北海道大学工学部研究報告
第167号（平成6年）

Bulletin of the Faculty of Engineering
Hokkaido University, No. 167 (1994)

分散制御型動的負荷分散における遺伝的操作の導入

棟朝　雅晴　　高井　昌彰　　佐藤　義治

（平成5年8月31日受理）

A Genetic Scheme for Distributed Dynamic Load Balancing

Masaharu MUNEMOTO, Yoshiaki TAKAI, and Yoshiharu SATO

(Received August 31, 1993)

Abstract

A distributed computing system (DCS) is a collection of autonomous computers which are loosely coupled via a communicating network. Efficiency of DCS depends on how even the processors' loads are distributed. A dynamic load balancing scheme distributes tasks while their execution.

We propose a distributed dynamic load balancing scheme which improves performance of a conventional sender-initiated algorithm by using genetic operators based on a genetic algorithm (GA). In our scheme, genetic operators are applied to a population, a multiset of strings, assigned to each processor in a DCS. A string in the population represents a set of processors to which requests for task migration are sent. Through empirical investigations, we show the effectiveness of our scheme compared with the sender-initiated algorithm from a view point of the mean response time of input tasks.

1．はじめに

分散計算システム(Distributed Computing Systems, DCS)とは，自律した計算機が比較的通信遅延の大きい通信ネットワークを通して多数結合されたシステムである[1]。DCSの性能の向上，すなわちタスクを起動してから終了するまでの平均応答時間を最小にするためには，各計算機における負荷（一般には，実行待ちのタスクの数で測られる）を平均化することが重要である。負荷分散アルゴリズムは，実行前にタスクの割当てを行なったり，実行中にタスクの転送を行なったりすることで負荷の平均化を行なう。負荷分散アルゴリズムを，静的負荷分散，動的負荷分散の2つの方式に大別することができる。静的負荷分散アルゴリズムでは，タスクを実行する前にその実行に要する時間や依存関係についての情報が得られており，それを用いて実行前にタスクを各計算機に割当てることで負荷の均衡をはかる。動的負荷分散アルゴリズムでは，各計算機の負荷情報を動的に観測し，それを用いて負荷の重い計算機から負荷の軽い計算機へタスクを転送することにより，負荷の平均化を行なう。動的負荷分散アルゴリズムはさらに集中制御型と分散制

情報図形科学講座

御型に分類することができる。集中制御型のアルゴリズムでは，ネットワーク全体の負荷状態を観測する計算機が存在し，それがタスク転送の決定を行なう。分散制御型においては，全体を管理する計算機が存在せず，タスク転送の決定はそれぞれの計算機で分散して行なわれる。

分散制御型の動的負荷分散アルゴリズムの一つとして，タスクの送り手側からタスク転送の要求をする sender-initiated アルゴリズム[2),3)]がある。この手法では，計算機に外部からタスクが入力されたときに待ち行列の長さを求め，それがある閾値を越えていた場合には，その計算機の負荷が重いと判断して，ランダムに一つの計算機を選択し，そこにタスク転送の要求をする。要求を受けた計算機の負荷が軽い，すなわち待ち行列の長さがある閾値よりも小さい場合にはタスク転送の要求が受け入れられ，タスクがそこへ転送される。要求が拒絶された場合には，別の計算機に対して要求が送られる。要求の送出数に関してある限度を設けておき，その限度を越えた場合にはタスク転送要求を中止し，タスクが入力されたプロセッサの待ち行列にそれを入力する。

この sender-initiated アルゴリズムは，システム全体の負荷が軽い場合には要求がほぼ満たされるため，効率良く動作する。しかしながら，システム全体の負荷が重い場合には，要求を行なっても多くの場合それが拒絶され，無駄な要求が繰り返されることとなるので，負荷分散を行なう能力が著しく低下する。本論文では，sender-initiated アルゴリズムにおいて無駄な要求が送出されるのを防ぐことを目的とし，遺伝的操作を導入した分散制御型動的負荷分散アルゴリズムを提案する。提案する手法においては，どの計算機に対してタスク転送の要求を送出するかを記述した文字列からなる集団を各計算機ごとに用意し，それに対して遺伝的アルゴリズムの操作を適用することで，適切な転送要求の送出方法を学習する。シミュレータを用いた実験により，従来の sender-initiated アルゴリズムとの比較を，タスクの入力から結果の出力までの応答時間の観点から行ない，提案する手法の有効性を示す。

