チューニング技法入門: Part 1
チューニングの基礎

一般財団法人高度情報科学技術研究機構
Research Organization for Information Science and Technology (RIST)

質問は随時受け付けます
E-mail (hpc-seminar@rist.or.jp) もご利用ください

ver. May2024

本資料の主要な参考文献

- [A] 青山幸也「チューニング技法虎の巻」(平成28年8月1日版)
  - HPCIポータルサイトから入手可能 (https://www.hpci-office.jp/events/seminars/seminar_texts)
- [HW] G. Hager and G. Wellein, Introduction to High Performance Computing for Scientists and Engineers (CRC Press, 2011)
- [S] 寒川光, 藤野清次, 長島利夫, 高橋大介「HPCプログラミング」(オーム社, 2009)
  - 備考: 寒川光「RISC超高速化プログラミング技法」 (共立出版, 1995)も参照
- [UO] 内田啓一郎,小柳滋「コンピュータアーキテクチャ」(改訂2版) (オーム社, 2019)

他の参考文献は随時記載
参照箇所の表記
[文献記号]:ページ番号 (or 節 or 章, etc.)
例) 経過時間 ([A]:1-2)

まえがき

- プログラムの性能に対する考え方
  - 本資料: 時間に着目 →プログラムの全体/部分の実行時間をどれだけ削減できるか
- 対象のプログラミング言語: FortranおよびC/C++ → コンパイラ言語
  - FortranとC/C++との共通事項を主に説明
    - 差異がある場合は明記 (例: 多次元配列のアクセスパターン)
    - プログラミング言語に特化した話題: 随時補足 & Appendix
  - FortranとC/C++のコード例を(可能な限り)併記
- 具体例 (モデル) による理解
  - チューニング技法の理解: ハードウェアの知識が要求
  - 本資料:本質を取り出したモデルで理解 (厳密さは犠牲に) [A]
    - より本格的な書籍を理解するための土台の構築

サンプルコード: scalar-sample_YYYYMMDD.tar.gz
https://www.hpci-office.jp/events/seminars/seminar_texts

アウトライン

- part1: チューニングの基礎
  - 経過時間とCPU時間
  - ホットスポットとチューニング
  - チューニングの第一歩: コンパイラの活用
  - 性能測定の方法
  - ホットスポットの特定
  - 性能分析の手順

Appendix
- Fortranに関する補足
- C/C++に関する補足
- Perfによる性能測定
- デバッグとコンパイルオプション
- 演算例外について

経過時間とCPU時間 (1/2)

- 経過時間
  - プログラムの実行に要する実時間 (Elapsed time, real time)
    - プログラム全体の場合: 計算ジョブの開始から実行終了までの時間

> - 一般に: 別のジョブが動く場合 → 待ち時間発生 → 経過時間の遅れ
> - ここでは:「別のジョブが全く動いていない」という設定で考察

- CPU時間
  - ユーザーCPU時間 (user CPU time)
    - ユーザプログラム自体に関するCPUの処理時間
  - システムCPU時間 (system CPU time)
    - オペレーティングシステム (OS) に関するCPUの処理時間 (例: 入出力割込み) ([UO]:4章)
      - ✓通常はわずかのため無視可能

> ユーザCPU時間とシステムCPU時間を正確に区別することは難しい → OSの動作に強く依存
> - D. A. Patterson and J. L. Hennessy (訳: 成田光彰), コンピュータの構成と設計: ハードウェアとソフトウェアのインターフェース 第5版 (日経BP, 2014) 第1章

経過時間とCPU時間 (2/2)

- 初歩的な性能分析: 経過時間とCPU時間の差を評価
  - プログラム全体の時間を測定
    - 経過時間とCPU時間が一致するとは限らない
      ✓典型的な挙動: [経過時間] > [CPU時間] (注意: プロセッサコアを1個のみ使用する場合)
    - 差の部分: CPUの処理が大きく関与しないと推測 → I/O処理の時間とみなす
      ✓例: ユーザプログラムによるI/O処理 → ファイルや画面への入出力等
    - I/O時間を定義できる: [I/O時間] = [経過時間] - [CPU時間]

> - I/O処理にもCPU時間は必要 → 厳密にI/O時間を取り出そうとは考えないこと
> - ページングによるI/O時間の可能性 ([UO]:3章) → ディスク (スワップ領域)とメモリ間の入出力

  - 分析の結果
    - CPUの処理とI/Oの処理のどちらでプログラムが支配されているかを検討できる

ホットスポットとチューニング (1/4)

- ホットスポットとは ([A]:1-3, [HW]:2.1節)
  - プログラムの実行時間の主要部
    - 本資料: プログラム内でCPU時間を多く消費する部分と定義

      ✓単一のプロセッサコアに関するチューニング技法における着眼点
      - cf. I/O時間が顕著 → I/Oチューニングの検討

Case study: 3重ループ

- 最も内側の部分 (処理3) が全体のほとんどを占める

→ ホットスポット

- 処理1の寄与はわずか

→ 無視してよい

Fortran

```
a=0; b=0; c=0
do k = 1, 100
  a = a+1
  do j = 1, 100
    b = b +1
    do i = 1, 100
      c = c + 1
    end do
  end do
end do
```

