**Journal of Information Processing Vol.0 No.0 1–16 (??? 1992)**

[DOI: 10.2197/ipsjjip.0.1]

**Regular Paper**

# Multilingualization Based on RPC for Job-Level Parallel Script Language, Xcrypt

Masaru Ueno[1,3] Tasuku Hiraishi[2] Motoharu Hibino[1,4] Takeshi Iwashita[2]
Hiroshi Nakashima[2]

Received: November 16, 2012, Accepted: April 30, 2013

**Abstract:** We are developing a job level parallel scripting language called Xcrypt, which helps us run a single program a number of times with different parameter values under computing environments where jobs are managed by a batch scheduler, such as supercomputers. Because Xcrypt is implemented as an extension to Perl, Xcrypt users are required to write Perl-based scripts. This becomes a barrier for programmers of other scripting languages when using Xcrypt. To solve this problem, we developed a framework that enables us to implement interfaces for using Xcrypt functionalities in other languages at a reasonable cost. To reuse existing Xcrypt implementations, we designed an RPC protocol between Perl and other languages. This protocol supports remote references to objects in another process, as well as callback functions. We can use Xcrypt APIs in any language by invoking Perl functions using RPCs. In our framework, Xcrypt extension modules, which Xcrypt users can implement as class extensions based on object oriented programming in Perl, can also be defined and used in any supported language. This paper also describes an example of a performance tuning of an electromagnetic field analysis simulation program performed with a Ruby-based parallel script that uses an extension module implemented in Lisp.

**Keywords:** parallel computation, scripting languages, domain specific languages, remote procedure calls, multilingualization

## 1. Introduction

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

Parameter sweeps for a car body design or, drug discovery are real examples of job-level parallelism; we often execute a single simulation program a number of times under different conditions. Job-level parallelism is also useful for the static automatic tuning of software by measuring the execution performance of a single program with various performance parameter settings [1], [2].

In most practical supercomputer systems, including the ACCMS supercomputer at Kyoto University and the "K Computer," computer resources are managed by a batch scheduler such as NQS [3], LSF [4], SGE [5], or Torque [6]. In such environments, we cannot run a user program by only executing a shell command; we need to write a job script in which the amount of computing resources and the commands to be executed are described, and submit the job to a job queue managed by the scheduler. The scheduler assigns the job to the resources and executes when a sufficient amount of available resources is detected for the job when considering the amount of free computing resources the user can utilize, the fairness among all system users, and so on. Of course, we can submit multiple jobs simultaneously, which can be executed in parallel as long as computing resources remain available.

We call the type of parallelism employing jobs in batch schedulers as parallelization units, "job parallelism." Because we can impose the management of computing resources onto the back-end scheduler in such environments, it seems easy to write job-level parallel programs in pre-existing script languages such as Perl or Ruby. However, many hard to implement tasks remain, such as generating job scripts for job submissions, and the management and synchronization of submitted jobs.

Furthermore, interfaces of batch schedulers, e.g., a shell command for submitting/checking/killing jobs, and a grammar for the job scripts, differ among batch scheduler implementations. Therefore, to reuse a job script in other systems, we need a mechanism for handling such differences. It is not difficult to implement such a mechanism, but a situation in which each supercomputer user implements his/her own mechanism is undesirable when considering of the overall productivity. We should therefore use a common programming environment.

[1] Graduate School of Informatics, Kyoto University
[2] Academic Center for Computing and Media Studies, Kyoto University
[3] Presently with Fujitsu Laboratories Ltd.
[4] Presently with Toyota Technical Development Corp.

© 1992 Information Processing Society of Japan

To achieve this, we developed a script language called Xcrypt [7], [8] for job-level parallel programming. This language is based on Perl, and various additional features are added to 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. Differences in system interfaces are handled by (Perl-based) configuration scripts. Because configuration scripts are written separately from Xcrypt scripts, end users do not need to be concerned about the differences in interfaces. In addition, Xcrypt has a resume function; even if submitted jobs or the Xcrypt process is aborted, we can restore the original state quickly. Furthermore, Xcrypt has a mechanism for adding various useful features, such as limiting the number of simultaneously running (or queued) jobs, as modules.

Because Xcrypt is implemented as an extension to Perl, users need to write Perl-based scripts as user scripts or extension modules. We believe Perl is a suitable base language because its grammar is similar to C, and it can be easily used by many computational scientists. However, this limitation is a barrier for programmers who are familiar with other script languages such as Ruby or Python. Furthermore, we often write scripts using typical features of languages other than Perl, such as interactive environments in Lisp or lazy evaluations in Haskell.

This paper proposes a mechanism enabling us to add an interface that allows end users to use Xcrypt features in other languages besides Perl. To reuse the existing implementation of Xcrypt for Perl, Xcrypt interfaces for a foreign language are realized using remote procedure calls (RPCs) between Perl and the foreign language. We also designed an RPC protocol called Xcrypt-RPC for such RPCs. This protocol employs JSON [9] as a data representation, and supports remote references of job objects and callback functions among different languages. A supported language can be added by implementing the Xcrypt-RPC for that language. Furthermore, we enabled extension modules for Xcrypt, which are implemented as class extensions in object-oriented Perl in the original Xcrypt, to be implemented in any supported language and used from user scripts in any supported language.

Note that we proposed a Common Lisp interface for Xcrypt in [10]. This current paper proposes a mechanism for adding support for languages other than Common Lisp, and for enabling us to define and use extension modules in other languages besides Perl.

The remainder of this paper is organized as follows. We describe Xcrypt in Section 2. In Section 3, we clarify the goal of this research and provide an overview of how to realize multilingualization for Xcrypt using RPCs. Based on this overview, we describe a user interface of the multilingualized Xcrypt in Section 4. We then provide the design details and implementation of Xcrypt-RPC in Section 5. We describe the performance evaluations in Section 6, and present other work related to this topic in Section 7. Finally, some concluding remarks and a description of future work are given in Section 8.

## 2. Xcrypt: a Job-Level Parallel Script Language

This section overviews the design and implementation of Xcrypt. In this paper, we do not describe the advanced features of Xcrypt, such as the handling of interface differences between systems or a fault resiliency support to restore the original state after a system failure. Refer to [8] for a description of these features and the details of the Xcrypt implementation.

### 2.1 User Scripts

#### 2.1.1 Declaration of Modules Used

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

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

*Module-name*$_1$ *module-name*$_2$ ... enumerates the extended modules used. Note that the `core` module realizes the core features of job objects, and must be included in the module list.

#### 2.1.2 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 job information. **Table 1** shows important keys that a template object should contain. The members `before` and `after` are defined as the preprocessing and postprocessing of each job, respectively. Their values are references of the function objects. The `before` function is invoked before submitting the job, whereas the `after` function is invoked after the job is completed.

When the input template object contains `RANGE`$n$ as its member, the `prepare` function 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 $k$-th object `id` is `example`$k$. When the values of the members `RANGE0`, `RANGE1`, ... of the template object are arrays with lengths of $n_1, n_2, \ldots$, respectively, $(n_1 \times n_2 \times \ldots)$ job objects are created.

We can ensure that the member values of the created job objects differ by postfixing `@` to the member names, such as `arg0_0@`, and we set a function to the corresponding member value; the member value of each created job object is the return value of the given function. In the function body, the assigned element of `RANGE`$n$ can be referred to as a $(n+1)$-

© 1992 Information Processing Society of Japan

**Table 1** Important keys 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 |
| RANGE$n$ | extraction ranges from the template |

th[*1] argument or by $VALUE[$n$].

The created job objects basically inherit the members of the template object and their values. In addition, the job objects have additional members, such as a member indicating the current state (e.g., running, completion) of a job. We can submit jobs by applying the Xcrypt function explained in the next section to job objects.

### 2.1.3 Submitting Jobs

We can submit jobs using the submit function as follows:

```
submit (@jobs)
```

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

The details of the submit function are as follows. The submit function creates a thread, called a job thread, for each job object, as explained in Section 2.1.2. The task of a job thread is as follows.

(1) Invoke the user-defined before function.
(2) Invoke all before methods defined in the declared modules in the use base, from left to right.
(3) Invoke the start method defined for the leftmost module among the modules used. The start method in the core module, which is invoked under usual settings, generates a job script, and submits a job by using the submission command provided by the underlying batch scheduler (e.g., /usr/bin/qsub).
(4) Wait for the submitted job to be completed.
(5) Invoke all after methods defined in the declared modules in the use base, from right to left.
(6) Invoke the user-defined after function.

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

Lightweight job threads are created using the Coro CPAN module [11], which enables us to create thousands of threads with a reasonable performance overhead.

### 2.1.4 Waiting for Jobs to Finish

We can wait for jobs to finish using the sync function:

```
sync (@jobs)
```

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

[*1] This misalignment occurs because the first argument is a reference to the job template object given to prepare.

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

**Fig. 1** Perl-based Xcrypt script for a parameter sweep.

## 2.2 Extension Modules

When only the core module is used, all job objects created by the prepare function are instance objects of the core class. Developers of Xcrypt libraries can extend Xcrypt by extending the core class.

