# Xcrypt: A Perl Extension for Job Level Parallel Programming

Tasuku Hiraishi
Academic Center for Computing and Media Studies, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
tasuku@media.kyoto-u.ac.jp

Tatsuya Abe
RIKEN Advanced Institute for Computational Science
Kobe, JAPAN 650-0047
abet@riken.jp

Takeshi Iwashita
Academic Center for Computing and Media Studies, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
iwashita@media.kyoto-u.ac.jp

Hiroshi Nakashima
Academic Center for Computing and Media Studies, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
h.nakashima@media.kyoto-u.ac.jp

## ABSTRACT

For the effective use of resources in large-scale parallel computing environments such as supercomputers, we often use job level parallelization, that is, plenty of sequential/parallel runs of a single program with different parameters. For describing such parallel processing easily, we developed a scripting system named Xcrypt, based on Perl. Using Xcrypt, even computational scientists who are not familiar with script languages can perform typical job level parallel computations such as parameter sweeps by using a simple declarative description. In addition, programmers who are familiar with Perl can write a wide variety of execution flows such as optimal parameter search. Xcrypt provides Perl libraries that enable us to write job executions simply as function calls, relieving us from annoying tasks such as creating job scripts for batch schedulers and managing states of jobs. Furthermore, Xcrypt provides a mechanism to add hooks invoked before/after jobs are submitted/finished modularly. This enables us to add various features such as search algorithms as modules. In this paper, features of Xcrypt, our implementation of Xcrypt, and practical examples of parallel job executions performed by using it are discussed.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks, Modules, packages*

## General Terms

Languages, Design

## 1. INTRODUCTION

We need to parallelize computations to use large-scale computing resources effectively. Parallelization is done not only at the *program level* by using OpenMP and/or MPI but also at the *job level* by running a single program with different parameters in parallel.

Parameter sweeps and optimal parameter searches for car body design, drug discovery, and static software automatic tuning [10] are real examples of job level parallel processing. For achieving coarse-grained parallelism, job level parallelization can be used since it is easy to implement. Furthermore, computing systems with a parallelism greater than 1M are emerging; in such systems, it is difficult to write a parallel program that runs efficiently. However, we can use such systems effectively by using two layers of parallelism at both the program level and job level.

In a system where a batch scheduler such as NQS [4], LSF [3], SGE [11], or Torque [2] is installed, we can impose computing resource assignment on the back-end scheduler. Therefore, we can just pre-existing script languages, e.g., Perl, Ruby, Python or shell scripts, to automate job level parallel executions. However, there remain many tasks that are hard to implement using these languages, such as preparing input files, generating a job script for each job, submitting jobs and waiting for them to be finished, extracting necessary parts from output files to analyze results, and handling for jobs that are abnormally finished. Furthermore, interfaces to batch schedulers, e.g., a shell command for submitting jobs and a grammar for job scripts, differ with the system used. These issues are especially serious for computational scientists who are not familiar with script languages.

Although we can use GUI-based workflow tools such as that

presented in [7], most of them lack flexibility. For example, it is hard to implement optimal parameter search using the bisection algorithm with such tools.

Therefore, we have developed a scripting system named Xcrypt (pronounced "*iks-krài-pt*"), based on Perl. This system provides various additional supports that facilitate the easy description of job level parallel processing.

In Xcrypt, a job is abstracted as a *job object* and we can write a job submission simply as an asynchronous procedure call with the job objects given as arguments. This enables us to seamlessly glue job executions that are components of a parallel processing job. In addition, Xcrypt has a mechanism to add various useful features, such as limiting the number of simultaneously running (or queued) jobs, as modules.

The advantages of Xcrypt are as follows.

