Automatic Code Selection for the Dense Symmetric Generalized Eigenvalue Problem Using ATMathCoreLib

Masato Kobayashi[1], Shuhei Kudo[1], Takeo Hoshi[2], and Yusaku Yamamoto[1](✉)

[1] The University of Electro -Communications, Tokyo 182-8585, Japan
yusaku.yamamoto@uec.ac.jp

[2] Tottori University, Tottori 680-8552, Japan

Abstract. Solution of the symmetric definite generalized eigenvalue problem (GEP) $A\mathbf{x} = \lambda B\mathbf{x}$ lies at the heart of many scientific computations like electronic structure calculations. The standard algorithm for this problem consists of two parts, namely, reduction of the GEP to the symmetric eigenvalue problem (SEP) and the solution of the SEP. Several algorithms and codes exist for both of these parts, and their execution times differ considerably depending on the input matrix size and the computational environment. So, there is a strong need to choose the best combination of codes automatically given these conditions. In this paper, we propose such a methodology based on ATMathCoreLib, which is a library to assist automatic performance tuning. Numerical experiments using performance data on the K computer, Fujitsu FX10 and SGI Altix show that our methodology is robust and can choose the fastest codes even in the presence of large fluctuations in the execution time.

Keywords: automatic code selection · automatic performance tuning · ATMathCoreLib · generalized eigenvalue problem · parallel computing · ScaLAPACK · ELPA · EigenExa · performance prediction

1 Introduction

Suppose that there are M computer programs that can perform a given task. Their functions are all equivalent, but their execution times may be different and may vary depending on the input problem size, the computing environment and random factors such as influence from other programs running on the same machine. Suppose also that we want to perform the task N $(\geq M)$ times using the same computing environment, using different inputs of the same size, and minimize the total execution time. If we have no prior knowledge on the execution time of each program, a possible strategy is to use each of the M programs once for the first M executions, choose the fastest one, and use it for the remaining $N - M$ executions. But the execution time may fluctuate due to random factors and therefore the estimations from the first M executions may not be accurate. Then, what is the best strategy?

More specifically, let the execution time of the mth program be denoted by $T(m, \mathbf{n}, \mathbf{p}, \mathbf{z})$, where $\mathbf{n}$, $\mathbf{p}$ are parameters that specify the input problem size and the computing environment, respectively, and $\mathbf{z}$ denotes the random factor. Note that $\mathbf{n}$, $\mathbf{p}$ and $\mathbf{z}$ are in general vector variables. For example, in the case of eigenvalue computation, $\mathbf{n}$ consists of the matrix size and the number of eigenvalues to be computed. The parameter $\mathbf{p}$ might consist of integers specifying the target machine and the number of processors to be used. Then, our objective is to choose the sequence $m_1, m_2, \ldots, m_N$ $(1 \leq m_i \leq M)$ judiciously to minimize the expected value $\mathbb{E}[\sum_{i=1}^{N} T(m_i, \mathbf{n}, \mathbf{p}, \mathbf{z}_i)]$, given $\mathbf{n}$, $\mathbf{p}$ and some assumptions on the probability distribution of $\{\mathbf{z}_i\}$. Note that m_i may depend on the already measured execution times, $\{T(m_j, \mathbf{n}, \mathbf{p}, \mathbf{z}_j)\}_{j=1}^{i-1}$. This problem is known as *online automatic tuning* [1].

There are two criteria in choosing $m_1, m_2, \ldots, m_N$. On one hand, we need to estimate the mean execution time $\mathbb{E}[T(m, \mathbf{n}, \mathbf{p}, \mathbf{z})]$ for each m accurately to find the fastest program. In general, the accuracy is improved as the number of measurement for each m is increased. On the other hand, we want to exploit the knowledge obtained by previous measurements as much as possible, by maximizing the use of the program estimated to be the fastest. These two objectives are conflicting, so there is a tradeoff between *exploration* and *exploitation.*

To solve this problem, Suda developed ATMathCoreLib [2], which is a library to assist online automatic tuning. It constructs a statistical execution time model for each of the M programs and chooses the one to be executed next time by considering the tradeoff between exploration and exploitation. After execution, it receives the actual execution time and updates the model using Bayes' rule. This process is repeated N times. In this way, the total execution time is minimized in the sense of expected value.

