抽象状態同期のカーネルレベル実装

大木 敦雄, 久野 靖†

抽象状態とは、データ構造ないしオブジェクトの状態を複数の(あまり多くない)集合に排他的に分割し、現在の状態がそのいずれに含まれるかを明示的に管理する方式を言う。抽象状態同期とは、排他領域によってガードされる対象の抽象状態を同期機構側で保持し、スレッドなどの実行主体が排他領域に入れる条件を抽象状態の集合として表現する方式であり、自然なコード記述で効率よい条件同期を実現可能である。筆者らは、POSIX スレッドライブラリ API を拡張して抽象状態同期に対応させることを提案し、その記述性と性能について評価を行って来た。本論文では、FreeBSD 6.0-RELEASE のマルチ CPU 対応スレッドライブラリとそれが利用するカーネル側プリミティブを改造して抽象状態同期を組み込んだ実装、およびこの実装と既存の同期機構との性能比較を中心に報告する。性能評価では、抽象状態同期はスレッド数が多い場合に、現在広く使われているプログラミングスタイルである反復判定(排他領域の入口において排他領域に入るためのガード条件式を繰り返し評価する方式)の数倍以上の性能を有するという結果が得られている。

Kernel-level Implementation of State Abstraction-based Synchronization

Atsuo OHKI, Yasushi KUNO†

"State Abstraction" is a framework in which states of a data structure (or an object) are divided into small number of exclusive sets ("Abstract States," or AST), and current abstract state is explicitly managed by the code. In "State Abstraction-based Synchronization" (or AST-sync), abstract states of the data structure guarded by a critical region are similary managed, and an activities can enter the region only when curent state is included in the pre-specified set (e.g, in a bounded buffer, "get" operation can proceed only in "full" or "mid" state, but not in "empty" state). AST-sync can express conditional synchronization in conscice and readable manner, and can be efficiently implemented. We have previously proposed an extension of POSIX Thread API which supports AST-sync, and have discussed its expressiveness and performance. In this paper, we report our kernel-level implementaiton of AST-sync based on multiprocessor-capable POSIX Thread libray included in FreeBSD 6.0-RELEASE and its performance evaluation. As the result, our implementaiton outperforms standard guard-based synchronization by factor of three to five, when the number of threads are large.

1. はじめに

マルチプロセッサシステムや分散システムなどの並列システムでは、動作主体間で同期(synchronization)を取る手段が不可欠である。並列システムのモデルとして共有メモリモデルが使われている場合(分散共有メモリを含む)、共有対象となるデータ構造やオブジェクトの整合性を保つために、排他領域(critical regions)が使用される。この場合、同期は基本的に、排他領域に入ろうとする実行主体の実行を他の実行主体が排他領域中を実行している間遅延させる形で実現される。

ただし、他の実行主体がいない時でも、場合によっては実行主体の実行を遅延させる必要がある。たとえば有限バッファでは、そこからデータを取り出そうとする実行主体は、バッファが空であればその作業を遂行できないので、他の実行主体がバッファにデータを追加してくれるまで待つ必要がある。このような同期を一般に、特定の条件が成り立つまで実行を遅延させることから「条件同期」(conditional synchronization)と呼ぶ。

条件同期の記述方法としては、条件変数(condition variable)[4]、セマフォ(semaphore)[5]、条件つき排他領域(conditional critcal cegion、CCR)[3]などが代表的であるが、いずれも弱点がある。

セマフォは一種のカウンタであり、蓄積されているデータ数などのように、個数として数えられるものの条件同期以外には適さない。

条件変数は待ち合わせのためのキューのようなもので

† 筑波大学ビジネス科学研究科
Graduate School of Business Sciences, University of Tsukuba, Tokyo.

あり、1つの待ち合わせ条件につき1つの条件変数を使用するため、条件が増えて来ると繁雑となり間違いを侵しやすい上、複数の待ち合わせ条件に重なりがある場合に対応できない。

CCR は実行主体が待つべき条件を論理式で表現することから一般性があるが、論理式が参照する値が変更される (可能性がある) ごとに毎回条件評価が必要であり、効率のよい実装が難しい。たとえば、CCR の論理式をトランザクションメモリ上で待ちなし評価することで効率的な実行を可能とする研究[1] があるが、論理式中のすべてのメモリアクセスをトランザクションとして実装するという強い制約を持つ。

筆者らは、上記の弱点を克服した複数の実行単位間の並行制御のための新たな機構として、抽象状態同期 (abstract state synchronization, AST) を提唱し、そのオブジェクト指向言語への組み込み[8)9)]、および C や C++ などの直列言語から呼び出すスレッドライブラリ API への組み込み[10)] について研究してきた。

本論文では、FreeBSD オペレーティングシステムに標準で搭載されているマルチプロセサ対応のスレッドライブラリおよびそれが使用するカーネル機能を改造し、抽象状態同期機構を組み込んだ経験について報告する。また、改造したスレッドライブラリ上での抽象状態同期の性能を、既存の条件同期方式と比較した結果についても述べる。

