3. 共有メモリを用いた並列プログラム

3.1 並列プロセスの fork-join

では共有メモリはプログラマから見るとどのように見えるだろうか？典型的な方法は、並列に動作するプロセス（スレッド）を想定し、それらの間で変数が共有されているモデルである。まずプログラマは単一のプログラムを想定し、ある処理を実行することで、複数の並行に動作するプロセスを生成する。この操作を fork と呼ぶ。このプロセスは一つのプロセッサ上で疑似的に並列実行される場合もあるが、マルチコアでは、実際に別々のコアに割り当てられて並列実行される。fork されたプロセスは、自分のプロセス id を取得し、その値に応じて個別の処理を行うことができる。プロセス間で変数は共有され、これを通じてデータのやり取りを行う。図 1 に示すように fork されたプロセスは、さらに fork して子プロセスを生成することができる。一定の処理が終了すると、複数のプロセスは一つにまとまり、最終的には単一のプロセスとなって全体の処理を終了する。この操作を join と呼ぶ。全てのプロセスが終了しなければ、join は行われないため、この操作は一種の同期の役割を果たすことができる。簡単な並列プログラムは fork-join の繰り返しで記述することができる。

3.2　不可分命令と Critical Section の実行

3.2.1 最も単純な不可分命令 Test&Set(x)

共有メモリを用いたデータ転送の問題は、他のプロセスあるいはプロセッサがいつ特定の変数＝特定の共有メモリアドレスに書き込むか分からない点にある。書き込むプロセッサが決まっている場合、あとはその値が有効かどうかを示すためのビット（フラグと呼ぶ）を設けてハンドシェークを行えば良い。しかし、場合によっては一つのプロセッサを選んで、それが排他的に変数を扱う必要がある。

単純な共有メモリを用いて排他を行うのは結構難しい。例えばある変数 x を 0 に初期化しておき、これを読み込んだプロセッサが即座に 1 を書き込み、0 を読み込んだプロセッサを勝者とすれば良いのではないか、と考えるかもしれない。しかし、一般的な共有メモリでは、0 を読み込んでから 1 を書き込むまでの間に、他のプロセッサが 0 を読み込む可能性がないとはいえず、単純な読み出しと書き込みでは、排他制御を保証することができない。この様子を図 2 a)に示す。すなわち、読み出しと書き込みが不可分（indivisible あるいは atomic）に行われる必要がある。これを保証するのが不可分命令で、最も単純な Test&Set（x）は、図 2b)に示す通り、x を読み出して 0 ならば 1 を書き込む操作を不可分に行う。Test&Set(x)命令を複数のプロセッサが実行した場合でも、0 を読み出すことのできるプロセッサは 1 つに限られる。この選ばれたプロセッサは、図 3 に示す通り、他と競合する資源を扱う。これは、変数に対する書き込み、占有して用いる入出力装置の読み書きなどで、ただ一つのプロセッサでのみ扱える操作を Critical Section（際どい領域）と呼ぶ。Critical Section を実行した後に、 x に 0 を書き込む。このことにより、他のプロセッサの Test&Set(x)は成功し、Critical Section を実行可能となる。

3.2.2 様々な不可分命令

不可分命令の本質は、読み出しと書き込みを一連の操作として他から邪魔されずに行うことにある。このため、用途に応じて様々な命令が使われる。代表的なものを紹介する。

○Swap(x,y): 共有メモリ領域の x とローカルメモリ領域の y を交換する。Compare & Swap はこれに比較の操作が入り、比較の結果に従って交換するかどうかを決める。

○Fetch&Dec(x):共有メモリ領域の x を読み出して 1 引いた値を書き戻す。複数プロセッサに番号を割り当てるのに便利である。x が 0 になったら引き算を行わない場合（飽和型）もある。1 足した値を書き戻す Fetch&Inc(x)もある。

○Fetch&Add(x,y):共有メモリ領域の x を読み出して y を加算して書き戻す。Fetch&Dec と Fetch&Inc の一般形である。

○And - write(x,y):共有メモリ領域の x 読み出して y の論理積を取って書き戻す。論理和を行う Or - write もある。共有メモリ上のビットマップを扱うのに便利。

○Fetch&ϕ（x ,y）：今までに挙げたすべての不可分命令の一般形。ϕ は操作を示す。これが交換ならば Swap,加算ならば Fetch&Add、論理積ならば And - write になる。

