マルチコアでソフトウェアはどのように動くのか

2022年11月17日

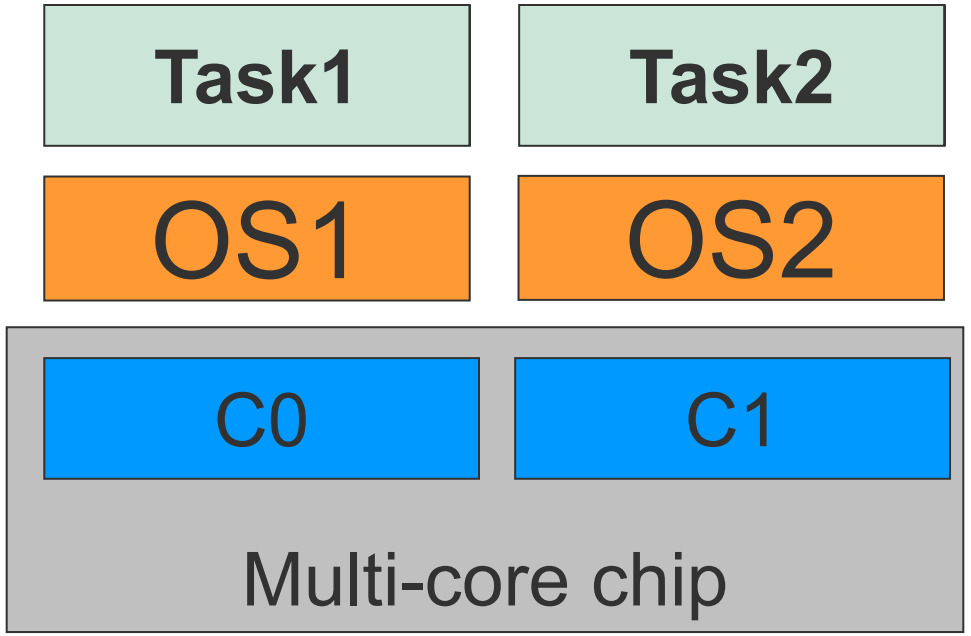
名古屋大学大学院情報学研究科
枝廣　正人

概要

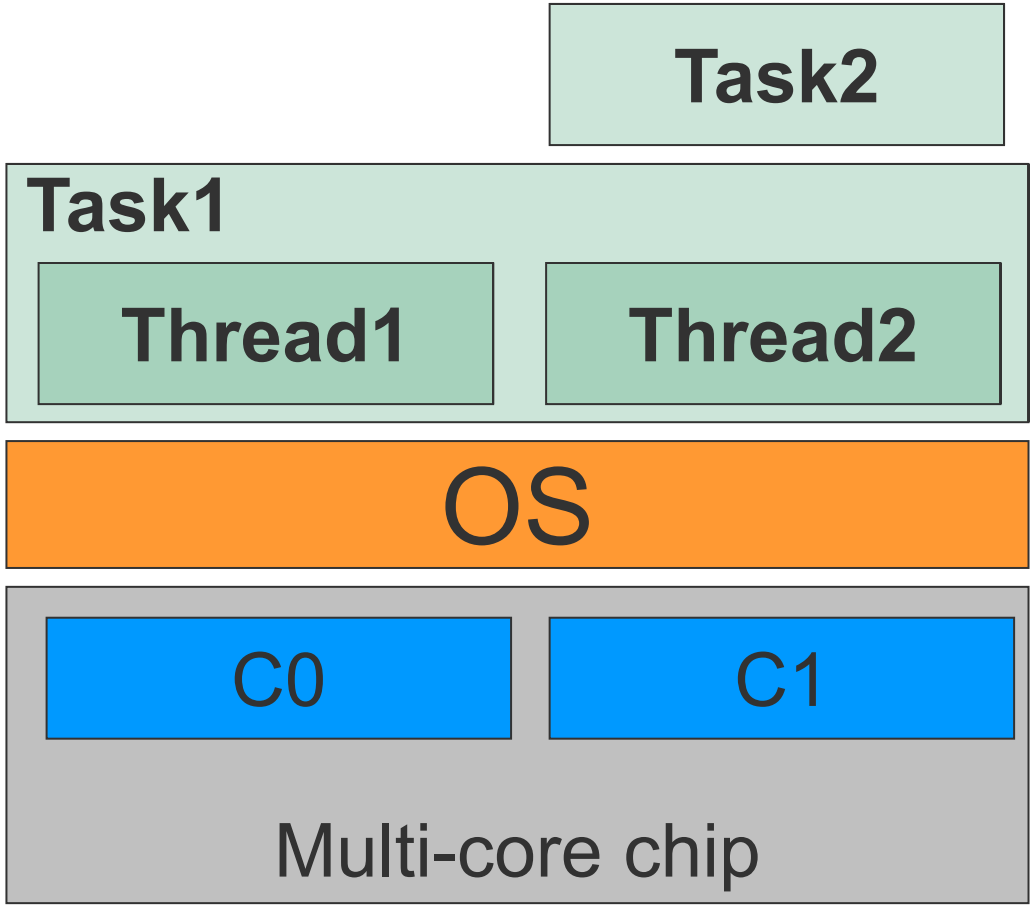
- 本講演ではマルチコア上でのプログラムや基本ソフトウェアの動きの基礎について説明する。並列動作やコア間連携におけるハードウェアとソフトウェアの基本的な動きについてわかりやすく説明する。

- 主に以下について説明
  - マルチタスクをマルチコア上に静的配置して動かす(AMPモデル)
  - マルチタスクをマルチコア上に動的配置して動かす(SMPモデル)
  - スレッドプログラミングをして動かす(SMPモデル)

マルチコア上のソフトウェアの動きは大きく分けて２種類



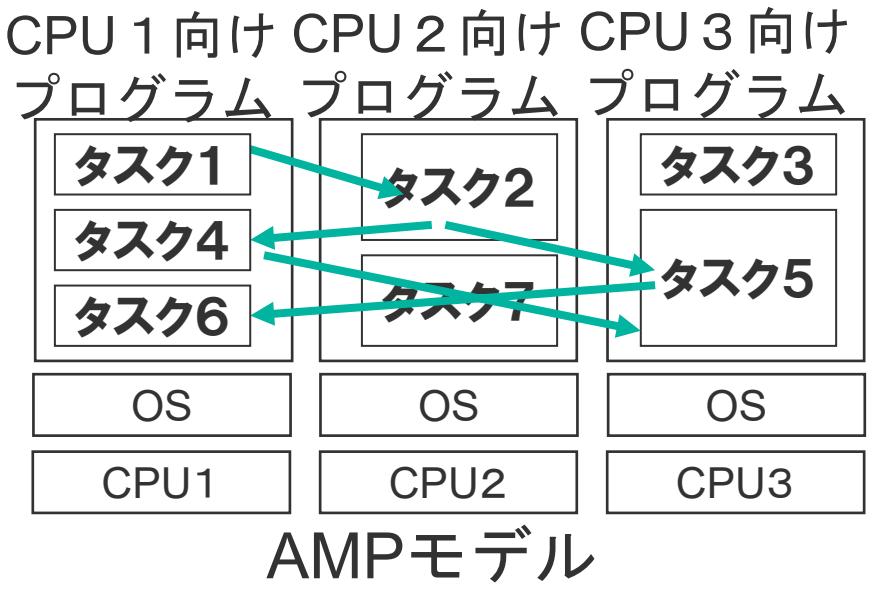
AMPモデル



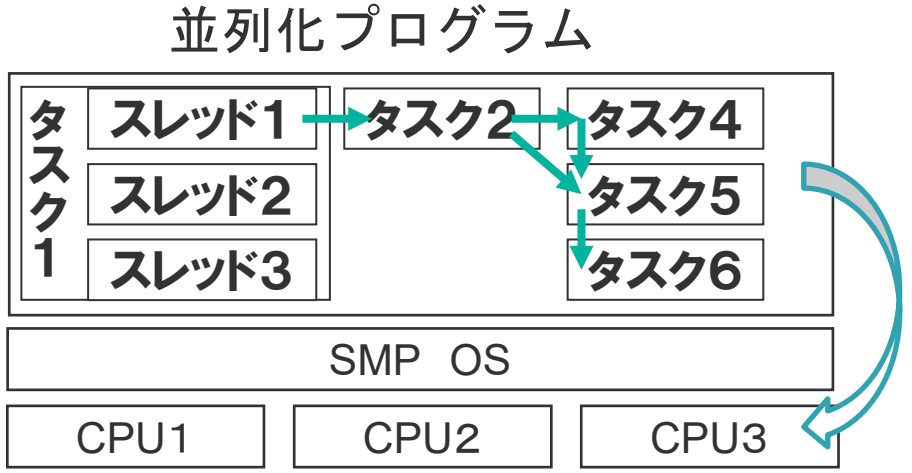
SMPモデル

AMPモデルとSMPモデル

- AMPモデルのプログラム
  - マルチタスクソフトウェア、タスクをコアに静的割付
  - 同期・通信以外は通常のソフトウェア
    - 他コアも周辺モジュールの一種だと考えることも可能
- SMPモデルのプログラム
  - マルチタスクソフトウェア、タスクをコアに動的割付
  - タスク内でスレッド・プログラミングにより並列化



