第8回
高速化チューニングとその関連技術1

渡辺宙志

東京大学物性研究所

Outline

1. チューニング、その前に
2. バグを入れないコーディング
3. デバッグの方法論

本講義の内容

プログラム開発時間の短縮 (今週)

- バグを入れないコーディング習慣
- バグを入れにくい開発手順
- バージョン管理システム
- デバッグの効率化

今後の人生の役に立ちます

プログラム実行時間の短縮 (来週)

- プロファイラの使い方
- メモリ最適化
- SIMD化等

今後の人生の役に立ちません

チューニング、その前に

チューニング、その前に (1/3)

最適化の第一法則：最適化するな

最適化の第二法則(上級者限定)：まだするな

Michael A. Jackson, 1975

チューニング、その前に (2/3)

足が速いからといって良いサッカー選手に
なれるとは限らない

H. Watanabe, 2012

チューニング、その前に (3/3)

なぜ最適化するのか？

プログラムの実行時間を短くするため

なぜ実行時間を短くしたいのか？

計算結果を早く手に入れるため

なぜ計算結果を早く手にいれたいのか？

論文を早く書くため ← ここがとりあえずのゴール

最適化、並列化をする際には、必ず「いつまでに論文執筆まで持って行くか」を意識すること。だらだらと最適化にこだわらない。

典型的な研究スパン

年に二編論文を書く → 半年で一つの研究が完結

調査	プログラム開発＋計算	執筆

調査：先行研究の調査や、計算手法についての調査 (1ヶ月)
開発＋計算：プログラム開発、計算の実行(4ヶ月)
執筆：結果の解析＋論文執筆＋投稿 (1ヶ月)

実態は・・・

調査	開発	デバッグ	計算	執筆

開発時間の大部分はデバッグに費やされている
初心者であるほど、デバッグの占める割合が長くなる
コードの高速化は、研究時間の短縮にさほど寄与しない

※ もちろん例外あり

デバッグについて

Q. 最適化、並列化でもっとも大事なことは何か？

A. バグを入れないこと

開発において最も時間のかかるプロセスはデバッグ
並列プログラムのデバッグは絶望的に難しい

デバッグは時間がかかり、集中力が要求され、達成感もある
しかし、結局は自分が入れたバグを自分で取っているだけ

「デバッグは仕事ではない」ということを肝に銘じること

バグの入り方

Q. バグはいつ入るか？

A. 機能を追加したとき

バグの種類:

- 機能追加直後に判明するバグ(即効性)
  → バグを入れないコーディング
- 機能追加後、後で判明するバグ(地雷)
  → デバッグの方法論

バグを入れないコーディング

バグを入れないコーディング

- バグが入りにくいプログラム習慣をつける
  - コンパイラの警告を無視しない
  - 普段からassertをいれる癖をつける
- バグが入りにくい開発プロセスを踏む
  - 単体テスト
  - sort+diffデバッグ
- それでもバグが入ってしまったら・・・
  - バージョン管理システムとの連携
  - デバッガの利用

コンパイラの警告を無視しない (1/4)

代入と比較の間違い

```
for(int i=0;i<10;i++){
   if(i=3) puts("i=3! ");
}
```

if (i==3) puts("i=3!");
本当はこれが正しい

コンパイラはデフォルトで上記のコードに警告を出さないが、
「-Wall」をつけると以下の警告を出してくれる

```
$ g++ -Wall test.cpp
test.cpp: 関数 ‘int main()’ 内:
test.cpp:6:11: 警告: 真偽値として使われる代入のまわりでは、
丸括弧の使用をお勧めします [-Wparentheses]
    if(i=3)puts("i=3! ");
```

注：この警告は、もしこれがミスでなく意図するコードなら

```
if ((i=3)) puts("i=3! ");
```

と書けという意味。こうすると警告が消える。

コンパイラの警告を無視しない (2/4)

このプログラムの間違い、すぐにわかりますか？

```
#include <stdio.h>

int add_three (int verylongname){
  int veryverylongname= veryverylongname+ 3;
  return veryverylongname;
}

int main(void){
  printf("%d¥n", add_three(1));
}
```

add_threeは、引数に3を加えた値を返す関数のつもり

コンパイラの警告を無視しない (3/4)

- 本来なら引数であるべき変数を、似た名前のローカル変数で書いてしまった

```
int add_three (int verylongname){
  int veryverylongname= veryverylongname+ 3;
  return veryverylongname;
}
```

int veryverylongname= verylongname+ 3;
ここは、本当はこれが正しい

コンパイラはデフォルトで以下のコードに警告を出さない

```
int a = a + 1;
```

コンパイラの警告を無視しない (4/4)

(1)「-Wall」オプションをつけてコンパイルすると・・・

```
$ g++ -Wall test.cpp
test.cpp: 関数‘int add_three(int)’ 内:
test.cpp:5:45: 警告: ‘veryverylongname’ はこの関数内で初期化されずに使用されています [-Wuninitialized]
   int veryverylongname = veryverylongname + 3;
```