The extension module implementation is based on the manner in which the class extension is carried out in object-oriented Perl programming. In addition, some method names in Xcrypt have a special meaning. For example, as explained in Section 2.1.2, methods called before, after, and start are used for extending the preprocessings, post-processings, and job submissions, respectively.

The declaration of the modules used by end users is as follows:

```
use base qw(module-name_1 module-name_2 ... core);
```

As explained in Section 2.1.1, this declaration lets the class for the job objects to have all the classes of the *module-name*$_1$ *module-name*$_2$ ... core as superclasses. We use the NEXT module [12] for enabling end users to write the modules used in the same place, and letting the *module-name*$_{k-1}$ class behave as a child class of the *module-name*$_k$ class.

Using the NEXT module, we can invoke method $m$ of *module-name*$_k$ by executing $*module-name*$_{k-1}$->NEXT::$m$(@*args*) in the body of the method $m$ of *module-name*$_{k-1}$ (if $m$ is not defined in *module-name*$_k$, the first defined $m$ of *module-name*$_{k+1}$, *module-name*$_{k+2}$, ... is called).

© 1992 Information Processing Society of Japan

```
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;}
```

**Fig. 2** Definition of the `limit` module.

### 2.3 Example

**Fig. 1** shows a simple Perl-based Xcrypt script for submitting 5000 jobs executing the single program `a.out`; each job uses a different command line argument.

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

Because the template has the member `RANGE0` with the value `[1..5000]`, the `prepare` function generates 5000 job objects with `id`s `psweep1`–`psweep5000`. The command line to be executed in a job is defined by `exe@`. Because the command line arguments (input and output file names) differ from job to job, the member name is postfixed by "@" and its value is a procedure. In the function body, `$VALUE[0]` binds the corresponding value in the range of `1` to `5000`.

An array of job objects generated by the `prepare` function is passed to the `submit` function, which submits all the jobs corresponding to the job objects in the list. Each submitted job is added to a job queue managed by a batch scheduler. When a queued job becomes executable, the scheduler assigns the resources to the job and executes it. Note that multiple jobs can run in parallel as long as the computing resources can be assigned. When each submitted job is completed, the computing resources are released, and the procedure set as the value of the `after` member is invoked. In this script, the message is printed to the standard output.

This script limits the simultaneous running (or queued) of jobs to ten using the `limit` extension module. This module is implemented as shown in **Fig. 2**. When this module is used, a semaphore is acquired before the submission of each job, and is released after the completion. The number of simultaneously running (or queued) jobs cannot exceed the number set by `limit::initialize` because job threads with excess jobs wait to acquire a semaphore.

## 3. Development Strategy

This section clarifies our goal and describes our reason for employing RPCs to achieve the multilingualization of Xcrypt. We then discuss the required specifications of the RPC protocol and how to implement it.

```
require "client.rb"
xcrypt_init "limit", "core"  #  Use the  limit module.

#  The number of simultaneous jobs is limited to 10.
xcrypt_call("limit::initialize", 10)

jobs = prepare (
  'id' => 'psweep_rb',       #  prefix of jobs' ID
  'RANGE0' => [*1..5000],    #  creates 5000 jobs

  #  The return value of the function is used as a command.
  'exe0@' => lambda|this, *args|
    "./a.out #args[0]"
  ,
  'after' => lambda |this|  #  executed after job completion
    puts "Job #get(this,'id') finished";

)

submit(jobs)                 #  submits jobs
sync(jobs)                   #  waits for completion of jobs
```

**Fig. 3** Ruby-based Xcrypt script for a parameter sweep.

```
(xcrypt-init "limit" "core") ;  Use the limit module.

;;  The number of simultaneous jobs is limited to 10.
(xcrypt-call "limit::initialize" 10)

(setq
 jobs
 (prepare
  '((:id . "psweep")          ;  prefix of jobs' ID
    (:RANGE0 . ,(loop for x   ;  creates 5000 jobs
                  from 1 upto 5000 collect x))
    ;;  The return value of the function is used as a command.
    (:exe0@ . ,#'(lambda (tmpl &rest vals)
                   (format nil
                    "./a.out input~A output~A"
                    (nth vals 0) (nth vals 0))))
    ;;  executed after job completion
    (:after . ,#'(lambda (job &rest vals)
                   (format t
                    "Job ~A finished."
                    (jobobj-get job "id")))))))

(submit jobs)                 ;  submits jobs
(sync jobs)                   ;  waits for completion of jobs
```

**Fig. 4** Lisp-based Xcrypt script for a parameter sweep.

### 3.1 Our Goal

Our goal is to enable developers to add an interface enabling end users to use the functionalities of Xcrypt in other languages than Perl with reasonable implementation costs.

We use the term "a supported language" to refer to a language that can use the functionalities of Xcrypt through an implemented interface, including Perl. That is, we can write user scripts and definitions of modules corresponding to Figs. 1 and 2 using the supported languages.

For example, after Ruby and Common Lisp are added as supported languages, we can execute user scripts such as shown in **Fig. 3** (Ruby) and **Fig. 4** (Common Lisp), and define the extension modules using scripts such as those provided in **Fig. 5** (ruby) and **Fig. 6** (Common Lisp). In addition, the extension modules defined in any supported language can be used in user scripts written in any supported language.

### 3.2 Implementation Strategy of Multilingualization

This section discusses how to implement the multilingualization of Xcrypt.

One strategy is to re-implement Xcrypt in other languages. The advantages of this strategy are a better perfor-

© 1992 Information Processing Society of Japan

```
###  list of public methods
### export: initialize initially finally

require 'thread'
require 'xcrypt_lib.rb'  #  tools for defining Xcrypt modules
include XcryptLib        #

module Limit
  module_function

  def initialize(n)
    @lock = Mutex.new
    @n = n
  end

  def initially(*args)
    while_cond = true
    while while_cond
      @lock.synchronize {
        if @n>0
          @n -= 1
          while_cond = false
        else
          Thread.pass
        end
      }
    end
  end

  def finally(*args)
    @lock.synchronize { @n+=1 }
  end

  def start(job)
    xcrypt_call_next(job) #  calls an ancestor method
  end
end
```

**Fig. 5** Definition of the `limit` module in Ruby.

```
;;  list of public methods
;; export: initialize initially finally

(require :process)

;;  defines a package
(defpackage "LIMIT_LSP"
  (:nicknames "LIMIT")
  (:use "CL" "CL-USER"))

;;  changes a current package
(in-package :limit)

;;  defines a semaphore and sets to a global variable
(defparameter *semaphore* (mp:make-gate nil))

;;  defines methos
(defun initialize (n)
  (setq *semaphore* (mp:make-gate nil))
  (loop repeat n
      do (mp:put-semaphore *semaphore*)))

(defun initially (self &rest vals)
  (mp:get-semaphore *semaphore*))

(defun finally (self &rest vals)
  (mp:put-semaphore *semaphore*))
```

**Fig. 6** Definition of the `limit` module in Lisp.

mance and the ability to use the libraries in each language effectively. However, both maintenance and implementation costs are high since we need to modify the implementations of all the supported languages when the Xcrypt specification is modified. Furthermore, since function calls among languages are not supported, we cannot use extension modules defined in another language, which is described as a requirement in Section 3.1.

Another possible strategy is to compile both the implementation code of Xcrypt and user scripts into object files and link these files together. This strategy is common in linking C and Fortran programs since the basic data structures and calling conventions used in these languages are the same. However, it is difficult to apply this strategy to script languages because each script language usually has its own execution mechanism. It is not impossible to implement a runtime system that knows the execution mechanisms of all of the supported languages such as "Perl for Ruby Module" [13], which is a runtime system created by linking Perl and Ruby interpreters. However, maintenance costs of such a system are high since we have to follow version upgrades of all the interpreters. Furthermore, the costs explode as the number of supported languages increases. Thus, this strategy is inappropriate for application in systems that should be maintained for a long period of time.

We therefore employed RPCs; we ran both the Xcrypt process and the processes of the supported languages in which the user scripts and extension modules are defined, and use the functionalities of these other languages and transfer the data among the processes using the RPCs. This strategy allows us to use the functionalities among the languages used without modifying each language implementation.

### 3.3 Required Specification and Implementation of the RPC Protocol

To achieve multilingualization of Xcrypt using RPCs, we designed a new RPC protocol called Xcrypt-RPC. This section discusses the required specifications of Xcrypt-RPC and shows the design of the protocol used to satisfy these requirements.

#### 3.3.1 Data Transfer

To design Xcrypt-RPC, we should first decide how the data are to be transferred among the languages when calling a function of another language and returning from it. Commonly, a pass-by-value or pass-by-reference is used to transfer data among the processes. For example, when transferring an array, all elements of the array are passed in the pass-by-value, and a pointer to the first element is passed in the pass-by-reference.

