ハイパフォーマンス
コンピューティング 72−17
（1998. 8. 7）

共有メモリ計算機における局所同期機構

建部修見† 関口智嗣†

分散メモリ型並列計算機でメッセージパッシングを用い，データ並列，パイプライン並列処理等を記述，実行することは容易であるが，共有メモリ計算機でそれらを記述する場合，POSIX Pthreads 等のスレッドライブラリに局所同期のためのプリミティブが用意されていないため，一般にそれらのプログラミングはコストがかかる．本研究では，必要な局所同期のためのプリミティブのデザインと実装を行い，プログラミング例を示すとともに Ultra Enterprise 3000, Origin2000 において評価を行う．メモリ，あるいはそのブロックの操作に対し 1-write 1-read の局所同期を用いることより，プログラミングが比較的容易になる．またそれらを不可分命令を用いて，あるいは用いないで実装することにより，低コストの局所同期を実現することができるが，Pthreads の排他ロック，条件変数を用いた実装でも粒度を大きくすれば性能を出すことができる．

Local synchronization facility for shared memory multiprocessors

Osamu Tatebe† and Satoshi Sekiguchi†

Data-parallel program and pipeline execution are easily programmed in message-passing style on distributed memory machines, however not easily on shared memory multiprocessors because the POSIX Pthreads interface does not support primitives for data synchronization. This paper proposes a design and some implementations of the primitives, and shows programming examples with these primitives, which are evaluated on Ultra Enterprise 3000 and Origin2000. Using the idea to access memory or memory block using 1-write and 1-read data synchronization, it is easy to program as well as the message-passing style. To get a low cost to synchronize, it is necessary to implement with or without atomic operations, however even though using POSIX mutex locks and condition variables, good performance is achieved when granularity of computation is large.

1. はじめに

共有メモリ機構を持つ計算機において並列プログラミングを行う場合，排他制御，同期制御が重要な役割を果たす．しかしながら Pthreads 等のスレッドライブラリは排他制御のためのプリミティブは提供しているが，バリア同期等の大域同期，またメッセージパッシング，生産者 / 消費者等の局所同期に関する同期機構のためのプリミティブは提供していないため，同期機構を伴うデータ並列，パイプライン並列処理等の処理のプログラミングにはコストがかかる．

データ並列に関しては大域同期であるバリア同期が用いられる．共有メモリ計算機におけるバリア同期に関しては既に多くの文献[4]がある．バリア同期は計算フェーズを揃え，ネットワーク，資源の不必要な混雑を解消することができるが，大域同期であるためコストが大きい．

パイプライン並列処理では大域同期は必要なく，軽い局所同期で実行を行うことができる．またデータ並列に関しても局所同期を用い実装を行った方が実装，実行の柔軟性が上がり，また同期のコストも下がる可能性がありメリットが多いと考えられる．

本論文では，局所同期のためのプリミティブを提案し，Pthreads の排他ロック，条件変数を用いた実装，また不可分命令を用いた実装，用いない実装を示す．そして，それらを用いたパイプライン処理についてのプログラミング例を示し，局所同期，バリア同期の基本性能評価，SOR 法によるパイプライン処理のプログラムの性能評価を SUN Enterprise 3000 (UltraSPARC I 168 MHz), SGI/Cray Origin2000 (MIPS R1000 195 MHz) を用い行う．

2. 局所同期機構

メッセージパッシングにおける point-to-point 通信では，送信完了，受信完了を待つことで送信，受信間の局所同期をとることができる．また，メッセージパッシングでは送信バッファ，受信バッファが別の領域であり，送信に関しては送信完了後には送信バッファは書き込

† 電子技術総合研究所
Electrotechnical Laboratory, AIST, MITI
E-mail: {tatebe,sekiguchi}@etl.go.jp

