e-science@京大：高生産スクリプト言語

中島 浩（代理：平石 拓）
京都大学学術情報メディアセンター

目次

背景

- 科学技術計算によるシミュレーション
  - 創薬，車体設計，天気予報
- パラメータスイープ，最適パラメータ探索

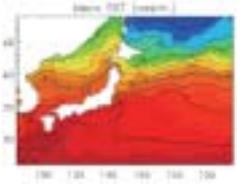
→ 同一のプログラムを多数の入力に対し繰り返し適用
=*PDCAサイクル*
  - Plan: 入力を生成
  - Do: ジョブ投入（タスク並列）
  - Check: 結果を処理
  - Action: 次の計算を検討

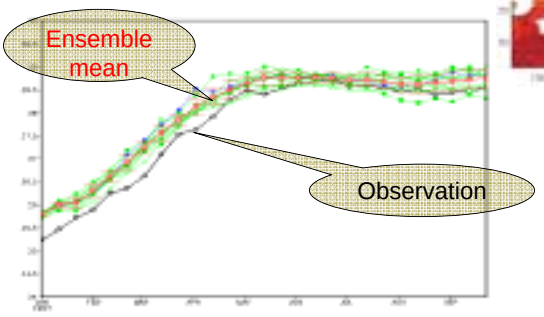
の繰り返し

- 自動化すべき
  - ワークフロー，出力から次の入力を生成

例：アンサンブルシミュレーション

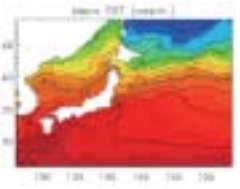
- 異なる多数の初期パラメータに対して同一の予測計算（1アンサンブル＝1ジョブ）
- 最後に結果をまとめる

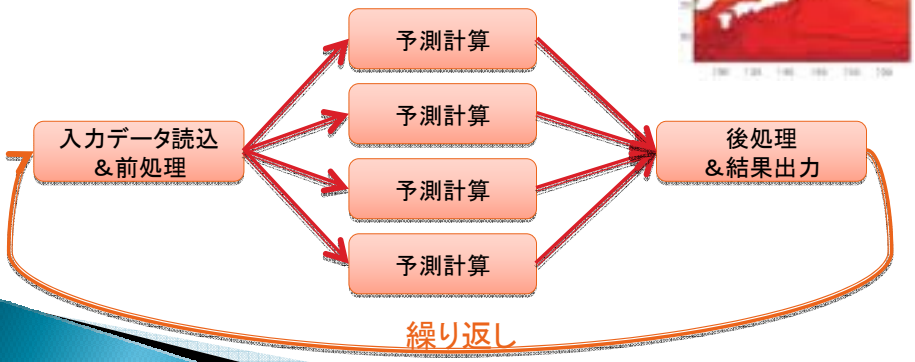




例：アンサンブルシミュレーション

- 異なる多数の初期パラメータに対して同一の予測計算（1アンサンブル＝1ジョブ）
- 最後に結果をまとめる





現状と目標

- 手動（非自動化）
- ジョブの実装言語（FortranやC）で自動化
  - 「計算科学者」に親しみのある言語
  - 複雑なフローやタスク並列計算の記述には適さない
- 汎用のスクリプト言語
  - シェルスクリプト，Perl，Ruby，．．．
  - 汎用性は高い
  - まだ手軽とは言い難い（特に「計算科学」研究者にとって）
- 専用スクリプト言語 (DSL)
  - ワークフローの自動化に特化した言語機能・ライブラリを提供
  - 手軽な記述と汎用性を両立
- ワークフローツール
  - 手軽な記述（GUI）
  - 複雑なフローの記述は困難

設計目標

- タスク並列スクリプト言語に求める要件
  - 簡便性
    - ユーザ＝計算科学研究者（≠計算機科学研究者）
    - スクリプト記述が手軽
      - 複雑な探索アルゴリズムなどを記述しなくてよい
      - ジョブの出力データから次の入力を生成も楽に（正規表現より楽に）
  - 汎用性
    - 複雑なフローにも対応（二分探索など）
    - 既存のプログラムから容易に移行可能
  - 可搬性
    - 様々な環境で動作する（特定のジョブスケジューラに依存しない など）

