高生産並列スクリプト言語 Xcrypt

平石拓
共同研究者：中島 浩，岩下武史，安部達也，三宅洋平
京都大学 学術情報メディアセンター

目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

背景(1)

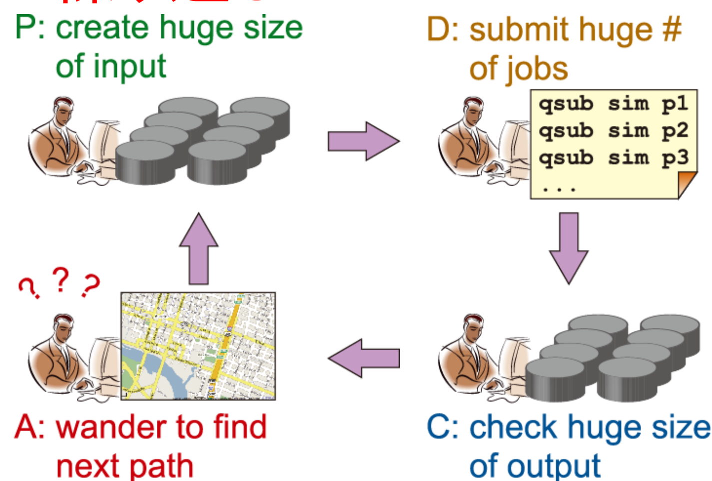
- スパコン上の計算科学分野のシミュレーション
  - 創薬，車体設計，天気予報
- パラメータスイープ，最適パラメータ探索