ちゃんと「初期化されてない変数を使ってるよ」と教えてくれる

(2)「-Wall -Wextra」とオプションを追加すると・・・

```
$ g++ -Wall test.cpp
test.cpp:4:5: 警告: 仮引数‘verylongname’ が未使用です [-Wunused-parameter]
 int add_three(int verylongname){
     ^
test.cpp: 関数‘int add_three(int)’ 内:
test.cpp:5:45: 警告: ‘veryverylongname’ はこの関数内で初期化されずに使用されています [-Wuninitialized]
   int veryverylongname = veryverylongname + 3;
                                          ^
```

「使われてない変数があるよ」とも教えてくれる

Intelコンパイラでは、-w2で(1)を、-w3で(2)を教えてくれる

コンパイラの警告を無視しないのまとめ

- 普段から「-Wall –Wextra」相当のオプションを指定する癖をつける
- コンパイラの警告を無視しない

普段から警告ゼロをキープすることが大事
警告が出たら
「あ、なんかやらかしたな」
と思うこと。

普段からassertをいれる癖をつける (1/4)

assertとは何か？

to state firmly that something is true

(From Longman Dictionary of Contemporary English)

C言語のassert

プログラムにおいて「成り立っていなければならない条件」を記述する

```
#include <assert.h>
...
assert(some condition);
```

中身が成り立っていれば何もしない
不成立なら、Assertion Failedと言ってプログラムがabortする

普段からassertをいれる癖をつける (2/4)

assertの例

```
void func(int a){
  assert(a<10);
  printf("%d¥n",a);
}

int main(void){
  func(8);   //OK
  func(11); //失敗する
}
```

入力となるaは10未満であるはず、と宣言する

Assertionが破られたこと、ソースのどこでAssertionが破られたか教えてくれる

実行結果

```
$ ./a.out
8
Assertion failed: (a<10), function func, file test.cpp, line 5.
zsh: abort      ./a.out
```

普段からassertをいれる癖をつける (3/4)

assertの無効化

そんなチェックをたくさん
入れたら遅くなるんじゃないの？

assertは「-DNDEBUG」オプションで
無効にできます

```
$ g++ -DNDEBUG test.cpp
$ ./a.out
8
11
```

← Assertion failedが起きない

開発中は有効に、プロダクトランの時には無効にする

普段からassertをいれる癖をつける (4/4)

assertに助けられた例

- 粒子のペアが p1[N], p2[N]という2つの配列として表現されている
- p1[i]とp2[i]がi番目のペアの粒子番号を表す
- 高速化のため一度ソートし、必ず p1[i] < p2[i] となっているはずだった
- しかし念のため

```
assert(p1[i] < p2[i]);
```

を入れておいた

- 後日、自分がassertを入れたことも忘れた頃に・・・

```
Assertion failed: (p1[i] < p2[i] ), function calcforce, file
calcforce.cpp, line 125.
```

あとで追加した関数が、ソート関数を呼び忘れていたのが原因

コンパイル、計算は実行できるが、結果を地味に間違える

assertについてのまとめ

☑ 普段からassertを入れる癖をつける

どこに入れればいいか
わからないんですが

入れているうちにだんだん
分かってきます

※ この関数を呼ぶ時にはこうなってるはず、みたいなところに入れると良い

assertは「自分の実装意図」を示すコメント
→ ただのコメントと違い、異常を検出してくれる
今日の自分が一ヶ月後の自分を助ける

転ばぬ先のassert

バグを入れない開発手順

バグを入れない開発手順

基本理念

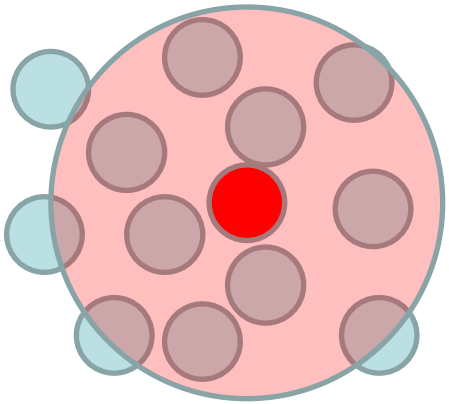
- なるべく頭を使わず、機械的にチェックできる仕組みを作る
- 「ここまでは大丈夫」という「砦」を築きながら進む

開発技法

- 開発技法には長い歴史があり、現在も研究が進んでいる
  XP、アジャイル開発、チケット駆動開発、テスト駆動開発
- print文デバッグ
  → 必要な情報を出力しながらデバッグする方法
  → 最も古典的かつ基本となるデバッグ方法
- sort+diffデバッグ
  → print文デバッグの一種
  → 一致すべき情報が一致しているか確認する
- 単体テスト
  → 開発したい部分だけを切り出してテストすること
  → いきなりコード全体でテスト(統合テスト)してはならない

sort+diff デバッグの例1：粒子対リスト作成 (1/2)