既存のタスク並列言語 (1)

- Megascript [大塚, …, 中島 03]
  - Rubyベース：オブジェクト指向
  - 「タスク」と「ストリーム」
    - 各タスクはオブジェクトとして定義
    - タスク間を「ストリーム」で接続することにより，データの授受を実現
  - プログラムに対する制約：データの授受は必ず「ストリーム」を利用
    - 既存プログラムの改変の手間
    - このモデルに合致しないアプリケーションも

```
class Task_A < Task
  def initialize(*arg)
    @exefile="./task"
    @parameter = arg
  end
  def behavior
    n = @parameter
    FOR n
      x=getValue.of(
          main(bt=1))
      compute(x)
    end
  end
end
```

既存のタスク並列言語 (2)

- PJO (Parametric Job Organizer) [富士通]
  - Perl（またはsh）ベース
  - ジョブ投入部（topブロック）＋完了後処理（whenブロック）
  - 各ジョブにIDを付与
    - パターンにマッチしたジョブに対応するwhenブロックが実行される
    - whenから次のジョブ投入も可能
  - ジョブの依存関係も考慮したジョブの再投入も可能

```
top {{
  foreach $param (1 .. 5) {
    do_job {{ param=$param }} {{
      $rc = system("./a.out");
      if (0 != $rc)
      PJO::abort "job executes failed.¥n";
    }}
  }
}}
when {{ {{ param=$x }} }} {{
  print "job param=$x finished.¥n";
}}
```

既存のタスク並列言語 (2)

- PJO (Parametric Job Organizer) [富士通]

問題点

- ジョブ投入とフローの処理が別のPerlプロセスで処理される
  - 全ワークフローが同一環境上で処理されない

→複雑なフローの記述が困難

- ジョブ間タスク授受のサポートは特になし

```
top {{
  foreach $param (1 .. 5) {
    do_job {{ param=$param }} {{
      $rc = system("./a.out");
      if (0 != $rc)
      PJO::abort "job executes failed.¥n";
    }}
  }
}}
when {{ {{ param=$x }} }} {{
  print "job param=$x finished.¥n";
}}
```

開発方針(1)

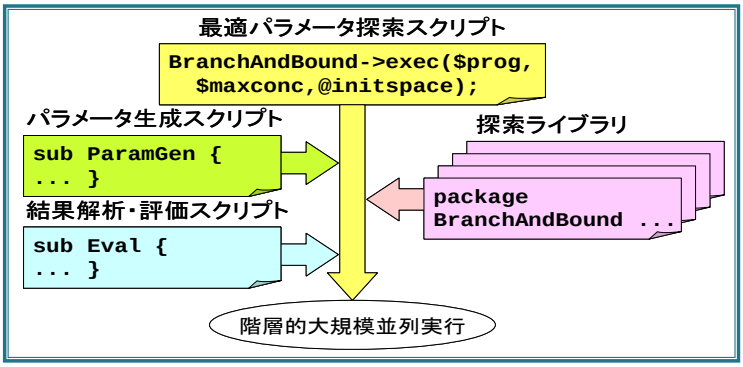
- MegascriptやPJOの資産(思想・実装)は継承
  - Megascript
    - ジョブをオブジェクトとして抽象化
    - ジョブ間データ授受の抽象化
  - PJO
    - 指定したジョブに対して完了後に指定した処理を非同期実行
    - スクリプト言語のバックエンド実装
      - ジョブ状態の監視機構
      - ジョブ再投入機構
      - . . .

