共有メモリ向けプリミティブとそのGCCを使った実現

八杉 昌宏[†,††] 高田 潤[†] 田畑 悠介[†]
小宮 常康[†] 湯淺 太一[†]

並列計算機は共有メモリ型だけを考えてもプロセッサやメモリモデルなどの点で様々なアーキテクチャがある．アーキテクチャの違いを吸収して信頼性・再利用性・実行効率の高いソフトウェアを開発するには，並列処理のための高水準プログラミング言語が有用である．高水準言語コンパイラでは，直接アセンブリコードを生成する代わりに，C 言語を実装用言語として利用することでプロセッサに依存するコード生成を C コンパイラに担当させることができる．しかしながら，C 言語では，共有メモリに関する不可分操作やメモリアクセス完了順序を直接記述することはできず，ライブラリや asm 文などを利用する必要があり，移植性や実行効率を低下させる原因となっていた．そこで移植性や実行効率を高めるため，C 言語の拡張による共有メモリ向けプリミティブを設計している．一方，プリミティブの普及を図るには既存のシステムでもできるだけ利用可能であることが望ましい．そこでプリミティブを少し変更し，GCC の拡張機能による実現を行った．

Primitives for Shared Memory and its Implementation with GCC

Masahiro Yasugi,[†,††] Jun Takada,[†] Yusuke Tabata,[†]
Tsuneyasu Komiya[†] and Taiichi Yuasa[†]

There are various architectures for shared-memory parallel computers in terms of processors and memory models. High-level programming languages for parallel processing are quite useful to develop reliable, reusable and efficient applications on various parallel computers by concealing their architectural difference. Compliers for high-level languages may directly generate assembly code, but they are implemented more easily by employing C language as an implementation language and using C compilers to generate processor-dependent code. In C, however, we cannot directly describe atomic operations and memory orders for the shared memory; we have to use library routines or `asm` statements, resulting in poor portability and lower performance. We designed an extended language to C with primitives for shared memory to obtain better portability and performance. On the other hand, in order to promote those primitives, it is desirable that they are available on the conventional system. Thus, we implement the slightly modified version of the primitives with GCC's extended functionality.

1. はじめに

計算機の高性能化のために並列計算機は魅力的である．並列アーキテクチャは，共有メモリ型アーキテクチャ，分散メモリ型アーキテクチャなど様々であり，その違いを吸収しつつ信頼性・再利用性・実行効率の高いソフトウェアを開発するには，並列処理のための高水準プログラミング言語が必要となる．

高水準言語コンパイラの実装には言語階層を用いると便利である．特に C 言語を経由することで，低レベルの最適化やレジスタ割り付け，アセンブリコード生成などの処理を C コンパイラに担当させることができる．このように C 言語を実装用言語として用いることで，処理系の共通に利用できる部分を増やし，処理系実装の労力を減らすことができる．

しかしながら，C 言語は，本来逐次処理用の言語であるため，並列処理のための実装用言語として用いるには記述性・実行効率ともに問題がある．C 言語の仕様/処理系では他のプロセッサというものの存在が基本的にあまり考慮されていないため，自プロセッサでの決められた処理を集中して進めるのは得意であるが，他のプロセッサとの協調はあまり得意ではない．他のプロセッサとの協調のためには，プロセッサ間での通信・同期の機能，他のプロセッサからの非同期の

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
†† 科学技術振興事業団さきがけ研究 21「情報と知」領域グループ
"Information and Human Activity", PRESTO, Japan Science and Technology Corporation(JST)

依頼に応える機能，自プロセッサでの処理の内容や順序を固定化せず他のプロセッサに合わせて調整する機能が求められる．そのような機能のために，通常は通信ライブラリやスレッドライブラリなどの並列ライブラリを用いる必要がある．通信ライブラリは，分散メモリ環境における通信・同期などに用いられるが，高速な並列アーキテクチャを生かしきれないという問題がある．また，スレッドライブラリは，同期の機能や，自プロセッサで進められる処理の流れ（スレッド）を複数持ち，他のプロセッサに合わせて処理順序を変える機能などを持つが（データの通信は共有メモリを利用），同期の度にライブラリ関数を呼び出すオーバヘッドや，スレッドを生成するオーバヘッドが大きく細粒度スレッドの実装には適さない．

そこで，本研究では，並列処理に適した実装用言語として並行協調拡張 C 言語 XC-cube を設計・開発している．また，どのような設計が望ましいか，またどのような記述性や性能の向上が可能となるのかを研究している．現在の基本設計方針は以下のようになっている：各プロセッサのためのコードを記述するための言語として用いる．C 言語と同じく，汎用言語，低水準言語として柔軟性を保ちつつ，並列処理に適するように拡張し，アセンブリコードを直接生成しなくては実装が困難だった高水準言語のための実装用言語としても利用できるようにする．そのため並列アーキテクチャの通信・同期機能を直接サポートする．また，他のプロセッサからの要求と自プロセッサの処理の多重実行に必要なマルチコンテキスト管理[12] 等を実現するための機能をサポートする．

本論文ではこれらのうち，並列アーキテクチャ，特に共有メモリ型並列計算機のアーキテクチャのサポートについて論ずる．C 言語では，共有メモリに関する不可分操作やメモリアクセス完了順序を直接記述することはできず，ライブラリや asm 文などを利用する必要があり，実行効率の低下や，異なるアーキテクチャ間の移植性の低下の原因となっていた．そこで移植性や実行効率を高めるため，拡張 C 言語 XC-cube の共有メモリ向けプリミティブを設計している．一方，プリミティブの普及を図るには既存のシステムでもできるだけ利用可能であることが望ましい．そこでプリミティブを少し変更し，GCC（GNU C コンパイラ）の拡張機能による実現を行った．

2. 共有メモリに関する問題

共有メモリ上の実行においては，他のプロセッサとの間で望ましくない干渉や行き違いが生じないようにしたい．この章では，共有メモリ上の実行に関する問題として，ロック，不可分操作，粒度，メモリモデルを取り上げ，Alpha[2]，MIPS[4),6]，Pentium[5]，PowerPC[8]，SPARC-V9[10]，SPARC-V8[9] といったプロセッサアーキテクチャ/マルチプロセッサシステムの場合について述べる．この章では，プロセッサによる機械語プログラムの実行と共有メモリの関係に関して議論し，C 言語のプログラムから機械語プログラムへと変換する際の問題は次章以降で述べる．投機的ロード，フォールト無しロードについては，直接記述するものというよりはプロセッサやコンパイラが高速化・最適化に用いるものと考えて本論文ではこれ以上扱わない．プリフェッチについては，基本的にはヒントであり，直接記述できるように言語を拡張することに特に問題はないと考えられる．プリフェッチについても本論文ではこれ以上扱わない．

2.1 ロック

ロックを相互排他に用いる場合，同時に一つのプロセッサだけがロックを獲得することができ，ロックが解放されるまで他のプロセッサによるロックの獲得は成功しないようにする．このようなロックは，メモリ上で共有されている構造体へのアクセスをクリティカルセクション内で他のプロセッサと競合せずに（排他的に/不可分に）行いたい場合などによく用いられる．（ここで，他のプロセッサはロックを獲得するまでクリティカルセクションに進まないという約束を守る必要がある．）

このようなロックは最低限備えるべき機構と考えれ，当然，共有メモリ型並列計算機に用いられるプロセッサには，このようなロックを実現するための機械命令が備えられている．いわゆるテスト&セット（test & set）命令に相当する命令である．Alpha，MIPS，Pentium，PowerPC，SPARC-V9，SPARC-V8 のプロセッサアーキテクチャのうち，SPARC-V8 を除いては，2.2 節で述べる不可分操作のための命令を使ってロックが実現できる．SPARC-V8 では，`ldstub`（atomic load-store unsined byte）命令か `swap` 命令を用いる[9]．`ldstub` 命令では，共有メモリ上から 1 バイトのデータをロードし，同時にそのバイトのビットをすべて 1 に書き換える．ロードとストアは不可分に実行されるため，`ldstub` 命令で 0 がロードされたときにロックの獲得に成功したとし，それ以外の場合は再試行することで（スピンロックの場合）ロックの獲得が実現できる．また，ロックの解放は単に該当するバイトに 0 をストアすればよい．実際には，2.4 節で述べるようにロックの獲得/解放と共有されている構

```
atomic_add(*s, v) =
 Lretry: load     [s],d
         add      d,v,n
         cas      [s],d,n
         br_fail  Lretry
;; = do{ d = *s ; n = d + v;
;;   }while( cas(*s, d, n) == fail );
```

