細粒度スレッドモデル向けデータレース検出アルゴリズムの提案

森 達矢[†1] 松崎秀則[†1]

従来のスレッドモデル向けに提案されてきたデータレース検出手法をそのまま細粒度スレッドモデルに適用すると，スレッド数 N に対してメモリ使用量が $O(N^2)$ になる．本稿では既存のデータレース検出アルゴリズム “happens-before” を改良し，トレードオフ関係にある検出率とメモリ使用量をハンドリング可能とするアルゴリズムを提案する．本アルゴリズムでは，検出率の低下を許容すればメモリ使用量を $O(N)$ に下げられる．また，本アルゴリズムを応用することでメモリ使用量を抑えつつ検出率を高める手法を合わせて提案する．実験では，検出率ほぼ 100%で 1 回のデータレース検出におけるメモリ使用量を 10 分の 1 以下に削減できることを確認した．

A Data-race Detection Algorithm for Fine-grain Thread-model

Tatsuya Mori[†1] and Hidenori Matsuzaki[†1]

In fine-grain thread-model, data-race detection algorithms proposed for conventional thread-model require $O(N^2)$ memory usage against the threads of N. This paper proposes an algorithm enabling to control a tradeoff between detection rate and memory usage by improving the “happens-before” algorithm. The proposed algorithm can reduce the memory usage to $O(N)$ if the decrease of the detection rate is permitted. Moreover, this paper proposes a method for raising detection rate and suppressing memory usage by applying this algorithm. Our experiments show the proposed method can reduce the memory usage to 1/10 or less in a single run while keeping almost 100% detection rate.

1. はじめに

マルチプロセッサ向け並列プログラミングでは，プロセスより粒度の小さいスレッド単位でプログラムを並列化するスレッドモデルが使われている[1)]．スレッドモデルでは，同一の共有データへアクセスする複数スレッドどうしで同期が正しく定義されない**データレース（競合）**が問題となる[2)]．この問題に対して，ソフトウェアの実行を解析してデータレースを検出する様々なアルゴリズムが提案されている（6 章参照）．

また昨今，プロセッサ数に比例してプログラムの処理性能が向上するスケーラビリティの実現を目指し，スレッドの粒度がさらに小さい**細粒度スレッドモデル**が研究されている[3)–6)]．細粒度スレッドモデルは従来のスレッドモデルに比べて相対的にスレッド数が増大するため，従来のスレッドモデル向けに提案されたデータレース検出手法の適用が困難になる．既存のデータレース検出アルゴリズム happens-before を細粒度スレッドモデルに適用すると，スレッド数 N に対してメモリ使用量が $O(N^2)$ になる．細粒度スレッドモデルを用いた並列プログラミングではスレッド数が数万に達することもあり，データレース検出のたびに $O(N^2)$ のメモリ量を使用するのは現実的でない．

本稿では，細粒度スレッドモデル向けデータレース検出アルゴリズムを提案する．これは happens-before をベースとし，検出率とメモリ使用量のトレードオフ関係をハンドリングできるよう改良を加えたものである．提案アルゴリズムでは，検出率の低下を許容すればメモリ使用量を $O(N)$ に下げることができる．

また，提案アルゴリズムを応用することでメモリ使用量を抑えつつ検出率を高める手法を合わせて提案する．実験では，検出率ほぼ 100%で 1 回のデータレース検出におけるメモリ使用量を 10 分の 1 以下に削減できることを確認した．

以降，2 章で本稿が扱う細粒度スレッドモデルを示し，3 章で happens-before アルゴリズムを細粒度スレッドモデルに適用する．4 章で提案アルゴリズムの説明を行い，5 章で実験結果と考察を述べたうえで提案アルゴリズムの応用方法とその効果を示す．6 章で関連研究に触れ，7 章でまとめを行う．

†1 株式会社東芝研究開発センター
Research & Development Center, TOSHIBA CORPORATION

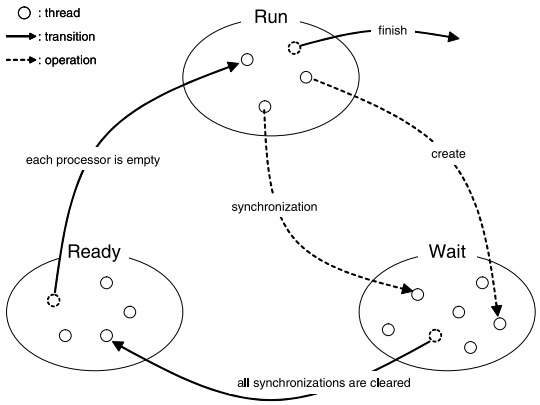


図 1 スレッド管理の概観
Fig. 1 Threads management overview.

2. 細粒度スレッドモデル

細粒度スレッドモデルは，これまで様々な研究がなされている[3)–6)]．本稿が前提とする細粒度スレッドモデルにおいても，スレッドの粒度が小さいことに依拠した以下の性質を持つ．

- **non-preemptive**
  スレッドは実行完了までプロセッサを占有する．
- **non-blocking**
  スレッドには同期待ちが起こらない．
- **non-mutex（mutual exclusion）**
  スレッドどうしで排他制御を行わない．

これらの性質を備えた細粒度スレッドモデルにおけるスレッド管理の概念を**図 1** に図示する．すべてのスレッドは *Wait* 状態から始まり，同期がすべて満たされると *Ready* 状態に移り，1 つ以上のプロセッサが空くとディスパッチされて *Run* 状態になる．1 度 *Run* 状態に移ったスレッドにはプリエンプションが発生せず，終了するまでプロセッサを占有する．新規スレッドの生成と *Wait* 状態スレッドに対する同期を行えるのは *Run* 状態スレッドのみである．また，最初に起動するスレッドだけはユーザや OS によってキックされ，同期待ちがなくかつすべてのプロセッサが空いているのでただちに $Wait \Rightarrow Ready \Rightarrow Run$ と遷移する．

