第6回
可読性と性能の両立を目指して

渡辺宙志

東京大学物性研究所

2015年11月5日@計算科学技術特論C

University of Tokyo

可読性と性能の両立を目指して: Agenda

- 本講義の対象者
- 本講義の目的
- 並列分子動力学法コードMDACPについて
- プログラム設計
  - クラス、継承、仮想関数
  - モジュールの結合度
  - インプットファイル
  - 複数プロジェクトの扱い
  - main関数の引数の扱い
  - 力の計算ルーチン
- 並列化と設計
  - 通信の設計
  - プロセス配置管理
  - 並列構造の隠蔽
- ソースを公開するということ

自己紹介

プロフィール

自分はプログラマだと思っている(最近はSEだと思っている)
普段使う
　Ruby/C++
使ったことがある/たまに使う
　Fortran/Java/Perl/Python/JavaScript/ActionScript

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～大学院	Borland C++ BuilderでWindowsのフリーソフト作成 PerlでCGI作成 Javaでバイト
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 RubyでCGI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

これまでに作ったもの (1/4)

スクリプト言語の簡易実行環境 (B3)

NEWS

MS-DOSウィンドウを表示せずにPerlスクリプトを実行できる「Copal」v1.50

実行結果やエラーを「Copal」のウィンドウ内に表示

(99/11/12)

MS-DOSウィンドウを表示せずにPerlやAWKなどのスクリプト言語を実行するソフト「Copal」v1.50が、11日に公開された。Windows 95/98で動作するフリーソフトで、現在作者のホームページからダウンロードできる。

「Copal」は、PerlやAWKなどのスクリプト言語を、MS-DOSウィンドウを表示せずに実行できるソフト。実行したいスクリプトを開くと、編集用のウィンドウにスクリプトの内容が表示され、ツールバーのボタンを押すだけでスクリプトを実行してくれる。

標準出力への実行結果や実行時に発生したエラーも、「Copal」のウィンドウ内に表示される。MS-DOSプロンプトを起動してコマンドラインから実行する場合と異なり、スクリプトを修正しながら手軽に実行できるようになる。なお、実行時にコンソールからの入力を求める対話型のスクリプトには対応していない。

窓の杜 (株式会社インプレス)

これまでに作ったもの (2/4)

パズルゲーム (M2)

NEWS

同じ色のブロックを4つ並べて消すパズルゲーム「☆星工房☆」v0.80β

"スターブロック"を上手に使ってすべてのブロックを消そう

(01/01/12)

同じ色のブロックを4つ並べて消すパズルゲーム「☆星工房☆」v0.80βが、12日に公開された。ブロックを押して移動させ、同じ色のブロックを4つ並べて消していく。Windows 95/98対応のフリーソフトで、現在作者のホームページからダウンロードできる。

「☆星工房☆」は、同じ色のブロックを4つ並べて消すパズルゲーム。空から落ちてバラバラになった星を空に返すため、プレイヤーは"星の掃除屋さん"を操作して星のかけらのブロックを集めていくというストーリーだ。カラフルなブロックを押して移動させ、同じ色のブロックを4つ並べると消すことができる仕組みで、荷物を押して所定の位置に配置する往年のパズルゲーム「倉庫番」と、同じ色の駒を4つつなげて消すパズルゲーム「ぷよぷよ」を組み合わせたようなルールとなっている。ブロックはまとめていくつでも押すことができ、すべてのブロックを消すとステージクリアとなる。星マークのついた"スターブロック"はオールマイティで、どの色と組み合わせても消せるだけでなく、色の違うブロックを複数組

窓の杜 (株式会社インプレス)

これまでに作ったもの (3/4)

量子計算シミュレータ (D1)

QCAD

GUI environment for Quantum Computer Simulator

English/Japanese

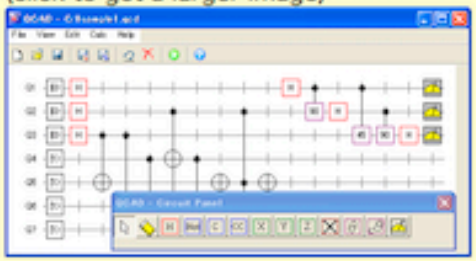
このソフトウェアについて

QCADは、量子計算機用の回路図を作るためのソフトです。 GUIで簡単に回路図が書けます。書いた回路は独自形式のほか、eps形式、bitmap形式で保存できます。 作成した回路はその場でシミュレートし、結果を表示することができます。

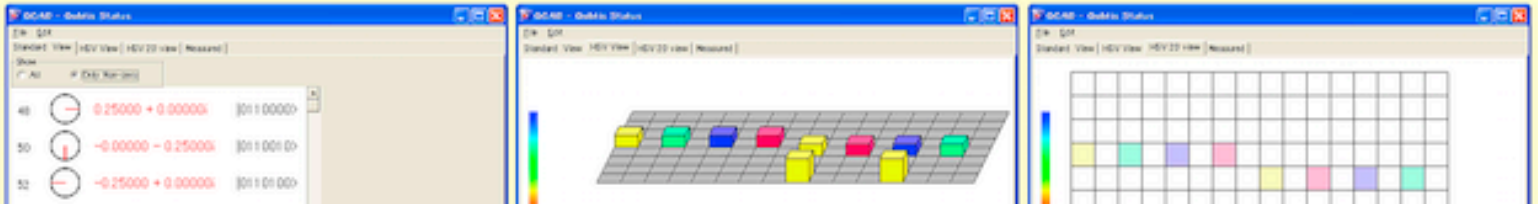
スクリーンショット

実行画面

(click to get a larger image)



結果出力画面



http://qcad.osdn.jp/

これまでに作ったもの (4/4)

迷路作成ツール (名大3年目)

オンラインソフト紹介サイト
窓の杜
WINDOWS FOREST
同一ジャンルソフト記事
● 同じ「ゲーム」ジャンルのソフト記事を読む

NEWS (06/09/04 12:50)

同じジャンルのソフトを探す ゲーム

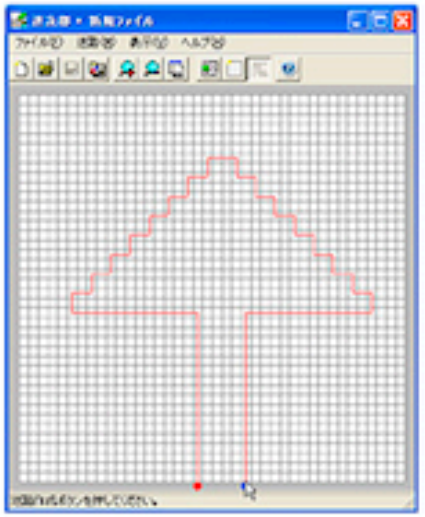
一筆書きでゴールに着くと絵が浮かびあがる迷路を作成「迷次郎」

作成した迷路は問題・正解の両方をBMP画像で保存可能

一筆書きでゴールに着くと絵が浮かびあがる迷路を、画像をトレースするように作成できるソフト「迷次郎」v1.01が、8月16日に公開された。Windowsに対応するフリーソフトで、編集部にてWindows XPで動作を確認した。現在作者のホームページからダウンロードできる。

「迷次郎」は、スタートからゴールまで正解ルートを通る線を引くと、絵が浮かびあがる迷路を作成できるソフト。本ソフト自身のウィンドウを透過表示できるのが特長で、別ソフトで表示させた画像などをなぞるようにして迷路を作成可能。

本ソフトを起動すると、方眼紙のような迷路作成画面が開く。画面



「迷次郎」v1.01

窓の杜 (株式会社インプレス)

University of Tokyo

本講義の対象者

- 趣味ではなく、仕事でプログラムを組もうと思っている人
- 自らプログラムを書き、そのプログラムで論文を書く人
- 何かソフトウェアを公開しようと思っている人

本講義の目的 (1/3)

プログラムを組む時に最も大事なことは何か？

妥協

本講義の目的 (2/3)

マーク・ザッカーバーグ(Facebookの創業者)曰く

Done is better than perfect.
完成を目指すより、まず終わらせよ

Code wins arguments.
コードは議論に勝る

Facebookの新規株式公開(IPO)申請書類に添付された手紙より
http://www.sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm

本講義の目的 (3/3)

- 完成しないプログラムに価値は無い
  →どこかで妥協
- 「妥協」に正解は無い
  →どこを妥協せず、どこを妥協すべきか？

以上について、主に短距離分子動力学法コードMDACPの開発、公開を通じて考えたことを紹介します。

MDACP (1/4)

MDACPとは？

Molecular Dynamics code for Avogadro Challenge Project
KAUST GRP Investors 賞受賞研究
「アボガドロ数への挑戦ー 非平衡現象の計算統計物理学 ー」
というプロジェクトで開発

アルゴリズム

短距離古典分子動力学法
現在はカットオフ付きLennard-Jonesポテンシャルのみ実装

研究対象

主に気液相転移を研究
気液転移の普遍性(平衡状態)
気泡生成待ち時間分布(準安定状態)
多重気泡生成(非平衡非定常状態)

MDACP (2/4)

