高生産並列スクリプト言語 Xcrypt の開発

平石　拓[†1]　安部達也[†1]　三宅洋平[†1]
岩下武史[†1]　中島浩[†1]

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スパコン上などで行う際，パラメータスイープや最適パラメータ探索のために，同一のプログラムをパラメータを変えながら同時に大量実行することで並列処理を実現することも多い．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複雑な処理が行えないことがある．一方，Perl 等の汎用スクリプト言語で記述することは特に一般の計算科学の研究者にとっては敷居が高い．そこで我々は，既存のスクリプト言語をベースとしつつ，このような大量のジョブの大量実行に特化することで簡便な記述を可能にするジョブ並列スクリプト言語 Xcrypt 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．本報告では，Xcrypt 言語およびその処理系のプロトタイプを示す．

Development of Xcrypt: Highly Productive Parallel Script Language

Tasuku Hiraishi,[†1] Tatsuya Abe,[†1] Yohei Miyake,[†1]
Takeshi Iwashita[†1] and Hiroshi Nakashima [†1]

Computational scientists often perform large scale simulations in their research or development such as car body design and drug discovery. Such a simulation often has plenty of sequential/parallel runs of a single program 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 complicated workflows with them. They can also use general 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 named Xcrypt, which is based on an existing script language and specialized for executing plenty of jobs to enable us to describe such automation easily. We will realize both flexibility and easiness to write by modularizing features, such as smart 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 report shows a prototype of the Xcrypt language and its system.

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．パラメータスイープや最適パラメータ探索は PDCA サイクルの典型的な例である．またこの際，依存関係のないジョブは並列に実行されることが望ましい．

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．既存の自動化ツールとしては，GUI ベースの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で非同期的に実行される多数のジョブの制御を記述するのは容易ではない．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