→ 同一のプログラムの大量実行の繰り返し

＝*PDCAサイクル*

- Plan: 入力を生成
- Do: ジョブ投入(ジョブ並列)
- Check: 結果を処理
- Action: 次の計算を検討

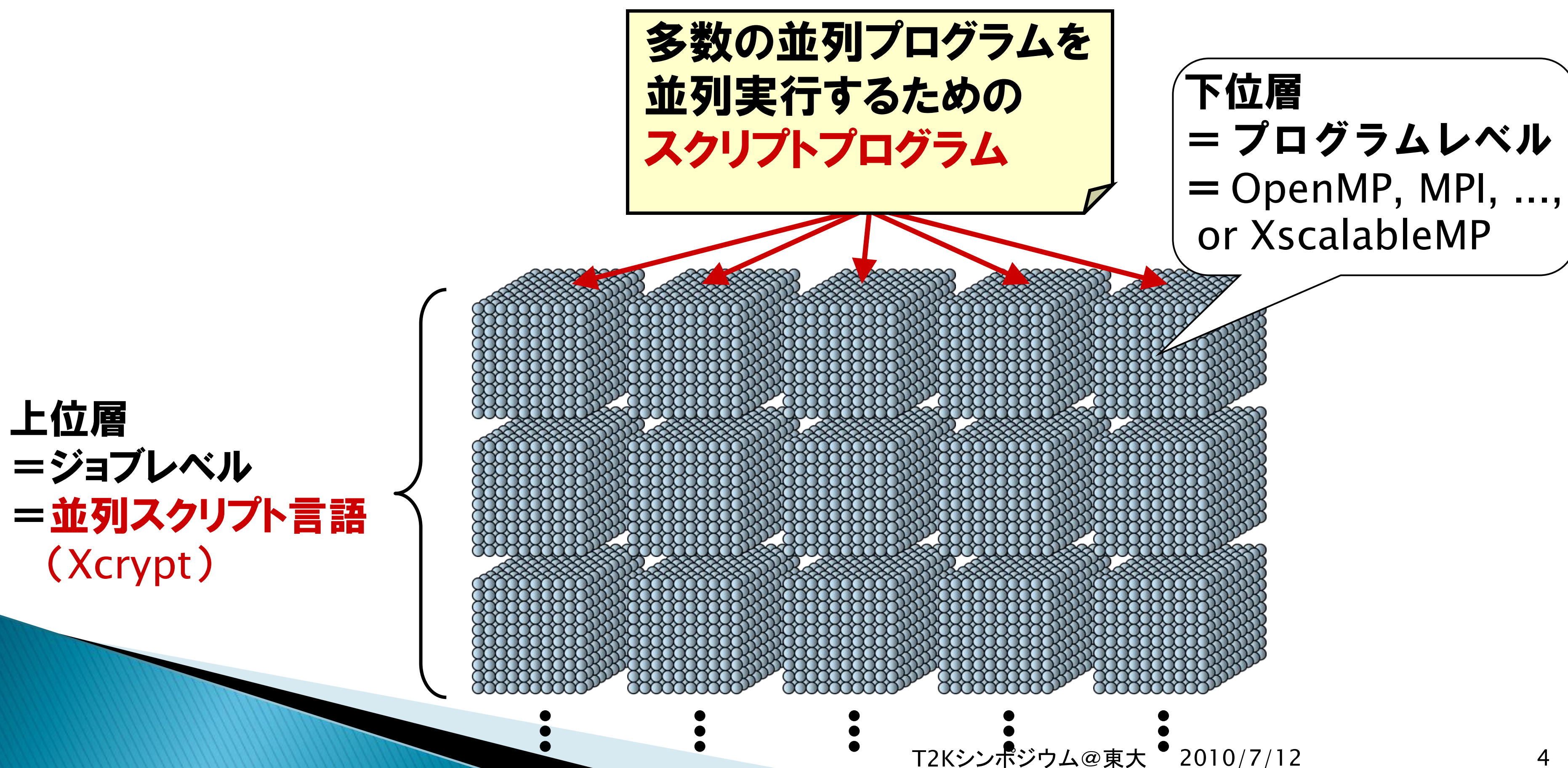
の繰り返し



- 自動化したい
  - ワークフロー，出力から次の入力を生成

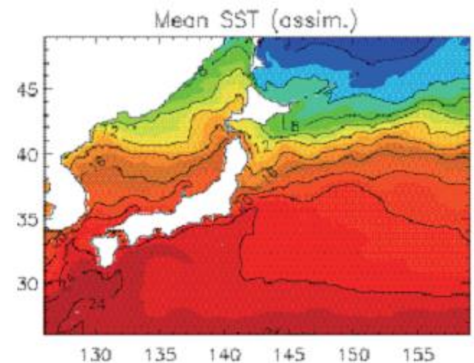
背景(2):階層的大規模並列処理

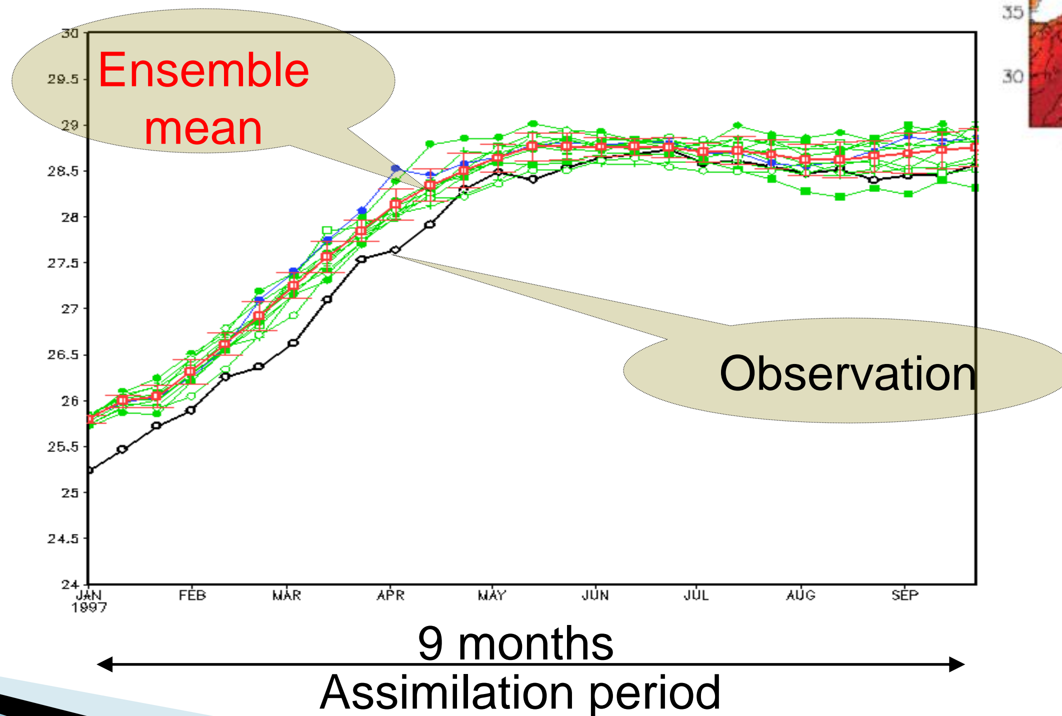
- ジョブレベル–プログラムレベル の2階層並列
  ➔1000並列 x 1000並列 ＝ 100万並列