図 1 cas 命令による不可分加算（疑似コード）
Fig. 1 atomic addition by cas instruction.

```
cas(*s, o, n) =
 Lretry: ll       [s],d
         br_ne    d,o,Lfail
         sc       [s],n
         br_fail  Lfail
 Lsucess:
          ...
 Lfail:
          ...
;; = { d = ll(*s);
;;     if(d != o) return fail;
;;     else return sc(*s, n); }
```

図 2 ll 命令，sc 命令による cas 命令の実現（疑似コード）
Fig. 2 implementation of cas instruction using ll and sc instructions.

```
atomic_add(*s, v) =
;;; = do{ d = *s ; n = d + v;
;;;   }while( cas(s, d, n) == fail );
 Lretry: load     [s],d
         add      d,v,n
         ll       [s],d2
         br_ne    d2,d,Lretry
         sc       [s],n
         br_fail  Lretry
;;; = do{ d = *s ; n = d + v;
;;;       d2 = ll(*s);
;;;       if(d2 != d) continue;
;;;   }while( sc(*s, n) == fail );
```

図 3 ll 命令，sc 命令で cas 命令としたときの不可分加算
Fig. 3 atomic addition by ll and sc instructions instead of cas instruction.

```
atomic_add(*s, v) =
 Lretry: ll       [s],d
         add      d,v,n
         sc       [s],n
         br_fail  Lretry
;; = do{ d = ll(*s); n = d + v;
;;   }while( sc(*s, n) == fail );
```

図 4 ll 命令，sc 命令による不可分加算（疑似コード）
Fig. 4 atomic addition by ll and sc instructions.

造体へのアクセスとの間のメモリアクセス完了順序についてさらに考慮する必要がある．

2.2 不可分操作

プロセッサ間で一つの変数を共有し，変数の値を不可分に更新したいという場合がある．例えば，カウンタ変数が保持する 4 バイトの整数値に，ある値を不可分に加算したいというような場合である．このような不可分操作のための機械命令としては，Pentium のように lock 命令プリフィックスを使ったものなどがある．Pentium の場合は，BTS（bit test and set）命令，INC 命令，ADD 命令などに lock プリフィックスを付けることで，メモリ上の値を不可分に更新できる．

加算以外の一般的な演算も可能とするため，演算を自由に記述したいという場合には，いわゆる比較交換（compare & swap）命令（以下 cas 命令と表記）を用いるか，リンク付きロード（load linked「linked」は，「locked」「reserve」などのこともある）命令（以下 ll 命令と表記）と条件付きストア（store conditional）命令（以下 sc 命令と表記）のペアを用いることができるとよい．

対象となる共有変数，変数の元の値，新しい値が与えられると，cas 命令は，対象となる共有変数の値が変数の元の値と比較して等しいときにのみ新しい値で共有変数の値を置き換える．比較して異なっていた場合は置き換えを行なわない（あるいは最新の値に戻す）. Pentium や SPARC-V9 の実際の cas 命令では，比較して異なっていた場合にも最新の変数の値がロードされるが，以下ではこれを無視し，単に比較の結果が分かるようなっていると考える．cas 命令を用いれば，図 **1** のように，元の値にある値を加えたものを新しい値として用いて，比較して異なっていた場合には再試行するようにすることでカウンタ変数への不可分加算が実現できる．また，図 1 において加算を行っている部分に自由に計算式/手順を記述することで，一般的な演算も可能となる（例えば，「n = (d + v) % N」など）

また，ll 命令は，共有変数の値をロードしたことを覚えておく．後に新しい値で共有変数の値を置き換えるために sc 命令を実行すると，元の値をロードしてからそれまでに共有変数に対して他のプロセッサからのストアなどがなかった場合は共有変数の値を置き換えるが，途中に邪魔が入っていたときには共有変数の値を置き換えないようにする．また，置き換えが成功したかどうかが分かるようになっている．ll/sc 命令を利用すれば，図 **2** のように cas 命令を合成することができる．この合成を利用すれば，カウンタ変数へ不可分加算は図 **3** のようにできる．さらに，合成した cas 命令を使用してカウンタ変数へ不可分加算をしなくても，図 **4** のように，ll 命令と sc 命令の間で新しい値を計算することで，より少ない命令数で不可分加算が実現できる．

cas 命令は SPARC-V9，Pentium で使用され，SPARC-V9 は 4 バイト（cas），8 バイト（casx）の値について，Pentium は 1 バイト，2 バイト，4 バイ

ト（CMPXCHG），8 バイト（CMPXCHG8B）の値についての cas 命令を持つ．

ll/sc 命令は，MIPS(R4000 以降)，Alpha，PowerPC で使用され，MIPS は 4 バイト（LL/SC），8 バイト（LLD/SCD，ただし 64bit mode 時のみ有効）の値について，Alpha では 4 バイト（LDL_L/STL_C），8 バイト（LDQ_L/STQ_C）の値について，32-bit の PowerPC は 4 バイト（lwarx/stwcx.）の値について ll/sc 命令を持つ．

なお，SPARC-V8 には，cas 命令や ll/sc 命令に相当する命令は存在しない．SPARC-V8 でカウンタ変数へ不可分加算を実現するためには，ロックによる相互排他を用いる必要がある．その場合，カウンタ変数とは別にロック用の変数を準備する必要がある．このため，移植性を考えるうえで，SPARC-V8 を除外して考えるべきかどうかという問題が残る．

また，ここでの cas 命令は，単一の共有変数のみ対象としたが，二つの共有変数を対象とした dcas（double compare & swap）命令があれば，いわゆる lock-free（non-blocking）/wait-free な同期データ構造が実際的に使われるようになると考えられる[3]．このような dcas 命令は現在は Motorola 68040 でしか提供していないが，将来のプロセッサアーキテクチャで提供される可能性を考えると，移植性を考える上で問題となる．

2.3 粒　　度

通常のロード/ストア命令や，cas 命令，ll/sc 命令が一括して扱うデータの粒度には，プロセッサによる違いが大きい．逐次処理では大きなデータは複数のロード/ストア命令などを用いて実現することに問題はないが，共有メモリに関してロード/ストア命令を分けてしまうと（例えば，64 ビット変数にアクセスするために 32 ビットのアクセスを二回行う場合など），他のプロセッサとの間で望ましくない干渉が発生する．粒度を保証するには適切な命令を用いてアクセスを行うように注意する必要がある．

例えば，32 ビットの PowerPC では，整数は最大 4 バイトの大きさで扱われ，8 バイトのサイズの整数に関して粒度を保証できない．このため，8 バイトの実数のための cas の機能を実現したい場合，実数値を整数レジスタに移して整数レジスタを使って cas を実現するというアプローチをとることはできない．

Pentium では，整数レジスタを使ったロード／ストア命令は通常 4 バイトまでであり，8 バイトの整数値についての粒度を保証したロード／ストアを行うことはできない．これを 4 バイトのアクセスを 2 回行ったのでは粒度が保証できない．ただし，後述するように CMPXCHG8B 命令を利用して 8 バイトのメモリアクセスの粒度を保証することは可能である☆．

SPARC-V8 では，整数レジスタ対上の 8 バイトのデータに関する ldd 命令や sdd 命令で，8 バイトのロード／ストアの粒度は保証されるが，ld 命令を 2 回用いたのでは粒度が保証できない．また，swap 命令などは 4 バイトしか粒度保証されない．

Alpha[2] には，1 バイトや 2 バイトを単位としてロード/ストアするための命令は準備されていない☆☆．4 バイトや 8 バイトを単位とする命令は準備されている．このため，1 バイトのデータをメモリに書き込む際には，そのバイトを含む大きな単位のロードを行ない，該当するバイトのみ変更した後，もとのメモリ位置にストアすることになる．これは逐次処理では問題ないが，共有メモリの場合は，ロードからストアまでを不可分にしないと同じメモリ位置に複数のプロセッサが 1 バイトのデータを書き込もうとした場合にその効果が失われてしまう可能性がある．