なお，以降で「スレッド」とは「細粒度スレッドモデルにおけるスレッド」を指すものとする．

3. happens-before アルゴリズム

happens-before アルゴリズムを細粒度スレッドモデルに適用する．happens-before アルゴリズムの中身は大きく 2 つに分けられる．1 つがセグメント間の半順序関係を特定するベクタクロックの計算であり，もう 1 つがシャドーメモリを用いたメモリ状態のチェックである．ここでセグメントとは，同期を境界にスレッドを分割した区間を指す．ベクタクロックの計算では，そのクロック値を比較するだけでセグメントどうしの半順序関係が特定できるようスレッド内でのクロック更新およびセグメント間でのクロック伝播を行う．シャドーメモリのチェックでは，ある特定のメモリサイズごとにデータレース判定を行い，同時にデータレース判定に必要なシャドーメモリの情報を更新する．つまり，ベクタクロックを用いてセグメントの半順序関係を特定し，ある同一のメモリ領域にアクセスした複数のセグメントに対してそれらが全順序関係にあるか判定する．全順序関係になければデータレースである．以降でベクタクロックとシャドーメモリについて説明し，happens-before アルゴリズムの詳細を述べた後，メモリ使用量と検出率の定式化を行う．

3.1 ベクタクロック

happens-before アルゴリズムでは，全スレッド分の要素数からなるベクタクロックをスレッドごとに保持する．ベクタクロックは 1 次元配列で表され，配列要素はすべてのスレッドと一意に対応している．スレッドが同期を発行すると自ベクタクロックにおける自要素を increment して更新し，更新後のベクタクロックを同期先のスレッドに転送する．スレッドの実行開始時には，同期によって転送されてきたすべてのベクタクロックの要素ごとに大小比較を行った最大値が選択される．

これらベクタクロック更新・伝播の理解を助けるため，$P = 3$ の例を**図 2** に示す．この図 2 は，プロセッサ p_0〜p_2 で実行されるスレッド t と，それらスレッドどうしの同期関係を図示している．数字の配列はセグメントごとのベクタクロックを表し，スレッド t ごとに自ベクタクロックの自要素 v_t を下線で強調している．

3.2 シャドーメモリ

シャドーメモリの判定および更新は，各プロセッサがメモリアクセス（read/write）するたびに行われる．シャドーメモリの設置単位は任意であるが，その単位 x ごとに以下の情報をシャドーする．

- $sm.x(t_r)$ ： t of last read
- $sm.x(v_r)$ ： v_t of last read
- $sm.x(t_w)$ ： t of last write
- $sm.x(v_w)$ ： v_t of last write

プロセッサから x に対してメモリアクセスが発生すると，まずデータレース判定を行う．シャドーメモリとベクタクロックを比較し，そのメモリアクセスが

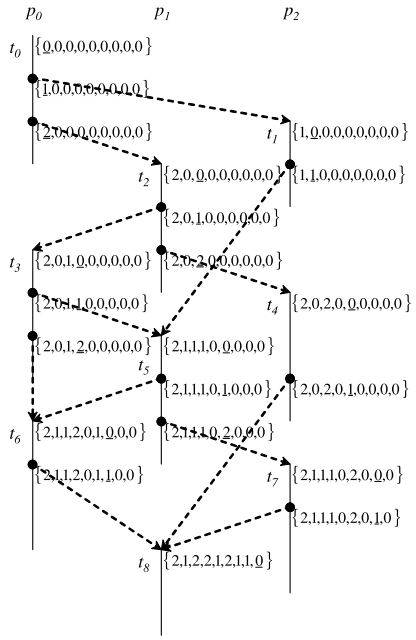


図 2 細粒度スレッドモデルに適用した happens-before アルゴリズムにおけるベクタクロックの更新および伝播

Fig. 2 Vector clock update & propagation in the happens-before algorithm applied to the fine-grain thread-model.

read の場合はアクセスしたスレッドのベクタクロックのうち $sm.x(t_w)$ に対応する要素が $sm.x(v_w)$ 以下，write の場合は $sm.x(t_r)$ に対応する要素が $sm.x(v_r)$ 以下もしくは $sm.x(t_w)$ に対応する要素が $sm.x(v_w)$ 以下であればデータレースである．

データレース判定の後，シャドーメモリの更新を行う．メモリアクセスが read であれば $sm.x(t_r)$ と $sm.x(v_r)$ を，write であれば $sm.x(t_w)$ と $sm.x(v_w)$ を更新する．

3.3 happens-before アルゴリズムの詳細

3.1 節と 3.2 節を合わせた詳細アルゴリズムを記載する．

```
/* initialize */
foreach p ∈ P {
    V_p = O;
}
foreach t ∈ T {
    G_t = O;
}
/* main */
repeat
    case: t starts on p {
        /* vector clock decision */
        V_p = G_t;
    }
    case: t on p posts synchronization to t' {
        /* vector clock update */
        V_p(t)++;
        foreach t'' ∈ T {
            G_t'(t'') = MAX(V_p(t''), G_t'(t''));
        }
    }
    case: t on p reads x {
        /* data-race detection */
        if V_p(sm.x(t_w)) ≤ sm.x(v_w) {
            report "write → read data-race";
        }
        /* shadow memory update */
        sm.x(t_r) = t;
        sm.x(v_r) = V_p(t);
    }
    case: t on p writes x {
        /* data-race detection */
        if V_p(sm.x(t_r)) ≤ sm.x(v_r) {
            report "read → write data-race";
        }
        if V_p(sm.x(t_w)) ≤ sm.x(v_w) {
            report "write → write data-race";
        }
        /* shadow memory update */
        sm.x(t_w) = t;
        sm.x(v_w) = V_p(t);
    }
until each t ∈ T exit;
```

$$
\begin{cases}
T & : number\ of\ threads \\
P & : number\ of\ processors \\
\mathbf{T} & = \{t_0, t_1, t_2, \cdots, t_{T-1}\} \\
\mathbf{P} & = \{p_0, p_1, p_2, \cdots, p_{P-1}\} \\
\mathbf{V}_p & = \{v_t : (\forall t \in \mathbf{T})\} \\
\mathbf{G}_t & = \{g_{t'} : (\forall t' \in \mathbf{T})\}
\end{cases}
$$

$\mathbf{G}_t$ は *Wait* 状態のスレッド t が保持するベクタクロックである．$\mathbf{G}_t$ は，t に対して同期が発行されるたびに更新される．

3.4 定 式 化

メモリ使用量

happens-before アルゴリズムのメモリ使用量はベクタクロックとシャドーメモリが支配的である．

- **ベクタクロック**：$I * T * P + I * T * T$
- **シャドーメモリ**：$I * E * N$

$$\begin{cases} I & : integer(=4\,Byte) \\ E & : elements\ of\ shadow\ memory(=4) \\ N & : number\ of\ variables \\ T & : number\ of\ threads \end{cases}$$

ベクタクロックの初項は $\mathbf{V}$，第 2 項は $\mathbf{G}$ に対応している．happens-before アルゴリズムのメモリ使用量は，スレッド数 T に対して $O(T^2)$ である．