2．遺伝的アルゴリズム

遺伝的アルゴリズム(Genetic Algorithms, GAs)[4)]は適合度関数(fitness function)の最大化問題を近似的に解くアルゴリズムである。適合度関数のすべての変数は，アルファベット(alphabet)中の文字から構成される文字列(string)の形式にコーディングされる。この1本の文字列を個体と呼ぶ。すなわち1個体は探索空間内のある1点を示している。個体からなる集団(population)に対して以下の基本3操作を繰り返し適用し，より高い平均適合度をもった集団に収束させる。

selection　個体の適合度に応じて次世代におけるその個体の数を定める。つまり，適合度の高い個体の数を増やし，逆に適合度の低い個体の数を減少させる。

crossover　個体を交叉させる。つまり，個体をペアにしてその間で部分列を相互交換する。

mutation　個体を突然変異させる。つまり，その中のある文字を別の文字に変化させる。

静的負荷分散はタスクの割当て問題とみなすことができ，通常の組合せ最適化問題として解くことができるので，GA を適用するのは容易である。Mansour と Fox[5)]は，静的負荷分散をデータグラフから計算機ネットワークグラフへの写像としてとらえ，その割当て問題を，改良を施した GA で解いている。一方，動的負荷分散に GA を適用する試みはこれまで行なわれてこなかった。特に分散制御による動的負荷分散では，環境の動的変化に追従する必要があるのみならず，状態

観測に関して通信ネットワークを用いているために，遅延のある情報しか手に入れることができず，単純な最適化問題とみなすことはできない。

3．分散制御型動的負荷分散アルゴリズム

提案する手法は，基本的にはsender-initiatedアルゴリズムにおいて，そのタスク転送要求の送出先を複数指定するマルチキャストを導入したものであり，その送出先を示した文字列からなる集団を，遺伝的操作を施す対象とする点に特徴がある。一般にタスク転送を要求する場合，ブロードキャストにより全ての計算機に要求する方法や，ランダムに一つの計算機を選択しユニキャストにより要求する方法が考えられる。ブロードキャストは通信量が非常に大きくなるという問題点がある。また，ユニキャストの場合にはシステム全体の負荷が重いと多数の要求を繰り返す必要が生じ，タスク転送が行なわれるまでに時間がかかる。提案する手法では，個体からなる集団に対して遺伝的操作を施すことで適切なタスク転送要求を行ない，効率的な負荷分散を実現する。

3．1　概要

本手法の概念図を示したものが図1である。図の中の P_0〜P_7 はDCSにおけるそれぞれの計算機を示している。負荷状態は，タスクの待ち行列の長さがある閾値を越えたかどうかで，(Light, Normal, Heavy)の3状態から選択する方式を採用する。自分自身の負荷がHeavyであるときにタスク転送の手続きを実行し，負荷がLightである計算機が見いだされたなら，それに対してタスクを転する。どの計算機に対してタスク転送の要求を行なうかを記述した列をstring $=\langle v_0, v_1, \cdots, v_{n-1}\rangle$ (nは全体の計算機の数)として定義する。これをGAの個体とする。ここで，v_j はタスク転送を計算機 P_j に要求する場合に1，しない場合に0の値を持つ。ただし，自分自身に対応する v_i の値は常に0とする。各計算機ごとに，m個の個体からなる集団を用意する。タスク転送の手続きが