例：アンサンブルシミュレーション

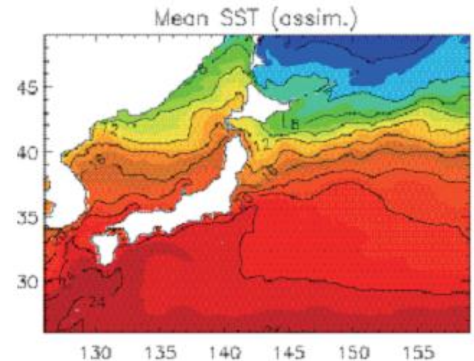
- 異なる多数の初期パラメータに対して同一の予測計算（1アンサンブル＝1ジョブ）
- 最後に結果をまとめる





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



入力データ読込
&前処理

予測計算

予測計算

予測計算

予測計算

後処理
&結果出力

繰り返し

何が問題か？

- 多くのスパコン環境ではバッチスケジューラの利用が前提（NQS, SGE, Torque, etc.）
  - 単なるコマンドラインではプログラムを走らせられない
    - qsubコマンドでスケジューラに「ジョブ」の実行依頼
    - 要求する計算資源や実行したい処理を記述した「ジョブスクリプト」を書く必要がある
  - ジョブの終了確認も大変
    - qstatコマンド or 出力ファイルができているか確認

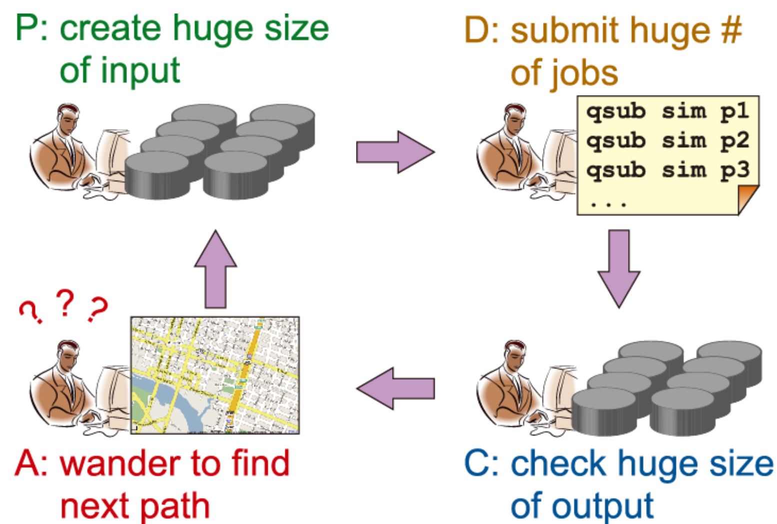
- ジョブ数が増えてくると管理が煩雑
  - （正常／異常）終了したジョブの把握
  - 各ジョブに対するジョブスクリプト・入力ファイル・コマンドラインオプションの生成
  - 出力ファイルの管理，解析

（PerlやRubyで書こうと思えば書けるが．．．）

→ ジョブ並列処理記述のための定まったプログラミング環境が存在しない

現状とXcryptの位置付け

- 手動でやっている人も多い
- ジョブの実装言語（Fortran／C, OpenMP／MPI）で自動化
  - 「計算科学者」に親しみのある言語
  - ジョブに依存関係がある場合など，複雑なフロー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．生産性も悪くない
  - これで直接書こうとするといろいろ煩雑
    - 非同期に実行されるジョブの管理 etc.
    - バッチスケジューラとのやりとり
- 専用スクリプト言語 (DSL)
  - ワークフローの記述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
- GUIワークフローツール
  - NAREGI WFT など
  - 非常に手軽だが，柔軟性・自由度が低い



目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

設計思想

- 簡便性・直観性
  - バッチスケジューラとの煩雑なやりとりからユーザを解放
    - ジョブ実行を，同期／非同期の手続き呼び出しのように簡単に書ける
      → 全体の実行フローの記述のみに注力できる
  - （典型的な）大量のジョブ実行を手軽に扱える
    - 入力ファイルやコマンドラインだけが一部異なるような
- 柔軟性・汎用性
  - 「プログラミング言語としての」自由度を失わない
  - 複雑な処理も「書こうと思えば」書ける
    - 例：反復処理，ジョブの結果に応じて次の処理を判断（パラメータ探索）
    - モジュールとして提供できる（エンドユーザが苦労して書かなくてよい）
- 可搬性
  - 実行するプログラムの書き換えを要求しない
  - 様々なシステムで（スクリプトの改変なしに）動作する
    - バッチスケジューラごとのインタフェースの違いを吸収
    - qsub，qdel，qstat（に相当する）コマンドの存在のみ仮定