†1 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University

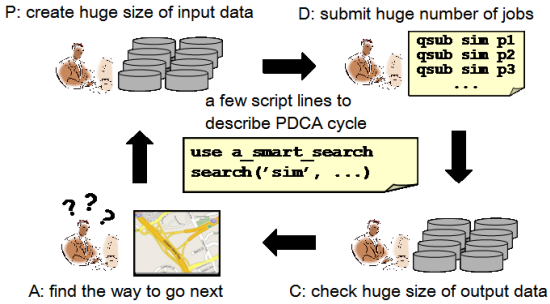

図 1 PDCA サイクル

```
# メイン処理
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 != $rc)
        PJO::abort "job executes failed.\n";
    }}
  }
}}
# 各ジョブの終了を待ち合わせて実行される処理.
# 各ジョブ投入時の$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

とが多い.

そこで我々は，汎用プログラミング言語としての柔軟性を維持しつつ，より簡便にジョブ並列実行の記述ができるようにすることを目的として，タスク並列スクリプト言語 Xcrypt，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，既存の汎用スクリプト言語である Perl をベースとしつつ，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（クラスライブラリ）として提供できるようにした. 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.

Xcrypt ではまた，ジョブスクリプトの生成や投入したジョブ状態の監視などをシステム側がサポートすることにより，プログラマが実行フローにおける本質的な部分のみの記述に集中できるようにしている.

本稿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 章で現状の Xcrypt 言語について例を用いて説明し，最後に 5 章でまとめと今後の課題を述べる.

2. 既存のジョブ並列言語

2.1 MegaScript

MegaScript[1] は，Ruby をベースとしたジョブ並列言語である. MegaScript では 1 つのジョブ実行を「タスク」（＝オブジェクト）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. 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は困難な場合も多いため，実用上の問題が生ずる.

また，Ruby ベースであるため，プログラマはある程度オブジェクト指向プログラミングに慣れている必要がある.

2.2 PJO

PJO[2] は富士通研究所によって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. [⋆2]

⋆1 `do_job` ブロック内のスクリプトが文字列として別 Perl プロセスに渡され，解釈実行される.

⋆2 OS の環境変数を介することによるデータの受け渡し手段は提供されているが，記述が複雑になるうえ，配列などの複雑なデータに対応しにくい.

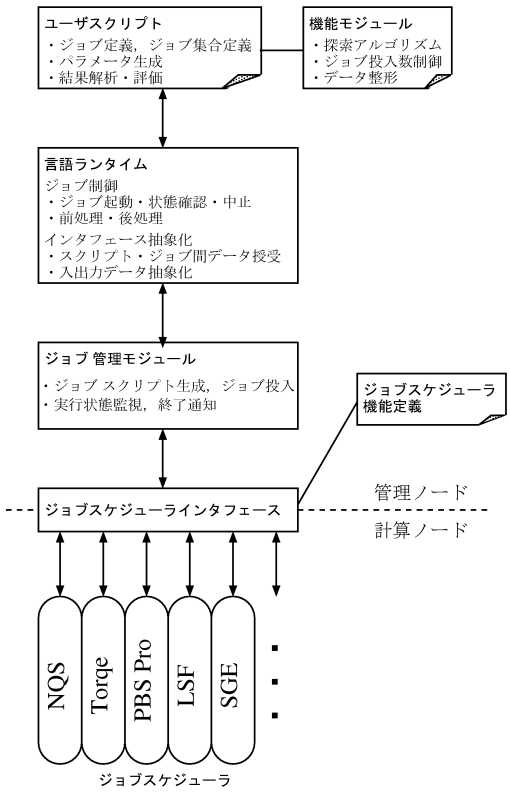

図 3 スクリプト処理システムの全体構成

また，あるジョブの完了に依存するジョブの投入は `when` ブロックに書く必要があるため，たとえば逐次で実行する必要がある複数個のジョブ投入を `while` 等のループ構文で直観的に書くということができない．

Xcrypt の開発は，この PJO をベースとして行っている．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についても，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. Xcrypt システムの概要

図 3 に Xcrypt システムの構成を示す．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 Xcrypt 言語で記述されたジョブの定義（実行ファイルや入力データの指定）およびジョブ投入や完了待ち合わせを含む実行のフロー．探索アルゴリズム等の機能モジュールも同じ Xcrypt 言語で実装することができる．そのような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記述で実現できるようにしている．

言語ランタイムライブラリ ジョブ投入や終了の待ち合わせを行ったり，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，スクリプトから利用できるランタイムライブラリ．

ジョブ管理モジュール 上記ランタイムからの要求を受けてジョブスクリプトを生成し，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．また，ジョブ状態（スケジューラへの投入前・投入済・プログラム実行中・正常終了・異常終了など）の変化の監視およびその（Xcrypt プロセスへの）通知を行う．

ジョブスケジューラインタフェース NQS，Torque，PBS Pro などの各種ジョブスケジューラのコマンドラインオプションやジョブスクリプトの記法等の違いを吸収するインタフェース．利用したいスケジューラに対応する設定ファイルを書いておく（Perl のハッシュオブジェクトとして記述する）と，このインタフェースがそれを参照し，ジョブスクリプトや qsub のコマンドラインオプションなどを当該スケジューラに適するように変換する．スケジューラ定義モジュール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，高い可搬性を実現できる．

なお，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おり，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だけ活用するようにしている．

4. Xcrypt 言語

本章では，開発中の Xcrypt 言語によるプログラミングについて簡単な例を用いて説明する．

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が，同時に実行するジョブの数は 10 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．
- 実際にジョブを投入する前に，スクリプトが意図通りにジョブを投入することを確認するためにドライ実行（プログラムの実行のみスキップするスクリプト実行）を行う．

これは単純な例ではあるが，気象予報におけるアンサ

```
use base qw(limit dry core);
$dry::dry = 0; # 1 に変更するとドライ実行
limit::init(10);

%sample = (
    'id' => 'example',        # 任意のジョブ ID
    'RANGE0' => [1..5000],
    'exe' => './a.out',
    'arg0S' => 'input$_[0]',  # 入力ファイル名
    'arg1S' => 'output$_[0]', # 出力ファイル名
    # プログラム実行に必要なファイル
    'copiedfile0' => 'a.out',
    'copiedfile1S' => 'input$_[0]',
    'queue' => 'myqueue',     # ジョブを投入する
                              # キューの名前
);
# ジョブの準備，投入および終了待ち
# まとめて prepare_submit_sync(%sample); でもよい
@jobs = prepare (%sample);
@threads = submit (@jobs);
@sync (@threads);
```

図 4 ユーザスクリプトの例

