システムプログラム
第11回

創域理工学部 情報計算科学科
松澤　智史

本日の内容

- 並列計算・並列処理
- 並列プログラム

復習：並行(Concurrent)と並列(Parallel)

- 並行処理
  - 複数のタスクを1人で，見せかけ上同時進行しているように見える処理
  - 目的はシステムの応答性の向上
    - 副作用的に速度の向上のケースもある
    - 詳細は第5回のジョブスケジューリング等を参考

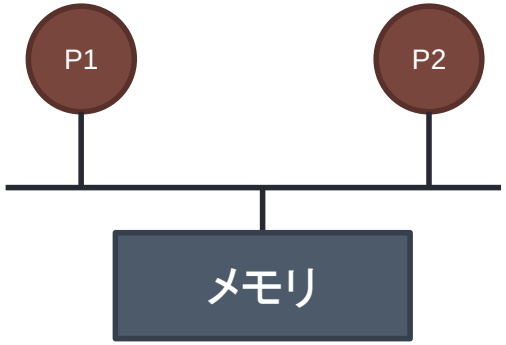
- 並列処理
  - 実際に処理する人数がタスクと同じ数で行う（実際に同時に行う）処理
  - 目的は速度の向上

計算機の高速化の歴史

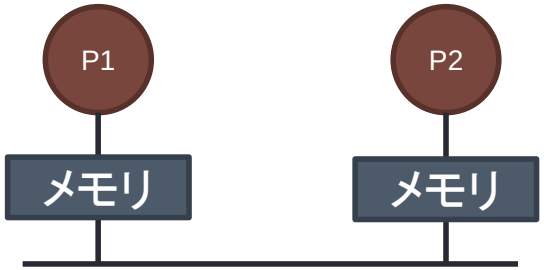
- プログラムは逐次的に計算される
  - 命令列はCPUで一度に一つずつ実行される
- プロセッサの高速化が性能強化につながる
  - クロック数増加のための工夫など
- プロセッサのクロック数の上昇には限界（問題）が見え始める
- 複数のプロセッサによる性能強化の路線へ移行する
  - マルチプロセッサは当初スーパーコンピュータなどの高価（大規模）システムでの設計であった
  - プロセッサ自体に複数の計算用コア（マルチコア）を導入する

マルチプロセッサ

- 共有メモリ型マルチプロセッサ
  - どのプロセッサからも同様に
    アクセス可能なメモリを持つ



- メッセージパッシング型マルチプロセッサ
  - 共有メモリを持たず
    メッセージ交換で処理を行う



マルチコア

- コア
  - チップ内の計算処理を行う中核の装置
  - 現在の汎用CPU：1プロセッサ内に多数のコアが入っている
- マルチコア
  - コア数2~16ぐらいのプロセッサ
  - 現在のPC，スマホなどはほぼマルチコアのプロセッサ
- メニーコア
  - コア数64~数千
  - アクセラレータとして使われることが多い(GPUなど)
    - NVIDIA RTX3090は1万超え

並列計算機

- もともとはスーパーコンピュータ用の設計

- 設計はもちろん，使用するソフトウェアにも特別な工夫が必要
  - 並列計算専用のプログラムの知識や技術が必要

並列OS

- 考慮する次元が異なる
  - 単一プロセッサ用OS：時間次元(並行処理)
  - 並列OS: 時間次元と空間次元
- もともとはスーパーコンピュータ用OS
- 近年はマルチプロセッサが汎用機にも普及したことにより
  汎用OSも並列OSの機能を備える

並列計算の可能性と問題点

- 理想的には，プロセッサの数に比例した性能向上が可能
  - 理想的な環境 = 複数のプロセッサが100%の性能を発揮する
- 複数のプロセッサ性能を100%使い切るのは難しい
  - 他のプロセッサの結果に依存する計算の排除
  - 各プロセッサの仕事を均等に割り当てる工夫
- 並列化に必要な処理（オーバーヘッド）が発生する
  - 並列化の恩恵をオーバーヘッドが上回るケースもある

プログラム中，複数のプロセッサで均等に処理できる部分の割合をプログラムの並列度と呼ぶ

アムダールの法則とグスタフソンの法則

- 並列アルゴリズムは対象のアルゴリズムをどれだけ並列化できるかに依存
- 並列化できない部分が並列化の性能向上を制限する
- アムダールの法則
  - $S = \frac{1}{(1-P)}$ Sはプログラムの性能向上率，Pは並列可能比率
- グスタフソンの法則
  - $S(P) = P - \alpha(P - 1)$ Pはプロセッサ数，Sは性能向上，αは並列化できない部分