- *Ease of use and Flexibility*
  - We can perform typical job level parallel computations such as parameter sweeps by using a simple declarative description.
  - In addition, programmers who are familiar with Perl can write a wide variety of execution flows. This is easier than writing them by using Perl or other pre-existing script languages since procedures involving job submission and waiting for job completion can be specified by using procedure calls without having to consider interfaces to batch schedulers.
- *Portability*
  - Once a user writes an Xcrypt script, it runs on various systems without requiring modifications. Note that generally, different batch schedulers are installed in different systems. Differences in interfaces among systems are handled by configuration scripts, which are written separately from Xcrypt scripts. These scripts need to be prepared for each system when installing Xcrypt. It is expected to be done by the system administrators.
  - We make no assumption regarding application programs. In contrast, MegaScript [12, 8], a Ruby-based job level parallel script language, assumes that applications receive inputs from the standard input and send outputs to the standard output. We only assume that the underlying batch scheduler provides the commands for submitting a job, canceling a job, and displaying the status of submitted jobs.
- *Extensibility*
  - We can extend Xcrypt by implementing extension modules. A module is defined as an ordinary Perl library or an extension of the base class for job objects. We can extend the base class by adding not only methods and members through object-oriented Perl programming but also preprocessings and postprocessings that are performed before and after job execution, respectively.
- *Fault resilience*
  - Even if submitted jobs or an Xcrypt process is aborted, we can, to a certain extent, restore the original state quickly by executing the same Xcrypt script again. This is possible since Xcrypt saves the state transition log of the jobs, and when Xcrypt is re-executed, it skips the previously executed tasks and executes the aborted jobs again.

# 2. XCRYPT FEATURES

This section explains the important features of Xcrypt added to Perl.

## 2.1 Declaration of Used Modules

An Xcrypt script should begin with a declaration of the modules to be included, as follows:

`use qw(`*module-name*$_1$ *module-name*$_2$ `...core);`

Note that the `core` module must be included.

## 2.2 Functions for Handling Jobs

### 2.2.1 Creating job objects

To create job objects, the `prepare` function is used as follows:

`@`*jobs* `= prepare (%`*template*`)`

`%`*template* is a job template object defined as a hash object that contains information on jobs. Table 1 shows important keys that a template object should contain. A key name prefixed with `JS_` implies that the corresponding value is referred to by *a batch scheduler configuration script* and passed to the underlying scheduler by following its interface.

Values of `before`, `after`, `before_in_job`, and `after_in_job` should be "code references" (anonymous functions or references to predefined named functions). `Before` and `after` are procedures that are invoked before job submission and after job completion, respectively. These procedures are executed in the Xcrypt process. On the other hand, `before_in_job` and `after_in_job` are procedures executed in Perl processes run during the job execution. When these functions are defined, Xcrypt serializes values of user-defined global variables and the job object to be submitted, and then creates Perl scripts. These scripts define the global variables whose values are to be serialized and then invoke `before_in_job` or `after_in_job`. These members are useful for preventing preprocessings and postprocessings (such as creating input files and analyzing output files) from being bottlenecks when plenty of jobs are submitted. We minimized the loss of programmability by allowing some of the variables defined in Xcrypt to be read in these procedures. We employed the Dump::Dumper [9] CPAN module for serialization; the module can serialize not only scalar values but also arrays, hash variables, and functions.

The created job objects basically inherit the members of the template object and their values. In addition, the job objects have additional (private or public) members and methods, such as the member that indicates the current state of a job.

**Table 1: Important members of job template objects**

| Name ($n$, $m$: integers equal to or greater than 0) | Meaning |
|---|---|
| id | a string to identify the job |
| exe$n$ | a command line to be executed as the job execution (exe0, exe1, ..., exe$n$ are executed in this order) |
| arg$n$_$m$ | The $m$-th command line option of exe$n$ |
| JS_cpu | # of CPUs required for the job |
| JS_node | # of nodes required for the job |
| JS_queue | name of the queue to which the job is submitted |
| before | a procedure invoked before submitting the job |
| after | a procedure invoked after the job is completed |
| before_in_job | a procedure invoked just before exe0 |
| after_in_job | a procedure invoked immediately after all the exe$n$s |
| RANGE$n$ | extraction ranges from the template |

When the input template object contains RANGE$n$ as its member, prepare creates multiple job objects. In this case, the created job objects are given different ids by postfixing sequential numbers: for example, the return value of prepare ('id'=>'example', 'RANGE0'=>[1..100]) is an array of job objects whose ids are example1–example100. When the values of the members RANGE0, RANGE1, ... of the template object are arrays with lengths $n_1, n_2, \ldots$ respectively, $(n_1 \times n_2 \times \ldots)$ job objects are created. The value of RANGE$n$ is not limited to successive integers; [3, -5, 2, -10] and ['--option', '--another-option'] are also permissible.

We can ensure that the member values of the created job objects are different from each other by postfixing @ to the member names, such as arg0_0@, and we set a function to the corresponding member value; each member value of the created job objects is the return value of a given function. In the function body, the assigned element of RANGE$n$ can be referred to as the $n$-th argument (can be referred to by $VALUE[$n$]). We will give an example later.