みが可であり，また受信に関しては受信発行以前は受信バッファに書き込まれることはない．

共有メモリ機構を持つ計算機の場合，送信バッファ，受信バッファを分ける必要がなく，また分けないことでそれらのバッファ間のコピーが必要なくなり，余分なオーバヘッドがなくなる．この時，データの移動はないためデータを書き込み可能か，あるいはデータが読み込み可能かという局所同期機構を用意すれば良い．この局所同期は 1-write 1-read の I-Structures[1]，あるいはバッファサイズ 1 のバッファに対する生産者 / 消費者における同期に相当するものである．I-Structures は 1-write N-read のデータ構造であるが，ここでは 1-write 1-read として，書き込まれていなければ読み込みはブロックし，書き込まれているものが読み込まれていなければ書き込みはブロックする構造としている．

また，このバッファを FIFO キューにすることにより書き込み可能かどうかの確認は必要なくなり，この時このバッファは Q-Structures に相当する．

3. 局所同期機構の実装

局所同期機構を実装するためには，I-Structures の実装と同様にメモリに presence bit と同等のものを付け加え，presence bit の更新とメモリの読み書きを atomic に行えば良い．Presence bit のデータ型を `I_st` とし，整数型データに対する書き込み，読み出しに対し，以下の様な関数を用意する．

```
void i_write_int(I_st* i_st, int* addr,
                 int val);
int i_read_int(I_st* i_st, int* addr);
```

これらの関数はデータの読み書きに対し局所同期を伴わせたものであり，メモリの書き込み，読み出しと presence bit の更新は atomic に行われる．しかしながら，上記のようなインターフェースでは以下のような問題がある．

- 必要なデータ型に全てに対しあらかじめ準備する必要がある．
- 構造体，連続データ，ストライドデータなどに対し局所同期をとる必要がある場合，細粒度にそれぞれのデータに対し同期をとると同期のコストが高くなってしまう．

始めの問題点に関しては C++ の関数テンプレートを用いれば解決することができるが，2 番目の問題点に関しては余分なコピーがおきてしまう．ここで，構造体については，構造体のポインタを渡すことにより余分なコピーが必要なくなるが，この場合参照先の構造体をさらに書き換えることに対し同期が必要になってしまう．そこで我々は以下のような関数を準備することによりこれらの問題の解決を行った．

```
void i_write_lock(I_st* i_st);
void i_write_unlock(I_st* i_st);
void i_read_lock(I_st* i_st);
void i_read_unlock(I_st* i_st);
```

`i_write_lock()` は書き込み可能になるまでブロックし，`i_write_unlock()` は書き込みの終了を表す．`i_read_lock()`, `i_read_unlock()` も同様にそれぞれ読み込み可能になるまでブロック，読み込み終了を表す．これらを用いれば `i_write_int()` 等は以下の様に定義することができる．

```
void i_write_int(I_st* i_st, int* addr,
                 int val)
{   i_write_lock(i_st);
    *addr = val;
    i_write_unlock(i_st);   }
```

ここで，局所同期とデータの読み書きを分離したため，厳密な意味のデータ依存による局所同期ではなく，書き手，読み手のそれぞれの操作に対するロックをかけることができる☆．

また，`I_st` の初期化のための関数も用意する．

```
void i_st_init(I_st* i_st);
```

ここで `i_write_lock()` と `i_read_lock()` はロックできない場合はブロックするため，MPI の point-to-point 通信において保証されている 2 プロセス間のメッセージの順序は保証される．

3.1 Pthreads による実装

基本的に実現したい局所同期機構は有限バッファの生産者 / 消費者であるため，セマフォを 2 つ使うか，あるいは同等の機構を条件変数，排他ロックを用い実装することになる．が，セマフォを用いると効率が悪いため同等の機構を条件変数，排他ロックを用い実装する．この時，ブロックした時に期限付きの spin-wait + sleep-wait する実装が実用的であるが，まず条件変数を用い sleep-wait する場合を示す．局所同期のタグのデータ型 `I_st` のとして presence bit と排他制御のための排他ロック，さらに書き込み，読み込みでブロックした時のための条件変数が必要となる．

```
typedef struct {
    volatile int tag;
    pthread_mutex_t mt;
    pthread_cond_t full, empty;
} I_st;
```

