# Effect of Auto-tuning with User's Knowledge for Numerical Software *

Takahiro Katagiri
Graduate School of Information Systems, The University of Electro-Communications
1-5-1 Choufu-gaoka, Choufu-shi, Tokyo 182-8585, Japan
PRESTO, Japan Science and Technology Agency (JST)
katagiri@is.uec.ac.jp

Kenji Kise
Graduate School of Information Systems, The University of Electro-Communications
1-5-1 Choufu-gaoka, Choufu-shi, Tokyo 182-8585, Japan
PRESTO, Japan Science and Technology Agency (JST)
kis@is.uec.ac.jp

Hiroki Honda
Graduate School of Information Systems, The University of Electro-Communications
1-5-1 Choufu-gaoka, Choufu-shi, Tokyo 182-8585, Japan
honda@is.uec.ac.jp

## ABSTRACT

This paper evaluates the effect of an auto-tuning facility with the user's knowledge for numerical software. We proposed a new software architecture framework, named FIBER, to generalize auto-tuning facilities and obtain highly accurate estimated parameters. The FIBER framework also provides a loop-unrolling function and an algorithm-selection function to support code development by library developers needing code generation and parameter registration processes. FIBER offers three kinds of parameter optimization layers—install-time, before execute-time, and run-time. The user's knowledge is needed in the before execute-time optimization layer. In this paper, eigensolver parameters that apply the FIBER framework are described and evaluated in three kinds of parallel computers: the HITACHI SR8000/MPP, Fujitsu VPP800/63, and Pentium4 PC cluster. Our evaluation of the application of the before execute-time layer indicated a maximum speed increase of 3.4 times for eigensolver parameters, and a maximum increase of 17.1 times for the algorithm selection of orthogonalization in the computation kernel of the eigensolver.

## Keywords

Auto-Tuning, Parameter optimization, Numerical library, Performance modeling, Eigensolver

*This article was submitted to ACM CF04 conference in November 14, 2003. This is also UEC IS Technical Reports, UEC-IS-2003-10, Graduate School of Information Systems, The University of Electro-Communications in November 17, 2003.



## 1. INTRODUCTION

Tuning parallel computers and other complicated machine environments is time-consuming, so an automated adjustment facility for parameters is needed. Moreover, library arguments should be reduced to make the interface easier to use, and a facility is needed to maintain high performance in all computer environments. To solve these problems, many packages of SATF (Software with Auto-Tuning Facility) have been developed.

There are two kinds of paradigms for SATF. The first paradigm is known as computer system software. Examples of this software for tuning computer system parameters such as I/O buffer size include Active Harmony [12] and Autopilot [9]. The other approach for the computer system software paradigm is known as agent system software. As an example, SANS (Self-adapting Numerical Software) [3] introduced a network agent into their system software to adjust parameters in numerical libraries for GRID environments.

The second paradigm is known as a numerical library. PHiPAC [2], ATLAS and AEOS (Automated Empirical Optimization of Software) [1, 13], and FFTW [4] can automatically tune the performance parameters of their routines when they are installed, while ILIB [6, 7] implements both install-time and run-time optimizations.

For formalization of an auto-tuning facility, K.Naono and Y.Yamamoto formulated the install-time optimization known as SIMPL [8], an auto-tuning software framework for parallel numerical processing.

Although many facilities of SATF have been proposed, these conventional facilities have a limitation from the viewpoint of general applicability, because each auto-tuning facility uses dedicated methods defined in each package. As an example of this limitation, several general numerical libraries contain direct solvers, iterative solvers, dense solvers, and sparse solvers. There is no software framework to adapt SATF to these solvers.

This paper proposes and evaluates a new and general software framework for SATF called FIBER (Framework of Install-time, Before Execute-time, and Run-time optimization layers) [11, 10], a framework containing three types of

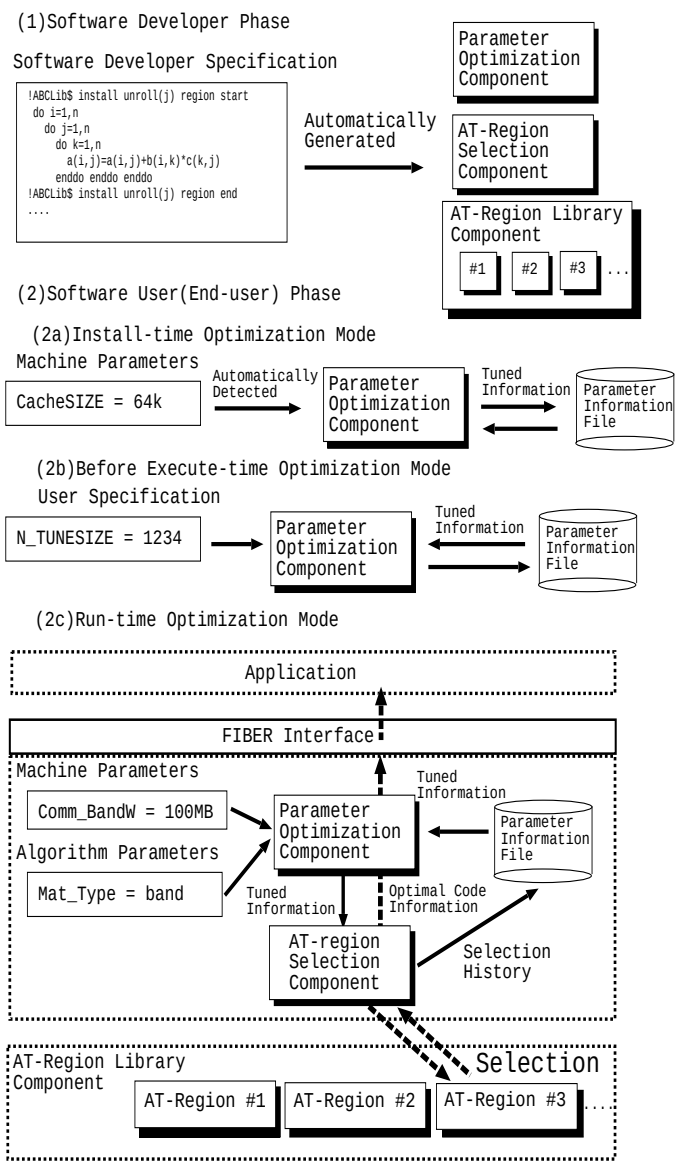


**Figure 1: Outline of process view for auto-tuning in FIBER.**

optimization layers to solve these problems.

The remainder of this paper is organized as follows: in Section 2 we explain the framework and scenarios for FIBER, in Section 3 we explain an adaptation of FIBER, in Section 4 we evaluate the three types of optimization layers in FIBER, in Section 5 we explain related work for auto-tuning software, and, finally, we conclude with a summary of our research.

# 2. FIBER FRAMEWORK

## 2.1 Process View

In the FIBER framework, there are two phases for making and using the auto-tuning facilities: the software developer phase and the end-user phase. Figure 1 shows the process view. In Figure 1, there are two viewpoints which define each phase: the viewpoint of the software developer and that of the end-user. The explanations of these two phases are shown below.

- *Software Developer Phase*:
  Software developers perform the following processes to generate auto-tuning facilities in the FIBER framework.

  (1a) Software developers, who want to add the auto-tuning facility to their software, write operations in their source programs using a language provided by FIBER for target regions, e.g., computation kernels in numerical computation routines.

  (1b) Software developers apply a pre-process tool which is provided by the FIBER toolkit developer. The pre-process tool generates new source codes with the auto-tuning facility. The automatically generated auto-tuning facilities contain three components: (i) Parameter Optimization Component, (ii) AT-Region Selection Component, and (iii) AT-Region Library Component. These components are explained in the section below.

