■6 群（基礎理論とハードウェア）- 3 編（アルゴリズムとデータ構造）

5 章 並列・分散アルゴリズム

（執筆者：久野　靖）[2012 年 7 月 受領]

■概要■

コンピュータの内部でプログラムの実行を行う部分である CPU が廉価になった今日では, 一つの計算に複数の CPU を用いることで計算効率を向上させることは重要な課題である．また, これと併せてネットワークが普及したことにより, 複数の地点にまたがった CPU 群の協調によりネットワーク上のサービスを実現することも必須となっている．本章では, 複数の CPU に分担して計算を行わせる並列計算, 及び異なる場所にある複数の CPU に協調して計算を行わせる分散計算のための計算モデルとアルゴリズムについて紹介する．

【本章の構成】

5-1 節では, 並列・分散アルゴリズムの基礎となる計算モデルの代表的なものとして, PRAM, logP, π 計算の各モデルを取り上げ紹介する．5-2 節では, 並列アルゴリズム解析の基本的な考え方とワーク最適, コスト最適などの概念について, 具体例を挙げて説明する．5-3 節では, 分散アルゴリズムについてその評価指標を解説した後, リーダ選挙問題を取り上げ様々な考え方を紹介する．

 © 電子情報通信学会 2012

■6 群 - 3 編 - 5 章

5-1 並列・分散計算モデル

（執筆者：久野　靖）［2012 年 7 月 受領］

5-1-1 並列・分散計算モデルの位置づけ

単一 CPU のプログラム実行（逐次計算，sequential computation）に対応する計算モデルの場合と同様に，並列・分散計算についてもそのアルゴリズムを検討したり書き表すうえで，計算モデルが必要である．そしてその計算モデルは，それを実行する計算機システムがどのような並列性を提供するかによって変わってくる．

並列性に基づく計算機システムの大まかな分類としては，Flynn[2)]らによる次のものが代表的である．

- **SISD（single instruction, single data）** — 従来のシングルプロセッサに相当し，並列性をもたないもの．
- **SIMD（single instruction, multiple data）** — 制御の流れは一つだが，個々の命令が多数のデータに作用するもの．
- **MIMD（multiple instruction, multiple Data）** — 制御の流れが複数あり，それぞれが別のデータに作用するもの．

SIMD システムは，一つの命令の流れによって多数の演算器が同一の演算を行うことを意味する．コンピュータによる大量の計算が必要な分野の一つに，行列やベクトルなど，多数のデータが並んだものに対する計算があり，そのような用途に適したモデルが SIMD だといえる．

ただし，今日では「一つの制御装置に多数の演算器」という古典的な意味での SIMD 型 CPU がつくられることはなく，多数の値を格納する装置とそれらの値に高速に同一演算を施すことのできる装置（ベクトルレジスタ・ベクトル演算器）を備えた CPU や，汎用 CPU の一部に複数データに同一演算を施す命令（Intel CPU の MMX 命令など）を搭載するかたちを取ったり，多数の汎用 CPU を用いてそれらの間で同期を取りながら同一のプログラムを実行していくかたちの実装を行うことが多い．最後のものは特に **SPMD（single program, multiple data）**と呼ぶ．また，近年普及してきているグラフィックエンジン（GPU）では，一つのコアに多数の演算器が接続されているが，各演算器は一つのスレッドに対応していて各スレッドが同一のプログラムを実行するかたちであり，これを **SIMT（single instruction, multiple thread）**と呼ぶこともある．

SIMD（SPMD，SIMT）型に近い計算モデルとしては，同期式共有メモリモデルである PRAM モデルがある．名称だけで見ると PRAM は「複数の RAM」つまり MIMD の一種であるように思えるが，PRAM ではすべてのプロセッサが同期して動作するため，そのアルゴリズムはむしろ SIMD に近いものとなる．PRAM モデルについては後で取り上げる．

一方，MIMD システムは更に，プロセッサ間の通信を共有されたメモリによって行う共有メモリ型と，共有メモリをもたずメッセージ交換によって行う非共有メモリ型のシステムに分けられる．共有メモリ型のシステムは，複数 CPU を搭載したマルチプロセサシステム，更

に単一 CPU 内に複数のコアを搭載したマルチコア CPU として広く普及してきている．これに対し非共有メモリ型のシステムは，多数の CPU を高速な通信機構で結合したクラスタシステムや，より廉価なかたちとして単に多数のシステムを一般的なネットワークハードウェアで結合したものに相当する．

システム的には，共有メモリ型のシステムでは共有されるデータに対する競合を制御しつつ計算を進めて行くことが主眼になるのに対し，非共有メモリ型のシステムではプロセサ間で適切に情報を交換しつつ計算を進めて行くことが主眼になる．

これに対し計算モデルという点では，共有メモリも単にプロセサ間でデータを交換する手段と考えることができるので，メッセージ交換に基づく計算モデルやアルゴリズム（分散アルゴリズム）が広く研究されている．

5-1-2 PRAM モデル

並列性を扱う計算モデルとして，一つのプロセサがメモリを読み書きしつつ計算を進めるモデルである **RAM（random access machin）**の処理装置を複数にする，というのは自然な発想である．これを **PRAM（pararell random access machine）**と呼ぶ（図 5･1）．PRAM は同期式のモデルであり，全プロセッサはクロックを共有して動作する．各クロックにおいて，メモリアクセスないし演算を 1 回実行できる．

