# Xcrypt on Lisp: A Scripting System for Job Level Parallel Programming in Lisp

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

Masaru Ueno
Graduate School of Informatics, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
m.ueno@sys.i.kyoto-u.ac.jp

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

Motoharu Hibino
Graduate School of Informatics, Kyoto University
Sakyo Kyoto, JAPAN 606-8501
m.hibino@sys.i.kyoto-u.ac.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 this paper, we propose a Common Lisp front-end for Xcrypt. This enables us to write scripts to perform various job level parallel executions and pre/post-operations that generate/analyze inputs/results of jobs using various powerful features of Common Lisp such as list processing and REPL. We realized this front-end by implementing a RPC mechanism between Lisp and Perl processes which supports remote object references and calling unnamed functions defined in the other language. This implementation design can be applied easily when realizing Xcrypt fornt-ends for other languages other than Lisp.

## 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. We can consider parallelization not only at the *program level* but also the *job level*. The program level parallelization means parallelization *inside* a program using languages and/or libraries such as OpenMP, MPI, Cilk, Intel TBB, parallelization libraries for Lisp, and so on. On the other hand, job level parallelization means running a single with different parameters in parallel.

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 or almost impossible to write a single parallel program that runs efficiently using all the computing resourses. However, we can use such systems effectively by using two layers of parallelism at both the program level and job level. For example, in car body design, they have to try crash simulations plenty of times by executing a single simulation program with many sets of initial parameters. In this case, they can not only parallelize the simulation program to speed up each simulation computation but also run plenty of simulation computations in parallel. Parameter sweeps and optimal parameter searches for drug discovery and static software automatic tuning [1, 2] are other real examples of job level parallel processing.

In most of real supercomputer systems, including K Computer and the supercomputer in the Kyoto University, computing resources are managed by a batch scheduler such as NQS [3], LSF [4], SGE [5], or Torque [6]. In order to execute a program in such systems, we have to request the batch scheduler to execute it by *submitting a job* to a job queue. When the scheduler detects sufficient available computing resources to execute the job[1], it assigns the job to the

[1]The batch scheduler consider amount of free computing resources, a mount of resources the user can use, fairness among all the users of the system, and etc.

```
#!/bin/bash
#QSUB -ug mygroup
#QSUB -q myqueue
#QSUB -A p=1:t=1:c=1:m=3840M
./a.out 10
```

**Figure 1: An example of a job script for the supercomputer in Kyoto University.**

resources and execute it. Of course, we can submit multiple jobs simultaneously and they can be executed in parallel as far as computing resources remain.

Because we can impose management of computing resources on the back-end scheduler in such environments, it seems to be easy to write a job level parallel program in pre-existing script languages such as Perl, Ruby or Python, regarding a *job* in batch schedulers as a parallelization unit. However, there remain many tasks that are hard to implement, such as 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.

The most significant issue is to implement interfaces to various kind of batch schedulers. For example, in order to submit a job to execute "`./a.out 10`" to a job queue in the supercomputer in the Kyoto University, we have to prepare a script file called a *job script* as Figure 1, which describes queue and group names (depending on account information), amount of computing resources used by the job (memory size and a number of nodes, hardware threads, and cores), and shell commands to be executed. If this script is saved as "`aout.sh`," we can submit a job by a shell command "`qsub aout.sh`." If we want to execute `a.out` with another command line argument, we have to prepare another job script and execute `qsub` command again. To make matters worse, formats of job scripts are different among supercomputer systems. It is not difficult to implement a job script generator for each system, but the situation that each supercomputer user implements his/her own generator is undesirable in consideration of the overall productivity. Note that these issues are especially serious for computational scientists who are not familiar with script languages.

To solve these problems, we have developed a scripting system named Xcrypt [7], 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. Differences in interfaces among systems are handled by (Perl-based) configuration scripts, which are written separately from Xcrypt scripts. In addition, Xcrypt has a mechanism to add various useful features, such as limiting the number of simultaneously running (or queued) jobs, as modules.

