ABCLib Working Notes No.9

ABCLibScript
User's Guide ver. 1.00

October 2004

Takahiro KATAGIRI

Department of Information Network Science
Graduate School of Information Systems
The University of Electro-Communications

"Information Infrastructure and Applications"
PRESTO, Japan Science and Technology Agency

Introduction

ABCLibScript is a scripting language (set of directives) with features that reduce the workload of developers of libraries with auto-tuning features. Specifically, it is a scripting language (set of directives) whose purpose is to auto-generate code required for auto-tuning by placing annotations in the source program, in order to reduce the workload of library developers.

Although the concept of a library with auto-tuning features is applicable to a wide range of processes, the current specification is limited to a language specification specialized for parallel-calculation processing.

1. Target Languages

ABCLibScript is a scripting language (set of directives) designed in order to make developing parallel-computation libraries more efficient*. Consequently, it must have an interface from a language that is suited to numerical calculation and parallel processing. The purpose of ABCLibScript is the generation of executable program code.

For these reasons, it is assumed that ABCLibScript will be used to generate program code that will run in an environment in which Fortran90 is available as a language for numerical calculation, and MPI (Fortran interface) is available as a language for parallel processing.

1.2 Sequence of Library Development and Use

Fig. 1 illustrates the sequence of library development using ABCLibScript, and the use of mathematics libraries with auto-tuning features by the user.

* The language is not limited to library development. It can be used to optimize any user program.

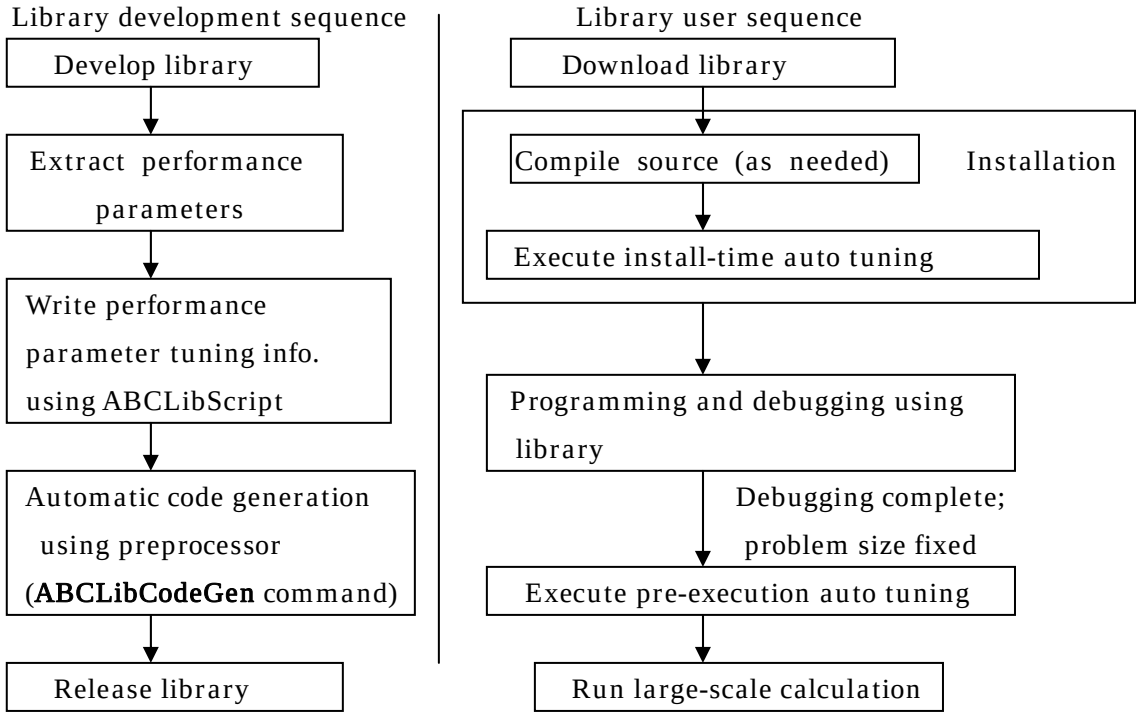


Figure 1. Sequence of Actions of Library Developers and Users Using ABCLibScript

1.3 Features of Programming with ABCLibScript

ABCLibScript was designed with the following four points in mind.

(1) Auto-tuning specification function specialized for numerical calculation

Auto-tuning features specialized for numerical calculations are provided. They facilitate the creation of numerical calculation libraries with auto-tuning features.

(2) Tuning specification method whereby the library developer inserts instructions into the program in the form of annotations

The addition of auto-tuning features is performed in the form of annotations by the library developer. Consequently, this makes it easy to specify auto tuning, while not interfering with the compilation of the original code.

(3) Provision of a preprocessor that parses the library developer's annotations and automatically generates code

This feature parses the commands written by the library developer via annotations, and automatically generates code with added auto-tuning processing. This allows

the library developer to manage library source code itself, with an auto-tuning framework automatically added on. It is also possible to understand what processes have been added by viewing the automatically generated source code. The makes the processing highly transparent and worry-free.

(4) Adoption of the concept of two types of parameters that affect performance: performance parameters (PP) and basic parameters (BP)
The adoption of the concept of performance parameters and basic parameters improves the outlook for auto-tuning processing. It also makes it easy for library developers to communicate their intentions to the system.

1.4 The ABCLibScript Software Architecture

The ABCLibScript Application Programming Interface (API) consists of the following three elements:

- Specifiers defining auto-tuning targets, and subtype specifiers defining auto-tuning process details
- Environmental parameters defining auto-tuning information (system parameters)
- Run-time routines that supplement auto tuning

Fig. 2 illustrates the relationship between these elements.

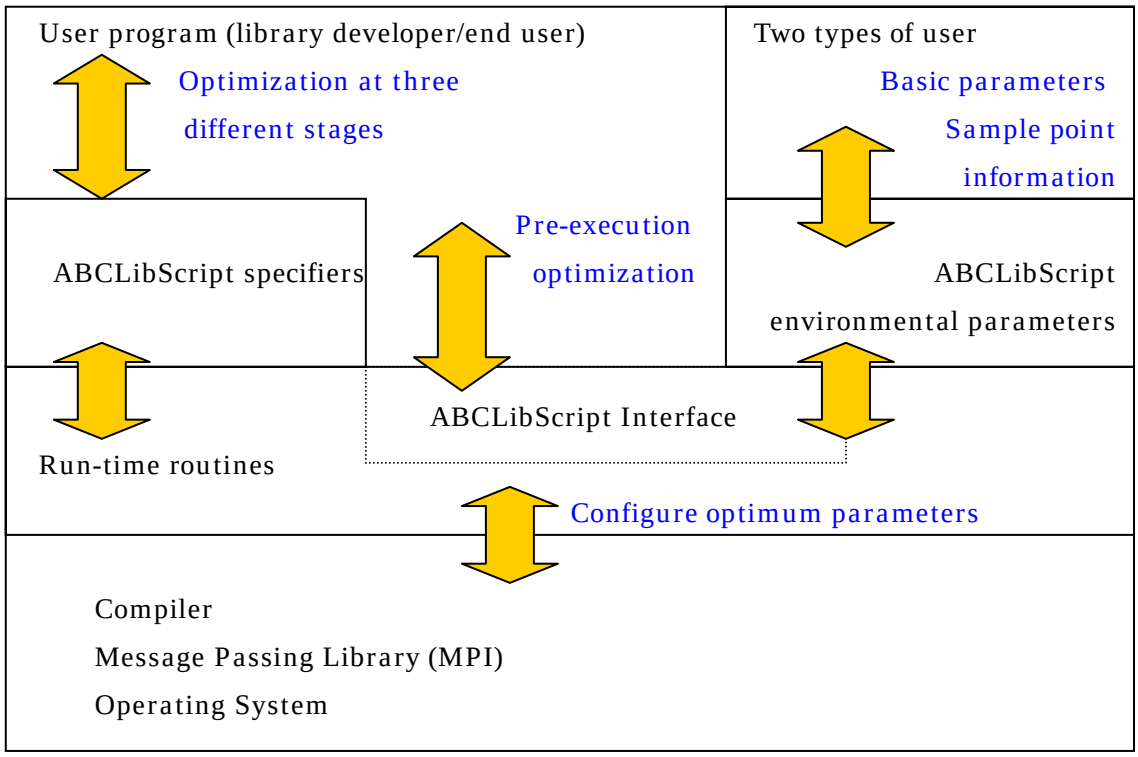


Figure 2. The ABCLibScript Software Architecture

2. Getting Started

2.1 Auto Tuning Features Envisioned by the System

The auto-tuning features provided by ABCLibScript can be classified into the following three categories:

- **(1) Install-time auto tuning**

  This feature automatically tunes performance parameters that can be determined when the library is installed.

- **(2) Pre-execution auto tuning**

  This feature auto-tunes performance parameters after the end user has fixed the problem size and other basic parameters.

- **(3) Run-time auto-tuning**

  This feature automatically tunes performance parameters that can be determined when the library is actually called from within a program.

These auto-tuning features are implemented as subroutines based on the Framework of Install-time, Before Execute-time and Run-time optimization feature (FIBER; patent application filed January 2003) software architecture, which is an auto-tuning library architecture proposed by the ABCLib project.

