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
4. ソース公開のススメ

自己紹介 (1/2)　私とプログラム

プログラム歴

小学校	PC-9801FでN88-BASICを触る
中学校～高校	Quick BASIC + アセンブラでMS-DOSのゲームを作成
大学～	Borland C++ BuilderでWindowsのフリーソフト作成
大学院	数値シミュレーション、単純並列計算
名古屋大学情報科学研究科	ActionScriptを覚えてFlashをいくつか作成
東京大学情報基盤センター	MPIによる非自明並列計算
東京大学物性研究所	大規模分子動力学法で研究

研究以外で作ったもの

MS-DOS

ドラクエ型RPG (文化祭用：サブプログラム、BGM、原画)
対戦アクションゲーム (コンテスト用：サブプログラム、BGM、原画)
ダンジョンRPG (卒業制作：サブプログラム、BGM)

Windows

対戦アクションゲーム (MS-DOSで作ったゲームの移植)
パズルゲーム2本 (プログラム、BGM)
スクリプト言語の統合開発環境、簡易エディタ
迷路作成プログラム ← ヤンメガの裏表紙
量子計算シミュレータ ← 未踏ソフトウェア

Flash

クリックアクションミニゲーム
パズルゲーム
交通流シミュレータや集団おにごっこシミュレータ

自己紹介 (2/2)　私とスパコン

2001年 (D1): 物性研 SGI 2800/384
2002年 (D2): 筑波大 CP-PACS
2003年 (D3): JAMSTEC 地球シミュレータ
2004年 名古屋大学情報科学研究科に赴任
2005年 物性研 SGI Altix 3700
2008年 東京大学情報基盤センターに赴任
2009年 九州大学 SR16000 (42ノード) 25.3TFLOPS
2010年 核融合研究所 SR16000 (128ノード) 77TFLOPS
2010年 物性研究所に赴任
2010年 物性研 SGI Altix 8400EX
2012年 情報基盤センター FX10 (4800ノード) 1PFLOPS
2014年 理研「京」10PFLOPS

本講義の目的

世の中の人は主に以下の二つのグループに分類される

一般人

普通の人
本講義の対象

逸般人

別名：廃人、人外
とても人間とは思えないプログラマ

本講義は、一般人の今後の作業時間の短縮を目的とする

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：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・・・

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

単体テストとsort+diffデバッグ

バグを入れないコーディング

バグを入れない方法

いろいろあるが、特に以下の二つの方法が有効 (一種のテスト駆動開発)

- 単体テスト
- sort + diff デバッグ

単体テスト

- テストしようとしている部分だけを切り出す
- その部分だけでコンパイル、動作するような最低限のインターフェース
- 最適化、並列化する前と後で結果が一致するかを確認する
- **本番環境でテストしない**

sort + diff デバッグ

- print文デバッグの一種
- 出力情報を保存し、sortしてからdiffを取る
- 単体テストと組み合わせて使う

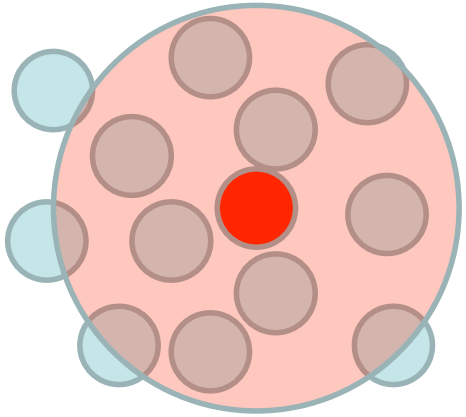
デバッグのコツ

「ここまでは大丈夫」という砦を築く

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？

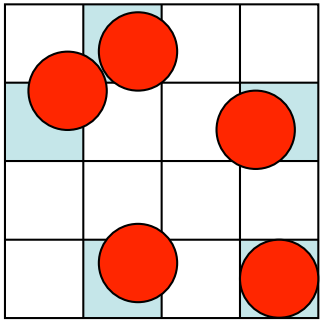
相互作用距離(カットオフの距離)以内にある粒子対のリスト
どの粒子同士が近いか？という情報
全粒子対についてチェックすると $O(N^2)$
高速に粒子対を作成する方法　→ グリッド探索