C

```
a=0; b=0; c=0;
for(k=0; k<100; ++k){
  a += 1;
  for(j=1; j<100; ++j){
    b += 1;
    for(i=0; i<100; ++i){
      c += 1;
    }
  }
}
```

処理1: 100回

処理2: 10000回

処理3: 10^6回

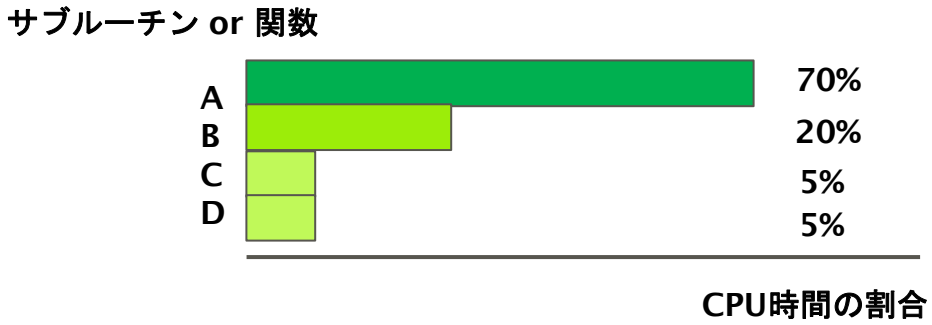
ホットスポットとチューニング (2/4)

- 基本的な戦略: ホットスポットのみをチューニング
  - 最小の努力で最大の効果を上げる
  - ホットスポット以外はできる限り触らない

Case study: チューニングのターゲット

- サブルーチンA or 関数AのCPU時間を0.5倍 → 全体で35%の性能向上
- A以外のCPU時間を10^-6倍 → 全体で高々30%の性能向上

$$\gamma = 0.35$$
$$\gamma < 0.3$$



性能向上の目安 $\frac{T^{\mathrm{CPU}}_{\mathrm{tuning}}}{T^{\mathrm{CPU}}_{\mathrm{original}}} = 1 - \gamma$

ホットスポットとチューニング (3/4)

- 補足: 並列性能における性能の着眼点
  - CPU時間のみに着目 → 問題あり
    - 使用プロセッサコア数が増加するとCPU時間も増加 (コアからの寄与の総和)
  - 単一プロセッサコアの場合とは異なる着眼点が必要
    - 着眼点1: 計測区間を定めた経過時間に着目 → I/O, 通信, 演算等からの寄与を区別する
    - 着眼点2: CPU時間の分布に着目→ プロセッサコア間の処理のインバランスを把握する
  - 実行時間の主要部以外も重要になる傾向
    - 典型例: 使用コア数増加で時間が減少しない (ほぼ一定) or 逆に増加する
      → スケーラビリティのボトルネックと呼ぶべき部分

参考情報 (RISTの支援活動)
高速化ノウハウ集: https://www.hpci-office.jp/user_support/tuning_knowhow

ホットスポットとチューニング (4/4)

- チューニングの手順
  - (0) 入力データの準備
    - メモリにおさまる配列サイズの設定
    - 計算の実行時間の短縮 (例: 総ステップ数を短く, 収束条件を緩く, etc.)
      - ✓配列サイズ縮小: 好ましくない (ホットスポット変更の可能性)
  - (1) 性能 (実行時間)の測定とホットスポットの特定 (part.1)
  - (2) ホットスポットに対してチューニング
    - 0. コンパイラオプションの検討と診断レポートの確認 (part.1)
    - 1. 適切な数値計算ライブラリの利用 (part.3)
    - 2. 高価な処理 (除算や組込関数) , 無駄な計算の削減, インライン展開の適用 (part.3)
    - 3. ループにおける配列アクセスパターンの検証 (part.2)
    - 4. ループにおけるメモリからのデータ移動の削減 (part.2, part.3)
    - 5. SIMD処理の利用の検討 (part.3)
  - (3) 結果が正しいこと, 性能向上の有無を確認
    - 結果の不正 or 性能向上の不足 → (2)へ戻る

(2) ←→ (3)では
バージョン管理を徹底

参考情報 (RISTの支援活動)
高速化ノウハウ集: https://www.hpci-office.jp/user_support/tuning_knowhow

チューニングの第一歩: コンパイラの活用 (1/3)

- コンパイルとは?
  - 高級プログラム言語で書かれたプログラムを機械向き言語のプログラムに翻訳すること
    - 翻訳のときに「最適化」が行われる (かもしれない)
      - ✓最適化=コンピュータへの命令に適した形にプログラムが自動的に整形される

        - Peter van der Linden (訳: 梅原系) 「エキスパートCプログラミング: 知られざるCの真相」 (アスキー出版,1996)
        - “The Compiler Design Handbook: Optimizations and Machine Code Generation” 2nd ed, Edited by Y. N. Srikant and P. Shankar (CRC Press, 2007)
        - 中田育男 「コンパイラ: 作りながら学ぶ」(オーム社,2017)

  - コンパイル時に指定するオプション (コンパイルオプション)
    - 例: 使用する計算機 (CPU) に適したプログラムを生成するための指定 → 最適化オプション
    - 例: デバッグ向けの指定
      - ✓[参照] Appendix:デバッグとコンパイルオプション