  (1c) Finally, software developers open the source codes to end-users as a library. Or, software developers compile the source codes with a compiler, and the compiled object code is also opened to end-users.

- *Software User (End-user) Phase*:
  Software users use the auto-tuning software opened by the software developer. The following three optimization modes are performed to obtain high performance at three different times.

  (2a) *Install-time Optimization Mode:* This mode is performed when the software is installed into the end-user's environment. The install-time optimizer tries to find the hardware parameters, e.g., cache sizes and message communication latency, for efficient implementations corresponding to the user's machine environments. End-users are unaware of this optimization, since the optimization can be totally hidden in the installation or compilation processes for the target software.

  (2b) *Before Execute-time Optimization Mode:* This mode is performed after special parameters are fixed, based on the end-user's knowledge. For example, the problem size to execute for their software is a special parameter, because the system cannot know this value in advance. For fixed parameter information, optimization is performed in this mode. This optimization also includes the selection of the best algorithm. Compilers cannot perform such algorithm selection, because of the lack of knowledge of the end-user's algorithms.

  (2c) *Run-time Optimization Mode:* This mode is performed when target auto-tuning regions are executed or called. The run-time optimizer tries to obtain run-time information, e.g., input data characteristics such as the band matrix for numerical computation, the migrated load of the machines, and the current communication performance (such as the communication bandwidth for

a GRID environment). Since run-time optimization optimizes target regions with "dynamic" information, it can set the best value for a parameter which cannot be optimized in the above two optimization modes.

## 2.2 Components

The FIBER framework supports the following components:

- **Parameter Optimization Component:** This component optimizes developer-specified library parameters in the PTL (Parameter Tuning Layer). There are three different timings for the optimizations (install-time, before execute-time, and run-time).
- **AT-Region Selection Component:** This component selects the best auto-tuning region, taking into account information from the parameter information file and run-time optimizer.
- **AT-Region Library Component:** This component is automatically generated according to pre-process specifications from the software developer. The commands of this pre-process are provided by the FIBER toolkit developer. The regions of auto-tuning (AT-Regions) are specified by the software developer through a dedicated language. According to software developer's instructions, the specified AT-regions are separated and formed into several components. The aggregation of these components form the *AT-region library*.

The PTL optimizes parameters to minimize a function specified by the software developers. Even the parameters in computer system libraries, such as MPI (Message Passing Interface), can be specified if the interface is open to software developers. PTL in FIBER, thus, can access system parameters.

In our current implementation, the Parameter Optimization and AT-Region Selection Components are implemented using Fortran90 and MPI. For this reason, the generated software can be executed on parallel machines which are available to Fortran90 compilers and MPI.

## 2.3 Scenarios and Optimization Classifications

### 2.3.1 *Two Scenarios for Using The FIBER Framework*

In this section, we will show two scenarios for using the FIBER framework from the viewpoint of software developers and end-users.

Figure 2 shows the process view for the software developers. In Figure 2, software developers specify the auto-tuning instructions for their software using a specification language. The loop-unrolling depth, block length for blocking algorithms, and method for algorithm selection are target instructions in this scenario. After specifying the instructions, the developer executes a pre-process command named *ABCLibCodeGen*. According to the developer's instructions, the pre-processor generates new source codes containing the auto-tuning facilities,. Finally, the source (object) codes are released to the public as a library.

Figure 3 shows the process view for the end-users. In Figure 3, end-users who want to use the released library can download and install the library to their computer environments. When they install the library, FIBER install-time

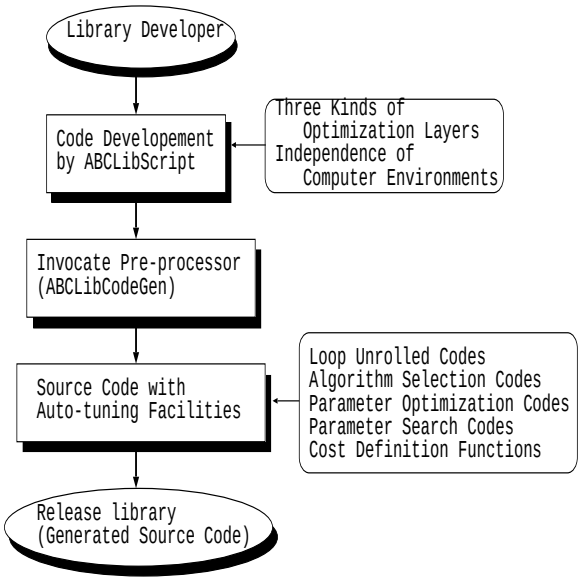


**Figure 2: Process view for software developers.**

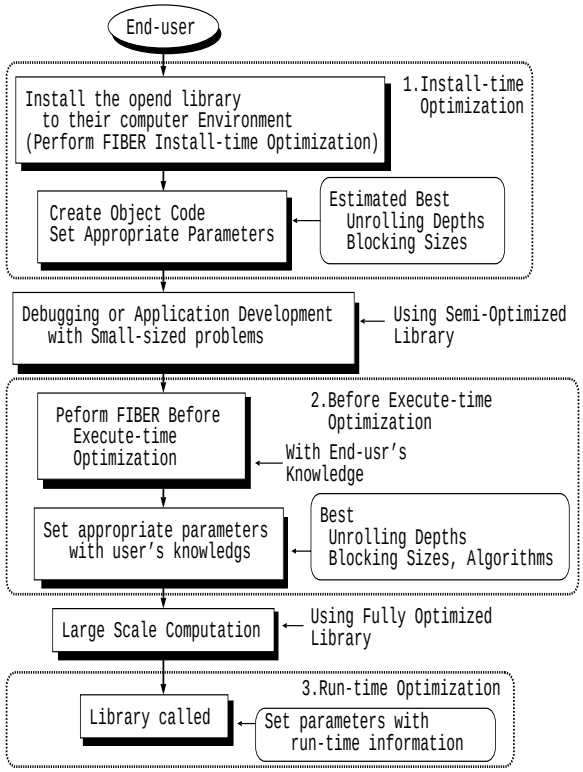


**Figure 3: Process view for the end-users.**

optimization is performed, but the end-users are not aware of this optimization. The FIBER install-time optimization sets the estimated best parameters with information from the end-user's machine environments. Finally, the installer generates a semi-optimized library with the estimated best parameters. The end-users develop or debug their software with the semi-optimized library.

After finishing developing or debugging the software, end-users perform an optimization, which is defined by the software developer, with the end-user's knowledge. This optimization is called the FIBER before execute-time optimization. The problem sizes to execute and the characteristics of the input matrix data are assumed information in this scenario. After optimizing is finished, the fully optimized object is generated. End-users can then perform a large-scale computation with the fully optimized object.

Finally, the target library is called, and the system optimizes the remaining parameters with run-time information. This optimization is called the FIBER run-time optimization. By using the run-time optimization, the target library is fully and completely optimized.

This is a typical scenario using the FIBER framework.

#### 2.3.2 Further Optimization Classifications

The optimization procedure is further classified in this section. The install-time, before execute-time, and run-time optimization modes are explained in the previous section, along with scenarios from the viewpoints of the software developer and end-user. Additional FIBER optimization modes for computer environments are classified and several scenarios are given below.

1. *Hybrid Install-time, Before Execute-time, and Run-time Optimizations (Fully Hybrid Model)*: This is a typical model for FIBER. We showed the scenario in Section 2.3.1.