開発の背景

類似の並列MDコードは多数存在する
→ LAMMPS/NAMD/SPaSM/MODYLAS
ただ論文を書くだけなら上記のコードでもなんとかなる
しかも現在のMDACPよりも高機能

僕が「物理学者」なら、既存のコードを使っていたと思う
でも僕は「プログラマ」なので・・・

自分で書いたプログラムで論文を書きたい
世界一に挑戦してみたい

MDACP (3/4)

コード概要

言語: C++
並列化: Ver. 1.0系 Flat-MPI
Ver. 2.0系 MPI/OpenMPのハイブリッド並列
規模: 5000行程度 (*.ccと*.h合わせて50ファイル)
公開:オープンソース (http://mdacp.sourceforge.net/)
ライセンス: 3条項BSDライセンス

設計思想

- 開発のしやすさと実行速度をなるべく両立
  → ただし、速度と設計がぶつかったら速度を優先
- 他人がアプリケーションがとして利用することは考えない
  → ソース公開の目的は「ユーザ」ではなく「開発者」に向けて
- なるべく外部ライブラリに依存しない

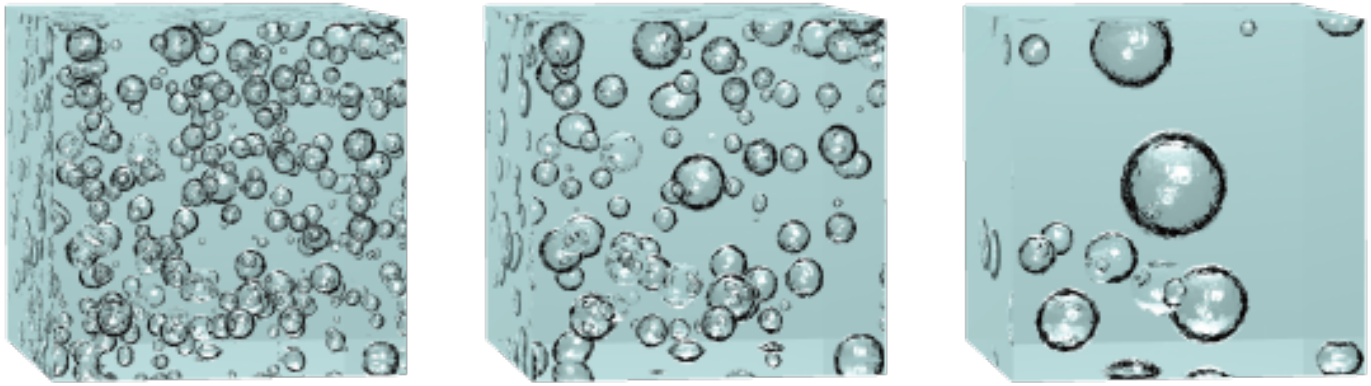
MDACP (4/4)

性能

京フルノード: 82944ノード (663552コア)
ベンチマーク
- 最大粒子数 : 3318億粒子 (1.77PF, 16.6%)
- 最高性能　　: 12億7千万粒子 (2.44 PF, 23.0%)

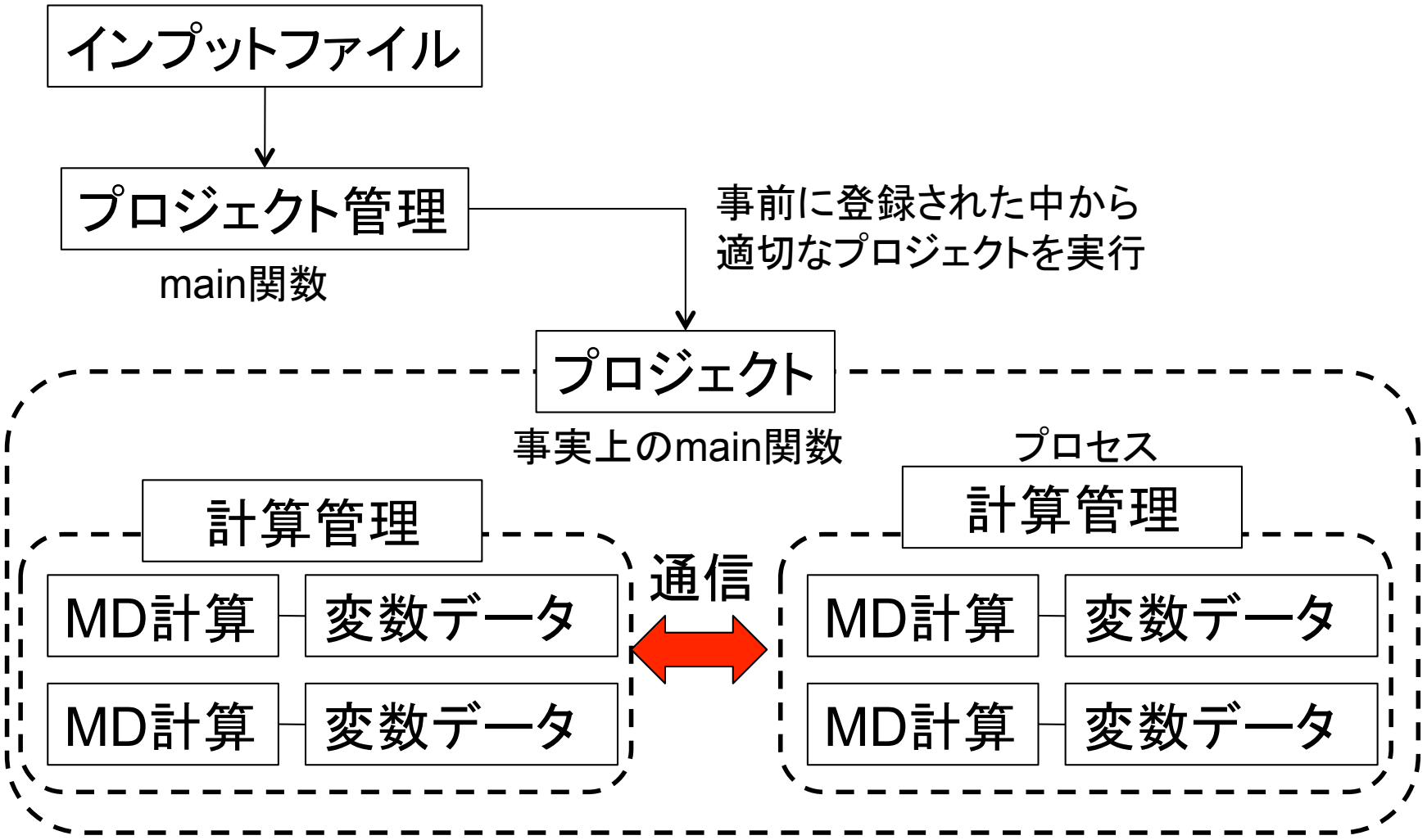
プロダクトラン :
- 13億7千万粒子　(1.81PF, 17.0%)
- 急減圧シミュレーション

急減圧直後
多数の泡が生成
→
大きな泡がより大きく
小さな泡がより小さく
→
最後には一つの
泡に収束



※ これらは4096ノードで計算した小さい系(2300万粒子)を可視化したもの

MDACPのプログラム構造



プログラム設計

クラス、継承、仮想関数 (1/2)

クラス

特定の機能を実現するため、データと処理(メソッド)をまとめたモジュール
継承により、子クラスを作ることができる

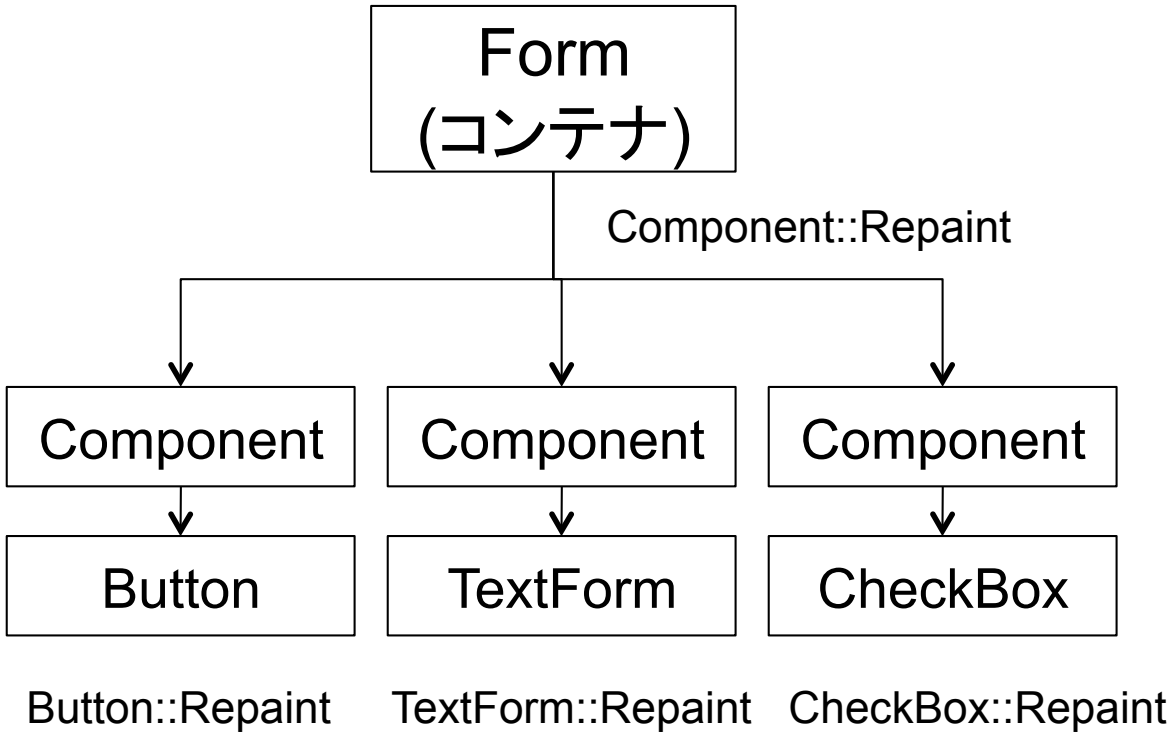
継承

既存のクラスを継承したクラスを派生クラスと呼ぶ
多くの場合、仮想関数と共に使われる

仮想関数

継承元のクラスの動作を上書きする
仮想関数は、実際にどのメソッドが呼ばれるかが実行時まで決まらない

クラス、継承、仮想関数 (2/2)