In this paper, we apply ATMathCoreLib to automatic code selection for the dense symmetric generalized eigenvalue problem (GEP) $A\mathbf{x} = \lambda B\mathbf{x}$, where $A, B \in \mathbb{R}^{n \times n}$ are symmetric and B is positive definite. For this problem, the standard procedure is to transform it to the standard symmetric eigenvalue problem and then solve the latter [3]. There are several algorithms both for the first and second parts and several implementations exist, such as ScaLAPACK [4], ELPA [5,6] and EigenExa [7,8]. Which one is the fastest depends on the problem size n and the computational environment. Since the dense symmetric GEP lies at the heart of many scientific computations and it requires long computing time, it is desirable to be able to choose the best code for a given condition automatically. As computing environments, we consider the K computer, Fujitsu FX10 and SGI Altix. In our experiments, we add artificial noise corresponding to $\mathbf{z}$ to measured data given in [9] and study if ATMathCoreLib can find the optimal code for each case even in the presence of noise.

The rest of this paper is structured as follows. In Sect. 2, we detail the operation of ATMathCoreLib. Section 3 explains algorithms for the dense symmetric GEP and their implementations. In Sect. 4, we apply ATMathCoreLib to the dense symmetric GEP and give experimental results in several computing environments. Finally, Sect. 5 gives some conclusion.

2 Operation of ATMathCoreLib

The operation of ATMathCoreLib is illustrated in Fig. 1 [10]. Here, we assume that there is a master program that executes one of the M equivalent codes depending on the code selection parameter k. The master program also measures the execution time of the code. ATMathCoreLib works interactively with this master program. At the ith iteration ($1 \leq i \leq N$), it selects the code to be executed in such a way that the expected value of the total execution time is minimized. To achieve this, it uses its internal execution time model, which holds the estimates of the mean and variance of the execution time of each code. A code is more likely to be selected if its mean is smaller (faster code) or its variance is larger (meaning that the model for the code is not yet accurate enough). This corresponds to choosing the code to be executed by considering the tradeoff between exploration and exploitation. Then it passes the code number k_i to the master program. The master program receives it, executes the k_i-th code, measures its execution time, and passes it to ATMathCoreLib. Then, ATMathCoreLib uses it to update its internal model. This process is repeated for $i = 1, 2, \ldots, N$.

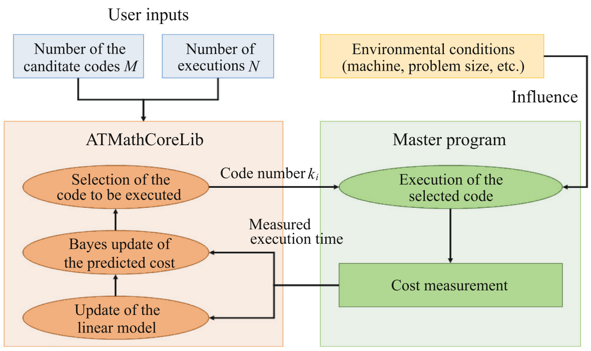
User inputs
Number of the canditate codes M
Number of executions N
Environmental conditions (machine, problem size, etc.)
Influence
ATMathCoreLib
Master program
Selection of the code to be executed
Code number k_i
Execution of the selected code
Measured execution time
Bayes update of the predicted cost
Cost measurement
Update of the linear model

Fig. 1. Operation of ATMathCoreLib.

Actually, the model update process in ATMathCoreLib is more intricate; it consists of two steps called update of the coefficients of the linear model and Bayes update. But we do not go into details here. Readers interested in mathematical foundations of ATMathCoreLib should consult [1,2].

3 Algorithms for the Dense Symmetric GEP and Their Implementations

Here, we consider computing all the eigenvalues and eigenvectors of a dense symmetric GEP $A\mathbf{x} = \lambda B\mathbf{x}$. The standard procedure to solve this problem consists of the following two parts:

1. Reduction of the GEP to a standard symmetric eigenvalue problem (SEP).
2. Solution of the SEP.

There are several algorithms for both of them. For the first part, the standard method is to use the Cholesky decomposition of B. In that case, the whole computation proceeds as follows:

(i) Compute the Cholesky decomposition $B = LL^{\top}$.
(ii) $C \equiv L^{-1}AL^{-\top}$.
(iii) Solve the SEP $C\mathbf{y} = \lambda\mathbf{y}$ and obtain the eigenvalues $\{\lambda_j\}_{j=1}^{n}$ and the eigenvectors $\{\mathbf{y}_j\}_{j=1}^{n}$.
(iv) $\mathbf{x}_j \equiv L^{-\top}\mathbf{y}_j$ for $j = 1, 2, \ldots, n$.

