

A Study on Large Scale Eigensolvers
for Distributed Memory Parallel Machines
分散メモリ型並列計算機における大規模固有値ソルバの研究

by
KATAGIRI Takahiro
片桐 孝洋

A Dissertation

Submitted to
the Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

December 2000

ABSTRACT

Information retrieval is one of the important processes today because of the internet technologies progress. Many people have regarded such retrieving process as non-numerical computation. However, one points out that the retrieving process demands an eigenvalue computation to find knowledge from the retrieved information by M.Berry *et.al.*. Thus, one can say that the process of eigenvalue computation is not only an important process on the conventional limited scientific and technical fields, but is also a basic process in the field of general information processing. Considering its computational scale and memory usage, the eigenvalue computation is always a large-scale problem, since databases were retrieved on the huge internet, and scientific and technical problems require computation of maximum performance and memory usage. From these reasons, our attention is always focused on solving eigenvalue problems at high speed with very large memory. On the other hand, today's high speed computers which can perform large-scale computations, such as super-computers, are distributed parallel computers. Therefore, to solve the large-scale eigenvalue problems, we should parallelize the eigenvalue computation considering algorithm and memory usage.

From these requests, large-scale parallel eigensolvers are focused on in this thesis. The "large-scale" means that it requires not only huge computations of memory spaces, but it also requires a large number of processor elements. This requires "massive parallel" processing. On the other hand, to compute eigenvectors corresponding clustered eigenvalues, most of all parallel execution time is consumed in the orthogonalization process. Therefore developing new orthogonalization algorithm is needed to manage "large-scale" eigenvectors corresponding clustered eigenvalues. Next is the management of huge codes to attain high performance. This is a time-consuming work. To remove this work, auto-tuning methodology is a key concept to manage "large-scale" codes.

This thesis is divided into four parts from the above viewpoints. In part I of this thesis, problems from the usage of parallel distributed memory parallel machines and how to develop parallel libraries are pointed out. In part II of this thesis, several parallel algorithms for eigensolvers are proposed and evaluated. The former part of part II proposes and evaluates a high efficiency parallel similarity transformation algorithm. The latter part of part II is for a new parallel orthogonalization algorithm for computing eigenvectors according to clustered eigenvalues. In part III of this thesis, an optimization feature for parallel processing is firstly proposed. To show effectiveness of the feature, it is evaluated by using several parallel machines. Finally, part IV of this thesis summaries the findings of part II and part III. Part IV also includes a discussion for adaption of proposed algorithm of part II. In addition, an advice to symmetric eigensolver users is described.

In part I of this thesis, the problems from the usage of distributed memory parallel machines are described. The problems are as follows: (1) Load imbalance for parallel

blocked algorithms based on sequential blocked algorithms; (2) Additional communications by using the symmetry of matrices; (3) Reduced communication method according to two dimensional data distribution; To solve these problems, a parallel algorithm is proposed in part II. On the other hand, the problems from how to develop parallel libraries are shown. The problems are as follows: (1) Coding time to attain high performance; (2) Ill-specified parameters by users; To solve these problems, an optimization feature is proposed in part III.

In part II of this thesis, a new algorithm of the Householder-Bisection method is proposed. This method is for dense symmetric eigenvalue computation. The Householder tridiagonalization is used in this method. In conventional algorithms, there was a problem that the communication time of the tridiagonalization increased when the number of processors increased. To solve this communication problem, a reduced communication algorithm is proposed in this thesis. In the tridiagonalization, an implementation technique for sequential processing, which can improve computation efficiency, has been used. The technique is called *blocking*, and the algorithm implemented the blocking technique is called *blocked algorithm*. For the parallelization of the blocked algorithms, distributing matrix data is needed. This distribution is performed as a continuous block, and this distribution is called *blocking data distribution*. Many conventional parallel blocked algorithms have been constructed by using the equivalent block lengths for the blocking algorithm and blocking data distribution. From this reason, they caused heavy load imbalance when the problem size was small or when in massive parallel processing. To solve this load balancing problem, we propose a new concept for developing parallel blocked algorithms. That is, different block lengths should be used in the parallel blocked algorithms. When this new concept was evaluated by using a distributed memory parallel machine, the maximal speed-up factor of about 5 times was obtained in comparison with a ScaLAPACK's routine, which is widely-used as a parallel library. For a real application, the small eigenproblem of 572 dimension is needed to solve more than 6000 times. By applying the proposed algorithm, more than 90% parallel efficiency is attained in 8 processors of a PC cluster.

Next, we focus the parallelization of the Gram-Schmidt orthogonalization method, which is widely-used in the computations of linear algebra. When computing eigenvalues, this orthogonalization method is used for computing eigenvectors corresponding to clustered eigenvalues. For the conventional parallel methods, there was a trade-off problem between accuracy and parallelism in the orthogonalization process of eigenvalue computation. To solve the trade-off problem, this orthogonalization was classified as "QR decomposition" and "re-orthogonalization." From this classification, parallel orthogonal algorithms can be classified by data distribution, and this is a new classification in the parallel orthogonal algorithms. For the QR decomposition process, we propose a new data distribution method for massive parallel processing (MPP). By using the new distribution method, in some cases a load imbalance is improved in MPP. For the re-orthogonalization

process, we propose a new orthogonalization method by applying a sorting process to the re-orthogonalization. The results of numerical experiments indicated that (1) this new orthogonalization method improves parallel performance and orthogonal accuracy; (2) it reduces iterative times until a routine converges; in an eigenvector computation routine. Therefore, these results show that the new re-orthogonalization method not only can solve the trade-off problem of performance and accuracy, but it can also be a high-speed eigenvector computation routine. In addition, we find that “dividing the computation” gives us higher accuracy. Therefore, large-scale computations should be performed in parallel machines in the viewpoints of execution time, memory usage and its accuracy.

In part III of this thesis, a new concept of parallel numerical software, named “optimization feature before execution” is proposed. To show the effectiveness of the optimization feature before execution, we implement a parallel eigensolver with the optimization feature before execution to HITACHI’s distributed memory parallel machines, and evaluate the performance. The optimization methods in the eigensolver are selecting the implementation of global reduction operation, the implementation of loop-unrollings for matrix-vector products and matrix updating, and the block length for communication in a blocked algorithm. We implement the optimization feature to the Householder tridiagonalization routine, Householder inverse iteration routine, and modified Gram-Schmidt orthogonalization routine. The kernels of these processes are composed of different types of computations, named BLAS1 (Basic Linear Algebra Subprograms level 1), BLAS2, and BLAS3, which are basic kernels of the computation of linear algebra. Therefore the eigensolver is a good benchmark of real application software in a sense. The optimization feature proposed by the author is a different method from the conventional methods. Our feature can perform “global optimizations” between computations and communications, while the conventional methods only perform “local optimizations” of BLAS. By using our feature, much higher performance can be attained compared to the conventional “local optimization” methods for BLAS. The proposed optimization method is rather simple because it selects the fastest routine by measuring the execution time of prepared routines. The physically separated routines can not affect the performance in theory, then it does not search every combination of the routines. By using the simple method, 1.1 – 2.3 speed-ups in the HITACHI SR2201 and SR8000 can be obtained. This results indicate that the effectiveness of the optimization feature before the execution is high.

Next in part III is a discussion of how to implement the auto-tuning software, just like the proposed library with optimization feature before execution. To discuss the implementation, “High performance” auto-tuning software is defined as follows: (1) It is capable of finding a parameters set quickly which can execute the program fast; (2) Optimization time for finding better parameters set is also fast; (3) The longer optimization time was specified, the better parameters set is found. To attain the high performance auto-tuning software, four optimization methods are discussed. As a result of performance evaluation, we found that “incomplete method” is one of the useful methods. The “incomplete

method” shows that using the basic operations of linear algebra computations, and it uses a fact that many linear algebra computations are composed of basic operations. As the result of performance evaluation, the incomplete method is found that it is capable of reducing the optimization time with the factor of 74.1. This shows that about 33 hours optimization is reduced to about 1591 seconds, therefore, the incomplete method is a sufficient method. There is another proposal of a new category of BLAS, named *Intelligent BLAS (IBLAS)*. The category of IBLAS can be obtained by extending the concept of optimization feature before execution. By using IBLAS, the speed-up factor of 1.02–1.26 was obtained compared to the execution time of conventional BLAS. IBLAS can easily be used by the software which uses BLAS. Therefore, the portability of IBLAS is very high. Thus, IBLAS proposed in this thesis will be one of the useful methods to develop the high performance parallel numerical libraries.

Finally, part IV summarizes the findings in part II and part III in this thesis. The key concepts of the former part of part II are as follows: (1) Using the reduced communication method; (2) Improving load imbalance from block lengths of algorithm and data distribution; (3) Reducing communications by using the non-symmetry of matrices; By using these concepts, a highly parallel tridiagonalization routine can be obtained. The proposed algorithm is also useful when we solve small dimension problems and we are in MPP environments. This indicates, therefore, our algorithm will be a crucial in real applications. The latter part of part II proposes a new parallel orthogonalization method by changing computation order to remove the accuracy problem in conventional methods. From numerical experiments, we found that the changing computation order can modify accuracy in a certain condition. Therefore a generalization for the changing computation order, such as the optimization process in compilers, will be a key concept. Part III proposes the optimization feature before execution to manage large-scale codes. From performance evaluation, the feature is an effective technique from the viewpoints of high performance. We suggests, therefore, all of dense linear algebra libraries should contain the proposed optimization feature. Containing these features gives us high performance numerical libraries, even if we use poor parallel computing environments or inefficient optimization compilers or poor performance communication libraries. Part III also proposes a new category of BLAS, named IBLAS. The IBLAS can be regarded as a flexible BLAS, since it can change own implementations according to the size of the input matrix. The results of performance evaluation show that the conventional BLAS is not optimal. Basically, the interface of IBLAS is as same as that of BLAS. These results indicate that libraries which call the BLAS should call IBLAS from the viewpoint of the execution time.

1.2.2	Overview of software libraries	17
1.2.2.1	ScaLAPACK's PDSYEVX	17
1.2.2.2	HJS	18
1.2.2.3	PeIGs	18
1.2.2.4	HITACHI's MATRIX/MPP	19
1.2.2.5	IBM's parallel ESSL	19
1.2.2.6	NAG's Parallel Libraries	19
1.2.2.7	Fujitsu-ANU Parallel Mathematical Subroutine	19
1.2.2.8	ILIB	20
1.3	Organization of this thesis	21
2.	Distributed Memory Machines and Parallel Algorithms Basics	22
2.1	Introduction	22
2.2	Target architectures	22
2.2.1	SIMD and MIMD	22
2.2.2	Distributed memory organization	23
2.2.3	Interconnection network	23
2.3	Parallel linear algebra algorithm basics	23
2.3.1	What is speed-up	23
2.3.2	Load balancing and data distribution	24
2.3.2.1	One-dimensional data distribution	24
2.3.2.2	Two-dimensional data distribution	25
2.3.3	Data dependence	26
2.3.3.1	Flow, anti, and output dependencies	26
2.3.4	Typical communication time	27
2.3.4.1	Models for performance estimation	27
2.3.4.2	Broadcast	28
2.3.4.3	Reduction operations	28
2.3.4.4	All-to-all communication	29
2.4	Experiment environments	29
2.4.1	The HITACHI SR2201	29
2.4.1.1	Organization	29
2.4.1.2	Pseudo vectorization	29
2.4.2	The HITACHI SR8000	29
2.4.2.1	Organization	29
2.4.2.2	Intra-node and Inter-node parallel processing	30

2.4.2.3	Pseudo vectorization	30
---------	--------------------------------	----

II Algorithm of Parallel Dense Eigensolvers Part 31

3. High Performance Parallel Householder-Bisection Algorithm 32

3.1	Introduction	32
3.2	Parallel processing environment and several notations	33
3.3	Outline of our parallel eigenvalue computation process	34
3.3.1	Outline of the entire process	34
3.3.2	The tridiagonalization process	35
3.3.3	The re-distribution process	39
3.3.4	The eigenvalue computation process	39
3.4	Performance evaluation	39
3.4.1	Performances in the Frank matrix	39
3.4.1.1	Performance for calculating all eigenvalues	39
3.4.2	Comparison to the ScaLAPACK	41
3.4.2.1	Experimental results	41
3.4.2.2	Discussion	42
3.4.3	Comparison to other packages	47
3.5	Conclusion	47

4. New Parallel Orthogonalization Method 49

4.1	Introduction	49
4.2	Sequential algorithms of GS orthogonalization method	50
4.2.1	QR decomposition	50
4.2.2	Re-orthogonalization	53
4.3	Parallelization of GS orthogonalization method	53
4.3.1	QR decomposition	53
4.3.1.1	A case of row-wise distribution	53
4.3.1.2	A case of column-wise distribution	55
4.3.1.3	Proposition of new data distribution	57
4.3.2	Re-orthogonalization	59
4.3.2.1	A case of row-wise distribution	59
4.3.2.2	A case of column-wise distribution	59
4.4	New method for improving accuracy in the CGS method	60
4.5	Summary of parallel algorithms	61

4.6	Performance evaluation	61
4.6.1	Conditions of our experiments	62
4.6.1.1	Definition of orthogonality	62
4.6.1.2	Test matrices	64
4.6.1.3	Implementation details of the CGSS method	64
4.6.2	QR decomposition	65
4.6.2.1	A case of row-wise distribution	65
4.6.2.2	A case of column-wise distribution	67
4.6.3	Re-orthogonalization	69
4.6.3.1	A case of row-wise distribution	69
4.6.3.2	Distribution of dot-producted values	70
4.6.4	Evaluation with an application program	71
4.6.4.1	Orthogonality	71
4.6.4.2	The number of iterations untill convergence	72
4.7	Conclusion	72

III Application of Parallel Dense Eigensolvers Part 79

5.	Optimization Feature Before Execution 80
5.1	Introduction 80
5.2	Optimization feature before execution 81
5.2.1	Definition of optimization feature 83
5.2.2	Relation between algorithms and implementations 84
5.2.3	Relation between order notation and optimization feature 85
5.2.4	Aims of optimization feature 86
5.3	Dense symmetric eigensolver with an optimization feature 87
5.3.1	Entire process 87
5.3.2	Householder tridiagonalization 87
5.3.2.1	Parallel implementation of the Householder tridiagonal- ization 89
5.3.3	Householder inverse transformation routine 90
5.3.3.1	Parallel implementation of the Householder inverse trans- formation 91
5.3.4	MGS orthogonalization routine 92
5.3.4.1	Blocked MGS orthogonalization algorithm 92

5.3.4.2	Parallel implementation of blocked MGS orthogonalization algorithm	92
5.4	Method for searching parameters	93
5.4.1	Householder tridiagonalization routine	93
5.4.1.1	Parameter for the global summations	93
5.4.1.2	Parameter for the matrix-vector product	94
5.4.1.3	Parameter for the process to update	94
5.4.1.4	How to search these parameters	95
5.4.2	Householder inverse transformation routine	96
5.4.2.1	Parameter for the Householder inverse transformation kernels	96
5.4.2.2	How to search this parameter	97
5.4.3	MGS orthogonalization routine	97
5.4.3.1	Parameter for communication times	97
5.4.3.2	Parameter for the pivot PE kernel	98
5.4.3.3	Parameter for the inner PE kernel	99
5.4.3.4	How to search these parameters	100
5.5	Experimental results	101
5.5.1	Householder tridiagonalization routine	101
5.5.1.1	The results of the SR2201	102
5.5.1.2	The results of SR8000	106
5.5.2	Householder inverse transformation routine	106
5.5.2.1	The results of the SR2201	106
5.5.3	MGS orthogonalization routine	110
5.5.3.1	The results of the SR2201	110
5.6	Conclusion	110
6.	Towards High Performance Auto-tuning Software	113
6.1	Introduction	113
6.2	Methods for reducing optimization time	114
6.2.1	Using machine specific parameters	114
6.2.2	Using small sized problems	115
6.2.3	Using incomplete decompositions	115
6.2.4	Using decreased searches	115
6.2.5	Evaluation of the methods	116
6.2.5.1	Method for using small sized problems	116

6.2.5.2	Method for using incomplete decompositions	118
6.3	New category of a new basic linear algebra subprograms	120
6.3.1	Evaluation of IBLAS	121
6.3.1.1	Results	121
6.3.1.2	Discussion	121
6.3.2	Implementation and sampling tuned parameter methods for IBLAS	124
6.3.2.1	Experimental results	124
6.3.2.2	Discussion	125
6.4	Conclusion	126

IV Conclusion Remarks Part 128

7. Adaptation and Future Work 129

7.1	Introduction	129
7.2	Adaptability of each algorithms	130
7.2.1	The algorithm in box A	130
7.2.2	The algorithm in box B	130
7.2.3	The algorithm in box C	132
7.2.4	The algorithm in box D	132
7.3	Applications for real problems	133
7.3.1	An example : Chemical field	133
7.4	Future work	134
7.5	Conclusion	134

8. Summary and Conclusion 136

8.1	Conclusion of main parts	136
8.1.1	Algorithm of parallel dense eigensolver part	137
8.1.1.1	High performance parallel Householder-Bisection algortihm	137
8.1.1.2	New parallel orthogonalization method	138
8.1.2	Application of parallel dense eigensolver part	138
8.1.2.1	Optimized feature before execution	138
8.1.2.2	Towards high performance auto-tuning software	140
8.2	Advice to symmetric eigensolver users	141
8.3	Closing remarks	142

References 143

Appendix

A. Information of Our Eigensolver	153
A.1 Code line of ILIB_DRSSSED	153
A.2 Results of auto-tuning for ILIB_DRSSSED routine	153
A.2.1 ILIB_TriRed : a Householder tridiagonalization routine	153
A.2.1.1 Tuning information	153
A.2.1.2 Distribution of performance parameter	156
A.2.2 ILIB_MGSAO : a MGS orthogonalization routine	157
A.2.2.1 Tuning information	157
A.3 How to obtain the ILIB_DRSSSED routine	159
B. ILIB_TriRed Manual: Version 0.61	162
B.1 Introduction	162
B.1.1 Householder tridiagonalization algorithm	163
B.1.2 Parallelization of the tridiagonalization	164
B.2 Automatically tuned parameters	166
B.2.1 Loop unrolling for matrix-vector product	167
B.2.2 Loop unrolling for updating process	167
B.2.3 Communication method	167
B.2.4 Information file of automatically tuned parameters	167
B.3 Sample program	168
B.4 Example of execution	172
B.4.0.1 How to execute	173
B.4.0.2 Results of execution	173
B.5 Input output formats	174
B.5.1 ILIB_TriRed	174
B.5.2 Run-time option	174
B.5.3 Other routines	175
B.5.3.1 ILIB_TriRed_Init()	175
B.5.3.2 cid2xy2	175
B.5.3.3 MakeTestMat2	175
B.5.3.4 PBisect	176
B.5.3.5 ChkEigErr	177

C. Bibliography: List of Publications by the Author	179
C.1 Papers with referee	179
C.1.1 Paper (Journal)	179
C.1.2 Papers (Symposium)	179
C.2 Papers without referee	180
C.2.1 IPS.Japan SIG Notes	180
C.2.2 Other reports	180
C.3 Co-authored papers	181

List of Figures

1.1	Typical performance of blocked algorithms.	6
1.2	Load imbalance caused by data distribution methods.	7
1.3	An example for using symmetry and non-symmetry of matrix in a matrix-vector products.	8
1.4	Reduced communication method in 2-D grid-wise data distribution.	10
3.1	Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution).	36
3.2	Performances in parallel tridiagonalization (MFLOPS, the percentages in parentheses are percentages of the theoretical peak performance). Total calculation amount of $8/3n^3$ is used for computing the MFLOPS values.	41
3.3	Execution time for SLP TRD and Our TRD in the tridiagonalization (SR2201).	45
3.4	Speed-up ratios between SLP TRD and Our TRD for the tridiagonalization ($n = 8000$, SR2201).	46
3.5	Efficiency to the theoretical peak performance.	48
4.1	The MGS method in the QR decomposition.	50
4.2	Data dependence graph of the MGS method in the QR decomposition.	51
4.3	The CGS method in the QR decomposition.	52
4.4	Data dependence graph of the CGS method in the QR decomposition.	52
4.5	The MGS method in row-wise distribution.	53
4.6	The CGS method in row-wise distribution.	54
4.7	An example for parallel MGS algorithms with blocked communication in the QR decomposition.	56
4.8	An example for the data dependence graph of the MGS algorithm by using non-blocked communication.	57
4.9	The Cyclic Triangular Distribution (*, CTD).	58
4.10	An example of advantage for (*, CTD).	59

4.11	The CGSS method for re-orthogonalization.	61
4.12	Execution time in the QR decomposition. (The row-wise distribution, randomized matrix, $n = 1500$)	65
4.13	Execution time in the QR decomposition. (The row-wise distribution, randomized matrix, $n = 1500$. The quick sort is used.)	66
4.14	Orthogonalities in QR factorization. (randomized matrix)	67
4.15	(* , CTD) vs. (* , Cyclic) with the non-blocked MGS method.	68
4.16	(* , CTD(m)) vs. (* , Cyclic(m)) with the blocked MGS method.	69
4.17	Orthogonalities with MGS orthogonalized vectors in re-orthogonalization. (the randomized matrix, $n = 1500$)	74
4.18	Orthogonalities with MGS orthogonalized vectors in re-orthogonalization. (The Lauchili matrix, $n = 1500$, $\epsilon = 1$)	75
4.19	Distribution of values for dot-products in each orthogonalization methods. ($p = 16$)	76
4.20	Orthogonalities in inverse iteration using each orthogonalization method.	77
4.21	Average number (per one eigenvector) of iteration until converge.	78
5.1	The concept of optimization feature before execution.	82
5.2	Changing complexity order in an optimized routine.	86
5.3	The overall of optimization feature for our eigensolver.	88
5.4	Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution).	90
5.5	Parallel Householder inverse transformation.	92
5.6	Parallel blocked algorithm of MGS orthogonalization.	93
5.7	Execution time for the SLP TRD and TRD in the tridiagonalization (SR2201).	105
5.8	The execution time in every auto-tuning process. (SR2201, $n = 8000$)	106
6.1	How Intelligent BLAS (IBLAS) works.	121
6.2	Execution time of two kinds of implementations for IBLAS.	125
7.1	An overview of the computational and communication complexities for parallel Householder reduction.	131
A.1	Execution time of ILIB_TriRed. (SR2201, PE=4, $n = 4000$)	156
A.2	Execution time of ILIB_TriRed. (SR2201, PE=1024, $n = 8000$)	156
A.3	Distribution of performance parameters for ILIB_TriRed. (SR2201, PE=4)	157
A.4	Execution time of ILIB_MGSAO I. (SR2201, PE=16, $n = 1000$)	160
A.5	Execution time of ILIB_MGSAO II. (SR2201, PE=16, $n = 1000$)	161

B.1 The Householder tridiagonalization algorithm in the cyclic-cyclic distribution.178

Acknowledgments

I would like to express my sincere thanks to my research adviser Professor KANADA Yasumasa for his support and guidance throughout my master and doctoral degree studies. I would also like to thank to our laboratory's Associate Professor SATO Hiroyuki, research associate KURODA Hisayasu at Kanada laboratory and ex-research associate Dr. TAKAHASHI Daisuke who is now at Saitama University, for their constant inspiration in research and support for our computer environments. I would also like to thank to our laboratory secretary, Ms. KAMEDA Fumiyo, for her support to my research environments, and all students of KANADA laboratory at the Department of Information Science, the University of Tokyo who have given me advice and suggestions.

I would like to express my sincere thanks to Dr. Aad van der Steen of Utrecht University for giving me useful comments in my published papers. I would also like to thank to Mr. NAONO Ken and Mr. YAMAMOTO Yuusaku of Hitachi, Ltd., Central Research Laboratory for giving me several useful comments and discussions for parallel eigensolvers.

I would like to thank to my grandmother, the late Ms. KATAGIRI Tomiko for her support in my childhood, and my parents, Mr. KATAGIRI Yasutaka and Ms. KATAGIRI Machiko, for their support in every aspect and permit me to study at the graduate school, the University of Tokyo.

At last, but not least, I thank my wife Akiyo for her love and never ending support for my studies.

Part I

Introduction Part

Chapter 1

Introduction

1.1 Motivation : interesting observations

This thesis is focused on large scale eigensolvers. Here, the “large scale” means that it requires not only huge computations or memory spaces, but it also requires a large number of processor elements. Thus naturally it requires “massive parallel” processing. In addition, the author think that the “large scale” also means managing or developing huge codes to attain high performance.

In the viewpoint of dense eigensolvers, developing high efficient similarity transformation algorithms is important to accomplish “massive parallel” processing. On the other hand, to compute eigenvectors corresponding clustered eigenvalues, most of all parallel execution time is consumed in the orthogonalization process. Therefore developing new orthogonalization algorithm is needed to manage “large scale” eigenvectors corresponding clustered eigenvalues. Next point is the management of huge codes to attain high performance. This is a time-consuming work. To escape this work, auto-tuning methodology is a key concept to manage “large scale” codes.

Thus, the author will discuss “large scale” eigensolvers based on the following three viewpoints of “large scale” in this thesis: (1) solving large scale dense eigenproblem in massive parallel processing; (2) managing eigenvectors corresponding clustered eigenvalues in parallel processing; (3) managing huge codes to attain high performance.

1.1.1 Background

Information retrieval is one of the most important processes today because of the internet technologies progress. Many people have regarded such retrieving process as non-numerical computation. However, one points out that the retrieving process demands an eigenvalue computation to find knowledge from the retrieved information [1, 2, 3]. Thus,

one can say that the process of eigenvalue computation is not only an important process on the conventional limited scientific and technical fields, but also is a basic process in the field of general information processing. Considering computational scale and memory usage, the eigenvalue computation is always a large-scale problem, since databases were retrieved on the huge internet, and scientific and technical problems require computation of maximum performance and memory usage.

From these reasons, our attention is always focused on solving eigenvalue problems at high speed. On the other hand, today's high speed computers which can perform large-scale computations, such as super-computers, are categorized as distributed parallel computers. To solve the large-scale eigenvalue problems, therefore, we should parallelize the eigenvalue computation considering algorithm and memory usage.

Under these restrictions, the standard eigenproblem of the following must be solved:

$$Ax = \lambda x, \tag{1.1}$$

where $A \in \mathfrak{R}^{n \times n}$ is an $n \times n$ matrix, $\lambda \in \mathbf{C}$ is an eigenvalue, and $x \in \mathbf{C}^n$ is an eigenvector. Whether the matrix is a real general matrix or a real symmetric matrix, there are many ways to solve this equation on sequential computers [4, 5, 6, 7, 8].

In many cases we transform the matrix A to a matrix, which has many zero entries without changing its eigenvalues for the reason of computational complexity. Such a transformation is called *similarity transformation*. By using similarity transformation, it is possible to transform a real dense symmetric matrix to a tridiagonal matrix, and a real dense non-symmetric matrix to a Hessenberg form. With this transformation, calculation of eigenvalues and eigenvectors can be easier. To perform the similarity transformation, the Householder method is used in many cases.

If the matrix A' is a symmetric matrix, we can reduce the matrix A' to a tridiagonal matrix T by using the similarity transformation:

$$Q^T A' Q = T, \tag{1.2}$$

where the matrix Q is such that $Q^T Q = I$, e.g. $Q^T = Q^{-1}$. By using Equation (1.2), the eigenproblem (1.1) is rewritten as the following:

$$(Q T Q^T) x' = \lambda' x', \tag{1.3}$$

where $A' \in \mathfrak{R}^{n \times n}$, $\lambda' \in \mathfrak{R}$, and $x' \in \mathfrak{R}^n$. By defining a new vector y as

$$y = Q^T x', \tag{1.4}$$

we obtain the following equation:

$$Ty = \lambda'y. \quad (1.5)$$

Equation (1.5) shows that the eigenvalues between the matrices A' and T are identical, and we can obtain the eigenvectors of A' by solving Equations (1.5) and (1.4).

The computational complexity for solving the eigenproblem (1.1) is $O(n^3)$. In order to solve the problem at higher speed, a new approach is needed. Especially, many of today's supercomputers are categorized as parallel architectures. Therefore, the focus must be on the performance of parallel implementations of the approach.

1.1.2 Sequential blocking techniques and load imbalance

Blocking is one of the well-known techniques to attain high performance for sequential processing. For symmetric dense eigenvalue problem, blocking the Householder tridiagonalization, which is used in the reduction from dense matrix to tridiagonal matrix, is a widely-used techniques for RISC based processors. Following is a description of this blocking algorithm in details.

Let us consider the m times successive execution of the Householder transformation. This successive execution is described as follows:

$$H^{(k+m-1)} \dots H^{(k)} A^{(k)} H^{(k)} \dots H^{(k+m-1)} = A^{(k)} - UZ^T - XU^T, \quad (1.6)$$

where $k = (q-1)m + 1$, and $q = 1, \dots, n/m$, where m is blocking length, and n is matrix dimension. To simplify the discussion, let n be dividable by m . Calculations in each step for $j = 1, \dots, m$ are:

$$H^{(k+j-1)} \left(A_{*,k+j-1}^{(k+j-1)} \right) = (U_{*,j}, \alpha_{k+j-1}), \quad (1.7)$$

$$\begin{aligned} X_{*,j} &= \alpha_{k+j-1} A^{(k+j-1)} U_{*,j} \\ &= \alpha_{k+j-1} (A^{(k)} - U_{*,1:j} Z_{*,1:j}^T - X_{*,1:j} U_{*,1:j}^T) U_{*,j} \\ &= \alpha_{k+j-1} A^{(k)} U_{*,j} - U_{*,1:j} f - X_{*,1:j} e, \end{aligned} \quad (1.8)$$

$$\begin{aligned} y &= \alpha_{k+j-1} A^{(k+j-1)T} U_{*,j} \\ &= \alpha_{k+j-1} (A^{(k)} - U_{*,1:j} Z_{*,1:j}^T - X_{*,1:j} U_{*,1:j}^T)^T U_{*,j}, \\ &= \alpha_{k+j-1} A^{(k)T} U_{*,j} - Z_{*,1:j} e - U_{*,1:j} d, \end{aligned} \quad (1.9)$$

$$\mu_{k+j-1} = \alpha_{k+j-1} U_{*,j}^T X_{*,j}, \quad (1.10)$$

$$Z_{*,j} = y - \mu_{k+j-1}U_{*,j}, \quad (1.11)$$

where

$$d = \alpha_{k+j-1}X_{*,1:j}^T U_{*,j}, \quad (1.12)$$

$$e = \alpha_{k+j-1}U_{*,1:j}^T U_{*,j}, \quad (1.13)$$

$$f = \alpha_{k+j-1}Z_{*,1:j}^T U_{*,j}. \quad (1.14)$$

If $j = 1$, the vectors which need to continue the computation must be calculated, and these results u_k , x and y must be stored in the matrices $U_{*,1}$, $X_{*,1}$, and $Z_{*,1}$, respectively. The case of $m = 1$ is equivalent to the unblocked algorithm. Accordingly, we can regard the blocked algorithm as an extension of the unblocked algorithm.

Generally, blocked algorithms as showed in this section can attain high performance in RISC based architectures because of high data usage in caches. Figure 1.1 shows the typical performance of the above blocked algorithm, since the performance of blocked algorithm is improved according to the problem size.

Next is discussion of the parallelization of the blocked algorithm. Generally, there is a problem for the blocked algorithm when we parallelize it. The problem is load imbalance caused by data distribution in the sequential blocked algorithm. For many parallel blocked algorithms, the block-cyclic data distribution is used. Why we use the block-cyclic data distribution is because the block-cyclic data distribution with length m — it is the same length as the blocked algorithm one — makes the parallel implementation easier. The nature of this distribution is explained as: implementing each blocked process is easily done in blocked algorithms.

This implementation, however, causes a heavy load imbalance when the block length of m is too large for the matrix dimension of n . Unfortunately, such condition is often caused in massive parallel processing (MPP) or when we have to solve small problems. Figure 1.2 shows the load imbalance problem in parallel blocked algorithms.

From the discussion, blocking algorithms have a trade-off problem between computation efficiency per processor and parallelism. Therefore, designing the parallel blocked algorithm with the different factors of blocking length among blocking algorithm and data distribution is one of the reasonable approaches to solve the trade-off problem. Such approaches are proposed by T.Katagiri and Y.Kanada [9] and B.Hendrickson *et.al.* [10] in the Householder reduction for eigenproblem.

Development of parallel blocked algorithms by using the different blocking factors, therefore, not only is a key aspect to attain high performance in “large scale” computation of eigensolvers, but it is also a key technique to accomplish high parallelism in “large scale” distributed memory parallel machines.

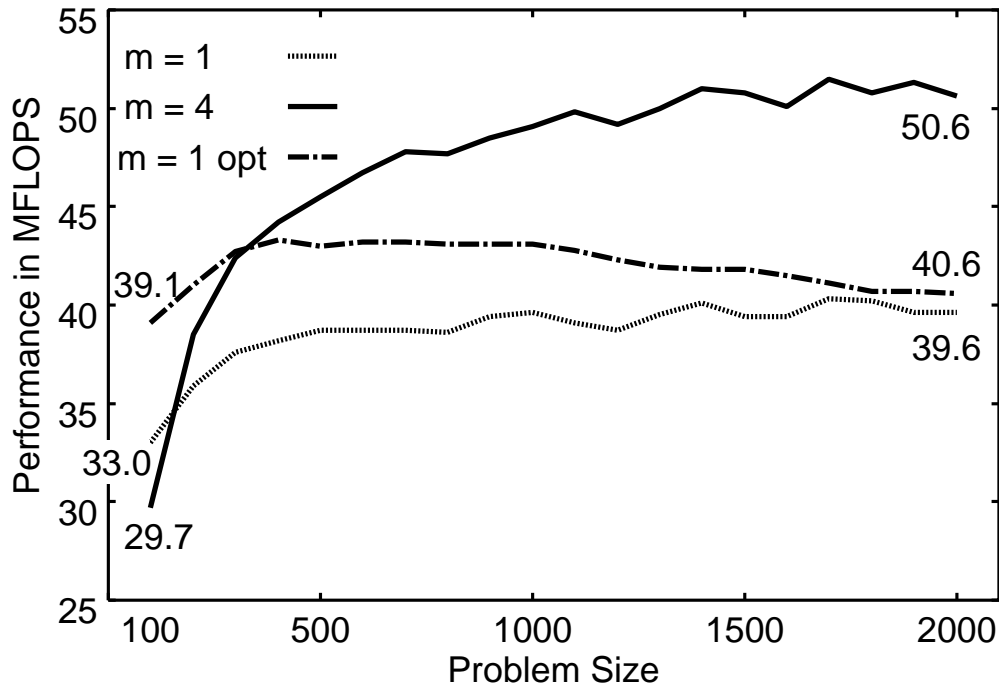


Figure 1.1: Typical performance for blocked algorithms. The measured algorithm is blocked Householder Hessenberg reduction using 4 RISC based processors on the Fujitsu AP1000+. The notation of “m=1 opt” is the performance of copy free version of the blocked algorithm. This shows high performance implementation in the blocked algorithm.

1.1.3 Using symmetry of matrices

Many symmetric eigenvalue solvers utilize the nature of symmetry in input matrix to decrease the computational complexity and computation time. For sequential processing, there is no room to discuss the use for symmetry because the access time of matrix elements can be regarded as constant time. When the algorithm is parallelized with distributed memory parallel machines, however, a problem is confronted — the access time of matrix elements among intra-processor and inter-processor is essentially different. In general, the algorithms which use the nature of symmetry cause a complicated data access pattern because they use a special data structure to compress the elements of symmetric matrix. The complicated data access pattern demands inter-processor data access because it is hard to decide the best data distribution which demands less inter-processor data access. In addition, the complicated data access sometimes prevents high performance sequential implementation because it may not be vectorized and/or it causes

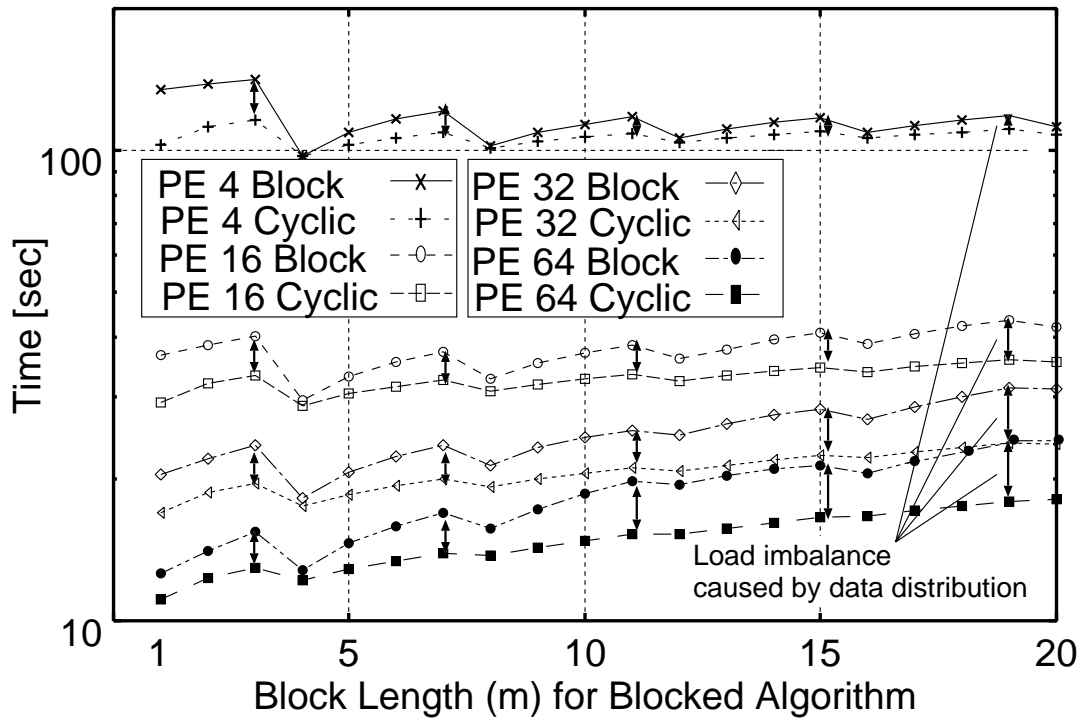


Figure 1.2: Load imbalance caused by data distribution methods. The x-axis shows the blocking length for the blocked algorithm (NOT the length of data distribution). The notation of “Block” means distribution length of m , which is the same value of the block algorithm, is specified. On the other hand, the notation of “Cyclic” is the different data distribution length to the blocked algorithm is specified. In the case of “Cyclic”, the length of data distribution is fixed as 1.

non-continuous memory access. Figure 1.3 shows this aspect in a matrix-vector products.

The algorithms which use the nature of non-symmetry, however, can not decrease the computation complexity. Therefore, there is a trade-off problem in the viewpoint of computational complexity. Although the computational complexity is two times larger than that of symmetry case, the method using non-symmetry is an effective method in MPP environments, since the communicational cost of MPP is much higher than that of small-scale parallel environments.

Using non-symmetry of matrix, therefore, is one of the choices to attain high performance in “large scale” distributed memory parallel machines.

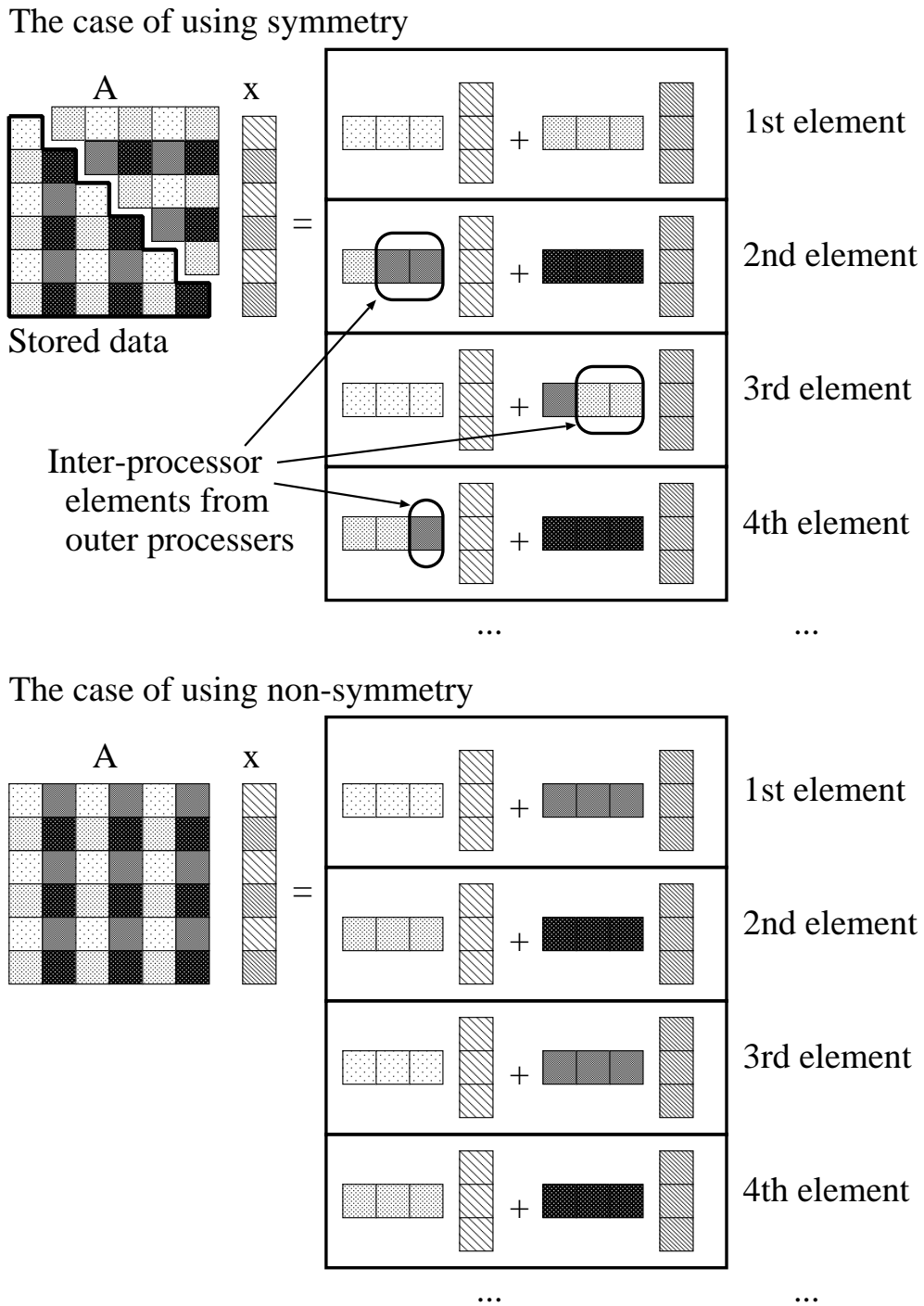


Figure 1.3: An example of using symmetry and non-symmetry of matrix in a matrix-vector products. The data distribution of the matrix A is cyclic-cyclic distribution. The data distribution of the vector x is 2-D cyclic distribution, and this distribution shows that the same elements of x are distributed to PEs which hold same row of the matrix A .

1.1.4 Reduced communication method and 2-D distribution

Using the 2-D data distribution, we can reduce communication complexity in computations. Consider the following computations:

$$y = Ax, \tag{1.15}$$

$$z^T = y^T A. \tag{1.16}$$

The computations of Formulas (1.15) and (1.16) appear in Householder reduction of eigensolvers.

By assuming the 2-D data distribution and using the nature of non-symmetry, the matrix-vector product of Formula (1.15) can be performed its computation without inter-processor communication, except for row-wise vector reduction. Next, to compute Formula (1.16), the inverse elements of the vector y is needed. To obtain the inverse elements of y , we can use the column-wise multi-casting for diagonal processors, since diagonal processors have the elements of vector y^T . As the result, the vector y^T is obtained by applying this multi-casting. Then, we can perform the computation of Formula (1.16) without inter-processor communication, except for column-wise vector reduction because of the nature of 2-D distribution.

Thus, the computations of Formulas (1.15) and (1.16) can be performed by using column-wise and row-wise communications. Figure 1.4 shows the process.

Note that the communications of Figure 1.4 show high efficiency in MPP environments, since the communication time is $O(n/\sqrt{p} \log \sqrt{p})$ in comparison with the other distribution cases (for example, 1-D cyclic distribution) of $O(n \log p)$, where p is the total number of processors. The idea of using the “reduced algorithm” for Householder tridiagonalization is proposed by H.Chang *et al.* [11], B.Hendrickson *et al.* [12], and T.Katagiri and Y.Kanada [13, 14, 15].

The reduced communication method shown in Figure 1.4 can also be applied efficiently for the case of using non-symmetry of matrices because there is no inter-processor communication in matrix-vector products under 2-D data distribution. This advantage is pointed out by T.Katagiri and Y.Kanada [16]. Note that the effect of using the reduced communication algorithm and the nature of non-symmetry is much conspicuous in MPP environments since the cost of communication is huge. The cost of communication can not be ignored in MPP environments.

The reduced communication method, therefore, is necessary to establish high performance in “large scale” distributed memory parallel machines.

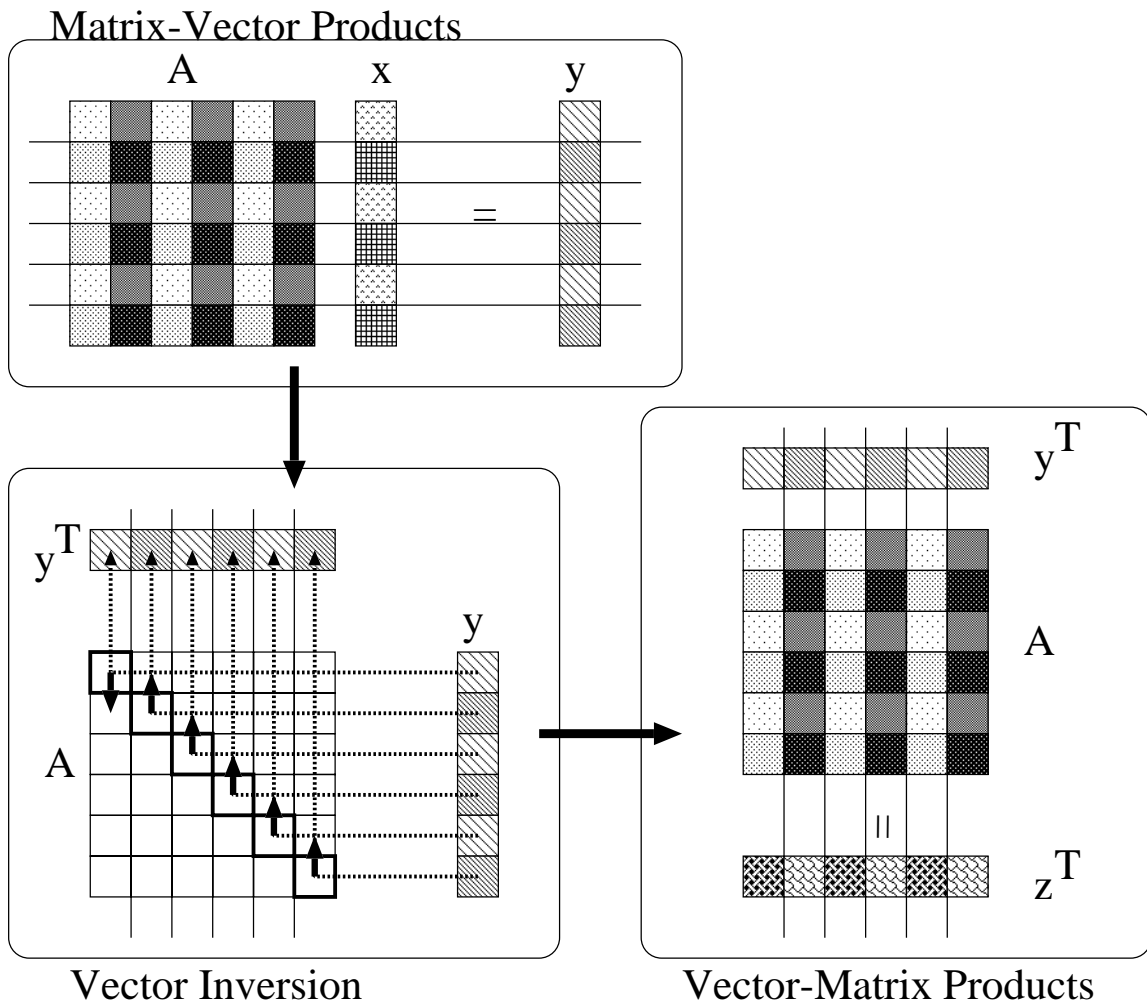


Figure 1.4: Reduced communication method in 2-D grid-wise data distribution. The initial data distribution of matrix A is cyclic-cyclic grid-wise distribution. That of vector x is 2-D cyclic distribution, which is same as the distribution shown in Figure 1.3.