2. *Hybrid Install-time and Before Execute-time Optimizations (Fully Install-time Optimization Model)*: This model is used for homogeneous computing environments, e.g., a PC cluster with homogeneous PEs and stable communication performance. Since the performance of computation and communication is stable, dynamic information for the computer environments is not needed. The optimization speed and execution time for the target process are faster than for the Fully Hybrid Model because this model does not need run-time optimization.

3. *Hybrid Before Execute-time and Run-time Optimizations (Feedback Optimization Model)*: This model is used for heterogeneous and unstable computing environments, e.g., GRID environments. In run-time optimization, dynamic information for computer environments is stored in the parameter information file. After that, the information is used to optimize target processes in the before execute-time optimization. This is a feedback system for the FIBER framework. In this model, a launch-time scheduler for GRID is one of the target processes, but the usage of this model is not limited to GRID environments. Before execute-time optimization with run-time information in this model can be used in code optimizations (Consider this as compiler code optimizations with run-time information.)

4. *Complete Install-time Optimization (Conventional Install-time Model)*: This model is classified as a conventional auto-tuning facility for a numerical library.

5. *Complete Before Execute-time Optimization (Fully User Knowledge Optimization Model)*: This model is used for optimizing algorithms, or reducing the total memory size for the library. Given the user's knowledge, the execution speed and total amount of memory for the library can be reduced.

6. *Complete Run-time Optimization*: This model is classified as conventional auto-tuning middleware at run-time.

## 3. ADAPTATION FOR FIBER AUTO-TUNING FACILITY

### 3.1 Specification of Performance Parameters by Library Developers

In the FIBER framework, library developers can implement detailed instructions to specify the performance parameters of *PP* (See [11, 10]) and target areas of auto-tuning in their programs. Hereafter, the target area is called the *tuning region*. Typical instruction operators, named *unrolling instruction operator* (`unroll`) and *selection instruction operator* (`select`), are shown below.

#### 3.1.1 Specification Format

First of all, we will explain the specification format in the dedicated language for FIBER. In the source program, the line `!ABCLib$` is regarded as a FIBER instruction. The overall notation is shown in Figure 4.

```
!ABCLib$ ⟨Auto-tuning Type⟩ ⟨Function Name⟩
      [ (Target Variables) ] region start
 [ !ABCLib$ ⟨Detail of Function⟩ [ sub region start ] ]

            Tuning Region

 [ !ABCLib$ ⟨Detail of Function⟩ [ sub region end ] ]
!ABCLib$ ⟨Auto-tuning Type⟩ ⟨Function Name⟩
      [ (Target Variables) ] region end
```

**Figure 4: Instruction format of auto-tuning in FIBER.**

The ⟨Auto-tuning Type⟩ and ⟨Function Name⟩ in Figure 4 are called instruction operators. The instruction operator ⟨Auto-tung Type⟩ can specify the three kinds of timing—install-time optimization (`install`), before execute-time optimization (`static`), and run-time optimization (`dynamic`). The way to process the target code and details for the auto-tuning method can be specified by the instruction operator ⟨Function Name⟩.

#### 3.1.2 Example of Unrolling Instruction Operator

The following code shows an example of an unrolling instruction operator.

```
!ABCLib$ install unroll (j) region start
!ABCLib$ varied (j) from 1 to 16
!ABCLib$ fitting polynomial 5 sampled (1-4,8,16)
```

```
 do j=0, local_length_y-1
    tmpu1 = u_x(j)
    tmpr1 = mu * tmpu1 - y_k(j)
    do i=0, local_length_x-1
       A(i_x+i, i_y+j) = A(i_x+i, i_y+j)
          + u_y(i)*tmpr1 - x_k(i)*tmpu1
    enddo
 enddo
!ABCLib$ install unroll (j) region end
```

The above code shows that the loop-unrolled codes for $j$-loop, which adapts unrolling to the tuning region of `region start` – `region end`, are automatically generated. The depth of the loop unrolling is also automatically parameterized as $PP$.

The instruction operator specifies detailed functions for the target instruction operator, and so it is called the sub-instruction operator. The sub-instruction operator `varied` defines the defined area of the target variables. In this example, the area is `{1,..,16}`. For the cost definition function, the types can be specified by the sub-instruction operator `fitting`. In this example, a 5th-order linear polynomial function is specified. The sub-instruction operator `sampled` defines the sampling points for estimating the cost definition function. This example of sampling is `{1-4,8,16}`.

### 3.1.3 *Example of Selection Instruction Operator*

The following code shows an example of the selection instruction operator.

```
!ABCLib$ static select region start
!ABCLib$ parameter (in CacheS, in NB, in NPrc)
!ABCLib$   select sub region start
!ABCLib$   according estimated
!ABCLib$         (2.0d0*CacheS*NB) / (3.0d0*NPrc)
           Target Process 1
!ABCLib$   select sub region end
!ABCLib$   select sub region start
!ABCLib$   according estimated
!ABCLib$         (4.0d0*CacheS*dlog(NB))/(2.0d0*NPrc)
           Target Process 2
!ABCLib$   select sub region end
!ABCLib$ static select region end
```

The above code shows that the selection procedure from several tuning regions, which are specified by `sub region start` – `sub region end`, is performed based on the values of the formulas, which are specified by the sub-instruction operator `according estimated`.

The variables referred to in the formulas are defined by the sub-instruction operator `parameter`. The sub-instruction operator `in` shows that the target variables are input variables. The values of the variables should have been stored in a parameter information file in the install-time optimization layer by using the sub-instruction operator `out`, since this example is defined as a before execute-time optimization.

Please note that the selection of Target Process 1 or Target Process 2 is parameterized as $PP$ in this example.

## 3.2 Objective of Auto-tuning

[**Example 1**] A conventional parallel numerical library interface:

```
call PEigVecCal(
     A, x, lambda, n,                    ...(i)
     nprocs, myid, iDistInd,             ...(ii)
     ictr, imv, iud, kbi, kort, ichit, ihitk,
     ibl, iop, isp, ioo, iso,            ...(iii)
     MAXITER, deps                       ...(iv)  )
```

For this paper, the arguments in Example 1 are called as (i) Basic information parameters, (ii) Parallel control parameters, (iii) Performance parameters, and (iv) Algorithm parameters. For example, the dimension sizes of matrix $A$ are specified in the parameters of (i), the data distribution information in the parameters of (ii), the unrolling depth or block size in the parameters of (iii), and the maximum iterative numbers in the parameters of (iv). Generally speaking, these arguments can be removed to design a better library interface for (ii), and to analyze numerical characteristics for (iv). The parameters of (iii), however, cannot be removed in conventional frameworks which do not have the auto-tuning facility.

The goal of the auto-tuning facility is to maintain performance and remove the parameters of (iii). Using the auto-tuning facility, the interface can be simplified as:

```
call PEigVecCal(A, x, lambda, n)
```

## 3.3 An Adaptation in The Install-time Optimization Layer (IOL)

The auto-tuning facility of FIBER is adapted to an eigensolver in this section. This is implemented using the Householder-Bisection-Inverse iteration method for computing all eigenvalues and eigenvectors in dense, real symmetric matrices. The cost definition function as the execution time for the solver is defined here.

Let the interface of the target library in the Householder-Bisection-Inverse iteration method be the same interface as `PEigVecCal`, shown in Example 1. The main arguments of this library are:

- $PP \equiv$ { `ictr`, `imv`, `iud`, `kbi`, `kort`, `ichit`, `ihit`, `ibl`, `iop`, `isp`, `ioo`, `iso` }
- $BP \equiv$ { `n`, `nprocs` }

The library interface `PEigVecCal` consists of the following four kinds of performance parameters for $PP$ in this library.