以下第 2 節では、抽象状態同期の基本概念およびそのスレッドライブラリ API への組み込み方式について説明する。第 3 節では、FreeBSD 6.0-RELEASE 付属のスレッドライブラリへの抽象状態同期の組み込みと、その作業途上で得た知見について報告する。第 4 節と第 5 節では、第 3 節で報告した実装の性能評価について述べる。第 6 節でまとめをおこなう。

2. 抽象状態同期とスレッドライブラリ

2.1 抽象状態と抽象状態同期

オブジェクト指向言語では、1つのオブジェクトがさまざまな状態を取ることができる。オブジェクトの状態はインスタンス変数群の値によって定義されるが、そのオブジェクトを外部から (抽象データ型として) 扱う場合、すべての状態の違いを区別することは必ずしも必要でない。

たとえば有限バッファの場合、そこにさまざまな値がさまざまな順序で格納されているとき、それらの値や順序の違いはすべて異る状態に対応する。しかし、有限バッファを利用する側から見れば、興味があるのはバッファが「空、中間、満杯のいずれの『状態』にあるか」だけのことが多い。そこで、これらの「状態」に {empty, mid, full} のように名前をつけて明示的に扱うことで、実際に必要な状態の違いだけを区別できる。これらの「状態」を、オブジェクト内部の状態を抽象化したものであることから、**抽象状態**と呼ぶ。

通常のオブジェクト指向言語では、プログラマが ad hoc に `isEmpty()`、`isFull()` などの述語メソッドを用意することで実質的に抽象状態の情報を提供しているが、(1)ad hoc なので何が状態であるか分かりにくく情報の系統的な活用が難しい、(2) メソッド呼び出しのオーバヘッドがある☆、(3) 抽象状態が変化していなくてもそのつどメソッド内で条件式を評価する無駄が生じる、という弱点がある。

これに対し抽象状態を用いる場合、特定のインスタンス変数等に現在の状態値を保持し、それが変化する場合は抽象データ型内部のコードがこの値を正しく更新することとした。これにより、条件式の評価を必要最低限にとどめることができ、外部からも効率よく抽象状態値を参照できる。

抽象状態同期は抽象状態を条件同期機構に利用するものであり、オブジェクトごとに抽象状態を保持することは前節と同様だが、オブジェクトに対応する排他領域に入る実行主体は、その入口で行おうとする操作が実行可能な抽象状態集合を明示し、現在の抽象状態がその集合に含まれない場合には実行を遅延する。また、排他領域内に入った実行主体は、出口において次の抽象状態を設定する。その結果、その状態で実行を許される遅延中の実行主体から1つが選ばれて実行を再開する。

抽象状態同期では、抽象状態の数がハードウェアの1語のビット数以内であれば、各状態を1ビットに対応させ、ビット演算により遅延が必要かどうかを効率よく判定できる[8)]。また、クラス間の継承関係がある言語に適用する場合でも、親クラスにおける抽象状態を子クラスの複数の抽象状態に系統的に分割することで継承異常問題を回避し、同期記述の再利用を可能にできる[9)]。

2.2 スレッドライブラリへの抽象状態同期の導入

抽象状態同期は、並行オブジェクト指向言語に組み込みの同期機構として採用することもできるが[8)9)]、C や C++ などの直列言語から呼び出すスレッドライブラリと組み合わせて使用することもできる。以下でその方式および C や C++ から呼び出すためのライブラリ API について解説する (プログラムの記述性に関する検討については[10)] を参照されたい)。

以下本稿では、普及しているスレッドライブラリ

☆ インライン展開などの最適化は可能である

表 1　抽象状態同期に基づく同期 API

呼び出し API	機能
int ast_mutex_init(struct ast_mutex *m, int state)	初期化
int ast_mutex_destroy(struct ast_mutex *m)	廃棄
void ast_mutex_enter(struct ast_mutex *m, int mask)	排他領域に入る
void ast_mutex_exit(struct ast_mutex *m, int state)	排他領域を出る

API である POSIX Thread[6] を取り上げる。POSIX Thread では、排他領域は mutex ロックによって実現され、その中での条件同期は条件変数によって実現される。これらを用いて、有限バッファにデータを格納する関数、そこからデータを取り出す関数はたとえば次のように書ける (mutex は mutex ロック型、nonfull、nonempty は条件変数型として定義されているものとする)。☆

```
void put(T p) {
  pthread_mutex_lock(&mutex);
  while(バッファが満杯)
    pthread_cond_wait(&nonfull, &mutex);
  /* 有限バッファにデータを格納 */
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&mutex);
}
void get(T *p) {
  pthread_mutex_lock(&mutex);
  while(バッファが空)
    pthread_cond_wait(&nonempty, &mutex);
  /* *p に有限バッファからデータを取り出し */
  pthread_cond_signal(&nofull);
  pthread_mutex_unlock(&mutex);
}
```

複数の条件変数を使い分けるのは明らかに繁雑で間違いを侵しやすい。別のプログラミングスタイルとして、上記のコードで条件変数を 1 つだけ使用し、signal の代わりに broadcast を使ってすべてのスレッドを起こすように直したものも多く使われる。この場合、条件同期は「while に書かれた条件式が成り立つまで待つ」という形で行われ、実質的に CCR として動作することになる。本稿ではこれを反復判定と呼ぶ。Java のように 1 つのロックに対して 1 つの条件変数しか提供しない言語では、反復判定が標準的な条件同期のスタイルとなる。