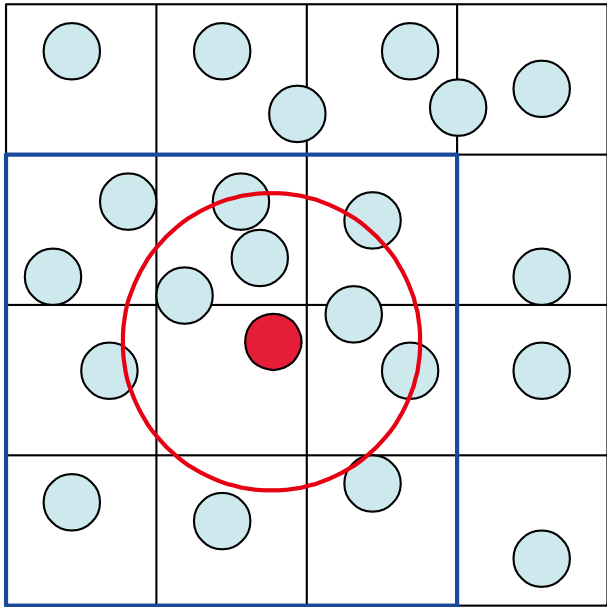
ペアリストとは？



相互作用距離(カットオフの距離)以内にある粒子対のリスト
全粒子対についてチェックすると$O(N^2)$
高速に粒子対を作成する方法　→ グリッド探索

グリッド探索

・空間をグリッドに切り、その範囲に存在する粒子を登録する→ $O(N)$



sort+diff デバッグの例1：粒子対リスト作成 (2/2)

ポイント

O(N)法とO(N^2)法は、同じconfigurationから同じペアリストを作る
O(N^2)法は、計算時間はかかるが信頼できる (砦)

手順

初期条件作成ルーチンとペアリスト作成ルーチンを切り出す(単体テスト)
O(N)とO(N^2)ルーチンに同じ初期条件を与え、ペアリストをダンプ
ダンプ方法：作成された粒子対の番号が若い方を左にして、一行に1ペア
リストの順番は異なるので、ソートしてからdiffを取る

```
$ ./on2code | sort > o2.dat
$ ./on1code | sort > o1.dat
$ diff o1.dat o2.dat
```

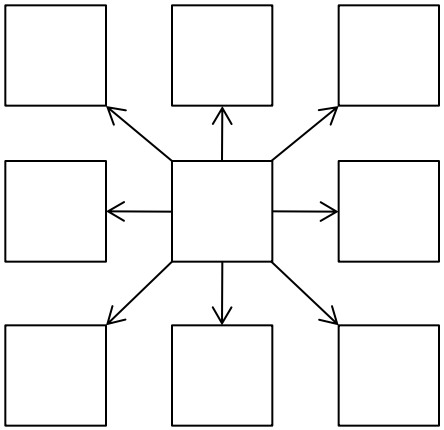
←結果が正しければdiffは何も出力しない

いきなり本番環境に組み込んで時間発展、などとは絶対にしない

sort+diff デバッグの例2：粒子情報送信(1/2)

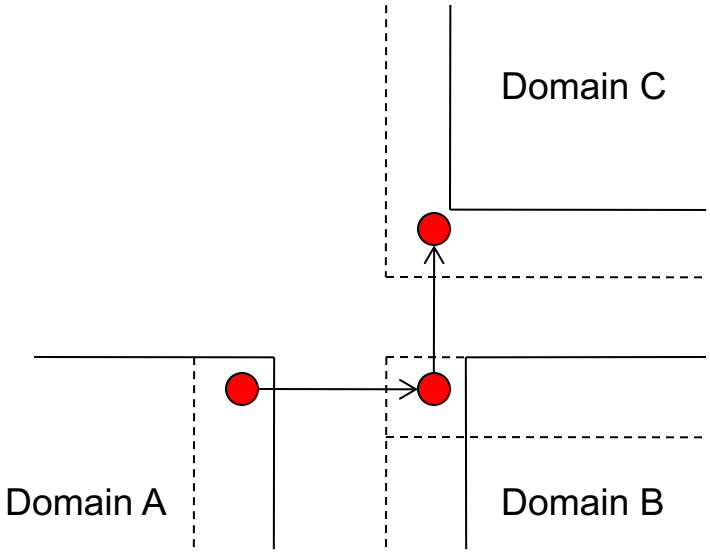
端の粒子の送り方

ナイーブな送り方



隣接するドメイン全てと通信を行う
3次元の場合、26回の通信が発生する

通信方法を減らした送り方



辺で接する領域からもらった粒子を、
別の方向で辺で接する領域へ転送

斜め方向の通信が必要なくなるため、
通信回数は6回で済む

sort+diff デバッグの例2：粒子情報送信(2/2)

デバッグの手順

(1) 初期条件作成ルーチンと通信ルーチンのみで実行　(単体テストの原則)
(2) 通信後、自分の担当する粒子を全て出力
　　(proc012.datなどの名前でファイルに出力する)
(3) ナイーブな通信(砦)と、転送式の通信の両方で実行
　　(出力先を test1/ test2/などと異なるディレクトリに)
(4) 粒子の座標が完全に一致することを確認 (sort + diff デバッグ)

