第 3 回 並列ゼミ

ゼミ担当者　：　真武信和，山本啓二，坂田大輔
指導院生　　：　下坂久司，輪湖純也，冨岡弘志
開催日　　：　2003 年 5 月 30 日

ゼミ内容: 本ゼミでは，スレッドレベルの並列処理についての説明を行う．まずスレッドレベルの並列化とは何かについて触れ，その後並列化の原理について説明する．そして最後にこれまでのまとめを行う．

1　スレッドレベルの並列化

1.1　スレッドとは

前回までの並列処理は，複数のプロセスを並列で処理させていた．1 つのプロセスは 1 つのプログラムを扱い，プロセッサがプロセス単位で処理を行う．またマルチプロセスプログラムでは，プロセスごとにメモリ空間が独立しているため，プロセス間でデータを送受信する場合には，MPI 等で通信を行う必要があった．

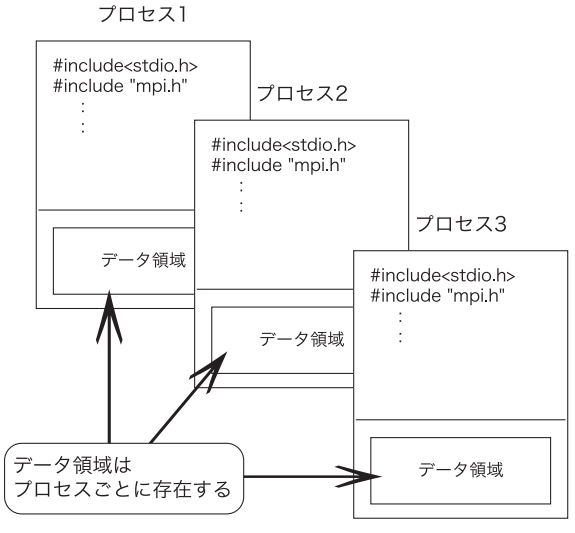

Fig. 1　マルチプロセス

一方，マルチスレッドプログラムでは，スレッドと呼ばれる実行単位で処理を行う．スレッドとは，プロセスの中に生成された実行単位である．スレッドを作ることにより，複数の CPU で 1 つのプロセスを実行することができる．スレッドは同一のプロセス内で動作するため，スレッド間でアドレス空間を共有している．プロセスがプログラム単位で処理を行っていたのに対し，スレッドでは関数単位での並列処理が可能となっている．

1.2　スレッドプログラミング

スレッドによる並列処理を行う場合，様々なライブラリを使用する．その中で代表的なものとして，“OpenMP” と “Pthread” がある．

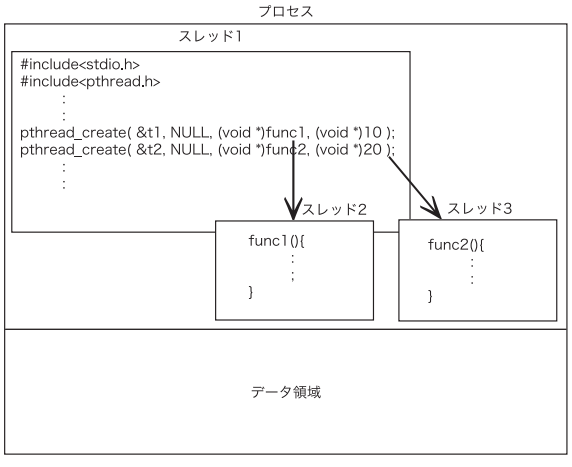

Fig. 2　マルチスレッド

- OpenMP
  共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデル．ベース言語（Fortran/C/C++）を directive（指示文）で並列プログラミング用に拡張する．

- Pthread(POSIX thread)
  POSIX 1003.1c-1995 という標準に準拠した並列プログラミングの規格．スレッドの生成や消去などをプログラム中で直接記述する．

1.3　MPI との違い

プロセスを生成する場合，新しくアドレス空間と，内部で使用するデータ構造を準備する必要があるが，スレッドの場合はアドレス空間を共有しているため OS への負荷が小さくなる．また，メモリを共有しているため，MPI 等でデータを送受信する必要も無くなる．