チューニングの第一歩: コンパイラの活用 (2/3)

- 最適化に関するコンパイルオプション
  - 性能向上に関する操作を指定:演算順序の変更, 無駄な計算の削減, etc. (“自動的に”チューニング*)

*: cf. 自動チューニング (AT) について: 「ソフトウェア自動チューニング」 (森北, 2021)

  - 最適化オプションの種類
    - “-O” (Optimizeの意味) による指定が多い
      - ✓大きい数字 → 最適化レベルの上昇 (例: -O0 < -O1 < -O2 < -O3)
        - より”aggressive”なオプションが用意されていることもある (例: -O3 < -Ofast)
    - 最適化レベルの上昇 → コンパイル時間は増大
      - ✓プログラムの文法エラー修正では「最適化なし (-O0) (or -O2レベル)」を推奨
    - 未設定時 (デフォルト) の最適化レベル → コンパイラに依存
      - ✓マニュアルで調べることを推奨
  - 最適化オプションの指針: そこそこ速くて安全
    - 典型的な選択: まずは-O2レベルからスタート
    - 高い最適化レベル → 副作用 (side effects)に配慮
      - ✓例: 計算結果の変化, コンパイラ自体のバグ (誤ったコード生成)

オプションの意味を調査するためのオプションの例
- GNU (https://gcc.gnu.org/)
  - gcc --help=optimizers
  - gcc -Q -O2 --help=optimizers
- LLVM (https://clang.llvm.org/docs/UsersManual.html)
  - clang --help
  - clang --print-supported-cpus
- Nvidia HPC SDK (https://docs.nvidia.com/hpc-sdk/index.html)
  - nvc --help=opt
  - nvc --help=target

チューニングの第一歩: コンパイラの活用 (3/3)

- 診断レポート (最適化レポート) に関するコンパイルオプション
  - コンパイラが適用した/適用できなかった最適化について表示
    - デフォルトでレポート表示はOFFのことが多い
      → マニュアルで表示方法を調べることを推奨
  - レポートを上手に使う → チューニング技法習得の近道 ([HW]:2章)
    - 診断レポートを表示させるオプションの例 (注意: コンパイラのバージョンに依存)
      ✓GNU: -fopt-info-all
        - https://gcc.gnu.org/
      ✓LLVM: -fsave-optimization-record -Rpass='.*' -Rpass-missed='.*' -Rpass-analysis=',*'
        - https://clang.llvm.org/docs/UsersManual.html
    - 最適化の内容 (副作用の有無), 最適化の阻害要因を把握
      → コンパイラを上手に利用するための情報を把握する

サンプルコード: prof-ex, simd-add, polynomial

性能測定の方法: プログラム全体

- timeコマンドの使用 ([A]:3-1)
  - UNIXやLinuxで提供されるコマンド → ジョブ全体の経過時間とCPU時間
    - ディスプレイに結果が出力
    - Windows*: “Linux-like”な環境経由で利用 → WSL, msys2

*:cf. PowerShellのMeasure-Commandも参照

```
program main
   write(6,’(“TEST”)’)
end program
```

Fortran

```
$ time ./a.out
TEST

real    0m1.036s← 経過時間
user    0m0.000s← ユーザCPU時間
sys     0m0.031s← システムCPU時間
```

```
#include <iostream>
int main(int argc, char **argv)
{
  std::cout<< “TEST” <<std::endl;
  return 0;
}
```

C++

```
$ time ./a.out
TEST

real    0m0.534s
user    0m0.000s
sys     0m0.093s
```

参考: GNUのtimeコマンド (/usr/bin/time) のマニュアル (GNU 1.7 ver.):

real %e
user %U
sys %S
%e (Not in tcsh.) Elapsed real time (in seconds).
%S Total number of CPU-seconds that the process spent in kernel mode.
%U Total number of CPU-seconds that the process spent in user mode.

性能測定の方法: プログラムの特定区間 [Fotran]

- 経過時間/CPU時間の測定方法

サンプルコード: timer

  - 基本的な考え方: 測定部分の前後で時間を取得→時間の差を計算
    - 経過時間: SYSTEM_CLOCK (Fortran90で提供)
      ✓カウンターの折り返しの効果を考慮する必要あり (= ielp1 > ielp2の場合がある)
    - CPU時間: CPU_TIME (Fortran95で提供)

```
integer :: ielp1, ielp2, icnt_rate, icnt_max
real(8) :: elp
call system_clock (ielp1,icnt_rate,icnt_max)
                経過時間を測定したい部分
call system_clock (ielp2,icnt_rate,icnt_max)
!カウンターの折り返し処理
if ( ielp1 .le. ielp2 ) then
   elp = (ielp2 - ielp1) / dble(icnt_rate)
else
   elp = (ielp2 + icnt_max +1 - ielp1) /
dble(icnt_rate)
end if

write(6, '("ELAPSED", 1F13.4,"SEC")') elp
```

```
real(8) :: cpu1, cpu2
call cpu_time (cpu1)
                CPU時間を測定したい部分
call cpu_time (cpu2)
write(6, '("CPU", 1F13.4,"SEC")') cpu2 - cpu1
```

