寄　　稿

# 並列プログラミングとメモリコンシステンシモデル

平　田　博　章*
hrt@kit.ac.jp

## 1. はじめに

現在のコンピュータは、複数のプロセッサコアがメモリ（主記憶装置）を共有する、共有メモリ型の並列コンピュータとして構成されるのが一般的です。いわゆるマルチコア CPU は、高価なハイエンドのサーバーコンピュータから、デスクトップコンピュータやポータブルコンピュータ、さらには組み込み用コンピュータやスマートフォンに至るまで、普通に使用されています。現在では、1 個の LSI チップの中に数十個のプロセッサコアを搭載する製品も製造・販売されるようになりました。しかし、複数のプロセッサコアを搭載していても、アプリケーションプログラムが逐次プログラムとして書かれていたのでは、実行時間を短縮することはできません。つまり、マルチコア CPU のメリットを得るためには、プログラマは並列プログラムを書かなければなりません。使用するアルゴリズムに固有の並列性を抽出し、例えばマルチスレッドライブラリを用いて、その並列処理単位を 1 つのスレッドとして実行するよう明示的に記述する必要があります。

共有メモリモデルに基づく並列プログラムを作成する場合に注意すべき問題に、排他制御や同期があります。複数のスレッドが共有する変数に対しては、その変数にアクセスする際にはロックを掛けて他のスレッドがアクセスすることを禁止します。このルールを忠実に守ってプログラムを書くだけでも、実際には骨の折れる作業ですが、一方、そのような行儀のよいプログラムを書いていたのでは実行時間が短くならない、ということもよくあります。そんな時、レベルの高いプログラマなら、ルールを少し破っても速いプログラムを書こうとするでしょう。また、「ロックを掛ける」というやり方自体に抵抗感を憶える人達もいます。Lock-free アルゴリズム（や Wait-free アルゴリズム）を開発したり使用しようとしたりする人達です。Lock-free というのは、本来は「スレッドが止まってしまう（ブロックされてしまう）ことがない」という性質を意味しているのですが、実際、やり方としてもロック機構を使用しません。

しかし、そのような試みは、容易には成功しません。俗語のレベルですが、ソフトウェアのバグの中にハイゼンバグ（Heisenbugs）と呼ばれる種類のものがあります。量子力学において不確定性原理を唱えた物理学者ハイゼンベルク（Heisenberg）の名前をもじったもので、バグの原因を突き止めようとするとバグの症状が変化したり、バグが起こらなかったりするようなバグを指します。Lock-free アルゴリズムを開発する場合や、故意にロック機構をすり抜けるようなプログラムを作成しようとすると、ハイゼンバグに遭遇していつまで経ってもデバッグが終わりそうにない、という悲惨な状況に陥ることも少なくないようです。

ただ、そのようなハイゼンバグの発生に対して「実際に何が起こっているのか？」を類推できる（あるいは、最初からそのようなバグの発生を抑制する）ためには、まず、コンピュータのハードウェアレベルで、メモリへのアクセスがどのようになされているのかを理解しておく必要があります。実際、長年ソフトウェアの開発に携わってきたベテランのプログラマでも、また、バリバリ働ける若いプログラマでも、プログラミングに活かせるハードウェアの知識を持つ人はあまりいないように感じます。そこで、本稿では、共有メモリ型並列コンピュータのメモリ機構について、プログラマからの視点を意識しながら、入門的な解説をしようと思います。

＊情報工学・人間科学系　教授

## 2. 簡単な排他制御の例

図1のプログラムを例に考えます。スレッドAは関数foo_A()を、スレッドBは関数foo_B()を、それぞれ実行するものとします。変数countのインクリメント（7行目、13行目）は変数countの値を読み出して、1を加算し、その加算結果を書き戻すRead-Modify-Write（RMW）の操作で、スレッドAとBが同時に行うことは阻止しなければなりません（そうしなければ、スレッドAとBが1ずつ加算したのに、結果が1しか増えていない、という事態が生じるかもしれません）。従って、このRMW操作は1つのスレッドのみが一気に（アトミックに）行わなければならず、このようなプログラム部分をクリティカルセクション（Critical Section: CS）と呼びます。

