プログラミング 11－12
(1997. 1. 23)

共有メモリマルチプロセッサシステムにおける同期時間最適な無待機時計合わせプロトコル

守屋 宣　井上美智子　増澤利光　藤原秀雄

奈良先端科学技術大学院大学 情報科学研究科
〒630-01 奈良県生駒市高山町 8916-5
e-mail:{sen-m, kounoe, masuzawa, fujiwara}@is.aist-nara.ac.jp

あらまし 共有メモリマルチプロセッサシステム，特に，システム内の全プロセッサが大域パルスを共有するフェーズ内システムにおける無待機時計合わせプロトコルに関して考察する．無待機時計合わせプロトコルとは，ある特定時間である同期時間以上正常に動作し続けているすべてのプロセッサの局所時計が同期することを保証するプロトコルで，プロセッサが任意の時間動作を停止し，動作停止に気づかずに動作を再開する，居眠り故障に耐性がある．フェーズ内システムにおける無待機時計合わせプロトコルとしては，これまで同期時間 $O(n^2)$ のプロトコルが提案されていた（n はプロセッサ数）．本稿では，同期時間 $O(n)$ のプロトコルを提案し，また，この同期時間がオーダー的に最適であることを示す．

キーワード 共有メモリマルチプロセッサシステム，時計合わせプロトコル，無待機性，同期時間

Optimal Wait-free Clock Synchronization Protocol on a Shared-memory Multi-processor System

Sen MORIYA　Michiko INOUE　Toshimitsu MASUZAWA　Hideo FUJIWARA

Graduate School of Information Science
Nara Institute of Science and Technology
8916-5, Takayamacho, Ikoma, Nara 630-01
e-mail:{sen-m, kounoe, masuzawa, fujiwara}@is.aist-nara.ac.jp

Abstract We consider wait-free clock synchronization protocols on a shared-memory multi-processor system, especially, on an in-phase system in which all processors share a common clock pulse. A wait-free clock synchronization protocol guarantees that, for a fixed k, local clocks of processors which have been working correctly for at least k time are synchronized. Such k is called synchronization time. The best previous protocol has synchronization time $O(n^2)$, where n is the number of processors. In this paper, we present a wait-free synchronization protocol with synchronization time $O(n)$. We also show that this synchronization protocol is asymptotically optimal.

key words shared-memory multi-processor system, clock synchronization, wait-freedom, synchronization time

1 まえがき

複数のプロセッサがそれぞれの局所時計を管理している分散システムを考える．このような分散システムにおいて，故障が存在するにも関わらず局所時計を同期させ続ける問題である時計合わせ問題は多くの応用問題にとって重要であり，またそれ自体も興味深い問題である．時計合わせ問題は，これまで，メッセージパッシングシステム，共有メモリマルチプロセッサシステム等，種々の分散システムにおいて研究されている．

メッセージパッシングシステムにおける時計合わせ問題は，各プロセッサがそれぞれ進捗率に偏りのある物理時計を持ち，かつ，メッセージの伝送遅延が有界であるという仮定のもとで，プロセッサのビザンティン故障、認証ビザンティン故障に耐性を持つプロトコルに関する多くの研究がなされた [1, 2, 3, 4, 5]．これらの研究で提案された多くのプロトコルは，正常なプロセッサの局所時計を同期させるもので，一度故障したプロセッサが復旧後に他の正常なプロセッサと局所時計を同期させることは，保証されていない．

また，Gouda ら [6], Arora ら [7] は，分散システムの任意の状況から実行を開始しても，各プロセッサの時計を同期させることができる，自己安定時計合わせプロトコルを提案した．これらのプロトコルは，局所時計の破壊などの一時故障が起きても，その後，十分長い時間故障が起きなければ，時計の値を同期させることができる．

共有メモリマルチプロセッサシステムにおける時計合わせ問題は，共有メモリマルチプロセッサシステムにおけるプロセッサの動作と故障パターンを考慮して，Dolev と Welch によって提案された [8]．彼らは，プロセッサの故障として，任意時間動作を停止し，動作停止に気づかずに動作を再開するという居眠り故障を対象とし，任意個のプロセッサの居眠り故障に耐性のある時計合わせプロトコルを提案した．提案されたプロトコルは，ある固定された値 kに対し，以下の２つの条件を保証している．