性能測定の方法: プログラムの特定区間 [C/C++]

- 経過時間/CPU時間の測定方法

サンプルコード: timer

  - 基本的な考え方: 測定部分の前後で時間を取得→時間の差を計算
    - 経過時間: clock_gettime + CLOCK_REALTIME (time.hが必要)
      - ✓備考: CLOCK_MONOTONICの利用も考えられる
        - https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic
    - CPU時間: clock_gettime + CLOCK_PROCESS_CPUTIME_ID (time.hが必要)

```
#include <time.h>
#include <stdio.h>
double get_elp_time() { /*タイマールーチン */
   struct timespec tp;
   clock_gettime(CLOCK_REALTIME, &tp);
   return tp.tv_sec+(double)tp.tv_nsec*1.0e-9;
}
int main(void) {
   double elp1, elp2;
   elp1 = get_elp_time();
                経過時間を測定したい部分
   elp2 = get_elp_time();
   printf("Elapsed time (sec)=%13.4f¥n", elp2-elp1);
}
```

```
#include <time.h>
#include <stdio.h>
double get_cpu_time() { /*タイマールーチン */
   struct timespec tp;
   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tp);
   return tp.tv_sec + (double)tp.tv_nsec*1.0e-9;
}
int main(void) {
   double cpu1, cpu2;
   cpu1 = get_cpu_time();
                CPU時間を測定したい部分
   cpu2 = get_cpu_time();
   printf("CPU time (sec)=%13.4f¥n", cpu2-cpu1);
}
```

- clock_gettime使用時にコンパイル(リンク)で失敗する場合 → リンク時に-lrtを設定 (librt)
  - 例: gcc a.o. b.o timer.o -lrt

性能測定の方法: プログラムの特定区間 [C++ STL]

- 経過時間の測定方法

サンプルコード: timer

  - 基本的な考え方: 測定部分の前後で時間を取得→時間の差を計算
    - 経過時間: std::chrono::steady_clock (<chrono>が必要, C++11以上)

```
#include <chrono>
#include <cstdio>

int main(void) {
   {
      const auto elp1 = std::chrono::steady_clock::now();
                  経過時間を測定したい部分
      const auto elp2 = std::chrono::steady_clock::now();
      const std::chrono::duration<double> elapsed = elp2 - elp1;
      printf("Elapsed time (sec)=%13.4f¥n", static_cast<double>(elapsed.count()));
   }
}
```

性能測定の方法: 補足

- タイマーの精度について
  - 信頼できる計測時間を把握しておくこと
    - 短すぎる計測時間で性能向上を議論するのは避けることを推奨
    - チェック用コードの例
      - ✓時間差が正になるまで処理量 (nnの値) を増やしながら計算する

サンプルコード: timer-res

Fortran

```
tval = -1.0D0
nn = 0
do while ( tval .le. ZERO )
   nn = nn + 1
   call cpu_time (tval0)
   i = func(nn) !nnに比例して処理量を増やす関数
   call cpu_time (tval)
   tval = tval - tval0
end do
```

C

```
tval = -1.0;
nn = 0;
while ( tval <= 0.0 ) {
   nn++;
   tval0 = get_elp_time ();
   func ( nn ); /*nnに比例して処理量を増やす関数*/
   tval = get_elp_time ();
   tval -= tval0;
}
```

I. Crawford and K. Wadleigh, "Software Optimization for HPC" (2000)

ホットスポットの特定

- ホットスポットを特定するためのツール (プロファイラ) ([A]:3-6, 3-8, [HW]:2.1節)
  - Function profiling: サブルーチン/関数単位の情報収取
    - 典型的なツール: gprof (GNUのパッケージの一部)

Case study: gprofによる性能評価の流れ

1. コンパイル

```
$ gfortran -pg -O3 sample.f
```

```
$ gcc -pg -O3 sample.c
```

-pgをつけてコンパイル

2. 実行・分析

```
$ ./a.out
```

実行後gmon.outが生成

3. データ取得

```
$ gprof ./a.out > prof.out
```

分析結果をテキストファイル
prof.outに書き出し

ホットスポットの特定

- ホットスポットを特定するためのツール (プロファイラ) ([A]:3-6, 3-8, [HW]:2.1節)
  - Function profiling: サブルーチン/関数単位の情報収取
    - 典型的なツール: gprof (GNUのパッケージの一部)
      - ✓Flat profile: サブルーチンごとのCPU時間，呼び出し回数

サンプルコード: timer, prof-ex

0.01秒毎に(関数の使用状況を)サンプリングすることで計測 (OSの割り込み機能を利用)

ホットスポット

測定用ルーチンの稼働時間 (無視すべき)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 78.55     58.43    58.43  1000000    58.43    58.43  sub3_
 14.63     69.31    10.88   200000    54.40   288.12  sub2_
  6.64     74.25     4.94   100001    49.40   166.26  sub1_
  0.19     74.39     0.14                             _mcount_private

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.
```

ホットスポットの特定

- ホットスポットを特定するためのツール (プロファイラ) ([A]:3-6, 3-8, [HW]:2.1節)
  - ◆Function profiling: サブルーチン/関数単位の情報収取
    - 典型的なツール: gprof (GNUのパッケージの一部)
      - ✓Call graph: コールグラフ (関数の呼び出し関係を分析)
        - サブルーチン/関数間の関連図を作成したい場合にも便利

サンプルコード: timer, prof-ex

