計算機アーキテクチャ　105－10
ハイパフォーマンス
コンピューティング　50－10
（１９９４．　３．　１１）

バリア同期のためのタスクスケジューリングアルゴリズムとその性能評価

高木浩光† 有田隆也† 川口喜三男† 曽和将容‡

†名古屋工業大学 電気情報工学科
〒 466 名古屋市昭和区御器所町

‡電気通信大学 情報システム研究科
〒 182 東京都調布市調布が丘 1–5–1

効率的な並列実行のためには，タスク間のデータ依存関係などにより必要となるプロセッサ間の同期操作を，高速に実現することが重要である．同期操作のソフトウェアによる実現では，同期操作自体に浪費される時間が無視できないほどに大きいものとなりうるのに対し，バリア同期の専用ハードウェアによる実現は，高速でしかも実現コストが小さいという特長を持っている．本稿では，ソフトウェアによる同期操作を一切併用することなく，バリア同期のみによって，与えられたプログラムの正しい実行を保証するような，バリア挿入位置を求めるアルゴリズムについて議論し，プロセッサの実行タイミングを推定しながらタスク割当てと同時にバリア挿入位置を決定することで，できるだけ全体の処理時間が短くなるような割当てを決定するアルゴリズムを示す．

A Task Scheduling Algorithm for Barrier Synchronizarion and its Evaluation

Hiromitsu TAKAGI† Takaya ARITA† Kimio KAWAGUCHI† Masahiro SOWA‡

†Department of Electrical and Computer Engeneering
Nagoya Institute of Technology
Gokiso, Showa-Ku, Nagoya 466, Japan

‡Graduate School of Information Systems
University of Electro-Communications
1-5-1 Chofugaoka, Chofu-City Tokyo 182, Japan

Efficient synchronization between processors is essential for efficient parallel execution, by which precedence constraints between tasks are satisfied. The wasting time produced by the operations in hardware implementation such as the barrier synchronization mechanisms can be made sufficiently small because it is organized by a simple AND logic connected by a wire in principle, while the overhead caused by the synchronization operations cannot be ignored in software implementation. This paper discusses the condition for the barrier points to be inserted, which assures the proper parallel execution without any other synchronization operation, and also proposes an algorithm which determine the task allocation and the points for the barriers to be inserted.

1. まえがき

複数台の逐次型プロセッサによる並列実行においては，データの依存関係などによりプロセッサ間の同期操作が必要となる．この同期を，ソフトウェアによる方法，例えば共有メモリ上の変数を用いた方法などで実現した場合，同期操作そのものに要する時間は無視できないほどに大きいものとなりうる．これに対し，同期専用のハードウェアによる実現では，同期操作に要する時間を十分に小さくすることができるが，ハードウェアコストが大きくなりかねない．

こうしたなか，バリア同期機構は，原理的には1本のワイヤーによる論理積回路だけで実現可能であり，高速でしかも単純ある．本来バリア同期は，データ並列性のようなfork/join 型の並列実行を実現するための同期方式として用いられるものであるが，生産者消費者型といった個々のタスク間の同期操作としても用いることができる．

本稿では，バリア同期専用ハードウェアを持つ並列計算機を前提とし，バリア同期以外の同期操作を併用することなく正しい実行が保証されるような，タスク配置およびバリア挿入位置を求めるアルゴリズムについて議論する．

我々は既に文献[1]において，タスク集合 T とタスク間の先行制約 $\prec_C$，タスク t の処理時間 $e(t)$ からなるプログラムと，プロセッサ集合 P，タスクのプロセッサへの割当て及び実行順序 S が与えられたとき，バリア同期以外の同期操作を必要としないようなバリア挿入位置 B を決定するアルゴリズム INSERT-BARRIERS-2 を示した．しかしそこでは，入力の S がリストスケジューリング[2,3,4]などのタスクケジューリングアルゴリズムによって予め求められているものとしていた．リストスケジューリングは，実行タイミングの予測に基づいて，できるだけ全体の処理時間が短くなるようにタスクを配置するアルゴリズムであるが，INSERT-BARRIERS-2 においてあるバリアが挿入されると，そのバリア以降に実行されるタスクの実行タイミングは S が決定される際に予測されたタイミングと異なってくる場合があり，これによって S の決定が十分優れたものでなくなり，バリアを挿入する場合の全体の処理時間を長くしてしまう可能性がある．

そこで本稿では，文献[1]では十分に詳しく示すことのできなかった，$T, \prec_C, P, e$ のみを入力として同時に S と B を決定するアルゴリズム ALLOC-TASKS-WITH-BARRIERS の詳細について述べ，また，このアルゴリズムによるスケジュール結果のより綿密な評価を行なう．

2. 諸定義

2.1 プログラム，タスク，先行制約，実行タイミング

問題として与えられる並列プログラムを

$$G \stackrel{\text{def}}{=} (T, \prec_C, e)$$

で表す．T はプログラムが含むタスクの集合であり，

$$T \stackrel{\text{def}}{=} \{t_1, t_2, \ldots, t_{|T|}\}$$

とする．$\prec$ は T 上の2項関係であり，任意の実行タイミングにおいて，$t \in T$ の実行終了時刻 $\tau_{fin}(t)$, $t' \in T$ の実行開始時刻 $\tau_{st}(t')$ について $\tau_{fin}(t) \leq \tau_{st}(t')$ が成り立つとき $t \prec t'$ と書く．またこのとき，$\prec$ を順序対の集合とみなして $(t,t') \in \prec$ とも書き，(t,t') を先行制約と呼ぶ．$\prec_C$ は与えられた問題が保証することを要求している先行制約の集合であり，

$$\prec_C \subseteq \prec_M$$

なる $\prec_M$ の先行制約が保証されればプログラムは正しく実行される†．

e は $e: T \to \{1,2,\ldots\}$ なる関数であり，$e(t)$ はタスク t の処理時間を表す．実行タイミングすなわち各タスクの実行開始/終了時刻は，ある先行制約集合 $\prec$ と e のみによって次のように与えられる．

- $\tau_{st}^{\prec}(t) = \begin{cases} \max\limits_{t' \in T, t' \prec t} \{\tau_{fin}^{\prec}(t')\} & \{t' \in T \mid t' \prec t\} \neq \phi \\ 0 & \text{otherwise} \end{cases}$
  $\prec$ のもとでタスク t が実行を開始する時刻．
- $\tau_{fin}^{\prec}(t) \stackrel{\text{def}}{=} \tau_{st}^{\prec}(t) + e(t)$
  $\prec$ のもとでタスク t が実行を終了する時刻．
- $\tau_{total}^{\prec} \stackrel{\text{def}}{=} \max\limits_{t \in T} \{\tau_{fin}^{\prec}(t)\}$
  $\prec$ のもとで最も遅く終了するタスクの終了時刻．すなわち，プログラム G の処理時間．