変数a、bは、値が1ならば、スレッドA、BがそれぞれCSに入っていることを示すための変数です。スレッドAは変数aを1にして、CSに入ることを宣言します（5行目）。次に、変数bの値を調べて、スレッドBがCSに入っていないことを確かめます（6行目）。変数bの値が0なら、スレッドBはCSには入っていないので、7行目へと進みますが、bの値が1なら、0になるまで（つまり、スレッドBがCSを出るまで）待ちます。CSを抜けると、変数aの値を0にして、スレッドAがCSにいないことをスレッドBから確認できるようにします（8行目）。スレッドBも同様の動作です。

もしも、スレッドAが変数aの値を1にするのと、スレッドBが変数bの値を1にするのがほぼ同時であったならば、両者はそれぞれ6行目、13行目で無限ループに陥り、デッドロックが生じてしまいます。この点で図1のプログラムは不完全と言わざるを得ません。しかし、少なくとも、このプログラムはスレッドAとBの両方が同時にCSに入ることを阻止できるはずですが、果たしてそれは本当でしょうか？

自分がCSに入ると宣言してから、相手がCSに入っているかどうかを調べているのですから、スレッドAとBの両方がCSに入ることはなさそうに思えます。しかし、実際にはそうとは限りません。スレッドAとBの両方がCSに入ることはないと期待したとすれば、それは恐らく、スレッドAが変数aの値を1にすると、即座にスレッドBでもそれが観測できる（読み出せる）と思い込んでいたからではないでしょうか。コンピュータにもよりますが、多くの場合、スレッドAを実行しているプロセッサコアで変数aの値を書き変えても、それがスレッドBを実行している別のプロセッサコアには即座には伝わりません。従って、スレッドAが変数aの値を0から1に変更しても、スレッドBが変数aの値を読み出すと、まだ0としか読めないかもしれないのです。この場合、スレッドBは14行目へと進みます。スレッドAも、変数bの値が1になったのがまだ伝わっていなければ、6行目から7行目へと進み、図1のプログラムでは排他制御できないことがわかります。

```
01:  int  count = 0;
02:  int  a = 0, b = 0;

03:  void foo_A(void)
04:  {
05:      a = 1;
06:      while( b == 1 );
07:      count ++;
08:      a = 0;
09:  }

10:  void foo_B(void)
11:  {
12:      b = 1;
13:      while( a == 1 );
14:      count ++;
15:      b = 0;
16:  }
```

**図1　排他制御のプログラム例（C言語）**

## 3. メモリコンシステンシモデル

メモリコンシステンシモデル（Memory Consistency Model）[1-3]は共有メモリへのアクセス順序とプログラム実行の正当性との関係を扱うモデルです。イベントオーダリングモデル（Event Ordering Model）と呼ぶ場合も

あります。

メモリ（主記憶装置）の速度はプロセッサに比べるとかなり遅いのが現実です。メモリに1回アクセスするだけの時間で、プロセッサでは整数の足し算を100回以上行うことができるほどの差があります。また、メモリの同じ番地に複数のプロセッサから同時にアクセスすることはできません。メモリ装置としての性質がそういうものですからどうしようもありません。そこで、プロセッサコア（ごと）にキャッシュメモリを配置して、プロセッサから見たメモリの性能を見かけ上、向上させています。キャッシュメモリは主記憶装置ほどの容量はありませんが、その代わり速度の速いメモリ素子を用いて構成します。頻繁に使用する命令語やデータのコピーを一時的にキャッシュメモリに格納し、プロセッサコアからは、通常、メモリではなくキャッシュメモリにアクセスすることで、メモリアクセス性能を向上します。