Here, steps (i), (ii) and (iv) correspond to part 1 and step (iii) corresponds to part 2 above. There are two options in computing steps (ii) and (iv). The first one is to use forward and backward substitutions to multiply L^{-1} or $L^{-\top}$. This approach is adopted by ScaLAPACK. The second one is to compute the inverse matrix L^{-1} explicitly and compute steps (ii) and (iv) by matrix multiplications. This approach has the advantage that the number of forward and backward substitutions, which have limited parallelism, is minimized and is adopted by ELPA.

Another method for reducing the GEP to SEP is to use the eigendecomposition of B. In this case, the computation proceeds as follows.

(i) Compute the eigendecomposition $B = WDW^{\top}$, where D is a diagonal matrix and W is an orthogonal matrix.
(ii) $C \equiv D^{-\frac{1}{2}}W^{\top}AWD^{-\frac{1}{2}}$.
(iii) Solve the SEP $C\mathbf{y} = \lambda\mathbf{y}$ and obtain the eigenvalues $\{\lambda_j\}_{j=1}^{n}$ and the eigenvectors $\{\mathbf{y}_j\}_{j=1}^{n}$.
(iv) $\mathbf{x}_j = D^{\frac{1}{2}}W^{\top}\mathbf{y}_j$ for $j = 1, 2, \ldots, n$.

This method has the advantage that the same SEP solver can be used both for steps (i) and (iii). It is used in EigenExa.

In the solution of the SEP, the matrix C is transformed to an intermediate symmetric tridiagonal matrix T or a penta-diagonal matrix P by orthogonal transformations, the eigenvalues and eigenvectors of T or P are computed, and the eigenvectors are transformed to those of C by back-transformation. There are several approaches to achieve this, as listed below.

(A) C is transformed directly to a symmetric tridiagonal matrix T by the Householder method. The eigenvalues and eigenvectors of T are computed by standard methods like the QR algorithm, the divide-and-conquer algorithm, or the MR^3 (Multiple Relatively Robust Representations) algorithm.
(B) C is first transformed to a symmetric band matrix S and then to a symmetric tridiagonal matrix T. The eigenvalues and eigenvectors of T are computed by the standard methods.
(C) C is transformed directly to a symmetric penta-diagonal matrix P. The eigenvalues and eigenvectors of P are computed by a specially designed divide-and-conquer method.

Approach (A) is a conventional one and is adopted by ScaLAPACK. In ELPA and EigenExa, there are also routines using this approach. We denote them as ELPA1 and EIGS, respectively. While this approach is the most efficient in terms of computational work, it has disadvantages that many inter-processor communications are incurred in the tridiagonalization step and that matrix-vector multiplications (DGEMV [11]) used in the tridiagonalization cannot use cache memory effectively. In contrast, approach (B) requires less inter-processor communications. Also, since most of the computations in the tridiagonalization can be done in the form of matrix-matrix multiplications (DGEMM [12]), cache memory can be used effectively. This approach is used by one of ELPA's routine, which we call ELPA2. Approach (C) is an intermediate approach between (A) and (B) and is used in one of the routines in EigenExa. We call this EIGX.

In summary, there are three routines we can use for reducing the GEP to SEP, namely, those from ScaLAPACK, ELPA and EigenExa. Also, there are five routines to solve the SEP, namely, ScaLAPACK, ELPA1, ELPA2, EIGS and EIGX. While ScaLAPACK, ELPA and EigenExa have different matrix storage formats and data distribution schemes, there is a middleware called EigenKernel [13] that allows the user to freely combine routines from these libraries, by providing automatic data conversion and re-distribution functions. Using EigenKernel, we can evaluate the performance of various combinations and choose the fastest one for a given matrix size and computational environment.