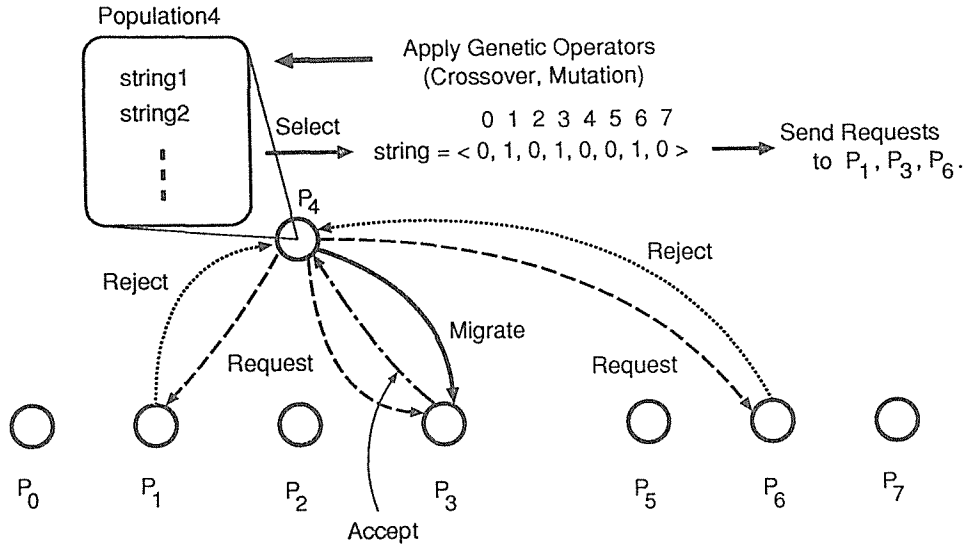

図1　本手法の概念図

起動されたとき，適合度に比例した確率で一つの個体を集団から選択し，それに応じたタスクの転送要求がなされる。

それぞれの個体の適合度は，以下で定義される payoff 値の過去 T 回の和として求められる。ある個体が選択され，それに従って要求が送出されたとき，その要求が受け入れられたかどうかにより，payoff の値が定められる。

$$\text{payoff} = \begin{cases} 0 & \text{If all requests were refused.} \\ (\text{全体の計算機数}) - (\text{要求メッセージの送出数}) & \text{Otherwise.} \end{cases}$$

タスク転送の要求を送った先の計算機すべてからそれを拒絶された場合，タスクの転送は行なわれず，payoff の値は0となる。要求を送った中で少なくとも一つの計算機から要求を受理された場合にはその計算機へタスクが転送される。2つ以上の計算機が要求を受理した場合には転送相手がその中からランダムに選択される。要求が受理された場合に，送出した要求の数が多い時には少ない payoff，逆に少ない時には大きい payoff 値を与える。つまり，少ない要求の送出数でタスク転送が成功する場合に最も高い payoff 値が与えられることとなる。これにより，無駄な要求メッセージが多数発生することを防ぐ。

さらに各計算機において，個体からなる集団に対して定期的に遺伝的操作である crossover，mutation を適用する。この遺伝的操作は payoff 値評価の一定回数ごとに実行され，生成された個体が集団内で適合度の最も低いものと置き換えられることにより selection が実行される。この操作により，適合度の高いすなわち性能の良い個体が集団内に残ることとなる。

3．2　アルゴリズムの詳細

各計算機で実行される負荷分散アルゴリズムは，Initialization，Check_load，Message_evaluation，String_evaluation，Genetic_operators の5つの手続きから構成されている。それぞれの依存関係を図 2 に示す。

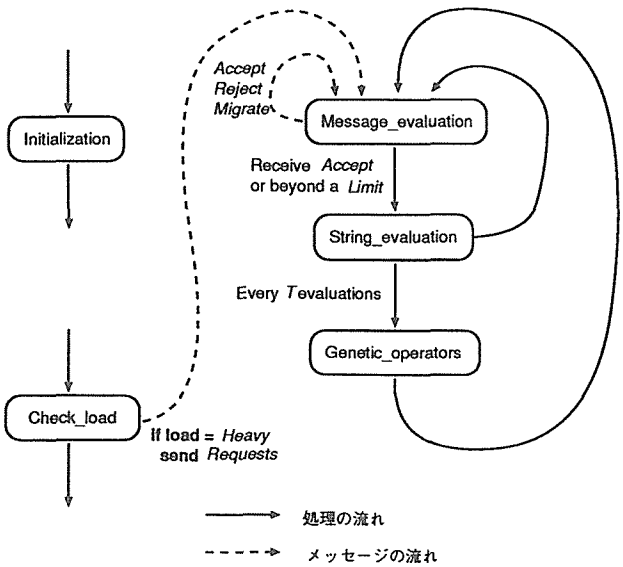

図 2　負荷分散アルゴリズムにおける手続き間の依存関係

Initialization は全体のシステムが起動されるときのみ，一度だけ起動される。このアルゴリズムを図3に示す。そこでは各計算機内の集団が初期化されるとともに，負荷情報が初期化される（すべての負荷情報が Light にセットされる）。

