シームレスな高生産並列スクリプト言語の実現に向けて

平石　拓　岩下武史　中島　浩

京都大学学術情報メディアセンター

車体の設計や創薬等の計算科学における大規模シミュレーションを行う際，パラメータスイープや最適パラメータ探索のために，同一のジョブをパラメータを変えつつ，大量に繰り返し，あるいは並列に実行することが多い．このような処理を自動化する際，既存のワークフローツールを用いると簡便ではあるが汎用性に欠け，Perl 等の既存のスクリプト言語で記述するのは計算科学の研究者には一般に敷居が高い．そこで我々は，既存のスクリプト言語をベースとしつつ，上記の自動化を簡便に記述するための機能を追加したジョブ並列スクリプト言語を開発している．複雑な探索アルゴリズムや同時投入ジョブ数の制限等の機能はオブジェクト指向の抽象クラスとしてモジュール化する．ユーザは提供済みのモジュールをそのまま使うことも，必要に応じて改造したり新たに開発することもできるため，汎用性と簡便性が両立できる．本発表では，バックエンドとなるジョブスケジューラを含むシステムの全体像と言語のプロトタイプを示す．

Towards Seamless and Highly-Productive Parallel Script Language

Tasuku Hiraishi, Takeshi Iwashita and Hiroshi Nakashima

Academic Center for Computing and Media Studies, Kyoto University

Computational scientists often perform large scale simulations in their research or development such as car body design and drug discovery. In such simulations, they often execute plenty of sequential and/or parallel jobs with different parameters for parameter sweep or optimal parameter search. Though they can use workflow tools in order to automate such tasks, it is difficult to describe some kind of workflows with them. They can also use script languages such as Perl, but it is hard for typical computational scientists to program in such a language. Therefore, we are developing a new language based on an existing script language that has additional features to enable us to describe such automation easily. We realize both flexibility and easiness to use by modularizing features, such as complicated search algorithms and limiting the number of simultaneously submitted jobs, as abstract classes of object oriented languages. Programmers can automate not only typical workflows easily by simply using provided modules but also more complicated workflows by modifying existing modules or developing new modules. This presentation shows an overview of our system, which includes a job scheduler as a backend, and a prototype of our script language.

1. はじめに

科学技術計算によるシミュレーションをスーパーコンピュータなどの環境で行う際，**図 1** のような *PDCA* **サイクル**の形態がとられることが多い．すなわち，プログラムへの入力データを用意し（plan），その入力を与えてジョブを投入し（do），その結果を確認し（check），それをもとに次に必要な試行を考え（action），必要なら新たな入力を用意してジョブの実行をやり直すということを繰り返すのである．パラメー

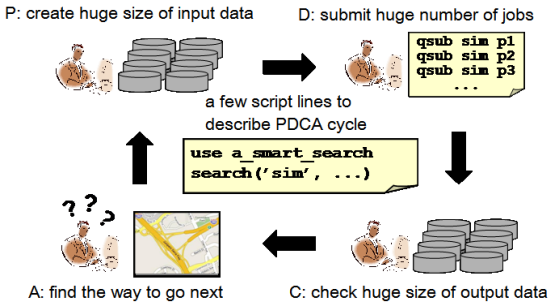

図 1　PDCA サイクル

タスイープや SA・DA などによる最適パラメータ探索は PDCA サイクルの典型的な例である．また，複数のジョブを同時に投入し，タスクの並列実行を行うことも多い．

このような作業を人手で行うのは手間がかかるため，なるべく自動化されることが望ましい．従来の自動化ツールとしては，まずワークフローツールが挙げられる．このようなツールを使うと手軽にフローを自動化することができるが，記述できる範囲はツールが想定するフローに限定され，ユーザが自動化したい作業を必ずしも記述できないという問題がある．一方，Perl や Ruby，シェルスクリプトなどのスクリプト言語のプログラムによって自動化を行うこともあるが，このような汎用言語によるプログラミングは手間がかかる．特に，計算科学のユーザはこのような言語に不慣れであることも多い．

