ベクトルキーを用いたプロセスグループへの資源割当てのための分散アルゴリズムの設計

学生会員 田島 中、学生会員 和田 裕、正会員 程 子学
会津大学コンピュータ理工学部

分散環境における資源割当て問題は、いままで分散システムの分野で盛んに研究されており、デッドロックやスターベーションを回避する分散アルゴリズムは多数開発されている。しかしながら、コンピュータネットワークの発展に伴い、グループウェアを用いて様々なグループ活動がネットワークを通じて行なわれている。その時、複数のグループはネットワーク上の複数の資源を競合し、複数のグループ間でグループデッドロックやグループスターベーションが起こり得る。いままでの分散アルゴリズムでは、デッドロックやスターベーションを回避できるが、グループデッドロックやグループスターベーションには対応できない。そこで、本稿では、グループデッドロックやグループスターベーションを回避するため、要求ベクトル概念にもとづいた到達度（既に確保した資源）を用いた複数のグループへの資源割当てのための分散アルゴリズムを提案する。

1　はじめに

現在まで、分散環境における資源割当ては重要な問題であり、多くの研究がなされている。[1][2][3] また、コンピュータネットワークの発展に伴い、グループウェアをはじめ様々なアプリケーションの開発が試みられている。そうしたなかで、グループを構成しているプロセスへの、資源割り当てを目的としたアルゴリズムが考案されている。[4] しかし、リソースの代替可能性を考慮し、且つグループを構成している複数のプロセスへの資源割り当てを目的としたアルゴリズムは課題として残されている。資源割当てを難しくしている要因として、デッドロック、スターベーションに加え、システムの中の複数のプロセスグループがリソースを競合するときに生じる、グループデッドロックや、グループスターベーションの解除または、回避を保証しなければならないということがある。

本稿では割当ての際に生じた競合を、リソースの「仮配分」と「到達度」という概念で解消させることを提案し、トークン方式によるリソース分配の分散アルゴリズムを示す。

以下、2.で資源割当ての前提となるモデルや諸定義を与える。3.で要求ベクトル概念を用いたアルゴリズムを提案し、4.でそのアルゴリズムの諸性質を述べる。

Desing of Distributed Algorithm for Resource Allocation among Process Groups Using Vector-key Naka TAJIMA Department of Computer Hardwear, Univ of Aizu, Yutaka WADA Department of Computer Softwear, Univ of Aizu, Zixue CHENG Department of Computer Softwear, Univ of Aizu

2　諸定義

2.1　資源割当てモデル

本稿では、リソースの集合 R を代替可能性[1]により分類し、部分集合 $\forall t \in T, R_t \subseteq R$ に分割される。ここで、$T = \{t_1, t_2, \cdots, t_{|T|}\}$ は、リソースタイプの識別子の集合である。また、各タイプに対応するリソースマネージャが一つずつ存在し、そのタイプに属するリソースを管理している。資源割当ての前提となる分散システムは、それぞれユニークな識別子を持つプロセスの集合 $P = \{p_1, p_2, \cdots, p_{|P|}\}$ とリソースマネージャの集合 $RM = \{rm_1, rm_2, \cdots, rm_{|T|}\}$ を頂点とし、それらの間のリンクを辺とする二部グラフとして表される。また、P は複数のグループという部分集合 $\forall g \in G, G_g \subseteq P$ に分割される。$G = \{g_1, g_2, \cdots, g_{|G|}\}$ はグループの識別子の集合である。割当てアルゴリズムは、プロセスとリソースマネージャの間でメッセージの送受信をし、資源へのアクセス権を制御する。また、以下の条件を仮定する。

1. 分散システム (P, RM) は完全二部グラフである、したがって、割り当ての対象となるリソースの集合 R は、P のすべてのプロセスから、アクセス可能である。