並列度の高いプログラム，低いプログラム

```
int a = 100;
int b = a;
int c = b;
```

このプログラムは3行とも並列処理できない

```
int a = 100;
int b = a;
int c = a;
```

このプログラムは2行目と3行目を並列処理可能

```
int a = 100;
int b = 100;
int c = 100;
```

このプログラムは3行とも並列処理可能

逐次実行の場合はどれも変わらない（3回の代入）が，
並列処理の場合は，3つ目のプログラムが性能が高い

分割と依存関係

- 並列実行を考慮する際，大きく二つの視点で考える
  - 分割可能の場所
    - タスク分割
    - データ分割
  - 分割した際の依存関係
    - 関連するタスク
    - 順序関係

※ `a = 100;`と `b = a;`は順序関係が発生する

タスク並列

- 各タスク間に依存関係のないものを集め並列化
- 独立したタスクが十分にあると性能向上
- タスク分割した数とプロセッサ数を意識する必要がある
  - プロセッサ4(コア4)で2つのタスクを並列にしても性能は2倍止まり

データ並列

- 前提:
  - 対象とするデータをサブデータへ分割可能なケース
  - サブデータの処理を行うことで元の全体データの処理になるケース

- 例
  - 配列のデータにすべて1加算する
    - 配列を分割して処理しても逐次でも結果は同じ

並列プログラミングとOpenMP

gccにはAPI(OpenMP)が標準装備されている

OpenMP

- Open MultiProcessing
- 並列環境と非並列環境でほぼ同一のソースコードを使用できる
- 共有メモリ環境においては導入が簡単
- 基本的にCPU計算の並列化が可能
  - gccの作り直し(再コンパイル)でGPU計算にも適用可能(omp v4.0以降)

似たようなAPIとしてOpenACCなどもある

※この講義では触れない

OpenMP ディレクティブ

- ディレクティブ
  - プリプロセッサ（コンパイル前）が処理する行
  - C言語(gcc)の場合 #で始まる行

- #pragma ディレクティブ
  - 環境依存するディレクティブ
  - 一般的なプログラムでは使用は推奨されない

- OpenMPのディレクティブ
  - #pragma omp ~

#pragma omp

使用例

```
#include <omp.h>
int main(){

#pragma omp parallel
    {
        /* 並列処理させたいプログラム */
    }
    return 0;
}
```

omp使用例

~

```
tusedls04$ cat omphello.c
#include <stdio.h>
#include <omp.h>

int main(){
#pragma omp parallel
  {
    printf("Hello from %d\n",omp_get_thread_num());
  }

}
tusedls04$ gcc omphello.c -o omphello -fopenmp
tusedls04$ ./omphello
Hello from 2
Hello from 0
Hello from 5
Hello from 9
Hello from 4
Hello from 8
Hello from 7
Hello from 3
Hello from 6
Hello from 1
tusedls04$
```

忘れないこと

大学のCentOSでは10プロセッサ(10スレッド)が立ち上がる

#pragma ompの解説

- #pragma行はコンパイラに対する指示文
- OpenMPコンパイラのみが処理する文
  - 他のコンパイラは無視

- 並列処理のスレッド数は指定せずとも
  実行開始時にその計算機のCPU数で開始される

- シェルの環境変数で明示的に指定することも可能
  - OMP_NUM_THREADS　変数

for文へ適用

~

```
tusedls04$ cat ompfor.c
#include <stdio.h>
#include <omp.h>

int main(){
  int i;
#pragma omp parallel for
  for(i=0;i<20;i++){
    printf("i=%d, t_number=%d\n",i, omp_get_thread_num());
  }
}
tusedls04$ gcc ompfor.c -o ompfor -fopenmp
tusedls04$ ./ompfor
i=16, t_number=8
i=17, t_number=8
i=12, t_number=6
i=13, t_number=6
i=10, t_number=5
i=11, t_number=5
i=2, t_number=1
i=3, t_number=1
i=18, t_number=9
i=19, t_number=9
i=4, t_number=2
i=5, t_number=2
i=0, t_number=0
i=1, t_number=0
i=8, t_number=4
i=9, t_number=4
i=6, t_number=3
i=7, t_number=3
i=14, t_number=7
i=15, t_number=7
tusedls04$
```

for文の場合はリージョンで
括る必要なし

10プロセッサがfor文を2回ずつ,
合計20回実行の確認ができる

並列度の低い例

```
int tmp;
  #pragma omp parallel for
  for (i = 0; i < 10000; ++i){
    tmp = array[i];
    array[i] = function(tmp);
  }
```

