

Masaaki Geshi *Editor*

The Art of High Performance Computing for Computational Science, Vol. 2

Advanced Techniques and Examples for
Materials Science



Springer

The Art of High Performance Computing
for Computational Science, Vol. 2

Masaaki Geshi
Editor

The Art of High Performance Computing for Computational Science, Vol. 2

Advanced Techniques and Examples
for Materials Science

 Springer

Editor
Masaaki Geshi
Osaka University
Toyonaka, Japan

ISBN 978-981-13-9801-8 ISBN 978-981-13-9802-5 (eBook)
<https://doi.org/10.1007/978-981-13-9802-5>

© Springer Nature Singapore Pte Ltd. 2019

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

This is the second of two volumes that are written about the basics of parallelization, the foundation of numerical analysis, and related techniques. Even if it is mentioned as a foundation, we do not assume a novice here completely in this field, so if you would like to know from the beginning of programming, you should learn from another book suitable for that. For readers, those who have learned physics, chemistry, biology (earth sciences, space science, weather, disaster prevention, manufacturing, etc.) are assumed. Furthermore, we assume those who use numerical calculation and simulation as research methods. In particular, we assume those who develop software code. Many of them have not learned systematically about programming and numerical calculation, but from the information science experts, many parts are included as contents of the undergraduate level.

This Volume 2 includes advanced techniques based on concrete applications of software applications for several fields, in particular, the field of materials science. Chapter 1 outlines supercomputers including a brief explanation of the history of hardware. Chapter 2 details a program tuning procedure. Chapter 3 describes concrete tuning results on the K computer for several software applications: RSDFT [1] and PHASE [2] (now the official name changes to PHASE/0) in materials science, nanoscience, and nanotechnology; Seism3D in earth science; FrontFlow/Blue in engineering. The above chapters are more practical than Chaps. 1–5 in the Volume 1. Chapter 4 explains how to reduce the computational cost of density functional theory (DFT) calculation from $O(N^3)$ to $O(N)$, so-called order-N method. This method is implemented in the software application, OpenMX [3]. Chapter 5 explains acceleration techniques of classical molecular dynamics (MD) simulations, for example, general techniques for hierarchical parallelization on the latest general-purpose supercomputers, in particular, connected by means of a three-dimensional torus network. These techniques are implemented in the software applications, MODYLAS [4]. This chapter also introduces the software application, GENESIS [5], which is developed for investigating the long-term dynamics of biomolecules by simulating a huge biomolecule system by using an efficient structure search method such as the extended ensemble method. Chapter 6 explains techniques for large-scale quantum chemical calculation techniques

including the order-N method. These techniques are implemented in software applications, DC-DFTB-K [6] and SMASH [7]. You can download and use some of them. MaterApps [8] is useful for finding software applications in the field of material science. This website introduces software applications from around the world as well as software applications made in Japan.

This book is revised and updated from the Japanese book published by Osaka University Press in 2017, *The Art of High Performance Computing for Computational Science 2* (Masaaki Geshi Ed., 2017). This book is based on the lectures, “Advanced Computational Science A” and “Advanced Computational Science and B”, broadcasted to maximally 17 campuses through videoconference systems from 2013. All the texts and videos are published on websites (only in Japanese). These were parts of the human resource development programs that we have tackled as part of the project of the Computational Materials Science Initiative (CMSI) organized by the Ministry of Education, Culture, Sports, Science and Technology (SPIRE) field 2 <New materials/energy creation>, so-called the project of K computer. These lectures are aiming to contribute to developing young human resources, centering on basic techniques that will not change for a long time being even though it is a computer that progresses day by day. These lectures were offered from the Institute for NanoScience Design, Osaka University. We have gathered up to about 150 participants per lecture. Participants exceeded 6500 net people in the past 6 years. The videos of the lectures and lecture materials are opened to the public on the web, and anyone can learn the content at any time in Japanese. Now we are using cloud-based video meetings service that connects many users across different devices to make it easier for more people to participate. These lectures continue to distribute with slightly changing the organizational structure even after the project ends.

I would like to express my deep appreciation to the authors, who cooperated in the writing of the series of books as an editor. I would like to thank the staff of Springer for publishing the English version. I think that the techniques cultivated in the project of K computer of Japan contain many useful contents for future HPC. I hope it will be shared in the world and it will contribute to the development of HPC and the development of science.

Osaka, Japan
April 2019

Masaaki Geshi

References

1. <https://github.com/j-iwata/RSDFT>
2. <https://azuma.nims.go.jp/software>
3. <http://www.openmx-square.org/>
4. <http://www.modylas.org/>

5. <https://www.r-ccs.riken.jp/labs/cbrt/>
6. http://www.chem.waseda.ac.jp/nakai/?page_id=147&lang=en
7. <http://smash-qc.sourceforge.net/>
8. <https://ma.issp.u-tokyo.ac.jp/en/>

Contents

1 Supercomputers and Application Performance	1
Kazuo Minami	
2 Performance Optimization of Applications	11
Kazuo Minami	
3 Case Studies of Performance Optimization of Applications	41
Kazuo Minami and Kiyoshi Kumahata	
4 $O(N)$ Methods	89
Taisuke Ozaki	
5 Acceleration of Classical Molecular Dynamics Simulations	117
Y. Andoh, N. Yoshii, J. Jung and Y. Sugita	
6 Large-Scale Quantum Chemical Calculation	159
Kazuya Ishimura and Masato Kobayashi	
Index	203

Chapter 1

Supercomputers and Application Performance



Kazuo Minami

Abstract Before the advent of modern supercomputers, single processors were approaching the limit of improvement in operating frequency, and there was a memory wall problem. Even if the computing capacity of a single processor could be increased, the data supply capacity of the memory could not match the computing capacity. The increase in operating frequency also caused the problem, that is, power consumption increased faster than performance improvement. In other words, the limit of performance improvement of a single processor was becoming apparent. To solve these problems, parallel architectures, in which many single processors are connected by a communication mechanism, have been used. The two points, “programming conscious of parallelism” and “programming conscious of execution performance”, are very important for users, researchers, and programmers to make effective use of the present supercomputers equipped with tens of thousands of processors.

1.1 What Is a Supercomputer?

In this section, we first describe the development of computers and the changes in the usage technologies of supercomputers in Sect. 1.1, and in Sect. 1.2 we describe two important points for developing high-performance applications. Computational science, which elucidates scientific phenomena by using numerical simulation, has long been described as the third science alongside theory and experiments, and in recent years, innovative scientific technology research and development using super-

¹<http://www.riken.jp/en/research/environment/kcomputer/>.

²The Institute of Physical and Chemical Research (<http://www.riken.jp/en/>).

³<https://www.top500.org/>.

K. Minami (✉)

RIKEN Center for Computational Science, RIKEN, Kobe, Hyogo, Japan
e-mail: minami_kaz@riken.jp

computers has been active all over the world. In Japan, the K computer¹ developed by RIKEN² in 2011 won the top 500³ ranking for two consecutive terms.

The application of supercomputers in Japan is an innovative way to elucidate various natural phenomena over a vast scale from the extremely fine quantum world to the universe, including an enormous number of galaxies, and the discoveries made are expected to contribute to society. For example, on the very small scale of ten to the minus several powers of meters, we expect to understand the behavior of viruses, liposomes (consisting of several hundred thousand atoms), and other organic phenomena through long-running simulations, and this is expected to contribute to the medical field, inexpensive biofuels, and new energy fields. On a slightly larger scale, we expect to accelerate innovation in next-generation electronics through design simulation of entire next-generation semiconductor nanodevices and the creation of new functional nonsilicon materials such as nanocarbons. On the scale of human society from several tens of meters to several hundreds of kilometers, we expect to contribute to detailed disaster-prevention planning by seismic simulation, combining seismic wave propagation and structure response. On a global scale of several thousand kilometers to several tens of thousands of kilometers, we expect to present high-resolution global weather forecasts and accurate predictions of the course and intensity of typhoons by climate simulation, and to contribute to climate change research. On the larger scale of more than 10 to the 20th power of meters, we expect to elucidate cosmic phenomena, such as the generation of stars and analysis of the behaviors of galaxies.

As described earlier, various applications are expected for supercomputers, but what is a supercomputer in the first place?

Although there is no clear definition of a supercomputer, it is regarded as a computer with extremely high speed and outstanding computing capacity, compared with the general computers of its era. For example, a supercomputer is defined in present government procurement in Japan (2016) as a computer capable of 50 trillion or more floating point operations per second (50 TFLOPS)⁴: This number is reviewed as necessary. In the mid-1940s, one of the first digital computers, named the ENIAC (an abbreviation of Electronic Numerical Integrator and Computer), appeared. In 1976, the CRAY-1, which was described as the world's first supercomputer, appeared; its theoretical computing performance was 160 MFLOPS. The performance of a personal computer using a Pentium IV in 2002 was about 6.4 GFLOPS: about 40 times the performance of the CRAY-1. At that time, the performance of the Earth Simulator, which was Japan's fastest supercomputer in 2002, was 40 TFLOPS, which was about 250,000 times the performance of the CRAY-1. The K computer, which achieved the world's fastest performance in 2011, achieves 10 PFLOPS, about 62.5 million times the performance of CRAY-1.

Computers have achieved these drastic performance improvements, but how?

⁴FLOPS denote a unit of calculation speed. One FLOPS is the execution of one floating point calculation per second. Thus, 160 MFLOPS is equivalent to 160 million floating point operations per second.

Early computers had a single processor. After the technology changed from vacuum tubes to semiconductors, the improvement of the frequency of the semiconductor devices promoted the performance improvement of the CPU and improved the performance of the computer. The memory was composed of one or more memory banks, and a memory bank could not be accessed until a certain time had elapsed after a previous access. The waiting time for memory accesses remains several tens of nanoseconds even now. However, until the 1970s, because the operating frequency of the computer was low and the operation of the computing unit was slow, memory access times were not a major problem. It was an era when the computation speed was the bottleneck rather than the memory transfer performance.

Since then, while the waiting time for memory access of several tens of nanoseconds has not reduced much, the number of cycles the CPU must wait for memory access has increased with the improvement in CPU operating frequency. Moreover, because of the miniaturization of the semiconductor process, more computing units can be mounted in one CPU. As a result, the data transfer capability of the memory reduces the computing capacity of the computing unit. This is called the memory wall problem.

Between the latter half of 1970 and the 1980s, although it was based on a single processor, a vector architecture was developed that enabled high-speed computation using vector pipelines, treating data that could be processed in parallel by paying attention to the parallelism within loops. To solve the memory wall problem in the vector computer, the number of memory banks was increased, and the CPU read data from different memory banks cyclically to supply data to the computing unit continuously. The problem of the waiting time to access the same memory bank was thus overcome. By adopting this mechanism, it was possible for the vector computer to balance the data supply capability of the memory and the calculation ability of the computing unit.

At that time, the processor's operating frequency was several tens of MHz or more, and as described earlier, more computing units could be added because of the progress in the miniaturization of the semiconductor process. This increase, together with the increased operating frequency of the computing unit, contributed to the realization of high-speed vector computers. Although the vector system was fast, the manufacturing cost and power consumption increased because of the expensive memory bank mechanism described above.

Around the same time as the development of the vector architecture, the computing unit of the scalar architecture also became RISC⁵ and was pipelined, taking advantage of the increases in processing units and operating frequency. The scalar architecture evolved into a superscalar architecture with multiple computing units. Furthermore, by using SIMD⁶ and other techniques, high-speed computation was made possible by utilizing advanced parallelism hidden in the program.

⁵Reduced instruction set computing.

⁶Single instruction multiple data: A class of parallel processing that applies one instruction to multiple data.

In the scalar architecture, to cope with the memory wall problem, a countermeasure other than the vector architecture was taken: a cache with high data supply capability was placed between the memory and the computing unit without increasing the number of memory banks. As much of data as possible was placed in the cache, and the data were reused to compensate for the limited data supply capacity of the memory to the computing unit. Although this method has performance disadvantages, it has benefits in terms of cost and power consumption over the multiple-bank method of the vector architecture.

The single processor was approaching the limits of improvement of the operating frequency and the memory wall problem remained. Even if the computing capacity of the single processor could be increased, the data supply capacity of the memory could not catch up with the computing capacity. Furthermore, with the increases in operating frequency, we also faced the problem of power consumption increasing faster than the improvement in performance. In other words, the limits of performance improvement of single processors were becoming apparent.

To solve this problem, a parallel architecture in which many single processors are connected by a communication mechanism has appeared. Without this development, it would be impossible to obtain the necessary computing power with realistic power consumption.

At present, hybrid, massively parallel computers are emerging, in which multiple calculation cores are built in a processor and thousands to tens of thousands of processors are connected by a communication network.

Although each node of a supercomputer is basically the same as an ordinary computer, computing capacity and computing performance in total are extremely high, and high-speed interconnection performance is required. Further, because low-power performance of the total system is required, it is essential that power saving is implemented at the processor level. Because the number of parts constituting the system is very large, extremely high reliability is required for individual parts, and high reliability of the total system is required.

Up to this point, we have explained the development of hardware. As the hardware evolved, how has its usage changed so that the performance of the hardware can be fully utilized?

In the early days of computers, the processing speed was a bottleneck compared with the memory transfer performance with a single processor, so development environments such as high-level languages and compilers emerged. It was common for researchers and programmers to reproduce formalized and discretized theoretical model equations in code. High-speed processing was realized by developing compilers that could interpret the parallelism hidden in the program.

From the latter half of 1970 to the 1980s, when vector architectures used multiple memory banks to cope with the memory wall problem, the parallel nature of loop indices was exploited and parallel-processable data were pipelined.

As a programming technique in the age of vector architecture, it was necessary to guarantee the parallel nature of loops. Eliminating recurrence (regression references) became an essential performance optimization technique.

In the scalar architecture, as described earlier, by dealing with a cache with high data supply capability, we coped with the memory wall problem. In addition, in the scalar architecture, SIMD was introduced and effectively used simplified instructions with RISC, superscaling with multiple operation pipelines, and software pipelining by compiler.

Similar changes were introduced for the scalar architecture as well. It was necessary to guarantee the parallelism of the loop index, and SIMD vectorization has become a performance optimization technique that requires eliminating recurrence. Efficient cache usage technology has also become indispensable.

After the limits of performance improvement of a single processor were seen and supercomputers changed to a parallel architecture, with parallelism among several to several hundred cores in the CPU, parallelism among thousands and tens of thousands of CPUs has been realized by introducing it explicitly in programs. In other words, it becomes necessary for the programmer to parallelize the code in consideration of the parallelism among the cores and the parallelism among the CPUs, and to program with consideration of the distribution of the data used by the cores and CPUs for calculation. In addition, a communication system usage technique that exploits the network topology between the nodes where the processes are located has become necessary.

As described earlier, modern computers still have the memory wall problem in which the computing capacity of the computing unit is increased but the data supply capacity of the memory is relatively insufficient. To cope with this problem, cache memories (level 1, level 2, and level 3) with high data supply capability are provided, and the data are placed in the cache and reused many times while performing a calculation.⁷ Thus, compared with programming on older computers, the necessity of programming with attention to multilevel memory structures such as cache became obvious. However, many programs cannot reuse data as described here. Because the capacity of the computing unit cannot then be fully used, programs may require the use of high-speed data access mechanisms such as prefetch.

1.2 What Is a High-Performance Application?

The two points mentioned in Sect. 1.1, “programming conscious of parallelism” and “programming conscious of execution performance” must be recognized by users, researchers, and programmers who use the present supercomputers equipped with tens of thousands of processors and containing various enhancements and new functions. Thus, high-performance applications require “performance optimization with high parallelism” and “performance optimization of single CPUs”. In Chaps. 1–3, these are the performance-optimizing techniques used to exploit the performance of modern supercomputers.

⁷This approach is called cache blocking.

1.2.1 Important Points in Optimizing High Parallelism

Parallelization is briefly described first. The basic idea is simple. As shown in Fig. 1.1, if a problem that is sequentially computed using one processor is computed in parallel using four processors, the computation should be four times faster and it should be executed in a quarter of the original calculation time.

In simulations of fluids and structural analysis, a mesh is constructed in the spatial direction, and calculations are performed for each mesh point. High parallelization is briefly explained by using this example.

To parallelize the calculation, the mesh is divided into multiple regions. These regions are distributed among the processors and the calculations are performed in parallel. Such a parallelization method is called a domain decomposition method and is depicted in Fig. 1.2. In this figure, after executing the calculation using four processors, data are exchanged using the communication network to achieve consistency of the calculations proceeding in parallel; these steps are then repeated to continue the calculation. As described earlier, in the parallel computation by domain decomposition, adjacent communications are performed to exchange the data of a part of the domain with the adjacent processors. When an inner-product calculation is performed over all the domains, global communication is required to obtain the sum of the data for all processors. An important point in achieving high parallelism is to minimize the amounts of adjacent and global communications mentioned here.

It is also important to make the calculation times for each processor as equal as possible. Differences in calculation time are called load imbalances.

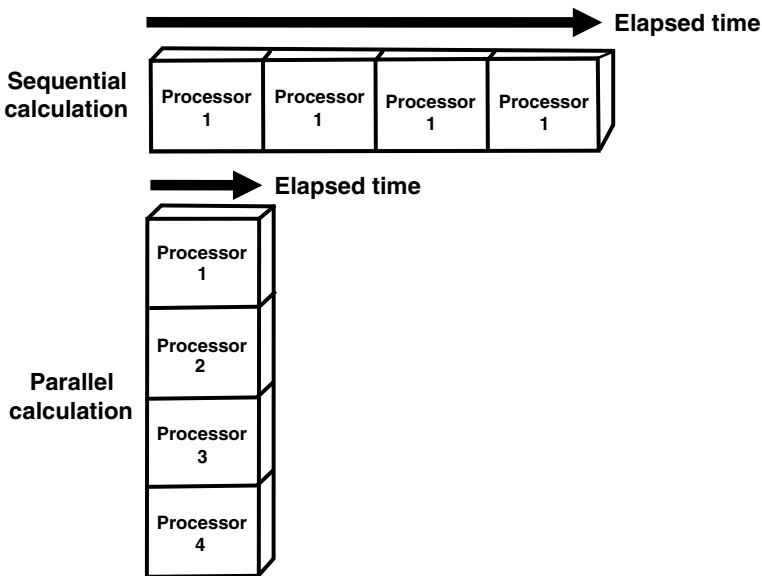


Fig. 1.1 Parallel calculation

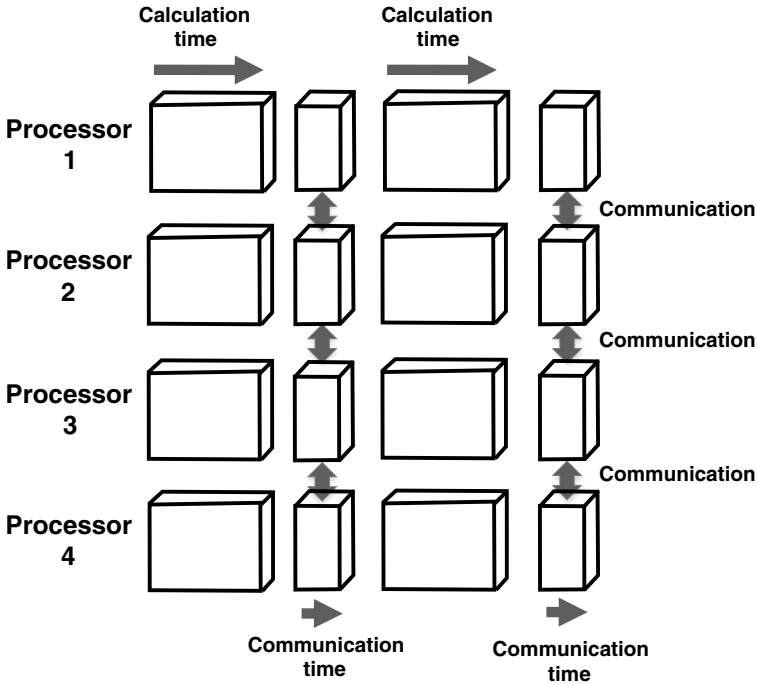


Fig. 1.2 Parallel calculation by domain decomposition

Next, the parallelization rate and parallelization efficiency will be described. Assume that 99% of a sequential calculation can be performed as parallel calculations but that the remaining 1% of the sequential calculation cannot be made parallel; the parallelization rate of this calculation is 99%. Assume that the sequential calculation time is 100 s in total, and the parallel computation is performed using 100 processors. The parallel computation time for the parallelizable component will then be $99 \text{ s}/100 = 0.99 \text{ s}$; 1 s of nonparallel computing time must then be added, so the calculation time with 100 processors is 1.99 s. Thus, the original calculation is 100 s and about 1/50 of that with 100 processors, so the parallelization efficiency is 50%.