キャッシュメモリをどのように制御するか、また、キャッシュメモリとメモリとをどのようなネットワークで接続するか、によって、他のプロセッサコアが変更したデータをいつ読めるようになるのか、あるいはどんな順序で読めるのか、も影響を受けます。プログラムの側から見れば、書き込んだ順に他のプロセッサコアで読めるのが最も望ましいのですが、決められたある順序を守るように制約を課すことは、概して高速化を妨げることになります。同時に処理することで高速化するのですから、順序化はそれとは逆の行為です。一方、順序制約を緩和しすぎて、やりたいことがプログラムとして表現できないということになれば、それはそれでコンピュータとして使い物になりません。そこで、高速化と正しいプログラムが書けることとの間での調整が必要になります。それを規定したものがメモリコンシステンシモデルです。

並列コンピュータのメモリアーキテクチャの発展の過程で、いくつかのメモリコンシステンシモデルが提唱されてきました。モデルとしては、ハードウェアの詳細は隠蔽して、結果的にどのような順序制約が緩和されたり追加されたりしているかのみを提示しています。ハードウェアの理解がなくてもそのモデルで定義されたルールに従えば良いのですが、逆に、そのルールの根拠が示されないので、理解しづらいものとなってしまっているとも言えます。以降、代表的なメモリコンシステンシモデルについて説明します。

## 4. Total Store Ordering

Intel社のx-86アーキテクチャで採用されているモデルがTotal Store Ordering（TSO）[4]です。古くからある命令セットアーキテクチャを基に並列コンピュータを構成する際に開発したもので、並列コンピュータとしては比較的速い時期に開発されたためか、ライトスルー方式のキャッシュアーキテクチャを色濃く反映したモデルとなっています。

あるプロセッサコアが書き込んだ値を他のプロセッサコアが読み出せるようになるまでに遅れが生じることを許していますが、最大の特徴は、モデル名が示す通り、あるプロセッサコアが行なった書き込みの順序は、そのままの順序で他のプロセッサコアに観測される点です。例えば、プロセッサコアAで変数xに3を書き込み、その後、変数yに4を書き込んだとします。さらにその後、別のプロセッサコアBが変数xを読み出す場合、3を書き込む前の値が読める（3はまだ読めない）かもしれません。一方、変数yを読み出して4が読めたのなら、その前に書き込まれたxの値は3が読めることが保証されます。

ライトスルー方式では、プロセッサコアがメモリに書き込む場合、実際にはキャッシュメモリとメモリの両方に書き込みます。しかし、書き込みを行うたびにメモリにアクセスしていたのでは高速化できません。そこで、一般的には、ライトバッファをキャッシュメモリに併設して、メモリに直接書き込む代わりに、ライトバッファに書き込みます。ライトバッファに書き込まれた値は、後で、書き込まれた順にメモリに書き込みます。従って、他のプロセッサコアからは、書き込まれた順番にその結果が読み出せるのです。TSOは、このようなライトバッファの存在に対応して考案されたモデルと言えます。

性能の点では、ライトスルー方式よりもコピーバック方式のキャッシュメモリの方が一般

的に優れているのですが、1台のプロセッサコアで行なった書き込みの順序が保存されるため、プログラムはまだ書き易いと言えます。

## 5. Partial Store Ordering

コピーバック方式（つまりはライトスルー方式ではない）のキャッシュメモリの場合や、ストアユニットを複数持つスーパスカラプロセッサの場合に自然に対応するモデルがPartial Store Ordering（PSO）です。コピーバック方式では、ライトバッファを用いません（ライトバッファが不要です）。ライトバッファがあることで、書き込みの順序をプログラムの通りに保存できたのですが、それがないため、書き込みの順序は保存できません。また、ストアユニットを複数設けるのは、キャッシュメモリに同時に複数のデータを書き込むのが目的です。しかし、この場合は、キャッシュメモリに書き込む時点で、すでに、プログラムの順序とは異なっていると考えられます。PSOは、このような特質を反映して考案されました。