When all the processes employ a common representation for structural data, the call-by-reference is effective since we can transfer data flexibly in both directions. For example, we can update remote objects efficiently. However, data representations employed by most implementations of script languages differ from each other. In such a case, it is difficult to access elements in the structural data of other languages using references to the data. Another issue is that objects in most script languages are managed through garbage collection (GC); all data references among the processes must be managed in order to allow the collectors to collect the objects properly, which is difficult to implement.

The pass-by-value is easier to implement than the pass-by-reference. Differences in data representations among languages can be bridged by transferring the data using an intermediate data representation. All each language has to know is the method for serializing and deserializing the data representations in the language and the intermediate data

© 1992 Information Processing Society of Japan

representation; each language does not need to know about the data representations in the other languages.

We employ this implementation strategy and use JSON as the intermediate data representation. We employ JSON because there are pre-existing JSON libraries for many script languages including Perl, Ruby, and Python. JSON is so flexible that it can represent many types of commonly used structural data. In addition, since most Xcrypt functions do not have side effects to their arguments, most data transfers needed in Xcrypt are achieved using this transfer mechanism.

Although JSON can represent most structural data in simple descriptions, some types of data cannot be serialized to a JSON representation. For example, file handles and functions in Perl cannot be serialized, and it is difficult to transfer such data among different languages. However, for practicality, we need to transfer functions and job objects (Section 2.1.2) among languages. We therefore designed Xcrypt-RPC to support the transfers of these objects as follows.

- *A job object* contains information managed by a job management thread in an Xcrypt (Perl) process, such as the status of the job, and this information is updated asynchronously. Therefore, if we use the pass-by-value to transfer job objects, it is difficult to achieve consistency among replicas. A job object also contains data that are difficult to be serialized, such as a job thread. In addition, a job object sometimes contains a large amount of structurally-complex data, which require high copying costs. Therefore, we use the pass-by-reference to transfer job objects; when a transfer of a job object to a non-Perl language is requested, an ID string of the job object is sent instead of a serialized object. The non-Perl process that receives the ID creates a proxy object corresponding to the ID. This proxy object provides interfaces to refer to and to update the members of the job object stored in the Perl process, using RPCs.
- It is difficult to transfer *functions* among different languages using serialization or passing function pointers. However, since many Xcrypt functions such as `prepare` use functions as arguments to be invoked as callback functions, transferring functions using Xcrypt-RPC should be supported for practicality. We therefore implemented a mechanism for transferring functions among the languages. In this mechanism, when the function $f$ defined in language $L$ is passed to language $L'$, $f$ is translated into function $f'$ of language $L'$ that calls $f$ as an RPC.

#### 3.3.2 Extension Modules

As described in Section 2.2, extension modules of Xcrypt are defined as extended classes in Perl. However, it is difficult to define a Perl class in non-Perl languages and inherit the members of the superclasses among these languages. In Xcrypt-RPC, we enable users to define and use extension modules in non-Perl languages through the following mechanism. Inside a non-Perl language, only method bodies of an extension module are implemented; a *stub module*, written in Perl, is generated from the definition of the extension module written in a non-Perl language; and a class tree construction and mechanism for calling a method of an ancestor module are achieved using stub modules.

For example, defining a module in Ruby and using it from a user script can be achieved as follows. A stub module written in Perl is generated from the definition of the extension module written by a Ruby developer. The generated stub module file includes a definition of the Perl class and definitions of the methods whose names correspond to the names of the methods defined in the Ruby module file. The methods in the stub module are called *stub methods*. When a stub method is invoked, it simply calls the corresponding Ruby method using an RPC. To use the module from a user script, the script does not load the Ruby module file directly, but requests the Xcrypt process (Perl process) to load the stub module file. A method in the Ruby module can be called using the corresponding stub method in the stub module (if the user script is written in a non-Perl language, the stub method is called using an RPC).

Since the tasks of all the stub methods are the same, we can generate a stub module file easily from only the list of names of the public methods, the name of the generating stub module, and the name of the Ruby module. Instead of implementing a parser to extract the method names from a script, we let the end users write a list of method names explicitly, as the second line in Fig. 5, to simplify the interface implementations for additional supported languages.

In addition, we should support calling an ancestor from a method defined in Ruby. Xcrypt-RPC achieves this by requesting the stub method that calls the Ruby method to search for and call the ancestor method.

The main advantage of our implementation strategy for using the extension modules among different languages is that, since a user script can call a module method in any supported language through the corresponding stub method defined in Perl, each language process has to support RPCs only for Perl.

#### 3.3.3 Support for Multithreading Environments

To handle RPC request messages, each process of a supported language has a message handler thread. As described in Section 2.1.3, the `submit` function generates asynchronous threads to submit jobs (job threads). In such environments, another RPC request may occur during an RPC execution. Therefore, if we implement the message handler to execute a function when it receives an RPC request and waits for another request after the execution, a deadlock may occur since the handler cannot handle an RPC request while executing a function for a different RPC request. Therefore, in Xcrypt-RPC, the message handler generates a thread to execute a function, and sends a reply message with its return value when it receives an RPC request. The message handler can handle another RPC request just after generating the thread.

© 1992 Information Processing Society of Japan

Although Xcrypt-RPC allows for calling a function that generates asynchronous threads, such as the `submit` function, its protocol is designed to support only synchronous RPCs. We did not support asynchronous RPCs because we seldom need asynchronous processing in the original Xcrypt except in the `submit` function, and we believe that the demand for calling any function asynchronously would be small. If necessary, we can easily extend Xcrypt-RPC to support asynchronous RPCs.

#### 3.3.4 Namespace

One of the common problems we should solve in developing a multilingual programming environment is a namespace problem. That is, we need to decide what name is used to specify a method in an Xcrypt module defined in a non-Perl language. In multilingualized Xcrypt, we can specify a method, regardless of the language in which the method is defined, using a pairing of a public function name and the name of the extension module in which the method is defined. When the module is defined in a non-Perl language, the public function name and the extension module name directly correspond to the function name and the package name (class name) in the stub module in Perl, respectively. The mapping between the pair and function name in the non-Perl language can be defined by the developer. The developer also can specify which functions to be published as module methods that can be called using the RPCs.

Note that, for programmer convenience, standard Xcrypt API functions such as `prepare` and `submit` can be specified only by their function names. This specification is implemented by defining wrappers in each supported language.

## 4. User Interface

This section shows the external specification that end users or developers of extension modules refer to when they use Xcrypt in supported languages for which interfaces have been implemented using our multilingualization mechanism.

Although we use Ruby as a supported language to explain the specification herein, the specifications of the interfaces for other supported languages are given in the same manner, unless explicitly stated otherwise.

### 4.1 Writing User Scripts in Ruby

This section shows the external specification for writing Ruby-based Xcrypt scripts.

#### 4.1.1 Declaration of Used Modules and Running a Perl Process

A declaration of the modules to be included should appear at the beginning of the script, in the same manner as in Perl.

`xcrypt_init(`*module-name*$_1$`,`*module-name*$_2$`,...,"core")`

As in Perl scripts, the `core` package must be included in the module list. Note that we can specify a module only by its module name in *module-name*$_1$, *module-name*$_2$, ... regardless of the language in which the module is defined, including Perl.

#### 4.1.2 Creating Job Objects, Submitting Jobs, and Waiting for Jobs to Finish

As in Perl scripts, we can generate job objects, submit jobs, and wait for the submitted jobs to finish using the following function calls:

*jobs* `= prepare(`*template*`)`
`submit(`*jobs*`)`
`sync(`*jobs*`)`

We can translate Perl statements for these tasks into Ruby statements in a straightforward manner. For example, procedures defined as values of the `before` and `after` members, which are invoked through callbacks from the job threads generated by the `prepare` function during a Perl process, can be defined as general Ruby functions. However, since these tasks are invoked using RPCs, users need to note the data transfers, as explained in Section 4.2.

#### 4.1.3 Calling Perl Functions

We can call any function defined in a Perl process using a function call as follows:

`xcrypt_call(`*fname*`,*`*args*`)`

In addition, we can call public methods defined in the extension modules, such as the `initialize` method in the `limit` module, using `xcrypt_call`. As explained in Section 3.3.4, when specifying the module method, we do not need to be concerned about the language in which the extension module is defined; we can call a public method called *method* defined in module $M$ as follows:

`xcrypt_call("`$M$`::`*method*`",*`*args*`)`

regardless of the language in which the method is defined.

### 4.2 Transferring Data among Languages

As discussed in Section 3.3.1, arguments and return values in RPCs are transferred among languages using serializers. The types of data we can transfer depend on the specification of JSON, which we use as a serialized data representation; we can transfer `null`, booleans, numbers, strings, arrays, and objects (represented as lists of key-value pairs). In addition, we can transfer job objects and functions, which cannot be serialized into JSON objects, as follows. Job objects are transferred to non-Perl languages as proxy objects. For example, we can access members of a job object from Ruby using the proxy object, as **Fig. 7**. When transferring a function to another language as an argument or a return value of an RPC, the function is translated into "a function in the receiver language that calls the transferred function in the sender language using an RPC." Thus, in user scripts, we can call a function that is defined in another language and transferred to the process executing the script, as if it is a native function.