4 Automatic Code Selection for the Dense Symmetric GEP Using ATMathCoreLib

Now we apply ATMathCoreLib to automatic code selection for the dense symmetric GEP and evaluate its performance. To this end, we use execution time data on three distributed-memory parallel computers, namely, the K computer, Fujitsu FX10 and SGI Altix ICE 8400EX, given in [9]. We add artificial random noise to these data and study if ATMathCoreLib can choose the optimal combination in the presence of error.

Among $3 \times 5 = 15$ possible combinations of the algorithms for reduction to the SEP and solution of the SEP, 8 promising combinations (workflows) are chosen as candidates in [9]. They are shown in Table 1. The specifications of the parallel computers are listed in Table 2. The size of test matrices is $n = 90,000$ and $n = 430,080$. They are matrices from the ELSES matrix library, which is a collection of matrix data from electronic structure calculations.

From the many test cases reported in [9], we picked up three cases for our evaluation: the problem of $n = 430,080$ on the K computer, $n = 90,000$ on SGI Altix and Fujitsu FX10. The number of nodes used is $p = 10,000$ and 256 for the K computer and SGI Altix, respectively. For Fujitsu FX10, p is either 1,024 or 1,369, depending on the workflow. This is because some library puts restrictions on the number of nodes that can be used. The total execution times for the three cases are shown in Table 3. Here, workflow D' is the same as workflow D except

Table 1. Combinations of the algorithms used in [9].

Workflow	Solution of SEP	Reduction to SEP
A	ScaLAPACK	ScaLAPACK
B	EIGX	ScaLAPACK
C	ScaLAPACK	ELPA
D	ELPA2	ELPA
E	ELPA1	ELPA
F	EIGS	ELPA
G	EIGX	ELPA
H	EIGX	EigenExa

Table 2. Specifications of the parallel computers.

Name	CPU	Clock	# of cores	Byte/Flop
K computer	SPARC 64 VIIIfx	2.0 GHz	8	0.5
Fujitsu FX10	SPARC64 IXfx	1.848 GHz	16	0.36
SGI Altix ICE 8400EX	Intel Xeon X5570	2.93 GHz	8	0.68

that it does not use SSE-optimized routines in the ELPA2 solver. For each case, the workflow with the shortest execution time is marked with bold letters.

In our numerical experiments, we operated ATMathCoreLib by using these data as inputs, instead of actually executing the GEP solver each time. More specifically, at the ith execution ($1 \leq i \leq N$), if the workflow selected by ATMathCoreLib was k_i, we picked up the execution time of the k_i-th workflow from Table 3, added random noise to it, and input it to ATMathCoreLib. As random noise, we used a random variable following normal distribution with mean zero and standard deviation equal to 10%, 20%, or 40% of the corresponding execution time. The number of total executions was set to $N = 100$ for all cases.

The results of automatic code selection is illustrated in Figs. 2 through 7. Figures 2, 4 and 6 show the execution time for each iteration, while Figs. 3, 5 and 7 show the workflows selected by ATMathCoreLib for each iteration. As can be seen from the latter graphs, ATMathCoreLib tries various workflows at the beginning, but gradually narrows down the candidates to one or two, finally chooses the one it considers the fastest and then continues executing only that one. In all the cases given here, the workflow finally chosen by ATMathCoreLib was actually the fastest one, even if the noise level was as high as 40%. Thus we can conclude that automatic code selection using ATMathCoreLib is quite robust against fluctuations in the execution time. These final choices show that ELPA is the fastest for reduction to the SEP in all three cases. For solution of the SEP, EIGX was the fastest for the $n = 430,080$/K and $n = 90,000$/Altix cases, while EIGS was the fastest for the $n = 90,000$/FX10 case.

Table 3. Execution time of each workflow for three cases (taken from [9]).