1. Householder tridiagonalization routine: $PP =$ { `ictr`, `imv`, `iud` }
2. Bisection routine : $PP =$ { `kbi` }
3. Inverse iteration routine: $PP =$ { `kort` }
4. Householder inverse transformation routine: $PP =$ { `ichit`, `ihit` }
5. QR decomposition routine with the Gram-Schmidt method: $PP =$ { `ibl`, `iop`, `isp`, `ioo`, `iso` }

The main definition area and process in each parameter are defined as:

The main definition areas in this example are:

- `ictr` $\equiv$ { 0, 1 } : Communication method for reduction operation in the Householder tridiagonalization routine. `ictr`=0 means using an implementation with one-to-one communication libraries (`MPI_SEND`, `MPI_RECV`.) `ictr`=1 means using an implementation with an MPI released library (`MPI_ALLREDUCE`.)

- `imv` ≡ { 1,2,...,16 } : Unrolling depth for the outer loop of a matrix-vector product in the Householder tridiagonalization. The kernel is formed as a double nested loop, BLAS2.
- `iud` ≡ { 1,2,...,16 } : Unrolling depth for the outer loop of an updating process in the Householder tridiagonalization. The kernel is formed as a double nested loop, BLAS2.
- `kort` ≡ { MG-S, CG-S, IRCG-S, NoOrt } : Types of algorithms in the inverse iteration routine for reorthogonalization algorithms to calculate eigenvectors corresponding to clustered eigenvalues.
- `ichit` ≡ { 1,2,3 } : Communication method for the gathering operation in the Householder inverse transformation routine. `ichit`=1 means an implementation using an MPI broadcast routine (`MPI_BCAST`). `ichit`=2 means an MPI blocking one-to-one communication routine (`MPI_SEND`, `MPI_RECV`.) `ichit`=3 means an MPI non-blocking one-to-one communication routine (`MPI_SEND`, `MPI_IRECV`.)
- `ihit` ≡ { 1,2,...,16 } : Unrolling depth for the outer loop of the Householder inverse transformation routine. The kernel is formed as a double nested loop, and this is classified as BLAS1.
- `ibl` ≡ { 1,2,3,4,8,16 } : Blocking length for the blocked algorithm of QR decomposition with the Gram-Schmidt method. The blocking length can also control communication frequency and volume. The kernel is formed as a triple nested loop, and this is classified as BLAS3.
- `iop` ≡ { 1,2,3,4 } : Unrolling depth of the outer loop for pivot PEs in the QR decomposition routine.
- `isp` ≡ { 1,2,3,4,8,16 } : Unrolling depth of the second loop for pivot PEs in the QR decomposition routine.
- `ioo` ≡ { 1,2,3,4 } : Unrolling depth of the outer loop for the updating process in the QR decomposition routine.
- `iso` ≡ { 1,2,3,4,8,16 } : Unrolling depth of the second loop for the updating process in the QR decomposition routine.

The set of parameters which are optimized in IOL is:

- $IOP$ ≡ { `ictr`, `imv`, `iud`, `kbi`, `ichit`, `ihit`, `ibl`, `iop`, `isp`, `ioo`, `iso` }.

The parameter `kort` cannot be included in the IOL parameters, since it depends on the input matrix characteristics.

#### 3.3.1 How To Estimate Parameters in IOL

The IOL parameters are estimated when the target computer systems, such as the computer hardware architecture or compilers, are fixed. This is because these parameters can be affected by the number of registers, the size of caches, vector processing factors, and other computer hardware characteristics.

The parameters are determined in the following way. First, the parameters in $BP$ are fixed, and several points for the execution time at the target process are sampled (hereafter referred to as sampled data). The cost definition function is then determined by using the sampled data.

# 4. EVALUATION FOR FIBER AUTO-TUNING FACILITY

## 4.1 Machine Environments

We use the following three kinds of parallel computers to evaluate FIBER optimization facilities.

- HITACHI SR8000/MPP
  - System configuration: The HITACHI SR8000/MPP nodes have 8 PEs. The theoretical maximum performance of each node is 14.4 GFLOPS. Each node has 16 GB memory, and the inter-connection topology is a three-dimensional hypercube. Its theoretical throughput is 1.6 Gbytes/s for one-way communication, and 3.2 Gbytes/s for two-way. For the communication library, the HITACHI optimized MPI was used.
  - Compiler : The HITACHI Optimized Fortran90 V01-04 compiler specified options, *-opt=4 -parallel=0* and *-opt=0 -parallel=0*, were used.
- Fujitsu VPP800/63
  - System configuration: This machine is a vector-parallel style of super-computer. The Fujitsu VPP800/63 at the Academic Center for Computing and Media Studies, Kyoto University was used. The total number of nodes for the VPP800 is 63. The theoretical maximum performance of each node is 8 GFLOPS for vector processing, and 1 GFLOPS for scalar processing. Each node has 8 GB memory, and the inter-connection topology is a cross-bar. Its theoretical throughput is 3.2 Gbytes/s. For the communication library, the Fujitsu optimized MPI was used.
  - Compiler : The Fujitsu optimized UXP/V Fortran/VPP V20L20 compiler specified options, *-O5 -X9* and *-O0 -X9*, were used.
- PC Cluster
  - System configuration: The Intel Pentium4 2.0 GHz, as a node of a PC cluster, was used. The number of PEs for the PC cluster is 4, and each node has 1 GB (Direct RDRAM/ECC 256 MB*4) memory. The system hardware board is the ASUSTek P4T-E+A (Socket 478). The network card is the Intel EtherExpressPro100+. Linux 2.4.9-34 and MPICH 1.2.1 are used as the operating system and communication library.
  - Compiler : The PGI Fortran90 4.0-2 compiler specified options, *-fast* and *-O0*, were used.

## 4.2 Target Processes

We will evaluate auto-tuning facilities in the FIBER framework by using the following four processes for the eigensolver.

1. Householder tridiagonalization routine:
   $PP$ = { `ictr`, `imv`, `iud` }
2. Inverse iteration routine: $PP$ = { `kort` }

3. Householder inverse transformation routine:
   $PP$ = { `ichit`, `ihit` }
4. QR decomposition routine with the Gram-Schmidt method:
   $PP$ = { `ibl`, `iop`, `isp`, `ioo`, `iso` }

## 4.3 Hypothesis

In this evaluation, we set the following hypotheses.

- Cost Definition Function (CDF): 5th-order linear polynomial function of $a_1x^5+a_2x^4+a_3x^3+a_4x^2+a_5x^1+a_6$.
- We define the CDF as the execution speed.
- The least-squares method with Householder QR decomposition is used to estimate the coefficient of the CDF.
- Sampling points for problem sizes of $n$:
  - SR8000:
    Option -*opt=4* : {100,200,...,1000,2000,...,6000}
    Option -*opt=0* : {100,200,...,1000,2000}
  - VPP800:
    Option -*O5* : {100,200,...,1000,2000,...,6000}
    Option -*O0* : {100,200,...,900}
  - PC Cluster:
    Option -*fast* : {100, 200,...,1000,2000,...,9000,10000}
    Option -*O0* : {100, 200,...,2000}

  These values were determined from the limitation of computation time for each machine.

## 4.4 Experiments on The Effect of IOL

The effect of IOL for the parameters of `iud` using several kinds of machine environments is also evaluated.

[**Experiment 1**] Evaluate the effect of IOL.

Tables 1, 2 and 3 show the IOL effect in this experiment. As a result, the following observations are obtained:

- The default parameters are not always nearly optimal. In the worst case, we found the default parameters to be a factor of 4.4 times slower than the best. Using constant parameters, hence, is not sufficient from the performance viewpoint.
- Taking into account the ratios between best and worst, we conclude that some kind of tuning facility is needed. This is because we found that the worst case has a huge speedup factor of 14.
- The QR decomposition routine has more potential for optimization compared to the tridiagonalization and Householder Inverse Transformation routines. This is because the routine of QR decomposition is blocked, and thus the specification of block length is sensitive to performance.