Xcryptの仕様概要

- Perlベースの手続型言語
- ジョブはオブジェクトとして抽象化
  - 実行ファイルなどの必要情報を宣言的に定義
  - 使用するコア数などもここで宣言（スケジューラ非依存）
  - ジョブの前処理／後処理も定義できる
- 大量の並行ジョブの投入を簡便に行うための組込機能
- ジョブ投入を，同期／非同期手続き呼び出し的に書ける
  - submit()/sync()
- バッチスケジューラごとのインタフェースの違いは，スクリプトとは独立に書かれた「設定ファイル」で吸収
  - 「設定ファイル」はインストール時に管理者が書く
- モジュール提供機構
  - ジョブ投入数制限などの複雑な機能をモジュールとしてエンドユーザに提供
  - Perlのオブジェクト指向（ジョブクラスを拡張）＋前処理／後処理を追加で「差し込む」ことができる

目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

Xcryptスクリプトの例1（単一ジョブ）

```
use base qw (core);

%template = (
    ‘ID’ => ‘example’,                      ジョブにつけるユニークID
    'exe' => './a.out',                     実行ファイル
    ‘arg0’ => ’input’,                      コマンドライン引数（ここでは入出力ファイル名）
    ‘arg1’ => ’output’,
    'copiedfile0' => 'a.out',               作業ディレクトリにコピーするファイル
    'copiedfile1' => 'input',
    ‘JS_queue’ => ’myqueue’,                バッチスケジューラのキュー名
    ‘after’ => sub { print “$_->{id} finished.¥n” }   後処理
);
@job = prepare (%template);                 ジョブオブジェクト生成
submit(@job);                               ジョブ投入
sync(@job);                                 ジョブ終了待ち
```

Xcryptプログラムの基本スタイル

- ジョブのテンプレートを定義
- prepare関数でテンプレートからジョブオブジェクトを生成
- submit関数にジョブオブジェクトを与えてジョブ投入
- sync関数でジョブの終了を待ち合わせ

Xcryptスクリプトの例2（並行ジョブ）

```
use base qw (core);

%template = (
    RANGE0 = [1..5000],          1～5000の5000個のジョブ
    ‘ID’ => ‘example’,           ジョブにつけるユニークIDの接頭辞
    'exe' => './a.out',          実行ファイル
    ‘arg0@’ => sub{”input$_[0]”},
    ‘arg1@’ => sub{”output$_[0]”},   連番のコマンドライン引数
    ‘JS_queue’ => ’myqueue’,     バッチスケジューラのキュー名
    ‘after’ => sub { print “$_->{id} finished.¥n” }   後処理
);
@jobs = prepare (%template);     ジョブオブジェクト列生成
submit(@jobs);                   ジョブ投入
sync(@jobs);                     ジョブ終了待ち
```

prepare関数

- 入力：ジョブテンプレートオブジェクト
- RANGEnに基づいて，ジョブオブジェクト列に展開
  - ジョブオブジェクト毎には異なるidがふられる（"example1"～"example5000"）
  - パラメータ名の末尾に'@'をつけると，ジョブ毎に異なるパラメータ値を与えることができる.
    - 文字列→eval関数の結果
    - 関数→その返り値

    Rn$, ... RANGE$n$ の範囲内の各ジョブに対応する値を参照できる
- 各ジョブの実行に必要なファイルのステージング
- 返り値：ジョブオブジェクト（列）

submit関数

- 入力:ジョブオブジェクト(列)
- 与えられた全てのジョブオブジェクトに対して,
  - ジョブの前処理(before())
  - (「設定ファイル」を参照しつつ)ジョブスクリプトの生成
  - ジョブ投入
  - ジョブの後処理(after())