- 少なくとも k時間連続して動作したプロセッサは，それ以降正常に動作し続ける限り，局所時計の調整を行わない．
- 少なくとも k時間連続して動作した２つのプロセッサの局所時計の値は一致する．

このプロトコルは，各プロセッサは少なくとも k時間連続して動作すれば，他のプロセッサの動作に関わらず，局所時計を同期させることができるという意味で，無待機なプロトコルと呼ばれる．また，上記の kを無待機時計合わせプロトコルの同期時間と呼ぶ．

本稿では，共有メモリマルチプロセッサシステム，特に，全プロセッサが共通の大域パルスを共有する，フェーズ内システムにおける無待機時計合わせプロトコルを考察する．フェーズ内システムにおける無待機時計合わせプロトコルとして，Dolev と Welch は，同期時間 $O(n^3)$ の自己安定無待機時計合わせプロトコルおよび，同期時間 $O(n^2)$ の無待機時計合わせプロトコルを提案した [8]．また，Papatriantafilou と Tsigas は，同期時間 $O(n^2)$ の自己安定無待機時計合わせプロトコルを提案した [9]．本稿では，同期時間 $O(n)$ の無待機時計合わせプロトコルの提案をする．また，提案するプロトコルの同期時間がオーダー的に最適であることを示す．

以下，２節ではシステムモデルに関する定義を行う．３節では，無待機時計合わせプロトコルの定義を行う．４節では，同期時間 $O(n)$ の無待機時計合わせプロトコルを示し，プロトコルの証明の概略を示す．５節では，無待機時計合わせプロトコルの同期時間の下界が$\Omega(n)$ であることを示す．

2 モデル

n 個のプロセッサがいくつかのレジスタを共有している共有メモリマルチプロセッサシステムを考える．プロセッサはレジスタを介してのみ通信を行うことができる．本稿では，各レジスタは $1WnR$レジスタ (single-writer n-reader register) であるとし，各レジスタ Rに対し，１プロセッサのみが書き込みをでき，全プロセッサが読み出しをできる．レジスタ Rに書き込みをできるプロセッサを Rの**所有者**と呼ぶ．

各プロセッサは状態機械としてモデル化される．状態 s のプロセッサ Pは，以下の操作を行い，次の状態 s'に遷移する．

(1) 状態 s により決定するレジスタ Rを読み込む．

(2) 状態 s とレジスタ Rの値を基に，Pが所有するレジスタを更新し，状態 s'に遷移する．

これらの操作を１ステップと定義する．

本稿では，フェーズ内システムと呼ばれる同期式マルチプロセッサシステムを扱う．フェーズ内システムとは，全プロセッサが共通の大域パルスを共有するシステムであり，全プロセッサは１ステップずつ同期し

て動作する．ただし，本稿では**居眠り**故障を考慮し，各パルスで全プロセッサが動作するとは限らないとする．このようなシステムを以下のようにモデル化する．

システムの大域状況は，全プロセッサの状態と全レジスタの値の組で表すことができる．以降，このシステムの大域状況のことを，単に**状況**と言う．システムの**実行**は，状況と，各パルスで動作したプロセッサの集合の無限または有限交互列 $E = c_0\pi_1c_1\cdots$で表すことができる．ここで，各 c_tは状況を表し，c_0は全プロセッサの初期状態，全レジスタの初期値から成る初期状況を表す．π_tは t 番目のパルス発生時に動作したプロセッサの集合を表す．また，実行 Eが有限であるときは，Eは状況で終る．この実行 Eは，各 t に対し，π_tに属する各プロセッサが c_{t-1}を基に1ステップを行い，状況が c_tになったことを意味する．以下では，パルスをこのプロセッサ集合として扱う．また，t 番目のパルスが発生した (大域) 時刻を時刻 t と呼ぶ．プロセッサ Pについて，$P \notin \pi_t$のとき，Pは時刻 t で**居眠り**したと言う．時刻 tで居眠りしたプロセッサ Pは c_{t-1}を基にしたステップは行わないため，c_{t-1}と c_tにおける Pの状態と Pが所有するレジスタの値は変わらない．