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

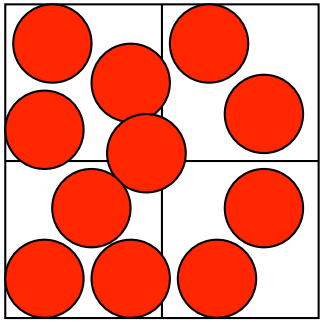
排他的グリッド(Exclusive Grid)法



一つのグリッドに粒子一つ

・短距離相互作用
・二次元
・高密度

非排他的グリッド(Non-Exclusive Grid)法



一つのグリッドに複数粒子

・長距離相互作用
・三次元
・低密度

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
```

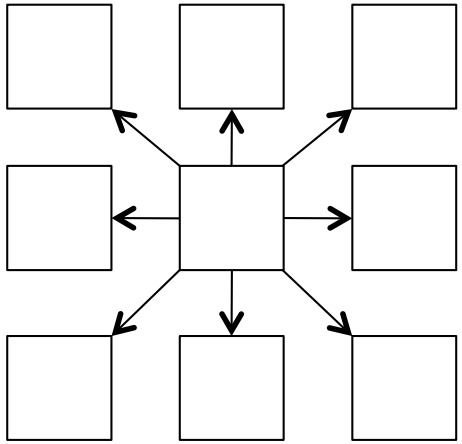
←結果が正しければdiffは何も出力しない

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)

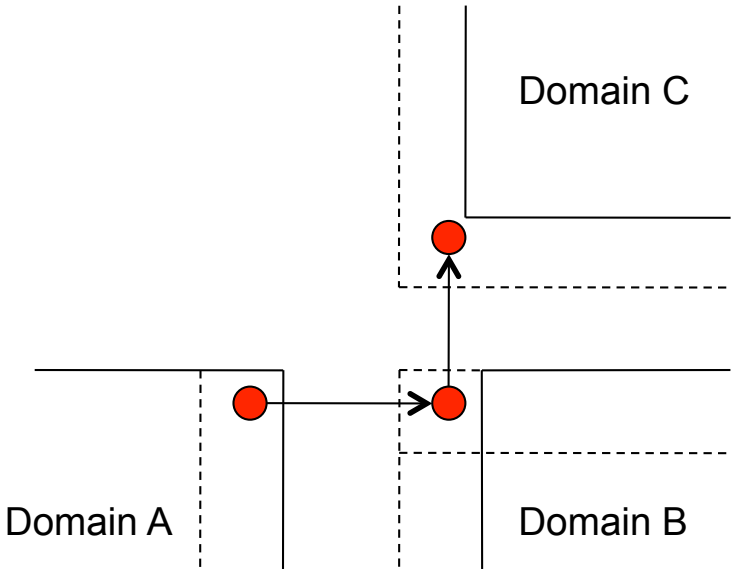
端の粒子の送り方

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

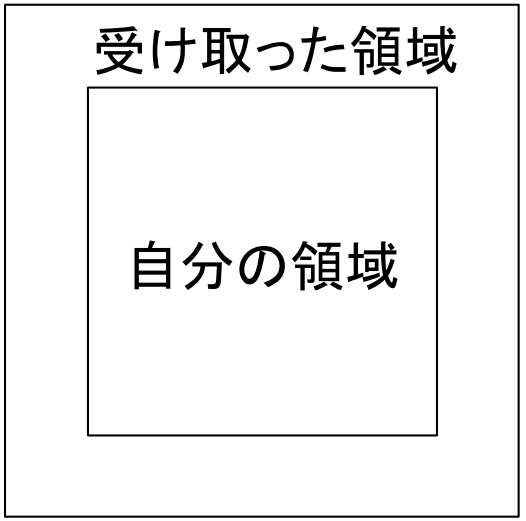
sort+diff デバッグの例2：粒子情報送信(2/2)

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
```

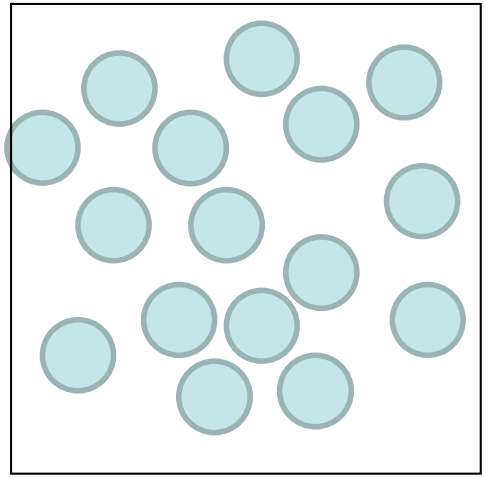
全てのプロセスについて一致することを確認
※ 複数の初期条件を試す事

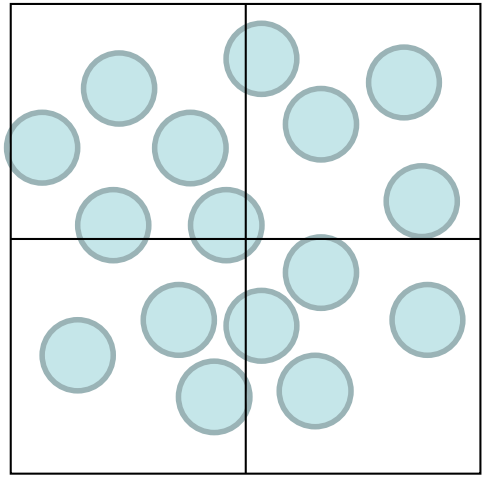


sort+diff デバッグの例3：並列版リスト作成(1/2)