© 1992 Information Processing Society of Japan

```
jobs = prepare(template);
j = jobs.first

j.get("id")         #  returns "psweep_20130101"
j.set("id", "psweep_20130101_rev")
j.get("id")         #  returns "psweep_20130101_rev"
```

**Fig. 7** Handling a remote reference to a job object in Ruby.

### 4.3 Defining and Using Extension Modules

We can define extension modules in Ruby as follows:

(1) Write a definition of the extension module in Ruby as shown in Fig. 5. We can write definitions in Ruby in almost the same manner as in Perl; we can define a module by defining Ruby functions as new methods or extensions to existing methods for extending the functionalities of the job objects. In addition, we need to explicitly enumerate a list of names of the public methods in the script. In Ruby, we can write the list as a comment line as follows:

`### export:` $method_1$ $method_2$ ... $method_n$

(2) Generate a Perl stub module file (Section 3.3.2) from the Ruby module definition file. The generator is provided in the implementation of the Xcrypt interface for Ruby.

As described above, the defined module can be included by writing its module name in `xcrypt_init` or `use base`, regardless of the language in which the definition is written. An public method defined in the included module can be called using the `xcrypt_call` function as explained in Section 4.1.3.

As described in Section 3.3.2, we can call a method defined in an ancestor module from a body of the method defined in a non-Perl language, using the mechanism provided by the `NEXT` Perl module. For example, we can execute `xcrypt_call_next(*`*args*`)` in the body of a module method in Ruby to search for a method based on the module list written in `xcrypt_init` or `use base` and can call the first method found, in the same manner as a `NEXT` call in Perl (Section 2.2).

## 5. Specification and Implementation of Xcrypt-RPC

In this section, we explain the details of the Xcrypt-RPC specification and show our implementation of Xcrypt-RPC.

In Section 5.1, we provide an overview for how a user script written in a non-Perl language and including modules defined in non-Perl languages can be run. In Section 5.2, we explain the messages used in Xcrypt-RPC. We then show our implementation of Xcrypt-RPC in Perl in Section 5.3. Finally, in Section 5.4, we explain how to implement Xcrypt-RPC in a non-Perl language using Ruby as an example.

### 5.1 How a Script Runs with Xcrypt-RPC

**Fig. 8** shows how the user script `user-script.rb` written in Ruby runs using the extension modules `P`, `R`, and `C`, which are implemented in Perl, Ruby, and Lisp, respectively.

`P.pm`, `R.pm`, `L.pm`, and `core.pm` in the upper-center of Fig. 8 are the Perl module files. In particular, `R.pm` and `L.pm` are the stub modules (Section 3.3.2) generated from `R.rb` and `L.lisp`, respectively.

To execute the user script, we first run a Ruby process (the leftmost process in Fig. 8). The Ruby process then calls `xcrypt_init` at the beginning of the script body to run a Xcrypt (Perl) process. The Perl process first loads the `P.pm`, `R.pm`, `L.pm`, and `core.pm` modules. Next, `xcrypt_init` in the Ruby process makes a connection to the process and executes the rest of the user script.

A Lisp process is not run until the first call to a stub method defined in the stub module `L.pm`. A module list (`L.lisp` in this example) that the Lisp process needs to load upon start-up is given by the Perl process as its runtime options.

As explained in Section 3.3.2, calling a method in an ancestor module from a body of the method defined in a non-Perl language is achieved by requesting a stub method to search for and call the ancestor method. For example, calling an ancestor method from the method defined in `R.rb` is achieved as follows:

- For the stub method in `R.pm` to call the method in `R.rb` as an RPC, it generates an asynchronous thread, which performs an RPC to the Ruby method. As an additional implicit argument, the RPC message includes a callback function to send a request to the main thread in Perl (the thread that generated the asynchronous thread). The main thread waits for a message for requesting an ancestor method call or a message for notifying that the execution of the RPC is completed.
- When `xcrypt_call_next` (Section 4.3) is invoked in the body of the Ruby method, the Ruby process invokes the callback function to send a request message to the main thread in Perl. The main thread searches for and calls an ancestor method, and then returns the value to the Ruby process as the result of the callback.

The implementation code used in the stub methods for executing these tasks is shown in Section 5.4.

### 5.2 Messages in Xcrypt-RPC

Messages in Xcrypt-RPC are classified into "funcall" messages, "return" messages, and "finish" messages, which are used to send RPC requests, send the return values of the RPCs, and close the connections between processes, respectively.

Each message is a single line string that represents a single JSON object. The types of data that a JSON object can contain are restricted by the specification of JSON, and can be listed as follows:

- `true` and `false` (boolean),
- numbers (integer and floating number),
- strings,
- arrays,
- objects (unordered sets of key-value pairs), and
- `null`.

In particular, to achieve the transfer of functions and job objects among different languages as described in Sec-

© 1992 Information Processing Society of Japan

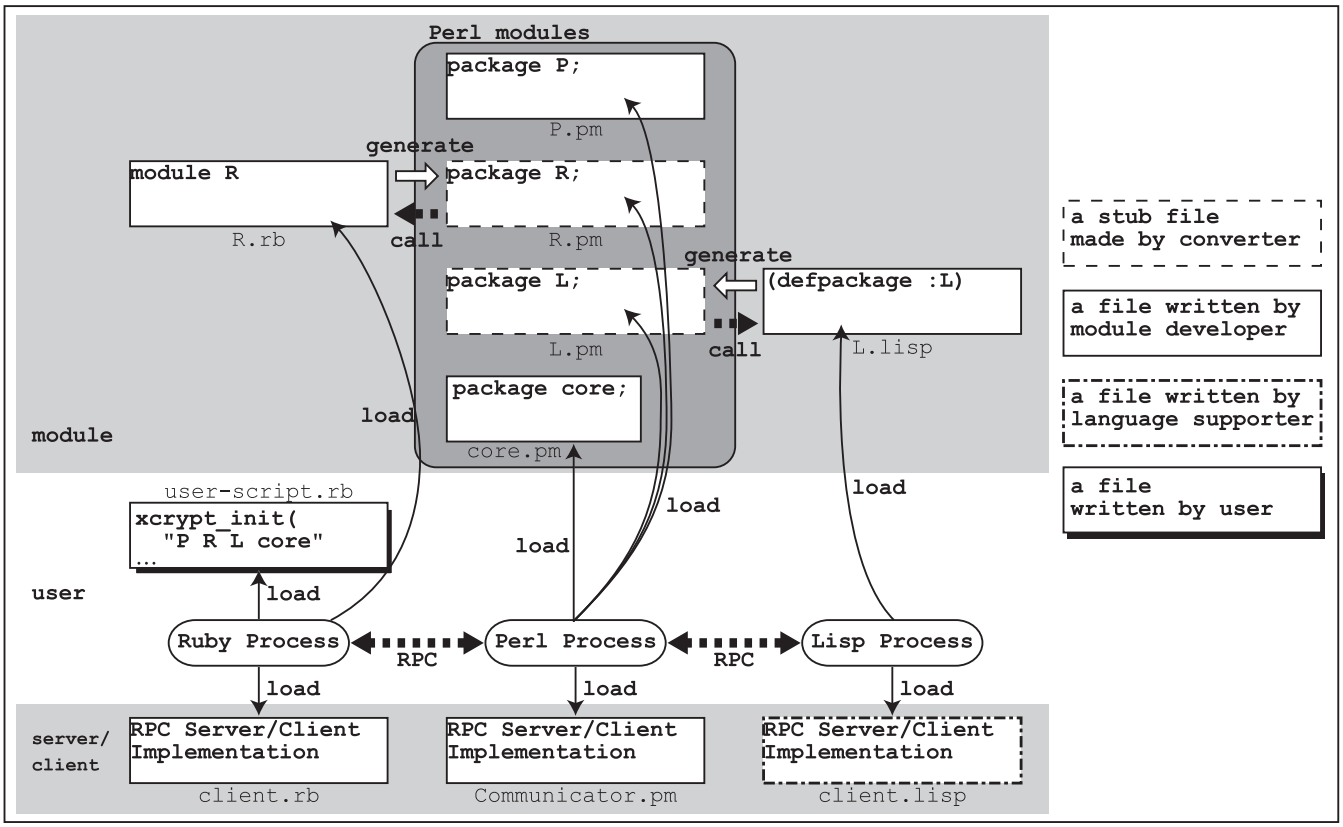



**Fig. 8** How the user script written in Ruby runs using the extension modules implemented in Perl, Ruby, and Lisp.

```
{
    "exec": "funcall",
    "function": "Limit::initialize",
    "args": 5,
    "thread_id": "#<Thread:0x8005acb4>",
    "next": {
        "type":"function/pl",
        "id":"CODE(0x8027d3e8)"
    }
}
```

**Fig. 9** An example of an Xcrypt-RPC message.

tion 3.3.1, objects that match one of the following patterns are treated as special objects.