At first, we developed Xcrypt based on Perl. However, some users would like to write such job level parallel execution in other scripting languages. Therefore we started a project to develop a multilingualization support for Xcrypt.

In this paper, we introduce a Common Lisp front-end for Xcrypt as a part of the multilingualization support. It enables us to write job level parallel executions with various powerful features of Common Lisp including higher-order functions, macros, and REPL.

We know that there are few Lisp programmers among supercomputer users and it will be hard to let them to use Lisp rather than other scripting languages such as Python, Perl, and Ruby. But we expect that Lisp programmers who are not familiar with supercomputers can perform large scale parallel computations easily using our system.

The contribution of this paper is as follows.

- We propose a scripting system to perform job level parallel computation on supercomputers, based on Common Lisp. It enables us to write various job level parallel executions easily using both the original Xcrypt features and various powerful features of Common Lisp.
- We designed and implemented a remote procedure call mechanism between Lisp and Perl, which is developed to realize the Lisp version of Xcrypt. Our RPC implementation supports callback functions and remote object references. This implementation can be used for the general purpose of using Perl features from Lisp and vice versa. In addition, this implementation design can be applied when realizing Xcrypt fornt-ends for other languages other than Lisp.

# 2. OVERVIEW OF XCRYPT

This section explains the important features of (the Perl version 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 realizes the core feature of job objects and must be included in the module list.

## 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 [8] 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.

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 `id`s by postfixing sequential numbers: for example, the return value of `prepare ('id'=>'example', 'RANGE0'=>[1..100])` is an array of job objects whose `id`s 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.

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$`]`).

Note that we provided the `RANGE` feature because a parameter sweep is one of the most typical usages of Xcrypt and there are many users who are not familiar with a programming style using map functions; since Perl provides the map function, (the Perl version of) Xcrypt users can write a parameter sweep using it.

### 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 `use`d 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 [9], 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 `use`d, 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 Implementation

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

**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 |

for communicating with jobs and the user interface and the other for monitoring the status of jobs.

When the `submit` function is called, Xcrypt generates a job script file as follows and execute the `qsub` command using it.

```
#!/bin/bash
#QSUB -ug groupname
#QSUB -q queuename
#QSUB -A amount of computing resources

inventory_write ID running
(Execute Perl to invoke before_in_job)
./a.out0 arg0_0 arg0_1 ...  # exe0
...
./a.outn argn_0 argn_1 ...  # exen
(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 been "running" or "done," by using some communication method (such as TCP/IP communication or creating a file in NFS). Then the Xcrypt process can detect the notification and update the job's status.

As mentioned in Section 1, Xcrypt have to generate job scripts in different format when it is executed in a different supercomputer system. Refer to [7] for how this issue is solved.

## 2.5 Example

Figure 2 shows a simple Perl-based Xcrypt script that submits 5000 jobs that execute the single program `a.out`; each job uses a different command line arguments.

Because the template has the member `RANGE0` with the value `[1..5000]`, `prepare` generates 5000 job objects with `id`s `psweep1`–`psweep5000`. The command line to be executed in a job is defined by `exe0`. 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).

```
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]"}
 'after_in_job'
  => sub { print "$_[0]->{exe0} is done."}
 'after'
  => sub { print "Job $_[0]->{id} finished."}
);
# prepare_submit_sync(%template);  is also allowed
@jobs = prepare (%template);
submit (@jobs);
sync (@jobs);
```

**Figure 2: Perl-based Xcrypt script for parameter sweep.**