### 2.2.2 *Submitting jobs*

We can submit jobs by using the submit function as follows:

```
submit (@jobs)
```

@*jobs* should be an array of job objects created by prepare. All the jobs contained in the array are submitted.

Details of the submit function are as follows. The submit function creates a thread, called *job thread*, for each job object. The task of a job thread is as follows.

1. Invoke the user-defined before.
2. Invoke all the before methods defined in the declared modules in use, in the left-to-right order.
3. Invoke the start method defined for the leftmost module among the used modules. The start method in the core module generates a job script for the batch scheduler by referring to configuration scripts, and it submits a job by using the command for submission provided by the underlying batch scheduler (e.g., qsub).
4. Wait for the submitted job to be completed.
5. Invoke all the after methods defined in the declared modules in use, in the right-to-left order.
6. Invoke the user-defined after.

The execution of submit itself is completed after the creation of job threads.

Job threads are created as lightweight threads by the using Coro CPAN module [5], which enables us to create thousands of threads at a reasonable memory/time cost.

### 2.2.3 *Waiting for finishing jobs*

We can wait for the jobs to be finished by using the sync function:

```
sync (@jobs)
```

It waits for all the job threads corresponding to the job objects included in the array @*jobs* to be finished.

## 2.3 Extension Modules

When only the core module is used, all the job objects created by prepare are instance objects of the core class. Developers of Xcrypt libraries can extend Xcrypt by extending the core class. End users can use such extensions only by adding the name of the extended class in use.

The manner of implementing extension modules is based on the manner in which class extension is carried out in object-oriented Perl programming. In addition, in Xcrypt, a method named new, before, start, or after has special meaning, as explained in the previous section.

We can extend the new and start methods to extend or modify the procedures of creating job objects and submitting them, respectively. We can also define the before/after methods as additional hooks that are invoked before/after job execution.

## 2.4 Batch Scheduler Configuration Scripts

Xcrypt has a mechanism to provide information about batch schedulers, such as the paths to the qsub (submitting a job), qdel (canceling a job), and qstat (displaying the status of

```
use config_common;
use File::Spec;

$jsconfig::jobsched_config{KYOTO} = {
  # command path
  qsub_command => "/thin/local/bin/qsub",
  qdel_command => "/usr/bin/qdel -K",
  qstat_command => "/thin/local/bin/qstat",
  # options
  jobscript_preamble => ['#!/bin/sh'],
  jobscript_workdir => '$QSUB_WORKDIR',
  jobscript_option_stdout
    => option_with_default('# @$-o ', 'stdout'),
  jobscript_option_stderr
    => option_with_default('# @$-e ', 'stderr'),
  jobscript_option_merge_output
    => boolean_option ('# @$-eo'),
  jobscript_option_node => '# @$-lP ',
  jobscript_option_cpu => '# @$-lp ',
  jobscript_option_memory => '# @$-lm ',
  jobscript_option_limit_time => '# @$-cp ',
  jobscript_option_limit_cputime => '# @$-lT ',
  jobscript_option_queue => '# @$-q ',
  jobscript_option_group => '# @$-g ',
  jobscript_option_verbose
    => boolean_option ('# @$-oi'),
  jobscript_option_verbose_node
    => boolean_option ('# @$-OI'),
  # Extract from output messages
  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; },};
```

**Figure 1: Batch scheduler configuration script for the supercomputer in Kyoto University operated from FY2008 to FY2011 (Fujitsu NQS is installed).**

```
use config_common;
use File::Spec;

$jsconfig::jobsched_config{KYOTO-LSF} = {
  # command path
  qsub_command => '/usr/share/lsf/.../qsub <',
  qdel_command => '/opt/dpc/bin/qkill',
  qstat_command => '/usr/share/lsf/.../qjobs',
  # options
  jobscript_preamble => ['#!/bin/sh'],
  jobscript_option_stdout
    => workdir_file_option('#QSUB -oo ', 'stdout'),
  jobscript_option_stderr
    => workdir_file_option('#QSUB -eo ', 'stderr'),
  jobscript_option_merge_output
    => boolean_option ('# @$-eo'),
  jobscript_other_options => sub {
    my $self = shift;
    my $node = $self->{JS_node} || 1;
    my $cpu = $self->{JS_cpu} || 1;
    my $thread=$self->{JS_thread}||$cpu;
    my $memory=$self->{JS_memory}||(61440/16*$cpu).'M';
    return
      "#QSUB -A p=$node:t=$thread:c=$cpu:m=$memory";},
  jobscript_option_limit_time => '#QSUB -W ',
  jobscript_option_queue => '#QSUB -q ',
  jobscript_option_group => '#QSUB -ug ',
  # Extract from output messages
  extract_req_id_from_qsub_output => sub {
    my (@lines) = @_;
    foreach my $ln (@lines) {
      if ($ln =~ /Job\s+<([0-9]+)>\s+is/) {return $1;}}
    return -1; },
  extract_req_ids_from_qstat_output => sub {
    my (@lines) = @_;  my @ids = ();
    shift (@lines); # no ID included in the first line
    foreach (@lines) {
      if ($_ =~ /^\s*([0-9]+)\s+/) {push (@ids, $1);}}
    return @ids; },};
```