通常の POSIX スレッドでは排他領域の入口で条件式を評価することになるのに対し、抽象状態同期では抽象状態の集合によって待ち合わせ条件を記述する。具体的には、1 つのロックが 1 つのオブジェクトに対応するものと考え、ロックのデータ構造中にそのオブジェクトの抽象状態を保持させる。そしてロック獲得時にライブラリ呼び出しの引数として、排他領域に入ることが許される抽象状態の集合を渡す。そして、各コードは、排他領域を出るところで新しい抽象状態を設定する。抽象状態の初期値は、オブジェクトの初期状態に対応したものをロック初期化の際に指定する。

今回設計した API を表 1 に示す。この API では、各抽象状態を 1 ビットで表わし、1 語 (int 型) のマスクで抽象状態の集合を表すこととした。このため、抽象状態の最大数は int のビット数である 32 (ないし 64) となるが、「データ構造が持つすべての状態を少数のものに抽象化して扱う」ためにはこの程度で十分と考えている。☆☆この API を用いると、有限バッファの格納 / 取得操作は次のように記述できる (初期値は S_EMPTY に設定し、またバッファサイズは 2 以上であるものとする)。

```
void put(T p) {
  ast_mutex_enter(&mutex, S_MID|S_EMPTY);
  /* 有限バッファにデータを格納 */
  if(バッファが満杯)
    ast_mutex_exit(&mutex, S_FULL);
  else
    ast_mutex_exit(&mutex, S_MID);
}
void get(T *p) {
  ast_mutex_enter(&mutex, S_FULL|S_MID);
  /* *p に有限バッファからデータを取り出し */
  if(バッファが空)
    ast_mutex_exit(&mutex, S_EMPTY);
  else
    ast_mutex_exit(&mutex, S_MID);
}
```

☆ 条件変数で待っていたスレッドが起きた時には通常は同期条件が成り立っているので、再度条件を確認する必要はないが、そのスレッドが起きる前に他のスレッドがロックを獲得してしまうことがあるため、繰り返し判定が必要である (実際にループを回る回数は少ない)。条件変数の仕様が「signal で起こされたスレッドが優先的にロックを獲得できる」となっていれば、ここに示すコードの while は if で済む。POSIX Thread ではそのような保証はない。

☆☆ たとえば通常のデータ構造を扱うときに、データ構造の状態を if 文などで 32 通り以上に枝分かれして扱うようなコードはほとんど現れないはずである。

本方式では ast_mutex_enter を通過した後では対応するデータ構造は直ちに操作可能な状態になっているので、条件判定やその反復は不要である。ロックは ast_mutex_exit により解放するが、そのときロックを操作後の新しい抽象状態に遷移させる。

まとめると、enter から exit までの間が排他領域であり、ある実行主体が状態集合を指定して enter しようとしたとき、現在の状態が集合に含まれないか、他の実行主体がその中を実行中の場合は待たされる。そして、実行主体が exit した時には設定された抽象状態に適合する実行主体の中から 1 つを選んで排他領域に入れるものとする。この場合、状態の遷移させ方によっては飢餓が起きることもあり得るが、それはプログラマの責任と考える。

3. 実　　装

筆者らは FreeBSD 6.0-RELEASE に付属するスレッドライブラリを改造して、前節で述べた API による抽象状態同期機構を実装した。FreeBSD には POSIX Thread API を提供するスレッドライブラリとして、(1) ユーザレベルのみでスレッド機能を実装するもの (libc_r)、(2) カーネルスレッドとユーザレベルスレッドが 1:1 に対応するもの (libthr)、(3) 複数のカーネルレベルスレッドが複数のユーザレベルスレッドに M:N 対応するもの (libpthread) の 3 つが標準で備わっている。今回はカーネルレベルスレッドの機能を拡張して抽象状態同期を導入したため、(2) をもとに改造を行った。☆

今回改造した部分に関係するユーザ空間側のデータ構造を図 1に示す。pthread_mutex はオリジナルのライブラリの mutex データ構造であり、その中にある umtx_t 型のデータはカーネル側のロック機構が使うデータ構造であり、このユーザ側データ構造の排他アクセスに使用する。pthread_cond は同じく条件変数のデータ構造であり、こちらはこのデータ構造の排他アクセスと、カーネル内部にあるキュー構造 (後述) の排他アクセスのため、2 つのカーネルロック (umtx_t) を持つ。ast_mutex は抽象状態同期 API が使う新たなデータ構造であるが、中身はオリジナルの pthread_mutex と現在の抽象状態値を表す 1 語とから成る。

カーネル空間側のデータ構造を図 2に示す。umtx_chain はハッシュ表であり、双方向連結リストのヘッダとなっている。mutex や条件変数の数が多くならなければ、1 つのスロットは 1 つの mutex や条件変数で待っているスレッドだけの連結リストと考えてよい。連結リストの各要素には、オリジナルのカーネルではスレッド ID とユーザ空間側のカーネルロック (umtx_t データ構造) のアドレスが格納されていたが、今回の改造ではそれに次の 2 つのフィールドを追加した