そこで我々は，既存のスクリプト言語である Perl をベースとしつつ，より手軽に PDCA サイクルの記述ができるよう機能を拡張したタスク並列スクリプト言語，およびそのバックエンドとなるジョブ管理システムの開発を行っている．柔軟性と簡便性を両立するため，複雑な探索アルゴリズムや同時投入ジョブ数といった機能をモジュールとして提供する．これにより，典型的な作業はモジュールを取り込んで必要なパラメータ（実行ファイル名や投入ジョブ数など）を設定するだけで自動化が行え，さらに既存のモジュールを改造したり新たなモジュールを開発することでより複雑なフローにも対応できる．これらのモジュールは，オブジェクト指向言語の抽象クラスとして提供することを検討している．

このような言語とモジュールの関係は，LaTeX と TeX の関係に近い．LaTeX ユーザは，TeX 言語で記述されたスタイルファイルを usepackage 等で取り込むことで，複雑なプログラミングなしに組版を作成できる．また，既存のスタイルファイルを改造したり新たに作成したりすることによって，柔軟なカスタマイズも可能である．

本論文の構成は以下の通りである．まず 2 章で既存のタスク並列言語を紹介する．次に 3 章でジョブスケジューラ等を含むスクリプト処理システムの全体像を示し，本言語の位置付けを明確にする．その後，4 章でスクリプト言語の現状の仕様案を示し，5 章で今後の課題を述べ，最後に 6 章でまとめる．

なお，本システムは開発の初期段階であるため，言語の仕様等についてインフォーマルな説明が多く含まれる点，ご容赦いただきたい．

```
top {{
  foreach $param (1 .. 5) {
    do_job {{ param=$param }} {{
      # ジョブの実行を記述.
      # このブロックの処理のみ別プロセスで実行
      $rc = system("./a.out");
      if (0 != $rc)
        PJO::abort "job executes failed.\n";
    }}
  }
}}
# 各ジョブの終了を待ち合わせて処理を実行.
# 各ジョブ投入時の$param の値が$x で参照可能.
when {{ {{ param=$x }} }} {{
  print "job param=$x finished.\n";
}}
```

図 2 PJO スクリプトの例

2. 既存のタスク並列言語

2.1 MegaScript

MegaScript[3] は，Ruby をベースとしたタスク並列言語である．MegaScript では外部プログラムの実行を「タスク」として定義し，各タスクを「ストリーム」と呼ばれる論理的な通信路で接続することでタスク間のデータの受け渡しを実現する．Ruby ベースであるため，ワークフロー記述の自由度も高い．

しかし，タスク間のデータの受け渡しがストリームを通すものに限定される関係上，外部プログラムはデータの入出力を標準入出力を通して行うように実装しなければならないという制約がある．そのため，既存のプログラムをそのような要請を満たすために変更するのが困難な可能性もあるなど実用上の問題がある．

2.2 PJO

PJO[4] は富士通研究所により開発されたジョブ管理システムである．システムのフロントエンドとして Perl を拡張した独自のタスク並列スクリプト言語を提供している．PJO スクリプトの例を**図 2** に示す．PJO は，top ブロックで必要なジョブの投入を終えた後，when ブロック（複数あってもよい）で投入した各ジョブの完了を待ち合わせ，それぞれのジョブに対応した後処理を実行するという実行モデルを採用している．待ち合わせるジョブは param=$x のようなパターンによる直観的な記述で指定できる．ジョブの依存関係も考慮して失敗したジョブの再投入を行えるなど優れたジョブ管理機能も備えている．

PJO の問題点は，各ジョブ投入の処理（プログラムの do_job ブロック内）と全体のジョブ投入のフローおよび待ち合わせ処理（それ以外）がそれぞれ別の Perl プロセス内で実行される[⋆1]ため，両者の処理の間で変