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

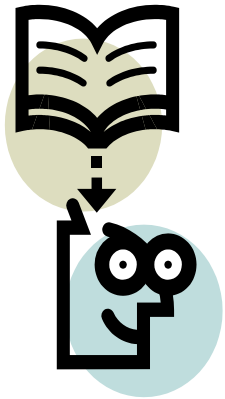
新しい機能の追加や高速化をするたびに単体テストする

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」

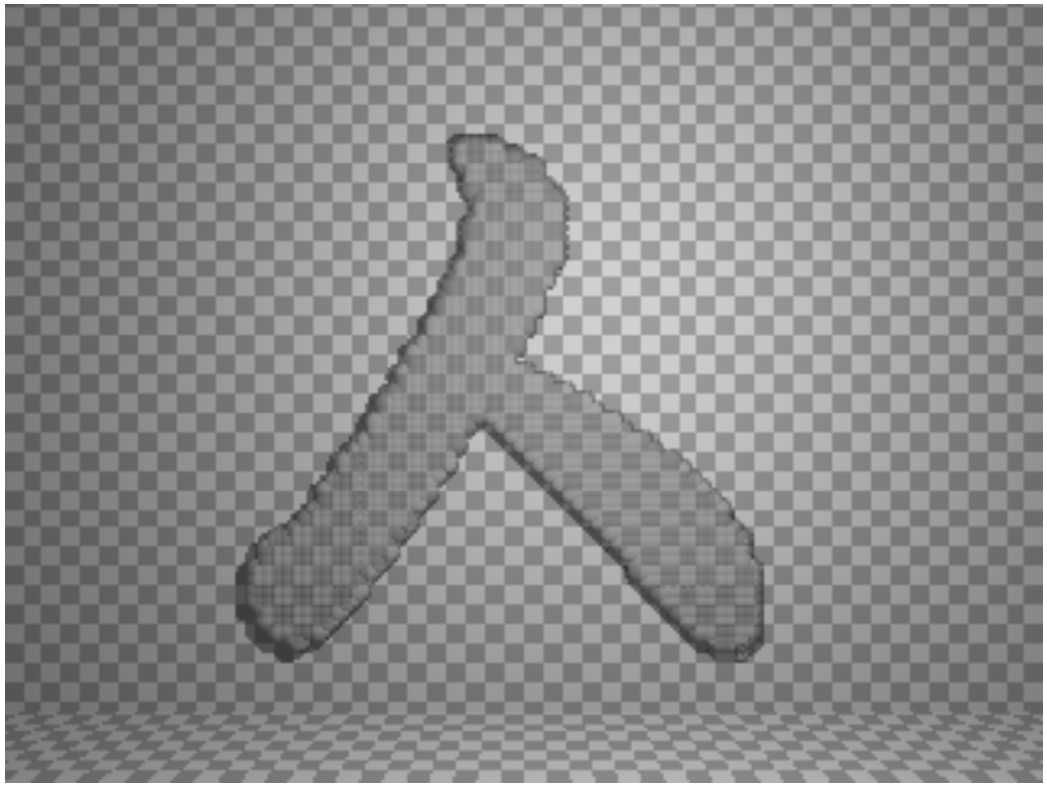
時間はかかるが信用できる方法と比較する
複数の機能を一度にテストしない



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

閑話休題

人と人が支えあったときのストレス