以下では，システムの n 個のプロセッサを $P_1, P_2, \cdots, P_n$と表し，各 P_iに対し i を識別子と言う．また，簡単化のため，本稿では各プロセッサ P_iが所有するレジスタは1個だけとし，そのレジスタを R_iと表す．P_iは R_iをいくつかのフィールドに分割して使用することができる．以下では，レジスタ Rのフィールド f を $R.f$，プロセッサ Pの変数 x を $P.x$ と表す．実行 $E = c_0\pi_1c_1\cdots$に対し，状況π_tでのフィールド $R.f$，変数 $P.x$ の値を，それぞれ $R.f(\pi_t), P.x(\pi_t)$ で表す．ただし，文脈から明らかなときは，単に $R.f(t), P.x(t)$ と表すこともある．

3 無待機時計合わせプロトコル

本節では，無待機時計合わせプロトコルを定義する．各プロセッサ P_iの所有するレジスタ R_iはフィールド $clock$を持つ．$R_i.clock$の値は，プロセッサ P_iの局所時計の値を保持する．実行 $E = c_0\pi_1c_1\cdots$に対して，プロセッサ Pが時刻 t まで連続して動作したパルス数を $work(P,t)$ と表す．すなわち，$work(P,t) = \max\{l | P \in \bigcap_{t-(l-1)\le t'\le t} \pi_{t'}\}$とする．ただし，$P \notin \pi_t$ のときは，$work(P,t) = 0$ と定義する．

任意の t に対し，時刻 t まで連続して動作したパルス数がある特定パルス以上であるような任意のプロセッサ Pの局所時計の値が一致し，以降 Pが動作し続ける限り局所時計の値が1パルス毎に1ずつ増えるようにするプロトコルを無待機時計合わせプロトコルと言う．無待機時計合わせプロトコルは，次のように定義される．

定義 1 ある正整数 kに対し，システムの任意の実行が次の2つの条件を満たすようなプロトコルを，**無待機時計合わせプロトコル**と言う．

Adjustment: 任意の $t \ge 1$, 任意のプロセッサ P_iに対し，$work(P_i,t) > k$ならば，

$$R_i.clock(t) = R_i.clock(t-1) + 1$$

が成り立つ．

Agreement: 任意の $t \ge 1$, 任意のプロセッサ P_i, P_j に対し，$work(P_i,t) \ge k, work(P_j,t) \ge k$ ならば，

$$R_i.clock(t) = R_j.clock(t)$$

が成り立つ

また，このときの正整数 kを**同期時間**と言う． □

4 プロトコル

本節で，同期時間 $O(n)$ の無待機時計合わせプロトコルを提案する．プロトコルを図1に示す．

4.1 プロトコルの概略

本プロトコルでは，各プロセッサ P_iは2つのモード，調整中モード，調整済モードを持つ．調整中モードでは時計の調整を行い，調整済モードでは1ステップ毎に局所時計 $R_i.clock$の値を1ずつ増やす．また，モードに関わらず，各ステップで，P_i自身あるいは他のプロセッサが居眠りしたかどうかのチェックを行う．

変数

各プロセッサ P_iは，P_iの変数および R_iのフィールドとして，以下の値を保持する．

- $clock$： 時計の値を表す．(初期値は 0)
- $clock\_gen$： 時計の世代番号を表す．局所再設定をする度に1増やされる．(初期値は 1)

- *work_count*: 現在の時計の世代番号の調整作業を始めてから, 動作したステップ数を表す. (初期値は 1)
- *count*: 初期状況からのステップ数を表す. (初期値は 1)
- *adjusted*: 時計の調整を終えたかどうかを表す論理型変数で, 時計調整中のときは 0, 時計調整が終了したら 1 である. (初期値は 0)
- *invalid*[1..*n*]: *invalid*[*j*] は P_jの時計の無効な世代番号のうち, P_iが知っている最大値を表す. (初期値は 0)

また, 上記以外に P_iの変数として, 以下の値を保持する.

- *last*[1..*n*]:*last*[*j*] は 4 つのフィールド *my_count*, *count*, *clock_gen*, *work_count* を持ち, それぞれ前回 R_jの読み出しをしたときの P_i.*count*, R_j.*count*, R_j.*clock_gen*, R_j.*work_count* の値を表す.(初期値は (1,1,1,1))
- *stay*:時計調整を終えるまでの足踏み (後述) の回数を表す. (初期値は 0)
- *j*:現在のパルスで読み出したレジスタが R_jであることを表す. (初期値は 1)

動作

調整中モードでは, 以下のように局所時計を調整する. 以下の場合, P_iは調整作業を開始する.