AMPモデル

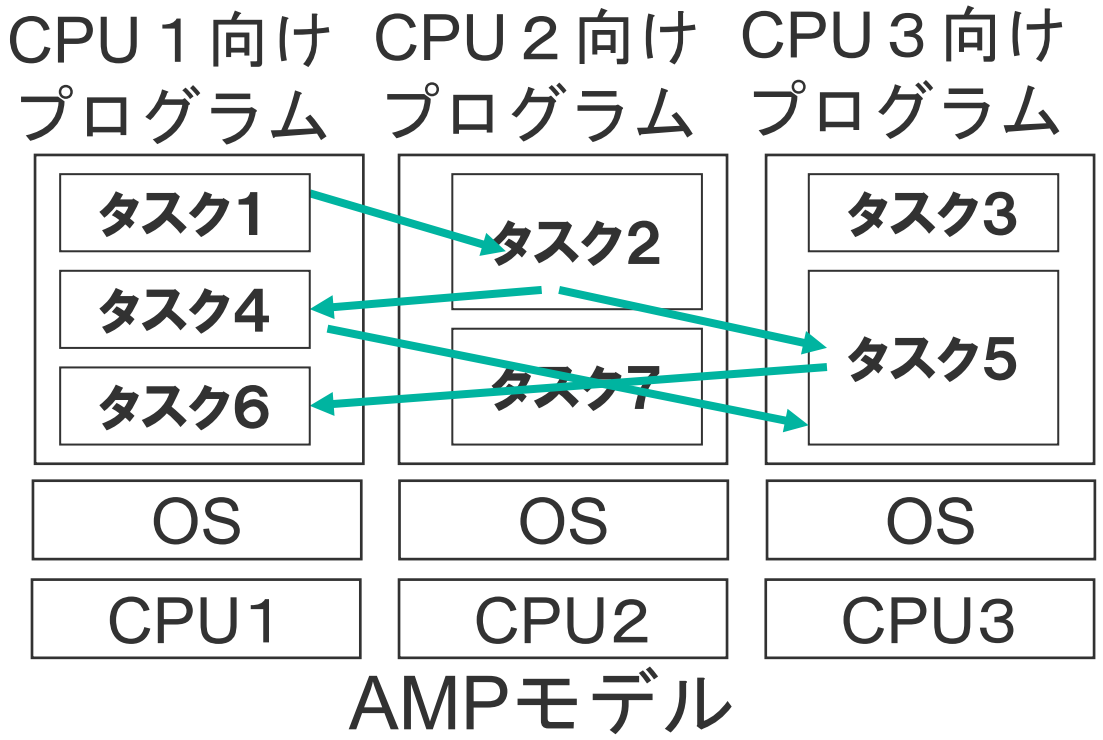


SMPモデル

SMP OS がタスクやスレッドをコアに動的に割り付け

AMPモデルのプログラム

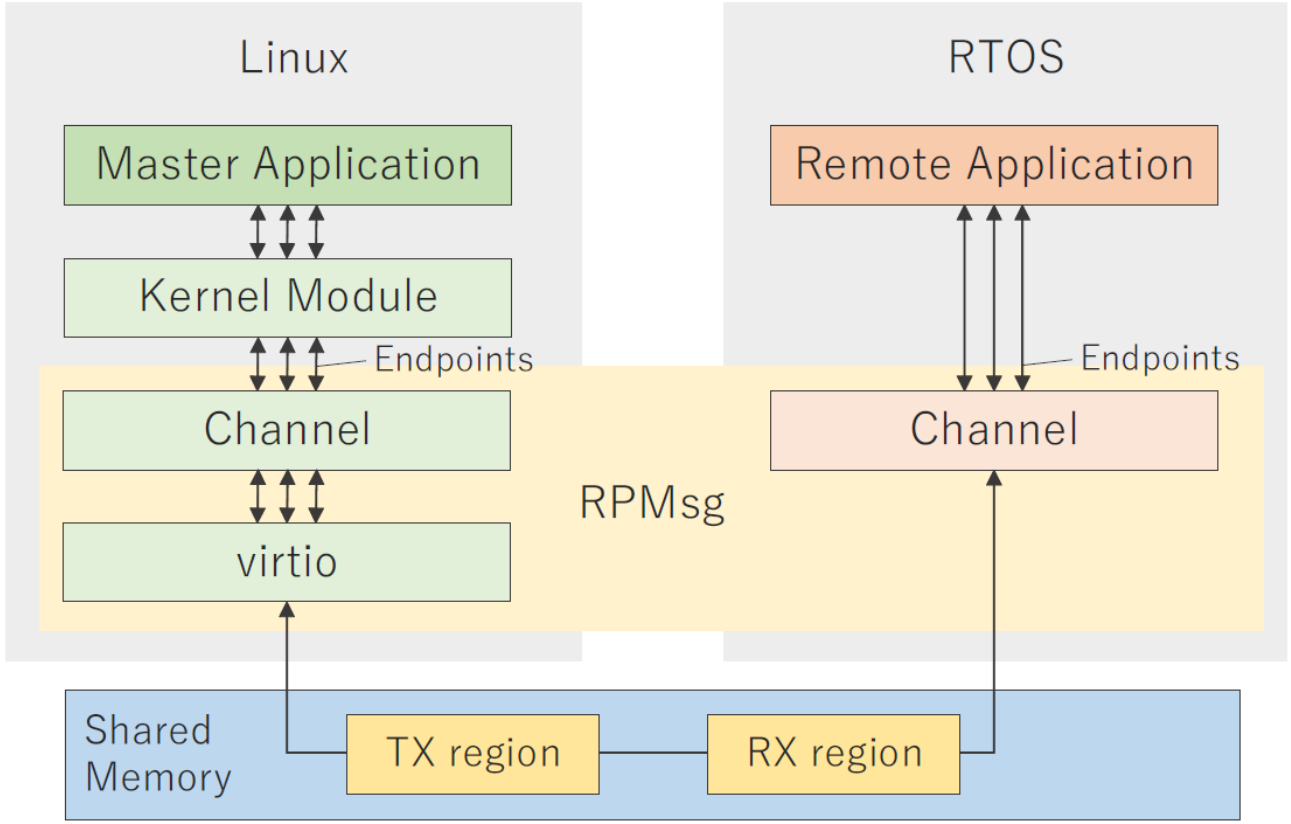
- マルチタスクソフトウェア、タスクをコアに静的割付
- 同期・通信以外は通常のソフトウェア
  - コア間通信が必要
    - 異なるOS、ベアメタルなど、ターゲットに合わせてコア間通信メカニズムを実装する必要がある
    - フレームワークも提案されている
      - 例：OpenAMP (MCA⇒Xilinx, etc.)、MDCOM (TOPPERS)、MCAPI (MCA), etc.



AMPモデル

コア間通信のフレームワーク（1）

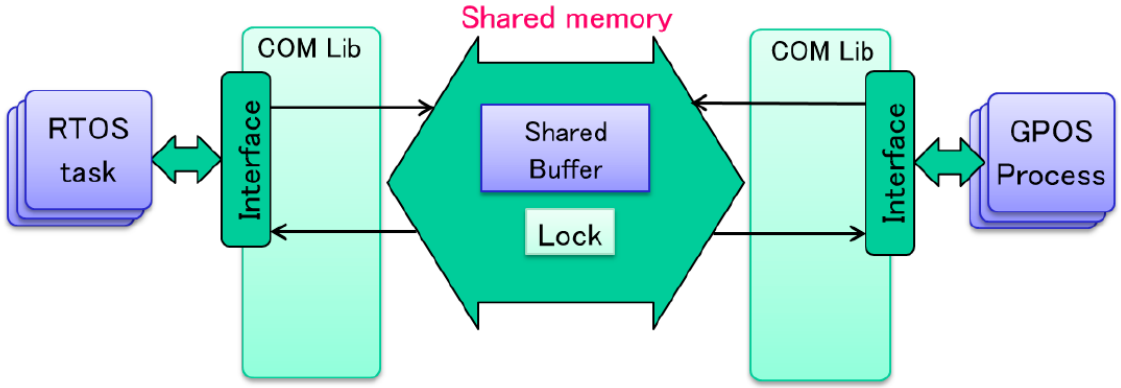
- OpenAMP
  - MCA (Multicore Association) で標準化。Xilinx等が推進
  - LinuxのvirtIO、remoteproc、RPMsgの仕組みを使って実現
  - RTOSまたはベアメタル上のリモートコンピューティングリソースを管理



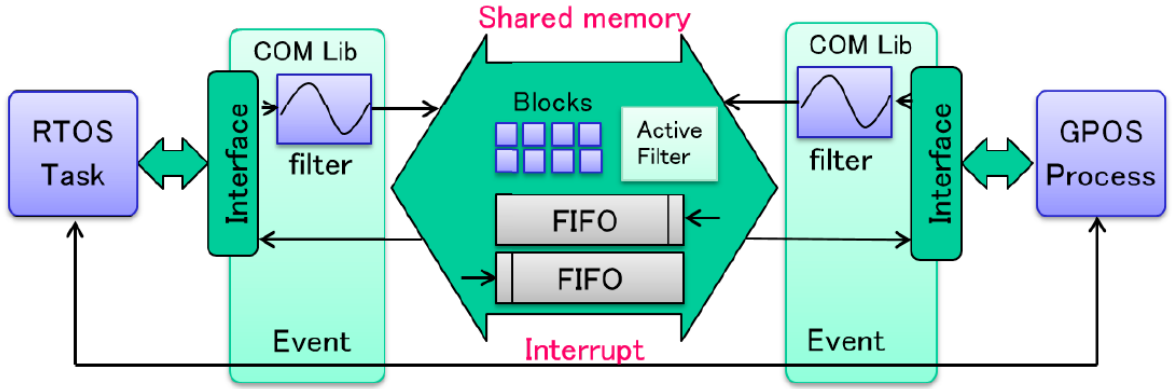
- Xilinx, libmetal および OpenAMPユーザー ガイド
- https://github.com/OpenAMP/open-amp

コア間通信のフレームワーク（2）

- MDCOM
  - TOPPERSプロジェクトで公開
  - さまざまなOS間で動作する通信ライブラリをユーザープログラムAPIとして提供
  - 共有メモリまたはFIFOを提供する「チャネル」を複数持つことが可能



共有メモリチャネル

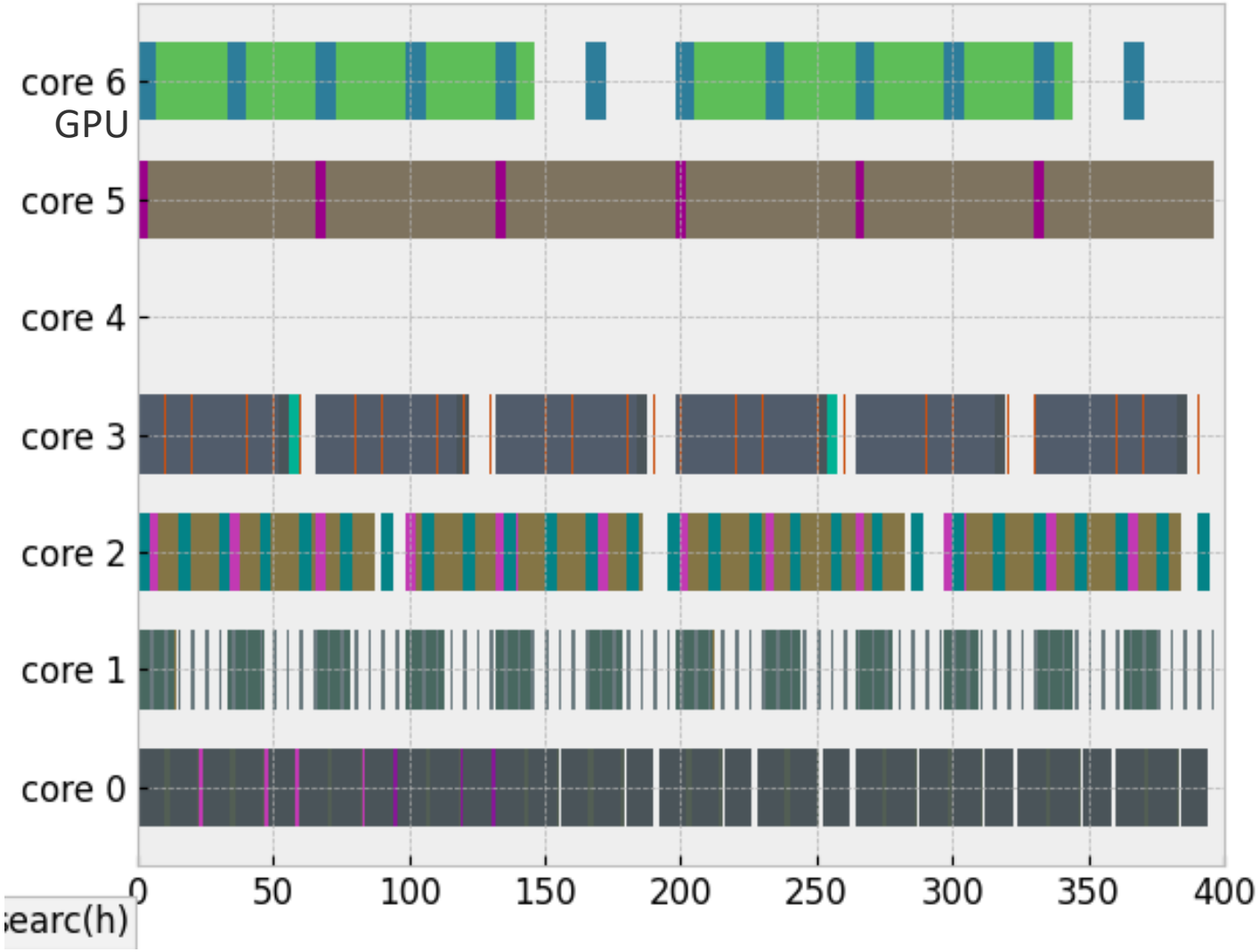


FIFOチャネル

大竹他, ヘテロジニアスプロセッサ向け通信ライブラリ MDCOM, ETNET2017, 2017.

AMPモデルのプログラム実行

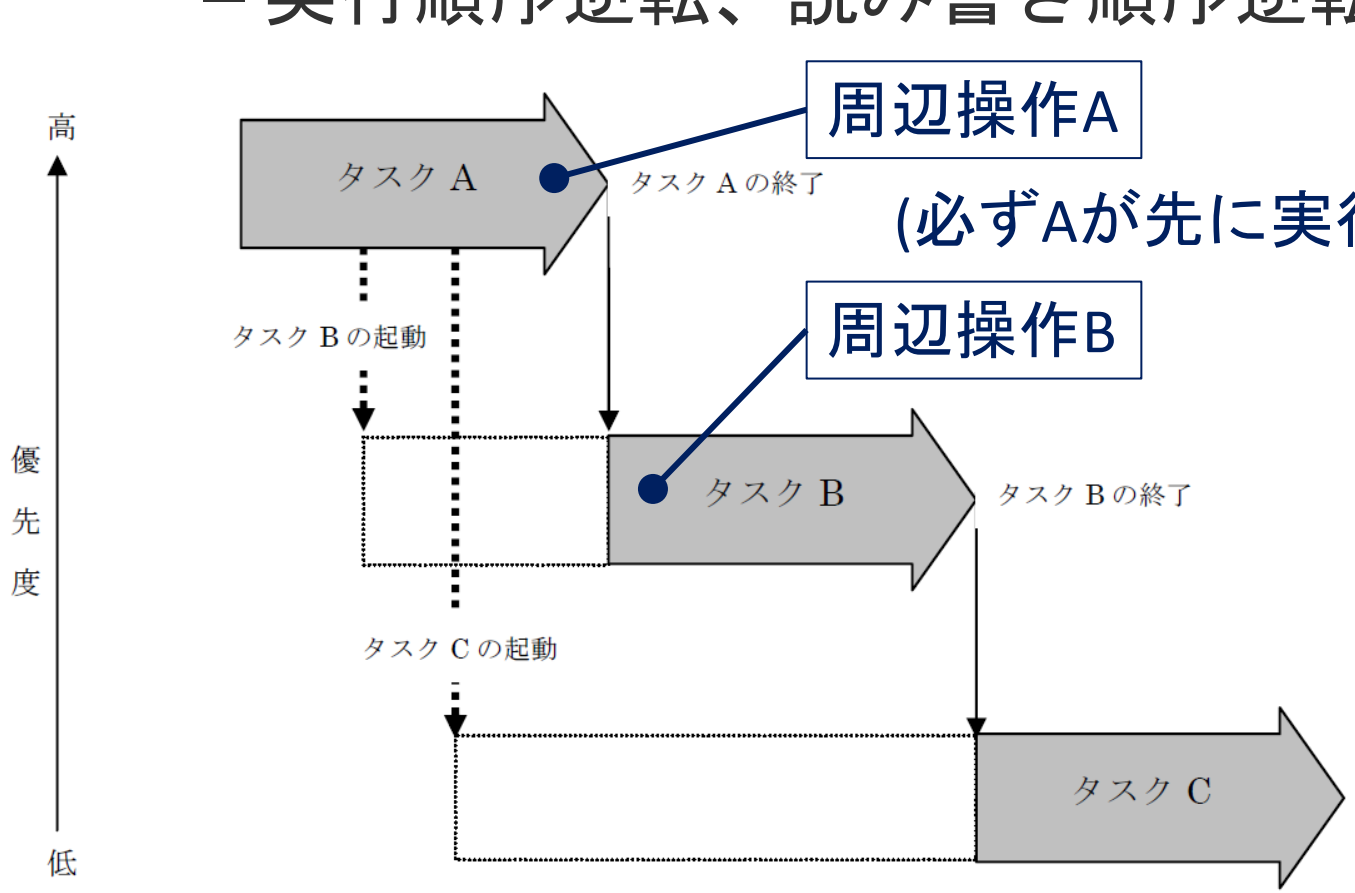
- 個々のシングルプロセッサシステムが同時に動いているという観点が重要
- アプリケーションは個々のプロセッサを意識する必要がある
  - データ依存関係、コア間通信オーバーヘッド、コアごとの特性や基本ソフトウェアを考慮し、タスクをコアに静的割当して実行