検出率

happens-before アルゴリズムでは，共有データに対するアクセス履歴を read/write それぞれで最新の 1 つしか持たない．よって false-negative が起きるが，1 回のプログラム実行に対してデータレースをデバッグするには十分である．false-positive は起こらない．

4. 提案アルゴリズム

オリジナルの happens-before アルゴリズムと同様，提案アルゴリズムの中身も大きく 2 つに分けられる．以降でベクタクロックとシャドーメモリについて説明し，提案アルゴリズムの詳細を述べた後，メモリ使用量と検出率の定式化を行う．

4.1 ベクタクロック

提案アルゴリズムでは，プロセッサ数分の要素からなるベクタクロックをプロセッサごとに保持し，同一のプロセッサで実行されたスレッドは 1 つのベクタクロックを共有することを基本とする．これにより，顕在化するデータレースがすべて検出されることを保障しつつ，ベクタクロックの要素数を P に縮小させることができる．逆に，同じプロセッサで実行されたスレッドどうしが引き起こすデータレース[⋆1]は検出できない．そこで，各プロセッサで実行されたスレッドどうしで，ある頻度 F ごとにベクタクロックを共有するよう拡張する．すなわち，プロセッサ数が P のとき，$F \times P$ の要素数を持った F 個のベクタクロックを各プロセッサごとに保持する．$F = 1$ であれば基本と同じであり，$F = \infty$ であれば happens-before と同じである．このとき，ベクタクロックは**図 3** に示す $F \times P$ の行列で表され，各スレッドはそれぞれの f_p と p より各 V の f_p 行 p 列目の要素 $v_{f_p,p}$ にマッピングされる．スレッド中で同期を発行すると自ベクタクロックの $v_{f_p,p}$ を increment して更新し，更新後のベクタクロックを同期先のスレッドに転送する．スレッドの実行開始時には f_p と p に対応する過去のベクタクロックを引き継ぎ，同期によって転送されてきたすべてのベクタクロックの要素ごとに大小比較を行って最大値を選択する．

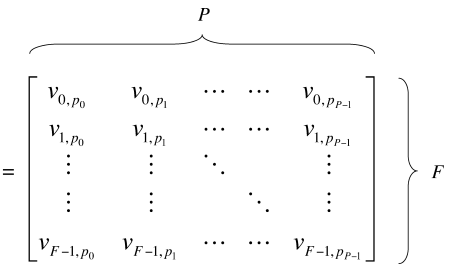


図 3 提案アルゴリズムにおけるベクタクロックの形式
Fig. 3 Form of vector clock in the proposed algorithm.

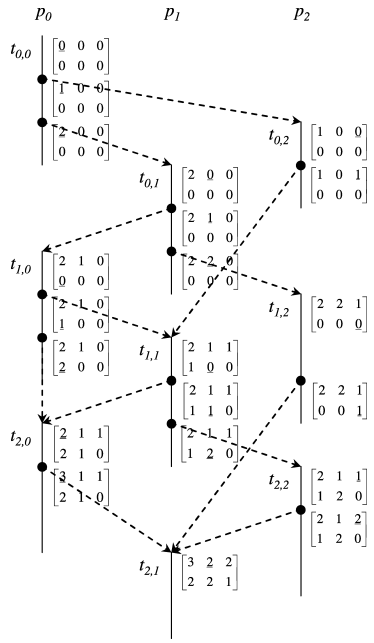


図 4 提案アルゴリズムにおけるベクタクロックの更新および伝播
Fig. 4 Vector clock update & propagation in the proposed algorithm.

これらベクタクロック更新・伝播の理解を助けるため，$F = 2$，$P = 3$ の例を**図 4** に示す．この図 4 は，プロセッサ p_0〜p_2 で実行されるスレッド $t_{i,p}$ と，それらスレッドどうしの同期関係を図示している．i はそのプロセッサで実行されたスレッドの順番を表し，プロセッサごとに $f_p = i \bmod F$ となる．図 4 では，スレッド t ごとに自ベクタクロックの自要素 $v_{f_p,p}$ を下線で強調している．

4.2 シャドーメモリ

happens-before アルゴリズムの場合と同様に，あるメモリ単位 x ごとに以下の情報をシャドーする．

- $sm.x(p_r)$ ： p of last read access
- $sm.x(f_r)$ ： f_p of last read access

⋆1 メモリアクセスが 1 つのプロセッサに閉じているのでデータレースは顕在化しないが，これらのスレッドがつねに同一のプロセッサへ割り当てられるとは限らないため，検出対象に含める．

- $sm.x(v_r)$ ： $v_{f_p,p}$ of last read access
- $sm.x(p_w)$ ： p of last write access
- $sm.x(f_w)$ ： f_p of last write access
- $sm.x(v_w)$ ： $v_{f_p,p}$ of last write access

プロセッサから x へアクセスが発生すると，まずデータレース判定を行う．シャドーメモリとベクタクロックを比較し，そのアクセスが read の場合はアクセスしたプロセッサのベクタクロックのうち $sm.x(f_w)$ 行 $sm.x(p_w)$ 列の要素が $sm.x(v_w)$ 以下，write の場合は $sm.x(f_r)$ 行 $sm.x(p_r)$ 列の要素が $sm.x(v_r)$ 以下もしくは $sm.x(f_w)$ 行 $sm.x(p_w)$ 列の要素が $sm.x(v_w)$ 以下であればデータレースである．

次に，シャドーメモリの更新を行う．read であれば $sm.x(p_r)$，$sm.x(f_r)$，$sm.x(v_r)$ を，write であれば $sm.x(p_w)$，$sm.x(f_w)$，$sm.x(v_w)$ を更新する．

