解　説

PRAM 上の並列アルゴリズム†

岩　間　一　雄††

1. はじめに

ある共通テストに5万人が受験したとしよう．結果は各受験生あてに，その受験生の得点と，その受験生の“次の順位”の受験生の電子メールアドレス（電話番号ではまずい理由を後で述べる）が知らされてくる．そこで，受験生は協力して，最高点と自分の順位を全ての受験生が知ることができる方法を考えることになった．すぐ思いつくのが次のような逐次法である．1番の順位の人（1番に限って採点者から知らされると仮定しても良いし，また，全ての人が次の人へ同時にメールを出せばメールが来なかった人が1番である）が2番の人へ自分の点数（最高点）と順位（1番）を知らせる．以後，メールが来たものから順々に最高点と自分の順位（前の人の順位＋1）を次の人にメールで知らせていく．この手法は単純ではあるが，特定の人にメールが集中しないとか，非同期で良い（メールが来るまで待っていれば良い）といった良い性質をもっていることに注意されたい．しかし，効率の点で実際的でないのは明らかで，1日10人進むとしても5000日もかかるのである．

そこで，次のような並列化を考えた．(i)第1のフェーズでは全ての受験生が同時に次の受験生に自分のアドレスを伝える（ただし，1番の人だけは以前と同様自分の点，つまり最高点を伝える）．なお，電話ではこのような同時伝送が（お話中になってしまって）できない．(ii)第2のフェーズでは全ての受験生が前のフェーズで受けとったアドレス（または最高点）を次の人に回す．これを第10フェーズまでやれば，各人は自分より10位上の人のアドレスを知ることができる．（ただし，1位から11位の人は最高点と自分の絶対順位が分かる．）次に，各自，自分より10位上の人に自分のアドレスを伝えた後で，上記の逐次法を10ストリーム並列に適用する．（たとえば，5位の人からは15位，25位の順にメールが出される．）こうして約10倍の速度が得られるが，それでもまだ500日で現実的ではない．そこで，上のストリーム数をより大きなkまで上げようとすると，今度は最初のk位上の人のアドレスを知る部分が問題になってきて，結局，簡単な計算で約44日より早くすることはできないことが分かる．

順位のほうは後回しにして(4.)，とりあえず最高点のみを伝える第3の方法を以下に与える．再帰的記述を行う．ある時点で，各自が自分より2^i順位の高い人と2^i順位の低い人のアドレスを知っていると仮定しよう．(**図-1**(a)に$i=2$の場合を示す．たとえば，Dが$2^2=4$位下のEのアドレスを知っていることをD→Eで表している．）ただし，1～2^i位までの人は2^i高い人のアドレスの代わりに最高点を，逆に低位の2^i人は2^i低い人のアドレスの代わりに“bottom”という特別な情報を保持する．そこで次の(i)と(ii)を実行する．(i)自分より2^i高い人のアドレスを2^i低い人へ知らせる．たとえば，図-1(a)で，DはCのアドレスをEに知らせる．(ii)自分より2^i低い人のアドレスを2^i高い人へ知らせる．たとえば，図-1(a)で，BがCのアドレスをAに知らせる．こうして，Cは自分より$2^3=8$位上(A)と下(E)のアドレスを知る（**図-1**(b)）．なお，(i)で2^i高い人が存在しない場合は代わりに（知っていると仮定されている）最高点を伝達するし，2^i低い人が存在しない場合は単にメールを出さない．(ii)においても，2^i低い人が存在しない場合はbottomを伝達するし，2^i高い人が存在しなければメールを出さない（**図-1**(c)）．こうして上記の$i=i+1$の状況が作り出せたことが

† Parallel Algorithms over PRAMs by Kazuo IWAMA (Department of Computer Science and Communication Engineering, Kyushu University).

†† 九州大学工学部情報工学科

分かるであろう．最初は，全ての受験生が自分のアドレスを次の順位の人に伝えることにより$i=0$の状況が作れる．$2^{16}=65536$であるから，$i=16$のとき全ての人が最高点を知ることになる．そのための日数は（前と同様1日 10 ステップ進むとすれば）2日ほどで十分である．

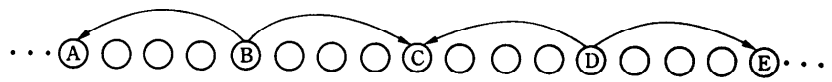
こうして，所要時間が最初の5000日から44日へ，さらに2日へと減少したわけであるが，第3の方式が“並列アルゴリズム”なのだと言われれば，その有効性を納得しないわけにはいかないであろう．第1の方式がまぎれもない直列アルゴリズムである．第2の方式がいわゆる“直列アルゴリズムの並列化”と呼ばれる方式であり，有名なアムダールの法則（並列計算の中に直列に計算する部分があれば，残りの部分に対していかに高度な並列化を行っても少なくても直列部分の時間だけはかかってしまう）を取り込んでいることに注意されたい．