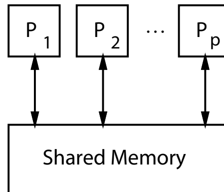

図 5･1　PRAM **モデル**

このように，PRAM は計算のみの並列化に着目したモデルであり，通信時間は考慮していない．PRAM は更に，複数のプロセッサが同一時点で同一メモリ番地にアクセスする場合の動作に基づいて，次のように分類される．

- **EREW（exclusive read, exclusive write）** PRAM — 複数のプロセッサが同一番地をアクセスすることは，読み書きともに禁止されている．
- **CREW（concurrent read, exclusive write）** PRAM — 複数のプロセッサが同一番地を読むことは可能だが，書込みは一つのプロセッサだけが行う必要がある．
- **CRCW（concurrent read, concurrent write）** PRAM — 書込みも複数プロセッサが同時に行ってもよい．更に次の三つのクラスに分類できる．
  - **common CRCW** PRAM — 書き込む全プロセッサにおいて，書き込む値は同一のものでなければならない．

- **arbitrary CRCW** PRAM — 複数プロセッサが書き込んだ場合，そのなかのいずれか一つのプロセサが任意に選ばれ，その書込み値がメモリに残る．
- **priority CRCW** PRAM — 複数プロセッサが書き込んだ場合，プロセサ ID が最小のものの書込み値がメモリに残る．

これらのモデルは後のものほど制約が緩くなるため，前のものを包含する．例えば EREW 用のアルゴリズムを CREW で実行することができるが，その逆は必ずしも成り立たない．

5-1-3 LogP モデル

PRAM モデルではプロセッサ間の通信を考慮していないが，実際のマルチプロセッサや分散システムでは通信時間や通信帯域も性能を制約する重要な要因となる．これらを考慮に入れようとしたものが Culler[1)] らによる **LogP** モデルである．

LogP では計算機システムはローカルメモリをもつプロセッサ多数が結合網（ネットワーク）によって接続されているものとして扱う．各プロセッサは 1 単位時間に一つの演算またはメッセージの送受信を実行する．そして，計算機システムは次の四つのパラメタでモデル化される（LogP という名前は四つのパラメタの名前 L，o，g，P によっている）．

- L（Latency）— 1 メッセージをほかのプロセッサに送るのに要する時間の上界．この時間にはプロセッサはほかの仕事ができる．
- o（overhead）— 1 メッセージの送信／受信に要する時間．送信／受信を実行している o 単位時間のあいだ，プロセッサはほかの仕事ができない．
- g（gap）— 一つのプロセッサによる連続した送信，ないし連続した受信の最小間隔．$\frac{1}{g}$ が通信の帯域幅となる．
- P（Processors）— プロセッサ（+ローカルメモリ）の台数．

LogP モデルを用いることで，通信時間や通信量を考慮したアルゴリズムの評価が行える．例えば，P=8，o=2，g=4，L=6 の場合の最適ブロードキャスト（P0 からほかのすべてのプロセサに情報を伝える手順で，時間が最短のもの）の例を図 5･2 に示す．

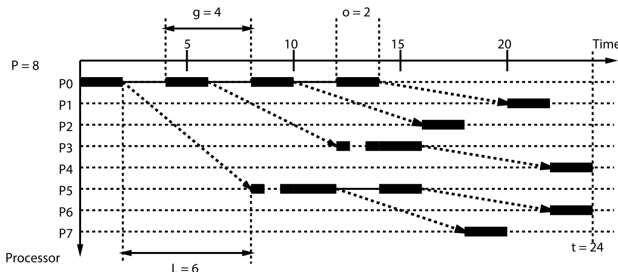

図 5･2　LogP モデルによる最適ブロードキャストの例（P=8，o=2，g=4，L=6，[1)]に基づく）

5-1-4 π 計算

LogP モデルなどが性能評価を目的としているのに対し，逐次計算における λ 計算のように，分散計算の振る舞いや正しさなどの性質を扱うことを目的としたモデルも複数存在している．ここでは**プロセス計算（process calculus）**ないし**プロセス代数**（process algebra）と呼ばれる一群の計算モデルの代表例である，**π 計算（π calculus）**を取り上げる[3)4)]．

λ 計算では名前が値を表していたのに対し，π 計算では名前は値と通信チャネルという二つのものを表すのに使われている．計算は次の構文によって表されるプロセスによって表現される（この構文は参考文献 4) による．また，プロセス表記中の「|」との混同を避けるため，BNF 記法の「または」をタイプフェースの「|」で表している）．

$$
\begin{aligned}
P &::= M \mid P|P \mid \nu zP \mid !P \\
M &::= \mathbf{0} \mid \pi.P \mid M + M \\
\pi &::= \overline{x}y \mid x(z) \mid \tau \mid [x = y]\pi
\end{aligned}
$$

まず，プロセスは $\pi.P$ のようにプレフィクス π をもつことができるが，これはプレフィクスの表す動作が起きてから P の動作に進むことを意味する．各プレフィクスの意味は次のとおり．