1.1.5 Parallelism of orthogonalization to compute eigenvectors

Many eigensolvers require orthogonalization for eigenvectors corresponding to clustered eigenvalues to keep accuracy. Normal orthogonalization methods, for example, the Gram-Schmidt method, have poor parallelism to orthogonalize eigenvectors in the inverse iteration method. According to T.Katagiri and Y.Kanada [39], the ratio of the parallel orthogonalization time to total eigenvalue computation time was more than 90%.

Therefore, a new approach is needed to compute “large scale” clustered eigenproblem in distributed memory parallel machines.

1.1.6 Coding time to attain high performance

To develop high performance libraries, there is another problem. Modern microprocessors can achieve high performance on linear algebra kernels. To achieve the high performance, however, extensive machine-specific software optimization has to be done, and the optimization time will be huge in general. For example, we may have to write native machine languages to achieve high performance. This kind of work is time-consuming. To solve the tuning problem, we should discuss how to optimize the libraries to reduce the optimization time automatically.

We also think that the optimization feature gives us another benefit — several auto-tuning facilities are needed to archive high performance, since the conventional routines without auto-tuning specify only one proper parameter for all dimensions of matrix. We think, however, they can not attain high performance. By using several tuned parameter sets obtained with the auto-tuning facility, much higher performance is realized.

To attain high performance in “large scale” eigensolver codes, therefore, a methodology of automatically optimized libraries is needed.

1.1.7 Ill-specified parameters

In addition to some advantages of optimization feature mentioned, above optimization feature can prevent from ill-specified parameters to decrease the performance of libraries. Users who are not well familiar to the library can not specify the proper parameters for performance, but the library automatically specifies the proper parameter sets from the nature of the optimization feature. Generally, many numerical libraries open the performance parameter sets to users. We believe that this causes poor performance by setting the unsuitable parameter sets by the inexperienced users. We also believe that a feature which can specify arbitrary parameter sets should be included in each package because expert users are sometimes familiar to the optimal parameter sets.

Therefore, the automatic optimization methodology is also needed from the viewpoint of library performance.

1.2 History of parallel algorithms and libraries

1.2.1 Overview of parallel algorithms

1.2.1.1 Reduction algorithms by using the Householder method

The similarity transformation using the Householder method is called “Householder reduction.” There are many kinds of Householder reduction based algorithms to perform eigendecomposition. For its parallelization, we have to explain how to distribute the objective matrix.

In the Householder based algorithm, G.Stewart parallelized the Hessenberg reduction (NOT tridiagonal reduction) by using the column-wise block distribution [17] in 1987. According to our knowledge, he was the first persons to parallelize the Householder based algorithm. In 1980’s, many parallel machines had very low performance communication hardware. From this fact, there was a researcher who thought that parallelizing Householder reduction is difficult because it needs heavy communications [18]. Reducing communication overhead has been focused on as a major problem. For example, T.Kalamboukis [19] proposed a parallel method which can hide the broadcasting overheads for Householder reductions.

Later, the communication hardware of parallel machines was improved, and many researchers proposed several data distribution strategies. H.Chang *et al.* used the column-wise cyclic distribution [20]. H.Chang *et al.* [11] also used the cyclic-cyclic 2-D distribution in 1988. We believe that H.Chang *et al.* were the first persons who use the 2-D distribution in the Householder reduction. B.Hendrickson *et al.* were the first to point out that using the 2-D distribution can reduce the communication complexity to the order of $1/\sqrt{p}$ in the tridiagonal reduction [12]. Later, T.Katagiri and Y.Kanada extended this idea to the Hessenberg reduction [13, 14, 15].

As for blocked algorithm, J.Dongarra and R.van de Geijn [21] proposed parallel blocked Householder algorithms in 1992. They used the column-wise block-cyclic data distribution. This distribution strategy, however, has a load imbalance problem. For this, many researchers [10, 9, 22] pointed out a blocking problem between *block length for data distributions* and *block length for blocked algorithms* in the Householder reduction. We call the block length in data distribution *BDD*, and block length in blocking algorithm *BBA*. B.Hendrickson *et al.* [10] and T.Katagiri and Y.Kanada [9] suggested that the *BBA* must

be 1 in order to maintain the load balance.

On the other hand, in order to reduce the communication time in the tridiagonalization, C.Bischof *et al.* proposed a parallel tridiagonalization algorithm through two-step band reduction [23]. This algorithm used a sequential blocking technique, called WY representation [24], and the WY representation leads the tridiagonalization to high computational efficiency and low cost in communication. It requires, however, more communications than normal one to transform the reduced band matrix to tridiagonal matrix in the second step.

1.2.1.2 Jacobi method

Parallel Jacobi method, or Jacobi-like method is far more accurate than conventional methods, such as the QR methods in a certain condition. Especially, if the input matrix is a spectrally dominant matrix, the Jacobi method is more accurate than the QR method [25]. As for problem size, it is said that the Jacobi method is useful in small dimension of $n < 100\sqrt{p}$ [26], where n is the size of matrix, and p is the number of processors.

There are two variations of the sequential Jacobi algorithm, and their difference is how to annihilate the non-zero elements. The former one is called the classical Jacobi algorithm, and it annihilates the biggest elements by the Jacobi rotation. The other one is called the threshold Jacobi algorithm, and it annihilates the non-zero elements in a continuous order. As for its parallelization, the threshold Jacobi algorithm is the suitable algorithm because the annihilation of the threshold Jacobi algorithm can be re-ordered to obtain high parallelism. A parallel Jacobi method for overlapping the annihilations is proposed by M.Pourzandi and B.Tourancheau [27]. Blocking techniques for parallel Jacobi method are studied by P.Arbenz and M.Oettli [28], or D.Gimenez *et al.* [29].

1.2.1.3 QR method

The QR method (or QR algorithm) is a method that the full symmetric matrices are first reduced to a tridiagonal form by using a similarity transformation, and then the QR method rapidly reduces the off-diagonal elements until they are negligible [4].

As for its parallelization, S.Chinchalkar and T.Coleman [30] implemented a parallel QR method with Givens rotation for dense symmetric eigendecomposition in 1991. Their data distribution was the row-wise 1-D cyclic distribution. On the other hand, blocking techniques for the parallel QR method were studied by C.Bischof [31] in 1988. He used the Householder QR method by using the WY representation, and also proposed a pipelined parallel Householder QR method.

For QR algorithm for non-symmetric matrices, it was said that the QR algorithm had poor parallelism because of its usage of the one shift strategy. This problem was first pointed out by G.Henry and R.van de Geijn [32]. And they improved the parallelism by using the multi-shift strategy [32]. The QR algorithm with multi-shift strategy, however, has the problem of its convergence. Later, R.Suda *et al.* proposed a new data distribution method for the non-symmetric QR algorithms [33] in 1999. By applying their data distribution, the parallelism was improved, and we can obtain parallel efficiency without using the multi-shift strategy.

1.2.1.4 Bisection method

There are two kinds of bisection methods for eigenvalue computation. They are normal-bisection and multi-section methods. The normal-bisection method is suitable for RISC based computers, and the multi-section method is suitable for vector computers [34]. Therefore, which methods are suitable depends on computer architectures.

These methods can be easily be parallelized because they have natural parallelism in each computation of eigenvalues.

1.2.1.5 Divide and conquer method

The divide and conquer method for the symmetric tridiagonal eigenvalue problem was first developed by J.Cuppen in 1981. This method is suitable for computing all eigenvectors corresponding to clustered eigenvalues in sequential processing.

As for its parallelization, J.Dongarra *et.al.* [35] and I.Ipsen and E.Jessup [36] used straightforward data distribution strategy, which uses one-dimensional block distribution to distribute matrices. This data distribution grows in communication with the number of processes, however, making it not scalable. Later, F.Tisseur and J.Dongarra [37] used a two-dimensional block cyclic distribution of the matrices, and they obtained the result of the parallel divide and conquer method which is 8 times faster than the QR method in a certain problem on 8-nodes IBM SP2.

The divide and conquer method is basically suitable for parallel processing. It requires communication times of $O(\log p)$ and communication volume of $O(n^2 \log p)$. These communication costs indicate that overheads are heavy enough for small dimension of n and large processor number of p . We think, therefore, it is not proper approach for small dimension problems or in MPP environments. In addition, the divide and conquer method basically requires $O(n^2)$ memory spaces to compute eigenvectors. This shows that there is no scalability for memory spaces. Then, it is not appreciate method for the distributed

memory parallel machines. It is one of the best method for the shared memory parallel machines.

1.2.1.6 Inverse iteration method — It causes the orthogonalization problem.

The inverse iteration method is the power method invoking the matrix A^{-1} [4]. That is,

$$\text{Solve } Ay_k = x_{k-1} \text{ for } y_k. \quad (1.17)$$

The iteration (1.17) can easily be replaced with $A - \sigma$ instead of A . We can calculate arbitrary eigenvalue y_k corresponding to the eigenvalue σ by this replacement. From the iteration (1.17), we can find that there is high parallelizm in the inverse iteration because each computations of eigenvectors y_k do not affect other computations.

The main, and perhaps the only, weakness of inverse iteration is that the computed eigenvectors for two close eigenvalues may be acceptable (because their residual are small) and yet not be mutually orthogonal [4]. A simple answer to this problem is to perform re-orthogonalization for clustered eigenvalues. This weakness is also the weakness of parallelization. Therefore, the parallelization of re-orthogonalization process is very important.

Today, many researchers proposed parallel re-orthogonalization methods. We will explain the methods in the following sections.

1.2.1.7 Multi-color inverse iteration method

The multi-color inverse iteration method is proposed by K.Naono *et.al.* [38] in 1995. This method has the following three advantages:

- (i) reducing the computation complexities for the eigenvectors corresponding to clustered eigenvalues,
- (ii) improving parallelism for the re-orthogonalizations, and
- (iii) improving load balance for the inverse transformation.

As for (i), conventional iteration methods have been used in Paters-Wilkinson method to find clustered eigenvalues. In the Paters-Wilkinson method, the eigenvalues among the following distance *eps* are regarded as clustered eigenvalues:

$$eps = 10^{-3} \|T\|_1, \quad (1.18)$$

where T is a tridiagonal symmetric matrix, and the notation of $\|\cdot\|_1$ is defined as

$$\|T\|_1 \equiv \sum_{i=1}^n |t_{i,i}| + \sum_{i=1}^{n-1} |t_{i,i+1}|. \quad (1.19)$$

In the Paters-Wilkinson method, eigenvalue in the distance of ϵps from eigenvalue A is regarded as the clustered eigenvalue (say eigenvalue B), and eigenvalue in the distance of ϵps from the eigenvalue B is also regarded as the clustered eigenvalue (say eigenvalue C). The eigenvalues A and C have the distance of $2 \times \epsilon ps$. From this reason, this distance is long enough to keep the accuracy of eigenvectors. Therefore, the extra re-orthogonalization is performed in this method. K.Naono *et.al.* pointed out this distance problem, and modified the method to reduce extra re-orthogonalization.

Next as for (ii), K.Naono *et.al.* improved the re-orthogonalization strategy by using the coloring problem in the SOR method for the sparse linear equation iterative solvers. From this improvement, each computations of eigenvectors can start the computation in parallel when the eigenvectors need no re-orthogonalization.

Finally, as for (iii), K.Naono *et.al.* implemented their solver to distribute the number of eigenvectors per PE equally. From this implementation, their solver does not cause heavy load imbalance in the inverse transformation routine. Of course, the improvement of the re-orthogonalization process (ii) is one of the reasons that give us the good load balance in the inverse transformation routine.

The multi-color inverse iteration method, however, can not parallelize ill-conditioned eigenvalue computations, for example, most of all eigenvalues are equivalent. From the nature of this method, the computations of the ill-conditioned problems are sequentialized in the orthogonalization process.

1.2.1.8 Inverse iteration method with the hybrid Gram-Schmidt orthogonalization

The hybrid Gram-Schmidt orthogonalization method is proposed by T.Katagiri and Y.Kanada [39, 40, 41]. This method is a hybrid method of the classical and modified Gram-Schmidt methods. They used a parallelism in the classical Gram-Schmidt method, and improved the parallelism in the re-orthogonalization process. The same idea appears in [42, 43]. We think, however, T.Katagiri and Y.Kanada were the first to use the hybrid re-orthogonalization in the inverse iteration method. In addition they discussed the method how to affect data distribution for the orthogonalization process in detail.

The hybrid Gram-Schmidt orthogonalization method works well in the ill-conditioned problems. Although additional communications are not needed to keep accuracy, however, it requires communications to compute clustered eigenvalues, since the orthogonal distance for clustered eigenvalues is unreasonably large. Simple solution to this problem is to apply the approach of the multi-color inverse iteration method.

1.2.1.9 Householder inverse iteration method

The Householder inverse iteration method is developed by Y.Yamamoto *et.al.* [44]. Note that this method is different from the inverse iteration method after the Householder tridiagonalization.

To reduce communication time, the Householder orthogonalization method is used in the re-orthogonalization process. By using the orthogonalization, the kernel of re-orthogonalization can be changed from level 2 BLAS to level 3 BLAS. From this reason, much better performance is obtained than the normal re-orthogonalization with level 2 BLAS.

The method has 1.5 times computational complexity compared to the normal inverse iteration.

1.2.1.10 Dhillon's new algorithm

Finally, we have to mention the Dhillon's new algorithm. I.Dhillon *et.al.* developed a new algorithm without orthogonalization in the inverse iteration method [45, 46].

As we have already mentioned, the inverse iteration method has basically high parallelism if the eigenvalues are well-separated, since we need no orthogonalizations. I.Dhillon *et.al.* used the natural parallelism of inverse iteration method, and developed an algorithm which can make good initial eigenvectors. To make the good initial eigenvectors, they used a special factorization with the object tridiagonal matrix, named *twisted factorization*.

The Dhillon's new algorithm, however, needs extremely high accuracy for computed eigenvalues. Therefore, conventional reduction methods can not be applied because the conventional reduction methods damage the accuracy. From this reason, we conclude that the Dhillon's new algorithm has an accuracy problem, although the Dhillon's new algorithm has high parallelism.

Conclusion from the above discussion is that we still have no perfect methods in the viewpoint of parallel performance and accuracy of eigenvalues and eigenvectors. We think that if the users can accept the low accuracy, the Dhillon's new algorithm is the best choice.

1.2.2 Overview of software libraries

1.2.2.1 ScaLAPACK's PDSYEVX

ScaLAPACK's PDSYEVX routine [26] uses the reduction to tridiagonal form, the bisection method, the inverse iteration method, and the inverse transformation to perform

eigendecomposition of dense symmetric matrices.

The PDSYEVX routine was designed by using the block-cyclic 2-D distribution. The memory usage of orthogonalization process of the PDSYEVX routine is limited. This is the explanation of the orthogonalization process.

Let the number of orthogonalized vectors in PE i be nv_i . In the PDSYEVX routine, the total number of orthogonalized vectors can be calculated as

$$\sum_{C \in \{\text{clusters assigned to } nv_i\}} \cdot \sum_{i=1}^{|C|} n_{itr}(i),$$

where $n_{itr}(i)$ is the number of the inverse iterations performed for the i -th eigenvalue [26]. If the size of the largest cluster is greater than n/p , the processor which has more n/p -th eigenvectors does not orthogonalize these eigenvectors. Therefore in this case, there is no guarantee for its orthogonality.

1.2.2.2 HJS

B.Hendrickson, E.Jessup, and C.Smith wrote a symmetric eigensolver, named HJS [10] for the Intel PARAGON. The HJS only works on the Intel PARAGON, and has never been released as public domain library.

HJS requires the data distribution to be a cyclic distribution of block size 1, and that intermediate matrices to be replicated across processor columns and distributed across processor rows.

1.2.2.3 PeIGs

PeIGs [47] uses the reduction to tridiagonal form, the bisection method, the inverse iteration method, and the inverse transformation to perform the eigendecomposition of dense symmetric matrices.

The PeIGs uses a new re-orthogonalization strategy. The strategy uses a simultaneous iteration to maintain orthogonality among eigenvectors associated with clustered eigenvalues. First of all, the inverse iteration method [48] with tridiagonal matrices is done at one time. This iteration method does not perform re-orthogonalization. As the next step, if the calculated eigenvalues are clustered, the eigenvectors associated with the eigenvalues are re-orthogonalized. This allows the re-orthogonalization to be performed efficiently in parallel processing.

If the input matrix has large clusters of eigenvalues, however, the PeIGs is slow because the cost of re-orthogonalization in the PeIGs is huge.

1.2.2.4 HITACHI's MATRIX/MPP

MATRIX/MPP (Matrix calculation subprogram library / Massive Parallel Processors) [49, 50] for parallel computers are provided by the HITACHI Limited. The MATRIX/MPP is designed to provide high performance on the HITACHI MPP System, such as the HITACHI SR2201 [51], and the HITACHI SR8000 [52]. The routines can be called from the user programs are written in FORTRAN or C.

It contains subroutines for linear algebra subprograms, eigensystem analysis, Fourier transforms, and uniform random number generators.

1.2.2.5 IBM's parallel ESSL

PESSL (Parallel Engineering and Scientific Subroutine Library) [53] is a mathematical library from IBM. The PESSL is designed to provide high performance on the IBM SP Systems or on clusters of IBM RS/6000 workstations [54].

It contains subroutines for linear algebra subprograms, matrix operations, linear algebraic equations, eigensystem analysis, Fourier transforms (2-D and 3-D), and uniform random number generators.

The PESSL components are ESSL for core computational routines (BLAS [55, 56] tuned for RS6000 and a subset of LAPACK), BLACS [57, 58], a subset of level 2 and 3 PBLAS [59], and a subset of ScaLAPACK [60] (eigensystem and singular value analysis routines).

1.2.2.6 NAG's Parallel Libraries

The NAG Parallel Libraries are the parallel version of the NAG Fortran Library. There is a PVM [61] based library and an MPI [62, 63] based library.

The NAG library includes NAG Fortran libraries for core computational routines (tuned and untuned BLAS, and the suite of numerical routines including most of LAPACK [64]), BLACS, Level 2 and 3 PBLAS, a subset of ScaLAPACK, quadrature, unconstrained minimization, sparse linear algebra, and uniform random number generators.

1.2.2.7 Fujitsu-ANU Parallel Mathematical Subroutine

The Fujitsu-ANU Parallel Mathematical Subroutine Project [65] was a joint research program involving staff at the Australian National University and Fujitsu Japan. The goal of the project was to produce a library of mathematical subroutines for the vector-parallel Fujitsu VPP300 [66]. The subroutine includes the SSLII mathematical subroutine

library for Fujitsu vector and vector/parallel supercomputers, in particular, for the Fujitsu VPP300.

Its parallel and serial vector algorithms have been developed in several areas, including: random number generators, eigenvalue solvers, sparse matrix solvers (direct and iterative), FFT's, multigrid preconditioner, wavelet transforms, and least squares solutions. Code developed is incorporated into the Fujitsu SSL2 scientific subroutine libraries, SSL2VP for one processor and SSL2VPP for multiple processors.

As for its eigenvalue solvers, a new tridiagonal reduction algorithm, named one-sided Householder method [67], is adopted. The one-sided Householder method can reduce communications against normal Householder method. It needs, however, a Cholesky factorization. Therefore the one-sided Householder method needs extra calculation. On the other hand, the eigensolver can perform outer-PE re-orthogonalization. If the clusters of eigenvalues extend across two or more PEs some will drop entirely from the computation [68].

1.2.2.8 ILIB

The Intelligent Library (ILIB) Project [69] is a partial project of High Performance Tools & Software (HINTS) Project [70] at Kanada Laboratory, the University of Tokyo.

They have proposed a new concept for libraries, which is named Intelligent Library (ILIB). The ILIB is a concept of libraries which can auto-tune for performances without setting parameters by users. They especially define the ILIB concept for numerical calculation. ILIB should have the following characteristics:

- it has less parameter interface,
- it tunes calculation kernels automatically,
- it tunes communication method automatically (if we use parallel architectures), and
- it selects efficient algorithms automatically.

They call such numerical libraries as ILIB for numerical calculation. Currently the ILIB project provides linear equations solvers for an iterative methods, named ILIB_GMRES [71] and a direct method, named ILIB_LU [72], and an eigensolver for symmetric matrices, named ILIB_TriRed [73].

Conclusion from the above discussion is that we still have no perfect libraries in the viewpoints of accuracy for eigenvectors, computation efficiency, memory usages, and parallel efficiency. To solve “large scale” eigenproblem on the “large scale” distributed memory parallel machines, we have to establish good eigensolver and release it to public.

1.3 Organization of this thesis

This thesis is divided into four parts.

Chapter 2 is organized as Part I of this thesis. Part I is an introduction and describes parallel processing basics. Chapters 3 and 4 are organized as Part II. Part II gives new algorithms and implementations for parallel dense eigensolver. Chapters 5 and 6 are organized as Part III. Part III gives new applications and concepts for parallel dense eigensolvers. Finally, Chapters 7 and 8 are organized as Part IV. Part IV gives adaption of this study and conclusions.

As Part I of this thesis, Chapter 2 describes distributed memory machines and parallel algorithms basics.

As Part II of this thesis, Chapter 3 studies a new algorithm and implementation for high performance parallel Householder-Bisection algorithm, and the algorithm is a well-used algorithm to compute dense symmetric eigenvalues. Chapter 4 describes a new parallel algorithm of Gram-Schmidt orthogonalization methods and the orthogonalization methods are also well-used methods to orthogonalize the computed eigenvectors in almost all eigensolvers. Chapter 4 also contains a new data distribution method, and new classification for the parallel orthogonalization methods.

As Part III of this thesis, Chapter 5 proposes a new concept to optimize the libraries automatically. We take a parallel eigensolver as a test case to show the effectiveness of the concept. On the other hand, such optimization software has a problem of optimization time. To solve the optimization problem, Chapter 6 suggests some methods to reduce the optimization time. In addition, Chapter 6 also proposes a new category of BLAS, named *Intelligent BLAS (IBLAS)*.

As Part IV of this thesis, Chapter 7 describes adaption of the parallel algorithm shown in Chapter 3, and future works. Finally, Chapter 8 gives conclusions in this thesis.

Chapter 2

Distributed Memory Machines and Parallel Algorithms Basics

2.1 Introduction

This chapter explains the target architectures in this thesis, and some definitions to perform parallel linear algebra operation. The computer environment in numerical experiments is shown in the later part.

2.2 Target architectures

2.2.1 SIMD and MIMD

Above all the classification of parallel computers, Michael J. Flynn's (1966) is the most well-known. He classified parallel computers in four classes, e.g. SISD, MIMD, MISD, and SIMD. Today, however, there are no more parallel computers for SISD and MISD. Therefore important classifications are SIMD and MIMD. The SIMD and MIMD are explained as the following.

SIMD : Single Instruction stream, Multiple Data stream.

In SIMD computers, the control unit of the computer broadcasts an instruction to all PEs at same time. Then, the PEs perform same computation in parallel.

MIMD : Multiple Instruction stream, Single Data stream.

MIMD computers are defined as the simultaneous execution of more than one instruction on several data stream. The individual processors run under the control of their programs. This allows flexibility in the tasks that the processors are performing at any given time.

Many of today's parallel computers for general usage are classified as MIMD computers. The dedicated usage parallel computers are SIMD computers. Therefore MIMD parallel computers are on the focus.

2.2.2 Distributed memory organization

Parallel computers can be also classified into shared memory and distributed memory parallel computers. Although both types of multiprocessors are developed as of today, we focused on the distributed memory processor. The following are the reasons.

First reason is that almost all supercomputers are the distributed memory parallel computers because the shared memory parallel computers have a limitation for the number of PEs, since memory band-width has a physical limitation. Because of this limitation, the theoretical performance is also limited against the distributed memory parallel computers. Second reason is that many scientific and technical computations are operated by using supercomputers, since the computation requires enormous computation time, so the normal computers, such as personal computers, can not perform such heavy computations in practical time.

We focus on the large scale scientific computation. Our target architecture, therefore, is distributed memory parallel computers.

2.2.3 Interconnection network

There are many interconnection topologies for parallel computers [74]. The interconnection topology can be classified as either static or dynamic.

The static connection is also called direct connection, and it limits data access to the selected PEs. Well-known examples are fully connected, star, mesh, torus, and hyper cube networks [74]. The interconnection of that is statically defined. This means that to communicate with arbitrary PEs, transmission PEs are necessary.

The dynamic connection is also called indirect connection, and it has a set of switches in every transmission PE. Examples are crossbar and omega networks [74]. By controlling the switches, we can communicate with arbitrary PEs.

2.3 Parallel linear algebra algorithm basics

2.3.1 What is speed-up

The speed-up ratio S_p is defined as the following.

T_s : Time for standard execution.
 (Sequential execution time or parallel execution time using s PEs)
 T_p : Parallel execution time using p PEs.

$$S_p = T_s/T_p. \quad (2.1)$$

Then, if T_s is measured by sequential time, the efficiency E_p is defined

$$E_p = S_p/p. \quad (2.2)$$

If T_s is measured by parallel time using s PEs, the efficiency E_p is also defined

$$E_p = S_p/(p/s). \quad (2.3)$$

When the E_p is 1 or near 1, the measured parallel algorithm is called as *highly efficiency parallel algorithm*. If the values of S_p/p do not decrease when p increases, this phenomenon is called as *linear speed-up*. When E_p is over 1, this phenomenon is called as *super linear speed-up*. In many cases, the super linear speed-up attributed to the high hit ratio of caches. To develop highly efficient parallel algorithms, therefore, techniques of using caches efficiently should be used.

2.3.2 Load balancing and data distribution

One of the key points to attain high efficiency in parallel linear algebra computations is to figure out how to distribute matrices. From the parallel processing point of view, this distribution is called “load balancing.” The next step in this section is the description of the notation for the matrix data distribution.

There are many variations for one-dimensional or two-dimensional data distributions. However, we only define the following distributions, since the following distributions are well-used and well-analyzed in this thesis.

2.3.2.1 One-dimensional data distribution

Let Π be a set of row indexes of matrix A , and Γ be a set of column indexes, where $j \in \Pi, \Gamma$ is $(1 \leq i \leq n)$, and n is a matrix dimension. These sets can vary according to each data distribution. By using these sets, the column-wise data distributions ($*$, Block) and ($*$, Cyclic) are defined as follows:

$$\begin{aligned}
(*, \text{Block}) : \quad \Pi &= \{1, 2, \dots, n\}, \\
\Gamma &= \{\lceil n/p \rceil \times myid + j\}, \\
j &= 1, 2, \dots, \mathbf{last}_b(\lceil n/p \rceil \times (myid + 1), \lceil n/p \rceil). \quad (2.4)
\end{aligned}$$

$$\begin{aligned}
(*, \text{Cyclic}) : \quad \Pi &= \{1, 2, \dots, n\}, \\
\Gamma &= \{myid + 1 + (j - 1)p\}, \\
j &= 1, 2, \dots, \mathbf{last}_c(myid + 1 + p \times \lfloor n/p \rfloor, \lfloor n/p \rfloor), \quad (2.5)
\end{aligned}$$

where $myid$ is defined as a processor identification number, and it takes from 0 to p .

The function $\mathbf{last}_b(a, b)$ is

$$\mathbf{last}_b(a, b) = \begin{cases} \text{if } a \leq n \text{ then } b, \\ \text{if } a > n \text{ then } b - (a - n), \end{cases} \quad (2.6)$$

and the function $\mathbf{last}_c(a, b)$ is

$$\mathbf{last}_c(a, b) = \begin{cases} \text{if } a \leq n \text{ then } b + 1, \\ \text{if } a > n \text{ then } b. \end{cases} \quad (2.7)$$

2.3.2.2 Two-dimensional data distribution

Similarly, the row-wise data distributions (Block, *) and (Cyclic, *) are defined as:

$$\begin{aligned}
(\text{Block}, *) : \quad \Pi &= \{\lceil n/p \rceil \times myid + j\}, \\
j &= 1, 2, \dots, \mathbf{last}_b(\lceil n/p \rceil \times (myid + 1), \lceil n/p \rceil), \\
\Gamma &= \{1, 2, \dots, n\}. \quad (2.8)
\end{aligned}$$

$$\begin{aligned}
(\text{Cyclic}, *) : \quad \Pi &= \{myid + 1 + (j - 1)p\}, \\
j &= 1, 2, \dots, \mathbf{last}_c(myid + 1 + p \times \lfloor n/p \rfloor, \lfloor n/p \rfloor), \\
\Gamma &= \{1, 2, \dots, n\}. \quad (2.9)
\end{aligned}$$

In addition, 2-D distributions, the grid-wise distributions (Block, Block) and (Cyclic, Cyclic) are easily defined as:

$$\begin{aligned}
(\text{Block}, \text{Block}) : \quad \Pi &= \{\lceil n/\sqrt{p} \rceil \times myidx + i\}, \\
i &= 1, 2, \dots, \mathbf{last}_b(\lceil n/\sqrt{p} \rceil \times (myidx + 1), \lceil n/\sqrt{p} \rceil), \\
\Gamma &= \{\lceil n/\sqrt{p} \rceil \times myidy + j\}, \\
j &= 1, 2, \dots, \mathbf{last}_b(\lceil n/\sqrt{p} \rceil \times (myidy + 1), \lceil n/\sqrt{p} \rceil). \quad (2.10)
\end{aligned}$$

$$\begin{aligned}
(\text{Cyclic, Cyclic}) : \Pi &= \{myidx + 1 + (i - 1)\sqrt{p}\}, \\
& i = 1, 2, \dots, \mathbf{last}_c(myidx + 1 + \sqrt{p} \times \lfloor n/\sqrt{p} \rfloor, \lfloor n/\sqrt{p} \rfloor), \\
\Gamma &= \{myidy + 1 + (j - 1)\sqrt{p}\}, \\
& j = 1, 2, \dots, \mathbf{last}_c(myidy + 1 + \sqrt{p} \times \lfloor n/\sqrt{p} \rfloor, \lfloor n/\sqrt{p} \rfloor)
\end{aligned}
\tag{2.11}$$

2.3.3 Data dependence

2.3.3.1 Flow, anti, and output dependencies

For a given program, we have to analyze its data dependences to parallelize the program or to attain high performance [75, 76]. Lets consider the following FORTAN code.

```

xx1 = x(1)
xx2 = x(2)
do i=1, n, 2
  do j=1, n, 2
    b(j ) = A(j ,i) * xx1 + A(j ,i+1) * xx2
    b(j+1) = A(j+1,i) * xx1 + A(j+1,i+1) * xx2
  enddo
  xx1 = x(i+2)
  xx2 = x(i+3)
enddo

```

For the loop of $i=1$ and $j=1$, the variables $xx1$ and $xx2$ are needed. However, the $xx1$ and $xx2$ can not be referred while the definitions $xx1=x(1)$ and $xx2=x(2)$ are performed to match sequential results. This dependence is called *flow dependence*. Similarly, the variables $xx1$ and $xx2$ can not be referred while the definitions $xx1=x(i+2)$ and $xx2=x(i+3)$ are done. This dependence is called *loop carried dependence*. The loop of $i=1$ and $j=n-1$, the variables of $xx1$ and $xx2$ are re-written after referring. This dependence is called *anti dependence*. The loop of $i=n-1$ and $j=n-1$, the values of the variables $xx1$ and $xx2$ (say $xx1old$ and $xx2old$) should be written as $xx1=x(n+1)$ and $xx2=x(n+2)$ (say $xx1new$ and $xx2new$). If $xx1old$ and $xx1new$ are allocated to same register, this causes a dependence. This dependence is called *output dependence*. (The case of $xx2old$ and $xx2new$ is also called output dependence.)

2.3.4 Typical communication time

2.3.4.1 Models for performance estimation

First of all, the parameters required for evaluation of performances on distributed memory machines in our model are presented here. Table 2.1 shows the list of parameters.

Table 2.1: The parameters of parallel computers for our performance estimation model.

Notation	Explanation
ϕ	CPU time per one double precision operation.
$\delta_c(p)$	Message setup time. (one-to-one communication)
$\tau_c(p)$	Transfer time per message. (one-to-one communication per byte)
$\delta_b(p)$	Message setup time. (broadcast)
$\tau_b(p)$	Transfer time per message. (broadcast per byte)
$\gamma(p)$	Scalar global summation time (double precision)

We assume that one-to-one communication time does not depend on the distance, e.g., the time is approximated by an linear expression in $\tau_c(p)$ for messages of length N . With this condition, the following formula is obtained:

$$\text{Communication time} = \delta_c(p) + \tau_c(p)N. \quad (2.12)$$

We assume that broadcasting time is given in a way as in the Expression (2.12). Below, n is the problem size, p is the number of PEs we use. We denote the total calculation time by $T_{calc}(n, p)$, the total communication time by $T_{comm}(n, p)$, and the total computational complexity by $Calc(n, p)$. The calculation time $T_{calc}(n, p)$ is expressed as:

$$T_{calc}(n, p) = Calc(n, p) \times \phi. \quad (2.13)$$

Let $CommT(n, p)$ be the number of one-to-one communications and $CommS(n, p)$ be the total message complexity with respect to one-to-one communication. Similarly, let $BroadT(n, p)$ be the number of broadcast operations and $BroadS(n, p)$ be the total message complexity for broadcasting. In addition, let $ReductS(n)$ be the number of scalar global summations. Using these expressions, the total communication time $T_{comm}(n, p)$ is described as the following:

The one-to-one communication time,

$$T_{comm}(n, p) = CommT(n, p)\delta_c(p) + CommS(n, p)\tau_c(p). \quad (2.14)$$

The broadcasting time,

$$T_{comm}(n, p) = BroadT(n, p)\delta_b(p) + BroadS(n, p)\tau_b(p). \quad (2.15)$$

The reduction operation time,

$$T_{comm}(n, p) = ReductS(n)\gamma(p). \quad (2.16)$$

Furthermore, $T_{theory}(n, p)$ can be described as:

$$T_{theory}(n, p) = T_{calc}(n, p) + T_{comm}(n, p). \quad (2.17)$$

The variable ϕ in Expression (2.13) indicates that an operation time for double precision calculation, and that variable ϕ has several values according to the conditions, such as problem size, number of PEs. Hence, estimating the value of ϕ is very hard. However, we must somehow determine the value of ϕ to estimate the execution time. A possible approach to estimate the value of ϕ is to measure the execution time of a communication-free program.

2.3.4.2 Broadcast

For broadcast operation, let a binary tree-structured communication be available in an inter connection network. Then, the broadcast time can be predicted as:

$$\begin{aligned} BroadT(n, p) &= \lceil \log p \rceil, \\ BroadS(n, p) &= \lceil \log p \rceil \times n, \\ Calc(n, p) &= 0. \end{aligned} \quad (2.18)$$

2.3.4.3 Reduction operations

There are many operations for reduction. For example, global summation, global subtraction, and finding max/min values are reduction operations. In every operations, a binary tree-structured communication can be used in an inter connection network. By using the communication, the reduction operation time is predicted as:

$$\begin{aligned} ReductS(n, p) &= \lceil \log p \rceil \times n, \\ Calc(n, p) &= \lceil \log p \rceil \times n. \end{aligned} \quad (2.19)$$

2.3.4.4 All-to-all communication

When the partial elements of a vector in each PE are gathered, the all-to-all communication is used. For the all-to-all communication, let a synchronized one-to-one communication be available in an inter connection network. Then, the all-to-all communication time can be predicted as:

$$\begin{aligned}CommT(n,p) &= p^2, \\CommS(n,p) &= np^2, \\Calc(n,p) &= 0.\end{aligned}\tag{2.20}$$

2.4 Experiment environments

2.4.1 The HITACHI SR2201

2.4.1.1 Organization

The HITACHI SR2201 system used in this thesis is a distributed memory, message-passing parallel machine of the MIMD class. It is composed of 1024 PEs, each with 256 Megabytes of main memory, interconnected via a communication network with the topology of a three-dimensional hyper-crossbar. The peak interprocessor communications bandwidth is 300 Mbytes/s in each direction. The HITACHI Optimized Fortran90 V02-06-/D compiler is used, and *-rdma -W0, 'OPT(O(SS))'* is used as a compile option.

2.4.1.2 Pseudo vectorization

The SR2201 system has a vector processor feature in a certain sense, i.e. Pseudo Vector Processor based on Slide-Windowed Registers (PVP-SW) [77]. For a linear algebra calculations, the PVP-SW gives us high performance even if we use RISC processors because the PVP-SW works efficiently for contiguous data access.

2.4.2 The HITACHI SR8000

2.4.2.1 Organization

The HITACHI SR8000 system used in this thesis is a distributed memory, message-passing parallel machine of the MIMD class, same as the HITACHI SR2201. It is composed of 128 nodes, each having 8 Instruction Processors (IPs), 8 Gigabytes of main memory, interconnected via a communication network with the topology of a three-dimensional

hyper-crossbar. The peak interprocessor communications bandwidth is 1 Gbytes/s in each direction.

2.4.2.2 Intra-node and Inter-node parallel processing

The SR8000 system has two types of parallel environments, one is intra-node parallel processing and the other, inter-node parallel processing. The intra-node parallel processing is so-called parallel processing as a sheard memory parallel machine, and it never performs interprocessor communications. On the other hand, the inter-node parallel processing is similar to parallel processing as a distributed memory parallel machine, and it must perform interprocessor communications. The HITACHI Optimized Fortran90 V01-00 compiler is used, and `-W0,'OPT(O(SS)),mp(p(0))'` is used as a compile option in the intra-node parallel processing, and `-W0,'OPT(O(SS)),mp(p(4))'` in the inter-node parallel processing.

2.4.2.3 Pseudo vectorization

The SR8000 system also has a vector processor feature in a certain sence, like the SR2201. For a linear algebra calculations, the pseudo vectorization gives us high performance even if we use RISC processors.

Part II

Algorithm of Parallel Dense Eigensolvers

Part

Chapter 3

High Performance Parallel Householder-Bisection Algorithm

3.1 Introduction

Many researchers have worked on parallelizing eigensolvers for symmetric dense matrices [78, 30, 79, 60, 38, 10, 80, 65, 81]. However, parallelization for massive parallel processing (MPP) has received little attention because (1) there were only few real MPP machines; (2) efficient MPP implementations are difficult to obtain. It is especially difficult to attain high performance when matrix size is small on a large number of processor elements (PEs). Such situation easily happens on MPP system.

Recently, various kinds of MPP architectures are becoming available, which include hundreds or more PEs. On these MPP systems many classical algorithms will work poorly since they cannot be readily adapted to a parallel environment.

It is widely known that the most efficient algorithm for computing all eigenvalues of dense symmetric matrices is the Householder algorithm. However, conventional parallel Householder algorithm based on a sequential Householder algorithm [60, 38, 10] has a problem of increasing communication complexity. Conventional algorithms based on the symmetry of the matrix increase communication complexity as compared to algorithms that assume asymmetry, even though the conventional algorithms have half of the computational complexity of non-symmetric ones.

In MPP environments, however, the increase of communication complexity must be considered. For instance, a typical conventional Householder tridiagonalization using symmetry (called HTS hereafter) have computational complexity of $4/3 \cdot n^3/p \cdot \delta_1$, and communication complexity of $\gamma_1 \cdot n^2 \log_2 p$, where n is the problem size, p is the number of PEs, δ_1 is an execution time per floating-point computation, and γ_1 is the time for a communications in the HTS. In the same way as in the HTS algorithms, we can

estimate execution time for Householder tridiagonalizations based on the non-symmetry (HTN hereafter). These algorithms have computational complexity of $8/3 \cdot n^3/p \cdot \delta_2$, and communication complexity of $\gamma_2 \cdot n^2 \log_2 p$, where δ_2 is an execution time per floating-point computation, and γ_2 is the time for a communications in the HTN. From these relations, a threshold size n_{thd} of the problem where the HTN is faster than the HTS:

$$n_{thd} < \frac{3}{4} C_{\gamma/\delta} \cdot p \log_2 p, \quad (3.1)$$

where $C_{\gamma/\delta} \equiv (\gamma_1 - \gamma_2)/(2\delta_2 - \delta_1)$. The n_{thd} depends on the number of PEs and the factor $C_{\gamma/\delta}$ for communication. For example, if $p = 4$, then $n_{thd} < 6 \cdot C_{\gamma/\delta}$, and if $p = 1024$, then $n_{thd} < 7680 \cdot C_{\gamma/\delta}$. This relation shows that (i) n_{thd} grows with increasing p ; (ii) a lower γ_2 gives us large values for n_{thd} . From this example, the conclusion is, decreased communication time is important even if the HTN algorithms have twice the computational complexity in MPP environments. As a consequence of (i) and (ii), a reduced communication algorithm assuming non-symmetry is proposed. In addition, high efficiency could be attained even if matrix size is small with the proposed algorithm.

This chapter is organized as the following. The description of parallel processing environment and the mathematical notations used throughout this paper in section 3.2. Section 3.3 is a description of an efficient parallel algorithm for computing all eigenvalues on an MPP system. In section 3.4 shows the execution time for our routine on the HITACHI SR2201 along with comparisons to the ScaLAPACK routine on the same machine. Finally, section 3.5 gives conclusions regarding this work.

3.2 Parallel processing environment and several notations

Let us assume that our target parallel computers are constructed with homogeneous PEs in processing speed, memory size and communication speed, and their PEs are labeled in a two-dimensional mesh of size $q \times r = p$, where p is the number of PEs. Let $P_{myidx, myidy}$, ($myidx = 0, 1, \dots, q-1$, $myidy = 0, 1, \dots, r-1$) be a two-dimensional index for the PEs. In addition, it is assumed that all PEs are connected with a network in order to broadcast messages and perform reduction operations, such as global summation for local data.

In our implementation, the Householder transformation is used for the similarity transformation. That is,

[Theorem](Householder transformation) Given a vector $z \in \Re^n$, the following vector $u \in \Re^n$ and scalar $\alpha \in \Re$ exist:

$$(I - \alpha uu^T)z = (\xi_1, \dots, \xi_k, \pm\sigma, 0 \dots, 0)^T, \text{ where } \sigma = \|z_{k+1:n}\|_2. \quad (3.2)$$

The vector $u \equiv (0, \dots, 0, \xi_{k+1} \pm \sigma, \xi_{k+2}, \dots, \xi_n)^T$ and the scalar $\alpha \equiv 1/(\sigma^2 + |\xi_{k+1}\sigma|)$ are a pair of quantities which satisfy the above theorem, and $\alpha u^T u = 2$ since $\|u\|_2^2 = (\xi_{k+1} \pm \sigma)^2 + \xi_{k+2}^2 + \dots + \xi_n^2 = \sigma^2 + \sigma^2 + 2|\xi_{k+1}\sigma| = 2(\sigma^2 + |\xi_{k+1}\sigma|) = 2/\alpha$. The sign of the scalar σ is same as that of ξ_{k+1} to minimize catastrophic cancellation of significant digits when calculating the elements of the vector u . The reflection $(I - \alpha uu^T)z$ in the theorem of Householder transformation is called *Householder transformation*. We represent the reflection $(I - \alpha uu^T)z$ by $H^{(k)}(z)$. This reflection does not affect the elements ξ_1, \dots, ξ_k . In addition, (u, α) is the pair to be required to perform the above reflection $H^{(k)}(z)$ in the formula (3.2).

Finally, the following is the table of the notation used in this paper in Table 3.1 to define algorithms.

Table 3.1: Mathematical notation and its explanation.

Notation	Explanation
α, μ, σ	scalars $\in \mathfrak{R}$
x, y, u	vectors $\in \mathfrak{R}^n$
χ_i, η_i, ν_i	i -th elements in the above vectors x, y, u
x_Π	a partial vector constructed from the arguments which are indexed by a set Π on the above vector x
A	a matrix $\in \mathfrak{R}^{n \times n}$
$A_{i:j,k}$	a partial vector constructed from the i, \dots, j -th rows and the k -th column of the above matrix A
$A_{\Pi,j}$	a partial matrix constructed by rows indexed by a set Π and the j -th column of the above matrix A
$A^{(k)}$	k -th iteration of matrix A

3.3 Outline of our parallel eigenvalue computation process

3.3.1 Outline of the entire process

Here, we assume that the data of our matrix is already distributed over the PEs. Under this condition, our parallel eigensolver calculates eigenvalues using the following well-known three steps:

- (1) Transforming a dense symmetric matrix to a tridiagonal matrix in parallel (*The tridiagonalization routine*).
- (2) Re-distributing the non-zero elements of the tridiagonal matrix over all PEs (*The re-distribution routine*).
- (3) Computing all eigenvalues for the gathered tridiagonal matrix in step (2) by the bisection method in parallel (*The eigenvalue computation routine*).

The Householder transformation is used in the process (1), which is called Householder-bisection method.

3.3.2 The tridiagonalization process

We consider the following transformation: $A^{(1)} \equiv A$ to tridiagonal form $A^{(n-2)}$, where $A^{(k)}$ is defined as in Table 3.1. This transformation is denoted by $H^{(k)}(z) = H^{(k)}(A_{k:n,k}^{(k)})$. By substituting $H^{(k)} = I - \alpha uu^T$ for $H^{(k)}(z)$ in $(k+1)$ -th iteration, the following equations are obtained:

$$\begin{aligned}
A^{(k+1)} &= H^{(k)} A^{(k)} H^{(k)} \\
&= A^{(k)} - \alpha A^{(k)} uu^T - \alpha uu^T A^{(k)} - \alpha^2 uu^T A^{(k)} uu^T \\
&= A^{(k)} - xu^T - uy^T + \alpha uu^T xu^T \\
&= A^{(k)} - uy^T + u\mu u^T - xu^T \\
&= A^{(k)} - u(y^T - \mu u^T) - xu^T,
\end{aligned} \tag{3.3}$$

where

$$x = \alpha A^{(k)} u, \quad y^T = \alpha u^T A^{(k)}, \quad \mu = \alpha u^T x. \tag{3.4}$$

For tridiagonalization process, the matrix A is symmetric. Therefore, $x = y$, and the following formula is obtained:

$$A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T. \tag{3.5}$$

Note that to execute k -th iteration, the column vector $A_{k:n,k}$, which is obtained from the partial matrix $A_{k:n,k:n}$, is needed.

The tridiagonalization and Hessenberg reduction routines [13] have already been developed by the Householder transformation. Figure 3.1 shows the parallel tridiagonalization algorithm.

<pre> c $P_{myidx,myidy}$ owns row set Π and c column set Γ of $n \times n$ matrix A. <1> do $k=1, n-2$ <2> if ($k \in \Gamma$) then <3> broadcast($A_{\Pi,k}^{(k)}$) to PEs sharing rows Π <4> else <5> receive($A_{\Pi,k}^{(k)}$) <6> endif <7> computation of (u_{Π}, α) <8> if (I have diagonal elements of A) then <9> broadcast(u_{Π}) to PEs sharing columns Γ <10> else <11> receive(u_{Γ}) <12> endif <13> do $j=k, n$ <14> if ($j \in \Gamma$) $x_{\Pi} = x_{\Pi} + \alpha A_{\Pi,j}^{(k)} v_j$ endif <15> enddo <16> global summation of x_{Π} to PEs sharing rows Π </pre>	<pre> <17> if (I have diagonal elements of A) then <18> broadcast(x_{Π}) to PEs sharing columns Γ <19> else <20> receive(x_{Γ}) <21> endif <22> do $j=k, n$ <23> $\mu = \alpha u_{\Pi}^T x_{\Pi}$ enddo <24> global summation of μ to PEs sharing rows Π <25> do $j=k, n$ <26> do $i=k, n$ <27> if ($i \in \Pi$ and. $j \in \Gamma$) then <28> update $A_{i,j}^{(k+1)} =$ $A_{i,j}^{(k)} - v_i (\chi_j^T - \mu v_j^T) - \chi_i v_j^T$ <29> endif enddo enddo c remove k from active columns and rows <30> if ($k \in \Gamma$) $\Gamma = \Gamma - \{k\}$ endif <31> if ($k \in \Pi$) $\Pi = \Pi - \{k\}$ endif <32> enddo </pre>
---	--

Figure 3.1: Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution).