  を非同期に行う
- スレッド生成が完了したら制御が戻る
  (完了は待たない)
- 返り値:入力のジョブオブジェクト(列)自身

sync関数

- 入力：ジョブオブジェクト（列）
- 与えられた全てのジョブオブジェクトに対応するジョブの完了を待ち合わせる
- 返り値：入力のジョブオブジェクト（列）自身

Xcryptスクリプトの例3（拡張モジュール適用）

```
use base qw (core limit);
limit::initialize (10);
%template = (
    RANGE0 = [1..5000],
    'ID' => 'example',
    'exe' => './a.out',
    'arg0@' => '"input$R0"',
    'arg1@' => '"output$R0"',
    'copiedfile0' => 'a.out',
    'copiedfile1@' => '"input$R0"',
    'JS_queue' => 'myqueue',
    'after' => sub { print "$_->{id} finished.¥n" }
);
@job = prepare (%template);
submit(@job);
sync(@job);
```

- `use base qw (core limit);` `limit::initialize (10);` — モジュールの取り込み
- `RANGE0 = [1..5000]` — 1～5000の5000個のジョブ
- `'ID'` — ジョブにつけるユニークID
- `'exe'` — 実行ファイル
- `'arg0@'`, `'arg1@'` — 連番のコマンドライン引数
- `'copiedfile0'`, `'copiedfile1@'` — 作業ディレクトリにコピーする連番のファイル
- `'JS_queue'` — バッチスケジューラのキュー名
- `'after'` — 後処理
- `prepare` — 作業ディレクトリ作成
- `submit` — ジョブ投入
- `sync` — ジョブ終了待ち

Xcrypt拡張モジュールの提供機構

- ジョブオブジェクトはcoreクラスのインスタンス
- 拡張モジュールの開発者は，
  - メンバ（⊃エンドユーザに提供するパラメータ）の追加
  - メソッドの拡張・追加

  によりcoreクラスを拡張する
- ただし，以下の名前のメソッドは特別な意味を持つ
  - new：ジョブオブジェクト生成時の処理 —— prepare() 内
  - before：ジョブ投入の（追加の）前処理
  - start：ジョブ投入処理
  - after：ジョブ投入の（追加の）後処理

  （before／start／after）—— submit() 内

例：limitモジュールの定義

```
package limit;
use strict;
use NEXT;
use Coro::Semaphore;

my $smph;
# モジュールの利用者が最初に呼び出す関数（同時ジョブ投入数の上限設定）
sub initialize {  $smph = Coro::Semaphore->new($_); }

# ジョブの追加前処理：セマフォ獲得
sub before {$smph->down;}
# ジョブの追加後処理：セマフォ解放
sub after  {$smph->up;}
```

limit以外の拡張モジュール

- dry.pm
  - 全てのジョブの実行を飛ばす（ドライ実行）
- convergence.pm
  - 今回と前回とのジョブの結果の差が設定値以下に収束するまで実行を繰り返す
- sandbox.pm
  - ジョブごとに作業ディレクトリ（箱庭）を作成して，そこに必要ファイルをコピー後，ジョブ実行（従来のデフォルトの挙動）
- dependency.pm
  - ジョブの依存関係を宣言的に記述できるようにする

```
%temp3 = (
  'ID' => 'job3', ..., 'depend_on' => ['job1', 'job2'],
)
```

バッチスケジューラ設定ファイル

- qsubのオプションやジョブスクリプトの書き方の違いを吸収
- xcrypt-*x.x*/lib/config/*mysched*.pm
  をシステムに合わせて定義
- 環境変数 XCRJOBSCHED を *mysched* にすると設定が反映される.
  - 実行時にも変更可能, 複数のスケジューラの使い分けも可能
- 書く人が多少Perlに慣れていることを要求
  - システム管理者が書くことを想定
- 定義すべき情報は:
  - qsub, qdel ,qstat のコマンドのパス
  - qsubの出力からrequest IDを取り出す方法
  - qstatの出力からrequest ID(列)を取り出す方法
  - ジョブスクリプトのオプションの書式　　など