In the same way, for 1000 processors, the calculation time is about 1/91. That is, if even 1% of the calculation is nonparallel, efficient parallel calculation cannot be performed no matter how many processors are used. This limit on the benefits of parallelization is known as Amdahl's law. From this, as well as minimizing the communication time, it is important to minimize the nonparallel computing component as much as possible.

1.2.2 Important Points for Single-CPU Performance Optimization

From the viewpoint of single-CPU performance, applications can be roughly divided into two types according to the ratio of the number of bytes of memory accessed to the number of floating point operations required to execute the application. In one type, the data transfer requests (in bytes) are small compared with the number of floating point operations (in FLOPS). Such calculations are called computations with small required byte/FLOP (B/F) values. For example, for the matrix–matrix product calculation shown in Fig. 1.3, in principle the B/F value is $1/N$ when the data movement amount is represented by the number of elements. In general, because the amount of data movement is represented by the number of bytes, for double precision calculations, the movement amount is multiplied by eight, so that it is $8/N$, and in principle the B/F value becomes smaller as N increases. In real programs (see Fig. 1.4), if N becomes a certain size and (a) the access direction is continuous in memory, (a) is in the cache but (b) is not in the cache. Therefore, using a technique called blocking, we divide the matrix into small matrices so that both (a) and (b) are in the cache. Applications with small required B/F values like the matrix–matrix product shown can achieve high single-CPU performance by reusing the data placed in the cache many times.

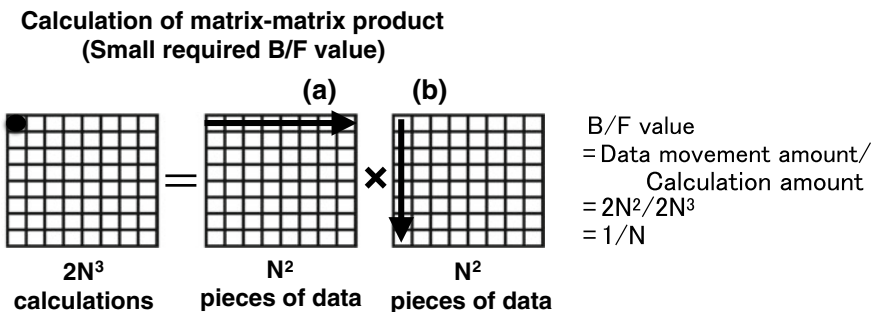


Fig. 1.3 Example of matrix–matrix product (1)

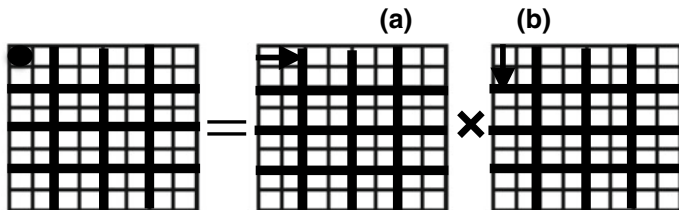


Fig. 1.4 Example of matrix–matrix product (2)

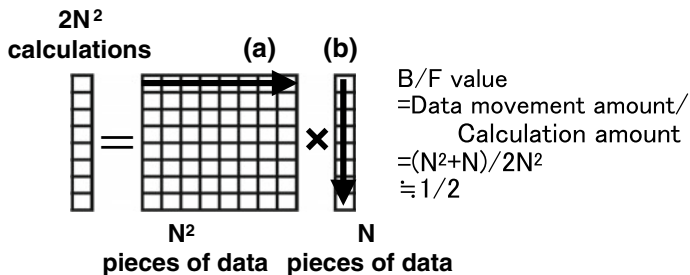


Fig. 1.5 Example of matrix–vector product

In the other type of application, the data transfer requests from memory are large compared with the number of floating point operations required to execute the application. These calculations are called calculations with large required B/F values. Such calculations have problems in effectively using the high performance of the CPU because it is difficult to use the cache effectively. For example, for the matrix–vector product calculation shown in Fig. 1.5, in principle, the B/F value is approximately 1/2 when the data movement is expressed by the number of elements. As above, for a double precision calculation, the movement amount is multiplied by 8 bytes, so becomes $8/2 = 4$; therefore, the B/F value is large when compared with the matrix–matrix product. In this way, as a viewpoint for improving the performance of the single CPU, the required B/F value of the application is important.

Exercises

1. Describe the memory wall problem, which is an important problem for single processors, and describe the characteristics of recent supercomputers.
2. There are various benchmark tests (BMTs) to evaluate the performance of supercomputers. The most famous BMT is the top 500 (<https://www.top500.org/>), which is evaluated by the performance of LINPACK, but there are others as well. For some other BMTs, discuss the relationship between the evaluation method and the field in which the evaluation is important.

Chapter 2

Performance Optimization of Applications



Kazuo Minami

Abstract In this chapter, we present procedures for performance evaluation that we have used for practical applications. The method is outlined in Sect. 2.1, and Sects. 2.2–2.6 describe the details of the method. The classification of problems related to high parallelism and the classification of applications from the viewpoint of single-CPU performance are described. Sections 2.7 and 2.8 then describe performance optimization techniques for each problem pattern related to high parallelism and the techniques for single-CPU performance optimization according to application classification.

2.1 Performance Evaluation Method

The performance evaluation of an application is divided into two parts: “highly parallel performance optimization” and “single-CPU performance optimization.” For each part, the performance evaluation method has two working phases: “current state recognition” and “understanding the problems.” The “current state recognition” is common to both working phases and is divided into “source code investigation” for analyzing the structure of source code, and “measurement of elapsed time” to understand the current state of application performance. The final procedure in the “current state recognition” is “calculation/communication kernel analysis,” in which we evaluate the results of “source code investigation” and “measurement of elapsed time.” The next phase of “current state recognition” begins with “problem evaluation method” with working phases of “understanding the problems.” In the “problem evaluation method” for “highly parallel performance optimization,” the problems related to high parallelization are classified into six patterns. In the “problem evaluation method” for “single-CPU performance optimization,” applications are also classified into six patterns.

Our approach is summarized in Table 2.1.

K. Minami (✉)
RIKEN Center for Computational Science, RIKEN, Kobe, Hyogo, Japan
e-mail: minami_kaz@riken.jp

© Springer Nature Singapore Pte Ltd. 2019
M. Geshi (ed.), *The Art of High Performance Computing for Computational Science*, Vol. 2,
https://doi.org/10.1007/978-981-13-9802-5_2

Table 2.1 Outline of performance optimization method

	Highly parallel performance optimization	Single-CPU performance
Current state recognition	Source code investigation	
	Measurement of elapsed time	
	Calculation/communication kernel analysis	
Understanding the problems	Problem evaluation methods	

2.2 Current State Recognition: Source Code Investigation

As the first step in current state recognition, we investigate the source code of the application. We investigate the structure of the source code and analyze the call structure of subroutines and functions. We also analyze the subroutines, the loop structure in the functions, and the control structure of the IF blocks, and organize and visualize the structure of the entire program. The visualized source code is divided into blocks of calculation and communication processing according to the algorithms of physics and mathematics used in the program, and the blocks are organized. We understand the physical/mathematical processing content of each processing block. By comparing these aspects of the processing blocks with the results of the investigated source code, the calculation characteristics for each calculation block are obtained. The calculation characteristics describe the processing of a calculation block as non-parallel, completely parallel, or partially parallel, and identify the calculation index (e.g., number of atoms or number of meshes), whether the calculation amount in the calculation block is proportional to N or proportional to N^2 when the calculation index is N , and so on. We also investigate the communication characteristics of each communication block: whether the processing of the communication block is global communication, adjacent communication, or whether the communication amount depends on the calculation index. These investigations are shown in Fig. 2.1.

The purpose of the investigation of the source code is to understand the characteristics of each processing block in the program. However, the visualization of the loop structure from the start to the end of the program and that of the entire control structure of the IF blocks mentioned here are large tasks if done manually. Therefore, we use a visualization tool for program structure, such as K-scope [1, 2].

2.3 Current State Recognition: Measurement Methods

In the sequential calculation before parallelization in the simulation, the calculation is sequentially performed for each calculation unit such as a mesh. To parallelize a code, we divide a set of calculation units such as meshes into plurality sets, share the divided sets among the processors, and perform the calculations in parallel. In such parallel computation, adjacent communications are performed in every calculation

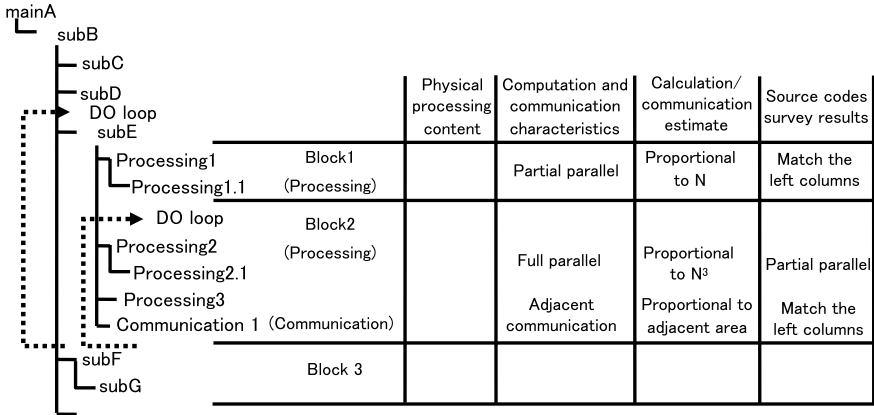


Fig. 2.1 Investigation of source code

step to exchange data for parts of areas with neighboring processors. In addition, when calculating inner products of scalar values for all areas, global communication between all processors is required. An important point in achieving high parallelism is to make the adjacent and global communication times as small as possible.

As described in Sect. 1.2.1, it is important in parallel computation, just like the reduction of communication time, to make the nonparallel computing parts as small as possible.

The next step of current status recognition is to conduct application performance measurement. It is important for these measurements to be useful for investigating parallel characteristics; that is, what kind of behaviors the adjacent and global communication times described here show during highly parallel calculation, and where nonparallel computing parts remain and their influence on behaviors in high parallelism. Therefore, where performance measurement is possible, it is carried out as follows.

In conducting application performance measurement as the next step of current status recognition, it is important to conduct performance measurements that clarify the parallel characteristics of applications: specifically, what kind of behaviors the adjacent global communication times display during the highly parallel calculation, which calculation parts are nonparallel, and how the nonparallel parts influence the application’s behavior in highly parallel execution. For clarification of parallel characteristics, where possible, the performance measurement is carried out as follows.

First, we define the problem to be solved, determine the number of parallel paths in the problem, and create a test problem that has the same problem size with one processor as the target problem that can be run with several levels of parallelism. Next, we perform the performance measurement using the prepared test problem. In the performance measurement, the execution time is measured for each process for each calculation block and communication block, as defined in the previous section. The parallel characteristics during parallel computation cannot be fully clarified by

measuring the entire application. Each processing block's influence on the parallel characteristics differs depending on whether it includes a nonparallel part and the number of parallel paths, and whether the communication time changes. Therefore, it is essential to measure the performance of each processing block for each process separately. These measurements allow us to identify the processing blocks that degrade parallel performance. In addition, because the communication behavior during parallel execution differs between adjacent and global communication, it is necessary to measure them separately. The adjacent communication time has the same value if the communication amount is the same, as described later, but the global communication time tends to increase as the number of parallel paths increases, even if the communication volume stays the same. Furthermore, because communication times may include waiting times caused by load imbalance, it is also important to measure the waiting time and the net communication time separately, thus allowing us to distinguish whether the problem is caused by communication or load imbalance. With respect to computation, simultaneously with the computation time, the amount of computation and the computation performance are also measured for each processing block in each process.

2.4 Current State Recognition: Determination of Computation and Communication Kernels

The analysis of the source code shows the correspondence between the physical/mathematical processing contents of each processing block and the source code, and the calculation characteristics of each calculation block and the communication characteristics of each communication block. By matching these results with measurement results, the calculation kernel and the communication kernel can be identified.

For example, suppose there is a parameter N that determines the amount of computation. Assume that the coefficient of computation amount proportional to the third power of N is m_1 , the coefficient of computation amount proportional to N is m_2 , and that m_2 is considerably larger than m_1 . When N is relatively small, the amount of computation for the two parts may be about the same. However, as N increases, the amount of computation for the part proportional to the third power of N becomes significantly larger, and the amount of computation for the part proportional to N may become negligible.

Both the amount of computation and the computation time also vary depending on the level of performance¹ that can be obtained relative to the theoretical peak performance. The essentially nonparallel parts may remain because of the adopted parallelization method. By considering the size of the parameters of the problem to be solved in this way, the parallelization method used, the parallelization method that may be adopted in the future, the prospects for effective performance, and so on,

¹Performance obtained by dividing the measured amount of computation by the execution time.

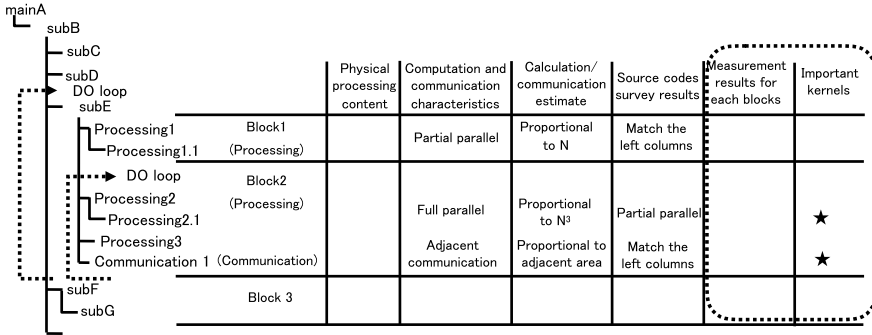


Fig. 2.2 Identifying kernel of calculation and communication

and the kernels to be evaluated, are determined (see Fig. 2.2). The kernels selected here can be reviewed at later stages of the evaluation.

2.5 Understanding the Problems: Evaluation of High-Parallelism Problems

We explain how to evaluate the problems of high parallelism by carrying out the measurements shown in Sect. 2.3 and how to measure parallel performance from several parallel processes to about 100, about 1000, or several thousand, step by step. There are two kinds of methods for measuring the performance by gradually increasing the number of parallel processes: strong scaling measurement and weak scaling measurement. Strong scaling measurement is a method of fixing the scale of the problem to be solved and increasing the number of parallel processes: for example, if the problem scale is fixed to $N = 10,000$, the number of parallel processes and the problem size per processor change is 1 and 10,000, 2 and 5000, 4 and 2500, and so on, respectively. In contrast, weak scaling measurement is a method of fixing the scale of the problem solved by each processor and increasing the number of parallel processes. For example, if the problem scale is measured first at $N = 1000$, the problem size per processor and the number of parallel processes is 1000 and 2, respectively, and the total problem scale is $N = 2000$. If the problem scale per processor and the number of parallel processes is 1000 and 4, the total problem scale is increased to $N = 4000$. The feature of weak scaling measurement is, ideally, that even when the number of parallel processes is increased, because the same computation is performed, and the adjacent communication amount is not changed, the execution time of the computation parts and the execution time of the adjacent communications are not changed. When a nonparallel part is included in the computation part, a significant increase in computation time should be measured, as the number of parallel processes becomes large in weak scaling measurement.

For example, assume that the execution time of the parallelizable part during sequential execution is T_p and the execution time of the nonparallelizable part during sequential execution is T_s . The execution time T_0 during sequential execution is represented by $T_0 = T_p + T_s$. The execution time when this problem is multiplied by N and executed sequentially is represented by $N \times T_0 = N \times T_p + N \times T_s$. When this problem is executed in N parallel processes, it corresponds to what we performed with weak scaling. If the execution time when executed in N parallel processes is T_{wn} , the parallelizable portion becomes N times faster but the nonparallelizable portion does not become faster, so $T_{wn} = T_p + N \times T_s$, and the term $N \times T_s$ increases. Incidentally, if T_{sn} is the execution time when run with strong scaling, then $T_{sn} = T_s + T_p/N$.

Even when the adjacent communication time increases in accordance with the number of parallel processes, it is easy to see that there are some problems in the corresponding adjacent communications. The global communication time generally increases in accordance with the number of parallel processes, and the increase can be predicted from the data on the basic communication performance by comparing the degree of increase with the predicted value. This can show whether there are some problems in the corresponding global communications.