Check_load は各計算機において定期的に起動され，その計算機の負荷情報を観測する。このアルゴリズムを図4に示す。負荷状態は上で述べたように，(Light, Normal, Heavy)の3状態からタスクの待ち行列の長さにより決定される。そして，負荷が Heavy の場合にタスク転送の要求手続きを起動する。タスク転送要求が起動されたなら，まず，適合度の値に応じた確率で一つの個体が選択される。その個体中の文字に従ってタスク転送要求がなされる。個体中の文字 v_i が1ならば計算機 P_i にタスク転送の要求である Request メッセージを送り，0ならば送らない。この手続きはタスク転送要求のメッセージ送出を行なうのみで，要求自体の処理は Message_evaluation にまかされる。

```
Initialization(i)
Processor_ID   i;
{
   load(i) = Light;
   Available_list(i) = NULL;
   for(j = 0; j < Number_of_strings; j++) {
      initialize_string(string(i, j));
   }
}
```

図3　集団の初期化手続き

Message_evaluation は他の計算機からのメッセージを受け取った時に起動される。これを図5に示す。ある計算機が Request メッセージを受け取った時，その計算機の負荷が Light である場合には Accept メッセージを，それ以外の場合には Reject メッセージを送り返す。一方，Request メッセージの送り元の計算機では，Accept メッセージが戻ってきた場合には，そのメッセージの送出元の計算機を Available_list に加える。Request メッセージの送り元の計算機では，それを送った先の計算機すべてから，Accept または Reject メッセージが戻ってきた後に，Available_list

```
Check_load(i)
Processor_ID   i;
{
   switch(observe_load(i)) {
      case Light  : load(i) = Light; break;
      case Normal : load(i) = Normal; break;
      case Heavy  : load(i) = Heavy;
         selected_string = roulette wheel selection from population(i);
         number_of_requests = 0;
         for(j = 0; j < number_of_processors; j++) {
            if(selected_string(j) == 1) {
               send Request to processor j;
               number_of_requests(i)++;
            }
         }
         break;
   }
}
```

図4　負荷状態の観測手続き

```
Message_evaluation(i)
Processor_ID   i;
{
   switch(received_message) {
      case Request from j:
         if(load(i) == Light) send Accept to processor j;
         else send Reject to processor j;
         break;
      case Accept from j:
         add j to Available_list(i);  number_of_requests(i)——;
         if(number_of_requests(i) == 0) {
            k = select from Available_list(i);
            send Migrate with a task to processor k;
            String_evaluation(i, selected_string, Success, total_requests);
         }
         break;
      case Reject:
         if(number_of_requests(i) == 0)
            if(available_list(i) == NULL)
               String_evaluation(i, selected_string, Failure, 0);
            else {
               k = select from Available_list(i);
               send Migrate with a task to processor k;
               String_evaluation(i, selected_string, Success, total_requests);
               available_list(i) = NULL;
            }
         break;
   }
}
```

図5　メッセージ処理手続き

```
String_evaluation(i, string, flag, total_requests)
Processor_ID   i;
String         string;
Success_flag   flag;
int            total_requests;
{
   if(flag == Success) payoff(string) = number_of_processors - number_of_requests;
   else payoff(string) = 0;
   number_evaluation++;
   if(number_of_evaluation == G) {
      Genetic_operators(i);
      number_of_evaluation = 0;
   }
}
```

図6　個体の評価手続き

```
Genetic_operators(i)
Processor_ID   i;
{
   (d1, d2, d3) = select worst three strings(i);
   (p1, p2, p3) = select three strings randomly(i);
   /* uniform crossover */
   for(i = 0; i < length; i++) {
      if(random[0,1) < 0.5) { p1'(i) = p1(i); p2'(i) = p2(i) }
      else { p1'(i) = p2(i); p2'(i) = p1(i) }
   }
   /* mutation */
   k = random[0,1,2,..,length-1];
   p3'(k) = 1 - p3(k);
   /* selection */
   replace (d1, d2, d3) with (p1', p2', p3');
   /* Fitness values are inherited from their parents */
}
```