この時 `i_write_int()`, `i_read_int()` は以下のように実装することができる．

```
void i_write_int(I_st* i_st, int* addr,
                 int val)
{
    pthread_mutex_lock(&i_st->mt);
    while (i_st->tag != EMPTY)
```

☆ Solaris threads の read/write lock は名前は似ているが，このロックは 1-write N-read の排他ロックであり，書き込みと読み込みの順序はつけることができない点で異る．

```
        pthread_cond_wait(&i_st->empty,
                          &i_st->mt);
    *addr = val;
    i_st->tag = FULL;
    pthread_cond_signal(&i_st->full);
    pthread_mutex_unlock(&i_st->mt);
}

int i_read_int(I_st* i_st, int* addr)
{
    int val;

    pthread_mutex_lock(&i_st->mt);
    while (i_st->tag != FULL)
        pthread_cond_wait(&i_st->full,
                          &i_st->mt);
    val = *addr;
    i_st->tag = EMPTY;
    pthread_cond_signal(&i_st->empty);
    pthread_mutex_unlock(&i_st->mt);

    return val;
}
```

また, i_write_lock(), i_write_unlock() の実装にはi_write_int() のそれぞれ前半部分と後半部分として実装することができるが, この場合, i_write_lock(), i_write_unlock() とi_read_lock(), i_read_unlock() を必ずペアで用いる必要がある. しかしながら, 書き込みを一度しか行わない場合, あるいは必ず書き込みをしてもいいと分かっている場合は, 書き込み時に i_write_lock() を発行するのは無駄であり, i_write_lock(), i_read_unlock() を省略し, 書き込み終了時にi_write_unlock() を発行し, 読み込み時に i_read_lock() を発行すれば良い. このような使い方のためには, 以下のように presence bit として FULL/EMPTY の他に変更中を示す UPDATING という状態を持たせ, かつ排他ロックをアンロックする必要がある.

```
void i_write_lock(I_st* i_st) {
    pthread_mutex_lock(&i_st->mt);
    while (i_st->tag != EMPTY) {
        pthread_cond_wait(&i_st->empty,
                          &i_st->mt);
    }
    i_st->tag = UPDATING;
    pthread_mutex_unlock(&i_st->mt);
}

void i_write_unlock(I_st* i_st) {
    pthread_mutex_lock(&i_st->mt);
    i_st->tag = FULL;
    pthread_mutex_unlock(&i_st->mt);
    pthread_cond_signal(&i_st->full);
}
```

次に spin-wait だけでブロックを行う場合であるが, この時 I_st のデータ型として presence bit と排他制御のための排他ロックだけが必要となり, 条件変数は必要なくなる. spin-wait に関して Pthreads では Solaris threads におけるthr_yield() に相当する命令を用意していないため, 以下のようにspin-wait することになる.

```
void i_write_lock(I_st* i_st) {
    pthread_mutex_lock(&i_st->mt);
    while (i_st->tag != EMPTY) {
        pthread_mutex_unlock(&i_st->mt);
        /* pthread_yield() */
        pthread_mutex_lock(&i_st->mt);
    }
    i_st->tag = UPDATING;
    pthread_mutex_unlock(&i_st->mt);
}
```

期限付きの spin-wait + sleep-wait の場合はこれらを組み合わせたものになる.