- ast_bits — 待っている状態集合を表すマスク値。
- owner — 待っているスレッドが起こされる時に umtx_t のロックを獲得するのに使用する値。

これらのフィールドは抽象状態同期によって待ちに入った場合にのみ使用される。☆☆

なお、連結リストに待ちスレッドを挿入する場合、先頭 / 末尾のいずれに挿入することも同じ手間で可能だが、オリジナルの実装では先頭に挿入するようになっていた。これは、同期条件などによる制約がない範囲で

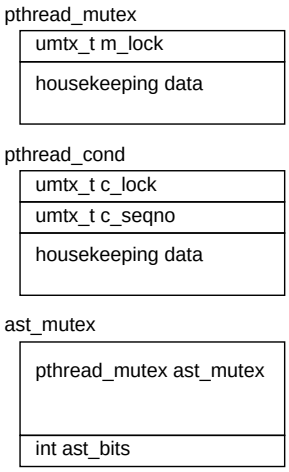

図 1　ユーザ空間側のデータ構造

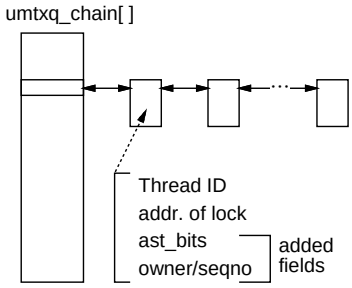

図 2　カーネル空間側のデータ構造

☆ (1) はユーザレベルのみでの実装であり、マルチプロセサでも単一 CPU しか使用しないため、また (3) はカーネルスレッドとユーザレベルのスレッドが M:N 対応であり、改造が複雑になることから、今回は利用しなかった。

☆☆ このほか一方のフィールドには、条件変数による待ち合わせ時に、後述するバグ除去作業の一環として「何番目の signal/broadcast 以降に待ちに入ったか」を表す番号を格納し、競合に起因するバグが発生していないことをチェックするのに使用するようにした。

は、一番最近まで動いていたスレッドが先にスケジュールされることになり、キャッシュヒット率などの点で有利なためと予想される。今回はこの点については変更しなかった。

ライブラリのコードからのカーネル呼び出しは_umtx_op() という専用のシステムコールにより、そのパラメタとしてオリジナルでは次の4つの操作が指定できた。

- UMTX_OP_LOCK — umtx_t をロックする
- UMTX_OP_UNLOCK — umtx_t を解放する
- UMTX_OP_WAIT — umtx_t で寝る
- UMTX_OP_WAKE — umtx_t で寝ているスレッドをパラメタで指定した個数起こす

ここで、抽象状態同期を実現するため、後2者の拡張版を次のように追加した。

- UMTX_OP_WAIT2 — 状態マスクを指定して umtx_t で寝る
- UMTX_OP_WAKE2 — 状態値を指定して umtx_t で寝ているスレッドからマスクに適合するもの1つを起こす

後者は連結リストの先頭から1個ずつ探索を行い、最初に見つかった適合するスレッドを起こすという単純な実装としている。より複雑なリスト構造を用いて探索を無くすこともできるが、次節で述べる評価の範囲 (数百スレッド程度) では単純な実装でも性能劣化は観察されなかった。

また、オリジナルの実装では pthread_mutex_lock はユーザ空間側の mutex データ構造の更新と umtx_t の獲得をまとめて行っていた。しかし抽象状態同期では、umtx_t を「解放することなく次の起きるべきスレッドに引き継ぐ」必要がある (上記UMTX_OP_WAKE2 は実際そのようになっている) ため、ユーザ空間側の mutex データ構造のみを更新する関数と umtx_t の操作を分離した。

しかしその後、評価のためのベンチマークを実行してみると、オリジナルの実装にバグがあることが分った。すなわち、条件変数内の2つの umtx_t のうち1つ目 (ユーザ空間側の mutex 構造をガードする) を解放してから、2つ目 (キューで待つための umtx_t) で待ちに入るために、競合が起きるというものであった。これを防ぐために、1つ目の umtx_t をロックしたままカーネルに入り、2つ目で待ちに入る直前に1つ目を解放するように修正した。

なお、このようなバグ取りを行ったものは、オリジナルの条件変数の実装に比べて性能が向上している。これは単純に1回の操作当たりのシステムコール回数が2回から1回に減少したためと思われる。次節で述べる性能評価では公平のために、オリジナルではなくこのバグ取り後の条件変数の実装を比較対象としている。

表 2 変更を行った行数

ファイル名	変更行数	全行数
thr_mutex.c	60	1678
thr_cond.c	16	344
thr_umtx.c	22	80
thr_umtx.h	3	87
kern_umtx.c(kernel)	181	772

変更したコードはスレッドライブラリ (libthr) 側の全60ファイル中4ファイル、カーネル側の1ファイルであった。変更量を表2に示す。変更に要した作業量は筆者の一人が作業して丸3日程であった (元のライブラリとカーネルの構造読解を含む。また前述のバグへの対処は期間にして1週間ほど要した)。