[1]あるプロセス p_i の要求が、r_a または r_b どちらか一方あれば満たされるとき、r_a と r_b は代替可能であるという

2. P, R の数は静的で変化しない。

3. 異なる二つのタイプに属するリソースは存在しない。つまり、$R_x \cap R_y = \emptyset \quad (x, y \in T)$。

4. 異なる二つのグループに属するプロセスは存在しない。つまり、$G_u \cap G_v = \emptyset \quad (u, v \in G)$。

5. P と RM の間のリンクは FIFO 型双方向通信路である。

6. 同じグループのプロセス同士の間には FIFO 型双方向通信路がある。

7. P, RM はローカルクロックを持つ。

8. メッセージ送信にかかる時間は一定ではないが、ある有限時間内に到着が保証されている。

9. 故障は生じない。

以下図 1 に分散システムモデルを示す。

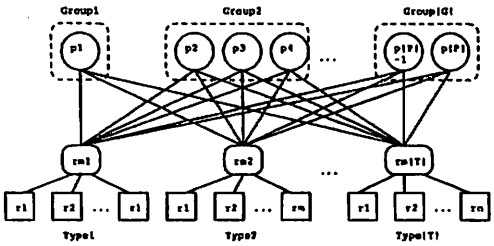

図 1: 分散システムモデル

2.2 問題

分散システム上の各プロセスはリソースとの関係の違いによる三つの状態 $Idle, Hungry, Woking$ からなるサイクルを遷移している。$Idle$ はリソースを使用せず且つ要求もない状態、$Hungry$ はリソースへの要求が起こっている状態、$Working$ は要求したリソースをすべて取得して使っている状態である (一括要求方式)。だだし、グループ G_u に属しているプロセスは、同じグループ内のすべてのプロセスがそれぞれに要求したすべてのリソースへのアクセス権を取得しない限り $Working$ に状態遷移することができない。

プロセスグループへの資源割当て問題とは、プロセスと RM とのメッセージの送受信のみで、以下の四つの条件を保証し、リソースへのアクセス権の制御をすることである。

1. デッドロックフリー：プロセス同士で、リソースの解放を待ち合い、要求したリソースが得られない状態に陥るということがない。

2. スターベーションフリー：要求したリソースを、永久に得られないプロセスが存在している状態になることはない。

3. グループデッドロックフリー：プロセスグループ同士で、リソースの解放を待ち合い、要求したリソースが得られない状態に陥ることがない。

4. グループスターベーションフリー：要求したリソースを、永久に得られないプロセスグループが存在している状態になることはない。

だだし、グループ内のプロセス同士でリソースへの競合は起こらないと仮定する。また、全てのプロセスは、有限時間内に取得したリソースを使用し解放する。

3 アルゴリズム

まず各グループのプロセス数が一つだけの特別な場合、言い替えれば、プロセスがグループを構成していないケースで、「仮配分」、「到達度」を用いたアルゴリズムを説明し、つぎにプロセスがグループ化された場合への拡張方法を述べる。

3.1 リソース空間と要求ベクトル

リソースのタイプを次元、そのタイプに属するリソースの数を成分として、リソースの集合をベクトルで表現する、これをリソース空間と呼び $RS = (|R_{t_1}|, |R_{t_2}|, \cdots, |R_{t_{|T|}}|)$ で表す。

各プロセス p_i の要求は、リソース空間の部分空間をなすベクトル $Demand(p_i) = (d_1, d_2, \cdots, d_{|T|})$ で表される。d_x は、タイプ t_x のリソースの必要数を表している。複数のプロセスが同時に要求を出した場合、それらの要求ベクトルを足し合わたもの $\sum Demand()$ の要素が、対応する RS の要素より大きかったとき、それらの要求を一度に満たすことは明らかに不可能である。例：$RS = \{3,2,4\}, Demand(p_1) = \{1,2,2\}$ このときの p_1 の要求はタイプ 1 のリソースが一つ、タイプ 2、3 のリソースが二つずつ必要としている。タイプ 1 のリソースの総数は 3 なので、仮にすでに二つ使われていても残りのリソースを割り当てることができる、しかしタイプ 2 のリソースについては、その総数と要求の数が一致しているため、もし、一つでも他の

