにタスクを生成していくような場合，LIFO 方式にすると FIFO 方式に比べて実行時のキューの最大長が短くなる．このため，転送方針のしきい値を FIFO 方式の場合よりも低く設定する必要がある．(`fib 19`) を計算する際，FIFO 方式と LIFO 方式の両方について実験してみたが，LIFO 方式の場合キューが伸びる速度が遅いため，負荷分散がうまくいかなかった．今後はこの点に関しても実験を重ねて詳しく調べていく予定である．

6 結論及び今後の予定

本稿では，高並列計算機 AP1000 上に，GM に基づく動的負荷分散機構を備えた分散 lisp システムを実装した．転送方針，情報獲得方針に関する工夫により，従来の GM と比べ良い負荷分散効率が得られたが，まだオーバーヘッドが大きく，プロセッサ数が増えたときの台数効果を得るためには，更なる効率改善が必要である．

今後は，動的負荷分散の処理にかかるオーバーヘッドの更なる削減を実現し，台数を増やしたときにも台数効果が得られるようにする予定である．また，副作用を入れたときの変化についても測定し，比較を行なう予定である．最終的には，実用的なアプリケーションを記述し，本システムで実行させる予定である．

参考文献

[1] Herber Kuchen, Rinus Plasmeijer, and Holger Stoltze. Efficient Distributed Memory Implementation of a Data Parallel Functional Language. *Parallel Architectures and Languages Europe: PARLE '94*, pp. 464–477, Jul. 1994.

[2] Frank C. H. Lin and Robert M. Keller. Gradient Model: A Demand-Driven Load Balancing Scheme. *IEEE Conference on Distributed Systems*, pp. 329–336, 1986.

[3] F. J. Muniz and E. J. Zaluska. Parallel Load-balancing: An Extension to the Gradient Model. *Parallel Computing Vol.21 No.1*, pp. 287–301, Jan. 1995.

[4] Christian Queinnec. Locality, Causality and Continuations. *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pp. 91–102, 1994.

[5] Thierry Le Sergent and Bernard Berthomieu. Balancing load under large and fast load changes in distributed computing systems - a case study. *Parallel Processing CONPAR '94(LNCS 854)*, pp. 854–865, 1994.

[6] Niranjan G. Shivaratri and Phillip Krueger. Two adaptive location policies for global scheduling algorithms. *IEEE Proc. 10th Int'l Conf. Distributed Computing Systems*, pp. 502–509, 1990.

[7] Niranjan G. Shivaratri, Phillip Krueger, and M. Singhal. Load Distributing in Locally Distributed Systems. *IEEE Computer Vol.25*, pp. 33–44, Dec. 1992.

[8] 松井俊浩, 関口智嗣. マルチスレッド並列 Euslisp の分割型メモリ管理手法. 情報処理学会プログラミング研究会研究報告, Vol. 95, No. 92, pp. 9–14, Sep. 1995.