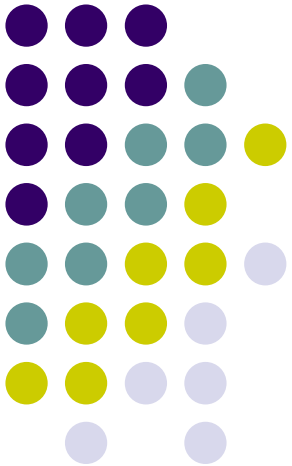
2015年度
実践的並列コンピューティング
第6回

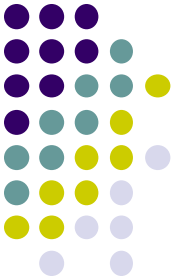
OpenMPによる
共有メモリプログラミング (4)



遠藤 敏夫

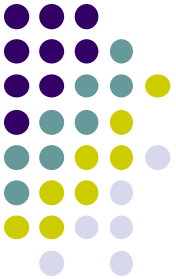
endo@is.titech.ac.jp

並列ソフトウェアを作る上で考えること



- 正しい逐次ソフトウェアか？
- 速い逐次ソフトウェアか？
- 正しい並列ソフトウェアか？
  - 依存関係を壊していないか
  - Race conditionはないか
- 速い並列ソフトウェアか？
  - ノード内で速いか
  - ノード間で速いか

「正しい」並列ソフトウェアに向けて



- 既存アルゴリズムから、並列化可能な場所を見つけ出す
  - 部分計算Aと部分計算Bは、どんな順序で行っても結果が同じか？
  - 独立した計算どうしと分かれば、別スレッドに行わせてもよい
    - 例えば、同じデータに対して書き込んでいない
    - Matmulの反復どうしやfibの2つの関数呼出どうしはokだった

  → データ並列構文やタスク並列構文を使って並列化
- 同じデータに対する書き込みがあると厄介
  - piサンプルの、共有カウンタへの書き込みなど

「正しい」並列ソフトウェアに向けて

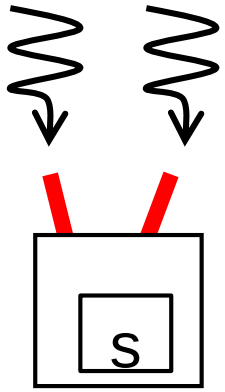
- OpenMPは指定個所を「並列に実行」してくれるが、「正しさの保証」はしてくれない ⇒プログラマの責任

sが共有変数のとき、s = s+1 と s = s+1 を並列に実行するとどうなるか？

このプログラムは正しいか？

```
   int s = 0;
#pragma omp parallel
   {
       s = s+1;
   }
```