Formは、Component達を管理していると思っている
再描画が必要な時、FormはComponent::Repaintを呼ぶ
実際には、派生クラスのメソッドButton::Repaint等が呼ばれる

コンポーネントの種類が増えても、Formクラスの再コンパイルは不要

University of Tokyo

モジュールの結合度

結合度

二つ以上のプログラムモジュール間の関係を表す
お互いの設計に依存関係があるほど、結合度が強い
お互いの設計が独立であるほど、結合度が弱い

モジュール間の結合が強いコード→局所的な修正が全体に波及

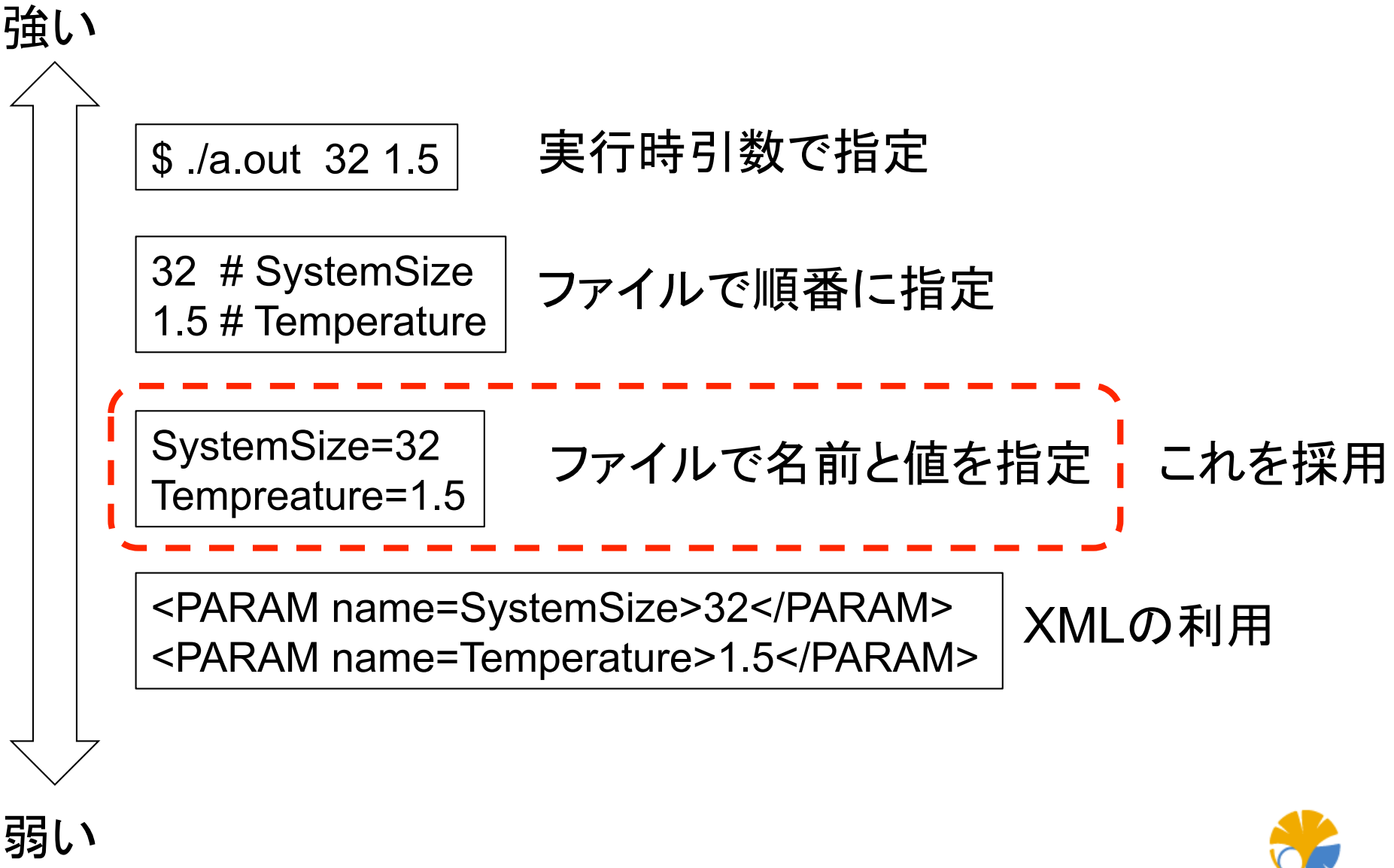
モジュール間の結合が弱いコード→修正の波及が局所で収まる

結合度が弱いほど独立性、拡張性が高い → 良いプログラム

・・・なのだが、実際には結合度を最低にしない設計を採用することが多かった

インプットファイル (1/3)

プログラムとの結合



インプットファイル (2/3)

結合度と可読性

XMLを採用するとlibXMLなどのライブラリの利用が必要
手で編集するのにXMLは不向き
いま与えたい情報に対してXMLはオーバースペック
→ XMLは使わない

※ 直接手でいじらないデータをXMLで保存するのは良いと思う

実装

- Parameterクラス(parameter.cc/h)
- 文字列ハッシュで管理 (std::map)
- 「名前=値」の形式でずらずらならべる
- Parameterクラスは全て文字列として保存
- 値を取り出す時に文字列を数値などに解釈

インプットファイル (3/3)

流れ

1. 実行可能ファイルにインプットファイル名を渡す
2. main関数でファイル名を受け取り、計算管理クラスに渡す
3. 計算管理クラスが、受け取ったファイル名からParameterクラスのインスタンスを作成
4. Parameterクラスのインスタンスを通じて値を受け取る

値の受け取り

ハッシュキーを渡し、値を受け取る
double Parameter::GetDouble(string key)
ハッシュキーが定義されていなかった場合、valueを値とする (より安全)
double Parameter::GetDoubleDef(string key, double value)