○Load Linked と Store Conditional。Load Linked は特定のアドレスから値を読み込むと共にそのアドレスを記録する。次に同じアドレスに対して Store Conditional を実行すると、Load Linked した後に、他のプロセッサが書き込みをするか、コンテキストスイッチが起きると、値の書き込みは失敗し 0 が返る。そうでなければ書き込みは成功し 1 が返る。Store Conditional に成功するかどうかで排他制御の実装は可能で、他の方式よりも実装上の効率が良い。Hennessy&Patterson のテキストで取り上げられて有名になった。

例題１：あるプロセッサ A が n 台のプロセッサにデータを転送したい。Fetch&Dec（x）を用いて、n 台全てが受け取ったことを確認したい。どのようにすれば良いか？

例題２：Test&Set（x）を用いて Fetch&Add(x,y)を実装せよ。

3.2.4 セマフォ

共有メモリ型の並列コンピュータにおけるプロセッサ間の同期は、OS によるプロセス間の同期と類似している。これはユニプロセッサであっても並行プロセスはどのタイミングで切り替わるか分からないので、操作の間コンテキストスイッチを起こさない命令が必要になる。セマフォは、P（ウェイト）命令と V（シグナル）命令から成り、それぞれ以下の機能を持つ。

P(s):s＝1 になるのを待ち、なったら不可分に 0 にする。

S(s): s=0 になるのを待ち、なったら不可分に 1 にする。

セマフォは、今まで紹介した不可分命令とほぼ同じ機能を持っている。上記は 0 と 1 からなるバイナリセマフォだが、カウンタを＋１、－１するカウンティングセマフォも使われ、これは Fetch&Inc, Fetch&Dec と類似する。

3.2.3 不可分命令の実装

読み出しと書き込みを他から妨害されないように行うためには、一連のバストランザクションで行いマスタ権を握りっぱなしにするなどの工夫が必要となる。いずれにせよ、不可分命令は、複数のプロセッサが Critical Section の実行権を取るまでに繰り返し実行する傾向（図 3 のビジーウェイティング）があり、共有バスの利用率が高くなってしまう。これを防ぐため、不可分命令の実行前に一度共有変数を読み出して事前チェックを行う。例えば、Test&Set(x)を実行する前に、まず x を読み出し、0 であることを確認する。 x が 1 であれば、Test&Set が成功する見込みはないので、読み出しを続ける。この読み出しは、全てスヌープキャッシュ上でヒットし、共有メモリを使わない。Critical Section の実行を終えたプロセッサが x に 1 を書き込むと、スヌープキャッシュ上の値が無効化され、読み出しがミスヒットし、共有バスを使って 1 がキャッシュまでコピーされる。ここで 1 になっていることを確認して、Test&Set を実行する。この時は共有バスが使われ、どれかのプロセッサのみが選ばれて x ＝0 を取得する。この方法を Test-and-Test&Set と呼ぶ。

Load Linked と Store Conditional は、各プロセッサにリンクレジスタを設け、スヌープキャッシュ同様、Store Conditional の実行時にアドレスをバスに流してやり、リンクレジスタを無効化することで、効率的な実装が可能である。

3.3　バリア同期

3.3.1 単純バリア

バリア同期は、関連する全てのプロセッサが足並みを揃えて待ち合わせをする同期方式である。図 4 に示すように、あるプロセッサは、バリア命令を実行すると、他のすべてのプロセッサが実行するまで先に進むことができない。全部のプロセッサが実行すると、バリアが成立し、先に進むことができる。実際の並列プログラムでは、それぞれのプロセッサが計算を行った結果を共有メモリに書き込み、他のプロセッサの結果を使って次の反復で同様の計算をする場合が多い。この際、自分の結果を書き込んでからバリア命令を実行すれば、バリアが成立した時には、次の反復で利用するデータが全て有効になっており、安心して次の反復に進むことができる。この方法は、遅いプロセッサの実行に全てが律速されるが、多くの場合には手軽で効率の良い同期が実現できる。バリア同期は不可分命令を用いて実装ることが可能だが、簡単なハードウェアで実現可能なので、デバッグ用に装備しておくのも便利である。