**Figure 2: Batch scheduler configuration script for the supercomputer in Kyoto University operated from FY2012 (LSF is installed).**

submitted jobs) commands, independently from end user scripts.

For example, the configuration script for Fujitsu NQS and LSF, which are installed on the former and current supercomputer system of Kyoto University, are defined as shown in Figure 1 and Figure 2, respectively. These scripts specify the paths to the `qsub`, `qdel`, and `qstat` commands, how to get request IDs from outputs of `qsub` and `qstat`, how to request the required number of CPUs from the scheduler, and so on.

A parameter named `jobscript_option_`*name* corresponds to the member named `JS_`*name* of job objects. Xcrypt refers to `jobscript_option_`*name* to know the format for specifying the job option and refers to `JS_`*name* to get the corresponding option value. Ther `jobscript_other_options` parameter in Figure 2 is used when a parameter given to parameter does not correspond one-to-one with a member of a job object.

For example, when a job object contains the following members:

```
JS_node=>4, JS_cpu=>16,
JS_thread=>32, JS_memory=>'28G',
```

Xcrypt (the job script generator invoked by `core::start` method) generates a job script that include the following lines:

```
# @$-lP 4
# @$-lp 16
# @$-lm 28G
```

for NQS and the following line:

```
#QSUB -A p=4:t=32:c=16:m=28G
```

for LSF. Note that `JS_thread` is ignored for NQS since the way of handling the member is not defined in Figure 1. Indeed, this version of NQS does not take the parameter for specifying number of threads. In contrast, the ways of handling `JS_verbose` and `JS_verbose_node` are defined in Figure 1 and not defined in Figure 2. Thus, these members are available for NQS but ignored for LSF.

We wrote configuration scripts also for Hitachi NQS, SGE,

```
[environment]
sched = KYOTO
[template]
JS_queue = myqueue
```

**Figure 3: An example of a user configulation script.**

**Table 2: Status of jobs in Xcrypt**

| State name | Description |
|---|---|
| initialized | The job object was initialized |
| submitted | The job was submitted to the batch scheduler |
| queued | The job submission was successful |
| running | The user application program is running |
| done | The user application program has been run |
| finished | All the tasks of the job thread finished |
| aborted | The job is abnormally completed |

and Torque environments and tested that Xcrypt works on these systems.

## 2.5 User Configuration Scripts

In Table 1, the value of `JS_queue` should differ with the system on which Xcrypt is executed. However, we cannot define the value in the batch scheduler configuration script because the value also depends on the user. For such parameters, Xcrypt provides the mechanism of user level configuration scripts.

Figure 3 shows an example of a user configuration script. When this script is loaded, the member value of `JS_queue` of a generated job object is set to `myqueue` if the value is not defined explicitly in the argument of the `prepare` function. This script also says that Xcrypt uses the `KYOTO` configuration script as the default configuration script.

Owing to the use of these two types of configuration scripts, we can completely remove system-dependent pieces of code from Xcrypt scripts.

# 3. IMPLEMENTATION

## 3.1 Overview

Almost the entire Xcrypt system is implemented as Perl modules. When Xcrypt is executed with an Xcrypt script, it performs simple translations of the script and executes Perl with the translated script. The translation includes the addition of declarations, some necessary global variables, and top-level statements to create two background threads: one for communicating with jobs and the user interface and the other for monitoring the status of jobs.

## 3.2 Managing States of Jobs

Xcrypt internally manages all the created jobs by constructing a set of pairs (*ID*, *state*). Table 2 shows a list of the status and Figure 4 shows the state transition diagram.

A job object is "initialized" just after it is created by `prepare`. Then, it becomes "submitted" when the job submission command is executed in `start`, and becomes "queued" when Xcrypt is notified that the submitted job is successfully queued by the output message of the job submission command.