- $\overline{x}y$ — 名前 y を名前（チャネル）x を通じて送信する（output）．
- $x(z)$ — 名前（チャネル）x を通じて任意の名前を受信し，名前 z に受け取る（input）．
- τ — プロセスの内部動作を表し，外部からはその中身は観測できない（unovservable）．
- $[x = y]$ — x と y が同じ名前であることを確認（match）．

プレフィクス以外のプロセスの構文は次の意味をもつ．

- $\mathbf{0}$ — 何もしない（inaction）．この状態になったプロセスは以後無視できる．
- $P + P'$ — 和（sum）．P か P' のいずれかが選ばれて動作する（他方は消滅）．
- $P|P'$ — 合成（composition）．P と P' は並行動作する（共通の名前を通じて通信可能）．
- νzP — 限定（restriction）．名前 z の使用は P のなかだけに限定される．
- $!P$ — 重複（replicaiton）．$P|P|P|\cdots$ と同等（無限個の P のコピーが生み出される）．

例えば，次のプロセスを見てみる．

$$\nu x(\overline{x}z.\mathbf{0} \mid x(y).\overline{y}x.x(y).\mathbf{0})|z(v).\overline{v}v.\mathbf{0}$$

チャネル x は $(\cdots)$ で囲まれた内部だけに限定されていおり，その左側のプロセスから中央のプロセスに向かって名前 z を渡すのに使用される．左側のプロセスは z を送った後は $\mathbf{0}$ で停止する．中央のプロセスは z を名前 y に受け取った結果，$\overline{z}x.x(z).\mathbf{0}$ となり，次はチャネル z を通じて名前 x を送信しようとする．ここで右側のプロセスがチャネル z を通じて v に名前を受け取るため，右側のプロセスは $\overline{x}v.\mathbf{0}$ となり，これと $x(z).\mathbf{0}$ とで送受信が成立してすべて $\mathbf{0}$ となる．このように，π 計算ではチャネルを通じて名前を送受信でき，その名前をチャ

ネルとできるため，柔軟な通信が記述可能である．

π 計算のプロセスに対しては，次のような**構造合同（structural congruence）**の公理が成り立つものと考える（≡ で構造合同を表す）．

- $[x = x]\pi.P \equiv \pi.P$
- $M_1 + M_2 \equiv M_2 + M_1$, $M_1 + (M_2 + M_3) \equiv (M_1 + M_2) + M_3$, $M + \mathbf{0} \equiv M$
- $P_1|P_2 \equiv P_2|P_1$, $P_1|(P_2|P_3) \equiv (P_1|P_2)|P_3$, $P|\mathbf{0} \equiv P$
- $\nu z \nu t P \equiv \nu t \nu z P$, $\nu z \mathbf{0} \equiv \mathbf{0}$, $\nu z(P_1|P_2) \equiv P_1|\nu z P_2$（$P_1$が自由変数 z を持たないとき）
- $!P \equiv P|!P$

そして，次の規則により式の還元（reduction）が行えるものとする．

- $(\overline{x}y.P_1 + M_1)|(x(z).P_2 + M_2) \longrightarrow P_1|P_2\{y/z\}$　（$\{y/z\}$ は z の y への置き換えの意味）
- $\tau.P + M \longrightarrow P$
- $P \longrightarrow P'$ のとき, $P|Q \longrightarrow P'|Q$, $\nu z P \longrightarrow \nu z P'$
- $P \equiv P'$, $Q \equiv Q'$, $P \longrightarrow Q$ のとき, $P' \longrightarrow Q'$

これらの規則を用いることで，先の例を次のように還元していくことができる．

$$
\begin{aligned}
&\nu x(\overline{x}z.\mathbf{0} \mid x(y).\overline{y}x.x(y).\mathbf{0}) \mid z(v).\overline{v}v.\mathbf{0} \\
&\to (\mathbf{0} \mid \overline{z}x.x(y).\mathbf{0}) \mid z(v).\overline{v}v.\mathbf{0} \\
&\to \overline{z}x.x(y).\mathbf{0} \mid z(v).\overline{v}v.\mathbf{0} \\
&\to x.x(y).\mathbf{0} \mid \overline{x}v.\mathbf{0} \\
&\to \mathbf{0} \mid \mathbf{0} \\
&\to \mathbf{0}
\end{aligned}
$$

別の定式化として，各ステップにおいてどのようなイベント（通信や内部動作）が起きるかを $\xrightarrow{\pi}$ というかたちのラベルつき遷移で記述することもできる．そして，ラベルつき遷移に基づく**双模倣（bisimulation）**により，二つのプロセス記述の同値性を定めることができる．この同値問題は，$!P$ におけるコピーの個数を有限にとどめるならば決定可能であることが示されている．

5-1-5 非同期分散システム

π 計算などの定式化はシステムの理論的な性質を検討するうえでは有用だが，現実の様々な分散アルゴリズムを検討するためには，記述が繁雑となりがちで適さない．

分散システムは歴史的には，単一プロセサのシステムをネットワークによって相互接続することで構築されてきた．このため分散システムのモデルも，逐次計算を実行する**プロセス（process）**の集合と，プロセスどうしを接続するリンク（link，通信路）の集合を合わせたものとして定式化するものが広く使われてきた（図 5･3）．ここではそのようなモデルのうちでも，