ここがs++でも話は同じ

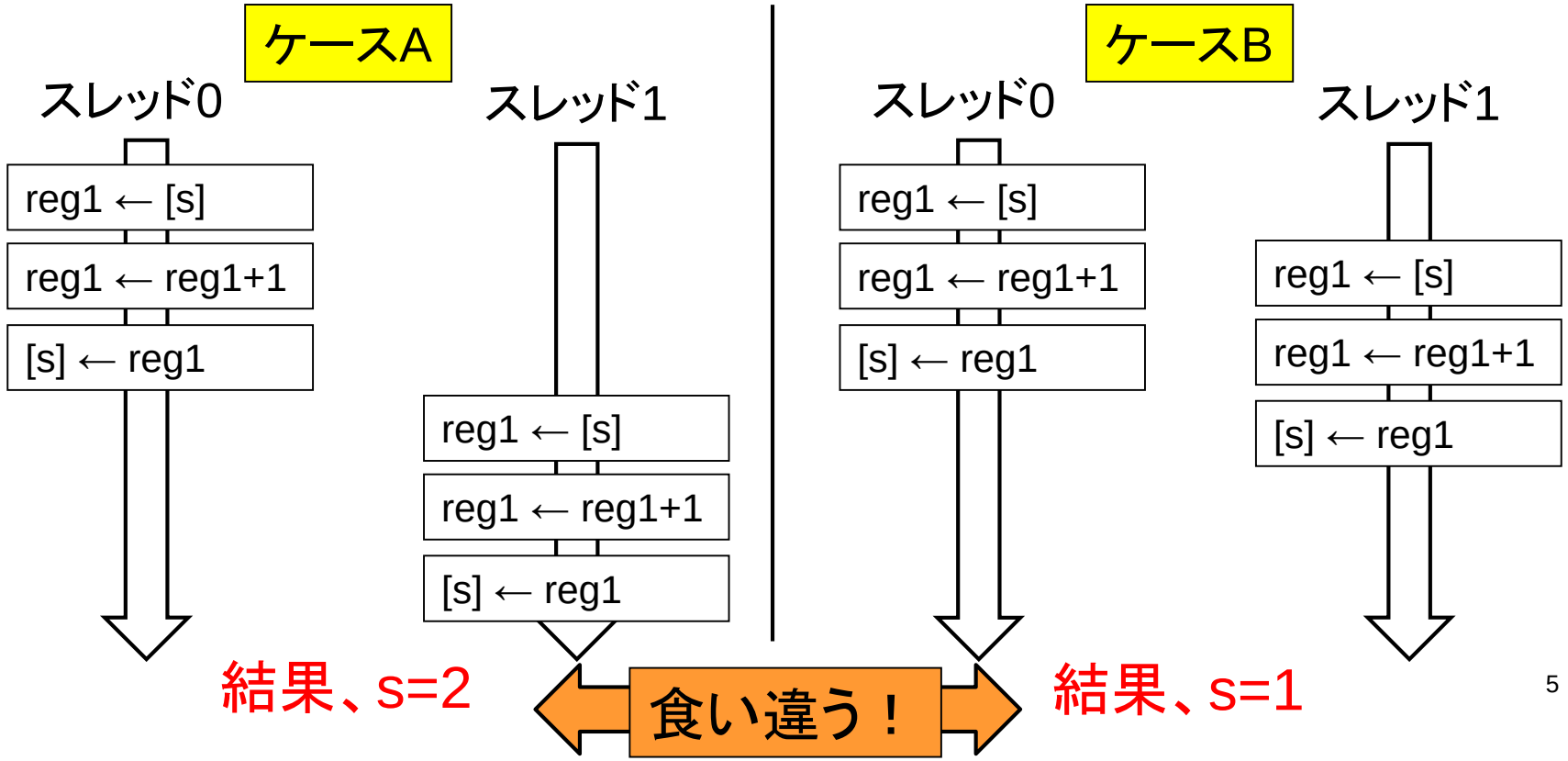


2スレッドがそれぞれsを1増やすので、結果はs=2 ??