Fig. 3 shows an overview of FIBER. With the exception of the auto-modeling routine shown in Fig. 3, ABCLibScript is a scripting language (set of directives) developed with the aim of providing the three above-mentioned auto-tuning features.

Fig. 4 illustrates the parameter information hierarchy. In other words, parameters determined by the install-time auto-tuning routine can be referenced by the pre-execution auto tuning and run-time auto tuning routines. Meanwhile, the parameters determined by the pre-execution auto-tuning routine can only be referenced by the run-time auto-tuning routine. Finally, the parameters determined by the run-time routine basically cannot be referenced by any other routine except the run-time routine[†].

[†] An exception, however, is in the FIBER feedback model, where it is possible for the pre-execution routine to reference parameters optimized by the run-time routine, and optimize them.

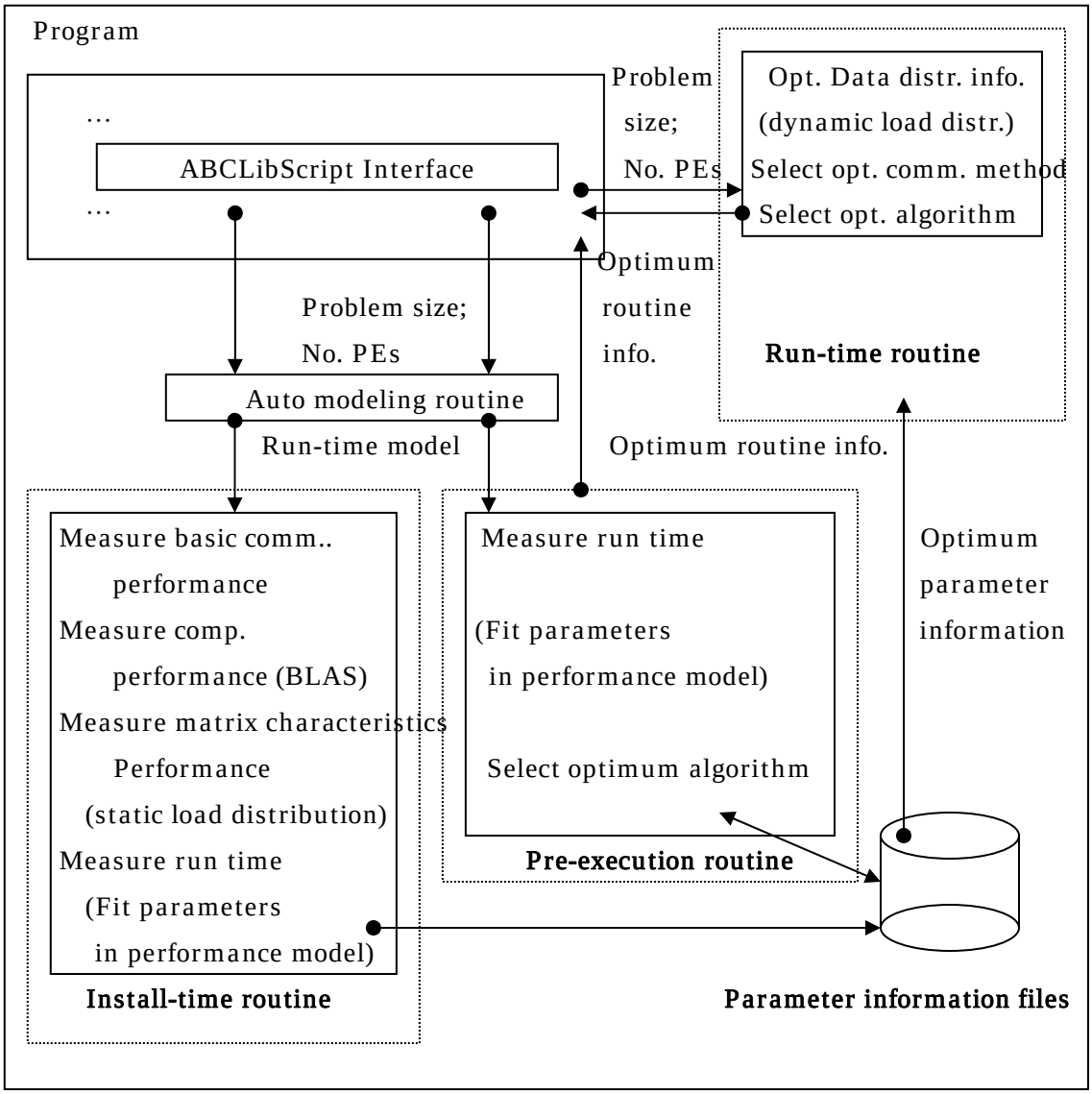


Figure 3. Structural Diagram of Auto Tuning Library in ABCLib Project (FIBER Software Architecture)

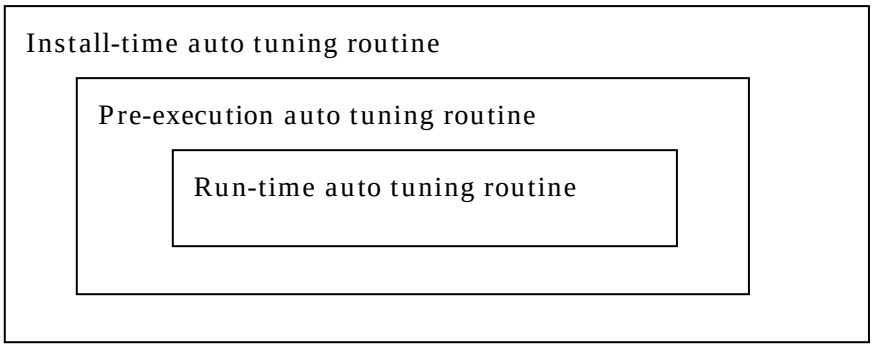


Figure 4. Hierarchy of Parameter Information Reference

2.2 Execution Order of ABCLib's Auto Tuning

ABCLib executes auto-tuning in the following order.

(1) Install-time auto tuning routine
(2) Pre-execution auto tuning routine
(3) Run-time auto tuning routine

Note that if the execution sequence is other than the above, an error code will be output, and auto-tuning will stop.

2.3 Parameter Types

FIBER tunes the following two types of parameter in libraries, subroutines, and a portion of programs developed by users (library developers and end users).

- **Basic parameters (*BP*)**
  These parameters must be set when the end user uses the library or the like. Some examples of BPs include matrix size and number of processors.

- **Performance parameters (*PP*)**
  These parameters are not absolutely necessary for the end user to utilize the library, but do impact performance. One example is the number of unrolling levels. The user guarantees that the optimum values for performance information parameters can be found if the basic parameters are set.

ABCLibScript is a scripting language (set of directives) developed to make it easy for library developers to specify the above-mentioned FIBER basic parameters and performance parameters. Consequently, one could say that the ABCLibScript notation described below is a scripting language for declaring, defining, and specifying performance parameters and basic parameters.

2.4 Notation

2.4.1 Overview

With ABCLibScript, auto-tuning features are achieved by coding processes in the form of annotations in the source program. Specifically, lines beginning with

```
!ABCLib$
```

are considered to be instructions to ABCLibScript.

Briefly, the notational conventions are as follows:

```
!ABCLib$ <auto-tuning type> <feature name> [(target param.)] region start
[ !ABCLib$ <feature details> [ sub region start ] ]
        Program targeted for processing
[ !ABCLib$ <feature details> [ sub region end] ]
!ABCLib$ <auto-tuning type> <feature name> [(target param.)] region end
```

The <auto-tuning type> and <feature name> above are called **specifiers**. The <feature details> above are called **subtype specifiers** .

Finally, the processing surrounded by !ABCLib$ region start … !ABCLib$ region end is called the **tuning region** (**AT region**).

2.4.2 Specifiers

The ABCLibScript specifiers are described in detail below.

List of specifiers

<auto-tuning type>::= (**install** | **static** | **dynamic** | <formula>)

install: Specifies install-time auto tuning

static: Specifies pre-execution auto tuning

dynamic: Specifies run-time auto tuning

<formula>::= Indicates a formula conforming to Fortran90 syntax

<feature name>::= (**define** | **variable** | **select** | **unroll**)

define: Specifies that this process sets a parameter

variable: Specifies that this is a variable parameter

select: Specifies that this process selects from multiple options

unroll: Specifies that the following process performs loop unrolling

2.4.3 Subtype Specifiers

The specifier <feature name> may require a further annotation. What is used to encode this is the subtype specifier. The following <feature details> subtype specifiers are available.

<feature details>::= (**name** | **parameter** | **select** | **according** | **varied** | **fitting** | **number** | **prepro** | **postpro**)

The <feature details> subtype specifiers that can be used depend on the specifier. This is shown below.

Specifier	Available subtype specifiers
define	name, parameter, number, prepro, postpro
variable	name, parameter, varied, fitting, number, prepro, postpro
select	name, parameter, select, according, number, prepro, postpro
unroll	name, parameter, varied, fitting, number, prepro, postpro

Details about each subtype specifier are shown below.