すなわち，どのタスクもそれに先行するすべてのタスクの実行が終了すると同時に実行を開始するものとする．このことは，同期操作などの実行順序の制御に要する時間は無視できると仮定していることを意味する．

2.2 実行系 M_0

M_0 は有限台の逐次実行型プロセッサによる実行系で，特別な同期機構を持たない．与えられた問題の先行制約は，プロセッサの逐次実行とソフトウェアなどによるプロセッサ間の同期操作により保証される．M_0 を

$$M_0 \stackrel{\text{def}}{=} (P, S)$$

で表す．P は実行系が持つプロセッサの集合であり，

$$P \stackrel{\text{def}}{=} \{p_1, p_2, \ldots, p_{|P|}\}$$

とする．$p_1, p_2, \ldots, p_{|P|}$ は能力の等しいプロセッサであり，任意の $p \in P$ が任意の $t \in T$ を実行することができる．

S は，タスクのスケジュールであり，

$$S \stackrel{\text{def}}{=} \langle s_{p_1}, s_{p_2}, \ldots, s_{p_{|P|}} \rangle$$

である．各 s_p は $p \in P$ に割当てられているタスクの列であり，

$$s_p \stackrel{\text{def}}{=} \langle t_{i_1}, t_{i_2}, \ldots, t_{i_{|s_p|}} \rangle$$

である．$|s_p|$ は p に割当てられているタスクの総数を表す．また，s_p の j 番目の要素を単に $s_p[j]$ と書く．ここで，プログラムの正しい実行が保証されるためには，同じタスクが異なる2つ上のプロセッサで実行されることがなく，T のすべての要素はいづれかのプロセッサで実行されるような S でなければならない．

各プロセッサは割当てられたタスク列を逐次的に，すなわち $t_{i_1}, t_{i_2}, \ldots, t_{i_{|s_p|}}$ の順に実行する．したがって，その実行によって p に割当てられたタスク間に

$$\forall_{i,j\,(1 \leq i < j \leq |s_p|)}\ s_p[i] \prec s_p[j]$$

の関係が成立する．この，プロセッサ集合 P の並列実行によって保証されるタスク間の先行制約の集合を $\prec_S$ とす

†ただし実行系がデッドロックをひき起こさないために $\prec_M$ が先行関係[1]でなければならない．

ると，

$$\prec_S \stackrel{\text{def}}{=} \{(s_p[i], s_p[j]) \mid p \in P,\ 1 \le i < j \le |s_p|\}$$

である．

一般には $\prec_C \subseteq \prec_S$ であると限らないので，実行系 M_0 は，$\prec_S$ のみでは保証されない残りの先行制約 $\prec_C \cap \overline{\prec_S}$ をなんらかの同期操作によって保証する．このとき，実行系 M_0 が保証する先行制約の集合は $\prec_S + (\prec_C \cap \overline{\prec_S})$ である†．またこのとき，実行系 M_0 によるプログラム G の処理時間は $\tau_{total}^{\prec_{M_0}}$ で表される．

2.3 バリア同期機構を持つ実行系 M_B

実行系 M_0 にバリア同期機構を加えた実行系を M_B とし，

$$M_B \stackrel{\text{def}}{=} (P, S, B)$$

とする．B は挿入されているバリアの集合であり，

$$B \stackrel{\text{def}}{=} \{b_1, b_2, \ldots, b_{|B|}\}$$

である．$b \in B$ は各プロセッサにおけるそのバリアの挿入点の列であり，

$$b \stackrel{\text{def}}{=} \langle i_{p_1}, i_{p_2}, \ldots, i_{p_{|P|}} \rangle \ (0 \le i_{p_j} \le |s_{p_j}|)$$

である．また，b の j 番目の要素 (すなわちプロセッサ p_j における b の挿入点) を単に $b[p_j]$ と書く．$b[p] = i$ であることは，$1 \le i \le |s_p| - 1$ の場合には，バリア b のプロセッサ p における挿入点が，タスク $s_p[i]$ とタスク $s_p[i+1]$ の間であることを意味し，$b[p] = 0$ の場合には，s_p の先頭のタスクの直前，$b[p] = |s_p|$ の場合には，s_p の最後のタスクの直後であることを意味する．

バリア同期は，すべてのプロセッサの実行がバリア挿入点に到達するまで，どのプロセッサもバリア挿入点を越えて実行しないという同期方式であるので，バリア b による同期操作によって

$$\forall_{p,p' \in P} \forall_{i,j (1 \le i \le b[p],\ b[p'] < j \le |s_{p'}|)}\ s_p[i] \prec s_{p'}[j]$$

の関係が成立する．したがって，バリア集合 B 全体が保証する先行制約の集合を $\prec_B$ とすると，

$$\prec_{B_*} \stackrel{\text{def}}{=} \sum_{b \in B_*} \prec_b$$

$$\prec_b \stackrel{\text{def}}{=} \{(s_p[i], s_{p'}[j]) \mid p, p' \in P,\ 1 \le i \le b[p],\ b[p'] < j \le |s_{p'}|\}$$

である．バリア同期以外の同期操作を併用せずに，実行系 M_B によってプログラム G が正しく実行されるための B の条件については文献 [1] の第 3 節を参照されたい．

3. スケジューリングアルゴリズム

文献 [1] で示したアルゴリズム INSERT-BARRIERS-2 (省略形 IB2) は，$T, \prec_C, P, S, e$ を入力としてバリア挿入点をを求めるものあり，入力の S はリストスケジューリング [2, 3, 4] などのタスクケジューリングアルゴリズムによって予め求められているものとしていた．リストスケジューリングは，実行タイミングの予測に基づいて，できるだけ $\tau_{total}^{\prec_{M_0}}$ が小さくなるようにタスクを配置するアルゴリズムである．しかし，INSERT-BARRIERS-2 において，あるバリアが挿入されると，そのバリア以降に実行されるタスクの実行タイミングは S が決定される際に予測されたタイミングと異