デバッグの方法論

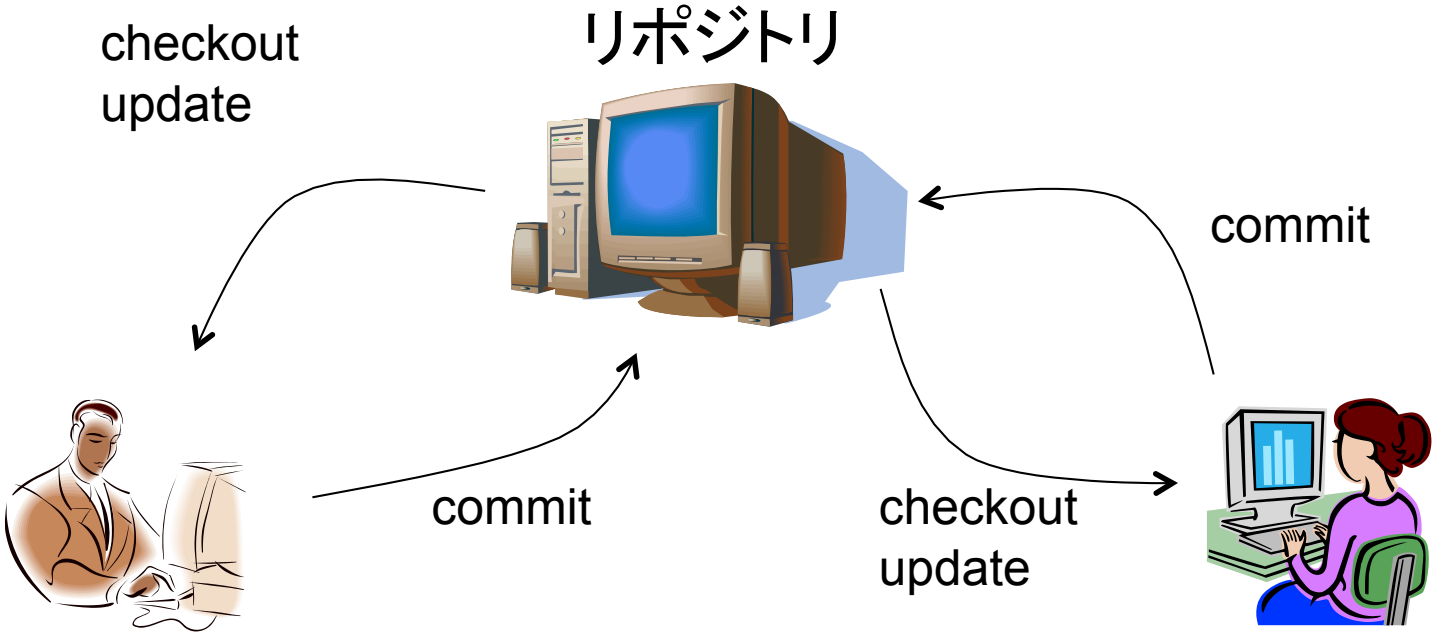
地雷型バグのデバッグ方法

デバッグの方法論・・・その前に

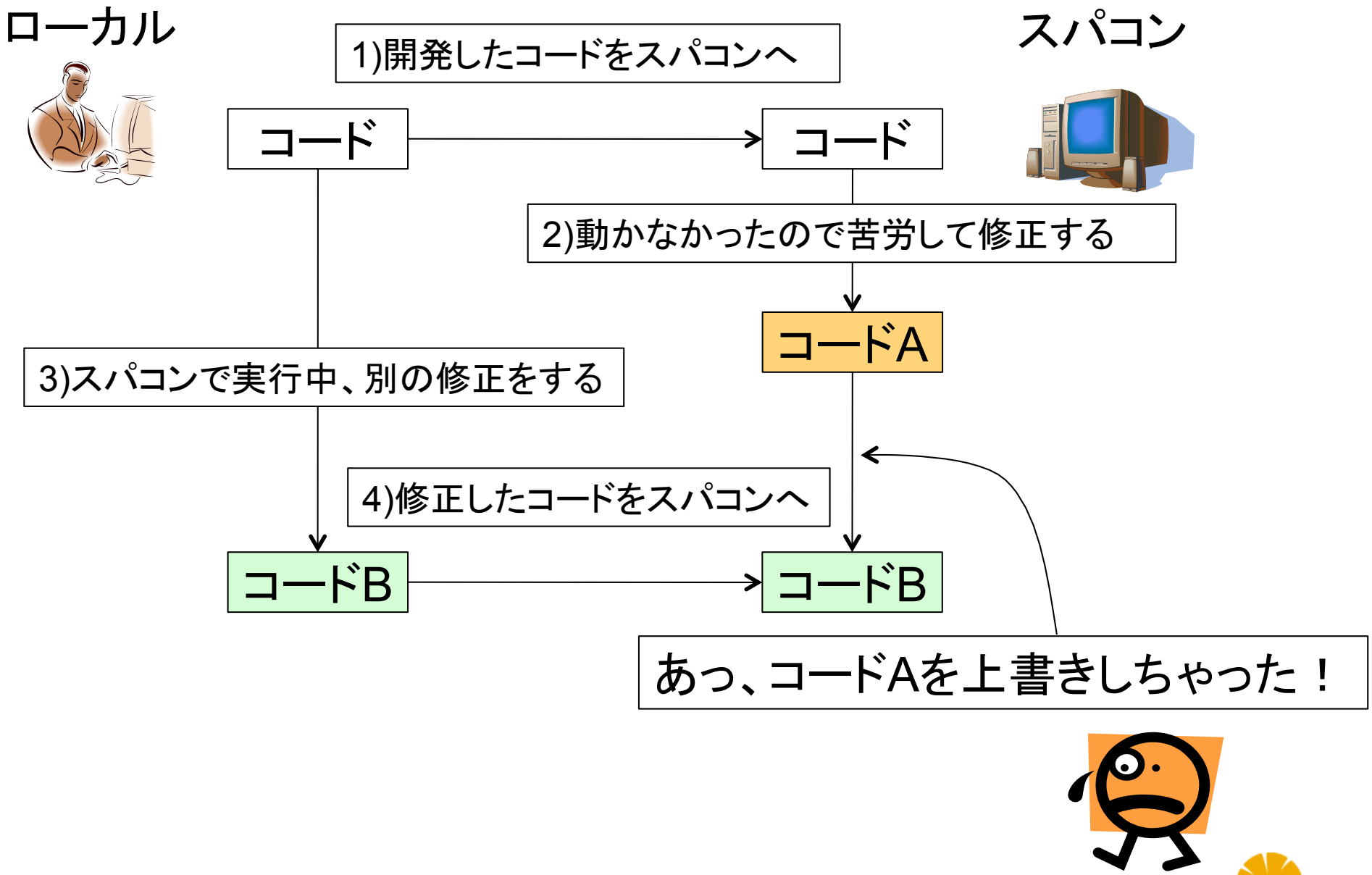
バージョン管理システム、使っていますか？(Y/y)

バージョン管理システムとは

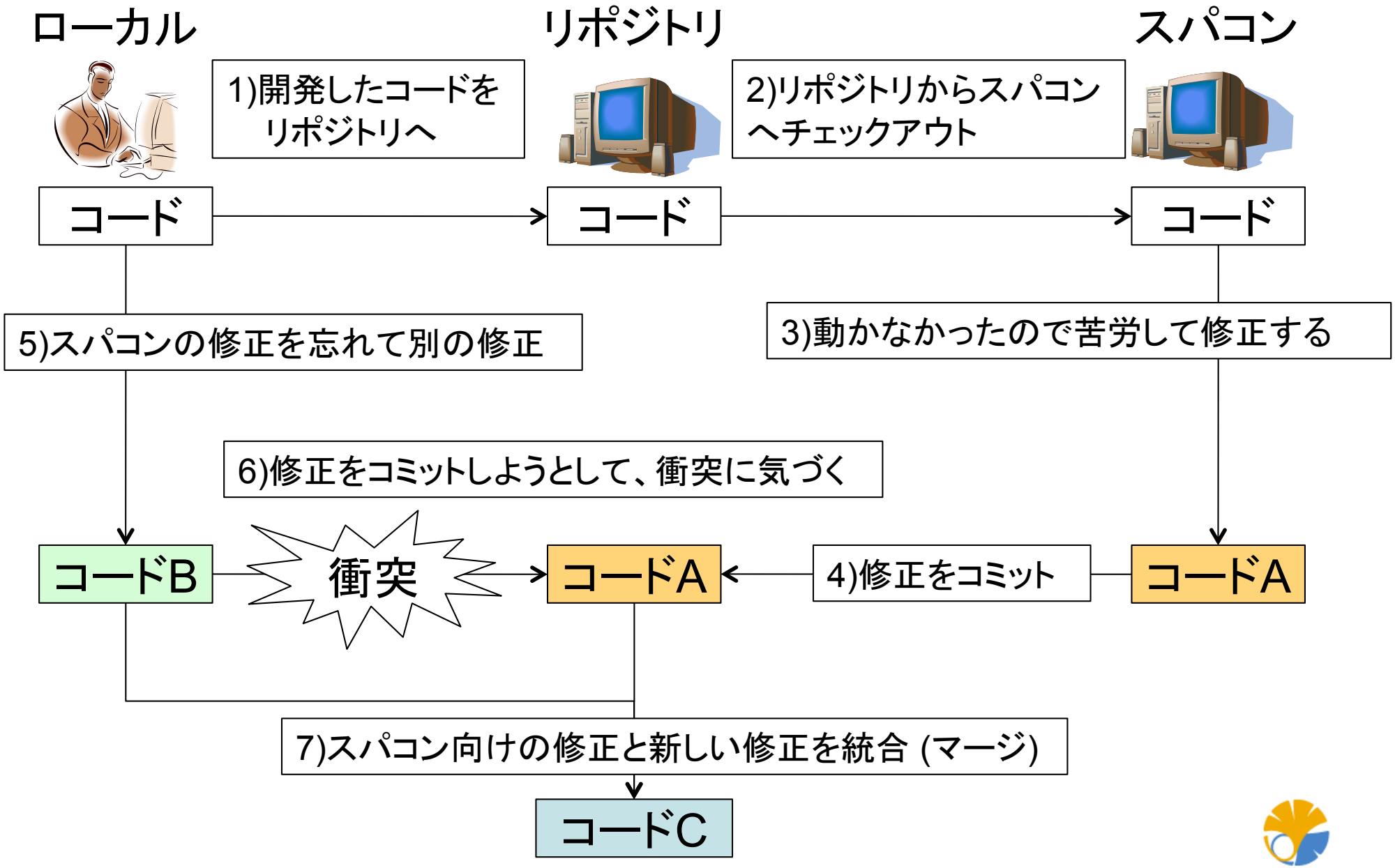
ファイルの編集履歴を管理するためのシステム
CVS, Subversion, Gitなどが有名
ファイルの編集履歴を全て保存する「リポジトリ」というデータベースをもつ
ユーザは、そのリポジトリにアクセスしながら開発を行う
超優秀な秘書のようなもの