この例の変数tmpは共有変数であり，2行目の参照結果が
必ずしもそのスレッドで実行された1行目の代入結果を使う保証はない

対処法

対処法

```
#pragma omp parallel for
  for (i = 0; i < 10000; ++i){
    int tmp;
    tmp = array[i];
    array[i] = function(tmp);
  }
```

並列度を意識した記述を心掛けると
並列計算への適用も見込めるプログラムとなる

または

```
#pragma omp parallel for private(tmp)
  for (i = 0; i < 10000; ++i){
    tmp = array[i];
    array[i] = function(tmp);
  }
```

メッセージパッシング型プロセッサ用環境

- MPI(Message Passing Interface)
  - 複数のCPUが情報(メッセージ)を送受信することで協調動作をサポート
  - 言語を問わず利用できる
  - プログラマが細密なチューニングを行う必要がある
    - メリットでもデメリットでもある
- OpenMPI(※OpenMPとは異なるので注意)
  - MPI用のライブラリ
  - スーパーコンピュータ京でも用いられていた
- MPICH
  - メッセージパッシングの標準の一つ
  - Linux含むUnix系OSで使用可能

まとめ

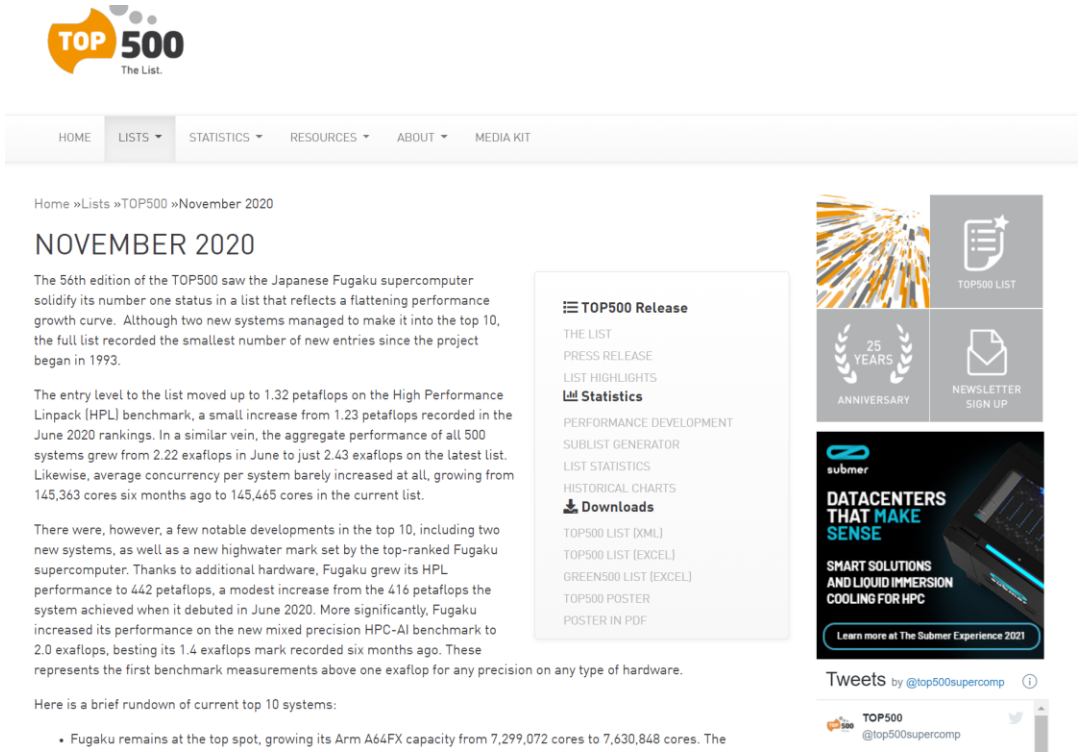
並列計算

- プロセッサを同時に走らせることで速度向上を目的とした処理
- 並列度
- タスク並列とデータ並列
- #pragma omp

おまけ

スーパーコンピュータの歴史

- TOP500
  - https://www.top500.org/
  - 計算機ランキングTOP500が掲載
  - 1993年～
  - 毎年6月と11月に更新



スーパーコンピュータの歴史

- スーパーコンピュータの歴史は計算機の歴史
  - 基本的にスパコン部門は赤字
- 高速計算を行うためには
  - CPU，メモリなどの演算に必要なハードウェアの強化・進化
  - 冷却システム・電力効率の進化
  - 並列計算のソフトウェアの進化
  - 通信路の高速化
- 様々な試行錯誤・失敗を重ねてチャレンジした
  研究者・技術者・組織の歴史を追ってみよう

データ可視化

- 近年注目を浴びている
- アニメーションや斬新なグラフなどが登場

スーパーコンピュータTOP500の推移

質問あればどうぞ