The routine of Figure 3.1 reduces communication and broadcast times for vector reduction to a ratio of $1/\sqrt{p}$. The same idea appears in [11, 12, 10]. Table 3.2 summarizes the communication complexity for the algorithm in Figure 3.1.

Table 3.2 shows that this algorithm needs at most 5 times reduction operations per iterations for the tridiagonalization. This means that parallelizing the tridiagonalization is more difficult than parallelizing other numerical decompositions, such as LU decomposition. In addition, we see that the communication complexity of our algorithm depend on the topology of PE grids.

This process does not depend on the symmetry of the matrix, so that our routine has twice computational complexity ($8/3n^3$) of that of the standard process ($4/3n^3$) [12,

Table 3.2: Communication complexities of reduction operation for the tridiagonalization of Figure 3.1 in k -th iteration.

Line No.	Total communication setup times	Message length per one communication	Comments
$\langle 7 \rangle$	$\lceil \log_2(r) \rceil$	1	
$\langle 8 \rangle$ – $\langle 12 \rangle$	$\lceil \log_2(r) \rceil$	$\lceil (n - k + 1)/q \rceil$	if $r = q$, broadcast
$\langle 16 \rangle$	$\lceil \log_2(q) \rceil$	$\lceil (n - k + 1)/r \rceil$	
$\langle 17 \rangle$ – $\langle 21 \rangle$	$\lceil \log_2(r) \rceil$	$\lceil (n - k + 1)/q \rceil$	if $r = q$, broadcast
$\langle 24 \rangle$	$\lceil \log_2(r) \rceil$	1	

60, 10]. However, this process has a lower communication complexity than the routine in [12, 60, 10], since data structures and data access patterns are simple. This is explained by the following reasons. To decrease computational complexity, the processes of $\langle 25 \rangle$ – $\langle 29 \rangle$ in Figure 3.1 are improved to update upper tridiagonal part of A . After the improvement, either of the following two methods must be implemented, since the lower tridiagonal part of A is not calculated.

- (1) *Method for compressed data form:* Additional communications and re-structuring the calculation processes of $\langle 13 \rangle$ – $\langle 15 \rangle$ are needed.
- (2) *Method for non-compressed data form:* After the modified processes of $\langle 25 \rangle$ – $\langle 29 \rangle$ in Figure 3.1, data re-distribution for lower tridiagonal part of A is needed.

For instance, the routine in [26] (the ScaLAPACK’s tridiagonalization routine [60]) is implemented according to the above method (1). By the method (1), the routine in [26] needs 4 reduction operations to execute the processes $\langle 13 \rangle$ – $\langle 15 \rangle$, while our routine needs only 1 reduction operation. In addition, note that blocked algorithm [26] needs more additional communications to perform block update. For instance again, the routine in [26] needs 4 spread communications and 1 reduction operation, while our routine does not need any communication.

Table 3.3 shows the communication complexities of each implementation method. In general, the communication complexities depend on the implementation of communication methods. For the communication complexities of Table 3.3, the complexities are calculated by using the implementation of ScaLAPACK[26] for the method (1), and all-to-all communication for the method (2).

Table 3.3: Communication complexities of matrix-vector product for the tridiagonalization.

Implementation	Communication times	Total communication volume
Method (1)	$4 n \lceil \log_2(r) \rceil$	$2 n^2 \lceil \log_2(r)/q \rceil$
Method (2)	$n \lceil \log_2(r) \rceil + n(p-1)$	$n^2/2 \lceil \log_2(r)/q \rceil$ $+ (p-1)(n^3/(3p) + n^2/(2p))$
Figure 3.1	$n \lceil \log_2(r) \rceil$	$n^2/2 \lceil \log_2(r)/q \rceil$

The communication complexities of Table 3.3 indicate that the method (2) has $O(n^3)$ for total communication amount, and this is much higher complexity in comparison with the other implementations. Therefore, the implementation of the method (2) is not addressed.

Let ϕ_1 be a time per floating-point operation in the method of Figure 3.1, and ϕ_2 be a time per floating-point operation in the implementation method (1). Let α be a message setup time, and β be a communication time per floating-point data. By using these variables, we can estimate a condition that the execution time for the method of Figure 3.1 is faster than that of the method (1):

$$4n^2/3 \cdot (2\phi_1 - \phi_2) < \log_2(r) \cdot (3\alpha + n\beta/q), \quad (3.6)$$

where $n, r, q, \alpha,$ and $\beta > 0$. The condition of (3.6) will be achieved when $2\phi_1 - \phi_2 < 0$. This indicates that the floating-point operation factor of ϕ_1 for the method (1) can greatly affect the condition. Since if the floating-point operation of Figure 3.1 is 2 times faster than that of the method (1), the method of Figure 3.1 is always faster than that of the method (1). Note that we can implement the method of Figure 3.1 at high performance with comparison to the method (1), since the kernel of the method of Figure 3.1 does not use the special data structure for symmetric matrices.

For these reasons, we expect that our routine is faster than the conventional routines when the number of PEs increases, and adaptability of the real problems is high. In addition, our routine does not support block-cyclic distribution. The block-cyclic distribution causes a poor load balance when n/p is small. This condition easily happens on MPP, hence, such distribution is not suitable for parallel routines for MPP.

3.3.3 The re-distribution process

In order to have the entire tridiagonal matrix on each PE, the grid-wise (Cyclic, Cyclic) distributed elements are re-distributed in this process.

3.3.4 The eigenvalue computation process

The bisection method is implemented to compute the eigenvalues. Implementation of the bisection method is the same as the routine BISECT in reference [82].

In our implementation, *nbi* which is the number of the iterations for improving the accuracy when an eigenvalue is isolated is set to 200 to calculate all eigenvalues. Note that the routine does not iterate 200 times, since the routine is finished when narrowing section is small enough (such as machine epsilon). Parallelizing the routine is easy because we can parallelize the routine with different initial values.

3.4 Performance evaluation

Our parallel eigensolver was implemented on the HITACHI SR2201, and performance was evaluated.

3.4.1 Performances in the Frank matrix

To evaluate performances and to check the program, eigenvalues and eigenvectors for the following matrix were calculated:

$$A_n = (a_{ij}), \quad a_{ij} = n - \mathbf{max}(i, j) + 1. \quad (3.7)$$

Its eigenvalues are known to be:

$$\lambda_k = \frac{1}{2 \left(1 - \cos \frac{(2k-1)\pi}{(2n+1)} \right)}, \quad (k = 1, 2, \dots, n). \quad (3.8)$$

3.4.1.1 Performance for calculating all eigenvalues

We measured the execution times of an order 8000 all eigenvalue problem with the SR2201 on 4–1024 PEs. Table 3.4 shows the execution times.

From Table 3.4, the tridiagonalization process occupies the largest part (more than 90%) of the total parallel execution time, while the time for re-distribution took 0.2% at the most. Therefore, improvement of the re-distribution routine is not an issue. Notwithstanding its scalar implementation, the time for calculation of the eigenvalues took only

Table 3.4: Calculation times of 8000 eigenvalues in seconds. ($nbi = 200$, the maximal relative error with respect to the analytical values is 0.2493×10^{-7})

PEs	4	8	16	32
(Grid)	(2×2)	(2×4)	(4×4)	(4×8)
Tridiagonalizaiton	1962	989.5	490.3	254.9
(Ratio %)	(95.1%)	(94.6%)	(94.0%)	(93.8%)
Re-distribution	0.002	0.004	0.005	0.006
Bisection	98.57	55.61	30.86	16.79
(Ratio %)	(4.7%)	(5.3%)	(5.9%)	(6.1%)
Total time	2061	1045	521.2	271.7
Speed-Up	1.00	1.97	3.95	7.58
PEs	64	128	256	1024
(Grid)	(8×8)	(8×16)	(16×16)	(32×32)
Tridiagonalization	119.0	70.42	47.90	63.16
(Ratio %)	(92.1%)	(92.7%)	(93.9%)	(98.6%)
Re-distribution	0.011	0.013	0.025	0.082
Bisection	10.15	5.469	3.060	0.783
(Ratio %)	(7.8%)	(7.2%)	(6.0%)	(1.2%)
Total time	129.2	75.90	50.99	64.02
Speed-Up	15.9	27.1	40.4	32.1

about 8% of the total time. Therefore, no improvement of the routine for the calculation of the eigenvalue routine is necessary on this machine. Finally, conclusion is that fast parallel tridiagonalization is the crucial part and the efficiency of tridiagonalization will decide the total performance for computing all eigenvalues.

Next shows the performances in FLOPS by the tridiagonalization routine for 4–1024 PEs in Figure 3.2. From Figure 3.2 it is clear that the performance for 4 PEs saturates at

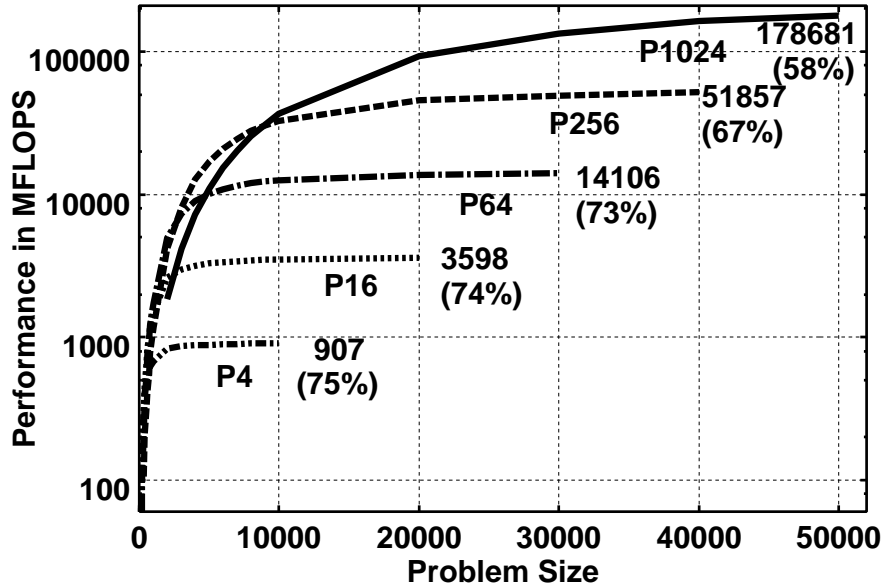


Figure 3.2: Performances in parallel tridiagonalization (MFLOPS, the percentages in parentheses are percentages of the theoretical peak performance). Total calculation amount of $8/3n^3$ is used for computing the MFLOPS values.

75% of the theoretical peak performance of 300 MFLOPS. The saturation performance for 1024 PEs was 178 GFLOPS with our algorithm. Figure 3.2 also shows that the efficiency decreases as the number of PEs increases. One of the reasons is that when the number of PEs increases, the ratio of calculation time to the total execution time decreases.

3.4.2 Comparison to the ScaLAPACK

3.4.2.1 Experimental results

Table 3.6 and Table 3.7 show execution times of ScaLAPACK tridiagonalization (hereafter SLP TRD) routine and ours (hereafter our TRD) respectively. HITACHI optimized ScaLAPACK version 1.2 [83] was used. Its communication library used is PVM, and the PBLAS which is computational kernel for ScaLAPACK is optimized by HITACHI limited.

The SLP TRD is implemented by using block-cyclic distribution, blocked algorithm, and symmetry of the matrix [26]. Since blocked algorithm was used, the size of blocking (BL) can greatly affect performance of the ScaLAPACK. Table 3.5 shows this fact.

Table 3.5: Execution time of the ScaLAPACK for varying BL in seconds. (SR2201, $n = 8000$, 256 PEs (PE grid:16×16))

BL	1	2	5	10	15	20	25	30
SLP TRD	517.37	296.53	201.97	174.02	171.50	156.86	157.28	155.70
Speed-Up	1.00	1.74	2.56	2.97	3.01	3.29	3.28	3.32

From the result of Table 3.5, we found that varying BL gives us about 3.2 times speed-up with respect to the execution time in $BL = 1$.

According to [83], if the problem size n is less than 4000, BL should be 60, and if n is over 4000, desirable $BL = 100$ is a good choice on the SR2201. Considering these values, we evaluated the performance of SLP TRD routines under $BL = \{40, 60, 80, 100, 120\}$ to find which BL gives the best performance. For PE grid, squared grid of $\sqrt{p} \times \sqrt{p}$ is best according to [83]. We tried to measure execution times in the PE grid when number of PEs is large. When the number of PE is small, such as 4, 32, and 64, we measured times in all combinations for the PE grid to find which PE grid gives the best performance. The BL size and the execution time are included in Table 3.6 and 3.7

Figure 3.3 shows execution times for varying number of PEs. From Figure 3.3(a), our TRD was about 2–6 times as fast as SLP TRD for a 2000×2000 matrix. On the other hand, from Figure 3.3(b), when matrix dimension was 8000, SLP TRD was faster than our TRD on 4–16 PEs. However, when number of PEs increased, our TRD was faster than SLP TRD. The threshold number of PEs was about 16.

Figure 3.4 shows speed-up ratio for a 8000×8000 matrix. From Figure 3.4, we could find the fact that our TRD speed-up 40 times, while SLP TRD only speed-up 10 times.

3.4.2.2 Discussion

From the results in Table 3.6 and Table 3.7, the conclusion is that if local matrix sizes were small enough, execution times of our TRD were 2–5 times as fast as the SLP TRD. This is caused by the following two reasons:

- (I) Our TRD has better load balance than the SLP TRD, since our TRD is permanently set to $BL = 1$ while SLP TRD allows to the arbitrary values;

Table 3.6: Performances for tridiagonalization I (SR2201). Unit is second.

(a) Case of $PE = 4$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP / Ours
100	0.02 (1×4, 100)	0.056 (2×2)	0.35
200	0.48 (1×4, 100)	0.133 (2×2)	3.6
400	1.73 (1×4, 40)	0.475 (2×2)	3.6
800	6.01 (1×4, 40)	2.454 (2×2)	2.4
1000	9.32 (2×2, 40)	3.785 (2×2)	2.4
2000	41.90 (2×2, 40)	26.937 (2×2)	1.5
4000	231.10 (2×2, 40)	242.010 (2×2)	0.95
8000	1422.69 (2×2, 100)	1962.512 (2×2)	0.72

(b) Case of $PE = 16$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP /Ours
100	0.03 (1×16, 100)	0.082 (4×4)	0.36
200	0.82 (8×2, 100)	0.195 (4×4)	4.2
400	1.92 (1×16, 40)	0.419 (4×4)	4.5
800	5.48 (4×4, 40)	1.733 (4×4)	3.1
1000	7.53 (2×8, 40)	1.824 (4×4)	4.1
2000	23.00 (4×4, 40)	8.649 (4×4)	2.6
4000	92.21 (4×4, 60)	56.239 (4×4)	1.6
8000	474.49 (4×4, 60)	490.346 (4×4)	0.96

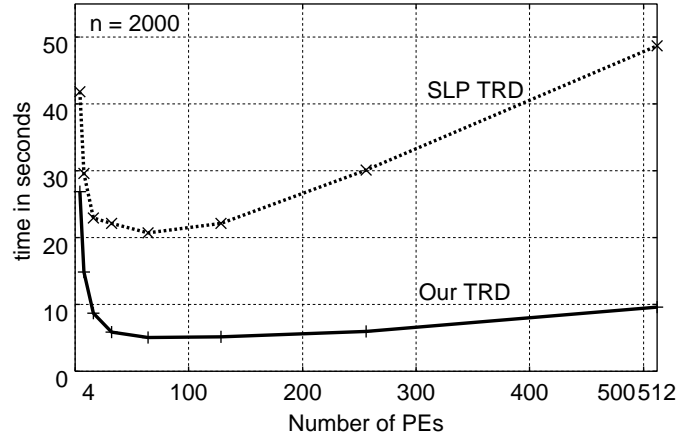
(c) Case of $PE = 64$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP / Ours
100	0.21 (4×16, 100)	0.153 (8×8)	1.3
200	0.98 (1×64, 100)	0.278 (8×8)	3.5
400	2.82 (4×16, 100)	0.638 (8×8)	4.4
800	6.60 (8×8, 40)	1.402 (8×8)	4.7
1000	8.79 (8×8, 40)	1.612 (8×8)	5.4
2000	20.73 (8×8, 40)	5.105 (8×8)	4.0
4000	57.6 (8×8, 40)	19.631 (8×8)	2.9
8000	210.49 (8×8, 60)	119.065 (8×8)	1.7

Table 3.7: Performances for tridiagonalization II (SR2201). Unit is second.

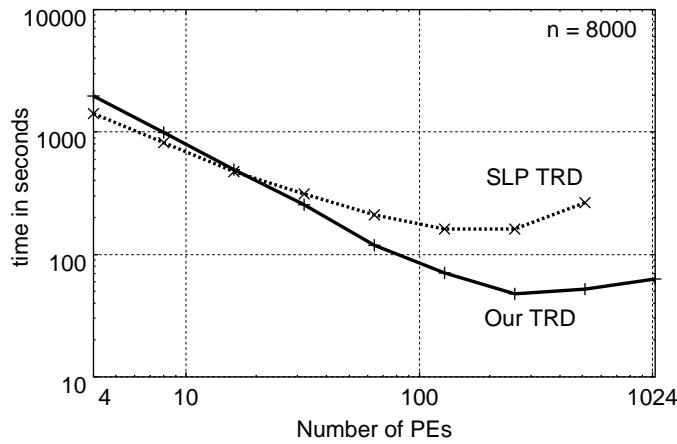
(a) Case of $PE = 128$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP / Ours
200	1.93 (8×16 , 100)	0.462 (8×16)	4.1
400	3.78 (8×16 , 60)	0.860 (8×16)	4.3
800	7.68 (8×16 , 80)	1.650 (8×16)	4.6
1000	9.74 (8×16 , 100)	2.109 (8×16)	4.6
2000	22.17 (8×16 , 40)	5.122 (8×16)	4.3
4000	54.13 (8×16 , 40)	15.420 (8×16)	3.5
8000	162.01 (8×16 , 40)	70.422 (8×16)	2.3
10000	245.60 (8×16 , 40)	123.891 (8×16)	1.9

(b) Case of $PE = 256$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP / Ours
400	5.69 (16×16 , 60)	1.373 (16×16)	4.1
800	10.17 (16×16 , 80)	2.480 (16×16)	4.1
1000	12.89 (16×16 , 100)	3.217 (16×16)	4.0
2000	30.12 (16×16 , 40)	5.964 (16×16)	5.0
4000	67.29 (16×16 , 40)	14.338 (16×16)	4.6
8000	161.76 (16×16 , 100)	47.906 (16×16)	3.3
10000	226.11 (16×16 , 40)	79.889 (16×16)	2.8
20000	774.10 (16×16 , 60)	454.267 (16×16)	1.7

(c) Case of $PE = 512$			
Size	SLP TRD (Grid, BL)	Our TRD (Grid)	SLP / Ours
1000	26.34 (16×32 , 40)	4.552 (16×32)	5.7
2000	48.76 (16×32 , 80)	9.613 (16×32)	5.0
4000	111.93 (16×32 , 40)	20.484 (16×32)	5.4
8000	265.77 (16×32 , 40)	52.397 (16×32)	5.0
10000	325.76 (16×32 , 60)	81.450 (16×32)	3.9
20000	827.96 (16×32 , 40)	302.541 (16×32)	2.7



(a) Case of $n = 2000$



(b) Case of $n = 8000$

Figure 3.3: Execution time for SLP TRD and Our TRD in the tridiagonalization (SR2201).

- (II) Our TRD has a lower communication complexity than the SLP TRD because of the non-symmetry in our TRD.

As for the reason (I), T.Katagiri and Y.Kanada [15], P.Stranzdings [22], and B.Hendrickson *et al.* [10] pointed out the following. Parallel libraries must be constructed by using different blocking factors of block length in data distribution (BDD) and block length in blocking algorithm (BBA). The BBA does not depend on the data distribution. Therefore, the value of BDD must be taken as small as possible from the viewpoint of parallel performance. In ScaLAPACK, however, the value of BDD is equal to BBA for the reason of easy construction of their libraries. This construction policy causes poor load balancing on MPP environments. Our process only supports (Cyclic, Cyclic) distribution (the value

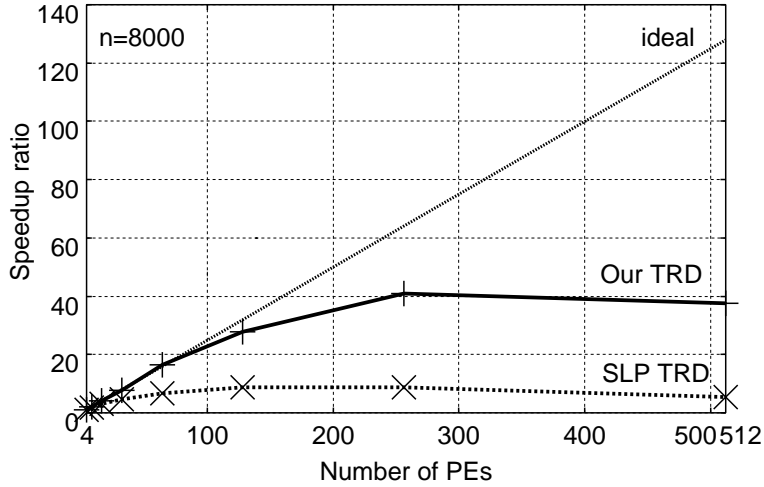


Figure 3.4: Speed-up ratios between SLP TRD and Our TRD for the tridiagonalization ($n = 8000$, SR2201).

of BDD is 1). Therefore, our routine never suffers from poor load balancing.

The reason (II) shows the fact that when n/p is very small, algorithms by the HTN will be faster than the HTS, since the HTN algorithms have less communication with respect to the HTS. Of course, when n/p is large enough, execution time will mainly depend on computation time since computational complexity is $O(n^3)$. This can be explained by Table 3.6(a), $p = 4$, $n = 8000$ case. In this case, SLP TRD was about 1.3 times as fast as our TRD. However, when the number of PEs increases, a larger matrix is needed to exceed the performance of our TRD. For example, Table 3.6(a) shows that the execution time of a 4000×4000 matrix was 231 seconds (SLP TRD) vs. 242 seconds (our TRD), and the ratio of executions was 0.95, and Table 3.6(b) shows that the execution time of a 8000×8000 matrix was 474 seconds (SLP TRD) vs. 490 seconds (our TRD), and the ratio of executions was 0.96. This means that in order to exceed our TRD, matrix sizes must be large in SLP TRD, and the execution time will be also large.

From Table 3.6(a) ($p = 4$, $n = 800$, and $n = 8000$), we can determine the factors in the formula (3.1) as $\delta_1 \approx 4268/(8000)^3$, $\gamma_1 \approx 3.58 \times 10^{-6}$, $\delta_2 \approx 5888/(8000)^3$, and $\gamma_2 \approx 3.83 \times 10^{-7}$. By using these factors, we will obtain the factor of $C_{\gamma/\delta} \approx 1011$, and this shows a fact that the $C_{\gamma/\delta}$ can have a considerable value. This factor affects execution times for an application. In applications in the chemical field, 6000 or more diagonalizations are needed [84]. Therefore in such applications, the execution time per diagonalization is limited, since total execution time become enormous. Even if execution time per one

diagonalization is 100 seconds, total execution time will be about 166 hours (about 7 days!) From the results of Table 3.6 and Table 3.7, under 100 seconds diagonalizations were about 2–5 times faster than SLP TRD. This shows that 35 days diagonalizations by using SLP TRD will be reduced by 7 days using our TRD. So, executing small size diagonalization at high speed is very important. For such applications, our routine will be a more effective tool than the conventional routines.

3.4.3 Comparison to other packages

Next is the comparison of compare efficiency to the theoretical peak performance when data per PE is small. Figure 3.5 shows the efficiency. In Figure 3.5, the efficiency of several tridiagonalization packages in different computer environments is also shown.

The efficiency in Figure 3.5 indicates that although algorithms and computer environments are different, our tridiagonalization routine is high in efficiency even if n/p is small. From this result, we can conclude that our method is very useful in the case of small n/p . This also shows that our method is efficient enough in MPP environments.

3.5 Conclusion

In this chapter, the author formulated, implemented and evaluated a parallel routine which can calculate all eigenvalues on an MPP system. By using the Householder tridiagonalization using a non-symmetry algorithm and the (Cyclic, Cyclic) data distribution, we could obtain a better performance than the ScaLAPACK routine which is widely used as a parallel library. This effect becomes stronger when the number of PEs increases. Therefore, our process is very efficient on MPPs.

In RISC based processors, it is known that blocking algorithms are more efficient than non-blocking algorithms [26]. However, blocking algorithms increase communication complexity for tridiagonalization [15]. In addition, using symmetry requires complex communication. Therefore, efficient inner processor algorithms and inter processor algorithms are different. To accomplish high performance, we have to analyze their parallel performances theoretically. The analysis and implementation of non-blocking vs. blocking algorithms, and symmetry vs. non-symmetry algorithms are the parts of future work.

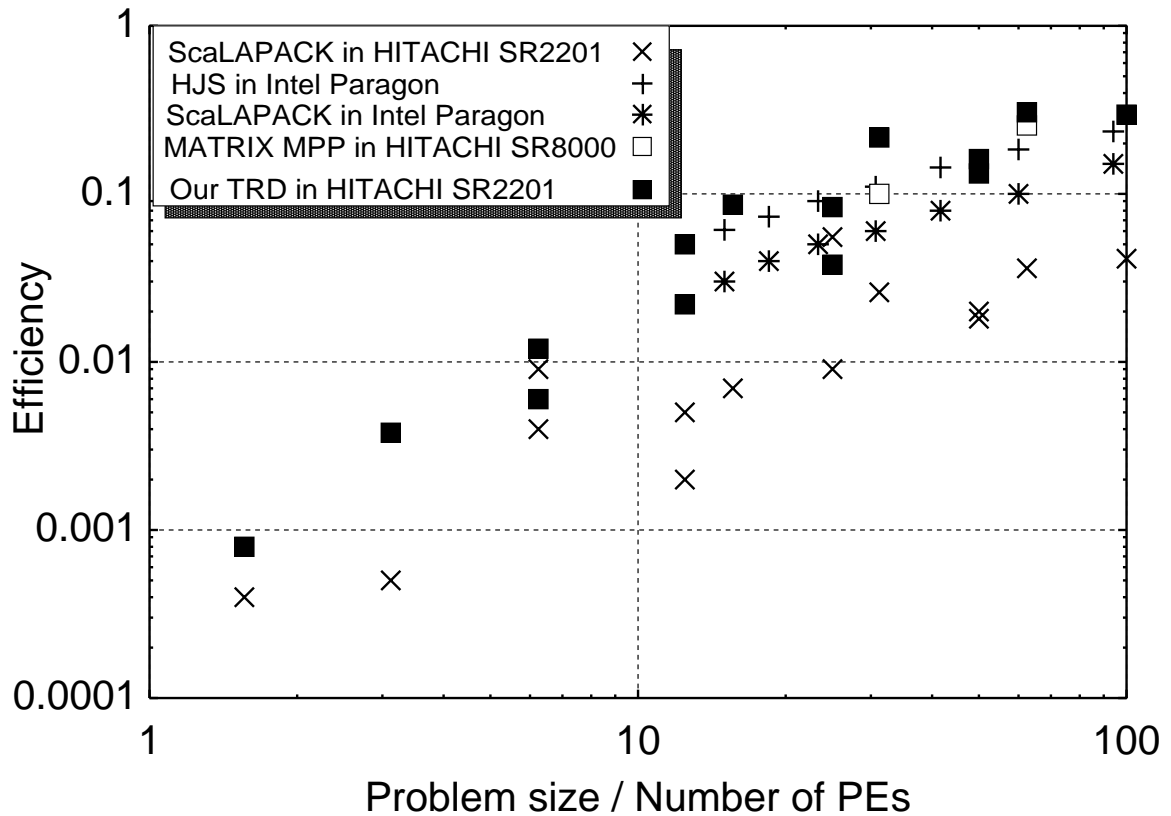


Figure 3.5: Efficiency to the theoretical peak performance. X-axis shows number of n / p , where n is problem size and p is total number of processors. The notation HJS means B.Hendirckson E.Jessup C.Smith solver (called HJS solver), and the performance of the HJS solver is shown in [10]. The performance of ScaLAPACK in Intel Paragon is also shown in [10]. The performance of the MATRIX MPP, which is provided by HITACHI Limited, is shown in [85].

Chapter 4

New Parallel Orthogonalization Method

4.1 Introduction

The orthogonalization process is one of the most important process to perform several linear algebra computations, such as eigendecomposition, and QR decomposition. It is known that the parallelism of the process depends on the number of orthogonalized vectors needed. Therefore, the orthogonalized process is classified as the following two kinds of processes in this chapter:

- (QR decomposition)

The orthogonalization process to obtain the normalized and orthogonalized vectors q_1, q_2, \dots, q_n by using the normalized vectors a_1, a_2, \dots, a_n .

- (Re-orthogonalization)

The orthogonalization process to obtain the normalized and orthogonalized vector q_i by using the normalized and orthogonalized vectors q_1, q_2, \dots, q_{i-1} .

This chapter discusses how to parallelize the above two orthogonalization process by using the Gram-Schmidt (GS) orthogonalization method. The focus is especially on the parallelization of Classical GS (CGS) method, since the method has high parallelism. However the CGS method has an accuracy problem.

First aim of this chapter is to propose the CGS method with sorting to improve the accuracy problem. This proposal is a new concept in the GS orthogonalization. This new method is called as *CGSS (Classical GS with Sorting)*. Many researchers reports the idea of changing calculation orders to improve accuracy. For example, several methods for changing floating-addition order and its analysis have been suggested by N.Higham[86]. Its application of multiple-precision calculation has reported in [87]. The dynamically

controlled method [43] to improve orthogonal accuracy, which is block algorithm of GS method, have been studied.

On the other hand, the Modified GS (MGS) method is useful in a data distribution. Second aim of this chapter is to propose a new data distribution for the MGS method.

First of all, Chapter 4.2 describes sequential algorithms of GS method. Chapter 4.3 discusses the parallel algorithms based on the sequential GS method, and explains the new data distribution. Chapter 4.4 explains the CGSS method in detail. Chapter 4.5 summarizes the CGSS method and conventional methods from the viewpoint of accuracy and parallel execution performance. Chapter 4.6 is the evaluation of the CGSS method from the viewpoint of accuracy and execution time for conventional methods by numerical experiments. In addition, we evaluate the new data distribution. Finally, we summarize new findings of this chapter in Chapter 4.7.

4.2 Sequential algorithms of GS orthogonalization method

4.2.1 QR decomposition

It is widely known that there are following two methods to perform QR decomposition by using the GS method. They are Classical GS (CGS) method and Modified GS (MGS) method. The CGS method is simply implemented in the formula of the GS orthogonalization, and the MGS method is modified in order to obtain high accuracy.

The MGS method in the QR decomposition is shown in Figure 4.1. The notation of (\cdot, \cdot) in Figure 4.1 means an inner product.

<pre> <1> do $i = 1, n$ <2> $a_i^{(0)} = a_i$ <3> do $j = 1, i - 1$ <4> $a_i^{(j)} = a_i^{(j-1)} - (q_j^T, a_i^{(j-1)})q_j$ <5> enddo <6> Normalization of $q_i = a_i^{(i-1)}$. <7> enddo </pre>
--

Figure 4.1: The MGS method in the QR decomposition.

Here, let the matrix A be a matrix which is composed by the row vectors of the initial vectors a_i , the matrix Q be an orthogonalized matrix which is composed by the row vectors of the orthogonalized vectors q_i in Figure 4.1, and the upper triangular matrix R

be a matrix which is composed by the inner-producted values $(q_j^T, a_i^{(j-1)})$ as its j -th row and i -th column. By using these matrices, we will obtain the following decomposition:

$$A = QR. \tag{4.1}$$

Therefore, we call the process in Figure 4.1 “QR decomposition.”

Next is the data dependency of the QR decomposition for the MGS method shown in Figure 4.2.

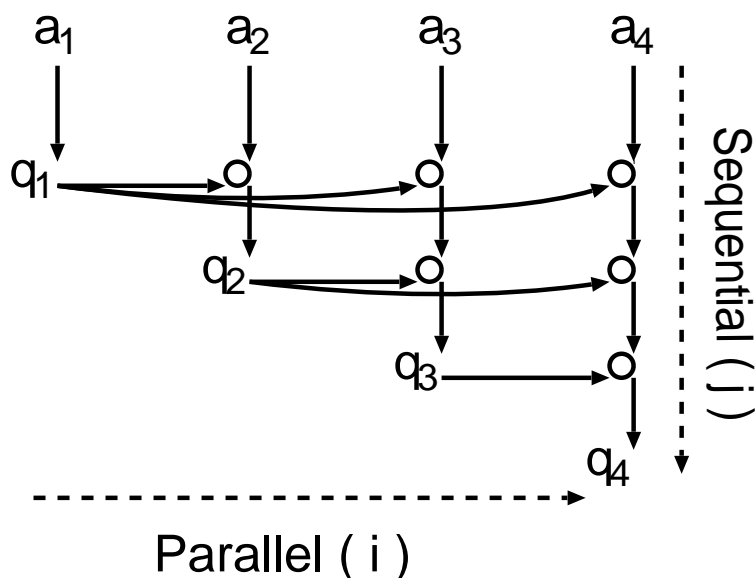


Figure 4.2: Data dependence graph of the MGS method in the QR decomposition. The notation \bigcirc shows the inner kernel process $\langle 4 \rangle$ of the MGS method in Figure 4.1.

Figure 4.2 shows that there is a parallelism for vectors which will be orthogonalized (the direction of the i -loops) when we obtain the vector q_i . On the other hand, we find that there is no parallelism for vectors which is orthogonalizing (the direction of the j -loops).

The CGS method to perform the QR decomposition is shown in Figure 4.3.

```

<1>  do  $i = 1, n$ 
<2>     $q_i = a_i$ 
<3>    do  $j = 1, i - 1$ 
<4>       $q_i = q_i - (q_j^T, a_i)q_j$ 
<5>    enddo
<6>    Normalization of  $q_i$ .
<7>  enddo

```

Figure 4.3: The CGS method in the QR decomposition.

Figure 4.4 shows the data dependence graph of the CGS method to perform the QR decomposition.

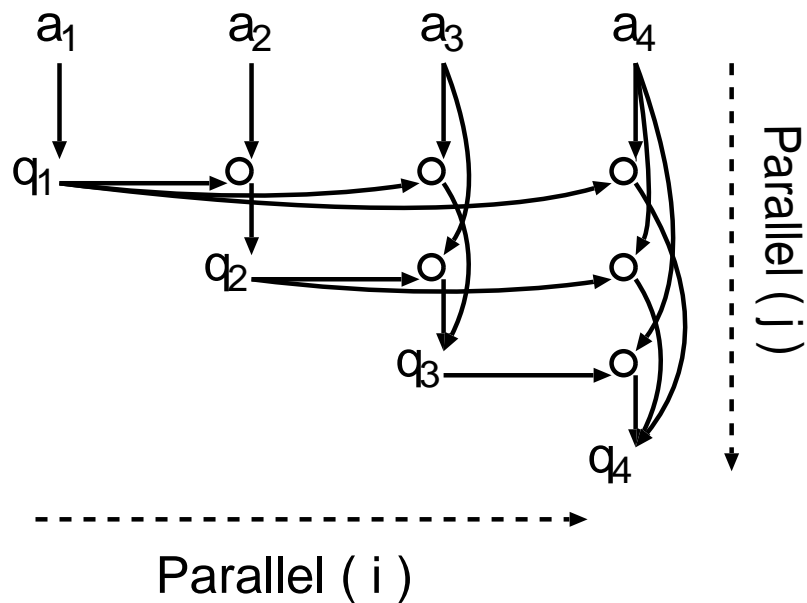


Figure 4.4: Data dependence graph of the CGS method in the QR decomposition.

From Figure 4.4, we find that there is a parallelism between vectors which will be orthogonalized (the direction of the i -loop) and vectors which is orthogonalizing (the direction of the j -loop).

4.2.2 Re-orthogonalization

The re-orthogonalization process is implemented by removing the outer loop of i in both of the MGS and CGS methods. Therefore, it is easy to write these data dependence graphs by removing the data dependency of $a_1 - a_3$ in Figures 4.2 and 4.4.

4.3 Parallelization of GS orthogonalization method

4.3.1 QR decomposition

4.3.1.1 A case of row-wise distribution

In the row-wise distribution, the elements of the normalized vectors a_i and the normalized and orthogonalized vectors q_i ($i = 1, 2, \dots, n$) are distributed to each PEs.

Let the matrix A be denoted as $A = (a_1, a_2, \dots, a_n)$, and the matrix Q be denoted as $Q = (q_1, q_2, \dots, q_n)$. By using these matrices, the row-wise distribution can be denoted as (Block, *) or (Cyclic, *) distribution for the matrices A and Q . To calculate the dot-products of $(q_j^T, a_i^{(j-1)})$ and (q_j^T, a_i) , we need a scalar reduction operation in both of MGS and CGS methods for the row-wise distribution, since each PEs does not have whole elements to calculate the dot-products.

Figure 4.5 shows a row-wise parallel algorithm for the MGS method. The notation “Global sum” in Figure 4.5 is the collective communication operation, which sums up distributed data and then distributes the result to all PEs. The collective communication can be implemented by using the `MPI_ALLREDUCE` function on the MPI (Message Passing Interface).

$\langle 1 \rangle$	do $i = 1, n$
$\langle 2 \rangle$	$a_i^{(0)} = a_i$
$\langle 3 \rangle$	do $j = 1, i - 1$
$\langle 4 \rangle$	Local $(q_j^T, a_i^{(j-1)})$
$\langle 5 \rangle$	Global sum of $\eta = (q_j^T, a_i^{(j-1)})$.
$\langle 6 \rangle$	Local $a_i^{(j)} = a_i^{(j-1)} - \eta q_j$
$\langle 7 \rangle$	enddo
$\langle 8 \rangle$	Normalization of $q_i = a_i^{(i-1)}$.
$\langle 9 \rangle$	enddo

Figure 4.5: The MGS method in row-wise distribution.

4.3.1.2 A case of column-wise distribution

In the column-wise distribution, the pair of vectors (a_i, q_i) for the normalized vectors a_i and the normalized and orthogonalized q_i are distributed to same PE. Matrices A and Q are defined as the same. By using the matrices, the column-wise distribution is denoted as $(*, \text{Block})$ or $(*, \text{Cyclic})$ distribution for the matrices A and Q . As for the data dependency, the $(*, \text{Block})$ distribution has worse load balancing than the $(*, \text{Cyclic})$ distribution. Therefore, the $(*, \text{Cyclic})$ distribution is desirable. If we select the $(*, \text{Cyclic})$ distribution, however, we have to broadcast a normalized and orthogonalized vector to every PEs in each step. To reduce the broadcasting, there is another choice of distribution. That is, the $(*, \text{Cyclic}(m))$ distribution.

Next is a discussion on how to implement parallel algorithm based on the column-wise distribution. From the difference of communication types, there are some variations of the parallel algorithm. The following is the description on the algorithms.

[A case of using blocked communication]

Blocked communication is chosen to broadcast each normalized and orthogonalized vectors q_i , there will be only one parallel implementation on both the MGS and CGS methods, since we cannot use the parallelism for the j -loop in the column-wise distribution.

Figure 4.7 shows an example for the QR decomposition with blocked communication.

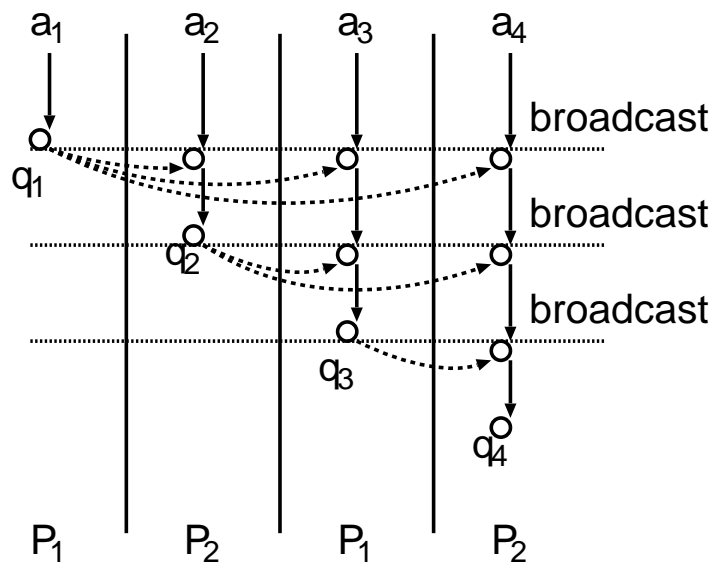
As shown in Figure 4.7, the block-cyclic distribution decreases communication time. The block-cyclic distribution, therefore, is a better distribution than the cyclic distribution.

[A case of using non-blocked communication]

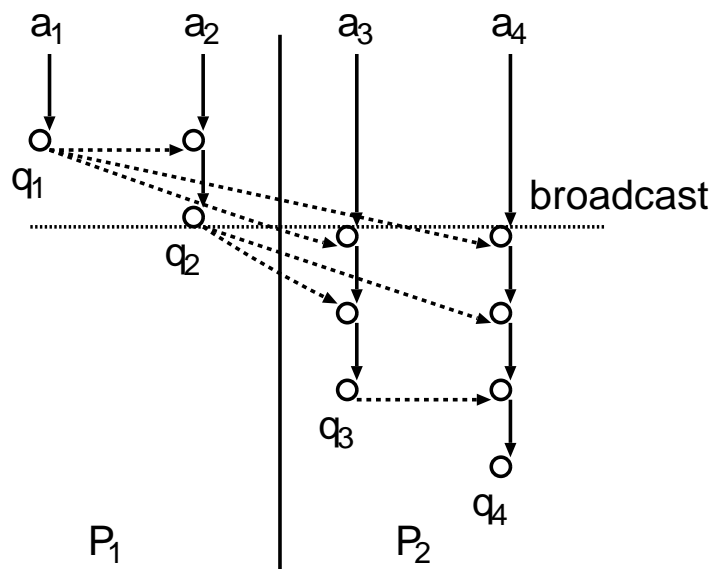
By using non-blocked communication, an overlapped parallel algorithm for communication and calculation can be implemented. In addition, we can also implement different types of parallel algorithms between the MGS and CGS methods.

For example, Figure 4.8 shows the data dependence graph of the MGS algorithm by using non-blocked communication.

In the algorithm of Figure 4.8, every PE calculates the orthogonalized and normalized vector which is needed later at first as soon as it obtains the essential vector, and sends the orthogonalized vector by using non-blocked communication. The overhead of communication is completely hidden in a condition in this algorithm. This implies that each PEs can obtain the essential vector as soon as they start the calculation, so there is not any waiting time in a condition.



(a) A case of cyclic distribution.



(b) A case of block-cyclic distribution. The block length m is 2.

Figure 4.7: An example for parallel MGS algorithms with blocked communication in the QR decomposition.

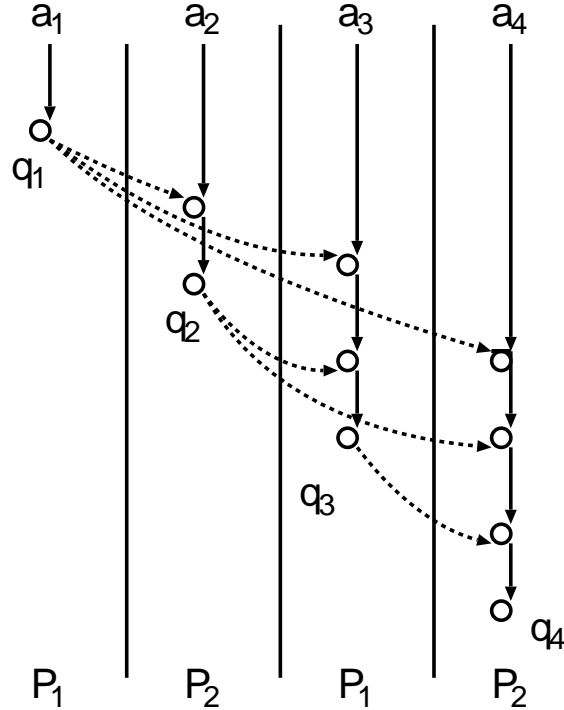


Figure 4.8: An example for the data dependence graph of the MGS algorithm by using non-blocked communication. The data distribution is the cyclic distribution.

Note that the cyclic distribution is used in the example in Figure 4.8, but the another non-blocking algorithm can be implemented by using block-cyclic distribution. This block-cyclic algorithm is reduced communication times against the cyclic ones.

Many kinds of non-blocking algorithm is implemented in the CGS method. These algorithms do not need the received orthogonalized and normalized vectors sequentially (q_1, q_2, \dots, q_n) . Therefore the calculation can be started whenever the vector is obtained, and this shows that there are more flexibility to parallel algorithms.

From the viewpoint of these variations, the column-wise distribution will be good choice. However, the discussion on this distribution is omitted in this chapter to simplify the discussion.

4.3.1.3 Proposition of new data distribution

This section proposes a new data distribution method for the QR decomposition with the column-wise distribution. The new data distribution is defined as the mapping that each column of A is mapped to $0, 1, 2, \dots, p-1, p-1, \dots, 2, 1, 0, 0, 1, \dots$, where

the number means the identification number of PEs. This distribution is called *Cyclic Triangular Distribution (CTD)*. The notation applying the CTD distribution to each rows of matrices is defined as $(*, \text{CTD})$. Figure 4.9 shows an example of the CTD distribution.

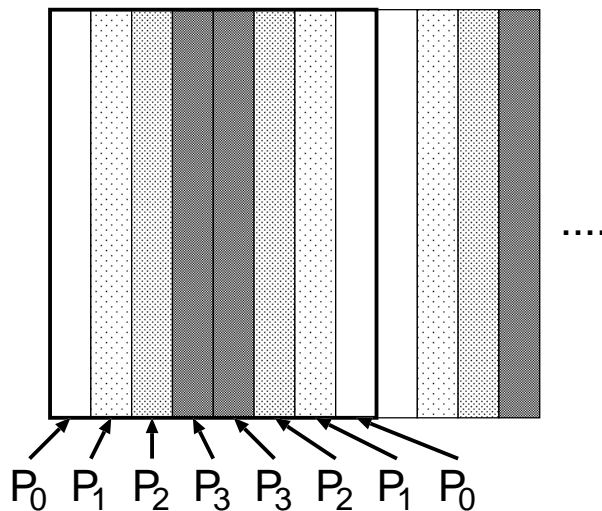


Figure 4.9: The Cyclic Triangular Distribution $(*, \text{CTD})$.

Same distribution strategy is proposed by T.Scheriber *et.al.*[88]. They called this distribution as *Team mapping*. However, they applied the Team mapping to QR decomposition with Givens method for hermitian matrix (NOT GS method). In addition, they did not implement the mapping to real parallel machines, and evaluate the performance. So, there is no data for its performance at present.

In the QR decomposition with column-wise distribution, why the CTD distribution is superior to the conventional distribution, such as $(*, \text{Cyclic})$? The answer is that the process features are formed by triangle (see Figures 4.2 and 4.4).

The example in Figure 4.10 is an explanation.

From Figure 4.10, we find that the $(*, \text{CTD})$ distribution has perfect load balancing when the number of vectors n can be divided by the number of PEs p . If it is not divided, however, the $(*, \text{Cyclic})$ and $(*, \text{CTD})$ distributions have a load imbalance. This load imbalance is caused by the nature of column-wise distribution, so the improvement of the load imbalance is difficult.