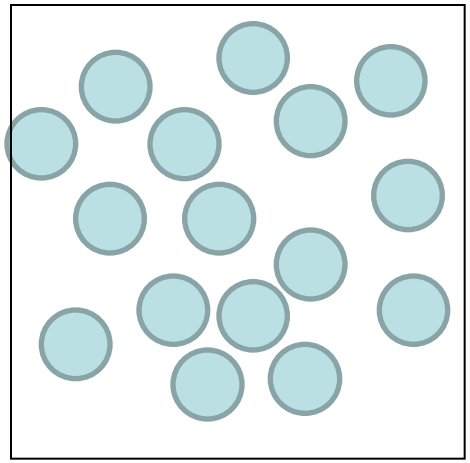
```
$ sort test1/proc000.dat > test1/proc000s.dat
$ sort test2/proc000.dat > test2/proc000s.dat
$ diff test1/proc000s.dat test2/proc000s.dat
```

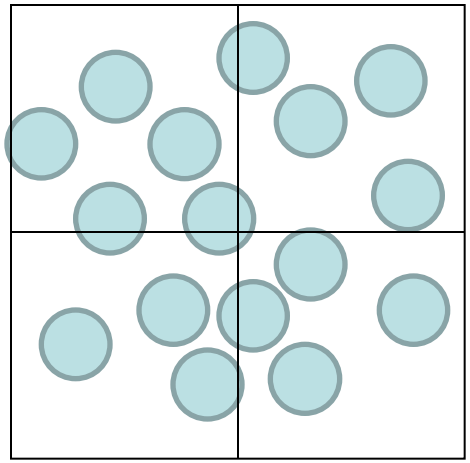
全てのプロセスについて一致することを確認
※ 複数の初期条件を試す事

sort+diff デバッグの例3：並列版リスト作成(1/2)

ペアリストの並列化

空間分割による並列化
各領域でそれぞれペアリストを作成
並列化の有無に関わらず同じconfigurationからは
同じペアリストを作成しなければならない





はじっこの粒子が正しく渡されているか？
周期境界条件は大丈夫か？

sort+diff デバッグの例3：並列版リスト作成(2/2)

ポイント

非並列版のペアリスト作成ルーチンはデバッグが終了しているはず (砦)
粒子情報の通信ルーチンはデバッグが終了しているはず(砦)

一度に複数の項目を同時にテストしない

手順

初期条件作成ルーチンとペアリスト作成ルーチンのみで実行 (単体テスト)
非並列版と並列版のペアリスト作成ルーチンを作る
非並列版はそのままペアリストをダンプ
並列版は「若い番号の粒子が自分の担当の粒子」であるときだけダンプ
並列版はプロセスごとにファイル(proc???.dat)に出力、catでまとめる
sort + diffで一致を確認する

```
$ ./serial | sort > serial.dat
$ ./parallel
$ cat proc???.dat | sort > parallel.dat
$ diff serial.dat parallel.dat
```