The method described here is shown in Fig. 2.3. The reason for using weak scaling measurement in this way is that it is easy to find problems. However, in weak scaling measurement, it is necessary to prepare separate execution data according to the number of parallel processes, which may be troublesome. In a simulation in which the amount of computation is proportional to the second or third power of the problem size N , weak scaling measurement is sometimes difficult. In such a case, strong scaling measurement is performed. For strong scaling measurement, it is necessary to model the computation and communication times with the number of parallel processes as a parameter, to predict these, and to compare the predictions with the actual measured times so as to find any nonparallel parts or communication problems. However, unlike weak scaling measurement, it is not necessary to prepare

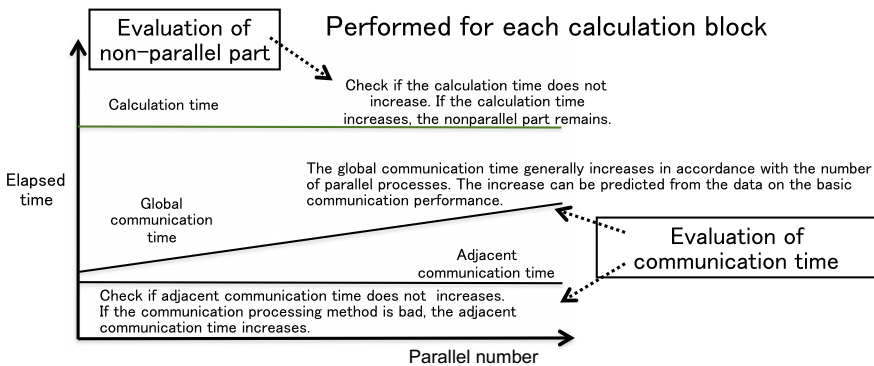


Fig. 2.3 Measurement with weak scaling

execution data according to the number of parallel processes. It is sufficient to prepare only one type of data.

For the calculation of kernel, we compare and evaluate the trend of the predicted computation amount as clarified in the investigation of the source code and the computation amounts from the measurement results. For example, assume the computational amount of the kernel is proportional to the problem size N and is completely parallelized and measured with weak scaling. Because the computational amount for each process is constant regardless of the number of parallel processes, the calculation time for the total system is constant. In this case, the value obtained by dividing the computational amount of the measurement result by N is the proportional coefficient. If it can be evaluated with weak scaling, as described above, and if it is possible to completely parallelize and there are no problems in the communication part, even if the number of parallel processes is increased, the execution time of the computation part will be constant, and the adjacent communication time will also be constant and should not increase. If the execution time of the computation part increases remarkably with the number of parallel processes, it is likely that some nonparallel parts remain in the operation kernel. In addition, if the adjacent communication time increases significantly according to the number of parallels, it is likely that some processing that is not adjacent communication is included in the communication kernel. For example, there may be some global communication that is used instead of adjacent communication to simplify programming. As for the global communication, as described at the beginning of this section, its communication time also increases as the number of parallel processes increases. However, because the extent of the increase can be predicted from the basic communication performance data, if the communication time increases significantly more than predicted, we can consider that there is some problem in the corresponding global communication. In the discussion of measurement methods, we described the method of measuring the communication time and waiting time separately. This measured waiting time often indicates some imbalance included in the computation part and communication part.

In parallel computing, some processing imbalance is physically unavoidable. However, where the extent of the imbalance is remarkably large or if the imbalance increases with the number of parallel processes, it is likely that some problem causing imbalance was introduced in the programming stage. In evaluations using strong scaling, as described above, the existence of nonparallel parts and communication problems are found by comparing the predicted computation and communication times with the measured times. The predicted times are obtained by modeling them with the parallel number N as a parameter.

For example, suppose that the computation of the kernel is proportional to the third power of the system parameter N and the computation kernel is completely parallelized. When measured with strong scaling, if the number of parallel processes is doubled, then the computation of each process should be halved.

The total computation amount is the number of processes multiplied by the measured computational amount of each process and the parallel number M . The total computational amount divided by the third power of M is the proportional coefficient for the third power of M . The investigation of these computational amounts

and proportional coefficients is performed using many parallel measurement results, and the evaluation is made as to whether the predicted value is consistent with the measurement results. If the evaluation results are consistent, it means that the source code is written according to the theory. If the evaluation results are not consistent and there is an increase in computational amounts with the increase in the number of parallel processes, it is likely that there is some problem such as the existence of nonparallel parts in the source code. A similar evaluation is required for the adjacent communications. For example, when a rectangular parallelepiped area is calculated using twice the number of parallel processes, the length of one side of the allocated area of each processor is $1/3$ to the power of $1/2$. The adjacent communication amount for the adjacent faces is $2/3$ to the power of $1/2$. Therefore, assuming the same communication performance, the communication time should also be $2/3$ to the power of $1/2$. For the global communication, a similarly modeled evaluation is required. As for the evaluation of the imbalance, it is necessary to evaluate the results as for weak scaling.

As repeatedly described, it is essential to carry out the evaluation shown here for each computation and communication kernel. Some tools provided by manufacturers have functions to measure the execution time, the amount of computation, and the computation performance for each subroutine or function, and it is usual to measure performance using these tools. However, the subroutines and the functions of the application do not generally match the range of the block, and because a function may be called from different blocks several times in different ways, these tools may not yield accurate measurement results for each block. Therefore, it is better to perform measurements on each block. However, this does not apply if the subroutine or function is configured to match the block.

2.5.1 Classification of Problems Related to High Parallelism

In the HPCC benchmark, applications are classified by using two axes. The first axis is defined by the locality versus nonlocality in the spatial direction of the data divided among the processors. The second axis is defined by the locality versus nonlocality in the temporal direction of data in the processors [3].

In addition, a study of application classification, the “Berkeley 13 dwarfs,” classified applications by the two axes of the communication and calculation patterns [4]. In this study, applications were classified among seven dwarfs in the HPC field, and 13 dwarfs by adding other fields.

In promoting performance optimization, we also classify the application and organize the execution performance optimization methods for applications based on the classification. For high parallelism, the locality versus nonlocality of the data is considered in the HPCC as one axis, and in the Berkeley 13 dwarfs, the pattern of communication is considered as one axis. In this section, we focus on the kinds of problems that occur and how we deal with those problems when optimizing the performance of existing applications, and we classify them according to highly parallel

patterns. The problems relating to high parallelism are classified into six patterns, as shown in Table 2.2.

The main problems relating to high parallelism are caused by calculations and communication. The first, second, and sixth problems are caused by calculation, and the third, fourth, and fifth problems are caused by communication. The six patterns are described as follows.

The first pattern is the mismatch of the degree of parallelism between applications and hardware. Researchers want to solve a problem within a certain time; suppose that to do so, it is necessary to use tens of thousands of parallel nodes on a super-computer. For example, the K computer makes it possible to use more than 80,000 parallel nodes in terms of parallelism of the hardware. However, sometimes only thousands of parallel nodes can be used because of the limitations of the application parallelization. This is the mismatch of degree of parallelism between the application and the hardware. When approaching the limitation of the parallelism of the application, the computation time becomes extremely small, whereas the proportion of communication time increases, leading to a deterioration of the parallel efficiency.

The second pattern is the presence of nonparallel parts. As mentioned at the beginning of this chapter, we can see that the parallel performance deteriorates because of Amdahl's law if nonparallel parts remain in the computation. Here, assuming that the execution time of a certain application at the time of sequential execution is T_s and the parallelization rate of the application is α , the nonparallelization ratio of the application is $1 - \alpha$. When this application is executed using n parallel processes, the execution time T_n is expressed as $T_n = T_s (\alpha/n + (1 - \alpha))$. For a parallelization efficiency of 50%, the parallelization ratio α is required to be 99.99% when $n = 10,000$. The easiest way to find remaining nonparallel parts is to measure the increase in execution time of the calculation part using weak scaling measurement as described above.

The third pattern is the occurrence of large communication sizes and frequent global communication. Communication times, particularly for global communications, have a large impact on the parallel performance. Consider an example of implementing the ALLREDUCE communication of M (bytes) between N nodes. Assume that the ALLREDUCE communication is performed using a binary tree

Table 2.2 Bottlenecks in parallel performance

	Bottleneck
1	Mismatch of the number of parallel processes between application and hardware (insufficient parallelism of applications)
2	Presence of nonparallel part
3	Large communication size and the occurrence of frequent global communication
4	Global communication among all nodes
5	Large communication size and the occurrence of communication times in adjacent communication
6	Load imbalances

algorithm and the communication performance is P_t (bytes/s). The communication time T_g for acquiring the total amount of M (bytes) after all nodes have communicated is $T_g = \frac{m \times \log_2 n}{P_t}$. To compare the global communications with the adjacent communications, we consider an example in which N nodes perform the adjacent communications of M (bytes) to the next rank. When the communication performance is matched with the above conditions, the communication time T_a to complete the communication of M (bytes) for all nodes is calculated by $T_a = \frac{m}{P_t}$. When comparing global and adjacent communications, it is found that the global communication time is larger by the coefficient of $\log_2 n$. Global communication should be a minimum.

The fourth pattern is the occurrence of global communications among all nodes. As described above, when the ALLREDUCE communication of M (bytes) is performed between N nodes using the binary tree algorithm, the communication time T is $T = \frac{m \times \log_2 n}{P_t}$ assuming the communication performance to be P_t (bytes/s). Because the communication time increases as the number of nodes N increases, it is better to limit global communication among all nodes as much as possible. However, calculation of inner products is inevitable in the iterative solution of simultaneous linear equations and other problems, so it is impossible to eliminate all-node global communication.

The fifth pattern is the occurrence of a large communication size and a large number of communications in the adjacent communication. In terms of the communication time, adjacent communication tends to be faster than global communication. However, useless adjacent communication, such as communicating data for the entire area for one mesh to the adjacent mesh, are sometimes performed. Such code should be reviewed and only communication of data on the adjacent surface should be made.

The sixth pattern is the occurrence of load imbalances. Differences in the amount of calculation for each node may occur, causing some load imbalance among nodes. When the load imbalance deteriorates as the number of nodes increases, or when the load imbalance is extremely large over a small number of nodes, it is a problem.

2.6 Understanding the Problems: Evaluation Methods for Problems in Single-CPU Performance

2.6.1 Application Classification for Single-CPU Performance

As mentioned in Sect. 2.5, the developers of the HPCC benchmark [3] and the Berkeley 13 dwarfs [4] classified applications. For the HPCC, applications were classified using locality versus nonlocality of data in the temporal direction with regard to the single-CPU performance. For the Berkeley 13 dwarfs, applications were classified using the calculation pattern.

Similarly, in promoting the study of performance optimization, we also classify applications and organize the application execution performance optimization techniques based on the classification. In Sect. 1.2, from the viewpoint of the single-CPU

performance, we mentioned that applications can roughly be classified into two types, one with a low required B/F value and one with a high required B/F value. This idea is close to the classification used for the HPCC. In this section, we will develop this view and show the classification of applications into six types as shown in Table 2.3.

The calculations for which the required B/F value is small are the first to the fourth types. The performance greatly varies depending on whether the DGEMM library or manual cache blocking can be used, even for calculations with small required B/F values. When cache blocking can be used, the performance varies depending on whether the data structure and loop structure are simple, or the data structure is slightly complicated such as using list vector indexing by integer arrays. Applications with more complex loop structures often fail to achieve high performance. These considerations led to the four types of calculations with small required B/F values.

The first type includes applications that can be rewritten as matrix–matrix product calculations. This type has small B/F values because in principle it can perform the calculations proportional to the third power of n by loading the data for a square of size n from memory. An example of this type of calculation is the application of the first principle quantum calculation based on density functional theory.

The second type includes applications that allow cache blocking although they are not rewritable to the matrix–matrix product, but still have small required B/F values. The calculation of the Coulomb interactions of molecular dynamics and the calculation of the gravity interaction of the gravitational multiple-body problem are examples. In both cases, by loading the data for n particles and performing cache blocking, calculations proportional to the square of n can be performed, so that the required B/F value is small. This type often uses list vector indexing by integer arrays for the particle access, and the loop body² is somewhat complicated.

The third type contains examples such as special high-precision stencil calculations,³ which make it possible to use the cache effectively, so the required B/F value

Table 2.3 Classification of applications from the standpoint of single-CPU performance

	Classification	Application examples
1	Rewritable to matrix–matrix products	Density functional theory calculations
2	Cache blocking is possible	Molecular dynamics, many-body gravity problems
3	The required B/F value is small, and the loop bodies are simple	Special stencil calculations
4	The required B/F value is small, but the loop body is complex	Plasmas, physical processes of meteorology, quantum chemical calculations
5	The required B/F value is large	Mechanical processes of meteorology, fluids, earthquakes, nuclear fusion
6	The required B/F value is large and list accesses are used	Structural calculations using finite-element methods, fluid calculations

²The code contained in the loop.

³The calculation using subscripts for differences such as $i, i - 1$ appearing in difference calculations.

is small and the loop body is a simple calculation. Although this type of calculation gives good performance, unfortunately there are few examples.

In the fourth type of calculations, the required B/F value is small, but the loop body is complex. Some weather calculations have mechanical processes to calculate the motion of a fluid and physical processes to calculate the microphysics of clouds; this physical process corresponds to the fourth type of calculation. By using small amounts of data loaded from the memory, complex and in-cache calculations are performed, but the loop body tends to be long and complicated. The calculation of the PIC method⁴ used for plasma calculations is also of this type. In this technique, although the mesh data around the particle are cached, list vector indexing by integer array is commonly used to access the particle data, resulting in complex program codes. The body of the calculation loop also tends to be long. For this type, we expect high performance because the data are cached, but in many cases we cannot obtain the expected performance because of the complexity of the program code.

The fifth and sixth types of calculation have high required B/F values. Even for program codes that have the same high required B/F values, the performance varies greatly, depending on whether discontinuous access to lists is required. This is the basis for classifying calculations with high required B/F values into the fifth and sixth types.

The fifth type of calculations has high required B/F values and do not use list accesses. There are many calculations of this type in the usual stencil calculation, and there are many other examples such as the dynamic processes in weather calculations described earlier, fluid calculations and calculations of earthquakes. The sixth type of computation has high required B/F values and uses list accesses. Such calculations occur frequently in engineering; examples are structural analysis and fluid calculations using finite-element methods. List accessing is the weak point for the modern scalar computer architecture because random accesses are required for each element.

In general, single-CPU performance decreases in the order from type 1 to type 6 calculations. However, there is usually little difference between types 2 and 3.

2.6.2 Evaluation by Cutting Out the Computation Kernel

First, we cut out the calculation kernel to form an independent test program that can be executed in one process. In cutting out the kernel, the following steps are carried out.

- (A) Dump the necessary data at the time of executing the original program to prepare the data such as arrays necessary for the execution of the test program. The data used by the conditional statements are important. For the data used for calculation, when only the performance is a problem, the appropriate data may be set without using dumped data.

⁴Particle-in-cell method. A method for arranging particles in the calculation lattice.

- (B) Dump the data for verification at the time of executing the original program to prepare the data such as arrays necessary for verification of the test program.
- (C) Cut out the kernel as the test program. If necessary, add a function to read the data from (A) to the test program and a function to write data to compare with the data dumped in (C).

The reason for cutting out the kernel in this way is that it makes it easier to handle in terms of execution time and the number of processes by allowing execution in one process. It is also easier to rewrite and test program code such as merging arrays and replacing indices.

Next, the various performance improvement techniques such as loop division, loop fusion, array merging, and replacement of array indexes can be tried using the cut-out kernel, and the effects of the changes are evaluated. When implementing a performance improvement technique throughout the original program, the amount of work may be increased. We identify work that affects the entire code when the performance improvement techniques are implemented, and estimate the cost of making those changes. We evaluate the trial results of the performance improvement technique from both aspects of the performance improvement and the work volume, and determine an accepted plan of performance improvement.

2.7 Performance Optimization Techniques Using Problem Patterns for High Parallelism

For each highly parallel problem pattern described in Sect. 2.5.1, we outline the applicable performance optimization techniques. Specific applications of the performance optimization techniques mentioned here are shown in Chap. 3.

2.7.1 *Mismatch of Parallelism Between Application and Hardware*

One applicable solution to the mismatch of parallelism between the application and the hardware is the extension of the parallelization axis. Some programs are parallelized using only one parallelization axis, even though it may be physically possible to adopt a plurality of parallelization axes. By extending this parallelization axis and parallelizing it with a plurality of parallelization axes, it becomes possible to expand the parallelism of the applications and to use more of the parallelism available in the hardware. However, the expansion of the parallelization axis often requires extensive rewriting of programs, and it is necessary to consider this increase in the programmer workload.

2.7.2 Remaining Nonparallel Part

If the parallelization adopted domain decomposition, many problems can be completely parallelized. However, as mentioned above, in spite of being physically able to adopt a plurality of parallelization axes, some programs are parallelized using only one parallelization axis. This may leave nonparallel parts in the program code. For example, suppose that there are two parallelization axes, A and B, physically, and the loop related to the B-axis is within the loop related to the A-axis. Of the two axes, if only the B-axis is parallelized, the processing between the loop related to the A-axis and related to the B-axis is not parallelized and nonparallel parts remain.

One way of dealing with the remaining nonparallel parts is to extend the parallelization axis. This makes it possible to parallelize all the processing of the loops concerning the A- and B-axes as described above. In the parallelization of sequential programs, it is important to create a solid parallelization design assuming full parallelization. An incomplete parallelization design will leave nonparallel parts in the parallel application.

2.7.3 Large Communication Size and Frequent Global Communication

As an example of the occurrence of large communication size and frequent global communication, consider an inner product with $N \times M$ elements distributed over all nodes when the number of nodes is N and each node has a vector consisting of M elements. For this processing, the inner-product processing of M vectors may be performed after performing the ALLREDUCE⁵ communication of M vectors between N nodes. In this case, if the ALLREDUCE communication for one element between N nodes is executed after calculating the sum of M elements in each node first, this can only be a global communication of one scalar element. In the former case, the global communication is an ALLREDUCE communication of a vector with M elements, and in the latter case it is an ALLREDUCE communication of one scalar element, which requires a shorter communication time. This kind of attention should be paid to such selection errors in the communication processing.

The global communication time may also be sharply reduced by extending the parallelization axis, as mentioned above. Figure 2.4 shows an outline of the improvement in global communication by expanding the parallelization axis. The upper part of Fig. 2.4 shows the state of the parallel execution of the original program. The whole space is parallelized by six processes, and global communication between the six processes occurs. The lower part of Fig. 2.4 shows the state of the parallel execution of the program after expanding the parallelization axis to the two axes of the space and energy band. It can be seen that the range of global communication of

⁵The global communication for holding calculation results such as sum of values of all nodes in the group of data of each node.

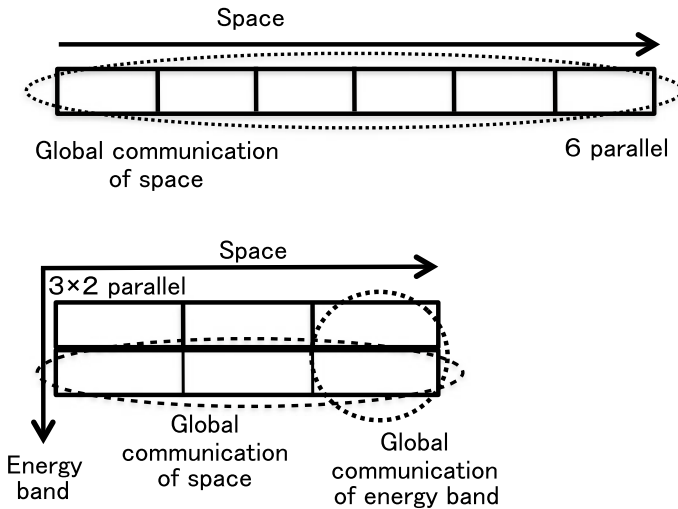


Fig. 2.4 Improvement effect of global communication by expanding the parallelization axis

the space is reduced from six processes to three processes. In principle, expanding the parallelization axis converts the global communication among n processes to that among $n^{\frac{1}{2}}$ processes, which allows a sharp reduction in the global communication space.