**Table 3: Skipped method/function invocations when Xcrypt is re-executed ("S" implies skipped).**

| State | prepare | before | start | after | sync |
|---|---|---|---|---|---|
| initialized | | | | | |
| submitted | | S | | | |
| queued | | S | S | | |
| running | | S | S | | |
| done | | S | S | S | |
| finished | | S | S | S | S |
| aborted | | | | | |

In order to make transitions to "running" and "done," Xcrypt has to be notified by the job, which is generally assigned to other nodes, that the job has been started and completed. Thus, Xcrypt generates a job script that executes the following commands:

`inventory_write` *ID* `running`
(*Execute Perl to invoke* `before_in_job`)
`./a.out0 arg0_0 arg0_1 ...  # exe0`
...
`./a.out`*n* `arg`*n*`_0 arg`*n*`_1 ...  # exe`*n*
(*Execute Perl to invoke* `after_in_job`)
`inventory_write` *ID* `done`

The `inventory_write` command notifies Xcrypt running on *hostname* that the status of job whose `id` is *ID* has become "running" or "done," by using some communication method (such as TCP/IP communication or creating a file in NFS). Then the Xcrypt's communication thread can detect the notification and update the job's status.

When a job running on computation nodes is suddenly aborted, the Xcrypt's monitoring thread can detect it by periodically executing `qstat` and checking its output. Xcrypt judges that a job is "aborted" if the job is "queued" or "running" but does not appear in the output of `qstat`.

## 3.3 Restoring the Execution State

Even if an Xcrypt process is aborted because of an some accident such as a node down, we can restore the original state quickly solely by re-executing Xcrypt with the same Xcrypt script if the Xcrypt script satisfies the following conditions.

- The same `id` is assigned to the same job even in the re-execution.
- It is permissible for side effects such as the creation of input files to occur again.

Xcrypt saves the log that contains a record of all the state transitions of the jobs. The restoration mechanism skips previously executed tasks when Xcrypt is re-executed. More concretely, if the job has previously been executed, invocations of `before`, `start`, `after`, and `sync` are skipped depending on the last state in the previous execution (see Table 3). Jobs that are being re-executed are recognized from their `id`s.

# 4. EXAMPLES

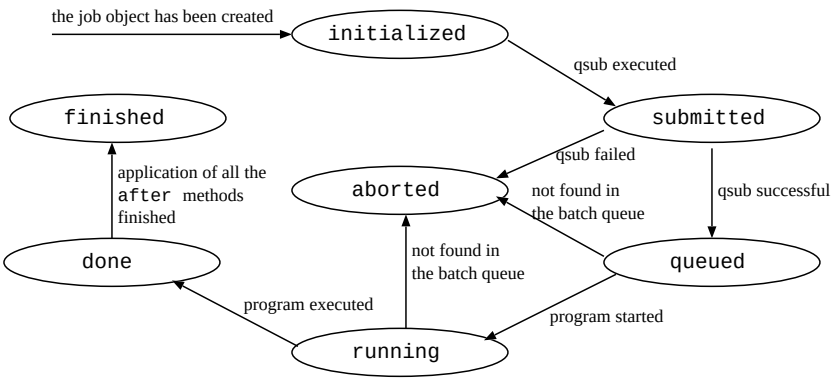


**Figure 4: State transitions of Xcrypt jobs.**

```
use base qw(limit core); #  use the  limit module
limit::initialize(10);

%template = (
 'id' => 'psweep',          #  job's ID
 'RANGE0' => [1..5000],     #  extraction range
 'exe0@'
  => sub {"./a.out input$VALUE[0] output$VALUE[0]"}
);
# prepare_submit_sync(%template);  is also allowed
@jobs = prepare (%template);
submit (@jobs);
sync (@jobs);
```

**Figure 5: Xcrypt script for parameter sweep.**

## 4.1 Parameter Sweep

### *4.1.1 User script*

Figure 5 shows a simple Xcrypt script that submits 5000 jobs that execute the single program `a.out`; each job uses a different command line arguments, and simultaneously running (or queued) jobs are limited to 10.

As the example shows, a typical Xcrypt script consists of the following components:

- preamble that specifies modules to be loaded, initializes some global parameters, and so on,
- definitions of job templates, and
- function calls to prepare for submitting jobs, submit the jobs, and wait for the jobs (job threads) to be finished.