```
package limit;

use strict;
use NEXT;
use Thread::Semaphore;
my $smph;

sub init {
  $smph = Thread::Semaphore->new($_);
}

sub new {
    my $class = shift;
    my $self = $class->NEXT::new(@_);
    return bless $self, $class;
}
sub start {
    my $self = shift;
    $self->NEXT::start();
}
sub before {
    my $self = shift;
    $smph->down;
    $self->NEXT::before();
}
sub after {
    my $self = shift;
    $self->NEXT::after();
    $smph->up;
}
```

図 5 limit モジュールの定義

ンブル予測などのアプリケーションにおいてよく利用さ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 ID を持つジョブ列をオブジェクトとして定義した後，prepare，submit，sync の関数呼び出しにより，それぞれジョブの実行前準備，投入および終了待ち合わせを行っている．

ジョブの各種パラメータは，オブジェクトのメンバの値として指定する．例ではまず，RANGE0 に配列 [1..5000] を設定することにより，このジョブが単一のジョブではなく，それぞれ example1〜example5000

```
package dry;

use strict;
use NEXT;
our $dry;

sub new {
  my $class = shift;
  my $self = $class->NEXT::new(@_);
  return bless $self, $class;
}
sub start {
  my $self = shift;
  $self->NEXT::start();
}
sub before {
  my $self = shift;
  if ($dry) {
    # プログラム実行のコマンドラインを空文字列に
    # 置き換える
    $self->{exe} = '';
    for ( my $i = 0; $i <= $user::max; $i++ ) {
      my $arg = 'arg' . $i;
      $self->{$arg} = '';
    }
  }
  $self->NEXT::before();
}
sub after {
  my $self = shift;
  $self->NEXT::after();
}
```

図 6 dry モジュールの定義

```
package core;

use strict;