†先行制約集合の和演算の定義については文献 [1] 参照．

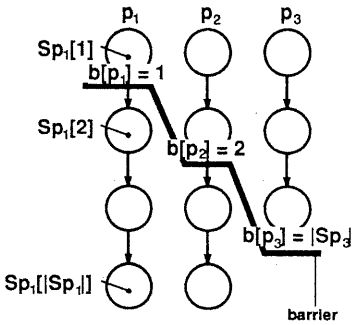


図 1：バリア挿入位置と S の関係

なってくる場合がある．これによって S の決定が十分優れたものでなくなり $\tau_{total}^{\prec_{M_B}}$ を大きくしてしまう可能性がある．

そこで，$T, \prec_C, P, e$ のみを入力として，同時に S と B を決定するアルゴリズム ALLOC-TASKS-WITH-BARRIERS(省略形 ATWB) を以下に示す．

アルゴリズム 1 ALLOC-TASKS-WITH-BARRIERS

ALLOC-TASKS-WITH-BARRIERS $(T, \prec_C, P, e)$ returns S, B is
$S \leftarrow \langle s_{p_1}, s_{p_2}, \ldots, s_{p_{|P|}} \rangle$ where $s_{p_i} = \langle\rangle$;
$B \leftarrow \phi$; $\tau_{cur} \leftarrow 0$;
$T_{rem} \leftarrow T$; $\prec'_{M_B} \leftarrow \phi$ ..... (1)
while $T_{rem} \neq \phi$ **do** ..... (2)
  $P_{idle} \leftarrow \{p \in P \mid \tau_{tail}(p) \le \tau_{cur}\}$; ..... (3)
  $T_{ready} \leftarrow \{t \in T_{rem} \mid \forall_{t' \in T (t' \prec_C t)}\ \tau_{fin}^{\prec'_{M_B}}(t') \le \tau_{cur}\}$; ..... (4)
  $T_{alloc} \leftarrow T_{alloc} \subseteq T_{ready}$ where
    $|T_{alloc}| = \min\{|T_{ready}|, |P_{idle}|\}$ $\wedge$
    $\forall_{t \in T_{alloc},\ t' \in (T_{ready} \cap \overline{T_{alloc}})} h_{CP}(t) \ge h_{CP}(t')$; ..... (5)
  $\prec_{alloc} \leftarrow \{(t', t) \in \prec_C \mid t \in T_{alloc}\}$; ..... (6)
  $\prec'_{alloc} \leftarrow \prec_{alloc}$;
  while $\prec'_{alloc} \neq \phi$ **do** ..... (7)
    $(t', t) \leftarrow (t'; t) \in \prec'_{alloc}$ where
      $\tau_{dist}(t', t) = \min_{(t''', t'') \in \prec'_{alloc}} \{\tau_{dist}(t''', t'')\}$; ..... (8)
    $p \leftarrow p$ where $\exists_i\ s_p[i] = t'$; ..... (9)
    if $t \in T_{alloc}$ **and** $p \in P_{idle}$ **then** ..... (10)
      $s_p \leftarrow s_p + \langle t \rangle$;
      $T_{alloc} \leftarrow T_{alloc} \cap \overline{\{t\}}$; $P_{idle} \leftarrow P_{idle} \cap \overline{\{p\}}$
    end ;
    $\prec'_{alloc} \leftarrow \prec'_{alloc} \cap \overline{\{(t', t)\}}$ ..... (11)
  end ;
  while $T_{alloc} \neq \phi$ **do** ..... (12)
    $t \leftarrow t' \in T_{alloc}$;
    $p \leftarrow p' \in P_{idle}$;
    $s_p \leftarrow s_p + \langle t \rangle$;
    $T_{alloc} \leftarrow T_{alloc} \cap \overline{\{t\}}$; $P_{idle} \leftarrow P_{idle} \cap \overline{\{p\}}$
  end ;
  $\prec'_{M_B} \leftarrow \prec_S + \prec_B + \prec_C$;
  if $\prec_{alloc} \cap \overline{\prec_S + \prec_B} \neq \phi$ **then** ..... (13)
    $\mathcal{A} \leftarrow$ POSSIBLE-ASSIGNMENTS(S, B, τ_{cur}) ; ..... (14)
    $(S, B) \leftarrow (S, B) \in \mathcal{A}$ where
      EVAL$(S, B) = \min_{(S', B') \in \mathcal{A}} \{$EVAL$(S', B')\}$; ..... (15)
    $\prec'_{M_B} \leftarrow \prec_S + \prec_B + \prec_C$;

$\tau_{cur} \leftarrow \max_{b \in B}\{\tau_{sync}(b)\}$ ……(16)
else
$\tau_{cur} \leftarrow \min_{p \in P,\, p \notin P_{idle}}\{\tau_{tail}(p)\}$ ……(17)
end ;
$T_{rem} \leftarrow \{t \in T \mid \forall_{p \in P}\, t \notin s_p\}$ ……(18)
end
end

- $\tau_{tail}(p) \stackrel{\text{def}}{=} \begin{cases} \tau_{fin}^{\prec'_{M_B}}(s_p[|s_p|]) & |s_p| > 0 \\ 0 & \text{otherwise} \end{cases}$
  プロセッサ p に現在最後に割り当てられているタスクの実行終了時刻.
- $\tau_{cp}(t) \stackrel{\text{def}}{=} \begin{cases} e(t) + \max_{t' \in T,\, t \prec_C t'}\{\tau_{cp}(t')\} & \{t' \in T \mid t \prec_C t'\} \neq \phi \\ e(t) & \text{otherwise} \end{cases}$
  タスク t のクリティカルパス長.
- $\tau_{dist}(t',t) \stackrel{\text{def}}{=} \tau_{st}^{\prec'_{M_B}}(t) - \tau_{fin}^{\prec'_{M_B}}(t')$
  先行制約 (t',t) の距離[5].
- $\tau_{sync}(b) \stackrel{\text{def}}{=} \max_{p \in P}\{tau_{fin}^{\prec'_{M_B}}(s_p[b[p]])\}$
  バリア b の同期完了時刻.