## 4.5 Experiments on The Effect of BEOL (Before Execute-time Optimization Layer)

In FIBER, the BEOL optimization is performed when the parameters in $BP$ (Basic Parameters, see [11, 10]) are specified by the library user before executing the target process. This section explains several adaptations of this layer.

[**Situation 1**] In Example 1, library users know the number of processors (=8 PEs), and matrix sizes (=8192) for the eigenvalue computation. (`n` $\equiv$ 8192, `nprocs` $\equiv$ 8) In Situation 1, the parameters to optimize in BEOL are

- $BEOP \equiv$ { `ictr`, `imv`, `iud`, `kbi`, `ichit`, `ihit`, `ibl`, `iop`, `isp`, `ioo`, `iso` }.

Please note that IOL uses totally estimated parameters with sampling points determined by the library developer. In BEOP, however, library users directly specify the real sampling points to inform the FIBER optimization system, even though BEOP has the same $PP$ parameters with respect to IOL. This is because library users know the real numbers of $PP$, such as problem size $n$. The accuracy of parameters estimated in BEOP, hence, is better than that of IOP. BEOP can be used in processes which need highly accurate estimated parameters.

### 4.5.1 Experiment of The Effect of Eigensolver Parameters

In this section, the FIBER BEOL is evaluated on the three kinds of parallel computers.

[**Experiment 2**] Evaluate the effect of BEOL in Situation 1, where library users know the real problem sizes to execute. For example, the problem sizes are 512, 5123, and 6123, which are not included in the sampling points of $n$.

Figures 7 and 8 show the IOL and BEOL effects in this experiment. The following can be pointed out from the results.

- IOL estimated parameters are not always optimal, but they are sufficient in many cases. This indicates that the 5th-order polynomial function is a sufficient estimation in a sense.
- BEOL effects are about 10%–50% speedups compared to the IOL estimated parameters.
- We found, however, the case that IOL estimations totally failed. In this case, a speedup factor of 3.4 for IOL estimated parameters is obtained.

The above points indicate that BEOL is a needed facility for the following reasons: (1) BEOL can improve the speed by about 10%–50% compared to IOL optimization results; (2) To avoid the failures of IOL estimation, BEOL should be performed. This also indicates that BEOL can assure users of library performance.

### 4.5.2 Experiment of The Effect of Algorithm Selection

[**Experiment 3**] Let the library users know the information that the target matrix coefficients are not changed. In this situation, evaluate the effect of the FIBER BEOL for the parameters of `kort`.

The following is an example of BEOL optimization, where a user wants to solve the eigenvalues and eigenvectors of a Frank matrix order 10,000 by using the HITACHI SR8000/MPP. Tables 4 and 5 show the results for the execution time and accuracy of the eigenvector with different parameters of `kort`.

Table 4 shows that the execution time differed according to the re-orthogonalization methods in the Frank matrix. Taking into account the numerical stability, the MG-S method was selected as the *de facto* parameter in several libraries.

**Table 1: IOL Effect on the Three Kinds of Parallel Machines (Householder Tridiagonalization Routine.) The default parameter is (0,8,6). One second is the unit for execution time.**

(a) HITACHI SR8000/MPP

(a-1) Compiler Option (*-opt=4*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.027 | 0.028 (0,8,14) | 0.026 (0,4,4) | 1.02 | 1.08 |
| 200 | 0.062 | 0.064 (0,8,8) | 0.059 (0,4,4) | 1.06 | 1.08 |
| 400 | 0.170 | 0.178 (0,8,1) | 0.166 (0,11,11) | 1.02 | 1.07 |
| 1000 | 1.05 | 1.23 (0,8,1) | 1.00 (0,13,13) | 1.04 | 1.23 |
| 6000 | 141 | 176 (0,8,1) | 130 (0,16,16) | 1.08 | 1.35 |

(a-2) Compiler Option (*-opt=0*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.036 | 0.040 (0,8,1) | 0.035 (0,4,4) | 1.02 | 1.14 |
| 200 | 0.129 | 0.169 (0,8,1) | 0.128 (0,6,6) | 1.00 | 1.31 |
| 400 | 0.793 | 1.28 (0,8,1) | 0.777 (0,9,9) | 1.02 | 1.65 |
| 1000 | 11.0 | 25.9 (0,8,1) | 10.8 (0,6,6) | 1.01 | 2.38 |
| 2000 | 83.1 | 207 (0,8,1) | 81.8 (0,6,6) | 1.01 | 2.53 |

(b) Fujitsu VPP800/63

(b-1) Compiler Option (*-O5*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.011 | 0.012 (0,15,2) | 0.011 (1,9,9) | 1.02 | 1.04 |
| 200 | 0.025 | 0.027 (0,9,10) | 0.024 (1,7,7) | 1.02 | 1.09 |
| 400 | 0.058 | 0.063 (0,1,8) | 0.056 (1,10,10) | 1.03 | 1.12 |
| 1000 | 0.227 | 0.265 (0,1,10) | 0.213 (1,16,16) | 1.06 | 1.24 |
| 6000 | 18.6 | 21.7 (0,1,2) | 18.2 (1,16,16) | 1.02 | 1.18 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.073 | 0.081 (0,8,16) | 0.072 (1,5,5) | 1.01 | 1.12 |
| 200 | 0.490 | 0.547 (0,8,16) | 0.485 (1,10,10) | 1.01 | 1.12 |
| 400 | 3.68 | 4.11 (0,8,16) | 3.65 (1,15,15) | 1.01 | 1.12 |
| 1000 | 56.0 | 62.5 (0,8,16) | 55.4 (1,15,15) | 1.01 | 1.12 |
| 2000 | 448 | 498 (0,8,16) | 442 (1,16,16) | 1.01 | 1.12 |

(c) PC Cluster

(c-1) Compiler Option (*-fast*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.097 | 1.32 (0,8,1) | 0.094 (1,13,13) | 1.03 | 14.06 |
| 200 | 0.235 | 0.762 (0,11,3) | 0.231 (0,11,11) | 1.01 | 3.28 |
| 400 | 1.97 | 2.41 (0,1,4) | 0.685 (0,10,10) | 2.88 | 3.51 |
| 1000 | 7.23 | 8.12 (0,8,13) | 3.36 (1,4,4) | 2.15 | 2.41 |
| 10000 | 1352 | 1552 (0,8,13) | 1076 (1,5,5) | 1.25 | 1.44 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (`ictr`,`imv`,`iud`) | Best (`ictr`,`imv`,`iud`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.100 | 0.515 (0,8,1) | 0.095 (1,13,13) | 1.04 | 5.39 |
| 200 | 0.248 | 0.875 (0,8,11) | 0.235 (1,8,8) | 1.05 | 3.72 |
| 400 | 0.807 | 1.29 (1,3,16) | 0.777 (0,8,8) | 1.03 | 1.66 |
| 1000 | 5.22 | 6.38 (0,14,6) | 5.08 (0,8,8) | 1.02 | 1.25 |
| 2000 | 26.6 | 29.8 (0,8,12) | 25.8 (0,6,6) | 1.02 | 1.15 |

**Table 2: IOL Effect on the Three Kinds of Parallel Machines (Householder Inverse Transformation Routine.) The default parameter is (1,1). One second is the unit for execution time.**