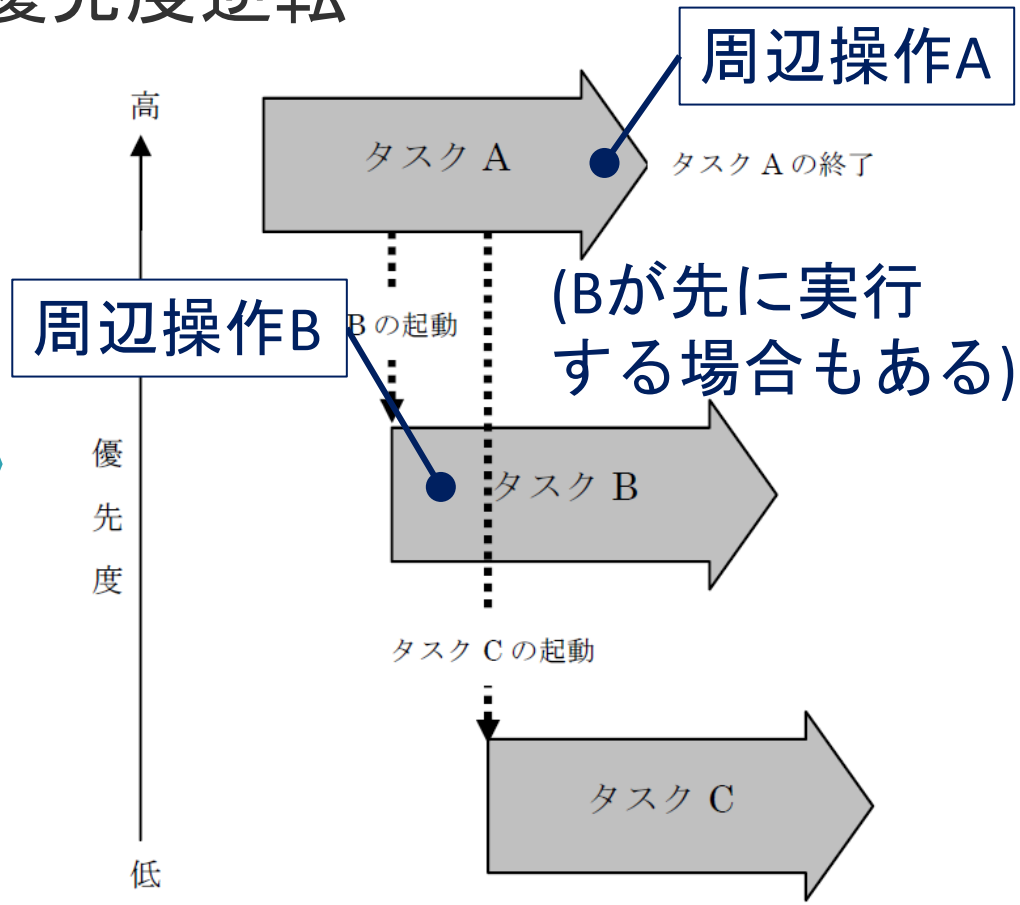


複数のプログラムが同時に動くということ

- 優先度で実行順序制御をしていると、、、
  - 実行順序逆転、読み書き順序逆転、優先度逆転

SMP T-Kernel仕様書より
(SMP T-Kernel 1.00.01, 2017)



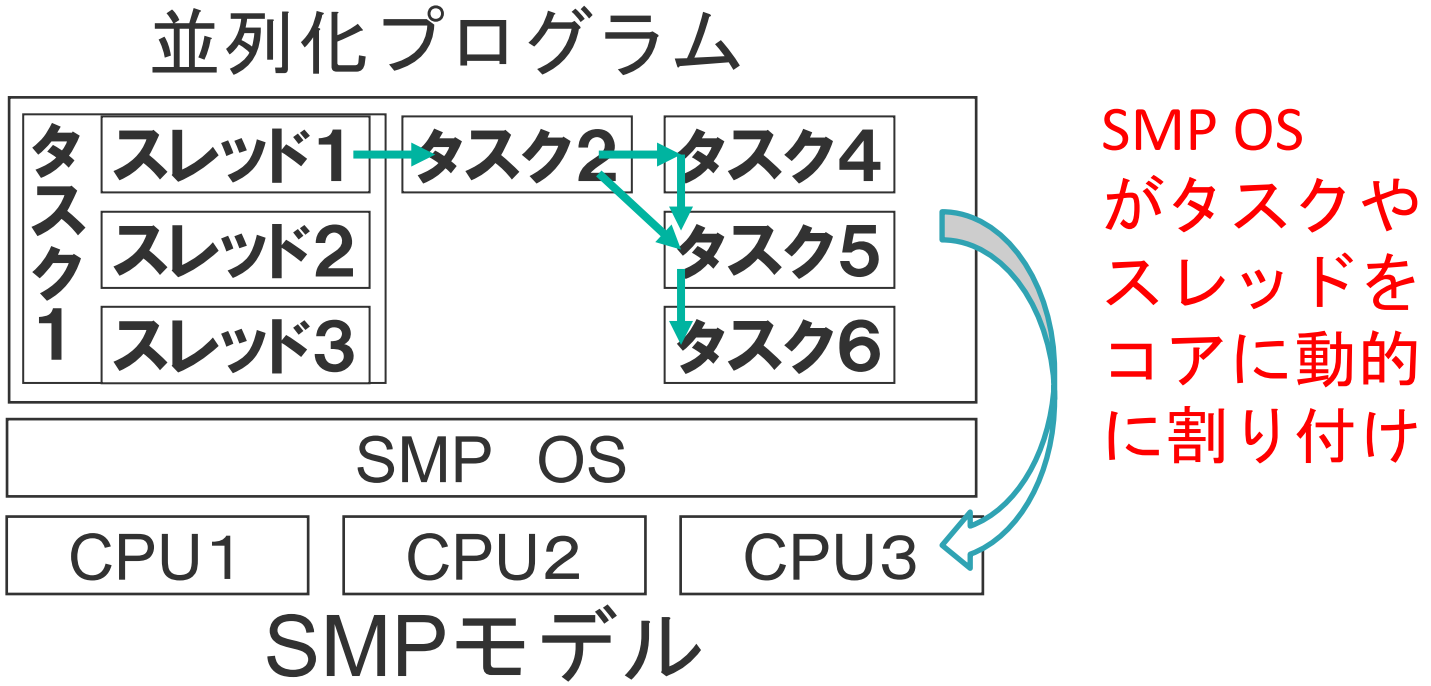


［図 2(a)］シングルプロセッサの T-Kernel によるタスク実行の例

SMPモデルのプログラム（マルチタスク）

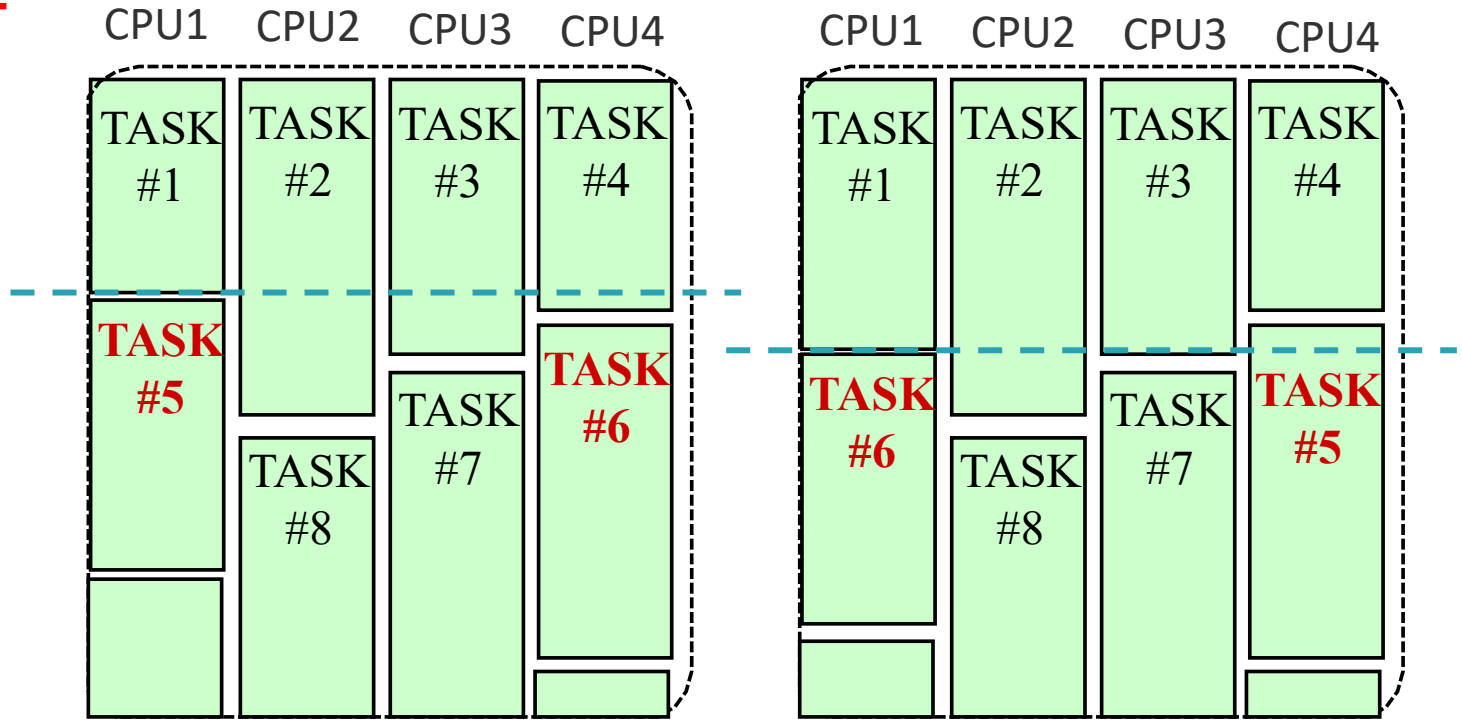
- マルチタスクソフトウェア、タスクをコアに動的割付
  - SMP OSはアプリケーション、プロセッサを一元管理
  - SMP OSがタスクをコアに動的に割り付けて実行

- タスク内でスレッド・プログラミングにより並列化



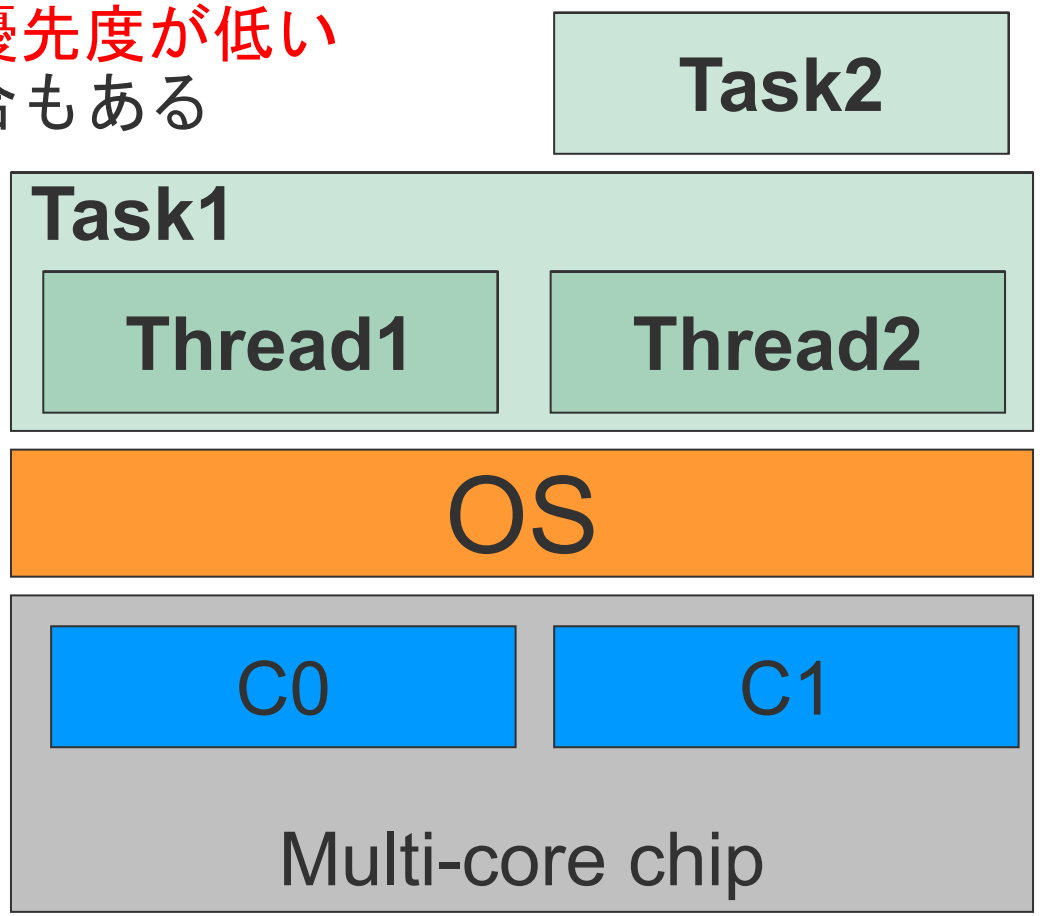
マルチタスクプログラムをSMP上で動かす（1）

- アプリケーションからは個々のコアはSMP OSにより隠蔽される。
  通常、アプリケーションはコアを意識する必要はなく、
  コアの個数を意識せずプログラム可能
- SMP OSがタスクをコアに
  動的に割り付けて実行
  - 動作コアやタイミングが
    変わることもある
- コアを意識する場合
  - 「アフィニティ」による
    静的割付
  - 割込みやデバイス制御