アルゴリズムは，割当てるタスクの順序はクリティカルパス法 (CP 法)[2]により決定し (行 (3)–(5),(12),(20))，どのプロセッサにどのタスクを割当てるかを DTSP 法[5]により決定する (行 (6)–(11)) ものとし，タスクを割当てながらバリア同期の挿入が必要となった時点で IB2 とほぼ同様の手法によりひとつのバリア挿入位置を決定する (行 (14)–(18)) ものである．またバリアを挿入したとき，これによってプロセッサに生じる待ち時間の部分に，新たなバリアを必要とすることなくそこで実行可能であるタスクを割当てる (POSSIBLE-ASSIGNMENTS の行 (6)–(13))．

アルゴリズムの詳細を以下に示す．変数 S 及び B はそれぞれ，(2) の **while** ループの各繰り返しの時点における，割当て済みタスク列の列，挿入されているバリアの集合を表し，どちらも空に初期化される．変数 τ_{cur} は，リストスケジューリングにおける割当てイベントの時刻を表し，0 に初期化される．変数 T_{rem} はまだ割当てられていないタスクの集合であり，T に初期化される (行 (1))．$\prec'_{M_B}$ は，繰り返しの各時点において，$\prec_S + \prec_B + \prec_C$ の値をとるもので，既に割当てられているタスクの実行タイミングを推定するために用いられる．この実行タイミングは，その時点で挿入されているバリア集合 B では保証されない残りの先行制約が，実行系 M_0 と同じ同期手段で保証されるとした場合のものである．初期状態では $\prec'_{M_B} = \phi$ である．アルゴリズム全体は，残りのタスクがなくなるまで (2) の **while** ループを繰り返す構造となっている．

ループの各繰り返しは次のような構造となっている．まず時刻 τ_{cur} の段階で，空いているプロセッサの集合 P_{idle} (行 (2))，及び，実行可能なタスクの集合 T_{ready} (行 (3)) を求める．次に，実行可能なタスクのうち，クリティカルパス長の長いものから順に $\min\{$実行可能タスク数,空きプロセッサ数$\}$ 個のタスクを取りだし，割当てタスク T_{alloc} とする (行 (5))．

さらに DTSP 法に基づいた割当てを決定するために，割当てタスクが後続タスクとなっている先行制約の集合 $\prec_{alloc}$ を生成する (行 (6))．そして (7) の **while** ループによりいくつかのタスクの割当てを決定する．その手順は以下の通りである．まず $\prec_{alloc}$ のうち最も距離 $\tau_{dist}(t',t)$ の短い先行制約を取り出し (行 (8))，その先行制約に先行するタスク t' (既に割当て済みのはず) が割当てられているプロセッサ p を求める (行 (9))．このプロセッサがまだ空きの状態であり，その先行制約に後続するタスク t がまだ割当てられていなければ，t を p に割当て ((10) の **if** ブロック)，処理した先行制約を $\prec_{alloc}$ から取り除く (行 (11))．これにより，プロセッサ間をまたがった距離の短い先行制約を少なく抑えることができ，挿入が必要となるバリア数を削減する効果が期待される．また，割当てプロセッサの競合により (7) の **while** ループだけでは割当ての決定されないタスクが生ずる場合があるので，(12) の **while** ループで，それらを適当なプロセッサに割当てる．

ここで，行 (6) で求めた $\prec_{alloc}$ すなわち時刻 τ_{cur} で割当ての決定されたタスクに関する先行制約のうち，プロセッサの逐次実行及びここまでで挿入されたバリアの同期操作によって保証されないものが存在するかをチェックする (行 (13))．もし存在しなければ，次のイベント時刻 (行 (17)) と未割当タスクの集合 (行 (18)) を求めて，一回の繰り返しを終える (この場合は CP-DTSP 法と同一のアルゴリズムとなる)．逆にこれが存在する場合には，バリアの挿入が必要であり，行 (14)–(16) によって適切な位置にバリアが挿入され，次のイベント時刻は最後に挿入されたバリアの同期完了時刻とされる (行 (16))．

バリアの挿入は次の方法で行なう．まず，アルゴリズム POSSIBLE-ASSIGNMENTS によりプロセッサ及びバリアの割当て (S,B) の候補の集合を求め (行 (14))，次に，この候補のうち評価関数 EVAL の値が最も小さくなるものを選んで，現在の割当てをそれに更新する (行 (15))．

以下に，アルゴリズム POSSIBLE-ASSIGNMENTS を示す．

アルゴリズム 2 POSSIBLE-ASSIGNMENTS

POSSIBLE-ASSIGNMENTS (S,B,τ_{cur}) **returns** $\mathcal{A}$ **is**
$\mathcal{T}_{sync} \leftarrow \{\tau \mid \tau_{min} \le \tau \le \tau_{cur},\ \exists_{p \in P,i}\ \tau = \tau_{bound}(p,i)\}$; ..(1)
$\mathcal{A} \leftarrow \phi$;
foreach $\tau \in \mathcal{T}_{sync}$ **do** ……(2)
$b \leftarrow \langle b[p_1],\ldots,b[p_{|P|}]\rangle$ where $b[p_i] = \max_{\{j \mid \tau_{bound}(p_i,j) \le \tau\}}\{j\}$ ….(3)
$B' \leftarrow B \cup \{b\}$; ……(4)
$S' \leftarrow \langle s'_{p_1}, s'_{p_2}, \ldots, s'_{p_{|P|}}\rangle$ where
$s'_{p_i} = \langle s_{p_i}[1], s_{p_i}[2], \ldots, s_{p_i}[b[p]]\rangle$; ……(5)
$T_{rem} \leftarrow \{t \in T \mid \forall_{p \in P}\, t \notin s'_p\}$ ……(6)
$T_{alloc} \leftarrow \{t \in T_{rem} \mid \forall_{t' \in T(t' \prec_C t)}\, \exists_p t' \in s_p\}$; ……(7)
while $T_{alloc} \neq \phi$ **do**
$t \leftarrow t \in T_{alloc}$ where $h_{CP}(t) = \max_{t' \in T_{alloc}}\{h_{CP}(t')\}$; ..(8)
$P_{avail} \leftarrow \{p \in P \mid \tau_{cur} - \tau_{last} \ge e(t),\ \forall_{t' \in T(t' \prec_C t)}$
$(\tau_{fin}^{\prec'_{M_B}}(t') \le \max_{b' \in B}\{\tau_{sync}(b')\} \vee \exists_i\, s_p[i] = t')\}$; ..(9)
if $P_{avail} \neq \phi$ **then** ……(10)
$p \leftarrow p \in P_{avail}$ where $\tau_{last}(p) = \min_{p' \in P_{avail}}\{\tau_{last}(p')\}$; (11)
$s_p \leftarrow s_p + \langle t\rangle$;
$T_{rem} \leftarrow T_{rem} \cap \overline{\{t\}}$;
$b[p] \leftarrow b[p] + 1$; ……(12)
$T_{alloc} \leftarrow \{t \in T_{rem} \mid \forall_{t' \in T(t' \prec_C t)}\, \exists_p t' \in s_p\}$ ….(13)
else
$T_{alloc} \leftarrow T_{alloc} \cap \overline{\{t\}}$ ……(14)
end

```
    end ;
    A ← A ∪ {(S', B')}
  end
end
```

- $\tau_{min} \stackrel{\text{def}}{=} \max_{(t',t)\in\prec_{alloc}} \{\tau_{fin}^{\prec'_{M_B}}(t')\}$
  ATWB の行 (7)–(12) で割り当てたタスクに先行するタスクのうち最も終了時刻の遅いものの終了時刻.
- $\tau_{bound}(p,i) \stackrel{\text{def}}{=} \begin{cases} \tau_{fin}^{\prec'_{M_B}}(s_p[i]) & i>0 \\ 0 & \text{otherwise} \end{cases}$
  p の i 番目のタスクが実行を終了する時刻. ただし $i=0$ の場合は 0 .
- $\tau_{last}(p) \stackrel{\text{def}}{=} \max\{\tau_{tail}(p), \max_{b\in B}\{\tau_{sync}(b)\}\}$
  プロセッサ p に次に割当てられるタスクの実行開始時刻.