We now consider the case in which a BROADCAST communication⁶ is performed for N processes. If the message length to be communicated is M (bytes), the communication performance is b (bytes/s), and the time for one communication is T_0 ; this is represented by $T_0 = M/b$. Assuming the BROADCAST communication uses the binary tree algorithm, the communication time of the entire BROADCAST is $T_0 \times \log_2 N$. By expanding the parallelization axis to two axes, global communication between processes of $N^{\frac{1}{2}}$ is performed, so the communication time is $T_0 \times \log_2 N \times N^{\frac{1}{2}} = (1/2) \times T_0 \times \log_2 N$, and the communication time is halved.

This tendency appears more prominently with ALLTOALL communication. Consider the case in which the ALLTOALL communication is performed using a two-dimensional torus. Let N be the total number of nodes, n be the number of nodes in each dimension, and let $N = n \times n$. Assuming that the message length that each node communicates with one node of the communicating partner is m (bytes), the message length communicated by each node is $M = mN$. The N nodes are divided into two on the left and right, and the amount D to be transferred to the adjacent region in each region is calculated. Each side includes $(n/2) \times n$ nodes. Because the communication is performed between all the nodes included both left and right, the amount of data to be transferred to the neighboring area is $(n/2) \times n \times (n/2) \times n \times m = n^2 \times (n/2)^2 \times m$. This amount is transferred to left and right in two directions, so the data amount D traversing between the boundaries divided into two

⁶The global communication that transmits data from one node to other nodes in the group.

is represented by $D = 2 \times n^2 \times (n/2)^2 \times m$. If each node can communicate in four directions simultaneously, the bisection bandwidth⁷ BY of the system is represented by $4 \times n \times b$, where b is the communication bandwidth for one direction of one node. When calculating the time t required for the communication, t is represented by D/BY , so that $t = (1/8) \times (n^3/b) \times m = (1/8) \times N \times n \times (m/b) = (1/8) \times M \times (n/b)$. Thus, t is proportional to Mn . The message length M when executing the ALLTOALL communication in the entire N node is calculated as $M = mN = mn^2$. However, in principle, when the parallelization axis is expanded to two axes, it is possible to calculate the message length M of ALLTOALL communication as the message length for one axis. Therefore, the message length M is calculated as $M = mN^{1/2} = mn$.

We see that the communication time can be reduced from the order of n^3 to the order of n^2 . The discussion described here can also be applied to the communications GATHER, ALLGATHER, SCATTER, and so on, of the type in which the message length of the communication is $M = mN$.

Although the effect of parallelization of multiple axes has been described here, this must be applied to all parts of the application. For example, suppose that the application has two processing blocks, A and B, and that two parallelizing axes of a and b can be adopted. Let us assume that both A and B contain calculations that depend on a and b . Under this assumption, the parallelization of the N nodes with respect to the a -axis (b -axis) is carried out for part A (B). This is expressed as A (a/N , b), B (a , b/N). Then, between processing block A and processing block B, the communication for returning a/N to a and for dividing b into b/N occurs. This usually requires global communication between N nodes, which causes a large performance degradation.

2.7.4 All-Node Global Communication

The calculation of the inner product appearing in the iterative method of solving simultaneous linear equations cannot be avoided. Therefore, the global communication by ALLREDUCE of all nodes cannot be eliminated. We should therefore use the high-speed ALLREDUCE communication provided by the hardware assistance available in the K supercomputer and others.

For other communications, it is effective to adopt multiple parallelization axes as described in Sect. 2.7.4. The reason is that the range of global communication can be changed from all nodes to the square root of the number of nodes, as explained above.

⁷The total bandwidth of the communication across the division plane that divides all nodes included in the system into two.

2.7.5 Large Communication Size in Adjacent Communication, Communication Frequency

Even though an algorithm may only need to communicate the physical quantities of the adjacent faces between the nodes, sometimes all the physical quantities within the node are communicated because this simplifies the coding. If this is found, the code should be rewritten, and communications should be restricted to the adjacent surfaces.

In preprocessing for simultaneous linear equation solvers by the iterative method, the adjacent communication time may increase by increasing the number of communications between adjacent nodes. Methods using preprocessing localized in the node are effective for reducing this; however, it is necessary to guard against an increase in the number of iterations.

2.7.6 Load Imbalances

Differences in the amount of calculation for each node may occur, leading to some load imbalance among nodes. There are no general countermeasures for large load imbalances. The problem parts must be specified by the method shown in Sect. 2.5, and the causes are investigated. Various examples of load imbalance are conceivable. For example, an imbalance in the number of atoms between nodes may be caused by atomic motion in molecular dynamics calculations. Similarly, particle motion in the PIC method may produce imbalances in the number of particles between nodes. An imbalance between nodes may occur because of the amounts of microsubstances constituting a cloud in the physical process calculations for climate prediction. To deal with these imbalance problems, the adaptive mesh method⁸ or a similar approach can be adopted.

2.8 Performance Optimization of Single-CPU Performance

2.8.1 Elements for Obtaining High Single-CPU Performance

To realize high single-CPU performance, it is important to achieve thread parallelization. Once this has been established, the factors considered important for improving the single-CPU performance are shown as follows:

(1) Effective utilization of prefetching

Prefetch is a function that reads data to cache memory before it is required.

Latency is the time of initial processing from the start of data access until the

⁸A method of dynamically changing the mesh division.

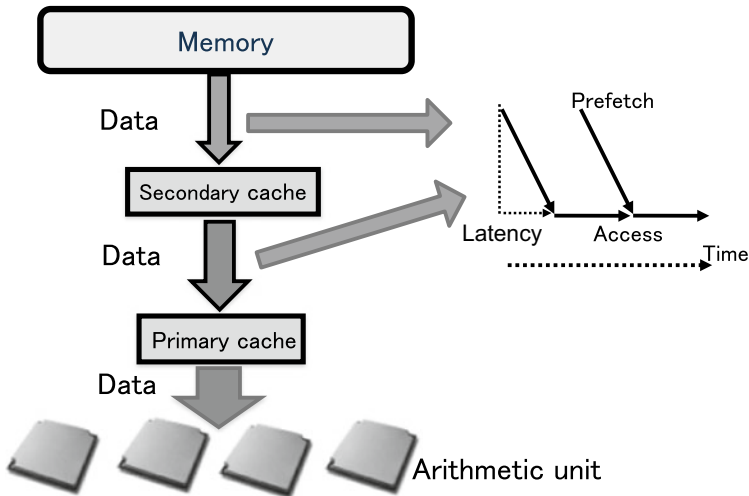


Fig. 2.5 Effective utilization of prefetch

actual data access is started. In accesses from memory to the L2 cache and from the L2 cache to the L1 cache, as shown in Fig. 2.5, effective prefetch operations are important for achieving high-performance data accessing. When a load instruction is executed without using a prefetch mechanism, the access to memory and L2 cache causes a large latency penalty (see Fig. 2.5). If the latency to the L1 cache is 1, the latency to the L2 cache from the L1 cache is typically 10 and the latency to the memory from the L2 cache is typically 100. In addition, if there is more computation than data accesses, not only the memory accessing latency but also the access to memory itself may be hidden by using prefetch mechanisms.

(2) Effective use of line accesses

In the CPU of the K computer, the data are accessed in line (128 bytes) units between the memory and the L2 cache. To obtain high performance, it is important to perform computations using as much as possible of each line of the data loaded. If only eight bytes from each line can be used, it is necessary to access the data of 16 lines to load 16 elements, which is a large penalty over using all 16 elements in one line. Therefore, the apparent memory access performance would be 1/16 (Fig. 2.6).

(3) Effective use of the cache

Consider the matrix–matrix product calculation of $(n \times n)$ matrices. The number of elements of the two matrices is $2n^2$ in total and the number of computational operations is $2n^3$ in total for the product and the sum. If the $2n^2$ elements are divided into multiple smaller sets of n^2 and each such element set is in the cache when its computations are executed, $2n^2$ pieces of data are reused and $2n^3$ computations can be executed. Such calculations can utilize the cache effectively and can perform high-speed calculations. In principle, for calculations that can

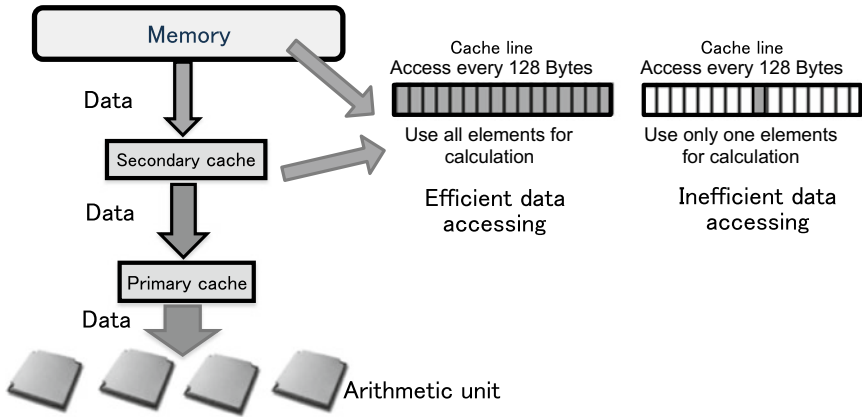


Fig. 2.6 Effective utilization of cache line

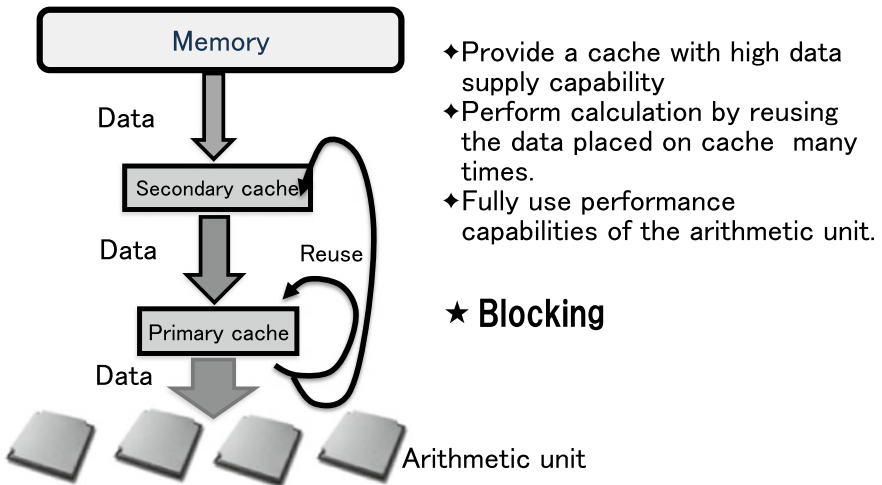


Fig. 2.7 Effective utilization of cache memory

perform n^2 computational operations using n data (or n^3 computational calculations using n^2 data), high-speed calculation is possible (Fig. 2.7). As mentioned earlier, this technique is called cache blocking.

Although their capacity is much smaller than that of the L1 cache, there are registers that provide high-speed data storage locations close to the computing unit. There is thus a similar technique called register blocking in which data are blocked in the registers, but the description is omitted here.

(4) Efficient instruction scheduling

The K computer is equipped with 256 floating-point registers. It is important for performance improvement that the compiler schedule load, arithmetic, and

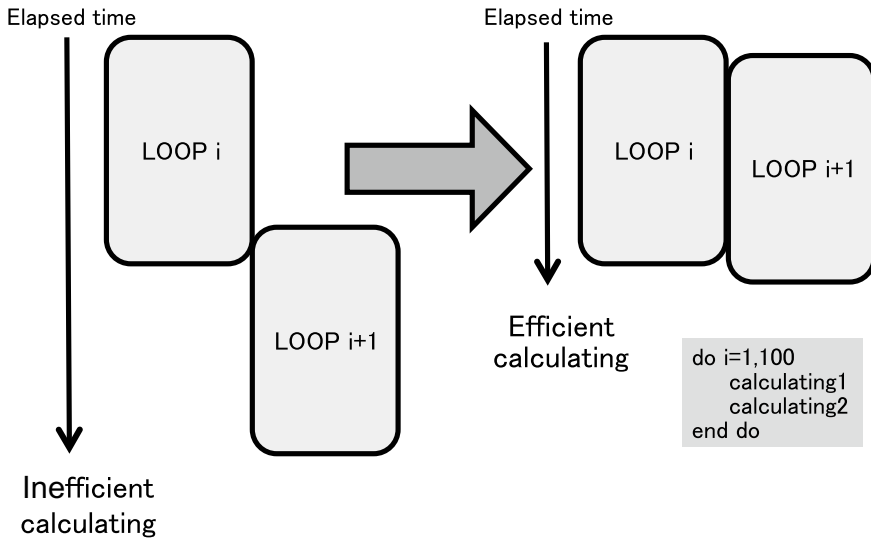


Fig. 2.8 Efficient instruction scheduling

store instruction belonging to different indexes of the loop index direction use these floating-point registers effectively (Fig. 2.8). If the compiler cannot find a good schedule, the performance may be improved by manually performing loop division or loop expansion.

(5) Effective use of SIMD arithmetic units

The K computer CPU core has two sets of two product–sum computing units. The product–sum operation (2 operations) \times two product–sum computing units \times two sets allows a total of eight operations with one clock cycle. Because the operation clock of the CPU is 2 GHz, 8 operations \times 2 GHz = 16 G calculation (16 GFLOPS) is the calculation peak performance per second in one core. Each of the two product–sum computing units operates as a SIMD arithmetic unit having a vector length of 2. Therefore, to realize high-performance computation, it is important that the product–sum operation works as SIMD, and that the two SIMD units operate simultaneously (Fig. 2.9).

2.8.2 Relationship Between Factors for High Performance and Required B/F Values

From the viewpoint of single-CPU performance in Sect. 1.2, we showed that applications can be classified into two types in which the required B/F value is large or small. For each of these types, we show the relationship between the type and five elements for obtaining the high performance shown in Sect. 2.8.1.

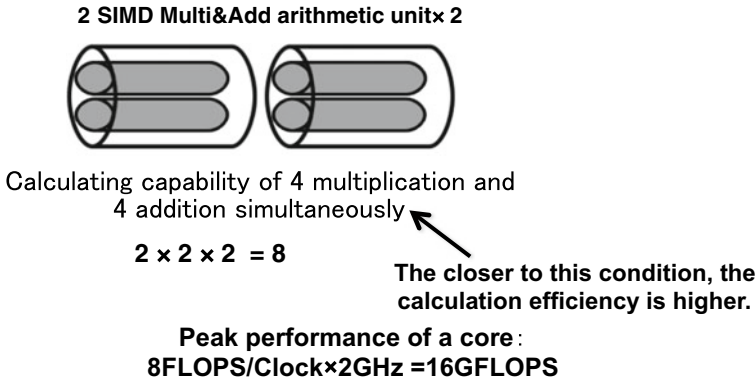


Fig. 2.9 Effective utilization of SIMD unit

For applications with small required B/F values, high memory data transfer performance (memory bandwidth) is not necessary in principle. It is most important to code the program to cache the data as shown in point (3) above. Next, because the L2 cache is accessed line by line, coding the program to utilize the data on each line of the L2 cache effectively (that is, point (2) above) is important (it is not necessary if the data are in the L1 cache). After realizing points (2) and (3), points (4) and (5) become important.

For applications with large required B/F values, it is important that the memory bandwidth should be utilized as fully as possible. It is more important to use the full memory bandwidth than to maximize the CPU computing performance. For these applications, the most important points are (1) and (2) above for obtaining the high performance shown in Sect. 2.8.1. It may be possible to reuse some cached data even though most data are accessed from memory, in which case point (3) becomes important. When points (1) to (3) are satisfied and the data necessary for calculation are supplied to the computing unit effectively, it is important to achieve efficient instruction scheduling in order to use these data efficiently and make effective use of the SIMD arithmetic units (points (4), (5)).

As seen here, the method required for the performance tuning varies depending on the required B/F value.

2.8.3 Thread Parallelization Common to Large and Small Required B/F Values

Regardless of whether the required B/F value is large or small, thread parallelization is indispensable for improving the single-CPU performance. Thread parallelization is impossible if there is a dependency relationship in the target loop, so the coloring method has been studied to eliminate dependencies. However, the coloring method

sometimes leads to the deterioration of single-CPU performance by increasing the number of iterations. The coloring method is explained in detail in Chap. 3.

2.8.4 When the Required B/F Value Is Small: DGEMM Conversion

As in the Gram–Schmidt orthogonalization shown below, it is sometimes possible to rewrite processes that use matrix–vector products in the normal coding to use matrix–matrix products. As described in Sect. 2.6, because the required Gram–Schmidt processing of the matrix–vector product program is large, high single-CPU performance cannot be expected. However, if it can be rewritten as a matrix–matrix product, the required B/F value can be reduced, and higher single-CPU performance can be expected. By using this matrix–matrix product as the mathematical library BLAS level 3 subroutine DGEMM, it becomes possible to calculate with very high performance optimized for each computer [5].

If we code the Gram–Schmidt orthogonalization algorithm straightforwardly, it becomes an algorithm to calculate the vector $(\psi'_1, \psi'_2, \dots, \psi'_i, \dots)$ as the orthogonalization of the vector $(\psi_1, \psi_2, \dots, \psi'_i, \dots)$, as shown in Fig. 2.10. This algorithm is coded with the matrix–vector product of the matrix $\langle \psi'_i | \psi_j \rangle$ and vector $|\psi'_i\rangle$. The indices 1, 2, ..., i are parallelized for each process.