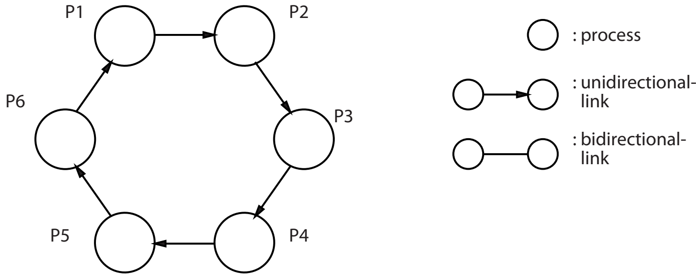

図 5･3　プロセスとリンク

現実のネットワーク通信に近い性質をもつ**非同期分散システム（asynchronous distributed system）**のモデルについて説明する．

非同期分散システムでは，プロセスはプロセス id によって識別される．一つのリンクは二つのプロセスを結ぶ（point-to-point）通信路であり，1 方向の場合と双方向の場合がある．リンクに対して送信命令を出したプロセスは，受信が完了するまで待つ必要がないものとし（non-blocking send，非ブロッキング送信），リンク中にあるメッセージの追い越しは起こらないものとする（FIFO）．

PRAM や LogP のような同期型の並列システムと異なり，ここでは各プロセスの進行に関して次のように扱う．

- プロセスの実行速度に対してとくに仮定を設けない．ただしプロセスの実行は**公平（fair）**である．すなわち，プロセスがある命令を実行しようとした場合，その命令は有限時間内に実行される．
- 通信遅延に対して特に仮定を設けない．ただし通信遅延は有限であるものとする．
- 各プロセスは**局所時計（local clock）**をもつが，プロセス間の局所時計の差についてはとくに仮定を設けない．

このようなモデルに基づくアルゴリズムの記述は，通常の擬似コード命令群にメッセージの送信と受信を表す命令「`send`（値, ポート）」「`receive`（変数, ポート）」を追加したもので記述できる．ここでポートはリンクの端点を識別する名前である．

アルゴリズムの実行は，一つまたは複数のプロセス（開始者，initiator）が自身の擬似コードを自発的に実行を開始することではじまる．それ以外のプロセスは，ほかのプロセスからメッセージを受信することで開始される．アルゴリズムの記述例などについては後の節で示している．

■参考文献

1) David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, Thorsten Von Eicken, “LogP: Towards a realistic model of parallel computation,” ACM SIGPLAN Notices, vol.28, issue 7, pp.1-12, 1993.

2) Michael J. Flynn, Kevin W. Rudd, “Parallel Architectures,” ACM Computing Surveys, vol.28, no.1, pp.67-70, 1996.
3) Robin Milner, David Walker, “A Calculus of Mobile Processes, I,” Information and Computation, vol.100, issue 1, pp.1-40, 1992.
4) Davide Sangiorgi, David Walker, “The π-calculus: A Theory of Mibile Processes,” Cambridge University Press, 2001.

■6 群 - 3 編 - 5 章

5-2 並列アルゴリズム

(執筆者：久野　靖)[2012 年 7 月 受領]

5-2-1 並列アルゴリズムの効率と最適性

本節では PRAM モデルの上の並列アルゴリズムとその効率について基本的な概念と考え方を紹介する（並列アルゴリズムはそれ自体非常に広範囲な分野であるため，より詳しくは[2)]などの専門書を参照されたい．参考文献 1) など公開されている講義資料で参考になるものもある）．先に述べたように PRAM モデルは同期式であり，各プロセッサ間で待ち合せを記述する必要はない．PRAM アルゴリズムの記述に固有の記法として，一般的な擬似コードに加えて次のものを追加する．

for $i \in SET$ **pardo** $STAT$

これは，集合 SET に含まれる各 i について並列に $STAT$ を実行することを表す．

ところで，例えば上記で $STAT$ の中身が 1 ステップで実行できるものとして，実際に全体の処理が 1 ステップで済むかどうかは，プロセッサ数が集合の大きさ $|SET|$ だけあるかどうかによって変わってくる．そこで，PRAM アルゴリズムについて検討する場合は，問題の大きさ n に対応して以下の指標群を併せて考える．

- プロセサ数: $P(n)$
- 所要時間: $T(n)$
- コスト: $C(n) = T(n) \times P(n)$
- ワーク: $Work(n)$ = オペレーションの総数

特に**コスト**（**cost**）は，「アルゴリズム実行のために占有しているプロセッサ数と時間の積（すなわち資源量）」であり，例え所要時間 $T(n)$ が小さくても余分の資源を必要とするようなアルゴリズムをチェックするという点で重要な指標である．言い換えれば，オペレーションの総数である**ワーク**（**work**）と比して余分なプロセサ数や時間を消費しないことが重要である．

例えば簡単な例として，配列 $A[1..n]$ にある n 個の値の総和を求める，次の EREW PRAM アルゴリズムを考える．

for $i \in \{1..\log n\}$**do**
　for $j \in \{1..\frac{n}{2^i}\}$ **pardo** $A[j] := A[2j-1] + A[2j]$