sub new {
  ジョブ名に対応するディレクトリを作成し
  copiedfile で指定されたファイルをそこにコピー
}
sub before {}
sub start {
  ジョブスクリプトを生成し，qsub コマンド実行
}
sub after {}
```

図 7 core 組み込みモジュールの定義

という ID を持つ 5000 個のジョブからなるジョブ列であることを宣言している．

次に，exe，arg0S，arg1S によりジョブで実行したいプログラムおよびそのコマンドライン引数，copiedfile0，copiedfile1S によりプログラムを実行するために必要なファイル，queue によりジョブを投入する（ジョブスケジューラの）キューの名前をそれぞれ指定している．

ここで，名前の末尾に 'S' が付加されているパラメータについては，ジョブ列中の各ジョブに異なる値が設定される．この場合のハッシュオブジェクトのメンバの値には，関数，配列，文字列のいずれかを設定する．各ジョブのパラメータの値は，このメンバの値が関数の場合はその戻り値，配列の場合はその配列とジョブ列の各要素を 1 : 1 対応させたときに対応する配列の要素の値，文字列の場合はそれを Perl の eval 関数で評価した結果となる．なお，関数や eval される文字列のコード中では，RANGEn の配列のうち各ジョブに対応する要素の値を$_[$n$] このようにジョブ別にメンバの値を設定する方法は，id および RANGEn を除く全てのメンバに対して適用することができる．例えば exe の代わりに exeS を用いてジョブごとに異なるプログラムを実行することも可能である．

図 4 のスクリプトは，同時ジョブ投入数制限とドライ実行の機能を提供する各モジュール limit（**図 5**），dry（**図 6**）および built-in モジュールである core（**図 7**）をクラスの多重継承により取り込んでいる．limit モジュールは同時に実行するジョブの数を limit::init 関数により設定することをそのモジュールの利用者に要請する．この例では 10 に設定している．

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 limit モジュール

図 5 が同時実行ジョブ数制限の機能を提供する limit モジュールの定義である．このモジュールを読み込むと，各ジョブの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）ようになる．ユーザスクリプトの先頭の，limit::init により設定した数を超えるジョブを一度に実行しようとすると，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 dry モジュール

図 6 がドライ実行を実現する dry モジュールの定義である．ドライ実行は，ジョブ投入直前に before メソッドにおいてジョブオブジェクトの exe，arg$n$ メンバの値を空文字列に置き換えることにより実現している．

4.3.3 core 組み込みモジュール

図 7 の core モジュールは，Xcrypt 言語システムの built-in として提供される．全ての Xcrypt ユーザスクリプトはこのモジュールを取り込む必要がある．

new メソッドは図 4 の prepare 関数によりジョブ投入前の準備のために呼び出され，各ジョブに対応する作業ディレクトリ（各ジョブの入力・出力ファイルや実行ファイル，中間ファイル，ジョブスクリプトファイル等はすべてこのディレクトリに保存される）を作成し，そこに必要なファイルをコピーする．[⋆1]

start メソッドは，図 4 の submit 関数を通して呼び出され，ジョブスクリプトの生成およびジョブの投入（qsub の実行）を行う．

4.4 スクリプトの実行の流れ

図 8 に図 4 のスクリプトの実行の流れを示す．

スクリプト中ではまず，ジョブオブジェクトに対して prepare 関数が適用されている．この関数は，まず RANGE パラメータを持つジョブオブジェクトに対してはその値である配列と "...S" の形のメンバを解釈することで，適切なジョブオブジェクト列を生成する．各ジョブオブジェクトはその生成時に，ロードした各モジュールの new メソッドが適用[⋆2]される．このメソッドは，ジョブ投入処理前に必要な準備（core モジュールの new メソッドにおける作業ディレクトリの生成など）を行う．prepare の返り値は，生成されたジョブオブジェクト列である．(RANGE パラメータを持たないジョブオブジェクトに対しては単に new メソッドの適用が行われる)