```
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 3: Definition of the `limit` module.**

```
(xcrypt-init "limit" "core")
(xcrypt-call "limit::initialize" 10)

(setq jobs
  (prepare
   '((:id . "psweep")
     (:RANGE0 . ,(loop for x from 1 upto 5000
                   collect x))
     (:exe0@ . ,#'(lambda (tmpl &rest vals)
                    (format nil
                     "./a.out input~A output~A"
                     (nth vals 0) (nth vals 0)))
     (:after_in_job . (lambda (job &rest vals)
                        (format t
                         "~S is done."
                        (jobobj-get job "exe0"))))
     (:after . ,#'(lambda (job &rest vals)
                    (format t
                     "Job ~A finished."
                     (jobobj-get job "id"))))
     ))))

(submit jobs)
(sync jobs)
```

**Figure 4: Lisp-based Xcrypt script for parameter sweep.**

An array of job objects generated by the `prepare` function is passed to `submit` function and it submits all the jobs corresponding to the job objects in the list. Each submitted job is queued to a "job queue" (the batch scheduler has a queue for each user or user group). When sufficient computing resources to execute the queued job is available, the scheduler assigns the resources to the job. During each job execution, `a.out` is executed and then the Perl function set to `after_in_job` is invoked in a computation node. The message printed by `after_in_job` is stored in a "standard output file" created for each job by a batch scheduler. When each submitted job completed, the computing resources are released and the procedure set as the value of the member `after` is invoked. The message by `after` is printed to the standard output of the Xcrypt (Perl) process.

This script limits simultaneously running (or queued) jobs to 10 using the `limit` extension module. This module is implemented as shown in Figure 3. 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.

# 3. LISP FORNT-END FOR XCRYPT

## 3.1 Overview

In the Lisp-based Xcrypt, we can write a script equivalent to Figure 2 as Figure 4. We can translate a Perl script to a Lisp one straightforward. This script uses the Xcrypt built-in feature of `RANGE` to create 5,000 job objects and submit 5,000 jobs. However, Lisp programmers would like to write such an execution with a map function. Of course, we can write a script as Figure 5 with `mapcar` and `compose`.

Furthermore, we can write a script as Figure 6. The `jmapcar` function in this script submits a job for each element in the list given as the second argument. During each job execution, a Lisp process is run (on a computation node) to call

```
(xcrypt-init "limit" "core")
(xcrypt-call "limit::initialize" 10)

(setq jobs
  (mapcar (compose
           #'prepare
           #'(lambda (val0)
               (let ((id (format nil "pwsweep~A" val0)))
                 `((:id . ,id)
                   (:exe0 . ,(format nil
                              "./a.out input~A output~A"
                              val0 val0))
                   (:after_in_job
                           . (lambda (job)
                               (format t
                                "~S is done."
                                (jobobj-get job "exe0"))))
                   (:after . ,#'(lambda (job)
                                  (format t
                                   "Job ~A finished." id
                                   ))))))))
          (loop for x from 1 upto 5000 collect x)))

(submit jobs)
(sync jobs)
```

**Figure 5: Lisp-based Xcrypt script for parameter sweep using a map function.**

```
(xcrypt-init "limit" "core")
(xcrypt-call "limit::initialize" 10)

(jmapcar '(lambda (val0)
           (let ((exe0
                  (format nil
                    "./a.out input~A output~A"
                    val0 val0)))
             (run-shell-comand exe0)
             (format t "~S is done." exe0)))
         (loop for x from 1 upto 5000 collect x)
         :after #'(lambda (job val0)
                    (format t "Job ~A finished."
                            (jobobj-get job "id"))))
```

**Figure 6: Lisp-based Xcrypt script for parameter sweep using a job level parallel map function.**

```
CL-USER(1): (xcrypt-init "limit" "core")
CL-USER(2): (xcrypt-call "limit::initialize" 10)
CL-USER(3): (setq jobs (mapcar #'prepare ...))
CL-USER(4): (submit jobs)
CL-USER(5): (mapcar #'get-job-status jobs)
("finished" "finished" "aborted" "running" "running" "queued" ...)
CL-USER(6):
```

**Figure 7: Using Xcrypt in a Lisp Read-Print-Eval Loop environment.**

the function given as the first argument of `jmapcar` taking the element as an argument.

