パラメータ自動チューニングにおける GPTune の性能評価

森下 誠[1,a)]　片桐 孝洋[2]　Osni Marques[3]　Yang Liu[3]　星野 哲也[2]　永井 亨[2]　河合 直聡[2]

概要：数値計算ライブラリには、その性能に影響を与える多くのパラメータがあることが多い。ライブラリの高い性能を得るためには、そのパラメータのチューニングが必要である。しかし、このようなパラメータのチューニングは、専門的な知識がなければ困難である。そこで、数値計算ライブラリの性能を向上させるために、ソフトウェアによるオートチューニング (AT) が期待されている。本研究では、DOE の Exascale Computing Project で開発された AT フレームワークである GPTune の手法を説明する。また、数値計算ライブラリとして広く使われている ScaLAPACK 上のルーチンを用いた GPTune の適応例を示す。

Performance Evaluation of GPTune for Parameter Auto-Tuning

Abstract: Numerical libraries often have many parameters that impact its performance. To obtain high performance of the libraries, tuning the parameters are required. However, it is difficult to tune such parameters without special knowledge on them. Software auto-tuning(AT), therefore, is one of promising approaches to establish high performance for numerical libraries. In this study, we explain the methodology of GPTune, which is an AT framework developed by DOE's Exascale Computing Project. In addition, we show an example of adaptation for the GPTune with a routine on ScaLAPACK, which is one of widely-used numerical libraries.

1. Introduction

In High Performance Computing (HPC), software often has many parameters that impact its performance. However, it is difficult to determine optimal values for such parameters in an impromptu way. The automatic tuning – autotuning – of parameters is therefore an area of great interest.

There is a rich research history in the field of autotuning [5]. Initially, the focus was on parameter tuning within numerical libraries [3] [8] [4]. Presently, this autotuning concept has expanded into machine learning, specifically targeting the tuning of hyperparameters. Consequently, autotuning remains a pivotal technology.

The purpose of this work is to understand the methodology of GPTune [1] [6], which is an autotuning framework developed by DOE' s Exascale Computing Project, and use the framework in a set of applications of interest. For the target numerical library, we choose ScaLAPACK [2], which is one of widely-used numerical libraries on supercomputer environments.

This report is organized as follows. In Section2, a rough function of GPTune is explained. Section 3 is performance evaluation of autotuning by GPtune. Section 4 give a conclusion in this report.

2. GPTune

GPTune [1] [6] is an autotuning framework that solves an underlying black-box optimization problem, using surrogate modeling. GPTune uses Bayesian optimization based on Gaussian Process regression and supports advanced features such as multi-task learning, transfer learning, multi-fidelity and objective tunings, and parameter sensitivity analysis. GPTune targets the autotuning of HPC codes, in particular applications that are very expensive to evaluate.

Problem description in GPTune

The following are Tuning Spaces defined by GPTune.

[1] Graduate School of Informatics, Nagoya University
[2] Information Technology Center, Nagoya University
[3] Lawrence Berkeley National Laboratory
[a)] morishita@hpc.itc.nagoya-u.ac.jp

(1) Input Space
- This space defines the problems to be tuned.
- Every point in this space represents one instance of a problem.

(2) Parameter Space
- This space defines the application parameters to be tuned.
- A point in this space represents a combination of the parameters.
- The tuner finds the best possible combination of parameters that minimizes the objective function associated with the application.
- The Listing 1 shows a sample code of parameter space definition by GPTune.

```
parameter_space = \
Space([Real(0., 1., transform="normalize", name="x")])
```
Listing 1: Sample code of parameter space definition

In Listing 1, we can define arbitrary parameters, named "x" . In addition, types of parameter values can be specified, such as real, etc. The following is explanation of output space of parameters.

(3) Output Space
- This space defines the objective of the application to be optimized.
- For example, this can be runtime, memory or energy consumption in HPC applications or prediction accuracy in machine learning applications.