TSOでは、1台のプロセッサコアで行なったすべての書き込みに対してその順序が保存されますが、PSOでは、同じアドレスに対して行った書き込みの順序のみが保存されます。異なるアドレスに対して行った書き込みは、プログラムの順序とは異なって観測される、という点に注意する必要があります。

再び、図1のプログラム例で考えてみましょう。今回はスレッドAとBがクリティカルセクションに入る際に競合せず、スレッドAが先に関数foo_A()の実行を終了したとします。関数foo_A()の中で、スレッドAは、まず、変数aに1を書き込み、次に変数countに1を書き込み、最後にaに0を書き込みます。その後、スレッドBがfoo_B()を実行します。この時点で変数aの値が0であることが観測できると、クリティカルセクションに入ってcountをインクリメントします。その結果、countの値が2になることを期待するのが普通でしょう。実際、TSOであったなら、期待通りの結果が得られます。しかし、PSOではそうとは限りません。最終的なcountの値は、2かもしれませんが、1かもしれないのです。変数aとcountは異なる変数ですので、スレッドAでの書き込みの順序はスレッドBからは異なって観測される可能性があります。つまり、8行目の結果として13行目で変数aの値が0と読めたからといって、14行目でcountの値を読み出すときに7行目の結果が読み出せるとは限りません。

しかし、これでは期待した通りのプログラムを書くことはできません。そこで、メモリアクセスの順序に制限を加える命令が用意され、必要なときのみその命令を用いてメモリアクセスの順序を制御できるようになっています。図1のプログラムでは、7行目と8行目の間、また14行目と15行目の間に、メモリバリア（またはメモリフェンス）と呼ぶ操作を行う命令を挿入する必要があります。これにより、メモリバリアの前に記述されたメモリアクセスが完了してから、メモリバリアの後に記述されたメモリアクセスを開始します。このようにメモリバリアを挿入することにより、スレッドAが変数countに1を書き込んだ後に、変数aに0を書き込んだことが（他のプロセッサコア上で実行されている）スレッドBからも観測できるようになります。つまり、変数aの値として0が読めたなら、変数countの値は必ず1が読めるようにできます。

メモリアクセスの順序に関する制約が緩和されるほど、高速にプログラムを実行できる可能性が高まります。しかし、その一方で、そのようなハードウェアの特質を最大限に活かすためには、プログラムを作成する際に上記のような注意が必要になります。メモリバリアに関する命令やメモリバリア機能の種類は、プロセッサの種類（アーキテクチャ）によって異なりますので、アセンブリ言語との組み合わせでプログラムを記述する必要があります。しかし、例えばgccの場合は、組み込み関数 __sync_synchronize()を呼び出すことで、一般的なメモリバリア操作を実行することができます。

## 6. Weak Consistency

PSOではメモリバリアなどの同期命令が必要になりますが、それも含めてメモリコンシステンシモデルとして定義した方がスッキリしま

す。ということで、メモリへのアクセスは読み出し / 書き込みだけでなく、同期も含めてその順序制約を考えます。そのようにして考案されたのが Weak Consistency（WC）です。WC は、IBM 社の POWER/PowerPC で採用されています。

WC での同期命令には、メモリバリア操作を行う命令のほか、同期機能を伴うメモリの読み出しや書き込みを行う命令（さらには演算を行うものもあります）も含めて考えます。実際にどのような命令が用意されているかはプロセッサのアーキテクチャによって異なります。基本的には、クリティカルセクションの入口と出口で同期命令を実行するものとして考えます。