また，粒度の問題は，境界整列（アラインメント）の話が出てくるとさらに複雑になる．境界整列については，整列してない場合に，粒度保証がなくなるだけで整列違反を起こさないもの，整列違反でフォールトするものやトラップハンドラで対処するものなどがある．境界整列の問題は逐次処理でも存在しており粒度の問題と異なりマルチプロセッサ固有の問題ではない．ただし，機械語プログラムの上では粒度が通常保証されている命令を使っていたとしても，整列違反の際にはトラップハンドラが 2 回に分けてアクセスするというような場合には，粒度保証はないものとして扱う必要がある．

2.4 メモリモデル

制御の流れにそって，機械語プログラム上の命令の論理的な実行順序（発行順序）は定まるが，必ずしもその順序通り，プロセッサによって命令が実行されるとは限らない．依存関係の制約を満たして命令を一つずつ順に実行したのと同じ結果となる限り，実際のプロセッサではレジスタの renaming なども利用して命令の同時実行や reordering が行われる．メモリアクセスについても，プロセッサ自身にとって一貫性が保たれてさえいれば，メモリアクセスがメモリによってどの順序で処理されるのかは自由にできるようにする

☆ 他にも MMX の命令を用いることも考えられるがプロセッサが直接保持するレジスタなどのコンテキストが多くなってしまうという問題がある．

☆☆ 比較的新しいものには 1 バイトや 2 バイトデータのロード/ストア命令が追加されている．

```
      P1                    P2
{                       {
  *p  = 10;               while(*f == 0);
  *q  = 20;               (membar;)
  (membar;)               r1 = *p;
  *f  = 1;                r2 = *q;
}                       }
```

図 5 2 プロセッサ間での通信・同期 (疑似コード)
Fig. 5 communication/synchronization between 2 porcessors.

ことで，性能を向上できると考えられている．

最近のプロセッサに関するメモリモデルは文献 1) などによくまとまっている．

メモリモデルには，最も強い sequential consistency (プログラム上で決まる命令の順序の通りにメモリアクセスが処理されなくてはならない)，プロセッサ自身にとって一貫しているならロードがストアを追い越してもよいという SPARC-V8[9] の TSO (total store ordering)，さらにメモリ上でストアが処理される順序が変ってもよい SPARC-V8 の PSO (partial store ordering) などがある．PSO ではストアバリア (`stbar`) 命令によりストアの順序を保つことができる．Intel P6 ファミリ (Pentium PRO 以降) の場合は，プロセッサによる投機的ロードが許されているものの，TSO と同等の processor ordering が用いられている[5]．TSO や P6 ファミリのメモリモデルは processor consistency と呼ばれる．これらでは，ロードがストアを追い越すのが許されるため，ストアバッファにストア要求を溜めておくことができ，その間にストア先と同じアドレスに関するロードがあれば，バッファ内のストア要求の持つデータを返すというような実装が可能となる．ここで，機械語プログラム上で，ロードがストアを追い越すのを防いで順序を保つには，SPARC-V8 では `swap` 命令，P6 ファミリでは `lock` プリフィックス付命令を使ったり，暗黙の `lock` プリフィックスを持つ `XCHG` 命令を副作用が生じないように注意しつつ用いる必要がある．これには，メモリとレジスタの値を交換する `swap` 命令などがロードとストアの両方の意味を持つことを利用している．

また，図 5 のように 2 プロセッサ間で 2 ワードのデータ (10, 20) をプロセッサ P1 からプロセッサ P2 へと伝えたい場合，processor consistency では，`(membar;)` としたところに何も指定しなくてよい．PSO ではプロセッサ P1 の `(membar;)` の部分にストアバリア (`stbar`) 命令を用いなくてはならない．

また，Alpha，MIPS，PowerPC では，特殊な命令によりメモリ順序を指定する weak ordering が使われている．Alpha にはメモリバリア (`MB`) 命令，MIPS，PowerPC では `SYNC` 命令などで，その命令の前のメモリアクセスの完了後に，その命令の後のメモリアクセスが完了することを制約として与えることができる．制約のないメモリアクセスの完了順序は自由である．これらのメモリモデルで図 5 のようなデータの授受を実現するには，プロセッサ P1 とプロセッサ P2 の両方の `(membar;)` の部分にメモリバリア (`MB/SYNC`) 命令を用いなくてはならない．ただし，MIPS については，実際の MIPS R10000 プロセッサなどでは sequential consistency が使われている．

Weak ordering の一種，あるいは拡張といえるのが，SPARC-V9 の RMO(relaxed memory ordering) で，順序を保存したい場合にはメモリバリア命令を用いるが，ストア–ストア間，ロード–ロード間のみ順序を保つなどの細かい指定が可能である．RMO モデルで図 5 のようなデータの授受を実現するには，プロセッサ P1 の `(membar;)` の部分にストア–ストア間のメモリバリア命令，プロセッサ P2 の `(membar;)` の部分にロード–ロード間のメモリバリア命令を用いることになる．SPARC-V8 の TSO，PSO では，ストア–ロード間の順序が保たれないため，いわゆる Dekker のアルゴリズムで，ストアの代りかストアの後に `swap` 命令を使うなどしなくてはならなかったが，ストア–ロード間の順序を保つメモリバリア命令を用いればその必要はなくなる．

さらに，ロックの獲得，解放に相当する命令を使って，獲得の場合はそれ以後のメモリアクセスと，解放の場合はそれ以前のメモリアクセスとの間でのみ順序を保つ release consistency なども提案されている．さらに，ロック (同期オブジェクト) の獲得，解放の際に，共有データを指定する entry release consistency なども提案されている．これらは，基本的には順序の制約を弱めていく方向で提案などが行なわれているといえる．

2.1 節で述べたロックの獲得/解放と，共有されている構造体へのアクセスとの間のメモリアクセス完了順序については，構造体へのアクセスの完了はロックを獲得より後かつロックの解放より前でなくてはならない．このために必要なメモリバリア命令はアーキテクチャにより異なる．processor consistency である Intel P6 ファミリや，SPARC の TSO ではロックの獲得がロードの意味を持つためメモリバリア命令は必要ない．SPARC の PSO であれば，ロックの解放前に `stbar` 命令を用いる必要がある．また，SPARC-V9 の RMO であれば，ロックの獲得のためのロードまたはストアの完了より後に構造体へのアクセスが完了する

ようにするためのメモリバリア命令，ロックの解放のためのストアの完了の前に構造体へのアクセスを完了させるためのメモリバリア命令が必要となる．Alpha の場合も同様であるがメモリバリア命令としては `MB` 命令の 1 種類しかない．MIPS については，ロックの獲得はメモリバリアの意味も持つので，ロックの解放時にのみ `SYNC` 命令を用いればよい．また，PowerPC の場合，Alpha の場合と同様に `sync` 命令を用いればよいが，ロック獲得時には獲得後のメモリアクセスのみが重要で獲得前のメモリアクセスが完了している必要はないので，`isync` 命令[8] を用いることもできる．

3. C 言語による記述の問題点

1 つ目の問題として，前章で述べた共有メモリ上での並列処理に利用すべき命令の多くが C 言語（あるいは処理系）では直接サポートされていないことが挙げられる．このため，粒度の保証や，メモリアクセス完了順序の保証，不可分操作，ロックなどを言語自身では記述することはできず，移植性や効率面で問題があると考えられる．これらの粒度の保証，メモリアクセス完了順序の保証，不可分操作，ロックなどは，機械語命令を含むライブラリルーチンへの呼出しや asm 文を用いて，サポートする必要がある．粒度保証に話を限っても，例えば，GCC や一部の C コンパイラがサポートしている型である `long long` 型，`long double` 型などのデータに関するメモリアクセスは単なるロードかストアであっても二回に分けられることも多い．また，`long` 型が 2 回に分けられることもあるし，`char` 型のデータのストアにおいて，そのバイトを含む大きな単位のロード命令を実行し，該当するバイトのみ変更した後，元のメモリ位置へのストア命令を実行するという場合もある．

一方で，境界整列の問題は，逐次の場合に共通であるので，通常，C コンパイラが適切に対処すると考えられる．C 言語で記述したプログラムで扱うデータについてメモリ上の位置を決める際には適切に境界整列を行い，境界整列を仮定した命令を使用できるようになっているといえる．