3.2 不可分命令による実装

compare-and-swap 等の不可分命令を用いることにより, より効率的な実装を行うことができる. SGI の C コンパイラは不可分命令を intrinsic として用意しており, compare-and-swap に関しては以下のようになっている.

```
int __compare_and_swap(<type>* ptr,
                       <type> old, <type> new, ...)
```

この命令は ptr の内容が old と等しい場合はその内容とnew を入れ換え, 等しくない場合は何もしない. 不可分命令を用いる場合, I_st のデータ構造は実質的には FULL/EMPTY/UPDATING を表す 2 ビットで十分であるが, 整数型で実装を行う. この時, i_write_lock(), i_write_unlock() は以下のように実装を行うことができる.

```
void i_write_lock(volatile int *i_st) {
    while(!__compare_and_swap(i_st,
                              EMPTY, UPDATING))
        while(*i_st != EMPTY)
            /**  spin-wait  **/;
}

void i_write_unlock(int *i_st)
{   *i_st = FULL;   }
```

__compare_and_swap() は LL/SC を用いて実装されるため, spin-wait の部分ではネットワークの負荷をあげないようにローカルにアクセスしている. これらの実装は, 必ず書き込み可で i_write_lock(), i_read_unlock() が不要な場合それらを省略すること

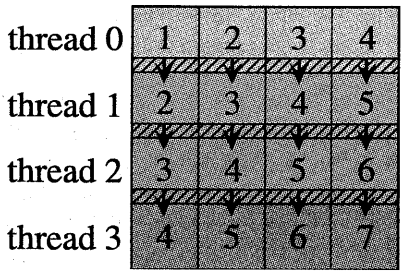

図1 SOR 法の実行の流れと I-Structures

ができる.

また, SPARC Architecture V9 では不可分命令の**compare-and-swap** 命令 **cas** を持つため, UltraSPARC I/II を搭載したマルチプロセッサでは **cas** を用い実装を行うことができる. 以下, **compare-and-swap** 命令を用いた局所同期の実装を CAS と呼ぶ.

3.3 不可分命令を用いない実装

同じ I-Structure に対し二つ以上のスレッドが書き込みを行ったり, 読み込みを行い, 競合状態になることがないプログラムの場合, presence bit の変更に関して不可分命令, あるいは排他ロックを用いる必要がない. このプログラムはメッセージパッシングに例えると, 送信元プロセス, メッセージタグに ANY を用いないプログラムに相当する. 従って, 多くのプログラムにおいて, 不可分命令を用いない局所同期命令を用いることができる. 以下, この不可分命令を用いない実装を NCAS と呼ぶ.

4. プログラミング例

この節では **i_write_lock()** 等の局所同期機構を用いたプログラミングについて解説を行う.

4.1 SOR 法

パイプライン並列処理の例として自然順序の SOR 法のプログラムについて考察を行う.

```
for(i = 1; i < N - 1; ++i)
  for(j = 1; j < N - 1; ++j)
    a[i][j] = .25 * (a[i - 1][j]
            + a[i][j - 1] + a[i][j + 1]
                  + a[i + 1][j]);
```

このループでは各反復において a[i - 1][j], a[i][j - 1] の参照にデータ依存がある. 2 次元配列 a に対し, 図1の様に行ブロックサイクリックに計算分割を行い, パイプライン処理で並列化を行う.