WC のルールは次のようなものです。まず、(i) プログラム上で同期命令よりも前に記述された読み出しや書き込みは、その同期命令を実行する前に完了しなければなりません。また、(ii) 同期命令よりも後に記述された読み出しや書き込みは、その同期命令が完了してから開始しなければなりません。そして、(iii) 同期命令同士は、プログラムに記述された順序で実行(完了)しなければなりません。異なるプロセッサコアで実行される同期命令の順序はどれが先でも構いませんが、全てのプロセッサコアにおいて同じ順序で観測される必要があります。ある同期命令から次の同期命令までの間に行われる読み出しや書き込みの順序には一切の制約がありませんので､それらの実行（や完了 / 観測）の順序が入れ替わっても問題ありません。

プロセッサコアのハードウェア機構と WC との関係は、PSO の場合とあまり変わりはありません。図 1 のプログラムを例にしたメモリバリアの話は、WC でもそのまま当てはまります。これはクリティカルセクションの出口での処理についての話であり、上記のルール（i）に関連します。

さて、メモリアクセスの順序が入れ替わる原因は、キャッシュメモリの制御方式だけではありません。例えば、高速化のためのデータの先読み（プリフェッチ）や、スーパスカラでの命令の動的スケジューリングによって命令の実行順序が入れ替わる場合、があります。再び、図 1 のプログラムを用いてその様子を見てみましょう。スレッド A が先にクリティカルセクションに入り、変数 count の値を 1 にしてメモリバリア命令を実行した後に、変数 a の値を 0 にします。その少し後にスレッドBがクリティカルセクションに入る場合を考えます。スレッドＢは変数 a の値が 0 であることを確認してからクリティカルセクションに入るので、14 行目で変数 count を読み出すと 1 が読めるはずです。ところが、データの先読みを行なっていると、変数 a の値を読むよりも先に count の値を読み出しているかもしれません。その場合、スレッド A が 1 を書き込む以前の値を読んでしまっていることになります。従って、ここでもメモリバリアが必要になります。つまり、6 行目と 7 行目の間、また 13 行目と 14 行目の間に、メモリバリア命令を挿入して、データの先読みを抑止する必要があります。これは先のルール（ii）に関連します。

これまで見てきたように、プロセッサの高速化を実現するアイデアを実装すると、メモリアクセスの順序がプログラムで記述された順序と異なってしまうことが起こり得ます。言い換えれば、プログラムで記述されたメモリアクセスの順序をそのまま守るのではなく、順序制約を緩和することでプロセッサを高速化していると言えます。どのような順序制約が残されているかを、ハードウェア機構の詳細を隠してモデル化したものがメモリコンシステンシモデルです。

高速な並列プログラムを書こうとするなら、メモリコンシステンシモデルの理解は必須です。

## 7. 実例 - 二分探索木の並列生成

一般に、クリティカルセクションに入る前にはロックをかけ、クリティカルセクションを出るときにロックを解放することで、排他制御を行います。きちんとロックをかけるプログラムを書く場合には、メモリコンシステンシモデルを意識する必要はありません。とは言っても、もちろん、メモリバリア命令などが不要なわけではありません。ロック関数やアンロック関数の中で必要な同期命令を実行するように書かれているために、それらの関数を使用してアプリケーションプログラムを作成する場合には、同期命令を意識する必要がないだけです。逆に、

ロック / アンロック関数を作成するシステムプログラマには、メモリコンシステンシモデルの正確な理解が要求されます。

さて、ここでは、二分探索木を並列に生成するプログラムを考えてみます。ただし、プログラム実装の詳細にはあまりふれずに、実行時間における効果を示すだけにとどめます。

まず、図2に二分探索木のノードの定義（Node）と、そのノード内のリンクの初期化関数init_node_link()を示します。ここでは、ノードのキーはint型で定義しています（8行目）。各ノードには、左右の子ノードを指すリンク（Link）が必要です（10行目）。1つのリンクは、子ノードを指すポインタとロック変数を対にして定義します（3～6行目）。