⋆1 do_job ブロック内のスクリプトが文字列として別 Perl プロセ

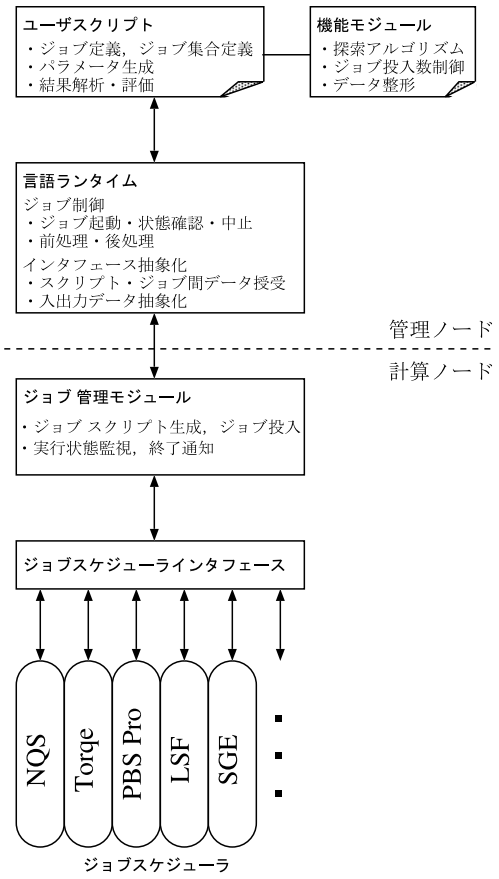

図 3 スクリプト処理システムの全体構成

数の値などの環境が共有されないことである．このことは直観性を損なうだけでなく，たとえば前に実行したジョブの結果を次に実行するジョブの入力に反映させるということが困難になる．[⋆1]

我々のスクリプト言語システムの開発は，この PJO をベースとする．具体的には，フロントエンドのスクリプト言語を再設計し，バックエンドのシステムも，現在はジョブスケジューラも現在は Parallelnavi-NQS にしか対応していないジョブスジケジューラインタフェースを他のスケジューラなどにも対応できるよう可搬化するなど，設計・実装を見直す．

3. システムの概要

図 3 に本スクリプト処理システムの構成を示す．2.2 節でも述べたように，本システムは PJO をベースとして開発を進めている．特にジョブ管理モジュールなどは，既存の PJO コンポーネントをできる限り活用する予定である．以下，各コンポーネントについて説明する．

ユーザスクリプト・機能モジュール 4 章で説明するスクリプト言語でジョブの定義（実行ファイルや入力データの指定）や，ジョブ投入や完了待ち合わせを含む実行のフローを記述する．同じスクリプト言語で実装された機能モジュールを取り込むことで，典型的な処理についてはなるべく簡単なユーザスクリプトのみで記述できるようにする．

言語ランタイム スクリプトで指示されるジョブ投入や終了待ち要求を実現したり，およびジョブとスクリプト間のデータの授受を支援するためのランタイム．

ジョブ管理モジュール 計算実行ノードで動作するモジュールであり，上記ランタイムからの要求を受けて実際にジョブスケジューラに対するコマンド（qsub など）を発行する．ジョブ完了などの監視・通知も行う．

ジョブスケジューラインターフェース ジョブスケジューラインターフェースには，ジョブ管理モジュールが発行したコマンドを NQS，Torqe，PBS Pro などの各ジョブスケジューラに対応するコマンドに変換するラッパ（開発中）が含まれる．ラッパを追加することで新たなジョブスケジューラにも対応できるため，高い汎用性を実現できる．

4. スクリプト言語の設計

本章では，開発中のスクリプト言語について例を用いて説明する．なお，プログラム例には，Perl に C++ 言語風のクラス定義の構文を追加したような言語を用いる．この設計はモジュールが抽象クラスとして提供されることをわかりやすく説明するためのプロトタイプであり，今後の開発で変更される可能性がある．(実際には，既存の Perl から構文を大きく拡張することはできるだけしない方針である．5.4 節も参照.)