本稿は，並列アルゴリズムについての解説であるが，よく行われる“最新の成果の紹介”ではないことを最初に断わっておく．上記の導入部からも感じていただけるように，目的は“並列アルゴリズムの原理”と“研究の目的およびゴール”を**できるだけ平易に**説明することである．例題は上記例題のみで最後までつきあっていただく．なお，関連する解説・教科書として，参考文献 4)～7) をあげておく．

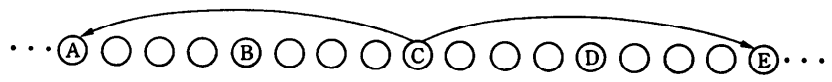
2. 並列乱アクセス機械

単に計算機（直列計算機）といえば，その歴史も長く，多くの人が共通のイメージをもつであろうが，並列計算機となるとそうはいかない．ある人はマルチ（せいぜい 4～8 程度の）CPU のワークステーションを思い浮かべるであろうし，他の人はハイパキューブ型のそれだと思うであろう．最初に明確にしておくべきことは，本稿で扱う並列処理の並列度である．われわれは“超並列”，あるいは，“極並列”（極限まで並列化を進める意味）の世界を対象にする．したがって，プロセッサは入力のサイズに応じていくらでも多く使えると仮定する．**1.** で述べた第3の方式は，各受験生をプロセッサとみたてれば，入力サイズ（受験生の数）nに対してプロセッサn台の並列処理ということになる．

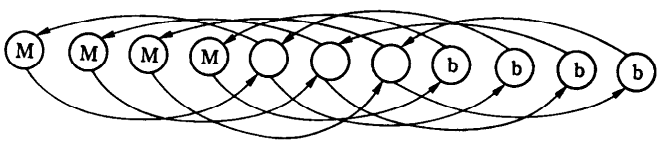
並列の度合いの次は，並列計算機モデルである．多数使う個々のプロセッサを通常の直列計算機とするところまでは異論がなかろうが，それらの間の通信機能には，数多くの異なった方式が提案されている．本稿では，最も強力な通信手段と考えられる理想的な共有メモリ（後述のように各プロセッサからの並列アクセスに対しほとんど制限がない）を有する**並列乱アクセス機械（PRAM）**を使用する．このことによって，われわれは，アルゴリズムを設計する際に，通信に係わるオーバヘッドをあまり意識しなくてよい．逆に，能力の低い通信機能を仮定した場合は，通信のための時間をいかに減らすかがアルゴリズム効率化の鍵になることが多い．こちらのほうが現実には合っているのかもしれないが，アルゴリズムの研究の面からは欠点もある．通信にどうしても時間がかかってしまうために，通信以外の部分の高速化が無意味になる（つまり通信時間が全体の時間を支配し



（a） $2^2=4$ 位上下の人のアドレスを知っている



（b） $2^3=8$ 位上下の人のアドレスを知る



（c） $i=2$ のときの最高点 (M) と bottom (b) を知るプロセッサ

図-1 2^i 位高（低）い人のアドレスを 2^i 位低（高）い人へ伝達する

てしまう）ことが多い．アルゴリズム的興味の度合い（工夫がむくわれる度合い）は，通信に係わる部分よりもそれ以外の部分のほうが格段に大きいというのが筆者の感想である．なお，PRAM をより弱いモデルで一般的に模倣する技術が知られており，後ほど簡単にふれる．

PRAM は多数のプロセッサとして，**乱アクセス機械 (RAM)** を使用し，それらを共有記憶によって結合する．個々の RAM はいわゆる von Neuman 型計算機と同じ構造でレジスタや記憶装置（共有記憶のほかに各自の局所記憶）を有している．通常の(不自然でない)機械語命令を使用できる．各 RAM は特別なレジスタを有しており，そこには常にその RAM のプロセッサ番号（各 RAM に一意で，0 から順に割り付けられる）が入っている．各 RAM は特別な LOAD と STORE 命令によって共有記憶にアクセスするが，そのときの規則によって分類が生じる．**CREW-PRAM** (Concurrent Read Exclusive Write PRAM) は読み出しに関してはまったく制限がないが，書き込みについては同じセルに二つ以上のプロセッサによる同時書き込みを禁止する（そのようなことが生じないようなアルゴリズムを考えなくてはいけない）．**CRCW-PRAM** は同時書き込みも許す．同時書き込みの処理で再び分類が生じ，**ARBITRARY**（どれか事前には分からないプロセッサの書いたデータが生きる）や **COMMON**（全てのプロセッサの書くデータが同一でなければいけない）などがある．一つのセルへのアクセスは読み出しも書き込みも一つのプロセッサに制限するのが **EREW-PRAM** である．