- 初期状態である場合.
- P_i自身の居眠りを検知して, または, 調整時間が長くなり過ぎて, *clock*, *work_count* 等のいくつかの変数を再設定した場合.

後者の場合のいくつかの変数の初期化を局所再設定と呼ぶ.

初期状態, または局所再設定を行った後, プロセッサはまず $R_1, R_2, \cdots R_n, R_1, R_2, \cdots, R_n$の順に, プロセッサの識別子の順で 2 周, 各ステップで 1 レジスタずつの読み出しを行う. レジスタ R_jの読み出しをしたとき, R_j.*work_count* の値は *last*[*j*].*work_count* に格納して記憶する. 2 周目の読み出しを終えたとき, 各レジスタから読み出した *work_count* の値を基に, 各プロセッサを *work_count* の降順にソートする. ソートの結果を Q とする. 3 周目は, 自分より *work_count* が大きいレジスタに対し, 系列 Q の順にレジスタの読み出しを行いながら, 時計の値を調整する. このとき, R_jの状態により, 次のように操作する.

- 以下の場合, R_j.*clock*の値は無効とし, 自分の *clock*の値を 1 増やす.

  (1) 前回の R_jの読み出し以降に P_jが局所再設定を行っている場合
  (P_jの時計の世代番号で判断する.)

  (2) 前回の R_jの読み出し以降の P_jの居眠りを検知した場合
  (居眠りのチェック法は後述する.)

- 上記以外の場合, 次のようにする. ソートの結果 Q において, P_iより先にあるすべてのプロセッサ Pに対し, Pの *clock*を無効と判定する, あるいは Pの *clock*に P_iの *clock*を合わせる操作を行ったら, P_iは調整済とする.

  (1) P_jが調整済の場合, P_iの *clock*の値を R_j.*clock*の値に合わせる. ただし, この調整作業中に少なくとも 1 回他のプロセッサの局所時計に P_iの時計を合わせたことがある場合, 以下のときは R_j.*clock*に P_iの *clock*を合わせることはない.

  - P_i.*clock* > R_j.*clock*である場合は, R_j.*clock*の値は無効とし, 自分の *clock*を 1 増やす.
  - P_i.*clock* < R_j.*clock*である場合は, P_i自身の居眠りを検知したことになるので, 局所再設定を行う.

  (2) P_jが調整中の場合は, 次ステップでも R_jを読み出すことにする. この操作を足踏みと呼ぶ. この調整作業中における足踏み回数 (変数 *stay*) が n になった場合は, P_iは局所再設定を行う.

調整済モードでは, まず調整中モードの 3 周目で読み出しをしなかったレジスタを系列 Q の順に読み出しを行い, そのあとは系列 Q の順に読み出しを続け, 各ステップでは, *clock*の値を 1 増やす.

居眠りのチェックは以下のようにして行う．プロセッサ P_iは，あるレジスタ R_jの読み出しをしてから再び R_jの読み出しをするまで，$P_i.count, R_j.count, R_j.clock\_gen$ の値をそれぞれ $last[j]$ のフィールド $my\_count, count, clock\_gen$ に格納して記憶する．レジスタ R_jの読み出しをしたとき，前回の R_jの読み出し時からの $R_j.count, P_i.count$ の増分の差を $dif\_\Delta count$ とする．このとき，R_jの前回の読み出しから今回の読み出しの間で，$dif\_\Delta count > 0$ ならば P_jが，$dif\_\Delta count < 0$ ならば P_iが，居眠りしたことがわかる．$dif\_\Delta count < 0$ のときは，P_iは局所再設定をし，調整作業を初めからやり直す．$dif\_\Delta count > 0$ のときは，$R_i.invalid[j]$ に $R_j.clock\_gen$ の値を書き込み，P_jが居眠りした世代を知らせる．また，逆に $R_j.invalid[i] = clock\_gen$ の場合は，P_iが世代番号 $clock\_gen$ での居眠りを検知したことを意味し，P_iは局所再設定を行う．

P_iは，各ステップの最後に，変数 $P_i.clock, P_i.clock\_gen, P_i.work\_count, P_i.count, P_i.adjusted, P_i.invalid[j]$ の値を，それぞれに対応する R_iのフィールドに書き込む．