アルゴリズム POSSIBLE-ASSIGNMENTS の詳細を以下に示す. まず始めに, 挿入されるバリアの同期完了時刻となる時刻の集合 (行 (1)) を求める. そして (2) の **foreach** ループで, この各要素ごとにひとつの対応するバリア挿入位置が決定される. なぜバリア挿入位置が同期完了時刻に一対一対応するかは, 文献 [1] の第 4.4 節の議論と同様の理由による. ただし, IB2 と異なり, 同期完了時刻の上限は τ_{cur} となっている.

ひとつの同期完了時刻 τ が選ばれると (行 (2)), それに対してひとつのバリア b が決定される. b の各プロセッサにおける挿入点は, p に割当てられているタスクのうち終了時刻が τ を越えず最大となるものの直後 (該当タスクがない場合は先頭のタスクの直前) とする (行 (3)). そして, このバリア b をこれまでに挿入されているバリア集合 B に加えたバリア集合 B' を生成する. 次に, 今加えたバリア以降に割当てられているタスクを S から取り除くことで割当てを取り消したスケジュール S' を生成する (行 (5)). この取り消しを踏まえた未割当タスクの集合 T_{rem} を求め (行 (6)), そのなかから割当て候補タスクの集合 T_{alloc} を生成する (行 (7)). このタスクは, バリア同期によってプロセッサに生じた待ち時間の部分で実行される候補である.

待ち時間の部分への割当ては次のようにして決定される ((8) の **while** ループ). 割当て候補タスクのうちクリティカルパス長が最も大きいタスクを選び† t とし (行 (8)), そのタスクが割当てられても, 新たなバリアを必要とすることなく, またバリア b の同期完了時刻を変化させないようなプロセッサの集合 P_{avail} を求める (行 (9)). そのようなプロセッサの条件は,「プロセッサの待ち時間が t の処理時間より短くなく ($\tau_{cur}-\tau_{last}\geq e(t)$), かつ, t に先行するすべてのタスク t' が, そのロセッサ p に割当てられている ($\exists_i\, s_p[i]=t'$) かまたは, 直前のバリアより前に割当てられている ($\tau_{fin}^{\prec'_{M_B}}(t') \leq \max_{b'\in B}\{\tau_{sync}(b')\}$)」である.

このような P_{avail} が空でなければ (行 (10)), そのなかから空き時間が最も長いプロセッサを選択し p とし (行 (11)), p に t を割当てる (スケジュール S' の上で). また, この割当てはバリア b より前としたいので, これによってずれるバリア挿入点を補正する (行 (12)). そして, この割当てによって新たに割当て候補となるタスクのことも考慮して T_{alloc} を再計算しておく. もし, P_{avail} が空で t をどのプロセッサにも割当てられない場合には, そのタスクを割当て候補から除いて (行 (14)) 次の候補を選択する (行 (8)).

いつかからなず T_{alloc} は空となり, この **while** ループは終了する. このとき求められた S' および B' を, ひとつのスケジュール候補として A に追加する.

次に, スケジュールの評価関数 EVAL を示す. 評価には $\tau_{total}^{\prec'_{M_B}}$ の値を用いている. この値は, 最後に挿入されたバリア以前の実行には M_B が用いられ, それ以降の実行には M_0 が用いられたとした場合の全体の処理時間である. 最後に挿入されたバリア以降の実行系 M_B による実行タイミングを推定するためには, そこに挿入されるバリアを再帰的に求める必要があり, これを用いるとアルゴリズムの計算時間が指数オーダになってしまうため, ここでは M_0 による実行タイミングで近似した.

アルゴリズム 3 EVAL

```
EVAL (S, B) returns τ is
  T_rem ← {t ∈ T | ∀_{p∈P} t ∉ s'_p} ;
  ≺_rem ← {(t', t) | t', t ∈ T_rem} ;  ..... (1)
  S' ← LIST-SCHEDULING-CP(T_rem, ≺_rem, P, e) ;  ..... (2)
  S'' ← ⟨s''_{p_1}, s''_{p_2}, ..., s''_{p_|P|}⟩ where s''_{p_i} = s_{p_i} + s'_{p_i} ;  ..... (3)
  ≺'_{M_B} ← ≺_C + ≺_{S''} + ≺_B  ..... (4)
  τ ← τ_total^{≺'_{M_B}}
end
```

まず, まだ割当てられていないタスクのみからなる部分グラフを生成し (行 (1) まで), これをリストスケジューリング (CP 法) でスケジュールする (行 (2)). この結果 S' を S に連結した S'' を求め (行 (3)), その全体の処理時間を求めている (行 (4) 以降).