4.3 提案アルゴリズムの詳細

4.1 節と 4.2 節を合わせた詳細アルゴリズムを記載する．

```
/* initialize */
foreach p ∈ P {
    f_p = −1;
    foreach f ∈ F {
        V_{f,p} = O;
    }
}
foreach t ∈ T {
    G_t = O;
}
/* main */
repeat
    case: t starts on p {
        /* vector clock decision */
        f_p = (f_p + 1) mod F
        foreach f′ ∈ F, p′ ∈ P {
            V_{f_p,p}(f′,p′) =
              MAX(V_{f_p,p}(f′,p′), G_t(f′,p′)); ···(1)
        }
    }
    case: p posts synchronization to t {
        /* vector clock update */
        V_{f_p,p}(f_p,p)++;
        foreach f′ ∈ F, p′ ∈ P {
            G_t(f′,p′) =
                  MAX(V_{f_p,p}(f′,p′), G_t(f′,p′));
        }
    }
    case: p reads x {
        /* data-race detection */
        if V_{f_p,p}(sm.x(f_w), sm.x(p_w)) ≤ sm.x(v_w) {
            report "write → read data-race";
        }
        /* shadow memory update */
        sm.x(p_r) = p;
        sm.x(f_r) = f_p;
        sm.x(v_r) = V_{f_p,p}(f_p,p);
    }
    case: p writes x {
        /* data-race detection */
        if V_{f_p,p}(sm.x(f_r), sm.x(p_r)) ≤ sm.x(v_r) {
            report "read → write data-race";
        }
        if V_{f_p,p}(sm.x(f_w), sm.x(p_w)) ≤ sm.x(v_w) {
            report "write → write data-race";
        }
        /* shadow memory update */
        sm.x(p_w) = p;
        sm.x(f_w) = f_p;
        sm.x(v_w) = V_{f_p,p}(f_p,p);
    }
until each t ∈ T exit;
```

$$
\begin{cases}
T & : number\ of\ threads \\
P & : number\ of\ processors \\
\mathbf{T} & = \{t_0, t_1, t_2, \cdots, t_{T-1}\} \\
\mathbf{P} & = \{p_0, p_1, p_2, \cdots, p_{P-1}\} \\
\mathbf{F} & = \{0, 1, 2, \cdots, F-1\} \\
\mathbf{V}_{f_p,p} & = \left[\ v_{f_p,p} : (\forall f_p \in \mathbf{F}, \forall p \in \mathbf{P})\ \right]_{F \times P} \\
\mathbf{G}_t & = \left[\ g_{f_p,p} : (\forall f_p \in \mathbf{F}, \forall p \in \mathbf{P})\ \right]_{F \times P}
\end{cases}
$$

happens-before アルゴリズム（3.3 節）との違いは (1) の部分で，同一プロセッサで実行された過去のスレッドから $\mathbf{V}_{f_p,p}$ を受け継いでいる．

4.4 定 式 化

メモリ使用量

- **ベクタクロック**：$I*(F*P)*(F*P)+I*(F*P)*T$
- **シャドーメモリ**：$I*E*N$

$$
\begin{cases}
I & : integer(=4\,Byte) \\
E & : elements\ of\ shadow\ memory(=6) \\
N & : number\ of\ variables \\
T & : number\ of\ threads
\end{cases}
$$

ベクタクロックの初項は $\mathbf{V}$，第 2 項は $\mathbf{G}$ に対応している．提案アルゴリズムのメモリ使用量は，スレッド数 T に対して $O(T)$ である．

検出率

提案アルゴリズムでは同一プロセッサで実行されるスレッドどうしがベクタクロックを共有することに起因して false-negatives が起こり，happens-before に比べて検出率が低下する．false-positive は起こらない．本稿では happens-before の検出率を 100%とおき，提案アルゴリズムの検出率を相対的に算出する．

なお，両アルゴリズムの時間計算量はメモリ使用量と近似できるため，提案アルゴリズムの時間計算量は happens-before 以下となる．

5. 実験と考察

メモリ使用量と検出率について実験を行う．まず，実験に用いるアプリケーションモデルと OS のスレッドスケジューラを準備し，その上で happens-before アルゴリズムと提案アルゴリズムを用いてデータレース検出を行った結果を比較し，考察する．その後，検出率とメモリ使用量の関係性を考察し，提案アルゴリズムの応用方法について述べる．

5.1 準　　備

提案アルゴリズムを用いた場合の検出率は，アプリケーションのスレッド構造・データ共有関係およびスレッドスケジューラの影響を受ける．よって，本実験ではアプリケーションモデルとスケジューラをそれぞれ 2 つずつ使用する．なお，実験は共有メモリ型マルチプロセッサシミュレータ上でプログラムを実行して収集したトレースに対してデータレース検出を行っているが，実用に際してはシステム構成に制限はない．

アプリケーションモデル

モデルには，細粒度スレッドモデルのターゲットアプリケーションとして想定しているデータ並列性の高いメディア処理を採用した．以下の両アプリは多数のデータレースを発生させるよう故意に設計されている．

- **Appli.A**：H.264 デコードの信号処理部分を用いる．並列化するにあたり，1 フレームのマクロブロック単位に処理を分割してスレッド化した．このように並列化するとスレッドどうしで大量のデータを共有するため，シャドーメモリのメモリ使用量を測定するのに適する．QCIF サイズを例に，スレッドの同期関係を**図 5** に示す．デコード対象フレームの総画素数を N ピクセルとすると，マクロブロックは $16 * 16$ ピクセルの矩形であるので 1 フレーム分のスレッド数は $N/(16 * 16)$ になる．データレース検出の対象となるデータ単位は 1 ピクセルごと，シャドー範囲は 1 フレーム分とした．また，すべてのスレッドにすべてのスレッドとデータの送受信をさせることで，故意にデータレースを発生させている．データの送受信はすべてスレッドの先頭セグメント内で行う．このとき，図 5 の灰色に色付けしたスレッドに注目すると，手書きで囲った枠内のスレッドとの間でデータレースを引き起こす．全体では計 1,980 のデータレースが発生している．

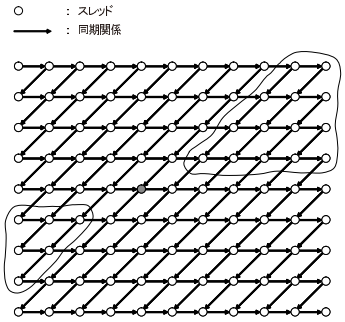


図 5 Appli.A のスレッド構造とデータレース
Fig. 5 Threads structure & data race of Appli.A.

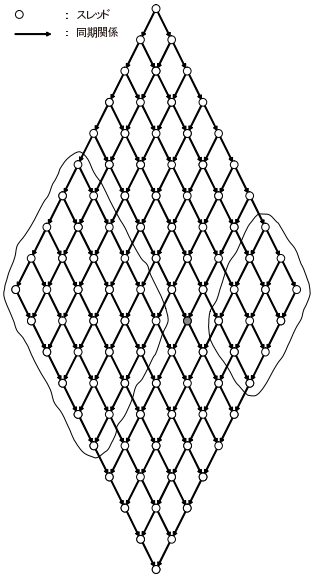


図 6 Appli.B のスレッド構造とデータレース
Fig. 6 Threads structure & data race of Appli.B.