例：T2K京大用の設定ファイル

- xcrypt-*x.x*/lib/config/KYOTO.pm

```
$jobsched::jobsched_config{“KYOTO"} = {
    qsub_command =>
      "/thin/local/bin/qsub",
    qdel_command => "/usr/bin/qdel -K",
    qstat_command =>
      "/thin/local/bin/qstat",
    jobscript_queue => '# @$-q ',
    jobscript_stdout => '# @$-o ',
    jobscript_stderr => '# @$-e ',
    jobscript_proc => '# @$-lP ',
    jobscript_cpu => '# @$-lp ',
    jobscript_memory => '# @$-lm ',
    jobscript_verbose => '# @$-oi',
    jobscript_verbose_node => '# @$-OI',
    jobscript_workdir =>
      '$QSUB_WORKDIR‘,
extract_req_id_from_qsub_output => sub
      {
        my (@lines) = @_;
        if ($lines[0] =~ /([0-9]*)¥.nqs/) {
            return $1;
        } else {
            return -1;
        }
    },
extract_req_ids_from_qstat_output =>
      sub {
        my (@lines) = @_;
        my @ids = ();
        foreach (@lines) {
            if ($_ =~ /([0-9]+)¥.nqs/) {
                push (@ids, $1);
            }
        }
        return @ids;
    },
};
```

目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

Xcryptの実装

- ほとんどPerlのライブラリ群として実装
  - Xcryptプロセス＝Perlインタプリタ
  - ただし，スクリプトへの前処理（変換）あり
- Perlインタプリタが起動すると，最初に
  - 異常終了ジョブ監視スレッド
  - 外部メッセージ処理スレッド

  を立ち上げた後，スクリプト本体の処理をPerlプログラムとして実行
- 上記の裏方スレッドでジョブの状態を管理・更新
  - initialized, prepared, submitted, queued, running, done, finished, aborted
  - 状態遷移の履歴はファイルにも残す
→ 中断後の再実行時に途中からやり直せるようにするため

Xcrypt実行イメージ

ログインノード

Xcryptプロセス＝Perlプロセス

ジョブスレッド1

ジョブスレッド2

ジョブスレッド3

異常監視スレッド

メッセージ処理スレッド

定期的qstat（投入したはずのジョブがあるか確認）

qsub

バッチスケジューラ（NQS等）

計算ノード

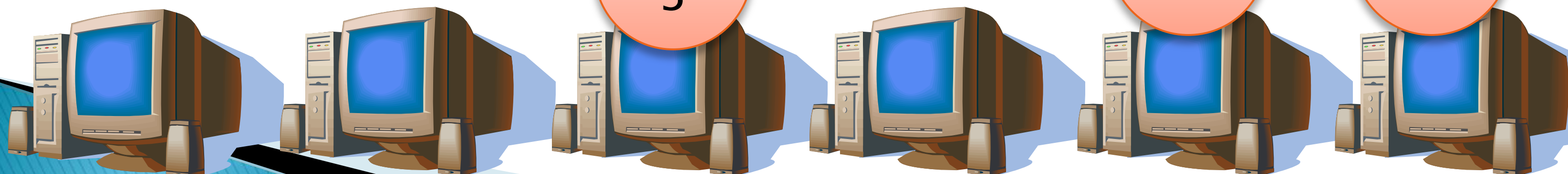
ジョブ開始通知
ジョブ終了通知

計算資源割り当て

Job 3

Job

Job



Xcryptの実行イメージ

ログインノード

Xcryptプロセス＝Perlプロセス

ジョブスレッド1

ジョブスレッド2

ジョブスレッド3

異常監視スレッド

メッセージ処理スレッド

定期的qstat
（投入したはずのジョブがあるか確認）

qsub

バッチ

ジョブ開始通知
ジョブ終了通知

計算ノード

TCP/IP通信でXcryptに通知

```
#!/bin/sh
...
inventory_write loginhost 9999 "running"
./a.out 3
inventory_write loginhost 9999 "done"
```