3.3.2 Fuzzy Barrier

単純なバリアは全部のプロセッサがバリア命令を実行すれば成立する。これを二つの段階に分離して、待ち時間を減らす方法を Fuzzy Barrier と呼ぶ。この方法では、自分の結果を全て書き込んだ時に Prepare(準備)命令を実行する。次に自分が他のプロセッサの計算し

た結果を読み出す直前に Synchronize（同期）命令を実行する。図 5 に示す通り、全てのプロセッサが Prepare または Synchronize を実行するとバリアは成立する。この手法は、自分の計算結果を書き込んでから、次の処理に移って他のプロセッサの処理した結果を利用するまでに一定の処理を行う場合に、より柔軟にバリアが成立することから進むことから効率が上がる。類似の方式に Elastic Barrier がある。単純バリアを線バリアと呼んで、この種のバリアを面バリアと呼ぶ場合もある。プロセッサのグループを定義して、そのメンバ内にのみバリアの成立を考えるグループバリアも用いられる。

3.4　OpenMP による並列プログラム

では最も簡単な並列プログラムの枠組みである OpenMP を用いてみよう。

3.4.1 OpenMP の指示文

OpenMP は C 言語、C++言語で書かれたプログラムに指示文（pragma, directive）を付けることにより並列化を明示的に指定する並列プログラムの方式である。指示文で並列に生成されたプロセス間は、基本的に全ての変数を共有し、これを介してデータをやり取りする。OpenMP は、gcc のライブラリとして標準的に装備されており、ほとんどの環境で利用可能である。ここでは簡単にこの利用法を紹介する。詳細は XX を参照されたい。

OpenMP の基本実行モデルを図 7 に示す。並列実行のための指示文は以下のとおりである。

```
# pragma omp parallel
{    … }
```

カッコで囲まれた BLOCK B は図 7 に示すように fork されて並列に実行され、終了後、join される。この間変数は共有される。実際にこの#pragma omp parallel の中で記述されるのは、基本的に同じ動作を並列に行う for 文、do 文と、異なった動作を行う section 文である。最も一般的に用いられる for 文の例を図 8 に示す。 # pragma omp parallel と#pragma omp for をまとめて、#pragma omp parallel for と書くことができ、実際上はこちらが良く使われる。指示文で指定された for 文のそれぞれの反復は、それが並列処理可能ならば、現在利用可能なプロセッサ数に均等に割り当てられて実行される。for 文では、基本的には同一の処理が各プロセッサで実行されるのに対して、section 文は、図 9 に示すようにそれぞれの関数（サブルーチン）を並列に実行する。

OpenMP では指示文に付け加えて補助的に指示を与える補助指示文（sub-directive）が用意されている。最も一般的なのは private sub-directive で、指定する変数をローカルに持たせる。図 10 の例では c はそれぞれのプロセッサにコピーされるので、高速に実行される。図 11 の例では、2 重ループの中の j を private 指定している。この指定により j はそれぞれのプロセッサでコピーされ独立にカウントアップされる。この指定がないと、複数のプロセッサがそれぞれに同じ j をカウントアップするため、プログラムが正しく実行されない。

リダクション補助指示文（reduction sub-directive）は、配列の要素すべての総和を取る場合などのリダクション演算を行う指示文である。リダクション演算は配列を並列に動作す

るプロセッサに分散して演算し、答えを集めて一つにする必要があり、並列処理の操作が面倒である。reduction sub-directive は図 12 に示すように演算子と答えの入る変数を指定することで、これを自動的に行ってくれる。

3.4.2　OpenMP の実行

　OpenMP の指示文を使うプログラムは#include <omp.h>を指定する必要がある。これにより、いくつかの組み込み関数が使える。omp_get_num_threads();は現在実行可能なスレッド数を返す。omp_get_thread_num ()；は自分のスレッド番号を返す。この値によって処理を変えることができる。また omp_get_wtime();は double で実行時刻を返してくれるので、チューニングに用いることができる。コンパイルには、-fopenmp オプションを付け、

gcc -fopenmp XXX.c -o XXX

のようにすればよい。

通常、Linux 環境ではその PC で実行可能なプロセッサ数で並列実行が行われるが、その数は環境変数 OMP_NUM_THREADS で制御することができる。例えば

export OMP_NUM_THREADS=8

などで実行するプロセッサ数を変えることができる。もちろん、実際に存在するプロセッサ数よりも多く指定しても性能は上がらない。