しかし，メモリが共有されていることにより，あるスレッドによるデータの変更が，他のスレッドに直接影響を及ぼす．このことにより，他のスレッドがデータの変更を行ったことにより，スレッド全体の処理がおかしくなってしまう可能性がある．また，異なるスレッドが同じデータにアクセスすることにより，計算結果がおかし

くなると言ったことも生じる．

1.4 スレッドレベルの並列化の注意点

1.4.1 競合 (race condition)

スレッドレベルでの並列化を行う場合，データが共有されているため注意する必要がある．例えば「変数 x（初期値は 100）にスレッド 1 が 10 を加え，スレッド 2 が 20 を加える」という処理を考える．通常ならば，x にスレッド 1 が 10 を加え 110 となり，110 にスレッド 2 が 20 を加えるため，x の値は最終的に 130 になる．

値を加える際には以下の 3 ステップに分けられる．

- 変数 x の値を読み込む
- 値を加える
- 値を x に書き込む

スレッドが 2 つなので，合計 6 つのステップとなる．まず以下のように処理が行われたとする．

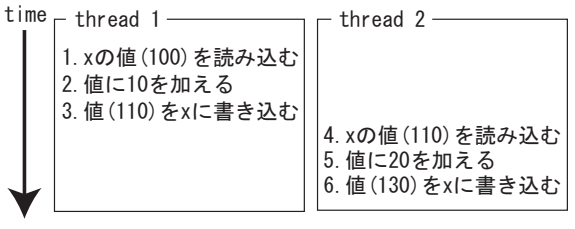

Fig. 3　処理 1

この場合 x の値は 130 となり，正しい値となる．しかし，スレッドが並列に処理を行うと，以下のような場合も起こりうる．

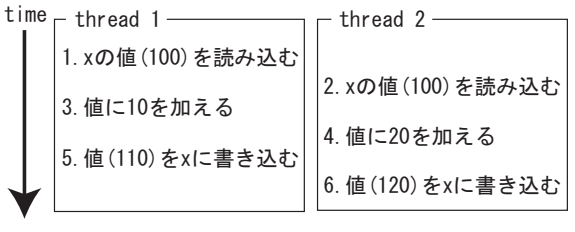

Fig. 4　処理 2

この場合 x の値は 120 となり，先ほどの結果と異なってしまう．

このような不具合を防ぐためには，処理中のデータを他のスレッドが使用できないようにするといった「排他制御」が必要となる．排他制御のアルゴリズムには，“Critical Section”，“Mutex”，“Semaphore”の 3 種類が存在する．それぞれの説明を Table 1 に示す．

Table 1　排他制御

アルゴリズム	説明
Critical Section	同一プロセス内のスレッドを対象に，排他制御を行う．
Mutex	他のプロセスやその中のスレッドも対象とした排他制御．1 つのデータに 1 つのプロセス，もしくはスレッドがアクセス可能である．
Semaphore	排他制御の対象は Mutex と同じ．1 つのデータに複数のプロセス，もしくはデータがアクセス可能になっている．

1.4.2 デッドロック (dead lock)

スレッドで排他制御を行った場合，他のスレッドがアクセスしているデータを利用したいときは，アクセス中のスレッドが利用し終えるまで待つことになる．しかし，この場合でも不具合が生じることがある．例えば「スレッド 1 が変数 x に変数 y の値を加え，スレッド 2 が変数 y に変数 x を加える」という処理を考える

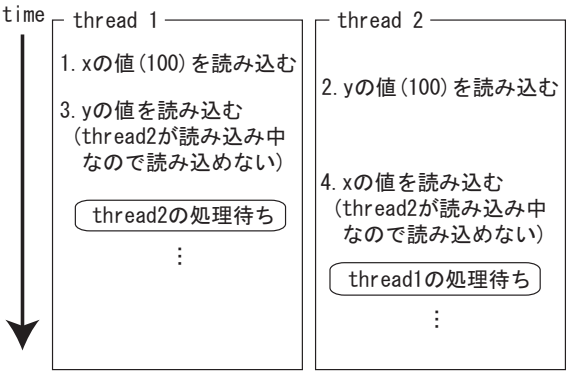

Fig. 5　処理 3

Fig. 5 のように処理が行われた場合，お互いに処理が終わるのを待ち続けることになってしまう．このような状態を「デッドロック」と呼ぶ (Fig. 6)．

x の値をスレッド 1 がアクセスしているため，x へのアクセスにロックがかかっており，スレッド 2 がアクセスしているため，y へのアクセスがロックがかかっている．このため，スレッド 1 が y にアクセスしようとしても，スレッド 2 のロックによりアクセスできないため処理待ちとなり，次の処理へ移る．続いてスレッド 2 が x にアクセスしようとするが，スレッド 1 のロックにより処理待ちとなる．このため，お互いに一方の処理が終わるのを待ち続けることになってしまう．