(a) HITACHI SR8000/MPP

(a-1) Compiler Option (*-opt=4*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.002 | 0.007 (3,3) | 0.002 (3,1) | 1.20 | 3.23 |
| 200 | 0.012 | 0.019 (8,3) | 0.008 (8,2) | 1.42 | 2.28 |
| 400 | 0.088 | 0.088 (1,1) | 0.057 (10,1) | 1.55 | 1.55 |
| 1000 | 1.16 | 1.16 (1,1) | 0.736 (5,2) | 1.58 | 1.58 |
| 6000 | 228 | 319 (15,3) | 158 (15,1) | 1.44 | 2.01 |

(a-2) Compiler Option (*-opt=0*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.030 | 0.030 (1,1) | 0.011 (13,1) | 2.64 | 2.64 |
| 200 | 0.126 | 0.126 (1,1) | 0.071 (5,1) | 1.76 | 1.76 |
| 400 | 2.00 | 2.00 (1,1) | 0.562 (10,1) | 3.57 | 3.57 |
| 1000 | 42.1 | 42.1 (1,1) | 9.40 (5,2) | 4.47 | 4.47 |
| 2000 | 334 | 334 (1,1) | 74.8 (5,2) | 4.47 | 4.47 |

(b) Fujitsu VPP800/63

(b-1) Compiler Option (*-O5*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.004 | 0.004 (1,1) | 0.002 (12,2) | 1.72 | 1.72 |
| 200 | 0.015 | 0.015 (1,1) | 0.007 (16,2) | 1.97 | 1.97 |
| 400 | 0.057 | 0.057 (1,1) | 0.025 (16,2) | 2.22 | 2.22 |
| 1000 | 0.409 | 0.409 (1,1) | 0.195 (13,2) | 2.09 | 2.09 |
| 6000 | 43.3 | 43.3 (1,1) | 26.2 (14,3) | 1.65 | 1.65 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.081 | 0.081 (1,1) | 0.063 (12,2) | 1.29 | 1.29 |
| 200 | 0.626 | 0.626 (1,1) | 0.479 (10,2) | 1.30 | 1.30 |
| 400 | 4.93 | 4.93 (1,1) | 3.76 (14,3) | 1.31 | 1.31 |
| 1000 | 76.4 | 76.4 (1,1) | 58.3 (13,3) | 1.30 | 1.30 |
| 2000 | 612 | 612 (1,1) | 467 (15,1) | 1.31 | 1.31 |

(c) PC Cluster

(c-1) Compiler Option (*-fast*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.086 | 0.250 (3,1) | 0.070 (13,3) | 1.24 | 3.57 |
| 200 | 0.265 | 0.337 (2,1) | 0.265 (1,1) | 1.00 | 1.27 |
| 400 | 1.06 | 1.12 (16,1) | 1.06 (1,1) | 1.00 | 1.06 |
| 1000 | 7.30 | 7.99 (11,1) | 7.30 (1,1) | 1.00 | 1.09 |
| 10000 | 1874 | 2876 (15,1) | 1871 (1,3) | 1.00 | 1.53 |

(c-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (`hitk`,`ichit`) | Best (`hitk`,`ichit`) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.085 | 0.109 (2,3) | 0.072 (2,2) | 1.19 | 1.51 |
| 200 | 0.288 | 0.32 (10,1) | 0.285 (2,1) | 1.01 | 1.13 |
| 400 | 1.24 | 1.32 (2,1) | 1.20 (11,1) | 1.03 | 1.09 |
| 1000 | 9.69 | 9.69 (1,1) | 9.01 (3,1) | 1.07 | 1.07 |
| 2000 | 52.8 | 52.8 (1,1) | 47.0 (3,3) | 1.12 | 1.12 |

**Table 3: IOL Effect on the Three Kinds of Parallel Machines (QR Decomposition Routine.) The default parameter is (4,4,8,4,8). One second is the unit for execution time.**

(a) HITACHI SR8000/MPP

(a-1) Compiler Option (*-opt=4*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.002 | 0.003 (16,4,8,4,8) | 0.002 | 1.00 | 1.54 |
| 200 | 0.014 | 0.019 (6,1,2,1,8) | 0.012 (6,1,2,3,3) | 1.14 | 1.60 |
| 400 | 0.103 | 0.139 (6,2,2,1,16) | 0.081 (6,2,2,4,6) | 1.27 | 1.72 |
| 1000 | 1.50 | 2.06 (8,1,4,1,16) | 1.08 (8,1,4,4,4) | 1.38 | 1.89 |
| 6000 | 379 | 405 (16,2,1,4,1) | 231 (16,2,1,4,4) | 1.63 | 1.74 |

(a-2) Compiler Option (*-opt=0*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.022 | 0.031 (8,1,3,2,16) | 0.017 (8,1,3,3,4) | 1.27 | 1.77 |
| 200 | 0.150 | 0.216 (8,1,2,1,16) | 0.083 (8,1,2,4,8) | 1.80 | 2.60 |
| 400 | 1.21 | 1.66 (8,1,2,1,16) | 0.583 (8,1,2,4,8) | 2.08 | 2.85 |
| 1000 | 17.9 | 25.6 (8,1,3,1,16) | 7.57 (8,1,3,4,8) | 2.37 | 3.38 |
| 2000 | 147 | 218 (16,4,3,3,1) | 68.7 (16,4,3,4,16) | 2.14 | 3.17 |

(b) Fujitsu VPP800/63

(b-1) Compiler Option (*-O5*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.002 | 0.004 (1,4,8,4,8) | 0.001 (8,1,3,4,8) | 1.61 | 2.78 |
| 200 | 0.008 | 0.011 (8,1,3,1,16) | 0.004 (8,1,3,4,8) | 1.96 | 2.70 |
| 400 | 0.034 | 0.052 (8,4,4,1,16) | 0.015 (8,4,4,4,8) | 2.28 | 3.45 |
| 1000 | 0.302 | 0.388 (16,2,6,2,1) | 0.153 (16,2,6,4,8) | 1.96 | 2.52 |
| 6000 | 48.2 | 67.3 (8,1,3,1,16) | 28.6 (8,1,3,3,8) | 1.68 | 2.35 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.154 | 0.234 (8,1,3,1,16) | 0.078 (8,1,3,4,8) | 1.97 | 2.98 |
| 200 | 1.20 | 1.82 (8,1,3,1,16) | 0.538 (8,1,3,4,8) | 2.24 | 3.39 |
| 400 | 9.37 | 14.5 (8,1,3,1,16) | 3.60 (8,1,3,4,8) | 2.60 | 4.03 |
| 500 | 18.3 | 28.4 (8,1,3,1,16) | 7.16 (8,1,3,4,8) | 2.56 | 3.96 |
| 900 | 106 | 165 (8,1,4,1,16) | 38.7 (8,1,4,4,8) | 2.75 | 4.27 |

(c) PC Cluster

(c-1) Compiler Option (*-fast*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.016 | 0.710 (6,1,5,1,3) | 0.015 (6,1,5,4,3) | 1.04 | 46.2 |
| 200 | 0.058 | 0.142 (3,1,3,4,16) | 0.056 (3,1,3,4,8) | 1.02 | 2.50 |
| 400 | 0.225 | 1.20 (3,3,16,4,16) | 0.219 (3,3,16,4,3) | 1.03 | 5.50 |
| 1000 | 2.07 | 16.0 (6,4,2,4,16) | 1.54 (6,4,2,4,5) | 1.34 | 10.3 |
| 10000 | 2016 | 16619 (8,4,4,4,16) | 1268 (8,4,4,4,4) | 1.59 | 13.1 |