バグを入れないコーディングのまとめ

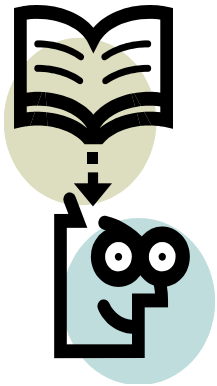
- 新しい機能の追加や高速化をするたびに単体テストする

  単体テストとは、必要なルーチンのみでコンパイル、実行すること
  全体のプログラムの一部に着目してテストすることではない

  単体テストとは、ミクロな情報がすべて一致するのを確認すること
  エネルギー保存など、マクロ量のチェックは単体テストではない

- 「確実にここまでは大丈夫」という「砦」を築く

  時間はかかるが信用できる方法と比較する
  複数の機能を一度にテストしない



デバッグとは、入れたバグを取ることではなく
そもそもバグを入れないことである

デバッグの方法論

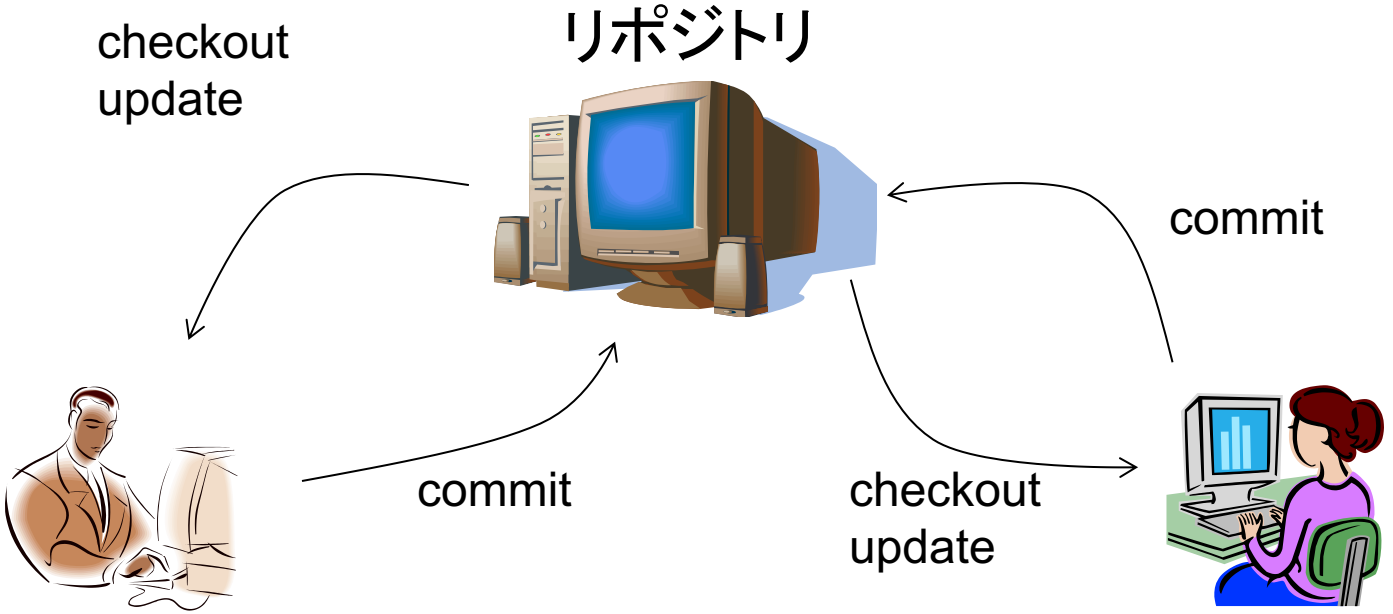
地雷型バグのデバッグ方法

デバッグの方法論・・・その前に

バージョン管理システム、使っていますか？(Y/y)

バージョン管理システムとは

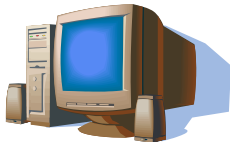
ファイルの編集履歴を管理するためのシステム
CVS, Subversion, Gitなどが有名
ファイルの編集履歴を全て保存する「リポジトリ」というデータベースをもつ
ユーザは、そのリポジトリにアクセスしながら開発を行う
超優秀な秘書のようなもの



ありがちなパターン

ローカル

スパコン



1)開発したコードをスパコンへ

コード → コード

2)動かなかったので苦労して修正する

コードA

3)スパコンで実行中、別の修正をする

4)修正したコードをスパコンへ

コードB → コードB

あっ、コードAを上書きしちゃった！

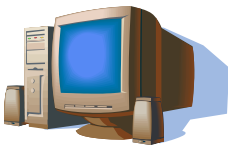
バージョン管理している場合

ローカル

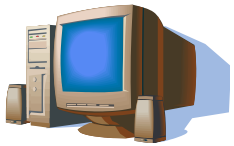
リポジトリ

スパコン

1)開発したコードを
リポジトリへ



2)リポジトリからスパコン
へチェックアウト



コード → コード → コード

5)スパコンの修正を忘れて別の修正

3)動かなかったので苦労して修正する

6)修正をコミットしようとして、衝突に気づく

コードB ← 衝突 → コードA ← 4)修正をコミット ― コードA

7)スパコン向けの修正と新しい修正を統合 (マージ)