プログラマはデッドロックが起こらないように，ス

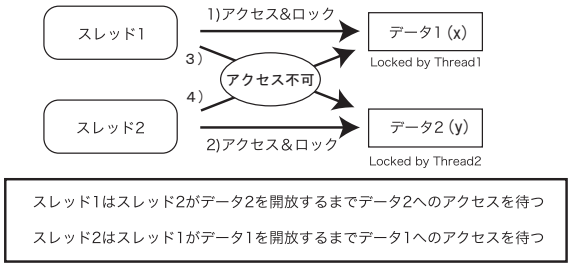

Fig. 6　デッドロック

レッドの並列処理を行えるようにプログラムする必要がある．

2　スレッドレベルの並列化の原理

2.1　並列化の流れ

スレッドレベルの並列プログラムは，基本的には以下のような流れで記述される．

1. スレッドの生成
2. 各スレッドでの処理
3. 同期
4. スレッドの消滅

スレッドの生成

スレッドの生成は，プロセスの生成よりも高速である．また各スレッドは一つのプロセス内に生成されるため，プロセスレベルの並列化の際のような「データ通信」は必要ない．データはプロセス内では共有されている．スレッドの生成時には，プロセス内で各スレッドにユニークなスレッド ID が与えられる．スレッドには，スレッドの担当する処理を記述した関数と，処理するデータの先頭のポインタが渡される．pthread ではスレッドの生成には pthread_create() を用いる．

各スレッドでの処理

各スレッドは与えられた関数に記述された処理を実行する．

同期

スレッドプログラミングはそもそも非同期で実現可能な問題のために用いられるものである．しかし実際には同期が必要となることも多い．多数のスレッド間でデータや処理の同期を取ることは，非常に複雑でプログラマに負担のかかる処理である．この同期については次節で詳しく扱う．

スレッドの消滅

pthread_create() で指定された関数からリターンすると，そ のスレッドが終了する．スレッドを終了させるには pthread_exit() を呼び出してもよい．ただし，初期スレッド が終了すると，プロセス全体が終了する．exit() システムコールを呼び出しても終了する．

2.2　スレッド間の同期

マルチスレッドプログラミングで難しいのは，複数のスレッドを協調させて動作させるために，スレッド間できちんと同期を取ることである．ここではこの同期について説明する．

2.2.1　排他制御

共有資源へのアクセスには，その領域が危険領域であるかどうかが重要である．危険領域へのアクセスを行う場合は，排他制御が必要となる．ここでは pthread での排他制御の方式を例に，排他制御の原理について説明する．

pthread では，排他制御の為に Mutex と呼ばれる同期変数を用意している．排他制御には，危険領域の入り口で Mutex のロックを行い，出口でアンロックを行う．Mutex がロックされている間は，Mutex の所有者以外のスレッドは，その領域へのロックがブロック（待機）させられる．所有者により Mutex がアンロックされると，他のスレッドが Mutex をロックできるようになる．こうすることにより，同時に Mutex をロックできるスレッドは高々1 つとなり，危険領域を保護できる．プログラマは，Mutex をロックしたスレッドだけが共有資源にアクセスするように，注意深くコーディングする必要がある．危険領域を保護するのは，プログラマの責任である．

複数の共有資源にアクセスするために，同時に複数の Mutex をロックする必要がある場合は，デッドロックが起きる可能性がある．これを避けるのもまたプログラマの責任である．共有資源の 1 つ 1 つに Mutex を使用すると，それだけ並列度は上がる．しかしそうするとロック・アンロックのコストがかかるようになるため，どの程度まとめて Mutex で保護するかは検討する必要がある．