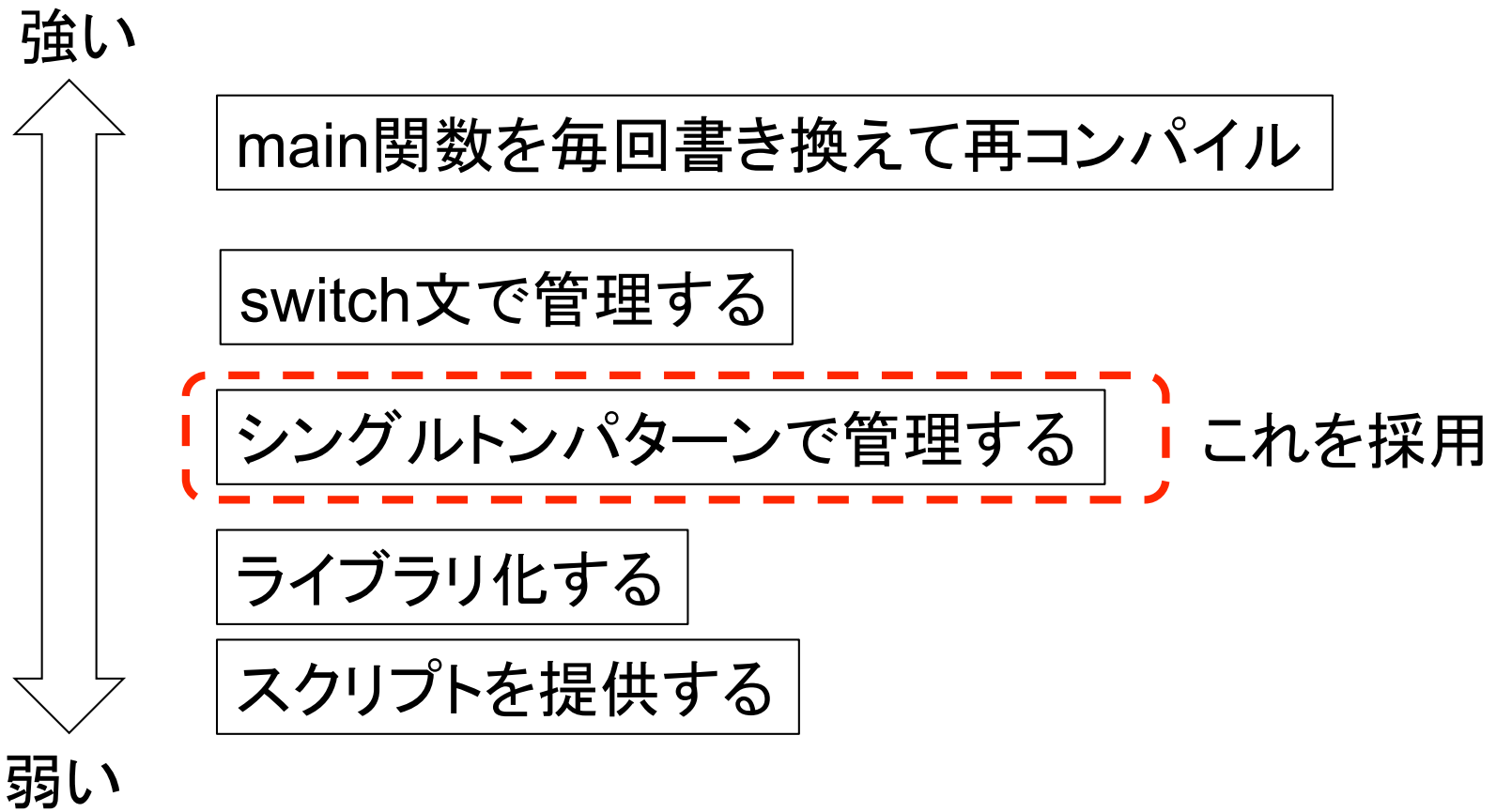
特殊キー「Mode」

インプットファイルは必ずModeを定義しなければならない
→複数のプロジェクトの管理(後述)

複数のプロジェクトの扱い(1/5)

プロダクトコードは、一つのカーネルを様々な研究に使う
(ベンチマークコードとの最大の違い)

プログラムとの結合



複数のプロジェクトの扱い(2/5)

プログラムと強結合

毎回mainを書き換える

```
int
main(void){
  //ProjectA();
  //ProjectB();
  ProjectC();
}
```

毎回再コンパイルが必要

switch文で管理する

```
switch(mode){
 case PROJECTA:
     ProjectA();
  break;
 case PROJECTB:
     ProjectB();
  break;
}
```

プログラムは呼ばれる可能性のあるプロジェクトを事前に知っていなければならない

こういうのは絶対イヤ

複数のプロジェクトの扱い(3/5)

プログラムと弱結合

プログラムにスクリプトを食わせる形式 (ex: LAMMPS)

```
# 2d polygon nparticle bodies
units          lj
dimension      2
atom_style     body nparticle 2 6
read_data      data.body
velocity       all create 1.44 87287 loop geom
pair_style     body 5.0
pair_coeff     * * 1.0 1.0
neighbor       0.3 bin
fix                        1 all nve/body
fix                        2 all enforce2d
#compute       1 all body/local type 1 2 3
#dump          1 all local 100 dump.body index c_1[1] c_1[2] c_1[3] c_1[4]
thermo         500
run            10000
```

次元、力場、温度制御、
ループ数などを指定する

github: lammps/examples/body/in.body

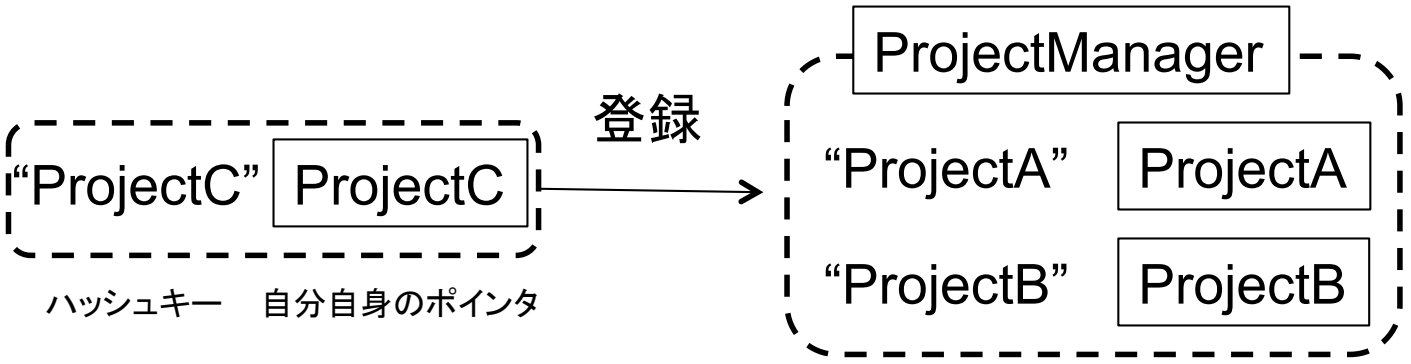
利点: 新しいプロジェクトを作っても再コンパイル不要
→ ユーザから見て使いやすい

問題点: プログラマは大変
→ 使う機能を全て事前に用意しておく必要がある

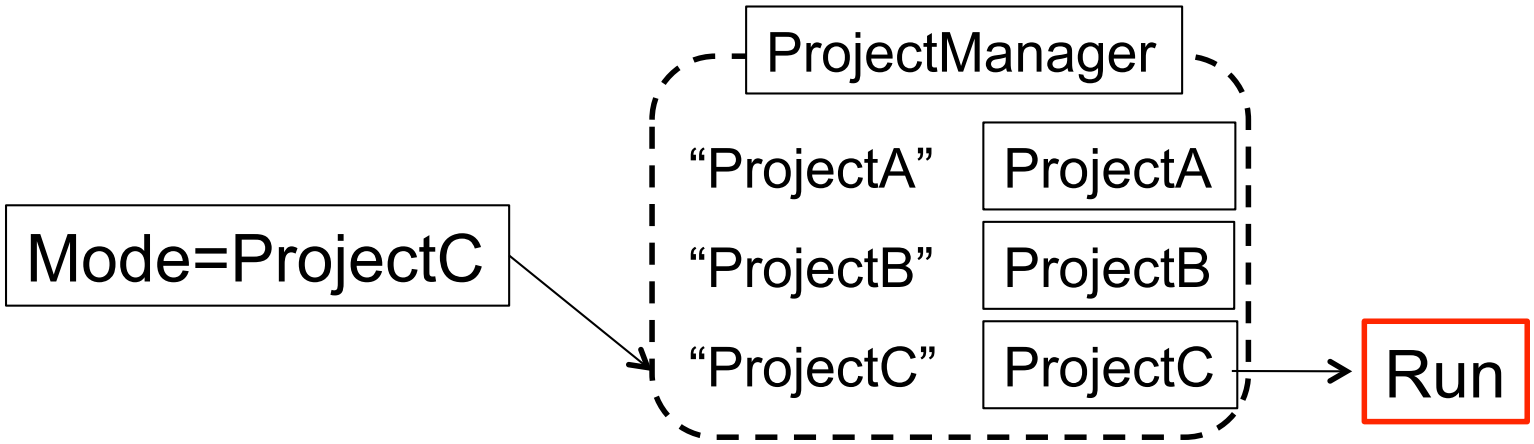
複数のプロジェクトの扱い(4/5)

Singletonパターンによる実装

1. ユーザはProjectクラスから派生したクラス(ProjectC)を実装
2.コンストラクタで自分をProjectManagerに登録する



3. ハッシュキーからProjectを探し、Runメソッドを呼ぶ



複数のプロジェクトの扱い(5/5)

設計思想

プロジェクトごとにmain関数を独立に書くイメージ
プロジェクトを増やしても、本体は再コンパイル不要

実装

ProjectManagerクラスをSingletonに
全てのプロジェクトはProjectクラスのサブクラスに
コンストラクタで自分をProjectManagerに登録
Project::Runの中身が事実上のmain関数
　ユーザはこの中身を書く