Consider dividing this algorithm into a triangular part and a rectangular part, as shown in Fig. 2.11. Each rectangular part in the direction $(\psi_1, \psi_2, \dots, \psi'_i, \dots)$ is parallelized for each process. First, in the process responsible for the triangular part, we use the original matrix–vector product algorithm. In the example in Fig. 2.11, the calculation of triangles for calculating ψ_1 and ψ_2 corresponds to this explanation. Using the calculated (ψ_1, ψ_2) data and calculating the square part in

$$\begin{aligned}
 \psi'_1 &= \psi_1 \\
 \psi'_2 &= \psi_2 - \langle \psi'_1 | \psi_2 \rangle \psi'_1 \\
 \psi'_3 &= \psi_3 - \langle \psi'_1 | \psi_3 \rangle \psi'_1 - \langle \psi'_2 | \psi_3 \rangle \psi'_2 \\
 \psi'_4 &= \psi_4 - \langle \psi'_1 | \psi_4 \rangle \psi'_1 - \langle \psi'_2 | \psi_4 \rangle \psi'_2 - \langle \psi'_3 | \psi_4 \rangle \psi'_3 \\
 \psi'_5 &= \psi_5 - \langle \psi'_1 | \psi_5 \rangle \psi'_1 - \langle \psi'_2 | \psi_5 \rangle \psi'_2 - \langle \psi'_3 | \psi_5 \rangle \psi'_3 - \langle \psi'_4 | \psi_5 \rangle \psi'_4 \\
 \psi'_6 &= \psi_6 - \langle \psi'_1 | \psi_6 \rangle \psi'_1 - \langle \psi'_2 | \psi_6 \rangle \psi'_2 - \langle \psi'_3 | \psi_6 \rangle \psi'_3 - \langle \psi'_4 | \psi_6 \rangle \psi'_4 - \langle \psi'_5 | \psi_6 \rangle \psi'_5 \\
 \psi'_7 &= \psi_7 - \langle \psi'_1 | \psi_7 \rangle \psi'_1 - \langle \psi'_2 | \psi_7 \rangle \psi'_2 - \langle \psi'_3 | \psi_7 \rangle \psi'_3 - \langle \psi'_4 | \psi_7 \rangle \psi'_4 - \langle \psi'_5 | \psi_7 \rangle \psi'_5 - \langle \psi'_6 | \psi_7 \rangle \psi'_6 \\
 \psi'_8 &= \psi_8 - \langle \psi'_1 | \psi_8 \rangle \psi'_1 - \langle \psi'_2 | \psi_8 \rangle \psi'_2 - \langle \psi'_3 | \psi_8 \rangle \psi'_3 - \langle \psi'_4 | \psi_8 \rangle \psi'_4 - \langle \psi'_5 | \psi_8 \rangle \psi'_5 - \langle \psi'_6 | \psi_8 \rangle \psi'_6 - \langle \psi'_7 | \psi_8 \rangle \psi'_7 \\
 \psi'_9 &= \psi_9 - \langle \psi'_1 | \psi_9 \rangle \psi'_1 - \langle \psi'_2 | \psi_9 \rangle \psi'_2 - \langle \psi'_3 | \psi_9 \rangle \psi'_3 - \langle \psi'_4 | \psi_9 \rangle \psi'_4 - \langle \psi'_5 | \psi_9 \rangle \psi'_5 - \langle \psi'_6 | \psi_9 \rangle \psi'_6 - \langle \psi'_7 | \psi_9 \rangle \psi'_7 - \langle \psi'_8 | \psi_9 \rangle \psi'_8 \\
 &\vdots
 \end{aligned}$$

Fig. 2.10 Original processing of Gram–Schmidt orthogonalization

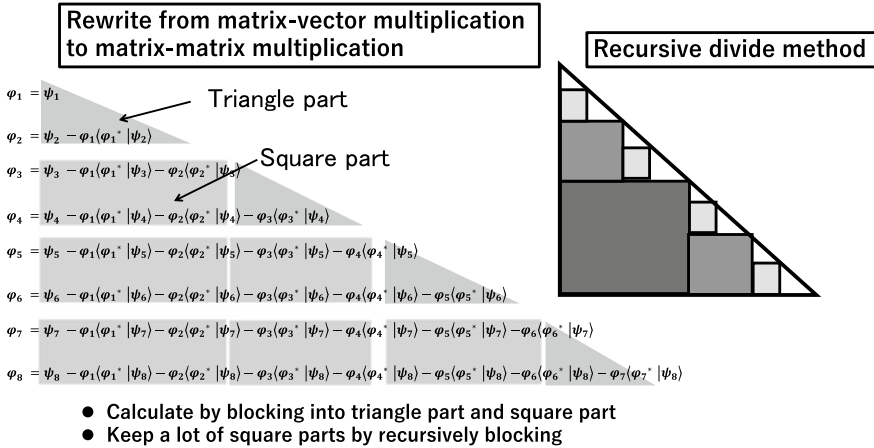


Fig. 2.11 Matrix–matrix productization of Gram–Schmidt orthogonalization

each process, this calculation can use the matrix–matrix product. In the example in Fig. 2.11, the calculation of the square in the left column using ψ_1 and ψ_2 corresponds to this explanation. The second triangular part is calculated using the original matrix–vector product algorithm. Similarly, ψ_3 and ψ_4 can be calculated perfectly in the example of Fig. 2.11. By repeating this process, it becomes possible to calculate the Gram–Schmidt orthogonalization using the matrix–matrix product. In the actual algorithm, the triangular part is recursively divided into a rectangular part and a triangular part as shown in Fig. 2.11, so that the matrix–matrix product calculation can be used as much as possible.

2.8.5 When the Required B/F Value Is Small: Cache Blocking

As described in Sect. 2.6, high single-CPU performance can be obtained by performing cache blocking, as in the matrix–matrix multiplication. The Coulomb force calculation in classical MD is one example. In this calculation, however, discontinuous list access is required to specify the particle pair, which degrades the single-CPU performance. Changing the blocking by rearranging the discontinuous data using the following procedure improves the single-CPU performance. This procedure is illustrated in Fig. 2.12.

- (1) Copy the N discontinuous data to a continuous area to enable blocking in the cache.
- (2) Perform the calculation N^2 times using N pieces of data.
- (3) Copy and return continuous calculated results to the discontinuous area.

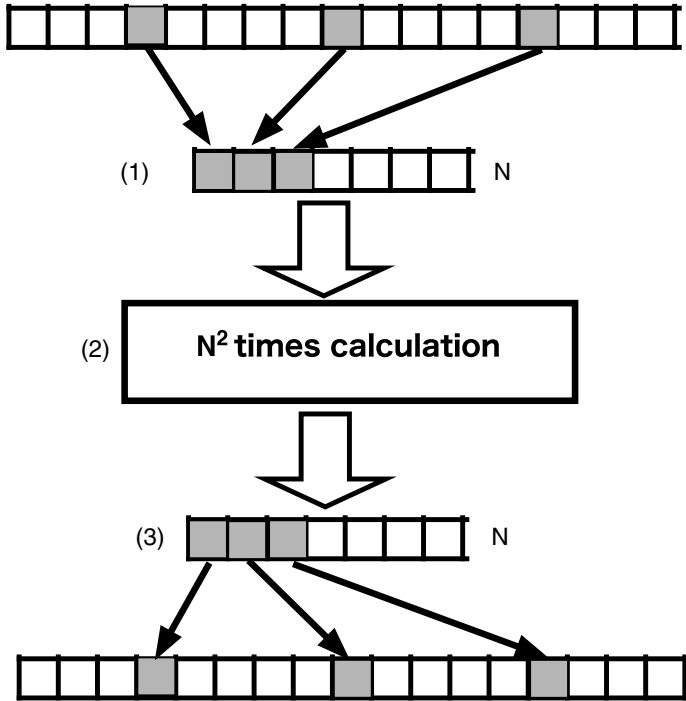


Fig. 2.12 Cache blockzation

In general, the copies of (1) and (3) require order-N processing, so the processing time is smaller than the computation time of order N^2 . Therefore, even if the copy is troublesome, a reduction in total execution time may be obtained.

2.8.6 When the Required B/F Value Is Small and the Loop Body Is Complex

When the loop body of a code is complex, the compiler may not be able to correctly interpret the required processing. In this case, more arrays are used in the loop, so more registers corresponding to the arrays are required, and there is a tendency for register shortage to occur. For these reasons, this type of application often fails to provide adequate performance even when the data are in the cache. There is no general solution for this problem, and we can only rewrite the code to achieve loop division, array index replacement, or array integration individually for each application.

2.8.7 Optimizing the Single-CPU Performance: When the Required B/F Value Is Large

The procedure for improving the single-CPU performance is shown in Fig. 2.13.

- (1) Profiler measurements
 Modern supercomputers are equipped with a profiler for acquiring performance information. The K computer has an excellent profiler function that can acquire abundant and useful information relating to the memory, cache, and arithmetic unit. This profiler should be used to measure the performance of the application kernel.
- (2) Detection of problems using the profiler’s measurement results
 Analyze the problem by looking at the profiler measurements. As described in Sect. 2.8.2, in applications that cannot efficiently utilize the caches, the important point for achieving high single-CPU performance is to use the memory performance fully. The profiler information includes the memory bandwidth used, and we can analyze the cause of the problem if the memory performance is poor.
- (3) Performance estimation
 Estimate the performance of the kernel using the performance estimation model (see Sect. 2.8.8). If the actual measurement results do not reach the estimation results, extend the analysis of step (2). Performance estimation is very important to judge the extent of tuning work required and where to stop.

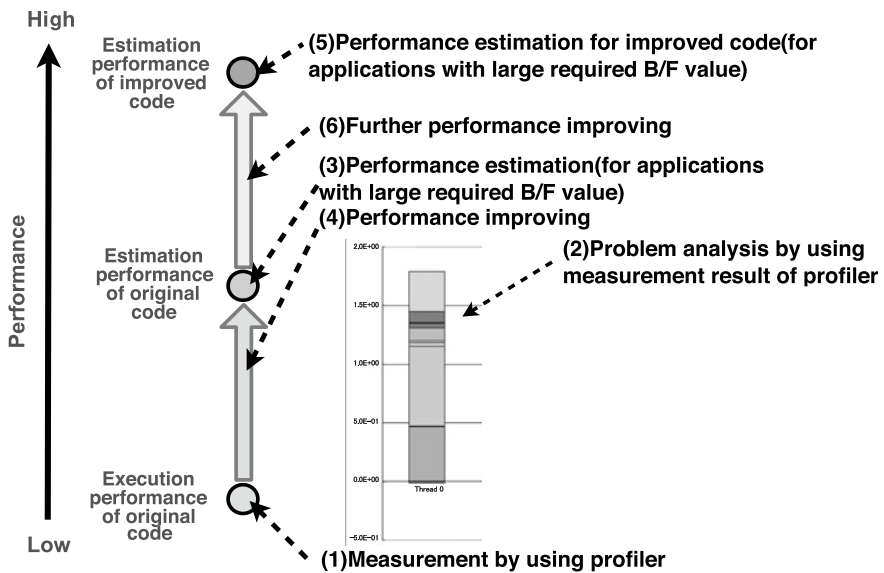


Fig. 2.13 Steps for improving performance

- (4) Performance tuning
We consider the available performance improvement methods and perform tuning work based on the results of the analysis of step (2). Work continues until the estimated performance is achieved.
- (5) Predict the performance of the improved version
Consider further methods of performance improvement and perform more tuning work. The performance of the kernel after further tuning is estimated according to the roof-line model (see Sect. 2.8.8).
- (6) Further performance tuning
Measure performance and analyze the results for the revised code after tuning. This cycle is repeated until the estimated result is approached.

2.8.8 Performance Estimation: When the Required B/F Value Is Large

Here, the performance estimation method when the required B/F value is large is described. The roof-line performance prediction model is used when data access is limited to the memory accesses and data access from the cache memory is not considered [6].

We define the theoretical memory bandwidth of the hardware as B , and the theoretical peak performance is defined as F . If the operational intensity of the application using the required FLOP value f of the application and the required byte value b of the application is $X = f/b$, the effective performance of the application is represented by $\min \{F, BX\}$. The operational intensity is the reciprocal of the B/F value. In this performance estimation model, an application that has the same operational intensity as the hardware can achieve peak performance. If the predicted performance of the application requires an operational intensity smaller than the operational intensity of the hardware, the model is proportional to the operational intensity required by the application. What we have described here is shown in Fig. 2.14. The performance of a memory-intensive application with a high required B/F value can be estimated

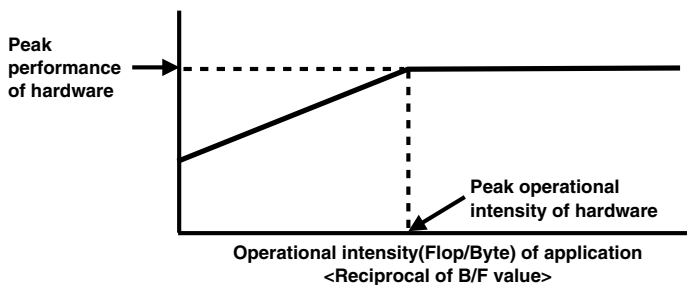


Fig. 2.14 Roofline model

more accurately by replacing the theoretical memory bandwidth of this model with the effective memory bandwidth [7].

2.8.9 Performance Optimization of the Sparse Matrix–Vector Product: When the Required B/F Value Is Large

A typical application with high required B/F value is the sparse matrix–vector product. This calculation often appears when discretizing partial differential equations. The features of the sparse matrix–vector product are shown below.

In general, when solving physical three-dimensional problems, the coefficients are three-dimensional and occupy sparse matrices to form three-dimensional arrays. However, these arrays may be expressed as one-dimensional or two-dimensional arrays for coding. In either case, the required memory capacity is large, so it is common to consume the memory bandwidth. However, the coefficient matrices may also be expressed not in three dimensions but in one- or two-dimensional arrays, or the coefficients may be represented as scalar quantities. In these cases, the calculation of the product of a sparse matrix and a vector does not consume as much memory bandwidth and the arrays can be stored in caches or registers. While the vectors are generally three-dimensional arrays when solving three-dimensional problems physically, they may, like the sparse matrices, be expressed as one- or two-dimensional arrays to simplify coding. An important feature of the vector arrays is that there is M or around M reusability when the average number of elements included in each row of the matrix is M and the dimension of the vector is L . Because the number of operations for the sparse matrix–vector products is $M \times L$ for each addition and multiplication, and the number of elements of vectors being used for performing operations is L , one element of the vector is referred to on average M times. Therefore, the memory bandwidth capacity of the vector is about $1/M$ of the memory bandwidth capacity of the matrix. Although access to the vectors also consumes memory bandwidth, the efficient utilization of the cache using the M reusability shown here is important for improving single-CPU performance.

2.8.10 Performance Optimization of the Sparse Matrix–Vector Product: Required B/F Value Is Large and List Vectors Used

For the sparse matrix–vector product discussed in Sect. 2.8.9, a stencil calculation in which the vector is continuously accessed is assumed. In addition to having the high required B/F value, there are applications for which the vectors are accessed using a list. In this case as well, because the vectors are reusable, it is important to make effective use of the cache. However, because the vector is not continuously accessed,

it is difficult to use cache memory effectively. For such applications, the cache can be utilized more efficiently by rearranging the order of the discrete points according to their physical positions, as discussed in the section on FFB in Chap. 3.

Exercises

1. Show that numerical solutions of differential equations by discretization are obtained using simultaneous linear equations, where A is a square matrix and x and b are vectors:

$$Ax = b. \quad (2.1)$$

2. Show that the simultaneous linear Eq. (2.1) can be transformed to:

$$x = Bx + b, \quad (2.2)$$

where B is a square matrix. When considering x^0, x^1, x^2 , that satisfy Eq. (2.3), if the column of vectors converges to vector x , show that the vector x satisfies Eq. (2.1):

$$x^{(m+1)} = Bx^{(m)} + b \quad (2.3)$$

3. The method of solving Eq. (2.2) as Eq. (2.3) is called an iterative method. One such method is the Jacobi method. If the equation to be solved is (2.1), the formula for the Jacobi iterative method is expressed as (2.4) and (2.5).

$$x^{(m+1)} = D^{-1}(E + F)x^{(m)} + D^{-1}b \quad (2.4)$$

$$a_{ii}x_i^{(m+1)} = - \sum a_{ij}^{(m)} + b_j$$

$$x_i^{(m+1)} = - \sum_{j=1, j \neq i}^n \left(\frac{a_{ij}}{a_{ii}} \right) x_{ij}^{(m)} + \frac{b}{a_{ii}}, \quad (2.5)$$

where $D, E,$ and F contain the diagonal elements of A , the lower triangular matrix of A , and the upper triangular matrix of A , respectively. Derive (2.4) and (2.5), which are expressions of the Jacobi method.

4. Code the Jacobi method in an appropriate programming language and parallelize it with MPI and/or OpenMP and investigate the parallelization efficiency. (The code for the nonparallel version of the Jacobi method can be found in textbooks or on several websites.)

References

1. M. Terai, E. Tomiyama, H. Murai, K. Minami, M. Yokokawa, in *Third International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI2012)*, vol. 434 (2012)
2. M. Terai, E. Tomiyama, H. Murai, K. Kumahata, S. Hamada, S. Inoue, A. Kuroda, Y. Hasegawa, K. Minami, M. Yokokawa, in *Proceedings of High Performance Computing and Computational Science* (in Japanese), vol. 84 (2013)
3. <http://icl.cs.utk.edu/hpcc/>
4. K. Asanović, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, Tech. Rep. of UCB/EECS-2006-183 (2006)
5. T. Yokozawa, D. Takahashi, T. Boku, M. Sato, in *Proceedings of Fourth International Workshop on Parallel Matrix Algorithms and Applications (PMAA'06)*, vol. 37 (2006)
6. S. Williams, A. Waterman, D. Patterson, *Commun. ACM* **52**, 65 (2009)
7. K. Minami, S. Inoue, S. Shigenobu, T. Maeda, Y. Hasegawa, A. Kuroda, M. Terai, M. Yokokawa, in *Proceedings of High Performance Computing and Computational Science* (in Japanese), vol. 23 (2012)

Chapter 3

Case Studies of Performance Optimization of Applications



Kazuo Minami and Kiyoshi Kumahata

Abstract In 2006, 7-year's project of MEXT (the Ministry of Education, Culture, Sports, Science and Technology) of the development and utilization of state-of-the-art high-performance supercomputers usually called the next-generation supercomputer project started. RIKEN undertook its main development, and the name "K computer" was determined. At the end of September 2010, the first K computer enclosure was carried into a building in Kobe, and the installation of the entire system was completed by September 2011 [1]. The K computer was completed in June 2012 after adjustment and improvement of the system software, and public use began in September 2012. Prior to the operation of the K computer, RIKEN had begun developing a set of tuned applications to demonstrate the system performance. We began programming and developed the high-performance computing technique to achieve high parallelization and fully utilize the enhanced functions introduced in the processor. The author was engaged in this application development work as a team leader. In this chapter, we first outline the set of tuned applications to demonstrate the system performance of the K computer. Next, we outline the K computer system, which is the premise of the subsequent section. Concrete examples of the applications mentioned in Chap. 2 are then described.

3.1 Applications for Demonstration of the Performance of the K Computer

3.1.1 Outline of Application Groups

The application group for the performance demonstration of the K computer was selected to demonstrate that applications in various fields can have a high performance by making full use of the versatility of the K computer. It also demonstrated

K. Minami (✉) · K. Kumahata
RIKEN Center for Computational Science, RIKEN, Kobe, Hyogo, Japan
e-mail: minami_kaz@riken.jp

© Springer Nature Singapore Pte Ltd. 2019
M. Geshi (ed.), *The Art of High Performance Computing for Computational Science*, Vol. 2,
https://doi.org/10.1007/978-981-13-9802-5_3

computer science characteristics that could be useful for future computer development. The term “computer science characteristics” as used herein means, specifically, two aspects:

- (a) in the parallelization, whether it is easy to obtain high-parallelization performance with relatively simple parallelization methods, or whether complicated methods must be used;
- (b) in the single-CPU performance, whether it is difficult or easy to obtain high single-CPU performance because of the high or low required B/F value of the application, and so on.