The definition of the function to be called in the job execution, its arguments, and its return values are serialized and sent to the computation node via generated lisp source files. Therefore we cannot use objects that cannot be serialized, such as function objects, streams, packages, and so on as the arguments and the return values. In addition, the environment (e.g., global variables) of the Lisp process invoking `jmapcar` cannot be referred to from a Lisp process on a computation node. Nevertheless this function is useful because we can write a script more simply.

We can use Xcrypt features in a Read-Eval-Print Loop (REPL) environment as Figure 7. We can submit jobs, check status of the submitted jobs, and kill them interactively. Although we also provide shell commands for these operations, we can manage jobs and their results more directly and intuitively in a REPL.

The lambda expressions in Figure 6 and in the values of `:after_in_job` in Figure 4 and Figure 5, which are to be serialized and evaluated in Lisp processes run during job executions, are not prefixed by "`#'`" but "`'`". This is because, unlike Perl, it is difficult to serialize a once interpreted (or compiled) function to a rereadable string.

## 3.2 Implementation

### *3.2.1 Overview*

The naïve solution to achieve multilingualization is to reimplement Xcrypt in the target language. But this solution is non-productive and it is hard to maintain the Xcrypt programs for all languages. So we did not employ it.

Another possible solution is to just implement wrapper functions for Xcrypt APIs such as prepare, submit, and sync that just perform remote procedure call to a Perl-based Xcrypt process. This is very easy, but in this solution we cannot use extension modules implemented in Perl. We also have to notice that Xcrypt APIs take various callback functions such as the member value of `after` in Figure 2. In this wrapper solution, we cannot define such procedures in the target language.

So we employed a third solution, to implement a general-purpose RPC mechanism between Perl and the target language that supports callbacks and remote object references. It takes more implementation cost, but we expect that once the design of the message protocol among Perl and other languages is fixed, we can easily implement supports for other various languages.

We implemented the following functions to achieve the Lisp version of Xcrypt.

- `(xcrypt-init &rest` *modules*`)`: runs a Perl Xcrypt process and make a TCP/IP connection to it. The Perl process first load the specified extension modules (defined in Perl) and listen a connection from the Lisp process. After the connection established, the Perl process executes a loop to handle messages from the Lisp process, which include RPC requests and return values of RPC from Perl to Lisp. The lisp process also creates a thread to handle messages from the Perl process.
- `(xcrypt-finish)`: finalizes the Perl process and close the connection to it.
- `(xcrypt-call` *fname* `&rest` *args*`)`: requests the Perl process to call the function specified by *fname* with the arguments *args*.

The arguments and return values from Lisp to Perl are serialized (translated into the JSON [10] format), sent to the message handler in the Perl process, and then deserialized. Numbers, strings, and symbols in Lisp are converted to numbers and strings in Perl. Booleans are converted into numbers that represents the corresponding booleans. Both lists and arrays are converted to arrays, except association lists are converted to hash objects.

When sending a function object, the Lisp process adds a pair of a generated function ID string and a reference to the function to a global function table, and then sends only the ID string. The message handler in the Perl process convert the function ID to a Perl function that requests the Lisp process to the call the function associated to the function ID. Thus, a Perl function that receives a callback function can call it without considering whether the function is a Lisp function or a Perl function.

Objects of the other types, which cannot be translated into the JSON format, such as streams, are converted into their printed representations.

Almost the same mechanism is applied when sending object from Perl to Lisp, except that, when a job object generated in Perl Xcrypt is sent to Lisp, only its job ID string is sent. The Lisp process converts the job ID into a proxy job object and it can access members of the job object in the Perl process via remote procedure calls such as `jobobj-get` in Figure 4. We also provide `jobobj-set` to update member values of job objects. We employed this implementation to avoid the consistency problem.