入力は共有記憶の最初の n セルに与えられ，そのサイズ n に対して $P(n)$ 個（プロセッサ番号 0 から $P(n)-1$）のプロセッサが同時に仕事を開始し，$T(n)$ ステップ後に全て停止する．このようなアルゴリズムを，$O(P(n))$ プロセッサ $O(T(n))$ 時間のアルゴリズムと呼ぶ．共に低次の項と比例定数は無視する（$O(8n^2+6n)$ 時間のアルゴリズムと $O(n^2)$ 時間のアルゴリズムとは性能的に等しいとみなされる）．共有メモリのサイズはあまり評価の対象にならないが，プロセッサ数 $P(n)$ と同数（定数倍）かそれ以下使用するものがほとんどである．たとえば，グラフアルゴリズムでプロセッサ数＝枝数に対して，メモリセル数を節点数の 2 乗使うものを見かけることがあるが，そのような場合には必ず明示しなくてはいけない．

前章の方式 3 を PRAM 上の $O(\log n)$ 時間アルゴリズムとして記述してみよう．入力は長さ n の二つの配列 $SCR[j]$ と $NXT[j]$ で与えられる．$SCR[j]$ には受験番号 $j(0\leq j\leq n-1)$ の受験生の得点が入り，$NXT[j]$ には受験生 j の次の順位の人の受験番号（前章のアドレスに対応，ただし最後の人の場合は -1）が入っている．上記約束によれば，たとえば，$SCR[0]$〜$SCR[n-1]$ はメモリのセル 0〜$n-1$，$NXT[0]$〜$NXT[n-1]$ はセル n〜$2n-1$ のように格納される．しかし，配列の種類が有限個の場合は，このような低レベルの約束が重要になる場合は少なく，たいていは記述を省略する．アルゴリズムではほかに 4 個の長さ n の配列 $FWD[j]$, $BWD[j]$, $MAX[j]$, $RNK[j]$ とさらに 2 個のワーク変数 $WKF[j]$, $WKB[j]$ を使う．$FWD[j]$ と $BWD[j]$ は前後の人へのポインタ（それらの人の受験番号が入る）のためのものであり，$MAX[j]$ の全てに $SCR[j]$ の中の最大値が入ったところでアルゴリズムは終了する．$RNK[j]$ は順位のための変数でこの章では使用されない．

アルゴリズム TOP_PASS：以下の同一のプログラムを全てのプロセッサが実行する．

ステップ 1.

p←自分のプロセッサ番号，
$FWD[p]\leftarrow -1$，
$BWD[p]\leftarrow NXT[p]$，
$FWD[BWD[p]]\leftarrow p$，

を実行する．

ステップ 2. もし $FWD[p]=-1$ なら

$MAX[p]\leftarrow SCR[p]$

を実行する．

ステップ 3. 次のサブステップ(i)，(ii)，(iii)からなり，本ステップは $[\log n]$ 回繰り返される．

(i)ポインタの退避．

$WKF[p]\leftarrow FWD[p]$
$WKB[p]\leftarrow BWD[p]$

(ii) $WKB[p]=-1$ なら本サブステップをスキップ．

$FWD[WKB[p]]\leftarrow WKF[p]$

を実行．もし $WKF[p]=-1$ なら

$MAX[WKB[p]]\leftarrow MAX[p]$

をさらに実行する．

（ⅲ）$WKF[p]=-1$ なら本サブステップをスキップ．

$BWD[WKF[p]]\leftarrow WKB[p]$

を実行する．

このように，より厳密なアルゴリズムの形に書いてみるとそれほど分かりやすいものではない．たとえば，ステップ3の（ⅱ）の $FWD[WKB[p]]\leftarrow WKF[p]$ は，$WKF[p]$ に入っている自分（Bとする）より 2^i 位（i はそのときまでの繰り返し回数）上の人（A）のアドレス（受験番号）を，$WKB[p]$ にそのアドレスが入っている 2^i 位下の人（C）の FWD に入れるところまでをBがやることを意味している．前章の説明では，BはAのアドレスをCに送るところまでをやり，Cの FWD の更新はC自身が行うことになっていたが，実質は同じである．ステップ3の1回の繰り返しは定数時間なので，結局全体で $O(\log n)$ 時間で終了する．またモデルは EREW-PRAM で十分である．いくつかの注意を与える．