コードC

バージョン管理システムのまとめ

バージョン管理システムはバックアップの代わりになる

svnのリポジトリや、gitのoriginは物理的に異なるサーバにすること
GithubやGitlabのプライベートリポジトリの活用など

バージョン管理システムは作業履歴が保存される

作業した結果が失われない
問題があった場合に遡って調べることができる

バージョン管理システムを使うと作業効率が倍以上になる
→ 使わないと人生を半分損する

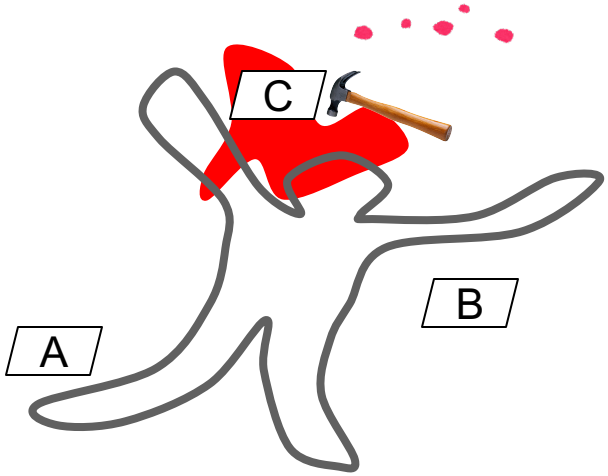
※使用者の感想であり、効果を保証するものではありません

地雷型バグ

地雷型バグとは？

バグを入れた後、しばらくしてから発見されるバグ
・最初から入っていたが、これまで気づかなかったタイプ
・機能追加時に、思わぬところに影響が波及したタイプ

バグを見つけたら？



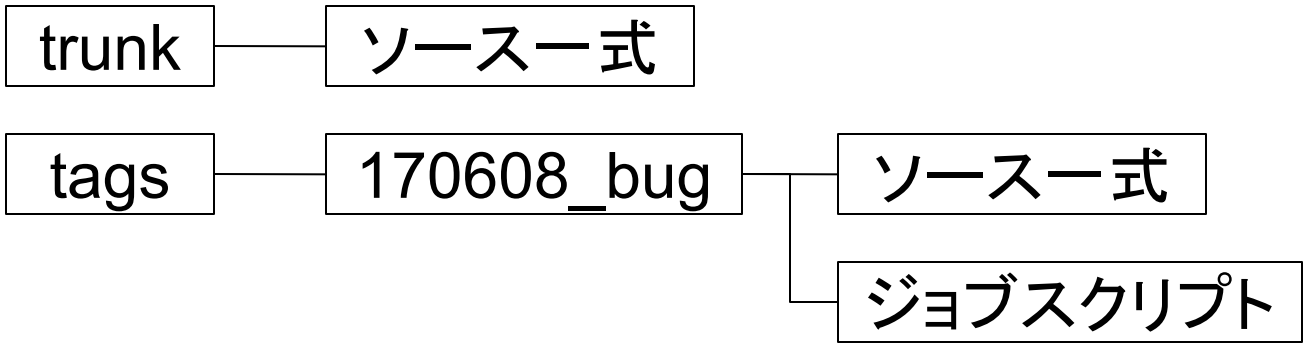
・いきなりデバッグをはじめない
デバッグにおいて重要なのは原因究明
「いつのまにかなおっていた」は一番まずい
→ 最初にやることは現場保全

(1) 再現性テスト (同じ条件で実行したら同じバグを発生するか？)
(2) バグを起こすソース一式を保存しておく (Subversionならタグ)
(3) バグを再現する最低限のコードを切り出す (容疑者の限定)

バグったコードの保存

バグったコードは保存しておく

Subversionを使っているなら、tagという機能を使う



Subversionにおいてタグとは、単にコピーのこと
Gitならブランチを切るなどする

なぜ保存しておくか？

デバッグしたつもりが、実はなおってなかったということがよくある
(別の原因でバグが発生しなくなったのを完治したと勘違い)
後で同様なバグが発生した時、同じ原因か、別のバグなのかを
確認したいことがよくあるため

問題の切り分け (1/2)

実行したらSegmentation Faultと言われて止まった

やってはならないこと

→ いきなりソースを見ながら原因を探る
(特にダメなのが頭の中でのトレース実行)

やるべきこと

- どこで止まったかを調べる
- どうやって調べるか？
  → print文による二分探索 (gdbでも可)

```
printf “1”;
    ・・・
printf “2”;
    ・・・
printf “3”;
```

出力が「1」であればこの間で止まっている

出力が「12」であればこの間で止まっている

上記を繰り返して、バグの発生箇所を特定する

問題の切り分け (2/2)

バグの発生箇所は、配列の領域外参照だった

```
const int N = 10;
double data[N];
・・・
double func(int index){
return data[index]; ← ここでindex=10だった
}
```

indexの値は0から9でないといけないのに、どこかでおかしな値が入った
(バグの発生箇所と、止まる箇所は一般に異なる)

おかしな値になった場所をどうやって探すか？
→ assertを入れまくる(if文でも可)

```
#include <assert.h>
double func(int index){
  assert(index<N); assertには「満たすべき条件」を記載する
  ・・・
}
```

assertにひっかかると、以下のようなエラーが出て止まる

```
Assertion failed: (i<10), function func, file test.cc, line 7.
```

実際に経験したバグ (1/2)

起きたこと

・ローカルPCで問題がなかったのに、スパコンでバグる
・スパコンでも条件によりバグったりバグらなかったりする
→ 当初、通信関連を疑ったが、乱数が原因だった

原因となった関数

与えられた整数Nについて、0からN-1までの数字をランダムに返す関数を意図して
こんなコードを書いた

```
double myrand_double (void){
return (double)(rand())/(double) (RAND_MAX);
}

int myrand_int (const int N){
return (int)(myrand_double()*N);
}
```

RAND_MAX=2147483647

実際には・・・ randは最高でRAND_MAXの値を返すので、
myrand_intは低確率(21億分の1の確率)でNを返す

実際に経験したバグ (2/2)

```
const int N = 10;
double data[N];
int index = myrand_int(N);
// (ずっと遠くで)
return data[index];
```