      → 改良・可搬化(ジョブスケジューラ非依存に)

開発方針(2)

- いろいろな種類の自動化を容易に実現
  →機能のモジュール化
  - 「探索アルゴリズム」「同時投入ジョブ数制限」「データ加工」などの典型的な機能を モジュール(ライブラリ) として提供
    - 処理の雛形はモジュール内で定義
    - アプリケーション依存部分(実行ファイル名, ジョブ数, . . .)はユーザスクリプトで実装
- 典型的な処理 → モジュール取り込み+最低限の記述
  → 簡便性の達成
- 複雑な処理 → モジュール改造 or 開発
  → 汎用性も失わない

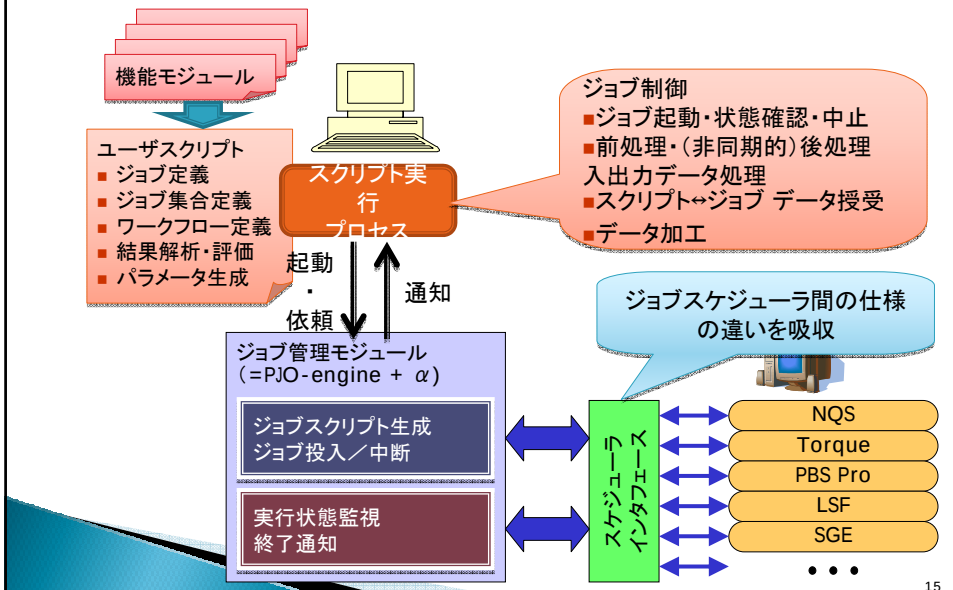
機能のモジュール化イメージ



目次

- はじめに
  - 研究の背景，目標
  - 既存のスクリプト言語
  - 開発方針
- スクリプト言語システムの設計
  - システムの全体構成
  - 言語のプロトタイプ設計
- 今後の計画・まとめ

全体構成



スクリプト仕様のプロトタイプ設計

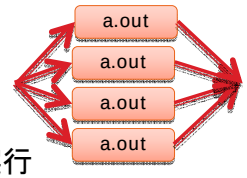
- Perl ＋ モジュール群 ＋ 最低限の言語拡張
  - 習得コストを節約
    (本発表のプロトタイプでは大幅に拡張)
- オブジェクト指向
  - ジョブ=オブジェクト
    - オブジェクトのクラスは"Job" built-inクラスを拡張(直接／間接)して定義
  - 機能モジュール=(抽象)クラス
    - "Job" built-inクラスを拡張(直接／間接)して定義
    - ユーザは，必要な機能に対応するクラスを多重継承(mixin)
  - メソッド定義(モジュール／ユーザ記述)で
    - 各ジョブ投入前・完了後に行うべき処理
    - 全体の実行フロー

    を実装

PJOのwhenブロック相当＋α

PJOのtopブロック相当

例題



- アンサンブル予測計算の基本部分
  - “a.out”を5000回，それぞれ異なる入力で実行
  - 各ジョブの間に依存関係はない（タスク並列可能）
  - 同時に投入するジョブの数は10個に制限する.
  - *i*番目のジョブの入力はファイル “input*i*”に予め用意
    結果は “output*i*”に出力
  - 入出力ファイル名はa.outのコマンドライン引数で指定
  - 各ジョブおよび全てのジョブ終了後にメッセージを表示
- 以下2つの「機能」をモジュールとして提供
  - *n*個のジョブのタスク並列実行
  - 同時投入ジョブ数を*m*個に制限