Objective Function

The user needs to define a (Python) function representing the objective function to be optimized in GPTune. Listing 2 is a sample code of definition for objective function.

```
def objectives(point):
    x = point['x']
    # call HPC code with parameter x
    ...
    # get the function value f
    ...

    return [f]
```
Listing 2: Sample code of definition of objective function

3. Performance Evaluation

3.1 Target Problem

In this work, we have focused on the autotuning of two algorithms implemented in ScaLAPACK [2], which are QR and LU decompositions. These two decompositions are basic and widely-used in mathematical computation. Hence improvement of computation time is very crucial for several applications.

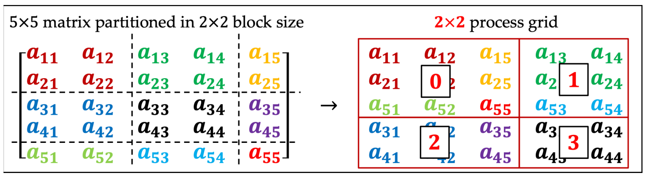

図 1 Example of 2D block-cyclic distribution (Source [2])

The QR decomposition is provided with the GPTune repository, while the LU decomposition was not provided. In this research, we have implemented a new code as my work.

In these two cases, the tuning parameters are the block size (for cache memory optimization), and the process grid size for distributed computing. Figure 1 shows an example of 2D block-cyclic distribution.

QR decomposition is the decomposition of matrix $A \in \mathbb{R}^{m\times n}$ into the product of a mth-order orthogonal matrix Q and a $m\times n$ matrix upper triangular matrix R. Formula (1) shows the QR decomposition.

$$A = QR \qquad (1)$$

LU decomposition is the decomposition of matrix $A \in \mathbb{R}^{n\times n}$ into the product of a lower triangular matrix L and an upper triangular matrix U. Formula (2) shows the LU decomposition.

$$A = LU \qquad (2)$$

In ScaLAPACK, distributed parallel interfaces (subroutines) for LU and QR decompositions are provided. Hence users can implement distributed parallel numerical computation by utilizing the interfaces of ScaLAPACK.

3.2 Experimental setting

We adapted GPTune in QR and LU decomposition code. This experiment is a preliminary evaluation of ability of autotuning by GPTune.

Experiments were conducted on a PC at hand and parameter tuning was performed on a small-scale problem before running in a massively parallel computing environment, such as distributed memory supercomputers. The execution environment and target tuning parameters are shown in the Table 1, 2, respectively.

In the Table 2, we choose two kinds of tunable parameters. First is block sizes, including two parameters for

表 1 Execution environment (proxy for a distributed environment)

Name	MacBookAir (M1, 2020)
OS	Ventura 13.4.1
CPU	Apple M1 chip, 8 cores
Memory	8 GB

表 2 Tuning parameters

parameter name	overview	parameter range
nb	row block size	8 ∼ 128
mb	column block size	8 ∼ 128
p	row process grid	1 ∼ 8

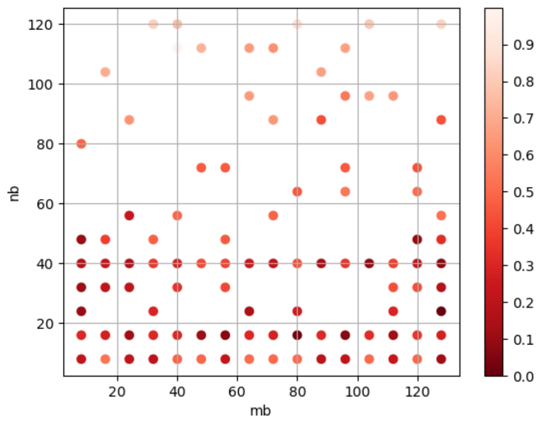

図 2 Execution time for 2 tuning parameters, block size (nb, mb) in QR decomposition. (matrix size 1000 × 1000)

row and column direction. The other is related to process grid.

Here, the process grid size changes only row (parameter p). This is tunable parameter of the experiments, since the parallel execution time affects configurations of processor grids. The causes by time of communications for the target routines, in this case, parallel implementations for QR and LU decompositions. In addition, size of matrices, and configurations of matrix, i.e. $n \times m$, affect execution time. Hence it is difficult to set constant parameters in advance.

Moreover, the block size (See Figure 1) is also tunable parameters. In general, block size should be optimized with respect to physical cache sizes. Hence it depends on installation environments. The block size is also known as crucial parameter in several numerical libraries.