4. 評価 1(2CPU)

改造したスレッドライブラリを評価するため、条件同期を必要とする小さなベンチマークによる実行時間計測を行った。ベンチマークとしては次の2つを用いた:

- 通常の有限バッファ。状態として {empty, mid, full} の3つがあり、スレッドは「full である間遅延」(put するスレッド)、「empty である間遅延」(get) するスレッド) の2つがある。
- ウォーターマークつき有限バッファ。バッファに半分以上データが入っているかどうかを区別するため、状態が {empty, midlow, midhigh, full} の4つとなる。put するスレッドに2番目の種類として「empty|midlow」である間遅延」を追加したもの。

いずれも、有限バッファの個数は1個、容量はデータ10個で、put するスレッドと get するスレッドを複数個用意して動作させた。

有限バッファ (内部で条件同期を行う) の実装としては、次の4種類を用意した。

- 条件変数を用いた条件同期 — 2.1 節の前半に掲載したもの。ただし、条件変数は待ち合わせ条件に重なりがあると使えないため、ウォーターマークつき有限バッファの実装はない。
- 反復判定による条件同期 — 2.1 節で説明したように、条件変数を1つだけとし、常に broadcast ですべてのスレッドを起こして条件を調べさせるようにしたもの。
- 可搬版の抽象状態同期 — 抽象状態同期 API を POSIX Thread API を用いて実装したもの[10]。実装内部では反復判定を行っている。

表 3 通常の有限バッファ / 待ち数小 (500x100:500x100)

実装の種類	ユーザ時間		システム時間		経過時間	
条件変数	1.273	(0.100)	12.277	(0.176)	7.16	(0.100)
反復判定	3.133	(0.942)	34.181	(11.461)	20.26	(6.805)
可搬版	3.080	(0.881)	33.637	(10.097)	19.91	(6.012)
カーネル版	0.601	(0.071)	8.269	(0.264)	6.41	(0.203)

表 4 通常の有限バッファ / 待ち数大 (500x100:1x50,000)

実装の種類	ユーザ時間		システム時間		経過時間	
条件変数	1.081	(0.087)	9.872	(0.244)	5.80	(0.106)
反復判定	2.713	(0.254)	28.684	(2.230)	16.99	(1.356)
可搬版	2.785	(0.215)	29.366	(1.890)	17.40	(1.128)
カーネル版	0.530	(0.060)	7.340	(0.129)	5.75	(0.083)

表 5 印つき有限バッファ / 待ち数小 (250x100:250x100:500x100)

実装の種類	ユーザ時間		システム時間		経過時間	
反復判定	4.161	(1.065)	46.892	(13.262)	27.83	(7.865)
可搬版	3.934	(0.894)	44.130	(11.312)	26.19	(6.698)
カーネル版	0.594	(0.073)	8.261	(0.229)	6.38	(0.169)

表 6 印つき有限バッファ / 待ち数大 (250x100:250x100:1x50,000)

実装の種類	ユーザ時間		システム時間		経過時間	
反復判定	3.158	(0.249)	33.392	(2.216)	19.82	(1.328)
可搬版	2.924	(0.198)	30.858	(1.562)	18.27	(0.930)
カーネル版	0.539	(0.060)	7.432	(0.126)	5.83	(0.080)

- カーネル版の抽象状態同期 — 前節で解説したもの。

いずれのケースについても、待ちスレッド数が少ない場合と多い場合を比較するため、put() するスレッドと get() するスレッドが同数の場合と、get() するスレッドを 1 個だけにした場合 (当然、多数の put() 側スレッドが待ち状態に入る) とで計測を行った。

以下では通常の有限バッファで put する側のスレッド数 N_w 個、各スレッドが put する回数 C_w 回、get する側のスレッド数 N_r 個、各スレッドが get する回数 C_r 回のとき (N_wxC_w:N_rxC_r) のように記す。さらにウォーターマークつき有限バッファでは、バッファに半分以上入っている場合に put しないスレッド数 N_v 個、これらが put する回数 C_v 回のとき (N_wxC_w:N_vxC_v:N_rxC_r) のように記す。

計測はすべて Pentium II 350MHz 2CPU 構成で行い、使用したコンパイラは GCC 3.4.4、最適化は -O4 を指定した。計測はすべて 100 回ずつ行い、平均値と標準偏差を示す (単位は秒)。計測結果は表 3 〜表 6 の通り (いずれも単位は秒、かっこ内は標準偏差)。