```
                     Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.01% of 74.39 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.8    0.00   74.25                 MAIN__ [1]
               10.88   46.74  200000/200000      sub2_ [3]
                4.94   11.69  100000/100001      sub1_ [4]
-----------------------------------------------
               11.69    0.00  200000/1000000     sub1_ [4]
               46.74    0.00  800000/1000000     sub2_ [3]
[2]     78.5   58.43    0.00 1000000           sub3_ [2]
-----------------------------------------------
               10.88   46.74  200000/200000      MAIN__ [1]
[3]     77.5   10.88   46.74  200000           sub2_ [3]
               46.74    0.00  800000/1000000     sub3_ [2]
```

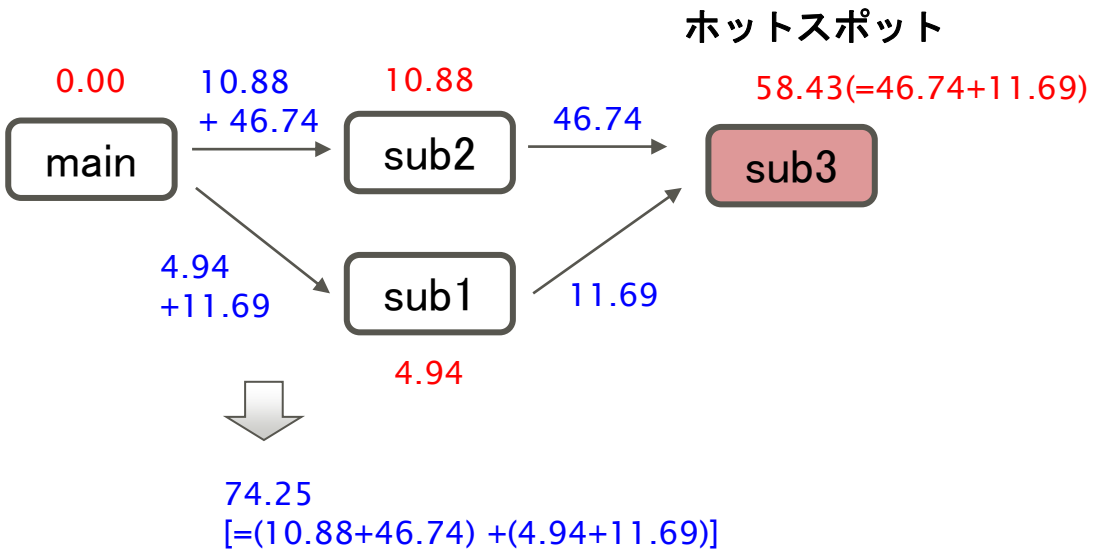
例: main関数の情報
- self =0: 自分自身の負荷は無し
- children=74.25: 呼び出し先の負荷
- 呼び出している関数: sub1とsub2

ホットスポットの特定

- ホットスポットを特定するためのツール (プロファイラ) ([A]:3-6, 3-8, [HW]:2.1節)
  - Function profiling: サブルーチン/関数単位の情報収取
    - 典型的なツール: gprof (GNUのパッケージの一部)
      - ✓Call graph: コールグラフ (関数の呼び出し関係を分析)
        - サブルーチン/関数間の関連図を作成したい場合にも便利

サンプルコード: timer, prof-ex

Case study: call graphから関連図を作成する



self
children