「正しい」並列ソフトウェアの敵: Race condition

Race condition (競合状態)：処理結果が、イベントの順序やタイミングにより、予期しないことになること

- 共有メモリの場合、複数スレッドが共有変数に(調停せずに)アクセスすると起こる。s=s+1の例では：



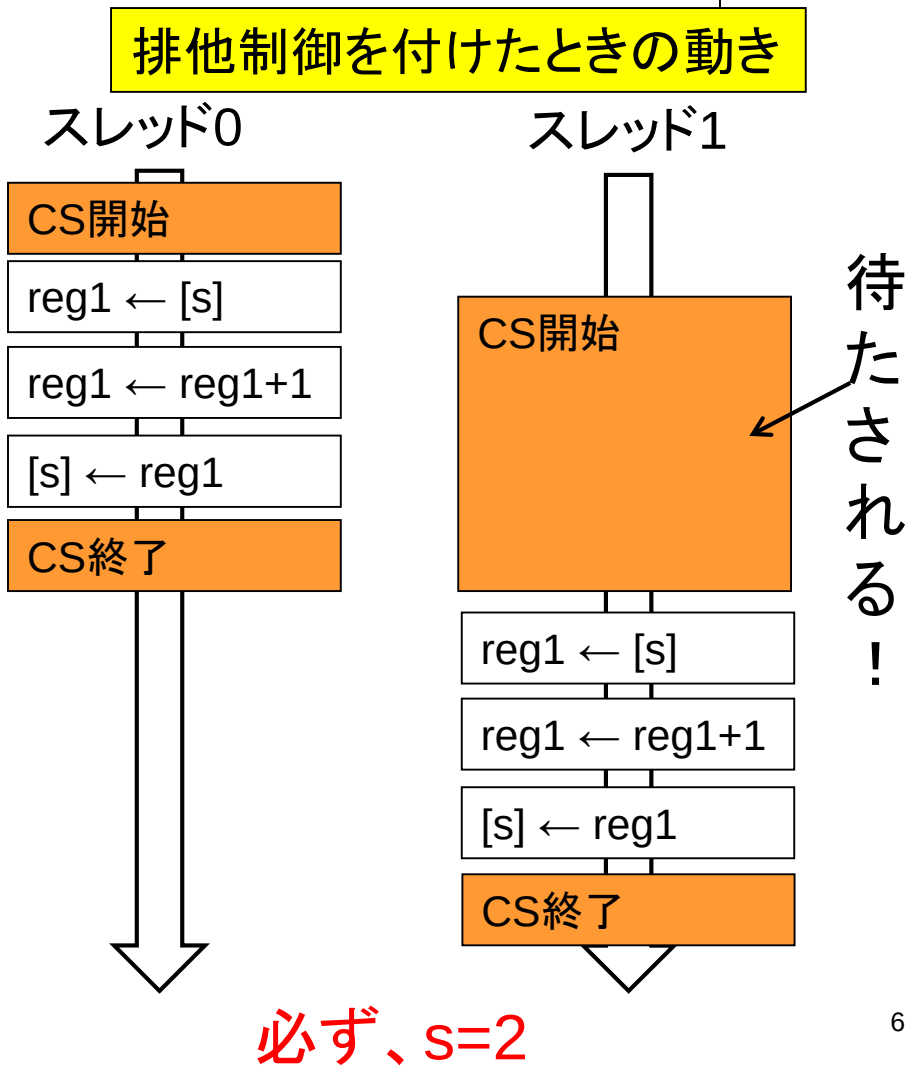
Mutual exclusion

Mutual exclusion(排他制御):

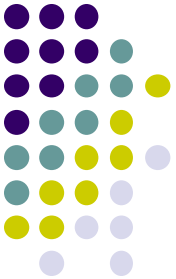
あるコード部分に、同時に1スレッドしか入れないようにする

- 1スレッドしか入れない部分をcritical sectionと呼ぶ

⇒　右の例では競合状態はなくなり、必ずs=スレッド数となる



OpenMPにおける Mutual exclusion



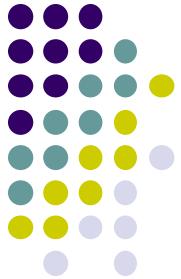
OpenMPでは「omp critical」をつけると、直後の文・ブロックはクリティカルセクションになる

```
    int s = 0;