マルチタスクプログラムをSMP上で動かす（2）

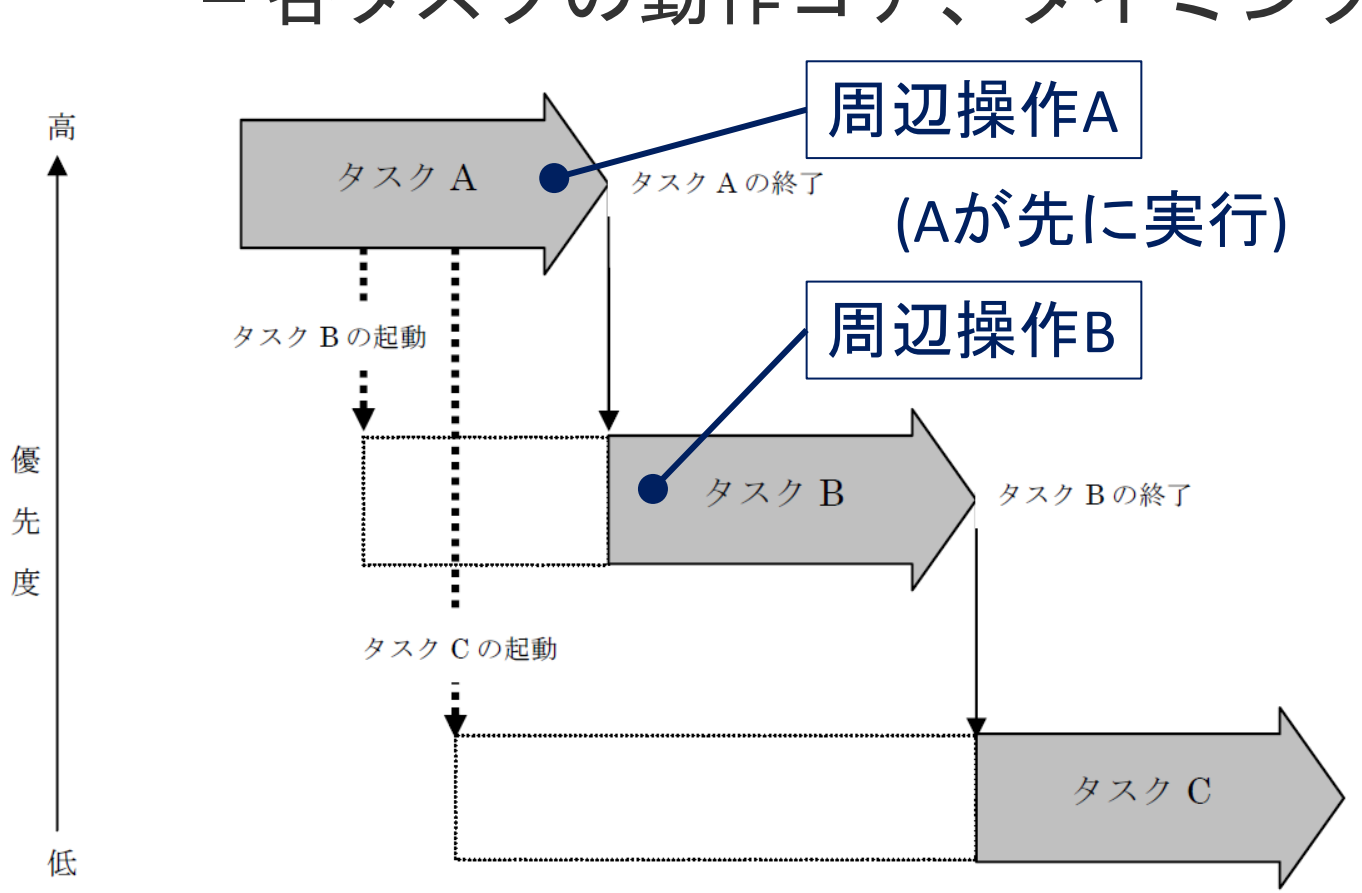
- 注意点
  - 実行状態のタスクが、最大でプロセッサ数ある
  - あるタスク実行中に、そのタスクよりも優先度が低いタスクが実行され、その影響を受ける場合もある
  - ハンドラ実行中にも他のプログラム（タスク、ハンドラ）が実行される
  - タスク動作コアやタイミングが変わることもある
  - 個々のコアを積極的に個別制御したいならば、AMPモデルも検討した方がよい
  - 共有メモリ上に共有変数を持つことは可能であるが、別コアで並行動作するタスクとの順序関係は不定であるため、順序逆転の可能性がある場合には同期等を利用すべき

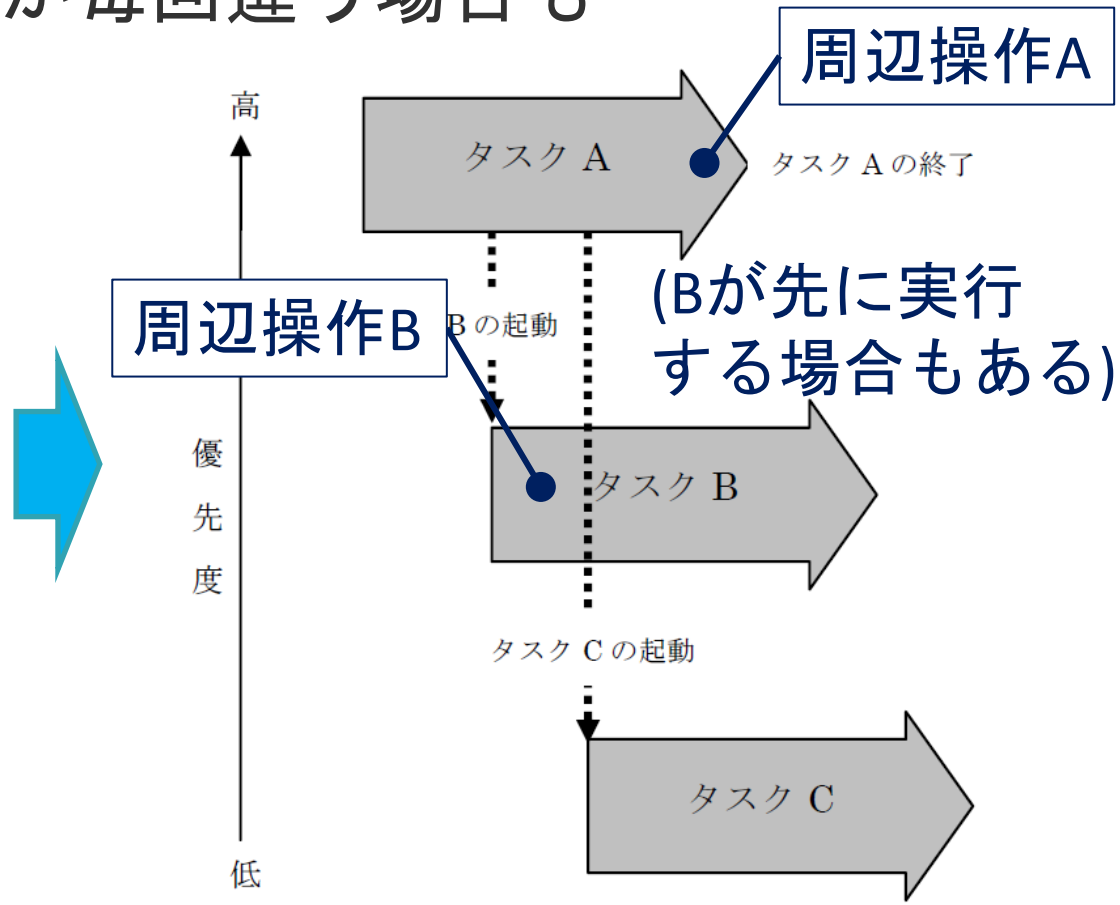


複数のプログラムが同時に動くということ

- SMPモデルではさらに問題が起こりやすい
  - 各タスクの動作コア、タイミングが毎回違う場合も

SMP T-Kernel仕様書より
(SMP T-Kernel 1.00.01, 2017)

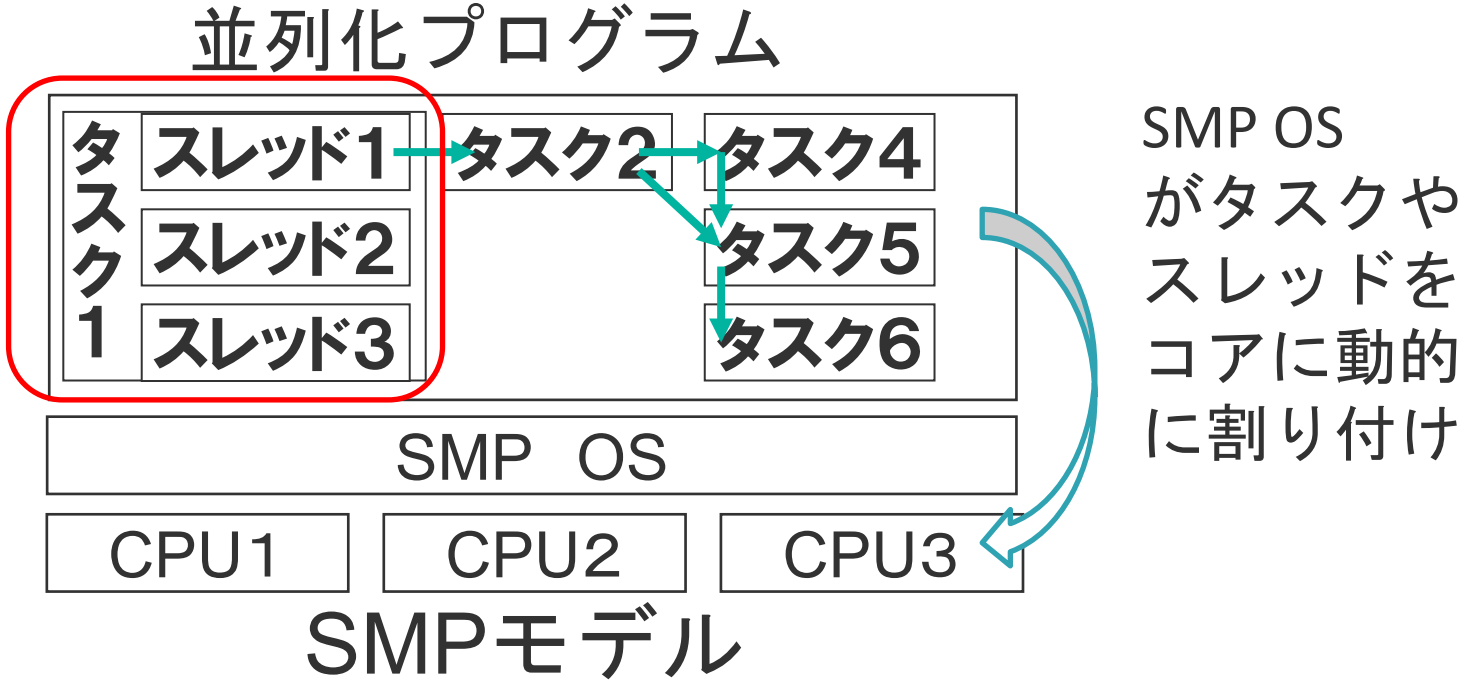




［図 2(a)］シングルプロセッサの T-Kernel によるタスク実行の例

SMPモデルのプログラム

- マルチタスクソフトウェア、タスクをコアに動的割付
- タスク内でスレッド・プログラミングにより並列化
  - 本講演ではOpenMPを例として説明
  - 実験ではIntel 4コアPC、48コアサーバを用いた（RTOSではない）
  - 順序逆転、共有変数の扱いに注意（後述）