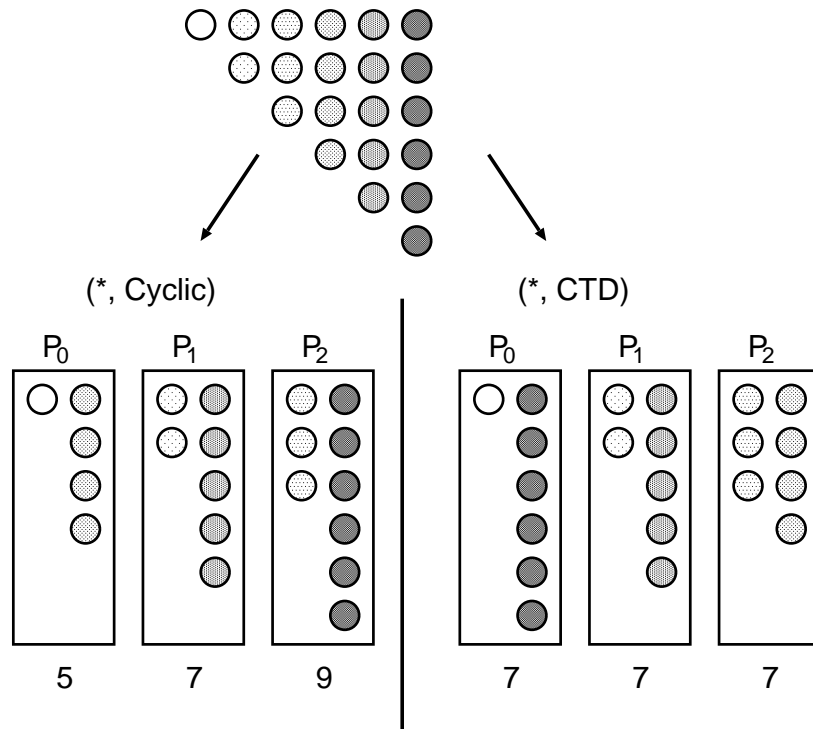


Figure 4.10: An example of advantage for (*, CTD).

4.3.2 Re-orthogonalization

4.3.2.1 A case of row-wise distribution

In both of the MGS and CGS methods, this distribution can be implemented by removing the most outer loop of i -loop, and then setting the number i in Figures 4.5 and 4.6 of vector which is orthogonalized by $i - 1$ orthogonalized vectors. This process is almost same as QR decomposition, so explanation on this process in detail is omitted.

4.3.2.2 A case of column-wise distribution

When the re-orthogonalization is attempted to be parallelized with the column-wise distribution in the MGS method, this algorithm cannot be parallelized because there is no parallelism. Hence, there is a flow dependency in the direction of the j -loop in Figure 4.5. On the other hand, since there is parallelism in the direction of the j -loop, the CGS method can be parallelized. From this reason, we have implemented a parallel algorithm using the CGS method with the column-wise distribution in our parallel eigensolver [39, 89].

4.4 New method for improving accuracy in the CGS method

This section considers the factors of break-down for accuracy in the CGS method. This break-down is caused by the following two factors:

Factor (i): A collapse of orthogonality in theory.

Factor (ii): A swamp in floating-point subtraction or addition.

As for the factor (i), this collapse can not be avoided when the QR decomposition is performed. In re-orthogonalization case, however, even if the CGS method is used, the collapse from the factor (i) in theory can be avoided if and only if the vectors which will be orthogonalized at high accuracy is computed. From this idea, we can create a new re-orthogonalization method, named *HCGS (Hybrid CGS)* method.

In the CGS method, we think that the factor (ii) can strongly affect its accuracy. To discuss the effect of the factor (ii), the Björck's error analysis is shown below. Björck shows that the main operation of the GS method:

$$a_i^{(j)} = a_i^{(j-1)} - \eta_{ji}q_j \quad (4.2)$$

causes heavy collapse under a condition [90]. The condition is

$$\|a_i^{(j)}\|_2 \ll \|a_i^{(j-1)}\|_2. \quad (4.3)$$

Note that this result can be applied to both of the MGS and CGS methods in the viewpoint of main operation, although each data dependency is different.

From the discussion above, an idea is obtained to solve the accuracy problem in the CGS method. The idea is to change the calculation order for the orthogonalization:

$$a_i^{(j)} = a_i - \eta_{1i}q_1 - \eta_{2i}q_2 - \cdots - \eta_{i-1,i}q_{i-1}$$

in order to make less difference from the point of each η_{ji} values. From applying the idea, the reduction of the probability of the condition (4.3) is expected. Therefore, the CGS method applied is taken after sorting by each values of η_{ji} . This new CGS orthogonalization is called *CGSS (CGS with Sorting)*. Figure 4.11 shows the CGSS method.

On the other hand, we can take the CGSS method instead of the CGS method in Figure 4.11. This method is called *HCGSS (HCGS with Sorting)* method.

⟨1⟩	$q_i = a_i$
⟨2⟩	do $j = 1, i - 1$
⟨3⟩	$w_j = (q_j^t, a_i)$
⟨4⟩	enddo
⟨5⟩	$(sw_1, sw_1, \dots, sw_{i-1}) = \text{Sort}(w_1, w_2, \dots, w_{i-1})$
⟨6⟩	do $j = 1, i - 1$
⟨7⟩	$idx = \text{Index}(sw_j)$
⟨8⟩	$q_i = q_i - sw_j \cdot q_{idx}$
⟨9⟩	enddo

Figure 4.11: The CGSS method for re-orthogonalization. The function `Sort()` in the Figure means a sorting function. The function `Index()` returns before-sorting indices from after-sorting indices.

4.5 Summary of parallel algorithms

This section summarizes the orthogonalization methods. Table 4.1 shows the classification of re-orthogonalization methods from the viewpoint of the referenced vectors.

Note that the QR decomposition can also be classified, since the QR decomposition is composed by the re-orthogonalizations. For instance, the HCGS method in the QR method is a process which changes its orthogonalization method before the whole process is finished * .

On the other hand, parallel GS methods in the viewpoint of parallel performance and accuracy is summarized in Table 4.2.

In Table 4.2, there is a tread-off in the viewpoint of accuracy and parallel performance except for row-wise distribution in the QR decomposition. Therefore, the best method is yet undetermined.

4.6 Performance evaluation

Each GS methods are implemented and evaluated on the HITACHI SR2201 at the Computer Division, the Information Technology Center, the university of Tokyo † .

*For example, it is remarkable method that a QR decomposition which uses the MGS method in the first few columns, and changes the CGS method after that. The method based in this idea is known as the blocked Gram-Schmidt algorithm[91].

†The experimentation days were January and February, 1999.

Table 4.1: Summary for parallel GS methods (1).

Referenced vectors	Re-orthogonalization method	Name
Orthogonalized by the CGS	CGS	CGS
	MGS	HMGS
	CGSS	HCGSS
Orthogonalized by the MGS	CGS	HCGS (Blocked GS [91])
	MGS	MGS
	CGSS	HCGSS
Orthogonalized by the CGSS	CGS	HCGS
	MGS	HMGS
	CGSS	CGSS

4.6.1 Conditions of our experiments

4.6.1.1 Definition of orthogonality

First of all is the definition of the orthogonality for orthogonalized vectors. The following norm for the matrix $Q = (q_1, q_2, \dots, q_n)$ which consists of each orthogonalized vectors q_j ($j = 1, 2, \dots, n$) is used:

$$\|I - Q^T Q\|_E, \quad (4.4)$$

where the notation $\|\cdot\|_E$ is computed by

$$\|B\|_E = \sqrt{\sum_{i,j} b_{ij}^2} \quad (4.5)$$

where $B = (b_{i,j})$. The orthogonality is defined for the QR decomposition by using the norm (4.4).

As for re-orthogonalization, its orthogonality is defined as

$$\begin{aligned} & \text{if } i \neq j, & \max \sqrt{|q_i^T q_j|} \quad (i, j = 1, 2, \dots, k), \\ & \text{otherwise,} & \max \left| 1 - \sqrt{|q_i^T q_i|} \right| \quad (i = 1, 2, \dots, k). \end{aligned} \quad (4.6)$$

These orthogonalities are checked by using quadruple precision floating operations. Double precision floating operations are used in the QR decomposition and re-orthogonalization.

Table 4.2: Summary for parallel GS methods (2). The notation of [Parallelism] “High” : High performance will be achieved when the number of PEs increases. “Middle” : High performance will not always be achieved when the number of PEs increases. “Low” : High performance will not be achieved when the number of PEs increases. “None” : There is no parallelism. The notation of [Utility] “○” : Now being used, or can be used. “×” : Not used because there is no benefit.

Process	Data distribution	Method	Parallelism	Accuracy	Utility
QR decomposition	Row-Wise	CGS	High	Low	○
		CGSS	High	High/Low	Unknown
		MGS	Low	High	○
	Column-Wise	CGS	High	Low	×
		CGSS	High	High/Low	×
		MGS	High	High	○
Re-orthogonalization	Row-Wise	CGS	High	Low	○
		CGSS	High	High/Low	Unknown
		HCGS	High	High/Low	Unknown
		HCGSS	High	High/Low	Unknown
		MGS	Low	High	○
	Column-Wise	CGS	Middle	Low	○ [39, 89]
		CGSS	None	High/Low	×
		HCGS	Middle	Middle	○ [39, 89]
		HCGSS	None	High/Low	×
		MGS	None	High	○ [39, 89]

4.6.1.2 Test matrices

The following two kinds of matrix were used as test matrix.

- Randomized matrix

The matrix which has the randomized elements of α . The elements of α are generated by a pseudo-random function, and its range is $0 \leq \alpha < 1$.

- Läuchli Matrix [90]

The matrix is defined as the following:

$$A = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \epsilon & & & \\ & \epsilon & & \\ & & \ddots & \\ & & & \epsilon \end{pmatrix} \quad (4.7)$$

The Läuchli matrix has a character that the values of ϵ can control the orthogonality of the QR decomposition and re-orthogonalization.

4.6.1.3 Implementation details of the CGSS method

It is expected that the orthogonalization process after sorting can affect the accuracy in the CGSS method. For example, the following factors must be considered:

- (i) Not consider ordering for positive and negative values.
- (ii) Switch operation following the sign of the values.
- (iii) After computations for positive and negative values, the two results are added.
 - (1) Sort as signed real value.
 - (2) Sort as absolute real value.
 - (a) Start computing from small values.
 - (b) Start computing from large values.

These experiments evaluate the CGSS method by using the following nine kinds of implementations.

Notation	Implementation	Notation	Implementation
CGSSsp	(i)(1)(a)	CGSSdsp	(ii)(1)(a)
CGSSsm	(i)(1)(b)	CGSSdsm	(ii)(1)(b)
CGSSap	(i)(2)(a)	CGSSdap	(ii)(2)(a)
CGSSam	(i)(2)(b)	CGSSdam	(ii)(2)(b)
		CGSSdsam	(iii)(2)(b)

4.6.2 QR decomposition

4.6.2.1 A case of row-wise distribution

Figure 4.12 shows the execution time of each orthogonalization methods in the QR decomposition for $n = 1500$. From Figure 4.12, it is clear that there is no parallel effect in

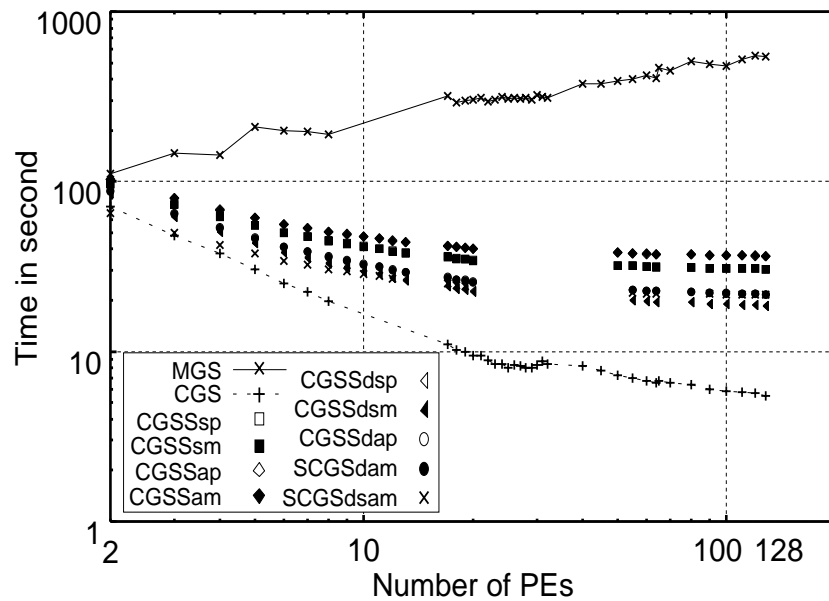


Figure 4.12: Execution time in the QR decomposition. (The row-wise distribution, randomized matrix, $n = 1500$)

the MGS method. We think that this result was caused by increasing communication time as the number of PEs increases, since the MGS method needs scalar reduction operation in every steps.

On the other hand, there are parallel effects in both of the CGS and CGSS methods. From these results, the conclusion is made that reducing scalar reduction operation can greatly affect total parallel performances.

The CGS method was faster than the CGSS method in Figure 4.12, since the CGSS method needs extra sorting process. The sorting time, however, can be reduced because the CGSS methods in Figure 4.12 used $O(n^2)$ selecting method. By using $O(n \log n)$ sorting algorithms, such as quick sort, this sorting time can be reduced. In addition, the CGSS method with considering signs was faster than the other CGSS methods. We think that this result was caused by reducing the number of data when we separated the data according to its sign, and this reducing was effective in $O(n^2)$ sort algorithms.

Figure 4.13 shows execution time in each method. The CGSS methods use the quick sort algorithm. The quick sort algorithm uses no recursive calls, and it stops its division when the length of data is less than 10, and then switches to the insertion method.

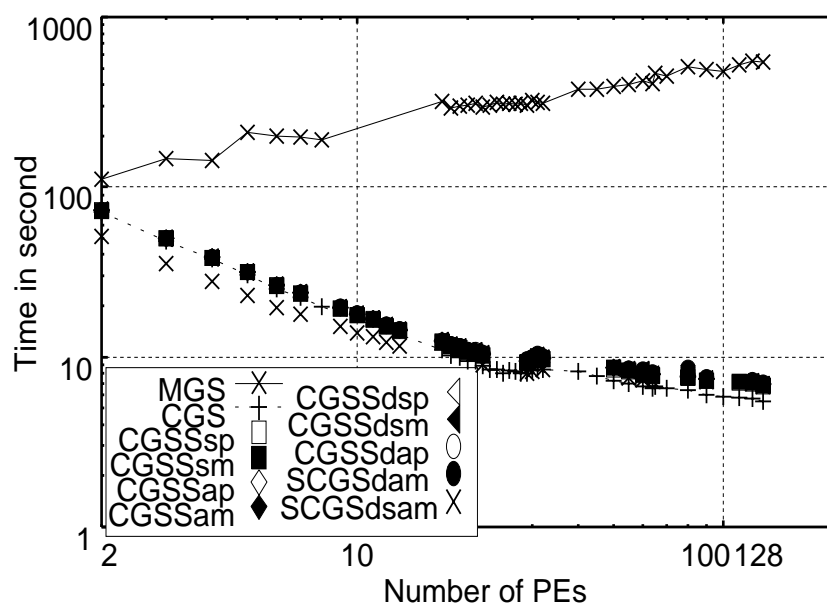


Figure 4.13: Execution time in the QR decomposition. (The row-wise distribution, randomized matrix, $n = 1500$. The quick sort is used.)

Figure 4.13 shows that by using the quick sort algorithm, the execution time can be reduced in the CGSS methods. In addition, it is certain that the CGSSdsam method was faster than the CGS method. We think that this result was caused by dividing the data, and the data was almost sorted data accidentally.

Figure 4.14 shows the orthogonalities of the QR decomposition. Figure 4.14 implies:

- i) The accuracy of the MGS method was half digit as high as the CGS method.
- ii) The CGSS methods does not always improve the accuracy against the CGS method.

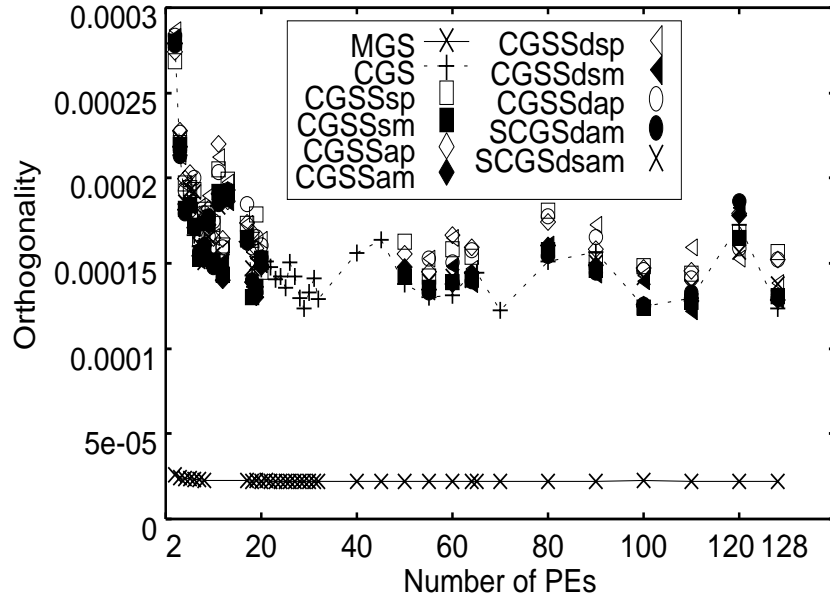


Figure 4.14: Orthogonalities in QR factorization. (randomized matrix)

- iii) There was a tendency to improve the accuracy when the number of PEs increased both of the CGS and CGSS methods.

From i) and ii), the CGSS methods are not always superior to the MGS method, and can not always obtain better accuracy than the CGS method. On the other hand, from iii), it is supposed that parallel processing gives us high accuracy. However, the parallel GS method in this chapter can be implemented sequentially by dividing the computation, and by saving the results into temporary buffers instead of local memory of each PEs. From this reason, conclusion can be made that “dividing the computation” gives us high accuracy. In addition, parallel processing gives us smaller memory sizes per PE and shorter execution time than sequential processing. From these merits, we can say that parallel GS methods are superior to sequential GS methods in the viewpoint of memory size and execution time.

4.6.2.2 A case of column-wise distribution

[Evaluation of the (*, CTD) distribution]

This section evaluates the (*, CTD) distribution of the QR decomposition for the MGS method with column-wise distribution. Figure 4.15 shows execution time of the QR decomposition for the MGS method with the (*, Cyclic) or (*, CTD) distributions by using blocked communication and non-blocked algorithm ($m = 1$).

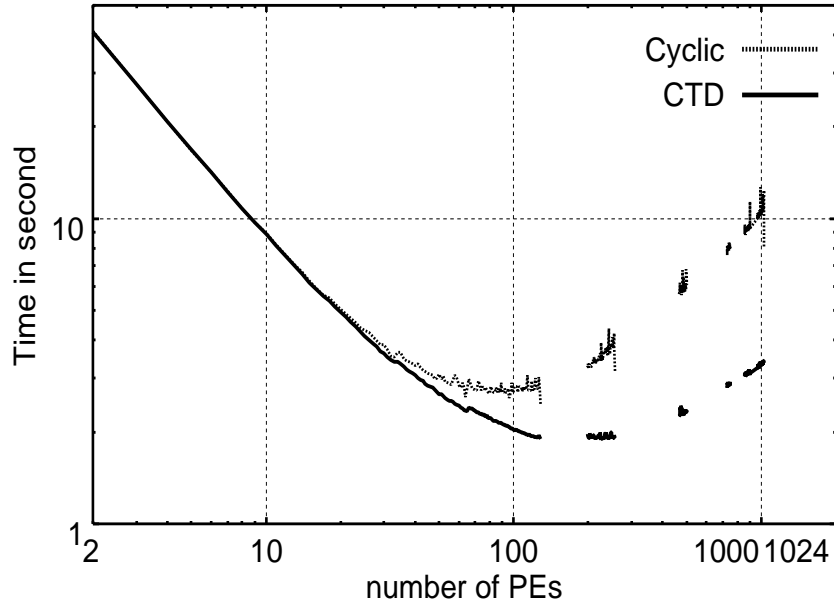


Figure 4.15: (*, CTD) vs. (*, Cyclic). ($n = 1500$, $m = 1$, the non-blocked MGS method using blocking communication.)

From the results of Figure 4.15, better performance could be obtained for the (*, CTD) distribution than the conventional (*, Cyclic) distribution when the number of PEs increased. It is assumed that these results are given by good load balancing for the (*, SWD), since the load balancing of (*, Cyclic) distribution becomes poor when the number of PE increases.

Figure 4.16 shows the fastest execution time of blocked MGS algorithm, where the length of blocking m is differed from 1 to 6.

Figure 4.16 shows that the blocking MGS algorithm gave us high performance, and there was no difference between the (*, Cyclic(m)) and (*, CTD(m)) distribution. We think that this result was caused by increasing data size per PE according to increase the block length. As a result, although the (*, CTD(m)) distribution, the load imbalance is caused, and then the (*, CTD(m)) distribution had no benefit.

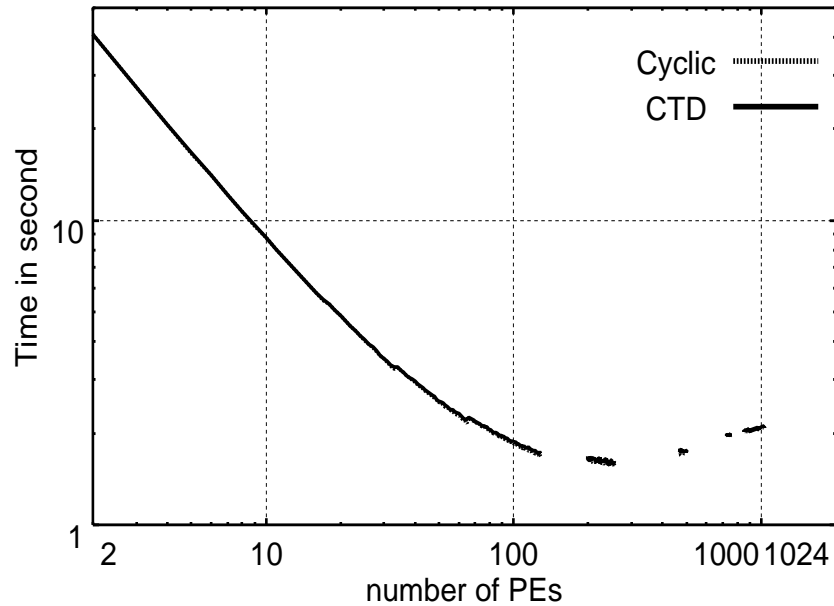


Figure 4.16: $(*, \text{CTD}(m))$ vs. $(*, \text{Cyclic}(m))$. ($n = 1500$, $m = \text{fastest}(1-6)$, the blocked MGS method using blocked communication.)

4.6.3 Re-orthogonalization

4.6.3.1 A case of row-wise distribution

First discussion of accuracy of re-orthogonalization for the CGSS method in a given matrix A . The focus is especially on the re-orthogonalization with the MGS orthogonalized vectors, since the vectors can greatly affect whole accuracy.

[A case of using the MGS orthogonalized vectors : randomized matrix]

Figure 4.17 shows the orthogonalities of re-orthogonalization with the MGS orthogonalized vectors for a randomized matrix. Figure 4.17 shows that

- i) The MGS method had better orthogonality.
- ii) “Dividing the computation” gave us high orthogonality.
- iii) There was no difference between the CGS and CGSS methods.

[A case of using the MGS orthogonalized vectors : the Lauchli matrix]

Figure 4.18 shows the orthogonalities of the Lauchli matrix for $\epsilon = 1$.

Figure 4.18 also shows that

- i) There was a case that the CGS or CGSS methods had higher accuracy than that of the MGS method.
- ii) “Dividing the calculation” gave us little benefit.
- iii) There was a difference for the orthogonalities between the CGS and CGSS methods.
- iv) In the CGSS method, the orthogonality of the implementation which starts the operation according to small values after sorting, was very poor.

We think that the findings of i) and iv) are very important, since it can be concluded from the following findings: (1) The orthogonality of the MGS and CGS methods depends on the characteristic of input matrices; (2) The orthogonality of implementations for the CGSS method also depends on the characteristic, and especially, the implementation which starts the operation according to small values after sorting may cause very deficient orthogonality.

4.6.3.2 Distribution of dot-producted values

Figure 4.19 shows the dot-producted values of (q_j^t, a_i) for re-orthogonalization in the CGS and CGSS methods. The number of PEs was fixed as $p = 16$, and the number of orthogonalized vectors was fixed as $k = 1500$ in Figure 4.19. The implementations of the CGSS method in Figure 4.19 are not-dividing and sorting method as signed values (CGSSsp, CGSSsm), and absolute values (CGSSap, CGSSam).

From Figure 4.19, we can find that

- i) There were relatively big values.
- ii) As for the randomized matrix, the dot-producted values changed around zero.
- iii) As for the Läuchli matrix, the values were all positive.

Especially, from i) and iii), we can conclude that

- Adding positive and relative big values from the smallest to the biggest value may cause very poor accuracy.

If the above hypothesis is true, it can be said that the CGS method has very low accuracy when all dot-producted values are positive and the values appear from small values. The CGSS method could improve the accuracy. Therefore, this conclusion shows that extra experiment and inspection are necessary.

4.6.4 Evaluation with an application program

This section evaluates the CGSS methods by using an application program. The application program is parallel eigensolver which can calculate all eigenvalues and eigenvectors by using the bisection and inverse iteration methods [39, 89]. The glued Wilkinson matrix W_g^+ which consists of the Wilkinson matrix of the order 21 dimension is used as the test matrix. This eigensolver is designed by the column-wise (*, Block) distribution for the storage of eigenvalues. From this distribution, the CGSS methods are sequentialized to determine calculation order after sorting, and so we cannot obtain parallel efficiency. Therefore, in this section, the discussion is only on the orthogonality and the convergence of the inverse iteration method.

In this experiment, the test matrices of 840, 1000, and 1260 dimensions were used. The eigenvalues for the test matrices are formed in a cluster of 40 eigenvalues (for 840 dimension) and 60 eigenvalues (for 1260 dimension) with value between -1.1 and 10.8 .

4.6.4.1 Orthogonality

Figure 4.20 shows the orthogonalities of all eigenvectors by using the CGSS re-orthogonalization. The notation “distance ” in Figure 4.20 means the value to control cluster eigenvalues. Setting large value of the distance gives us a big cluster of eigenvalues.

From Figure 4.20, the following is pointed out when the distance was set as big values (formed a big cluster of eigenvalues):

- When (a) $n = 840$ case, the CGSS methods were worse in orthogonality than conventional methods (the MGS and CGS methods).
- When (b) $n = 1000$ case, the CGS method had deficient orthogonality, but the CGSS methods could improve the orthogonality.
- When (c) $n = 1280$ case, the orthogonalities of the CGSS methods (CGSSsp, CGSSap) were one digit as high as the conventional methods.
- As for (a) and (c) cases, there was a case that the CGSS methods had worse orthogonality than the conventional methods.

The above results imply that there is a case to improve the accuracy by using the CGSS method. However, the results also implies that if we use an unsuitable implementation for the after sorted data, the orthogonality becomes very deficient.

4.6.4.2 The number of iterations until convergence

Figure 4.21 shows the average number of iterations until the convergence per eigenvector when the CGSS re-orthogonalization as for the orthogonalization process of our eigensolver is used. Figure 4.21 shows that the number of iterations until the convergence was reduced by using the CGSS method against that of the conventional methods. The conclusion is that the CGSS method may reduce execution time against that of the conventional methods.

4.7 Conclusion

This chapter proposes a new orthogonalization method of sorting, named the CGSS method. According to the results of numerical experiments, the CGSS method does not always improve accuracy, but it can dramatically improve accuracy in a case against conventional methods.

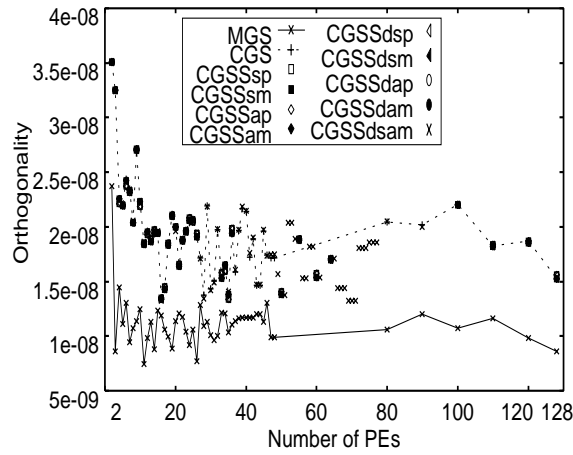
On the other hand, we found that “Dividing the computation” gives us higher accuracy than not dividing case from the numerical experiments. From this result, one can say that parallel GS methods have “double benefit” to sequential ones, since the parallel GS method can execute the orthogonalizations in short time and less memory size.

However, the CGSS methods cause a deficient orthogonality in a case of our experiments. From this, conclusion is that it is very important how to determine the process for data after sorting. It is expected that by analyzing the dot-producted values, the appropriate calculation order can be determined. Therefore, we have to analyze the relation between the values and the accuracy as a future work. If we can find the relation between the values and the accuracy, an automatical selection method in the CGSS methods is established, and then the CGSS method will be a good method in the viewpoint of accuracy and parallelism.

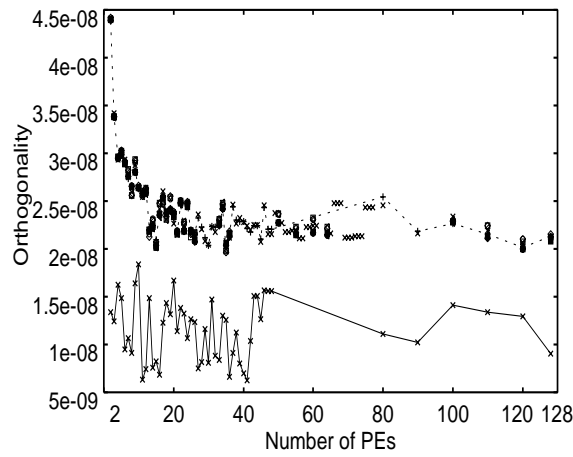
The followings are the summary of some findings in this chapter.

- The CGSS method may be a useful method. If so, a hardware mechanism is in need to perform sorting process. This mechanism gives not only high speed processing, but also high accuracy.
- The CTD distribution can improve load imbalance in the orthogonalization process. The improvement is achieved by triangularly formed process. There are many triangular formed processes in the numerical processing, so we have to implement the CTD distribution strategy on auto-parallelizing compilers, such as HPF or FortranD.

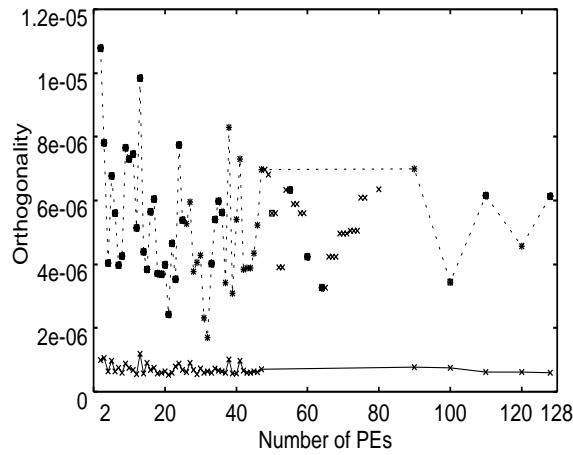
- “Dividing the computation” gives us “double benefit” in the viewpoint of speed and accuracy. Therefore, the parallel GS methods are superior to sequential ones in the parallel execution speed, the orthogonality, and memory usage. This means that the large scale numerical processing should perform on parallel distributed memory computers.



(a) A case of $k = 15$.

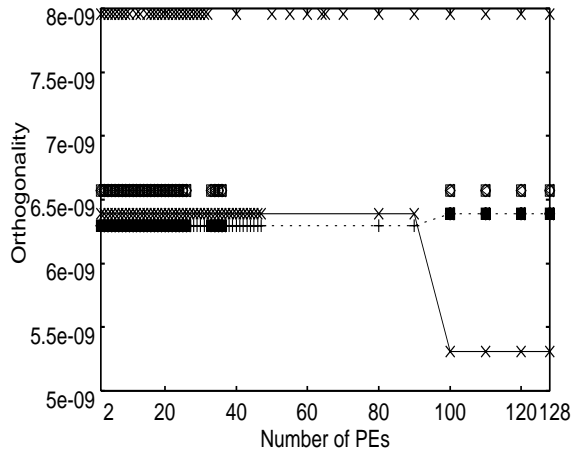


(b) A case of $k = 150$.

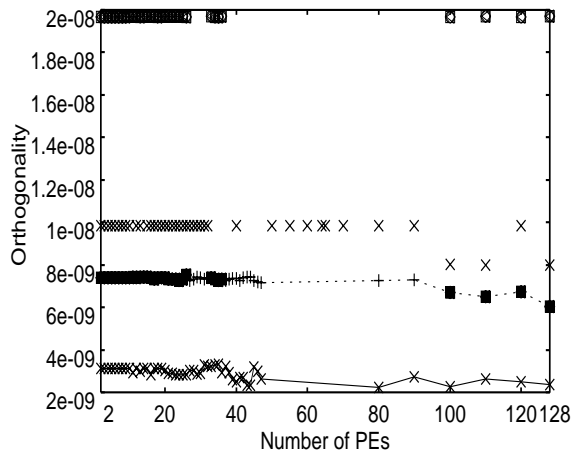


(c) A case of $k = 1500$.

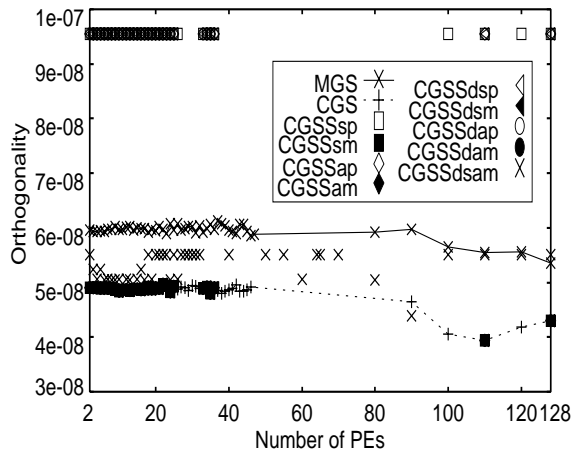
Figure 4.17: Orthogonalities with MGS orthogonalized vectors in re-orthogonalization. (the randomized matrix, $n = 1500$)



(a) A case of $k = 15$.

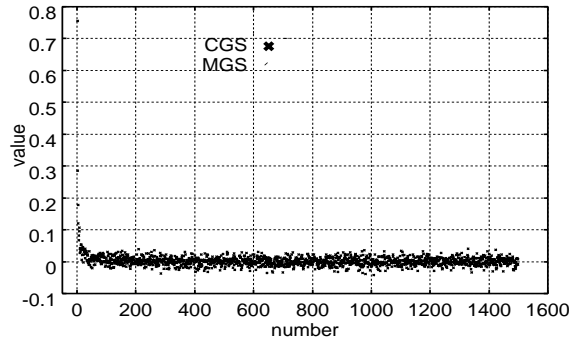


(b) A case of $k = 150$.

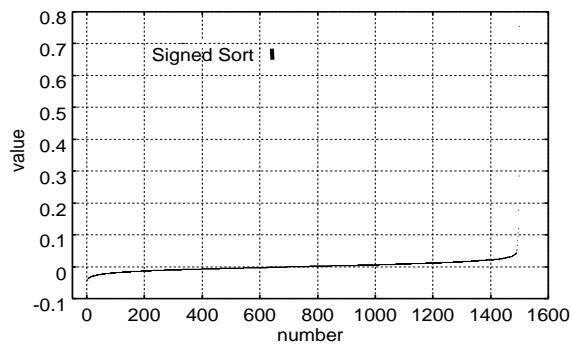


(c) A case of $k = 1500$.

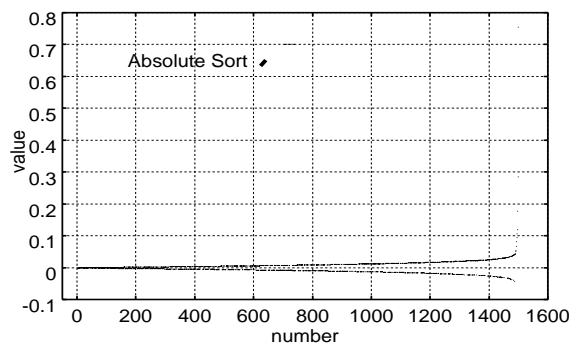
Figure 4.18: Orthogonalities with MGS orthogonalized vectors in re-orthogonalization. (The Läuchli matrix, $n = 1500$, $\epsilon = 1$)



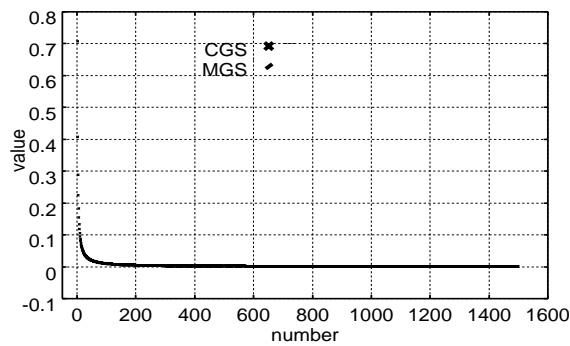
(a-1) A case of randomized matrix. ($k = 1500$) (MGS and CGS)



(a-2) A case of randomized matrix. ($k = 1500$) (CGSSsp, CGSSsm)

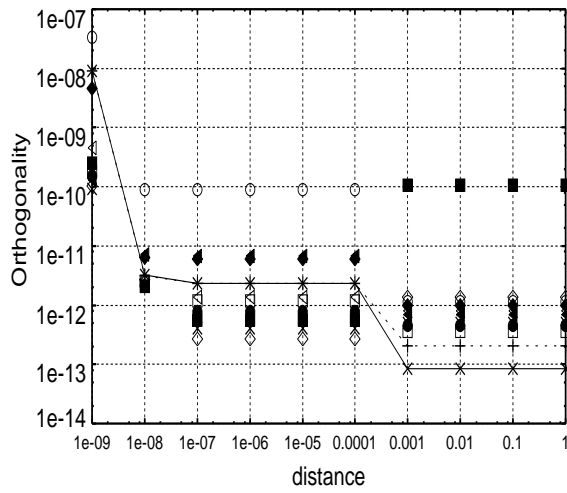


(a-3) A case of randomized matrix. ($k = 1500$) (CGSSap, CGSSam)

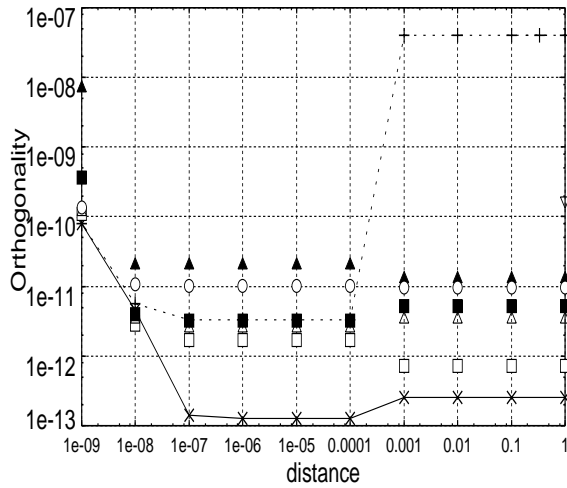


(b) A case of Lauchli matrix. ($k = 1500$) (MGS and CGS)

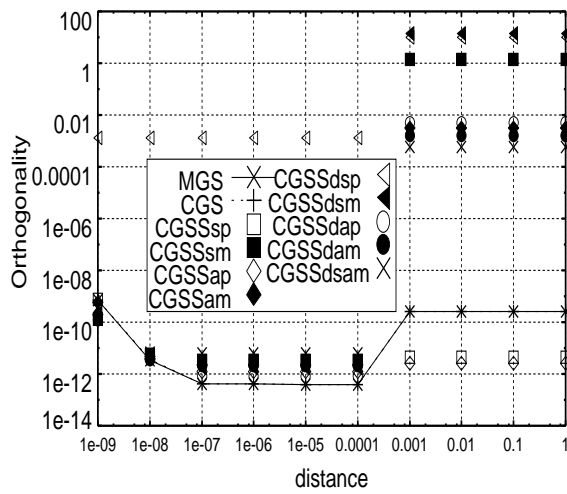
Figure 4.19: Distribution of values for dot-products in each orthogonalization methods.
($p = 16$)



(a) A case of $n = 840$.

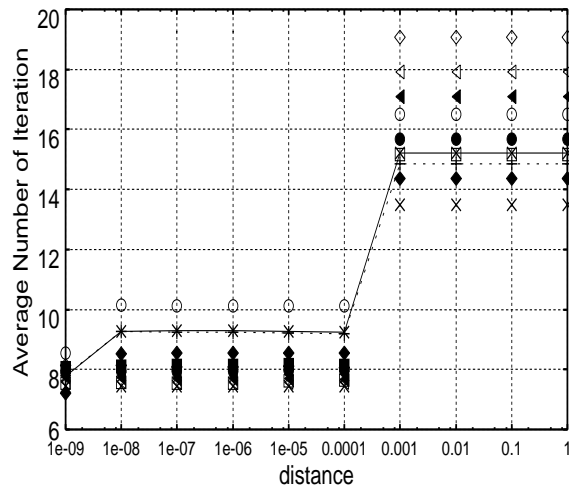


(b) A case of $n = 1000$.

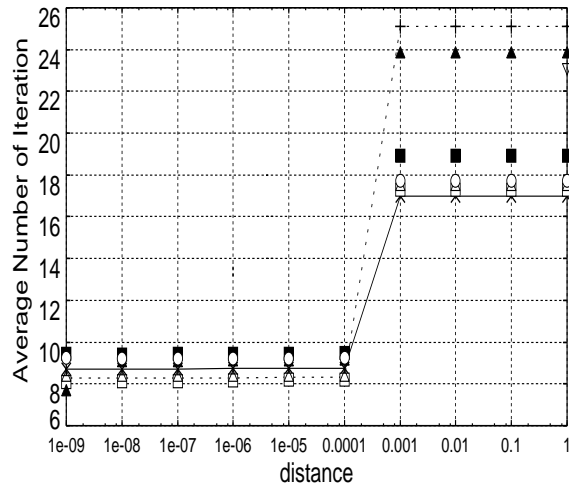


(c) A case of $n = 1260$.

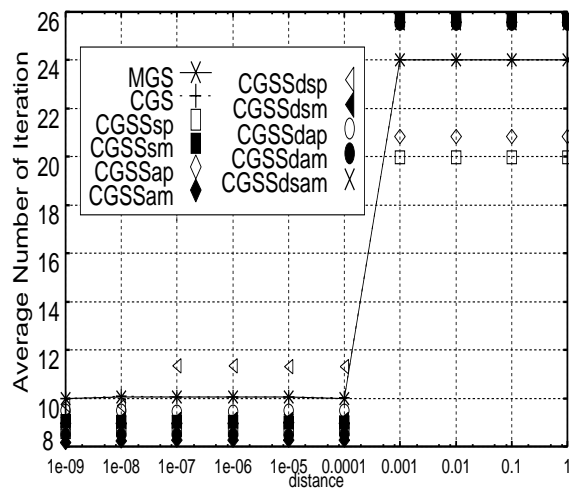
Figure 4.20: Orthogonalities in inverse iteration using each orthogonalization method.



(a) A case of $n = 840$.



(b) A case of $n = 1000$.



(c) A case of $n = 1260$.

Figure 4.21: Average number (per one eigenvector) of iteration until converge.

Part III

Application of Parallel Dense Eigensolvers

Part

Chapter 5

Optimization Feature Before Execution

5.1 Introduction

Tuning computational kernels is a time-consuming work. Several techniques still have to be used to attain high performance. To avoid the tuning work, many linear algebra programs are constructed by using vendor-tuned BLAS (Basic Linear Algebra Subprograms) routines. The BLAS routines give us high efficiency if the BLAS routines were implemented optimally. However, if the BLAS routines were implemented with low efficiency, the performance will be poor. Solution for such implementation problem for BLAS is to use auto-tuning software for BLAS, such as PHiPAC [92] or ATLAS [93]. Other solution is High-Level Library [94, 95], which can automatically select the optimal library based on a statistical modeling. These software packages are called *auto-tuning software for general usage*.

On the other hand, using automatically tuning software which does not or can not use BLAS is hard. Accordingly, every piece of software that can be tuned automatically has a special auto-tuning facility. For example, FFTW [96] for the discrete Fourier transformation, and the auto-tuning libraries [97] in run-time for sparse linear equation solvers. These software packages are called *auto-tuning software for dedicated usage*.

This chapter is for the development of such auto-tuning software for dedicated usage. The reasons are as the following:

1. Presently, auto-tuning software for parallel processing is not available.
2. We believe that an auto-tuning facility should be contained in each package.
3. To obtain high performance, global optimizations are needed.

As for the reason 2, if the auto-tuning facility is separated from the package, users will be in trouble to attain high performance because they have to install auto-tuning software

into their environments separately. In addition, the time needed for auto-tuning may be enormous because it may tune even non-relevant subroutines (consider the tuning time of all BLAS subroutines.) As for the reason 3, the global optimizations of program are needed to obtain much higher performance in general. The global optimizations, therefore, are focused on this chapter rather than local optimizations of BLAS by using auto-tuning software for general usage.

Hence, routines should contain such auto-tuning facility. The auto-tuning facility is called “optimization feature” in this chapter. A concept of “optimization feature before execution” is proposed too — dense linear algebra routines should perform the optimization feature before calling them. Figure 5.1 shows the concept of optimization feature before execution. To show the effectiveness of the concept, an eigensolver is taken as a case study.

This chapter is organized as follows. Section 5.2 is the explanation and definition of optimization feature before execution. Section 5.3 describes our parallel dense eigensolver with the optimization feature before execution. Section 5.4 is the parameters of auto-tuning, and how to search for the optimal parameters. Section 5.5 shows the results of the auto-tuned parameters and execution time of our routines using the optimization feature on the HITACHI SR2201 and HITACHI SR8000. The result of the SR2201 includes a comparison with a ScaLAPACK routine. Finally, Section 5.6 is the conclusion of this chapter.

5.2 Optimization feature before execution

Modern microprocessors can achieve high performance on linear algebra kernels. To achieve the high performance, we have to do extensive machine-specific software tuning. For example, programs in native machine languages must be modified to achieve high performance. This is a time consuming work. To solve this tuning problem, we propose *optimization feature before execution*.

Auto-tuning software in numerical computations is classified as follows:

- (i) Optimization feature at run-time.
- (ii) Optimization feature before execution.