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



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の利点

- (ちゃんとコミットしていれば)全ての編集履歴が保存される
  好きな時点のバージョンの呼び出しや任意のバージョン間の比較が可能
  → どのようにデバッグに役に立つかは後述
- 複数の環境でコードを開発しても混乱が少ない
- バックアップの代わりにもなる

バージョン管理システムの欠点 (面倒な点)

- 修正前に最新の状態にアップデートしなければならない
  → 慣れると習慣になります
- 全ての修正を「コミット」しなければならない
  → 慣れると習慣になります
- 衝突(コンフリクト)が発生した時に対処しなければならない。
  → 衝突に気づかずに修正してしまうほうが怖いです

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

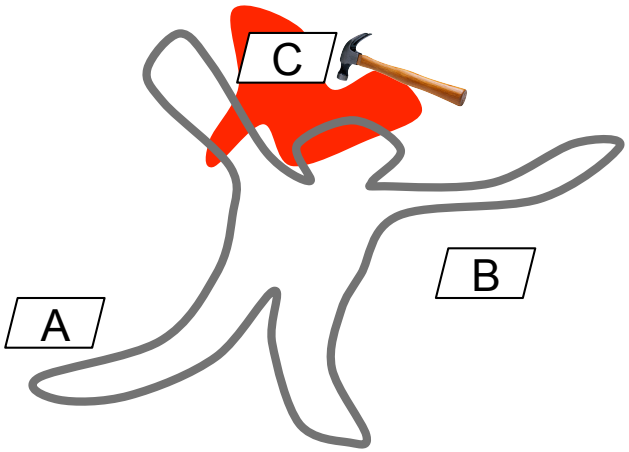
地雷型バグとは？

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

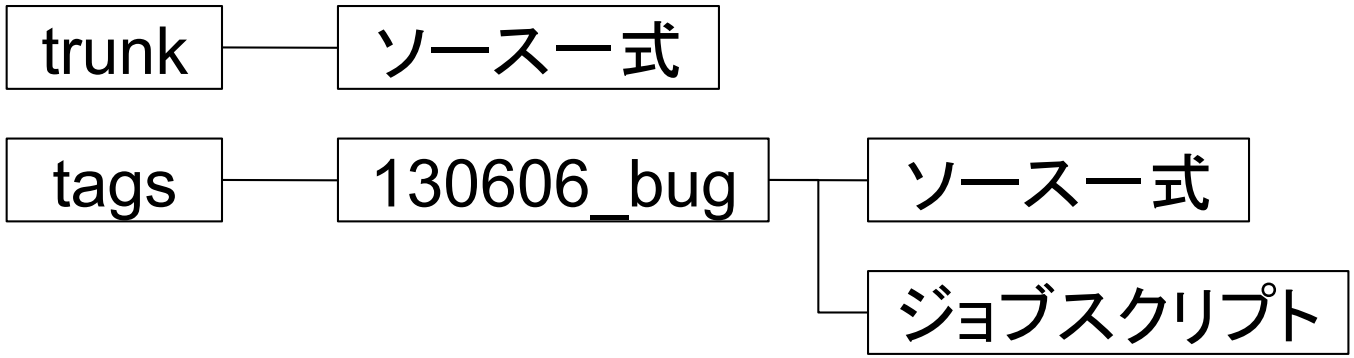


(1) 再現性テスト (同じ条件で実行したら同じバグを発生するか？)
(2) バグを起こすソース一式を保存しておく (Subversionならタグ)
(3) バグを再現する最低限のコードを切り出す (容疑者の限定)

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
デバッグが終了したら消しても良い (消去したことも含めて記録される)

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.
```

バグの例

与えられた整数Nについて、N未満の数字をランダムに返す関数が欲しかった

randは0からRAND_MAXまでの整数を返す関数
(RAND_MAX=2147483647)
それをRAND_MAXで割れば、0から1までの実数を返すはず？

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

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

randは最高でRAND_MAXの値を返すので、myrand_intは低確率でNを返す