次にこのジョブオブジェクト列を引数として，submit 関数を呼び出している．submit 関数は各ジョブオブジェクトに対して，そのジョブの前処理・投入処理・後処理を行うためのスレッドを生成する．[⋆3]

このスレッドの具体的な処理は以下の通りである．まず，ロードした各モジュールの before メソッドを呼び出す．[⋆4]（limit モジュールの同時ジョブ実行数制限による実行待ちはここで発生する.）次に，start メソッドを呼び出し，ジョブスクリプトの生成およびジョブの投入を行う．そしてその投入したジョブの終了を待ち合わせ，その後，ロードされた各モジュールの after メソッドを呼び出す．

submit 関数の戻り値は，与えられたジョブオブジェ

⋆1 ここでは，管理ノードと計算ノードのファイルが NFS 等により共有されている環境を想定している．共有されていない場合は通常のコピーではなく staging 処理になる．

⋆2 適用順序は Perl の NEXT モジュールによりクラス階層グラフの左深さ優先順となっている．

⋆3 現行の Perl（v5.10）のスレッドは大量のメモリを消費する．実際，1 ジョブに 1Perl スレッドを対応させる現在の実装では，200 程度のジョブ投入で 32GB のメモリを消費し尽くしてしまう．そのため，利便性を失わない程度に並行性を制限することで，大量の（少なくとも数千程度）ジョブ投入に耐えられる実装モデルに切り替えることを検討中である．

⋆4 現状，before メソッドや after メソッドの呼び出し順序は「各メソッドの最後あるいは先頭で必ず NEXT のメソッドを呼び出すこと」という Xcrypt ライブラリの coding convention によって実現している．

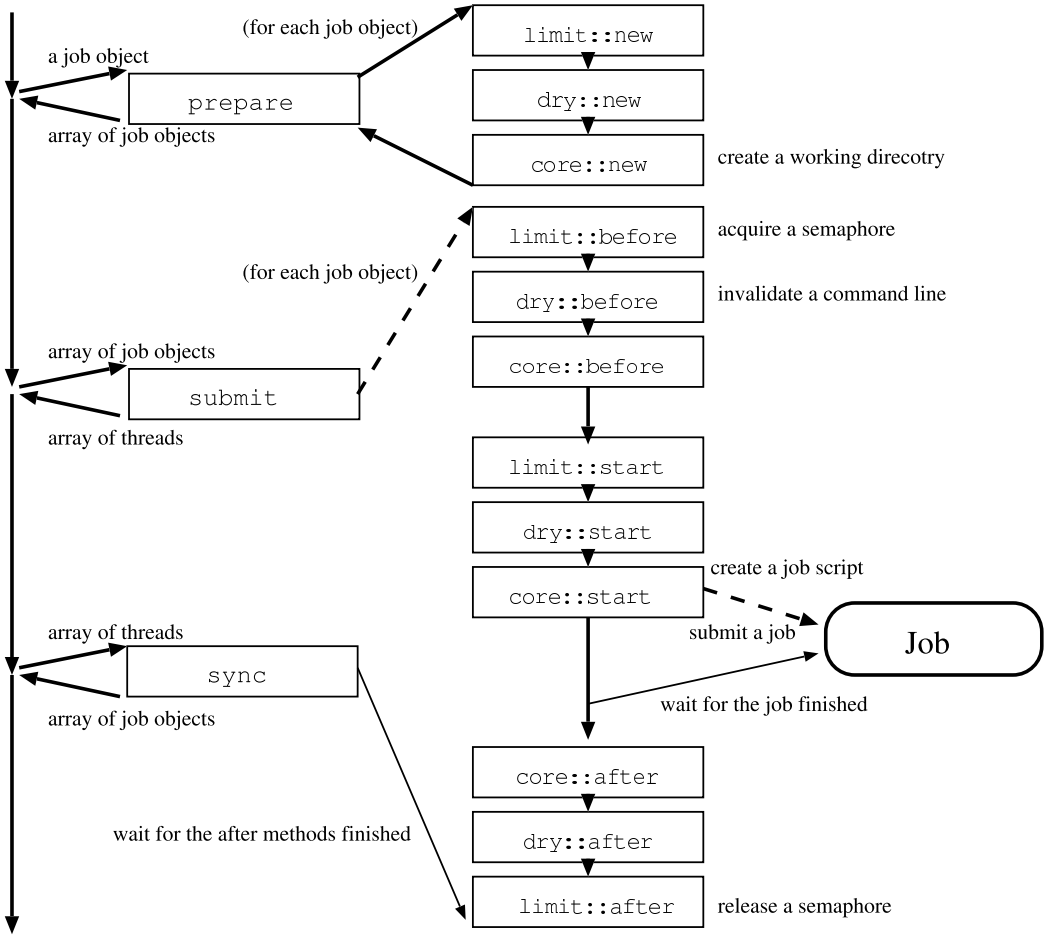

図 8　図 4 のスクリプトの実行の流れ（破線は非同期実行を表す）

クト列に対応する上記のスレッドオブジェクト列である．sync 関数をこのスレッドオブジェクト列を引数として呼び出すと，これらのスレッドの終了，すなわちジョブおよび after メソッドの実行完了を待ち合わせる．

4.5 ま と め

以上のように，Xcrypt の基本的なプログラミングスタイルは，投入したいジョブの各種パラメータ（ID，実行ファイル名，コマンドラインオプション，バッチキューシステムのキュー名など）を指定したジョブオブジェクトを（Perl のハッシュオブジェクトとして）定義し，そのオブジェクトを引数とする prepare 関数呼び出しで実行のための前準備を行った後，submit 関数でジョブを投入し，その終了を sync 関数で待ち合わせる，というものである．ジョブスクリプトの生成や投入したジョブの状態監視などの処理をシステムに任せることができるため，submit と sync という簡便な記述のみでジョブの投入，終了待ちを行うことができる．

なお，prepare と submit の呼び出しが分離されているため，prepare が準備した入力ファイルにジョブ別に手を加えるような Perl コードを両者の間に挿入することもできる．また，submit と sync が分離されているため，複数の独立するジョブ（列）を submit で投入した後，最後に一度に同期待ちを行うような処理も直観的な記述により実現できる．

追加モジュールの開発者は，独自のメンバやメソッドを追加することにより，ジョブオブジェクトのクラスを拡張して Xcrypt の機能を拡張することができる．このうち，特に new，before，start，after メソッドは特別な意味を持ち，prepare 実行時やジョブ投入前，投入時，終了後に行われる処理を追加・変更することができる．これにより様々な機能を柔軟に追加でき，それらの機能の利用も簡単にできるようになっている．

5. まとめと今後の課題

本稿では，主に大規模シミュレーション計算におけるタスク並列処理をともなう PDCA サイクルの自動化を，強力なプログラマビリティ失わずに簡便に実現するための並列スクリプト言語 Xcrypt について示した．システム設計はまだプロトタイプ段階であるが，ジョブ実行処理において典型的な各種機能をクラス Perl モジュール）として提供する枠組み等により，柔軟性と簡便性を両立できている．