In the optimization feature at run-time, the target routines or libraries are optimized at run-time. The optimization feature at run-time can be applied to many sparse solvers or iterative solvers for linear equations, since the nature of them depends on the characteristics of problems. For example, the performance of sparse solvers greatly depends on

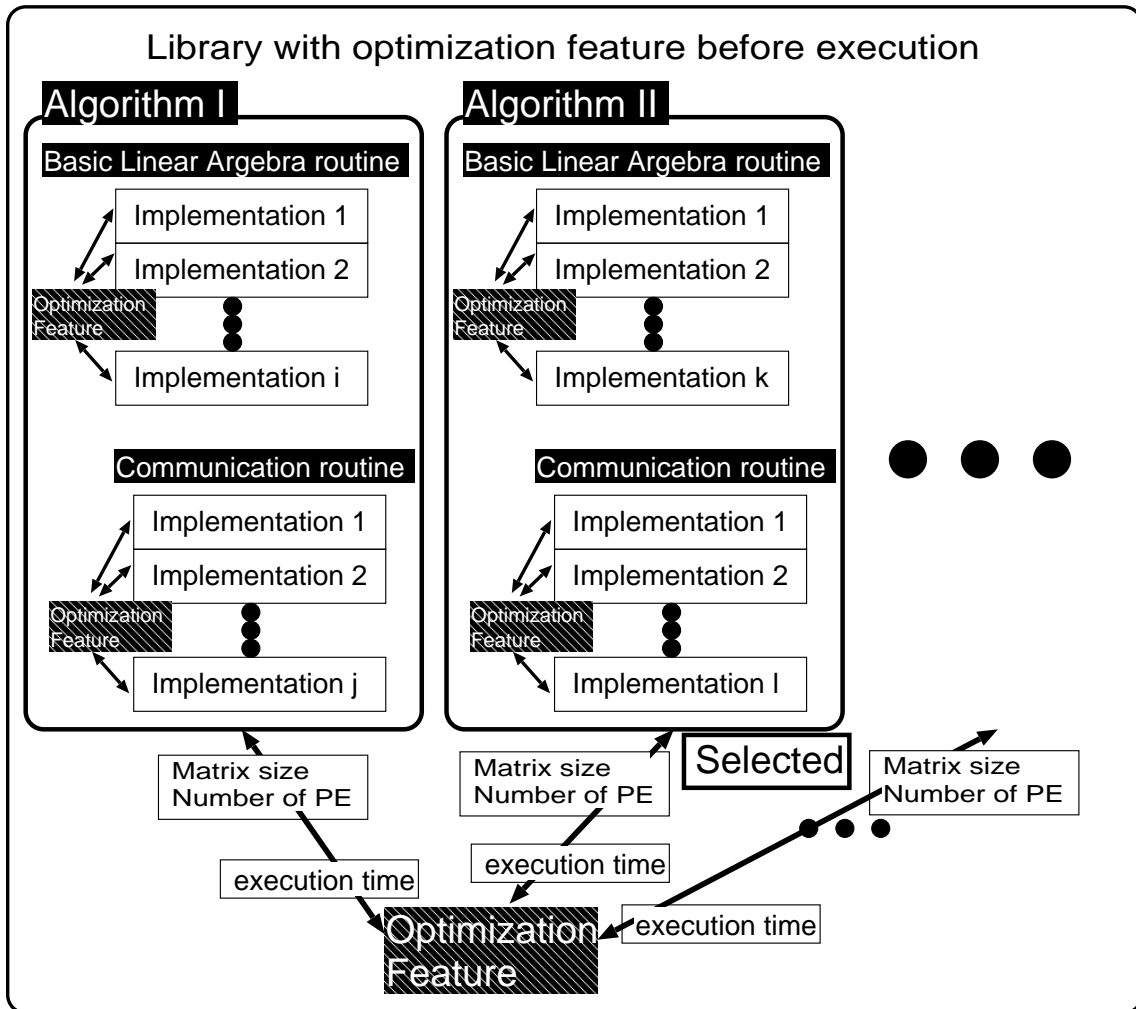


Figure 5.1: The concept of optimization feature before execution.

the location of non-zero elements of matrix. The performance of iterative solvers for linear equation solvers depends on the numerical characteristics of the matrix. For the application, H.Kuroda and Y.Kanada proposed an optimization feature at run-time for GMRES method in linear iterative solvers [97]. The optimization at run-time can not be performed by hands, since one can not select the appropriate implementations at run-time.

On the other hand, the optimization feature before execution is performed before calling the target routines or libraries. The optimization before execution can be applied to dense solvers, since the nature of them does not depend on the characteristics of problems in many cases. For example, the performance of dense direct solver, e.g. LU factorization, does not depend on the the numerical characteristics of the matrix, since computations of LU factorization are mainly determined by the matrix dimensions. The

optimization before execution can be performed by hands. However, the work is a time-consuming. In addition, modifying the codes by hands easily makes several bugs. An auto-tuning facility, therefore, is necessary to accomplish this optimization.

In this section, the (ii) optimization feature before execution is focused on the optimization of dense direct methods (for example, a tridiagonalization routine.)

Table 5.1 summarizes the characteristics of auto-tuning facilities in numerical libraries.

Table 5.1: Summary for the characteristics of auto-tuning facilities in numerical libraries.

Classification	Usage	Optimization	Parallelization	Library name
Run-time	Dedicated	Global	Yes	ILIB_GMRES [97]
Before execution	General	Local	No	PHiPAC [92] ATLAS [93]
		Local	No	FFTW [96]
	Dedicated	Global	Yes	<u>This work</u> (ILIB_DRSSSED)

5.2.1 Definition of optimization feature

At the first step in this section, the optimization feature is defined.

Definition 1 (Library and Routines) *Library is composed of many routines to satisfy certain specifications. To satisfy the specifications of the library, routines also have the specifications.*

Definition 2 (Optimization Feature) *Let a set of all parameters for performance be Ω in a library, and a set of currently using parameters be ω_i in routine i , thus $\omega_i \subseteq \Omega$. Then, the optimization feature is defined by searching ω_i of $\mathbf{min}_i T(\omega_i)$, where $T(\psi)$ means the execution time with the parameter set ψ .*

As for the set of parameters for performance, we define the following.

Definition 3 (Parameters for Performance) *The ω_i which is parameter for some performance of routine i is defined by subroutines which satisfy same output specifications.*

The subroutines which satisfy same output specifications are implemented by using different algorithms and detailed implementation. For example, in linear equation solvers, two of the parameters for performance are defined as LU factorization algorithm and conjugate gradient algorithm. Therefore in this case, ω_i is defined as: $\omega_i \equiv \{ \text{LU factorization, conjugate gradient} \}$. The other parameters in the linear equation solver are defined by blocked algorithms, how to implement loop-unrollings, data storage formats. Therefore, Ω is defined by $\{ \text{blocked algorithm, unblocked algorithm, no unrolling, 2-stride unrolling, ...} \}$. Appropriate parameters are determined by the machine architectures, operating systems used, matrix dimensions, and so on. Thus, searching best parameters is difficult.

There is another problem. It is difficult to enumerate all parameters in Ω because we can not implement routines which satisfy all functions of the parameters. Then the definition is made as the following.

Definition 4 (Practical Parameters for the Performance) *The practical parameters for the performance of each libraries are defined as finite routines in each libraries. Let the set of the practical parameters be Ω' , and the subset of Ω' be ω'_i . Then $\Omega' \subseteq \Omega$, and $\min_i T(\omega_i) \leq \min_i T(\omega'_i)$.*

For the selection for the practical parameters, the selection for parameters should be done by developers of each numerical library. Users who use the library can select the parameters when they desire it.

Finally, the optimization feature before execution in the routine i is defined in the following.

Definition 5 (Optimization Feature Before Execution) *The feature which can optimize $\min_i T(\omega'_i)$, $\omega'_i \subseteq \Omega'$ before the routine is called optimization feature before execution in the routine i .*

Note that the optimization feature before execution can be applied to dense linear algebra computations because several dense linear computations require direct methods to dense matrices. The nature of direct methods does not depend on the characteristics of the problem. Thus the computations of routines are determined before calling the target routines.

5.2.2 Relation between algorithms and implementations

Next is the relation between algorithms and implementations.

(Relation Between Algorithms and Implementations) *In the viewpoint of optimization feature, even if the same algorithm in mathematical or methodological meanings is used in a routine, we have to regard that the routines are implemented with different algorithm if the optimized routines use different implementations.*

For example of the direct method of linear equation solver, the LU factorization has many variations for its implementations, such as blocked LU factorization. It is called same algorithm of “LU factorization”, however, their performance are greatly different. This means that we have to regard these variations as different algorithm in the viewpoint of performance. Thus, we can conclude that the routine which is optimized on an architecture by using the optimization feature is regarded as using a different algorithm against the other optimized routines in another architecture.

5.2.3 Relation between order notation and optimization feature

(Relation Between Order Notation and Optimization Feature) *We can not know statical computational order complexity for optimized routines with an optimization feature.*

The optimized routines are composed of many different ordered routines. Therefore, the order of optimized routine is changed when the characteristics of the input is changed. Figure 5.2 shows this change.

For example, when an $O(n^2)$ routine is selected in $0 < n < 100$, and an $O(n \log n)$ routine is selected in $100 \leq n < 1000$, the computational order of optimized routine can not be estimated by using only one big-O notation for all n region of $0 < n < 1000$. Generally, the regions for input can not be known until we finish optimization, and it is more difficult to know its real order. This shows that optimization feature can change computational orders dynamically.

Of course, the big-O notation has a limitation for the finite definition of n , since the big-O notation is defined as “averaged order” for $n \rightarrow \infty$. However as showed in the previous examples, analyzing the computational order of routines in finite n is important in many applications. To estimate optimized routines, we have to expand the big-O notation. As for the simple solution for this problem is to use the optimization feature before execution which is proposed in this section.

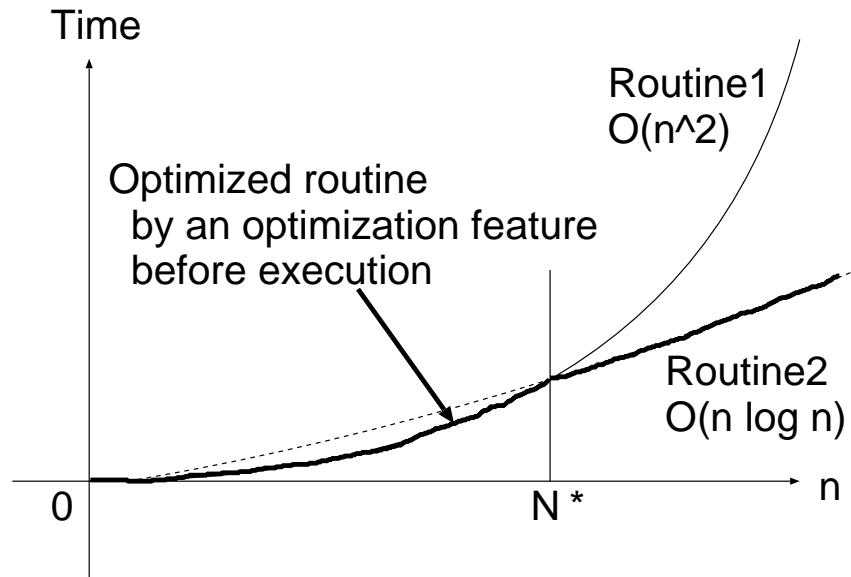


Figure 5.2: Changing complexity order in an optimized routine.

5.2.4 Aims of optimization feature

This is to explain why the optimization feature is needed. The aims are the following:

- (i) to reduce time for tuning, and
- (ii) to achieve high performance.

As for (i), the tuning work is a time-consuming work. Therefore we want to eliminate this work. From this point, the optimization feature will be one of the good choices.

As for (ii), we can not achieve high performance by using only one set of parameters for performance. Since the performance depends on the computer architectures, operating systems and the sizes of the problem, we can not specify the optimal parameters in practical. Solution to this problem is to use the optimization feature because the appropriate parameters for performance can be set dynamically. This idea is a new concept in a sense, since all conventional implementations try to find the only one appropriate set of parameters statically.

5.3 Dense symmetric eigensolver with an optimization feature

5.3.1 Entire process

Our eigensolver can perform the following eigendecomposition:

$$A = X\Lambda X^{-1}, \quad (5.1)$$

where $A \in \mathfrak{R}^{n \times n}$ is a symmetric dense matrix, $\Lambda \in \mathfrak{R}^{n \times n}$ is a diagonal matrix which contains eigenvalues $\lambda_i \in \mathfrak{R}$, ($i = 1, 2, \dots, n$) as the i -th diagonal elements, and $X \in \mathfrak{R}^{n \times n}$ is a matrix which contains eigenvectors $x_i \in \mathfrak{R}^n$ as the i -th row vectors, where n is problem size. In our eigensolver, the decomposition (5.1) is performed by using a well-known method, the Householder-bisection method. To perform the Householder-bisection method, the following four processes are needed.

1. Tridiagonalization by the Householder transformation: $T = QAQ$.
2. Eigenvalues of the tridiagonal matrix T are calculated by using the bisection method.
3. By using the inverse-iteration method, eigenvectors of the tridiagonal matrix T are calculated.
(The processes 2 and 3 yield the eigendecomposition $T = Y\Lambda Y^{-1}$.)
4. Reconstructing eigenvalues for the matrix A : $X = QY$.

Concerning the above four processes, the processes 1 and 4 can affect the whole performance if orthogonalization is not needed in process 3. Process 4 depends on the data distribution of the matrices Q and Y [10]. For this reason, determining the optimal parallelization of process 4 is hard, and hence, the parallelization has not been treated in this chapter.

If the calculated eigenvectors have less orthogonal accuracy, performing orthogonalization for the calculated eigenvectors is one of the choices to improve the accuracy. Therefore, an orthogonalization feature in our eigensolver is provided. The modified Gram-Schmidt (MGS) method is used in the orthogonalization.

Figure 5.3 shows the overall of optimization feature for our eigensolver.

5.3.2 Householder tridiagonalization

Consider the following transformation: $A^{(1)} \equiv A$ to tridiagonal form $A^{(n-2)}$, where $A^{(k)}$ is defined as the k -th iteration of the matrix A . This transformation is denoted by $H^{(k)}(x) = H^{(k)}(A_{k:n,k}^{(k)})$, where $A_{k:n,k}^{(k)}$ is a row vector of A which is constructed by the k -th row and

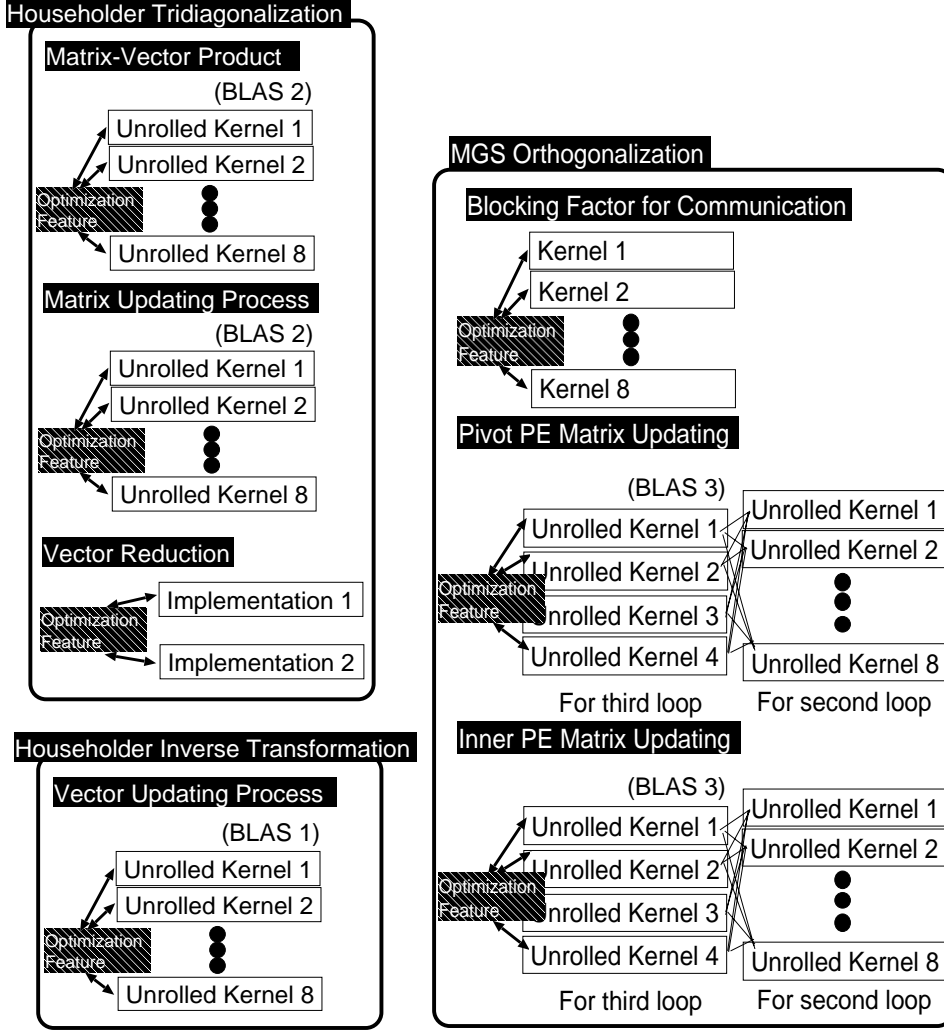


Figure 5.3: The overall of optimization feature for our eigensolver.

from the k -th to the n -th columns in the k -th iteration. By substituting $H^{(k)} = I - \alpha uu^T$ for $H^{(k)}(x)$ in the $k + 1$ -th iteration, the following equations are derived:

$$\begin{aligned}
A^{(k+1)} &= H^{(k)} A^{(k)} H^{(k)} \\
&= A^{(k)} - \alpha A^{(k)} uu^T - \alpha uu^T A^{(k)} + \alpha^2 uu^T A^{(k)} uu^T \\
&= A^{(k)} - xu^T - uy^T + \alpha uu^T xu^T \\
&= A^{(k)} - uy^T + u\mu u^T - xu^T \\
&= A^{(k)} - u(y^T - \mu u^T) - xu^T,
\end{aligned} \tag{5.2}$$

where

$$x = \alpha A^{(k)} u, \quad y^T = \alpha u^T A^{(k)}, \quad \mu = \alpha u^T x. \tag{5.3}$$

Here $\alpha, \mu \in \Re$, and $u, x, y \in \Re^n$. As matrix A is symmetric, $x = y^T$, and we obtain the following formula:

$$A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T. \quad (5.4)$$

Note that to execute the k -th iteration, the column vector $A_{k:n,k}$ from the partial matrix $A_{k:n,k:n}$ is needed.

5.3.2.1 Parallel implementation of the Householder tridiagonalization

Let p be the number of processor elements (PEs). The objective matrix A is distributed by $r \times q$ 2-D grid distribution, called grid-wise distribution (Cyclic, Cyclic), where $r \times q = p$. Block-cyclic distribution is not supported because the block-cyclic distribution causes poor load balance when n/p is small.

The parallel tridiagonalization and Hessenberg reduction routines based on the Householder transformation have already been developed by T.Katagiri and Y.Kanada [13]. Figure 5.4 shows our parallel tridiagonalization algorithm. The routine of Figure 5.4 reduces communication and broadcast time for vector reduction to a ratio of $1/\sqrt{p}$. The same idea appears in [11, 12, 10]. Symmetry of the matrix A was not used in the algorithm of Figure 5.4, and hence, the algorithm has the computational complexity of $8n^3/3$, while the algorithm using symmetry has $4n^3/3$. The algorithm based on the symmetry causes complex data accesses, and the complex data accesses prevent easy parallel implementation.

Figure 5.4 gives a conclusion that implementations of the following three operations affect the total performance.

1. The global summations of the lines ⟨7⟩, ⟨16⟩, and ⟨24⟩ in Figure 5.4.
2. The matrix-vector product of the lines ⟨13⟩–⟨15⟩ in Figure 5.4 .
3. The process to update the matrix A of the lines ⟨25⟩–⟨29⟩ in Figure 5.4.

These three operations are the basic operations for parallel tridiagonalization, and the system will tune the three basic operations automatically in our auto-tuning process.

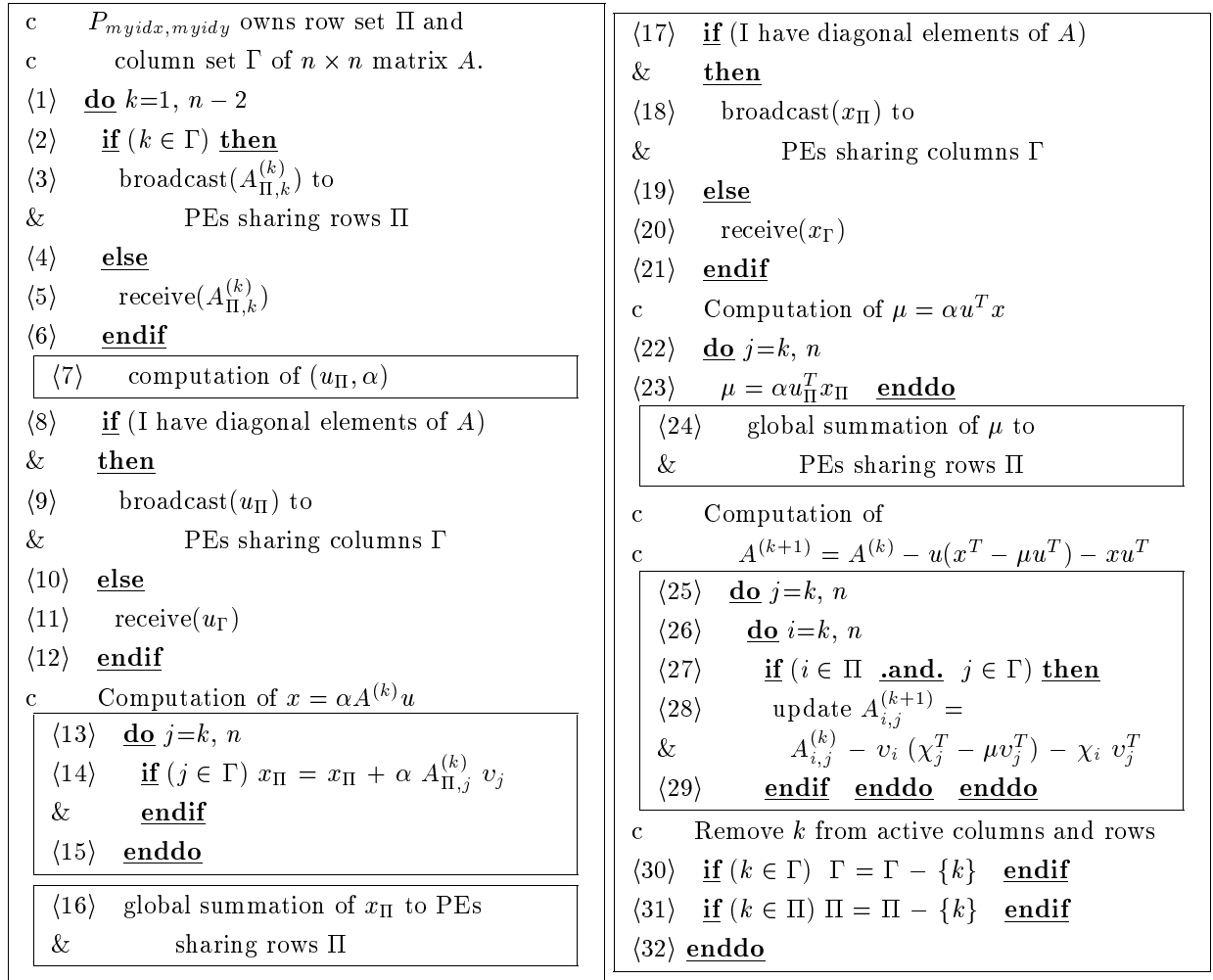


Figure 5.4: Parallel algorithm for the tridiagonalization (the (Cyclic, Cyclic) grid-wise distribution). The squared processes mean the subject of auto-tuning.

5.3.3 Householder inverse transformation routine

The Householder inverse transformation routine transforms eigenvectors \hat{w}_i , ($i = 1, 2, \dots, n$) for a tridiagonal matrix T to eigenvectors w_i , ($i = 1, 2, \dots, n$) of the original symmetric matrix A . The process is expressed as the following:

$$\begin{aligned}
w_i &= H^{(1)} H^{(2)} \dots H^{(n-2)} \hat{w}_i \\
&= (I - \alpha_1 u_1 u_1^T) (I - \alpha_2 u_2 u_2^T) \dots (I - \alpha_{n-2} u_{n-2} u_{n-2}^T) \hat{w}_i, \quad (i = 1, 2, \dots, n), \quad (5.5)
\end{aligned}$$

where the scalars α_i and vectors u_i are calculated in the Householder tridiagonalization routine.

To reduce computation complexity, the following formula is used in the computation

of the Householder inverse transformation.

$$\begin{aligned}
\tilde{w}_k &= (I - \alpha_{k-1}u_{k-1}u_{k-1}^T)\tilde{w}_{k-1}, \\
&= \tilde{w}_{k-1} - \alpha_{k-1}u_{k-1}u_{k-1}^T\tilde{w}_{k-1}, \\
&= \tilde{w}_{k-1} - \sigma_{k-1}u_{k-1} \quad (k = 2, 3, \dots, n-2),
\end{aligned} \tag{5.6}$$

where $\tilde{w}_1 = \hat{w}_i$, ($i = 1, 2, \dots, n$), and the scalar σ_{k-1} is calculated by

$$\sigma_{k-1} = \alpha_{k-1}u_{k-1}^T\tilde{w}_{k-1}. \tag{5.7}$$

5.3.3.1 Parallel implementation of the Householder inverse transformation

Note that in order to perform the above process, the calculated pivot vectors u_1, u_2, \dots, u_{n-2} and the scalars $\alpha_1, \alpha_2, \dots, \alpha_{n-2}$ in the tridiagonalization routine are required. The vectors and scalars are distributed over all PEs. Therefore, we have to gather these data to calculate all w_i . With this gathering requirement, the method based on the Equation (5.5) is inefficient. Hence, we implemented the following loop-exchanged method:

$$\begin{aligned}
\hat{w}_i^1 &= H^{(n-2)}\hat{w}_i, & (i = 1, 2, \dots, n) \\
\hat{w}_i^2 &= H^{(n-3)}\hat{w}_i^1, & (i = 1, 2, \dots, n) \\
&\vdots \\
\hat{w}_i^{n-3} &= H^{(2)}\hat{w}_i^{n-4}, & (i = 1, 2, \dots, n) \\
w_i &= H^{(1)}\hat{w}_i^{n-3}, & (i = 1, 2, \dots, n).
\end{aligned} \tag{5.8}$$

To calculate all eigenvectors, each row in Equation (5.8) is clearly parallel in the calculation. The routine was implemented based on Equation (5.8). The gathering of the pivot vectors u_k was implemented as the following:

- (1) The PEs labeled $myidx = 0$ perform one-to-many broadcasting (multi-casting). By performing this multi-casting, the elements of pivot vector u_k are sent to PEs equivalent to $myidy$, where ($myidx = 0, 1, \dots, r-1$, $myidy = 0, 1, \dots, q-1$)
- (2) Similarly, the PEs labeled $myidx = 1, \dots, r-1$ perform multi-casting.

Figure 5.5 shows the parallel Householder inverse transformation algorithm. Figure 5.5 shows that there is one kernel in this Householder inverse transformation algorithm. Therefore, we can say that the following kernel is the basic operation in the Householder inverse transformation algorithm:

1. The Householder inverse transformation kernel : lines ⟨3⟩ – ⟨6⟩ in Figure 5.5.

```

c  $P_{myid}$  owns the number of first row  $kstart$ ,
c   and the number of last row  $kend$  for each PEs.
⟨1⟩  do  $i=1, n$ 
⟨2⟩    Gather the vector  $u_i$  and scalar  $\alpha_i$  by using row-wise multi-casting.
      ⟨3⟩    Computation of  $\sigma_i$ .
      ⟨4⟩    do  $k = kstart, kend$ 
      ⟨5⟩       $w_k = w_k - \sigma_i u_i$ 
      ⟨6⟩    enddo
⟨7⟩  enddo

```

Figure 5.5: Parallel Householder inverse transformation. The row-wise data distribution (*, Block) is used in the data distribution of eigenvector matrix W . The squared processes mean the subject of auto-tuning.

5.3.4 MGS orthogonalization routine

5.3.4.1 Blocked MGS orthogonalization algorithm

Parallel blocked MGS orthogonalization algorithm is used in our MGS orthogonalization routine. The blocked MGS algorithm can reduce communication time with comparison to normal MGS algorithm. In addition, the kernel of blocked MGS algorithm can replace a two-nested loop (level 2 BLAS) with a three-nested loop (level 3 BLAS). From this reason, the efficiency of the kernel (level 3 BLAS) can be dramatically improved with comparison to normal MGS kernel (level 2 BLAS).

5.3.4.2 Parallel implementation of blocked MGS orthogonalization algorithm

In Figure 5.6 which shows the parallel MGS blocked algorithm, it is implied that there are two computational kernels in this MGS algorithm. Note that the kernels form three nested loops because the notation of w_j stands for a vector. For this reason, the following two kernels are the basic operations in the blocked MGS algorithm:

1. The pivot PE kernel : lines ⟨3⟩ – ⟨10⟩ in Figure 5.6.
2. The inner PE kernel : lines ⟨15⟩ – ⟨20⟩ in Figure 5.6.

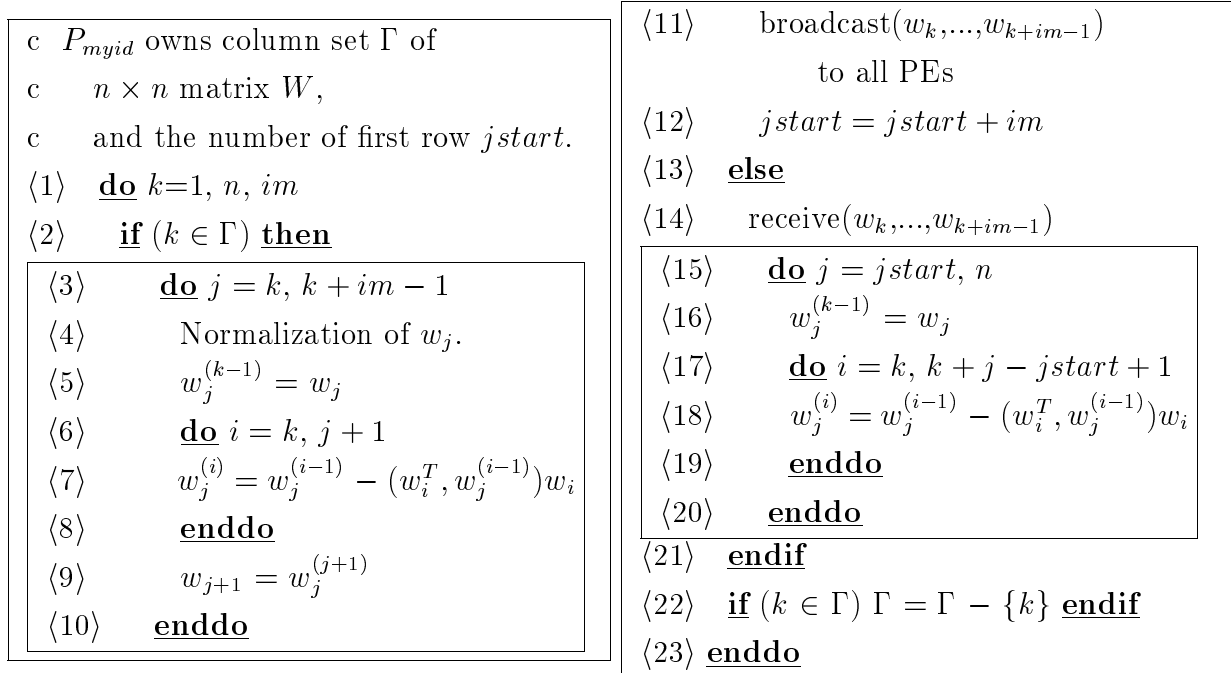


Figure 5.6: Parallel blocked algorithm of MGS orthogonalization. The data distribution is the $(*, \text{Block})$ column-wise distribution. The squared processes in Figure mean the subject of auto-tuning.

5.4 Method for searching parameters

This section describes the method of tuning the basic operations we have mentioned. Hereafter, MPI (Message Passing Interface) is used as the communication library.

5.4.1 Householder tridiagonalization routine

5.4.1.1 Parameter for the global summations

To perform the global summations, the following two implementations were selected.

1. A routine based on the binary tree-structured communication, or
2. The MPI_ALLREDUCE function on MPI.

Which implementation has the higher performance depends on the implementation of MPI functions. Hence, measuring their real performance is necessary to select the best implementation. For that reason, our auto-tuning routine has a parameter for the above two implementations.

5.4.1.2 Parameter for the matrix-vector product

To perform the parallel matrix-vector product ($x = \alpha A^{(k)}u$) at high performance, the size of the stride for loop unrolling must be selected. The size of the stride depends on the machine architectures, operating systems, and compilers we use. Therefore, selecting the optimal number of stride without measuring its real execution time is hard.

For example, a three-stride unrolled routine on the matrix-vector product are shown, where the value of `ilocal_length_x` is dividable by 3 to simplify the explanation.

```
m = ilocal_length_x/3
j = 1
do k=1, m
  dt1 = 0.0d0
  dt2 = 0.0d0
  dt3 = 0.0d0
  do i=1, ilocal_length_y
    du_y = u_y(i)
    ix = init_x+i
    iy = init_y+j
    dt1 = dt1 + A(ix, iy ) * du_y
    dt2 = dt2 + A(ix, iy+1) * du_y
    dt3 = dt3 + A(ix, iy+2) * du_y
  enddo
  x_k(j ) = dt1 * a1
  x_k(j+1) = dt2 * a1
  x_k(j+2) = dt3 * a1
  j = j + 3
enddo
```

This example shows the case of the loop unrolling for the outer-loop `k` only. We can unroll the inner-loop `i` or both of the loops `k` and `i`. Current target machines are vector architecture machines. Then, we only unrolled the outer-loop, since unrolling the inner-loop shortens the loop length which is not good for the vector architecture machines. For the auto-tuning parameter, size of the stride is taken.

5.4.1.3 Parameter for the process to update

As for the matrix-vector product, it is necessary to set the size of the stride for unrolling in the process to update ($A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T$). For example, a two-stride unrolled routine on the process to update is shown, where the value of `ilocal_length_x` is also dividable by 2 to simplify the explanation.

```

m = ilocal_length_x/2
do k=1, m
  j = 2*(k-1)+1
  dtu1 = u_x(j )
  dtu2 = u_x(j+1)
  dtr1 = mu * dtu1 - x_k(j )
  dtr2 = mu * dtu2 - x_k(j+1)
  do i=1, ilocal_length_y
    du_y = u_y(i)
    dx_k = x_k(i)
    ix = init_x+i
    iy = init_y+j
    A(ix, iy ) = A(ix, iy ) + du_y * dtr1 - dx_k * dtu1
    A(ix, iy+1) = A(ix, iy+1) + du_y * dtr2 - dx_k * dtu2
  enddo
enddo

```

For the same reason as for the matrix-vector product, we only unroll the outer loop `k`. The auto-tuning parameter for the process to update is the number of stride for unrolling.

5.4.1.4 How to search these parameters

Let the parameters for the global summation, the matrix-vector product, and the process to update be denoted as `Comm.Type`, `Mat-Vec`, and `Updating`, respectively. The `Comm.Type` can take on the values `{ Tree, MPI_ALLREDUCE }`, where `Tree` means a routine based on binary tree-structured communication, and the `MPI_ALLREDUCE` means communication by a MPI function. The `Mat-Vec` can have the values `{ None, 2, 3, 4, 5, 6, 8, 16 }`, where the numbers show the size of the stride for unrolling. The `Updating` can be chosen as `{ None, 2, 3, 4, 5, 6, 8, 16 }` like in the `Mat-Vec` case.

Following is the description of how to search for the optimal parameters. First, the following default parameter values are set:

$$\text{Comm.Type} \equiv \text{Tree}, \text{Mat-Vec} \equiv 8, \text{Updating} \equiv 6. \quad (5.9)$$

Secondly, the optimal parameters are searched by using the above initial parameters. Method for varying the parameters is as the following.

1. `Comm.Type=Tree`, `Mat-Vec=8`, and `Updating` is varied as { `None`, `2`, `3`, `4`, `5`, `6`, `8`, `16` }.
2. `Comm.Type=Tree`, `Mat-Vec` is varied as { `None`, `2`, `3`, `4`, `5`, `6`, `8`, `16` }, and `Updating`={ the selected value from the process 1 }
3. `Comm.Type` is varied as { `Tree`, `MPI_ALLREDUCE` }, `Mat-Vec`={ the selected value from the process 2 }, and `Updating`={ the selected value from the process 1 }.

This method can not find optimal parameters if there is a dependency among the three parameters. However, the basic operations mentioned are separated physically (see Figure 5.4), hence, there is no dependency in the three parameters. Therefore, we are confident that our method can find almost the optimal set of parameters.

As for the problem sizes, $n = 100$ is specified as the initial values. The problem size is increased by using the stride of 100 while problem size n is under 1000, the stride of 1000 while $1000 \leq n < 10000$, and the stride of 10000 while n is over 10000. This increment is used in each searching process.

5.4.2 Householder inverse transformation routine

5.4.2.1 Parameter for the Householder inverse transformation kernels

To perform the Householder inverse transformation at high performance, we must select the number of stride for loop unrolling. Same as the tridiagonalization routine, the number of stride depends on the machine architectures, operating systems, and compilers we use. Therefore, we can also conclude that selecting the optimal number of stride without measuring its real execution time is hard.

For example, a two-stride unrolled routine on the Householder inverse transformation routine is shown, where the value of `nend-nstart+1` is dividable by 2 to simplify the explanation.

```

im = (nend-nstart+1)/2
i = nstart
do ii=1, im
  local_i1 = i - nstart + 1
  local_i2 = local_i1 + 1
  dnorm1 = 0.0d0
  dnorm2 = 0.0d0
  do k=j, n

```

```

        dbuf = buf(k)
        dnorm1 = dnorm1 + dbuf * W(k, local_i1)
        dnorm2 = dnorm2 + dbuf * W(k, local_i2)
    enddo
    dalp = ALP(j)
    dnorm1 = dalp * dnorm1
    dnorm2 = dalp * dnorm2
    do k=j, n
        dbuf = buf(k)
        W(k, local_i1) = W(k, local_i1) - dnorm1 * dbuf
        W(k, local_i2) = W(k, local_i2) - dnorm2 * dbuf
    enddo
    i = i + 2
enddo

```

This example shows a case of the loop unrolling for the outer-loop i only. Of course, we can unroll the inner-loop k or both of the loops i and k . Our current target machines are vector architecture machines. Then, we will unroll the outer-loop only because unrolling the inner-loop shortens the loop length which is not good for the vector architecture machines. We have the number of the stride as for the auto-tuning parameter.

5.4.2.2 How to search this parameter

Let the parameter for the kernel be denoted as `HITkernel`. The `HITkernel` can take { `None`, 2, 3, 4, 5, 6, 8, 16 }, where the numbers show the number of stride for unrolling.

To search optimal parameters is quite simple, since we have only one parameter. Therefore, we change all of the parameter, and then chose the optimal parameter by measuring each execution time.

5.4.3 MGS orthogonalization routine

5.4.3.1 Parameter for communication times

From the parallel blocked MGS orthogonalization algorithm in the Figure 5.6, the parameter im for communication times is needed. If the im is a small value, the algorithm causes a lot of communication. On the other hand, if the im is a large value, the algorithm causes large overhead to start the algorithm because the PEs which are not pivot PE have no job until they receive the vectors w_k, \dots, w_{k+im-1} . Thus, the parameter im affects total parallel performance.

5.4.3.2 Parameter for the pivot PE kernel

To perform the pivot PE kernel at high performance, we must select the number of stride for loop unrolling. This kernel forms three-nested loop, all of the three loops can be unrolled. However, the loops except for the most inner loops are unrolled because unrolling the inner-loop shortens the loop length which is not good for the vector architecture machines.

For example, we show a two-stride and three-stride unrolled routine on the pivot PE kernel, where the value of il is dividable by 2, and $il-(ib+jjj-1)$ can be divided by 3 to simplify the explanation.

```

    ibml = il/2
    ib = 1
    do ibib=1, ibml
        i_local1 = ILOC(ig + ib - 1)
        i_local2 = ILOC(ig + ib      )
        dw1 = W(k, i_local1)
        dw2 = W(k, i_local2)
        dtemp1 = 0.0d0
        dtemp2 = 0.0d0
        do k=1, n
            dtemp1 = dtemp1 + dw1*dw1
            dtemp2 = dtemp2 + dw2*dw2
        enddo
        dtemp1 = dsqrt(dtemp1)
        dtemp2 = dsqrt(dtemp2)
        dtemp1 = 1.0d0 / dtemp1
        dtemp2 = 1.0d0 / dtemp2
        do k=1, n
            W(k, i_local1) = dtemp1 * W(k, i_local1)
            W(k, i_local2) = dtemp2 * W(k, i_local2)
        enddo
    do jjj=1, 2
        jml = (il-(ib+jjj-1))/3
        j = ig+ib+ jjj-1
        do jj=1, jml
            j_local1 = ILOC(j      )
            j_local2 = ILOC(j+1)
            j_local3 = ILOC(j+2)
            dtemp1 = 0.0d0
            dtemp2 = 0.0d0
            dtemp3 = 0.0d0
            do k=1, n
```

```

        dq = q(k, ib+jjj-1)
        dtemp1 = dtemp1 + dq * W(k, j_local1)
        dtemp2 = dtemp2 + dq * W(k, j_local2)
        dtemp3 = dtemp3 + dq * W(k, j_local3)
    enddo
do k=1, n
    dq = q(k, ib+jjj-1)
    W(k, j_local1) = W(k, j_local1) - dtemp1 * dq
    W(k, j_local2) = W(k, j_local2) - dtemp2 * dq
    W(k, j_local3) = W(k, j_local3) - dtemp3 * dq
enddo
j = j + 3
enddo
enddo
enddo

```

As for the auto-tuning parameter, we have the two numbers for the stride.

5.4.3.3 Parameter for the inner PE kernel

To perform the inner PE kernels at high performance, we must select the number of stride for loop unrolling. Same as the pivot PE kernel, this kernel also forms three nested loop, so we can unroll the three loops. However, we will unroll the loops except for the most inner loops because of the same reason of the pivot PE kernel case.

For example, this is a two-stride and three-stride unrolled routine on the inner PE kernel, where the value of j_1 is dividable by 2, and i_1 is dividable by 3 to simplify the explanation.

```

jbm1 = j1/2
jb = 1
do jbjb=1, jbm1
    j_local1 = ILOC(jg + jb - 1)
    j_local2 = ILOC(jg + jb    )
    im1 = i1/3
    ib = 1
    do ii=1, im1
        dtemp11 = 0.0d0
        dtemp12 = 0.0d0
        dtemp13 = 0.0d0
        dtemp21 = 0.0d0
        dtemp22 = 0.0d0
        dtemp23 = 0.0d0
    enddo
enddo

```

```

do k=1, n
  dw1 = W(k, j_local1)
  dw2 = W(k, j_local2)
  dq1 = q(k, ib )
  dq2 = q(k, ib+1 )
  dq3 = q(k, ib+2 )
  dtemp11 = dtemp11 + dq1 * dw1
  dtemp12 = dtemp12 + dq2 * dw1
  dtemp13 = dtemp13 + dq3 * dw1
  dtemp21 = dtemp21 + dq1 * dw2
  dtemp22 = dtemp22 + dq2 * dw2
  dtemp23 = dtemp23 + dq3 * dw2
enddo
do k=1, n
  dq1 = q(k, ib )
  dq2 = q(k, ib+1 )
  dq3 = q(k, ib+2 )
  W(k, j_local1) = W(k, j_local1) - dtemp11 * dq1 - dtemp12 * dq2
                                - dtemp13 * dq3
  W(k, j_local2) = W(k, j_local2) - dtemp21 * dq1 - dtemp22 * dq2
                                - dtemp23 * dq3

enddo
ib = ib + 3
enddo
jb = jb + 2
enddo

```

The auto-tuning parameters have two numbers of the stride.

5.4.3.4 How to search these parameters

Let the parameter *im* for the communication be denoted as **MGSBL**. Let the parameters for unrollings of the pivot PE kernel be denoted as **MGSPib** and **MGSPj**, respectively. Let the parameters for unrollings of the inner PE kernels be denoted as **MGSijb** and **MGSiib**, respectively.

The **MGSBL** can take $\{ 1, 2, 3, 4, 5, 6, 8, 16 \}$, where the numbers show the number of blocking factor for communication. The **MGSPib** can take $\{ \text{None}, 2, 3, 4 \}$, and the **MGSPj** can take $\{ \text{None}, 2, 3, 4, 5, 6, 8, 16 \}$, where the numbers show the strides of the loop unrollings. Same as these parameters, the **MGSijb** can take $\{ \text{None}, 2, 3, 4 \}$, and the **MGSiib** can take $\{ \text{None}, 2, 3, 4, 5, 6, 8, 16 \}$. The total number of combination of parameters in MGS kernels (for pivot PE and inner PE) is $(4 \times 8) \times 2 = 64$.

Next describes how to search optimal parameters. First, the following default parameter values are set:

$$\text{MGSBL} \equiv 4, \text{MGSPib} \equiv 4, \text{MGSPj} \equiv 8, \text{MGSijb} \equiv 4, \text{MGSiib} \equiv 8. \quad (5.10)$$

Secondly, the optimal parameters are searched by using the above initial parameters. Method for varying the parameters is as the following.

1. MGSBL is varied as $\{ 1, 2, 3, 4, 5, 6, 8, 16 \}$, MGSPib = 4, MGSPj = 8, MGSijb = 4, and MGSiib = 8.
2. MGSBL = $\{ \text{The selected value from the process 1.} \}$, MGSPib is varied as $\{ \text{None}, 2, 3, 4 \}$, MGSPj is varied as $\{ \text{None}, 2, 3, 4, 5, 6, 8, 16 \}$, MGSijb = 4, and MGSiib = 8.
3. MGSBL = $\{ \text{The selected value from the process 1.} \}$, MGSPib = $\{ \text{The selected value from the process 2.} \}$, MGSPj = $\{ \text{The selected value from the process 2.} \}$, MGSijb is varied as $\{ \text{None}, 2, 3, 4 \}$, and MGSiib is varied as $\{ \text{None}, 2, 3, 4, 5, 6, 8, 16 \}$.

Note that this method can not find the optimal parameters, since there is a dependency among the parameter MGSBL and (MGSPib, MGSPj), or (MGSijb, MGSiib). To avoid the auto-tuning time to be huge, all of the combination of the parameters is not searched. Therefore, it is not guaranteed that this method can find optimal parameter sets.

As for the problem sizes, $n = 100$ is specified as the initial values. The problem size is increased by using the stride of 100 while problem size n is under 1000, the stride of 1000 while $1000 \leq n < 10000$, and the stride of 10000 while n is over 10000. This increment is done in each searching process.

5.5 Experimental results

5.5.1 Householder tridiagonalization routine

The optimization feature on the HITACHI SR2201 and HITACHI SR8000 was implemented.

The communication library used for the SR2201 and SR8000 was MPI. Both machines have vector PEs in a sense, i.e. the Pseudo Vector Processor [77]. Therefore, we can regard both machines as vector-parallel machines.

We implemented our tridiagonalization routine by using dedicated subroutines which satisfy functions for the three parameters. For instance, our routine contains a two-stride unrolled matrix-vector product subroutine, or a three-stride unrolled subroutine to update, and so on. By using such subroutines, we can specify the arbitrary parameters. Note that our software does not generate Fortran codes dynamically in this experiments. All auto-tuning was done at measuring time.

5.5.1.1 The results of the SR2201

Results of auto-tuning

Table 5.2 shows parameters auto-tuned on the SR2201. The tuning time depended on the number of PEs, and the CPU elapsed time was about 33 hours at most. The tendency of the tuned parameter of `Comm.Type` were different between 4 and 32 PEs, and the tuned parameters of `Mat-Vec` and `Updating` was different on every problem sizes. From these facts, it is expected that the routine is effective in speeding up.