#pragma omp parallel
    {
#pragma omp critical
        {
          s = s+1;
        }
    }
```

使用例:
~endo-t-ac/ppcomp/15/race-condition/

- rc-bad: 間違ったプログラム
- rc-good: 正しいが、遅いプログラム
- rc-fast: 正しく、速いプログラム

Critical 指示文について補足情報



- プログラム文中で複数criticalセクションがあるときの挙動は？

⇒ どれかのcriticalセクションに入れるのは，同時に1スレッドのみ

補足：

#pramga omp critical (name)

という文法があり，「同じnameを持ったcriticalセクションに入れるのは，同時に1スレッドのみ」

デフォルトでは，「名無し」という名前を持つとみなされる

「速い」並列ソフトウェアの敵：ボトルネック

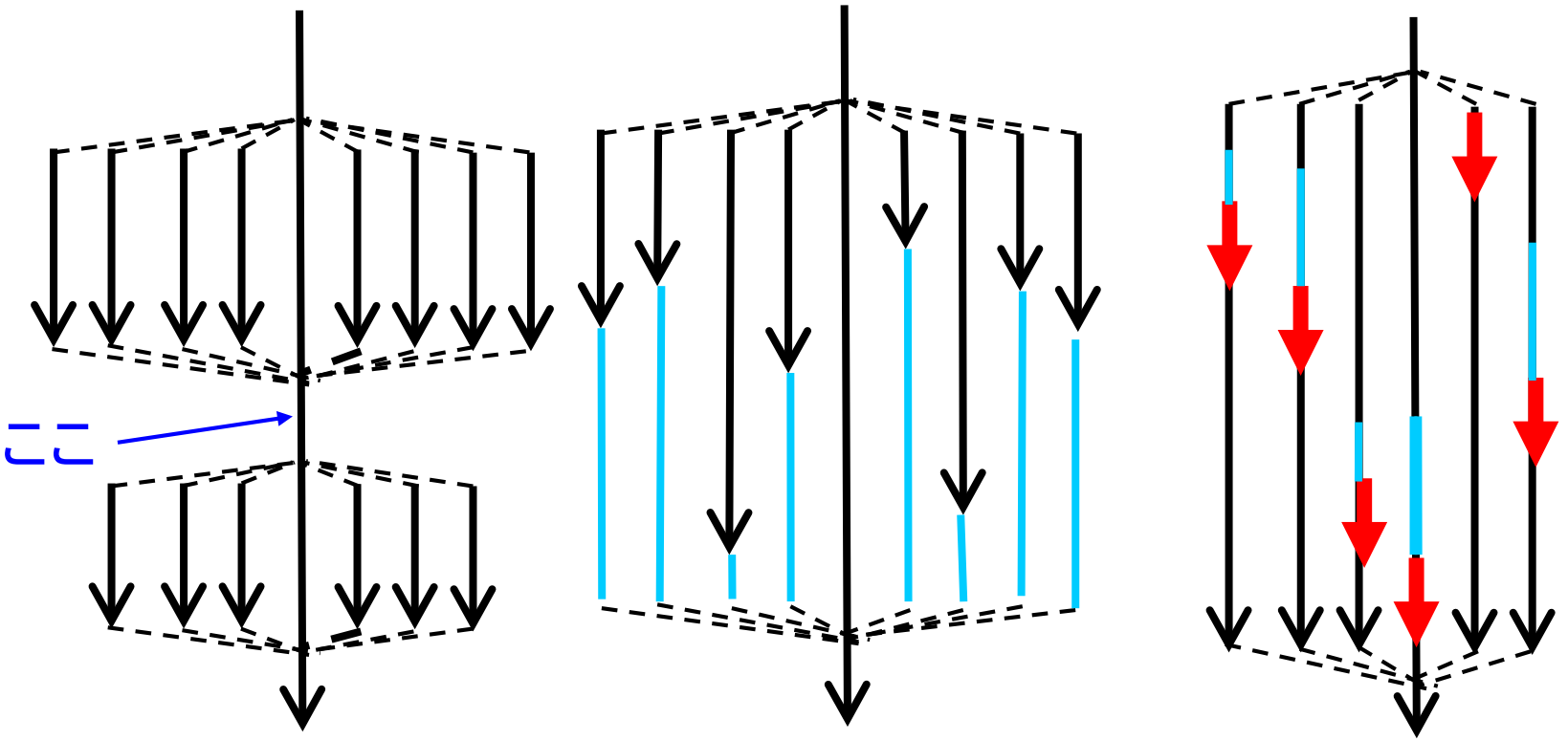
- 多くのアルゴリズムは、並列化できる場所とできない場所を含む⇒できない場所をボトルネックと呼ぶ