Because the template has the member `RANGE0` with the value `[1..5000]`, `prepare` generates 5000 job objects with `ids` `psweep1`–`psweep5000`.

The command line to be executed in a job is defined by `exe@`. Because command line arguments (input and output file names) differ from job to job, the member name is postfixed by "@" and its value is a function. In the function body, `$VALUE[0]` binds the corresponding value in `RANGE0` (1–5000).

```
package limit;

use strict;
use NEXT;
use Coro::Semaphore;

my $smph;

sub initialize {
  $smph = Coro::Semaphore->new($_);
}

sub new {
    my $class = shift;
    my $self = $class->NEXT::new(@_);
    return bless $self, $class;
}

sub before {$smph->down;}
sub after  {$smph->up;}
```

**Figure 6: Definition of the `limit` module.**

### *4.1.2 Extension module*

The script in Figure 5 includes the `limit` module, the module for limiting the number of simultaneously running (or queued) jobs. This module requires end users to specify the limit number in the `limit::initialize`, which is in the second line in the script.

The `limit` module is implemented as shown in Figure 6. When this module is used, the semaphore is acquired before the submission of each job, and it is released after the completion of each job. The number of simultaneously running (or queued) jobs cannot exceed the number set by `limit::initialize` since job threads of excess jobs wait for acquiring a semaphore.

Figure 7 shows the execution flow of the script in Figure 5.

## 4.2 Optimal Parameter Search Using Bisection Method

Figure 8 shows the script for finding the solution of $f(x) = 0$ using the bisection method. One computation of $f(x)$ is done by one job execution. This is not a job involving "parallel" processing because only one job is executed in each iteration. However, Xcrypt is also useful for such sequential job processing.

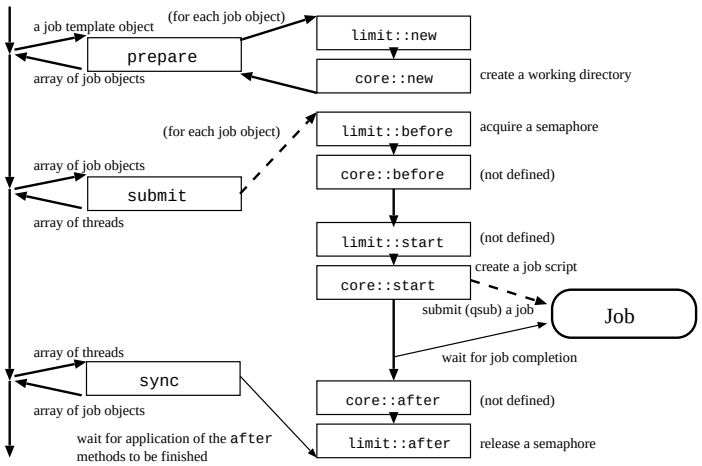


**Figure 7: Execution flow of the script in Figure 5 (broken lines indicate asynchronous executions).**

Note that this example represents a real application [6]: obtaining the amount of electrostatic charge $f(x)$ on the satellite when the amount of plasma discharge is $x$. The program is parallelized at the program level using MPI.

This script uses the `sandbox` module. If this module is loaded, then when `prepare` is invoked, a working directory for each job is automatically created and files labeled `copiedfile`$n$ are copied to the directory.

The process corresponding to iteration of the `until` loop is as follows.

1. Create a job object by calling `prepare`. `Prepare` also creates a working directory for the job and copies the two specified files to the directory.

2. Create an input file for the job by modifying the copy of `plasma.inp` in the working directory. This input file is a FORTRAN namelist. `KeyValueReplace` replaces the value of `wp(3)` ($x$) with the value of `$wpem`.

3. Submit the job and wait for it to be finished.

4. Extract the necessary part (the third column of the last line) of the output file to get the value of $f(x)$.

5. Exit the loop if $|f(x)|$ is sufficiently small. Otherwise, proceed to the next iteration after narrowing the search range.

In the next iteration, the template object used in the previous iteration can be used. However its `id` should be modified to give it a new `id`. Nevertheless, the job execution is skipped as Xcrypt judges the job to have been finished.

This example shows that Xcrypt is flexible. We can describe an execution flow using Perl's control flow constructs such as `if`, `until`, and `foreach`. Furthermore, Xcrypt's additional features enable us to glue job executions seamlessly as components of the execution flow.