- **Appli.B**：一般的なメディア処理を想定して，**図 6** に示すスレッド構造を持ったベンチマークを設計した．スレッド総数は 100 で，すべてのスレッドは最初と最後のスレッドに直接または間接的に依存している．スレッドどうしのデータ共有関係は Appli.A と同様とした．全体で計 4,050 のデータレースが発生している．

スレッドスケジューラ

提案アルゴリズムの検出率は，同一のプロセッサに

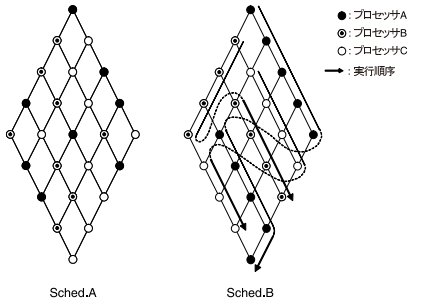


図 7 Sched.A と Sched.B によるスケジューリング結果
Fig. 7 Scheduling result by Sched.A & Sched.B.

ディスパッチされたスレッドどうしの同期関係の有無に左右される．したがって，2 つのスケジューラを用意する．

- **Sched.A**：*Ready* 状態になったスレッドから順にプロセッサへディスパッチする．
- **Sched.B**：同期関係にあるスレッドどうしを同じプロセッサへ連続してディスパッチする．

理解を助けるために，スレッド数を 25 に縮小した Appli.B を $P = 3$ で実行したときの Sched.A と Sched.B それぞれによるスケジューリング結果を**図 7** に示す．Sched.A は各スレッドをプロセッサへランダムにディスパッチし，Sched.B は同期関係にあるスレッドどうしを連続して同一のプロセッサへディスパッチしている（詳細な比較については付録を参照されたい）．

5.2 メモリ使用量

Appli.A において，$P = 6$ のとき画面の総画素数に対するメモリ使用量のグラフを**図 8** に示す．720p サイズで比べるとシャドーメモリは共通で 21.2 MB であるが，ベクタクロックは happens-before が 51 MB であるのに対して提案アルゴリズムでは $F = 1$ で 84KB，$F = 64$ でも 5.4 MB に収まっている．

次に，F をパラメータとし，$P = 6$，$T = 100$ のときに happens-before と提案アルゴリズムがベクタクロックに使用するメモリ量を**図 9** に示す．happens-before は一律 41.4 MB であるのに対し，提案アルゴリズムでは $F = 11$ 付近まで happens-before を下回っている．これは，4.4 節で定式化したベクタクロックが F に対して $O(F^2)$ であることに起因している．

5.3 検 出 率

提案アルゴリズムの検出率は，アプリケーションとスケジューラに加えて P と F の影響を受ける．Sched.A と Sched.B において Appli.A および Appli.B をそれぞれ $P = 1$〜6 で並列処理し，$F = 1$〜100 でデータレース検出を行った検出率のグラフを**図 10** に示す．なお，happens-before の検出率はつねに 100%である

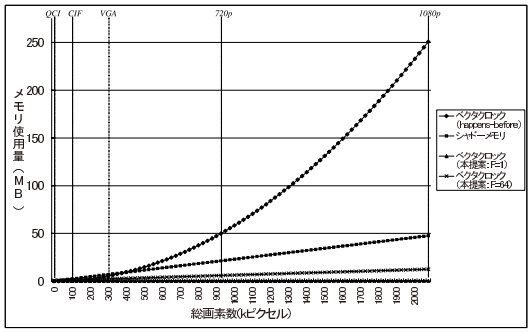


図 8 Appli.A における総画素数に対するメモリ使用量
Fig. 8 Memory usage against number of pixels on Appli.A.

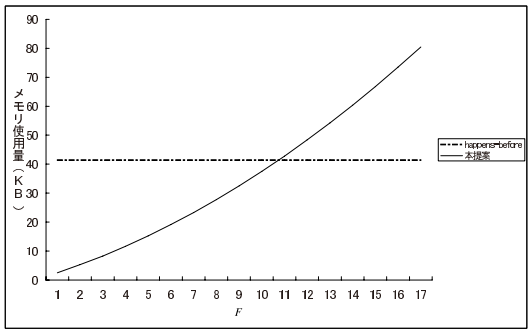
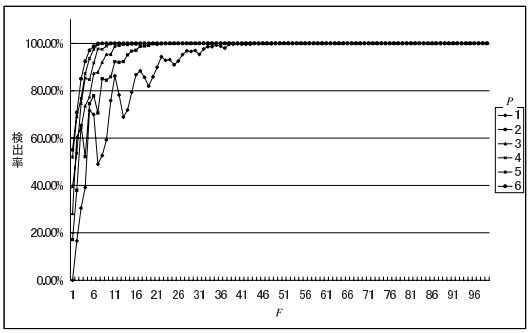


図 9 F に対するメモリ使用量
Fig. 9 Memory usage against F.

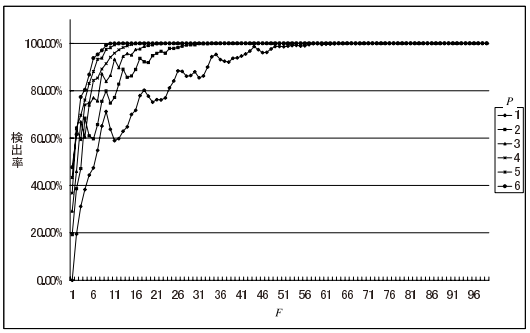
（検出結果の詳細については付録を参照されたい）．

(a) と (b) および (c) と (d) を比較すると，スケジューラが検出率に大きく影響していることが分かる．(c) と (d) において検出率の変動が大きいのは，Appli.A および Appli.B と Sched.B の相互作用による．提案アルゴリズムにおいて検出率が低下するのは，全順序関係にないスレッドどうしがベクタクロックを共有した場合である．(d) について考察すると，図 7 において右斜め下向きの同期関係を持ったスレッドどうしは連続して同じプロセッサへディスパッチされる．すると，あるプロセッサにおいて i 番目に実行されたスレッドがつねに $i \pm 10 * k$ 番目のスレッドとのみベクタクロックを共有すると，全順序関係にないスレッドどうしがベクタクロックを共有することがなくなる．これが，$F = 10 * k$ の場合である．