We used 8 cores on CPU. The experiments were conducted in an environment where $p \times q \leq 8$ is always satisfied when the process grid size of column is q.

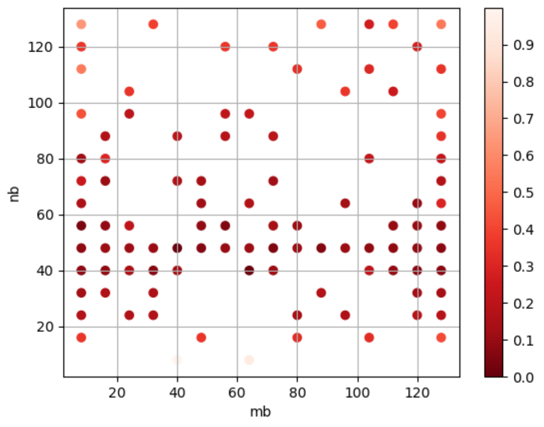

図 3 Execution time for 2 tuning parameters, block size (nb, mb) in QR decomposition. (matrix size 2000 × 2000)

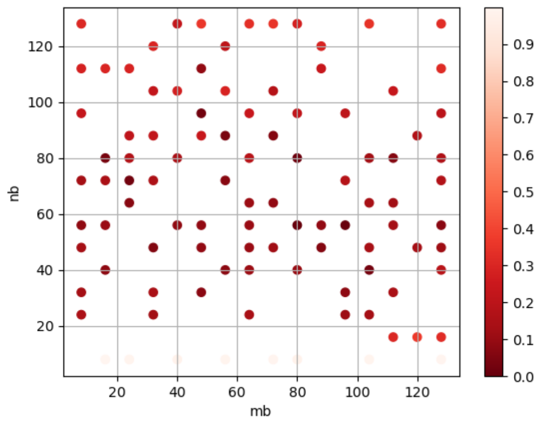

図 4 Execution time for 2 tuning parameters, block size (nb, mb) in QR decomposition. (matrix size 3000 × 3000)

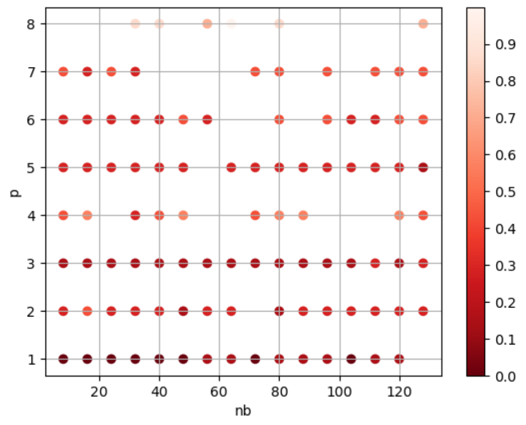

図 5 Execution time for 2 tuning parameters, block size and process grid (nb, p) in LU decomposition. (matrix size 1000 × 1000)

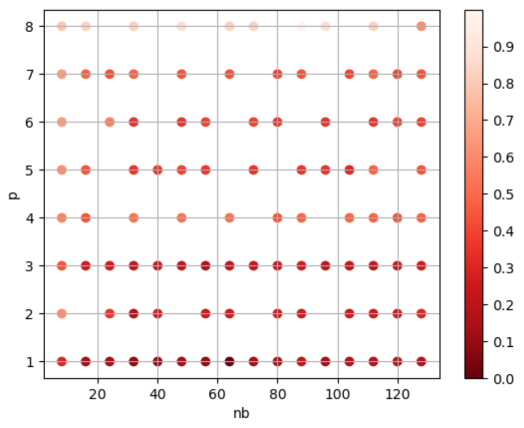

図 6 Execution time for 2 tuning parameters, block size and process grid (nb, p) in LU decomposition. (matrix size 2000×2000)