4.2 プロトコルの正当性

図1のプロトコルが，同期時間 $O(n)$ の無待機時計合わせプロトコルであることを示す．ただし，紙面の都合上，いくつかの補題に対しては証明のアイデアのみを示す．

任意の実行 $E = c_0\pi_1c_1\cdots$に対し，条件 Adjustment と Agreement が成り立つことを示す．2つの系列 s_1, s_2に対し，s_2が s_1からいくつかの要素を削除して得られる系列であるとき、**s_2は s_1の順序を保存する**と言う．

プロセッサ P_iが時刻 t_sで手続き $sort$ を実行するとする．P_iが，ソートして得た系列 $P_i.\mathcal{Q}(t_s) = (P_{q_1}, P_{q_2}, \cdots, P_{q_n})$ に対し，その接頭部系列 $(P_{q_1}, P_{q_2}, \cdots, P_l)$ を $P_i.\mathcal{Q}(t_s)^l$と記す．また，時刻 t_s以降，P_iが居眠りおよび局所再設定を行なわずに調整を終えた（変数 $adjusted$ の値を 1 にした）場合に，調整を終えるまでに局所時計の値を合わせたプロセッサの集合を $A(P_i, t_s)$ と表す．$A(P_i, t_s)$ に属するプロセッサ P_jは，P_iが局所時計の値を合わせたときに $P_j.adjust = 1$ であるので，それ以前に最後に調整を終えた時刻 t_j^1，最後に手続き $sort$ を行なった時刻 t_j^2が存在する．$A(P_i, t_s)$ に属するプロセッサ P_jのうち，t_sより後で P_iが P_jの局所時計の値に合わせる以前に調整を終えるプロセッサ，すなわち，$t_s < t_j^1$であるプロセッサの集合を $A'(P_i, t_s)$ とする．このとき，$work(P_i, t_s) > 5n$ であれば，任意の $P_j \in A(P_i, t_s)$ に対し，$P_i.\mathcal{Q}(t_s)^j$ は $P_j.\mathcal{Q}(t_j^2)^j$の順序を保存すること，また，任意の $P_j, P_{j'} \in A(P_i, t_s)$ に対し，$P_i.\mathcal{Q}(t_s)$ において P_jが $P_{j'}$に先行すれば，$P_{j'}.\mathcal{Q}(t_{j'}^2)^j$は $P_j.\mathcal{Q}(t_j^2)^j$の順序を保存することが保証される．このことから，P_iが時刻 t_s以降，最初に時計の調整を終える，または，局所再設定を行なうまでに行なう足踏みは高々 $n-1$ であることが導ける．

$work(P_i, t_s + 2n) > 7n$ の場合を考える．このとき，P_iは区間 $[t_s, t_s + 2n]$ では，自身の居眠りを検知して局所再設定を行なうこともない．このことから，P_iは区間 $[t_s + 1, t_s + 2n]$ 中に調整を終える．すなわち，以下の補題が成り立つ．

補題 1 プロセッサ P_iが時刻 t_sで手続き $sort$ を実行するとする．$work(P_i, t_s + 2n) > 7n$ ならば，

$$P_i.adjusted(t_s + 2n) = 1$$

である． □

補題 2 プロセッサ P_iが時刻 t_rで局所再設定するとする．$work(P_i, t_r + 10n) > 10n$ ならば，

$$P_i.adjusted(t_r + 10n) = 1$$

である．

(証明) $t_r + 5n$ までに調整が終らない場合，すなわち，$P_i.adjusted(t_r + 5n) = 0$ のときを考える．このとき，区間 $[t+4n, t+5n)$ 中のある t_r'で $stay = n$ となり，P_iは局所再設定を行う．P_iは $T_r' + 3n$ で再び手続き $sort$ を行うが，このとき $work(P_i, (t_r' + 3n) + 2n) > 7n$ となるので，$T_r' + 3n + 2n < T_r + 10n$ までに調整を終える． □

P_iが，時刻 t で R_jを読み出したときに，自分の居眠りの検知による局所再設定を行うのは，次の場合である．

- $dif\_\Delta count < 0$ のとき
- $R_j.invalid[i](t-1) \geq P_i.clock\_gen(t-1)$ のとき
- $0 < P_i.clock(t-1) < R_j.clock(t-1)$ かつ $P_i.invalid[j](t) < R_j.clock\_gen(t-1)$ のとき

それぞれの場合に, $work(P_i,t) < 7n$ が成り立つ. よって, 以下の補題が成り立つ.