長所/短所

ライブラリに近いイメージだが、ライブラリ化はしなくてよい
新規プロジェクトの追加は、ユーザにはハードルが高い

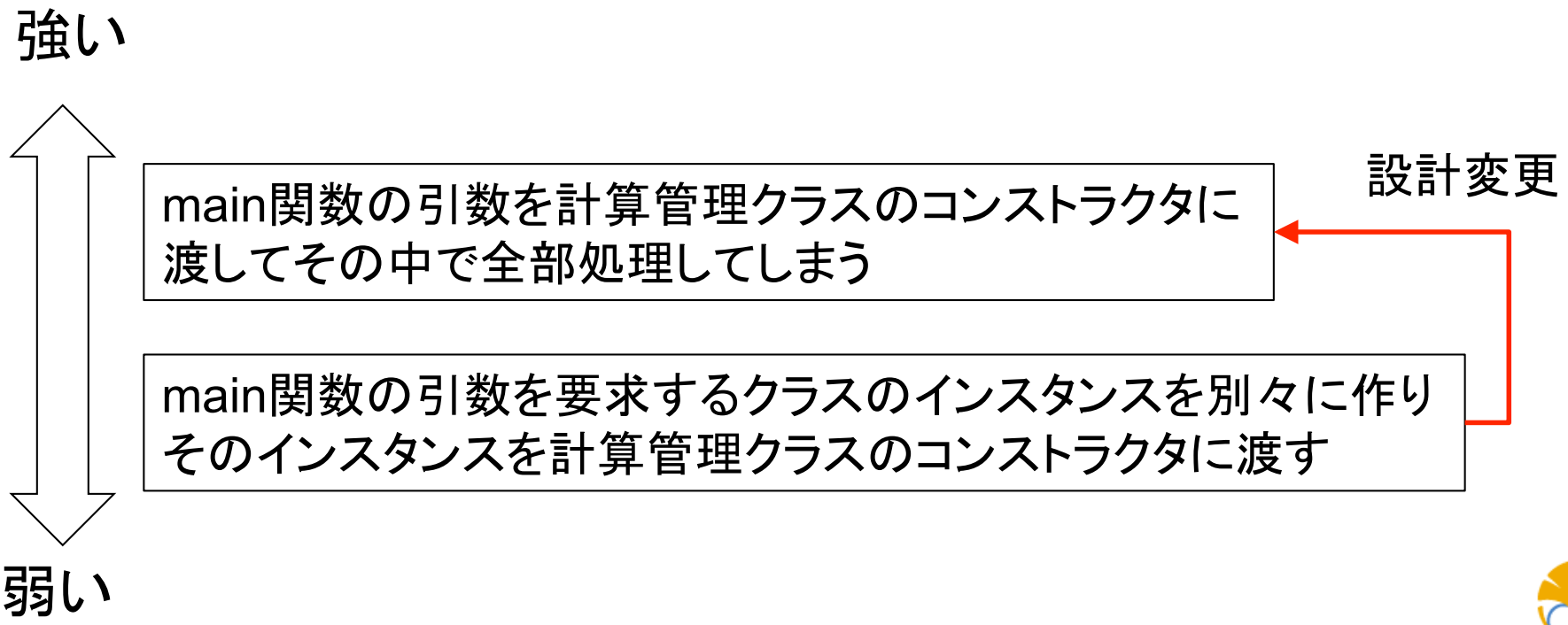
ユーザは僕だけだし・・・

main関数の引数の扱い (1/4)

main関数の引数を要求するクラスが、少なくとも2つある
- MPI_Initは引数としてargc, argvを要求
- パラメータクラス: ファイル名の取得にargvが必要

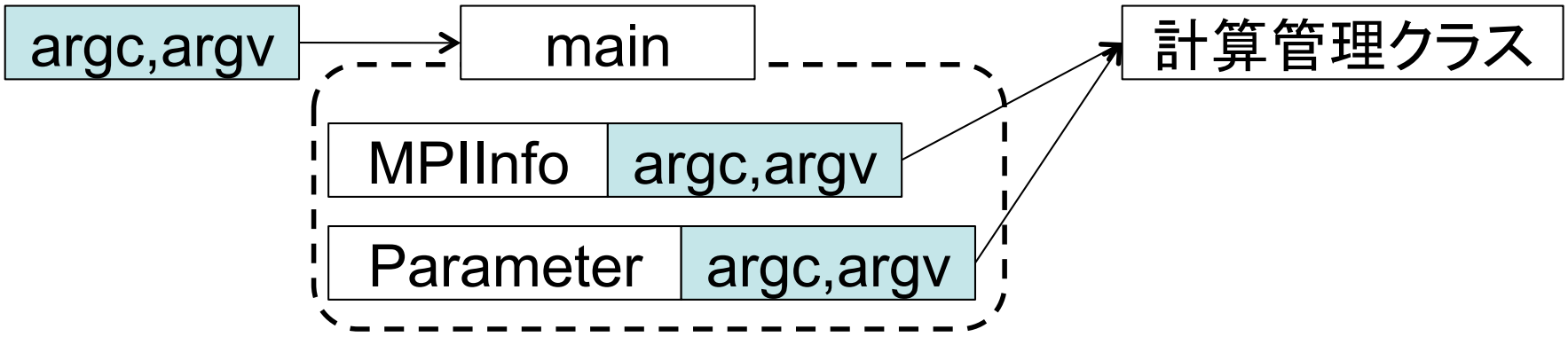
誰がどうやって受け取るべきか？

プログラムとの結合



main関数の引数の扱い (2/4)

案1: argc, argvをMPI管理クラス(MPIInfo)とパラメータ管理クラス(Parameter)それぞれのコンストラクタに渡し、それらのポインタを管理クラスのコンストラクタに渡す。

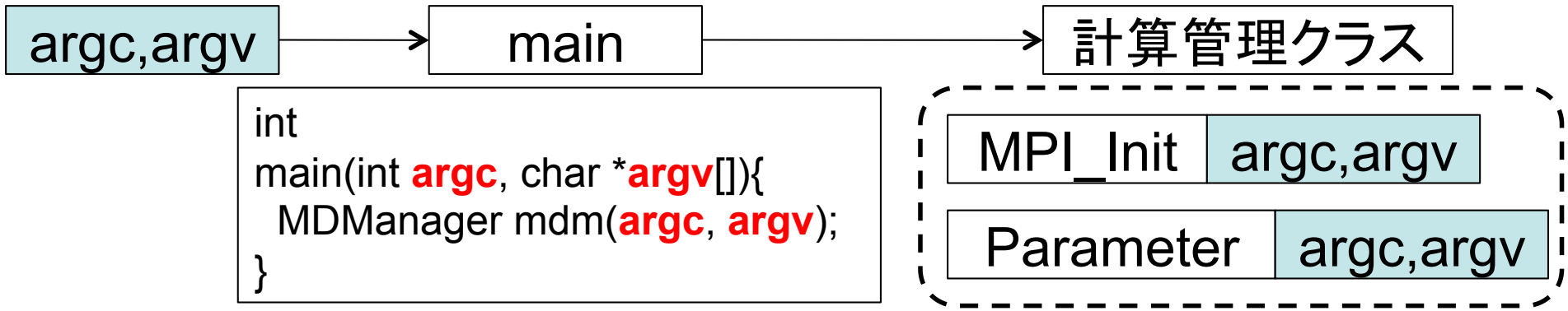


MPIInfo, Parameterのインスタンスを作成
管理クラスのコンストラクタに渡す

flat-MPI版コードではこちらを採用
設計的にはこれがまっとうな気もする。

main関数の引数の扱い (3/4)

案2: 計算管理クラスにargc, argvを渡し、コンストラクタで
MPI_Initの処理やParameterのインスタンスを作る



main関数は何もしない。渡されたargc, argvをそのまま
計算管理クラスのコンストラクタにスルー

ハイブリッド版コードではこちらを採用
プログラム設計的にはあまりよろしくない

main関数の引数の扱い (4/4)