4.1 例　　題

説明のための例題として，ユーザが以下のようなシミュレーション実験を行いたい場合を考える．

- 同一の実行ファイル（“`a.out`” とする）のプログラムを 5000 回，それぞれ異なる入力で実行する．
- 各ジョブの間に依存関係はなく並列実行が可能である．ただし，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” に用意されており，結果は “`output`i” に出力する．これらのファイル名はプログラム実行時のコマンド

スに渡され，解釈実行される．

⋆1 OS の環境変数を通じてデータを受け渡す手段は提供されているが，配列などの複雑なデータに対応しにくい．

```
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();
```

図 4 ユーザスクリプトの例

```
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();
  }
}
```

図 5 Restrict モジュールの定義

```
class Parallel : Job
{
  static $Par_njob;   # ジョブの数（ユーザ指定）
  static $Par_nsubmit=0; # 投入したジョブの数
  static $Par_nfinish=0; # 完了したジョブの数
  # Job で定義されたメソッドの拡張
  static main () {
    while ($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 ();
}
```

図 6 Parallel モジュールの定義

ライン引数で指定する．

- 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する．

単純な例ではあるが，気象予報におけるアンサンブル予測など，よく行われる実行パターンである．

4.2 ユーザスクリプト

4.1 節の例題を自動化するユーザスクリプトを**図 4** に示す．このスクリプトでは，Example という名前のジョブクラスを定義した後，最後の Example::main() でそのエントリポイントを起動している．Example クラスは，同時ジョブ投入数制限と複数ジョブの並列実行の機能を提供する各モジュール Restrict（**図 5**）と Parallel（**図 6**）をクラスの（多重）継承により取り込んでいる．また，built-in クラスである Job クラス（**図 7**）も Restrict や Parallel クラスを介して間接的に継承している．

```
class Job
{
 $Job_exec; $Job_args; $Job_env; ...
 # メソッド
 new (...) {...} # コンストラクタ
 static main (); # エントリポイント（ユーザ定義）
 before () {};   # ジョブ投入前の処理（ユーザ定義）
 do () { # ジョブを投入するために main から呼び出す
  # 全ての before を指定の順序で実行
  invoke_all_before();
  # Job_exec を Job_args のコマンドライン引数で実行
  exec ("$job_exec @Job_args");
  # ジョブの終了待ちの後，全ての after を実行
  #  （非同期実行）
  spawn wait_finish_and_invoke_all_after($self);
 }
 after () {};    # ジョブ完了後の処理（ユーザ定義）
}
```

図 7 Job built-in モジュールの定義

Restrict は制限するジョブの数を，Parallel は全ジョブ数をそれぞれクラスメンバ Restrict_max, Par_njob の値として設定することを，子クラスに要請する．この例ではそれぞれ 10，5000 に設定している．Parallel の子クラスはさらに，after_all メソッドを定義することで全ジョブ完了後の処理を指定することができる．ここではメッセージの表示を行っている．

Job_exec，Job_args，before，after は Job クラスで宣言・定義されているメンバ・メソッドである．Job_exec と Job_args には実行ファイル名とそのコマンドライン引数をそれぞれ設定する．また，before メソッドを拡張することによって，各ジョブ投入前に行う処理を定義できる．ここでは，コマンドライン引数の設定を行っている．after メソッドには，各ジョブについて実行終了を待ち合わせた後行いたい処理をで記述する．ここではメッセージの表示を行っている．ここで，before メソッド本体中で参照している Par_nsubmit は Parallel クラスが提供しているクラスメンバであり，今から投入しようとしているジョブが何番目かを参照できる．