OpenMP

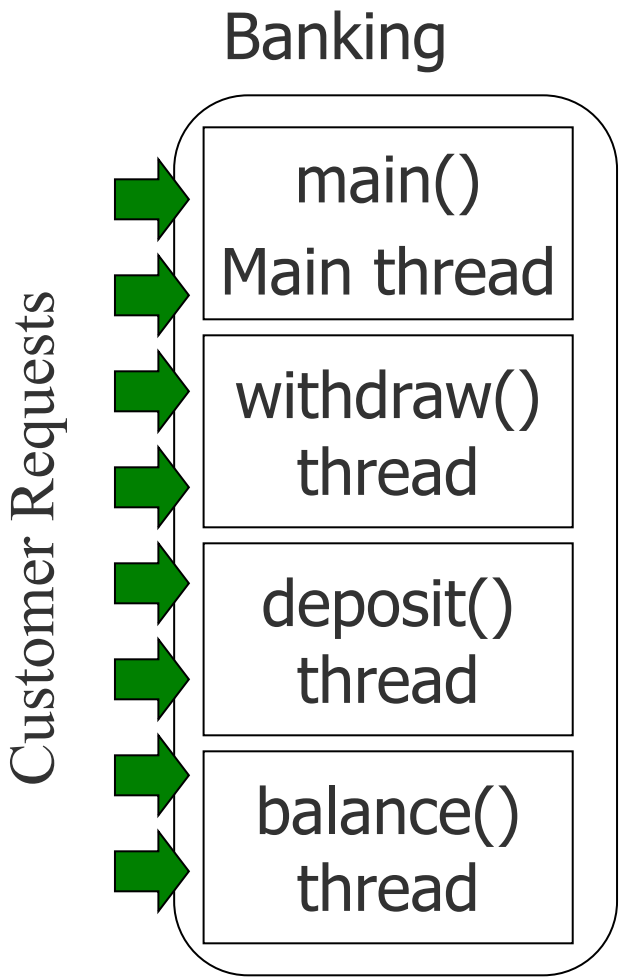
- C/C++/FORTRANの指示行として並列を記載
- USのコンパイラベンダが集まって開発
- PC向けの開発環境などでサポートされている
- http://www.openmp.org/
- Fork-Join Model
- 実行粒度はランタイムによって決められる

OpenMPの例

- ‘sections’ブロックにおいて
  複数の‘section’が並列実行

```
#pragma omp parallel sections
{
#pragma omp section
   main();
#pragma omp section
   withdraw();
#pragma omp section
   deposit();
#pragma omp section
   balance();
}
```

- ‘sections’ブロックの‘}’で同期
  (すべてのsectionは’}’で同期)



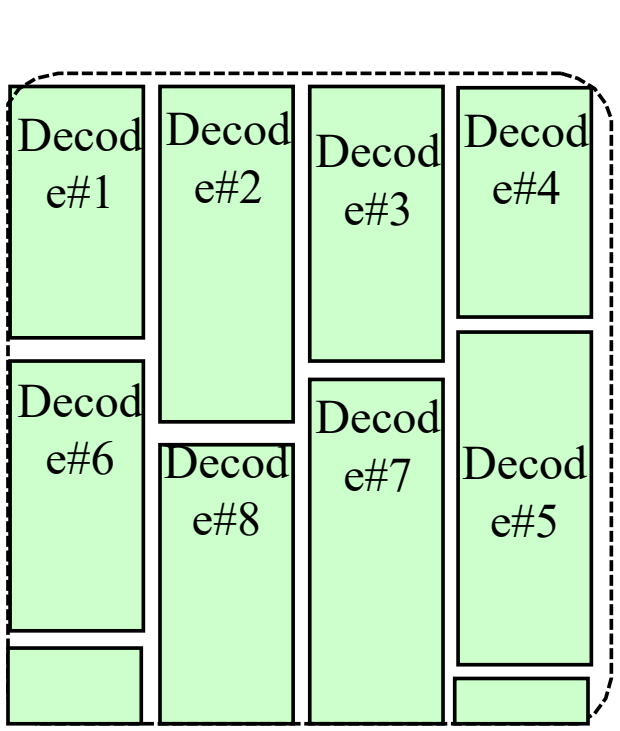
OpenMPの例

- Forループの前の指示行により並列実行

```
#pragma omp parallel for
    for(i=1; i<=N; i++)
        Decode#i;
```

–その他の指示文
  - 総和
  - バリア
  - アトミック
  - …

（注）細かく言えば、
OSのカーネルスレッドに
マップされるのか、ユーザレベル・スレッドスケジューラを
持つのか、気にする必要もある



プログラム例１（hello world）

```
int main()
{
	char a[11]={'h','e','l','l','o',' ','w','o','r','l','d'};
	int b[11]={0,0,0,0,0,0,0,0,0,0};
	int i;
	for(i=0;i<11;i++)
	{
		b[i]=a[i];
		printf("%c¥n”, b[i]);
	}
	for(i=0;i<11;i++)
		printf("b[%d]=%c¥n",i,b[i]);

	return 0;
}
```

実行結果

```
gcc main_ex1.c

eda@edapc10 ~/prog/test
$ ./a.exe
h
e
l
l
o

w
o
r
l
d
b[0]=h
b[1]=e
b[2]=l
b[3]=l
b[4]=o
b[5]=
b[6]=w
b[7]=o
b[8]=r
b[9]=l
b[10]=d
```

OpenMP化（hello worldの並列実行）

```
#include<omp.h>
#include<stdio.h>
int main()
{
	char a[11]={'h','e','l','l','o',' ','w','o','r','l','d'};
	int b[11]={0,0,0,0,0,0,0,0,0,0};
	int i;
#pragma omp parallel for
	for(i=0;i<11;i++)
	{
		b[i]=a[i];
		printf("%c, スレッド番号=%d, スレッド数=%d¥n",
				b[i], omp_get_thread_num(), omp_get_num_threads());
	}
	for(i=0;i<11;i++)
		printf("b[%d]=%c¥n",i,b[i]);
	return 0;
}
```

実行すると順序がめちゃくちゃ

```
eda@edapc10 ~/prog/test
$ gcc -fopenmp main_ex1_omp.c

eda@edapc10 ~/prog/test
$ ./a.exe
w, スレッド番号=2, スレッド数=4
h, スレッド番号=0, スレッド数=4
l, スレッド番号=1, スレッド数=4
l, スレッド番号=3, スレッド数=4
o, スレッド番号=2, スレッド数=4
e, スレッド番号=0, スレッド数=4
o, スレッド番号=1, スレッド数=4
d, スレッド番号=3, スレッド数=4
r, スレッド番号=2, スレッド数=4
l, スレッド番号=0, スレッド数=4
 , スレッド番号=1, スレッド数=4
b[0]=h
b[1]=e
b[2]=l
b[3]=l
b[4]=o
b[5]=
b[6]=w
b[7]=o
b[8]=r
b[9]=l
b[10]=d
```

再実行すると：さきほどと違う

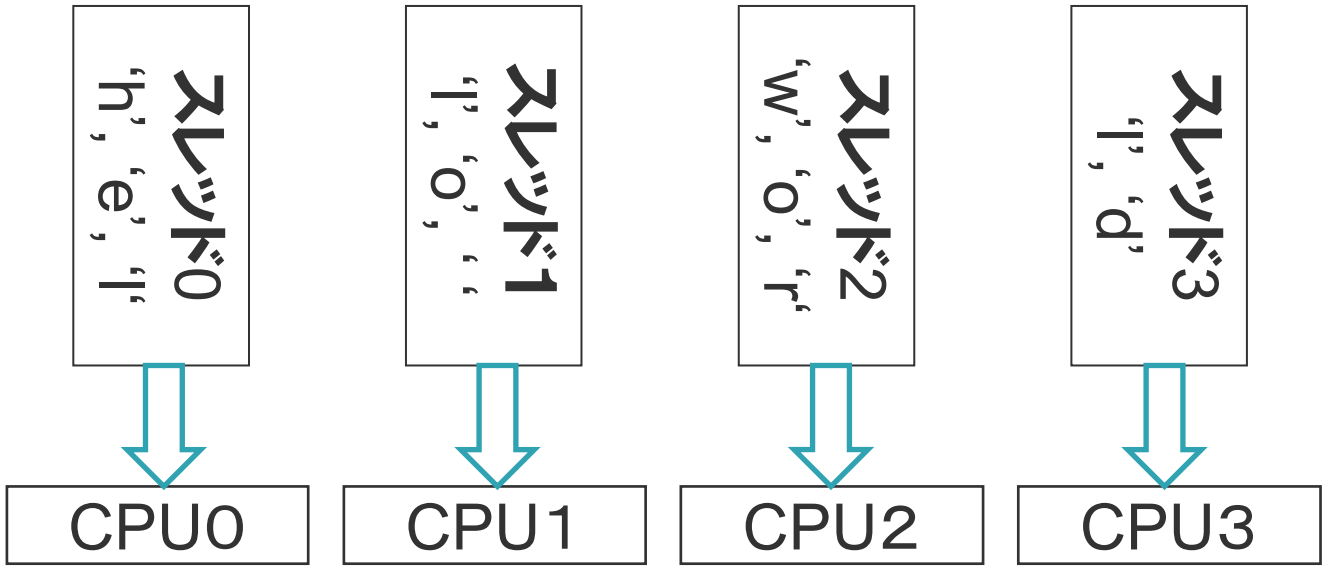
```
eda@edapc10 ~/prog/test
$ ./a.exe
l, スレッド番号=3, スレッド数=4
l, スレッド番号=1, スレッド数=4
h, スレッド番号=0, スレッド数=4
w, スレッド番号=2, スレッド数=4
d, スレッド番号=3, スレッド数=4
o, スレッド番号=1, スレッド数=4
e, スレッド番号=0, スレッド数=4
o, スレッド番号=2, スレッド数=4
 , スレッド番号=1, スレッド数=4
l, スレッド番号=0, スレッド数=4
r, スレッド番号=2, スレッド数=4
b[0]=h
b[1]=e
b[2]=l
b[3]=l
b[4]=o
b[5]=
b[6]=w
b[7]=o
b[8]=r
b[9]=l
b[10]=d
```

よく見ると

```
eda@edapc10 ~/prog/test
$ ./a.exe
l, スレッド番号=3, スレッド数=4
h, スレッド番号=0, スレッド数=4
w, スレッド番号=2, スレッド数=4
l, スレッド番号=1, スレッド数=4
d, スレッド番号=3, スレッド数=4
e, スレッド番号=0, スレッド数=4
o, スレッド番号=2, スレッド数=4
o, スレッド番号=1, スレッド数=4
l, スレッド番号=0, スレッド数=4
r, スレッド番号=2, スレッド数=4
 , スレッド番号=1, スレッド数=4
b[0]=h
b[1]=e
b[2]=l
b[3]=l
b[4]=o
b[5]=
b[6]=w
b[7]=o
b[8]=r
b[9]=l
b[10]=d
```

スレッド番号=0 ⇒’h’, ‘e’, ‘l’
スレッド番号=1 ⇒ ‘l’, ‘o’, ‘ ‘
スレッド番号=2 ⇒ ‘w’, ‘o’, ‘r‘
スレッド番号=3 ⇒ ‘l’, ‘d’

スレッド実行



SMPモデル

SMP OS
がタスクや
スレッドを
コアに動的
に割り付け

すべてのプロセッサが同時実行するため順序が不定

プログラム例２（画像処理）

```
int main(int argc, char *argv[])
{
	Image *colorimg;
	if((colorimg = Read_Bmp(argv[2])) == NULL){
		exit(1);
	}
	int i, maxp = colorimg->height * colorimg->width;
	for(i=0; i<maxp; i++){
		colorimg->data[i].b = 255 - colorimg->data[i].b;
		colorimg->data[i].g = 255 - colorimg->data[i].g;
		colorimg->data[i].r = 255 - colorimg->data[i].r;
		（同様の処理を連続）
	}
	if(Write_Bmp(argv[3], colorimg)){
		exit(1);
	}
	Free_Image(colorimg);
	return 0;
}
```