図7　遺伝的操作

から一つの計算機が選択され，それに対して，転送対象となるタスクとともにMigrateメッセージが送られる。Available_listが空である場合にはタスクの転送は実行されない。以上の操作の後にString_evaluationが起動される。

String_evaluationは選択された個体のpayoff値を求める。これを図6に示す。少なくとも一つの要求が受け入れられたなら，要求に要したコスト（送ったRequestの数）を考慮したpayoff値が与えられ，要求が失敗したなら，payoff値は0となる。適合度の値はこのpayoff値の過去T回の和として与えられる。さらに，payoff値の評価G回毎に遺伝的操作であるGenetic_operatorsが起動される。

Genetic_operatorsは個体からなる集団に対して遺伝的操作であるselection, crossover, mutationを施す処理であり，全体の個体評価G回毎に起動される。この詳細を図7に示す。まずはじめに適合度の値の低いものから順に3つの個体(d1, d2, d3)が選択される。さらにランダムに3つの個体(p1, p2, p3)が選択される。そして(p1, p2)についてuniform crossover[6]により部分列が交換され，(p1', p2')となる。さらに，p3の中のある文字が別の文字に変化することでmutationが実行され，p3'となる。最後に(d1, d2, d3)が除去され，(p1', p2', p3')と置き換えることによりselectionが実行される。(p1', p2', p3')の適合度の値は(p1, p2, p3)から継承される。

4．シミュレーション実験

UNIX-WS上にシミュレータを開発し，以下の条件でシミュレーション実験を行ない，タスクの入力から出力までの平均応答時間を観測することで，sender-initiatedアルゴリズムと提案する手法の比較をした。

計算機ネットワーク：16台の計算機が，通信速度10Mbpsのバス結合型ネットワークにより接続されている場合を想定した。

外部から入力されるタスク：実行時間に関しては平均100msの指数分布，サイズに関しては平均10kBytesの指数分布にそれぞれ従う。

負荷分散アルゴリズムのパラメータ：sender-initiatedアルゴリズムでは，要求を行なう回数の上限を8とした。提案する手法では，それぞれの集団に含まれる個体数を10，適合度値は過去3回のpayoff値の和とした。また，遺伝的操作はpayoff値の評価20回ごとに起動した。

全体のシミュレーション時間：10,000個のタスクを実行するまでシミュレーションを実行した。

実験回数：10回の実験の平均を求めた。

16台の計算機すべてに対して一様に同じ平均到達割合でタスクを入力した場合における，タスクの平均到達割合と平均応答時間の関係についての結果を図8に示す。負荷分散を行なわない場合(NoBalancing)，sender-initiatedアルゴリズム(Sender)，提案する手法(Genetic)の3つの手法について比較を行なった。ここで，図の横軸はタスクの平均到達割合であり，縦軸は平均応答

時間（単位 millisecond）である。

この結果によると，負荷分散を行なわない場合，平均到達割合が1に近づくにつれて平均応答時間が著しく大きくなる。提案する手法が最も少ない平均応答時間を示しているが，sender-initiated アルゴリズムとの差は顕著なものではない。例えば平均到達割合が0.99であるときには，sender-initiated アルゴリズムでは1105.3ms（標準偏差106.2ms），提案する手法では1032.85ms（標準偏差182.5ms）であるので，この場合2つの手法の間に有意な差が存在するとはいえないが，少なくとも一様負荷の場合には遺伝的操作を導入することによる性能の悪化は見られないことが分かった。

次に，負荷分散が重要になると考えられる非一様負荷の場合について平均応答時間を求めた結果を図9に示す。一様負荷の場合と同数のタスクを，4台の計算機に対して均等に割当てている。一様負荷の時の各プロセッサに対するタスクの平均到達割合を λ とすると，この場合，4台の計算機に対して平均到達割合 $4\ \lambda$ でタスクが到達し，残りの12台の計算機に対してはタスクが入力されないこととなる。