ユーザ記述スクリプト

モジュール使用宣言（親クラス指定）

```
class Example : Restrict Parallel {
  $Job_exec = "a.out";
  $Restrict_max = 10;
  $Par_njob = 5000;
  before () {
    @{$self->{Job_args}} = ("input$Par_nsubmit", "output$Par_nsubmit")
  }
  after () {
    print "Job @{$self->{Job_args}} finished.";
  }
  after_all () { print "All jobs finished." }
}

Example::main();
```

実行ファイル名

同時投入ジョブ数

全ジョブ数

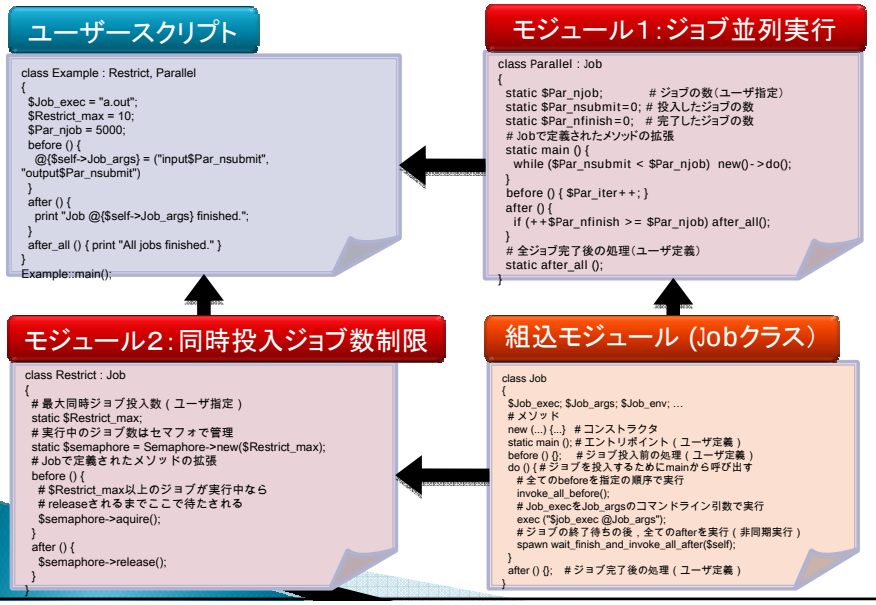
コマンドライン引数のセット

各ジョブ完了後のメッセージ

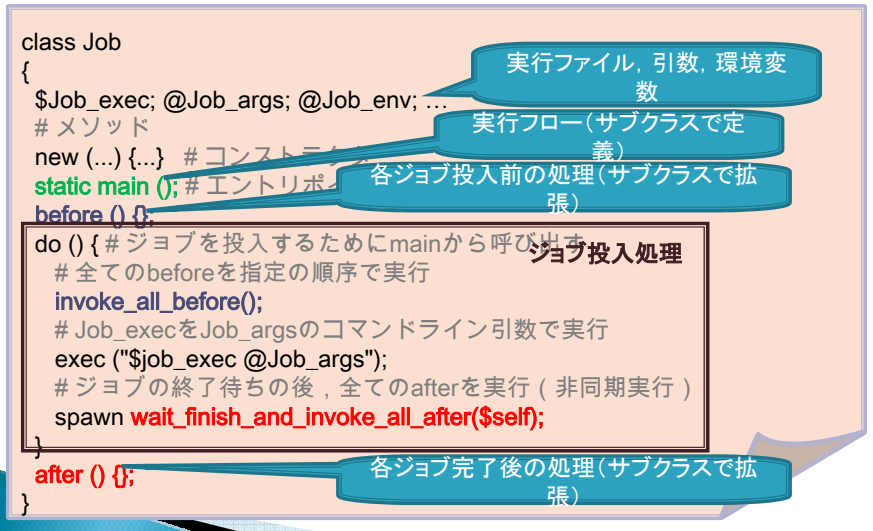
全ジョブ完了後のメッセージ

実行開始

クラス継承関係



Jobクラス（組み込みクラス）



モジュール1：ジョブ並列実行

```
class Parallel : Job
{
  static $Par_njob;
  static $Par_nsubmit=0;
  static $Par_nfinish=0;
  # Jobで定義されたメソッドの拡張
  static main () {
    while ($Par_nsubmit < $Par_njob)  new()->do();
  }
  before () { $Par_iter++; }
  after () {
    if (++$Par_nfinish >= $Par_njob) after_all();
  }
  # 全ジョブ完了後の処理（ユーザ定義）
  static after_all ();
}
```

全ジョブ数＝n（サブクラスで指定）

投入済みジョブ数の管理

完了済みジョブ数の管理

Job::main（ジョブ実行フロー）の実装

ジョブ投入前・完了後の処理の拡張：カウンタインクリメント

メソッド追加（サブクラスで定義）
「全ジョブ完了後何かする」機能を提供

モジュール2：同時投入ジョブ数制限

```
class Restrict : Job
{
  # 最大同時ジョブ投入数（ユーザ指定）
  static $Restrict_max;
  # 実行中のジョブ数はセマフォで管理
  static $semaphore = Semaphore->new($Restrict_max);