変数treeは二分探索木の根ノードを指すリンクです（12行目）。ポインタはNULL、ロック変数はアンロックされている状態を示す値（ここではLockInitValとします）で、それぞれ初期化します。また、関数init_node_link()は、引数に与えられたノードの左右のリンクを初期化する関数です。

二分探索木の並列生成アルゴリズムは、逐次アルゴリズムから簡単に導出できます。図3は、きちんとロックをかけて二分探索木を生成するプログラムを示しています。各スレッドは関数insert()を実行して、その引数ndが指す新しいノードを二分探索木に挿入します。関数insert()では、根ノードから葉ノードに向かって、ノードの挿入場所を探しながら二分探索木中のノードを順にたどります。変数curが、現在立ち寄っている二分探索木中のノードを指すリンクです。そのノードを直接指すポインタとして、変数tを用いています。

複数のスレッドが同時に同じ場所にノードを追加しようとするかもしれませんので、そのポインタへの書き込みには排他制御が必要になります。また、あるスレッドが読み出そうとするポインタに、別のスレッドが書き込もう（ノー

```
01: typedef struct node  Node;
02: typedef struct link  Link;
03: struct link {
04:   Node    *ptr;
05:   Lock_t  lock;
06: };
07: struct node {
08:   int   key;
09:   …
10:   Link  left, right;
11: };

12: Link  tree
          = { NULL, LockInitVal };

13: void init_node_link(Node *nd)
14: {
15:   nd->left.ptr = NULL;
16:   nd->left.lock = LockInitVal;
17:   nd->right.ptr = NULL;
18:   nd->right.lock = LockInitVal;
19: }
```

**図2　ノード定義と初期化関数（C言語）**

```
01: void insert(Node *nd)
02: {
03:   Link  *cur = &tree;
04:   Node  *t;

05:   while(1) {
06:     Lock(&(cur->lock));
07:     if( (t=cur->ptr) == NULL )  {
08:       init_node_link(nd);
09:       cur->ptr = nd;
10:       Unlock(&(cur->lock));
11:       return;
12:     } else {
13:       Unlock(&(cur->lock));
14:     }
15:     if( nd->key < t->key )
16:       cur = &(t->left);
17:     else
18:       cur = &(t->right);
19:   }
20: }
```

**図3　並列ノード挿入プログラム（C言語）**

ドを追加しよう）としているかもしれませんから、書き込み同士だけではなく、読み出しと書き込みの間でも排他制御が必要です。従って、リンクを辿るときにはまずロックを掛けて（6行目）、ポインタを読み出します（7行目）。それがNULLであれば、そこが挿入場所ですから、新たに挿入するノードのリンク欄を初期化して（8行目）、ノードを追加します（9行目）。これで挿入を完了しましたから、ロックを解放して（10行目）、関数insert()での処理を終了します（11行目）。

まずは正しい並列プログラムを書く、という観点では、図3のプログラムで良しとするべきでしょう。しかし、後で述べますが、このプログラムでは期待するほど高速に二分探索木を生成できません。むしろその結果にがっかりしてしまって、これが最初の並列プログラミング体験だったとしたら、並列処理などやる価値がないと思ってしまうかもしれません。

これを高速化したプログラムは、図3の6～14行目を図4の2～13行目で置き換えることによって得られます。ノードの追加は、値がNULLのポインタを、追加するノードへのポインタ値で置き換えることによって行います。NULLでないポインタを書き換えることは行いません。アルゴリズム上のこのような特徴を利用して、まずはロックを掛けずに、いま立ち寄るノードへのポインタ(cur->ptr)を読み出し(2行目)、それがNULLでなければそのノードをたどります（図3の15行目に進みます)。一方、NULLであった場合には、そこが挿入場所である可能性がありますので、この時点でロックを掛け（3行目)、そののちに再度、cur->ptrの値を読み直します（4行目)。