Q. なぜ全て計算管理クラスに詰め込んだのですか？
分けたほうが設計がきれいだと思いますが？

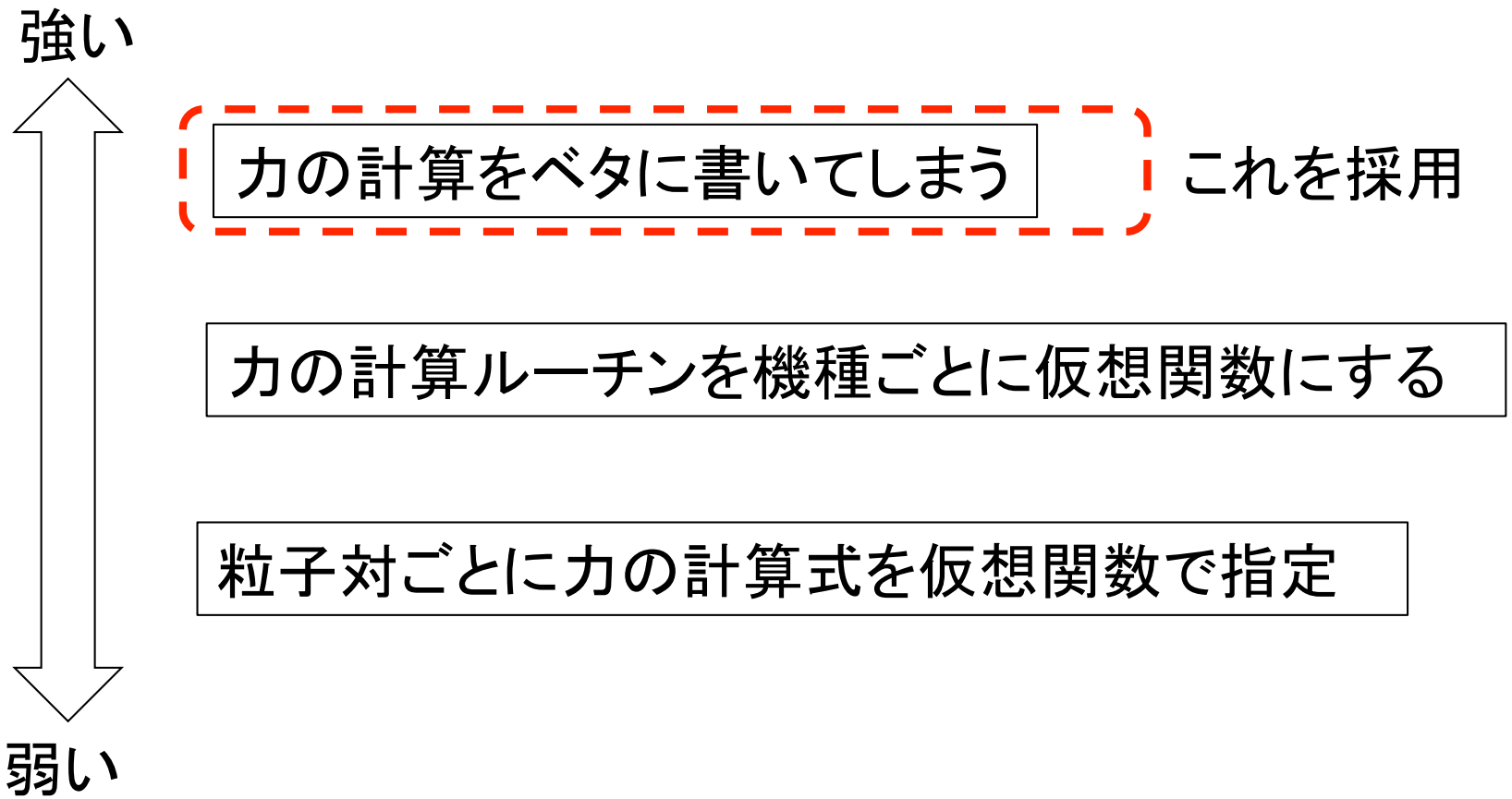
A. 分けるご利益があまりないと考えたから

その他雑多な言い訳
・MPIInfo、Parameterクラスのインスタンスは、どちらもMDManagerのメンバになっており、ライフタイムを共有している。MDManagerとライフタイムを共有するクラスのインスタンスを外で作って渡す、というのがどうにも気持ち悪かった。
・プロセスの化身であるMDManagerが、自分のランクを自分で知らない、というのが気持ち悪い気がした。「プロセスの化身」は誰か？MDManagerか？MPIInfoか？
・main関数はなるべく簡素化したい(これは単に趣味)。

力の計算ルーチン (1/2)

- 力の計算はなるべく高速に実行したい
- アーキテクチャごとに異なる最適化が必要
- 様々な種類の相互作用に対応したい

プログラムとの結合



力の計算ルーチン (2/2)

相互作用計算を仮想関数化

粒子の種類によって力の計算方法を変えたい
設計としては、力の計算を仮想関数にするのが一番きれい
→ 死ぬほど遅くなる

力の計算ルーチンを仮想関数化

せめてアーキテクチャ毎の力の計算を仮想関数にする？
ForceCalcクラスからForceCalcforSPARC64を派生して・・・
→ やっぱり遅くなる

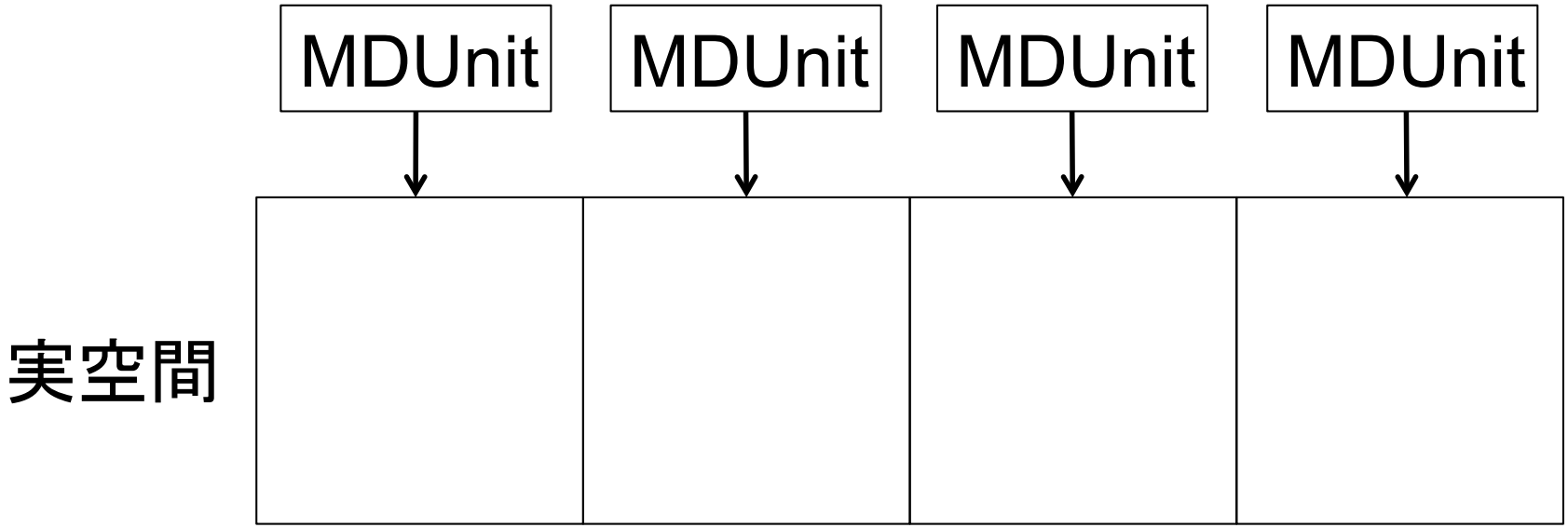
コンパイル時にどの関数が呼ばれるか決まらないと、プロシージャ間最適化(Interprocedural optimization, IPO)が効かなくなることがあるため

結局 #ifdef等でベタに指定する形に・・・

並列化と設計

通信の設計 (1/4)

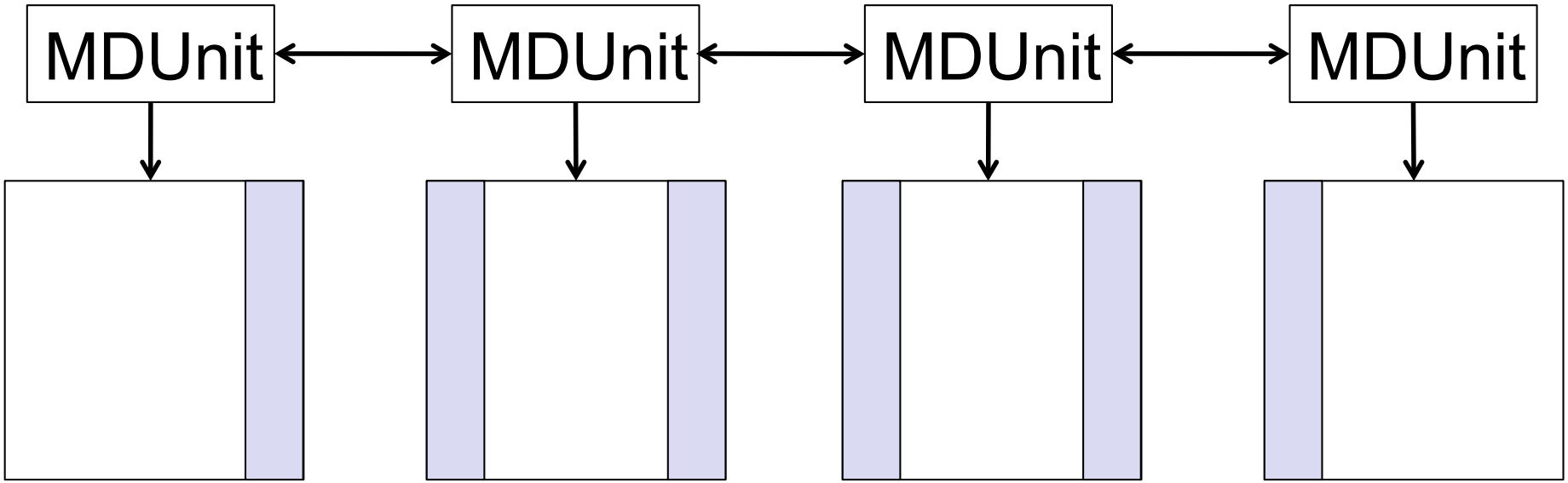
簡単のため単純領域分割、flat-MPIのみ考える
領域更新を担当するクラスを作るのが自然
→ ここではMDUnitと名付ける



分割された領域それぞれをMDUnitのインスタンスが管理
→　通信まわりをどう設計すべきか？

通信の設計 (2/4)

案1: MDUnit同士が行う (強結合)