（1） 問題を復習すると，要するに，**図-2** のように与えられた入力 SCR の中で一番大きな値を MAX に入れればよい．340 が最大であることはすぐ分かるし，それを各 $MAX[j]$ に書き込むためにプロセッサ1個なら時間がかかるかもしれないが，今の場合プロセッサはたくさんあるから，これも瞬時にできる．なんでこんな面倒な作業が必要なのか？ 時折り，単純な誤解や勘違いが生じることがあるが，その原因は，（ⅰ）最大値などの，直列処理では自明な問題を並列処理でも自明と思い込むこと（直列処理における線形時間はそれ以上改良できないベストであるが，並列処理では通常線形より格段に速い $\log n$ 時間などを目標にしている），（ⅱ）図-2 のような規模の小さい状況を眺めたときに，個々のプロセッサが“全ての”データを“都合の良い順に”しかも“瞬時に”読めると錯覚することなどにある．

（2） 上記 WKF などについては各プロセッサに備わっている局所記憶を使っても良い．しかし本稿では必要記憶容量を明確にするため，局所記憶は一切使用しない．

$j=$	0	1	2	3	4	5	6	7
$SCR=$	130	250	69	340	185	98	67	290
$MAX=$ プロセッサ	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7

図-2 TOP_PASS への入力例

（3） 入力サイズ n をいかにして知るか？ 天下り的に全てのプロセッサが知っているという設定もあるが，たとえば，以下のようにして知ることもできる．本例題の場合，入力の与えられるセルは $0\sim 2n-1$ 番地なので $2n$ 番地に特殊な値（たとえば負の最大値）が入っていると仮定する．最初に全ての（無限に多くの）プロセッサが同時に自分のプロセッサ番号のセルとそのセル $+1$ 番地のセルを読めば，$2n-1$ 番のプロセッサが入力の終わりを知ることができ，簡単な計算で n が求まる．後は，CREW-PRAM なら定数ステップで，EREW-PRAM なら $O(\log n)$ ステップで全てのプロセッサに知らせる（(5)参照）．

（4） PRAM は完全同期式である．同期の調整は NOP (No Operation) 命令によって行われるが，アルゴリズムの記述では省略されることが多い．たとえば，上のステップ3の（ⅱ）でサブステップをスキップするといっているが，本当は同じステップ数の NOP 命令を実行する必要がある．たいていの場合は全てのプロセッサがプログラムの同じ場所を実行しているので，このような省略が混乱を生むことはない．しかし同時刻にいくつかのプロセッサがプログラムの異なった部分を実行することも許されており，PRAM が SIMD 型であるという分類[4]には疑問もある．

（5） EREW-PRAM よりも CREW-PRAM，さらにCRCW-PRAMのほうが能力が高いことは明らかであるが，その違いはどの程度なのであろうか．たとえば，$T(n)$ 時間 $P(n)$ プロセッサ（かつ $P(n)$ メモリセル数）の CREW アルゴリズム M が与えられたとき，次のような（あまり効率は良くないが分かりやすい）EREW-PRAM 上での $O(T(n)\log n)$ 時間，$(P(n))^2$ プロセッサ（$(P(n))^2$ メモリセル）の模倣アルゴリズム M' が作れる．M' のメモリに対し2次元に添字 (i,j) をふり（同様にプロセッサも P_{ij} とする），$(i,0)$ を M のセル i に対応させる．M' では $P_{i,0}$ を使って M の P_i の各ステップをそのまま模倣した後で $P_{i,1}\sim P_{i,P(n)-1}$ を使い，$O(\log n)$ 時間かけて，$(i,0)$ の内容を全ての (i,j) $(1\leq j\leq P(n)-1)$ へコピーする．（この方法については，上記 TOP-PASS の簡単な場合にすぎないので説明の必要はなかろう．）M で P_i

がセル l を読むときは，M' では $P_{i,0}$ がセル (l,i) を読む．こうして，同時読み出しを回避することができる．CRCW-PRAM の CREW-PRAM による模倣にも同じアイデアが使える．ところで，キューブ網やシャフル交換網などソーティングが $O(\log n)$ あるいは $O(\log^2 n)$ 時間ほどでできる PRAM 以外のいわゆるネットワークモデルが数多く知られている．EREW-PRAM の各メモリセルをそのようなモデルの各プロセッサに対応させるなら PRAM の1ステップにおけるメモリアクセスの全てをネットワーク上での1回のソーティングによって模倣することができることに気づいてほしい．つまり，これらネットワークモデルは，$O(\log n)$ またはその二乗程度のオーバヘッドで EREW-PRAM を模擬できるのである．