### *3.2.2 Implementation of Xcrypt tools in Lisp*

We can implement the functions `prepare`, `submit`, `sync`, `jobobj-get` and `jobobj-set` easily only by calling the corresponding built-in functions in Perl using `xcrypt-call`. In addition, we can implement `jmapcar` by using the `prepare`, `submit` and `sync` functions and the `:after_in_job` mechanism.

For example, the `prepare`, `submit` and `jobobj-set` functions in Lisp can be implemented as follows:

```
(defun prepare (tmpl)
  (xcrypt-call "builtin::prepare" tmpl))
(defun submit (jobs)
  (xcrypt-call "builtin::submit" jobs))

(defun jobobj-set (jobj field newval)
  (car (xcrypt-call "rpc::set" jobj field newval)))
```

where that the Perl function `set` is defined as follows:

```
package rpc;
sub set {
    my ($jobj, $field, $newval) = @_;
    return $jobj->{$field} = $newval;
}
```

When the `prepare` Lisp function is called, a job template (an association list) is serialized into a JSON object (an unordered collection of key-value pairs) and passed to a Perl process, and then it is deserialized into a Perl hash object. Then the Perl process calls the `prepare` Perl function with the hash object as the argument. If the job template passed to `prepare` in Lisp contains a function object, a pair of a function ID and a reference to the function is added to a global table. For example, when the argument value of the `prepare` in Figure 4 is passed to a Perl process, pairs corresponding to the values of `:exe0@` and `:after` are added to a global table. In our current implementation, a pair of *fn* and the return value of (`write-to-string` *fn*) is added to the hash table named `*function-table*`, where *fn* is a reference to a function object. Then a pair of the string and a tag indicating that the object is a Lisp function is sent to the Perl process, and it is translated to a Perl function which performs a remote procedure call to the Lisp function associated with the string, or *fn*.

In executing the `prepare` function in the Perl process, it needs to call the `:exe0@` function in the job template in order to decide the `:exe0` value of each job object being generated. In this example, the function is defined in Lisp but Perl can call it as if it is an ordinary Perl function because it is already translated to a Perl function that performs a remote procedure call.

When returning a list of job objects as the return value of the `prepare` to the Lisp process, each job object is translated to a pair of its ID string and a tag indicating that the object is a job object. Then a list of the pairs are serialized to a JSON array and deserialized to a list of proxy job objects (instance objects of the structure defined by `defstruct`).

A Lisp user can refer to member values of a job object using `jobobj-get` and `jobobj-set`. When a serialized proxy object is passed to the Perl process, its message handler find the job object corresponding to the ID and passes a reference to the object to a Perl function such as `set`[2].

When the Lisp process invoked `submit` with a list of proxy objects as the argument, the Perl `submit` function is called with a list of job objects and it generates 5,000 job threads as explained in Section 2.2.2. Calling the `:after` function in each thread works as well as calling the `:exe0@` function from `prepare` function. In order to invoke the `:after_in_job` function in a Lisp process run during a job execution, a job thread generates a job script that runs the following commands:

```
inventory_write psweepk running
./a.out inputk outputk
# Execute Lisp to invoke after_in_job
alisp -qq -s psweepk_after_in_job.lisp
inventory_write psweepk done
```

and generates following Lisp script as `psweep`$k$`_after_in_job.lisp`:

```
(defconstant +self+
  '(("id" . "psweepk")
    ("RANGE0" . (1 2 3 ... 5000))
    ("exe0" . "./a.out inputkoutputk")
    ...))

(defun jobobj-get (jobj memb)
  (cdr (assoc memb jobj :test #'equal)))
Definitions of other utility functions.

(let ((retval
       ((lambda (job &rest vals)
          (format t "~S is done."
                  (jobobj-get job "exe0")))
        +self+ k)))
  (with-open-file (o "psweepk_return"
                   :direction :output
                   :if-exists :append)
    (print (cons :after_in_job retval) o)))
```

The lambda expression in the last form is the copy of the value of `:after_in_job` in the job template in Figure 4. The job object is also serialized and translated to an association list in order to enable its member values to be referred to in the `:after_in_job` function using the `jobobj-get` interface. Note that the implementation of `jobobj-get` is different from the one in the Xcrypt Lisp process.