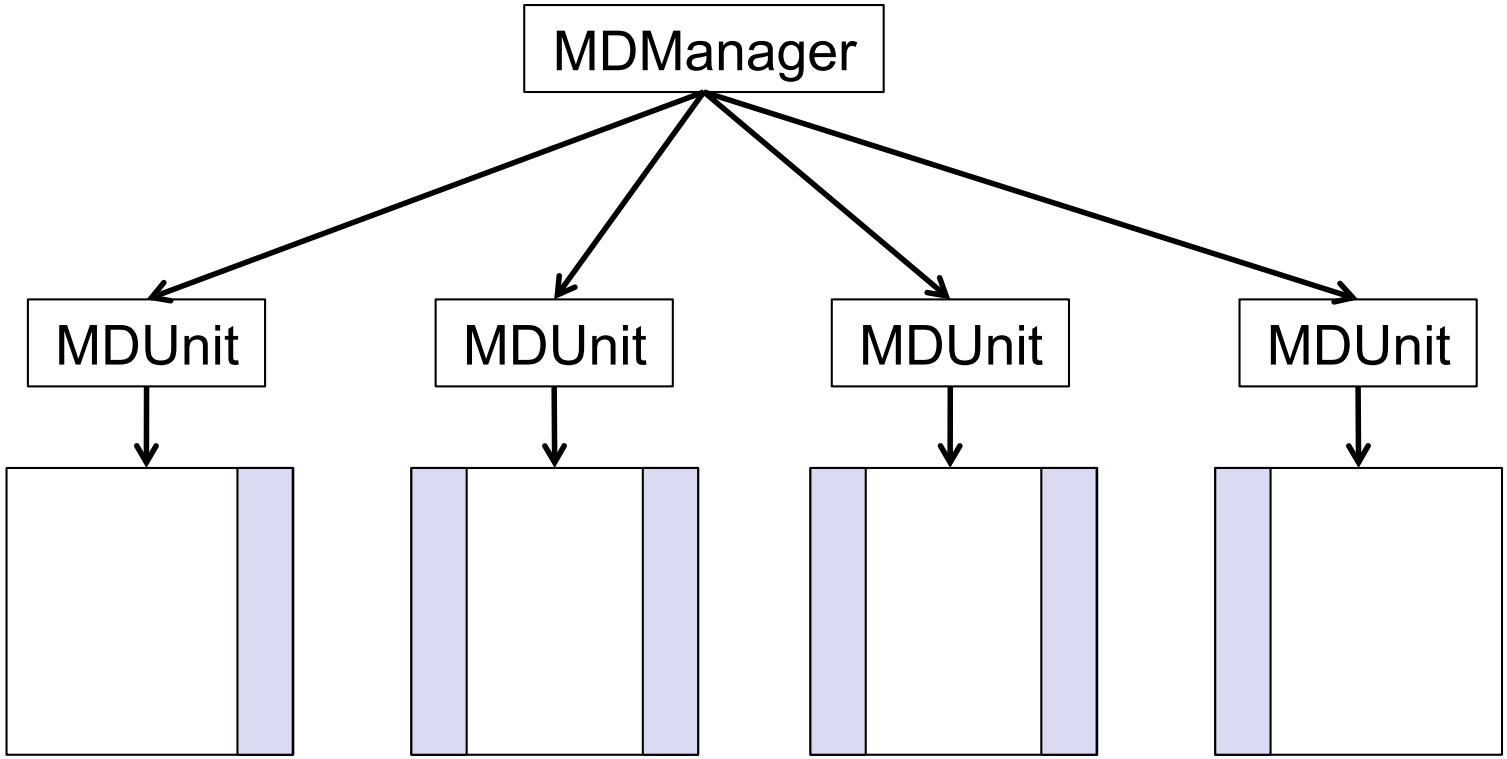


- 「隣の領域に誰がいるか」をMDUnitが自分で知っている必要がある
- 「領域更新」という局所的な役割と「通信」という大局的な役割の同居がとても気持ち悪い

→ flat-MPI版では案1を採用

通信の設計 (3/4)

案2: MDUnitを管理するMDManagerクラスを作る



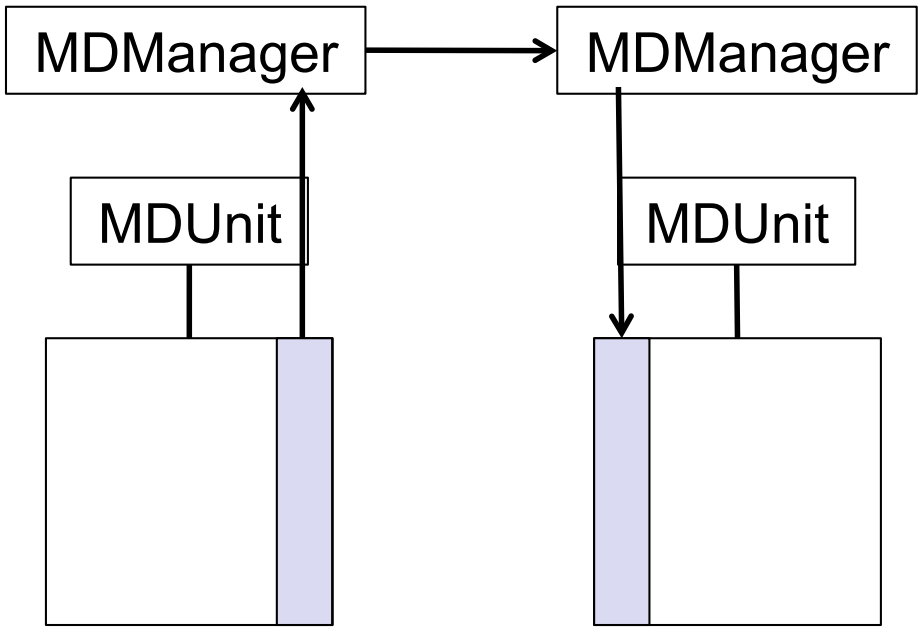
- MDUnitは自分が全体のどこに存在するか知らない
- 通信は全てMDManagerを通して行う
- 局所的役割と大局的役割の分離

→ ハイブリッド版では案2を採用

通信の設計 (4/4)

MPI通信とスコープ

通信はMDManagerを通してのみ行いたい



しかし実際には、ソースのどこからでもどこへでもMPI通信できる
→ MPIには本質的に「スコープ」が存在しない

MDUnitに隣接する領域のランクを教えないことで
擬似的に「スコープ」を導入したつもりだが・・・

プロセス配置管理 (1/2)

MPIでは、ノードをまたぐ通信量をなるべく減らすようにプロセスを配置する

1ノード4プロセス、4ノード計算のプロセス配置例

0	1	4	5
2	3	6	7
8	9	11	12
10	11	13	14

ハイブリッドだとさらにややこしくなる。
→ どの領域に誰がいるかの「地図」の管理が必要

プロセス配置管理 (2/2)

案1: MPIInfoクラスを作って、そこで地図を管理 (疎結合)
　　通信するクラスがMPIInfoクラスのインスタンスを持つ

案2: MDManagerクラスが地図を直接管理してしまう (強結合)

flat-MPIコードの開発では案1を採用したが、
ハイブリッドコードの開発では案2を採用

ハイブリッドコードでは、MDManagerのコンストラクタ、デストラクタで
MPI_Init/Finalizeを呼び出すなど、MPIに関する情報を分離していない

Flat-MPI版ではなるべく分離したが、分離するメリットがあまり感じられな
かったため

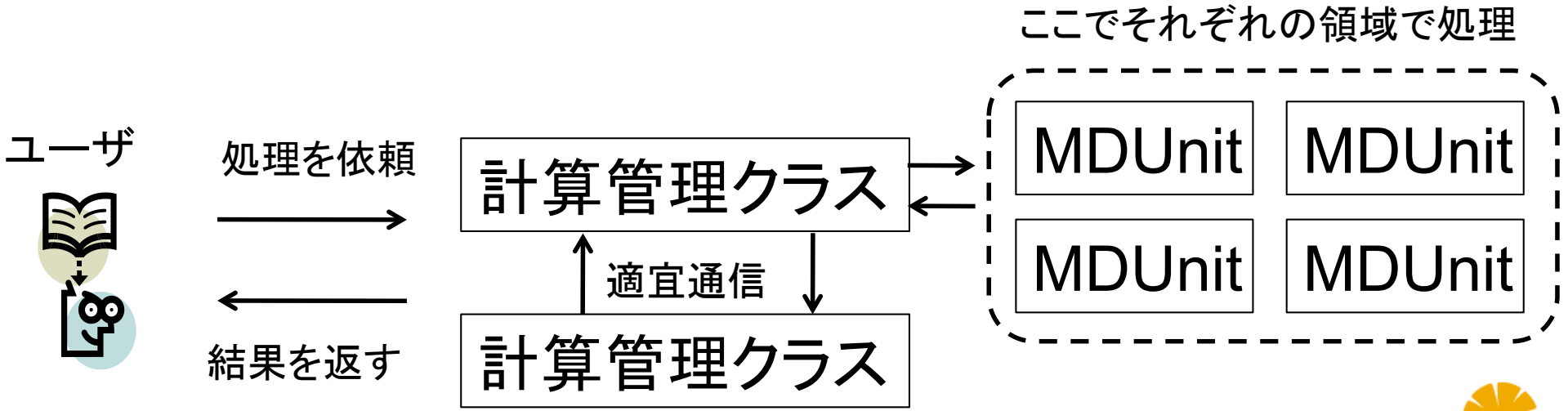
並列構造の隠蔽 (1/3)