Using these two issues as a basis, we evaluated target applications, adding the evaluation based on the analysis of parallelization characteristics described later, and selected several applications that could be expected to result in high-parallelization performance. The selected applications are listed in Table 3.1. Specifically, we selected two applications in the earth sciences field (NICAM [2] and Seism3D [3, 4]), two in the nanoscience field (PHASE [5] and RSDFT [6]); one in the engineering field (FrontFlow/Blue (FFB) [7, 8]); and one application in the basic physics field (LatticeQCD [9]) for a total of six applications. For our purposes, these have the following characteristics.

(A) Earth sciences field (NICAM, Seism3D)

- (a) For high parallelization, a relatively simple domain decomposition method is used. Because most communications are adjacent, high-parallel performance tends to be obtained relatively easily, even at high parallelization.
- (b) For single-CPU performance, the Earth Simulator (vector-parallel computer), shows high performance of about 40% peak performance ratio and tends to require high memory bandwidth performance. On the other hand, for scalar parallel computers with relatively low memory bandwidth performance relative to the computational peak performance, these applications require careful programming to obtain high performance. For this reason, it is essential to utilize the newly introduced high-speed computing mechanism in the K computer, including the effective use of cache.

(B) Nanoscience/nanotechnology field (PHASE, RSDFT)

- (a) For high parallelization, we found it difficult to adapt to the parallelization in the K computer with tens of thousands of nodes with the current parallelization method, following an analysis of the parallelization characteristics. Reviewing the fundamental parallelization method is the key point. (b) For single-CPU performance, high performance was expected by replacing the main processing with the processing of the matrix–matrix product.

Table 3.1 Applications for the performance demonstration of the K computer

Name	Field	Outline of application	Computational scientific features of code	Physical model/method
NICAM	Earth sciences	Global high-resolution atmospheric general circulation simulation	For ES ^a , the peak performance ratio is 40%. For programming, it is essential to use high-speed arithmetic mechanisms such as effective use of the cache to meet B/F performance	Atmospheric general circulation/FDM
Seism3D	Earth sciences	Seismic propagation/strong vibration simulation	For ES, the peak performance ratio is 40%. For programming, it is essential to use high-speed arithmetic mechanisms such as effective use of the cache to meet B/F performance	Seismic wave/FDM
PHASE	Nanoscience, nanotechnology	First principles of molecular dynamics based on DFT with plane wave expansion	The improvement of single-unit performance may be possible by replacing the main processing with the processing of matrix–matrix products. However, to secure performance in ultrahigh parallelism, it is necessary to increase the number of atoms considerably and it is necessary to consider high parallelization	DFT/plane wave method

(continued)

Table 3.1 (continued)

Name	Field	Outline of application	Computational scientific features of code	Physical model/method
RSDFT	Nanoscience, nanotechnology	First principles of molecular dynamics based on DFT with real-space difference method	The improvement of single-unit performance may be possible by replacing the main processing with the processing of matrix–matrix products. However, it is necessary to increase the number of meshes considerably to secure performance in ultrahigh parallelism and it is necessary to consider high parallelization	DFT/real-space method
LatticeQCD	Physics	Elementary particle nuclear analysis using Lattice QCD simulation	Advanced parallelization tuning that considers communication topology and measures to improve single-unit performance that are conscious of actual machines are indispensable	QCD/path integral method
FrontFlow/Blue (FFB)	Engineering	Unsteady flow analysis based on large eddy simulation (LES)	For ES, the peak performance ratio is 25%. For programming, high B/F performance is required, and for list access it is essential to use a high-speed arithmetic mechanism such as the effective use of the cache and to improve the efficiency of data access	Fluid/FEM

^a ES Earth Simulator

(C) Engineering field (FFB)

- (a) For high parallelization, the domain decomposition method, which is relatively easy to parallelize, is used. However, because the unstructured grid method is adopted, the parallelization becomes somewhat more complicated than that for applications in the earth science field that use the structural grid method. From the parallelization characteristics analysis, we found that adjacent communication does not become a large load even when it is highly parallelized; however, the time for global communication tends to increase with high parallelization. (b) For the single-CPU performance when using the Earth Simulator, about 20% of peak performance could be obtained. However, high memory bandwidth performance is required. Because the main part of the calculation requires list accesses, in a scalar computer with low memory bandwidth performance relative to the peak performance, it tends to be very difficult to obtain high performance.

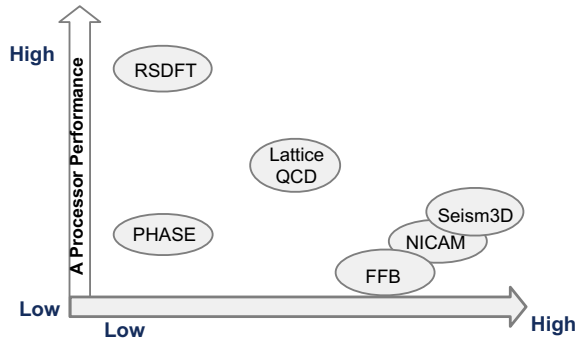
(D) Physics field (Lattice QCD)

This application has the same characteristics as (A). However, because it was adjusted to make effective use of the cache by reducing the problem scale executed on one node, the application has the following features. (a) For high parallelization, the domain decomposition method as in (A) was used, which made it easy to achieve high parallelization. However, because the problem size in one node is small, more communication is required compared with the computation, and the number of communications also increases. Therefore, this application has intermediate properties between (A) and (B). The parallelization characteristics analysis confirmed the above characteristics, and an advanced parallelization method aware of the utilization of the communication topology is required. (b) For the single-CPU performance, this application tends to require high B/F performance as in (A), and in principle, it is difficult to obtain high performance on the scalar architecture. Because the problem size in one node is small, it is easy to make effective use of the cache, and the application again has intermediate properties between (A) and (B). However, the programming is complicated, and this application requires measures to improve the single-CPU performance in accordance with the architecture of K computer.

The features described here are shown in Fig. 3.1.

In this chapter, we describe examples from two points of view. One is the evaluation method of the highly parallel characteristics and the examples of the ultrahigh-parallelization methods found to solve the problems. The other is the evaluation method of the single-CPU performance and the examples of the improvement of the CPU performance to solve the problems. For the former, we mainly discuss RSDFT and PHASE from the applications shown in this section; for the latter, we mainly discuss Seism3D and FFB.

Fig. 3.1 Computer science characteristics of applications for the performance demonstration



3.2 System Outline of the K Computer

In this section, we outline the system of the K computer as a basis for discussion.

3.2.1 Processor Features

We first describe the features and give a system outline of the processor of the K computer. First, we describe the outline of the CPU of the K computer [10]. One compute node consists of one CPU (SPARC 64 TM VIII fx, manufactured by Fujitsu Limited), 16 GB of memory, and an interconnect LSI (interconnect controller, ICC), which performs data transfers between compute nodes. The CPU has eight processor cores, a shared secondary cache memory (6 MB, 12-way write-back cache), and a memory control unit. CPU chips are 22.7 mm × 22.6 mm. Each core has an L1 data cache (32 KB, 2-way write-back cache), four product–sum operation units, and 256 double-precision floating-point registers. With one SIMD instruction, two product–sum operation units can be operated at the same time. By executing two SIMD instructions simultaneously, one core can perform eight floating-point operations per clock cycle. Therefore, the theoretical performance of each core is 16 GFLOPS, and the theoretical performance of the CPU (eight cores) is 128 GFLOPS for both single and double precision. The CPU also has various mechanisms for scientific and technical calculations, such as a hardware barrier mechanism for synchronizing the parallel processing between the cores, a prefetch mechanism for taking data necessary for calculation into the cache in advance, and a sector cache mechanism enabling programmable cache control. The theoretical bandwidth of the memory is 64 GB/s, and the B/F value is 0.5. The theoretical bandwidth of the L2 cache is 256 GB/s and the B/F value is 2.0. The theoretical bandwidth of the L1 cache is 64 GB/s, and the B/F value is 4.0. The memory and the L2 cache are accessed by lines of 128 bytes. The DGEMM performance of the CPU is 123.6 GFLOPS (ex-

cution efficiency 96.6%), and the hardware barrier performance between the cores is 49 ns. The memory access performance by a triad of the STREAM benchmark code is 46.6 GB/s.

We now describe the outline of the system. The complete K computer is a very large system composed of more than 80,000 CPUs, with the aforementioned node as the smallest structural unit. Four nodes are mounted on one system board, and 24 system boards are mounted in a computer rack. The entire K computer consists of 864 computer racks and has computing performance of 10 PFLOPS or more and the memory capacity is 1 PB or more. The K computer has also realized excellent power-saving performance (at the beginning of system operation) and high reliability. The SPARC64 TMVIII fx provides 128 GFLOPS per chip while the power consumption is only 58 W. It has excellent power-saving performance compared with the CPUs used in other supercomputers (at the beginning of system operation). This is realized by various technologies for power saving, such as a low operating frequency (2 GHz) (at the beginning of system operation) and a mechanism for cutting power to unused circuits. The operating temperature of this CPU is about 30 °C, which is extremely low (compared with the CPUs installed in other supercomputers), contributing to the reduction of the failure rate.

The execution time of the LINPACK benchmark program took about 28 h and no malfunction occurred even though the system operated with a high load for that time. This is an outstandingly large value compared with other supercomputers, and it shows the low failure rate and high reliability of the K computer. In this way, the K computer has not only the world's best computing performance but also various functions to improve utilization as a shared system, such as power-saving performance, reliability, availability, and operability.

3.2.2 Outline of Tofu Interconnect

In the K computer, the Tofu (torus fusion) interconnect [10, 11] was adopted to construct a communication network between the nodes, and the communication among the 82,944 nodes is realized in the whole system. Each node is equipped with an ICC, which has four Tofu network interfaces (TNIs) and a Tofu network router (TNR); simultaneous communication in up to four directions is possible with the TNI.

The TNR has ten links. Four of them handle communications for a three-dimensional (3D) mesh/torus network within 12 nodes (Tofu units), which is the basic unit of Tofu. The remaining six links handle communications for a 3D mesh/torus network between Tofu units. A six-dimensional (6D) mesh/torus network consists of two 3D mesh/torus networks.

3.2.3 6D Network

The physical coordinate axes in the 6D mesh/torus network are denoted as (x, y, z, a, b, c) [10, 11]. The three dimensions a, b, and c consist of 12 nodes of $2 \times 3 \times 2$, which is a unit of Tofu, and are arranged in physically close positions in the rack. The three dimensions x, y, and z are configured over several racks. In the K computer, the node allocation is carried out in units of Tofu to reduce the load of the job scheduler. With this combination of three dimensions plus three dimensions, the communication can avoid faulty nodes and the construction of the 3D torus network by job unit becomes possible [10]; this arrangement allows high availability¹ and flexibility to meet various operational needs. Users can specify the dimensions for assigning the nodes when submitting jobs to the K computer. Here, we call it “node shape”. Node shapes may be one-dimensional (1D), two-dimensional (2D), or 3D, and it is possible to construct a torus network by job unit. For example, when the node shape is specified as 3D, it is realized by a combination of six dimensions ((xa), (yb), (zc)). The MPI environment of the K computer is based on OpenMPI. For the collective communication function of MPI, in addition to the communication algorithm derived from OpenMPI, a communication algorithm optimized for the Tofu interconnect (Tofu-dedicated algorithm) was developed and high-communication performance is realized. For example, for MPI_ALLREDUCE communication in a 3D torus network, we have adopted an algorithm called Trinaryx 3, which realizes high performance by 3-way simultaneous communication, by dividing a message into three. Of the OpenMPI algorithm and the Tofu-dedicated algorithm, the optimum one is automatically selected according to the size of the message at the time of communication.

3.3 Computer Environment for Performance Evaluation

In the examples described below, the K computer was mainly used once it was completed. However, before its completion, we used the T2 K-Tsukuba system of the University of Tsukuba and the RIKEN integrated cluster of clusters (RICC) system, which was the largest computer system in Japan at that time. We also used the FX1 system as a small parallel computer environment. The CPU configuration, the theoretical performance, and the main memory of each of these computer systems are shown below.

(1) T2K-Tsukuba

CPU configuration: Opteron Barcelona B 8000 648 node (quad-core \times 4 sockets/node)

¹Here, availability means the ability of the system to operate continuously.

Theoretical performance: $2.3 \text{ GHz} \times 4 \text{ arithmetic} \times 4 \text{ cores} \times 4 \text{ sockets} = 147.2 \text{ GFLOPS/node}$, 95.3 TFLOPS for the system
 Main memory: $32 \text{ GB/node} = 20.7 \text{ TB/system}$

(2) RICC parallel cluster

CPU configuration: Intel Xeon $\times 2048 \text{ CPUs}$ (8192 core) (1024 units)
 Theoretical performance: $2.93 \text{ GHz} \times 4 \text{ operations} \times 8192 \text{ cores} = 96.0 \text{ TFLOPS}$
 Main memory: 12.0 TB ($12 \text{ GB} \times 1024 \text{ units}$)

(3) Fujitsu FX 1

CPU configuration: SPARC64 VII (2.5 GHz) 4 core/CPU, 1 CPU/node, 32 nodes
 Theoretical performance: 40 GFLOPS/node , $1280 \text{ GFLOPS/system}$
 Main memory: 32 GB/node .

3.4 Outline of the Examples of High-Parallelization Performance Optimization

3.4.1 *Enhancement of the Parallelization Axis*

As described later in Sect. 3.5.4 for RSDFT and in Sect. 3.6.2 for PHASE, problems arise when a software application cannot use the full parallelism of the hardware or the global communication time increases. For PHASE, another problem is the large residue of nonparallel parts. As described in Sect. 2.7 as a countermeasure against these problems, the enhancement of the parallelization axis is adopted.

Details of the enhancement of the parallelization axis for RSDFT are given in Sect. 3.5.5, and the results are shown in Sect. 3.5.6. Similarly, the enhancement of the parallelization axis was applied to PHASE as shown in Sect. 3.6.3, and the results are shown in Sect. 3.6.4.

3.4.2 *Improvement of Communication and Load Imbalances*

For RSDFT, the problems of the increasing global communication time are described later in Sect. 3.5.4, and load imbalance problems in the Gram–Schmidt calculation and the eigenvalue calculation are described in Sect. 3.5.6.

To deal with the increased global communication of the former, the enhancement of the parallelization axis was adopted as described in Sect. 2.7. By applying the optimum Tofu mapping method together with the enhancement of the parallelization axis, significant performance improvement in communications was obtained. Details of its application are given in Sect. 3.5.5, and the results are shown in Sect. 3.5.6.

3.5 Performance Optimization of RSDFT

3.5.1 Overview of RSDFT

RSDFT is a program to perform electronic state calculations based on density functional theory [6]. It aims to elucidate quantum theory phenomena at the nanoscale based on first principles and to predict nanomaterials and structures having new functions. For example, the magnitude of leakage current of conventional semiconductors has become a problem as the process becomes finer. To solve these problems, RSDFT is used for a simulation to investigate the characteristics of new semiconductors with low power consumption.

The Kohn–Sham (KS) equation expressed as follows is derived from density functional theory within a local density approximation (LDA):

$$\left[-\frac{1}{2}\nabla^2 + v_{eff} \right] \psi_i(\vec{r}) = \varepsilon_i \psi_i(\vec{r}), \quad (3.1)$$

$$v_{eff} = V_{nucl}(\vec{r}) + \int \frac{n(\vec{r}')}{|\vec{r} - \vec{r}'|} d\vec{r}' + \frac{\delta E_{xc}[n]}{\delta n(\vec{r})}, \quad (3.2)$$

$$n(\vec{r}) = \left| \sum \psi_i(\vec{r}) \right|^2. \quad (3.3)$$

In Eq. (3.1), i represents the quantum number of an energy band, ψ_i represents a wave function, and r represents a space coordinate. Equation (3.1) is the eigenvalue equation that will be solved. From the wave function ψ_i , i is obtained by solving the eigenvalue equation and the electron density $n(\vec{r})$ is calculated using Eq. (3.3). The calculation is performed repeatedly until the input electron density matches the output electron density within a certain range (self-consistent field calculation (SCF)). RSDFT adopts the real-space method, which introduces a 3D lattice of the real space into the eigenvalue equation and solves discretized physical property values on each lattice as a difference equation. The lattice forming the 3D space is divided into equal parts, ML1, ML2, and ML3, and the KS equation is solved as the Hermitian eigenvalue problem with dimension ML (=ML1 \times ML2 \times ML3). The parallel processing is implemented by dividing the space lattice into several small areas and assigning each small area to a node.

3.5.2 Understanding the Software Application Characteristics by Investigating the RSDFT Source Code

Examination of the RSDFT source code showed that the main parts of the software program are as follows: DTCG, which implements the conjugate gradient method; GS, which carries out the standard Gram–Schmidt orthogonalization; and DIAG, which carries out partial diagonalization. The main part of the calculations and the update of the electronic density and potential are repeated until the condition of self-consistent field (SCF) is satisfied. Figure 3.2 shows the calculation flow of RSDFT.