The return value of `:after_in_job` function is written to a text file so that a Lisp user can get it.

# 4. PRACTICAL EXAMPLE

Automatic performance tuning is one of the most significant applications of Xcrypt. Actually, we performed performance tuning for an electromagnetic field analysis program using the Perl version of Xcrypt. The target analysis program employs the well-known optimization technique called tiling. The program using this technique takes four performance parameters, the tile size $(x, y, z)$ and the number of tiling steps. The tuning space is too large to try all the combination by parameter sweep. So we employed parameter sweeps with limiting the search space step by step using heuristics. As the result, we got 25% better performance than hand-tuning [11].

We can write an Xcrypt script in Lisp for this automatic performance tuning as Figure 8[3]. In each parameter sweep step, trials of performance evaluations are executed in parallel and the best parameter set is selected from their results. In the Lisp version of Xcrypt, we can treat parameter sets

[2]The original Xcrypt implementation has the mechanism to find a job object from its ID employing a global job object table. Therefore we did not have to newly implement a mechanism such as a function table.

[3]The tuning in [11] performed more tuning steps but they are omitted because showing all the steps is not significant here.

and results more easily as lists. In addition, we can follow the progress of tuning by referring to `*results*` while the script is running in an REPL.

## 5. CONCLUSION AND FUTURE WORK

We proposed an extension to Lisp which enables Lisp users to write a wide variety of job level parallel processing employing computing resources of supercomputers. Using this system, we can not only write a parallel script employing execution files implemented in C or FORTRAN, but also modify an existing Lisp program so that some parts of procedures in the program are executed in parallel as jobs.

We can realize basic features of a front-end for an additional language only by implementing `xcrypt-call`, `xcrypt-init` and `xcrypt-finish` in the target language based on the message protocol explained in this paper; the implementation cost is not very high. Actually, we are now developing front-ends for other languages such as Ruby.

With the current implementation, we can write user script in Lisp but extension modules such as the `limit` module need to be implemented in Perl. We will extend the multi-lingualization support to enables us to use modules written in any language from a user Xcrypt script written in any language.

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

## 6. REFERENCES


[1] Seymour, K., You, H. and Dongarra, J.: A comparison of search heuristics for empirical code optimization., *CLUSTER*, IEEE, pp. 421–429 (2008).

[2] Abe, T. and Sato, M.: Auto-Tuning of Numerical Programs by Block Multi-Color Ordering Code Generation and Job-level Parallel Execution, *The Seventh International Workshop on Automatic Performance Tuning (IWAPT2012)* (2012).

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

[4] Computing, P.: Platform LSF: The HPC Workload Management Standard. `http://www.platform.com/workload-management/high-performance-computing/lp`.

[5] Sun Microsystems, Inc.: The Grid Engine project. `http://gridengine.sunsource.net/`.

[6] Cluster Resources Inc.: TORQUE Resource Manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[7] Hiraishi, T., Abe, T., Iwashita, T. and Nakashima, H.: Xcrypt: A Perl Extension for Job Level Parallel Programming, *Second International Workshop on High-performance Infrastructure for Scalable Tools WHIST 2012 (held as part of ICS'12)*, Venice, Italy (2012).

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

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

[10] Crockford, D.: RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON) (2006). `http://www.json.org/`.