```
const int N = 10;
double data[N];
int index = myrand_int(N); ← ここがバグの原因
// (ずっと遠くで)
return data[index]; ← 低確率で領域外参照が発生
```

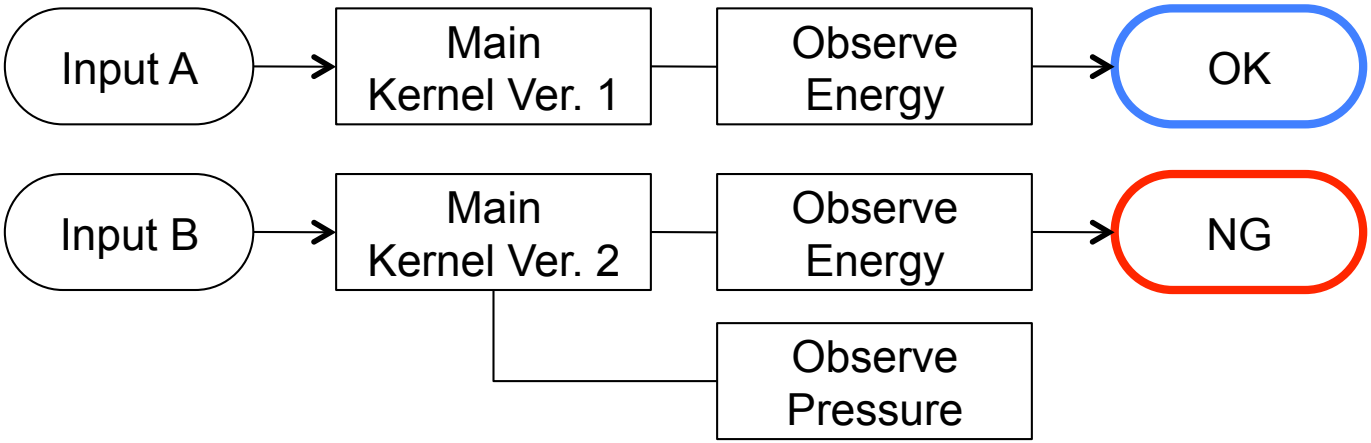
この種のバグの原因に「最初から思い至る」のは難しい
print文+assert文デバッグが有効

問題の切り分けとバージョン管理 (1/2)

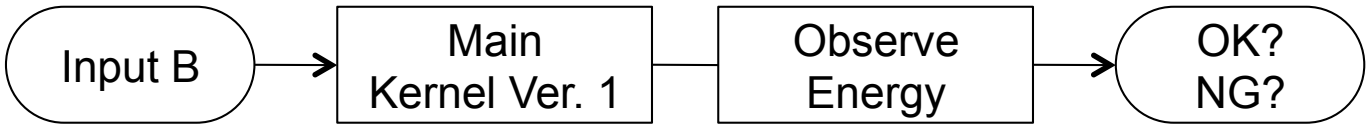
機能を追加したらバグった？

→ 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



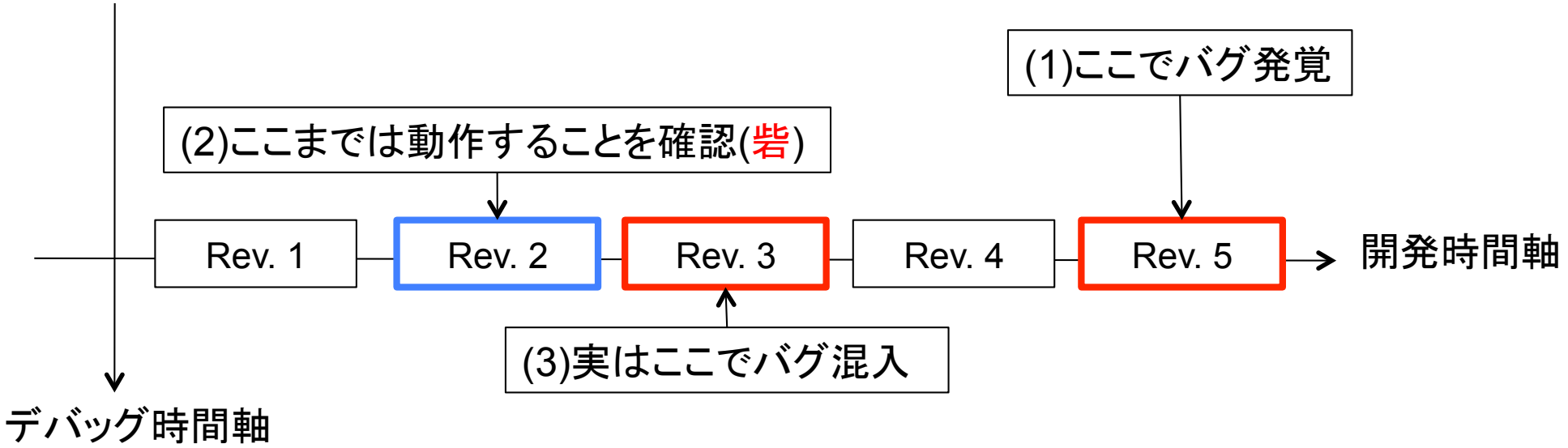
バージョン管理をしていると、問題の切り分けが容易

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

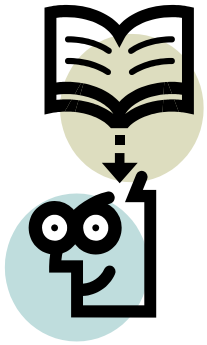
University of Tokyo

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を使っても良いが、
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



ソース公開のススメ

作ったプログラムをどうするか

ソフトウェア資産の一生

何かプロジェクトを提案して予算を獲得する
その予算でPDを雇ってプログラムを開発する
プロジェクト終了とともにプログラム開発ストップ
そのまま誰にも使われずに朽ちて行く・・・

なぜそうなるのか？

プログラムは生き物であり、メンテしないと死んでしまう
プログラムのメンテには開発者としての愛着が必要
予算ありきでプログラムを作ると基本的には同じ道を辿る

どうにかできないのか

予算を取ってプログラムを作るのではなく、開発されているプログラムを
予算によって支援する
開発段階からソースを公開する

なぜソースを公開するか

ソース非公開ということ

ソースが非公開だと、そのプログラムはブラックボックスになる