The modules `Data_Generation` and `Data_Extraction` used in lines 2 and 3 are included in the library that we have developed as part of the Xcrypt system. Of course, we can use legacy Unix tools such as grep, sed, and awk to generate input files and extract data from output files for a huge number of jobs. However, it is not very easy for users who are unfamiliar with regular expressions to generate a large number of FORTRAN namelists that are slightly different from each other or to extract certain elements from an output file that represents a matrix. These generation/extraction libraries provide higher-level interfaces specialized for use in computational science, for example, for the purpose of modifying a FORTRAN namelist and extracting data from output files by specifying row and column numbers. In addition, these libraries support huge files (>Gbytes). While naïve implementations of text processing tend to load the entire input/output file onto memory, our libraries are designed to work with reasonable memory consumption through the use of a buffering mechanism.

# 5. PERFORMANCE

The overheads of job level parallelization with Xcrypt include the scheduling cost of a batch scheduler and the cost of Xcrypt's tasks such as creating job objects and generating job scripts. The scheduling cost depends on the implementation of the underlying batch scheduler. On the supercomputer of system Kyoto University with NQS, the total overhead is sufficiently low when the execution time of each job is longer than several tens of seconds.

# 6. ADVANCED FEATURES

This section shows some remarkable additional features of Xcrypt and extension modules we have implemented.

## 6.1 Remote Job Submission

Xcrypt provides a remote job submission facility. It enables us to run Xcrypt process not on a login node of the supercomputer system but on a local computer. When this facility is activated, Xcrypt makes a SSH connection to the login node and execute the `qsub`, `qdel`, and `qstat` commands on the login node via the connection. Although transfer of

```
# use the sandbox module
use base qw(sandbox core);
use Data_Generation;
use Data_Extraction;

%template = (
 'id' => 'bisec',
 'exe0' => 'mpiexec -n 64 '
          . './mpikempo3D 4 4 4 <plasma.inp',
 'JS_node' => 64,  # # of MPI processes
 'JS_cpu'  => 1,   # # of cores per process
 # files to be copied to the working directory
 'copiedfile0' => 'mpikempo3D',
 'copiedfile1' => 'plasma.inp',
);

my $wpe1 = 1.0; my $wpe2 = 1.5;
my $phi = 1;

# Iterate until the error becomes smaller than 0.1
until (abs($phi) < 0.1) {
  # The value of x
  my $wpem = ($wpe1+$wpe2)/2;
  # Update the job's id
  $template{id} = $template{id} . '+';
  # Create a job object
  my $job = prepare(%template);
  # Modify the input file
  my $generator = Data_Generation
      ->new ("$job->{id}/plasma.inp");
  $generator->KeyValueReplace ('wp(3)', $wpem);
  $generator->Execute();
  # Submit the job and wait for finishing it
  submit_sync($job);
  # Extract f(x) from the output file
  my $extractor = Data_Extraction
      ->new ("$job->{id}/result.dat");
  $extractor->LastLine();
  $extractor->GetColumn(3);
  $phi = $extractor->Execute();
  # Output f(x)
  print "Potential: ", $phi, "\n";
  # Narrow the range according to the value of f(x)
  if ($phi < 0) { $wpe1 = $wpem; }
  else { $wpe2 = $wpem; }}

print "When wp(3) = ", $wpem, ",\n";
print "phi = ", $phi, ".\n";
```

**Figure 8: Bisection method in Xcrypt.**

input/output files and execution files needs to be done manually, we provide handy APIs for it.

Figure 9 shows an example of using the remote job submission facility.

This facility is useful especially for systems in which running processes on a login node is limited by its system policy. You can also use this feature to write a single script that performs job parallel processing among multiple different supercomputers.

## 6.2 Spawn-sync Style Notation

Xcrypt allows us to write a script in a multithreaded language style as shown in Figure 10, which is almost equivalent to Figure 5. In this script, `spawn` is used instead of `prepare` and `submit`. The `spawn` submits a job to execute its body; the procedure of the body is serialized (using the same mechanism used for `before_in_job` and `after_in_job`) and invoked in a Perl process during the job execution. Note that the global variable `$i` can be referred to from the spawn block since it is also serialized and sent to the Perl process.

```
use base qw(limit core);
limit::initialize(10);

# define a remote login node
my $env1 = &add_host({
  'host' => 'user@supercom.kyoto.ac.jp',
  'sched' => 'KYOTO-LSF'});
put_into ($env1, 'input*.txt'); # send input files
put_into ($env1, 'a.out'); # send an execution file
%template = (
 'id' => 'psweep',
 'RANGE0' => [1..5000],
 'exe0@'
  => sub {"./a.out input$VALUE[0] output$VALUE[0]"}
);
@jobs = prepare (%template); submit (@jobs); sync (@jobs);
get_from ($env1, 'output*.txt'); # receive output files
```