4行目ではマクロMR()を用いて読み直していますが、その定義は、1行目に示すように、一見しただけでは非常に理解しづらい型にキャストする（だけの）ものです。cur->ptrの値は2行目で既に読み出していますので、コンパイラが、4行目ではメモリから読み出さずに2行目で読み出した値を使うようにする可能性があります。しかし、2行目ではcur->ptrはNULLであったものの、4行目に進むまでの間に他のスレッドが書き換えているかもしれませんから、4行目でもNULLであるとは限りません。そこで、コンパイラの最適化機能を制止して、再度、cur->ptrの値をメモリから（実際にはキャッシュメモリからかもしれませんが）読み出すようにするために、volatile修飾子をつけてキャストしています。

ところで、ロックを掛けて実行しなければならないクリティカルセクションの中でアクセスするデータに、ロックを掛けずにアクセスすることを許すのは、本来はやってはいけないことです。しかし、そのやってはいけないことを図4のプログラムではやるのですから、それによってどのような不具合が生じ得るかをすべて把握し、対策できていなければなりません。

例えば、図4では、6行目でメモリバリア操作を行っています。TSOを用いているx86系のコンピュータでこのプログラムを実行する場合には、ここでのメモリバリアは不要（冗長）です。しかし、メモリコンシステンシモデルにPSOやWCを採用しているコンピュータで実行する場合は必須となります。

PSOやWCのコンピュータで、6行目が欠落したプログラムを実行した場合にどのようなことが起こり得るかを考えてみましょう。スレッドAが、ノードXを二分探索木中のノー

```
01: #define MR(A)  ¥¥
          (*((Node * volatile *)(A)))

02: if( (t = cur->ptr) == NULL )  {
03:   Lock(&(cur->lock));
04:   if( (t = MR(cur->ptr))
                          == NULL ) {
05:     init_node_link(nd);
06:     __sync_synchronize();
07:     cur->ptr = nd;
08:     Unlock(&(cur->lock));
09:     return;
10:   } else {
11:     Unlock(&(cur->lock));
12:   }
13: }
```

**図4　高速なノード挿入プログラム（C言語）**

ドＹの子ノードとして追加するとします。スレッドＡはノードＸ内のリンクを初期化して（5行目）から、ノードＹにノードＸを追加（7行目）します。別のスレッドＢはロックを掛けずに、ノードＹからノードＸへとたどってくるかもしれません。そしてさらに、ノードＸの子ノードへと進むでしょう。プログラム上ではノードＸ内のリンクを初期化した後にノードＹからノードＸへリンクを貼っていますが、スレッドＢからは、ノードＹからノードＸへとたどった後でも、ノードＸ内のリンクは、初期化する前の値がまだ見えているかもしれません。その結果、スレッドＢは誤った場所をたどろうとするか、あるいはセグメント違反を起こして異常終了してしまうことになります。このようなことを防ぐために、クリティカルセクションの中であるにもかかわらず、メモリバリア操作を6行目で行う必要があります。

最後に、図3のプログラムと図4のプログラムで、性能がどのように異なるかを見てみます。12コア24スレッドのインテルXeonプロセッサで実行した結果［5］を図5に示します。横軸がスレッド数を、縦軸が1スレッドで実行した場合に比べて何倍速いかを示しています。M3とS3が図3のプログラム、M4とS4が図4のプログラムの結果です。M3とM4ではロックにサスペンドロック（Mutex）を、S3とS4ではスピンロックを用いました。測定条件などの詳細は省きますが、ノード数が1万個の二分探索木を生成する場合（図5（a））には、図3のプログラムでは速くなるどころか、逆に1スレッドで実行するよりも遅くなっています。1億ノードの場合（図5（b））はスレッド数を増やすと性能は向上しますが、それでも図4のプログラムと比べると半分以下の性能です。