  # Jobで定義されたメソッドの拡張
  before () {
    # $Restrict_max以上のジョブが実行中なら
    # releaseされるまでここで待たされる
    $semaphore->aquire();
  }
  after () {
    $semaphore->release();
  }
}
```

制限ジョブ数＝m（サブクラスで指定）

実行中ジョブ数を管理するためのセマフォ

ジョブ投入前にセマフォ獲得

ジョブ完了後にセマフォ解放

ユーザ記述スクリプト(再)

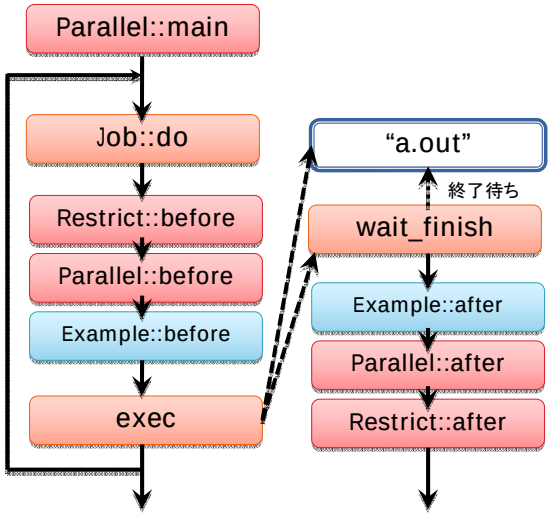
```
class Example : Restrict, Parallel
{
  $Job_exec = "a.out";
  $Restrict_max = 10;
  $Par_njob = 5000;
before () {
  @{$self->{Job_args}} = ("input$Par_nsubmit", "output$Par_nsubmit")
}
after () {
  print "Job @{$self->{Job_args}} finished.";
}
after_all () { print "All jobs finished." }
}

Example::main();
```

Job::Job_execの値設定
Restrict::Restrict_maxの値設定
Parallel::Parallel_njobの値設定
Job::beforeの拡張:ジョブ投入前にコマンドライン引数指定
Job::afterの拡張:各完了後のメッセージ
Parallel::after_allの実装
Parallelで定義されたmainを実行

スクリプトの動作

- mainから実行開始
  - 各ジョブ実行前
    - 全てのbeforeメソッドを「妥当」な順で実行
  - 各ジョブ完了後
    - 全てのafterメソッドを逆順に実行(非同期)

(method combination)



議論

- ユーザ記述の簡便性
  - 利用するモジュールの宣言
  - インポートしたモジュールが要請するパラメータおよび
    アプリケーション依存の処理

  のみ定義
- 汎用性
  - モジュールが十分用意されていれば，多重継承で組み合わせ
    様々な種類のワークフローに対応可能
    （たとえば，Parallel を SmartSearch に置き換え）
    - ただし，安全に組み合わせ可能とは限らない
  - 既存モジュールにない機能は，既存のものを拡張／改造すること
    で対応できる.