このアルゴリズムでは，外側ループの 1 周回目で $A[1] + A[2]$ を $A[1]$ に，$A[3] + A[4]$ を $A[2]$ に，というふうに二つずつのペアの和を求めて $A[1..\frac{n}{2}]$ に入れる．次の周回ではこれらのペアの和を $A[1..\frac{n}{4}]$ に入れる．これを繰り返していくことで，$\log n$ 周回目に $A[1]$ に総和が求まる．

このアルゴリズムについて各指標を見てみよう．まず，ワークは $\frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{n} = n \subset O(n)$

となる．ここで十分な数のプロセッサがある場合，つまり $P(n) = \frac{n}{2}$ の場合について考えてみると，$T(n) = \log n$ となり，したがって $C(n) = T(n) \times P(n) = \frac{n \log n}{2} \subset O(n \log n)$ となる．しかしこの場合，図 5･4 からも分かるように，全部のプロセッサが動作するのは 1 周回目だけであり，あとは遊んでいるプロセッサがどんどん増えていく．

ここで，$P(n)$ をもっと小さくすることを考える．$P(n)$ が小さいときは，各周回の処理を $\lceil \frac{n}{P(n)} \rceil$ 回に分けてやる必要があるので $T(n)$ は多くなるが，その代わり途中までは遊んでいるプロセッサが少なくなる．具体的には $T(n) = \lceil \frac{n}{2P(n)} \rceil + \lceil \frac{n}{4P(n)} \rceil + \cdots + \lceil \frac{n}{nP(n)} \rceil \subset O(\frac{n}{P(n)} + \log n)$ となる．ここで，$P(n) = \frac{n}{\log n}$ のように決めるとすると，$T(n) \subset O(\log n + \log n) = O(\log n)$ であり，$C(n) = T(n) \times P(n) \subset O(\log n \times \frac{n}{\log n}) = O(n)$ となる．このように，適切なプロセッサ数を選ぶことで，アルゴリズムのコストを低くすることができる．

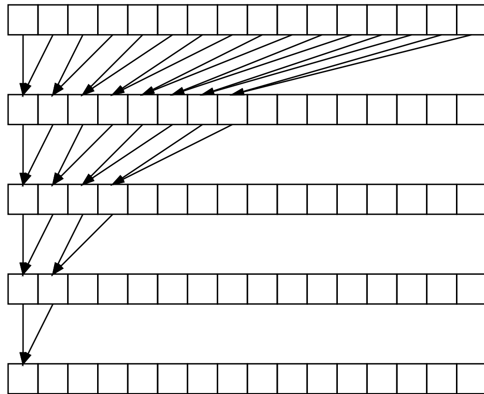

図 5･4　PRAM による総和アルゴリズム

では，「どれくらい」低ければよいだろうか? ここで，逐次アルゴリズムについては永年研究がなされていて多くの知見が得られていることから，知られている最速の逐次アルゴリズムの時間計算量 $T^*(n)$ を基準とすることを考える．ある並列アルゴリズムについて，$Work(n) \subset O(T^*(n))$ であるなら，そのアルゴリズムは最善の逐次アルゴリズムよりも余分な仕事をしていないといえる．これを**ワーク最適（work optimal）**と呼ぶ．そして，同様に，$C(n) \subset O(T^*(n))$ であるなら，そのアルゴリズムは最善の逐次アルゴリズムよりも余分な資源を占有していないといえる．これを**コスト最適（cost optimal）**と呼ぶ．これらを先の例にあてはめると，総和アルゴリズムはワーク最適であり，かつ，$P(n) = \frac{n}{\log n}$ のときにコスト最適となる．

5-2-2 並列アルゴリズム: 最大値アルゴリズム

本節では配列中の最大値を求める問題を例に取り上げ，様々な並列アルゴリズムやその考え方を紹介する．

(1) アルゴリズム 1: 2 分木

前節の総和プログラムの加算を最大値演算に変更することで，全要素の最大を求めることができる．このアルゴリズムは前節の分析がそのまま当てはまるので，EREW PRAM で動作し，$P(n) \subset O(n/logn)$，$T(n) \subset O(logn)$，$C(n) \subset O(n)$，$Work(n) \subset O(n)$ となり，ワーク最適，コスト最適である．

(2) アルゴリズム 2: 最速アルゴリズム

n^2 個のプロセッサを使用し，すべての $A[i], A[j]$ の組を同時に比較することで定数時間で終了するアルゴリズムを考える．作業用に 2 次元配列 $B[1..n, 1..n]$ を使用し，最終結果は $M[1..n]$ において最大値に対応する i についてのみ $M[i]$ が 1 であることで表す．

for $i, j \in \{1..n\}$ **pardo if** $A[i] \geq A[j]$ **then** $B[i, j] := 1$ **else** $B[i, j] := 0$
for $i \in \{1..n\}$ **pardo** $M[i] := 1$
for $i, j \in \{1..n\}$ **pardo if** $B[i, j] = 0$ **then** $M[i] := 0$