21億分の1の確率でNを返す

21億分の1の確率で配列外参照

だいたい2000ノード、1日ジョブで確率50%くらいで失敗した
→ ローカルPCでは10年くらい流しても踏まないバグ

この種のバグの原因に「最初から思い至る」のは難しい

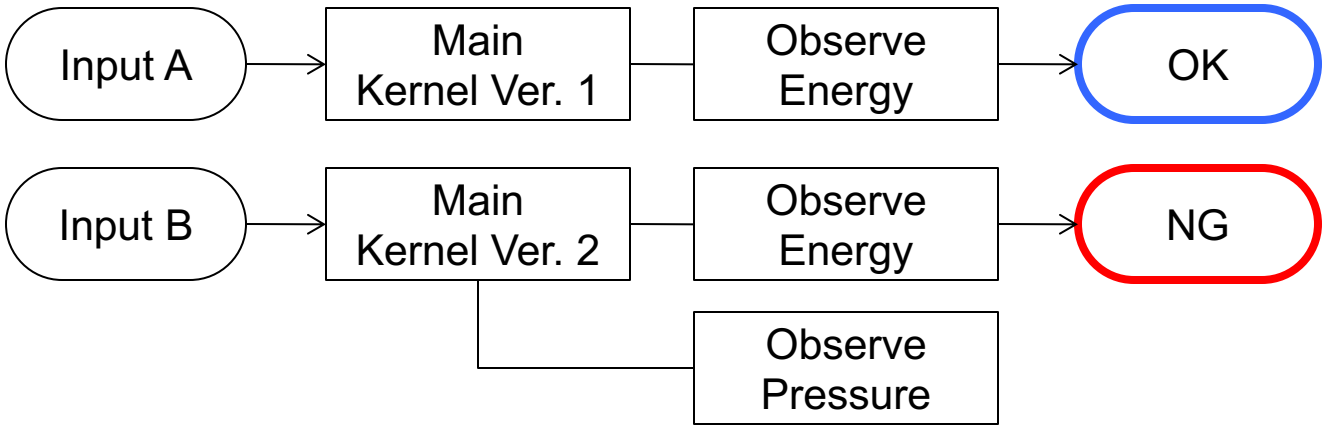
- 確実にバグを再現するプログラムを保存しておく
- print文+assert文デバッグを行う
- 必ず原因を究明し、放置しない

問題の切り分けとバージョン管理 (1/2)

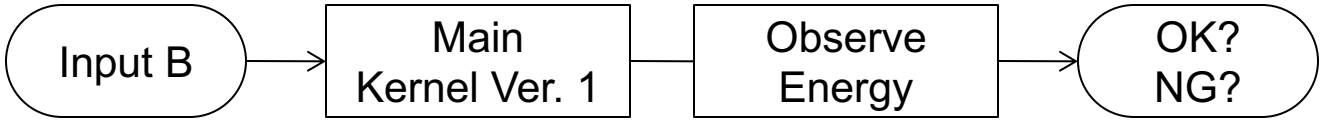
機能を追加したらバグった？

→ その機能を追加したことによるバグ？
もともとバグっていたものが顕在化？

例：圧力測定ルーチンを追加したら、エネルギーが発散した



圧力測定ルーチンのせいか？それともInput Bのせい(元々バグっていた)か？
→ ルーチン追加前のソースを取って来て、Input Bを食わせれば良い



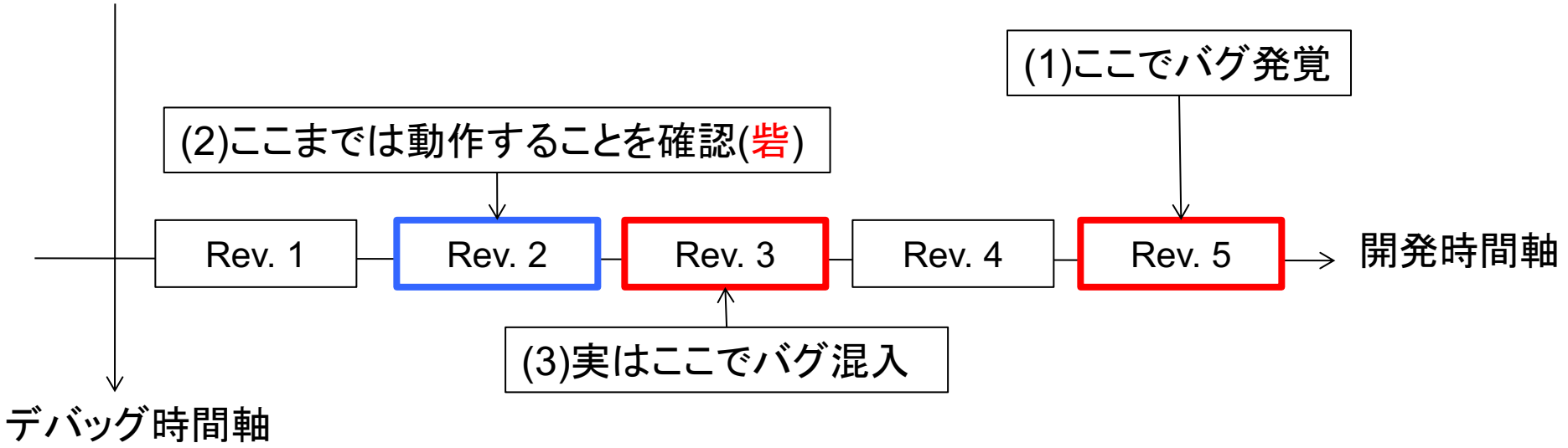
バージョン管理をしていると、問題の切り分けが容易

問題の切り分けとバージョン管理 (2/2)

昔入れたバグほど、デバッグが困難に (修正内容を忘れているから)