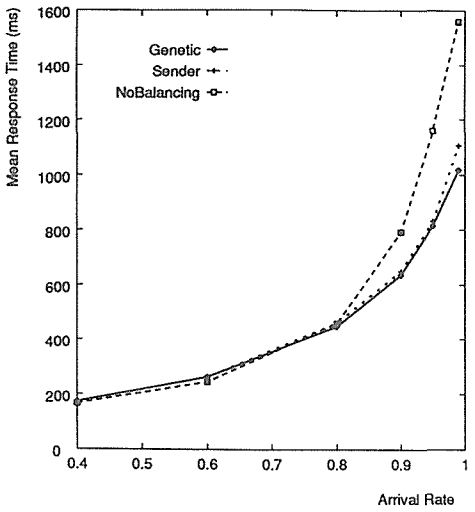

図8　平均応答時間（一様負荷）

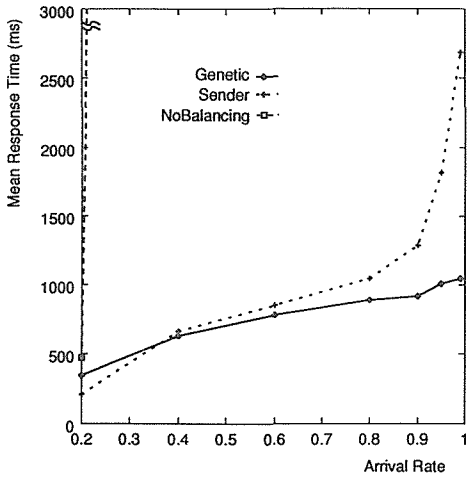

図9　平均応答時間（非一様負荷）

負荷分散を行なわないときには，4台の計算機に対しては16台の計算機の場合の4倍のタスクが入力されているので，平均到達割合が0.25の時に一様負荷における平均到達割合1.0の場合と同じ状況となり，0.2を越えたあたりから応答時間が急激に増大している。この場合には提案する手法の sender-initiated アルゴリズムに対する優位性は明らかである。例えば，到達割合が0.99であるとき，sender-initiated アルゴリズムでは平均応答時間が2683.5ms（標準偏差358.6ms），提案する手法では1050.7ms（標準偏差36.2ms）と，有意な差が認められる。

つまり，この手法は負荷が一様である場合においても従来の手法に比べて性能は低下せず，負荷が一様ではない場合において有効であるといえる。一様負荷の場合には，どの計算機に対してもタスクが入力されているため負荷の不均衡があまりなく，タスク転送先を学習する必要が少ない。一方，非一様負荷の場合には，タスクが入力されていて負荷が重い計算機から，タスクの入力がなく負荷の軽い計算機へタスクを転送するような負荷分散に対応する個体が，遺伝的操作に

より集団内で優位を占め，無駄なタスク転送の要求が避けられる。

5．おわりに

遺伝的操作を用いた分散制御型動的負荷分散アルゴリズムを提案し，タスクの入力から結果の出力までの平均応答時間に関してのシミュレーション実験を行ない，sender-initiated アルゴリズムと比較した場合の有効性，特に非一様負荷に対する有効性を確認した。今後の課題としては，負荷の軽い計算機からもタスク転送の要求メッセージを送出することを考慮したアルゴリズムの改良があげられる。

参考文献

1) Suen, T. T. Y. and Wong, J. S. K.: "Efficient Task Migration Algorithm for Distributed Systems", IEEE Trans. on Parallel and Distributed Systems, vol. 3, No. 4, pp. 488-499 (1992).
2) Eager, D. L., Lazowska, E. D. and Zahorjan, J.: "Adaptive Load Sharing in Homogeneous Distributed Systems", IEEE Trans. on Software Eng., vol. 12, No. 5, pp. 662-675 (1986).
3) Shivaratri, N. G., Krueger, P. and Singhal, M.: "Load Distributing for Locally Distributed Systems", IEEE COMPUTER, vol. 25, No. 12, pp. 33-44 (1992).
4) Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley (1989).
5) Mansour, N. and Fox, G. C.: "A Hybrid Genetic Algorithm for Task Allocation in Multicomputers", Proc. of the 4th Int. Conf. on Genetic Algorithms, pp. 466-473, Morgan Kaufmann Publishers (1991).
6) Gilbert Syswerda.: "Uniform Crossover in Genetic Algorithms", Proc. of the 3rd Int. Conf. on Genetic Algorithms, pp. 2-9 (1989).