補題 3 P_iが時刻 t で局所再設定するならば,

$$work(P_i.t) < 7n$$

が成り立つ. □

補題 4 任意の $t \geq 1$, 任意のプロセッサ P_iに対し, $work(P_i,t) > 17n$ ならば,

$$P_i.clock(t) = P_i.clock(t-1) + 1$$

が成り立つ.

(**証明**) P_iが t 以前で最後に居眠りした時刻を t_νとする.

まず, P_iが t_ν以降に居眠りの検知による局所再設定をしない場合を考える. このとき, $t_\nu+5n$ までに調整を終えないならば, 補題2と同様に, 遅くとも $t_\nu+10n$ までに調整を終える.

次に, P_iが t_ν以降に居眠りの検知による局所再設定をする場合を考える. このとき, 補題2より, 局所再設定を行うのは遅くとも $t_\nu+7n$ であり, それ以後は居眠り検知による局所再設定はしない. よって, 補題3より, P_iは遅くとも $t+17n$ までに調整を終える.

P_iは, 調整を終えたあとは局所再設定しないので, プロトコルより, 各ステップで $clock$の値を1ずつ増やす. よって, $work(P_i,t) > 17n$ のときは

$$P_i.clock(t) = P_i.clock(t-1) + 1$$

が成り立つ. □

次に条件 Agreement が成り立つことを示す.

補題 5 任意の $t \geq 1$, 任意のプロセッサ P_i, P_jに対し, $work(P_i,t) \geq 17n, work(P_j,t) \geq 17n$ ならば,

$$P_i.clock(t) = P_j.clock(t)$$

が成り立つ.

(**証明**) 補題2, 3より, $P_j.adjusted(t) = 1$が成り立つ. よって, P_i, P_jともに, t 以前に調整作業を終えたことがある. P_i, P_jが t 以前に終えた最後の調整作業で局所時計を合わせたプロセッサの集合を, それぞれ, $\hat{A}(P_i,t)$, $\hat{A}(P_j,t)$ とする. このとき, $P_j \in \hat{A}(P_i,t)$, または $P_i \in \hat{A}(P_j,t)$ が成り立つ（証明略）.

一般性を失うことなく, P_iを,

(1) $P_j \in \hat{A}(P_i,t)$ かつ $P_i \notin \hat{A}(P_j,t)$ であるプロセッサ,

または,

(2) $P_i \in \hat{A}(P_j,t)$ かつ $P_j \in \hat{A}(P_i,t)$ なら, t 以前に終えた最後の調整作業で, P_jが P_iの局所時計に合わせた時刻$t_{j,i}$と P_iが P_jの局所時計に合わせた時刻 $t_{i,j}$に対し, $t_{j,i} \leq t_{i,j}$が成り立つプロセッサ

であるとする. このとき,

$$P_i.clock(t_{i,j}) = P_j.clock(t_{i,j} - 1) + 1$$

が成り立つ.

時刻 $t-17n+2$ 以降に P_iが最初に R_jを読み出す時刻と $t_{i,j}$の最大値を $t'_{i,j}$とする. P_i, P_jは $t_{i,j}$以降, 局所再設定をすることはないので, $t_{i,j} < t'_{i,j} - 1$ のとき, 区間 $[t_{i,j}, t'_{i,j} - 1]$ で P_iが居眠りする回数を nap_i, P_jが居眠りする回数を nap_jとすると, $nap_i = nap_j$ が成り立つ. P_i. P_jは区間 $[t_{i,j}, t'_{i,j} - 1]$ で、それぞれ, 各ステップで時計の値を1ずつ増やすだけであるので,

$$P_i.clock(t'_{i,j}) = P_j.clock(t'_{i,j} - 1) + 1$$

が成り立つ. また, P_i, P_jともに区間 $[t'_{i,j} - 1, t]$ では居眠りをしないので,

$$P_i.clock(t) = P_j.clock(t)$$

が成り立つ. □

補題4と補題5より, 次の定理が成り立つ.

定理 6 図1のプロトコルは, 同期時間 $O(n)$ の無待機時計合わせプロトコルである. □

5 同期時間の下界

前節で, 同期時間が $O(n)$ の無待機時計合わせプロトコルを提案した. 本節では, 無待機時計合わせプロトコルの同期時間の下界が$\Omega(n)$ であることを示す.