これらを見ると、次のことが分かる。

- 反復判定と可搬版の性能はほぼ等しい。これは可搬版では内部で反復判定を行っているので、予想された結果である。
- 条件変数とカーネル版抽象状態同期では、カーネル版抽象状態同期の方がユーザ時間、システム時間ともやや優っている。これは、ユーザ空間側でもカーネル内でも抽象状態同期の方がコード的には簡潔であることと関係している可能性がある。
- ただし、待ちスレッドが多くなった場合はその優位は縮小している (表 3 と表 4 では表 4 が全スレッド数が半分になっていることに注意)。これは、今回の実装では待ちスレッドの双方向連結リストを線形探索して起こすスレッドを探していることと関係している可能性がある。
- カーネル版抽象状態同期に比べ、反復判定 (CCR に相当) および可搬版は 4 〜 5 倍程度の時間を要している (ユーザ時間、システム時間、経過時間とも)。これは、カーネル版抽象状態同期が動ける状態のスレッドを 1 つだけ起こすのに対し、反復判定および可搬版はすべてのスレッドを起こして条件チェックないしビットマスクのチェックを行わせることから予想された結果である。
- 印つき有限バッファでは、通常の有限バッファに比べて判定すべき状態が多くなり、またスレッドの種類数が増えていることから、反復判定および可搬版では、通常の有限バッファより所要時間が増大している。これに対し、カーネル版抽象状態同期ではこ

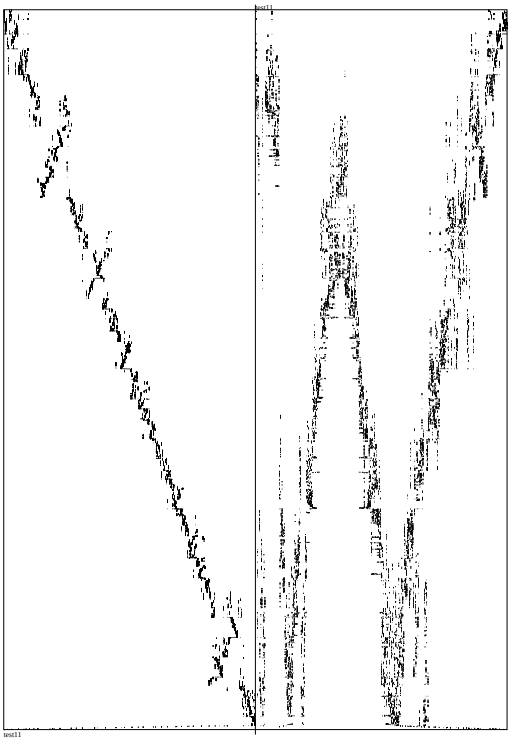

図 3　条件変数版のプロット

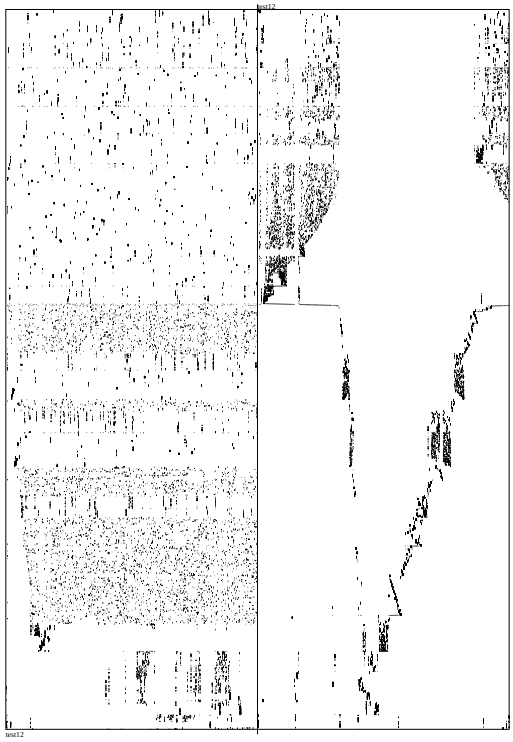

図 4　反復判定版のプロット

れらの要因にもかかわらず性能低下は見られない。さらに、条件変数版はウォーターマークつき有限バッファのように待ち合わせる複数の条件にオーバラップがある場合には適用できないが、抽象状態同期ではそのような制約はない。

より細かいスレッドの挙動を調べるため、有限バッファを通じて転送されるデータに時刻順に 0 〜 99999 の番号を割り当て、put() する側のスレッド、get() する側のスレッドにもそれぞれ 0 〜 499 の番号を割り当てて、何番のスレッドが何番のデータを put()/get() したかをプロットしてみた。図 3、図 4、図 5にそれぞれ条件変数、反復判定、カーネル版抽象状態同期のプロットを示す (すべて通常の有限バッファで読み書きスレッド数とも 500、縦軸がデータの番号、横軸がスレッド番号で、左半分が put()、右半分が get())。

これを見ると、反復判定版では毎回すべてのスレッドが起きるため、動くスレッドがバラバラになり易い傾向が見て取れる。これに対し、条件変数版では反復判定よりも動くスレッドに局所性があるが、やはり複数のスレッドが競合しながら動作していく様子が見て取れる。これに対し、カーネル版抽象状態同期ではそれぞれのスレッドがまとまって動き、競合が少なくなっている様子が分かる。