- A function object:
  {`"type":"function/`*lang*`","id":"`*functionID*`"`}
  (*Lang* indicates the language in which the function is defined, e.g., `perl`, `ruby`, or `lisp`).
- A job object:
  {`"type":"job_obj", "id":"`*jobID*`"`}

For serializing the structural data during a language process, the data are traversed, replacing elements into JSON objects. In particular, functions and job objects are replaced with the special JSON objects described above. For deserializing JSON objects into data during a language process, the JSON objects are traversed, replacing elements into data in the native format. In particular, JSON objects that match the patterns described above are replaced with functions and job objects (proxy objects in a non-Perl language).

Each type of messages used in Xcrypt-RPC can be explained as follows:

- A *"funcall" message* is sent to request an RPC. This message is represented as a JSON object that contains the following keys:
  - `exec`: a string `"funcall"`;
  - `thread_id`: a string used to identify the thread (unique inside a process) that sent this "funcall" message;
  - `function`: a string or special JSON object used to specify the calling function;
  - `args`: an array of arbitrary length that contains arguments of the calling function; and
  - `next` (used only when this message is sent from a stub method used in a Perl process): a callback function (a special JSON object) for requesting the process to invoke an ancestor method (see Section 5).

  After a thread sents a "funcall" message, it waits until receiving a "return" message whose `thread_id` value is the same as that of `thread_id` of the "funcall" message. When a process receives a "funcall" message, it calls the function corresponding to the `function` value of the message. After the execution of the function finished, it sends back a "return" message with the return value.
- A *"return" message* is sent to notify a completion of an RPC, which is requested by a "funcall" message, and return its result value. This message is represented as a JSON object that contains the following keys:
  - `exec`: a string `"return"`;
  - `thread_id`: a string that is contained in the corre-

© 1992 Information Processing Society of Japan

sponding "funcall" message; and
- `message`: an array of arbitrary length that contains the return values of the RPC.

- A *"finish" message* is sent to close a connection between processes. When a process receives this message, it closes the connection to the sender and terminates the process. A "finish" message is represented as a JSON object that contains the following keys:
  - `exec`: a string `"finish"`; and
  - `message`: a string used to indicate the exit status.

**Fig. 9** shows an example of a "funcall" message.

### 5.3 Implementation of Xcrypt-RPC for Perl

#### 5.3.1 Message Handler

When a Perl process is run as an Xcrypt process, the process first loads the libraries for the Xcrypt functionalities and extension modules of Xcrypt. It then generates a thread for handling Xcrypt-RPC messages. Each time this thread receives a message, it deserializes the message and performs one of the following tasks according to the message type (the value of the `exec` key). For a "funcall" message, the message handler generates an asynchronous thread. This thread calls a function specified by the `function` value using arguments specified by the `args` values. After the execution of the function is finished, this thread sends a "return" message to the sender of the "funcall" message. The "return" message includes the return values of the function as its `message` values. The `thread_id` value of the "return" message is the same as the `thread_id` value in the received "funcall" message. For a "return" message, the message handler notifies the thread corresponding to the `thread_id` value in the message that the RPC is completed and sends the `message` values as results of the RPC. For a "finish" message, the message handler closes a connection and terminates the Perl process.

The reason for the message handler thread to generate a thread for a "funcall" message is described in Section 3.3.3.

#### 5.3.2 Implementation of RPC Functions

When `xcrypt_call(`*lang*`,` *function*`,` *args*`)` is executed during a Perl process, it sends a "funcall" message to the process corresponding to *lang*. The `thread_id` value in the "funcall" message is a unique string used to identify the thread that executed the function call. After sending this message, this thread waits for a "return" message that contains this string as the `thread_id` value. The return values of `xcrypt_call` are the `message` values in the "return" message.

The *args* values are serialized by employing an existing JSON library for Perl. While traversing the values, the functions and job objects are replaced with special JSON objects as explained in Section 5.2. When replacing a function into a special JSON object, the serializer adds a pairing of a generated function ID and a reference to the function into the global function table. Referring to this table, we can specify a function from a special JSON object. Although we need the same mechanism for job objects, too, we do not have to prepare a new table because the original Xcrypt implementation manages all the generated job objects using such a global table.

When the deserializer receives a special JSON object for a function or job object, it deserializes it as follows.

- If the special object represents a function, the following cases may apply:
  - if *lang* in the object is `perl`, the deserializer refers to the global function table and replaces the object with a reference to the function that corresponds to the *functionID* value in the object.
  - if *lang* in the object is not `perl`, the deserializer generates an unnamed function that performs an RPC to call the function specified by the object, and replaces the object with a reference to the unnamed function.
- If the special object represents a job object, the deserializer refers to the job table and replaces the object with a reference to the job object that corresponds to the *jobID* value in the special object.

A Perl process runs a process of another language when `xcrypt_call` calls a function in that language for the first time. When the Perl process runs a language process, it sends a list of modules that the new process should load, that is, a list of modules required by the user script and implemented in that language.

### 5.4 Implementation of a Ruby Interface

#### 5.4.1 Startup Processes

When a Ruby process is run, it first loads extension modules defined in Ruby, and then generates a thread for handling messages from a Perl process. We can implement the message handling in the same manner as in Perl.

#### 5.4.2 Implementation of RPC Functions

When `xcrypt_init(`*module-list*`)` is called during the Ruby process, it runs an Xcrypt (Perl) process and makes a connection to it. When the Ruby process runs an Xcrypt process, it sends *module-list* to Xcrypt, and lets Xcrypt know that it is being run by the Ruby process because Xcrypt should know that a Ruby process is already running.

The implementation of `xcrypt_call(`*function*`,` *args*`)` is almost the same as in the Perl version except that *lang* is not necessary because a Ruby process directly communicates only with an Xcrypt process.

#### 5.4.3 Generation of Stub Modules

As described in Sections 3.3.2 and 4.3, the definition of an extension module in Ruby includes the definition of the module name and implementations of the module methods, and we need to generate the definition of a stub module from the module definition in Ruby. The developer of the Xcrypt interface for Ruby has to implement the stub module generator.

The stub generator takes a module name in Ruby and a stub module name as runtime arguments and extracts a list of names of the public methods from "`### export: ...`" in a Ruby module file. It then generates a stub module file, as shown in **Fig. 10**, which is generated from the Ruby-based

© 1992 Information Processing Society of Japan

definition of the `Limit` module shown in Fig. 5.

In Fig. 10, we omit the method bodies except for the `start` stub method. However, the bodies of the other methods are the same as `start` except for the method name that appears in `Limit::start` and `$self->NEXT::start(@next_args)`. As described in Section 5.1, a stub method first generates a thread for performing an RPC to the corresponding method in Ruby (`$remote_thr`). The main thread then waits for a message requesting an ancestor method call or a message notifying that the execution of the RPC has been completed. The generated thread sends an RPC request by calling the `xcrypt_call_with_next` function, which is implemented as an extended version of `xcrypt_call`. As in `xcrypt_call`, `xcrypt_call_with_next` takes a language name as the first argument, a function name as the third argument, and argument values of the calling function as the remaining arguments. In addition, `xcrypt_call_with_next` takes a callback function for sending a message to the main thread in a stub method as the second argument. This callback function is set to `next` in the "funcall" message, and is called when `xcrypt_call_next` is called in the body of the Ruby method for requesting a calling of an ancestor method. When the main thread in the Perl process receives a `next_call` message through a callback function, it searches for and invokes an ancestor method using a `NEXT` call. The return value of the ancestor method is sent to the Ruby process as the return value of the callback function.

```
package Limit_rb;

#  A stub method
sub start {
  my ($self, @args) = @_;
  my $sig=new Coro::Signal;
  my $msg='';
  my @next_args, $next_ret;

  #  Generates an asynchronous thread for performing an RPC.
  my $remote_thr = Coro::async {
    my $remote_ret = user::xcrypt_call_with_next(
      'ruby',
      sub {  #  This function is set to next in a ''funcall''
             #  message.
        @next_args = @_;
        $msg='next_call'; $sig->broadcast();
        until ($msg eq 'next_ret') {
          $sig->wait;
        }
        return $next_ret;
      },
      'Limit::start', $self, @args);
    $msg='remote_ret'; $sig->broadcast();
    return $remote_ret;
  };

  #  The main thread waits for a message and calls NEXT::start
  #  when it receives a 'next_call' message.
  until ($msg eq 'remote_ret') {
    $sig->wait();
    if ($msg eq 'next_call') {
      $next_ret = $self->NEXT::start(@next_args);
      $msg='next_ret';  $sig->broadcast();
    }
  }
  return $remote_thr->join();
}

#  The following stub methods also can be generated
#  in the same manner as start .
sub initialize { ... }
sub initially { ... }
sub finally { ... }
```

**Fig. 10** The `Limit` stub module in Perl

## 6. Performance Evaluations

We executed two kinds of performance measurements to evaluate the performance impact of our RPC mechanism. In Section 6.1, we measure the overhead of the RPCs by employing a script where the overhead of the job submissions and scheduling overhead by a batch scheduler are removed. In Section 6.2, we then evaluate the impact of the RPC overhead during a practical use by employing a practical script that conducts actual job submissions.