[11] Hibino, M., Minami, T., Hiraishi, T., Iwashita, T. and Nakashima, H.: Automatic Performance Tuning of a Program with the 3D FDTD Method Using Xcrypt, *Annual Meeting of The Japan Society for Industrial and Applied Mathematics (JSIAM)* (2012). (in Japanese).

```
(xcrypt-init "limit" "core")
(xcrypt-call "limit::initialize" 16)

;; The list to store evaluation results
(defvar *results* (list))

;; Problem size
(defparameter *dom-x* 600) (defparameter *dom-y* 600) (defparameter *dom-z* 600)
(defparameter *nstep* 90)

;; Number of threads
(defparameter *nthread* 16)
;; Cache size per CPU socket [KB]
(defparameter *cache-size* 22784)
;; Cache size per node [KB]
(defparameter *cache-node* (* 2 *cache-size*))
;; Maximum number of grid points stored in cache
(defparameter *grid-max* (/ (* *cache-node* 1000) 52))

;; Submit a job to run evaluation program once and return the job object
(defun submit-fdtd (tile-x tile-y tile-z tile-step)
  (let ((output-file
         (format nil "result_~A_~A_~A_~A"
                 tile-x tile-y tile-z tile-step)))
    (prepare-submit
     `((:id . "jobfdtd")
       (:exe0 . ". /usr/Modules/3.2.9/init/bash")
       (:exe1 . "module load intel/12.1")
       (:exe2 . ,(format nil "export OMP_NUM_THREADS=~A" *nthread*))
       (:exe3 . ,(format nil
                  "mpiexec.hydra ./myfdtd_tiling2_2011_03_01_sysB.out ~A ~A ~A ~A ~A ~A ~A ~A~ > ~A"
                  *dom-x* *dom-y* *dom-z* *nstep* tile-x tile-y tile-z tile-step output-file))
       (:JS_cpu . ,*nthread*) (:JS_node . 1) (:JS_memory . "60G") (:JS_limit_time . "1:00")
       (:after . ,#'(lambda (job)
                      (push (list (get-elapsed-time output-file)
                                  tile-x tile-y tile-z tile-step)
                            *results*)))
       ))))

(defun product-if (test &rest sets)
  ;;  Make a product set removing an element that does not satisfy test.
  )

;; Step 1: find the optimal tile size assuming a tile is a cube
(sync (mapcan #'(lambda (tile-xyz)
                  (submit-fdtd tile-xyz tile-xyz tile-xyz 10))
              (loop for x from *nthread* to *dom-x* collect x)))
(setq *results* (sort *results* #'< :key #'first))
(defparameter *optimal-cube-size* (second (car *results*)))

;; Step 2: find the optimal number of tiling steps for the optimal cube size.
(setq *results* nil)
(sync (mapcan #'(lambda (tile-step)
                  (submit-fdtd *optimal-cube-size* *optimal-cube-size* *optimal-cube-size* tile-step))
              (loop for s from 1 to 30 collect s)))
(setq *results* (sort *results* #'< :key #'first))
(defparameter *optimal-tile-step* (second (car *results*)))

;; Step 3: find the optimal tile size (not assuming a tile is a cube)
(setq *results* nil)
(sync (mapcan #'(lambda (tile-size)
                  (submit-fdtd (first tile-size) (second tile-size) (third tile-size) *optimal-time-step*))
              (product-if #'(lambda (tx ty tz)
                              (<= (- (expt *optimal-cube-size* 3) (* *grid-max* 0.05))
                                  (* tx ty tz)
                                  (+ (expt *optimal-cube-size* 3) (* *grid-max* 0.35))))
                          (loop for x from *nthread* to *dom-x* by 16 collect x)
                          (loop for y from *nthread* to *optimal-cube-size*
                                by (floor (+ (/ *optimal-cube-size* 20) 0.5)) collect y)
                          (loop for z from *nthread* to 350 by 9 collect z))))
(setq *results* (sort *results* #'< :key #'first))

;; Print the optimal parameter set and the elapsed time with it
(print (car *results*))
```

**Figure 8: Lisp script for performance tuning of a FDTD program.**