さまざまなボトルネック

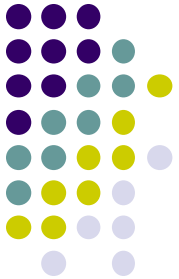
逐次部分による
ボトルネック

負荷不均衡による
ボトルネック

Critical section
によるボトルネック



ほかにも、アーキテクチャ上のボトルネックなども



アムダールの法則

- アムダールの法則(Amdahl's law)
  - アルゴリズムのうち、
    - 並列実行できる部分の割合をα
    - 逐次にしか実行できない部分の割合を1-α

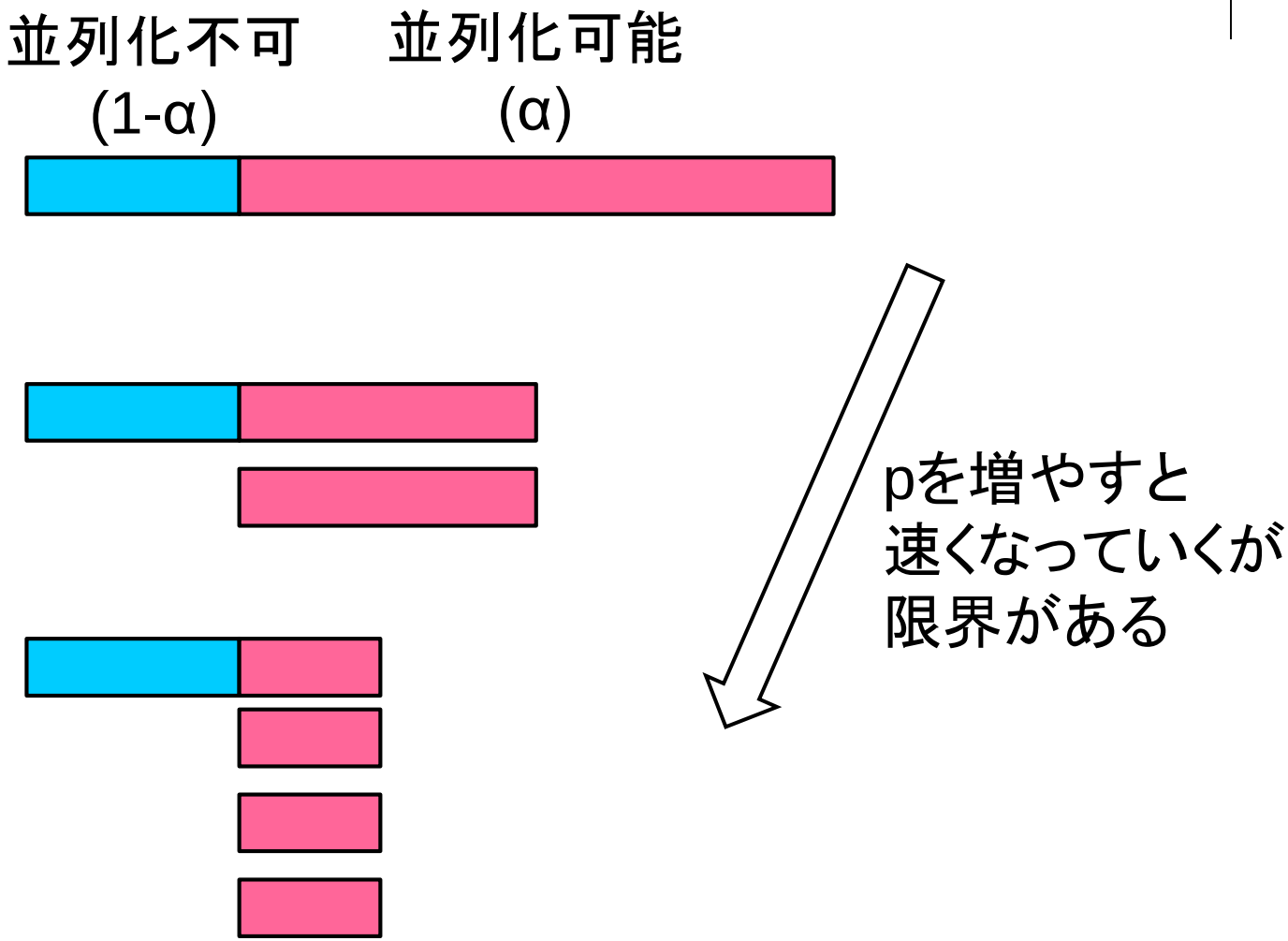
⇒ pプロセッサでの相対実行時間(逐次=1)は

$(1 - \alpha + \alpha / p)$

速度向上率は逆数の$1 / (1 - \alpha + \alpha / p)$

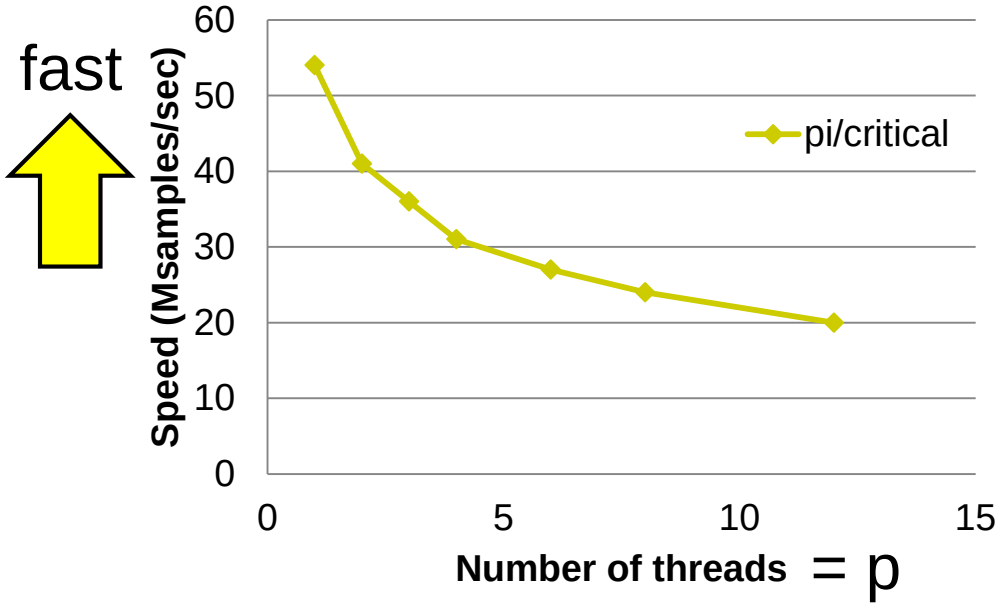
  - ボトルネックだらけのプログラムは $\alpha \fallingdotseq 0$ なので、どんなにpを増やしても速くならない

アムダールの法則



事態はアムダールの法則より さらに悪い

- piサンプル(omp版)を、reductionの代わりにcriticalを使った場合の性能
  - TSUBAME2の1ノードで実行, PGIコンパイラ12.8