3. 並列化可能性

クラスPに入る，(つまり直列計算機でたとえば $O(n^2)$ のような入力サイズ n の多項式時間で解ける) 問題 Q に対し，多項式プロセッサ数，$O(\log^k n)$ (多項式対数) 時間の PRAM アルゴリズムが存在するとき Q はクラス NC に入るといい，**並列化可能**であると位置付ける．NC はニックのクラスの略称で提唱者 Nick Pippenger からとられた．一応，k は定数であればよいし，プロセッサ数も多項式の範囲なら良い．前章の最後の議論から分かるように，このような大雑把な議論に対しては，上記 PRAM のいくつかの分類は問題にならず，k の 1〜2 の増減で対処できる．さらには，上で述べたような性質をもつ数多くのネットワークモデルを使っても同一の定義になる．しかし，より精密な (たとえば $O(\log n)$ 時間，$O(n^2)$ プロセッサといった) 議論をするときは上記分類は重要で，必ず使用モデルを明確にする必要がある．

クラス NC を並列化可能とする根拠について考えてみよう．たとえば，問題 Q が $O(n^2)$ 直列時間で解けるとする，"並列化" の素朴な概念が，"m 個のプロセッサで m 倍の加速" であることを思いだすなら，Q が $O(n^2)$ 個のプロセッサで定数時間(時間プロセッサ積$=O(n^2)$) で解ければ文句ない．しかし，時間がたとえば $O(\log^3 n)$ になっても，時間プロセッサ積は $O(n^2\log^3 n)$ であり，これはどんなに小さな α に対しても (漸近的には) $n^{2+\alpha}$ よりは小さいので我慢できる範囲内であろう，というのが根拠である．したがって，NC に入るといっても，$O(n^2)$ 直列時間で解ける問題に対して，多項式対数時間を得るのに n^3 個のプロセッサが必要になるなら，この意味での合理性には欠けるといわざるをえない．

並列アルゴリズムの研究が計算複雑度の高い問題に対してではなく，比較的複雑度の低い (多項式時間の) 問題に対して主に行われているのは次のような合理的理由によるものである．クラス NP 以上の難しい問題に対しては通常自明な並列化が存在して理論的興味にとぼしい上，並列化による著しい効果が期待しにくい．たとえば，2^n 直列時間の問題に対し n の多項式といった少ない数のプロセッサを用意しても，実用的な時間で解ける問題のサイズは n の値にしてせいぜい 2〜3 上がる程度かもしれない．逆に複雑度の低い問題に対しては並列化による効果が大きく，実際，問題のサイズの巨大化から並列化の要求が強い回路シミュレーションや方程式の数値解を求める問題の多くがこの範疇に入る．

問題が並列化可能であることが分かったら，次はより効率の良い並列化を追求するのが自然なステップである．ある問題に対し，$P(n)$ プロセッサ $T(n)$ 時間の並列アルゴリズム M が得られたとする．すると $1\leq p\leq P(n)$ の任意の p に対し，p プロセッサ $O(T(n)P(n)/p)$ 時間の並列アルゴリズムがシミュレーションを利用して容易に作れる．特に $p=1$ としたときは $O(T(n)P(n))$ 時間の直列アルゴリズムを意味する．もしこの，時間とプロセッサ数の積が，知られている最良の直列アルゴリズムの時間と等しいなら M は**最適である**といわれる．最適なアルゴリズムは，プロセッサをまったく無駄使いしていない．つまり，正確にプロセッサの台数分の加速が得られているという意味で，理想的な並列アルゴリズムである．

$O(\log^2 n)$ 時間 $O(n/\log^2 n)$ プロセッサのアルゴリズムと，$O(\log n)$ 時間 $O(n/\log n)$ プロセッサのアルゴリズムは共に明らかに最適である (時間プロセッサ数の積が n の線形になる)．しかし，上記の議論を適用するなら前者は後者からシミュレーションを利用して簡単に作れることに注意されたい．最適でかつ時間がそれ以上改良できない並列アルゴリズムは直列・並列を超えた "究極のアルゴリズム" と位置付けることができる．

4. リストランキングアルゴリズム

2. のアルゴリズム TOP_PASS は，より形式的に述べれば，長さ n の一次元リスト ($NXT[j]$) の先頭のノードが有する情報（最高点）をリスト上の全てのノードに伝達する $O(\log n)$ 時間 $O(n)$ プロセッサのアルゴリズムである．本章では，リスト上の各ノードの先頭からの距離（つまり各受験生の順位）を計算する問題（リストランキング問題）に対する同様の性能のアルゴリズム LIST_RANK を与える．TOP_PASS が十分理解できていれば LIST_RANK の開発はそれほど難しくない．TOP_PASS のステップ3の，繰り返しが $i+1$ 回目のとき（最初は $i=0$，つまり1回目）の状況を復習しよう．なお，プロセッサ番号 p のプロセッサを単に“プロセッサ p”と呼び，各配列の p 番目の要素，たとえば $WKF[p]$ を“プロセッサ p の WKF”と呼ぶ場合があることを断わっておく．

（i） プロセッサ p の WKF には，p より 2^i 上のノード番号が入っている．ただし，そのようなノードがない，つまり自分の順位が 2^i までなら -1 が入る．