このように，モジュール化された機能を取り込むことで，ユーザスクリプトでは最低限のパラメータのみを設定するだけで行いたい自動化を実装できる．

以下の節で，このスクリプトが動作する仕組みを各モジュールの実装を交えて説明する．

4.3 モジュールの実装

4.3.1 Restrict モジュール

図 5 が同時投入ジョブ数制限の機能をを提供する Restrict モジュールの定義である．このクラスを継承したクラスのジョブでは，その各投入前にセマフォの獲得が行われ（before メソッド），実行完了後に解放が行われる（after メソッド）．Restrict_max を超える数のジョブを投入しようとすると，セマフォの獲得待ちが発生するため，目的の機能が果たされる．

4.3.2 Parallel モジュール

図 6 が複数ジョブの並列実行を実現する Parallel モジュールの定義である．ここでは，エントリポイントとなる main メソッドが定義されている．図 4 の Example::main() の呼び出しは（他のクラスでは main をオーバーライドしていないため），ここで定義されている main に制御を移す．main メソッドには，ジョブクラスのインスタンスの生成およびそのジョブの投入を Par_njob 回繰り返す処理が記述されている．

before，after メソッドでは，それぞれジョブの投入数，完了数のカウンタをインクリメントしている．after メソッドではさらに，全てのジョブが終了したかを判定して，終了していれば after_all メソッド（図 4 の子クラス Example 定義されている）を呼び出す．

4.3.3 Job built-in モジュール

図 7 の Job クラスは，本スクリプト言語システムの built-in として提供され，全てのジョブクラスはこのクラスを継承する．このクラスでは，全ジョブクラスに共通する情報である実行ファイル名（Job_exec）やコマンドライン引数（Job_args），環境変数の情報 (Job_env) を設定するためのメンバや，before，after，main の各メソッドおよびコンストラクタ new が宣言・定義されている．

Parallel の main メソッドでジョブ起動のために呼び出している do メソッドもここで定義されている．このメソッドは，これまで説明してきた全ての before メソッドを実行した後プログラムを起動し，その終了待ちおよび after の実行のためのスレッドを生成（この例では非同期実行を行うためのキーワードを仮に spawn としている）する．

4.4 実行の流れ

図 4 のスクリプトの実行の流れを**図 8** にまとめる．ジョブ実行前後に起動する before，after メソッドはアスペクト指向言語における before，after アドバイス[2] や CLOS (Common Lisp Object System) の before, after メソッド[1] に相当する．複数の before，after メソッドの実行順序はこれらと同様に，before は親から子，after は子から親の順としている（CLOS の仕様では多重継承が存在する場合の優先順位も定義されている）．

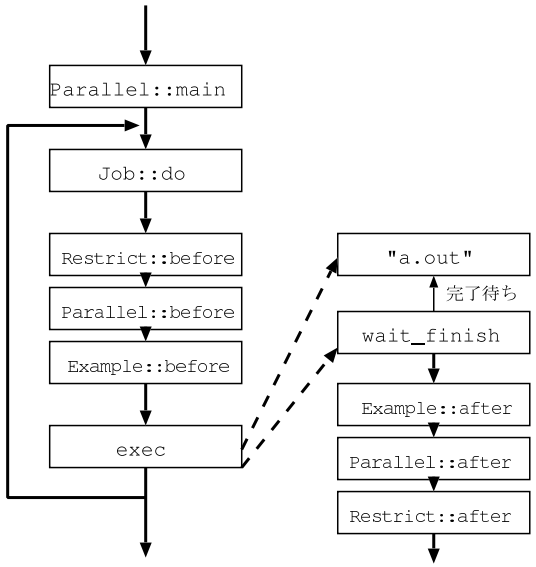

図 8 図 4 のスクリプトの実行の流れ（破線は非同期実行を表す）

5. 今後の検討課題

本システムは開発段階であり，今後開発を進めるにあたって様々な課題に衝突すると思われる．本章では，これまでの開発で浮上した検討課題について議論する．

5.1 複数のモジュール適用の安全性

4 章の例のように，複数の機能（モジュール）を同時に取り込むために多重継承を行った場合，互いのモジュールが干渉せずに正常に動作するかは保証できない．これは一般のオブジェクト指向言語の多重継承について古くから議論されている問題である．また，LaTeX においても複数のパッケージが干渉を起こす場合があることは知られている．