Job 3

Job

Job

目次

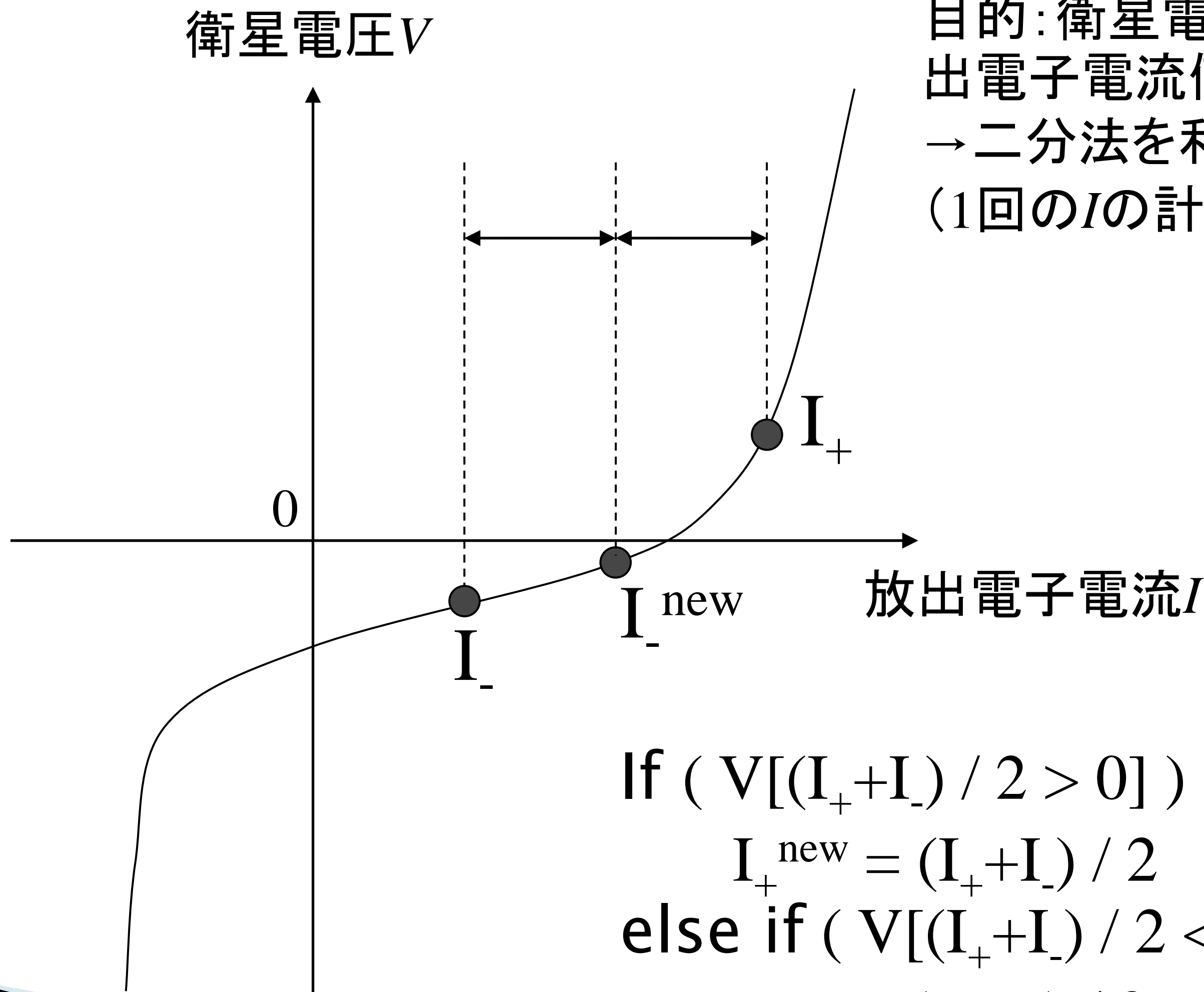
- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

実用例(1)：並列アルゴリズムの性能評価

探索空間: RANGE0–RANGE2

```
prepare_submit_sync (
  'id' => 'parLU_eval',
  'RANGE0' => [1,2,4,8,16],      # # of workers per node
  'RANGE1' => [2000,4000],       # Matrix size
  'RANGE2' => [1..3],            # # of trials
  'exe0' => './lu',
  'JS_cpu@' => sub { $_[0]; },     # # of cores
  'JS_node' => 1,                  # # of nodes
  'arg0_0@' => sub {"-n $_[0]";},  # # of threads
  'arg0_1@' => sub {"$_[1]"},      # problem size
  'JS_queue' => 'myqueue',
  'JS_limit_time' => 180         # 制限時間
  );
```

実用例(2):二分法による最適パラメータ探索



目的:衛星電圧Vを0とする最適放出電子電流値Iを求める
→二分法を利用した最適値探索
(1回のIの計算=1ジョブ)

If ($V[(I_+ + I_-) / 2 > 0]$) then

  $I_+^{new} = (I_+ + I_-) / 2$

else if ($V[(I_+ + I_-) / 2 < 0]$) then

  $I_-^{new} = (I_+ + I_-) / 2$

end if

テンプレート定義

```
%template = {
    'id' => 'job00',
    'exe' => 'mpiexec',
    'cpu' => '1',    'proc' => '64',
    'arg0' => '-n',    'arg1' => '${QSUB_VNODES}',
    'arg2' => './mpikempo3D',
    'arg3' => '4';,    'arg4' => '4',    'arg5' => '4',
    'arg6' => '< plasma.inp',
    'stdofile' => 'stdout',
    'stdefile' => 'stderr',
    'JS_queue' => 'ghxxxxx',
    'JS_group' => 'ghxxxxx'
};
```