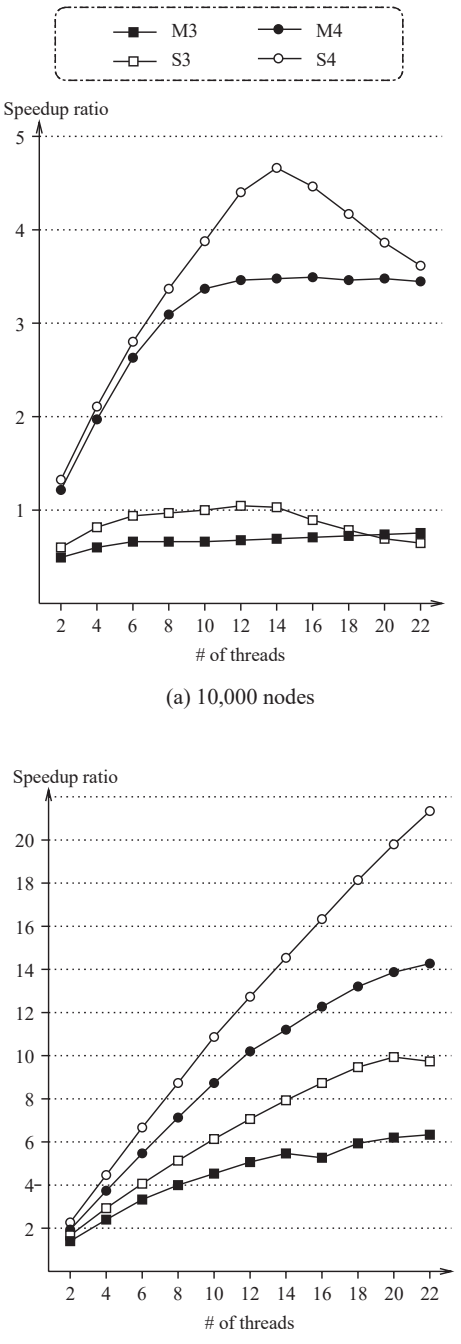


**図5　性能比較**

## 8. おわりに

メモリコンシステンシモデルについて、並列プログラミングの観点から解説しました。実は、本稿では紹介しませんでしたが、WCの順序制約をさらに緩和したモデルとして、Release Consistency（RC）［6］が考案されており、例えばARM（v8）やRISC-Vなど、比較的最近に開発されたプロセッサアーキテクチャはRCを採用しています。

期待通りに安定して動かすためには、あまりアグレッシブな並列プログラムは書かない方がよいのですが、それでは並列コンピュータの性能を十分に引き出すことはできません。一方、プログラム実行時に生じる種々の動作タイミングの組み合わせで何が起こり得るかを正確に把握して、それをプログラム中で対処・制御するのは容易ではありません。私もハイゼンバグに

悩まされることもあれば、バグの原因の見当をつけようと真剣に考える過程で、あまりにもややこしくて脳ミソがとろけそうな感覚を覚えることもありますが、それでも絶望的な状況に陥らないのは、やはり、ハードウェアの動作やメモリコンシステンシモデルの理解が助けになっているのだろうと感じています。

**参考文献**


[1] H. Hennessy, and D. A. Patterson. “Computer Architecture – A Quantitative Approach（6th Edition）.” Section 5.6, Morgan Kaufmann Publishers (2019).

[2] 天野英晴著「並列コンピュータ」第3章, 昭晃堂 (1996).

[3] D. J. Sorin, M. D. Hill, and D. A. Wood. “A Primer on Memory Consistency and Cache Coherence.” Morgan & Claypool Publishers (2012).

[4] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors.” Communications of the ACM, vol. 53, no. 7, pp. 89–97 (2010).

[5] H. Hirata, and A. Nunome. “Parallel Binary Search Tree Construction Inspired by Thread-Level Speculation.” Proceedings of the 23rd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 74–81 (2022).

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.” Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15–26 (1990).