Comparison to ScaLAPACK

To evaluate execution time of the conventional local optimized routine of tridiagonalization (hereafter TRD), the HITACHI optimized ScaLAPACK version 1.2 [83] was used. Its communication library used was PVM, and PBLAS (Parallel BLAS) which is the computational kernel for ScaLAPACK optimized by HITACHI limited. ScaLAPACK's tridiagonalization routine (hereafter SLP TRD) is implemented by using block-cyclic distribution, a blocked algorithm, and symmetry of the matrix [26]. Since blocked algorithm was used, the size of blocking (BL) can greatly affect the performance of ScaLAPACK. According to [83], if the problem size n is less than 4000, the desirable BL is 60, and if n is over 4000, the desirable BL is 100 on the SR2201. Considering these recommended values, we evaluated the performance of the SLP TRD routines with $BL = \{40, 60, 80, 100, 120\}$ to find which BL gives the best performance. It is shown that $\sqrt{p} \times \sqrt{p}$ is the best layout for the PE grid in [83]. We measured execution time in the PE grid for a large number of PEs. When the number of PEs is small, such as 4, 32, and 64, time was measured in all combinations for the PE grid to find which PE grid gives the best performance.

Table 5.3 shows execution time of the TRD1 (not auto-tuned), TRD2 (auto-tuned), and SLP TRD. Reasonable parameters set of `Comm.Type` \equiv `Tree`, `Mat-Vec` \equiv 8, and `Updating` \equiv 6 are specified in the TRD1 (not auto-tuned). Note that the optimal BL size and PE grids for the SLP TRD are used, and the values are included in Table 5.3.

Table 5.3 shows that 1.6–1.8 times speed-ups were obtained to respect to the TRD1 (not auto-tuned) when problem sizes were large, such as 4000, 8000. As for the SLP TRD

Table 5.2: The auto-tuned parameters on the SR2201. (Householder tridiagonalization routine)

(a) Case of 4 PEs

Size	Comm. Type	Mat-Vec	Updating
100	MPI_ALLREDUCE	6	3
200	Tree	8	4
300	Tree	8	6
400	Tree	5	2
500	Tree	8	5
600	Tree	5	6
700	Tree	8	6
800	Tree	3	3
900	Tree	8	4
1000	Tree	5	5
2000	Tree	5	6
3000	Tree	5	5
4000	Tree	3	3
5000	MPI_ALLREDUCE	5	5
6000	MPI_ALLREDUCE	5	5
7000	MPI_ALLREDUCE	5	5
8000	MPI_ALLREDUCE	3	2
Tuning time		118401 [Sec.]	(32.8 [Hours])

(b) Case of 32 PEs

Size	Comm. Type	Mat-Vec	Updating
100	MPI_ALLREDUCE	6	16
200	MPI_ALLREDUCE	4	5
300	MPI_ALLREDUCE	4	4
400	MPI_ALLREDUCE	6	3
500	MPI_ALLREDUCE	6	4
600	MPI_ALLREDUCE	6	4
700	MPI_ALLREDUCE	8	3
800	MPI_ALLREDUCE	5	3
900	MPI_ALLREDUCE	4	3
1000	MPI_ALLREDUCE	5	3
2000	MPI_ALLREDUCE	5	5
3000	MPI_ALLREDUCE	8	5
4000	MPI_ALLREDUCE	5	5
5000	MPI_ALLREDUCE	8	5
6000	MPI_ALLREDUCE	5	5
7000	MPI_ALLREDUCE	5	5
8000	MPI_ALLREDUCE	3	3
Tuning time		15555 [Sec.]	(4.3 [Hours])

Table 5.3: Execution time on the SR2201. (Householder tridiagonalization routine) Unit is in second.

(a) Case of 4 PEs				
Size	SLP TRD (Grid, BL)	TRD1 (not AT)	TRD2 (AT)	TRD1/TRD2
100	0.02 (1×4, 100)	0.056 (2×2)	0.056 (2×2)	1.00
200	0.48 (1×4, 100)	0.131 (2×2)	0.133 (2×2)	0.98
400	1.73 (1×4, 40)	0.435 (2×2)	0.475 (2×2)	0.91
800	6.01 (1×4, 40)	3.732 (2×2)	2.454 (2×2)	1.5
1000	9.32 (2×2, 40)	3.817 (2×2)	3.785 (2×2)	1.0
2000	41.90 (2×2, 40)	28.165 (2×2)	26.937 (2×2)	1.0
4000	231.10 (2×2, 40)	411.666 (2×2)	242.010 (2×2)	1.7
8000	1422.69 (2×2, 100)	3589.175 (2×2)	1962.512 (2×2)	1.8

(b) Case of 32 PEs				
Size	SLP TRD (Grid, BL)	TRD1 (not AT)	TRD2 (AT)	TRD1/TRD2
100	0.09 (4×8, 100)	0.108 (4×8)	0.106 (4×8)	1.01
200	0.87 (2×16, 100)	0.250 (4×8)	0.240 (4×8)	1.04
400	2.33 (2×16, 60)	0.514 (4×8)	0.516 (4×8)	0.99
800	6.27 (2×16, 60)	1.207 (4×8)	1.228 (4×8)	0.98
1000	8.28 (2×16, 60)	1.654 (4×8)	1.687 (4×8)	0.98
2000	22.18 (4×8, 40)	5.930 (4×8)	5.886 (4×8)	1.00
4000	72.74 (4×8, 40)	32.961 (4×8)	32.124 (4×8)	1.02
8000	313.25 (4×8, 40)	427.267 (4×8)	254.937 (4×8)	1.6

execution time, when problem size is small, the TRD was faster than the SLP TRD. On the other hand, when problem sizes per PE were large, the SLP TRD was faster than the TRD. It is considered that this is explained from the computational complexity of the TRD, since the TRD has twice computational complexity to the SLP TRD.

Figure 5.7 shows the execution time of the TRD1 (not auto-tuned), TRD2 (auto-tuned), and SLP TRD in $n = 2000$ and 8000 cases. Note that the execution time of the SLP TRD in Figure 5.7 was the optimal BL and the PE grid case. Conclusion from

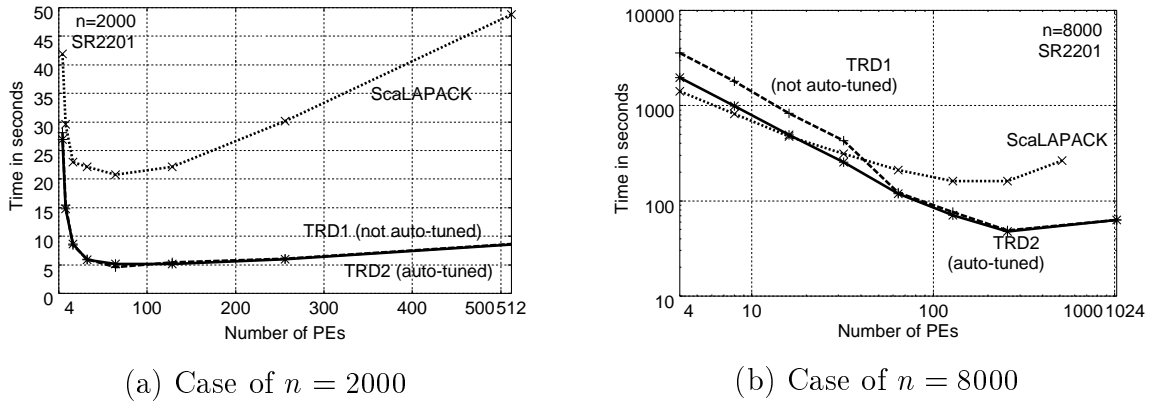


Figure 5.7: Execution time for the SLP TRD and TRD in the tridiagonalization (SR2201).

Figure 5.7 is that when $n = 2000$, the TRD is always faster than the SLP TRD, and the speed-up ratios are about 2–6 times. On the other hand, when $n = 8000$, the execution speed of the TRD was slower than the SLP TRD when the number of PEs was under 64, however, when over 64, the TRDs became faster than the SLP TRD. The effect of auto-tuning was high when the number of PEs was under 64.

Conclusion from the experimental results is that our global optimization methodology is useful compared to the conventional local optimized libraries based on the optimization of BLAS.

The execution time in every auto-tuning process

To evaluate the auto-tuning process in detail, the execution time in each of our auto-tuning process was analyzed. Figure 5.8 shows the time when the problem size was 8000.

From Figure 5.8 (a), the execution time for the specified initial set of parameters (Comm.Type \equiv Tree, Mat-Vec \equiv 8, Updating \equiv 6) was slower than the case of Figure 5.8 (b). Since the 6-stride of the process 1 (Updating) and the 8-stride of the process 2 (Mat-Vec) in Figure 5.8 (a) were not optimal parameters, there was the change of the elapsed time when varying these strides was high. Hence, the conclusion is that the initial parameters were not good for the case of 4 PEs, and this caused high speed-up ratios. On

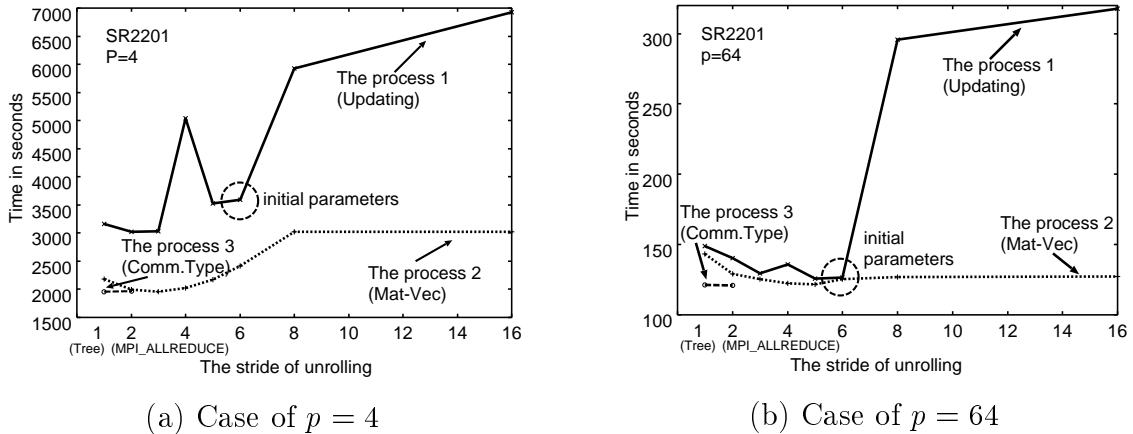


Figure 5.8: The execution time in every auto-tuning process. (SR2201, $n = 8000$)

the other hand, Figure 5.8 (b) shows that the initial set of parameters we specified was almost optimal set of values. For this reason, we did not obtain better speed-ups on 64 PEs than the speed-ups on 4PEs on the SR2201.

5.5.1.2 The results of SR8000

Results of auto-tuning and execution time

Table 5.4 shows auto-tuned parameters on the SR8000. From Table 5.4, the tendency of tuned parameters was found to be different between the inner-node parallel and inter-node parallel environments. From this fact, the cases for the speed-up are also found. Table 5.5 shows execution time of the TRD1 (not auto-tuned) and TRD2 (auto-tuned). From Table 5.5, we obtained about 1.1–1.3 times speed-ups with respect to the TRD1 (not auto-tuned). The effect became stronger when problem size increased. Therefore, the author concludes that our optimization method is also useful on the SR8000.

5.5.2 Householder inverse transformation routine

5.5.2.1 The results of the SR2201

Results of auto-tuning

Table 5.6 shows parameters auto-tuned on the SR2201. Table 5.7 shows execution time on the SR2201, where the notation HIT(not AT) means execution time with $HIT_{kernel}=1$. The results of Table 5.6 indicate that 1.2 – 1.4 times speed-ups were obtained by applying our optimization method.

Table 5.4: The auto-tuned parameters on the SR8000. (Householder tridiagonalization routine)

(a) Case of 1 Node (8 IPs)			
(SR8000, inner-node parallel, sheard memory)			
Size	Comm.Type	Mat-Vec	Updating
100	MPI_ALLREDUCE	None	None
200	MPI_ALLREDUCE	4	None
300	MPI_ALLREDUCE	8	None
400	MPI_ALLREDUCE	4	None
500	MPI_ALLREDUCE	5	None
600	MPI_ALLREDUCE	6	3
700	MPI_ALLREDUCE	6	None
800	MPI_ALLREDUCE	6	3
900	MPI_ALLREDUCE	6	None
1000	MPI_ALLREDUCE	6	3
2000	MPI_ALLREDUCE	6	None
3000	MPI_ALLREDUCE	6	None
4000	MPI_ALLREDUCE	6	None
5000	MPI_ALLREDUCE	4	None
6000	MPI_ALLREDUCE	4	None
7000	MPI_ALLREDUCE	6	None
8000	MPI_ALLREDUCE	6	None
Tuning time		16325 [Sec.]	(4.5 [Hours])
(b) Case of 4 Nodes (32 IPs)			
(SR8000, inter-node parallel, distributed memory)			
Size	Comm.Type	Mat-Vec	Updating
100	Tree	None	2
200	Tree	None	None
300	Tree	None	2
400	Tree	None	None
500	Tree	None	None
600	Tree	None	None
700	Tree	None	None
800	Tree	4	None
900	Tree	4	None
1000	Tree	6	None
2000	MPI_ALLREDUCE	6	4
3000	MPI_ALLREDUCE	6	4
4000	MPI_ALLREDUCE	4	16
5000	MPI_ALLREDUCE	4	16
6000	MPI_ALLREDUCE	4	16
7000	MPI_ALLREDUCE	6	16
8000	MPI_ALLREDUCE	6	16
Tuning time		4443 [Sec.]	(1.2 [Hours])

Table 5.5: Execution time on the SR8000. (Householder tridiagonalization routine) Unit is in second.

(a) Case of 1 Node (8 IPs)
(SR8000, inner-node parallel, sheard memory)

size	TRD1 (not AT)	TRD2 (AT)	TRD1 / TRD2
100	0.024 (2×4)	0.022 (2×4)	1.09
200	0.053 (2×4)	0.049 (2×4)	1.08
400	0.162 (2×4)	0.145 (2×4)	1.11
800	0.678 (2×4)	0.587 (2×4)	1.15
1000	1.155 (2×4)	0.988 (2×4)	1.16
2000	7.098 (2×4)	5.595 (2×4)	1.26
4000	50.451 (2×4)	39.263 (2×4)	1.28
8000	389.297 (2×4)	308.307 (2×4)	1.26

(b) Case of 4 Nodes (32 IPs)
(SR8000, inter-node parallel, distributed memory)

Size	TRD1 (not AT)	TRD2 (AT)	TRD1/TRD2
100	0.038 (2×2)	0.036 (2×2)	1.05
200	0.077 (2×2)	0.072 (2×2)	1.06
400	0.176 (2×2)	0.162 (2×2)	1.08
800	0.490 (2×2)	0.450 (2×2)	1.08
1000	0.714 (2×2)	0.648 (2×2)	1.10
2000	2.806 (2×2)	2.345 (2×2)	1.19
4000	14.957 (2×2)	11.392 (2×2)	1.31
8000	102.369 (2×2)	75.398 (2×2)	1.35

Table 5.6: The auto-tuned parameters on the SR2201. (Householder inverse transformation routine)

(a) Case of 4 PEs		(b) Case of 16 PEs	
Size	HITkernel	Size	HITkernel
100	6	100	6
200	3	200	4
300	5	300	4
400	3	400	3
500	5	500	3
600	6	600	6
700	5	700	8
800	3	800	3
900	5	900	5
1000	6	1000	6
2000	3	2000	3
3000	6	3000	6
4000	3	4000	3
Tuning time	6246 [Sec.] (1.73 [Hour])	Tuning time	3129 [Sec.] (0.86 [Hour])

Table 5.7: Execution time on the SR2201. Unit is in second. (Householder inverse transformation routine)

(a) Case of 4PEs				(b) Case of 16 PEs			
size	HIT1 (not AT)	HIT2 (AT)	HIT1 /HIT2	Size	HIT1 (not AT)	HIT2 (AT)	HIT1 /HIT2
100	0.010	0.007	1.42	100	0.005	0.004	1.25
200	0.067	0.051	1.31	200	0.034	0.026	1.30
400	0.473	0.375	1.26	400	0.234	0.189	1.23
800	3.469	2.837	1.22	800	1.735	1.436	1.20
1000	6.624	5.200	1.27	1000	3.332	2.660	1.25
2000	49.46	39.51	1.25	2000	24.81	19.86	1.24
4000	384.9	312.9	1.23	4000	192.96	158.93	1.21

5.5.3 MGS orthogonalization routine

5.5.3.1 The results of the SR2201

Results of auto-tuning

Table 5.8 shows parameters auto-tuned on the SR2201. Table 5.9 shows execution time on the SR2201, where the notation MGSAO1(not AT) means execution time with the default parameters set of MGSBL=4, MGSPib=4, MGSPj=8, MGSIjb=4, and MGSIib=8.

The results of Table 5.8 indicate that 1.1 – 2.3 times speed-ups were obtained by applying our optimization method.

5.6 Conclusion

The author has implemented and evaluated an eigensolver by using an auto-tuning methodology. Selecting suitable implementations for the global summation, the matrix-vector product, the process to update, and the blocking factor for communication on the parallel eigensolver is the auto-tuning methodology mentioned in this chapter, and the methodology is quite simple. Even though this quite simple methodology was used, the author could obtain about 1.1–2.3 times speed-ups with respect to the routine for which the reasonable parameters in the SR2201 and the SR8000 were specified. From these results, the author concluded that such an auto-tuning methodology is an effective technique.

The auto-tuning methodology is for vector-parallel machines. The auto-tuning methodology for the RISC based parallel machines, such as unrolling the most inner loop, and evaluation on the RISC based parallel machines are parts of the future work.

Table 5.8: The auto-tuned parameters on the SR2201. (MGS orthogonalization routine)

(a) Case of 4 PEs					
Size	MGSBL	MGSPib	MGSPj	MGSIjb	MGSIib
100	6	3	4	2	6
200	6	None	None	2	6
300	6	None	None	2	6
400	6	None	None	2	3
500	6	None	None	2	6
600	6	None	None	2	6
700	6	None	None	2	6
800	6	None	None	2	3
900	6	None	None	2	6
1000	6	None	None	2	6
2000	6	None	6	2	3
3000	6	None	16	2	6
4000	6	None	None	None	3
Tuning time	91407 [Sec.]	(25.3 [Hours])			

(b) Case of 16 PEs					
Size	MGSBL	MGSPib	MGSPj	MGSIjb	MGSIib
100	5	4	8	4	2
200	5	None	None	2	5
300	5	None	None	None	4
400	5	None	None	2	5
500	6	None	None	2	6
600	6	None	None	2	6
700	5	None	None	2	5
800	5	None	None	2	5
900	6	None	None	2	6
1000	5	None	None	2	5
2000	6	None	8	2	3
3000	6	None	6	2	6
4000	6	None	8	2	6
Tuning time	26607 [Sec.]	(7.3 [Hours])			

Table 5.9: Execution time on the SR2201. Unit is in second. (MGS orthogonalization routine)

(a) Case of 4PEs

Size	MGSAO1 (not AT)	MGSAO2 (AT)	MGSAO1 /MGSAO2
100	0.017	0.010	1.70
200	0.103	0.051	2.01
400	0.733	0.389	1.88
800	6.379	3.174	2.00
1000	12.598	5.378	2.34
2000	108.665	48.879	2.22
4000	892.796	420.207	2.12

(b) Case of 16 PEs

Size	MGSAO1 (not AT)	MGSAO2 (AT)	MGSAO1 /MGSAO2
100	0.014	0.012	1.16
200	0.047	0.033	1.42
400	0.294	0.136	2.16
800	1.932	0.859	2.24
1000	3.865	1.668	2.31
2000	32.407	14.310	2.26
4000	260.168	119.003	2.18

Chapter 6

Towards High Performance Auto-tuning Software

6.1 Introduction

Today, many packages of auto-tuning software for the computation of linear algebra have been developed [92, 93, 98, 96, 97, 99, 16]. Such packages of software are very useful because the tuning process is a time-consuming process. However these packages of software have a big problem — the tuning time becomes huge when the better parameters are to be obtained. To solve such tuning time problem, we will discuss how to reduce the tuning time in this chapter.

First of all, what is “high performance” for the auto-tuning software? Below defines the high performance auto-tuning software.

Definition 6 (High performance auto-tuning software) *(1) It can quickly find a parameter set which can execute the program at fast; (2) Optimization time for finding better parameter set is also fast; (3) If much longer optimization time was set, a much better parameter set is found;*

First this chapter proposes some methods to reduce the optimization time and attain high performance for the auto-tuning software.

Next this chapter proposes a new category of basic linear algebra subprograms (BLAS) [55, 56]. By using the tuned parameter sets, the new BLAS can be constructed. This new BLAS can change its implementations according to the dimension of matrix. Possibility of higher performance than that of the conventional BLAS is shown by changing the implementations.

This chapter is as the following. Section 6.2 explains the methods for reducing optimization method, and evaluates the methods by using a developed auto-tuning software.

Section 6.3 is the proposition of new category of basic linear algebra subprograms, called *Intelligent BLAS*. Finally, Section 6.4 is the conclusion of this chapter.

6.2 Methods for reducing optimization time

Increasing the optimization time is a main problem in auto-tuning software. This section discusses several methods to reduce the optimization time.

To reduce the optimization time, the following methods can be used:

- (1) using machine specific parameters,
- (2) using small sized problems,
- (3) using incomplete decompositions, and
- (4) using decreased searches.

6.2.1 Using machine specific parameters

The method (1) shows that machine specific parameters, such as cache related information can be used to reduce optimization time. This method is so called “heuristic” method, and we must analyze how the target kernels use caches. From the analysis, we can reduce combination of parameters for optimization.

ATLAS [93] uses this machine specific parameters to reduce optimization time. For the search heuristic of ATLAS, the theoretically optimal register blocking in terms of maximizing flops/load are near-square cases. Therefore, the ATLAS generator requires square factors, such as m_u , n_u , these M and N loops unrollings are then used to find an initial blocking factor. The initial blocking factor is found by simply using a loop unrollings, and seeing which of the blocking factors appropriate to the detected L1 cache size produce the best result. With this initial blocking factor, which instructions set to use, and a guess as to pipeline length, the search routine loops over all M and N loop unrollings possible with the given number of registers. Once an optimal unrolling has been found, ATLAS again tries all blocking factors, and various latency and K-loop unrolling factors, and chooses the best.

Thus, ATLAS reduces the optimization time by using the size of L1 cache and theoretically analysis of the target computation kernels. However, the found parameter sets may not be the optimal parameter set, since ATLAS uses the heuristic search method.

6.2.2 Using small sized problems

The method (2) shows that the parameter sets of small problem can be used when we estimate execution time for large problem. By using small sized problem, the total optimization time is reduced. However, good parameters may not be obtained, since the problem size is too small (consider a case that whole data in cache.)

6.2.3 Using incomplete decompositions

The method (3) shows that the incomplete decomposition or computations can be used to reduce optimization time. For example, by stopping LU decomposition in the k -th iteration, where $k < n$, and the case of $k = n$ is complete LU decomposition. For the Householder tridiagonalization, the incomplete decomposition of $k < n$ is shown as:

$$Q_k \cdots Q_2 Q_1 A_1 Q_1 Q_2 \cdots Q_k = A_k. \quad (6.1)$$

Clearly, the time for incomplete decomposition is reduced with compared to the complete case, hence the total optimization time is also reduced. For example, the computational complexities of the LU decomposition which is stopped in the k -th iteration, are estimated by the following:

$$\approx \frac{2}{3}n^3 - \frac{2}{3}(n-k)^3 - \frac{1}{3}(n-k)^2 - \frac{1}{3}(n-k). \quad (6.2)$$

The computational complexities of the full LU decomposition is $\approx 2/3n^3$. Hence, the speed-up ratio between the incomplete and full one can be estimated by

$$\approx \frac{\frac{2}{3}n^3 - \frac{2}{3}(n-k)^3}{\frac{2}{3}n^3} = 1 - \frac{(n-k)^3}{n^3}. \quad (6.3)$$

Note that all $O(n^3)$ decompositions can be estimated by using Equation (6.3), since the coefficient of the n^3 term does not affect the speed-up ratio. Therefore in general, if $k = n/2$ then the ratio is $7/8 \approx 0.87$, $k = n/4$ then the ratio is $37/64 \approx 0.57$, and $k = n/8$ then the ratio is $169/512 \approx 0.33$.

Thus, the use of incomplete decompositions is one of the choices in the viewpoint of optimization time. However, in general, determining the suitable k is difficult.

6.2.4 Using decreased searches

The method (4) shows that instead of brute force search, decreased search can be used. This method is classified as “heuristic search” method, or “ad-hoc search” method. Many

basic kernels for linear algebra software are physically separated in general. Therefore many parameters do not have a dependency between them.

If each parameter has the definition area of $O(m)$, and the number of parameters is $O(n)$, then search space is $O(m^n)$ in brute force search. But by using the decreased search, the search space is reduced to $O(mn)$ if each parameter has no dependency.

6.2.5 Evaluation of the methods

The methods for reduction optimization time were evaluated by using ver.0.61 of ILIB_TriRed routine [73]. The ILIB_TriRed routine is an auto-tuning routine for the Householder tridiagonalization for eigenproblems. The basic linear algebra kernels of ILIB_TriRed routine is physically separated, therefore the parameters have no dependency. The optimization method of ILIB_TriRed is used in the method (4) — the decreased search method.

The HITACHI SR2201 is used in this experiment.

6.2.5.1 Method for using small sized problems

Table 6.1 shows the optimization time by the method for using small sized problems. The

Table 6.1: Optimization time on the SR2201 by using small sized problem. Unit is second. Full(8000) is the full optimization time until problem size of 8000. SP(1000) is the optimization time until problem size of 1000. The numbers in parentheses show decreased ratios to the time of Full(8000).

#PEs	Full(8000)	SP(1000)	SP(2000)	SP(4000)
4	118401 (1/1)	333 (1/355)	979 (1/120)	9754 (1/12.1)
8	59262 (1/1)	231 (1/256)	596 (1/99.4)	5142 (1/11.4)
16	28768 (1/1)	167 (1/172)	372 (1/77.3)	2266 (1/12.6)

results of Table 6.1 shows that using small sized problems can reduce optimization time dramatically. We could obtain 11 – 333 times speed-ups to the full optimization case of $n = 8000$.

Table 6.2 shows execution time by using auto-tuned parameters with small sized problems. From Table 6.2, using tuned parameters for large problem sizes does not always attain high performance.

Table 6.2: Execution time on the SR2201 by using auto-tuned parameters with small sized problems. Unit is second. Full(8000) is the execution time by using full optimized parameter sets with problem size of 8000. SP(1000) is the execution time by using the optimized parameter set of problem size of 1000.

(a) The case of PE = 4.

Size	Full(8000)	SP(1000)	SP(2000)	SP(4000)
1000	3.785	—	—	—
2000	26.937	29.852	—	—
4000	242.010	473.132	309.387	—
8000	1962.512	4176.515	2676.937	1959.871

(b) The case of PE = 8.

Size	Full(8000)	SP(1000)	SP(2000)	SP(4000)
1000	2.476	—	—	—
2000	14.838	15.790	—	—
4000	124.205	136.865	158.702	—
8000	989.504	1094.535	1348.025	988.183

(c) The case of PE = 16.

Size	Full(8000)	SP(1000)	SP(2000)	SP(4000)
1000	1.824	—	—	—
2000	8.649	8.836	—	—
4000	56.239	60.907	58.565	—
8000	490.346	704.611	835.198	627.009

These results conclude that using auto-tuned parameters with small sized problems is not suitable method in the viewpoint of performance, since better performance can not be obtained even if more time was spent on the optimization.

6.2.5.2 Method for using incomplete decompositions

Table 6.3 shows optimization time with incomplete decomposition method. Table 6.3 shows that the optimization time can be reduced by using incomplete decomposition.

Table 6.3: Optimization time on the SR2201 by using the incomplete decomposition method. Unit is second. Full(8000) is the full optimization time until problem size of 8000. ICD(1/2) is the optimization time with the incomplete decomposition of 1/2 length. The numbers in parentheses show decreased ratios to the time of Full(8000).

#PEs	Full(8000)	ICD(1/2)	ICD(1/4)	ICD(1/8)
4	118401 (1/1)	102437 (1/1.15)	71962 (1/1.64)	39305 (1/3.01)
8	59262 (1/1)	52502 (1/1.12)	34726 (1/1.70)	20477 (1/2.89)
16	28768 (1/1)	25461 (1/1.12)	16785 (1/1.71)	9817 (1/2.93)
#PEs		ICD(1/16)	ICD(1/32)	ICD($\approx 1/n$)
4		21848 (1/5.41)	12020 (1/9.85)	1591 (1/74.4)
8		11426 (1/5.18)	6479 (1/9.14)	1280 (1/46.2)
16		5504 (1/5.22)	3150 (1/9.13)	646 (1/44.5)

The results of Table 6.3 also shows that using incomplete decompositions can reduce optimization time dramatically. The maximal speed-up factors in 4, 8 and 16 PEs were 74.4, 46.2 and 44.5, respectively.

Table 6.4 shows execution time by using auto-tuned parameters with incomplete decompositions. From Table 6.4, using tuned parameters for incomplete decompositions always attains high performance, since all execution time using incomplete decompositions was almost the same as that of complete decomposition Full(8000). This result concludes that using incomplete decompositions is a very useful method in the viewpoint of execution time.

Table 6.4: Execution time on the SR2201 by using the incomplete decomposition method. Unit is second. Full(8000) is the execution time by using full optimization parameter sets until problem size of 8000. ICD(1/2) is the execution time by using the parameter sets in the incomplete decomposition of 1/2 length.

(a) The case of PE = 4.

Size	Full(8000)	ICD(1/2)	ICD(1/4)	ICD(1/8)	ICD(1/16)	ICD(1/32)	ICD($\approx 1/n$)
1000	3.785	3.759	3.861	3.870	3.865	3.764	3.868
2000	26.937	26.794	26.729	26.787	26.752	26.797	26.807
4000	242.010	241.394	241.403	241.393	241.383	241.398	241.492
8000	1962.512	1959.273	1959.311	1959.433	1959.388	1959.323	1959.568

(b) The case of PE = 8.

Size	Full(8000)	ICD(1/2)	ICD(1/4)	ICD(1/8)	ICD(1/16)	ICD(1/32)	ICD($\approx 1/n$)
1000	2.476	2.521	2.530	2.510	2.551	2.515	2.511
2000	14.838	14.591	14.713	14.659	14.562	14.582	14.557
4000	124.205	123.696	123.843	123.893	123.764	123.714	123.735
8000	989.504	987.318	987.717	987.583	987.382	987.480	987.348

(c) The case of PE = 16.

Size	Full(8000)	ICD(1/2)	ICD(1/4)	ICD(1/8)	ICD(1/16)	ICD(1/32)	ICD($\approx 1/n$)
1000	1.824	1.809	1.819	1.805	1.809	1.813	1.805
2000	8.649	8.398	8.343	8.386	8.437	8.366	8.360
4000	56.239	55.447	55.448	55.396	55.400	55.376	55.407
8000	490.346	489.498	489.796	489.724	489.761	489.621	489.828

6.3 New category of a new basic linear algebra subprograms

This section proposes a new category for basic linear algebra subprograms (BLAS). The conventional BLAS (CBLAS) is categorized as the following:

Definition 7 (Conventional BLAS) *The conventional BLAS is constructed by using only one set of performance parameters combination.*

For example, the performance parameter combination consists of loop unrolling depth and blocking length, and so on.

Next is a definition of a new category of BLAS, named *Intelligent BLAS* (IBLAS).

Definition 8 (Intelligent BLAS) *The Intelligent BLAS can change performance parameters set corresponding to the inputted data.*

For example, the performance parameters set consists of matrix size, the number of PEs, and so on.

Main problem is focused on how to change the performance parameter in IBLAS. Using the tuned set of parameters is one of the choices. For example, we consider that an IBLAS is used in a Householder tridiagonalization routine. If tuned parameters sets for the matrix size of 100 (parameter set Υ), 200 (parameter set Φ), and 300 (parameter set Ψ) are obtained in the IBLAS routine, and then the tridiagonalization routine is called with the matrix size of 300. The tridiagonalization routine uses the IBLAS with the matrix size of \hat{n} , ($\hat{n} = 300, 299, \dots, 3$). Therefore, in this case, the IBLAS in the tridiagonalization routine uses the parameters sets by: Ψ for $200 < \hat{n} \leq 300$, Φ for $100 < \hat{n} \leq 200$, and Υ for $2 < \hat{n} \leq 100$. On the other hand, CBLAS uses only one parameters set of Ψ for all \hat{n} , ($\hat{n} = 300, 299, \dots, 3$).

Figure 6.1 shows how IBLAS works.

The characteristics of IBLAS are summarized as follows:

- By using the tuning information of the optimization feature before execution, IBLAS can specify several implementations compared to implementations by hand.
- No additional time is needed to perform IBLAS by using the tuning information of the optimization feature before execution.
- Programs which can apply BLAS can also easily apply IBLAS, since the interfaces of them are same.

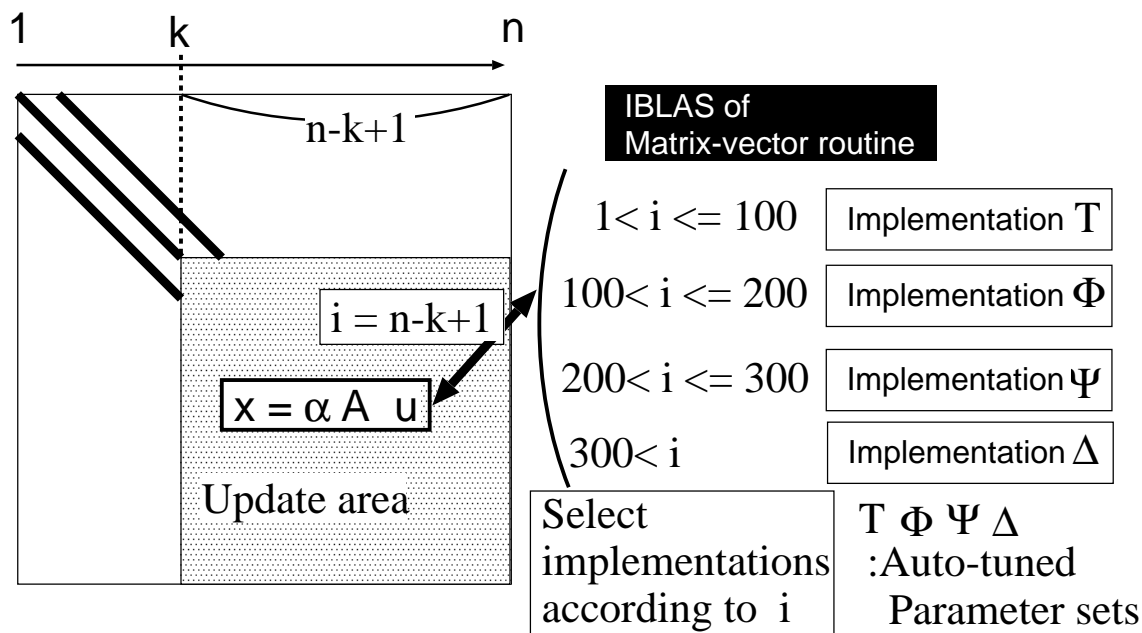


Figure 6.1: How Intelligent BLAS (IBLAS) works.

6.3.1 Evaluation of IBLAS

IBLAS was evaluated by using ILIB_TriRed routine. The ILIB_TriRed routine is an auto-tuning routine for the Householder tridiagonalization for eigenproblems. IBLAS is used for matrix-vector product routine, matrix updating routine, and vector reduction routine in the ILIB_TriRed.

6.3.1.1 Results

Tables 6.5 and 6.6 show the execution time of ILIB_TriRed with CBLAS and IBLAS in 4 PE of the SR2201. Tables 6.5 and 6.6 results that 1.02 – 1.23 times speed-ups could be obtained in comparison with the CBLAS. But we found out that 0.93 – 0.99 times speed-downs were observed by using IBLAS.

6.3.1.2 Discussion

The result of Tables 6.5 and 6.6 shows that CBLAS can not attain optimal performance, since there are cases of decreasing execution time by using IBLAS. Then the tuned parameters set is not always optimal when the matrix dimension changes in the tridiagonalization routine. For example, the tuned parameter set of (Tree, 8, 4) for the matrix size of 601

Table 6.5: ILIB_TriRed with IBLAS on the SR2201. (4PE, $n \leq 1001$) The execution time of tuned BLAS is used for the following tuned parameters sets. The execution time of IBLAS is used for the tuned parameters sets which are less than the specified size of the matrix.

(a) Tuned parameters sets.			
Size	Comm.Type	Mat-Vec	Updating
101	MPI_ALLREDUCE	5	3
201	Tree	6	4
301	MPI_ALLREDUCE	8	4
401	Tree	8	4
501	Tree	8	4
601	Tree	8	4
701	Tree	8	6
801	Tree	3	6
901	MPI_ALLREDUCE	8	4
1001	Tree	6	6

(b) Execution time.			
Size	Tuned CBLAS [sec.]	IBLAS [sec.]	Tuned CBLAS / IBLAS
101	—	—	1.00
201	0.135	0.137	0.98
301	0.272	0.266	1.02
401	0.468	0.456	1.02
501	0.749	0.724	1.03
601	1.385	1.091	1.26
701	1.972	1.765	1.11
801	2.763	2.416	1.14
901	3.528	2.943	1.19
1001	4.051	3.840	1.05

Table 6.6: ILIB_TriRed with IBLAS on the SR2201.(4PE, $n \geq 1001$) The execution time of tuned BLAS is used for the following tuned parameters sets. The execution time of IBLAS is used for tuned parameters sets which are less than the specified size of the matrix.

(a) Tuned parameters sets.				(b) Execution time.			
Size	Comm.Type	Mat-Vec	Updating	Size	Tuned CBLAS [sec.]	IBLAS [sec.]	Tuned CBLAS / IBLAS
2001	Tree	5	6	2001	—	—	1.00
2101	MPI_ALLREDUCE	5	6	2101	31.765	29.655	1.07
2201	MPI_ALLREDUCE	5	6	2201	36.394	33.737	1.07
2301	Tree	8	6	2301	42.701	43.881	0.97
2401	MPI_ALLREDUCE	5	6	2401	47.823	47.235	1.01
2501	MPI_ALLREDUCE	8	6	2501	52.182	48.700	1.07
2601	MPI_ALLREDUCE	5	6	2601	59.294	54.423	1.08
2701	MPI_ALLREDUCE	5	6	2701	65.205	60.593	1.07
2801	MPI_ALLREDUCE	5	6	2801	72.957	67.360	1.08
2901	MPI_ALLREDUCE	6	6	2901	81.169	75.584	1.07
3001	MPI_ALLREDUCE	8	6	3001	89.853	83.505	1.07
3101	Tree	8	6	3101	99.136	102.612	0.96
3201	MPI_ALLREDUCE	5	6	3201	111.354	111.851	0.99
3301	MPI_ALLREDUCE	5	6	3301	113.411	109.390	1.03
3401	MPI_ALLREDUCE	8	6	3401	124.729	119.330	1.04
3501	MPI_ALLREDUCE	8	6	3501	138.383	130.158	1.06
3601	MPI_ALLREDUCE	5	6	3601	151.839	140.829	1.07
3701	MPI_ALLREDUCE	5	6	3701	159.136	155.256	1.02
3801	MPI_ALLREDUCE	5	6	3801	170.913	167.113	1.02
3901	MPI_ALLREDUCE	8	6	3901	197.165	204.716	0.96

may not be the optimal parameters set in the matrix size of 600.

On the other hand, we also observed the cases of increasing execution time by using IBLAS. This results indicate that the changed parameters sets by IBLAS are not optimal sets in some dimensions. Therefore, how parameters set should be changed is difficult problem because we can not obtain all parameters sets in all dimension of the matrix. However, it was observed that the tridiagonalization routine with IBLAS was faster than that of CBLAS.

Therefore, the conclusion is that an auto-tuning routine with IBLAS is crucial in optimization process of auto-tuning software.

6.3.2 Implementation and sampling tuned parameter methods for IBLAS

This section is a discussion on how to implement IBLAS. The IBLAS can be implemented by using the following two methods when the system tries to optimize parameters set:

- (1) Without using the tuned parameters sets which are less than the specified matrix size.
- (2) With using the tuned parameters sets which are less than the specified matrix size.

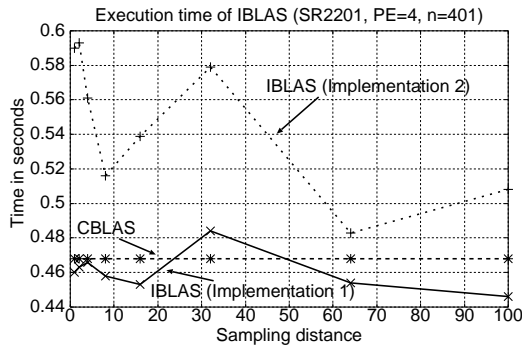
The implementation (1) is used for the previous section, and is defined the IBLAS with the implementation (1). On the other hand, the implementation (2) is one of the reasonable methods in theory because the implementation (2) can use “better” parameter sets with comparison to the case of the fixed parameters set.

Next problem for IBLAS is how to sample the optimized parameters sets. In theory, the routine can attain the best performance when all parameters sets of matrix sizes are obtained. If reduced parameters sets in comparison with the full parameters sets are obtained, the performance of the routine must be decreased with comparison to the routine using full optimized parameters sets. From this discussion, it concludes that we obtain better performance according to the number of tuned parameters sets in theory.

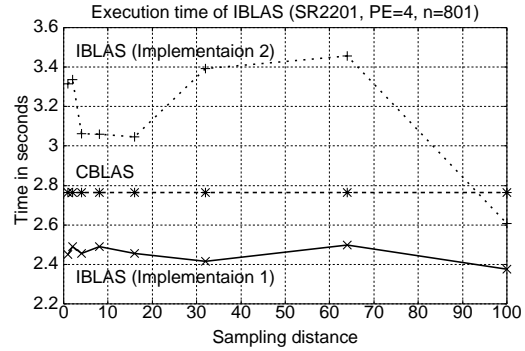
6.3.2.1 Experimental results

This section evaluates the implementation (1) and (2), and sampling distance for IBLAS by using ILIB_TriRed in 4 PEs of the SR2201. This experiments optimize all parameter sets from the matrix size of 1 to 1001. Figure 6.2 shows the results.

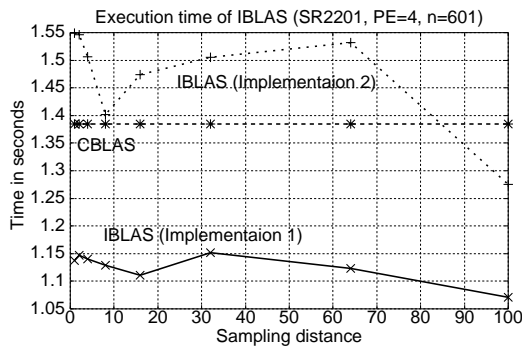
The Figure 6.2 indicate that the execution time of the implementation (2) was slower than that of the implementation (1). In addition, there was a tendency that increasing the number of sampling distance gave us high performance.



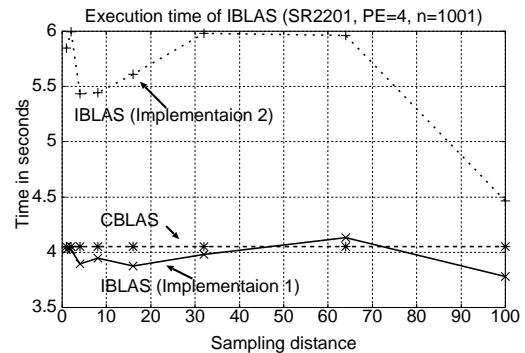
(a) The case of $n = 401$.



(c) The case of $n = 801$.



(b) The case of $n = 601$.



(d) The case of $n = 1001$.

Figure 6.2: Execution time of two kinds implementations for IBLAS. The “sampling distance” in this Figure shows how distance the tuned parameters sets are sampled. For example, the sampling distance 1 stands for optimizing all parameters sets from the matrix size of 1 to 1001, and the sampling distance 2 stands for optimizing parameters sets by stride 2, such as 1, 3, 5,...

6.3.2.2 Discussion

The result of Figure 6.2 is a surprising result in a sense because we thought that the implementation (2) is superior to the implementation (1), and decreasing the number of sampling distance gives us high performance.

The main reason of the results is probably the low hit ratio of instruction cache. Since the implementation (2) causes heavy switching of the implementation of BLAS kernels, this causes heavy misses for Instruction caches. In addition for the same reason, decreasing the number of sampling distance causes low hitting ratio because it performs heavy switching them. Thus, the implementation (2) and decreasing the number of sampling distance gave us low performance.

Conclusion of these results is that the implementation (2) and decreasing sampling distance prevents high performance implementation of IBLAS.

6.4 Conclusion

The author discussed how to develop high performance auto-tuning software in this chapter.

First, the author discussed how to decrease the optimization time for the auto-tuning software. And the following are found:

- Using auto-tuned parameters sets with small sized problems is not a suitable method, since we can not always obtain better parameters sets even if much time is spent on the optimization.
- Using the incomplete decomposition, thus measuring the first step of decomposition, is a sufficient method. By using the incomplete decomposition, the maximal speed-up factor of 74.4 could be obtained in comparison with the execution time of complete one.

The method of using incomplete decomposition can apply many linear algebra computations, since many linear algebra computations are composed of decompositions, such as QR decomposition, LU decomposition, and similarly transformation $Q^{-1}AQ$. Therefore the conclusion is that the method of using incomplete decompositions will be useful method for auto-tuning software in linear algebra computations.

Second, the author proposed a new category of Basic Linear Algebra Subprograms (BLAS), called Intelligent BLAS (IBLAS). The followings are the findings through an evaluation:

- Conventional BLAS can not attain the optimal performance. To attain better performance, BLAS should change the performance parameters sets according to the specified dimension of matrix, like IBLAS.
- IBLAS can not always attain better performance than conventional BLAS, since the optimized parameter sets referred by IBLAS are not always optimal in all dimension of the matrix. But IBLAS is crucial in auto-tuning software because there is a case that the performance is improved.
- Do not use tuned parameter sets which are less than the specified matrix size when on the optimization process. In addition, the sampling distance of parameter sets

should be large value, since these methods cause heavy instruction cache miss in IBLAS routine. This miss prevents high performance implementation of IBLAS.

The concept of IBLAS can easily apply to all subroutines which uses BLAS. Therefore, the author thinks that the proposed idea is very useful to attain high performance in real numerical computations for linear algebra.

Part IV

Conclusion Remarks Part

Chapter 7

Adaptation and Future Work

7.1 Introduction

As explained in part I of this thesis, we have proposed and developed a new parallel Householder reduction algorithm, which uses a reduced communication method [13, 14, 39, 15, 40, 41, 89, 84, 16]. Especially, the algorithm works well when we are in MPP environments, and the small dimension problem must be solved. Why the algorithm works well is explained as: Using reduced communication scheme and the nature of non-symmetry explained in Chapter 1.

Figure 7.1 summarizes the overview of the parallel Householder reduction algorithm in the viewpoint of computational and communication complexities. Note that the overview of Figure 7.1 can easily extend the algorithm to the non-symmetric reduction, called Hessenberg reduction. Therefore, the overview shows the nature of the Householder similarity transformation.

In the overview of Figure 7.1, there are four types of algorithms in the viewpoint of calculation and communication costs. The algorithms can be classified as:

- A: High computation efficiency algorithm based on the conventional communication method.
- B: Conventional algorithm by using the conventional communication method [20, 100, 19, 21].
- C: Proposed algorithm in this thesis.
- D: Parallel algorithm by using the reduced communication method based on the sequential blocked algorithm [11, 12, 101, 10].

The four types of algorithms showed in the above have an adaptability according to the parallel processing environments. In this chapter, we explain and discuss the adaptability of four algorithms, and an example of the real applications is shown.

7.2 Adaptability of each algorithms

7.2.1 The algorithm in box A

This algorithm is composed of normal communication method and the nature of non-symmetry. Therefore it needs the calculation complexity of $8/(3p)n^3$, the communication complexity of $n^2 \log p$, and the communication time of n .

The nature of using normal communication method, the algorithm prefers small-scale parallel processing to MPP because of the small communication times. The algorithm also prefers performing the small dimension problems because of the high computation efficiency.