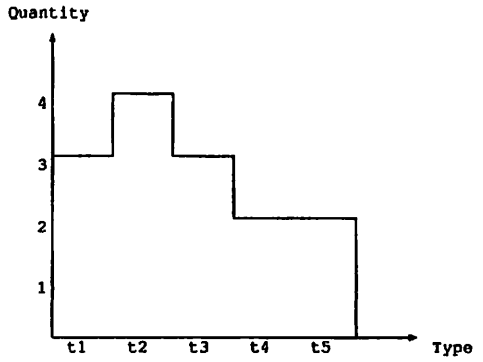

図 2: 例 :$RS = (3,4,3,2,2)$

プロセスに割り当てられていれば p_1 の要求は満たすことができない。

3.2 基本となるアイデア（到達度による競合の解消）

分散システム上のプロセス p_i は、$Idle$ から $Hungry$ へ状態遷移をする際に、必要とするすべてのリソースタイプ t_x に対応するリソースマネージャ rm_x へリクエストメッセージを送信する。

各リソースマネージャ rm_x は、プロセスからの要求を保持するための長さ $|P|$ のキュー $Queue_x$ を持っている。また、初期状態において、rm_x は自分が管理しているリソースの数 $|R_x|$ だけリソーストークンを保持している。

各プロセス p_i が出した要求のメッセージ $Req(Demand(p_i))$ は rm_x に受信されると、$Queue_x$ へ次々とエントリされる。このとき、あるいはプロセスからの、リソースの使用を終了したメッセージ $Release(Demand(p_i))$ を受信するごとに rm_x は要求が満たせる限り、それらのプロセスに対しリソーストークンを送信する。その、リソーストークンをプロセスが受信したとき、そのプロセスはタイプ t_x のリソースに関して「仮配分」されたという。

そして、プロセスが仮配分されたリソースを到達ベクトル $Status(p_i) = \{s_1, s_2, \cdots, s_{n(T)}\}$ で表す。s_x はタイプ t_x に関して仮配分されたリソースの数である。

もし、rm_x が要求を満たせなくなり競合が起きた場合、つまりもともと rm_x が保持しているリソーストークンの数 $|R_x|$ を、プロセスのタイプ t_x に対する要求の総和が上回ってしまった場合、競合を起こしているプロセスの集合（リソーストークンを保持しているプロセスと、たった今要求を受信され競合のトリガーになったプロセス）に対し、現在それらのプロセスが要求全体のうちどのぐらい仮配分されているかを問い合わせる。そして、「到達度」というものを計算し、その値の大きいプロセスを優先的に仮配分し直し、デッドロックを解消する。もし、到達度が同じ値であった場合には、プロセスの識別子を利用して、要求の全順序を確保する。

到達度 $Rate(p_i)$ は到達ベクトルの長さを要求ベクトルの長さで割った値である。

$$Rate(p_i) = \frac{|Status(p_i)|}{|Demand(p_i)|}$$

自分の要求 $Demand(p_i)$ と同じだけの仮配分をされたプロセスは、直ちに $Hungry$ から $Working$ へと、状態遷移をして、リソースへのアクセスを開始する。また、このときに、$Status(p_i) = \{0,0,\cdots,0\}$ と初期化される。

到達度の大小比較により、一度送信されたリソーストークンをリソースマネージャに返還しなければならなくなったプロセスは、たとえすべてのリソーストークンを返還してしまっても、再要求した場合キューの先頭にエントリされ、しかも、過去に仮配分されたときの自分の最大到達ベクトルを、現在の到達ベクトルとして使用する。これにより、スターベーションを防止する。