(c-2) Compiler Option (*-O0*)

| Dim. | Default | Worst (ibl,iop,isp,ioo,iso) | Best (ibl,iop,isp,ioo,iso) | Default/Best | Best/Worst |
|---|---|---|---|---|---|
| 100 | 0.016 | 0.070 (1,4,8,4,8) | 0.016 (6,4,8,3,3) | 1.03 | 4.39 |
| 200 | 0.060 | 0.077 (1,4,8,4,8) | 0.059 (6,1,4,3,5) | 1.02 | 1.30 |
| 400 | 0.393 | 0.482 (1,4,8,4,8) | 0.246 (16,1,5,4,4) | 1.59 | 1.95 |
| 1000 | 5.76 | 6.90 (8,3,3,1,16) | 1.97 (8,3,3,4,8) | 2.92 | 3.50 |
| 2000 | 44.2 | 56.9 (8,3,3,1,16) | 16.2 (8,3,3,4,8) | 2.73 | 3.51 |

**Table 4: Execution time of each re-orthogonalization method in the inverse iteration method. (HITACHI SR8000/MPP, $n = 10,000$). The unit is second. The notation of $>$ means the iteration did not converge within the limitation for execution in the supercomputer environments.**

| #PEs (nprocs) | CG-S | MG-S | IRCG-S | NoOrt |
|---|---|---|---|---|
| 8 | 6,604 | 38,854 | 12,883 | 23 |
| 16 | 3,646 | 24,398 | 6,987 | 12 |
| 32 | 2,061 | 28,050 | 3,906 | 7 |
| 64 | 1,633 | 27,960 | 3,059 | 3 |
| 128 | 2,091 | $>$ | 3,978 | 1 |

**Table 5: The accuracy of eigenvectors calculated with each re-orthogonalization method in the inverse iteration method. (HITACHI SR8000/MPP, $n = 10,000$). The unit is the norm of Frobenius.**

| #PEs (nprocs) | CG-S | MG-S | IRCG-S | NoOrt |
|---|---|---|---|---|
| 8 | 6.4E-13 | 6.6E-13 | 6.4E-13 | 1.4 |
| 16 | 6.6E-13 | 6.6E-13 | 6.6E-13 | 1.4 |
| 32 | 6.8E-13 | 6.6E-13 | 6.8E-13 | 1.4 |
| 64 | 9.4E-13 | 6.6E-13 | 9.4E-13 | 1.4 |
| 128 | 1.5E-12 | – | 1.5E-12 | 1.4 |

If library users can specify the accuracy of eigenvectors in this situation, i.e., less than $1.5E-12$, the system can determine the suitable re-orthogonalization method. In this case, CG-S was the best parameter. Consequently, BEOL can determine the parameter of `kort` as CG-S in this situation, using the value of accuracy from library users. The accuracy of eigenvectors and the code of algorithm selection in the re-orthogonalization methods can be implemented by the selection instruction operation in Section 3.1.

Figure 6 shows the speedup ratio compared to the case of the normal default parameter (the MG-S method) in Experiment 3.

**Table 6: Speedup ratio to specified normal default parameter of the MG-S method, when library users specify the accuracy of eigenvectors as less than $1.5E-12$ in Experiment 3. (HITACHI SR8000/MPP)**

| #PEs (nprocs) | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| Speedup | 5.8 | 6.6 | 13.6 | 17.1 | – |

Figure 6 indicates that we can obtain 5.8–17.1 times speedups in this case. This is a typical case for applying the BEOL optimization in FIBER. This speedup is crucial; hence, we can conclude that the FIBER BEOL is an important function for optimizing algorithms according to the user's knowledge.

## 5. RELATED WORK

We can classify the conventional auto-tuning software into the following three categories.

*Complete Run-time Optimization Software*: In this category, the software performs the parameter adjustments at run-time. For example, to tune computer system parameters such as I/O buffer size, Active Harmony[12] and Autopilot[9] can be used.

The SANS [3] project provides a framework based on run-time optimization by a network agent[1].

*Complete Install-time Optimization Software*: In this category, the software performs the parameter adjustments at their install-time. For example, PHiPAC [2], ATLAS and the paradigm of AEOS (Automated Empirical Optimization of Software) [1, 13], and FFTW[4] can automatically tune the performance parameters of their routines when they are installed.

For formalization of an auto-tuning facility in this category, Naono and Yamamoto formulated the install-time optimization in the SIMPL [8] auto-tuning software framework, which is a paradigm for parallel numerical libraries. For reducing the search time, a theory for optimal parameters in an eigensolver was studied by Imamura and Naono [5].

*Hybrid Install-time and Run-time Optimization Software*: In ILIB [6, 7], the facility of install-time and run-time optimizations is implemented.

The concepts of the execute-time optimization layer with the user's knowledge, in order to improve parameter accuracy and to generalize auto-tuning facilities, are not clear and rarely discussed in the conventional auto-tuning software mentioned above. We therefore believe that BEOL in FIBER[11] is a very new concept.

## 6. CONCLUSION

In this paper, we evaluated an optimization layer with the user's knowledge for numerical software. The originality of the FIBER framework lies in its innovative optimization layer, the BEOL (Before Execute-time Optimization Layer). The experiment for BEOL indicated that the speed increased 3.4 times in eigensolver parameters and 17.1 times in the algorithm selection of orthogonalization compared to conventional optimization layers of installation in a computation kernel for the eigensolver. In addition, BEOL can guarantee the performance in a sense.

The key feature of the FIBER framework is how it determines the cost definition function $F$ according to the characteristics of libraries, sub-routines, or other parts of a program. Moreover, to extend the adaptation of auto-tuning and to obtain high quality in the estimated parameters, a more sophisticated method is needed. Evaluation of the cost definition function and extending the adaptation of FIBER will be important future work.

A parallel eigensolver, named `ABCLibDRSSED`, has been developed which contains a part of the FIBER BEOL facility. The source code and manual for the alpha version are available at `http://www.abc-lib.org/`. The `ABCLibScript` language will be developed to support code generation, parameterization, and its registration for auto-tuning facilities based on the FIBER concept, as shown in Section 3.1.

[1] They also provide an install-time optimization scenario [3]. However, the agent approach is basically classified as run-time tuning, because it decides the appropriate parameters at run-time.

**Table 7: IOL and BEOL Effects on the Three Kinds of Parallel Machines for an Eigensolver. One second is the unit for execution time.**

(a) HITACHI SR8000/MPP

(a-1) Compiler Option (*-opt=4*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 0.607 | 0.588 | (0,14,6)(3,3) | 0.534 | (0,14,14)(8,2) | 1.13 | 1.10 |
| 5123 | 247 | 223 | (0,16,16)(2,2) | 195 | (0,15,15)(7,1) | 1.26 | 1.14 |
| 6123 | 485 | 370 | (0,16,16)(15,1) | 331 | (0,15,4)(16,1) | 1.46 | 1.11 |

(a-2) Compiler Option (*-opt=0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 9.15 | 3.73 | (0,14,6)(3,3) | 3.42 | (0,5,5)(16,2) | 2.67 | 1.09 |
| 1234 | 102 | 41.8 | (0,13,16)(5,2) | 40.8 | (0,6,14)(5,2) | 2.50 | 1.02 |
| 2345 | 731 | 270 | (0,13,16)(5,2) | 273 | (0,13,14)(7,2) | 2.67 | 0.98 |