List of subtype specifiers

- **name** <string>: Denotes the name of the tuning region
  (available for all functions)
- **parameter** (<attribute specification> <parameter name>
  [<attribute specification> <parameter name>,...])
  Specify a parameter to output to a performance characteristics file, or input from a performance characteristics file

  <attribute specification>::=[**in** | **out** | **bp**]
  in: Input (defined externally and referenced) parameter
  out: Output (defined in this tuning region) parameter
  bp: Basic parameter

  (Available for all functions)
- **select sub region**(**start** | **end**): Specifies that this is a selection procedure
  (For function-name select specification)
- **according** (<conditional expression> | **estimated**): Specifies a selection procedure by the standard specified below

  <conditional expression>::=
  [(**min**(<parameter name>) | **condition** (<condition>)) <connector>]
  <connector>::=[**.and.** | **.or.**] <conditional expression>

  estimated <mathematical expression>: Specifies that the optimum process should be selected and run, based on the user-defined cost (mathematical expression) associated with the selections.

  (For function-name select specification)
- **varied** (<parameter>[, <parameter>]) **from** X **to** Y
  Specifies the range of variation of the specified parameter (from X to Y)
  (For function-name variable/unroll specification)
- **fitting** <method> **sampled** <scope>: Specifies the method to use for inferring parameters

  <method>::=
  [**least-squares** <order> | **user-defined** <mathematical expression> | **auto**]
  least-squares: Specifies to infer the parameter by means of the least squares method via a polynomial expression.
  <order> Sets the order of the polynomial expression.
  user-defined: Infer using the least squares method, using the mathematical expression specified by the user.
  auto: Allow the system to infer the parameter. (Continued)

<scope>::= [<number> | **auto**] Specifies the scope required for parameter inference. Note that this can be omitted when <method> = auto

<number>: Specifies the concrete numerical value of the parameter

auto: Automatically set the parameter's sampling interval.

If the *fitting* subtype specifier is omitted, the optimum parameter is determined by measuring the entire range specified by the *varied* subtype specifier (i.e. exhaustive search).

(For function-name variable/unroll specification)

- **number** <number>: Used to specify the order in which to process the tuning regions. If this is omitted, the regions are processed first to last. In the case of nested specifiers, a number can only be assigned to the outermost specifier (Available for all functions)
- **prepro sub region** (**start** | **end**): Specifies processing to apply before calling the tuning region (Available for all functions)
- **postpro sub region** (**start** | **end**): Specifies processing to apply after calling the tuning region (Available for all functions)
- **debug** (<parameter>, [<parameter>, …]): Specifies that the variable is to be shown in the debug display when the tuning region is executed

(Available for all functions)

<parameter>::=[**bp** | **pp** | **any**]

bp: Display basic parameter information

pp: Display performance parameter information

Any: Display information about the specified parameter

Sample Program 1: Perform auto tuning upon installation to unroll a matrix product loop from 1 to 16 levels. The parameters are inferred by means of the least squares method, using a fifth-order polynomial equation. Additionally, set the sample points at 1-5, 8, and 16. Specify the performance parameter (number of unrolling levels) for debug display.

```
!ABCLib$ install unroll region start
!ABCLib$ name MyMatMul
!ABCLib$ varied (i, j) from 1 to 16
!ABCLib$ fitting least-squares 5 sampled (1-5, 8, 16)
!ABCLib$ debug (pp)
do i=1, n
  do j=1, n
    do k=1,n
      A(i, j) = A(i, j) + B(i, k) * C(k, j)
    enddo
  enddo
enddo
!ABCLib$ install unroll (i, j) region end
```

3. Implementing an Auto Tuning Library via ABCLibScript

3.1 The Initialization Interface

ABCLibScript provides an interface for executing auto tuning. The user can specify auto tuning by means of this interface.

Specifically, ABCLibScript provides the following interface for auto tuning.

- **ABCLib_ATexec** (ABCLib_ATkinds, ABCLib_ATroutines)

The ABCLib_ATexec procedure performs the auto tuning specified by ABCLib_ATkinds within the auto-tuning region specified by ABCLib_ATroutines.

The ABCLib_ATkinds parameter is used to specify the type of auto-tuning to perform. One of the following four constants defined in the ABCLibScript.h header file can be specified.

ABCLib_INSTALL: Install-time auto tuning
ABCLib_STATIC: Pre-execution auto tuning
ABCLib_DYNAMIC: Run-time auto tuning
ABCLib_ALL: All auto-tuning

The ABCLib_ATroutines parameter specifies which tuning region to perform the processing on. This parameter can be declared by the user, using the type ABCLib_ATname defined in the header file ABCLibScript.h, or it can be specified using a global variable common-defined in ABCLibScript.h:

ABCLib_AllRoutines: For all routines
ABCLib_InstllRoutines: For install-time auto tuning routines
ABCLib_StaticRoutines: For pre-execution auto tuning routines
ABCLib_DynamicRoutines: For run-time auto tuning routines

The ABCLib_ATset subroutine substitutes auto-tuning processor information.

Note that during install-time auto tuning and pre-execution auto tuning, auto tuning is performed when ABCLib_ATexec is called. During run-time auto tuning, however, execution is configured only. Actual auto tuning is performed when the section is question is called.

Sample Usage

!ABCLib$ call ABCLib_ATexec (ABCLib_INSTALL, ABCLib_InstallRoutines)

Performs install-time optimization.

- **ABCLib_ATset** (ABCLib_ATkinds, ABCLib_ATroutines)

The ABCLib_ATset procedure performs the procedures configured in ABCLib_ATroutines on the tuning region specified by the ABCLib_ATkind directive.

The ABCLib_ATkinds parameter is used to specify the type of auto-tuning to perform. One of the following four constants defined in the ABCLibScript.h header file can be specified.

ABCLib_INSTALL: Install-time auto tuning
ABCLib_STATIC: Pre-execution auto tuning
ABCLib_DYNAMIC: Run-time auto tuning
ABCLib_ALL: All auto-tuning

The name of the target tuning region is substituted in the ABCLib_ATroutines parameter.

Note that the following routine, ABCLib_ATdel, is used to delete specific routine names.

- **ABCLib_ATdel** (ABCLib_ATroutines, DelName)

ABCLib_ATdel procedure deletes the tuning region with the name specified by DelName, from the parameter containing the tuning region name information specified by means of ABCLib_ATroutines.

In the DelName parameter, write the name of the tuning region to be deleted. Note that the tuning region name is written for every tuning region in the annotation format (see below for details).

Sample Usage

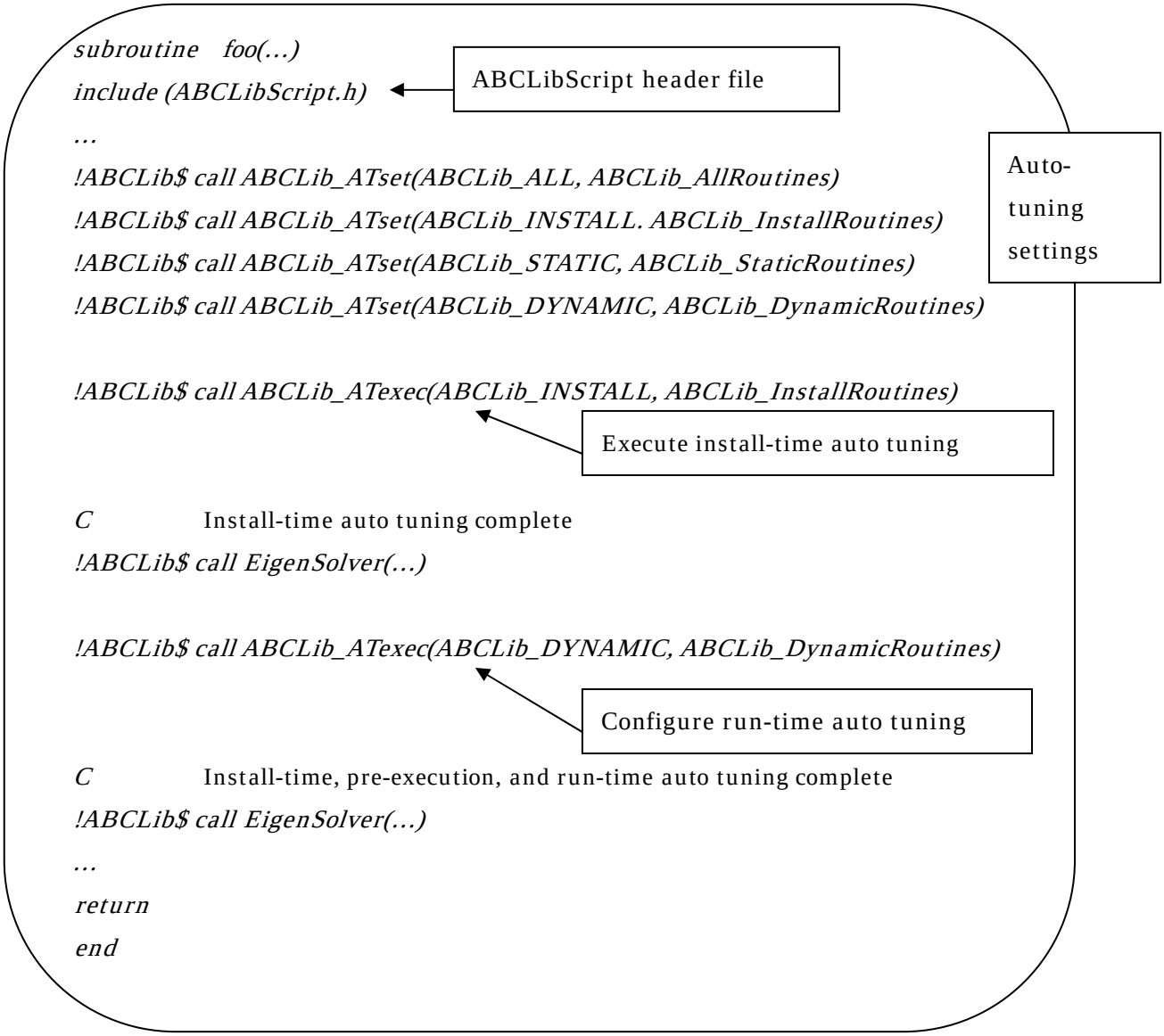
!ABCLib$ call ABCLib_ATdel(ABCLib_InstallRoutines, "MyMatMul")

Deletes the MyMatMul tuning region from the candidates for install-time optimization.