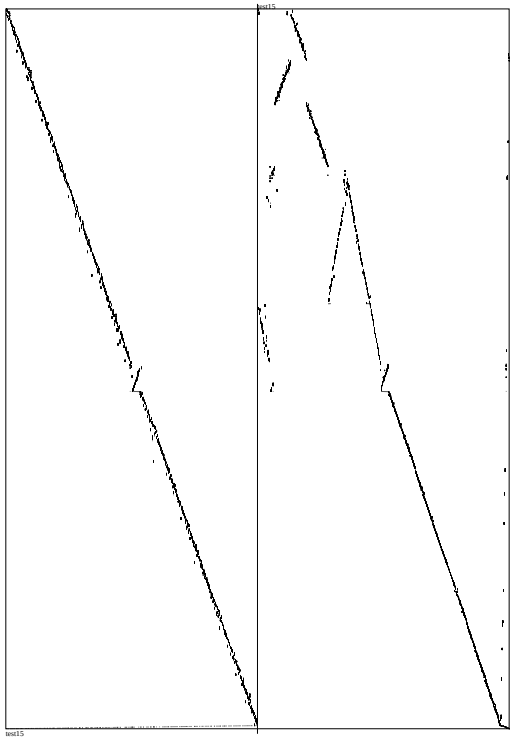

図 5　カーネル版抽象状態同期のプロット

これは、たとえばput()するスレッドが満杯で止まった場合はget()するスレッドが1つ選ばれて起こされて動き出し、次にこのスレッドが空で止まった時は最後に動いていた先のput()するスレッドが再び再開する、という形で制御が少数のスレッド間で渡され合うためではないかと想像される(ただし、他プロセスがスケジューリングされる等の要因による揺乱の影響も見られるので、決定的な動作になっているわけではない)。

なお、このような動作は前述の「待ち状態に入るスレッドを双方向連結リストの先頭に挿入する」という方式に起因しているものと考えられる。

5. 評価2(4CPU)

より並列度の高い場合の挙動について調べるため、Pentium D 3.2GHz 2CPU構成のマシンを用い、HyperThreading機能がoffの場合(論理CPU数2)とonの場合(論理CPU数4)で前節と同じベンチマークを実行した(CPU性能の増大に対応し、スレッド数とデータ量を増やしている)。その結果を表7〜表10に示す。

論理CPU数が2の場合のデータは当然、前節と同じ傾向を示している。これを基準に比較して、論理CPU数が4の場合のデータは待ち数が少ない場合(表7、9)と多い場合(表8、10)では変化の傾向が異っている。

待ち数が少ない場合:

- 条件変数版、反復判定版、可搬版とも、システム時間が3〜4倍と増えている。一方、カーネル版ではシステム時間が約20%増で済んでいる。
- ユーザ時間については、条件変数版、カーネル版は微増であるが、反復判定版、可搬版では約50%増となっている。
- 経過時間で見ると、条件変数版、反復判定版、可搬版とも所要時間が倍増しているが、カーネル版では約10%増程度である。

待ち数が多い場合:

- 条件変数版の傾向は、全体に待ち数が少ない場合と同様である。
- 反復判定版と可搬版では、システム時間は微増し、ユーザ時間が約半分となっている。全体として、経過時間は20%〜30%の減少となっている。
- カーネル版では、ユーザ時間は変わらず、システム時間は約50%増大しているが、経過時間は約20%増程度である。

とくに反復判定版と可搬版において、待ち数の多少で大きな傾向の違いが現れたのは、待ち数が少ないと多くのスレッドがスケジュールされ、それらが動き出して見ると条件が適合せずに再度寝ることが多いためと思われる(broadcastによって多数のスレッドが起こされるので、長い実行待ちキューができてしまい、自分が実際にスケジュールされたときに条件が適合している確率はほぼ半々になると予想される)。これに対し、待ち数が多い場合は多数の待ちスレッドが起こされるのは同じだが、get()側のスレッド(1個だけ)は常時動き続けるため、起きた後再度寝る数が少なくなっているものと予想される(ユーザ時間が減っていることでもこの予想を裏付けるものである)。

一方、条件変数版で論理CPU数が2から4になったときに一貫してシステム時間が大きく増えたことは意外であった。これはカーネル内部の無駄な競合が増えたことによるものと想像されるが、現時点では確からしい原因を推定するには至っていない。

全体として、さまざまな場合を通じて経過時間がもっとも少なく、その変動も小さく安定していたのはカーネル版であった(計測値の標準偏差も小さい値となっている)。カーネル版では論理CPU数が増えても無駄なユーザ時間の増大が見られず、システム時間はカーネルの作業量が増えるため増大したものの、経過時間への影響はさほど見られない。

6. ま と め

本稿では、スレッドライブラリに条件同期のための機構として筆者らが提唱している抽象状態同期を組み込む方式、およびFreeBSD 6.0-RELEASEのマルチCPU対応スレッドライブラリと対応するカーネル機能を改造して抽象状態同期を組み込んだ実装について解説した。性能評価の結果、抽象状態同期はさほど大きな手間なしに組め、多くの場面で使われている反復判定(ないしCCR)より優れた性能を持ち、スケジューラの方式にもよるが稼働するスレッドの局所性が高まるという特性を持つことが分かった。さらに、論理CPU数が増えた時にも安定して性能を発揮可能であった。

参 考 文 献

1) Tim Harris, Keir Fraser, Language Support for Lightweight Transactions, OOPSLA'03 Proceedings, pp. 388-402, 2003.
2) Maurice Herlihy, A Methodology for Implementing Highly Concurrent Data Objects, TOPLAS, vol. 15, no. 6, pp. 745-770, 1993.
3) C. A. R. Hoare, Monitors: Toward a theory of parallel programming, International Seminar on Operating System Techniques, A.P.I.C.Studies in Data Processing, vol.9, Academic Press, pp. 61-71, 1972.