ブラックボックスのプログラムは
- 安定していなければならない
- マニュアルが整理されていなければならない
- 開発がとまった時がプログラムの死ぬ時

オープンソースソフトウェア

ソースを公開していれば・・・
- ユーザが必要な時に自分で機能変更ができる
- 質問があったら「ソース読め」と言える
- 開発が止まっても、別の人が開発を引き継ぐ可能性がある
  (そのプログラムの一部機能が取り込まれていくこともある)
- 公開するつもりで書くと、プログラムがきれいになる

ソース公開の難しさ

えらい先生が反対する

せっかくの技術、ノウハウが流出する
→　技術、ノウハウの流出は分野振興にとって望ましいことのはず

そもそも「サイエンス第一」なんでしょ？

公開は恥ずかしい

自分のプログラムはつたないので、公開するのが恥ずかしい
→公開して恥ずかしくないようなプログラムを組めるように努力する
バグってたら恥ずかしい
→バグってるプログラムで論文書いちゃダメです

公開するためには

最初からソースを公開すると宣言し、賛同する人だけでプロジェクトを開始する
→最初あいまいにしておいて、複数人が関わったあとに公開するのは不可能

※ おそらく多くの「えらい先生」は、ソースを公開するというマインドがない

ソフトウェアの公開方法

公開場所

公開場所として大学のサーバは良くない
→開発者が異動することが多いから
まして年限つきのプロジェクトのサーバに置くのはダメ
→プロジェクト終了後、サーバも消えるかやっかいもの扱いされる運命

OSSホスティングサービス

ソースコードリポジトリがおすすめ
SourceForge.net, OSDN, GitHub ...

リポジトリのみならず、バグトラック、フォーラムなどがあり、
ソフトウェア開発基盤として有用

※ ソース公開が難しい場合もgoogle sitesなどを利用
なるべく大学におかない

SorceForge.net の例

- URLが取れる

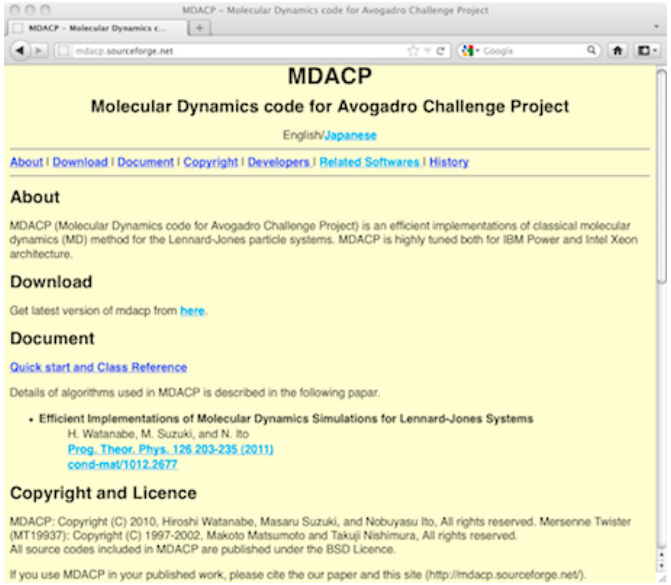
  http://mdacp.sourceforge.net/

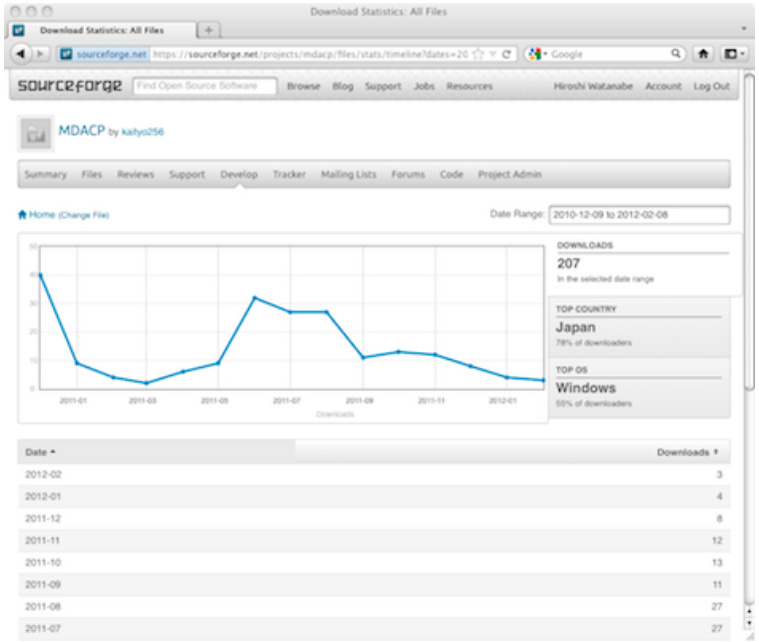
  - ウェブサイトは自由に作成できる (ウェブホスティング)