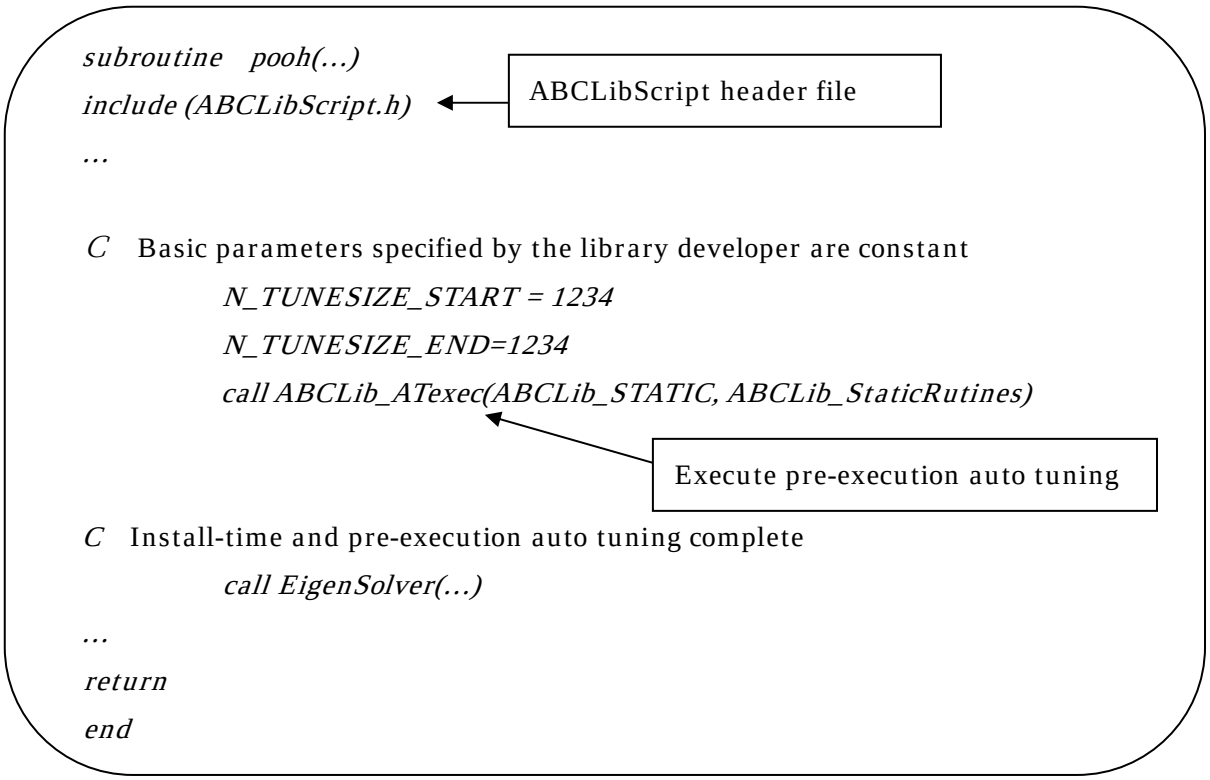
3.2 Overview of Implementation Methods

This section describes each function, using concrete examples.

A library developer's subroutine EigenSolver can be written in the following sequence, using the ABCLibScript interface in the library developer's subroutine foo which performs auto tuning.



Here, an end user is using the EigenSolver feature of the library provided by the library developer. Pre-execution auto tuning can be performed in the following sequence, using the ABCLibScript interface in the library user's subroutine pooh.



3.2.1 Sample Use of Install-time Auto Tuning Routines

Install-time auto-tuning routines are generally meant to be run once after library installation. Consequently, by default they are not executed the second and subsequent times they are specified. If you would like to process them again, you must initialize them by calling the ABCLib_ATInstallInit initialization interface below.

- **ABCLib_ATInstallInit** (ABCLib_InstallRoutines)

The ABCLib_ATInstallInit procedure undoes the tuning on the install-time tuning region specified in ABCLib_InstallRoutines.

Sample Usage

!ABCLib$ call ABCLib_ATInstallInit(ABCLib_InstallRoutines)

We assume that you want to run install-time auto tuning again.

In addition to the parameter inference performed upon installation as shown in **Sample Program 1**, install-time auto tuning routines are meant to measure those parameters that can be measured during installation. This specification is written using the following define function.

Sample Program 2:

Parameter determination using the define function in an install-time routine

```
!ABCLib$ install define (CacheSize, CacheLine) region start
!ABCLib$ name SetCachePram
!ABCLib$ parameter (out CacheSize, out CacheLine)
...
CacheSize =....
CacheLine=....
!ABCLib$ install define (CacheSize, CacheLine) region end
```

The install-time routine saves these parameter specification values in a **parameter information file**. The file name is ABCLib_InstallPram.dat, and it is stored in text format.

Example: Contents of ABCLib_InstallParam.dat parameter information file

```
(SetCacheParam
  (CacheSize    64)
  (CacheLine    8)
 )
```

3.2.2 Sample Use of Pre-execution Auto Tuning Routine

Pre-execution auto tuning determines parameters based on detailed information before the end user executes the library. Pre-execution auto tuning tunes parameters on the condition that the end user guarantees the values of the basic parameter described in

section 2.4.

The user specifies the parameters to be provided before library execution in the ABCLib_StaticParamDef.dat parameter information file. Basic parameters can be configured via substitution statements in the program. Here, **basic parameters**are parameters required for install-time and pre-execution auto tuning. By default, they are referred to internally using parameter names specified by the system.

The default basic-information parameter settings are shown below.

List of Default Basic Parameters

<Default Basic Parameters>::=
(**ABCLib_NUMPROCS** |**ABCLib_STARTTUNESIZE** | **ABCLib_ENDTUNESIZE** | **ABCLib_SAMPDIST**)

ABCLib_NUMPROCS: Integer. Indicates the number of processors to use.

ABCLib_STARTTUNESIZE: Integer. The starting problem size for auto tuning with regard to the default basic parameters.

ABCLib_ENDTUNESIZE: Integer. The ending problem size for auto tuning with regard to the default basic parameters.

ABCLib_SAMPDIST: Integer. The size of the increment to increase the problem size for auto tuning with regard to the default basic parameters.

Note that pre-execution auto tuning will not run if the basic parameters are not set. Additionally, install-time auto tuning will not run unless ABCLib_NUMPROCS, ABCLibSTARTTUNESIZE, ABCLib_ENDTUNESIZE, and ABCLib_SAMPDIST are set.

Library developers wishing to set new basic parameters should use the ABCLib_BPset and ABCLib_BPsetName procedures below.

- **ABCLib_BPset**(BPvalName)

The ABCLib_BPset procedure makes the basic parameter name specified by the parameter BPvalName into a new basic parameter.

Sample Usage

!ABCLib$ call ABCLib_BPset (“nprocs”)

Makes the parameter nprocs into a basic parameter.

- **ABCLib_BPsetName**(Kind, BPvalName, Name)

The ABCLib_BPsetName procedure sets the name of the information parameter relating to the basic parameter of the type specified by the parameter Kind to the name specified by the parameter Name, for the basic parameter specified in the parameter BPValName.

Here, the parameter Kind is as follows.

Kind ::=[**STARTTUNESIZE** | **ENDTUNESIZE** | **SAMPDIST**]

STATTTUNESIZE: Sample start point information relating to basic parameter BPvalName

ENDTUNESIZE: Sample end point information relating to basic parameter BPvalName

SAMPDIST: Sample point interval information relating to basic parameter BPvalName

Sample Usage

!ABCLib$ call ABCLib_BPsetName(“STARTTUNESIZE”, “nprocs”,
!ABCLib$ & “ABCLib_NprocsStartSize”)

Sets the sample start point parameter name for auto tuning for basic parameter nprocs to ABCLib_NprocsStartSize.

- **ABCLib_BPsetCDF**(BPvalName, CFDKind)

The ABCLib_BPsetCDF procedure sets the method of inferring non-sample points relating to the basic parameter name specified in the parameter BPvalName to the type of cost definition function specified by the parameter CFDKind. Here, the parameter CFDKind is as follows.