```
index % time    self  children    called     name
                                                <spontaneous>
[1]     99.8    0.00   74.25                 MAIN__ [1]
               10.88   46.74  200000/200000      sub2_ [3]
                4.94   11.69  100000/100001      sub1_ [4]
-----------------------------------------------
               11.69    0.00  200000/1000000     sub1_ [4]
               46.74    0.00  800000/1000000     sub2_ [3]
[2]     78.5   58.43    0.00 1000000          sub3_ [2]
-----------------------------------------------
               10.88   46.74  200000/200000      MAIN__ [1]
[3]     77.5   10.88   46.74  200000          sub2_ [3]
               46.74    0.00  800000/1000000     sub3_ [2]
```

性能分析の手順

- 手順のまとめ
  1. timeコマンドで全体を計測
     - 単一プロセッサコア使用の場合: 経過時間(real)とCPU時間 (user+sys)の差分を評価
       - ✓ I/Oチューニングの有無を判断
     - 計算機センター等でジョブ実行ログが出る場合 → ログの経過時間もおさえておく
  2. 実行プログラムが出力するログを確認
     - プログラム開発者が予めタイマーを挿入済み →ルーチンごとの時間内訳を把握可能
       - ✓ ログの表示レベルを上げることで性能情報が表示されるケースもある

実際には: 2までで十分なことが多い → 負荷の内訳を把握する

  3. (必要に応じて) プロファイラによる分析を実施
     - 関数ごとの時間内訳を把握 (ログ情報が利用できない場合)
       - ✓ ログ情報が利用可能な場合: ログ情報とプロファイラの結果の対応関係をおさえる
     - 負荷が顕著なルーチンについて: ソースコードの行単位で負荷を把握
     - ソースコードの高負荷部分を観察: プログラムの特徴をおさえる
       - ✓ コンパイラの診断レポートと比較する

[参照] Appendix: Perf による性能測定

まとめ

- part1: チューニングの基礎
  - 計算の実行時間の内訳を意識する (問題の切り分け)
  - ホットスポット・チューニング
    - 最小の努力で最大の効果
    - 原則: ホットスポット以外は触らない
  - コンパイラを活用する
    - 最適化オプションの設定:「そこそこ」速く安全なコンパイルオプションの選択
    - 診断レポートを表示 → チューニング技法習得の近道
  - 性能の測定
    - タイマーによる時間計測
    - プロファイラの利用: ホットスポットの特定に便利

Appendix

- Fortranに関する補足
- C/C++に関する補足
- Perfによる性能測定
- デバッグとコンパイルオプション
- 演算例外について

Fortranに関する補足 (1/3)

- 参考資料
  - 基本
    - 冨田博之, 齋藤泰洋「Fortran90/95プログラミング」(培風館, 2011)
    - 牛島省「数値計算のためのFortran90/95プログラミング入門」第2版 (森北出版, 2020)
    - A. Markus, Modern Fortran in Practice (Cambridge, 2012)
  - 包括的
    - M. Metcalf, J. Reid, and M. Chohen, Modern Fortran explained: Incorporating Fortran 2018 (Oxford, 2018)
  - 便利なウェブサイト
    - Fortran Wiki; http://fortranwiki.org/fortran/show/HomePage
    - Fortran Discourse; https://fortran-lang.discourse.group/

Fortranに関する補足 (2/3)

- On Modern Fortran (.GE. 2003): Fortran 2003以降で様々な機能が拡充
  - 最適化促進の機能
    - do concurrent: SIMD処理可能なループであることを指定
    - contiguous attribute: 配列セクションへのポインタ, 形状引き継ぎ配列でのデータの連続性を保証
  - Interoperability with C
    - ISO_C_BINDINGの利用 (use, intrinsic::ISO_C_BINDING): Cの型が利用可能
    - Cで書かれた関数のFortranからの呼び出し (interface + bind(C))
      - ✓cf. 伝統的な方法: Cで書かれた関数を呼び出す → 関数名の最後に下線(_)を付けてリネイム

サンプルコード: timer

- A. Markus, Modern Fortran in Practice (Cambridge, 2012) Chap.6

  - オブジェクト指向
    - 派生型 (derived type) に関する機能の拡充
      - ✓元指定割付け (sourced allocation) による派生型変数の一括代入(クローン生成); allocate(A, source=B)
    - 多相的変数 (polymorphic variables) に関する機能の拡充
      - ✓無制限多相的変数 (class(*))によるオブジェクトの遅延結合 (deferred binding)
      - ✓多相的変数の初期化: sourced allocation, alloc_move, structure constructor

Fortranに関する補足 (3/3)

- On Modern Fortran (.GE. 2003): コード開発上の考慮点
  - 使用する機能をサポートしている規格を明確にする
    - 特定の機能に対し全てのコンパイラが対応している保証なし
      - ✓新しい機能であるほど対応状況が十分とは限らない傾向
    - コーディング上の対応の例
      - ✓コンパイル・オプションで規格を陽に指定
        - [参考] LLVM: https://releases.llvm.org/13.0.0/tools/flang/docs/OptionComparison.html
      - ✓インストールの要求事項として明記
      - ✓プリプロセッサの使用で規格に応じたプログラムの切り替え
        - 伝統的なマクロ: #if defined, #if not defined, etc.
        - fypp (Python powered Fortran metaprogramming): https://github.com/aradi/fypp
  - 性能に影響を与える可能性がある機能の有無を確認する
    - ミニチュアコード等で実験して採用の可否を決める
      - ✓調査対象の典型例: 配列に関する組み込み関数 (reshape, spread, etc.)
        - 検討点: 素朴にdoループで記述するのと比べてどうか?

C/C++に関する補足 (1/3)

- 参考資料
  - 最適化・各種tips
    - P. van der Lindern (梅原系 訳)「エキスパートCプログラミング」(アスキー, 1996)
    - S. Meyers, “Effective Modern C++” (O’REILLY, 2015)
      - ✓“More Effective C++” (O’REILLY, 1995): 古い記述に注意して読むこと
    - K. Guntheroth, “Optimized C++” (O’REILLY, 2016)
    - D. Viswanath, “Scientific Programming and Computer Architecture” (MIT, 2017)
  - 包括的
    - P. Prinz and T. Crawford (黒川利明 訳)「Cクイックレファレンス」第2版 (オライリー, 2016)
    - S. B. Lippman, J. Lajoie, and B. E. Moo (神林靖 監修, 株式会社クイープ 翻訳)「C++プライマー」第5版 (翔泳社,2016)
    - D. Vandevoorde, N. M. Josuttis, and D. Gregor, “C++ Templates: The Complete Guide” 2nd ed. (Addison-Wesley, 2017)
  - 便利なウェブサイト
    - cppreference.com; https://en.cppreference.com/w/
    - cplusplus.com; https://cplusplus.com/

C/C++に関する補足 (2/3)

- [C++] 特殊化: 特定のテンプレートパラメータの実装(インスタンス化)のみを変更
  - 性能分析やチューニングをする際に便利(なこともある)