まず SOR 法のプログラムを図1の様にブロック化を行う. そして, 図1の各ブロックの斜線の部分の操作に対し局所同期を用い, マルチスレッドによる並列化を行う. ブロックサイズを **BLK_X** × **BLK_Y** スレッド番号を **tid**, 総スレッド数を **nthds** とすると, 図2となり, point-to-point 通信における送信, 受信がそれぞれ **i_write_lock()**, **i_write_unlock()**

```
for (t_i = tid * BLK_Y + 1, ist_i = tid;
     t_i < N - BLK_Y;
     t_i += BLK_Y * nthds, ist_i += nthds)
  for (t_j = 1, ist_j = 0;
       t_j < N - BLK_X;
       t_j += BLK_X, ++ist_j) {
    i = t_i;
    if (t_i > 1)
      i_read_lock(&i_st[ist_i-1][ist_j]);
    if (i < t_i + BLK_Y)
      for(j = t_j; j < t_j + BLK_X; ++j)
        a[i][j] = .25 * (a[i - 1][j]
                + a[i][j - 1] + a[i][j + 1]
                      + a[i + 1][j]);
    if (t_i > 1)
      i_read_unlock(&i_st[ist_i-1][ist_j]);
    for(++i; i < t_i + BLK_Y - 1; ++i)
      for(j = t_j; j < t_j + BLK_X; ++j)
        a[i][j] = .25 * (a[i - 1][j]
                + a[i][j - 1] + a[i][j + 1]
                      + a[i + 1][j]);
    if (i < N - 2)
      i_write_lock(&i_st[ist_i][ist_j]);
    if (i < t_i + BLK_Y)
      for(j = t_j; j < t_j + BLK_X; ++j)
        a[i][j] = .25 * (a[i - 1][j]
                + a[i][j - 1] + a[i][j + 1]
                      + a[i + 1][j]);
    if (i < N - 2)
      i_write_unlock(&i_st[ist_i][ist_j]);
  }
```

図2 局所同期を用いた SOR 法のプログラム

と**i_read_lock()**, **i_read_unlock()** になっていることが分かる. またメッセージパッシングとの大きな違いは共有メモリなのでローカルアドレスに直す必要がないということであり, ブロック化を行いそれぞれのブロックの計算をマルチスレッドで行い, 同期が必要な部分に局所同期命令を入れるだけである.

また, バリア同期を用いる場合は局所同期の時と違い, 計算ブロックについて局所同期をとる部分ととらない部分を分ける必要がないため, プログラム上は簡単になる.

SOR 法は通常収束するまで, あるいは規定回数反復を行う. 局所同期を用いて書かれたプログラムの場合は, 図2のプログラムをそのまま反復させるだけで, パイプライン処理の反復毎のプロローグとエピローグがつながり, 並列性が下がらない. バリア同期を用いる場合は, 図3の前後のバリア同期を SOR 法全体の反復の外に出すことで同様の最適化を行うことができる.

```
for (i = 0; i < tid; ++i)
    barrier(tid, nthds);

for (t_i = tid * BLK_Y + 1;
     t_i < N - BLK_Y;
     t_i += BLK_Y * nthds)
  for (t_j = 1; t_j < N - BLK_X;
       t_j += BLK_X)
    for(i = t_i; i < t_i + BLK_Y; ++i) {
      for(j = t_j; j < t_j + BLK_X; ++j)
        a[i][j] = .25 * (a[i - 1][j]
                + a[i][j - 1] + a[i][j + 1]
                        + a[i + 1][j]);
      barrier(tid, nthds);
    }

for (i = tid + 1; i < nthds; ++i)
    barrier(tid, nthds);
```

図3 バリア同期を用いた SOR 法のプログラム

表1 局所同期の遅延

	NCAS	CAS	sleep-wait	spin-wait
Enterprise	1.95	1.62	51.82	5.91
Origin2000	1.75	2.35	84.44	4765

(usec.)

5. 性能評価

局所同期とバリア同期の基本性能と SOR 法を用いパイプライン処理の並列実行の関する性能評価を行う.

5.1 局所同期とバリア同期

メッセージパッシングのピンポンベンチマークと同様の方法を用い, 局所同期の遅延の計測を行う. 局所同期のタグを二つ用い, `i_write_lock()`, `i_write_unlock()` を送信として, `i_read_lock()`, `i_read_unlock()` を受信として用い, 二つのスレッドでそれらを交互に実行する. 表1にその結果を示す. 用いた計算機は Ultra Enterprise 3000 と Origin2000 であり, Enterprise は 168 MHz UltraSPARC I の 6 プロセッサ構成, Origin2000 は 195 MHz R10000 の 16 プロセッサ構成である.