CFDKind ::=[**least-squares** <order> | **user-defined** <mathematical expression> | **auto**]

least-squares <order>: Specifies to infer the parameter by means of the least squares method via a polynomial expression. <order> Sets the order of the polynomial expression.

user-defined <mathematical expression>: Infer using the least squares method, using the mathematical expression specified by the user

auto: Allow the system to infer the parameter.

Note that by default, the cost definition function specified for the tuning region in question is used as-is.

Sample Usage

!ABCLib$ call ABCLib_BPsetCFD("nprocs", "least-squares 5")

Infer parameters for basic parameter nprocs my means of the least squares method, as a fifth-order polynomial equation.

Sample Program 3: Specify the basic-parameter sample points when performing pre-execution tuning for 1,024, 2,048, and 3,072 dimensional parameters using 4 processors.

```
!ABCLib$   ABCLib_TUNESTATIC = 1
!ABCLib$   ABCLib_NUMPROCS = 4
!ABCLib$   ABCLib_STARTTUNESIZE = 1024
!ABCLib$   ABCLib_ENDTUNESIZE = 3072
!ABCLib$   ABCLib_SAMPDIST = 1024
!ABCLib$   call ABCLib_ATexec(ABCLib_STATIC,   ABCLib_StaticRoutines)
```

Another way of specifying is to write the following in the file ABCLib_StaticParamDef.dat.

```
(BasicParam
  (ABCLib_TUNESTATIC   1)
  (ABCLib_NUMPROCS   4)
  (ABCLib_STARTTUNESIZE   1024)
  (ABCLib_ENDTUNESIZE   3072)
  (ABCLib_SAMPDIST    1024)
)
```

It is expected that the unroll feature will be used in pre-execution routines. Below is a sample of this usage.

Sample Program 4a:

This pre-execution routine determines the optimum parameter, unrolling the loop in the program below to 16 levels. Parameter measurement is performed at each of levels 1 to 16 (i.e. exhaustive search). Note, however, that it is assumed that the sample points for the basic parameter have been set as in **Sample Program 3**. Since the only loop entry variable is *n*, parameter *n* is interpreted as the default basic parameter.

```
!ABCLib$  static  unroll  (i, j)  region  start
!ABCLib$  name  MyMatMul
!ABCLib$  varied  (i, j)  from  1  to  16
do i=1, n
  do j=1, n
    do k=1, n
      A(i, j) = A(i, j) +B(i, k)*C(k, j)
    enddo
  enddo
enddo
!ABCLib$  static  unroll  (i, j)  region  end
```

In the case of this example, the optimum output parameters (i.e. the parameters for which processing is fastest) are the values i, j.

The pre-execution auto tuning routine writes this optimized parameter to the ABCLib_StaticParam.dat parameter information file.

Example: Contents of ABCLib_StaticParam.dat parameter information file:

```
(MyMatMul
    (ABCLib_NUMPROCS    4)
    (ABCLib_SAMPDIST     1024)
    (ABCLib_PROBSIZE    1024
      (MyMatMul_I     4)
      (MyMatMul_J     8)   )
    (ABCLib_PROBSIZE    2048
      (MyMatMul_I     4)
      (MyMatMul_J     9)   )
    (ABCLib_PROBSIZE    3072
      (MyMatMul_I     5)
      (MyMatMul_J     10)   )
)
```

Pre-execution auto-tuning routine parameter determination can be performed using only parameters specified in the program, and also by calling the parameters of the install-time auto-tuning routine (data in the parameter information file). **Sample Program 5** shows an example of this.

Sample Program 4b:

This pre-execution routine determines the optimum parameter, unrolling the loop in the program below to 16 levels. Parameter measurement is performed at each of levels 1 to 16 (i.e. exhaustive search). Note, however, that it is assumed that the basic information parameters have been specified as in **Sample Program 3**.

In the example below, because *n* is not the only loop-termination variable (the variable *nprocs* also exists), it is not known whether the variable *n* or *nprocs* is the default basic parameter. Consequently, the user must specify the basic parameter via the parameter subtype specifier.

```
!ABCLib$ static unroll (i, j) region start
!ABCLib$ name MyMatMul
!ABCLib$ parameter(bp n)
!ABCLib$ varied (i, j) from 1 to 16
do i=1, n/nprocs
 do j=1, n
    do k=1, n
      A(i, j) = A(i, j) +B(i, k)*C(k, j)
    enddo
 enddo
enddo
!ABCLib$ static unroll (i, j) region end
```

Sample Program 4c:

This pre-execution routine determines the optimum parameter, unrolling the loop in the program below to 16 levels. Parameter measurement is performed at each of levels 1 to 16 (i.e. exhaustive search). Note, however, that it is assumed that the basic information parameters have been specified as in **Sample Program 3**. In the example below, both *n* and *nprocs* are specified as basic parameters.

```
!ABCLib$ call ABCLib_BPsetVal("nprocs")
!ABCLib$ call ABCLib_BPsetName(STARTTUNESIZE, "nprocs",
!ABCLib$ &          "ABCLib_NprocsStartSize")
!ABCLib$ call ABCLib_BPsetName(ENDTUNESIZE, "nprocs",
!ABCLib$ &          "ABCLib_NprocsEndSize")
!ABCLib$ call ABCLib_BPsetName(SAMPDIST, "nprocs",
!ABCLib$ &          "ABCLib_NprocsSampDist")
!ABCLib$ ABCLib_NprocsStartSize = 1
!ABCLib$ ABCLib_NprocsEndSize = 8
!ABCLib$ ABCLib_NprocsSampDist = 1
!ABCLib$ static unroll (i, j) region start
!ABCLib$ name MyMatMu
!ABCLib$ parameter(bp n, bp nprocs)
!ABCLib$ varied (i, j) from 1 to 16
do i=1, n/nprocs
  do j=1, n
    do k=1, n
      A(i, j) = A(i, j) +B(i, k)*C(k, j)
    enddo
  enddo
enddo
!ABCLib$ static unroll (i, j) region end
```

Sample Program 5:

Determine the optimum implementation method referring to the parameter CacheSize of the install-time auto-tuning routine, and using the basic parameter information for problem size and number of processors set by the end user before execution. Here, the selection is made automatically using a given standard (in this case, the run-time estimate provided by the user)[‡].

```
!ABCLib$ static select region start
!ABCLib$ name ATfromCacheSize
!ABCLib$ parameter (in CacheSize, in ABCLib_PROBSIZE,
!ABCLib$ &                in ABCLib_NUMPROC)
!ABCLib$    select sub region start
!ABCLib$    according estimated
!ABCLib$ &        2.0d0*CacheSize*ABCLib_PROBSIZE*ABCLib_PROBSIZE
!ABCLib$ &        / (3.0d0*ABCLib_NUMPROC)
                     Target process 1
!ABCLib$    select sub region end
!ABCLib$    select sub region start
!ABCLib$    according estimated 4.0d0*CacheSize*ABCLib_PROBSIZE
!ABCLib$ &        *dlog(ABCLib_PROBSIZE) / (2.0d0*ABCLib_NUMPROC)
                     Target process 2
!ABCLib$    select sub region end
!ABCLib$ static select region end
```

3.2.3 Sample Use of Run-time Auto Tuning Routine

Run-time auto tuning routines determine performance parameters based on information that can be obtained at run-time. When determining these performance parameters, it is possible to reference parameters specified in the program, parameters determined by an install-time auto-tuning routine, and parameters specified by a pre-execution auto-tuning routine.

[‡] In this case, the selection is based on execution time. Note that execution time is generally not the only standard for AT region selection. For example, subroutine cost is also a possible selection standard. This type of selection standard can be encoded using the select specifier.

Below is a description of the select function, whose use is expected in run-time routines.

Sample Program 6:

Select the optimum process based on the parameters *eps* and *iter* defined in the program. Specifically, set condition *iter* to less than 5, in order to make *eps* the minimum value.

```
!ABCLib$ dynamic select ( eps, iter ) region start
!ABCLib$ name PricondSelect
!ABCLib$ parameter (in eps, in iter )
!ABCLib$ according min ( eps ) .and. condition ( iter < 5 )
!ABCLib$ select sub region start
      Target process 1
       eps=...
!ABCLib$ select sub region end
!ABCLib$ select sub region start
      Target process 2
     eps=...
!ABCLib$ select sub region end
!ABCLib$ dynamic select ( eps, iter ) region end
```

- **ABCLib_DynPefThis**(Name)

The ABCLib_DynPefThis procedure specifies that the optimization of the tuning region name performed by the run-time auto-tuning routine specified in the parameter Name is to be performed at the location written in this procedure. Note that when using this procedure, the execution of the tuning region name is performed using optimized parameters. In other words, no parameter tuning is performed by the tuning region specified by Name.