ジョブ投入の反復実行

```
my $wpe1 = 1.0;   my $wpe2 = 1.5;  my $wpem = 1.0;  my $phi = 1;
until (abs($phi) < 0.1) {  # 結果が誤差範囲内になるまで反復
    $wpem = ($wpe1+$wpe2)/2;
    $template{‘id’} = $template{‘id’} . ‘+’;   # ジョブ名が重複しないようにIDを変更
    my @job = prepare(%template);
    foreach (@job) {   # 入力ファイルの作成（ネームリストplasma.inpのwp(3)の値のみ$wpemに変更）
        my $generate = CF("$ENV{'PWD'}/plasma.inp","$ENV{'PWD'}/"."$_->{id}");
        $generate -> KR("wp(3)", "$wpem");
        $generate -> do(); }
    @results = submit_sync(@job), "¥n";
     foreach (@results) {  # 出力ファイルの抽出（最終行3列目の値を$phiに代入）
        my $phiext = EF("file:$ENV{PWD}/$_->{id}/pbody");
        $phiext->ED('L/E');   $phiext->ED('C/3');
        my @phis = $phiext->ER();
        $phi = $phis[0];  }
    print "Potential: ", $phi, "¥n";
    # $phiの正負に応じて左右どちらかから範囲を狭める
    if ($phi < 0) { $wpe1 = $wpem; } else{ $wpe2 = $wpem; }
}
```

目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

Xcryptが落ちたときの復帰

- Xcryptの実行は長時間(〜数週間)におよぶ
- 途中で落ちる(マシンダウン，ログインノードのプロセス制限時間)も考慮しなければならない
- ジョブの状態遷移をログファイルに残す
- Xcryptが落ちたとき
  - 実行中だったジョブはそのまま実行を続ける
  - 終了したらそのことをログに残す
- Xcrypt再実行時
  - すでに終了しているジョブは実行をスキップ
  - まだ実行中のジョブは投入をスキップして終了待ち
  - ジョブの同一性は「ジョブID」でとる

リモートからのジョブ投入

- スパコンのログインノードで長時間プロセスを走らせられない
- 複数のサイトに跨ってジョブ並列処理
- 「計算場所」をオブジェクトとして定義 → ジョブオブジェクトのメンバに指定

```
my $env0 = &add_env({'host' => 'abet@super.para.media.kyoto-u.ac.jp'});
my $env1 = &add_env({'host' => 'abet@t2k.ccs.tsukuba.ac.jp'});
my $env2 = &add_env({'host' => 'h22029@ha8000.cc.u-tokyo.ac.jp'});
my $env3 = &add_env({'host' => 'b30331@thin.kudpc.kyoto-u.ac.jp'});

%template = (
    'id'          => 'job4',
    'exe0@'       => sub { 'echo ' . (40 + $_[0]) . " > job4_$_[0]_out"},
    'RANGE0'      => [0,     1,      2,      3,      4           ],
    'env@'        => [$env0, $env1,  $env0,  $env3,  $default_env],
);
&prepare_submit_sync(%template);
```

目次

- はじめに
- Xcryptの開発方針
- Xcrypt言語の説明
- 実装
- 実用例
- その他の機能
- 今後の課題・まとめ

今後の課題

- 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

  → 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
- Xcrypt側で行う各ジョブごとの前処理，後処理がボトルネック

  → 計算ノード（ジョブ）側で実行するようにしたい
- ユーザビリティの（定量的，定性的）評価
- Perl以外のフロントエンド？
  - Lisp, Ruby, Python, ...
  - 言語中立な仕様設計

まとめ

- スパコン（バッチスケジューラ）上におけるタスク並列処理を簡便に記述するためのスクリプト言語Xcryptの開発
  - バッチスケジューラに関する煩雑な処理をユーザから隠蔽し，フローの記述のみに注力できるように
  - （大量の）ジョブ投入を既存言語（Perl）の手続き呼び出しとして書ける
  - ジョブの前処理／後処理を「差し込む」ことによる機能追加
  - リモートからのジョブ投入も可能
  - 本体が中断したときの復帰にも対応