2 つ目の問題として，C コンパイラが，C ソースプログラムを機械語プログラムへコンパイルする際においても，ソースプログラム上の実行順序とは異なる順序で機械語命令が実行されるような，実行順の入れ替えが起こり得ることが挙げられる．特に最適化コンパイル時においては，ソースプログラム上の実行順序で「自プロセッサだけが」実行したときと同一の結果が得られる範囲で機械語命令の実行順序を入れ替えたり，無駄と思われるメモリアクセスを削除する可能性がある．これは，プロセッサが与えられた機械語プログラムの実行順序を入れ替えることがあるのと似たケースであり，「他プロセッサも考慮すれば」入れ替えてはならないものを入れ替えないようにするための仕組みが必要である．

これに関しては，C 言語では，データ型に関して型修飾子 `volatile`[7] を指定することで，コンパイル時に，そのデータに関するロードやストアを省略したり，アクセスの順序を入れ替えたりはしないという意味になる☆．ただし，順序を入れ替えないのは `volatile` とされたデータ型に関するメモリアクセスについてのみであるため，図 5 のように (10,20) のデータをプロセッサ P1 からプロセッサ P2 へと伝えたい場合，`*f` と `*p` のアクセス順序のコンパイル時の保存，`*f` と `*q` のアクセス順序のコンパイル時の保存を考慮すると，結局図 5 のすべてのアクセスを `volatile` で行う必要が生じ，`*p` と `*q` のアクセス順序についての制約は本来必要ないにもかかわらず，それも指定されてしまって最適化が妨げられるという問題がある．なお，`volatile` は機械語プログラム上の制約に過ぎず，メモリ上のアクセス完了順序については何も保証しないことにも注意する必要がある．

実際に C 言語で共有メモリ上のプログラミングを行う際には，POSIX threads などのスレッドライブラリが用いられることが多いが，これらでは，データを授受するためのアクセスと同期のためのアクセスを分離し，同期のためのアクセスについてはライブラリのプリミティブで行うようになっている．同期はライブラリルーチンの呼出しで行われ，ルーチンの処理内容を C コンパイラが解析できないとすれば，ルーチン呼出し前の命令の実行と，ルーチン呼出し後の命令の実行の順序を入れ替えることはできないため，2 つ目の問題は解決される．このため，データのアクセスについては，`volatile` などの指定をせずにすみ，また最適化を妨げないという効果もある．ここで，ルーチン内部で機械語命令を用いてメモリアクセス完了順序の保証をしてやれば，C 言語自身にメモリアクセス完了順序の保証のための機能が備わっていなくてもよい．

一方，スレッドライブラリが提供する同期は mutex 変数，条件変数に関する同期などに限られていて，図 5 のようなごくシンプルで高速な同期を行うためのライブラリルーチンすら用意されていない．この他にも 4.4

☆ ただし，ANSI C の仕様では `volatile` の指定に処理系独立な意味は与えておらず，単に無視する処理系もある得るとしている．

節の記述例のような様々な高速な同期の記述ができないという問題がある．その他のスレッドライブラリの問題点としては，関数呼出しのオーバヘッドが必要なことに加え，スレッドは mutex 変数，条件変数上での待ち合わせが長くなりそうな場合には勝手にブロックして他のスレッド（あるいはプロセス）に切り換わることがあり，実行が長時間ブロックする可能性がある．

4. 提案する言語設計

この章では，並行協調拡張 C 言語 XC-cube（an eXtended C language for Concurrency and Cooperation = XC^3）の言語設計において，2 章で述べた共有メモリ上での並列処理に利用すべき命令（に関連する機能）を直接サポートするなどにより，共有メモリ上での並列処理の際の各プロセッサの処理の記述を容易とし，また実行性能も高められるようにする．ここでは，言語レベルでどのように抽象化するかを，利用頻度，実現可能性，効率，機種非依存性などを考慮して設計を行った．XC-cube の新しいプリミティブは組み込みの機能とし，asm 文などとは異なるアプローチである．

記述方式に関する方針としては，基本的なプリミティブを提供するものとし，高級機能はプリミティブを組み合わせて実現するものとした．さらに，よく使う機能については簡単に記述できるようにする．

理想的には，性能を犠牲にせずに同じ記述でプログラムが書け，あとは各プロセッサ / アーキテクチャ毎にコンパイラで最適化ということができるとよい．しかし，現実的には，プロセッサの種類毎に保証可能な粒度に違いがあったり，利用可能な命令の種類が異なったりするため，完全な移植性を提供することは困難である．そこで，一通りのプリミティブの設計・提供は行うが，プロセッサの種類によってはプリミティブの一部が実装されていないかもしれないというアプローチをとることにした．例えば，保証可能な粒度や，cas 命令に相当する機能が提供されるかは，プロセッサのアーキテクチャに依存する．

基本的なプリミティブとしては，

- 同期変数へアクセスするための粒度を保証したプリミティブ
- メモリバリア用プリミティブ

を用いるものとした．さらに，よく使う機能として，

- ロックのプリミティブ

を用いるものとした．

ここで，POSIX threads と同様に，同期変数へのアクセスと一般のデータに関するアクセスは分けて考

文法
⟨atomic_read 式⟩ ::= atomic_read(⟨左辺式⟩)
⟨atomic_write 式⟩ ::= atomic_write(⟨左辺式⟩,⟨式⟩$_1$)
⟨atomic_swap 式⟩ ::= atomic_swap(⟨左辺式⟩,⟨式⟩$_1$)
⟨cas 式⟩ ::= cas(⟨左辺式⟩,⟨式⟩$_1$,⟨式⟩$_2$)
型
⟨左辺式⟩ : τ
⟨atomic_read 式⟩ : τ ⟨atomic_write 式⟩ : τ
⟨swap 式⟩ : τ ⟨cas 式⟩ : int
⟨式⟩$_1$: τ_1 $(\tau_1 \le \tau)$ ⟨式⟩$_2$: τ_2 $(\tau_2 \le \tau)$
τ は整数型かポインタ型か浮動小数点数型（一部除く）
型の他にアラインメントも確認

図 6 粒度保証同期変数アクセスプリミティブ
Fig. 6 atomic access primitives for synchronization variable.

えるとともに，同期変数へのアクセスは明示するものとした．例えば，任意のサイズの構造体などに対して何らかの不可分な操作（例えば 4.4 節の図 **9** のような一貫性のとれた構造体の読み出し / 書き込み）をしたい場合などに，XC-cube コンパイラが自動的に同期変数を準備して適切にアクセスするようなコードを生成するようなことはしない．そのような高級機能は高水準言語で持つべき機能であり，XC-cube 言語上では，限られたサイズ（1 ワード等）の同期変数に関するプリミティブを組み合わせて実現する．これはさまざまな高級機能が実現できるようにするという低水準言語としての柔軟性を保つためである．

以下では各プリミティブについて説明する．

4.1 粒度保証同期変数アクセスプリミティブ

粒度保証同期変数アクセスプリミティブの構文と型付けを図 **6** に示す．

atomic_read 式は⟨左辺式⟩の値を読み出し，その値を結果とする．その際，その値は不可分に読み出される．つまり，例えば，32bit の値を読み出す際に，16bit ずつ読み出すようなことをして，その間に値が変化するようなことはないようにする．

atomic_write 式は⟨左辺式⟩に⟨式⟩$_1$ の値を書き込み，その値を結果とする．その際，その値は不可分に書き込まれる．つまり，例えば，32 ビットの値を書き込む際に，16 ビットずつ書き込むようなことをして，その間に値が変化したり，読み出されたりするようなことはないようにする．

atomic_swap 式は，⟨左辺式⟩の値を読み出し，代わりに⟨式⟩$_1$ の値を⟨左辺式⟩へ代入することを不可分に行う．読み出した値を結果とする．

cas 式は，⟨左辺式⟩の値と⟨式⟩$_1$ の値を比較し，一致すれば（値が等しければ）⟨式⟩$_2$ の値を⟨左辺式⟩へ代入することを不可分に試みる．一致して代入も成功

文法

⟨`membar` 文⟩ ::= `membar(`⟨opt 左辺式範囲リスト⟩$_1$`:`⟨opt 左辺式範囲リスト⟩$_2$`:`
⟨opt 左辺式範囲リスト⟩$_3$`:`⟨opt 左辺式範囲リスト⟩$_4$`);`
⟨`start_access` 文⟩ ::= `start_access(`⟨opt 左辺式範囲リスト⟩$_1$`:`⟨opt 左辺式範囲リスト⟩$_2$`);`
⟨`finish_access` 文⟩ ::= `finish_access(`⟨opt 左辺式範囲リスト⟩$_1$`:`⟨opt 左辺式範囲リスト⟩$_2$`);`

⟨opt 左辺式範囲リスト⟩ ::= | ⟨左辺式範囲リスト⟩
⟨左辺式範囲リスト⟩ ::= ⟨左辺式範囲⟩ | ⟨左辺式範囲⟩`,`⟨左辺式範囲リスト⟩
⟨左辺式範囲⟩ ::= ⟨左辺式⟩ | `*` | ⟨左辺式⟩ `...` ⟨左辺式⟩

型

⟨左辺式⟩：任意の型

図 7 メモリバリアプリミティブ
Fig. 7 Memory barrier primitives.