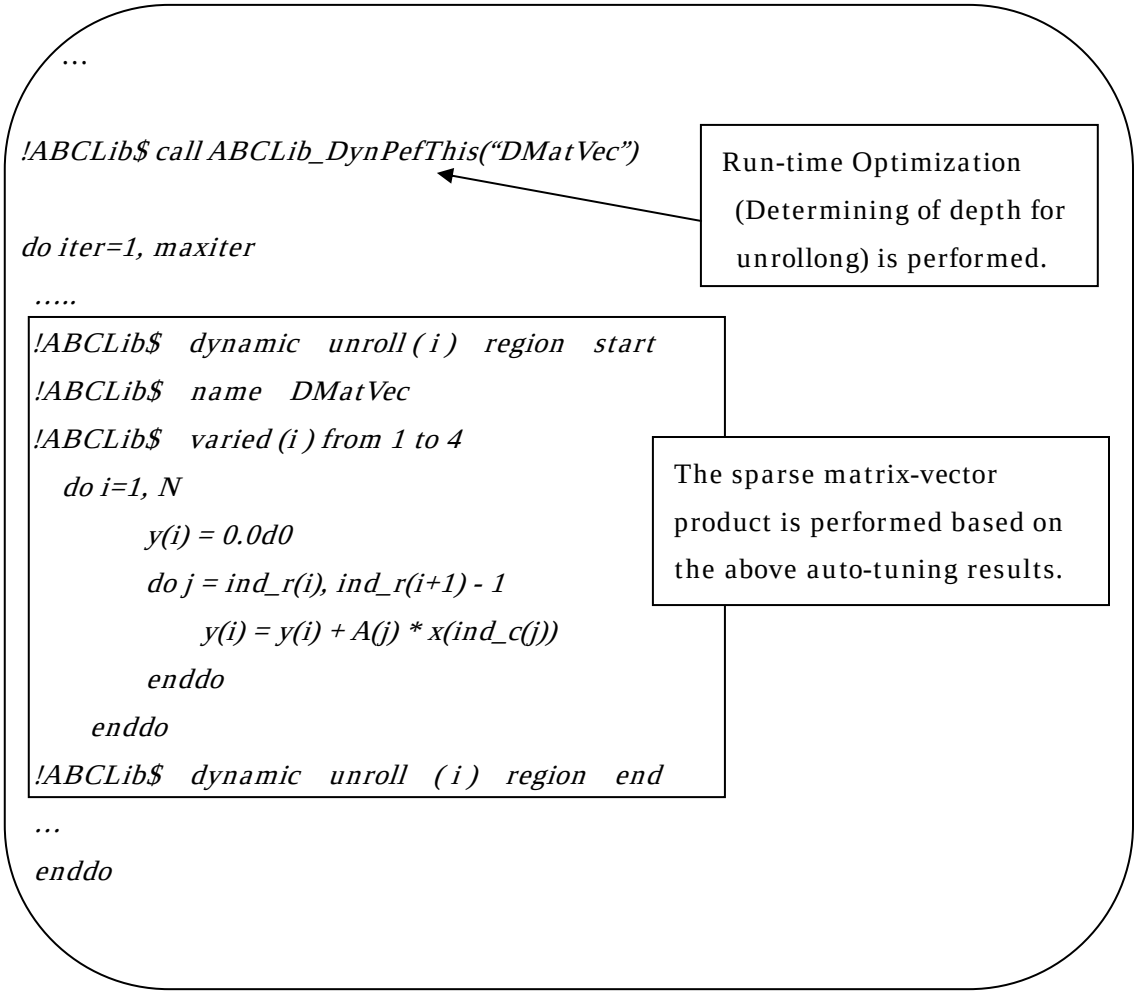
Sample Usage

!ABCLib$ call ABCLib_DynPefThis ("DMatVec")

Perform the auto tuning of the auto tuning region DMatVec here.

Sample Program 7:

Regarding the determination of the optimum number of loop-unrolling levels, re-use the parameters optimized by run-time optimization, and execute the process optimized in the AT region in question.



3.3 Sample Use of Auto Tuning Code Generation Command

To generate actual parallel-computing library code with auto-tuning features (Fortran90 + MPI), do the following:

```
>ABCLibCodeGen   test.f
```

Here we assume that test.f is a program including ABCLibScript directives.

After executing the command above, an ABCLib directory is created beneath the current directory, in which the following files are created:

./ABCLib/ABCLib_test.f:
Fortran90 + MPI source code with embedded auto-tuning code in test.f
./ABCLib/ABCLib_InstallRoutines.f:
Subroutines extracted from test.f that perform install-time auto tuning
./ABCLib/ABCLib_StaticRoutines.f:
Subroutines extracted from test.f that pre-execution auto tuning
./ABCLib/ABCLib_DynamicRoutines.f:
Subroutines extracted from test.f that run-time auto tuning
./ABCLib/ABCLib_ControlRoutines.f:
Subroutines that control auto-tuning code

If these files already exist when the preprocessor starts, non-overlapping source code and subroutines are added to the above files.

3.3.1 Run-time Options

ABCLibCodeGen can specify the following run-time options.

－debug

OFF: Do not generate debugging code

ON: Generate code debugging code at debug level x specified by the ABCLib_PRINT parameter.

－visualization

OFF: Do not output an auto-tuning trace file (default value)

ON: Output an auto-tuning trace file, and use it to create a visualization

The name of the auto-tuning trace file created when －visualization is set to ON is:

- ABCLibATlog.dat

Sample Usage

> ABCLibCodeGen －debug ON －visualization ON test.f

Specify the generation of debugging code, and output of an auto-tuning trace file of the trace of auto-tuning mode for program test.f containing ABCLibScript specifiers. This auto-tuning trace file can be used to dynamically view tuning progress in a visualizer (*VizABCLib*).

4. ABCLibScript Internal Specifications

4.1 System Parameters

Below is a list of system parameters kept by ABCLibScript. Note that these parameters are reserved words, and cannot be defined by the user.

List of System Parameters (Reserved Words)

Tuning Type Specifiers

ABCLib_ALL: An integer type with value 0. Indicates all tuning (install-time, pre-execution, and run-time)

ABCLib_INSTALL: An integer type with value 1. Indicates install-time optimization.

ABCLib_STATIC: An integer type with value 2. Indicates pre-execution optimization.

ABCLib_DYNAMIC: An integer type with value 3. Indicates run-time optimization.

Tuning Region Name Storage

ABCLib_AllRoutines:A string type. All tuning region names are stored.

ABCLib_InstallRoutines: A string type. The names of tuning regions for which install-time auto tuning is to be performed are stored.

ABCLib_StaticRoutines: A string type. The names of tuning regions for which pre-execution auto tuning is to be performed are stored.

ABCLib_DynamicRoutines: A string type. The names of tuning regions for which run-time auto tuning is to be performed are stored.

Default Basic Parameters

ABCLib_NUMPROCS: Integer. Holds the number of processors to use.

ABCLib_STARTTUNESIZE: Integer. Holds the starting sample point value for the default basic parameters.

ABCLib_ENDTUNESIZE: Integer. Holds the ending sample point value for the default basic parameters.

ABCLib_SAMPDIST: Integer. Holds the sample point interval (increment) value for the default basic parameters.

System Control

ABCLib_TUNESTATIC: Boolean. Specifies whether to execute pre-execution auto tuning.
.true.: Execute. **.false.**: Do not execute.

ABCLib_TUNEDYNAMIC: Boolean. Specifies whether to execute run-time auto tuning.
.true.: Execute. **.false.**: Do not execute.

ABCLib_DEBUG: Integer. Specifies the debug print level.
0: None. **Value x of 1 or greater**: Debug printing at level x.

4.2 Input and Output Files

This section describes the input and output files (parameter information files) handled by ABCLibScript. There are two types of I/O file: Files automatically generated by ABCLibScript (**system specification files**) and files specified by the user for debugging and the like (**user specification files**).

System specification files:

ABCLib_InstallParam*X*.dat: For install-time auto tuning routine parameter output
ABCLib_StaticParam*X*.dat: For pre-execution auto tuning routine parameter output

User specification files

ABCLib_InstallParamDef*X*.dat: For install-time auto tuning routine parameter specification
ABCLib_StaticParamDef*X*.dat: For pre-execution auto tuning routine parameter specification
ABCLib_DynamicParamDef*X*.dat: Input file for pre-execution auto tuning routine parameter specification

Note: *X* holds the name of the AT region in question.

4.2.1 Input Files

Input files consist of both user specification files and system specification files.

User specification files

ABCLib_InstallParamDef*X*.dat: For install-time auto tuning routine parameter specification

ABCLib_StaticParamDef*X*.dat: For pre-execution auto tuning routine parameter specification

ABCLib_DynamicParamDef*X*.dat: For run-time auto tuning routine parameter specification

System specification files:

install-time routines: None

Pre-execution routines:

ABCLib_InstallParam*X*.dat

Run-time routines:

ABCLib_StaticParam*X*.dat,

ABCLib_DynamicParam*X*.dat

4.2.2 Output Files

Output files consist of system specification files only.

System specification files:

install-time routines:

ABCLib_InstallParam*X*.dat

Pre-execution routines:

ABCLib_StaticParam*X*.dat

Run-time routines:

ABCLib_DynamicParam*X*.dat

4.2.3 Inupt/Output File Format

The format of user specification files and system specification files is as follows.

```
<format>::=
(<name>
    (<key> <value>)
    [(<key > < value >)]
      …
    )
    [<format>];
  < key >::= (<format> | parameter name) ;
  < value >::=[parameter value];
```

<name> specifies a tuning region name or basic parameter name. To specify a basic parameter, write BasicParam.