（ii） p の WKB には 2^i 下のノード番号（または -1）が入る．

そこで，次の(iii)を追加してみよう．

（iii） もし，p の WKF が -1 なら p の RNK に p の順位が入っている．

まず，初期状況（$i=0$）の構築については，最上位のプロセッサ(TOP_PASS のステップ2で自分の FWD が -1 のプロセッサ）が自分の RNK 1を入れればよい．i から $i+1$ への更新は TOP_PASS のステップ3に次の操作を追加することによって実現できる．

> 自分の WKF が -1 から何も行わない．
> そうでないなら $WKF[WKF[p]]$ をみて，
> -1 なら $RNK[WKF[p]]+2^i$ を自分の RNK に入れる．

本アルゴリズムの正しさの証明と，その（TOP_PASS 程度のより厳密な）記述は読者への宿題としたい．

LIST_RANK をさらに変更することにより，プレフィクスサムと呼ばれる同様に重要度の高い問題に対するアルゴリズム PRE_SUM が作れる．PRE_SUM では新たな配列 $SUM[p]$ を用意し，そこに自分より上の順位の人の得点合計（リスト構造の整数データの先頭から各ノードまでの和）を入れることを目標とする．したがって，求まった後で，(SUM+自分の得点）を自分の順位で割れば自分以上の順位の人の平均を知ることができるし，最後の人のそれは全体の平均になる．上記(i)〜(iii)に次の(iv)を追加する．

（iv） p の WKF が -1 なら p の SUM には p より順位が上の人全体の得点合計が，そうでないなら p の直前 2^i 人（(p の順位 -1) 位から (p の順位 -2^i) 位までの人）の合計が入る．

初期状況の構築（ただし，0人の和は0）や i から $i+1$ への更新作業などは LIST_RANK の簡単な応用でできる．

以上 $O(\log n)$ 時間 $O(n)$ プロセッサのアルゴリズム TOP_PASS, LIST_RANK, PRE_SUM を与えた，しかし，これらは全て最適ではない．なぜなら，いずれの場合にもほとんど自明な $O(n)$ 時間の直列アルゴリズムが存在するからである．そこで，これらの改善を試みよう．時間をより速く（たとえば $O(1)$ に）することは難しそうなので，プロセッサ数を $O(n/\log n)$ に減らすことを考える．$O(n)$ プロセッサの場合は全てのノードに1台ずつのプロセッサが行き渡ったわけであるが $O(n/\log n)$ プロセッサの場合は，直観的には，$\log n$ ノードに対して1台のプロセッサしか割り当てることができない．そこで，たとえば最後の PRE_SUM の場合，次のような方法はどうであろうか．

（i） プロセッサは1位，($\log n+1$) 位，($2\log n+1$) 位，…というように，$\log n$ 位飛ばしで均等に割り当てる．プロセッサが割り当てられたノードを代表ノードと呼ぶ．

（ii） 各プロセッサは自分担当の代表ノードから下位（自分のノードも含めて）$\log n$ 個のノードを配列 NXT を利用してたどりながらそれらのノードの SCR の値を直列演算で合計する ($O(\log n)$ 時間)．

（iii） (ii)で計算された値を代表ノードに置き，代表ノードのみからなるリスト構造を作りあげ，サイズ $n/\log n$ の入力として，PRE_SUM をそのまま適用する ($O(\log(n/\log n))=O(\log n)$ 時間)．

(iv) 各代表ノードより上位の(たとえば(2log n+1) 位の代表ノードだったら1位から2log n 位までの) 総合計が求まるので，再び(ii)と同様の直列アルゴリズムによって，代表以外のノードまでの合計を求める．

つまり，プロセッサが足りない分を各プロセッサによる単純な直列計算による前後処理で対処しているわけである．重要な点は(ii)と(iv)の直列計算に必要な時間が(iii)の(メインの)並列計算の部分の時間と釣りあっていることであって，われわれの約束のもとでは $O(\log n)$ 時間を3回実行しても依然として $O(\log n)$ 時間である．(ii)と(iv)を具体的にどう実現するかはほとんど自明であろう．(iii)も，NXT のポインタの付け変えなどが必要になってくるが，ここまで読み進んでくださった読者にとっては難しいことは何もないであろう．(i)は，単に自分のプロセッサ番号を $\log n$ 倍してその次の順位のノードを見に行けばよい．こうして，最適アルゴリズムが比較的簡単にできてしまったようにみえるが，本当だろうか．実は重大な見落しを含んでいる．一見単なる前処理に見える(i)の説明がまったく不十分だからである．再確認であるが，(i)はプロセッサを“順位”に対して均等に割り当てることを要求している．その順位の計算を忘れているではないか．