定理 7 同期時間が $n-2$ 以下の無待機時計合わせプロトコルは存在しない。

(**証明**) 同期時間が $k \leq n-2$ の時計合わせプロトコル A が存在すると仮定し, 矛盾を導く. 以下の2つの条件を満たすプロトコル A の2つの実行 $E = c_0\pi_1c_1\cdots, E' = c'_0\pi'_1c'_1\cdots$を考える.

- Eのc_tまでの接頭部は，E'のc'_tまでの接頭部と一致する．
- 任意のプロセッサPに対し，$work(P,t) \geq k$が成り立つ．

プロトコルAは同期時間kのプロトコルであるので，全プロセッサP_iに対し，$R_i.clock(t)$が一致する．この値を$clock(t)$とする．

実行Eにおいて，$\pi_{t+1} = \pi_{t+2} = \cdots = \pi_{t+n-1} = \{P_1\}$とする．このとき，条件 Adjustment より

$$R_1.clock(c_{t+n-2}) = clock(c_t) + n - 2 \qquad (1)$$

が成り立つ．実行Eでは，P_1は時刻$t+1$から$t+n-2$までに$n-2$ステップの動作を行うので，その間に読み出していないレジスタが少なくとも1つ存在する．そのレジスタをR_iとする．

実行E'において，$\pi'_{t+1} = \{P_i\}, \pi'_{t+2} = \pi'_{t+3} = \cdots = \pi'_{t+n-1} = \{P_1, P_i\}$とする．このとき，条件 Adjustment より

$$R_i.clock(c'_{t+n-1}) = clock(c'_t) + n - 1 \qquad (2)$$

が成り立つ．また，P_1, P_i以外のプロセッサは，$\pi_{t+1}, \cdots, \pi_{t+n-1}, \pi'_{t+1}, \cdots, \pi'_{t+n-1}$のどのパルスにも属していない．よって，$R_1, R_i$以外のレジスタの値は実行$E$では$c_t$から$c_{t+n-1}$まで，実行$E'$では$c'_t$から$c'_{t+n-1}$まで更新されない．故に，$P_1$は実行$E'$での時刻$t+2$から$t+n-1$では，$E$の時刻$t+1$から$t+n-2$までと全く同じステップを行う．従って，式(1)より実行E'において，

$$R_1.clock(c'_{t+n-1}) = clock(c'_t) + n - 2 \qquad (3)$$

が成り立つ．ところが，実行E'では$work(P_1, t+n-1) = n-2 \geq k, work(P_i, t+n-1) = t+n-1 \geq k$なので，条件 Agreement より

$$R_1.clock(c'_{t+n-1}) = R_i.clock(c'_{t+n-1})$$

となり，式(2), (3)に矛盾する． □

以上より，無待機時計合わせプロトコルの同期時間の下界は$\Omega(n)$である．従って，前節で提案したプロトコルの同期時間はオーダー的に最適である．

6 むすび

本稿では，共有メモリマルチプロセッサシステムにおける無待機時計合わせプロトコルについて考察した．同期時間が$O(n)$の時計合わせプロトコルを提案した．さらに，提案したプロトコルの同期時間がオーダー的に最適であることを示した．今後の課題としては，同期時間が$O(n)$の自己安定性のある無待機時計合わせプロトコルの考察などが挙げられる．

謝辞

日頃より，分散アルゴリズムについて熱心にご討論いただく本学情報科学センターの片山喜章助手，本学藤原研究室の上田英一郎氏に感謝致します．また，井上智生助手を始めとする本学藤原研究室の諸氏に感謝致します．なお，本研究の一部は，文部省科学研究費補助金(奨励(A)08780279)，財団法人電気通信普及財団の助成による．

参考文献

[1] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults ," Journal of the ACM, Vol. 32, No. 1, ,1985, pp1-36.

[2] J.W. Welch and N. Lynch, "A New Fault-Torelant Algorithm for Clock Synchronization ," Information and Computation, Vol. 77, No. 1, 1988, pp1-36.

[3] D. Dolev, J.Y. Halpern and H. R. Strong, "On the Possibility and Impossibility of Achieving Clock Synchronization," Journal of Computer System Science Vol. 32, No. 2, 1986, pp230-250.