した場合は 0，一致しないか失敗したら 0 以外の値を結果とする．

これらは，粒度が保証される．ただし，全ての整数型に対応できるとは限らない．また，浮動小数点数型やポインタ型に対しても利用できるが，全ての浮動小数点数型やポインタ型に対応できるとは限らない．たとえば，SPARC-V8 では`cas` 式には対応できない．また，対象となるデータ型は `volatile` としても問題ないが，通常のものでよいとした．

Fetch & add などの不可分操作は `cas` 式を利用して記述すればよい．その際，対象となるプロセッサアーキテクチャによっては生成する機械語コードを図 3 ではなく図 4 のようになるように最適化するといったことは，XC-cube コンパイラが担当すべき仕事ということになる．プログラムの解析（検証，デバッグ），XC-cube 処理系の実装，高水準言語の実装者の言語の修得，においてその手間を小さくするという点から，むやみにプリミティブを増やすようなことはしないことにした．

4.2 メモリバリア用プリミティブ

メモリバリアプリミティブの構文と型付けを図 **7** に示す．

メモリバリアプリミティブを用いない場合のメモリアクセス完了順序は，C 言語と同様とする．つまり，自プロセッサが行うメモリアクセスは自プロセッサにとって順序通りに処理されるようにさえ見えれば，共有メモリ上のアクセスが実際にどのように行われるか，どのような順序で完了していくかについては自由とする．つまり，コンパイラは「自プロセッサだけが」実行したときと同一の結果が得られる範囲で機械語命令の実行順序を入れ替えたり，無駄と思われるメモリアクセスを削除する可能性がある．さらに，生成された機械語プログラムをプロセッサが実際に実行する際にも，2.4 節の緩いメモリモデルのように共有メモリ上のアクセスの完了順序が入れ替わる可能性がある．ここにメモリバリアプリミティブを用いることでアクセス完了順序の制約を与える．

既存の C 言語では機械語レベルの緩和されたメモリモデルがある意味そのまま見えてしまっていたが，ここでは，拡張 C 言語の仕様としてアーキテクチャに依存しない C 言語レベルのメモリモデルに統一する．このモデルは既存の機械語レベルのメモリモデルのどれよりも緩い（制約の少ない）ものになっているため，最適化の余地が広くなる．実現としては，メモリバリアプリミティブを用いることで，C プログラムが機械語命令列に変換される際に，アクセスの省略や順序の入れ替えを防いだり，必要なメモリバリア命令を含めたりして，アクセスの完了順序の保証がされるようにする．

`membar` 文は，`membar` 文以前の⟨opt 左辺式範囲リスト⟩$_1$ に含まれる⟨左辺式範囲⟩の（メモリ）位置からの読み出し，⟨opt 左辺式範囲リスト⟩$_2$ に含まれる⟨左辺式範囲⟩の（メモリ）位置への書き込み，が完了してから，`membar` 文以後の⟨opt 左辺式範囲リスト⟩$_3$ に含まれる⟨左辺式範囲⟩の（メモリ）位置からの読み出し，⟨opt 左辺式範囲リスト⟩$_4$ に含まれる⟨左辺式範囲⟩の（メモリ）位置への書き込みが完了することを保証する．ここで，⟨左辺式範囲⟩は，「⟨左辺式⟩」でそのメモリ位置，「`*`」は全てのメモリ位置，「⟨左辺式⟩$_1$ `...` ⟨左辺式⟩$_2$」は⟨左辺式⟩$_1$ のメモリ位置から⟨左辺式⟩$_2$ のメモリ位置までの連続した範囲のメモリ位置を表す．`membar` 文は同期変数アクセス間の順序を保証するのに主に利用する．そのためだけなら⟨左辺式範囲リスト⟩の代りに単に⟨左辺式⟩であれば十分であるが，同期変数アクセスと通常のデータに関するアクセスの間の順序保証をより一般的に行うためにも利用できるようにしてある．

`start_access` 文は，直前の同期位置へのアクセスが完了した後に，⟨opt 左辺式範囲リスト⟩$_1$ に含まれる⟨左辺式範囲⟩の（メモリ）位置からの読み出し，

⟨opt 左辺式範囲リスト⟩$_2$ に含まれる⟨左辺式範囲⟩の（メモリ）位置への書き込み，が完了することを保証する．一方，finish_access 文は，直後の同期位置へのアクセスが完了する前に⟨opt 左辺式範囲リスト⟩$_1$ に含まれる⟨左辺式範囲⟩の（メモリ）位置からの読み出し，⟨opt 左辺式範囲リスト⟩$_2$ に含まれる⟨左辺式範囲⟩の（メモリ）位置への書き込み，が完了することを保証する．これらは，同期変数アクセスと通常のデータに関するアクセスの間の順序を保証するのに利用する．ここで，関数呼出しなどがあって直前や直後の同期変数へのアクセスが不明な場合は，保証を保守的に行う．

順序保証を保守的に行う分には高速化の可能性が減る以外の問題はない．特に，同期変数へのアクセスの直前にfinish_access(*:*)，直後にstart_access(*:*) を常に記述することで，メモリバリアが不十分かもしれないということについて深く考えなくてすむ．同様に 4.1 節の同期変数アクセスのプリミティブについて，その直前にfinish_access(*:*)，直後にstart_access(*:*) を自動的に行うような API をマクロなどで提供しておけば，メモリバリアプリミティブの記述はしなくて済む．ただし，同期において対象とすべき通常のデータについて，データの範囲はしばしば明らかであるので，その範囲を明示的に記述することで，最大限に高速化の余地を残せるようにしている．さらには，同期変数のアクセスの前後どちらで対象となる通常のデータのアクセスが行われるかもも分かっているはずなので，必要なメモリバリアプリミティブのみを記述することで，性能向上が図れるようにしている．

4.3 ロックプリミティブ

ロックプリミティブの構文と型付けを図 8 に示す．

try_lock 式は，⟨左辺式⟩のロックの獲得を試み，成功したら 0，失敗したら 0 以外の値を結果とする．spin_lock 文は，スピンロックにより⟨左辺式⟩のロックを獲得する．release_lock 文は，⟨左辺式⟩のロックを解放する．

try_rlock 式は，⟨左辺式⟩の read ロックの獲得を試み，成功したら 0，失敗したら 0 以外の値を結果とする．spin_rlock 文は，スピンロックで⟨左辺式⟩の read ロックを獲得する．release_rlock 文は，⟨左辺式⟩の read ロックを解放する．

try_wlock 式は，⟨左辺式⟩の write ロックの獲得を試み，成功したら 0，失敗したら 0 以外の値を結果とする．spin_wlock 文は，スピンロックで⟨左辺式⟩の write ロックを獲得する．release_wlock 文は，⟨左辺式⟩の write ロックを解放する．

文法
⟨lock 型⟩ ::= lock_t
⟨try_lock 式⟩ ::= try_lock(⟨左辺式⟩)
⟨spin_lock 文⟩ ::= spin_lock(⟨左辺式⟩);
⟨release_lock 文⟩ ::= release_lock(⟨左辺式⟩);
型
⟨try_lock 式⟩ : int ⟨左辺式⟩ : lock_t