今後の課題としては，各ジョブに与える入力ファイルを簡単に生成したり出力ファイルから必要なデータを簡単に取り出せるようにするためのライブラリの開発がまず挙げられる．Fortran プログラムの入力ファイルとして用いられるネームリストや，行列データを記述する CRS 形式等に特化することで，sed や awk 等の従来のツールより高機能かつ簡単に利用できるようにすることを目指している．

また，スクリプトの実行中に一部のジョブだけが失敗してしまった場合やスクリプトそのものが中断してしまった場合に，効率的に実行を再開できるようにするためのチェックポインティング・リトライ機能をサポートする必要がある．さらに，最適パラメータ探索等に対応した様々なアルゴリズムを Xcrypt のモジュール機構においてうまくモジュール化できるかも検証していく予定である．

一方，最も重要な点として，Xcrypt の記述が実際に計算科学分野の開発者に受け入れられるかについても検証する必要がある．そのため，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) 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).
2) 富士通株式会社：*Parametric Job Organizer.*

質 疑 応 答

Q.（竹内） Xcrypt という名前の意味・由来は？

A. Xperience，eXa などを意味する X に script を足したが，なぜか（語呂の問題で？）i が y に変化した．

Q.（鵜川） グリッドやクラウドへジョブを投げるものを連想する．どう違う？

A. GUI ベースのワークフローツールというものは存在するが，複雑なフローには対応できない．

Q.（鵜川） 小規模なクラスタで使いたい．可能か?

A. 使えるだろう．通常のシェルスクリプト記述より便利に使えると考えている．

C. 基本的にはスパコン等のバッチキューシステムでの利用を想定しているが，単純なコマンド実行（OS スケジューラがジョブスケジューラに対応していると考えることができる）や rsh ベースの実行にも対応できるような汎用性も持たせている．(設定ファイルをユーザやシステム管理者が書くことでそれぞれの環境に対応することができる)

Q.（大島） クラスタなどでリモートに実行させるのが簡単に書けると思って良いのか．

A. はい．基本的にバッチキューシステムにおいて多数のジョブを実行する際に，ジョブスクリプト作成などの負担を軽減することが目的の一つ．

Q.（大島） Perl は使いやすいのだろうか？導入として GUI があるのもいいのではないか？

A. Perl は有名なスクリプト言語の中では，C や Fortran のプログラマにも馴染みやすい部類に入ると思う．例えば Ruby で書こうと思うとオブジェクト指向を覚えなければならない．手軽に書けて，かつ複雑なフローも記述もできることが目標であるため，プログラミング言語である必要があると考えた．

C. Xcrypt で投入したジョブの状態などを視覚的に確認できるようにするための GUI ラッパーのようなものはあってもよいと考えている．

Q.（横山） 東大では GXPC を推奨している．これは Makefile ベース．Xcrypt は探索など書けるのが良さそうだ．他に違いなどがあれば教えて欲しい．

A. Makefile は宣言ベース．Xcrypt も当初は，ジョブの依存関係をメンバに名前で指定するような宣言ベースの言語設計だった．しかし，特に，主に想定している利用者層である計算科学者（C や Fortran のプログラマ）にとっては，命令的でないとわかりにくいとの意見があった．そのため現在の Xcrypt は，ジョブの定義は宣言的に行うが，投入は命令的に行うというスタイルになっている．

Q.（笹田） 発表は DSL としての書きやすさの話が中心で，どのあたりの機能によって並列処理が書きやすくなっているかは良くわからなかった．

A. ジョブの状態管理とかをシステム任せにできるのでお手軽なのではないか．

Q.（笹田） 並列処理部分と逐次（管理）処理部分を分けたのがポイントなのか？

A. ジョブ投入部分以外は並列を意識させないつもりである．

C. ジョブスクリプト自動生成機能のほか，ジョブの終了等の状態監視をシステムに任せることができるので，ユーザ側はジョブのパラメータ定義＋`submit()`/`sync()` によって簡単に（非同期に実行される）ジョブの管理を行えるようになっていることがポイントだと考える．