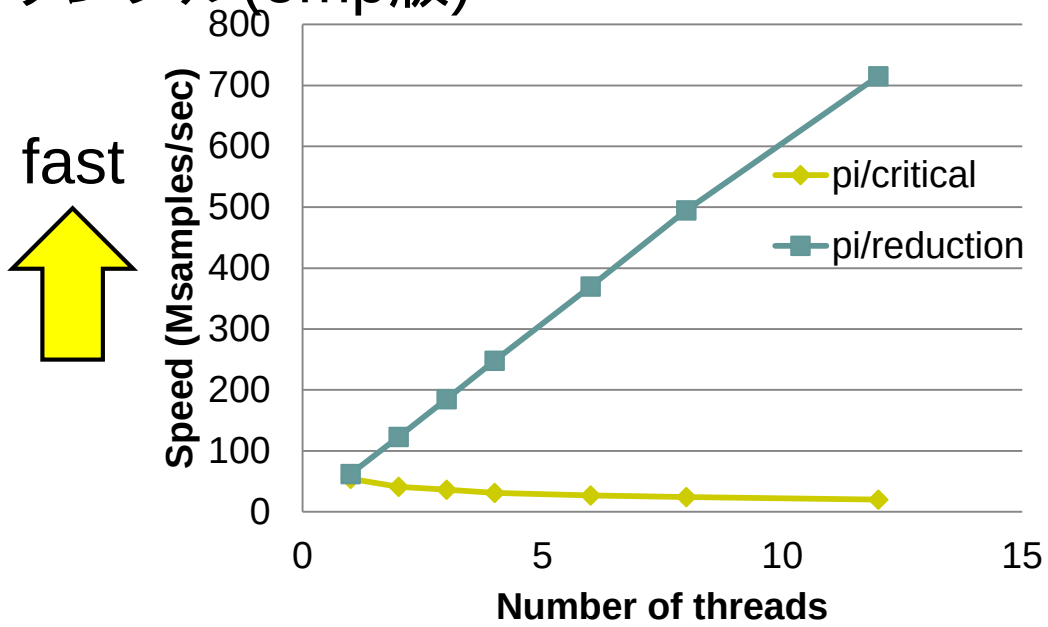
アムダールの法則どおりなら、どんなにαが悪くても(0に近くても)、pに対して単調増加のはず

⇒実際には遅くなってしまっている！

※ この理由の説明のためには、キャッシュコヒーレンシなどの理解が必要

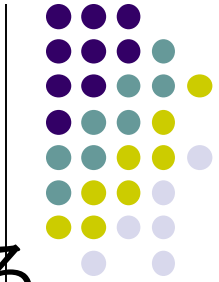
CriticalとReductionの性能比較

- piサンプル(omp版)



- どちらも意味は同じで、正しいプログラム
- Reductionのほうがはるかに速く、12スレッドどうしだと35倍もの差
- pを増やすと性能向上することを、スケーラブルであるという

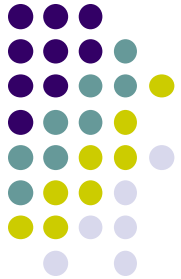
CriticalとReductionの関係



どちらも競合状態を防ぎつつ演算をすることができる

- Reductionでできること ⊂ Criticalでできること
- Critical:
  - 一般的な排他制御であり、クリティカルセクションの中にはどんな計算でも記述できる
    - 例: Wikipedia「排他制御」にリスト操作の例
  - Reductionで記述可能なことは、criticalでも可能・ただし遅い
- Reduction:
  - #pragma omp forでのみ可能
  - 可能な計算は、共有変数への加算・積算・AND・ORの繰り返しのみ (reduction処理)
  - はまれば速い

プログラムはソフトウェア部品から成り立っている



- 一から十まで、自分で作ったソフトウェアということはほぼない

  例:

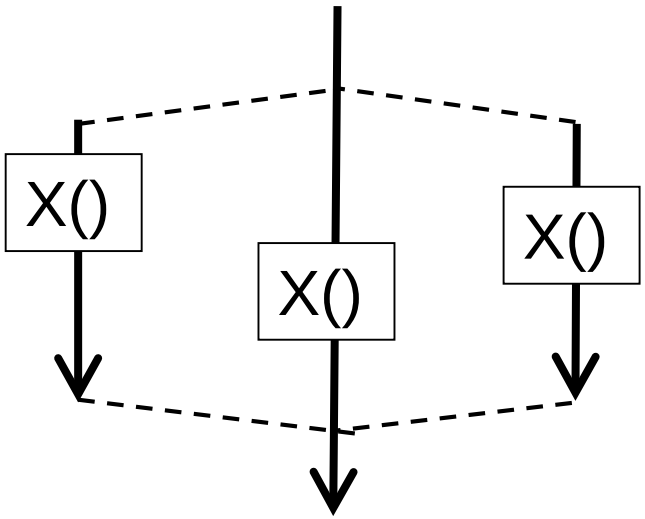
  - 標準ライブラリ
  - オープンソースのライブラリ
  - (フレームワーク)