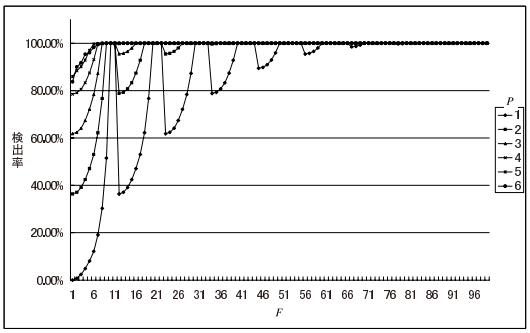
一方，(a) と (c) および (b) と (d) を比較すると，アプリケーションが異なっても検出率にはさほど違いがないことが分かる．いずれにしても，プロセッサの数が増えるほどアプリケーションとスケジューラの影響は小さくなり，$P = 6$ の場合には $F = 6$ で検出率 90%以上，$F = 10$ で検出率 100%をすべてのグラフ



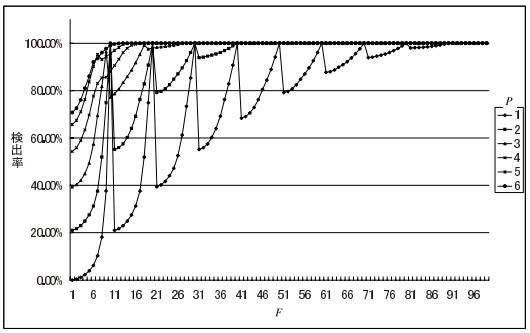
(a) Appli.A on Sched.A



(c) Appli.B on Sched.A



(b) Appli.A on Sched.B



(d) Appli.B on Sched.B

図 10 F に対する検出率
Fig. 10 Detection rate against F.

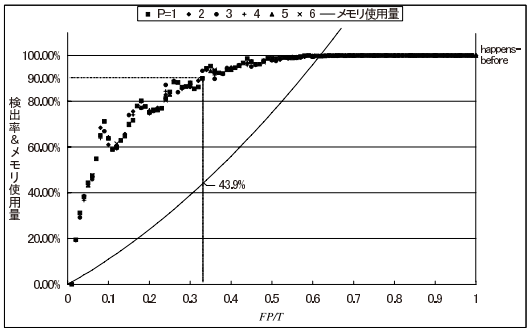


図 11 ベクタクロックのメモリ使用量と検出率の相関
Fig. 11 Correlation between detection rate and memory usage of vector clock.

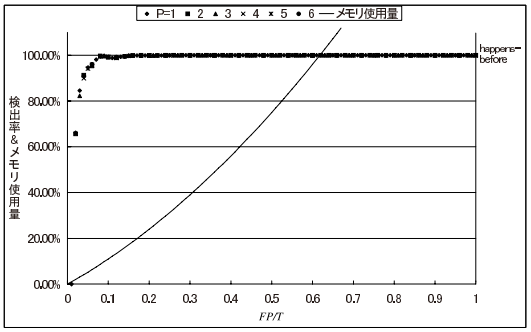


図 12 ベクタクロックのメモリ使用量と検出率の相関（$L = 5$）
Fig. 12 Correlation between detection rate and memory usage of vector clock ($L = 5$).

で達成している．したがって，細粒度スレッドモデルで想定されるプロセッサ数であれば比較的小さな F で高い検出率を維持できると予想される．

5.4 メモリ使用量と検出率の関係性

提案アルゴリズムでは F が検出率に影響を及ぼすが，FP と T の比が一定であれば検出率はほぼ変わらない．FP/T をパラメータとし，提案アルゴリズムにおけるベクタクロックのメモリ使用量と検出率の相関を**図 11** に示す．Appli.B と Sched.A を使用し，検出率とメモリ使用量はどちらも happens-before を100%とする．この図 11 を見ると，まず，FP/T に対する検出率のばらつきが小さいことを確認できる．また，$FP/T = 0.62$ で検出率・メモリ使用量ともに100%に達している．これは図 9 における $F = 11$ の点とも合致する．

次に，検出率が 90%でよい場合を仮定する．図 11 で検出率 90%となるのは $FP/T = 0.33$ のときであり，このときメモリ使用量は 43.9%で済むことになる．つまり，10%の検出率低下を許容することでメモリ使用量を 56%削減できる．

5.5 提案アルゴリズムの応用

図 11 は 1 回の実行に対してデータレース検出を行った場合の検出率であるが，スケジューリング結果が異なるよう実行回数を L 倍すれば検出率は L 乗に比例して向上する．例として，$L = 5$ としたときの検出率と 1 回のデータレース検出に必要なベクタクロックのメモリ使用量を**図 12** に示す．この図 12 によると，検出率 $\fallingdotseq$ 100%を実現しつつ 1 回のデータレース検出におけるベクタクロックのメモリ使用量を 10%以下に削減可能なことが確認できる．

6. 関 連 研 究

データレース検出は静的手法（形式的手法）と動的手法に大別され，プログラムが実行可能な空間を静的にすべて探索する形式的手法は NP 困難に属することが Netzer ら[2] によって示されている．動的手法はプログラム実行を通して得られた情報に対して解析を行うため，潜在的なデータレースをすべて見つけ出すことはできない．動的手法では，Lockset アルゴリズム[7)–10)] と happens-before アルゴリズム[11)–18)] が広く知られている．Lockset は，グローバルなロック変数を用いてスレッドどうしが互いに排他制御を行うロックベースの同期機構に特化したアルゴリズムである．happens-before は，スレッドどうしが相手側のスレッドを明示して同期信号をやりとりし合うシグナルベースの同期機構に特化したアルゴリズムである．また，ロックベースの排他制御はロック変数を介して同期信号を送受信している（相手を明示していないが）とも見なせるため，happens-before アルゴリズムはロックベースの同期機構に対しても適用可能となっている．本稿が前提とする細粒度スレッドモデルではシグナルベースの同期機構のみを使用しているため，Lockset アルゴリズムは適合しない．

Lamport[11] は，happens-before アルゴリズムの基となるセグメントの概念とロジカルクロックを定義している．しかし，セグメントの並列関係を完全には扱えていなかった．Mattern[12] はロジカルクロックを拡張したベクタクロックを提案し，セグメントの並列関係を定義している．Ronsse ら[13] は，happens-before アルゴリズムを応用したデータレース検出機構を Solaris 上に構築している．Pozniansky ら[14] は，C++ 言語で記述されたプログラムに対して happens-before アルゴリズムを用いたデータレース検出を適用している．また，Banerjee ら[15),16)] は happens-before アルゴリズムを実用化し，Intel 社の x86 系プロセッサで実行される OpenMP や PThread プログラミングモデルに従って記述されたプログラムに対してデータレース検出を実現している．しかし，これらの方式ではメモリ使用量が考慮されておらず，細粒度スレッドモデルには適合しない．