仮に順位 RNK がすでになんらかの方法で計算されている（あるいは与えられている）と仮定しても上の説明は依然として不十分である．(**2.** で注意した錯覚を思いだして欲しい．各プロセッサ p はどのようにして ($p\log n$+1) 位のノードを“すばやく”見に行けるのであろうか．) 以下のように修正する．プロセッサ p は直接**順位が** ($p\log n$+1) のノードに割り当てるのではなく，とりあえず，**ノード番号が** ($p\log n$+1) のノードに割り当てるのである．各配列はもちろんノード番号順に整列しているので，これなら各プロセッサが自分に割り当てられたノード（に対する配列要素）に自明な方法でアクセスできる．

$IND[p]$ という新たな配列を用意する．各プロセッサは自分のノード(たとえばノード番号 j=2log n+1) から log n 先（ノード番号 j=3log n) までを j を増やしながら順々に見ていき，各 j に対し，$IND[RNK[j]]$ に j を入れるという作業をしていく．$O(\log n)$ で完了した後には各 $IND[j]$ には j 位のノードのノード番号が入っている．こうして，プロセッサは j 位のノードを見たいとき，$IND[j]$ にアクセスしてそのノードの“位置”を知ることができる．

したがって，RNK の計算が $O(\log n)$ 時間 $O(n/\log n)$ プロセッサでできれば目標達成である．しかし，これはやさしくない．**図-3** にノード数9の場合の例を与える．矢印は NXT のポインタを示す．ノード番号順に均等にプロセッサを配置すると，順位としては，2位 (P_2)，4位 (P_0)，8位 (P_1) のノードに配置されることになる．そこで，各プロセッサは初期配置されたノードから NXT に従って見ていき，次にプロセッサが配置されたノードの直前までに対して直列処理を行うという方法が考えつくであろう．図の例の場合，P_0 は4個，P_1 は2個，P_2 は（特別の場合として，1位のノードも含め）3個のノードを処理することになる．このように順位の連続した何個かのノードをひとかたまりにできれば，後は前述の直列処理方式の併用によって RNK を計算できることはすぐ分かる．問題は“ひとかたまり”の大きさのバランスであるが，上の例でもみられるように，さらには乱数などを利用すれば，それほどひどくはならないのではなかろうか．

このような論述は実際的場面ではおおむね正しい．しかし，**2.** で与えたように，今われわれが使用している計算時間の評価方式は，全ての入力に対しその時間内に計算が終わること，すなわち，最悪の場合の評価なのである．(このルールの善し悪しについてはもちろん大きな論争がある．) したがって，たとえば上で述べた初期配置を行い，できたかたまりに対して素直に直列計算を適用すれば，明らかに都合の悪い入力（たとえば，P_0, P_1, P_2 に1, 2, 3位のノードが配置される）が存在し，計算時間は $O(n)$ まで増加してしまう．並

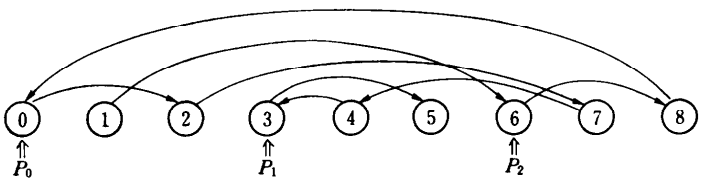


図-3 プロセッサの初期配置

列アルゴリズムとしては無意味なものになってしまうのである.

完全な解決[3]は本稿の範囲を越えるので,以下にそのアイデアのみをあたえる.

(1) 最初は,上と同様,プロセッサをノード番号について等間隔に配置する.こうしてノード番号について連続する(順位についてはなんらそのような関係のない)$\log n$ 個のノードが1個のプロセッサの担当になる.

(2) 各プロセッサは最初に割り当てられたノード(N_0 とする)から順位のポインタに従ってノードをたどりながら,"かたまり"を作っていく.かたまりは"最後尾"($=N_0$),"先頭"(=今見ているノード),"大きさ"(かたまり中のノードの個数)などのデータを有し(図-4),定数時間で処理できる(定数時間で1ノード分前へ進める)範囲で各ノードのこれらのデータを更新する.たとえば,最後尾ノードは常にこれら全ての最新データを保持できる.大きさが $\log n$ になればそれは"最終的なかたまりとして確定"させる.最初の割当が偶然順位についても等間隔になれば,全てのかたまりが同時に確定することになる.これは理想的な場合で,アルゴリズムはほとんど終了である.