Mutex をロックしている間，他のスレッドは危険領域に入れないため，プログラムの性能に影響を及ぼす場合がある．Mutex をロックする期間は最小限にしなければならない．

Mutex は次のように使用する．

1. 初期化
   Mutex は使用する前に初期化する必要がある．初期化には 2 通りの方法がある．動的に初期化するには pthread_mutex_init() を用いる．引数で Mutex 属性オブジェクトへのポインタで，生成する Mutex の属性を指定することができる．(Mutex の属性オブジェクトについては省略する）静的に初期化するには，PTHREAD_MUTEX_INITIALISER を

用いる．

不要になったMutexは，pthread_mutex_destroy()で破棄することができる．ただしMutexがロックされている場合は破棄することはできない．

2. ロックとアンロック

int pthread_mutex_lock(pthread_t *mutex);

→ Mutexのロック

int pthread_mutex_unlock(pthread_t *mutex);

→ Mutexのアンロック

Mutexのロック･アンロックにはpthread_mutex_lock(), pthread_mutex_unlock()を用いる．

2.3 並列化の枠組み

マルチスレッドでは，Fig. 2に示すように，プログラム内（プロセス内）で処理を並列に実行できる．シングルプロセッサではこれは時分割で実行されるが，マルチプロセッサでは実際に並列に処理が可能となる．スレッドレベルでの並列化を行う際には,「スレッド間で扱うデータに関連性が無い」ことが重要である．扱うデータが共通であったり関連性が強い場合，スレッド間で同期を取る必要が出てくるため，並列化の効率が悪くなるばかりでなく，デッドロックなどの問題が生じ，さらにはプログラマにも負担をかける．よって，マルチスレッドにすべき処理としては，次のようなものが向いていると言える．

- I/Oの処理
- ネットワークサーバ
- ユーザインタフェース

これらは入力が非同期的にやってくるため，一つのプロセス内でも他との独立性が強い．スレッドの処理の大半を占める「入力待ち」という処理は，大域的な変数への書き込みなどを一切必要としないからである．これらの処理をスレッド化してやることで，処理を並列に実行することが可能となり，マルチプロセッサにおいてはさらにその実力を発揮させることができる．

I/Oの多重化を例に挙げると，シングルスレッド環境で，あるスレッドがディスプレイに表示を行っているとき，CPUには待ち時間が発生する．しかしマルチスレッド環境においては，I/Oを扱うスレッドを多重化すると，表示を行うスレッドが待ちに入っている間でも，他のスレッドでキーボードからの文字入力やその他の処理が実行可能となる．これによりアプリケーションの性能向上の可能性が高まる．

3 まとめ

3回を通して行った並列ゼミの内容をここで簡単にまとめとして紹介する．

3.1 第1回で学んだこと

第1回では，並列処理を対象問題，アルゴリズム，ソフトウェア，通信ネットワーク，ハードウェアの各レベルに分け，問題領域からハードウェアに至るまで並列処理の概観の説明を行った．

3.1.1 対象問題

・並列処理の必要性

今や1つのCPUの高速化による計算機の高速化は限界にきている．クロック周波数が1GHzの場合，周期は1nsとなり，電気信号はその間に30cmしか進まない．集積回路の大きさや配線の基盤の寸法などを考えても，これ以上クロック周波数を飛躍的に上げることは原理的に困難である．

・Amdahlの法則

Amdahlの法則とは，全体の処理速度を並列処理の部分とそうでない部分の比率の関数として表したものである．s $(0 < s < 1)$ を全処理中逐次処理しなければならない部分の割合とし，p をプロセッサ数，$R(s)$ を総合的な速度向上率とすると，$R(s)$ は次式で表せる．