4.3 Collisions with Parameters in User Specification Files

When auto tuning is performed on a parameter specified in a user specification file (when an attempt is made to determine the parameter), this is called a **parameter collision**.

When there is a parameter collision, auto tuning halts, and the user-specified parameter is forcibly set.

When the system detects a parameter collision, it assumes that the user wants to halt the auto-tuning feature in order to debug or the like. To put it the other way around, the user can perform debugging by defining parameter information in a user specification file.

4.4 Nesting of Statements

This section defines specifier nesting.

4.4.1 Nesting Availability and Depth

The type of nesting available in auto-tuning is defined in Table 1.

Table 1. Availability of Nesting by Auto Tuning Type

Nesting part (superior part)	Nested part (subordinate part) install	static	dynamic
install	yes	no	no
static	yes	yes	no
dynamic	yes	yes	yes

Table 2 defines combinations of features that can be nested.

Table 2. Nesting Availability by Feature

Nesting part (superior part)	Nested part (subordinate part) define	variable	select	unroll
define	yes	yes	yes	yes
variable	yes	yes	yes	yes
select	yes	yes	yes	yes
unroll	no	no	no	no

The maximum nesting depth (how far down elements can be nested) is currently as shown below. In other words, the maximum nesting depth is three.

Nesting depth = 3 or less

4.4.2 Parameter Search Order (extended feature)

The method of searching for nested parameters is determined by the method specified by the outermost tuning region. The parameter search method can be annotated as follows.

```
subtype specifier <parameter search method>::=
                    (exhaustive search | AD-HOC method)
```

Now, let us assume that there are *m* tuning regions with parameters P_i (i=1, 2, … m), each needing to vary N_i parameters. In this case, the parameters are expressed as follows:

P = (V(P_1), V(P_2), …, V(P_m)),

where V(P_i) expresses 1 of the N_i parameters of parameter P_i.

Exhaustive search:

```
!ABCLib$  search  Brute-force
```

In an exhaustive search, all combinations are investigated. In other words, under this method all combinations of parameter P are searched.

Consequently, the number of parameter combinations is ΠN_i.

AD-HOC method:

```
!ABCLib$  search  AD-HOC
```

When the AD-HOC method is used, all combinations of parameter P are not searched.

The search starts with a given parameter with a set initial value, varied P_m, and the optimum parameter found and set. Next, it is varied P_m-1, the optimum parameter found and set, and so on. The algorithm then repeats the process until P_1.

Consequently, the number of parameter combinations is ΣN_i.

Action When Different Search Methods are Specified for Different Nested Specifiers

In general, the search begins from the innermost AT region, and made to match the outermost search method. However, of the inner method is AD-HOC, and the outer method is exhaustive, it will be treated as if the parameters of the AD-HOC specified AT regions are constant values.

Sample Program 8:

How is parameter searching carried out for the following nested processes?

```
!ABCLib$ static variable (BL) region start
!ABCLib$ name ABlockRoutine
!ABCLib$ varied 1 from 16
do iter=1, n, BL
…
!ABCLib$ static unroll ( i, j ) region start
!ABCLib$ name Kernel1
!ABCLib$ varied ( i, j ) from 1 to 32
  do i=1+iter, n
   do j=1+iter, n
     do k=1+iter, n
       …..
     enddo
    enddo
  enddo
!ABCLib$ static unroll ( i, j ) region end
….
!ABCLib$ static unroll (i, j) region start
!ABCLib$ name Kernel2
!ABCLib$ varied ( l, m ) from 1 to 32
  do l=1+iter, n
    do m=1+iter, n
      do p=1+iter, n
        …..
      enddo
    enddo
  enddo
!ABCLib$ static unroll ( l, m ) region end
…
enddo
!ABCLib$ static variable (BL) region end
```

Here, we assume that the parameter ordering is (BL, (i,j),(l,m)).

In the case of the above example, an exhaustive search is performed for all tuning region: AblockRoutine, Kernel1, and Kernel2. Here, the parameter search proceeds as follows:

(1,(1,1),(1,1)), (1,(1,1),(1,2)),….(1,(1,1),(1,32)),
(1,(1,1),(2,1)), (1,(1,1),(2,2)),…,(1,(1,1),(2,32)),
…,
(1,(1,2),(1,1)),(1,(1,2),(1,2)),…,(1,(1,2),(1,32)), ….,

Thus, there are 16*32*32*32*32 = 1,677,216 searches.

Let us assume that in the above example, the method for all tuning regions (AblockRoutine, Kernel1, and Kernel2) is AD-HOC. In this case, the search will be as follows:

(1,(1,1),(1,1)),(1,(1,1),(1,2)),…,(1,(1,1),(1,32)): Fastest parameter is determined (e.g. 8)
(1,(1,1),(1,8)),(1,(1,1),(2,8)),…,(1,(1,1),(32,8)): Fastest parameter is determined (e.g. 4)
(1,(1,1),(4,8)),(1,(1,2),(4,8)),…,(1,(1,32),(4,8)): Fastest parameter is determined (e.g. 5)
(1,(1,5),(4,8)),(1,(2,5),(4,8)),…

In other words, there are 16+32+32+32+32 = 144 parameter searches.

Let us assume that in the above example, the method for tuning region AblockRoutine is exhaustive search, and that for tuning regions Kernel1 and Kernel2 is AD-HOC. In this case, the search will be as follows:

(1,(1,1),(1,1)),(1,(1,1),(1,2)),…,(1,(1,1),(1,32)): Fastest parameter is determined (e.g. 8)
(1,(1,1),(1,8)),(1,(1,1),(2,8)),…,(1,(1,1),(32,8)): Fastest parameter is determined (e.g. 4)
(1,(1,1),(4,8)),(1,(1,2),(4,8)),…,(1,(1,32),(4,8)): Fastest parameter is determined (e.g. 5)
(1,(1,5),(4,8)),(1,(2,5),(4,8)),…,(1,(32,5),(4,8)): Fastest parameter is determined (e.g. 6)
(1,(6,5),(4,8)),(2,(6,5),(4,8)),…,

In other words, there are 16+32+32+32+32 = 144 parameter searches.

Now let us assume that in the above example, the method for tuning region AblockRoutine is AD-HOC, and that for tuning regions Kernel1 and Kernel2 is exhaustive search. In this case, the search will be as follows:

(1,(1,1),(1,1)),(1,(1,1),(1,2)),…,(1,(1,1),(1,32)) ,
(1,(1,1),(2,1)),(1,(1,1),(2,2)),…,(1,(1,1),(2,32)) ,
…
(1,(1,1),(32,1)),(1,(1,1),(32,2)),…,(1,(1,1),(32,32)): Fastest parameter is determined (e.g. (3,9))
(1,(1,1),(3,9)),(1,(1,2),(3,9)),…,(1,(1,32),(3,9)),
(1,(2,1),(3,9)),(1,(2,2),(3,9)),…,(1,(2,32),(3,9)),
…
(1,(32,1),(3,9)),(1,(32,2),(3,9)),…,(1,(32,32),(3,9)): Fastest parameter is determined (e.g. (2,8))
(1,(2,8),(3,9)),(2,(2,8),(3,9)),…,(16,(2,8),(3,9)): Fastest parameter is determined (e.g. 6)

In other words, there are 16+32*32+32*32 = 2,064 parameter searches.

Note that if no search method is specified, the default methods are as follows.

Feature: Default search method
define: None (no need for search)
variable: Exhaustive search
select: AD-HOC search
unroll: Exhaustive search

5. Conclusion

This user's guide has described the specifications and usage of the ABCLibScript auto-tuning processing directives for auto-tuning software developers.

The auto-tuning directives encoded in ABCLibScript could be called the unique knowledge of software developers. Consequently, encoding the craftsman like knowledge of software engineers using ABCLibScript will pass on this knowledge to other engineers via the source code.

In the past, this knowledge had to be obtained individually, and was not something that could be published for others. That is what turned the expertise and knowledge of individuals into a kind of obscure craftsman like technical knowledge.

In light of this background, the author believes that the clear encoding of knowledge in the source code using ABCLibScript will have a major ripple effect.