Fig. 3.2 Calculation flow of RSDFT

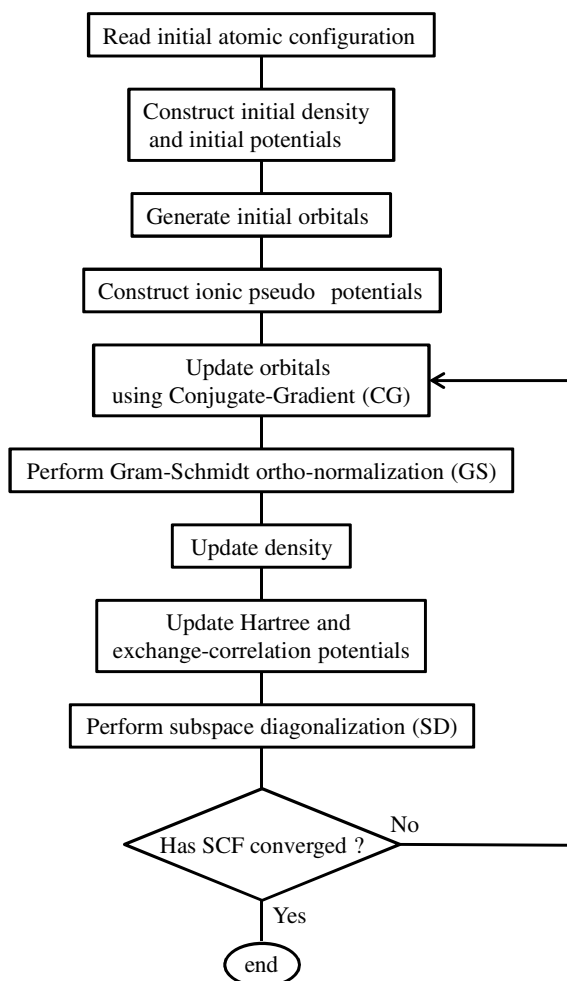


Table 3.2 Investigation results of RSDFT source code

Name	Explanation of processing	Amount of computation
DTCG	MB eigenvalues and eigenvectors of a symmetric $ML \times ML$ matrix calculated with a conjugate gradient method in order from the smallest eigenvalues	$O(ML \times ML)$ $O(N^2)$
Gram–Schmidt	Orthogonalization	$O(ML \times MB^2)$ $O(N^3)$
DIAG	Diagonalization of Hamiltonian limited to a subspace of ML dimension	
	Creating matrix elements (MatE)	$O(ML \times MB^2)$ $O(N^3)$
	Solving eigenvalues (pdsyevd)	$O(MB^3)$ $O(N^3)$
	Calculation of rotating (RotV)	$O(ML \times MB^2)$ $O(N^3)$

DIAG consists of three parts: MatE, which generates matrix elements; pdsyevd, which performs the eigenvalue calculation; and RotV, which performs rotation calculation. Table 3.2 shows the processing and the amount of computation found in the investigation of the source code. Here, ML represents the number of grids and MB represents the number of energy bands. Because MB and ML are proportional to the number of atoms N, the amount of computation in DTCG is of the order of N^2 , while the amount of computation in Gram–Schmidt is of the order of N^3 . MatE, pdsyevd, and RotV were also found to require computation of the order of N^3 . These investigation results are consistent with the theoretical background.

As a supplement to these results, the cost distribution was measured under the following conditions.

Measurement computer: RICC parallel cluster

Calculation parameters: number of atoms 8000, grid size $120 \times 120 \times 120$

Number of energy bands: 16,000

Parallelism: $8 \times 8 \times 8 = 512$, space parallel only

The following cost distribution was obtained. As described in the survey results of the source code, the cost analysis also confirmed that DTCG, Gram–Schmidt, and DIAG together form the main calculation part.

Initialization: 0.4%

SCF part (assuming SCF 100 times): 99.6%

DIAG: 30.5%

DTCG: 27.4%

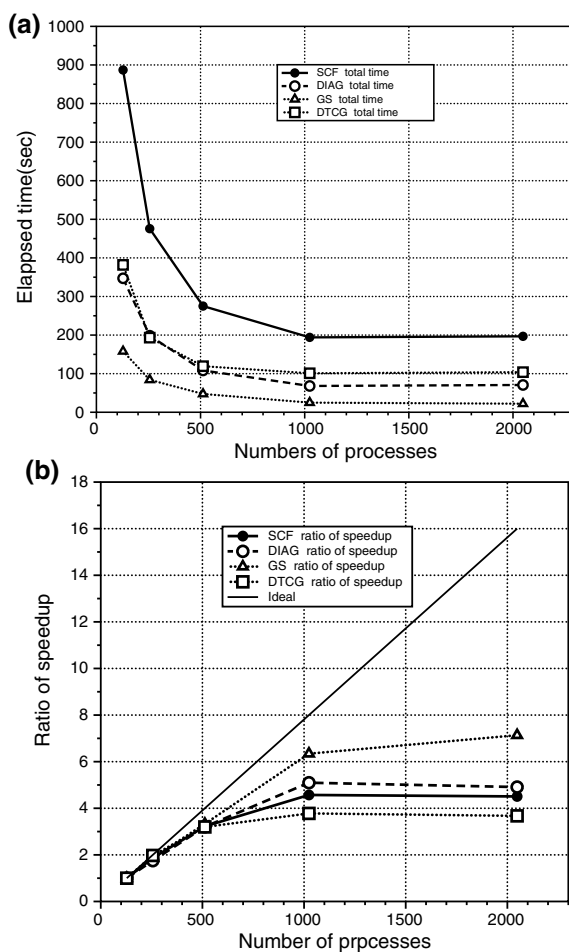
Gram–Schmidt: 38.6%

Mixing and output of intermediate result: 3.1%.

3.5.3 Measurement of the Parallelization Features of RSDFT

Scalability measurements were performed on the three blocks found to be the main computing parts: DTCG, Gram–Schmidt, and DIAG. The T2 K-Tsukuba of the University of Tsukuba and the PGI compiler were used, and mvapich2-medium was used as a communication library. Five block divisions were used in the spatial direction: 128, 256, 512, 1024, and 2048. The calculation system used for the measurement was 4096 atoms of Si, with 8192 energy bands and $96 \times 96 \times 96$ grids for each block. The calculation time, the global communication time, and the adjacent communication time were separated and measured using strong scaling. Figure 3.3 shows the measurement results for each block, including calculation and communication time. DTCG, Gram–Schmidt, and DIAG have good scalability only up to about 256

Fig. 3.3 Scalability of each calculation block of RSDFT



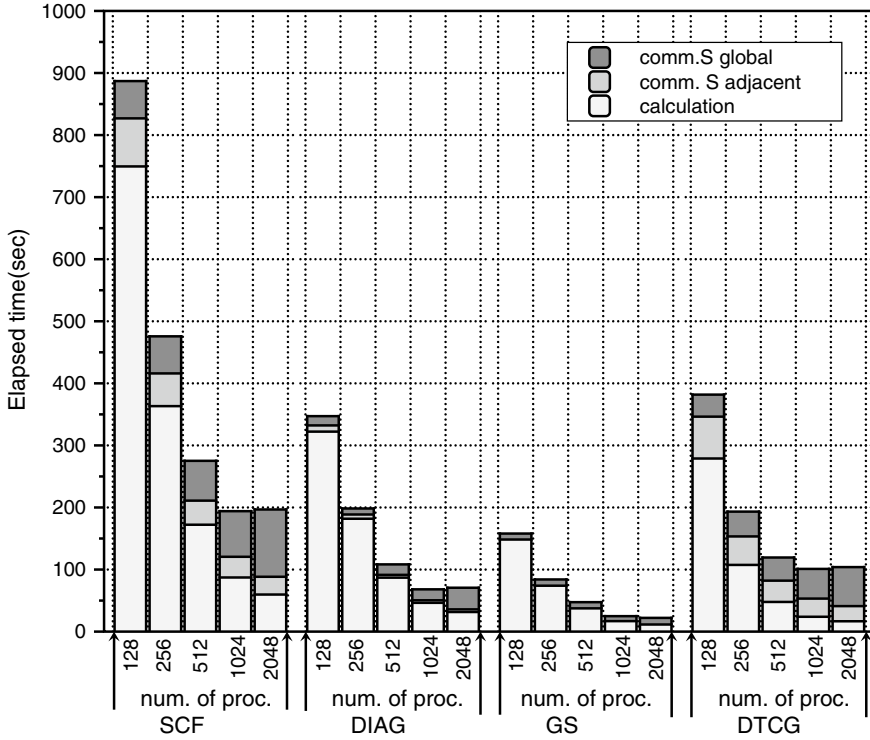


Fig. 3.4 Scalability of each calculation/communication block of RSDFT

parallel paths. Figure 3.4 shows the measurement results for each block divided into the calculation, the global communication, and the adjacent communication times. For each block of DTCG, Gram–Schmidt, and DIAG, the calculation time shows good scaling up to 1024 parallel paths but shows a deterioration in scaling from 2048 parallel paths. We found that the global communication time increased as the number of parallel paths increased for DTCG and DIAG.

3.5.4 Evaluation of RSDFT Kernel and High-Parallelization Feature

For RSDFT, the calculation and communication kernel was evaluated. From the investigation of the source code shown in Sect. 3.5.2, when targeting 100,000 atomic systems, we found that the calculation parts of Gram–Schmidt, MatE, pdsyevd, and RotV, with computation of the order of N^3 , computationally formed a kernel. From the measurement results of the parallelization characteristics shown in Sect. 3.5.3,

we found that DTCCG, which causes increases in communication time in accordance with the increase of the number of nodes, is also the kernel.

First, we consider the parallelization characteristics for computation time. By dividing the number of grids shown in the calculation scheme in Sect. 3.5.2 by the number of parallel paths, $96 \times 96 \times 96/2048 = 432$ is obtained. It turns out that the parallelization characteristics of the operation worsen at 432 grids per process. At this point, assuming that the target problem of 100,000 atoms is solved using all 82,944 nodes of the K computer, $100,000 \times 216 = 21,600,000$ grids are required. When this value is divided by the number of nodes, 82,944, the value becomes 260.4, which is much smaller than the number of grids, 432 per process, at which the parallelization characteristics worsen. This means that for the target problem, in the parallelization of the current grid space, there is a problem that not all nodes of the K computer can be used; that is, there is a mismatch in the number of parallel paths between the hardware and the application.

Next, we consider the high-parallelization characteristics of the communication time. For the two kernels, DTCCG and DIAG, the global communication time increases drastically, and for DIAG and Gram–Schmidt the global communication time for 2048 parallels is almost the same as the computation time. For DTCCG, the global communication time for 2048 parallel paths is nearly four times larger than the computation time. This is a major problem when aiming for high parallelization. The parallelization characteristics of each kernel are summarized in Table 3.3.

3.5.5 Code Modifications for High Performance of RSDFT

Enhancement of the parallelization axis

As described in Sect. 3.5.4, the first problem for the high parallelization of RSDFT is to deal with the limited parallelism of RSDFT compared with the hardware. The second problem is the high global communication time.

RSDFT solves the eigenvalue equation of Eq. (3.1). In the original RSDFT code, the variable r representing the space in this eigenvalue equation is parallelized. This equation contains the quantum number of the energy band i , and because there is no dependence between different energy bands, in principle it is also possible to parallelize by i . By enhancing the parallelization axis as described in Sect. 2.7, it is possible to increase the parallelization of the application. At this point, the parallelization was carried out for the energy bands in addition to the spatial direction, to enhance the parallelization axis, allowing us to use several tens of thousands of parallel paths rather than 1000, as shown later.

Investigation of the communication time after enhancing the parallelization axis

As shown in Sect. 2.7, the enhancement of the parallelization axis may eliminate an increase in global communication time, which is the second problem of RSDFT. An outline of the communication used in RSDFT is shown below. The underlined steps

Table 3.3 Parallel characteristics of RSDFT kernel

Name	Explanation of processing		Amount of computation	Parallel characteristics	Single-CPU performance
DTCG	MB eigenvalues and eigenvectors of a symmetric $ML * ML$ matrix calculated with a conjugate gradient method in order from the smallest eigenvalues	Minimize of Rayleigh quotient $\frac{\langle \psi_m H_{KS} \psi_m \rangle}{\langle \psi_m \psi_m \rangle}$	$O(ML \times ML)$ $O(N^2)$	Increased communication time. It is reversed calculation time. Lack of parallelism	Matrix-vector product performance is bad
Gram-Schmidt	Orthogonalization	$H_{m,n} = \langle \psi_m H_{KS} \psi_n \rangle$	$O(ML \times MB^2)$ $O(N^3)$	Communication time does not decrease. It is comparable to calculation time. Lack of parallelism	Good due to matrix-matrix product
DIAG	Diagonalization of Hamiltonian limited to a subspace of ML dimension				
	Creating matrix elements (MatE)	$\psi'_n = \psi_n - \sum_{m=1}^{n-1} \psi_m \langle \psi_m \psi_n \rangle$	$O(ML \times MB^2)$ $O(N^3)$	Increased communication time. It is comparable to calculation time. Lack of parallelism. Scalability of Scalapack is bad	Good due to matrix-matrix product
	Solving eigenvalues (pdsyevd)	$(H_{N \times N})(\vec{c}_n) = \varepsilon(\vec{c}_n)$	$O(MB^3)$ $O(N^3)$		Performance of Scalapack is bad
	Calculation of rotating (RotV)	$\psi'_n(r) = \sum_{m=1}^N c_{m,n} \psi_m(r)$	$O(ML \times MB^2)$ $O(N^3)$		Good due to matrix-matrix product

show the communication in the direction of the energy band, which is necessary because of the enhancement of the parallelization axis.

- Global communication
- MPI_ALLREDUCE
 - Gram–Schmidt: inner product array, normalization variable
 - DTCG: scalar variable
- MPI_REDUCE
 - DIAG (MatE)
- MPI_BCAST
 - DIAG (Rot V)
 - Gram–Schmidt: Deliver the updated wave functions of the triangle parts in Fig. 3.5
- MPI_ALLGATHERV
 - DIAG
 - Gram–Schmidt
- Adjacent communication
- Exchange of the boundary data: BCAST
- Nonlocal term calculation: HPSI
- Exchange the symmetric block data: DIAG (MatE)

Because of the enhancement of the parallelization axis, new communication processing on the enlarged energy band axis is required. Figure 3.5 shows the outline of the MPI_BCAST communication related to the Gram–Schmidt calculation as a

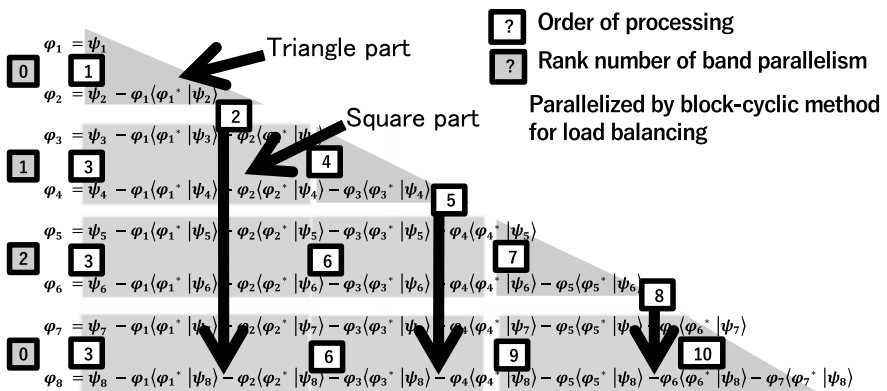


Fig. 3.5 MPI_BCAST communication related to the Gram–Schmidt calculation on the energy band axis

typical example of the new communication processing on the energy band axis. First, we perform the calculation of the triangle parts with rank 0. Next, we transfer the calculation result to other ranks where the transferred data is used to calculate the rectangle parts using DGEMM. After these, the procedure is repeated.

As described in Sect. 2.7.3, a reduction in communication time can be expected for MPI_ALLREDUCE or MPI_BCAST by applying the enhancement of the parallelization axis, but this reduction is less than the reduction effect for MPI_ALLGATHERV and MPI_ALLTOALL. This means that if new communications of the types MPI_ALLGATHERV and MPI_ALLTOALL are required in the code, the penalty for the addition of the communication may be large. The above outline of the communication used in RSDFT shows that the center of the global communication of the original RDFT code is MPI_ALLREDUCE and MPI_BCAST, and MPI_ALLGATHERV and MPI_BCAST are additions. That is, the effect of reducing the communication time is not large, and the penalty may actually increase. Therefore, the amount of communication and the number of each communication were investigated. Part of the result is shown in Table 3.4. The portions of the communication added are indicated by hatching in this table. This table shows that for MPI_ALLGATHERV, which carries a large penalty, neither the amount of communication nor the number of communications is large. We can also see that neither the amount nor the number of communications is large for MPI_BCAST.

Evaluation of the parallelization and communication by the enhancement of the parallelization axis of RSDFT

To verify the results of these investigations, we implemented the energy band parallelization and evaluated the increase in parallelization and the communication time. Table 3.5 shows the measurement patterns used for the evaluation. Normally, increasing the number of atoms increases the number of energy bands. Here, to measure the weak scaling, we performed the measurement while fixing the number of energy bands to 19,200, the number of parallel paths for the energy band to eight, and the number of energy bands in parallel to 2400. For the parallelization with respect to the grid, the number of grids per process was fixed to $12 \times 12 \times 12$, the number of parallel processes was $4 \times 4 \times 4 = 64$, $5 \times 5 \times 5 = 125$, $6 \times 6 \times 6 = 216$, or $8 \times 8 \times 8 = 512$. Finally, the measurement was performed using the four parallelism numbers $64 \times 8 = 512$, $125 \times 8 = 1000$, $216 \times 8 = 1728$, and $512 \times 8 = 4096$.

In the measurements, data were acquired for each of the four blocks GS, DTCG, MatE/DIAG, and RotV/DIAG. Figure 3.6 shows the evaluation results. The calculation time is shown as “CALCULATION”, the global communication time for space is shown as “COMM S GLOBAL”, the global communication time for energy bands is shown as “COMM B GLOBAL”, and the adjacent communication time for space is shown as “COMM S NEIGHBOR”. The execution times of the four blocks are shown as the name of each process: “GS”, “DTCG”, “MatE”, and “RotV”.