明日の自分は他人

バージョン管理していれば・・・



Rev. 2とRev. 3のdiffを取れば、どこが原因かがすぐわかる

デバッグ目的以外にも「あのジョブを実行した時のソースが欲しい」ということはよくある

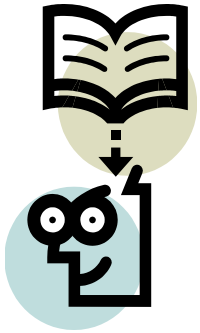
バージョン管理システムはタイムマシン

デバッグのまとめ

- バグったら、再現するコードを保存する (現場保全)
- いつバグが混入したか確認する (砦)
- バグに関係のないルーチンを削除していく (問題の切り分け)
- print文、assert文デバッグ (頭を使わない)

※ 統合開発環境やデバッガなどのツールも活用
とにかく原則として頭を使わないこと

デバッグ (プログラミング)とは
「ここまでは絶対大丈夫」
という砦を築いていく作業



デバッガの利用 (1/5)

デバッガとは？

デバッグの支援ツール
デバッグに便利な機能がたくさん含まれている
ほとんどの統合環境(IDE)にはデバッグ支援機能が含まれる
コマンドラインツールだと gdb が有名

何ができるか？

- ブレークポイント
- ステップ実行
- スタックトレース
- 変数監視
- その他非常に多機能

デバッガの利用 (2/5)

変数の書き換えタイミングを知りたい

変数の値がおかしくなった (assertにひっかかった)
でもソースのどこでその変数を書き換えているかわからない
(特にポインタやグローバル変数を多用したコードなどで発生)

```
int a = 0;
int main() {
  func1();
  func2();
  func3();
  func4();
  func5();
  func6();
  func7();
  func8();
  func9();
  assert(a < 10);
}
```

グローバル変数a (常に10未満であるはず)

このどこかでaを変な風にいじっている

ここでassertに失敗している

ウォッチポイント(watch)を使う

デバッガの利用 (3/5)

1. プログラムを「-g」オプションつきでコンパイル
2. 実行ファイルを指定してgdbを起動
3. ウォッチポイントの指定（条件 a>=10)
4. 実行

```
$ g++ -g test.cpp   (1)
$ gdb ./a.out       (2)
(gdb) watch a >=10  (3)
Hardware watchpoint 1: a >=10
(gdb) run           (4)
Thread 2 hit Hardware watchpoint 1: a >=10

Old value = false
New value = true
0x0000000100000cf8 in func5 () at test.cpp:9
9 void func5(){a = 15;}
```

test.cpp の 9行目にあるfunc5の関数内で問題の代入がされていることがわかった

デバッガの利用 (4/5)

不正な引数による関数呼び出しを検出したい

```
void func(int a){
  assert(a < 10);
  // Do something
}

int main(void){
  func1();
  func2();
  func3();
  func4();
  func5();
  func6();
  func7();
  func8();
  func9();
}
```

引数の値として a<10が想定されている

```
Assertion failed: (a < 10), function func, file test.cpp, line 7.
```

不正な引数で呼ばれたことはわかるが、
どこで不正な値が入ったかまではわからない

このどこかでfuncを不正な引数で呼んでいる

ブレークポイント(break)とバックトレース(bt)を使う

デバッガの利用 (5/5)

1. プログラムを「-g」オプションつきでコンパイル
2. 実行ファイルを指定してgdbを起動
3. funcにブレークポイントを指定
4. 先のブレークポイントに、条件(a>=10)追加
5. 実行 (a=11になっので止まる)
6. バックトレース(呼び出し履歴)の表示
7. 呼び出し元を表示(up)

```
$ g++ -g test.cpp   (1)
$ gdb ./a.out       (2)
(gdb) break func (3)
Breakpoint 1 at 0x100000ce1: file test.cpp, line 7.
(gdb) condition 1 a >= 10 (4)
(gdb) run (5)
Thread 2 hit Breakpoint 1, func (a=11) at test.cpp:7
7   assert(a < 10);
(gdb) bt (6)
#0  func (a=11) at test.cpp:7
#1  0x0000000100000cb1 in func7 () at test.hpp:8
#2  0x0000000100000d39 in main () at
(gdb) up (7)
#1  0x0000000100000cb1 in func7 () at test.hpp:8
8 void func7(void){func(11);}
```

func7の呼び出し方がまずいことがわかる

test.hppの8行目、func7内で、func(11)と呼んでいることがわかった

デバッガのまとめ

- ウォッチポイントにより、変数がいつ誰によって書き換えられたか検出できる
- バックトレースにより、ある関数がどういう履歴で呼び出されたのかをたどることができる
- 実行中の変数の値を逐一チェックできる
  - デバッガを使うとプログラムを「生きたまま」解析できる
  - print文デバッグ→静的な解析

デバッガは、使い方を覚えるまでのハードルは高いが、
プログラムを日常的に組むならその学習コストは
「必ず」元が取れる

今日のまとめ

- 頭を使うなツールを使え
- バージョン管理システムを使う
- デバッグのコストを意識する
  → バグを入れないプログラミング
  → すばやくデバッグするコツ

次回は高速化、チューニング、並列化のコツを扱います