References

[A-1]Brewer,A.E., Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization, Ph.D Thesis, Massachusetts Institute of Technology (1994)

[A-2] Bilmes, J., Asanovic, K, Chin,C.–W. and Demmel, J.:Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology, Proceedings of International Conference on Supercomputing 97, pp.340―347 (1997)

[A-3]高田広章：特集「組み込みシステム開発の現状」、情報処理、Vol.38、No.10、pp.870―903 (1997)

[A-4]黒田久泰、片桐孝洋、佃良生、金田康正：自動チューニング機能付き並列数値計算ライブラリ構築の試み ―対称疎行列用の連立一次方程式ソルバを例にして―、情報処理学会第 57 回全国大会講演論文集(１)、pp.1-10―1-11 (1998)

[A-5] Frigo, M.: A Fast Fourier Transform Compiler, Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, pp.169―180 (1999)

[A-6] Whaley, R., Petitet, A. and Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS project, Parallel Computing, Vol.27, pp. 3―35 (2001)

[A-7] 片桐孝洋、黒田久泰、大澤清、工藤誠、金田康正：自動チューニング機構が並列数値計算ライブラリに及ぼす効果、情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム、Vol.42、No.SIG 12 (HPS 4)、pp. 60―76 (2001)

[A-8] 直野健、山本有作：単一メモリ型インタフェースを有する自動チューニング並列ライブラリの構成方法、情報処理学会研究報告、No. 2001-HPC-87、pp.25―30 (2001)

[A-9] Ribler, R.L., Simitci, H. and Reed, D.A.: The Autopilot Performance-Directed Adaptive Control System, Future Generation Computer Systems, Special Issue (Performance Data Mining), Vol.18, No.1, pp. 175―187 (2001)

[A-10] Kuroda, H., Katagiri,T., and Kanada,Y: Knowledge Discovery in Auto-tuning Parallel Numerical Library, Progress in Discovery Science, Final Report of the Japanese Discovery Science Project. Lecture Notes in Computer Science 2281 Springer 2002, ISBN 3-540-43338-4, pp.628―639 (2002)

[A-11] Tapus, C., Chung, I.-H. and Hollingsworth, J. K.: Active Harmony : Towards Automated Performance Tuning, Proceedings of High Performance Networking and Computing (SC2002), Baltimore, USA (2002)

[A-12] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：実行起動前最適化層を有する自動チューニングソフトウエア構成方式の提案、２００３年先進的計算基盤システムシンポジウム（Symposium on Advanced Computing Systems and Infrastructures (SACSIS)、ＳＡＣＳＩＳ２００３ 論文集、pp.159―160 (2003)

[A-13] Katagiri,T., Kise,K., Honda,H., and Yuba,T., ： FIBER: A Framework of Installation, Before Execution-invocation, and Run-time Optimization Layers for Auto-tuning Software, 電気通信大学情報システム学研究科技術報告, UEC-IS-2003-3 (2003)

[A-14] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：FIBER：汎用的な自動チューニング機能の付加を支援するソフトウエア構成方式、第 94 回 ハイパフォーマンスコンピューティング（HPC）研究会、平成 15 年 6 月 13 日（金)、情報処理学会研究報告 2003-HPC-94, pp. 1―6 (2003)

[A-15] Katagiri, T., Kise K., Honda, H., and Yuba,T.,: FIBER: A General Framework for Auto-Tuning Software, Springer LNCS 2858, pp.146―159, The Fifth International Symposium on High Performance Computing (ISHPC-V), Tokyo Fashion Town Building, Tokyo International Trade Center (Odaiba, Tokyo, JAPAN), October 20-22 (2003)

[A-16] 今村俊幸、直野健：性能安定化を目指した自動チューニング型固有値ソルバーについて、先進的計算基盤システムシンポジウムＳＡＣＳＩＳ２００３論文集、pp.145―152, (2003)

[A-17]須田礼仁：ERXPP-数値ライブラリにより並列計算性能を簡易かつ適応的に引き出す方式の提案、情報処理学会研究報告、2003-HPC-96、pp.19―24 (2003)

[A-18] 高田広章：組み込みシステム開発の現状と課題、先進的計算基盤システムシンポジウムＳＡＣＳＩＳ２００３チュートリアル資料 (2003)

[A-19] Katagiri,T., Kise,K., Honda,H., and Yuba,T.,: Effect of Auto-tuning with User's Knowledge for Numerical Software, Proceedings of ACM Computing Frontiers (CF) 04, pp.12--25, Island of Ischia, Italy, 14―16 (2004)

[A-20] Cuenca, J., Gimenez, D., and Gonzalez, J.: Architecture of an Automatically Tuned Linear Algebra Library, Parallel Computing, Vol.30, pp.187—210 (2004)

[A-21] 今村俊幸、直野健：キャッシュ競合を制御する性能安定化機構内蔵型数値計算ライブラリについて、ハイパフォーマンスコンピューティングと計算科学シンポジウムＨＰＣＳ２００４論文集、pp.173—180　(2004)

[A-22] 直野健、今村俊幸、恵木正史：ＧＲＩＤコンピューティング環境における行列ライブラリ向け性能保証方式の検討、ハイパフォーマンスコンピューティングと計算科学シンポジウムＨＰＣＳ２００４論文集、pp.51—58 (2004)

[A-23] 石井良規、片桐孝洋、本多弘樹、弓場敏嗣：Autopilot を用いた疎行列ソルバにおける実行時自動チューニング機構の設計、電子情報通信学会総合大会論文集、D-3-9、pp. 28、(2004)

[A-24] Eijkhout,V., Fuentes,E: Statistical Techniques for Algorithm Ranking In Self-Adapting Numerical Software, Eleventh SIAM Conference on Parallel Processing for Scientific Computing (PP04), Hyatt at Fisherman's Wharf, San Francisco, CA, USA, February 25, 2004, Organized Session of MS22, Multimethod Sparse Solvers for Large Scale Simulations (2004)

[A-25]渡辺政彦：組み込みソフトウエア向け開発支援環境、情報処理、Vo.45、 No.1、pp.10—15 (2004)

[A-26] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：自動チューニング処理記述用ディレクティブ ABCLibScript の設計と実装、２００４年先進的計算基盤システムシンポジウム（Symposium on Advanced Computing Systems and Infrastructures (SACSIS)、ＳＡＣＳＩＳ２００４ 論文集、pp.43—52 (2004)、および、自動チューニング処理記述用ディレクティブ ABCLibScript、電気通信大学情報システム学研究科技術報告、UEC-IS-2004-1

(2004)：

[A-27] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：自動チューニング処理記述用ディレクティブ ABCLibScript の設計と実装、２００４年先進的計算基盤システムシンポジウム（Symposium on Advanced Computing Systems and Infrastructures (SACSIS)、ＳＡＣＳＩＳ２００４ 論文集、pp.43—52 (2004)

[A-28] 直野健、惠木正史：ＧＲＩＤコンピューティング環境下での行列ライブラリにおける性能保証ライン算出の改良型アルゴリズム、２００４年先進的計算基盤システムシンポジウム（Symposium on Advanced Computing Systems and Infrastructures (SACSIS), ＳＡＣＳＩＳ２００４ 論文集, pp.295—304 (2004)

[A-29] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：ユーザ知識を活用するソフトウエア自動チューニングについて、第 回 ハイパフォーマンスコンピューティング（HPC）研究会、平成 15 年 8 月 1 日（金）、情報処理学会研究報告 2004-EVA-10、pp.19— 24 (2004)

[A-30]マイクロソフト SQL Server 2000：
http://www.microsoft.com/japan/sql/default.mspx

[A-31] 東京大学 生産技術研究所 桑原研究室：交通流シミュレーション、ネットワークシミュレーションの容量パラメタの自動調整法：http://www.transport.iis.u-tokyo.ac.jp/

[A-32] 電子情報通信学会、ディペンダブルコンピューティング研究会、
http://www.ieice.org/iss/dc/jpn/

[A-33] 片桐孝洋、吉瀬謙二、本多弘樹、弓場敏嗣：データ再分散を行う並列 Gram-Schmidt 再直交化、情報処理学会論文誌：コンピューティングシステム、Vol.45、No. SIG 6 (ACS 6)、pp.75―85 (2004)：

[A-34] 直野健、猪貝光祥：マルチカラー逆反復法による直交化法とその性能、日本応用数理学会年会、オーガナイズドセッション：数値線形代数、中央大学後楽園キャンパス、2004 年 9 月 16 日～18 日

Related Projects:

[B-1] PHiPAC プロジェクト；http://www.icsi.berkeley.edu/~bilmes/phipac/

[B-2] ATLAS プロジェクト； http://www.netlib.org/atlas/index.html.

[B-3] ILIB プロジェクト（HINTS プロジェクト）；http://www.hints.org/

[B-4] ABCLib プロジェクト；http://www.abc-lib.org/

[B-5] FIBER プロジェクト；
http://www.yuba.is.uec.ac.jp/~katagiri/FIBER_Prj/FIBER-Prj.html

[B-6] Autopilot プロジェクト：
http://www-pablo.cs.uiuc.edu/Project/Autopilot/AutopilotOverview.htm

[B-7] Active Harmony プロジェクト：http://www.dyninst.org/harmony/

[B-8] SANS プロジェクト； http://icl.cs.utk.edu/sans/index.html

[B-9] SALSA プロジェクト； http://icl.cs.utk.edu/salsa/

[B-10] BeBOP プロジェクト； http://bebop.cs.berkeley.edu/

[B-11] Sparsity プロジェクト； http://www.cs.berkeley.edu/~yelick/sparsity/

Patents:

[C-1]片桐孝洋：プログラム、記録媒体およびコンピュータ、 日本国特許出願、
特願 2003-022792 (平成 15 年 1 月 30 日)

[C-2] 片桐孝洋: 計算装置、計算方法、プログラムおよび記録媒体、 日本国特許出願、
特願 2003-092592 (平成 15 年 3 月 28 日)。

[C-3] 片桐孝洋：計算装置、計算方法、プログラムおよび記録媒体、日本国特許出願、特願２００３－１４９７０１、平成１５年5月２７日、(特願２００３－９２５９２の国内優先権出願