### 6.1 RPC Overheads

To measure the overhead of the RPCs, we applied a dry execution module to a user script that executes $n$ jobs in parallel and waits for all the jobs to finish. Here, the dry execution module is an extension module used to change the behavior of the `submit` function to skip the job submission and wait for the completion of the jobs. Although this module is often used for debugging a script, we use this module to remove the impact of the job execution time, overhead of the job submissions, and scheduling overhead by the batch scheduler.

We implemented user scripts and the dry execution module in both Perl and Ruby, and measured the execution time of the following four scripts.

- Perl-Perl: a Perl-based user script that uses a Perl-based definition for the dry execution module.
- Perl-Ruby: a Perl-based user script that uses a Ruby-based definition for the dry execution module.
- Ruby-Perl: a Ruby-based user script that uses a Perl-based definition for the dry execution module.
- Ruby-Ruby: a Ruby-based user script that uses a Ruby-based definition for the dry execution module.

The user script and definition of the modules used for dry executions in Perl are shown in Appendixes A.1.1 and A.1.2, respectively. The user script and definition of the modules used for dry executions in Ruby are shown in Appendixes A.1.3 and A.1.4, respectively.

During the executions of scripts other than the Perl-Perl script, the RPCs are used as follows:

- an RPC for the `prepare_submit_sync` function in the Ruby-Perl and Ruby-Ruby scripts, and
- the following two RPCs for each job in the Perl-Ruby and Ruby-Ruby scripts:
  - an RPC for the `Dry::start` method, and
  - an RPC from the `Dry::start` method for updating the value of the `signal` member of the job object through its remote reference.

Thus, there are $2n$ RPCs in the Perl-Ruby script, 1 in the Ruby-Perl script, and $2n + 1$ in the Ruby-Ruby script.

The evaluation environment is as follows:

- CPU: Intel Core i7-640LM 2.13 GHz
- Memory: 2GB of PC3-8500 DDR3 SDRAM
- OS: Linux 3.5.3-1 (x86-64)
- Perl: perl 5.10.1

© 1992 Information Processing Society of Japan

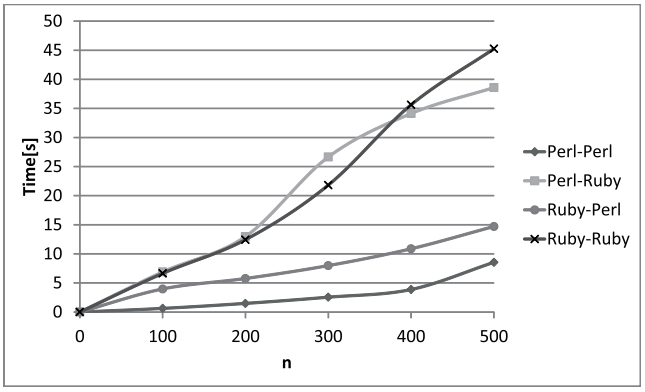



**Fig. 11** Execution times of the dry-run scripts.

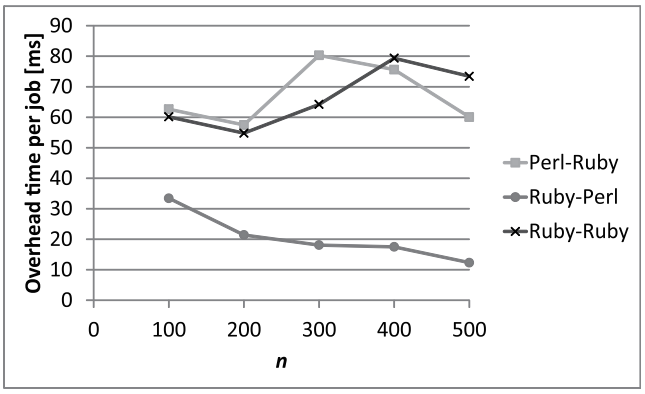



**Fig. 12** Relative execution times to the Perl-Perl script per job.

- Ruby: ruby 1.9.3dev

We changed the number of jobs ($n$) to 0, 100, 200, 300, 400, and 500 and took the average lengths of the execution time of ten trials for each setting. The measurement results are shown in **Fig. 11**. We can see that the execution time increases proportionally according to the number of jobs.

**Fig. 12** shows the relative execution times of the Perl-Ruby, Ruby-Perl, and Ruby-Ruby scripts to the Perl-Perl script per job. From this figure, we can see that the overhead of the RPCs per job is at most 80ms. Such overhead can be ignored because a job executed on supercomputer systems usually takes from minutes to days, and the scheduling overhead for each job is usually longer than a second.

### 6.2 Evaluation of a Practical Application

To evaluate the impact of the RPC overhead measured in the previous section for a practical use, we evaluated the performance when employing Xcrypt scripts for automatic performance tuning when actually submitting jobs.

Automatic performance tuning is one of the most significant applications of Xcrypt. Actually, we performed a performance tuning for an electromagnetic field analysis program using the Perl version of Xcrypt. The target analysis program employs a 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 combinations through a parameter sweep. We therefore employed parameter sweeps by limiting the search space step by step using the heuristics. As a result, we obtained a 25% better performance than hand-tuning [14].

Appendix A.1.5 shows a Ruby-based Xcrypt script equivalent to the Perl-based Xcrypt script we used in the performance tuning in [14]. Furthermore, this Ruby-based script uses the `limit` module implemented in Lisp (Fig. 6) to limit the number of simultaneously running jobs.

We compared the execution time of the Ruby-based script (which we call the Ruby-Lisp script) and the Perl script used in [14]. Since it takes too long to execute these scripts under the same conditions described in [14], we used a smaller problem size for the electromagnetic field analysis program, allowing each program execution to take about ten minutes, and limited the number of jobs to 59. We first submit 29 jobs and wait them to finish; we then submit another 30 jobs and wait for these jobs to also finish.

We used the Laurel supercomputer system at ACCMS, Kyoto University. This system is a Xeon E5 Sandy Bridge cluster, and employs the LSF batch scheduler [4]. We occupied a job queue where we can execute at least eight jobs at the same time[*2]. The versions of the language implementations we used for the performance evaluations can be summarized as follows:

- Ruby: ruby 1.9.3p194
- Perl: perl v5.12.5
- Lisp: Allegro Common Lisp 8.1

We executed the Perl and Ruby-Lisp scripts six times each. The results of the performance evaluations are shown in **Table 2**. The lengths of the execution times varied widely among the executions because of the variability of the scheduling overhead, or the lengths of the waiting time before the batch scheduler assigns jobs for execution.

A comparison of the average lengths of the execution times shows that the additional time of the Ruby-Lisp script to the Perl script is 66 ms per job. When we compare the minimum lengths of the execution times to minimize the impact of the scheduling overhead, the execution time of the Ruby-Lisp script is 5.2 s longer than the Perl version, and the overhead per job is 88 ms. This is shorter than 1% of the average job execution time, and we can conclude that the overhead of the RPCs is ignorable relative to the length of the job execution time.

## 7. Related Work

### 7.1 RPCs and Remote Object References among Different Languages

This section presents existing research and products for achieving calling methods among different languages. We employed JSON [9] as a representation of serialized ob-

[*2] Note that we do not submit more than eight jobs at the same time since we limit the number of simultaneously running jobs using the `limit` module. When we submit more than eight jobs simultaneously in the evaluation environment, the judgment of runability of the ninth and the following jobs competes with jobs of other uses in the supercomputer. We limited the number of simultaneously running jobs to remove such disturbances.

© 1992 Information Processing Society of Japan

**Table 2** Execution times of the scripts used for auto-tuning.

| | Perl [s] | Ruby-Lisp [s] |
|---|---|---|
| | 617.4 | 601.4 |
| | 731.4 | 858.1 |
| | 435.2 | 522.0 |
| | 455.8 | 440.4 |
| | 500.9 | 478.0 |
| | 707.5 | 571.5 |
| Average | 574.7 | 578.6 |

jects because it is highly readable, and we can easily implement RPC mechanisms for adding supported languages without describing type information in the Interface Description Language (IDL). In addition, we designed Xcrypt-RPC, which supports remote references to functions and job objects, which are not supported by JSON-RPC [15].

#### 7.1.1 XDR/ONCRPC (SunRPC)

The External Data Representation (XDR) [16] is a data serialization format developed by Sun Microsystems, which allows data to be transferred among different operating systems.

ONCRPC (Open Network Computing Remote Procedure Call), developed by Sun Microsystems as part of their Network File System (NFS), is an RPC system that uses XDR in a data serialization format and allows communication among different operating systems. In ONCRPC, we describe RPC interfaces, as well as the values and structures to be transferred. Since the RPC interface does not include type information, each node has to share the type information described in the IDL before starting a communication.

#### 7.1.2 DCE/RPC

Distributed Computing Environment / Remote Procedure Calls (DCE/RPC) [17] is an RPC system that enables us to efficiently develop software for large-scale distributed systems. The RPC protocol used in this system to achieve high level functionality, such as showing a distributed system as if it was a single node, is rather complicated.