(3) もちろんいつもそううまくはいかず,あるかたまりを成長させていく過程で,サイズが $\log n$ になる前に,別のプロセッサが成長させているかたまりに"追突"してしまう.一般的には,多くのプロセッサが球突き状に同時に追突することになるが,その場合,巧妙な手法[2]によって,定数時間でプロセッサの"間引き"をする.これは,図-5 に示すように,プロセッサの連鎖に対して,2以上 $\log\log n$ 以下の非決定的な間隔で"切れ目"を入れることによる.自分の直後が切れ目に当たったプロセッサ(図の P_1, P_4, P_8)以外は自分のかたまりを成長させることをあきらめ,最初に決められた自分の担当ノードの次を見に行く.そのノードがすでに処理されている(あるかたまりの中に入っている)ならさらに次へ行く.間引きされなかったプロセッサ(P_1, P_4, P_8)は自分のかたまりをさらに成長させる.ただし,このときはすでにできている前方のかたまりを

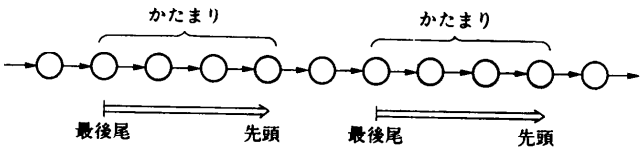


図-4 かたまりの成長

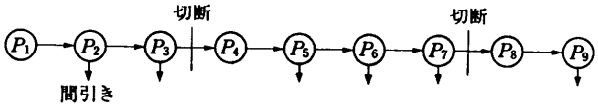


図-5 プロセッサ連鎖の切断

(複数のノードを)1ステップで取り込める.再び追突が起きるかもしれないが,そのときは同様の処理を行う.

(4) 上記(2),(3)の作業を $C\log n$ 時間(C はある定数)行った後,その時点でのかたまり(確定しているものは除く)を1個のノードとみなすとその数が $\frac{1}{k}n$ (k も1より大きな定数)になることが証明できる.そこで次のフェーズでは,これらの(広義の)ノードに対して再び均等にプロセッサを割り当て(1プロセッサ当り $\frac{1}{k}\log n$ ノード),(2),(3)を今度は $\frac{1}{k}C\log n$ 時間行う.こうして,等比級数的に時間および1プロセッサ当りのノード数を減らしていき,$\log\log n$ フェーズ後に1プロセッサ当り1ノードとする.(かたまりの数はおおよそ $n/\log n$ になっている.)問題はプロセッサの再割り当てに要する時間であるが,これは前述の RNK が与えられているときのプレフィクスサムの計算そのものになる.前は $O(\log n)$ 時間のアルゴリズムを与えたが,これも上手いやり方[1]があって,$O(\log n/\log\log n)$ 時間に減らせる.こうして全体で $O(\log n)$ 時間に収まってしまう.

かたまりが完成した後の処理はすでに述べた.リストランキングの最適並列化の影響はきわめて大きく,この結果がただちに相当な数の問題に対する最適並列アルゴリズムを導いた.

5. あ と が き

並列アルゴリズム研究のゴールはある意味で際限ない.単に一つの問題をとっても,(i)並列化可能であることを示し,(ii)最適化し,さらに(iii)最適性を保ったままでより高速化するという

目標までは述べた．実は，まだあって，(iv)バラエティに富んだ，より弱い（がより現実的な）モデル上でできるだけ PRAM に負けないアルゴリズムを開発するという，気の遠くなるような道のりが待っている．読者の挑戦を期待したい．

参考文献

1) Cole, R. and Vishkin, U.: Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems, In *Proc. 27th IEEE Symp. on Foundations of Computer Science*, pp. 478-491 (1986).
2) Cole, R. and Vishkin, U.: Deterministic Coin Tossing and Accerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, In *Proc. 18th ACM Symposium on Theory of Computing*, pp. 206-219 (1986).
3) Cole, R. and Vishkin, U.: Faster Optimal Parallel Prefix Sums and List Ranking, *Infom. and Comput.*, Vol. 81, pp. 334-352 (1989).
4) Gibbons, A. and Rytter, W.: *Efficient Parallel Algorithms*, Cambridge University Press (1988).
5) 岩間：極並列アルゴリズム，情報処理，Vol. 31, pp. 913-920 (1990).
6) 岩間：並列計算の理論，電子情報通信学会誌，Vol. 75, pp. 56-65 (1992).
7) 宮野：並列アルゴリズムの複雑さ，情報処理，Vol. 32, pp. 171-179 (1991).

(平成4年2月12日受付)

岩間 一雄（正会員）

昭和26年生．昭和48年京都大学工学部電気工学科卒業．昭和55年同大学院博士課程修了．工学博士．昭和54年京都産業大学理学部計算機科学科講師．昭和57年同助教授．昭和58年—59年カリフォルニア大学バークレー客員準教授．平成2年6月より九州大学工学部情報工学科助教授．計算の複雑さの理論，並列アルゴリズム等の研究に従事．