プログラム例２（画像処理並列化、高速化）

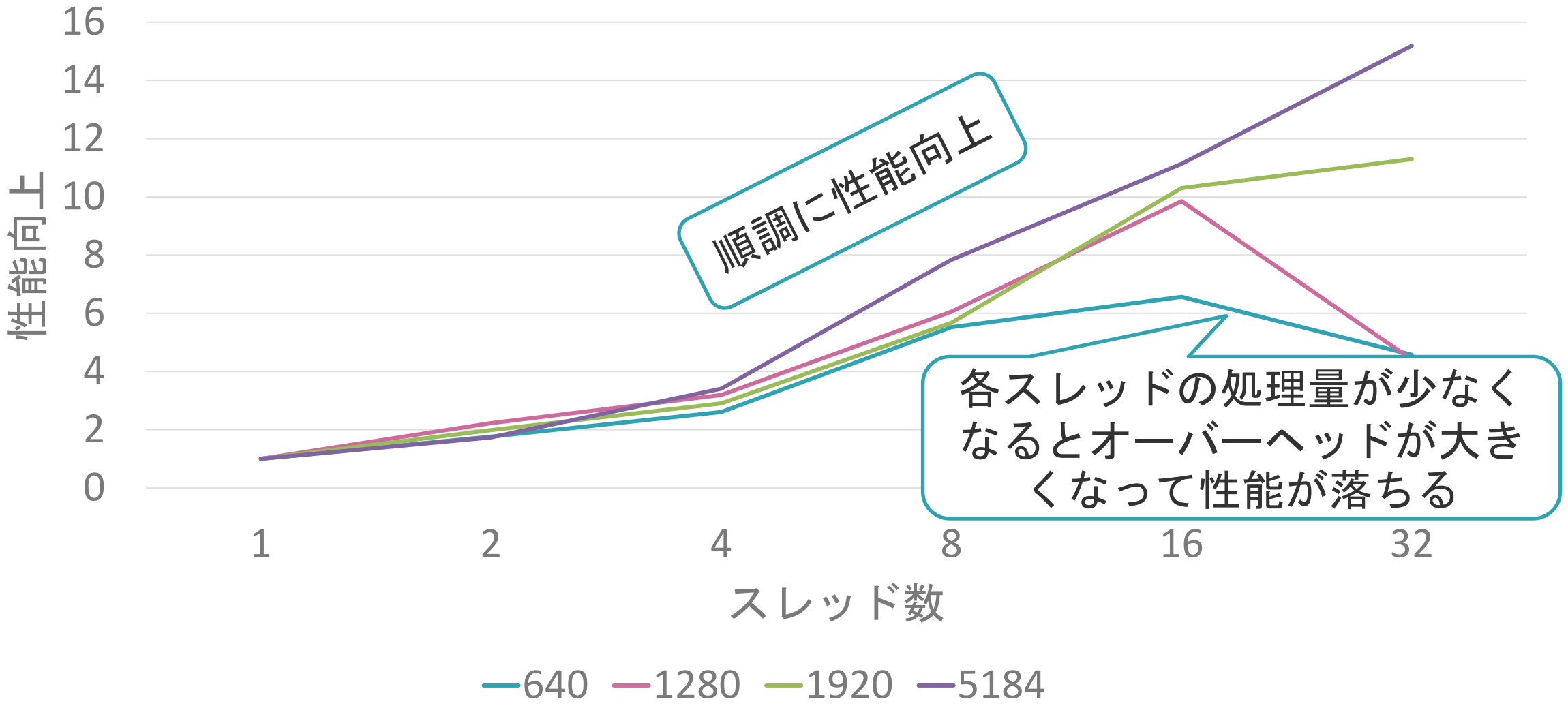
```
int main(int argc, char *argv[])
{
        （省略）
      int i, maxp = colorimg->height * colorimg->width;
#pragma omp parallel for
      for(i=0; i<maxp; i++){
            colorimg->data[i].b = 255 - colorimg->data[i].b;
            colorimg->data[i].g = 255 - colorimg->data[i].g;
            colorimg->data[i].r = 255 - colorimg->data[i].r;
             （同様の処理を連続）
      }
      if(Write_Bmp(argv[3], colorimg)){
            exit(1);
      }
      Free_Image(colorimg);
      return 0;
}
```

実験方法

- プロセッサ台数
  - 48プロセッサの計算機において、1，2，4，8，16，32スレッドを用いて実験

- 画像サイズ
  - 640x426 [640]、1280x853 [1280]、1920x1280 [1920]、5184x3456 [5184]の４種類で実験

評価結果２（ほぼ順調に性能向上）



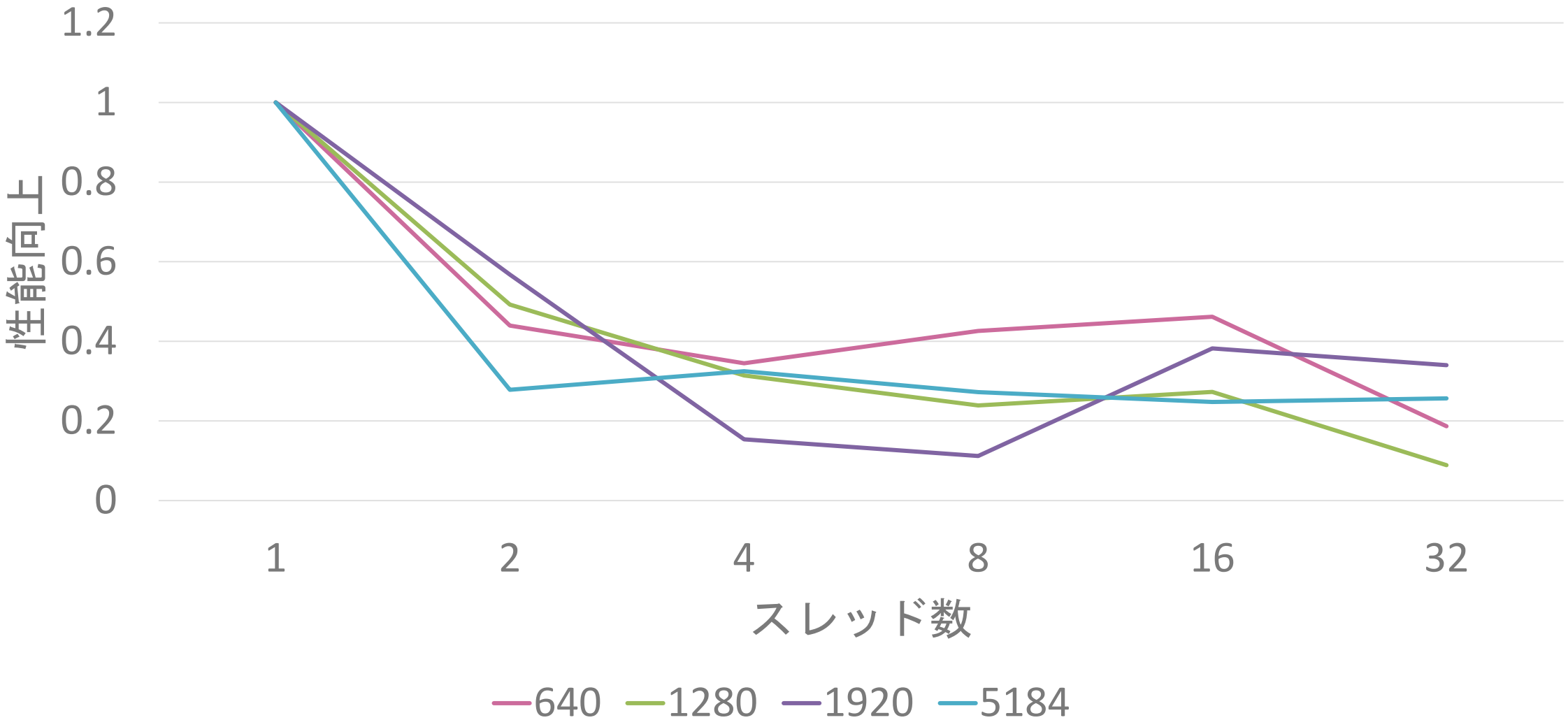
プログラム例２－２（少し変えた画像処理）

```
int main(int argc, char *argv[])
{
        Image *colorimg;
        if((colorimg = Read_Bmp(argv[2])) == NULL){
                exit(1);
        }
        int i, x, y, maxp = colorimg->height * colorimg->width;
        for(i=0; i<maxp; i++){
                x = (colorimg->data[i].b + colorimg->data[i].g + colorimg-
>data[i].r) / 3;
                y = colorimg->data[i].b + x;
                if(y > 255) y = y % 253;
                colorimg->data[i].b = y;
                （同様の処理をg,rにも）
        }
        if(Write_Bmp(argv[3], colorimg)){
                exit(1);
        }
        Free_Image(colorimg);
        return 0;
}
```

プログラム例２－２（少し変えた画像処理、OpenMP）

```
int main(int argc, char *argv[])
{
        （省略）
      int i, x, y, maxp = colorimg->height * colorimg->width;
#pragma omp parallel for
      for(i=0; i<maxp; i++){
            x = (colorimg->data[i].b + colorimg->data[i].g +
colorimg->data[i].r) / 3;
            y = colorimg->data[i].b + x;
            if(y > 255) y = y % 253;
            colorimg->data[i].b = y;
             （同様の処理をg,rにも）
      }
        （省略）
}
```

評価結果２－２（並列により性能低下）

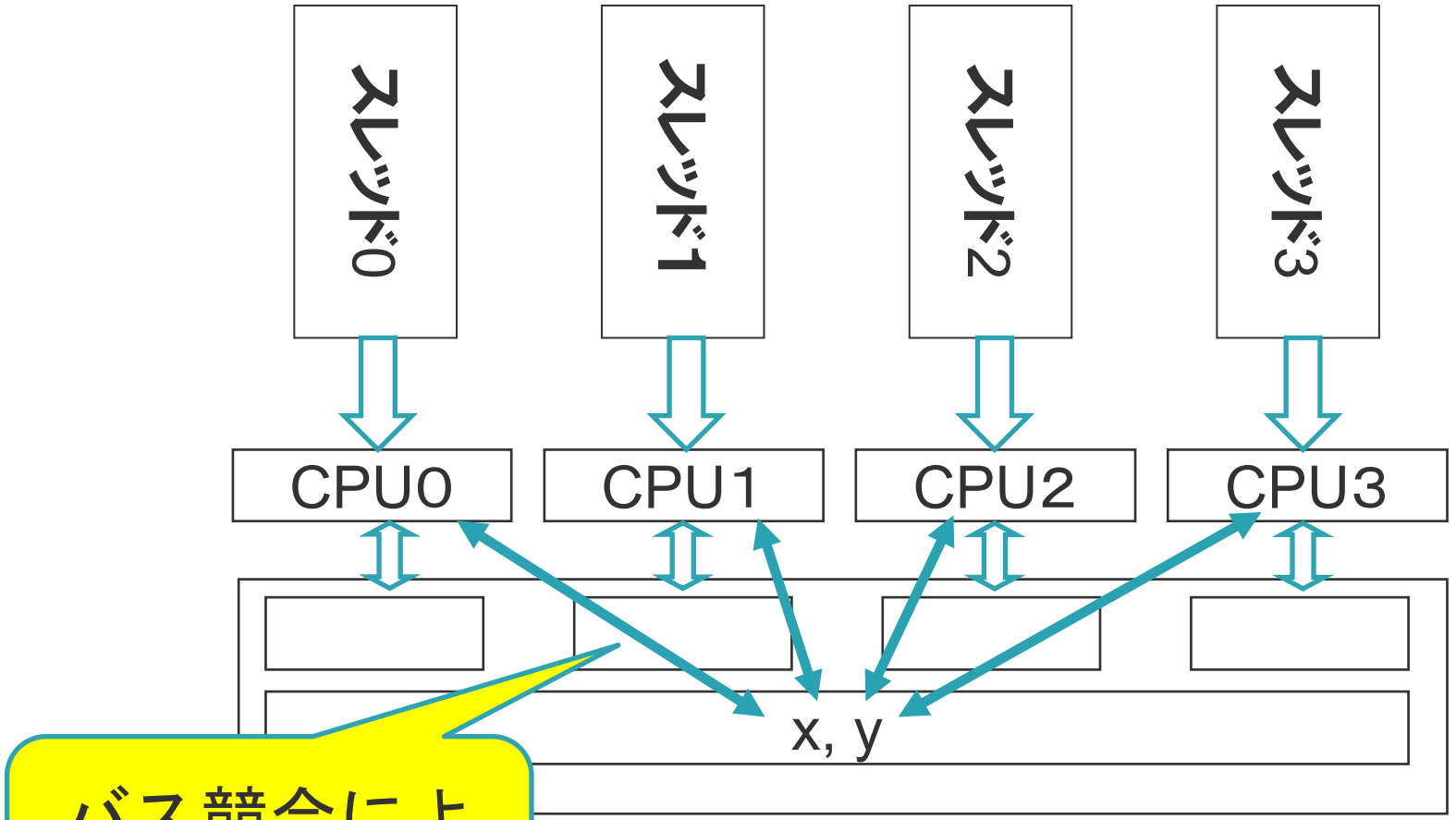


何が違う？

```
      int i, maxp = colorimg->height * colorimg->width;
#pragma omp parallel for
      for(i=0; i<maxp; i++){
            colorimg->data[i].b = 255 - colorimg->data[i].b;
            colorimg->data[i].g = 255 - colorimg->data[i].g;
            colorimg->data[i].r = 255 - colorimg->data[i].r;
      }
```

```
      int i, x, y, maxp = colorimg->height * colorimg->width;
#pragma omp parallel for
      for(i=0; i<maxp; i++){
            x = (colorimg->data[i].b + colorimg->data[i].g +
colorimg->data[i].r) / 3;
            y = colorimg->data[i].b + x;
            if(y > 255) y = y % 253;
            colorimg->data[i].b = y;
      }
```