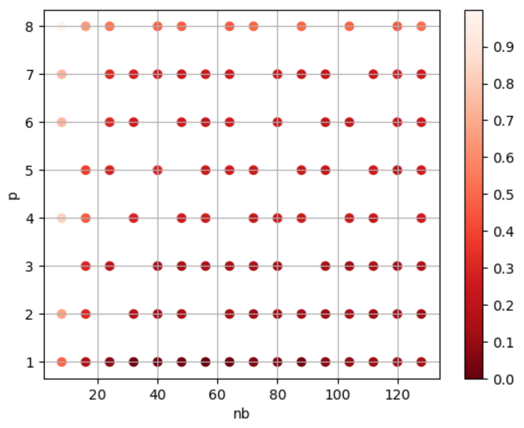

図 7 Execution time for 2 tuning parameters, block size and process grid (nb, p) in LU decomposition. (matrix size 3000×3000)

表 3 Optimal parameter set (nb, mb) in QR

Matrix size N	(nb, mb)
1000	(128, 24)
2000	(40, 48)
3000	(80, 56)

表 4 Optimal parameter set (nb, p) in LU

Matrix size N	(nb, p)
1000	(16, 1)
2000	(64, 1)
3000	(64, 1)

3.3 Result

Figures 2, 3 and 4 show the distribution of the execution time when block size nb and mb were tuned using GPTune. Table 3 shows the optimal parameter set in QR decomposition. Color of points in Figures 2, 3 and 4 is normalized execution time in seconds.

From the results, when the problem size is small such as $N = 1000$, the range with the lower nb value is searched, but as the problem size increases such as $N = 2000, 3000$, it can be said that the entire range is searched. In general, the closer the block size is to square ($nb = mb$), the more efficient the processing is. This result is consistent with that fact.

Figures 5, 6 and 7 show the distribution of the execution time when block size nb and p were tuned using GPTune. Table 4 shows the optimal parameter set in LU decomposition. Color of points in Figures 5, 6 and 7 is normalized execution time in seconds.

From results, for all problem sizes, processing is fastest when row process grid $p = 1$. This indicates that when the matrix to be calculated is distributed per process, it is more efficient to split it into one dimension such as $p \times q = 1 \times 8$, rather than into two dimensions such as $p \times q = 2 \times 4$ or $p \times q = 4 \times 2$.

4. Conclusion

We conducted experiments by utilizing GPTune to optimize performance parameters for QR and LU decompositions. Our preliminary results indicate improved performance compared to default parameter executions. We elucidated the process of selecting specific tuning parameters by modifying the space and objective functions within GPTune. Furthermore, we demonstrated how to integrate the target application, ScaLAPACK in our case, with GPTune effectively. Notably, we developed original code for parameter tuning in the LU decomposition of ScaLAPACK using GPTune, which is a novel aspect of our preliminary experiment. Moving forward, our future endeavors include deploying GPTune on the Supercomputer ”Flow” at Information Technology Center, Nagoya University to assess its performance on the ARM architecture, similar to the Supercomputer “Fugaku.” Additionally, we plan to apply GPTune to various applications, such as Fast Fourier Transform (e.g., FFTX, the exascale successor to the FFTW open-source discrete FFT package). Furthermore, we have explored hyperparameter tuning for quantum-related technologies [7], such as quantum annealers and quantum circuit simulators. The adaptation and evaluation of GPTune for these purposes constitute significant aspects of our future work.

謝辞 This work was supported by Japan Science and Technology Agency (JST) as part of SICORP, Grant Number JPMJSC2201.

参考文献

[1] Github - gptune/gptune.
[2] Scalapack—scalable linear algebra package.
[3] Jeff Bilmes et al. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347, 1997.
[4] Takahiro Katagiri et al. ABCLib_DRSSED: A parallel eigensolver with an auto-tuning facility. *Parallel computing*, 32:231–250, 2006.
[5] Takahiro Katagiri and Daisuke Takahashi. Japanese autotuning research: Autotuning languages and fft. In *Proceedings of the IEEE*, pages 2056–2067, 2018.
[6] Yang Liu et al. Gptune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 234 – 246, 2021.
[7] Makoto Morishita et al. 量子コンピューティングへの自動チューニングの適用と評価. 2023-HPC-188:22 2, 2023.
[8] R Clint Whaley et al. Automated empirical optimizations of software and the atlas project. *Parallel computing*, 27:3–35, 2001.