  このような場合でもマルチスレッドでokなのか？

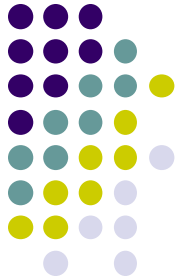
  マルチコアは広まる前に設計されたライブラリでは問題発生することも

Thread safe

- 複数スレッドが、排他制御なしに関数X()を呼び出しても、正常動作するとき、「関数Xはスレッドセーフ(thread safe)である」と呼ぶ
  - Multithread safe, MT-safeも同じ

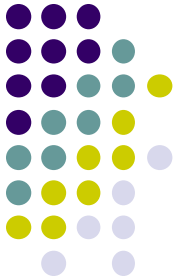


標準ライブラリ関数はthread safeか？



- スレッドセーフなもの
  - printf, malloc, gethostname…
    - mallocは、共有ヒープに対して内部で排他制御をしていると予想される
  - rand_r, strtok_r, gethostbyname_r…
- スレッドセーフでないもの
  - rand, strtok, gethostbyname…

randがthread safeでない理由



- rand(): 擬似乱数を返す
  - あるseedによりsrand(seed)で初期化された後、randが繰り返し呼ばれると、同じ乱数列を返す

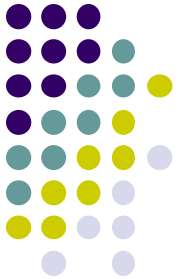
srand(123);
A = rand(); → 128959393
A = rand(); → 1692901013
A = rand(); → 436085873
A = rand(); → 748533630
:

srand(456);
A = rand(); → 1929505231
A = rand(); → 1800653403
A = rand(); → 1030306120
A = rand(); → 2023122793
:

srand(123);
A = rand(); → 128959393
A = rand(); → 1692901013
A = rand(); → 436085873
A = rand(); → 748533630
:

seedが同じなら、同じ乱数列

randがthread safeでない理由



- 予想されるsrand, randの実装

```
int g_seed;

void srand(int seed)
{   g_seed = seed;  }

int rand()
{
    int res = XXX(g_seed);
    g_seed = YYY(g_seed);
    return res;
}
```

共有変数を使っている点が、thread safeでなくしている！

“man rand” より

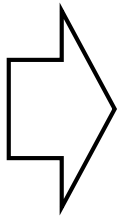
The function rand( ) is not reentrant or thread-safe, since it uses hidden state that is modified on each call.

rand単体の中で排他制御してもダメ！
守りたいのは複数のrand()からなる列なので

randのthread safe版: rand_r

- int rand_r(int *seedp)
  - seedは呼び出し側で管理すること
  - マルチスレッドプログラムからrandを使いたいなら、代わりにrand_rを使って書き換えること

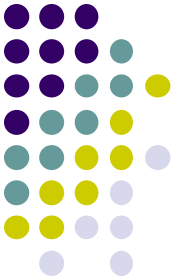
```
srand(123);
A = rand();
A = rand();
A = rand();
A = rand();
    :
```



```
int seed = 123;
A = rand_r(&seed);
A = rand_r(&seed);
A = rand_r(&seed);
A = rand_r(&seed);
    :
```

※サンプルプログラムpi_ompもrand_rを使っている

マルチスレッド版rand_r



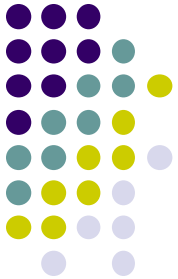
- 予想されるrand_rの実装

```
int rand_r(int seedp)
{
    int res = XXX(*seedp);
    *seedp = YYY(*seedp);
    return res;
}
```

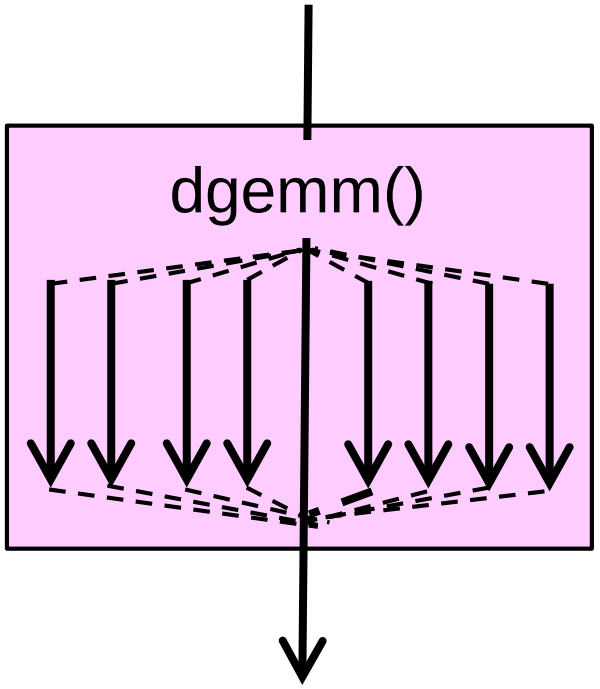
共有変数なし。
状態(今のseed)をためて
おくのはユーザの責任とする

ライブラリ関数を「使う際」も「作る際」も、
Thread-safeか否かに
気をつけるべき

別のケース:ライブラリ関数内部で並列化されている場合



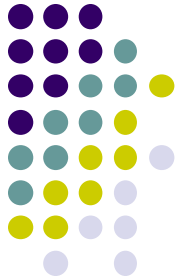
- Intel MKL, AMD ACML, GotoBLASなど
  - 行列演算などをとても速く実行するライブラリ