スレッド実行（共有変数を使う）



SMP OS
がタスクや
スレッドを
コアに動的
に割り付け

スレッドロー
カルメモリ

共有メモリ

バス競合によ
る速度低下

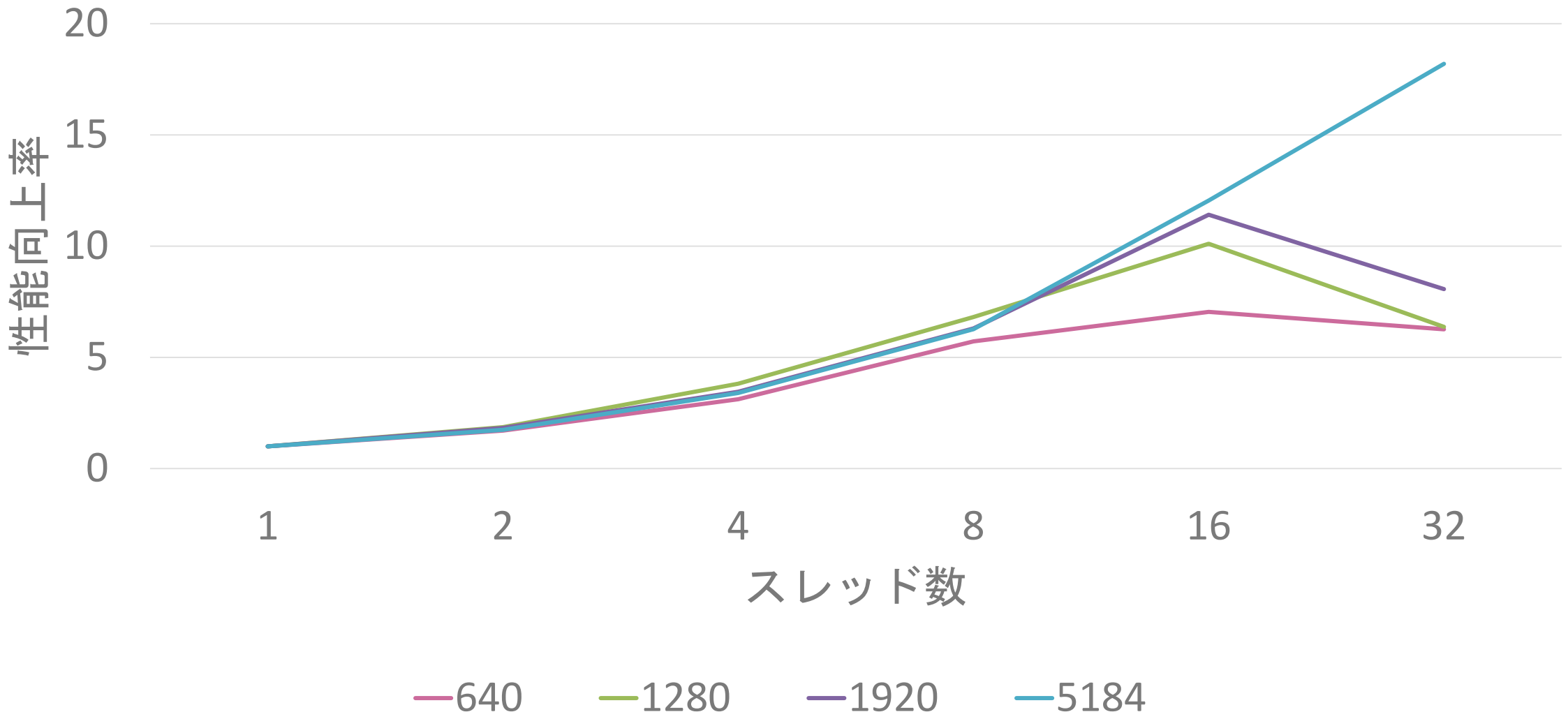
SMPモデル

何も指定しな場合、変数は
共有変数として扱われる

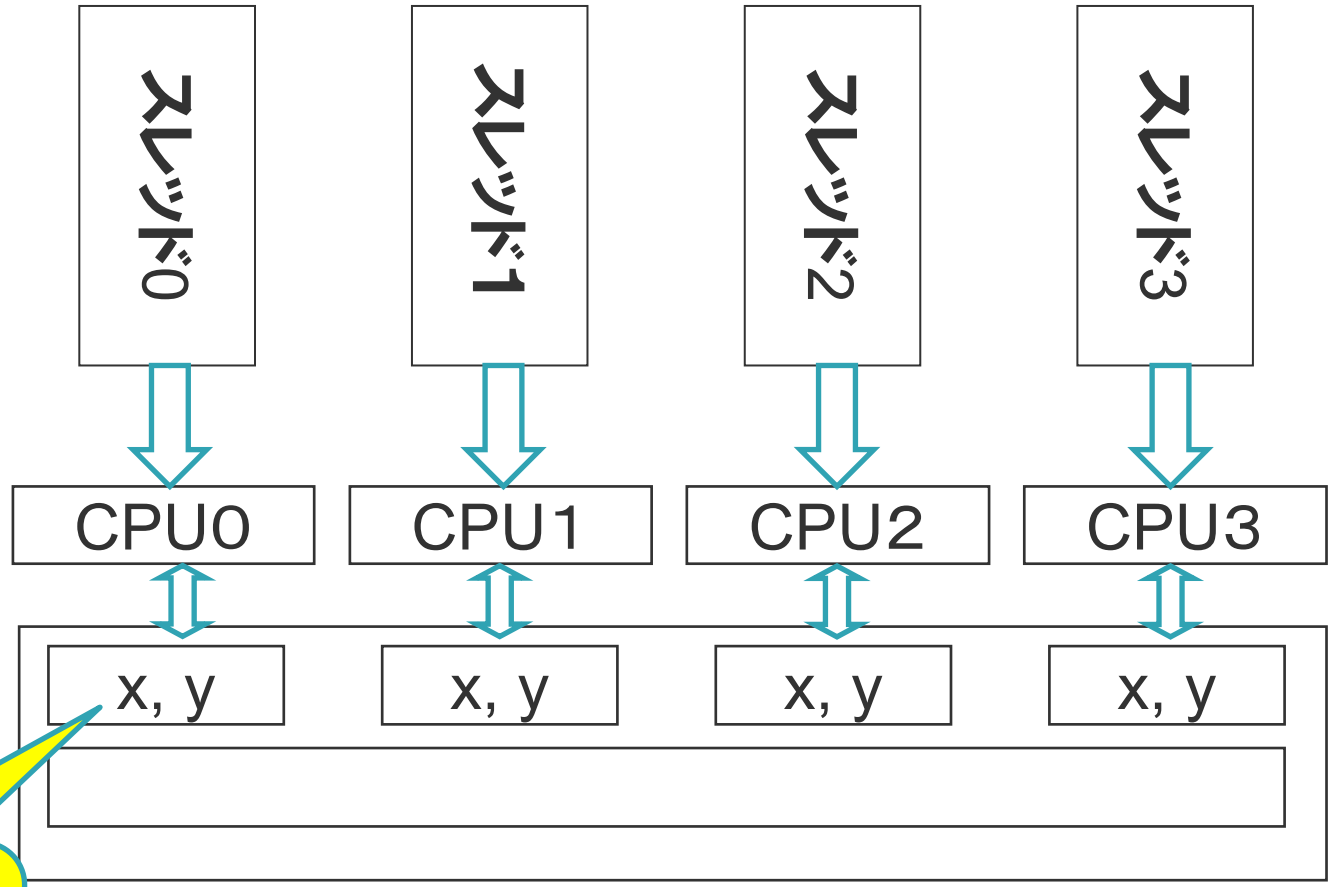
プログラム例２－３（画像処理、OpenMP）

```
int main(int argc, char *argv[])
{
        （省略）
      int i, x, y, maxp = colorimg->height * colorimg->width;
#pragma omp parallel for private(x,y)
      for(i=0; i<maxp; i++){
            x = (colorimg->data[i].b + colorimg->data[i].g +
colorimg->data[i].r) / 3;
            y = colorimg->data[i].b + x;
            if(y > 255) y = y % 253;
            colorimg->data[i].b = y;
             （同様の処理をg,rにも）
      }
        （省略）
}
```

評価結果２－３（無事に性能向上）



スレッド実行（SharedとPrivateの違い）



SMP OS
がタスクや
スレッドを
コアに動的
に割り付け

(Private変数
が置かれる)
スレッドロー
カルメモリ

共有メモリ
(Shared変数
が置かれる)

競合は発
生しない

SMPモデル

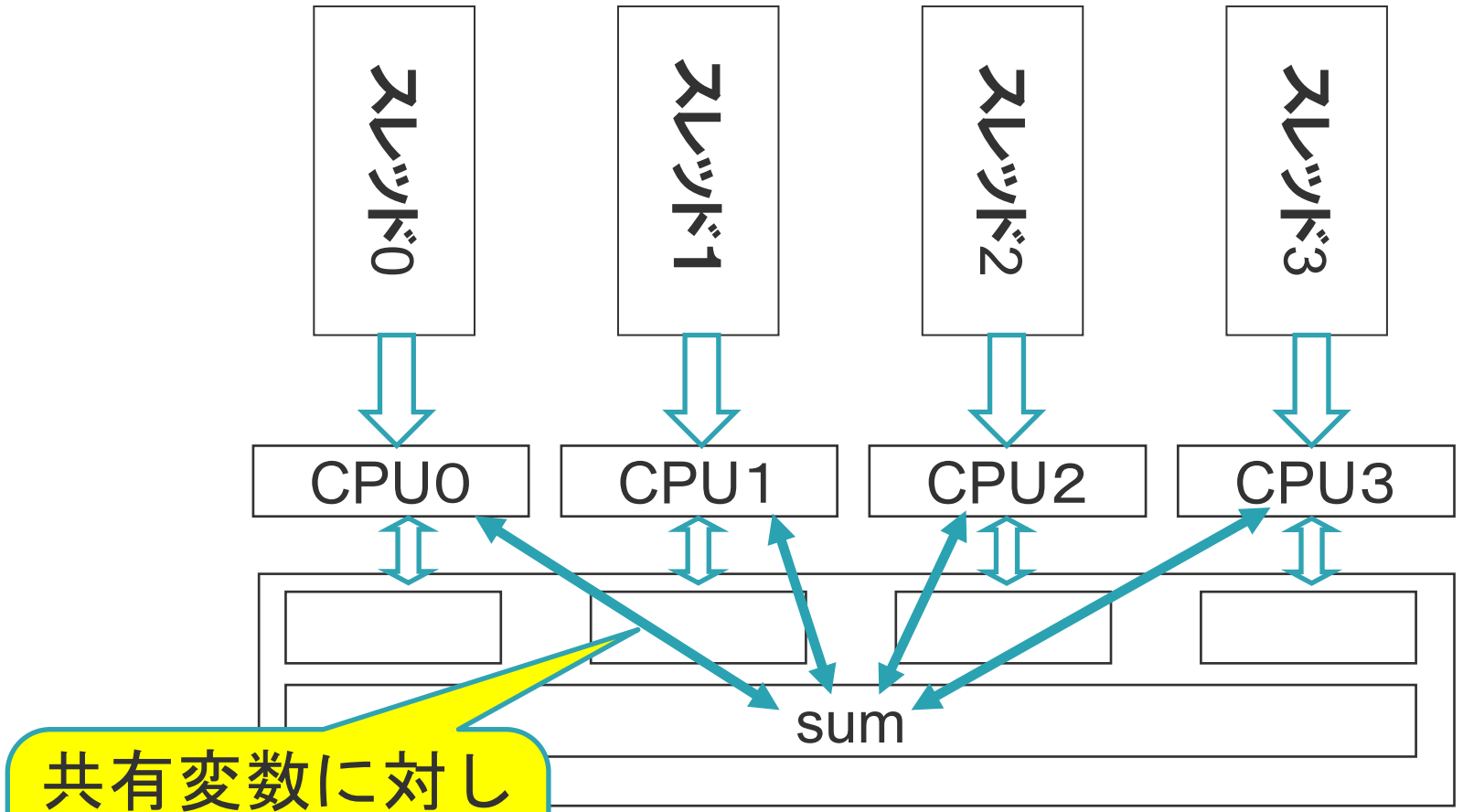
プログラム例３（総和の並列実行）

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

int main(void) {
        int i, sum=0;
#pragma omp parallel for
        for (i=1; i<=1000000; i++)
        {
                sum++;
        }
        printf("sum=%d¥n", sum);
}
```

答えが毎回違う

```
$ gcc -fopenmp main_omp2.c

eda@edapc10 ~/prog/test
$ ./a.exe
sum=267102

eda@edapc10 ~/prog/test
$ ./a.exe
sum=359541

eda@edapc10 ~/prog/test
$ ./a.exe
sum=269597

eda@edapc10 ~/prog/test
$ ./a.exe
sum=306387

eda@edapc10 ~/prog/test
$ ./a.exe
sum=267172
```

スレッド実行（共有変数を使う）



SMP OS
がタスクや
スレッドを
コアに動的
に割り付け

スレッドロー
カルメモリ

共有メモリ

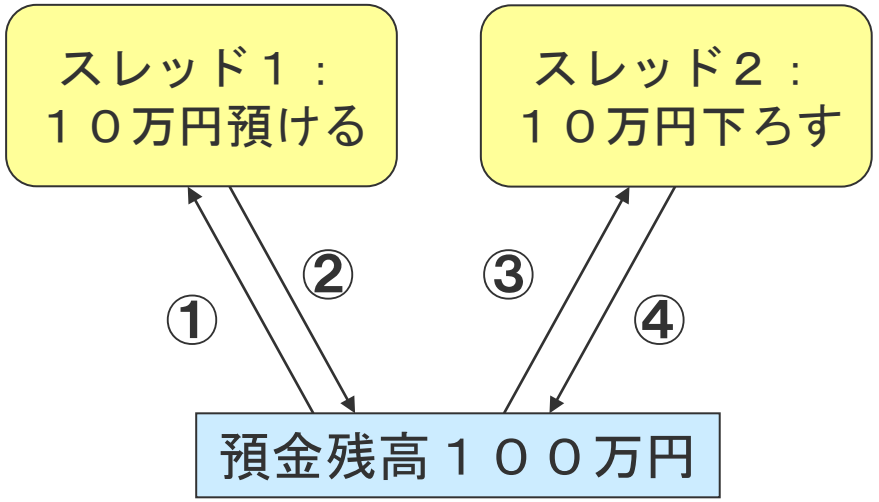
共有変数に対して同時にRead-Modify-Write

SMPモデル

何も指定しな場合、変数は
共有変数として扱われる

共有変数を用いて複数プロセッサからRead-Modify-Writeをかけるときの問題

- 下図のような銀行預金システムにおいて、二つのスレッドから同時に読み書き可能な場合、問題が起こることがある。
  - ①→②→③→④の場合
  - ①→③→②→④の場合



- シングルプロセッサ上のマルチタスクでも同じ問題が起こるが、マルチコアでは複数プロセッサが同時動作するため、現象が起こりやすくなる

どうしたら並列化できるのか？

- 4コア実行ならば100万回のループを25万回ずつに分割