終わった後でもし $M[i]$ に 0 が書き込まれていなければ，$B[i, *] = 1$ だったということであり，これはつまりすべての j について $A[i] \geq A[j]$ だったということなので $A[i]$ が最大値となる．このプログラムでは最後の **pardo** において，$B[i, *] = 0$ であるような $M[i]$ に各プロセッサが一斉に 0 を書き込むので，common CRCW PRAM 上で動作する．そのコストは $P(n) \subset O(n^2)$，$T(n) \subset O(1)$，$C(n) \subset O(n^2)$，$Work(n) \subset O(n^2)$ であり，ワーク最適でもコスト最適でもない．

(3) アルゴリズム 3: 2 重対数深度木

アルゴリズム 2 は $T(n) \subset O(1)$ と非常に高速だがプロセッサ数が n^2 個と非常に多数必要となる問題があった．そこで考え方を変えて，全体をそれぞれ $\sqrt{n}$ 個の要素を含む $\sqrt{n}$ 個の部分木に分け，それぞれの最大を求めたとすると，その $\sqrt{n}$ 個の最大から全体の最大を求めるのは，アルゴリズム 2 を使えば $(\sqrt{n})^2 = n$ 個のプロセサで $O(1)$ でできることになる．では，$\sqrt{n}$ 個ずつの要素の最大を求めるには? それは $\sqrt{\sqrt{n}}$ 個ずつの要素を含む $\sqrt{\sqrt{n}}$ 個の部分木それぞれの最大を求めてから，それらの最大を求めればよく，それには $\left(\sqrt{\sqrt{n}}\right)^2 = \sqrt{n}$ 個のプロセサでアルゴリズム 2 を実行すれば $O(1)$ で求まる（それを $\sqrt{n}$ 組並列に行うことになる）．

このような分割を繰り返していくと，最後は 2 個の要素を含む部分木それぞれの最大を求めることになるので，これは 1 個のプロセサで $O(1)$ で可能である．その状態までの分割の段数（木の深さ）を d とすると，$n^{(\frac{1}{2})^d} = 2$ より $(\frac{1}{2})^d \log n = 1$，したがって $\log n = 2^d$ だから $d = \log\log n$ となる．これを **2 重対数深度木（doubly logarithmic-depth tree）**ないし DLT と呼ぶ．DLT を用いたこのアルゴリズムのコストは，$P(n) \subset O(n)$, $T(n) \subset O(d) = O(\log\log n)$, $C(n) \subset O(n \log\log n)$（ワークも同じ）となる．したがってこのアルゴリズム（アルゴリズム 2 を利用するので common CRCW PRAM 用となる）は，かなり高速だが，コスト最適ではないことになる．

(4) アルゴリズム 4: Accelerated Cascading

次に，コスト最適だが遅いアルゴリズム 1 と，コスト最適でないが高速なアルゴリズム 3 を組み合わせて使うことを考える．具体的には，ある整数 m を決めて，最初にすべての要素

を $\frac{n}{m}$ 個ずつに分けてそれぞれに対してアルゴリズム 1 を適用し，その結果求まった m 個の最大値に対してアルゴリズム 3 を適用する．

ここで，$m = \frac{n}{\log\log n}$ としたとすると，アルゴリズム 1 を m 個並列に実行する部分については，$\frac{n}{m}$ 要素に対してコスト最適/ワーク最適となるようアルゴリズム 1 と同様にプロセッサ数を選び，$P(n) \subset O(m \times \frac{n/m}{\log n/m}) = O(\frac{n}{\log\log n} \times \frac{\log\log n}{\log\log\log n}) = O(\frac{n}{\log\log\log n})$ とすると，$T(n) \subset O(\log\log\log n)$ となる．一方，アルゴリズム 3 を m 要素に対して実行する部分では，$P(n) \subset O(m) = O(\frac{n}{\log\log n})$，$T(n) \subset O(\log\log m) = O(\log(\log n - \log\log\log n)) = O(\log\log n)$ となる．これらを合わせると，$P(n) \subset O(\frac{n}{\log\log n})$，$T(n) \subset O(\log\log n)$ より，$C(n) \subset O(n)$（ワークも同じ）となり，コスト最適，ワーク最適でありつつアルゴリズム 1 よりも高速なアルゴリズムが得られたことになる．

このように，コスト最適だが速くないアルゴリズムを用いてデータを統合した後に速いアルゴリズムに切り替えることで全体としてコスト最適で速いアルゴリズムを得る手法を **accelerated cascading** と呼ぶ．

■参考文献

1) 井上美智子, “計算理論 II（講義資料）,” http://fan.naist.jp/ kounoe/lecture/compII/（2012 年 1 月確認）
2) 宮野　悟, “並列アルゴリズム — 理論と設計 —,” 近代科学社, 1993.