文法
⟨rwlock 型⟩ ::= rwlock_t
⟨try_rlock 式⟩ ::= try_rlock(⟨左辺式⟩)
⟨spin_rlock 文⟩ ::= spin_rlock(⟨左辺式⟩);
⟨release_rlock 文⟩ ::= release_rlock(⟨左辺式⟩);
⟨try_wlock 式⟩ ::= try_wlock(⟨左辺式⟩)
⟨spin_wlock 文⟩ ::= spin_wlock(⟨左辺式⟩);
⟨release_wlock 文⟩ ::= release_wlock(⟨左辺式⟩);
型
⟨try_rlock 式⟩ : int ⟨try_wlock 式⟩ : int
⟨左辺式⟩ : rwlock_t

図 8 ロックプリミティブ
Fig. 8 lock primitives.

```
/* writer */
vn = obj->vn;
atomic_write(obj->vn, 0);
start_write(obj->f1, obj->f2, obj->f3);
obj->f1 = f1;
obj->f2 = f2;
obj->f3 = f3;
finish_write(obj->f1, obj->f2, obj->f3);
atomic_write(obj->vn, vn+2);

/* reader */
do{
  do{
    llbar(obj->vn : obj->vn);
    vn = atomic_read(obj->vn);
  }while(vn == 0);
  start_read(obj->f1, obj->f2, obj->f3);
  f1 = obj->f1;
  f2 = obj->f2;
  f3 = obj->f3;
  finish_read(obj->f1, obj->f2, obj->f3);
}while(atomic_read(obj->vn) != vn);
```

図 9 バージョン番号による一貫性の保たれた構造体の更新と読み出し
Fig. 9 consistent data structure reading/writing using version number.

4.4 記 述 例

この節では，POSIX threads などのスレッドライブラリではうまく効率のよい記述ができなかった共有メモリ上での並列処理の記述例を示す．

図 9 は，バージョン番号による一貫性の保たれた構造体の更新と読み出しの例である．ここで，llbar 文は membar 文の一部のフィールドを省略するもので，ロード–ロード間の順序の保証を行なうものとする．start_write 文はstart_access 文の一部のフィールドを省略するもので，直前の共有変数のアクセスとス

```
/* processor 1 */         /* processor 2 */
atomic_write(a1, 1);      atomic_write(a2, 1);
slbar(a1:a2);             slbar(a2:a1);
v = atomic_read(a2)       v = atomic_read(a1)
```

図 10 Dekker のアルゴリズム（一部）
Fig. 10 Dekker's algorithm (part).

```
struct s1 { lock_t lk; int a; double b; };

void trans_half(struct s1 *x1, struct s1 *x2,
                double *hh){
  spin_lock(x1->lk);
  start_read(x1->a, x1->b);
  spin_lock(x2->lk);
  start_read(x2->a, x2->b);
  {
    double b = (x1->b), h = b * 0.5;
    x1->b = b - h;
    x2->b += h;
    *hh += h;
    x1->a++; x2->a++;
  }
  finish_write(x2->a, x2->b);
  release_lock(x2->lk);
  finish_write(x1->a, x1->b);
  release_lock(x1->lk);
}
```

図 11 ロックの獲得中の変数アクセス
Fig. 11 access to variables with acquired lock.

```
  do{
    v = count;
  }while(cas(count, v, (v+i)%N));
```

図 12 剰余環における不可分な加算
Fig. 12 atomic addition in residue ring.

トアの間の順序の保証を行なうものとする．同様に，`start_read` 文は`start_access` 文の一部のフィールドを省略するもので，直前の共有変数のアクセスとロードの間の順序の保証を行い，`finish_write` 文は `finish_access` 文の一部のフィールドを省略するもので，直後の共有変数のアクセスとストアの間の順序の保証を行うものである．また，バージョン番号（`vn`）へのアクセスでは粒度を保証する必要がある．この例では，ロックなどのコストが高く，同時アクセスを過度に制限するプリミティブを用いずに，一貫性のとれた状態の構造体の読み出しを可能としている．このような手法は，オブジェクト指向並列言語を高水準言語として用いたとき，オブジェクトデータの効率よい一貫性管理の実装に利用できる．

図 **10** には，共有メモリ上でのロードとストアだけで，相互排他を実現する Dekker のアルゴリズムの一部を示す．ここで，`slbar` 文は `membar` 文の一部のフィールドを省略するもので，ストア–ロード間の順序の保証を行なうものとする．Dekker のアルゴリズムでは，後のロードがストアを追い越してはならないので，そ

```
/* push elm at the head */
finish_access(*elm:*elm);
do{
   elm->next = h = list_head;
   finish_write(elm->next);
}while(cas(list_head, h, elm));

/* pop elm at the head */
do{
   elm = atomic_read(list_head);
   start_read(elm->next);
}while(cas(list_head, elm, elm->next));
start_access(*elm:*elm);
```

図 13 リストの先頭での不可分なリスト要素の追加/抽出
Fig. 13 atomic insertion/extraction of a list element at the list head.

れをメモリバリアプリミティブで示している．この例では，ロックなどのコストが高く，同時アクセスを過度に制限するプリミティブを用いずに「早い者勝ち」を判定できている．このような手法は，遅延タスク生成を行うマルチスレッド言語を高水準言語として用いたとき，タスクスティールの成否/有無の判定の実装に利用できる．

図 **11** は，単なるロックを使った例であるので POSIX threads でも同様の記述はできる．違いは，オーバヘッドやブロッキングの可能性の他に，メモリバリアを対象となるデータについてのみとしている点がある．この例では，ロックを獲得する前に変数を読み出してしまったり，ロックの解放が変数への書き込みを追い越さないようにメモリバリアプリミティブを用いていて，対象となるデータを特定している．ただし，対象となるデータがよく分からなかったり，ポインタを含むなどして辿れるデータすべてについてメモリバリアを用いたりしたい場合は，図 11 の`start_read` の代わりに，`start_access(*:*)`，`finish_write` の代わりに，`finish_access(*:*)` としておけば高速化の余地が減ることを除いて問題ない．

図 11 の例で alias なしと分かっているとすると：

- `*hh` からの読み出しをロックの獲得前に移動させてもよい．
- `*hh` への書き込みをロックの解放後に移動したり，遅らせたりしてよい．
- `spin_lock(x1->lk);` の後であれば，`x1->a`，`x1->b` の読み出しを`spin_lock(x2->lk);` の前に移動させてもよい．
- `release_lock(x1->lk);` の前であれば，`x1->a`，`x1->b` の書き込みを `release_lock(x2->lk);` の後に移動させたり，遅らせたりしてよい．
- `spin_lock(x1->lk);` と`spin_lock(x2->lk);` の間に順序の制約はない．

などとなる．ここで移動したり遅らせたりするのは，プロセッサだけが実行時にしてもよいのではなく，C コンパイラがコンパイル時にしてもよい．これらは範囲を指定せず全てのアクセスについてメモリバリア指定するとできなくなるものである．さらに，分散共有メモリを実現する処理系などのキャッシュ等の管理において，ロックと結び付くデータを限定する（例えば，`x2->lk` には `x2->a`，`x2->b` であり，`*hh` や `x1->a`, `x1->b` は含まないとする）ことで通信量や回数等を減らすことができる．

図 **12** の例は不可分な加算の応用例であり，更新のための新しい値に関する式を自由に記述できることを示している．POSIX threads などでは，mutex 変数を使った相互排他で不可分な加算などを記述する必要があるが，それと比べてロックなどのコストが不要，ブロックしてしまう可能性がないという大きな利点がある．このような `cas` 式は，並列言語等を高水準言語として用いたとき，バリア同期，終了判定，リダクション，ごみ集めの実装などに利用できる．

図 **13** は，`cas` プリミティブを使ってロックなどを用いずに，リンクトリストの先頭に要素を追加する例を示している．他のプロセッサがリストを先頭から辿ったときに次の要素をまだ指していないことがないようにするためにメモリバリアプリミティブを用いる必要がある．また，逆に先頭から要素を抽出する例も示す．このような手法は，オブジェクト指向並列言語を高水準言語として用いたとき，メッセージキューの実装などに利用できる．

5. GNU C コンパイラを使った実現

前章では，移植性や実行効率を高めるため，拡張 C 言語 XC-cube の共有メモリ向けプリミティブとその記述例を示した．一方，プリミティブの普及を図るには既存のシステムでもできるだけ利用可能であることが望ましい．そこでプリミティブを少し変更し，GCC の拡張機能による実現を行った．