  - リポジトリを作ることができる

  CVS, Subversion, Git, mercurial ...
  - その他開発環境のためのサービス

    チケット、Wiki、バグトラッキング

  - ダウンロード統計なども取得できる

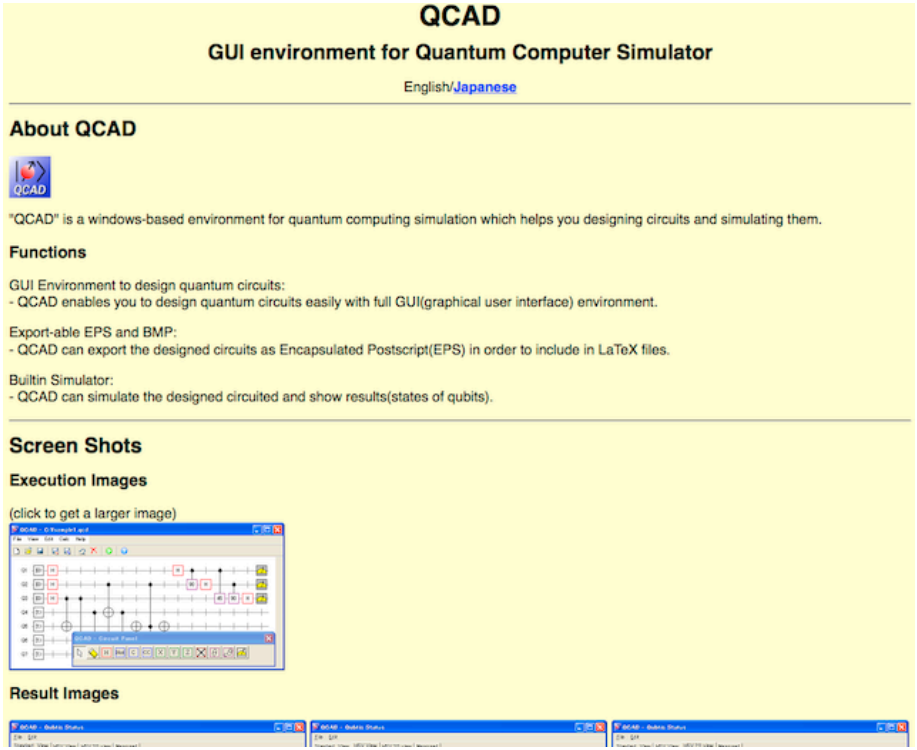




OSDN の例

SourceForgeの日本版。本家と若干インタフェースが異なる

http://qcad.osdn.jp/



ウェブサイトは自由に作成/編集できる

https://osdn.jp/projects/qcad/devel

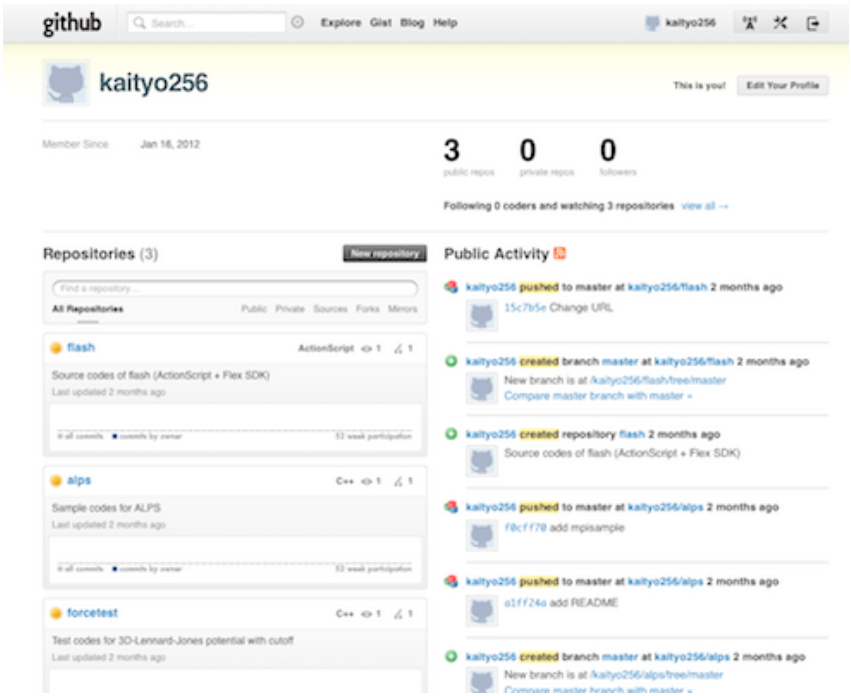
ダウンロード統計なども取得できる

※ 2015年5月にSourceForge.JPからOSDNに名称変更
※ 個人的には本家よりSF.jpの方が使い易かった

University of Tokyo

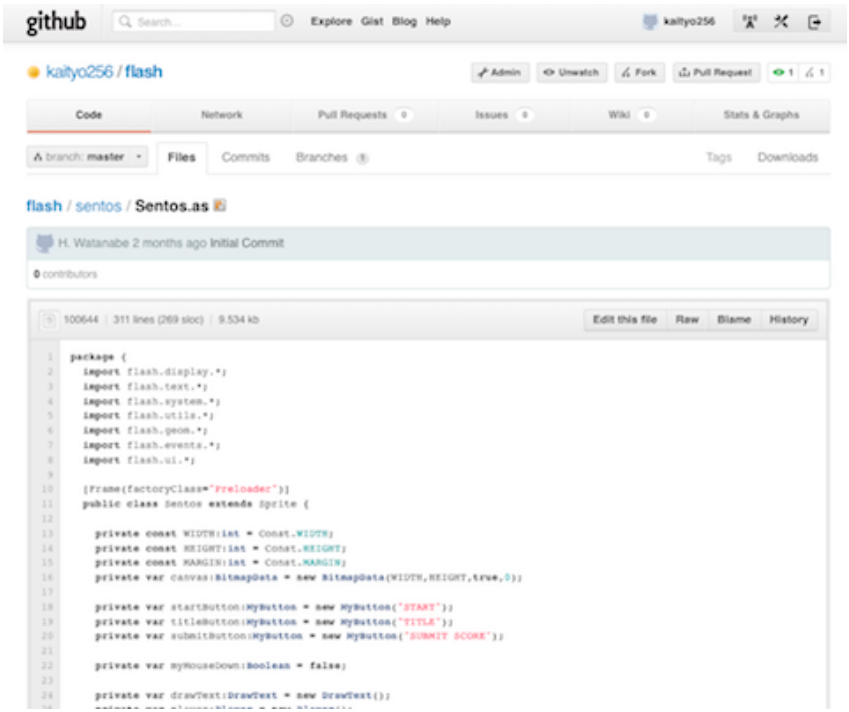
GitHubの例

https://github.com/kaityo256



SNS機能をもつ

https://github.com/kaityo256/flash/blob/master/sentos/Sentos.as



ソースが色付きで表示されたりする

※ 他にもプルリクやチケットなど様々な機能がある

ソース公開のまとめ

・Academiaで開発されたプログラムはソースを公開しないと
ほぼ死ぬ運命(偏見)

・プロジェクトでソフトウェアを作るとだいたいうまくいかない。

・うまくいくのは、「もともと作って公開するつもりだったソフトウェア」
をプロジェクトの形で支援、環境を整えること

・プロジェクトの支援とは？
  ・公開場所を提供することではない
  ・マンパワーを提供することでもない
  ・開発者がそのプログラムに集中することを仕事として認めること

今日のまとめ

- 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
- ソースはなるべく公開する
  → 開発に費やしたコストを意識する

次回は高速化、チューニング、並列化のコツを扱います