全体処理の扱い

領域分割による並列化により、変数データが分散している
初期化や観測ルーチンなどの全体処理をどう書くべきか？

設計思想

ユーザから見えるのは計算管理クラス(MDManager)のみ
計算クラス(MDUnit)が管理するデータを触りたい
なるべく並列化構造を意識したくない



並列構造の隠蔽 (2/3)

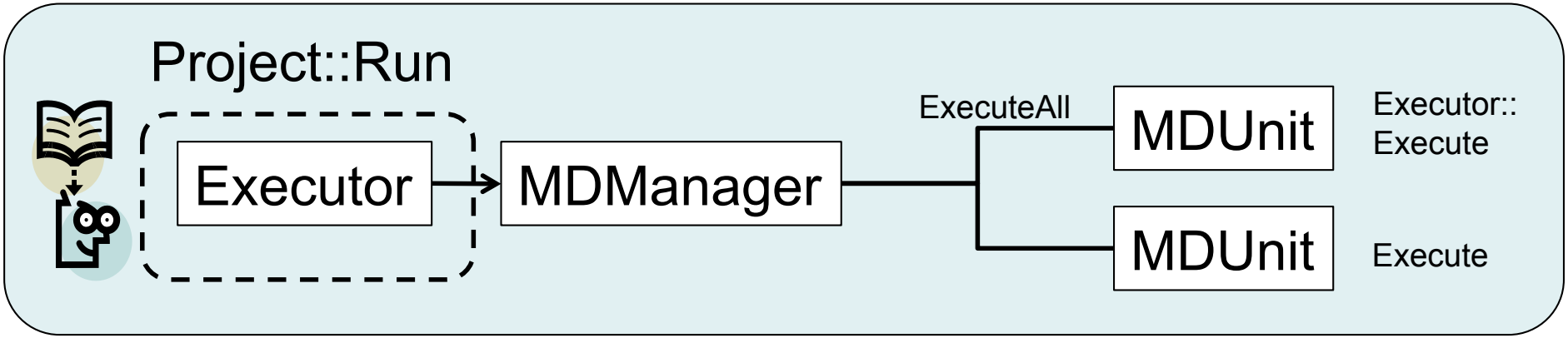
実装

MDManagerはExecuteAllメソッドを持つ
MDManager::ExecuteAllはExecutorクラスのインスタンスを
支配下のMDUnit::Executeに渡すだけ

```
void
MDManager::ExecuteAll(Executor *ex) {
  #pragma omp parallel for schedule(static)
  for (int i = 0; i < num_threads; i++) {
    mdv[i]->Execute(ex);
  }
}
```

ユーザはExecutorクラスを派生したクラスを作り、
その中のExecuteメソッドをオーバーライドする

並列構造の隠蔽 (3/3)



- ユーザが定義したExecutorのインスタンスはMDManagerを通じてMDUnitに配布される
- Executor::ExecuteにはMDUnitのインスタンスが渡されるので、それを通じて好き放題できる
- MDUnitは系のサイズや時間刻み等を知らないが、Executorクラスのインスタンスを作るのはProject::Runの中なので、系のサイズなどの情報を使える(コンストラクタで渡せる)

要するにコールバック関数

形の上では並列構造は隠蔽できたが、結局並列構造を理解していないと中身を組めない・・・

ソースを公開するということ

ソースを公開するということ (1/5)

ソースを公開して良いことはあるだろうか？

あまりフィードバックの価値のない文句は言われる
バグ報告などはほとんど無い
→ 公開してもあまりいいことはない

そもそもデファクトスタンダードを目指していない
ユーザを増やそうともしていない

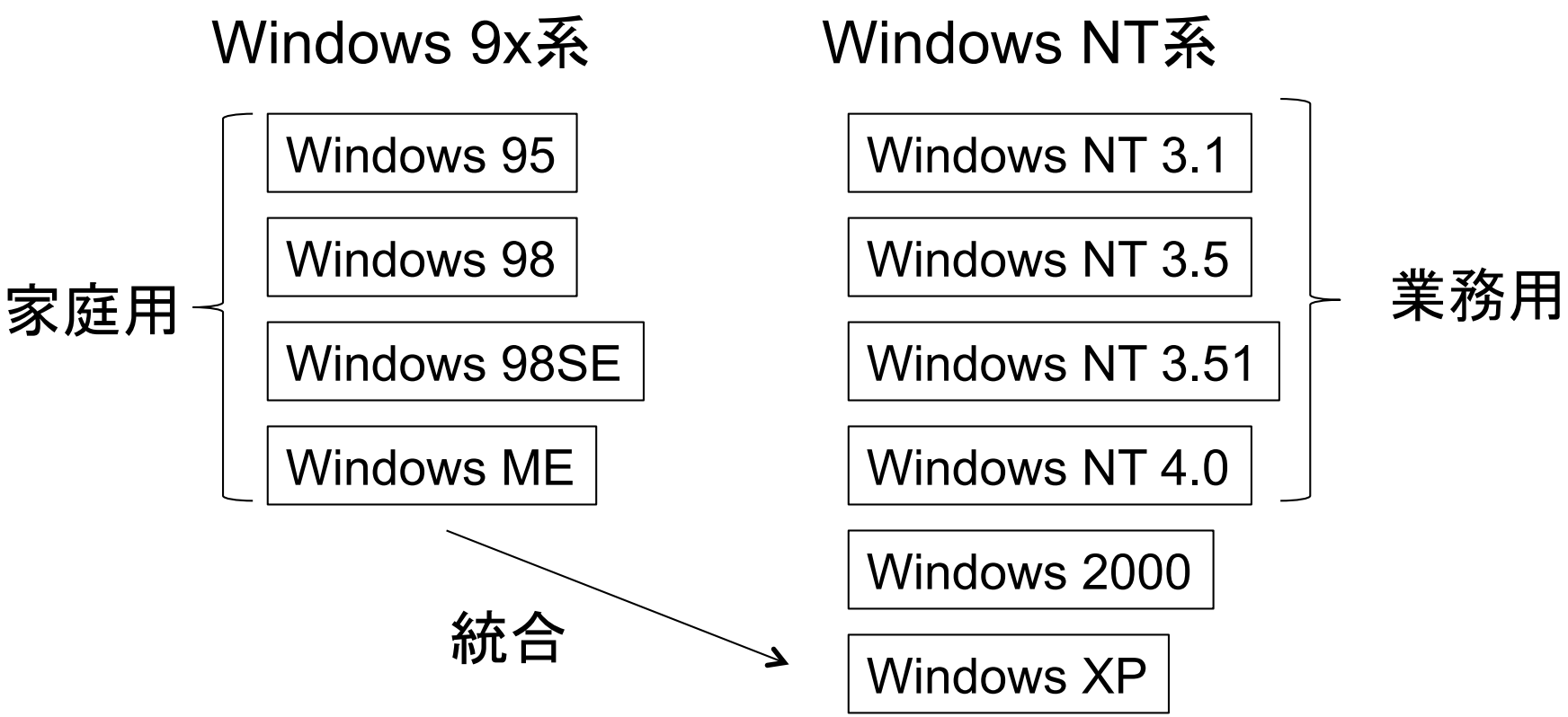
では、なぜそれでもソースを公開するのか

ソースを公開するということ (2/5)

原題：SHOWSTOPPER

「伝説のプログラマー」D. カトラーがWindows NTを作るお話

ソースを公開するということ (3/5)



9x系とNT系は全く異なるOS
その統合期には多くの混乱があった

ソースを公開するということ (4/5)

Windows プログラミング

WindowsにおけるプログラミングはAPIと呼ばれるものを使う

> API (Application Programming Interface)
> OSが提供する機能(ファイルアクセスやマウスイベント取得)の利用

異なるOSでも、同じAPIを使えば同じ動作が保証される・・・はず

9x系からNT系へ

当時Windows 98で開発していたソフトが、Windows 2000で動作しないという報告を受け取る
実機が無いためどうにもならなかったが、同様な機能を持つソフトがソースを公開しており、それを参考に修正→動作するように

> それまでソースは公開していなかったが、この一件以来公開するように

ソースを公開するということ (5/5)

ソースは公開すべきだろうか?

メリット/デメリットで言えば、個人的には公開する意味は薄いと思う

しかし我々は、インターネットの海に浮かぶ様々な栄養を日常的に摂取している。であれば、自らも何かフィードバックすべきでは？

MDACPのソースを公開する理由

- 短距離MDのデファクトスタンダードを目指しているわけではない
- 多くのユーザを獲得するためでもない
- バグ報告などを期待しているわけでもない

「京」フルノードで動作が確認されたリファレンス実装として、これから並列コードを組む人が参考にするための資料として公開

まとめ

- 設計に正解はない
  - 特に性能と拡張性・保守性の両立は難しい
  - とりあえず書いてみる (考えるより組む方が早い)
  - ただし、設計が良くないと思ったら書き直すこと

- 並列プログラムは面倒くさい
  - 並列化構造を隠蔽しようと試みた
  - 複雑なことをやろうとすると、並列化構造の理解必須

- ソースの公開について
  - 無理にやらなくても良いと思う
  - 文句を言うより、どうせなら言われる側になろう