■6 群 - 3 編 - 5 章

5-3 分散アルゴリズム

（執筆者：久野　靖）[2012 年 7 月 受領]

5-3-1 分散アルゴリズムの評価指標

本節では非同期分散システムを前提として，分散アルゴリズムの例とその効率について基本的な事柄を解説する．この分野も非常に多くの内容を含んでおり，より詳しくは参考文献 2) などの文献を参照されたい．また，本節では通信路やノードの故障がない場合を前提としている．信頼のおけない通信路があったり，**ビザンチン合意（Byzantine agreement）**問題に代表されるような，ノードのいくつかが故障・誤動作している場合も含めた問題については，6 群 7 編（分散システムのディペンダビリティ→ 6 群 7 編 2 章）を参照のこと．

分散アルゴリズムの評価指標としては，**通信複雑度（communication complexity）**と**時間複雑度（time complexity）**とが用いられる．前者はアルゴリズム実行終了までに送受されるメッセージの量を表し，更にメッセージの個数を用いる**メッセージ複雑度（message complexity）**とメッセージの総ビット数を用いる**ビット複雑度（bit complexity）**の 2 種類がある．後者は「プロセスが m_1 を受信したのち，その結果に基づいて m_2 を送信する」という関係にあるメッセージの連鎖のうち最長のものの長さを意味する．

メッセージ複雑度はアルゴリズムが完了するまでのメッセージの送受信を互いに重ならないという条件下で実行したステップ数，時間複雑度は逆に互いに最大限重ならせるという条件下で実行したステップ数だと考えることができる．例えば図 5･5 において，網掛けのプロセスがもつ情報をメッセージによりほかの全プロセスに伝達するとする．メッセージを複数並行して送ることにより，時間複雑度は 3 となるが，メッセージの総数は 5 個なので，メッセージ複雑度は 5 となる．

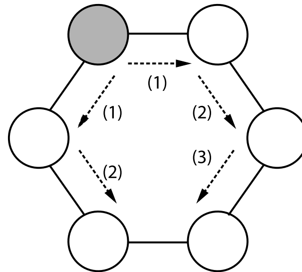

図 5･5　メッセージ計算量と時間計算量

これらはいずれも，メッセージのみに着目していてプロセス内部の計算時間を無視しているが，これは CPU 間の通信時間が CPU 内部の命令実行時間よりもはるかに大きいという，一般的に見られる状況をモデル化している．ただしこれについては，プロセス内部で複雑な計算を行う場合や非常に通信時間が短いシステムのモデルとして適切でないとする批判もある．

5-3-2 分散アルゴリズム: リーダ選挙問題

本節では単方向リングにおけるリーダ選挙問題を例に取り上げる．**リーダ選挙問題（leader election problem）**は，各プロセスがリーダの id について合意することを目的とする．ここでは，リンクの配線は初期状態から変化することはなく，プロセスやリンクの故障もないものとする（静的問題）．更に，各プロセス i の ID を I_i とし，各プロセスが最大の I_i を変数 Max に格納して停止するようにする（この番号がリーダの番号となる）．また，各プロセスがもつ一つずつの入力端点と出力端点を IN，OUT で表す．

（1）アルゴリズム 1: 素朴版

素朴版のリーダ選挙アルゴリズムを以下に示す．先述のように分散アルゴリズムは一つ以上の開始者から実行を開始するが，ここでは開始者であるか否かを論理値 *initiator* で表すものとし，いずれかによる実行内容の違いを冒頭の if 文で表現している．

if *initiator* **then**
  $Max := I_i; send(I_i, OUT)$
else
  $receive(M, IN); Max := max(I_i, M); send(I_i, OUT); send(M, OUT)$
endif
loop
  $receive(M, IN)$
  if $M = I_i$ **then** $STOP$ **endif**
  $Max := max(Max, M); send(M, OUT)$
endloop

すなわち，このアルゴリズムでは開始者はまず自分の ID を隣に送信し，ほかのプロセスは最初に受け取った ID と自分の ID をともに隣に送信する．続いて以後，繰り返し受け取った ID を隣に中継するが，この間に遭遇した ID の最大値を Max に保持し続ける．そして，自分の ID が 1 周して戻ってきたときに実行を停止するが，そのときにはすべての ID を自分が中継し終えているので，その最大値が全 ID の最大値ということになる．

しかし，あるノードが停止したときに，まだ中継すべきメッセージが残っていることはないのかという危惧があるかもしれない．実際には図 5･6 のように，例えば自分がノード C だとしたら，自分の ID より「後に」回っているメッセージは自分のメッセージより多くのリンクを経てきていて，1 周した時点で当該ノードに回収される．このため，停止したときに中継すべきメッセージが残っていることはない．

このアルゴリズムは，n 個のノードが自分のものも含む n 個のノード番号のメッセージを隣に送信するため，メッセージ複雑度が $O(n^2)$ となる．

（2）アルゴリズム 2: Chang-Roberts

素朴版のアルゴリズムを改良する方向として，中継する必要のない（より大きい値があると分かった）値は隣に送らないというものがある．以下は Chang と Roberts[1)]の提案によるアルゴリズムの[2)]による改訂版である．このアルゴリズムでは，各プロセスは自分の ID と受け取った ID の大きい方を覚え，かつ隣に中継する．ある ID が 1 周して元のプロセスに戻っ