目次

- はじめに
  - 研究の背景，目標
  - 既存のスクリプト言語
  - 開発方針
- スクリプト言語システムの設計
  - システムの全体構成
  - 言語のプロトタイプ設計
- 今後の検討課題・まとめ

今後の検討課題(1)

- 言語設計
  - この仕様で他の種類のワークフローにも対応できるか？
    - 事例を積み重ねつつ仕様を改善していく必要
    - 「任意」のパターンへの対応を目指しているわけではない
  - ユーザ向けの記述はできるだけ単純に
    - エンドユーザには「オブジェクト指向」を見せたくない
    - Perlの言語拡張はなるべくしない(ライブラリ改変で対応)

記述の簡略化の例

```
$myjobset  (parallel, restrict) {
  Jobset_exec="./fib";
  Jobset_args="%d";
  Jobset_after = ' {print "Job finished.¥n"} ';
  Par_njob=5000;
  Par_after_all = '{ print "All jobs finished.¥n" }';
  Restrict_max = 10;
}
$myjobset->start();
```

今後の検討課題(2)

- タスク間のデータ授受のサポート
  - 標準入力・標準出力等の扱い(抽象化)
  - 入出力ファイルの扱い(staging/destagingなど)
  - データの加工
    - 前のジョブの結果から次のジョブの入力を生成
    - 巨大なデータの圧縮，インデックス付与

  → これらも(Jobクラスの継承とは別に)
  モジュールとして提供

今後の検討課題(3)

- チェックポインティング／リトライ
  - 「継続」の管理
    - スクリプトの実行ポイント
    - 投入したジョブの実行状態(実行中／終了)
  - ジョブの依存関係を記述できる仕組み
  - PJOの資産も活用
- スクリプト言語レベルのデバッグ機能
  - プログラムは簡単でもジョブの実行時間・必要資源は膨大

  → 間違ったスクリプト実行による損失は膨大
  - 簡単なプログラムでも間違いは生じうる

  → ジョブ実行を省略・簡略化してスクリプトだけをデバッグ

まとめ

- 大規模シミュレーションにおけるPDCAサイクル，タスク並列計算を容易に自動化するためのスクリプト言語システムの開発
  - オブジェクト指向設計
    - ジョブ＝オブジェクト
    - 複雑な処理 ← モジュール＝抽象クラス として提供
  - 簡便性と汎用性を両立
  - 特定の環境・プログラミングスタイルに依存しない
- 開発方針
  - 実際の応用プログラムを使って簡便化を実現
    → 事例を積み重ねながら適用範囲を広げていく

スケジュール

H20(2nd half)	H21	H22	H23
ベース言語選定 ■ 第1近似＝perl ← PJO との親和性 ■ Python, Ruby も視野内 プロトタイプ基本機能実装 ■ ジョブスケジューラインタフェースの可搬化 ■ スクリプト／ジョブ間のデータ授受メカニズム再設計 ■ 例題ベース（海洋物理アンサンブルシミュレーション，etc.）でのユーザスクリプト／基本EPライブラリのAPI設計	プロトタイプ開発 ■ ジョブ制御機能／インタフェース抽象化機能 ● フル機能実装 ● 超簡易 end-user インタフェース機能実装 ■ 探索ライブラリ（例題ベース） ● 探索機能 ● パラメータ生成機能 ● 結果解析・評価機能 ■ 拡張機能設計 ● スクリプトのバッチ実行 ● スクリプトレベルデバッグ	評価 改良 最終仕様決定	最終版 システム開発 総合評価