このような問題を言語レベルで完全に解決するのは困難である．ただし，本スクリプト言語で本格的なアプリケーションを書くことを目的としているわけではないので，厳密な安全性を追求する必要性は比較的小さい．モジュールの開発時にこの問題点を留意しておくことで干渉を最小限に抑えることは可能である．

5.2 スクリプト言語の並列性

現在の方針では，プログラミングが複雑になることを防ぐため，スクリプト言語そのものに並列処理のセマンティクスを持たせることはなるべく避けるべきであると考えている．

ただし，投入するジョブは並列に実行できる必要があり，それら各ジョブの終了の待ち合わせ（`after` メソッドの起動）については非同期に行うしかない．この場合でも，複数の `after` メソッドが同時に処理されないこと（直列化）を言語レベルで保証しておくなど，マルチスレッドプログラミングに不慣れなユーザが混乱しないよう考慮すべきである．

5.3 ジョブ間のデータ授受

現実の応用では，複数の種類のジョブを連携させて一つのシミュレーションを行うことが多い．例えば，データの初期化，シミューレーション本体，結果解析用にそれぞれ実行ファイルを用意し，本体部分のみをタスク並列で実行したい場合が考えられる．このような応用で，初期化したデータなどの本体ジョブへの受け渡しをどのように実現するかは重要な検討課題である．

ファイル経由での受け渡しが最も現実的な解であると考えられるが，その場合，実行プログラム中のデータ（巨大な行列，グラフなど複雑な構造のデータも含む）を言語・アーキテクチャに依存しない形で書き出す，またそのデータの取り出す処理を簡単に行えるようにする仕組みを，たとえば Fortran や C のライブラリとして提供する必要がある．

5.4 Perl との言語仕様の連続性

ユーザの学習コストや言語の実装コストを考慮すると，本論文の現状の仕様のように，Perl のセマンティクスや構文を大きく拡張することはあまり好ましくない．特に，オブジェクト指向的な記述を計算科学のユーザに強要するべきではないと考えている（ただし，モジュール実装のための記述に関してはその限りではない）．

今後は，本論文のコンセプトを維持しつつ，なるべく単純な実装（構文解析を伴わない Perl へのトランスレータや Perl ライブラリのみでの実装）が可能な仕様を検討していく予定である．

6. おわりに

本論文では，大規模シミュレーション計算における PDCA サイクルを自動化するためのスクリプト言語システムの構想を示した．言語設計はまだプロトタイプ段階であり，今後詳細を詰めていく必要があるが，探索アルゴリズムなどの各種機能を抽象クラスとして定義したモジュールとして提供するという枠組みにより，柔軟性と簡便性を両立できる見込みである．

設計においては実用性を重要視しており，実際に応用分野の研究者と連携し実プログラムを例題として扱いつつ開発を進めている．

謝辞 本研究の一部は，文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」の支援による．

参考文献

1) GUY L. Steele Jr.: *Common Lisp: The Language,* Second Edition, Digital Press (1990).
2) Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings European Conference on Object-Oriented Programming* (Akşit, M. and Matsuoka, S., eds.), Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp.220–242 (1997).
3) 湯山紘史，津邑公暁， 中島浩：タスク並列言語 MegaScript における高精度実行モデルの構築, 情報処理学会論文誌. コンピューティングシステム， Vol.46, No.12, pp.181–193 (2005).
4) 富士通株式会社：*Parametric Job Organizer.*