4. アルゴリズムの評価

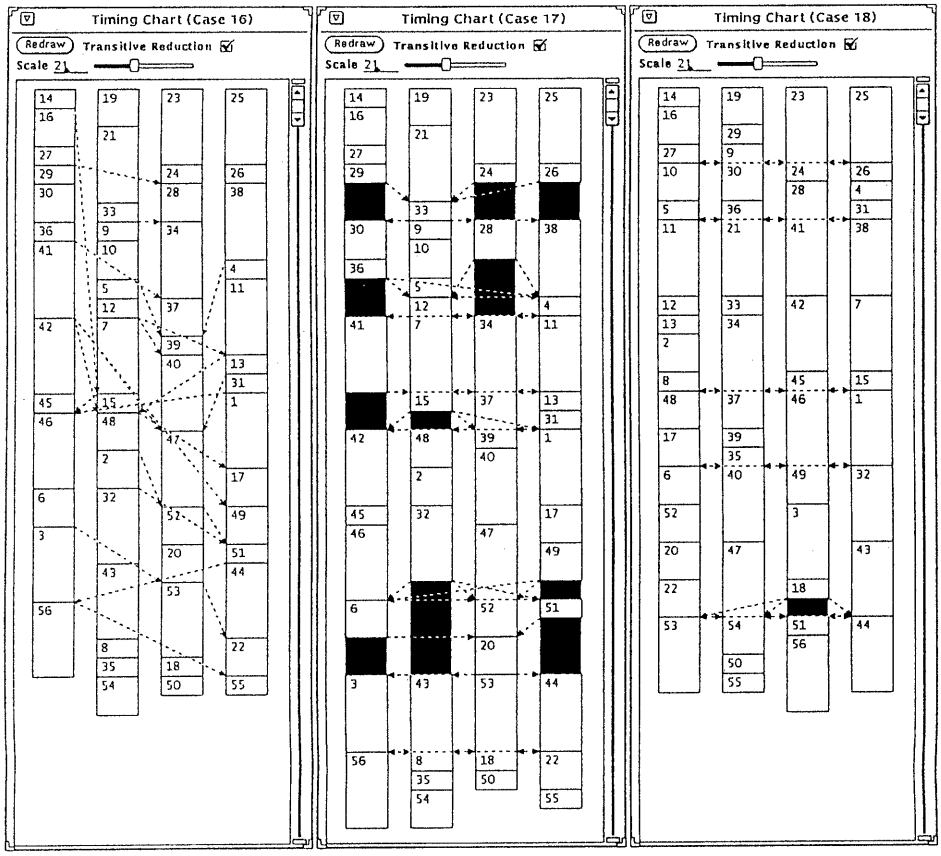
4.1 サンプルプログラムによる評価

アルゴリズムに与える入力サンプルとして, 文献 [6] の図 7 より引用した $G_{\text{IEICE-J73DI-9-p762-F7}}$ を用いる. このサンプルは, スパースな係数行列を持つ連立一次方程式の求解プログラムである[6]. タスク数は 56 で, 先行制約集合は文献のプログラムから読みとって作成した. 各タスクの処理時間は文献から読みとられなかったので, それぞれ 1, 2, 4 のなかからランダムに設定した.

このサンプルに対して次の 3 つの場合について実行タイミングを求めた. 第一は, CP-DTSP 法により S を求めた上での, 実行系 M_0 による実行であり, このとき保証される先行制約の集合を $\prec_{M_0}^{\text{CP-DTSP}}$ と書く. 第二は, CP-DTSP 法により S を求め, これを入力として文献 [1] のバリア挿入アルゴリズム INSERT-BARRIERS-2 により B を求めた上での, 実行系 M_B による実行であり, このとき保証される先行制約の集合を $\prec_{M_B}^{\text{IB2}}$ と書く. 第三は, 本稿で示したアルゴリズム ALLOC-TASKS-WIRH-BARRIERS で S と B を求めた上での, 実行系 M_B による実行であり, このとき保証される先行制約の集合を $\prec_{M_B}^{\text{ATWB}}$ と書く.

この結果を図 2 に示す. この例ではプロセッサ数を 4 として実行タイミングを求めている. 図中, 左上角に番号の書かれた白色の長方形がひとつのタスクを表し, 番号はタスク番号 (t_i の i) を示す. この長方形の縦の長さがそのタ

†クリティカルパス長の大きいものから優先して割当てるのは, CP 法がそうするのと同じ理由による.



(a) $\prec_{M_0}^{\mathrm{CP-DTSP}}$ (b) $\prec_{M_B}^{\mathrm{IB2}}$ (c) $\prec_{M_B}^{\mathrm{ATWB}}$

図 2：サンプルプログラム $G_{\mathrm{IEICE-J73DI-9-p762-F7}}$ の各実行系による実行タイミング ($|P| = 4$)

スクの処理時間を表しており，縦向き下方向が時間経過を表している．縦に接して並んでいる長方形群はひとつのプロセッサで実行されるタスクの集まりである．図中横向きに 4 つ並ぶ長方形群は，4 台のプロセッサによる並列実行を表している，プログラム全体の処理時間は，最も下の長方形底辺の位置によって比較される．また，点線の矢印は先行制約を表しており，図 (a) の t_{21} の上を横切っている先行制約は $t_{27} \prec t_{28}$ を表している．ただし図中では，$\prec_S$ および，$\{(t,t') \mid \exists_{t''}\, t \prec t'' \prec t'\}$ なる先行制約集合の表示が省略されている．図 (b),(c) に見られる黒色の長方形は，その時刻の間そのプロセッサがどのタスクも実行することなく休止していた (待ちが生じた) ことを意味している．

アルゴリズム IB2 および ATWB はどちらも，まずはじめに $t_{27} \prec_C t_{28}$ の先行制約を保証するためにバリアの挿入を検討している．アルゴリズム IB2 では，挿入するバリアの同期完了時刻として $\tau = \tau_{fin}^{\prec'_{M_0}}(t_{21})$ が選ばれており，バリア挿入点は $b_1 = \langle 4,2,2,2\rangle$ と決定されている．その他の挿入位置の候補として $\tau = \tau_{fin}^{\prec'_{M_0}}(t_{27})$，$b_1 = \langle 3,1,1,1\rangle$ や $\tau = \tau_{fin}^{\prec'_{M_0}}(t_{24})$，$b_1 = \langle 4,1,2,2\rangle$ などが考えられるが，この場合，$\prec_{M_0}^{\mathrm{CP-DTSP}}$ で最も遅く終了するプロセッサ p_2 の実行を遅らせることになるため得策でないと判断されたと推測される．

これに対して ATWB の場合では，$\tau = \tau_{fin}^{\prec'_{M_0}}(t_{27})$，$b_1 = \langle 3,1,1,1\rangle$ が選択されている．このときプロセッサ p_2 に 2 単位時間の待ち時間が生ずるが，IB2 の場合と異なり，バリア以降のタスク割当てはこの事態に応じて対応できるため問題とならない．さらにこの場合には，この待ち時間の時刻において t_{29} 及び t_9 の実行が可能であったため，これらがバリアの前に割当てられて p_2 には待ちが生じない結果となっている (この割当てによりバリア挿入点は $b_1 = \langle 3,3,1,1\rangle$ に補正されている)．

IB2，ATWB それぞれにおけるこれ以降のバリア挿入については，この最初のバリア挿入によって，バリア以降の実行タイミングは図 (a) とは異なるものとなり，それぞれのアルゴリズムにもとづいて挿入位置が決定されている．