$$R(s) = \frac{1}{s + \frac{1-s}{p}} \tag{1}$$

$R(s)$：総合的な速度向上率

$s(0 < s < 1)$：全処理中逐次処理しなければならない部分の割合

p：プロセッサ数

これをグラフにすると Fig. 7 となる．

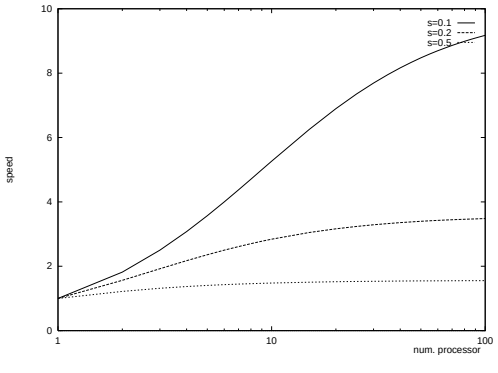

Fig. 7 Amdahlの法則

3.1.2 メッセージ通信ライブラリ

メッセージ通信は，あるプロセスのメモリ空間から，もう一方のプロセスのメモリ空間へのデータの転送と見ることができる．もちろん一般には，メッセージをやり取りするのは2つのプロセスとは限らず，もっと多くのプロセスが関与し得る．

分散メモリ型の並列計算機では，メッセージ通信によるプログラミングが最もハードウェアの構成に即したプログラミングであり，最高の性能が期待できる．

3.1.3 MPI と PVM

MPI(Message Passing Interface) とは，分散メモリ環境における並列プログラミングの標準的なメッセージ通信 API 仕様である．Message Passing とは，プロセッサ間で互いに通信してメッセージ（データ）交換を行ったり，同期を取ったりすることを指している．しかし，通信といっても様々な問題が存在する．例えば，同じ並列処理マシンでの通信や TCP/IP によるネットワーク越しの通信もある．並列処理マシンのアーキテクチャにより，通信方法が異なるために，その実装がまったく異なったものになってしまう．この部分を代行してくれるものが MPI である．

3.1.4 通信ネットワーク

Ethernet と Myrinet について説明した．

Myrinet は高性能なパケット通信及びスイッチングテクノロジーを持ち，PC クラスタのインターコネクトとして広く利用されている．Myrinet は以下のような特徴をもつ．

- 全 2 重 2Gbps でリンクされるスイッチ及びインターフェイス
- フロー制御，エラー制御，リンクのモニタ
- 低レイテンシ，カットスルー型のクロスバースイッチ
- 障害発生時の代替経路の探索
- 様々なネットワークトポロジーを構築可能

3.1.5 共有メモリ型並列コンピュータ

複数のプロセッサがメモリバス／スイッチ経由で，主記憶に接続される形態である．このアーキテクチャを有するシステムのことを SMP(Symmetric MultiProcessor) と呼ぶ．このシステムは，いうならば巨大なマザーボード上に複数個のプロセッサを取り付けた形である．

3.1.6 分散メモリ型並列コンピュータ

プロセッサと主記憶から構成されるシステムが複数個互いに接続された形態である．つまり，一連のコンピュータが内部ネットワークまたは，外部ネットワークを介して通信することであたかも一つのコンピュータのように動作する．大規模なメモリを得るシステムを組みやすい．

3.2 第 2 回で学んだこと

第 2 回では，分散メモリ環境における並列プログラミングである MPI について学んだ．プロセッサ間通信方式であるメッセージパッシング方式に触れ，MPI の実装ライブラリである MPICH を用いた実際のプログラミング方法を説明した．

3.2.1 1 対 1 通信とグループ通信

1 対 1 通信とは，メッセージパッシングにおける最も基本的な通信機能であり，一つのプロセスが送信元，相手のもう一つのプロセスが受信先になって行われる．

グループ通信とは，プロセスがグループ内での集団通信動作，例えば，一台のマシンが他のマシン全体に対してデータを送信することである．

3.2.2 ブロッキング通信とノンブロッキング通信

ブロッキング通信では，送受信の各プロセスは，通信が完了するまでそのプロセス内の他の作業を実行することができない．しかしノンブロッキング通信では，送受信の各プロセスは，通信が完了しなくても，プロセス内のほかの作業を実行することができる．この場合，通信が終了するのを待つ必要が無いため，より効率の良いプログラムを作成することができる．

3.2.3 プロセスの認識

通信を行うには，送信元と送信先を明確に識別できなければならない．送信元や送信先はプロセスであるから，各プロセスが一意に識別できる必要がある．MPI では，プロセスを「ランク」によって識別する．

3.2.4 MPICH の使用法

コンパイルは，通常 SSH（Secure SHell）で接続した相手であるマスタで行う．できた実行ファイルは，実行する全マシンに存在している必要があるので Sun が開発した NFS（Network File System）というファイル共有システムを利用する．