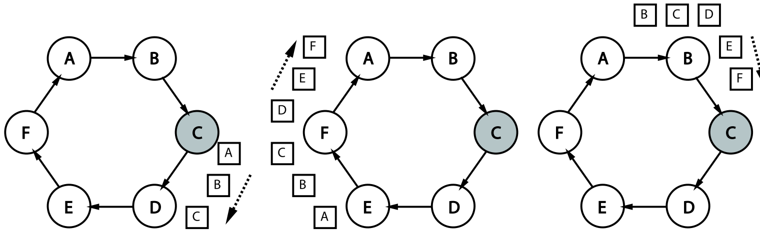

図 5･6　リーダ選挙問題: 素朴版アルゴリズム

てきたら，その値が最大値であると分かる．分かったら，そこで終了を示すメッセージを更に 1 周させる．すると各ノードはこれまでに遭遇したメッセージの最大が最大値だと分かるので直ちに停止すればよい．

if *initiator* **then**
　$Max := I_i; send(I_i, OUT)$
else
　$receive(M, IN); Max := max(I_i, M); send(Max, OUT)$
endif
loop
　$receive(M, IN)$
　if $M = I_i \,\|\, M =$ "$FINISH$" **then** $send($"$FINISH$"$)$; $STOP$ **endif**
　if $M > Max$ **then** $Max := M$; $send(Max, OUT)$ **endif**
endloop

このアルゴリズムのメッセージ計算量は，最善の場合（たまたま最大値の ID をもつプロセスだけが開始者だった場合）には $O(n)$，最悪の場合（すべてのプロセスが開始者でそこからの連鎖が平均 $\frac{n}{2}$ ノード先まで進む場合）には $O(n^2)$ であるが，平均的には $O(n \log n)$ となることが示されている．

(3) アルゴリズム 3: Peterson

別の考え方として，局所的なメッセージの交換に基づいて最大ではないと分かったプロセス ID をできるだけ速く「除外」するという考え方に基づくアルゴリズムがある（Peterson[3] によるアルゴリズムの変形版[2]，図 5･7）．

if *initiator* **then**
　$T1 := I_i$; $send(T1, OUT)$; $receive(T2, IN)$
else
　$receive(T2, IN)$; $T1 := I_i$; $send(T1, OUT)$
endif;
while $T1 \neq T2$ **do**

$send(T2, OUT);\ receive(T3, IN);$
if $T2 < max(T1, T3)$ **then**
loop
$receive(M, IN);\ send(M, OUT);$
if $M =$ "$FINISH$" **then** $STOP$ **endif**
endloop
endif
if $T2 > max(T1, T3)$ **then** $T1 := T2$ **endif**;
$send(T1, OUT);\ receive(T2, IN)$
endwhile;
$Max := T1;\ send($"$FINISH$"$, OUT);\ STOP$

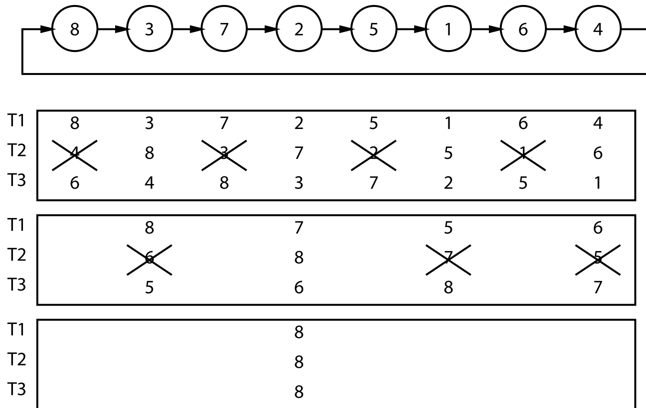

図 5･7　リーダ選挙問題: Peterson **アルゴリズム（例は参考文献** 2) **による）**

このアルゴリズムでは，まず自分の ID，左隣の ID，更に左隣の ID を変数 $T1 \sim T3$ に入れる．ここで，中央の値 $T2$ が「自分の担当する ID」であり，これが $T1$ または $T3$ いずれかより小さければ，$T2$ が最大ということは起こらないので，そのような $T2$ をもつプロセスは「除外」モードとなり，以後単にメッセージの中継のみを行う．残ったプロセスについては，$T2$ が $T1$，$T3$ のいずれよりも大きければそれが新たな $T1$（そうでなければ $T1$ は同じまま)，新たな $T2$ と $T3$ は左隣の $T1$，$T2$ をそれぞれ受け取ることで，再び自分の担当 ID$T2$ とその前後が $T1 \sim T3$ に入った状態になる．最後は一つのプロセスを除くすべてが「除外」となり，自分が送った $T1$ が $T2$ に受け取られて等しくなるのでそこで "$FINISH$" を送信して終了し，このプロセスの $T1$（$= T2$）が全体の最大となる．

このアルゴリズムではラウンド（while ループ内部の実行）1 回につき，少なくとも半分のプロセスが「除外」になるため，ラウンドの回数は $O(\log n)$ となり，1 ラウンドにつき各ノード 2 個，全体で $O(n)$ のメッセージが送られるため，メッセージ計算量は $O(n \log n)$ となる．

■参考文献

1) Ernest Chang, Rosemary Roberts, “An improved algorithm for decentralized extrema-finding in circular configurations of processes,” Communications of the ACM, vol.22, no.5, pp.281-283, 1979.
2) 亀田恒彦, 山下雅史, “分散アルゴリズム,” 近代科学社, 1994.
3) G.L. Peterson, “An $O(nlogn)$ unidirectional algorithm for the circular extrema problem,” ACM Transactions on Programming Languages and Systems, vol.4, no.4, pp.758-762, 1982.