最終的に各実行系の処理時間は $\tau_{total}^{\prec_{M_0}^{\mathrm{CP-DTSP}}} = 33$，$\tau_{total}^{\prec_{M_B}^{\mathrm{IB2}}} = 39$，$\tau_{total}^{\prec_{M_B}^{\mathrm{ATWB}}} = 33$ となった．IB2 によるスケジューリングが，M_0 に対して 14.6% の性能低下をひき起こしているのに対し，ATWB では M_0 と同じ性能を達成している．

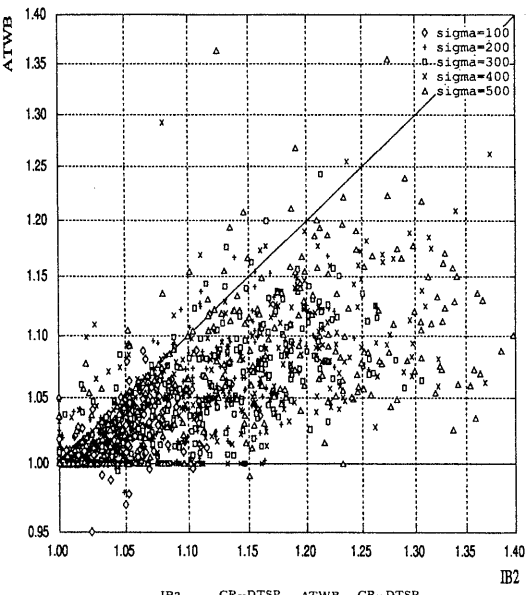


図 3 : $\tau_{total}^{\prec_{M_B}^{IB2}}/\tau_{total}^{\prec_{M_0}^{CP-DTSP}}$ と $\tau_{total}^{\prec_{M_B}^{ATWB}}/\tau_{total}^{\prec_{M_0}^{CP-DTSP}}$ の分布

4.2 ランダムに生成したプログラムによる評価

スケジューリングアルゴリズムの性能は，入力となる G に強く依存する．ここでは，アルゴリズムの一般的な性能を把握するために，ランダムに生成された多数の G に対する性能を評価する．

G は，文献 [1] で示したアルゴリズム CREATE-RANDOM-PRECEDENCE により $\prec_C$ をランダムに決定し，$e(t)$ が 1000 を平均とする標準偏差 σ の正規分布をなすように e をランダムに決定して生成した．以下の評価では，同じ条件に対してそれぞれ $G_1, G_2, \ldots G_{300}$ の 300 例を用いている．

まずはじめに，前節で挙げた 3 つの実行系によるプログラムの処理時間を比較する．図 3 は，横軸に IB2 の処理時間，縦軸に ATWB の処理時間をとり，同じ G に対する両者の処理時間をひとつの点で表している．ただし各処理時間は $\tau_{total}^{\prec_{M_0}^{CP-DTSP}}$ により正規化されている．ここでは $|T| = 50, prob = 0.025, |P| = 5$ とし，$\sigma = 100, 200, 300, 400, 500$ のそれぞれについて 300 例の結果をプロットしている．図中斜めの直線は，その直線上にプロットされた場合が，両アルゴリズムの性能が等しかった場合であることを意味しており，直線より右下 (左上) にプロットされた場合は，ATWB によるスケジュールの方が処理時間が短かった (長かった) ことを意味している．

この結果から読みとれることとして，まず第一に，ほとんどの場合において ATWB 法によるスケジュールの方が処理時間を短くしていること，第二に，ATWB 法によって実行系 M_B が実行系 M_0 と同じ性能を達成できる場合が，かなり多く存在すること (X軸上のプロット)，第三に，しかし ATWB 法の方が IB2 より悪い結果をもたらしている場合もまれではあるが存在していること，そして最後に，ATWB 法により，実行系 M_B が実行系 M_0 より良い性能を達成する場合もごくまれではあるが存在すること (Y軸が負の値のプロット) などが挙げられる．特に最後のケースは，CP-DTSP 法による M_0 における Sの決定が最適でなかった場合に，ATWB 法によるスケジュールによって，偶然にも，より最適に近い S が求められたことを意味している．

σ	0	100	200	300	400	500
$[0.00, 1.00)$	0	9	1	1	0	2
$[1.00, 1.05)$	300	265	193	176	158	119
$[1.05, 1.10)$	0	25	82	75	93	94
$[1.10, 1.15)$	0	1	20	41	29	47
$[1.15, 1.20)$	0	0	4	6	16	27
$[1.20, 1.25)$	0	0	0	1	1	7
$[1.25, 1.30)$	0	0	0	0	3	1
$[1.30, 1.35)$	0	0	0	0	0	0
$[1.40, \infty)$	0	0	0	0	0	3
最小値	1.000	0.950	0.978	0.994	1.000	0.991
平均値	1.000	1.018	1.038	1.048	1.056	1.072
最大値	1.000	1.118	1.168	1.243	1.292	1.408

表 1 : $\tau_{total}^{\prec_{M_B}^{IB2}}/\tau_{total}^{\prec_{M_0}^{CP-DTSP}}$ の度数分布と最大/最小/平均値

またこのときの $\tau_{total}^{\prec_{M_B}^{IB2}}/\tau_{total}^{\prec_{M_0}^{CP-DTSP}}$ の度数分布と平均値を表 1 に示す．σ の値が大きくなるにつれて，性能劣化率の大きい場合の頻度が高くなっていることが読みとられる．これを平均値で評価すると，スケジューリングアルゴリズムとして ATWB を用いることにより，実行系 M_0 に対する実行系 M_B の性能劣化率を 7% 程度以内に抑えることができたといえる．

文献 [7] では，バリア同期用のあるスケジューリングアルゴリズムを用いて，本稿とほぼ同様の評価方法で実行系 M_B の性能を評価しているが，その結果として本稿における M_0 に対して 20 ～ 44% の性能劣化が生ずることを示している．ここで用いられたアルゴリズムに比較すれば，本稿で示した ATWB が十分に優れているものであることがわかる．

次に，ATWB の絶対性能を把握するために，$\tau_{total}^{\prec_{M_B}}$ の下界と比較する．下界 $LB_{\tau_{total}^{\prec_{M_B}}}(G)$ は，G の正しい実行を保証するような任意の S, B に対して

$$LB_{\tau_{total}^{\prec_{M_B}}}(G) \leq \tau_{total}^{\prec_{M_B}}$$

を満すような関数である．また，

$$\prec_{M_0} \subseteq \prec_{M_B}$$

であることから，

$$LB_{\tau_{total}^{\prec_{M_0}}}(G) \leq LB_{\tau_{total}^{\prec_{M_B}}}(G)$$

がいえる．ここでは $LB_{\tau_{total}^{\prec_{M_0}}}(G)$ として文献 [8] で示された Fernandez-Bussel の下界式を用いている．