```
template<> void MyClass::impl<0,1>(int na)
{ // 特殊化:  impl<0,1>
  printf("  [Specified] na = %d, nb = %d¥n", na, -100);
}
void MyClass::func(int aflag, int bflag, int na)
{
  if ( aflag ) {
    if ( bflag ) impl<1,1>(na);
    else impl<1,0>(na);
  } else {
    if ( bflag ) impl<0,1>(na);
    else impl<0,0>(na);
  }
}
template<int AFLAG, int BFLAG> void MyClass::impl(int na)
{ // 元の関数テンプレートの実装
  int nb;
  nb = 0;
  if ( AFLAG ) nb = na + 1;
  if ( BFLAG ) nb = -1;
  printf("  [Original] na = %d, nb = %d¥n", na, nb);
}
```

```
MyClass mc;
aflag = bflag = 0;
for ( int i = 0; i < 10 ; ++i ) {
  printf("[Itr = %d] ", i);
  if ( i%3 == 0 ) aflag = 1;
  if ( i%7 == 0 ) bflag = 1; //i=7のときのみaflag=0, bflag=1
  mc.func(aflag,bflag,i);
  aflag = bflag = 0;
}
```

```
[Itr = 0]   [Original] na = 0, nb = -1
[Itr = 1]   [Original] na = 1, nb = 0
[Itr = 2]   [Original] na = 2, nb = 0
[Itr = 3]   [Original] na = 3, nb = 4
[Itr = 4]   [Original] na = 4, nb = 0
[Itr = 5]   [Original] na = 5, nb = 0
[Itr = 6]   [Original] na = 6, nb = 7
[Itr = 7]   [Specified] na = 7, nb = -100
[Itr = 8]   [Original] na = 8, nb = 0
[Itr = 9]   [Original] na = 9, nb = 10
```

- D. Vandevoorde, N. M. Josuttis, and D. Gregor, “C++ Templates” 2nd ed. (Addison-Wesley, 2018) Chap.20
- S. B. Lippman, J. Lajoie, and B. E. Moo (神林靖 監修, 株式会社クイープ 翻訳) 「C++プライマー」第5版 (翔泳社,2016) 第16章

C/C++に関する補足 (3/3)

- Cのコードを C++から呼び出す場合の注意
  - extern “C” を設定
    - C++のコンパイラにCでコンパイルされたオブジェクトであることを伝える
  - マクロ変数__cplusplusを利用した条件付きコンパイル
    - Cのコードとしてコンパイルする場合 (extern “C”は不要) への対応

サンプルコード: io-format

https://stackoverflow.com/questions/3789340/combining-c-and-c-how-does-ifdef-cplusplus-work

Case study: Cのタイマールーチンを C++のコードから呼び出す

```
/* timer.h */
#if defined(__cplusplus)
extern "C" {
#endif
double get_elp_time(); /* timer.cで実装 */
#if defined(__cplusplus)
}
#endif
```

```
gcc -c timer.c -o timer.o
g++ -c main.cpp -o main.o
g++ timer.o main.o -o a.out
```

```
// main.cpp
#include <cstdio>
#include “timer.h”

int main (int argc, char **argv)
{
  double elp1, elp2;
  elp1 = get_elp_time();
  …
  elp2 = get_elp_time();
  printf(“Elapsed time (sec)=%13.4f¥n”,
          elp2-elp1);
}
```

Perfによる性能測定 (1/4)

- perfの概略 (https://perf.wiki.kernel.org/index.php)
  - Linuxで使用可能: カーネルを利用した様々な性能情報の収集
    - perf listコマンド: 収集可能な情報 (event) を表示
    - FortranやC/C++で生成されたプログラム, スクリプト (例: python)について情報収集が可能
      - ✓プログラムの再コンパイルは不要
  - インストールについて
    - 確認
      - ✓[Red Hat] yum list installed |grep -E -i “perf”
      - ✓[Ubuntu] apt list --installed |grep -E -i “perf”
    - 方法 (システム管理者のみ) (cf. ソースからコンパイルも可能)
      - ✓[例: Ubuntu] sudo apt install linux-tools-common
  - 利用を推奨しないケース
    - ノードを跨ぐ分散並列処理, 演算加速器を利用 → 専用の性能分析ツールを使うほうがよい
    - 計算機センター等でツールが提供 → 提供ツールを優先して使用すべき

- B. Gregg (監訳: 西脇靖紘, 訳: 長尾高弘)「詳解システム・パフォーマンス」(オライリー・ジャパン, 2017) 6章

Perfによる性能測定 (2/4)

- 使い方の例: Counting (プログラム実行中のカウンター情報を計測)
  - perf stat [実行コマンド] → 画面に性能情報が出力
    - 例: pythonスクリプトの実行

```
$ cat test.py
#! /usr/bin/env python3
import numpy as np

a = np.arange(3*3).reshape(3,3)
b = np.arange(3*3).reshape(3,3)
c = np.matmul(a, b)
print(c)
```