7. お わ り に

本稿では既存のデータレース検出アルゴリズム happens-before を細粒度スレッドモデル向けに改良し，検出率とメモリ使用量のトレードオフ関係をハンドリング可能にした．従来はスレッド数 N に対してメモリ使用量が $O(N^2)$ であったが，提案アルゴリズムでは検出率の低下を許容すればメモリ使用量は $O(N)$ となる．さらに，提案アルゴリズムを応用することで，ほぼ 100%の検出率を維持しつつメモリ使用量を 10 分の 1 以下に削減できることを実験的に確認した．

また，提案アルゴリズムの見方を変えると，限られたメモリ量で検出率の最大化が可能であるといえる．たとえば，MPSoC（multi-processor system on a chip）などの組み込み機器においてデータレースの on-the-fly 検出を想定すると，使用できるメモリ量が制限される．このとき，メモリ使用量が許容範囲内の上限になるよう F を調整することによって，制限されたメモリ量下で最大の検出率が引き出される．そのうえで L を増やせば，検出率はさらに向上する．

なお，本稿の実験ではデータレース検出を行う対象プログラムにメディア処理アプリケーションを使用したが，提案アルゴリズムの評価をより一般化するためには SPEC-OMP[19),20)] などを用いて様々なベンチマーキングを行う必要がある．

最後に，現状の課題を以下にまとめる．

L の削減 アプリケーションの特性に合わせてスケジューラを調整することで，L を効果的に削減できる．そのようなスケジューラをいかに実現するか？

データレースの再現性 データレース検出のたびに検出されるデータレースが異なると，デバッグの成否が判断しにくい．したがって，デバッグの前後でスケジューリング結果が同じでなければならない．

検出率 100%の保障 検出されるデータレースは F とスケジューラに依存するが，最終的にはすべてのバグが解消されたことを厳密に保障しなければならない．

これらの課題は，いずれもスケジューラと密接に関係している．今後は，提案アルゴリズム，アプリケーション，スケジューラの関係性について解析を進める．

参 考 文 献

1) Cheriton, D.R., Malcolm, M.A., Melen, L.S. and Sager, G.R.: Thoth, a portable real-time operating system, *Comm. ACM*, Vol.22, No.2, pp.105–115 (1979).
2) Netzer, R.H. and Miller, B.P.: What are race conditions?: Some issues and formalizations, *ACM Lett. Program. Lang. Syst.*, Vol.1, No.1, pp.74–88 (Mar. 1992).
3) Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y.: Cilk: An efficient multithreaded runtime system, *SIGPLAN Not.*, Vol.30, No.8, pp.207–216 (1995).
4) Lowenthal, D.K., Freeh, V.W. and Andrews, G.R.: Using fine-grain threads and run-time decision making in parallel computing, *J. Parallel Distrib. Comput.*, Vol.37, No.1, pp.41–54 (1996).
5) Hendren, L.J., Tang, X., Zhu, Y., Ghobrial, S., Gao, G.R., Xue, X., Cai, H. and Ouellet, P.: Compiling C for the EARTH multithreaded architecture, *Int. J. Parallel Program.*, Vol.25, No.4, pp.305–338 (1997).
6) Culler, D.E., Sah, A., Schauser, K.E., von Eicken, T. and Wawrzynek, J.: Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine, *SIGARCH Comput. Archit. News*, Vol.19, No.2, pp.164–175 (1991).
7) Hoare, C.: An Operating System Structuring Concept, *Comm. ACM*, Vol.17, No.10, pp.549–557 (1974).
8) Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs, *ACM Trans. Comput. Syst.*, Vol.15, No.4, pp.391–411 (1997).
9) Cheng, G., Feng, M., Leiserson, C.E., Randall, K.H. and Stark, A.F.: Detecting data races in Cilk programs that use locks, *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures* (*SPAA '98*), Puerto Vallarta, Mexico, June 28–July 02, 1998, pp.298–309, ACM Press, New York, NY (1998).
10) Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V. and Sridharan, M: Efficient and precise datarace detection for multithreaded object-oriented programs, *SIGPLAN Not.*, Vol.37, No.5, pp.258–269 (2002).
11) Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
12) Mattern, F.: Virtual time and global states of distributed systems, *Proc. International Workshop on Parallel and Distributed Algorithms*, Corsnard, M. (Ed.), North-Holland, pp.120–131 (Oct. 1988).
13) Ronsse, M. and De Bosschere, K.: RecPlay: A fully integrated practical record/replay system, *ACM Trans. Comput. Syst.*, Vol.17, No.2, pp.133–152 (1999).
14) Pozniansky, E. and Schuster, A.: Efficient on-the-fly data race detection in multithreaded C++ programs, *SIGPLAN Not.*, Vol.38, No.10, pp.179–190 (2003).
15) Banerjee, U., Bliss, B., Ma, Z. and Petersen, P.: Unraveling Data Race Detection in the Intel® Thread Checker, Presented at *The First Workshop on Software Tools for Multi-core Systems* (*STMCS*), in conjunction with *IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), March 26, 2006, Manhattan, New York, NY (2006).
16) Banerjee, U., Bliss, B., Ma, Z. and Petersen, P.: A theory of data race detection, *Proc. 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging* (*PADTAD '06*), Portland, Maine, USA, July 17–17, 2006, pp.69–78, ACM Press, New York, NY (2006).
17) Dinning, A. and Schonberg, E.: Detecting access anomalies in programs with critical sections, *Proc. 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (*PADD '91*), Santa Cruz, California, United States, May 20–21, 1991, pp.85–96, ACM Press, New York, NY (1991).
18) O'Callahan, R. and Choi, J.: Hybrid dynamic data race detection, *Proc. 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP '03*), San Diego, California, USA, June 11–13, 2003, pp.167–178, ACM Press, New York, NY (2003).
19) Aslot, V. and Eigenmann, R.: Performance characteristics of the SPEC OMP2001 benchmarks, *SIGARCH Comput. Archit. News*, Vol.29, No.5, pp.31–40 (2001).
20) Aslot, V. and Eigenmann, R.: Quantitative performance analysis of the SPEC OMPM2001 benchmarks, *Sci. Program.*, Vol.11, No.2, pp.105–124 (2003).