表1の中で, NCAS は不可分命令を用いない実装, CAS は`compare-and-swap` を用いた実装, sleep-wait は Pthreads を用いブロックした場合条件変数で待つ実装, spin-wait は 同じく Pthreads を用い条件変数を用いず spin-wait して待つ実装である.

共有メモリ計算機におけるバリア同期に関して Mellor-Crummey[4] によると, キャッシュコヒーレンシの方式等共有メモリ機構のアーキテクチャにより効率的なバリア同期の方式は変わり, ブロードキャストベースのキャッシュコヒーレントマルチプロセッサの場合は, 集中カウンタ方式, あるいは 4 分木と集中極性フラグを用いた方式が良く, コヒーレントキャッシュを持たない, あるいはディレクトリベースのコヒーレントキャッシュを持つマルチプロセッサの場合は, dissemination バリア[2] あるいは 4 分木と集中極性フラグを用いた方式が良いと報告されている. が, 評価に用いた N-write 1-read 極性バリア[3] は, サイズ N のバッファに対し衝突なしに極性フラグの書き込みを行い, 代表スレッドが線形にバリア成立を調べ, 集中極性フラグによりバリアの成立を知らせる方式である.

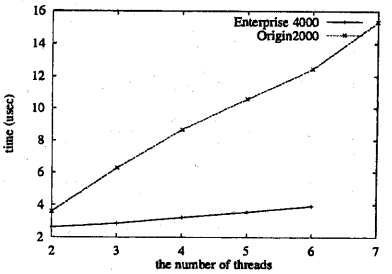

図4 N-write 1-read 極性バリアの遅延

N-write 1-read 極性バリアの性能を図4に示す. メモリに対する同期命令を発行しないため, 非常に高速なバリア同期を実現しているのが分かる.

5.2 SOR 法

4.1節で説明した SOR 法の 1000 反復を Enterprise 3000, Origin2000 で実行させた結果を表 2, 3に示す. 問題の大きさは表 2は $N = 80 \times 80$, ブロックサイズは 20×20 であり, 表3は $N = 320 \times 320$, ブロックサイズは 80×80 である. この時, いずれの場合もスレッド数は 4 となる. 表中の効率 (1) は逐次の 4 倍の性能に対する割合であり, 効率 (2) は問題サイズがブロックサイズと同じ問題の性能の 4 倍に対する割合である. 効率 (2) は各スレッドに対するキャッシュサイズを揃えた評価であり, スレッドの数が多くなりキャッシュサイズの増大によるキャッシュヒット率の上昇の効果を除いた評価となっている.

いずれの場合も NCAS, CAS による局所同期機構の実装を用いたものがもっとも良い性能を出している. バリア同期はそれらについで良い. Pthreads による実装ではブロックの粒度が小さい場合はそれらの遅延が現れてしまうが, 粒度を大きくするとそれらの差は縮まり, 十分な性能を出すことができるといえる.

Origin2000 は表 2では並列効率にして半分も出ていないが, 表3の様に問題を大きくし, ブロックの粒度を大きくすると 8 割以上の並列効率となり, もっと粒度を大きくすることでさらに性能が上がることが期待される. また, 条件変数を用いた Pthreads による実装もほどほどの性能となっている.

しかしながら, これらの評価は 4 スレッドという低い

表2 SOR 法の性能評価 (1) (N = 80 x 80)

		逐次	$N = 20$	NCAS	CAS	barrier	sleep-wait	spin-wait
Enterprise	time (msec)	562.1	29.98	146.3	146.2	156.8	236.2	179.0
	MFLOPS	45.54	53.38	175.0	175.1	163.3	108.4	143.0
	効率(1)	1.0	-	.96	.96	.90	.60	.79
	効率(2)	-	1.0	.82	.82	.76	.51	.67
Origin	time (msec)	160.4	7.928	85.67	93.60	96.89	280.3	11288
	MFLOPS	159.6	201.8	298.8	273.5	264.2	91.33	2.268
	効率(1)	1.0	-	.46	.43	.41	.14	.00
	効率(2)	-	1.0	.37	.34	.32	.11	.00