図 4は，図 3と同じ条件で $\tau_{total}^{\prec_{M_B}^{ATWB}}/LB_{\tau_{total}^{\prec_{M_0}}}$ の分布を示したものである．グラフの横軸は G_i の i であり横方向の分布に特別な意味はない．この結果から，下界に一致した場合 (X軸上にプロットされている)，すなわち最適解が求められている場合が数多く存在していることがわかる (1500 例中 305 例)．

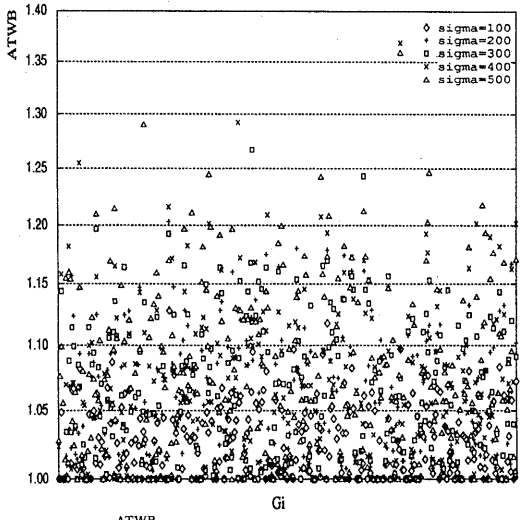


図 4：$\tau_{total}^{\prec_{M_B}^{ATWB}} / LB_{\tau_{total}^{\prec_{M_0}}}$ の分布 (横軸はサンプル番号)

4.3 アルゴリズムの計算コストについて

本稿で示されたアルゴリズムの書式は，アルゴリズムがどのような結果を返すかを正確に記述することだけを意図して書かれたものであり，このアルゴリズム中の各行をそのまま，より抽象度の低いコードに置き換えただけでは，アルゴリズム全体の計算コストはかなり大きなものとなってしまう．本稿の評価で実際に用いたコードは，C 言語で書かれており，約 700 行に及ぶものである．この実際のコードでは，同じ計算をできるだけ繰り返さないように，$\prec'_{M_0}$ などの複雑な計算には，ループの繰返しごとで差分だけを更新するようにするなどの工夫が施されている．この C 言語で書かれたコードの計算コストを見積もったところ，プロセッサ台数がタスク数に比べて十分に小さいとみなして，最悪時で $O(|T|^5)$ であった．しかし計算コストの削減に関して，さらなる改良を加えられる余地は残されている．

5. むすび

タスク集合 T とタスク間の先行制約 $\prec_C$，タスクの処理時間 e からなるプログラムと，プロセッサ集合 P が与えられたとき，プロセッサの実行タイミングを推定しながら，バリア同期以外の同期操作を併用することなくプログラムの正しい実行を保証し，かつ，できるだけ全体の処理時間 $\tau_{total}^{\prec_{M_B}}$ を短くするようなタスク割当て S およびバリア挿入位置 B を決定するアルゴリズム ALLOC-TASKS-WITH-BARRIERS を示した．

このアルゴリズムの入力としてランダムに生成した多数のプログラムを与え，スケジュール結果を評価したところ，理想的な実行系 M_0 に対して 7% 程度の性能劣化に抑えられることが示された．

この結果を文献 [9] と比較すると，文献 [9] では，バリア同期の性能を比較基準として拡張型バリア同期の評価を行なっているが，拡張型バリア同期の性能が十分に実行系 M_0 に近いと仮定して小さめに見積もっても，実行系 M_B の性能劣化率が 50% 以上であることを示している．この文献では，実際に与えられた G に対してスケジューリングを行なうといった評価方法ではなく，バリア間に存在するタスクの処理時間をランダムに決定するという評価方法が用いられている．これは，ランダムなスケジューリングを前提とした場合の評価とみなすことのできるものである．これに対して本稿の結果は，スケジューリングアルゴリズムを工夫することによって，拡張されていないバリア同期であっても十分な性能を達成できることを示したといえよう．

また，実行系 M_B が実行系 M_0 より平均的に性能が劣るという結果ではあるが，本稿では同期などの操作にかかる時間的コストは無視できると仮定しており，これは，M_B では十分に現実的な仮定であっても，ソフトウェアによる同期操作などを必要とする M_0 では現実的な仮定ではない．この点を考慮に入れれば，7% 程度の性能差しかないという今回の結果は，本来 fork/join 型の並列処理形態で用いられてきたバリア同期も，任意の先行制約を保証する並列処理方式においても，それなりに実用的に用いることができるということを示している．

参考文献

[1] 高木浩光, 有田隆也, 川口喜三男, 曽和将容: バリアを唯一の同期手段とした場合のタスクスケジューリング, 信学技報, CPSY93-22 (1993).

[2] Kohler, W.H.: A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems, IEEE Trans. Comput., No.12, pp.1235–1238 (1975).

[3] Kasahara, H. and Narita, S.: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, IEEE Trans. on Computers, Vol.C-33, No.11 (1984).

[4] Shirazi, B and Wang, M.: Analysis and Evaluation of Heuristic Methods for Staric Task Scheduling, Journal of Parallel and Distributed Computing, Vol.10, pp.222–232 (1990).

[5] 高木浩光, 有田隆也, 曽和将容: 実行タイミングの動的変動に強い静的プロセッサスケジューリング, 電子情報通信学会論文誌 (D-I), J75-D-I, No.9, pp.431–439(1990).

[6] 本多弘樹, 水野 聡, 笠原博徳, 成田誠之助: OSCAR 上での Fortran プログラム基本ブロックの並列処理手法, 電子情報通信学会論文誌 (D-I), J73-D-I, No.9, pp.756–766(1990).

[7] 高木浩光, 有田隆也, 曽和将容: 細粒度並列実行を支援する種々の静的順序制御方式の定量的評価, 並列処理シンポジウム JSPP'91 論文集, pp.269–276 (1991).

[8] Fernandez, E.B. and Bussel, B: Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules, IEEE Trans. on Computers, Vol.C-22, No.8, pp.745–751 (1973).

[9] 山家 陽, 村上和彰: バリア同期モデル — Taxonomy と新モデルの提案, および, モデル間性能比較 —, 並列処理シンポジウム JSPP'93 論文集, pp.119–126 (1993).