From these characteristics, the algorithm can be used when the number of processors is small, such as 4, 8 and 16, and the matrix size is also small. The kernel of the algorithm is simple, and its inner-loop length is longer than that of symmetry case. Therefore we think that the algorithm is suitable for Vector-Parallel distributed machines when we need to solve small dimension problems.

7.2.2 The algorithm in box B

This algorithm is composed of normal communication method, and the nature of symmetry. Therefore it needs the calculation complexity of $4/(3p)n^3$, the communication complexity of $n^2 \log p$, and the communication time of n .

The nature of using normal communication method, the algorithm prefers small-scale parallel processing to MPP because of the small communication times. The algorithm also prefers performing the large dimension problems because of the low computation complexity.

From these characteristics, the algorithm can be used when the number of processors is small, such as 4, 8 and 16, and the matrix size is large. The kernel of the algorithm has a complex access pattern, and its inner-loop length is shorter than that of non-symmetry case. Therefore we think that the algorithm is suitable for Scalar-Parallel RISC-Based distributed machines when we need to solve large dimension problems.

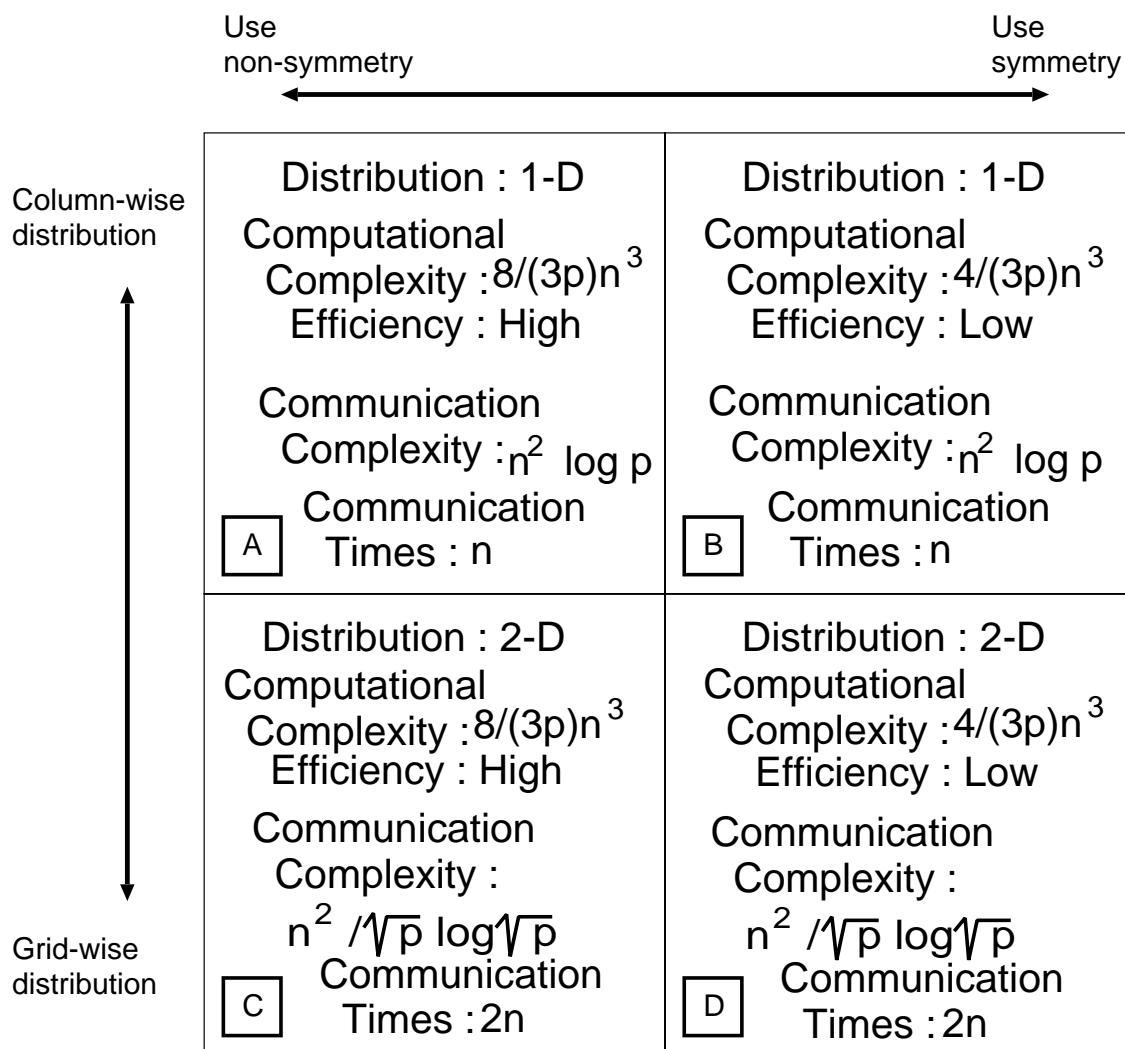


Figure 7.1: An overview of the computational and communication complexities for the parallel Householder reduction. The algorithm in box A will be used when data per processor is small, and the number of processors is small. The algorithm in box B shows the conventional parallel algorithms. The algorithm in box C shows the proposed parallel algorithm in this thesis. The algorithm in box D shows a considerable algorithm, when data per processor is huge, and the number of processors is big enough value.

7.2.3 The algorithm in box C

This algorithm is composed of the reduced communication method, and the nature of non-symmetry. Therefore it needs the calculation complexity of $8/(3p)n^3$, the communication complexity of $(n^2/\sqrt{p}) \log \sqrt{p}$, and the communication time of $2n$.

The nature of using reduced communication method, the algorithm prefers MPP to small-scale parallel processing because of the communication complexities. The algorithm prefers performing the small dimension problems because of the high computation efficiency.

From these characteristics, the algorithm can be used when the number of processors is large, such as 128, 256 and 1024, and the matrix size is small. The kernel of the algorithm is simple, and its inner-loop length is longer than that of symmetry case. Therefore we think that the algorithm is suitable for MPP Vector-Parallel distributed machines when we need to solve small dimension problems.

We believe that almost all real super-computing applications can apply the algorithm, since they need the MPP environments to attain speed-up, and the dimension of the problem should be small because of MPP environments.

7.2.4 The algorithm in box D

This algorithm is composed of the reduced communication method, and the nature of symmetry. Therefore it needs the calculation complexity of $4/(3p)n^3$, the communication complexity of $(n^2/\sqrt{p}) \log \sqrt{p}$, and the communication time of $2n$.

The nature of using reduced communication method, the algorithm prefers MPP to small-scale parallel processing because of the communication complexities. The algorithm also prefers performing the large dimension problems because of the low computation complexity.

From these characteristics, the algorithm can be used when the number of processors is large, such as 128, 256 and 1024, and the matrix size is also large. The kernel of the algorithm has a complex access pattern, and its inner-loop length is shorter than that of non-symmetry case. Therefore we think that the algorithm is suitable for MPP Scalar-Parallel RISC-Based distributed machines when we need to solve large dimension problems.

7.3 Applications for real problems

This section explains an adaptation of the algorithm in box C which is proposed algorithm in this thesis to a real problems.

7.3.1 An example : Chemical field

According to S.Tajima *et.al.* [84], there is a case where about 7000 times eigendecomposition is needed in an application for chemical field. In the application, the main contributor is Householder tridiagonalization, since the Householder-Bisection algorithm is used. In addition, the scale of the tridiagonalization should be small because of total execution time (Consider 7000 times tridiagonalization. If the time of one tridiagonalization is 1 minute, the total execution time is ≈ 116.7 hours ≈ 4.9 days!)

Thus, from the limitation of total execution time, the execution time per diagonalization should be fast, and this also indicates that the dimension of Householder tridiagonalization should take small value. Although such application requires the small dimension problem, it also requires that the execution time per tridiagonalization should be fast because of total execution time. Simple answer to attain the speed-up is using parallel machines. However, accomplishing the speed-up is hard in this case, since the dimension of matrix is small. In addition, the Householder tridiagonalization in this case has less parallelism in the nature of computation and communication complexities.

Our developed Householder-Bisection routine based on the algorithm in box C is used in their application as a diagonalization routine. For the total speed-up ratio of their application, they attained about 300 times speed-ups in comparison with a conventional method. One of their main efforts is modifying the method by using only eigenvalue to calculate molecular energy, but the high parallel efficiency of Householder reduction routine is also a considerable factor. The parallel efficiency of the routine was more 90% in 8 processors of PC cluster, which was used for the DEC Alpha2116A/533MHz processor and 100 BaseT switch for the communication. The problem size was 572 dimension of matrix (230 molecules) [84]. Of course, to attain the high performance of 90% more in the very small matrix size of 572 dimension, there is no room to discuss the effectiveness for the algorithm in box C. If a free-released library based on the algorithm in box D, such as ScaLAPACK, is used in this case, such high efficiency can not be obtained because of the low computational efficiency.

From the discussion above, the proposed algorithm in this thesis (the algorithm in box C in Figure 7.1) will be a powerful method when we need to perform a lot of tridiagonalization. We believe that most of all software in application field is same as the software

showed in this section, and the effectiveness of the proposed algorithm is much stronger in MPP environments. Therefore the proposed algorithm will be a powerful and useful algorithm in real problems.

7.4 Future work

The future work is summarized as follows.

- (1) Extension to band matrices.
- (2) Extension to sparse matrices.

The reduced communication algorithm proposed in this thesis is for dense symmetric or non-symmetric Householder reductions. To remove the limitation, we have to extend the reduced communication algorithm to general matrices.

As for (1), a reduced computational complexity algorithm for band matrices was proposed in 1975 by K.Murata [102]. The algorithm is called *Murata method*, and the Murata method uses many small-sized Householder transformation to keep the shape of a band matrix. We think that the concept of the reduced algorithm may be adapted to the small-sized Householder transformation. When the Murata method in MPP environments has to be performed, the algorithm may be useful algorithm. In addition, we have to summarize the algorithm for band matrices in the viewpoint of symmetry and data distribution as shown in this chapter.

As for (2), developing the reduction algorithm for sparse matrices is much more difficult than that of band matrices in general. From the nature of sparse matrix, the parallelism of the reduction algorithm is low. Therefore parallelizing the algorithm is a challenging work. However, several applications require solving the eigenproblem of sparse matrix. Thus, we need some methods to parallelize the reduction algorithm for sparse matrices. Adapting the reduced algorithm to the reduction algorithm for sparse matrices is now useful and challenging work.

7.5 Conclusion

In this chapter, the author discussed the adaptability of proposed Householder reduction algorithm in Chapter 1 of this thesis. The reduction algorithms can be classified as 4 types according to the nature of symmetry and data distribution methods. By analyzing the computation and communication complexities and the computational efficiency, the suitable algorithm for parallel machines can be determined.

The suitable algorithms are summarized as the followings.

- The algorithm in box A : To solve Small-Dimension problems with Small-Scale Vector-Parallel Machines.
- The algorithm in box B : To solve Large-Dimension problems with Small-Scale Scalar-Parallel RISC-Based Machines.
- The algorithm in box C : To solve Small-Dimension problems with MPP Vector-Parallel Machines.
- The algorithm in box D : To solve Large-Dimension problems with MPP Scalar-Parallel RISC-Based Machines.

The author believes that many real applications can apply the algorithm in box C which is proposed in this thesis, since the nature of the applications requires the MPP environments, and it makes the problem sizes small. In addition, from the limitation of total execution time for real application, the problem size should take much smaller values. Therefore the author concludes that proposed algorithm can be adapted to a various real applications in many scientific and technical fields.

Chapter 8

Summary and Conclusion

In this thesis, the author proposed several new algorithms and new implementations for parallel large scale eigensolvers. Here, the “large scale” means that it requires not only huge computations or memory spaces, but it also requires a large number of processor elements. In addition, it also means managing or developing huge codes to attain high performance.

To accomplish “high performance” for environments of massive parallel processing in eigenproblem, the author firstly proposed a new parallel approach for Householder reduction. The new parallel approach is suitable for massive parallel processing environment by developing parallel library with the different performance parameters of computation and communication.

To accomplish “high performance” for computation, the author proposed a new orthogonalization method which needs to compute “clustered” eigenvalue problem. The new orthogonalization method also can attain high-speed because the convergence of eigensolver is improved in a numerical experiment.

To accomplish “high performance” for managing or developing huge codes, the author proposed a methodology for optimization of codes to attain high performance, named optimization feature before execution. By using the tuned performance parameter, the author derived the new category of basic linear algebra subprograms.

8.1 Conclusion of main parts

Overall in this thesis, the author met the implementation and its effects on real parallel machines. The key aspects are to keep load balancing and to implement computations and communications well. If there is a trade-off problem between accuracy and parallel efficiency, using high efficient parallel algorithm is the best approach if the algorithm can

be modified to the accuracy in general.

The following subsections summarize the author's conclusions for major aspects in main chapters of this thesis.

8.1.1 Algorithm of parallel dense eigensolver part

8.1.1.1 High performance parallel Householder-Bisection algorithm

- The author formulated, implemented and evaluated a parallel eigensolver based on the proposed new approach on an Massive Parallel Processing (MPP) system.
- The proposed tridiagonal reduction algorithm uses a non-symmetry of matrices and the reduced communication method by using the (Cyclic, Cyclic) grid-wise data distribution. Using the non-symmetry and the reduced communication methods on Householder tridiagonal reduction is quite new approach because the conventional algorithms have been using the symmetry of matrices, and have not always been using the reduced algorithm. In addition, the proposed algorithm can easily extend to non-symmetric matrices (Hessenberg reduction).
- Parallel libraries should be constructed by using different blocking factors of block length in data distribution (*BDD*) and block length in blocking algorithm (*BBA*), since the *BBA* does not depend on the data distribution in many cases.
- By using these different parameters of *BDD* and *BBA*, we can construct high performance Householder tridiagonal routine on MPP environments.
- Our approach is also useful when the problem of small matrix sizes is solved. This indicates that our approach is very important in real applications which need many tridiagonalizations, for example it performs 6000 times or more tridiagonalizations. In a chemical field problem, more than 90% efficiency in 8 processors of a PC cluster was attained by using our approach for a very small problem of 572 dimension.
- As a result in this chapter, the author obtained the speed-up factor of 1.3 – 5.7 in comparison with a ScaLAPACK routine. This shows that our proposed approach is enough fast to conventional algorithms. In addition, the effect of parallelism becomes stronger when the number of PEs increases. Our approach, therefore, is very efficient on MPPs.

8.1.1.2 New parallel orthogonalization method

- The author proposed a new orthogonalization method with sorting, named the *Classical Gram-Schmidt with Sorting (CGSS)* method. As the results of numerical experiments, the author found that the CGSS method not always improve accuracy, but it can dramatically improve accuracy in a case against conventional methods.
- The author classified the algorithms of classical Gram-Schmidt, modified Gram-Schmidt, and CGSS in the viewpoint of parallel processing. In addition, useful points of these algorithms are shown in this section. The classification shown in this section is a new classification in the viewpoint of parallel Gram-Schmidt orthogonalizations.
- The CGSS method causes a poor orthogonality in a case. From this, the author can conclude that determination of the process for data after sorting is very important.
- The CGSS method may be a useful method. If so, we should have a hardware mechanism to perform its sorting and after adding process. This mechanism gives us not only high-speed processing, but also high accuracy.
- The proposed data distribution method, named *Cyclic Triangular Distribution (CTD)*, can improve load imbalance in the orthogonalization process. The improvement is archived by triangularly formed processes. There are many triangularly formed processes in the numerical processing. Therefore we have to implement the CTD strategy on some of auto-parallelizing compilers, such as HPF or FortranD.
- The author found that “Dividing the computation” gives us higher accuracy than non dividing case from numerical experiments. The parallel Gram-Schmidt methods, therefore, are superior to sequential ones in the parallel execution speed, the orthogonality, and memory usage. This means that large scale numerical processing should perform on parallel distributed memory computers.

8.1.2 Application of parallel dense eigensolver part

8.1.2.1 Optimized feature before execution

- The author focused on the *optimization feature before execution* in this chapter, which performs an optimization to attain high performance before target routine is called. The main aim of the feature is to remove tuning work of large-scale codes, since the work is time-consuming. The effect of the optimization feature also

contains that we can obtain high performance routines easily, and we can reduce mistakes by specifying wrong parameter sets.

- The author has implemented and evaluated a parallel eigensolver by using an optimization methodology. Selecting suitable implementations for the global summation, the matrix-vector product, the process to update, and the blocking factor for communication on the parallel eigensolver is the optimization methodology.
- Our methodology is a different method from the conventional methods of PHiPAC or ATLAS, since our methodology can perform global optimizations between computations and communications, while the conventional methods only perform local optimizations of BLAS. By using our methodology, much higher performance can be attained compared to the conventional local optimization methods for BLAS. In addition, our methodology can also be easily applied to parallel libraries, while the conventional methods can only apply sequential libraries.
- The target processes of the parallel eigensolver are a Householder tridiagonalization process, a Householder inverse iteration process, and a blocked Modified Gram-Schmidt (MGS) orthogonalization process. The computational kernels of these processes form BLAS 2, BLAS 1, and BLAS 3, respectively. The target processes, therefore, is one of the good benchmarks in a sense because these processes include many kinds of computation in linear algebra calculations. On the other hand, the target processes have important communication processes. They are reduction operation and blocking factor for control of communication times.
- Even though the author used a quite simple methodology to optimize the routines, the author obtained about 1.1–2.3 times speed-ups with respect to the routine for which the reasonable parameters in the SR2201 and the SR8000 were specified.
- From results of this chapter, the author concluded that such an auto-tuning methodology is an effective technique. The author, therefore, suggests that all of linear algebra libraries should contain the proposed optimization features to attain high performance and reduce optimization time by hands.
- To accomplish high portability and uniform accuracy in every packages of auto-tuning software, the author suggests that we have to develop the common interface of auto-tuning software.

8.1.2.2 Towards high performance auto-tuning software

- The author discussed how to develop “high performance” auto-tuning software. The “high performance” auto-tuning software is defined : (1) It can quickly find a parameter set which can execute the program at fast; (2) Optimization time for finding better parameter set is also fast; (3) If much longer optimization time was set, a much better parameter set is found. To accomplish the high performance, the author discussed the four methods to reduce optimization time.
- Using auto-tuned parameters sets with small sized problems is not a suitable method, since we can not always obtain better parameters sets even if we spend much time on the optimization.
- Using the “incomplete decomposition method”, thus measuring the first step of decomposition, is a sufficient method. By using the incomplete decomposition method, we obtained the maximal speed-up factor of 74.4 in comparison with the execution time of complete one. For optimization time, we needed about 33 hours to perform the optimization of Householder tridiagonalization routine. By using this method, the optimization time is reduced by 1591 seconds to obtain same parameters sets.
- The method of using incomplete decomposition can be applied to many linear algebra computations, since many linear algebra computations are composed of basic decompositions. For example, the proposed method for reducing optimization time can apply LU decompositions, QR decompositions, FFTs, and so on. Therefore, the author believes that the method of using incomplete decompositions will be a useful one for auto-tuning software in linear algebra computations.
- Next in this chapter, the author showed a new category of Basic Linear Algebra Subprograms (BLAS), called *Intelligent BLAS (IBLAS)*. By extending the concept of the optimization feature, the idea of IBLAS can be derived.
- Conventional BLAS (CBLAS) can not attain the optimal performance. To attain better performance, BLAS should change the performance parameters sets according to the specified dimension of matrix, like IBLAS. By using IBLAS, the author obtained the speed-up factor of 1.02–1.26 compared to CBLAS.
- IBLAS can not always attain better performance than CBLAS, since the optimized parameters sets referred by IBLAS are not always optimal for all dimension of matrix. But IBLAS is crucial in auto-tuning software because there is a case that the performance is improved.

- Do not use tuned parameters sets which are less than the specified matrix size, when the optimization process. In addition, the sampling distance of parameters sets should be a large value, since these methods cause heavy instruction cache miss in IBLAS routine. This miss prevents high performance implementation of IBLAS.
- The concept of IBLAS can easily apply all subroutines which use BLAS. Therefore, the author concludes that IBLAS is very useful to attain high performance in real numerical computations for linear algebra.

8.2 Advice to symmetric eigensolver users

The author warns parallel library users for eigenproblem that the parallelism or performance of these libraries greatly depends on the characteristics of eigenvalues. If the calculated eigenvalues form clusters, we can not obtain high performance in theory. Even though many useful computation methods for clustered eigenvalues have been proposed, these methods do not satisfy these theory, such as it needs extremely high accuracy. Basically, only one method to compute the clustered eigenvalues is orthogonalizing eigenvectors corresponding to the clustered eigenvalues. The orthogonalization, however, prevents high performance parallel implementation.

There are many reasonable approaches for eigenvalue computations. If input matrix is sparse and you need small number of eigenpairs, use a sparse eigensolver, which uses Lanczos algorithm for example. If you need almost all or high accuracy eigenpairs and the matrix size for input matrix is small ($n < 100\sqrt{p}$), use parallel Jacobi eigensolver [26]. If you need almost all eigenpairs and the input matrix is dense symmetric, use parallel eigensolver based on Householder-bisection type methods, as shown in this thesis.

Currently, we can use many public domain software for symmetric eigenproblem. The most readily available parallel dense symmetric solvers are ScaLAPACK's PDSYEV and PDSYEVX. These solvers, however, have a problem of orthogonality because inter-processor orthogonalization is not performed. If the routines perform the inter-processor orthogonalization, the parallelism of them are decreased. As a result, we obtain poor performance for these routines. If you use HITACHI's parallel computer, the eigensolver of MATRIX MPP is one of the good choices, since the eigensolver uses a new approach, named *multi-color inverse iteration method* [38]. This method gives us high parallel efficiency even if there are clustered eigenvalues. The multi-color inverse iteration method cannot compute ill-conditioned eigenvalues at high performance, such as almost all eigenvalues are same values because of the feature of their method. If you need to solve the ill-conditioned

eigenvalue problem, use parallel inverse eigensolvers which implemented orthogonalization process. For example, the *Householder inverse iteration method* [44], and the *inverse iteration with hybrid CGS method* [41] or CGSS method proposed in this thesis by the author. The new inverse iteration methods by the author will be opened through HINTS project (<http://www.hints.org/>) at Kanada laboratory, the University of Tokyo. Finally, if you know there are no ill-conditioned eigenvalues in input matrices, I.Dhillon's parallel algorithm [45] is the best method, since his algorithm is based on inverse iteration without orthogonalization, and there are no inter-processor and intra-processor orthogonalizations.

8.3 Closing remarks

The author obtained the speed-up factor of 1.3–5.7 by applying the author's new approach in the Householder tridiagonalization, the speed-up factor of 1.1–2.3 by using the proposed optimization feature, and the speed-up factor of 1.02–1.26 by using IBLAS which is a new category of BLAS. From these results, the author concludes that 1.45–16.5 times faster routine compared to conventional routines is obtained by using new methods and approaches proposed in this thesis. In addition, these speed-up factors become higher when the number of PEs increases. This indicates that the important aspect is to know how to implement and design the parallel algorithms with real parallel machines.

Finally, I chose this topic because I believe that computer scientists have been studied and developed too many parallel algorithms, but not enough implementation and evaluation by using real parallel machines. I hope this study leads to a new break-through in eigenvalue computation algorithms.

References

- [1] Berry, M. W., Dumais, S. T. and O'Brien, G. W.: Using Linear Algebra for Intelligent Information Retrieval, *SIAM Review*, Vol. 37, No. 4, pp. 573–595 (1995).
- [2] Berry, M. W. and Fierro, R. D.: Low-Rank Orthogonal Decompositions for Information Retrieval Applications, *Numerical Linear Algebra with Applications*, Vol. 3, No. 4, pp. 301–328 (1996).
- [3] Berry, M. W., Dramač, Z. and Jessup, E. R.: Matrices, Vector Spaces, and Information Retrieval, *SIAM Review*, Vol. 41, No. 2, pp. 335–362 (1999).
- [4] Parlett, B. N.: *The Symmetric Eigenvalue Problem*, SIAM (1980).
- [5] Oguni, T., Murata, K., Miyoshi, T., Dongarra, J. and Hasegawa, H.: *Matrix Calculation Softwares, WS, Supercomputer, Parallel Machines*, Maruzen (1991). in Japanese.
- [6] Demmel, J. W.: *Applied Numerical Linear Algebra*, SIAM (1997).
- [7] Trefethen, L. N. and BauIII, D.: *Numerical Linear Algebra*, SIAM (1997).
- [8] Stewart, G. W.: *Matrix Algorithms Volume I: Basic Decompositions*, SIAM (1998).
- [9] Katagiri, T. and Kanada, Y.: Performance Evaluation of Blocked Householder Algorithm on Distributed Memory Parallel Machine, *Trans. IPS. Japan*, Vol. 39, No. 7, pp. 2391–2394 (1998). in Japanese.
- [10] Hendrickson, B., Jessup, E. and Smith, C.: Toward an Efficient Parallel Eigensolver for Dense Symmetric Matrices, *SIAM J. Sci. Comput.*, Vol. 20, No. 3, pp. 1132–1154 (1999).
- [11] Chang, H., Utku, S., Sakama, M. and Rapp, D.: A Parallel Householder Tridiagonalization Stratagem Using Scattered Square Decomposition, *Parallel Computing*, Vol. 6, pp. 297–311 (1988).

- [12] Hendrickson, B. A. and Womble, D. E.: The Tours-Wrap Mapping for Dense Matrix Calculation on Massively Parallel Computers, *SIAM Sci. Comput.*, Vol. 15, No. 5, pp. 1201–1226 (1994).
- [13] Katagiri, T. and Kanada, Y.: Performance Evaluation of Householder Method for the Eigenvalue Problem on Distributed Memory Architecture Parallel Machine, *IPAJ SIG Notes 96-HPC-62*, pp. 111–116 (1996). in Japanese.
- [14] Katagiri, T. and Kanada, Y.: An Unblocked Hessenberg Reduction Algorithm for Distributed Memory Architecture Parallel Machine and its Evaluation, *Proceedings of Joint Symposium on Parallel Processing JSPP'97*, pp. 385–392 (1997). in Japanese.
- [15] Katagiri, T. and Kanada, Y.: An Unblocked Hessenberg Reduction Algorithm for Distributed Memory Architecture Parallel Machine and Its Effectiveness, *Trans. IPS. Japan*, Vol. 39, No. 11, pp. 3065–3075 (1998). in Japanese.
- [16] Katagiri, T., Kuroda, H. and Kanada, Y.: A Methodology for Automatically Tuned Parallel Tri-diagonalization on Distributed Memory Parallel Machines, *Proceedings of Vector and Parallel Processing (VECPAR) 2000*, Porto, Portugal, pp. 265 – 277 (2000).
- [17] Stewart, G.: A Parallel Implementation of the QR-Algorithm, *Parallel Computing*, Vol. 5, pp. 187–196 (1987).
- [18] Sekiguchi, T. and Oyanagi, Y.: Methods in Parallel Scientific Computation, *J. IPS Japan*, Vol. 27, No. 9, pp. 985–994 (1986). in Japanese.
- [19] Kalamboukis, T.: The Parallel Algorithm for the Dense Symmetric Eigenvalue Problem on a Transputer Array, *Parallel Computing*, Vol. 18, pp. 207–212 (1992).
- [20] Chang, H., Utku, S., Sakama, M. and Rapp, D.: A Parallel Householder Tridiagonalization Stratagem Using Scattered Row Decomposition, *I. J. Num. Meth. Eng.*, Vol. 26, pp. 857–874 (1988).
- [21] Dongarra, J. J. and van de Geijn, R. A.: Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures, *Parallel Computing*, Vol. 18, pp. 973–982 (1992).

- [22] Strazdins, P.: Optimal Load Balancing Techniques for Block-Cyclic Decompositions for Matrix Factorization, Technical Report TR-CS-98-10, The Australian National University, Joint Computer Science Technical Report Series (1998).
- [23] Bischof, C., Sun, X. and Lang, B.: Parallel Tridiagonalization Through Two-Step Band Reduction, Technical Report 17, PRISM Working Note (1994).
- [24] Bischof, C. and van Loan, C.: The WY Representation for Products of Householder Matrices, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 1, pp. s2–s13 (1987).
- [25] Demmel, J. and Veselic, K.: Jacobi’s Method is More Accurate than QR, Technical Report 15, LAPACK Working Note (1989). also University of Tennessee, Knoxville, Technical report CS-89-88.
- [26] Stanley, K. S.: Execution Time of Symmetric Eigensolvers, *Ph.D Thesis, Computer Science Division, University of California at Berkeley* (1997).
- [27] Pourzandi, M. and Tourancheau, B.: A Parallel Performance Study of Jacobi-like Eigenvalue Solution, Technical Report CS-94-226, University of Tennessee, Knoxville (1994).
- [28] Arbenz, P. and Oettli, M.: Block Implementations of the Symmetric QR and Jacobi Algorithms, Technical Report 178, Eidgenoessische Technische Hochschule Zurich (1992). This report is also available via anonymous ftp from ftp.inf.ethz.ch as doc/tech-reports/1992/178.ps.
- [29] Gimenez, D., Hernandez, V., van de Geijn, R. and Vidal, A. M.: A Jacobi Method by Blocks on a Mesh of Processors, *Concurrency : Practice and Experience*, Vol. 9, No. 5, pp. 391 – 411 (1997).
- [30] Chinchalkar, S. and Coleman, T.: Computing Eigenvalues and Eigenvectors of a Dense Real Symmetric Matrix on the NCUBE 6400, Technical Report 91-074, Cornell University, Theory Center (1991).
- [31] Bischof, C. H.: QR Factorization Algorithms for Coarse-Grained Distributed Systems, *Ph.D Thesis, Department of Computer Science, Cornell University* (1988).
- [32] Henry, G. and van de Geijn, R.: Parallelizing the QR algorithm for the Unsymmetric Algebraic Eigenvalue Problem : Myths and Reality, *SIAM J. Sci. Comput.*, Vol. 17, No. 4, pp. 870–883 (1996).

- [33] Suda, R., Nishida, A. and Oyanagi, Y.: A High Performance Parallelization Scheme for the Hessenberg Double Shift QR Algorithm, *Parallel Computing*, Vol. 25, pp. 729–744 (1999).
- [34] Shimasaki, M.: Parallel Eigenvalue Search for Symmetric Tridiagonal Matrices, Technical Report 514, Suuri kaiseiki kenkyujyo koukyuroku, Kyoto university (1984). in Japanese.
- [35] Dongarra, J. J. and Sorensen, D. C.: A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 2, pp. s139–ss154 (1987).
- [36] Ipsen, I. C. and Jessup, E. R.: Solving The Symmetric Tridiagonal Eigenvalue Problem on The Hypercube, *SIAM J. Sci. Stat. Comput.*, Vol. 11, No. 2, pp. 203–229 (1990).
- [37] Tisseur, F. and Dongarra, J.: A Parallel Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem on Distributed Memory Architectures, *SIAM J. Sci. Comput.*, Vol. 20, No. 6, pp. 2223–2236 (1999).
- [38] Naono, K., Igai, M. and Yamamoto, Y.: Development of a Parallel Eigensolver and its Evaluation, *Proceedings of Joint Symposium on Parallel Processing (JSPP)'96*, pp. 9–16 (1996). in Japanese.
- [39] Katagiri, T. and Kanada, Y.: A Parallel Implementation of Eigensolver and its Performance, *IPSJ SIG Notes, 97-HPC-69*, pp. 49–54 (1997).
- [40] Katagiri, T.: A Study on Parallel Implementation of Large Scale Eigenproblem Solver for Distributed Memory Architecture Parallel Machines, *Master's Degree Thesis, Department of Information Science, Graduate School of Science, the University of Tokyo* (1998).
- [41] Katagiri, T. and Kanada, Y.: A Parallel Implementation of Eigensolver and an Improvement of Its Parallelism, *Proceedings of Joint Symposium on Parallel Processing (JSPP)'98*, pp. 223–230 (1998).
- [42] Vanderstraeten, D.: A Generalized Gram-Schmidt Procedure for Parallel Applications, *unpublished reports* (1997).

- [43] Vanderstraeten, D.: A Parallel Block Gram-Schmidt Algorithm with Controlled Loss of Orthogonality, *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing* (1999).
- [44] Yamamoto, Y., Igai, M. and Naono, K.: A New Algorithm for Accurate Computation of Eigenvectors on Shared-Memory Parallel Processors, *Proceedings of Joint Symposium on Parallel Processing (JSPP)'2000*, pp. 19–26 (2000). in Japanese.
- [45] Dhillon, I.: A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue / Eigenvector Problem, *Ph.D Thesis, Computer Science Division, University of California at Berkeley* (1997).
- [46] Dhillon, I., Fann, G. and Parlett, B.: Application of a New Algorithm for the Symmetric Eigenproblem to Computational Quantum Chemistry, *Proc. 8th SIAM conference on Parallel Processing for Scientific Computing* (1997).
- [47] Fann, G. and Littlefield, R.: Performance of a fully parallel dense real symmetric eigensolver in quantum chemistry applications, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computation, SIAM* (1994).
- [48] Jessup, E. R. and Ipsen, I. C. F.: Improving the Accuracy of Inverse Iteration, *SIAM J. Sci. Stat. Comput.*, Vol. 13, No. 2, pp. 550–572 (1992).
- [49] HITACHI: The MATRIX/MPP WEB page.
<http://www.hitachi.co.jp/Prod/comp/soft1/hmpp-e/m0010049.htm>.
- [50] HITACHI: The HI-UX/MPP for SR8000 WEB page.
<http://www.hitachi.co.jp/Prod/comp/soft1/sr8000/index-e.html>.
- [51] HITACHI: The HITACHI SR2201 Web Page.
<http://www.hitachi.co.jp/Prod/comp/hpc/eng/sr1.html>.
- [52] HITACHI: The HITACHI SR8000 Web Page.
<http://www.hitachi.co.jp/Prod/comp/hpc/eng/sr81e.html>.
- [53] IBM: The Parallel ESSL WEB page.
<http://www.rs6000.ibm.com/software/sp-products/esslspec.html>.
- [54] IBM: The IBM RS6000 Web Page. <http://www.ibm.co.jp/rs6000/index.html>.

- [55] Dongarra, J. J., Croz, J. D., Hammarling, S. and Hanson, R. J.: An Extended Set of FORTRAN Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, pp. 1–17 (1988).
- [56] Dongarra, J. J., Croz, J. D., Hammarling, S. and Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 1–17 (1988).
- [57] Dongarra, J. J. and Whaley, R. C.: A User’s Guide to the BLACS v1.1, Technical Report 94, LAPACK Working Note (1997).
- [58] Whaley, R. C. and Dongarra, J.: BLACS Home Page. <http://www.netlib.org/blacs/>.
- [59] PBLAS Home Page. http://www.netlib.org/scalapack/html/pblas_qref.html.
- [60] Choi, J., Dhillon, I., Dongarra, J., Ostrouchv, S., Petitet, A., Stanley, K., Walker, D. and Whaley, R.: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance, Technical Report 95, LAPACK Working Note (1995).
- [61] Geist, A., Beguelin, A., Dongarra, J., Manchek, R., Jiang, W. and Sunderam, V.: *PVM: A Users’ Guide and Tutorial for Networked Parallel Computing*, The MIT Press (1994).
http://www.epm.ornl.gov/pvm/pvm_home.html.
- [62] Message_Passing_Interface_Forum: MPI : A Message-Passing Interface Standard (1995).
<http://phase.etl.go.jp/mirrors/mpif/docs/docs.html>.
- [63] Message_Passing_Interface_Forum: MPI-2 : Extension to the Message-Passing Interface (1997).
<http://phase.etl.go.jp/mirrors/mpif/docs/docs.html>.
- [64] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D.: LAPACK: A Portable Linear Algebra Library for High-Performance Computers, Technical Report 20, LAPACK Working Note (1990).
- [65] Brent, R., Grosz, L., HarrarII, D., M, H., Kahn, M., Keating, G., Mercer, G., Nielsen, O., Osborne, M. and Zhou, B.: Development of a Mathematical Subrou-

- tine Library for Fujitsu Vector Parallel Processors, *Proceedings of International Conference on Supercomputing*, pp. 13–20 (1998).
- [66] Fujitsu: The VPP Family WEB page.
http://www.fujitsu.co.jp/hypertext/Products/Info_process/hpc/vpp-e/.
- [67] HarrarII, D. L., Kahn, M. H. and Osborne, M. R.: Parallel Eigenvalue Routines on the Fujitsu VPP300 (1997). http://anusf.anu.edu.au/Area4_Working_Notes/.
- [68] Brent, R., Grosz, L., HarrarII, D., Hegland, M., Kahn, M., Keating, G., Mercer, G., Osborne, M., Zhou, B. and Nakanishi, M.: Design of the Scientific Subroutine Library for the Fujitsu VPP300 (1998). Submitted to HPCAsia 98, Singapore, http://anusf.anu.edu.au/Area4_Working_Notes/.
- [69] ILIB Project Home Page.
http://www.super-computing.org/staff/katagiri/public_html/nadia.html.
- [70] HINTS Project Home Page. <http://www.hints.org/>.
- [71] Kuroda, H.: ILIB_GMRES ver.0.91 Users Manual (2000).
 This manual is now opened via HINTS Project Home Page as an online manual.
<http://www.hints.org/doc/gmres.html>.
- [72] ILIB_LU Users Manual (2000).
 This manual will be opened via HINTS Project Home Page.
<http://www.hints.org/>.
- [73] Katagiri, T., Kuroda, H. and Kanada, Y.: ILIB_TriRed ver.0.61 Users Manual (2000). This manual will be opened in the HINTS Project Home Page.
<http://www.hints.org/>.
- [74] Tomita, S.: *Parallel Computer Engineering*, Shokodo (1996). in Japanese.
- [75] Tomita, S.: *Computer Architecture I*, Maruzen (1994). in Japanese.
- [76] Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1996).
- [77] Boku, T., Itakura, K., Nakamura, H. and Nakazawa, K.: CP-PACS: A Massively Parallel Processor for Large Scale Scientific Calculations, *Proceedings of International Conference on Supercomputing 97*, Vienna, Austria, pp. 108–115 (1997).

- [78] Lo, S-S., Philippe, B. and Sameh, A.: A Multiprocessor Algorithm for The Symmetric Tridiagonal Eigenvalue Problem, *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 2, pp. s155–s165 (1987).
- [79] Demmel, J. and Stanley, K.: The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers, Technical Report UT-GS-94-254, University of Tennessee, Knoxville (1994).
- [80] Katagiri, T. and Kanada, Y.: A Parallel Implementation of Eigensolver and its Performance, *IPSJ SIG Notes, 97-HPC-69*, pp. 49–54 (1997). in Japanese.
- [81] Reeve, J. and Heath, M.: An Efficient Parallel Version of the Householder-QL Matrix Diagonalization Algorithm, *Parallel Computing*, Vol. 25, pp. 311–319 (1999).
- [82] Murata, K., Natori, R. and Karaki, Y.: *Large Scale Numerical Simulations*, Iwanami Shoten, pp. 157–165 (1990). in Japanese.
- [83] HITACHI: Using ScaLAPACK and PBLAS Libraries for the HITACHI SR2201, *Computer Centre News, the University of Tokyo*, Vol. 30, No. 2, pp. 36–58 (1998). in Japanese.
- [84] Tajima, S., Katagiri, T., Kanada, Y. and Nagashima, U.: Parallel Processing of Molecular Geometry Parameter Optimization by Extended Heuckel Method – An Attempt of Simple Fast Generation of Molecular Structure –, *The Journal of Chemical Software*, Vol. 6, No. 2, pp. 67–74 (2000). in Japanese.
- [85] Naono, K., Igai, M., Yamamoto, Y. and Hirayama, H.: High Performance Implementation of Eigenvalue Computation for the SR8000, *Proceedings of 1999 annual conference of the Japan Society for Industrial and Applied Mathematics(JSIAM)* (1999). in Japanese.
- [86] Higham, N. J.: The Accuracy of Floating Points Summation, *SIAM J. Sci. Comput.*, Vol. 14, No. 4, pp. 783–799 (1993).
- [87] Crandall, R. and Fagin, B.: Discrete Weighted Transforms and Large-Integer Arithmetic, *Mathematics of Computation*, Vol. 62, No. 205, pp. 305–324 (1994).
- [88] Schreiber, T., Otto, P. and Hofmann, F.: A New Efficient Parallelization Strategy for the QR Algorithm, *Parallel Computing*, Vol. 20, pp. 63–75 (1994).

- [89] Katagiri, T. and Kanada, Y.: A Parallel Implementation of Eigensolver and Its Performance, *Ninth SIAM Conference on Parallel Processing for Scientific Computing (PP'99)* (1999). As a poster.
- [90] Å. Björck: Numerics of Gram-Schmidt Orthogonalization, *Linear Algebra and its Applications*, Vol. 197–198, pp. 297–316 (1994).
- [91] Jalby, W. and Philippe, B.: Stability Analysis and Improvement of the Block Gram-Schmidt Algorithm, *SIAM J. Sci. Stat. Comput.*, Vol. 12, No. 5, pp. 1058–1073 (1991).
- [92] Bilmes, J., Asanović, 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).
- [93] Whaley, R. C. and Dongarra, J. J.: Automatically Tuned Linear Algebra Software, ATLAS project, <http://www.netlib.org/atlas/index.html>.
- [94] Brewer, E. A.: Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization, *Ph.D Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology* (1994).
- [95] Brewer, E. A.: High-Level Optimization via Automated Statistical Modeling, *Proceedings of Principles & practice of Parallel Programming (PPoPP)'95*, Santa Barbara, CA, USA, pp. 19–21 (1995).
- [96] 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).
- [97] Kuroda, H. and Kanada, Y.: Performance of Automatically Tuned Parallel Sparse Linear Equations Solver, *IPSJ SIG Notes, 99-HPC-76*, pp. 13–18 (1999).
- [98] Kuroda, H., Katagiri, T., Tsukuda, Y. and Kanada, Y.: Constructing Automatically Tuned Parallel Numerical Calculation Library — A Case of Symmetric Sparse Linear Equations Solver —, *Proc. 57th National Convention IPSJ*, pp. 1–10 – 1–11 (1998).
- [99] Kuroda, H., Katagiri, T. and Kanada, Y.: Performance of Automatically Tuned Parallel GMRES(m) Method on Distributed Memory Machines, *Proceedings of Vector and Parallel Processing (VECPAR) 2000*, Porto, Portugal, pp. 251 – 264 (2000).

- [100] Kalamboukis, T.: The Symmetric tridiagonal eigenvalue problem on a transputer network, *Parallel Computing*, Vol. 15, pp. 101–106 (1990).
- [101] Smith, C., Hendrickson, B. and Jessup, E.: A Parallel Algorithm for Householder Tridiagonalization, *Proc. 5th SIAM Conf. Appl. Linear Alg.*, pp. 361–365 (1994).
- [102] Murata, K.: A New Algorithm to Tridiagonalize Symmetric Band Matrices, *J. IPS Japan*, Vol. 16, No. 2 (1975). in Japanese.

Appendix A

Information of Our Eigensolver

Information of our eigensolver, named ILIB_DRSSSED which contains optimization feature before execution is shown in this appendix.

A.1 Code line of ILIB_DRSSSED

The ILIB_DRSSSED means ILIB routine for Dense Real Symmetric Standard Eigenvalue Decomposition. Table A.1 shows the total code line of ILIB_DRSSSED routines. The total code length of ILIB_DRSSSED is about 20,000 lines. Most of them are unrolled codes for matrix-vector product and updating processes of matrix.

A.2 Results of auto-tuning for ILIB_DRSSSED routine

A.2.1 ILIB_TriRed : a Householder tridiagonalization routine

ILIB_TriRed is a parallel Householder tridiagonalization routine with optimization feature before execution.

A.2.1.1 Tuning information

Figure A.1 shows execution time in 4 PEs of the HITACHI SR2201 with all combination of parameters in the performance evaluation of Chapter 5. The number of combination for the parameters is $2 \times 8 \times 8 = 128$. Table A.2 shows the tuning information of ILIB_TriRed for PE=4, $n = 4000$ in the SR2201.

Figure A.2 also shows execution time in the SR2201, but this is for the case of 1024 PEs. The number of combination for the parameters is $2 \times 8 \times 8 = 128$. Table A.3 shows the tuning information of ILIB_TriRed for PE=1024, $n = 8000$ in the SR2201.