以下が、使用されるメッセージである。

1. $Req(Demand(p_i))$ プロセスからの要求メッセージ
2. $Query(t_x)$ リソースマネージャからの到達度の問い合わせメッセージ
3. $Answer(Rate(p_i))$ プロセスからの到達度の問い合わせに対する返答
4. $Release(Demand(p_i))$ プロセスからのリソースの解放、及びリソーストークンの返却
5. $OK(t_x)$ リソースマネージャからのリソーストークン配分
6. $Return(t_x)$ リソースマネージャからのリソーストークン返却要請

3.3 分散アルゴリズム

プロセス：p_i :

RM へ $Req(Demand(p_i))$ を送信

```
loop
  if(RM すべてから OK() を受信){
    working へ遷移
    リソースを使用し終ったら
    RM すべてに Release(t_x) を返信
    exitloop
    }
  if(Query(t_x) を受信){
    RM すべてから Req(Demand(p_i)) の
    返答 (OK() もしくは、Query()) を受信したら
    Answer(Rate(p_i)) を Query() の
    送信主すべてに送信
    }
  if(Return(t_x) を受信){
    rm_x へ Release(t_x) を返信
    Req(Demand(p_i)) を rm_x へ送信
    }
endloop


リソースマネージャ：rm_x
loop
  if(Req(Demand(p_i)) を受信){
    if(p_i ∈ List_x)
      Queue_x の先頭にエントリ
    else
      Queue_x の最後部にエントリ
    if(Demand(p_i)_{t_x} ≤ (リソースの残存数)){
      OK(t_x) を p_i に送信
      List_x へ p_i を加える
      }
    if(Demand(p_i)_{t_x} > (リソースの残存数)){
      Query(t_x) を List_x に記憶されている
      すべてのプロセス及び p_i へ送信
      wait for all reply
      if(Release() を受信)
        リソースの残存数を更新
      Answer() を返信してきたプロセスに対し、
      Rate() の大きい順に要求の数を
      足してゆきながら、List_x に入っていない
      プロセスがないかチェックし、もし見つか
      れば OK(t_x) を送信して、List_x に加える。
      |R_x| を超過した時点であふれたプロセスに
      Retrurn(t_x) を送信
      }
    }
endloop
```

$Demand(p_i)_{t_x}$ は $Demand(p_i)$ の t_x 番目の要素をあらわす。

3.4 グループへの拡張

3.4.1 グループ到達度

システムの中の複数のプロセスがグループを形成している場合では、グループ内のそれぞれのプロセスが、デッドロックフリーを保っていても、グループデッドロックを避けることはできない。

グループ単位での競合を解消するためには、グループ単位で比較し、全順序を決定できる指標が必要である、そのため 3．2節で定義したプロセスの到達度をグループの到達度へと拡張する。グループ内のすべてのプロセスの到達ベクトルの和をグループ到達ベクトルと呼ぶ。

$$Status(G_u) = \sum_{\forall k, p_k \in G_u} Status(p_k)$$

また、グループ要求ベクトルとは、グループ内のすべてのプロセスの要求ベクトルの和である。

$$Demand(G_u) = \sum_{\forall k, p_k \in G_u} Demand(p_k)$$

グループ到達度とは、グループ到達ベクトルの長さをグループ要求ベクトルの長さで割った値である。

$$Rate(G_u) = \frac{|Status(G_u)|}{|Demand(G_u)|}$$

3.4.2 一貫性の保護

グループ内のすべてのプロセスはこのグループ到達度を統一された値をして持っている必要がある、そのため $Enquiry(p_i)$ 及び $Notify(Demand(p_i))$, $Notify(Status(p_i))$ というメッセージを用いてグループ到達度の一貫性を保護する。

まずグループ内のすべてのプロセスはグループ要求ベクトルを知らなければならないので、プロセス p_i が $Idle$ から $Hungry$ に状態遷移をする際に、グループ内の自分以外のすべてのプロセスに $Notify(Demand(p_i))$ を送信する。またそれらのプロセスすべてから $Notify(Demand(p_j))$ が送られるのを待つ、すべて受信した時点ではじめて自分の要求するリソースへ $Req(Demand(p_i))$ を送信することができる。