#### 7.1.3 SOAP, XML-RPC

The Simple Object Access Protocol (SOAP) [18] is an XML-based protocol that allows network communication. The message format in SOAP is similar to a header and a body in HTTP; the header contains information necessary for communication, such as session information, and the body contains the main data. This protocol has the advantage of extensibility. However, the messages tend to be verbose. This protocol therefore does not fit our research since we mainly use communication among the processes in a single host.

XML-RPC is a simpler RPC protocol upon which SOAP is based. It employs XML for encoding and HTTP as a data transfer mechanism. We can use XML-RPC, but chose JSON because of the readability of the encoded data.

#### 7.1.4 JSON-RPC

JSON-RPC [15] is an RPC protocol that transfers data using HTTP or TCP/IP. As with Xcyrpt-RPC, it employs JSON as a serialized data representation.

**Fig. 13** shows an example of messages in JSON-RPC. An

```
Request
{
    "jsonrpc": "2.0", // JSON-RPC version 2.0
    "method": "subtract",
    "params": [42, 23],
    "id": 1
}

Response
{
    "jsonrpc": "2.0",
    "result": 19,
    "id": 1
}
```

**Fig. 13** Messages in JSON-RPC.

```
{ "compact" => true } #=> "\x81\xA7compact\xC3"
[1,2,3]               #=> "\x93\x01\x02\x03"
```

**Fig. 14** Messages in MessagePack.

RPC request is represented as an object that contains the members `jsonrpc`, `method`, `params` and `id`. A reply message for an RPC request is represented as an object that contains the members `jsonrpc`, `result` (when the RPC is successful), `error` (when the RPC fails), and `id`. We can pack multiple RPC requests into an array, and send the array for a batch process. In such a case, a reply message is also an array in which multiple reply messages are packed. JSON-RPC also supports multithreaded environments. However, it does not support transferring functions or remote references to objects.

#### 7.1.5 MessagePack-RPC

MessagePack [19] is a data representation format that supports the same data types as JSON (it also supports binary data not supported in JSON). Since data are represented in binary format, as shown in **Fig. 14**, the message size is smaller, and we can transfer data more efficiently. An RPC system employing MessagePack (MessagePack-RPC), which many languages support, was developed [20].

We expect to achieve a better efficiency using MessagePack, but chose JSON because we mainly use communication inside a single host, and for our purpose, the readability of the encoded data is more important than the communication performance.

#### 7.1.6 CORBA

The Common Object Request Broker Architecture (CORBA) [21] is a technique that allows the mutual exploitation of distributed objects among different operating systems and different languages. To serialize the objects, we need to describe the external interfaces of objects in the IDL.

CORBA is used for implementing a system enabling us to use Java RMI (remote method invocation) [22], API for sharing objects, and remote procedure calls among Java Virtual Machines, in non-Java languages [23].

#### 7.1.7 Other RPC Implementations

In large-scale mission-critical systems that need to handle large amounts of messages, the performance of a serializer and small amounts of encoded data are more important. There has been a considerable amount of research on satisfying such requirements, and many products have been

© 1992 Information Processing Society of Japan

developed, some of which support RPCs among different languages. Such products include Protocol Buffers developed by Google [24]; Avro, developed in a subproject of the Hadoop project [25]; Thrift, originally developed by Facebook and later as an Apache project [26]; and MessagePack-RPC. In these systems, with the exception of MessagePack-RPC, we need to define the schemas in IDL for transferring data.

To achieve multilingualization for Xcrypt, we mainly use communication inside a single host, and the serialization and communication performance is less important. JSON is therefore desirable from the viewpoints of readability, flexibility, and the fact that it is already supported in many script languages.

### 7.2 Process Containing Interpreters of Multiple Languages

The "Perl for Ruby Module" [13] is a Ruby module that enables us to call Perl functions from a Ruby script. This is achieved by creating an execution file by linking Perl and Ruby interpreters.

This system supports remote references to Perl objects using proxy objects, and calling Ruby functions from Perl using callback functions. However, unlike Xcrypt-RPC, the method used for calling a callback function from Perl differs from that of a Perl function.

While this strategy has a performance advantage, it does not fit our purpose because the maintenance costs of such a system are high since we have to follow version upgrades of all the interpreters. Furthermore, the costs explode as the number of supported languages increases.

### 7.3 Additional Strategies for Multilingualization

We often re-implement an existing functionality in other languages to enable us to use their functionality. For example, Catalyst [27] and Django [28] are re-implementations of Ruby on Rails [29] in Perl and Python, respectively. Although this strategy has the advantages of performance and usability, it has high implementation and maintenance costs.

On the other hand, we can implement a functionality as a dynamic-link library, and link an interpreter of the target language with this library. For example, using this strategy, cl-mpi [30] and the Ocaml/MPI Interface [31] allow us to use MPI in Common Lisp and Ocaml, respectively.

Note that neither strategy fits our purpose because we cannot achieve the use of extension modules among different languages.

## 8. Conclusion and Future Work

In this research, we developed a multilingualization mechanism for Xcrypt based on RPCs, which enables end users to use the functionalities of Xcrypt, such as job management and job submissions, in many languages. To achieve this, we also developed an RPC protocol that supports the transferring of functions and remote references to job objects, and a mechanism that allows us to use extension modules in any supported language from scripts in any supported language. Using our system, an end user who is unfamiliar with Perl can use the functionalities of Xcrypt naturally without have to learn Perl. We can also develop a script using the typical features of various languages. In addition, we show that the performance impact from the RPCs is ignorable in practical use.

Currently, the supported languages besides Perl include only Ruby and Common Lisp. However, we can add new supported languages easily referring to these existing implementations, which will be included in future work. In addition, Xcrypt has a feature that lets computing nodes execute part of the user scripts during a job execution. This is difficult to implement because we need to serialize bodies of the function definitions. However, we implemented them in Perl using a byte decompiler included as a standard Perl library, and in Lisp by letting the programmers quote the function definition. We will implement this feature for other languages including Ruby.

**Acknowledgments** This work was partially supported by JSPS KAKENHI Grant Number 22700030.

### References

[1] Seymour, K., You, H. and Dongarra, J.: A comparison of search heuristics for empirical code optimization., *CLUSTER*, IEEE, pp. 421–429 (online), available from ⟨http://dblp.uni-trier.de/db/conf/cluster/cluster2008.html#SeymourYD08⟩ (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] Platform Computing: 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., Miyake, Y., Iwashita, T. and Nakashima, H.: Xcrypt: Flexible and Intuitive Job-Parallel Script Language, *Symposium on Advanced Computing Systems and Infrastructures (SACSIS2010)*, pp. 183–191 (2010). (in Japanese).

[8] 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).

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

[10] Hiraishi, T., Ueno, M., Abe, T., Hibino, M., Iwashita, T. and Nakashima, H.: Xcrypt on Lisp: A Scripting System for Job Level Parallel Programming in Lisp, *Proceedings of the International Lisp Conference*, Kyoto, pp. 107–114 (2012).

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

[12] Conway, D.: NEXT.pm - Provide a pseudo-class NEXT (et al) that allows method redispatch. http://search.cpan.org/~flora/NEXT-0.65/lib/NEXT.pm.

[13] Masato, Y.: Perl module for Ruby Version 0.2.9, http://www.yoshidam.net/Ruby_ja.html#perl (2001). Retrieved 2012-12-05.

© 1992 Information Processing Society of Japan

[14] 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).
[15] JSON-RPC Working Group: JSON-RPC, http://www.jsonrpc.org/ (2012). Retrieved 2012-12-05.
[16] Eisler, M.: RFC 4506: XDR - External Data Representation Starndard (2006).
[17] The Open Group: DCE Portal, http://www.opengroup.org/dce/ (2012). Retrieved 2013-02-25.
[18] W3C Recommendation: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), http://www.w3.org/TR/soap12-part1/ (2012).
[19] Sadayuki, F.: MessagePack, http://msgpack.org/ (2012). Retrieved 2012-12-05.
[20] Sadayuki, F.: Specification of the MessagePack-RPC protocl (draft) and its Implementation, http://d.hatena.ne.jp/viver/20100406/p1 (2012). Retrieved 2012-12-05.
[21] Object Management Group, Inc.: CORBA, http://www.corba.org/ (2012). Retrieved 2012-12-05.
[22] Downing, T. B.: *Java RMI: Remote Method Invocation*, IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition (1998).
[23] Curtis, D.: Java, RMI and CORBA, http://www.omg.org/news/whitepapers/wpjava.htm (1997). Retrieved 2012-12-05.
[24] Google: Protocol Buffers, https://developers.google.com/protocol-buffers/ (2012). Retrieved 2012-12-05.
[25] The Apache Software Foundation: Apache avro, http://avro.apache.org/. Retrieved 2012-12-05.
[26] The Apache Software Foundation: Apache Thrift, http://thrift.apache.org/ (2012). Retrieved 2012-12-05.
[27] Catalyst Foundation: Catalyst — Perl MVC web application framework, http://www.catalystframework.org/ (2012).
[28] Foundation, D. S.: Django web Framework, https://djangoproject.com/ (2012).
[29] Rails Core Team: Ruby on Rails, http://rubyonrails.org/ (2012).
[30] Fukunaga, A.: cl-mpi: MPI bindings for Common Lisp (2009). http://code.google.com/p/cl-mpi/.
[31] Leroy, X., Gall, S. L. and Ozkural, E.: Ocaml/MPI Interface (2003). http://forge.ocamlcore.org/projects/ocamlmpi/.