Table A.1: Total code line of ILIB_DRSSSED. (in 14th Jan. 2000)

File name	lines	Details
HouseCom.f	131	Private communication routines.
Houselib.f	58	General usage routines.
ILIB_TriRed.f	622	Parallel Householder tridiagonalization routine.
vec-mat.f	483	Matrix-vector products kernels for the Householder tridiagonalization routine.
update.f	547	kernels to update matrix for the Householder tridiagonalization routine.
Redist.f	356	Data re-distribution routine.
PBisec.f	209	Parallel bisection routine.
PTriInvNoOrt.f	534	Parallel inverse iteration routine. (without re-orthogonalization)
PTriInvMGS.f	727	Parallel inverse iteration routine. (with the MGS re-orthogonalization)
PTriInvCGS.f	755	Parallel inverse iteration routine. (with the CGS re-orthogonalization)
PTriInvIRCGS.f	866	Parallel inverse iteration routine. (with the IRCGS re-orthogonalization)
TriDiagLU.f	192	LU factorization routine for Tridiagonal matrices.
ILIB_HouseInvTrs.f	113	Parallel Householder inverse transformation routine.
HITkernels.f	791	Kernels for Householder inverse transformation routine.
ILIB_PAllMGSblk.f	162	The MGS orthogonalization routine.
MGSblkBkernels.f	5,406	Kernels for the MGS orthogonalization routine (for pivot PEs.)
MGSblkIkernels.f	4,777	Kernels for the MGS orthogonalization routine (for inner PEs.)
ILIB_DRSSSED.f	532	Main routine of Dense Real Symmetric Standard Eigenvalue Decomposition.(DRSSSED)
ILIB_TestDRSSSED.f	434	Test driver of the DRESSSED.
TestSym.f	574	Accuracy test routines of the DRESSSED.
MakeMat.f	229	Test matrices generators.
ILIB_DRESSSEDautotune.f	1,008	Optimization feature before execution routines.
Total codes	19,506	

Table A.2: Tuning information of ILIB_TriRed. (SR2201, PE=4, $n = 4000$)

(a) Optimal and tuned parameters.
(Comm.Type, Mat-Vec, Updating)

Optimal parameter	Tuned parameter
(Tree, 3, 3)	(Tree, 3, 3)

(b) Maximal speed-up factor of each parameters.
(Comm.Type, Mat-Vec, Updating)

Comm.Type	
1.008 (MPI_ALLREDUCE, 3, 2) / (Tree, 3, 2)	
Mat-Vec	Updating
1.44 (Tree, 8, 3)/(Tree, 3, 3)	2.56 (Tree, 3, 16)/(Tree, 3, 3)

Table A.3: Tuning information of ILIB_TriRed. (SR2201, PE=1024, $n = 8000$)

(a) Optimal and tuned parameters.
(Comm.Type, Mat-Vec, Updating)

Optimal parameter	Tuned parameter
(MPI_ALLREDUCE, 8, 5)	(MPI_ALLREDUCE, 8, 5)

(b) Maximal speed-up factor of each parameters.
(Comm.Type, Mat-Vec, Updating)

Comm.Type	
1.48 (Tree, 4, 5) / (MPI_ALLREDUCE, 4, 5)	
Mat-Vec	Updating
1.51 (Tree, None, 5) / (MPI_ALLREDUCE, 8, 5)	1.59 (Tree, 8, 16) / (MPI_ALLREDUCE, 8, None)

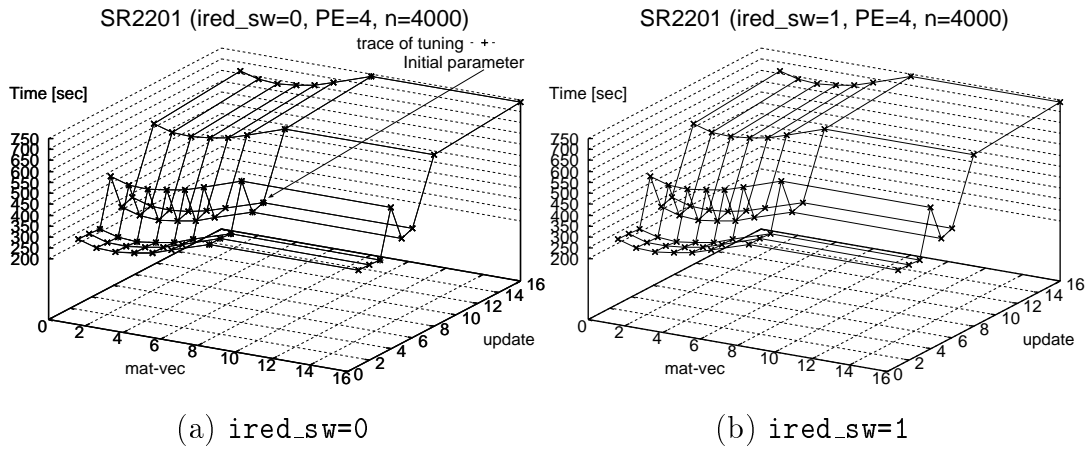


Figure A.1: Execution time of ILIB_TriRed. (SR2201, PE=4, $n = 4000$)

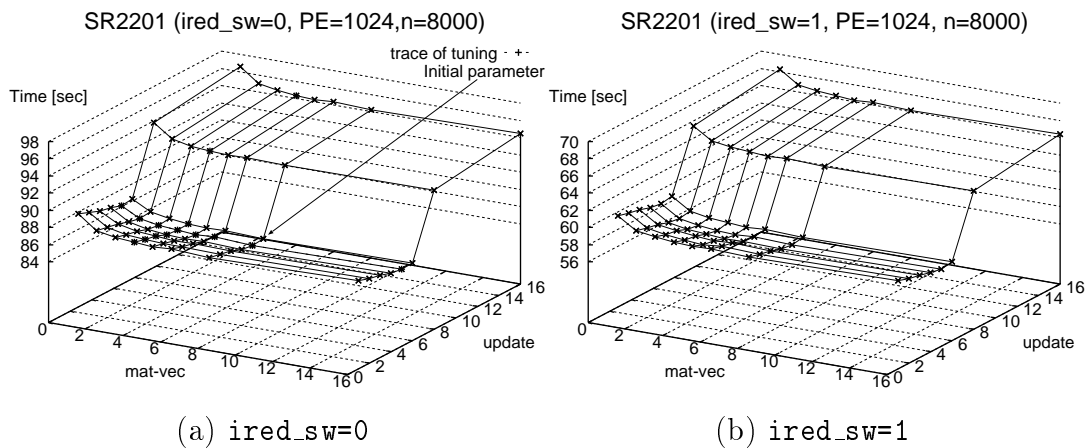
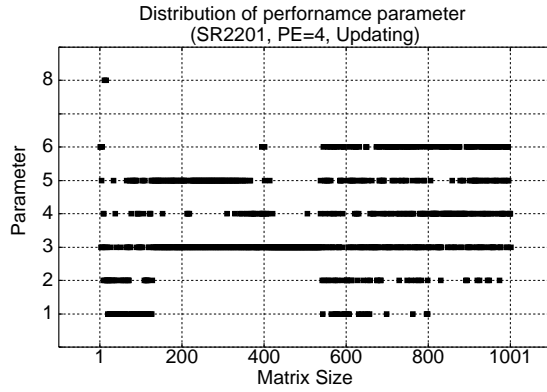


Figure A.2: Execution time of ILIB_TriRed. (SR2201, PE=1024, $n = 8000$)

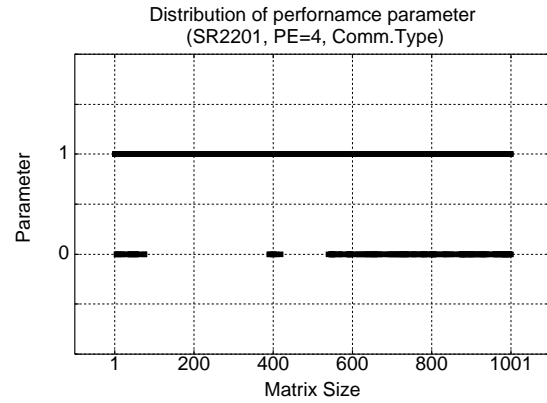
From the results of Figures A.2 and A.3, the ad-hoc optimization method showed in Chapter 5 can find the optimal parameters sets in this case. Therefore specified optimization method is a good for the Householder tridiagonalization. We think the main reason of this is explained as: the routines in the Householder tridiagonalization is physically separated. Therefore each parameter had no dependency. Then we found the optimal parameters set.

A.2.1.2 Distribution of performance parameter

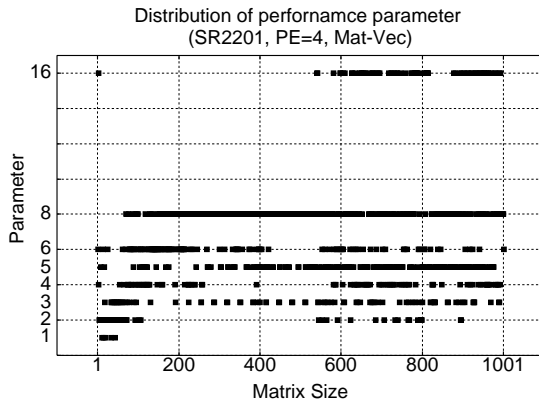
Figure A.3 shows the distribution of the three performance parameters of ILIB_TriRed from the matrix size of 1 to 1001.



(a) Parameter of Update.



(c) Parameter of Comm.Type.



(b) Parameter of Mat-Vec.

Figure A.3: Distribution of performance parameters for ILIB_TriRed. (SR2201, PE=4)

The results of Figure A.3 indicate that there is no tendency of performance parameters according to the matrix size. Therefore, selecting the optimal parameters sets is difficult problem in the ILIB_TriRed routine.

A.2.2 ILIB_MGSAO : a MGS orthogonalization routine

ILIB_MGSAO is a parallel Modified Gram-Schmidt (MGS) routine with optimization feature before execution.

A.2.2.1 Tuning information

Table A.4 shows tuning information in 16 PEs of the HITACHI SR2201 with all combination of parameters in the performance evaluation of Chapter 5. The total number of combination for the parameters is $8 \times 4 \times 8 \times 8 = 8192$.

Table A.4: Tuning information of ILIB_MGSAO. (SR2201, PE=16, $n = 1000$)

(a) Optimal and tuned parameters, and its execution time.

(MGSBL, MGSPib, MGSPj, MGSIjb, MGSIib)

Optimal parameter	Tuned parameter
(6, None, None, 2, 6)	(5, None, None, 2, 5)
Its execution time	Its execution time
1.604 [sec.]	1.668 [sec.]
Speed-down ratio	
between optimal and tuned time	3.9 %

(b) Maximal speed-up factor of each parameters.

(MGSBL, MGSPib, MGSPj, MGSIjb, MGSIib)

MGSBL	
2.17	
(5, None, None, 2, 6) / (6, None, None, 2, 6)	
MGSPib	MGSPj
1.21	1.21
(6, 2, None, None, 16) / (6, None, None, None, 16)	(4, None, 2, None, 16) / (4, None, None, None, 16)
MGSIjb	MGSIib
2.98	2.45
(6, None, None, 3, 6) / (6, None, None, 2, 6)	(8, None, None, 2, 8) / (8, None, None, 2, 6)

The results of Table A.4(a) shows that the ad-hoc approach for optimization in Chapter 5 did not find the optimal parameters set. The reason is that the parameters have a dependency since the routines are physically located in same place. However, a near optimal parameters set was found in this case because the execution time was almost same. The ratio of speed-down between optimal and tuned execution time was only 4%. Therefore the conclusion is the optimization method in Chapter 5 is an appropriate method.

The results of Table A.4(b) indicate that parameters of MGSBL, MGSIjb, and MGSIib can greatly affect the total performance, and parameters of MGSPib and MGSPj can not affect the total performance, since parameters of MGSPib and MGSPj are for pivot PE to determine the orthogonal vectors. The computation complexities are lower than the other computations. Therefore it can be assumed the parameters of MGSPib and MGSPj do not affect total performance.

To show the performance effect of these parameters, parameters of MGSBL, MGSIjb, and MGSIib were checked. Figures A.4 and A.5 show the execution time in the SR2201 with all parameter combination for MGSBL, MGSIjb, and MGSIib. Figures A.4 and A.5 show that there are several stationary points. Therefore, by fixing the parameters of MGSPib and MGSPj, several local optimal points are found. Developing optimization methods by using this fact is part of future work.

A.3 How to obtain the ILIB_DRSSSED routine

The ILIB_DRSSSED routine is developed in the part of HINTS (HIgh-performance Numerical Tools & Software) project at Kanada Laboratory, Computer Centre Division, Information Technology Center, the University of Tokyo. For information of ILIB_DRSSSED routine go to the following HINTS project home page.

<http://www.hints.org/>

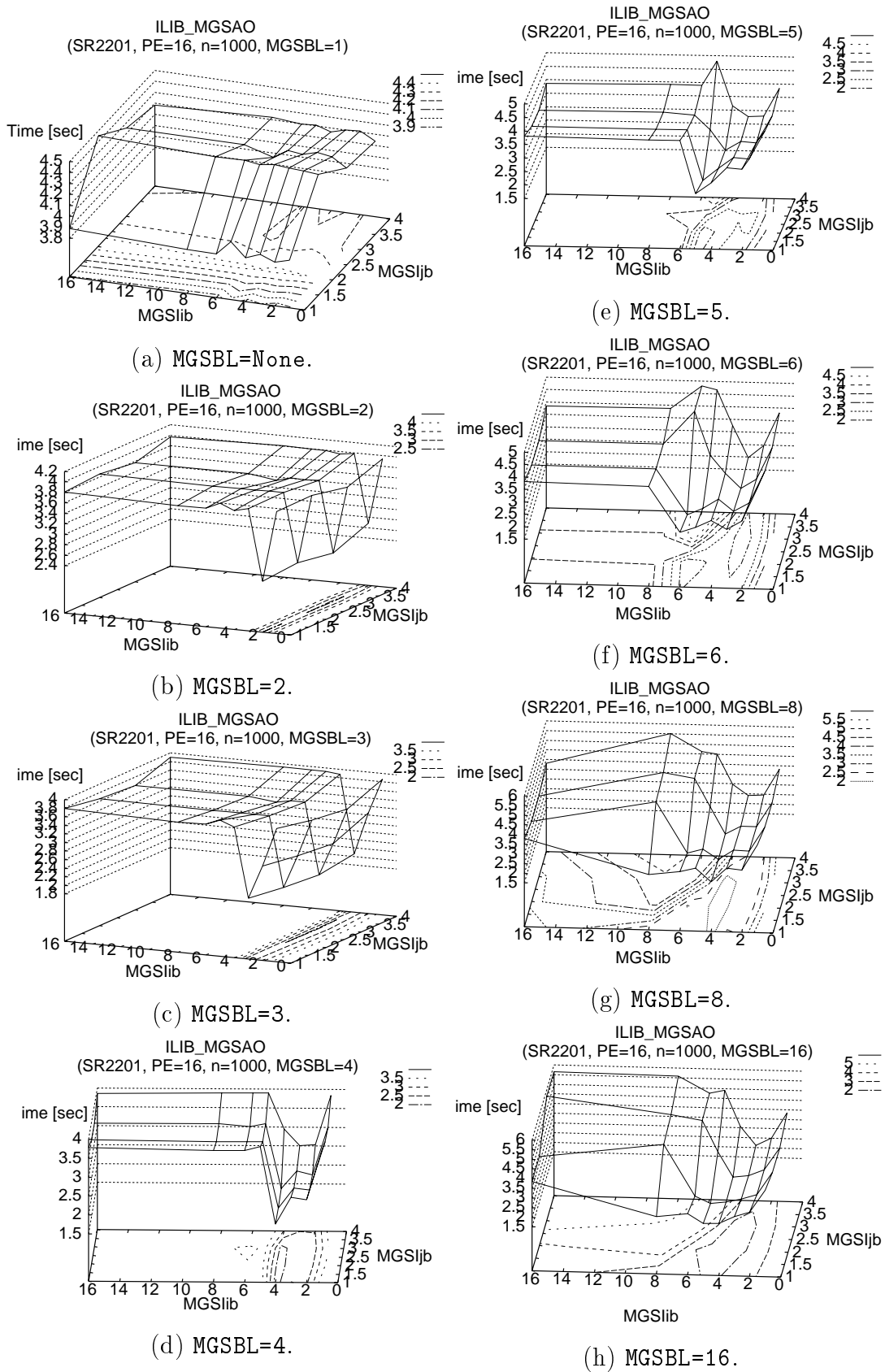


Figure A.4: Execution time of ILIB_MGSAO I. (SR2201, PE=16, $n = 1000$)

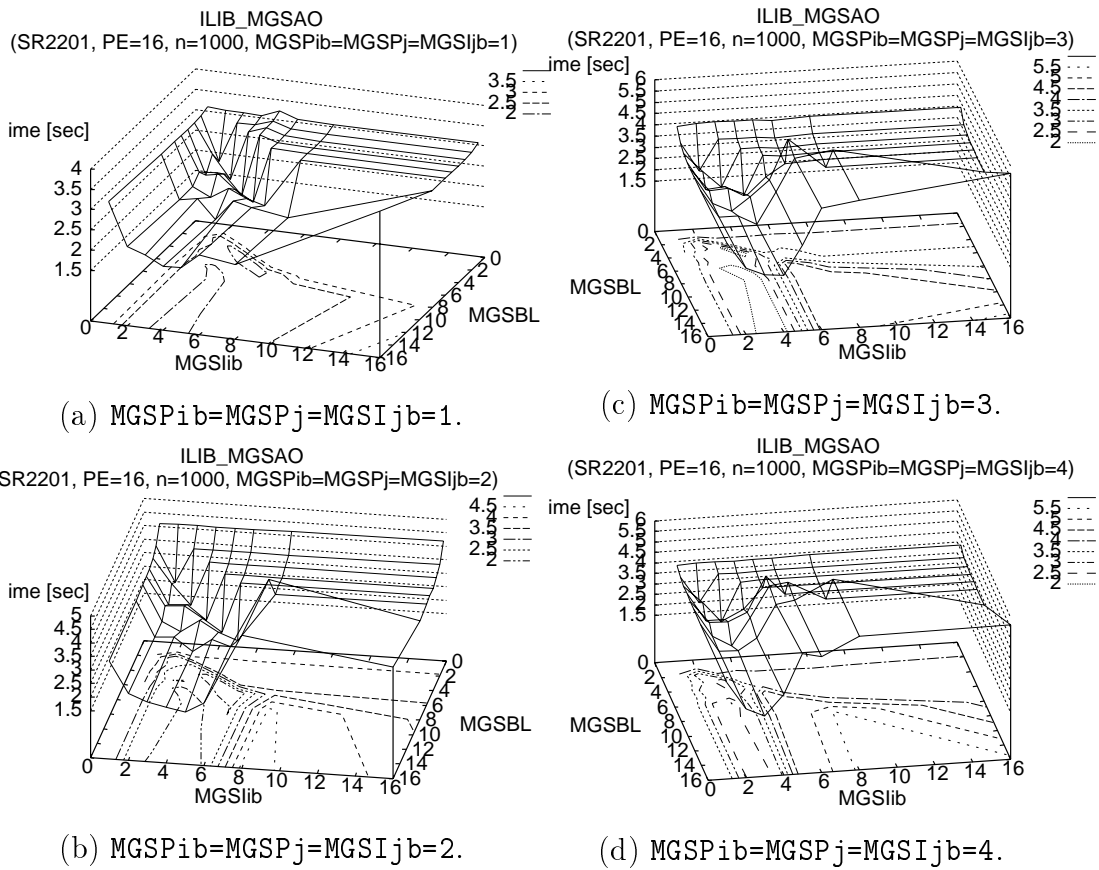


Figure A.5: Execution time of ILIB_MGSAO II. (SR2201, PE=16, $n = 1000$)

Appendix B

ILIB_TriRed Manual: Version 0.61

This manual describes the use of ILIB_TriRed for eigenvalue solver for real symmetric dense standard eigenvalue problem. This library has the following features.

- ILIB_TriRed supports parallel processing.
- ILIB_TriRed includes an auto-tuning methodology.

This library provides several subroutines which can be called by FORTRAN with MPI (Message Passing Interface). We carefully checked our routine by using many test matrices, but just in case bugs are found, please report to us by e-mail. The e-mail address is www-admin@pi.cc.u-tokyo.ac.jp.

B.1 Introduction

This library can be used for solving real symmetric dense standard eigenproblems. The solving of real symmetric dense standard eigenproblem is defined as : Performing eigendecomposition $A = X\Lambda X$, where the A is a coefficient matrix, the X is a matrix contained all eigenvectors as its row vector, and the Λ is a diagonal matrix whose diagonal elements have all eigenvalues. However, this library reduces a matrix A to a tridiagonal matrix T by using the Householder tridiagonalization method. To perform the eigendecomposition, we need an eigenvalue computation of the matrix T by using the bisection method or the other methods, and an eigenvector computation of T by using the inverse iteration method or the other methods. After the eigenvector computation, a re-transformation of the eigenvectors of T is needed by using scalars and vectors in the Householder tridiagonalization. The full eigendecomposition routine will be released in future projects. This library uses MPI (Message Passing Interface). Therefore, this library runs on all parallel computers on which MPI is installed.

B.1.1 Householder tridiagonalization algorithm

In this library, the following well-known and well-used transformation, named “Householder Method”, is used as a symmetrical transformation:

[Theorem]

When a vector $x \in \mathfrak{R}^n$ is given, there are the following vector $u \in \mathfrak{R}^n$ and scalar $\alpha \in \mathfrak{R}$: $(I - \alpha uu^T)x = (v_1, \dots, v_k, \sigma, 0, \dots, 0)^T$, where $\sigma = \|x_{k+1:n}\|_2$.

The vector $u = (0, \dots, 0, v_{k+1} \pm \sigma, v_{k+2}, \dots, v_n)^T$, and scalar $\alpha = 1/(\sigma^2 + |v_{k+1} \pm \sigma|)$ which are a pair of quantities which satisfy the theorem, and $\alpha u^T u = 2$, since $\|u\|_2^2 = (v_{k+1} \pm \sigma)^2 + v_{k+2}^2 + \dots + v_n^2 = \sigma^2 + \sigma^2 + 2|v_{k+1} \pm \sigma| = 2(\sigma^2 + |v_{k+1} \pm \sigma|) = 2/\alpha$. The sign of the scalar σ is same as that of v_{k+1} to minimize catastrophic cancellation of significant digits when calculating the elements of the vector u . We represent the reflection of $(I - \alpha uu^T)x$ by $H^{(k)}(x)$. The reflection does not affect the elements v_1, \dots, v_k . In addition, (u, α) is the pair that is required to perform the above reflection $H^{(k)}(x)$.

Here, we consider the reduction from $A^{(1)} = A$ to $A^{(n-2)} = T$. By applying the $H^{(k)} = I - \alpha uu^T$ to the $k + 1$ -th iteration of A , we can obtain the following fomula:

$$\begin{aligned}
 A^{(k+1)} &= H^{(k)} A^{(k)} H^{(k)} \\
 &= A^{(k)} - \alpha A^{(k)} uu^T - \alpha uu^T A^{(k)} + \alpha^2 uu^T A^{(k)} uu^T \\
 &= A^{(k)} - xu^T - uy^T + \alpha uu^T xu^T \\
 &= A^{(k)} - uy^T + u\mu u^T - xu^T \\
 &= A^{(k)} - u(y^T - \mu u^T) - xu^T,
 \end{aligned} \tag{B.1}$$

where

$$x = \alpha A^{(k)} u, \tag{B.2}$$

$$y^T = \alpha u^T A^{(k)}, \tag{B.3}$$

$$\mu = \alpha u^T x. \tag{B.4}$$

Now A is symmetric matrix, then $x = y$. We can obtain the following:

$$A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T \tag{B.5}$$

Note that we need the column vector $A_{k:n,k}$ on the partial matrix $A_{k:n,k:n}$ in the k -th iteration. We call the column vector $A_{k:n,k}$ as *the row to calculate the pivot vector*, and the vector u as *the pivot vector*.

The Figure B.1 shows the elements of parallel algorithm for the Householder tridiagonalization.

In the Householder tridiagonalization of Figure B.1, the pivot vector is sent between lines 2 – 12, the matrix-vector product ($x = \alpha A^{(k)}u$) is performed between lines 13 – 15, the dot product ($\mu = \alpha u^T x$) is performed between lines 25 – 29, and the updating process ($A^{(k+1)} = A^{(k)} - u(x^T - \mu u^T) - xu^T$) is performed between lines 25 – 29.

B.1.2 Parallelization of the tridiagonalization

For data distribution method of the matrix A , the cyclic-cyclic distribution is used in the library.

The input data is the matrix A in the symmetric standard eigenvalue problem of $Ax = \lambda x$. The matrix A is distributed by the cyclic-cyclic distribution. Therefore, partial data of A is given in each PE.

The cyclic-cyclic distribution is explained as follows: Let `nprocs` be the total number of PEs, and `pe_num` be a one-dimensional identification number of each PE. The `pe_num` can take:

$$\text{pe_num} : 0, 1, \dots, \text{nprocs} - 1.$$

For the `nprocs`, it has the following conditions:

$$\text{nprocs} = \text{ncelx} \times \text{ncely},$$

and

- $\text{ncelx} < \text{ncely}$, where ncelx should not be 1.
- $\text{ncely}/\text{ncelx}$ is dividable.

Next, the transformation from one-dimensional identification number of PE to two-dimensional one is performed by the following program:

```
subroutine cid2xy(pe_num, ncelx, nmyidx, nmyidy)
  integer pe_num, ncelx, nmyidx, nmyidy

  nmyidx = modulo(pe_num, ncelx)
  nmyidy = (pe_num - modulo(pe_num, ncelx))/ncelx
  return
end
```

In this program, the two-dimensional number of $(nmyidx, nmyidy)$ can be obtained.

The distribution of $n \times n$ matrix of A can be implemented by the following programs. (This program is not high performance, but it can be implemented easily.)

```
do i=0,n-1
  li = LOC(i)
  if (nmyidx .eq. OWNER(i)) then
    do j=0,n-1
      if (nmyidy .eq. OWNER(j)) then
        A(li, LOC(j)) = n - i
      endif
    enddo
  endif
enddo
```

Note that the subscript of matrix A is changed from 0 to $n - 1$ in this program. The array $OWNER(i)$ is a list of owner PE number for the i -th column of the matrix A , and the $LOC(i)$ is that of subscription number in local meanings for the i -th column of the matrix A . These arrays are calculated by:

```
do i=0, n-1
  LOC(i) = CYCLIC_LOC(i,ncelx)
  OWNER(i) = CYCLIC_OWNER(i,ncelx)
enddo
```

The functions $CYCLIC_LOC(i,ncelx)$, $CYCLIC_OWNER(i,ncelx)$ are defined as:

```
integer function CYCLIC_OWNER(i, nprocs)
  integer i, nprocs
  CYCLIC_OWNER = modulo(i, nprocs)
  return
end
```

```
integer function CYCLIC_LOC(i, nprocs)
  integer i, nprocs
  CYCLIC_LOC = idfloor(i/nprocs)
  return
end
```

Since the owner information for row and column direction are same in the cyclic-cyclic distribution, we do not need owner information.

For example, (`nprocs` = 4, `n` = 3) $A = (a_{ij})$, ($i, j = 1, 2, 3$)

$$\begin{array}{ccc} \downarrow (i) & \rightarrow & (j) \\ a_{11}, & a_{12}, & a_{13} \\ a_{21}, & a_{22}, & a_{23} \\ a_{31}, & a_{32}, & a_{33} \end{array}$$

`OWNER(i)`, Owner Info. ($nmyidx, nmyidy$)

$$\begin{array}{cccc} 0, & (0,0) & (0,1) & (0,0) \\ 1, & (1,0) & (1,1) & (1,0) \\ 0, & (0,0) & (0,1) & (0,0) \end{array}$$

`LOCAL(i)`, Local Subscript Info. ($local_i, local_j$)

$$\begin{array}{cccc} 0, & (0,0) & (0,0) & (0,1) \\ 0, & (0,0) & (0,0) & (0,1) \\ 1, & (1,0) & (1,0) & (1,1) \end{array}$$

`pe_num`, ($nmyidx, nmyidy$)

$$\begin{array}{cc} 0, & (0,0) \\ 1, & (1,0) \\ 2, & (0,1) \\ 3, & (1,1) \end{array}$$

The owner information shows the two-dimensional identification number of PEs which owns the element a_{ij} . The local subscript information shows the local subscription number of local arrays that a_{ij} should be stored. For example, the 0 PE (0, 0) has the following elements of matrix A by referring to the owner information:

$$a_{11}, a_{31}, a_{13}, a_{33}$$

In addition, the local subscription number by referring to the local subscript information will be as:

$$(0,0), (1,0), (0,1), (1,1)$$

B.2 Automatically tuned parameters

This section explains the auto-tuning facilities in this library.

B.2.1 Loop unrolling for matrix-vector product

Our library has loop unrolled kernels for matrix-vector products. For unrolling, the second loop from the most inner loop is unrolled as its depth of 1,2,...,6,8,16. By measuring the execution time of kernels, the library automatically determine the most suitable depth.

B.2.2 Loop unrolling for updating process

Our library also has loop unrolled kernels for updating process. For unrolling, the second loop from the most inner loop is unrolled as its depth of 1,2,...,6,8,16. By measuring the execution time of kernels, the library automatically determine the most suitable depth.

B.2.3 Communication method

Our library automatically determine the optimal communication method of vector reduction in the tridiagonalization by measuring its execution time. The following is the communication method.

- Method for binary-tree structured communication : This method uses the MPI functions of `MPI_recv` and `MPI_send`.
- Method for using MPI function : This method uses the MPI function of `MPI_ALLREDUCE`.

B.2.4 Information file of automatically tuned parameters

Before calling our library, please make a file for tuned parameters in current directory as “autotuneTRD.dat”.

This file is automatically made by performing auto-tuning process in our library. You can modify the file by hand to specify the arbitrary parameters.

The following is the data format of this file:

```
Matrix size / Communication method (0:binary tree ,1:MPI function) / Unrolling
depth of matrix-vector product (1,2,3,4,5,6,8,16) / Unrolling depth of updating
process (1,2,3,4,5,6,8,16)
```

```
100 1 6 2
200 1 6 3
300 1 4 4
400 1 5 3
```

```

500 1 5 3
600 1 5 3
700 0 8 3
800 1 3 3
900 1 8 3
1000 1 5 4
2000 0 5 5
3000 0 5 5
4000 0 3 3
5000 1 5 5
6000 1 5 5
7000 1 5 5
8000 1 3 2
-1 0 8 6
0 0 8 6
0 0 8 6

```

As the end of the data, please specify -1 as the matrix size. You can not specify more than 20 parameters in this version. If you solve the problem whose problem size is not specified in this file, the library automatically specify the parameters of the smaller problem. For example, when the above file was used and we call our library by using the size of 150, the library specifies the parameter of 100 dimension (1,6,2).

We strongly recommend that users do auto-tuning by using the same size of execution one in the viewpoint of the performance.

B.3 Sample program

```

program main

include 'ILIB_TriRed.h'
common /ILIBpval/ nprocs, myid, ncelx, ncely, nx, ny, ierrcode

c === real*8 definition
c =====

c === for tridiagonalzation
real*8 A(0:ILIB_MAXNX-1, 0:ILIB_MAXNY-1)
real*8 alpha(0:ILIB_MAXN+ILIB_MAXP-1)
real*8 beta(0:ILIB_MAXN+ILIB_MAXP-1)

```

```

real*8  ALP(1:ILIB_MAXN)
real*8  U(1:ILIB_MAXNX,1:ILIB_MAXN)

c      === for eigenvalue computation
real*8  beta2(0:ILIB_MAXN)
real*8  xi(0:ILIB_MAXN)
real*8  xs(0:ILIB_MAXN)
real*8  epsbi0, epsbi

c      === for checking answer
real*8  dtemp
real*8  EigErr

c      === for measuring time
real*8  t1, t2, bt, t_all
real*8  MPI_WTIME

c      =====

c      === integer definition
c      =====

c      === for parallel control
integer  ncelx, ncely, nx, ny
integer  myid, nmyidx, nmyidy
integer  n, nprocs

c      === for error flag
integer  ierrcode

c      === for selecting test matrices
integer  isw_mat

c      === for eigenvalue computation
integer  nc(0:ILIB_MAXN)
integer  nstart, nend, NBI
integer  np, nstart, nend

c      =====

c      --- Start program
c      =====

c      === MPI Init.
c      =====

```

```

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs, ierr )
c =====

c === Print title
c =====
if (myid .eq. 0) then
  print *, ""
  print *, "=== Parallel Eigensolver Check Program ==="
  print *, "Parallel Eigensolver on MPI Ver. 0.61 alpha 7/2000"
  print *, "Composed by KATAGIRI Takahiro"
  print *, ""
endif
c =====

c === Set Test Condition
c =====
if (myid .eq. 0) then
  write(6,*) "problem size >"
  read(5,*) n

  === set matrix type
  isw_mat = 1
endif
c =====

c == Set Parameters
c =====
call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
call MPI_BCAST(isw_mat,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c =====

c === Set Parallel Control Values
c =====
if (myid .eq. 0) then
  write(6,*) " ncelx ="
  read(5,*) ncelx
  write(6,*) " ncely ="
  read(5,*) ncely
endif
call MPI_BCAST(ncelx, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
call MPI_BCAST(ncely, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

```

```

call cid2xy2(myid, ncelx, ncely, nmyidx, nmyidy)
nx = idceiling(dfloat(n)/dfloat(ncelx))
ny = idceiling(dfloat(n)/dfloat(ncely))
c =====

c === Generate Test Matrix
c =====
call MakeTestMat2(A, n, nmyidx, nmyidy, isw_mat)
c =====

c === Initialize ILIB_TriRed routine
c =====
call ILIB_TriRed_Init()
c =====

c === Call Tridiagonalization Routine
c =====
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t1 = MPI_WTIME()

call ILIB_TriRed(A, n, alpha, beta, ALP, U)

call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t2 = MPI_WTIME()

t_all = t2 - t1
call MPI_REDUCE(t_all, bt, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0,
& MPI_COMM_WORLD, ierr)
t_all = bt
c =====

c === Calculate Eigenvalues & Check Its Error
c =====
if (ierrcode .eq. 0) then
  if (myid .eq. 0) print *, "Normal return"
else
  if (myid .eq. 0) print *, "Abnormal return: error code",ierrcode
endif

c === calculate all eigenvalues
NBI = 100
call PBisect(alpha, beta, beta2, xi, xs, nc,
& n, epsbi0, epsbi, nstart, nend, NBI)

```

```

    if (isw_mat .eq. 1) then
      if (myid .eq. 0) print *,
&      "Check of maximal error for eigenvalues...."
      call ChkEigErr(xi, n, EigErr)
      if (myid .eq. 0) print *, "End of checking."
    endif
c    =====

c    === Output Results
c    =====
    if (myid .eq. 0) then
      print *, ""
      print *, "=== Result "
      print *, "===== "
      print *, "n = ", n, " ,Number of processors = ", nprocs
      print *, "Test Matrix:"
      if (isw_mat .eq. 1) print *, "Frank"
      if (isw_mat .eq. 2) print *, "Rnd (-32768 ~ 32765)"
      print *, ""
      if (isw_mat .eq. 1) then
        print *, "Eigenvalue Max Error = ", EigErr
      endif
      write(*, 1000) t_all
1000  format(" Total Calculation Time = ",F10.3," [sec]")
      print *, ""
    endif
c    =====

c    ===== MPI Finalize
c    =====
    call MPI_FINALIZE(ierr)
c    =====

c    =====
    stop
    end

```

B.4 Example of execution

This section explains how to execute the test program of this library.

B.4.0.1 How to execute

For example, when the library is called by the problem size of 100 and 8 PEs, input the following sequence:

```
|> mpirun -n 8 -part ALL ILIB_TestTriRed
```

After the input, the following message will be printed:

```
|=== Parallel Eigensolver Check Program ===  
|Parallel Eigensolver on MPI Ver. 0.61 alpha 7/2000  
|Composed by KATAGIRI Takahiro  
|  
|problem size >  
|?
```

Then, input “100”. Next, the message of

```
|ncelx =  
|?  
|ncely =  
|?
```

is printed. This message shows input mode for processor construction. If you would like to specify the 2×4 processor grid, input “2” and “4”, respectively.

B.4.0.2 Results of execution

If the execution has no problem, the following messages will be printed:

```
|Auto-tuned : 100 1 1 4  
|Tridiagonalization Time = 0.358 [sec]  
|Re-distribution Time = 0.001 [sec]  
|Normal return  
|Check of maximal error for eigenvalues....  
|End of checking.  
|  
|=== Result  
|=====  
|n = 100 ,Number of processors = 8
```

```

|Test Matrix:
|Frank
|
|Eigenvalue Max Error = 0.275628866364506343E-012
|Total Calculation Time = 0.372 [sec]

```

B.5 Input output formats

B.5.1 ILIB_TriRed

ILIB_TriRed(A, n, alpha, beta, ALP, U)

A (Input): The elements of matrix A .

n (Input): The matrix dimension.

alpha (Output): The diagonal elements of tridiagonal matrix.

beta (Output): The sub-diagonal elements of tridiagonal matrix.

ALP (Output): The scalars α_k of Householder transformation:

$$H^{(k)} = I - \alpha_k u_k u_k^T, (\alpha_k, k = 1, \dots, n - 2).$$

U (Output): The vectors u_k of Householder transformation:

$$H^{(k)} = I - \alpha_k u_k u_k^T, (u_k, k = 1, \dots, n - 2).$$

B.5.2 Run-time option

```

-autotune      : Specify auto-tuning parameter.
  |0|no|No|NO| : No auto-tuning.
  |1|yes|Yes|YES| : Do auto-tuning.
-starttunesize : Specify matrix size for starting the auto-tuning.
  Number      : The matrix size.
-maxtunesize   : Specify matrix size for ending the auto-tuning.
  Number      : The matrix size.
-print         : Show execution time.
  0|no|No|NO   : No execution time of each routine is printed.
  1|yes|Yes|YES : Execution time of each routine is printed.
                The unit is second.

```

Please make a file for the above options as “.ilib_trired”. The file can contain more than 80 characters. If the library can not find the .ilib_trired file, or wrong options are specified, the library automatically selects the following default options :

```
-autotune no -starttunesize 100 -maxtunesize 8000 -print yes
```

B.5.3 Other routines

B.5.3.1 ILIB_TriRed_Init()

ILIB_TriRed_Init()

The ILIB_TriRed_Init() is an initialize routine for our library. Please call this routine at first whenever you call our library. The routine reads the run-time options. Please make a file for the run-time option as “.ilib_trired”.

B.5.3.2 cid2xy2

cid2xy2(myid, ncelx, ncely, nmyidx, nmyidy)

myid (Input) : Processor identification number.

ncelx (Input) : Number of processors for x-direction.

ncely (Input) : Number of processors for y-direction.

nmyidx (Output) : Processor identification number for x-direction.

nmyidy (Output) : Processor identification number for y-direction.

This routine is for calculation of two-dimensional processor identification number ($nmyidx, nmyidy$) by using $myid$.

B.5.3.3 MakeTestMat2

MakeTestMat2(A, n, nmyidx, nmyidy, isw_mat)

A (Output) : The elements of matrix.

n (Input) : The dimension of matrix.

nmyidx (Input) : Processor identification number for x-direction.

nmyidy (Input) : Processor identification number for y-direction.

isw_mat (Input) : Kinds of generated matrix. (1:Frank matrix, 2:randomized matrix)

This routine is for generating test matrix. The matrix A is generated with cyclic-cyclic distribution by calling this routine.

B.5.3.4 PBisect

PBisect(alpha, beta, beta2, xi, xs, nc, n, epsbi0, epsbi, nstart, nend, NBI)

alpha (Input) : The diagonal elements of tridiagonal matrix.

beta (Input) : The sub-diagonal elements of tridiagonal matrix.

beta2 (Output) : The squared sub-diagonal elements of tridiagonal matrix.

xi (Output) : The lower bound of eigenvalue λ_i , ($i = \text{nstart}, \dots, \text{nend}$). But this array has **xi(1), ..., xi(n)** elements.

xs (Output) : The upper bound of eigenvalue λ_i , ($i = \text{nstart}, \dots, \text{nend}$). But this array has **xs(1), ..., xs(n)** elements.

nc (Output) : The group number of clustered eigenvalues.

n (Input) : The matrix dimension.

epsbi0 (Output): Machine epsilon.

epsbi (Output) : The distance for determining clustered eigenvalue.

nstart (Output): The number of the smallest eigenvalue in this PE. (Global meanings)

nend (Output) : The number of the biggest eigenvalue in this PE. (Global meanings)

NBI (Input) : The number of iterations for the separated eigenvalues. If you need high accuracy for eigenvalues, please specify big values. However, the computation time will be increased.

This routine is for computing all eigenvalues by using the bisection method (parallel version). The calculated lower bounds of eigenvalues are stored in **xi**, upper bounds are stored in **xs**. The calculated eigenvalues are stored in each PE in distributed forms. For example, they are stored in **xi(nstart) – xi(nend)** in each PE.

B.5.3.5 ChkEigErr

ChkEigErr(xi, n, EigErr)

xi (Input) : Calculated eigenvalues.

n (Input) : The matrix dimension.

EigErr (Output) : The maximal error of eigenvalues.

This routine is for checking maximal error of eigenvalues of the Frank matrix.

```

c  Pmyidx,myidy owns row set  $\Pi$  and column set  $\Gamma$  of  $n \times n$  matrix  $A$ .
1:  do  $k = 1, n - 2$ 
2:    if ( $k \in \Gamma$ ) then
3:      Broadcast( $A_{\Pi,k}^{(k)}$ ) to PEs sharing rows  $\Pi$ .
4:    else
5:      receive( $A_{\Pi,k}^{(k)}$ )
6:    endif
7:    Computation of  $(u_{\Pi}, \alpha)$ .
8:    if (I have diagonal elements of  $A$ ) then
9:      Broadcast( $u_{\Pi}$ ) to PEs sharing columns  $\Gamma$ .
10:   else
11:     receive( $u_{\Gamma}$ )
12:   endif
13:   do  $j = k, n$ 
14:     if ( $j \in \Gamma$ )  $x_{\Pi} = x_{\Pi} + \alpha A_{\Pi,j}^{(k)} u_j$    endif
15:   enddo
16:   Global summation of  $x_{\Pi}$  to PEs sharing rows  $\Pi$ .
17:   if (I have diagonal elements of  $A$ ) then
18:     Broadcast( $x_{\Pi}$ ) to PEs sharing columns  $\Gamma$ .
19:   else
20:     receive( $x_{\Gamma}$ )
21:   endif
22:   do  $j = k, n$ 
23:      $\mu = \alpha u_{\Pi}^T x_{\Pi}$    enddo
24:   Global summation of  $\mu$  to PEs sharing rows  $\Pi$ .
25:   do  $j = k, n$ 
26:     do  $i = k, n$ 
27:       if ( $i \in \Pi$  .and.  $j \in \Gamma$ ) then
28:          $A_{ij}^{(k+1)} = A_{ij}^{(k)} - u_i(y_j^T - \mu u_j^T) - y_i u_j^T$ 
29:       endif enddo enddo
c  Remove  $k$  from active columns and rows.
30:   if ( $k \in \Gamma$ )  $\Gamma = \Gamma - \{k\}$    endif
31:   if ( $k \in \Pi$ )  $\Pi = \Pi - \{k\}$    endif
32: enddo

```

Figure B.1: The Householder tridiagonalization algorithm in the cyclic-cyclic distribution.

Appendix C

Bibliography: List of Publications by the Author

C.1 Papers with referee

C.1.1 Paper (Journal)

1. KATAGIRI Takahiro and KANADA Yasumasa: Performance Evaluation of Blocked Householder Algorithm on Distributed Memory Parallel Machine, *Trans. IPS. Japan*, Vol.39, No.7, pp.2391–2394 (1998), in Japanese.
2. KATAGIRI Takahiro and KANADA Yasumasa: An Unblocked Hessenberg Reduction Algorithm for Distributed Memory Architecture Parallel Machines and Its Effectiveness, *Trans. IPS. Japan*, Vol.39, No.11, pp.3065–3075 (1998), in Japanese.
3. KATAGIRI Takahiro and KANADA Yasumasa: An Efficient Implementations of Parallel Eigenvalue Computation for Massively Parallel Processing, *Trans. IPS. Japan*, Vol.41, No.5, pp. 1558–1566 (2000), in Japanese.

C.1.2 Papers (Symposium)

4. KATAGIRI Takahiro and KANADA Yasumasa: An Unblocked Hessenberg Reduction Algorithm for Distributed Memory Architecture Parallel Machine and its Evaluation, *IPS. Japan, Proceedings of JSPP (Joint Symposium on Parallel Processing)'97*, pp.385–392 (1997), in Japanese.
5. KATAGIRI Takahiro and KANADA Yasumasa: A Parallel Implementation of Eigensolver and an Improvement of Its Parallelism, *IPS. Japan, Proceedings of JSPP (Joint Symposium on Parallel Processing)'98*, pp.223–230 (1998), in Japanese.

6. KATAGIRI Takahiro and KANADA Yasumasa: A Parallel Implementation of Eigensolver and Its Performance, *Ninth SIAM Conference on Parallel Processing for Scientific Computing* (San Antonio, TX, USA, 21 Mar. 1999), short paper, as a poster.
7. KATAGIRI Takahiro, KURODA Hisayasu and KANADA Yasumasa: A Methodology for Automatically Tuned Parallel Tridiagonalization on Distributed Memory Vector-Parallel Machines, *Proceedings of Vector and Parallel Processing 2000*, pp.265 – 277, (Faculdade de Engenharia da Universidade do Porto, Portugal, 21st – 23rd June, 2000).
8. KATAGIRI Takahiro, KURODA Hisayasu, OHSAWA Kiyoshi and KANADA Yasumasa: I-LIB : An Automatically Tuned Parallel Numerical Library and Its Performance Evaluation, *IPS. Japan, Proceedings of JSPP (Joint Symposium on Parallel Processing)'2000*, pp.27–34 (2000), in Japanese.

C.2 Papers without referee

C.2.1 IPS.Japan SIG Notes

9. KATAGIRI Takahiro and KANADA Yasumasa: Performance Evaluation of Householder Method for the Eigenvalue Problem on Distributed Memory Architecture Parallel Machine, *IPSJ SIG Notes 96-HPC-62*, pp.111–116 (1996), in Japanese.
10. KATAGIRI Takahiro and KANADA Yasumasa: Performance Evaluation of Blocked Householder Algorithm for the Eigenvalue Problem on Distributed Memory Architecture Parallel Machine, *IPSJ SIG Notes 97-ARC-123*, pp.13–18 (1997), in Japanese.
11. KATAGIRI Takahiro and KANADA Yasumasa: A Parallel Implementation of Eigensolver and its Performance, *IPSJ SIG Notes 97-HPC-69*, pp.49–54 (1997), in Japanese.
12. KATAGIRI Takahiro and KANADA Yasumasa: CGSS: A New Gram-Schmidt Orthogonalization Method with Sorting, *IPSJ SIG Notes 99-HPC-76*, pp.37–42 (1999), in Japanese.

C.2.2 Other reports

13. KATAGIRI Takahiro and KANADA Yasumasa: A Parallelization of Householder Method on Distributed Memory Machines, *Computer Centre News*, the University of Tokyo, Vol.28,No.6,pp.85–95 (1996), in Japanese.

14. KATAGIRI Takahiro and KANADA Yasumasa: An Unblocked Hessenberg Reduction Algorithm for Distributed Memory Machines and Its Evaluation, *Computer Centre News*, the University of Tokyo, Vol.29, No.3, pp.69–82 (1996), in Japanese.
15. KATAGIRI Takahiro and KANADA Yasumasa: Implementation of Parallel Eigensolver and Its Performance, *Computer Centre News*, the University of Tokyo, Vol.29, No.6, pp.27–40 (1997), in Japanese.
16. KATAGIRI Takahiro and KANADA Yasumasa: Implementation of Parallel Large Scale Dense Eigensolver on Distributed Memory Parallel Machine and Its Performances — Viewpoint of processing for badly separated eigenvalues —, *Computer Centre News*, the University of Tokyo, Vol.30, No.4, pp.29–46 (1998), in Japanese.
17. KATAGIRI Takahiro, KURODA Hisayasu and KANADA Yasumasa: Development of Automatically Tuned Parallel Numerical Libraries — Applications of Techniques for Knowledge Discovery Science —, “*Discovery Science*” *Reports of Team A05*, pp. 112–119, (Hokkaido, Hokkaido University, August 9th and 10th 1999), in Japanese.
18. KATAGIRI Takahiro, KURODA Hisayasu, OHSAWA Kiyoshi and KANADA Yasumasa: Performance of Automatically Tuned Parallel Numerical Libraries I-LIB as Linear Equation Solver — Knowledge Discovery by using tuned parameters —, “*Discovery Science*” *Reports of Team A05*, pp. 10–19, (Ehime, Matsuyama University, August 5th and 6th 2000), in Japanese.

C.3 Co-authored papers

19. KURODA Hisayasu, KATAGIRI Takahiro, TSUKUDA Yoshio, and KANADA Yasumasa: Constructing Automatically Tuned Parallel Numerical Calculation Libaray — A Case of Symmetric Sparse Linear Equation Solvers —, *Proceedings of 57th National Convention IPSJ*, Vol.1, pp.1-10 – 1-11 (1998), in Japanese.
20. KURODA Hisayasu, KATAGIRI Takahiro, TSUKUDA Yoshio and KANADA Yasumasa: A Discussion of Parallel Numerical Libraries Using Knowledge Discovery in Databases — A Case of Sparse Symmetric Liner Equation Solvers —, “*Discovery Science*” *Reports of Team A05*, pp.84 – 89, (Tokyo, Tokyo university of science, October 2nd and 3rd 1998)
21. KURODA Hisayasu, KATAGIRI Takahiro and KANADA Yasumasa: Performance of Automatically Tuned Parallel GMRES(m) Method on Memory Machines, “*Dis-*

covery Science” *Reports of Team A05*, pp. 11–19, (Tokyo, the University of Tokyo, December 5th 1999)

22. TAJIMA Sumie, KATAGIRI Takahiro, KANADA Yasumasa and NAGASHIMA Unpei: Parallel Processing of Molecular Geometry Parameter Optimization by Extended Huckel Method — An Attempt of Simple Fast Generation of Molecular Structure —, *The Journal of Chemical Software*, Vol.6, No.2 (2000), in Japanese.
23. OHSAWA Kiyoshi, KATAGIRI Takahiro, KURODA Hisayasu, and KANADA Yasumasa: ILIB_RLU: An Automatically Tuned Parallel Dense LU Factorization Routine and Its Performance Evaluation, *IPSJ SIG Notes 2000-HPC-82*, pp.25–30 (2000), in Japanese.
24. KURODA Hisayasu, KATAGIRI Takahiro and KANADA Yasumasa: Performance Evaluation of Linear Equations Library on Parallel Computers, *IPSJ SIG Notes 2000-HPC-82*, pp. 35–40 (2000), in Japanese.
25. KURODA Hisayasu, KATAGIRI Takahiro and KANADA Yasumasa: Performance of Automatically Tuned Parallel GMRES(m) Method on Distributed Memory Machines, *Proceedings of VecPar2000*, pp.251– 264, (Faculdade de Engenharia da Universidade do Porto, Portugal, June, 21 – 23, 2000).
26. OHSAWA Kiyoshi, KATAGIRI Takahiro, KURODA Hisayasu and KANADA Yasumasa: Performance Evaluation of a Parallel Automatically Tuned Sparse Direct Solver — Knowledge Discovery from tuning information —, *”Discovery Science” Reports of Team A04 and A05*, pp. 129–136, (Osaka, October 28th 2000), in Japanese.