```
int i, j, p, sum;
for (p=0,sum=0; p<4; p++) {
    for(i=0,j=p*250000+1,s=0; i<=250000; i++, j++)
{
        s++;
    }
    sum += s;
}
return sum;
```

- Reductionという

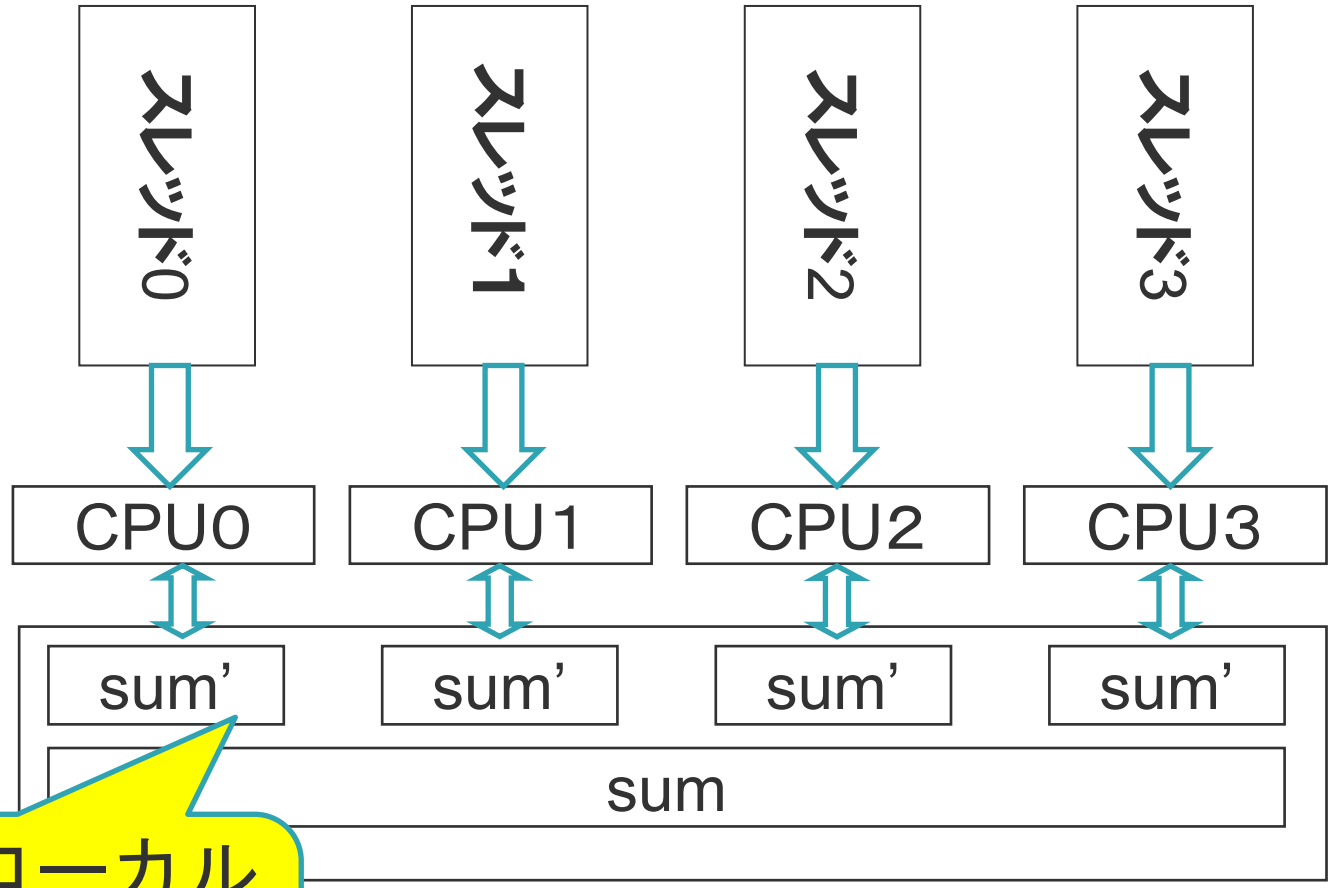
OpenMPでReduction

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

int main(void) {
        int i, sum=0;
#pragma omp parallel for reduction(+:sum)
        for (i=1; i<=1000000; i++)
        {
                sum++;
        }
        printf("sum=%d¥n", sum);
}
```

正しくなった

```
gcc -fopenmp main_omp2.c

eda@edapc10 ~/prog/test
$ ./a.exe
sum=1000000

eda@edapc10 ~/prog/test
$ ./a.exe
sum=1000000

eda@edapc10 ~/prog/test
$ ./a.exe
sum=1000000

eda@edapc10 ~/prog/test
$ ./a.exe
sum=1000000

eda@edapc10 ~/prog/test
$ ./a.exe
sum=1000000
```

スレッド実行（Reduction）



SMP OS
がタスクや
スレッドを
コアに動的
に割り付け

スレッドロー
カルメモリ

共有メモリ

スレッドローカル
に「小計」を持つ
ことにより並列化

SMPモデル

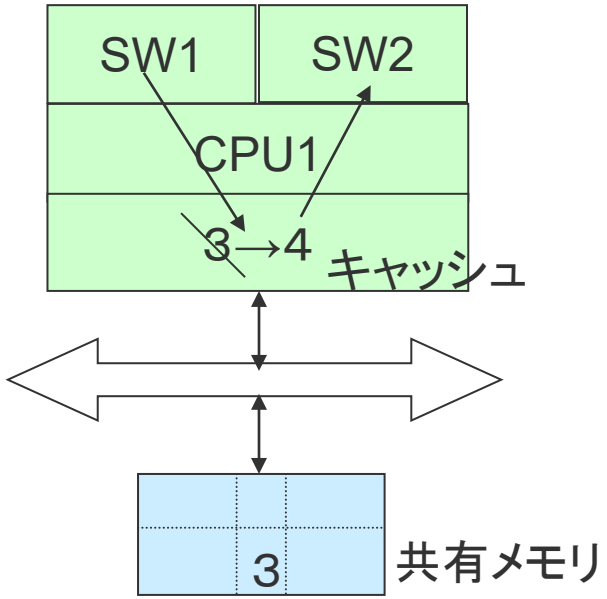
何も指定しな場合、変数は
共有変数として扱われる

まとめ

- マルチコア上でのプログラムや基本ソフトウェアの動きの基礎について説明
- AMPモデル
  - マルチタスクプログラムにおいてタスクを静的割当
- SMPモデル
  - マルチタスクプログラムにおいてタスクを動的割当
  - タスクのスレッド化についてOpenMPを例として説明

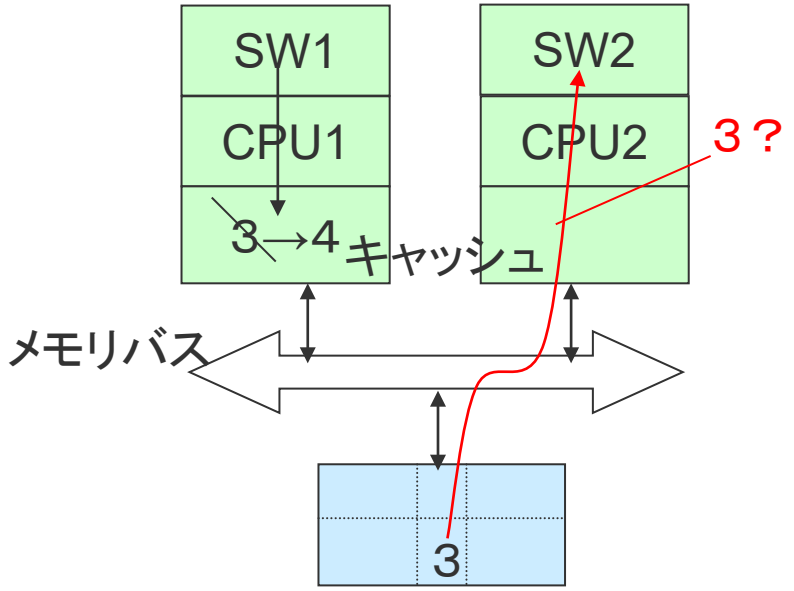
付録：SMP向けハードウェアにおける共有メモリアクセスの高速化（コヒーレントキャッシュ）

シングル
プロセッサ



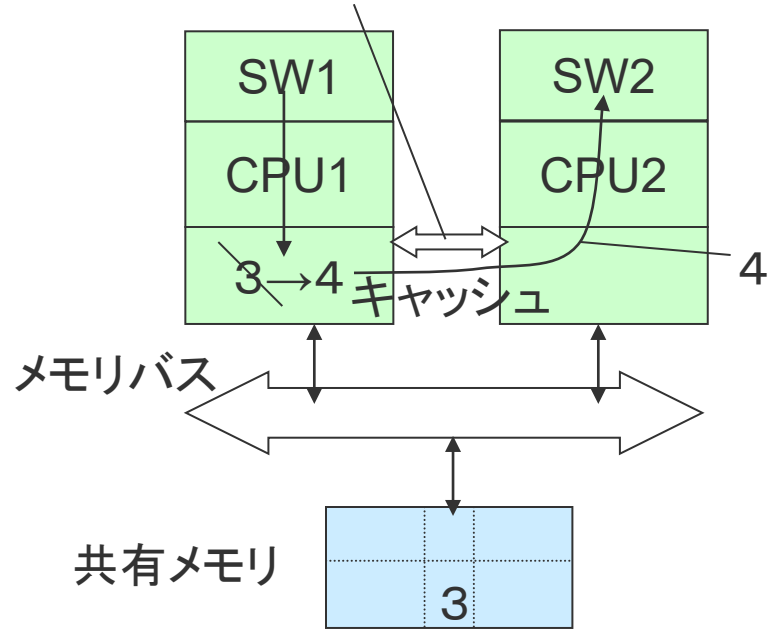
シングルプロセッサの場合、SW1もSW2も同じキャッシュから読むので、オーバーヘッドなく正しい値が読める

AMPモデル：ノンコヒーレント・キャッシュ
（ハードウェアサポートなし）



ノンコヒーレント・キャッシュの場合、SW2でメモリ上の同じデータを使いたいとき、SW1はキャッシュの内容を一度共有メモリに戻した後、SW2に通知する必要があり、大きなオーバーヘッドとなる

SMPモデル：コヒーレント・キャッシュ
（Snoop（盗み見）機構）



コヒーレット・キャッシュでは隣のCPUのキャッシュの内容を盗み見る（Snoop）ハードウェア機構を持つ。ソフトでは気にせずにSW1とSW2のデータ共有ができるため、オーバーヘッドが小さい

付録：メモリ整合性（メモリコンシステンシ）

```
P1: A = 0;                    P2: B = 0;
      .....                         .....
L1: A = 1;                    L2: B = 1;
    if (B == 0) …                 if (A == 0) …
```

- プロセッサP1とP2でそれぞれのプログラムが動く
- 変数A、Bは共有メモリ上にあるとする
- 両方のIF文が同時に真になることはない**はず**。ある種のスイッチ/排他制御になっているつもり
  - 例えばP1でIFが真になったとすると、P2はL2の前を実行しているはずだからP2のIF文の評価のときには既にA=1になっているはず
- しかしながら、実際にはうまく動かない場合が多い
- なんらかの順序関係の記述が必要

順序関係

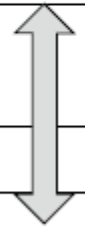
- 順序関係の保証をハードでやるのか、ソフトでやるのか

- ハードですべて保証しようと思えばできないことはないが、オーバーヘッドが大きい（プログラムのごく一部の話で全体が遅くなっていいのか）

- 一部ソフトで保証することにしてハードを簡単化
- 緩和する順序関係の候補
  - Read → Write
  - Write → Read
  - Write → Write
  - （Read → Read）

メモリ整合性モデル

緩和する順序関係（ソフトウェアでの対応必要）	メモリ整合性モデル	ハードウェアによる高速化手法	性能
なし	SC(Sequential Consistency)	なし	低い
W→R	TSO(Total Store Ordering), PC(Processor Consistency)	ライトバッファ	
W→R, W→W	PSO(Partial Store Order)	ライトマージ	
すべて	WO(Weak Ordering), RC(Release Consistency)	ノンブロッキングキャッシュ	高い



ソフトウェアでの対応：
同期をとって、（例えば）ライトバッファをはきだす命令を出す。
sync()**や**flush()**など**