Matrix size/Machine	Workflow	Total execution time (sec)
$n = 430,080$/K	A	11,634
($p = 10,000$)	B	8,953
	C	5,415
	D	4,242
	E	2,990
	F	2,809
	G	**2,734**
	H	3,595
$n = 90,000$/Altix	A	1,985
($p = 256$)	B	1,883
	C	1,538
	D	1,621
	D’	2,621
	E	1,558
	F	1,670
	G	**1,453**
	H	2,612
$n = 90,000$/FX10	A	1,248 ($p = 1,369$)
($p = 1,024/1,369$)	B	691 ($p = 1,024$)
	C	835 ($p = 1,369$)
	D	339 ($p = 1,024$)
	E	262 ($p = 1,024$)
	F	**250** ($p = 1,369$)
	G	314 ($p = 1,024$)
	H	484 ($p = 1,369$)

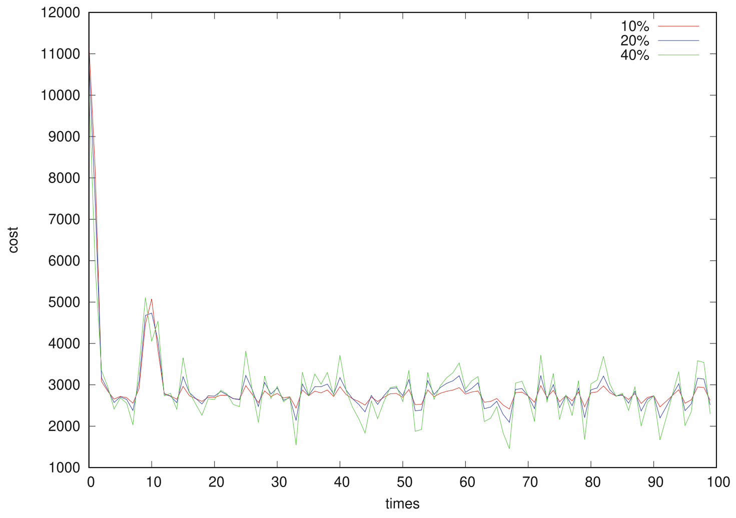

Fig. 2. Execution time for the 430,080/K case.

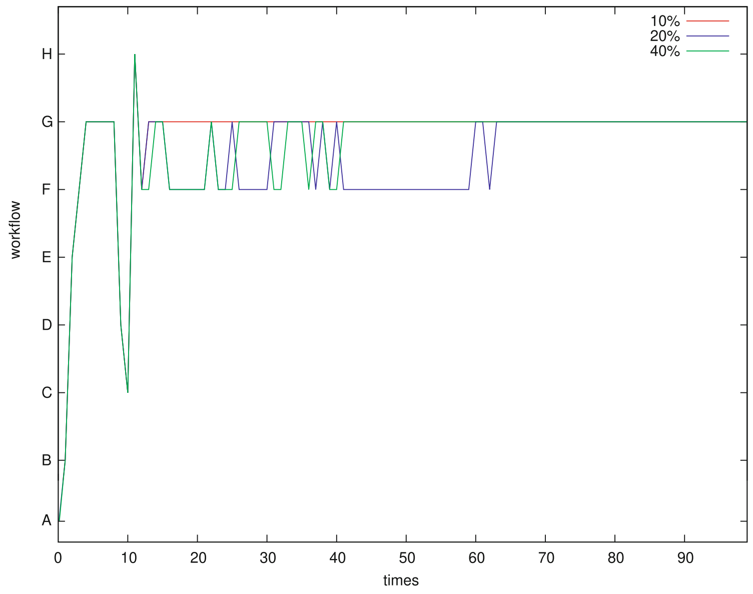

Fig. 3. Workflows selected by ATMathCoreLib in the 430,080/K case.

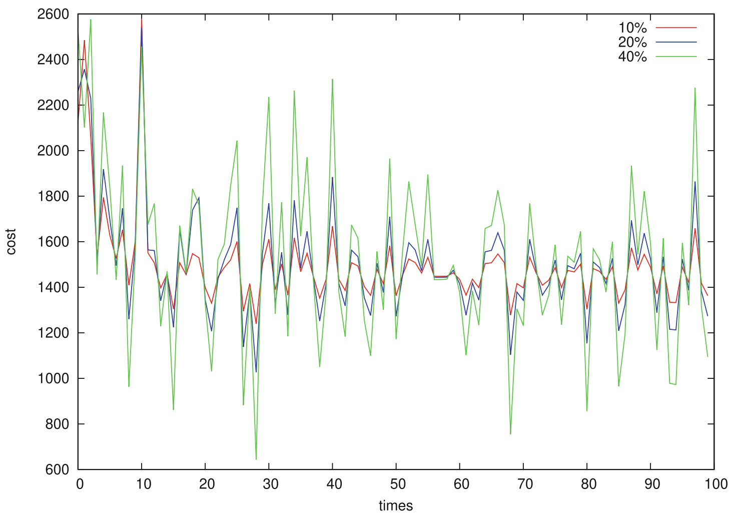

Fig. 4. Execution time for the 90,000/Altix case.