・マスタからスレーブへの実行の依頼

MPICH では，UNIX のシステムで用いられている “rsh” を使用してマスタからスレーブにログインし，一連の MPI プログラムのプロセスを実行している．これらの一連の作業は MPI に用意されているスクリプトである “mpirun” が行ってくれる．

3.3 今回学んだこと

今回は，スレッドを用いた並列化について OpenMP や pthread を例に挙げて説明した．

3.3.1 プロセスとスレッド

プログラムの実行単位をプロセスという．各プロセスにはそれぞれ独立したメモリ空間が割り当てられる．どのプロセスも他のプロセスの持っているメモリ空間にアクセスすることはできない．このため，プロセス間のデータのやり取りをしたい場合は，外部的・内部的な何らかの通信手段を介して行うことになる．つまり，プログラム中にデータの受け渡しを明示的に示す必要がある．

一般に 1 プロセスが 1 プログラムに対応する．しかしながら，プロセスの中にはさらに分けることが出来る部分が存在する．それをスレッドとして扱うことができる．あるプロセス内の複数のスレッドは，同じメモリ空間を

共用するため，プロセスと比べてダイレクトにメモリをアクセスできる．このため，パフォーマンスが良い．

3.4 並列のまとめ

3.4.1 プロセス長所と短所

長所

- 複数のマザーボードにメモリを点在できるので，巨大なメモリ空間を得るシステムが構築できる．
- 一般的に，同程度のシステムを組んだ場合，分散メモリ型，特に PC クラスタでは安価にシステムを構築できる．

短所

- プロセスの生成はスレッドよりもコストがかかるので短時間で終わる処理には向いていない．

3.4.2 スレッドの長所と短所

長所

- 分散したメモリ間での通信が無いため，プロセッサ数が一桁程度の範囲では実行性能の理論最大性能に対する落ち込みが少ない．
- 得たいデータ領域に対して直接そのアドレスを参照することができる

短所

- 他タスク・他ジョブとのメモリアクセス競合による性能低下が発生する

4 プログラム例

4.1 プログラム例 1〜sum.c〜

```
#include<stdio.h>
#include<pthread.h>

extern  int  *sum(int  *);
pthread_t  thread1, thread2;

void main()
{
    int  *rc1, *rc2;
    int  arg1 = 1000, arg2 = 2000;
    /* 2つのスレッドを生成する */
    pthread_create( &thread1, NULL, (void* (*) (void*))sum, &arg1 );
    pthread_create( &thread2, NULL, (void* (*) (void*))sum, &arg2 );
    /* スレッドが終了するのを待ち，終了ステータスを受け取る */
    pthread_join( thread1, (void**)&rc1 );
    pthread_join( thread2, (void**)&rc2 );
    /* 結果を表示 */
    printf( "SUM(1..%d) + SUM(1..%d) = %d\n" , arg1, arg2, *rc1 + *rc2);
    /* スレッドを開放 */
    free(rc1);  free(rc2);
    exit(0);
}

/* スレッドとして実行される関数 */
int *sum( int *arg_ptr )
{
    /* これらの自動変数は，スレッドスタック上に確保されるので，値はスレッドごとに独立している */
    int  max = *arg_ptr;
    int  i, sum;
    int  *p;
    for(i=1, sum=0; i<=max; i++){
        sum += i;
    }
    /* 計算結果を返す場所を確保する */
    p = (int *)malloc(sizeof(int));
    *p = sum;
    return p;
}
```

4.2 プログラム例 2〜counter.c〜

```
#include<pthread.h>
/* カウンタ（共有資源） */
```

```
int  counter;
/* カウンタを保護する Mutex */
pthread_mutex_t   counter_mutex = PTHREAD_MUTEX_INITIALIZER;

/* カウンタをインクリメントする（複数のスレッドから安全に呼べる） */
void  counter_inc()
{
    pthread_mutex_lock( &counter_mutex );
    counter++;
    pthread_mutex_unlock( &counter_mutex );
}

/* 現在のカウンタ値を返す（複数のスレッドから安全に呼べる */
int counter_get()
{
    int  rc;
    pthread_mutex_lock( &counter_mutex );
    rc = counter;
    pthread_mutex_unlock( &counter_mutex );
    return  rc;
}
```