5.1 プリミティブの変更

構文解析器，型チェッカ，コード生成器の拡張などを伴わずに，GCC の拡張機能の範囲内でプリミティブを実現するためには，プリミティブの変更が必要となる．

メモリバリアプリミティブについては，対象となる範囲指定を記述できないものとした．不定長のリストが構文拡張以外では扱いにくいこと，範囲指定を有効利用するにはコンパイラで最適化などにこれを生かす必要があるが，コンパイラに手を入れない以上それが望めないことがその理由である．`membar` 文の代わりに，`ssbar();`，`slbar();`，`sabar();`，`lsbar();`，`llbar();`，`labar();`，`asbar();`，`albar();`，`aabar();` というプリミティブの文を用意し，ストア–ストア間なら`ssbar();`，ストア–ロード間なら`slbar();`，その両方なら`sabar();` というふうに用いる．

また，`start_access` 文などでは，直前の同期変数へのアクセスは自動的には見つけられないので，`start_access_after_lock();`，`start_access_after_read();`，`start_access_after_write();`，`finish_access_before_unlock();`，`finish_access_before_read();`，`finish_access_before_write();` というプリミティブの文を用意する（`start_read_after_lock();` なども）

同期変数へのアクセスについては，技術的な理由から`atomic_write(⟨左辺式⟩,⟨式⟩);` は文にすることにする．また，同期変数の型ごとに`atomic_read_char`，`atomic_read_int`，`atomic_read_double`，`atomic_read_ptr` といったプリミティブを用意する．これは GCC の拡張機能の範囲内で扱う以上，型付けの結果で生成するコードを切り替えることができないためである．ここで `unsigned` は無視することにする．

また，GCC 以外でも同じマクロを使えるようにするためにも，プリミティブをデータ型に分けて準備する必要がある．これは，ライブラリルーチンの関数は `int cas_int(int *, int, int)` のような型を持たなくてはならないためである．

また，ポインタ型には注意が必要である．C の仕様では，`char *`は 1 バイト単位のアドレス，`int *`は 4 バイト単位のアドレスを用いるなどポインタとしては同じ場所を指すが，整数としてみたときは異なる値であるというのが許されている☆．このため，各基本型のポインタ毎にもプリミティブを分ける必要がある．基本型のポインタ以外は `void *` で指すしかない．これ以外の変更に係わる問題点については 5.8 節で述べる．

5.2 GNU C コンパイラの構成と拡張機能

GNU C コンパイラ[11)]では，ソース言語における C のプログラムは，構文解析と型付けの後，まず RTL と呼ばれる中間言語のプログラムに変換され，解析・最適化などを行った後，対象プロセッサの機械語プログラムへ変換される（実際には，アセンブラがアセ

☆ GCC の場合は当てはまらないようであるが，実際，ポインタの型によっては異なるアドレス値を用いる処理系もある．

ンブリ言語のプログラムを機械語プログラムに変換するが，特に区別する必要がなければ，アセンブリ言語のプログラムも機械語プログラムというレベルで捉えておくことにする．) C 言語，RTL，機械語の各レベルではそれぞれのデータ型，プログラム表現が用いられる．

GNU C コンパイラには C 言語に対するさまざまな機能拡張が施されているが，なかでも asm 文は，アセンブリ言語プログラムをどのように生成したいかを指示することができる．アセンブリ言語プログラム片のテンプレートと，オペランドの指定，オペランドに関するレジスタ割付の制約の記述や副作用の及ぶ範囲の記述が可能であり，かなり強力なものとなっている．

5.3 言語拡張の方法

コンパイラを変えない言語拡張の実現方法としては，言語の意味の部分，特に特殊な命令の利用については，asm 文かアセンブリ言語で記述されたルーチンの呼出しが考えられる．言語の意味の部分，特に型チェックや型付けについては，5.1 節で述べたようにプリミティブから型が分かるようにするしかない．構文の部分については，マクロか（インライン展開される）ルーチンの呼出しが考えられる．今回は GCC を用いた実現であるので，できるだけ asm 文を用いればよいが，asm 文が使えない C コンパイラ上で将来実現することを考えるとルーチン呼出しを用いる可能性がある．今回は，まずはプリミティブをマクロとして提供し，asm 文を必要とするところには asm 文を用いることにした．

GCC を用いた実現では，コンパイルの過程と同等の処理を行うことにした．つまり，RTL のプログラムから機械語プログラムへどうマッピングするかと，C のプログラムから RTL のプログラムへどうマッピングするかを考える必要がある．RTL のプログラムから機械語プログラムへのマッピングでは適切な機械語命令の選択などがある．また，C のプログラムから RTL のプログラムへのマッピングでは，データ型の変換と必要な部分の実行順序の保存がある．

2 章で扱ったプロセッサのうち，SPARC-V8 以外の部分については図 **14** ようなマクロを用いて，ロックや read/write ロックは実現できる．ただし，図 14 では，マクロ中でマクロを使う結果，`int *_loc=(_loc)` のように展開されて不都合が生じるので，実際には `extern __inline__`によるインライン展開を用いている．また，SPARC-V8 については，`ldstub` 命令を用いるとロックや read/write ロックが実現できる．

```
#define lock_t int
#define try_lock(loc) cas_int((loc),0,1)
#define spin_lock(loc)\
do{int *_loc=&(loc);\
   while(try_lock(*_loc))llbar();}while(0)
#define release_lock(loc)\
   atomic_write_int((loc),0)

#define rwlock_t int
#define try_rlock(loc)\
({int *_loc=&(loc), _c=atomic_read_int(*_loc);\
    ((_c<0) || cas_int(*_loc,_c,_c+1));})
#define spin_rlock(loc)\
do{int *_loc=&(loc);\
   while(try_rlock(*_loc))llbar();}while(0)
#define release_rlock(loc)\
do{int *_loc=&(loc), _c; do{_c = *_loc;}\
while(cas_int(*_loc,_c,_c-1));}while(0)

#define try_wlock(loc) cas_int((loc),0,-1)
#define spin_wlock(loc)\
do{int *_loc=&(loc);\
   while(try_wlock(*_loc)) llbar();}while(0)
#define release_wlock(loc)\
   atomic_write_int((loc),0)
```

図 **14** 不可分アクセスプリミティブを用いたロックの実現
Fig. 14 lock implementation using atomic access primitives

5.4 適切なプロセッサ命令の選択

適切なプロセッサ命令の選択は，提案するプリミティブに対応する命令がそのままあればそれを選ぶだけでよい．ここでは注意を要する点について述べる．

粒度保証は，Alpha の 1 バイト, 2 バイトや，Pentium の 8 バイトが問題となる．Alpha の 1 バイト, 2 バイトについては，1 バイトのストアのための大きな単位のロードとストアに `ll/sc` 命令を使ってやることで，他のプロセッサと競合することなく 1 バイトのストアが実現できる．Pentium で，8 バイトのデータを一括して扱いたい場合は，`CMPXHG8B` 命令を `cas` 式の実現以外に，`atomic_read` 式や`atomic_write` プリミティブの実現にも利用してやればよい．つまり，`cas` 命令のロード・比較・ストアを一括して行う機能のうち，ロードの部分，ストアの部分だけを利用する．

実際には，4 バイト，8 バイト整数についての `cas` 命令（または `ll/sc` 命令）を用いて `cas` 式の機能が実現できていさえすれば，効率の問題は別として，1，2，4，8 バイトの，整数型，実数型，ポインタ型に関する 4.1 節の粒度保証同期変数アクセスプリミティブはすべて実現できる．

メモリバリアについては，SPARC-V9 については，そのまま当てはまる `membar` 命令を用いればよいが，Alpha, MIPS, PowerPC では，`MB` 命令や `SYNC` 命令（または `isync` 命令）で保守的に近似してやればよい．一方，Pentium, SPARC-V8 の TSO については，

slbar() プリミティブに相当するものについてのみ xchg 命令, swap 命令をダミーのメモリ位置とレジスタを用いて行えばよい．

5.5 ソースレベルの型からの変換

実は GCC だけを考えるなら，データ型のマッピングは sizeof()，__alignof()__の境界整列の情報について if 文を用いれば，マクロ展開時に型のサイズ情報などが得られなくてもコンパイルはできる．