# Appendix

## A.1 Scripts for Performance Evaluations

### A.1.1 Perl-based User Script for Dry Executions

```
use base qw(dry core);
use Time::HiRes;

my $n = 100;
print "($n jobs)\n";

my $start = Time::HiRes::time;
my %template = (
    'id' => 'job_dry_perl',
    'RANGE0' => [1..$n],  # generates n jobs
    'exe0' => 'dummy',
);
prepare_submit_sync(%template);
printf("%0.6f\n", Time::HiRes::time - $start);
```

### A.1.2 Perl-based Definition of the dry Module

```
package dry;
use core;

sub start {
    my $self = shift;
    $self->{signal} = 'sig_invalidate';
}

1;
```

### A.1.3 Ruby-based User Script for Dry Executions

```
require_relative "client.rb"
require_relative "lib/dry-rb.rb"
#  loads the module for dry executions
xcrypt_init "DryRb", "core"

n = 100
STDOUT.puts "(#njobs)"
start = Time.now
template =
  'id' => 'job_dry_ruby',
  'exe0' => 'dummy',
  'RANGE0' => [*(1..n)]      #  generates n jobs

prepare_submit_sync(template)
STDOUT.puts(Time.now - start)
```

### A.1.4 Ruby-based Definition of the dry Module

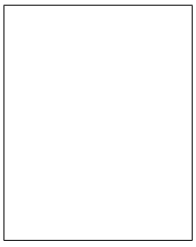
```
### export: start            # list of public methods
require_relative './xcrypt_lib.rb'
include XcryptLib

module DryRb
  def self.start(job)
    # Force the job completed using a remote reference to
    # the job object.
    job.set("signal", "sig_invalidate")
  end
end
```

### A.1.5 Ruby-based User Script for Auto-Tuning

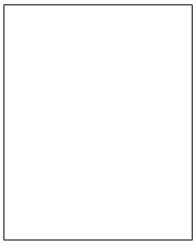
```
require "client.rb"
xcrypt_init "limit_lsp","core"

$result = [] # The list to store evaluation result
@jobid = 0

# Problem size
DOM_X = DOM_Y = DOM_Z = 300; N_STEP = 90

# Number of threads
N_THREAD = 8

# Cache size per CPU socket [KB]
CACHE_SIZE = 22784

# Cache size per node [KB]
CACHE_NODE = 2 * CACHE_SIZE

# Maximum number of grid points stored in cache
GRID_MAX = CACHE_NODE * 1000 / 52

OUTPUT_DIR = "result_\
#{Time.now.strftime('%Y-%m-%d_%H:%M:%S')}"

Dir.mkdir(OUTPUT_DIR) unless File.exists?(OUTPUT_DIR)

def submit_fdtd(tile_x, tile_y, tile_z, tile_step)
  output = File.join(OUTPUT_DIR, "result_%d_%d_%d_%d_%d" \
    % [DOM_X,tile_x,tile_y,tile_z,tile_step])
  template = {
    'id' => "jobfdtd_rb#{@jobid+=1}",
    'exe0' => "export OMP_NUM_THREADS=#{N_THREAD}",
    'exe1' =>
      "mpiexec.hydra ./fdtd_tiling.out %d %d %d %d \
%d %d %d %d > #{output}" \
    % [DOM_X, DOM_Y, DOM_Z, N_STEP,
      tile_x, tile_y, tile_z, tile_step],
    'JS_cpu' => N_THREAD,
    'JS_node' => 1,
    'JS_limit_time' => '1:00',
    'after' => lambda {|this|
      $result << [
        get_elapsed_time(output),
        tile_x, tile_y, tile_z, tile_step]
    }
  }
  prepare_submit(template)
end

def get_elapsed_time(file)
  ...
end

# product_if: Make a product set from *sets and then remove
# elements from it using the predicate defined as &block.
def product_if(*sets, &block)
  ...
end

time_start = Time.now
xcrypt_call("limit_lsp::initialize", N_THREAD)

# ---------------------------------
# Step 1 : determine OPTIMAL_CUBE_SIZE
# ---------------------------------
puts "step 1"
N_THREAD.step(DOM_X, 10) {|x| submit_fdtd(x, x, x, 10) }

sync

fastest_params = $result.min_by{|e|e.first}
OPTIMAL_CUBE_SIZE = fastest_params[1]

# ---------------------------------
# Step 2 : determine OPTIMAL_TILE_STEP
# ---------------------------------
```

© 1992 Information Processing Society of Japan

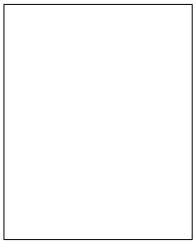
```
puts "step 2"
$result = []
1.upto(30).each do |s|
  submit_fdtd(OPTIMAL_CUBE_SIZE,
    OPTIMAL_CUBE_SIZE,
    OPTIMAL_CUBE_SIZE, s)
end
sync
fastest_params = $result.min_by{|e| e.first}
OPTIMAL_TILE_STEP = fastest_params.last
# --------------------------------
# Step 3: determin optimal tile size
# --------------------------------
puts "step 3"
$result = []
step_y = (OPTIMAL_CUBE_SIZE/20 + 0.5).floor
param_tx = N_THREAD.step(DOM_X, 16)
param_ty = N_THREAD.step(OPTIMAL_CUBE_SIZE, step_y)
param_tz = N_THREAD.step(350, 9)
test_sets = product_if(
  param_tx, param_ty, param_tz) {|tx,ty,tz|
  (OPTIMAL_CUBE_SIZE**3 - GRID_MAX * 0.05) <= tx*ty*tz \
    and tx*ty*tz <= (OPTIMAL_CUBE_SIZE**3 + GRID_MAX * 0.35)\
    ? true : false
}
test_sets.each do |tx,ty,tz|
  submit_fdtd(tx,ty,tz, OPTIMAL_TILE_STEP)
end
sync
fastest_params = $result.min_by{|e| e.first}
puts "**RESULT**\n tx: %f, ty: %f, tz: %f, ts: %f" \
  % fastest_params[1..4]
puts "elapsed time: #{Time.now - time_start} s"
```



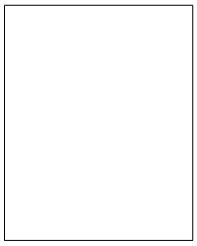
**Masaru Ueno** was born in 1989. He received a B.E. in Science in 2011 and an M.E. in Informatics in 2013, both from Kyoto University. He currently works for Fujitsu Laboratories Ltd.



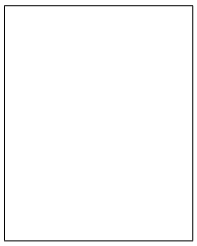
**Tasuku Hiraishi** was born in 1981. He received a B.E. in Engineering in 2003, an M.E. in Informatics in 2005, and a Ph.D. in Informatics in 2008, all from Kyoto University. In 2007–2008, he was a fellow of JSPS at Kyoto University. Since 2008, he has been working at Kyoto University as an assistant professor at the Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of IPSJ and Japan Society for Software Science and Technology (JSSST).



**Motoharu Hibino** was born in 1989. He received a B.E. in Information Science from Ritsumeikan University in 2011 and an M.E. in Informatics from Kyoto University in 2013. He currently works for Toyota Technical Development Corp.



**Takeshi Iwashita** was born in 1971. He received a B.E., an M.E., and a Ph.D. from Kyoto University in 1992, 1995, and 1998, respectively. In 1998–1999, he worked as a post-doctoral fellow of the JSPS project in the Graduate School of Engineering, Kyoto University. He moved to the Data Processing Center of the same university in 2000. Since 2003, he has been working as an associate professor in the Academic Center for Computing and Media Studies, Kyoto University. His research interests include high performance computing, linear iterative solver, and electromagnetic field analysis. He is a member of IEEE, SIAM, IPSJ, IEEJ, JSIAM, JSCES, and JSAEM.



**Hiroshi Nakashima** was born in 1956. He received his M.E. and Ph.D. degrees from Kyoto University in 1981 and 1991, respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a Professor at Toyohashi University of Technology in 1997, and a Professor at Kyoto University in 2006. His current research interests are the architecture of and programming in parallel systems. He received the Motooka Award in 1988 and the Sakai Award in 1993. He is a Fellow of IPSJ, and a member of IEEE-CS, ACM, ALP and TUG.

© 1992 Information Processing Society of Japan