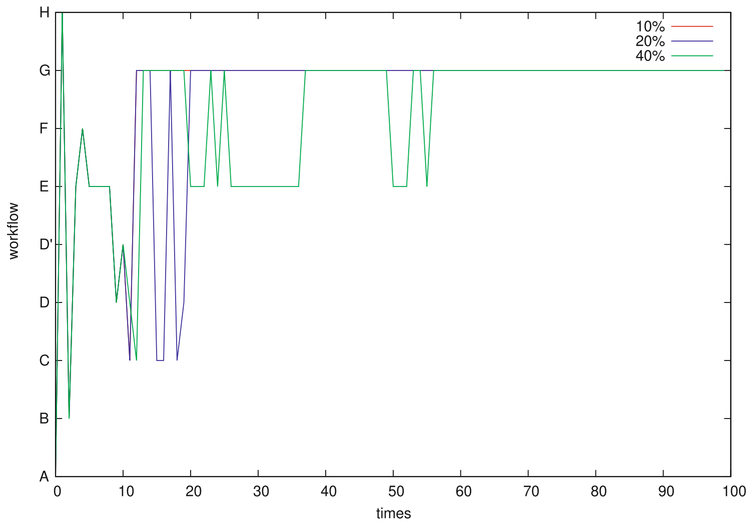

Fig. 5. Workflows selected by ATMathCoreLib in the 90,000/Altix case.

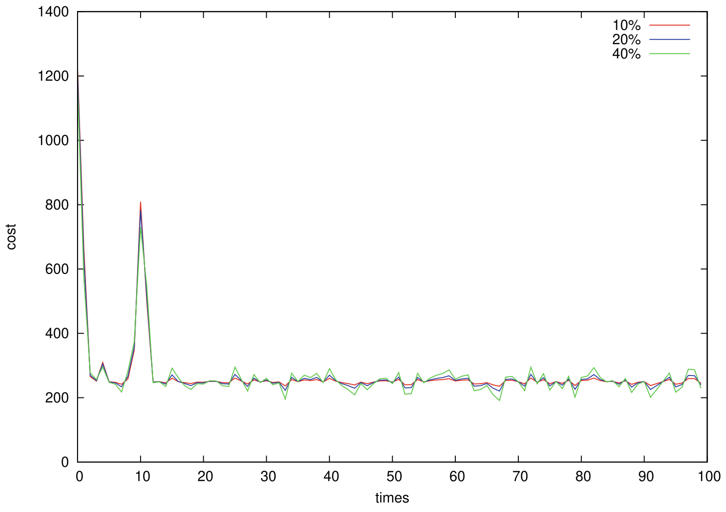

Fig. 6. Execution time for the 90,000/FX10 case.

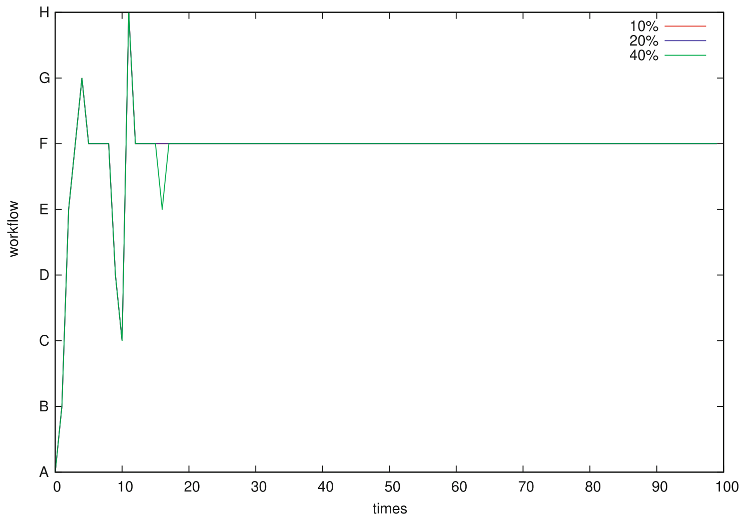

Fig. 7. Workflows selected by ATMathCoreLib in the 90,000/FX10 case.