[4] T. K. Srikanth and S. Toudg, "Optimal Clock Synchronization," Journal of the ACM, Vol. 34, No. 3, 1987, pp626-645.

[5] J. Halpern, B. Simons, R. Strong and D. Dolev, "Fault-Tolerant Clock Synchronization ," Proceeding of the 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp89-102.

[6] M. G. Gouda and T. Herman, "Stabilizing Unison ," Information Processing Letters, Vol. 35, 1990, pp171-175.

[7] A. Arora, S. Dolev and M. Gouda, "Maintaining Digital Clocks in Step ," Parallel Processing Letters, Vol. 1 No. 1, 1991, pp11-18.

[8] S. Dolev and J.L. Welch, "Wait-Free Clock Synchronization ," Proceedings of the 12th ACM Symposium on Principles of Distributed Computing, 1993, pp97-108.

[9] M. Papatriantafilou and P. Tsigas, "On Self-Stabilizing Wait-Free Clock Synchronization ," Proceedings of the 4th Scandinavian Workshop on Algorithm Theory(LNCS 824), 1994, pp267-277.

```
1   begin
2     repeat forever do on receipt of a pulse
3       read(R_j);
4       check_a_nap();
5       case work_count of
6         work_count = 0 : wait();
7         work_count = 2n : sort();
8         work_count > 2n : adjust();
9       end
10      count := count + 1;
11      if wait ≠ 0
12        then work_count := work_count + 1;
13      next_read();
14      write(R_i)
15  end.

16  procedure check_a_nap();
17    begin
18      Δcount := count − last[j].my_count;
19      Δcount_j := P_j.count − last[j].count;
20      dif_Δcount := Δcount − Δcount_j;
21      if dif_Δcount > 0 then
22        if R_j.clock_gen = last[j].clock_gen
23          then invalid[j] := R_j.clock_gen;
24          else invalid[j] := R_j.clock_gen − 1;
25      if dif_Δcount < 0
26        or R_j.invalid[i] ≥ clock_gen
27        then local_reset();
28    end;

29  procedure local_reset();
30    begin
31      clock := 0; work_count := 0;
32      adjusted := 0; stay := 0;
33      if wait = 0
34        then clock_gen := clock_gen + 1;
35             wait := n;
36    end;

37  procedure wait();
38    begin
39      wait := wait − 1;
40      position := position + 1(mod n);
41    end;

42  procedure sort();
43    begin
44      for l := 1 to n do
45        if last[i].clock_gen > invalid[i]
46          then now_work_count[l] :=
47                 last[l].work_count + count
48                 −last[l].my_count;
49          else now_work_count[l] := 0;
50      sort P_1, ..., P_n (各 P_l の
51        (now_work_count[l], l) の辞書式順)
52        into Q = (P_q1, P_q2, ···, P_qn);
53      for l := 1 to n do
54        m := 1;
55        while P_qm ≠ P_l do m := m + 1;
56        ranking[l] := m;
57      position := 1;
58      if ranking[i] = 1 then adjusted := 1;
59    end;

60  procedure adjust();
61    begin
62      if adjusted = 1
63        then clock := clock + 1;
64             position := position + 1(mod n);
65        else
66          adjust_0();
67          if stay = n
68            then local_reset();
69          if position = ranking[i]
70            then adjusted := 1;
71    end;

72  procedure adjust_0();
73    begin
74      if dif_Δcount > 0
75        or R_j.clock_gen > last[j].clock_gen
76        or clock > R_j.clock
77        then
78          if clock ≠ 0 then clock := clock + 1;
79          position := position + 1(mod n);
80        else if clock ≠ 0 or clock < R_j.clock
81          then local_reset();
82        else
83          if R_j.adjusted = 1
84            then
85              clock := R_j.clock + 1;
86              position := position + 1(mod n);
87            else
88              if clock ≠ 0
89                then clock := clock + 1;
90              stay := stay + 1;
91    end;

92  procedure next_read();
93    begin
94      case work_count of
95        work_count = 0 ∨ work_count > 2n :
96                m := 1;
97                while position ≠ ranking[m]
98                  do m := m + 1;
99                j := m;
100       1 ≤ work_count ≤ n :
101               j := work_count;
102       n + 1 ≤ work_count ≤ 2n :
103               j := work_count − n;
104     end
105   end;
```

図1：同期時間 $O(n)$ の無待機時計合わせプロトコル