ただし， GCC が分かるだけでなくて，プログラマにもサイズに関するマクロを提供できたほうがそれはそれで便利なので，サイズ等に関するマクロを提供し，自身でも利用する．サイズ等はコンパイラオプションで異なるので「gcc -E -dM a.c」で出力されるマクロ定義に応じて適切にマクロを定義する必要がある．

対応関係に従って，ソースレベルの int を RTL レベルの SImode に対応させたり，char を QImode に対応させることになる．

5.6 ソースレベルの実行順序の保存

C コンパイラで順序を入れ替えさせないためには，

- サブルーチンコールか
- ある場所の値をアドレスとした不明な場所への書き込みか，
- メモリ（"memory"）に関して副作用があるとする GCC の asm 文

を用いる方法が考えられる．今回は GCC の asm 文を用いている．実際には，メモリバリアのための asm 文で，必要なメモリバリア命令（空のこともある）とともに，副作用の及ぶ範囲に"memory"を指定すればよい．

5.7 コンパイルエラーの出力

マクロでプリミティブを提供する際に問題になるのが，適切でないサイズが用いられたときに，どのようにそれをコンパイルエラーにするかという問題がある．特にエラーであるかどうかが sizeof の値に依存する場合は，普通の if 文を用いたのでは，実行時にならないとエラーが報告できない．

これは，例えば，sizeof() の値が 4 でなくてはならなかったとすると，次のようなトリックを使うことで対処できる．

```
{char not_supported[sizeof(tp)==4? 1: -1];}
```

これにより，コンパイル時にエラーが報告され，また，GCC の場合は変数名を表示するので，この場合「not_supported」というエラーメッセージを出力することができる．

5.8 コンパイラを拡張しないことによる限界

コンパイラ拡張ならできて，マクロを被せた asm 文

```
atomic_add(s, -1);
//  = do{ int v = s; }while( cas(s, v, v-1) );
//          or
//  = mutex_lock(sl); s--; mutex_unlock(sl);
while(s > 0);
```

図 15 不可分加算によるバリア同期
Fig. 15 barrier synchronization by atomic addition.

あるいはルーチン呼び出しでの実現などでは難しい問題点を整理しておく．コンパイラを拡張しないことの影響には，言語仕様に及ぼす影響と，実装に及ぼす影響がある．

言語仕様に及ぼす影響としてはコンパイラに型を付けてもらえないので型を明示しないといけないことがある．例えば，同じatomic_write(x,a) を，a の型に応じてatomic_write_int(x,a) としたり，atomic_write_short(x,a) としたり，あるいは atomic_write_double(x,a) のように書き分けなくてはならない．これは面倒であるばかりでなく，a の型が，union { char c[4]; short s[2]; } のようなユーザ定義の型だとあらかじめその型のプリミティブを準備しておくことはできない．ここで a の型が union { char c[4]; short s[2]; } * というポインタ型なら，void *型に関するプリミティブを準備して，atomic_write_ptr((void **)x, (void *)a) とはできる．しかし，型チェックが回避されてしまうので，本来コンパイル時に静的検出できるはずのバグで実行時に型クラッシュするかもしれない．

その他の言語仕様に及ぼす影響として，不定長のリストを用いても動的に処理すると遅いため，それならないほうがよいという点がある．また「start_access」などで直前の同期変数のアクセスを特定することができないので，これも明示してもらう必要が生ずる．

実装に及ぼす影響としては，限定したエラー処理しかできないこと，最適化をしないため，あるいは最適化が妨げられるため効率が悪くなること，ルーチン呼出しの場合は呼出しのオーバヘッドが必要となることが挙げられる．最適化に関していえば，拡張部分を asm 文や，アセンブリ言語で書かれたルーチンの呼出しで与えられてもその部分の意味がわからないので，プログラムの意味を保存する範囲で命令の削除，順序の入れ替え，結合，合成を行うなどといった最適化ができない．

6. 予 備 実 験

予備実験として，IBM RS/6000 SP の 1 ノード上で評価を行った．ノードは 4 プロセッサの SMP となるよう構成した．各プロセッサは 332MHz PowerPC

表 **1**　予備実験の結果（バリア同期：マイクロ秒）
Table 1　result of preliminary experiments (barrier synchronization, μs).

# of PEs	1	2	3	4
non-opt	0.092	0.63	1.78	3.7
opt	0.086	0.62	1.74	3.4
lib/opt	0.093	0.65	1.74	3.6
mutex	0.69	4.2	200	8000

604e で，L1 cache は命令 32KB，データ 32KB，L2 cache は 0.25MB である．不可分加算を利用したバリア同期（図 **15** のコードに相当）の一回当りの処理時間をプロセッサ数を変化させて測定した．測定対象としたのは，拡張 C 言語で `cas` 機能を組み込みとして，不可分加算を `do{ int v = s; }while( cas(s, v, v-1) );` のように 記述したとして，これを PowerPC の `lwarx/stwcx.` 命令を用いて，図 3 のように単純にコード生成した場合（non-opt），またこれを最適化して図 4 のようなコードとした場合（opt），さらに図 4 のようなコードをライブラリ関数として呼び出した場合（lib/opt）である．拡張 C 言語のコンパイラは開発中なので不可分加算以外の部分については gcc 2.95.2 が生成した`-O2` のコードを利用した．測定結果を表 **1** に示す．

POSIX threads の`pthread_mutex_lock` 等を用いた場合（mutex）は，スレッドがブロックして極端に遅くなっていることが分かる．それ以外の場合はそれほど変わらないので，性能の面では，提案するプリミティブをコンパイラに組み込んでもそれほど変わらない可能性があることが分かる．ただし，もっとスレッド間の衝突が少ないケースでは差が大きくなる可能性がある．また，コンパイラに組み込むことで，記述しやすく，コンパイラに型チェックや最適化や適正なエラー処理をさせることができるようになる．

7. おわりに

本論文では，C 言語を拡張して共有メモリ上での並列処理を容易に，効率よく行うために，C 言語レベルのメモリモデルとその上のプリミティブの設計を示すとともに，その GNU C コンパイラを用いた実現について述べた．今後は，検討を進めると共に，そのコンパイラ版の実装について研究していきたい．

参考文献

1) Adve, S. V., Pai, V. S. and Ranganathan, P.: Recent Advances in Memory Consistency Models for Hardware Shared Memory Systems, *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, Vol. 87, No. 3, pp. 445–455 (1999).
2) DEC: Alpha AXP アーキテクチャ概要, 共立出版 (1993). (日本 DEC 監訳・編, 高澤嘉光 訳).
3) Greenwald, M. B.: *Non-Blocking Synchronization and System Design*, PhD Thesis, Stanford University (1999).
4) Heinrich, J.: MIPS R4000 Microprocessor User's Manual, Second Edition (1994).
5) Intel: Intel Architecture Software Developer's Manual (1999).
6) Kane, G. and Heinrich, J.: *MIPS RISC Architecutre*, Prentice Hall (1992).
7) Kernighan, B. W. and Ritchie, D. M.: プログラミング言語 C 第 2 版, 共立出版 (1989). (石田晴久 訳).
8) Motorola Inc.: PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors (1997).
9) SPARC International, Inc.: *The SPARC Architecture Manual – Version 8*, Prentice Hall (1992).
10) SPARC International, Inc.: *The SPARC Architecture Manual – Version 9*, PTR Prentice Hall (1994).
11) Stallman, R. M.: *Using and Porting GNU Compiler Collection*, Free Software Foundation, Inc., for gcc-2.95 edition (1999).
12) 八杉昌宏, 馬谷誠二, 小宮常康, 湯淺太一: マルチコンテキスト管理をサポートする実装用言語, 情報処理学会プログラミング研究会 (SWoPP'99) (1999).

(平成？年？月？日受付)
(平成？年？月？日採録)

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学大学院理学系研究科情報科学専攻博士課程修了．1993 ~ 1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．1998 年より科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

高田　　潤

1978 年生．2001 年京都大学工学部情報学科卒．同年より同大学大学院情報学研究科修士課程に在学中．並列処理と言語処理系に興味を持つ．

田畑　悠介

1976 年生．2000 年京都大学工学部情報学科卒．同年より同大学大学院情報学研究科修士課程に在学中．並列処理と言語処理系に興味を持つ．

小宮　常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．

湯淺　太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門（共著）」，「Scheme 入門」，「C 言語によるプログラミング入門」他．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．