5 Conclusion

In this paper, we proposed a strategy for automatic code selection for the dense symmetric generalized eigenvalue problem. We consider the situation where N GEPs of the same size are to be solved sequentially in the same computational environment and there are multiple GEP solvers available whose performance we do not know in advance. Then, our objective is to choose the GEP solver to try for each execution judiciously, by taking into account the tradeoff between exploration and exploitation, and minimize the expected value of the total execution time. This can be realized by using ATMathCoreLib, which is a library to assist automatic performance tuning. Numerical experiments using the performance data on the K computer, Fujitsu FX10 and SGI Altix show that ATMathCoreLib can actually find the best solver even if there are large fluctuations in the execution time. Thus we can conclude that our method provides a robust means for automatic code selection.

Our future work includes applying this methodology to other matrix computations and extending it to optimization of parameters in solvers.

Acknowledgements. The authors thank the anonymous reviewers for valuable comments that helped us to improve the presentation of this paper. The present study is supported in part by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research Nos. 19KK0255 and 22KK19772.

References

1. Naono, K., Teranishi, K., Cavazos, J., Suda, R. (Eds.): Software Automatic Tuning: From Concepts to the State-of-the-Art Results, Springer, 2010. https://doi.org/10.1007/978-1-4419-6935-4
2. Suda, R.: ATMathCoreLib: mathematical core library for automatic tuning (in Japanese), IPSJ SIG Technical Report, Vol. 2011-HPC-129, No. 14, pp. 1–12 (2011)
3. Golub, G.H., Van Loan, C.F.: Matrix Computations, 4th edn. Johns Hopkins University Press, Baltimore (2012)
4. Blackford, L.S., et al.: ScaLAPACK Users' Guide, SIAM. Philadelphia (1997). https://doi.org/10.1137/1.9780898719642
5. Auckenthaler, T., et al.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. **37**(12), 783–794 (2011). https://doi.org/10.1016/j.parco.2011.05.002
6. Marek, A., et al.: The ELPA library - scalable parallel eigenvalue solutions for electronic structure theory and computational science. J. Phys.: Condens. Matter **26**, 213201 (2014). https://doi.org/10.1088/0953-8984/26/21/213201
7. Imamura, T., Yamada, S., Yoshida, M.: Development of a high performance eigensolver on a peta-scale next-generation supercomputer system. Prog. Nucl. Sci. Technol. **2**, 643–650 (2011). https://doi.org/10.15669/pnst.2.643
8. Imamura, T., Hirota, Y., Fukaya, T., Yamada, S., Machida, M.: EigenExa: high performance dense eigensolver, present and future, 8th International Workshop on Parallel Matrix Algorithms and Applications (PMAA14), Lugano, Switzerland, 2014. http://www.aics.riken.jp/labs/lpnctrt/index_e.html
9. Imachi, H., Hoshi, T.: Hybrid numerical solvers for massively parallel eigenvalue computation and their benchmark with electronic structure calculation. J. Inform. Process. **24**(1), 164–172 (2016). https://doi.org/10.2197/ipsjjip.24.164
10. Nagashima, S., Fukaya, T., Yamamoto, Y.: On constructing cost models for online automatic tuning using. ATMathCoreLib, In: Proceedings of IEEE MCSoC 2016, IEEE Press (2016). https://doi.org/10.1109/MCSoC.2016.52
11. Dongarra, J., Du Croz, J., Hammarling, S., Hanson, R.J.: An extended set of fortran basic linear algebra subprograms. ACM Trans. Math. Softw. **14**(1), 1–17 (1988). https://doi.org/10.1145/42288.42291
12. Dongarra, J.J., Du Croz, J., Hammarling, S., Hanson, R.J.: Algorithm 656: an extended set of basic linear algebra subprograms: Model implementation and test programs. ACM Trans. Math. Softw. **14**(1), 18–32 (1988). https://doi.org/10.1145/42288.42292
13. Tanaka, K., et al.: EigenKernel. Jpn. J. Ind. Appl. Math. **36**(2), 719–742 (2019). https://doi.org/10.1007/s13160-019-00361-7
14. http://www.elses.jp/matrix/