**Figure 9: Xcrypt script using the remote job submission facility.**

```
use base qw(limit core);
limit::initialize(10);
foreach $i (1..5000) {
  spawn {
    system("./a.out input$i output$i");
  } (id => "psweep$i");
}
sync;
```

**Figure 10: Xcrypt script in a spawn-sync style notation.**

This feature is useful when you already have a Perl script and want to execute some part of procedures as jobs. Further examples will appear in [1].

## 6.3 Extension Modules

We implemented several useful extension modules besides `limit` and `sandbox`. They include the `dependency` module, which enables us to describe dependency among jobs declaratively. For example, let `$j1`, `$j2`, and `$j3` are job objects. Then we can write:

```
$j1->{depend_on} = [$j2, $j3];
```

to indicate that `$j1` must be executed after `$j2` and `$j3` are finished.

We also implemented the `dry` module for dry executions, the `bulk` module to combine multiple job objects and execute their bodies in a single job. In addition, we developed "algorithm modules" for divide-and-conquer, tree search, and the Monte Carlo method. These modules enable us to perform these algorithms only by defining application-dependent parameters.

# 7. CONCLUSION AND FUTURE WORK

This paper proposed a job level parallel script language named Xcrypt. Xcrypt relieves us from various annoying tasks such as writing job scripts for batch systems, generating/analyzing a huge number of input/output files, and managing states of asynchronously running jobs, and it can be made to run a wide variety of job level parallel processing by writing simple scripts.

Future work includes the evaluation of the performance and usability of the system by considering more practical and large-scale examples. We will also develop multilingualization support for Xcrypt to enable us to write scripts in languages other than Perl (e.g., Ruby and Python); this will involve the development of a foreign function interface (FFI) between Perl and other languages so that we can implement interfaces for various languages at low cost.

Xcrypt is now available at `http://super.para.media.kyoto-u.ac.jp/xcrypt`.

## 8. ACKNOWLEDGMENTS

This work was partly supported by a MEXT Grant-in-Aid for "Research and Development of Software for System Integration and Collaboration to Realize the E-Science Environment."

## 9. REFERENCES

[1] T. Abe and M. Sato. Auto-tuning of numerical programs by block multi-color ordering code generation and job-level parallel execution. In *The Seventh International Workshop on Automatic Performance Tuning (IWAPT2012)*, Jul 2012. (to appear).

[2] Cluster Resources Inc. Torque resource manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[3] P. Computing. Platform LSF: The HPC workload management standard. `http://www.platform.com/workload-management/high-performance-computing/lp`.

[4] Fujitsu, Inc. HPC middleware Parallelnavi. `http://jp.fujitsu.com/solutions/hpc/products/parallelnavi.html` (in Japanese), installed on the supercomputer system of Kyoto University.

[5] M. Lehmann. Coro: the only real threads in perl. `http://search.cpan.org/~mlehmann/Coro-5.372/`.

[6] Y. Miyake and H. Usui. New electromagnetic particle simulation code for the analysis of spacecraft-plasma interactions. *Physics of Plasmas*, 16(6):062904, 2009.

[7] National Institute of Informatics. *NAREGI Middleware WFT (GUI Workflow Tool) Users Guide*. `http://middleware.naregi.org/Download/Docs/UG-NAREGI-WFT-e.pdf`.

[8] K. Ohno, M. Matsumoto, T. Sasaki, T. Kondo, and H. Nakashima. An adaptive scheduling scheme for large-scale workflows on heterogeneous environments. In *Proc. Intl. Conf. Parallel and Distributed Computing and Systems*, Nov. 2009.

[9] G. Sarathy. Data::dumper: stringified perl data structures, suitable for both printing and eval. `http://search.cpan.org/~smueller/Data-Dumper-2.128/`.

[10] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *CLUSTER*, pages 421–429. IEEE, 2008.

[11] Sun Microsystems, Inc. The grid engine project. `http://gridengine.sunsource.net/`.

[12] H. Yuyama, T. Tsumura, and H. Nakashima. Construction of accurate task models for the MegaScript task parallel language. *IPSJ Trans. Advanced Computing Systems*, 46(12):181–193, Aug. 2005. (in Japanese).