付 録

A.1 Sched.A および Sched.B の詳細

図 7 で比較した Sched.A と Sched.B のスケジューリング結果について，各々のスレッドがどのプロセッ

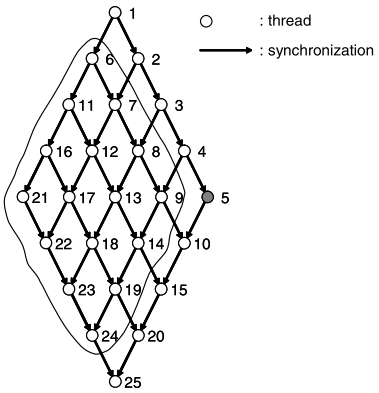


図 13 ベンチマークのスレッド構造およびデータレース
Fig. 13 The thread structure & data-race of the benchmark.

サでどのような順序で実行されたか具体的に示す．

まず，**図 13** でベンチマークの各スレッドに ID を与え，Sched.A によるスケジューリング結果を**図 14** に，Sched.B によるスケジューリング結果を**図 15** にそれぞれ図示する．なお，図 14，図 15 の太線はスレッド，点線は同期，四角内の数字はスレッド ID を表している．また，これらのグラフは時間の経過に沿って上から下へ描かれている．

A.2 データレース検出結果の詳細

ベンチマークのスレッド No.5 はすべてのスレッドと故意にデータ共有を行っている．図 13 に示すとおり，スレッド No.5 は手書きで囲った枠内のスレッドとの間でデータレースを起こし，全部で 16 個のデータレースが発生している．このとき，Sched.A および Sched.B 上で実行したベンチマークに対して，提案アルゴリズムによるデータレース検出を行った結果を図 14 と図 15 にそれぞれ示す．データレースは灰色の直線で表されている．

図 14 では 10 個のデータレースが検出されており，スレッド No.5 と No.6，7，14，19，23，24 間のデータレースは false-negatives により検出されていない．また，図 15 では 12 個のデータレースが検出されており，スレッド No.5 と No.18，19，23，24 間のデータレースは false-negatives により検出されていない．

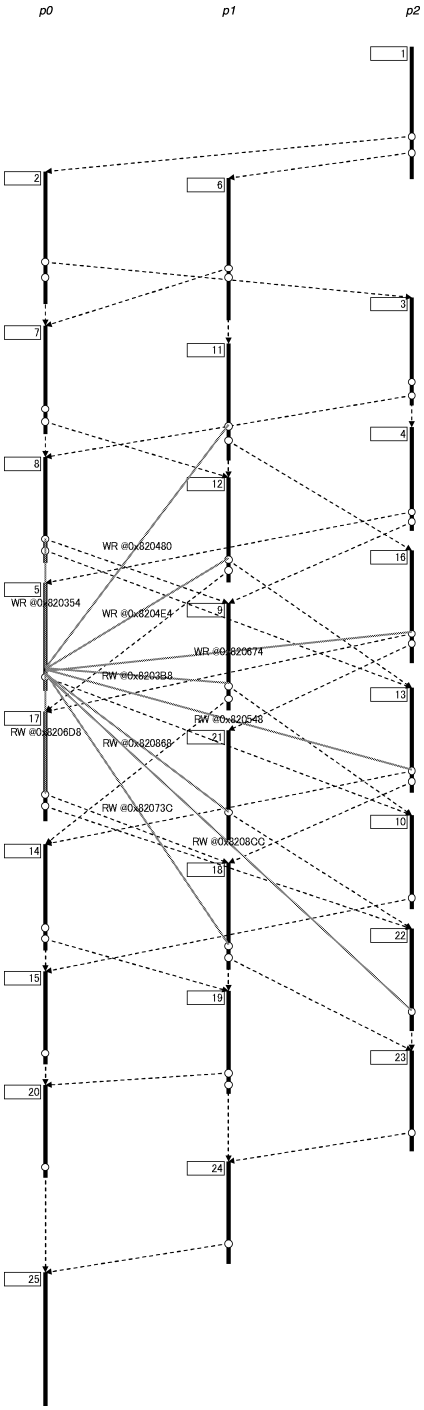


図 14 $P = 3$，Sched.A 上で実行されたベンチマークのスケジューリング結果，および $F = 2$ で提案アルゴリズムを適用したデータレース検出結果
Fig. 14 A scheduling result of the benchmark on Sched.A with $P = 3$ & data-race detection result by applying the proposed algorithm with $F = 2$.

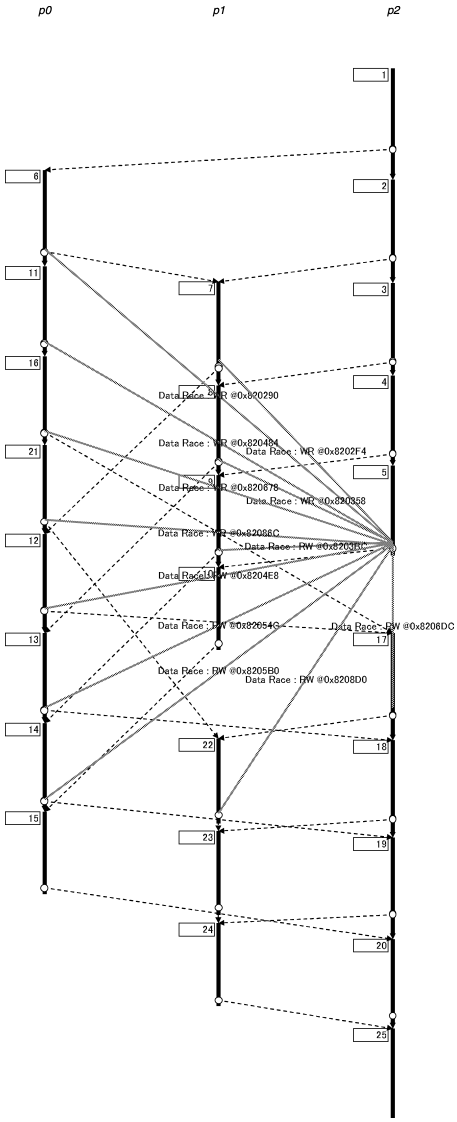


図 15 $P = 3$, Sched.B 上で実行されたベンチマークのスケジューリング結果，および $F = 2$ で提案アルゴリズムを適用したデータレース検出結果

Fig. 15 A scheduling result of the benchmark on Sched.B with $P = 3$ & data-race detection result by applying the proposed algorithm with $F = 2$.

(平成 19 年 7 月 23 日受付)
(平成 19 年 11 月 29 日採録)

森　達矢（正会員）

昭和 54 年生．平成 17 年九州大学大学院システム情報科学府情報工学専攻修士課程修了．同年（株）東芝入社．並列プログラミングの研究に従事．IEEE-CS 会員．

松崎　秀則

昭和 48 年生．平成 10 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．同年（株）東芝入社．並列プログラミングの研究に従事．