ユーザプログラムは並列化を意識しなくても、勝手にマルチコアを利用して速くなってくれる

複数スレッドがdgemmを呼んだらどうなるか？は実装・バージョン次第・・・

本授業のレポートについて

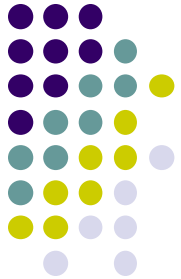


- 各パートで課題を出す。2つ以上のパートのレポート提出を必須とする

  予定パート:

  - OpenMPパート
  - MPIパート
  - GPUパート

OpenMPパート課題説明 (1)



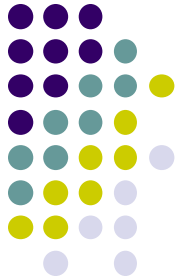
以下の[O1]—[O3]のどれか一つについてレポートを提出してください

[O1] diffusionサンプルプログラムを、OpenMPで並列化してください.

オプション:

- 配列サイズや時間ステップ数を可変パラメータにしてみる。引数で受け取って、配列をmallocで確保するようにする、など。
- より良いアルゴリズムにしてみる。ブロック化・計算順序変更でキャッシュミスを減らせないか？

OpenMPパート課題説明 (2)



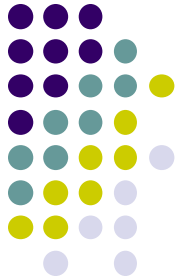
[O2] sortサンプルプログラムを、OpenMPで並列化してください.

- 注意：OpenMP3.0以上のtask対応コンパイラである必要
  - TSUBAMEではpgcc (-mpつき)や icc (-openmpつき)

オプション：

- クイックソート以外のアルゴリズムではどうか?
  - ヒープソート？マージソート？

OpenMPパート課題説明 (3)



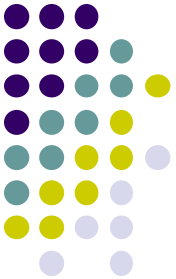
[O3] 自由課題：任意のプログラムを，OpenMPを用いて並列化してください．

- 単純な並列化で済む問題ではないことが望ましい
  - スレッド・プロセス間に依存関係がある
  - 均等分割ではうまくいかない、など
- たとえば，過去のSuperConの本選問題
  http://www.gsic.titech.ac.jp/supercon/
  たんぱく質類似度(2003)，N体問題(2001)・・・
  入力データは自分で作る必要あり
- たとえば，自分が研究している問題

課題の注意

- いずれの課題の場合も，レポートに以下を含むこと
  - 計算・データの割り当て手法の説明
  - TSUBAME2などで実行したときの性能
    - プロセッサ(コア)数を様々に変化させたとき
    - 問題サイズを様々に変化させたとき(可能な問題なら)
    - 「XXコア以上で」「問題サイズXXX以上で」発生する問題に触れているとなお良い
  - 高性能化・機能追加などのための工夫が含まれているとなお良い
    - 「XXXのためにXXXをしてみたが高速にならなかった」のような失敗でもgood
  - 作成したプログラムについても、zipなどで圧縮して提出
    - 困難な場合，TSUBAME2の自分のホームディレクトリに置き，置き場所を連絡

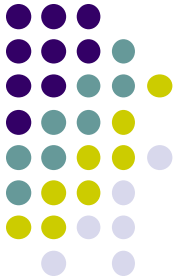
課題の提出について



- OpenMPパート提出期限
  - 6/8 (月) 23:59 (変更しました)
- OCW-i ウェブページから下記ファイルを提出のこと
- レポート形式
  - レポート本文：PDF, Word, テキストファイルのいずれか
  - プログラム：zip形式に圧縮するのがのぞましい

- OCW-iからの提出が困難な場合、メールでもok
  - 送り先： ppcomp@el.gsic.titech.ac.jp
  - メール題名： ppcomp report

次回: 6/8(月)



- MPIによる分散メモリ並列プログラミング(1)
  - 複数プロセスによる並列化 → 複数コンピュータを利用可能
  - メッセージパッシング