プロセス p_i が $Query(t_x)$ を受信した場合、$Enquiry(p_i)$ をグループ内のすべてのプロセスに送信する、そして、$Enquiry(p_i)$ を受信したプロセスすべてからその返答である、$Notify(Status(p_j))$ を受信したら、そこでグループ到達度を計算し、rm_x に $Answer(Status(G_u))$ を送信する。

以上のことを、3．2のプロセス p_i のルーチンへ付け加えたものが以下である。

プロセス：p_i
G_u へ $Notify(Demand(p_i))$ を送信
wait for $\forall j \in G_u, Notify(Demand(p_j))$
RM へ $Req(Demand(p_i))$ を送信
loop
  if(RM すべてから $OK()$ を受信)
  G_u すべてに $Enquiry(p_i)$ を送信
    if($Enquiry(p_j)$ を受信)
      $Notify(Status(p_i))$ を送信
    if($Status(G_u) = 1$){
      $working$ へ遷移
      リソースを使用し終ったら
      RM すべてに $Release(t_x)$ を返信
      exitloop
      }
  if($Query(t_x)$ を受信){
    RM すべてから $Req(Demand(p_i))$ の
    返答 ($OK()$ もしくは、$Query()$) を受信したら、
    G_u すべてに $Enquiry(p_i)$ を送信
    wait for $\forall j \in G_u, Notify(Status(p_j))$
    $Answer(Rate(p_i))$ を $Query()$
    の送信主すべてに送信
    }
  if($Return(t_x)$ を受信){
    rm_x へ $Release(t_x)$ を返信
    $Req(Demand(p_i))$ を rm_x へ送信
    }
endloop

リソースマネージャは、$Rate(p_i)$ の代りに、$Rate(G_u)$ を大小比較して、リソーストークン送信の優先順をきめる。

4 アルゴリズムの性質

提案されたアルゴリズムは以下の性質を持つ。

1. デッドロックフリー：到達度とプロセスの識別子により、プロセス間の全順序性が維持されるので、互いに他を待ち合う状態は生じない。

2. スターベーションフリー：到達ベクトル $Status(p_i)$ 及び到達度 $Rate(p_i)$ はそのプロセスが、すべての要求を満たされるまで初期化されることはなく、仮配分を重ねる毎に大きくなることはあっても、小さくなることはない。よって、永久に配置を待つようなプロセスは存在しない。

3. グループデッドロックフリー：グループ到達度とグループ識別子により、グループ間の全順序性が維持されるので、互いに他を待ち合う状態は生じない。

4. グループスターベーションフリー：プロセスが単体の場合と同様に、グループ到達ベクトル $Status(G_u)$ 及びグループ到達度 $Rate(G_u)$ はそのグループが、すべての要求を満たされるまで初期化されることはなく、仮配分を重ねる毎に大きくなることはあっても、小さくなることはない。また、グループ内の個々のプロセスに対してスターベーションフリーが保証されている場合、自動的にグループスターベーションフリーも保証される。

5 まとめ

本稿では、「仮配分」の概念を用いて「到達度」を定義し、グループデッドロックやグループスターベーションを回避するような、複数のグループへの資源割当てのための分散アルゴリズムを提案した。今後の課題として、1.リソースマネージャの役割をプロセス間で代行するような、より一般的なモデルへ対応したアルゴリズムの考案、2.グループ間メッセージ数を減らすため、または、リソース利用の並列性を向上させるために、到達度の算出方法を工夫する、3.プログラムで実装させ、正当性、有効性などを検証する、などがあげられる。

参考文献

[1] 前川 守 " 岩波講座ソフトウェア科学 7 ソフトウェア実行／開発環境"

[2] 山下 雅史、亀田 恒彦 " 分散アルゴリズム"

[3] Chandy Misra "Parallel Program Design"

[4] Zixue CHENG, Tongjun HUANG, Nario SHIRATORI "A Distributed Algorithm for Resource Allocation among Process Groups" ICOIN