```
$ perf stat python3 test.py
[[ 15  18  21]
 [ 42  54  66]
 [ 69  90 111]]

 Performance counter stats for 'python3 test.py':

            105.20 msec task-clock:u                     #    0.850 CPUs utilized
                 0      context-switches:u               #    0.000 K/sec
                 0      cpu-migrations:u                 #    0.000 K/sec
              3750      page-faults:u                    #    0.036 M/sec
         265513316      cycles:u                         #    2.524 GHz                      (82.78%)
           7212200      stalled-cycles-frontend:u        #    2.72% frontend cycles idle     (82.76%)
          54856880      stalled-cycles-backend:u         #   20.66% backend cycles idle      (84.16%)
         406471397      instructions:u                   #    1.53  insn per cycle
                                                         #    0.13  stalled cycles per insn  (85.20%)
          85223514      branches:u                       #  810.096 M/sec                    (81.67%)
           2571692      branch-misses:u                  #    3.02% of all branches          (83.44%)

       0.123743955 seconds time elapsed

       0.081265000 seconds user
       0.022612000 seconds sys
```

性能情報

イベント(event)の種類に応じた計測結果

Perfによる性能測定 (3/4)

- 使い方の例: Sampling (プログラムの実行状況を一定間隔ごとに計測)

サンプルコード: prof-ex

  - 基本: perf record ./a.outでプログラム実行 → perf report で情報取得
    - 統計情報はperf.dataに集約

  - Function profilingの実行 → gprofの標準的な使用方法とほぼ同等
    - サブルーチン/関数毎の負荷情報の取得 (cpu使用率)
      - ✓1. `perf record --event=cpu-cycles a.out`
      - ✓2. `perf report --header --stdio --input=perf.data` → 標準出力にデータ出力
    - コールグラフの取得
      - ✓1. `perf record --event=cpu-cycles --call-graph fp a.out`
        - fp: frame pointerを利用 (デフォルト) → fpが設定されていない場合はdwarf等で対応
      - ✓2. `perf report --stdio --input=perf.data --call-graph`
    - eventの設定変更
      - ✓ユーザ関数のみの負荷取得: `perf record --event=cpu-cycles:u ./a.out`
      - ✓キャッシュミス率の取得: `perf record --event=cache-misses ./a.out`
      - ✓複数イベントの計測: `perf record --event=cpu-cycles,page-faults ./a.out`

Perfによる性能測定 (4/4)

- 使い方の例: Sampling (つづき)
  - ソースコードの行単位の負荷分析 (Line profiling)

サンプルコード: prof-ex

    - 注釈付き出力の生成 (負荷, 機械語命令, ソースコード) + 高負荷の行数表示
      - ✓1. perf recordで情報収集: `perf record --event=cpu-cycles a.out`
      - ✓2. perf reportで負荷情報を出力: `perf report --header --stdio --input=perf.data`
        - 高負荷のサブルーチン/関数名を調査 (Symbol欄)
      - ✓3. `perf annotate --input=perf.data --print-line --source --symbol=<symbol>`
        - <symbol>: 対象とするサブルーチン名 (perf reportの出力のSymbol欄に従う)
        - 備考:ソースコードに負荷を出力するにはデバッグオプション (-g)が必要

GNU系のツールでも注釈付き出力 (annotated source) の生成は可能
gprof -A
gcov (https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)

デバッグとコンパイルオプション

- デバッグ時のオプション
  - 通常のコンパイルでエラーとならない問題を検出するものがある
    - 不正なメモリアクセス (領域外参照), サブルーチン/関数コール間の矛盾, etc.
      - ✓例 (GNU, LLVM): -fsanitize=address
  - 性能は低下 → デバッグ時のみの指定を推奨
  - 各種デバッグオプションの設定や意味

  → 使用しているコンパイラのマニュアルで確認する必要がある
    - GNU: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
    - AddressSanitizer in LLVM: https://clang.llvm.org/docs/AddressSanitizer.html
    - Fujitsu: https://software.fujitsu.com/jp/manual/manualindex/p21000154.thml

演算例外について

- aggressiveな最適化オプション利用時の注意
  - 副作用により演算例外が発生する場合がある
    - 浮動小数点(実数)の演算例外: オーバーフロー, アンダーフロー, ゼロ除算, 無効演算
      ✓結果出力ファイルにNaN (Not a Number)やInf (Infinity)が表示されることがある

- 演算例外に対する対応
  - 基本的には無視すべきではない: 「性能追求」より「安定して正しい計算をする」ほうが優先
    - コード上の該当箇所において最適化を抑制する
      ✓分割コンパイルでオプションを変える, コンパイラの指示文を利用する, 等々
  - NaN検出のコーディング上の工夫
    - [Fortran] IEEE_IS_NAN関数 (USE, INTRINSIC :: IEEE_ARITHMETICが必要)
    - [C] isnan関数 (#include <math.h>が必要)
    - [C++] std::isnan関数 (#include <cmath>が必要)
  - NaN発生箇所の特定 (gdb利用)
    - http://www.ecs.shimane-u.ac.jp/~stamura/NumericalComputation-Tips.html

2024年5月
一般財団法人高度情報科学技術研究機構（著作者 ）