表 7 論理 CPU 数 2 と 4 の比較: 通常の有限バッファ / 待ち数小 (1,000x1,000:1,000x1,000)

実装の種類	論理 CPU 数	ユーザ時間		システム時間		経過時間	
条件変数	2	5.189	(0.206)	59.255	(1.250)	33.58	(0.757)
	4	6.252	(0.448)	192.580	(11.929)	65.82	(3.523)
反復判定	2	6.658	(1.267)	85.804	(18.400)	49.19	(10.964)
	4	10.267	(1.305)	292.771	(37.942)	105.80	(12.686)
可搬版	2	6.884	(1.508)	86.683	(23.160)	50.09	(13.813)
	4	10.277	(1.214)	290.425	(34.602)	104.98	(11.606)
カーネル版	2	2.110	(0.113)	45.436	(1.634)	31.00	(0.874)
	4	2.200	(0.126)	54.224	(1.002)	33.01	(0.519)

表 8 論理 CPU 数 2 と 4 の比較: 通常の有限バッファ / 待ち数大 (1,000x1,000:1x1,000,000)

実装の種類	論理 CPU 数	ユーザ時間		システム時間		経過時間	
条件変数	2	4.763	(0.205)	44.719	(1.754)	26.46	(1.044)
	4	5.581	(0.226)	148.897	(3.933)	53.02	(1.271)
反復判定	2	14.599	(1.422)	196.267	(19.348)	116.47	(12.232)
	4	6.892	(0.252)	208.172	(4.350)	76.04	(1.336)
可搬版	2	14.044	(1.223)	185.283	(15.694)	110.19	(9.840)
	4	7.013	(0.250)	206.813	(5.044)	75.59	(1.538)
カーネル版	2	2.022	(0.125)	37.319	(0.622)	30.40	(0.412)
	4	1.953	(0.121)	56.588	(0.371)	35.22	(0.211)

表 9 論理 CPU 数 2 と 4 の比較: 印つき有限バッファ / 待ち数小 (500x1,000:500x1,000:1,000x1,000)

実装の種類	論理 CPU 数	ユーザ時間		システム時間		経過時間	
反復判定	2	7.315	(1.645)	95.298	(23.623)	54.61	(13.812)
	4	11.047	(1.488)	318.203	(40.249)	114.55	(13.871)
可搬版	2	7.711	(1.660)	97.840	(24.206)	56.56	(14.365)
	4	11.671	(1.610)	322.737	(43.995)	116.27	(15.231)
カーネル版	2	2.191	(0.127)	45.188	(1.581)	31.00	(0.845)
	4	2.495	(0.133)	54.739	(0.221)	33.38	(0.140)

表 10 論理 CPU 数 2 と 4 の比較: 印つき有限バッファ / 待ち数大 (500x1,000:500x1,000:1x1,000,000)

実装の種類	論理 CPU 数	ユーザ時間		システム時間		経過時間	
反復判定	2	14.472	(0.878)	191.331	(11.743)	112.04	(6.910)
	4	7.132	(0.269)	216.581	(4.407)	78.80	(1.442)
可搬版	2	13.371	(0.752)	170.729	(9.325)	100.36	(5.312)
	4	7.705	(0.272)	218.137	(3.281)	79.47	(1.073)
カーネル版	2	2.126	(0.105)	37.175	(0.528)	30.25	(0.364)
	4	2.200	(0.133)	56.487	(0.365)	35.42	(0.235)

4) C. A. R. Hoare, Monitors: an operating system structuring concept, CACM, vol.17, no.10, pp. 549-557, 1974.

5) E. W. Dijkstra, The Structure of THE Multiprogramming System, CACM, vol. 11, no. 5, pp. 341-346, 1968.

6) ISO/IEC 9945-1:1996 Information Technology – Portable Operating System Interface (POSIX) – Part 1, IEEE, 1996.

7) James Gosling, Bill Joy, Guy Steele, Guy L. Steele, The Java Language Specification, Addison-Wesley, 1996.

8) 久野 靖, 大木敦雄, 抽象状態同期に基づく並列オブジェクト指向言語 p6, 情報処理学会論文誌, vol. 38, no. 3, pp. 563-573, 1997.

9) Yasushi Kuno, Solving Inheritance Anomaly Problems by State Abstraction-Based Synchronization, in Jean-Paul Bahsoun, Takanobu Baba, Jean-Pierre Briot, Akinori Yonezawa eds., Object-Oriented Parallel and Distributed Programming, pp. 167-186, Hermes, 2000.

10) 大木敦雄, 久野　靖, スレッドライブラリへの抽象状態同期の導入, 情報処理学会プログラミング研究会発表資料, 2005-4-(5), 2005.

11) Butler W. Lampson, David D. Redell: Experience with processes and monitors in Mesa, CACM, vol. 23, no. 2, pp. 105-117, 1980.