(b) Fujitsu VPP800/63

(b-1) Compiler Option (*-O5*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 0.815 | 0.771 | (1,2,1)(16,3) | 0.757 | (1,10,9)(16,2) | 1.07 | 1.01 |
| 5123 | 71.8 | 60.2 | (1,16,2)(14,3) | 60.3 | (1,16,2)(14,1) | 1.19 | 0.99 |
| 6123 | 110 | 92.01 | (1,16,2)(14,3) | 91.9 | (1,16,4)(14,3) | 1.19 | 1.00 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 123 | 0.653 | 0.620 | (1,9,8)(14,1) | 0.616 | (1,9,9)(15,2) | 1.06 | 1.00 |
| 512 | 20.4 | 18.1 | (0,11,9)(12,3) | 17.9 | (1,10,9)(16,3) | 1.13 | 1.01 |
| 912 | 107 | 93.9 | (1,15,9)(13,3) | 93.6 | (1,16,9)(12,2) | 1.14 | 1.00 |

(c) PC Cluster

(c-1) Compiler Option (*-fast*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 2.88 | 3.32 | (1,10,4)(1,2) | 2.68 | (1,5,2)(1,1) | 1.07 | 1.00 |
| 5123 | 396 | 359 | (1,5,2)(1,3) | 366 | (0,5,1)(1,2) | 1.08 | 0.98 |
| 10123 | 2804 | 2497 | (1,5,2)(1,3) | 2526 | (1,5,3)(1,2) | 1.11 | 0.98 |

(c-2) Compiler Option (*-O0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL-Optimized Time [s] | Parameters (ictr,imv,iud) (hitk,ichit) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 3.55 | 3.45 | (0,13,11)(7,1) | 3.31 | (1,13,3)(3,1) | 1.07 | 1.04 |
| 1234 | 17.6 | 19.00 | (0,13,8)(14,1) | 16.7 | (1,5,8)(4,3) | 1.05 | 1.13 |
| 2345 | 97.4 | 98.6 | (1,14,15)(4,3) | 84.5 | (0,6,6)(4,3) | 1.15 | 1.16 |

**Table 8: IOL and BEOL Effects on the Three Kinds of Parallel Machines for a QR Decomposition Routine. One second is the unit for execution time.**

(a) HITACHI SR8000/MPP

(a-1) Compiler Option (*-opt=4*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 0.217 | 0.290 | (6,4,8,4,8) | 0.171 | (8,2,2,4,4) | 1.26 | 1.69 |
| 5123 | 391 | 149 | (16,4,8,4,8) | 146 | (8,4,1,4,4) | 2.67 | 1.02 |
| 6123 | 762 | 270 | (16,4,8,4,8) | 276 | (8,2,5,2,8) | 2.76 | 0.97 |

(a-2) Compiler Option (*-opt=0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 2.42 | 1.05 | (8,4,8,4,8) | 0.982 | (8,1,3,4,8) | 2.45 | 1.06 |
| 1234 | 33.9 | 15.2 | (8,4,8,4,8) | 16.9 | (8,2,3,3,8) | 2.00 | 0.89 |
| 2345 | 240 | 119 | (16,4,8,4,8) | 114 | (8,2,5,4,8) | 2.10 | 1.04 |

(b) Fujitsu VPP800/63

(b-1) Compiler Option (*-O5*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 0.061 | 0.026 | (16,4,8,4,8) | 0.026 | (8,1,5,4,8) | 2.34 | 1.00 |
| 5123 | 31.3 | 19.0 | (8,4,8,4,8) | 18.2 | (16,4,5,1,16) | 1.71 | 1.04 |
| 6123 | 51.4 | 30.6 | (16,4,8,4,8) | 30.2 | (8,1,1,3,8) | 1.70 | 1.01 |

(b-2) Compiler Option (*-O0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 123 | 0.292 | 0.290 | (3,4,8,4,8) | 0.152 | (8,1,3,3,4) | 1.92 | 1.90 |
| 512 | 19.6 | 7.11 | (8,4,8,4,8) | 7.05 | (8,1,3,4,8) | 2.78 | 1.00 |
| 912 | 110 | 40.6 | (8,4,8,4,8) | 40.5 | (8,1,4,4,8) | 2.71 | 1.00 |

(c) PC Cluster

(c-1) Compiler Option (*-fast*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 0.520 | 0.590 | (6,4,8,4,8) | 0.466 | (3,1,2,1,3) | 1.11 | 1.26 |
| 5123 | 261 | 151 | (8,4,8,4,8) | 152 | (8,3,4,4,8) | 1.71 | 0.99 |
| 10123 | 2017 | 2079 | (8,4,8,4,8) | 1320 | (16,4,16,3,4) | 1.52 | 1.57 |

(c-2) Compiler Option (*-O0*)

| Dim. | Default Parameters Time[s] | IOL-Estimated Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL-Optimized Time [s] | Parameters (ibl,iop,isp, ioo,iso) | BEOL Effect (Def/BEOL) | BEOL Effect (IOL/BEOL) |
|---|---|---|---|---|---|---|---|
| 512 | 1.11 | 0.523 | (16,4,8,4,8) | 0.532 | (16,2,3,4,16) | 2.08 | 0.98 |
| 1234 | 12.7 | 3.60 | (8,4,8,4,8) | 3.59 | (8,2,6,4,8) | 3.53 | 1.00 |
| 2345 | 86.0 | 87.3 | (6,4,8,4,8) | 25.4 | (8,3,1,4,8) | 3.38 | 3.43 |

## 7. ACKNOWLEDGMENTS

This study was partially supported by PRESTO, a Japan Science and Technology agency (JST).

## 8. ADDITIONAL AUTHORS

Additional authors: Toshitsugu Yuba (Graduate School of Information Systems, The University of Electro-Communications 1-5-1 Choufu-gaoka, Choufu-shi, Tokyo 182-8585, Japan, email: yuba@is.uec.ac.jp)

## 9. REFERENCES

[1] ATLAS project; http://www.netlib.org/atlas/index.html.

[2] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. *Proceedings of International Conference on Supercomputing 97*, pages 340–347, 1997.

[3] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications. *The International Journal of High Performance Computing and Applications*, 17(2):125–131, 2003.

[4] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, Atlanta, Georgia, May 1999.

[5] T. Imamura and K. Naono. An evaluation towards an automatic tuning eigensolver with performance stability. In *Proceedings of Symposium on Advanced Computing Systems and Infrastructures (SACSIS)2003*, pages 145–152, 2003.

[6] T. Katagiri, H. Kuroda, K. Ohsawa, M. Kudoh, and Y. Kanada. Impact of auto-tuning facilities for parallel numerical library. *IPSJ Transaction on High Performance Computing Systems*, 42(SIG 12 (HPS 4)):60–76, 2001.

[7] H. Kuroda, T. Katagiri, and Y. Kanada. 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*, 2281:628–639, 2002.

[8] K. Naono and Y. Yamamoto. A framework for development of the library for massively parallel processors with auto-tuning function and with the single memory interface. *IPSJ SIG Notes*, (2001-HPC-87):25–30, 2001.

[9] R. L. Ribler, H. Simitci, and D. A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems, special issue (Performance Data Mining)*, 18(1):175–187, 2001.

[10] K. Takahiro, K. Kise, H. Honda, and T. Yuba. FIBER: A general framework for auto-tuning software. *Proceedings of The Fifth International Symposium on High Performance Computing*, Springer Lecture Notes in Computer Science(2858):146–159, 2003.

[11] K. Takahiro, K. Kise, H. Honda, and T. Yuba. FIBER: A framework of installation, before execution-invocation, and run-time optimization layers for auto-tuning software. *IS Technical Report, Graduate School of Information Systems, The University of Electro-Communications*, UEC-IS-2003-3, May 2003.

[12] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active Harmony : Towards automated performance tuning. In *Proceedings of High Performance Networking and Computing (SC2002)*, Baltimore, USA, November 2003.

[13] R. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27:3–35, 2001.