表3 SOR 法の性能評価 (2) (N = 320 x 320)

		逐次	$N = 80$	NCAS	CAS	barrier	sleep-wait	spin-wait
Enterprise	time (msec)	11345	562.1	2338	2356	2361	2540	2390
	MFLOPS	36.10	45.54	175.2	173.9	173.5	161.3	171.4
	効率(1)	1.0	-	1.21	1.20	1.20	1.12	1.18
	効率(2)	-	1.0	.96	.95	.95	.89	.94
Origin	time (msec)	2421	160.4	759.8	772.8	798.0	990.0	13851
	MFLOPS	169.2	159.6	539.1	530.0	513.3	413.7	29.57
	効率(1)	1.0	-	.80	.78	.76	.61	.04
	効率(2)	-	1.0	.84	.83	.80	.65	.05

並列度で行ったものであり, この並列度でも局所同期機構はバリア同期よりも性能が良くなっている. バリア同期のコストは並列度に対し $O(N)$ あるいは $O(\log N)$ で増えるのに対し, 局所同期は (理想的には) 増えないため, 大規模なシステムに対して局所同期を用いた実行方式は極めて有望であるといえる.

6. まとめと今後の課題

共有メモリ機構を持つ並列計算機において局所同期に関するプリミティブを提案し, Pthreads の排他ロック, 条件変数を用いた実装, またcompare-and-swap 命令を用いた実装, 不可分命令を用いない実装を行った. さらにそれらの局所同期機構を用いたパイプライン並列処理に関するプログラミング手法を示し, 局所同期機構の基本性能を含む性能評価を Sun Ultra Enterprise 3000, SGI/Cray Origin2000 において行った. その結果, 局所同期にかかる時間は不可分命令を用いることにより 1 ～ 2 マイクロ秒であり, また不可分命令を用いる必要がない場合はさらに速くなり, 非常に軽く実装できることが分かった.

共有メモリ計算機において局所同期機構を用いることにより, メッセージパッシングで提供している局所同期機構を容易に実現することができ, また分散メモリのローカルアドレスを仮定しているメッセージパッシングに比べローカルアドレスに変換する必要がないため, プログラミングが容易となる.

パイプライン並列処理として SOR 法を局所同期機構を用い評価した結果, Enterprise においては 82 %, 96 % の並列効率を達成した. また, 4 スレッドという低い並列度にも関わらず, バリア同期を用いる実行方式より性能が上であった. また, Pthreads の排他ロック, 条件変数を用いても, 局所同期の粒度を大きくすることにより十分性能をあげることができることが分かった.

局所同期を用いた実行方式は, バリア同期を用いるものに比べより柔軟な実行方式であり, また大規模な並列計算機において効率的に実行できる可能性を持っている. 今後, より大規模な共有メモリ計算機による評価, C++ による実装, これらの実行方式を OpenMP 等のコンパイラに組み込むことなどを計画している.

謝　辞

本研究を遂行するにあたり貴重なご助言, ご討論いただいた大蒔和仁部長, 電総研 HPC グループ, 新情報処理開発機構, 筑波大, 電総研 TEA グループ, 電総研並列アーキテクチャラボのメンバ諸氏に感謝致します.

参 考 文 献

1) Arvind, R., Nikhil, S. and Pingali, K. K.: I-structures: Data structures for Parallel Computing, *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 4, pp. 598–632 (1989).
2) Hensgen, D., Finkel, R. and Manber, U.: Two algorithms for barrier synchronization, *Int. J. Parallel Program*, Vol.17, No.1, pp.1–17 (1988).
3) Matsuda, M.: Personal communication.
4) Mellor-Crummey, J. M. and Scott, M. L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprosessors, *ACM Trans. on Computer Systems*, Vol. 9, No. 1, pp. 21–65 (1991).