There was no significant increase in global communication with respect to space, as seen in Sect. 3.5.3, and no significant increase in the global communication with respect to the newly added energy bands was observed. We also measured with weak scaling, and no increase in the computation time was observed for any process-

Table 3.4 Communication size and number of communication times of RSDFT

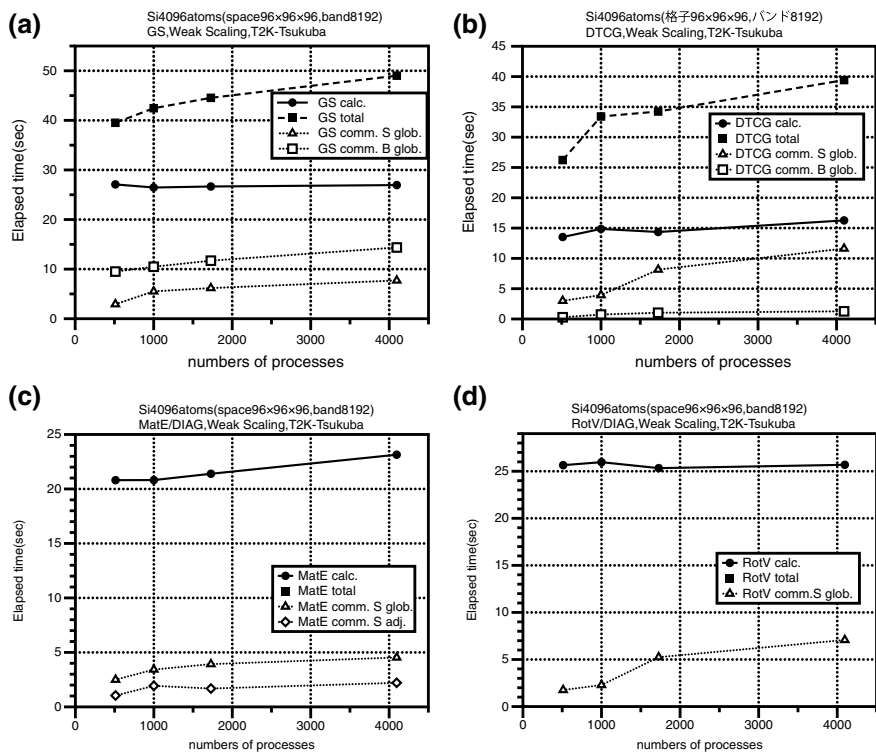
Name	MPI function	Type	Data size	Number of communications
Gram-Schmidt	mpi_allgatherv	mpi_real8	MB/NPB	1
	mpi_allreduce	mpi_real8	$MBLK * NBLK \sim (NBLK1+1) * (NBLK1+1)$	$MB/NBLK * MBLK/NPB + \text{Int}(\log(NBLK/NBLK1)) * (MB/NBLK/NPB)$
	mpi_allreduce	mpi_real8	$NBLK1 \sim 1$	$NBLK1 * (MB/NBLK/NPB)$
	mpi_allreduce	mpi_real8	1	$MB/NBLK * MBLK/NPB + \text{Int}(\log(NBLK/NBLK1)) * (MB/NBLK/NPB) + NBLK1 * (MB/NBLK/NPB)$
	mpi_bcast	mpi_real8	$MLO * NBLK$	$MB/NBLK/NPB$
DIAG	mpi_allgatherv	mpi_real8	MB/NPB	1
	mpi_reduce	mpi_real8	$MBLK * MBLK$	$(MB/MBLK * MB/MBLK)/NPB$
	isend/irecv	mpi_real8	$MBLK * MBLK$	1
	Omit communication in Scalapack (pdsyevd)			
	mpi_bcast	mpi_real8	$MSIZE * NBSIZE$	$(MB/MBSIZE * MB/NBSIZE)/NPB$
HPSI	mpi_isend	mpi_real8	$lma_nsend(irank) * MBLK$	$6 * NCNONL * MB/MBLK/NPB$
	mpi_irecv	mpi_real8	$lma_nsend(irank) * MBLK$	$6 * NCNONL * MB/MBLK/NPB$
	mpi_waitall		–	$MB/MBLK/NPB$
BCAST	mpi_isend	mpi_real8	$Md * MBLK$	$6 * MB/MBLK/NPB$
	mpi_irecv	mpi_real8	$Md * MBLK$	$6 * MB/MBLK/NPB$
	mpi_waitall		–	$MB/MBLK/NPB$

MB Number of energy Band, *NBLK* Max size of DGEMM, *NBLK1* Minimum size of DGEMM, *MBSIZE* Row block size of $MB \times MB$ matrix, *NBSIZE* Column block size of $MB \times MB$ matrix, *MBLK* $\min(MBSIZE, NBSIZE)$, *Md* Order of higher order finite difference, *lma_nsend* Number of nonlocal terms, *NPB* Number of energy band parallelism, *MLO* Number of grids processed by one process, *NCNONL* Number of adjacent communications in nonlocal term calculation for each direction

ing block, and no increase in the adjacent communication time was observed. The absence of an increase in the computation time for the weak scaling measurements implies that there is no nonparallel part in the computation, and the absence of an increase in the adjacent communication time means that there are no problems in the adjacent communication.

Table 3.5 Evaluation pattern of parallel axis expansion effect

	Number of atoms	Number of space grids	Number of energy bands	Number of processes
Pattern1	512	$48 \times 48 \times 48$	19,200	512 ($4 \times 4 \times 4 \times 8$)
Pattern2	1000	$60 \times 60 \times 60$	19,200	1000 ($5 \times 5 \times 5 \times 8$)
Pattern3	1728	$72 \times 72 \times 72$	19,200	1728 ($6 \times 6 \times 6 \times 8$)
Pattern4	4096	$96 \times 96 \times 96$	19,200	4096 ($8 \times 8 \times 8 \times 8$)

**Fig. 3.6** Evaluation result of effect expanding the parallelization axis

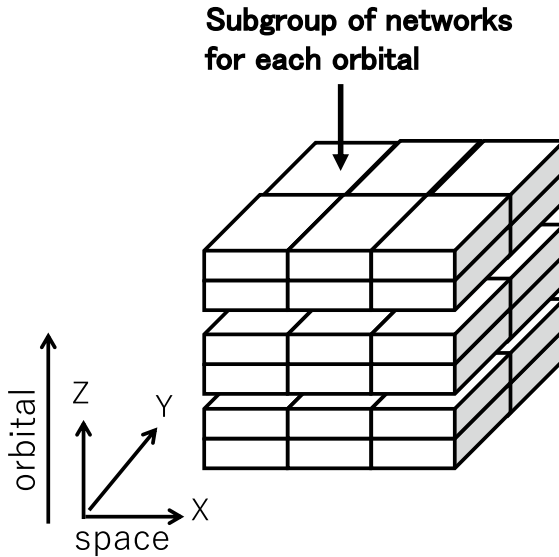


Fig. 3.7 Process mapping using for RSDFT on Tofu topology

By dividing the number of grids used for this evaluation by the number of parallels, we obtained $96 \times 96 \times 96 / 4096 = 216$, which is the same value as that for the target problem evaluated in Sect. 3.5.3. We can conclude that the scalability up to about 4000 parallel paths is secured while satisfying the (grid number/parallelization number) value of the target problem by enhancing the parallelization axis.

3.5.6 Results of High Performance of RSDFT

Adoption of optimal Tofu mapping

In RSDFT, global communications such as MPI_BCAST and MPI_ALLREDUCE, occur, and have large message lengths. They require efficient communication. In dealing with the above extremely massive parallelization of RSDFT, the product of the grid points and the number of divisions of energy bands is allocated to all compute nodes through the two-axes parallelization.

In the K computer, a manually prepared rank mapping file² can allow the optimum mapping of the addresses of the network topology to the rank numbers. In this stage of the performance tuning of RSDFT, we used this feature and mapped the computation nodes to the Tofu network, as shown in Fig. 3.7. In this figure, “orbital” represents an orbital corresponding to an energy band. When all the compute nodes are divided into three parts, as shown in Fig. 3.7, the compute nodes in each part can always

²This is specified with the parameters at the job execution.

be connected in a torus network, thanks to the 6D Tofu network. Orbitals can be partitioned almost equally into the same number of orbital groups as the number of parts and each orbital group is allocated to each part. The parallelized tasks at the grid points are executed within the part. With this allocation, communication among the parallel tasks in the grid points is kept within a part and does not affect the communications within other parts. The number of parts is increased to the number of parallel paths in the orbitals. By adopting this mapping, the performance improvement shown in Fig. 3.8 is obtained. By optimizing the mapping, the optimal communication algorithm is applied for the Tofu topology, and the communications are about 4.5 times faster.

Improving communication performance through two-axes parallelization

An analysis of silicon nanowires using 19,848 silicon atoms was carried out to verify the improvement in communication performance through two-axes parallelization. The number of energy bands was 41,472. We used 12,288 nodes of the K computer and 98,304 cores. The number of grids used for the calculation was $320 \times 320 \times 120 = 12,288,000$. Figure 3.9 shows the comparison between the computation and the communication time for space-only parallel processing for each processing block and for two-axes parallel processing. The number of grid divisions when using space-only parallelization was 12,288, and each node handled 1000 grids. With two-axes parallelization of space and energy band, there were 4096 grid divisions and each node handled 3000 grids. The energy band blocks were divided into three, and each node handled 13,664 energy bands. The figure shows the great reduction in global communication time that can be achieved for any processing block.

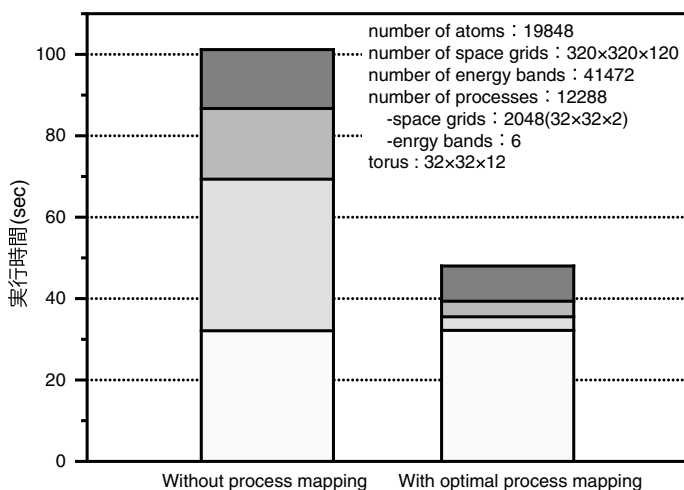


Fig. 3.8 Process mapping effect using for RSDFT on Tofu topology

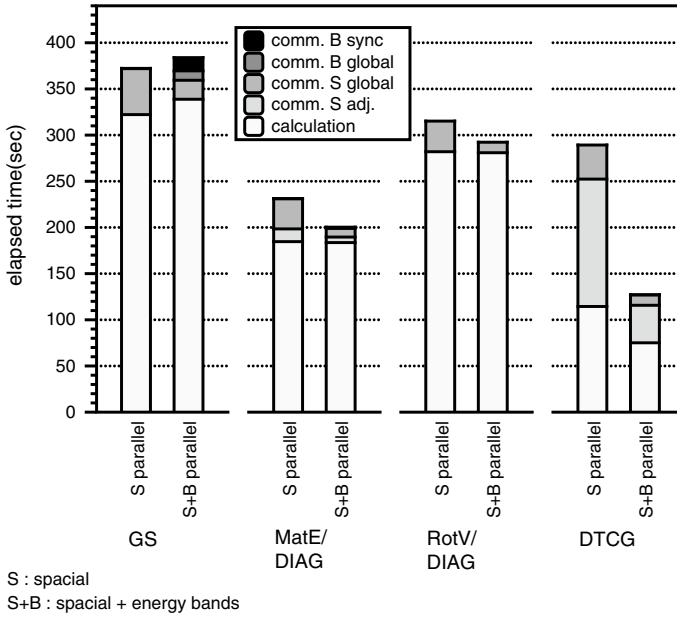


Fig. 3.9 Improving effect of communication performance by expanding the parallelization axis

3.5.7 Total Performance of RSDFT

Figure 3.10 shows the strong scaling performance of RSDFT with two-axes parallelization. The number of block-parallel divisions in the space was fixed to 1536, and the energy band blocks were divided into 1, 2, 3, and 6 groups. Therefore, the total parallelization was 1536, 3072, 4608, and 9216, respectively. The numbers of cores used were 12,288, 24,576, 36,864, and 73,728, respectively. The horizontal axis in Fig. 3.10 shows the number of cores and (a), (b), (c), and (d) show the calculation and communication times for each processing block of GS, CG, MatE/SD, and RotV/SD. The dashed line with open circles indicates the theoretical calculation time, and the solid line with black squares indicates the measured calculation time. They are consistent with each other, which shows that good scalability is obtained with two-axes parallelization. It is also clear that there is no problem because, for both the space and the energy band, the global communication and adjacent communication times decrease as the number of parallel paths increases or when their absolute values are small. The graphs in Fig. 3.10 show the good effects of two-axes parallelization using the space and the energy bands.

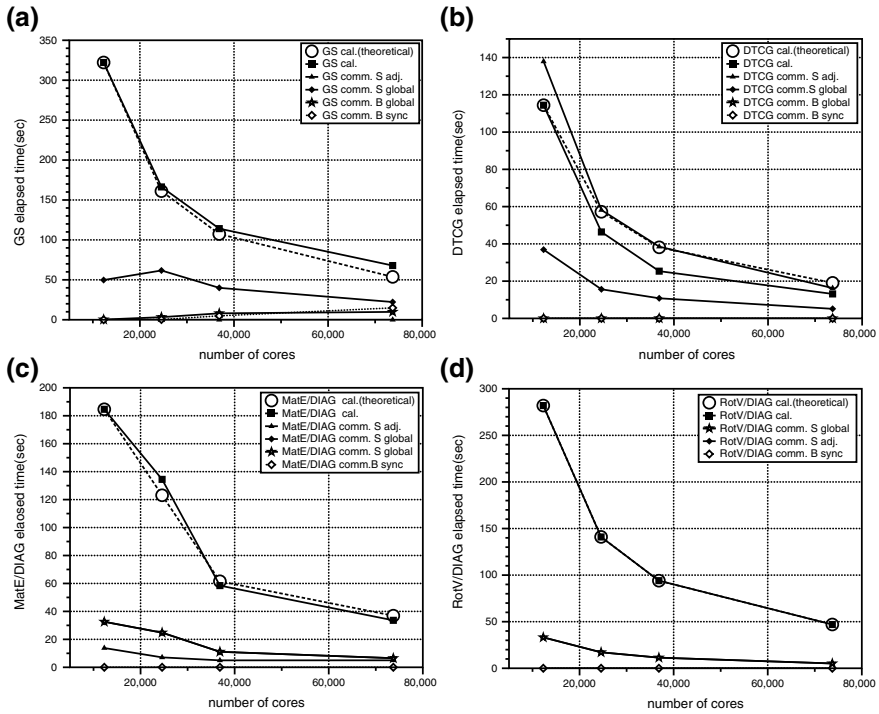


Fig. 3.10 Strong scaling performance of RSDFT with two-axis parallelization

For the total performance verification, silicon nanowires were analyzed using 107,292 silicon atoms. The number of energy bands was 229,824. The number of nodes of the K computer used was 55,296 with 442,368 cores. The number of grids used for the calculation was $576 \times 576 \times 192 = 63,700,992$, with 18,432 grid block divisions, and each node handled 3456 grids. The energy band blocks were divided into three groups, and each node handled 76,608 energy bands. The total performance of this system is shown in Table 3.6. The complete SCF calculation achieved 3.08 PFLOPS, achieving a peak performance ratio of 43.6%. This experiment was awarded the Gordon Bell Award at SC'11, which is a large international conference related to supercomputers [12].

Table 3.6 Total performance of RSDFT

Procedure block	Execution time (s)	Computation time (s)	Communication time (s)				Performance PFLOPS/%
			Adjacent/space	Global/space	Global/energy band	Wait/energy band	
SCF	5456.21	4417.15	83.18	899.05	15.87	40.93	3.08/43.63
SD	3710.01	3218.73	27.70	458.87	4.70	-	2.72/38.52
MatE	1084.70	717.45	27.70	337.85	4.70	-	3.09/43.72
EigenSolve	1322.16	879.61	-	442.39	-	-	0.04/0.61
RotV	1298.16	1177.15	-	121.01	-	-	5.18/73.25
CG	209.29	57.66	55.48	96.14	0.01	-	0.05/0.74
GS	1536.90	1140.76	-	344.04	11.16	40.93	4.37/61.87

3.6 Performance Optimization of PHASE

3.6.1 Overview of PHASE

PHASE [5] performs electronic structure calculations based on density functional theory, like RSDFT. It clarifies quantum theory phenomena at the nanoscale from first principles and predicts nanomaterials and nanostructures with new functions. The purpose of PHASE is to calculate structural relaxation and dynamics from first principles and to calculate various physical properties of the structures obtained.

Solving the Kohn–Sham equation obtained from density functional theory is ultimately equivalent to solving the eigenvalue equation:

$$H\psi_{ik}(\vec{G}) = \varepsilon_i\psi_{ik}(\vec{G}). \quad (3.4)$$

Here, ψ_{ik} represents the wave function defined in the periodic boundary condition, i represents the quantum number of the energy band, \vec{G} represents the reciprocal lattice vector, and k represents the k th point in the reciprocal lattice space. In contrast to RSDFT, PHASE uses a periodic boundary condition and the wave function is expressed as a function of the reciprocal lattice vector \vec{G} . To solve the Kohn–Sham equation, fast Fourier transformation (FFT) is used. Plane waves are employed as the basis function and the wave function is expressed by a linear combination of plane waves.

3.6.2 Understanding the Application Characteristics by Investigating PHASE Source Code

Because RSDFT and PHASE have parts in common, we will briefly describe the programming of the parts that are different from RSDFT.

Our investigation of the source code of PHASE showed that the program is divided into 11 blocks. When these processing blocks are categorized according to their calculation characteristics, they can be classified into three kinds of processing blocks:

- Blocks that can be rewritten as matrix–matrix products;
- Blocks including FFT;
- Blocks that perform the diagonalization of matrices.

The first group of processes can be rewritten as matrix–matrix products, as in RSDFT, as Gram–Schmidt orthonormalizations (described in Sect. 2.8.4). The amount of computation of these processes is of the order of N^3 , as in RSDFT.

The second group of processes involves FFT, which is not in RSDFT. PHASE performs the FFT processing from the reciprocal lattice space to the real space once while calculating the product of Hamiltonian and wave functions in solving

the Kohn–Sham equation, and performs inverse FFT processing after inner product processing in the real space. In PHASE, using the discretization in the reciprocal lattice space, the second group of processes appears frequently. The calculation amount of the FFT processing is of the order of $N^2 \times \log_2 N$.

The third group of processes performs diagonalizations, and these are also present in RSDFT. The amount of computation of the diagonalization processing is of the order of N^3 .

In PHASE, as with RSDFT, there is no dependency on the energy band quantum number i (Eq. (3.4)), and this independence is used to parallelize the energy bands. There is also a part parallelized for the reciprocal lattice (G in Eq. (3.4)). The transpose operation is used, so the wave functions parallel to the energy bands before G -parallelization can be G -parallelized. In addition, the transpose operation for returning the wave functions parallelized in G is parallel to the energy band parallelization after the G -parallel calculation has completed. The transpose operation shown in Fig. 3.11 is the global communication between all nodes (MPI_ALLTOALL type), and both the transfer amount and the transfer count are large contributors to the increase in the communication cost.

From the source code investigation, the loop structure of block 2 is shown in Fig. 3.12. This loop structure is displayed in a simplified form; originally, it was a more complicated structure, but the essential parts are shown here. The innermost energy band parallel part is the loop originally parallelized in PHASE. There is a

Fig. 3.11 Transpose operation in PHASE

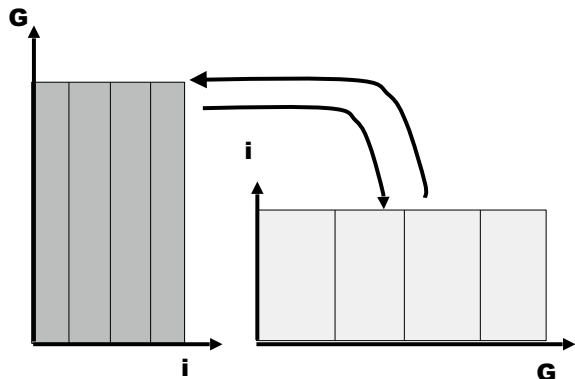


Fig. 3.12 Loop structure of PHASE block 2

```

subroutine m_es_vnlocal_w(ik,ikn1,ispin,switch_of_eko_part)
  --call tstato0_begin
  loop_ntyp: do it = 1, ntyp
    loop_natm : do ia = 1, natm --- Loop of atom number
      --call calc_phase
      T-do lmt2 = 1, lmt(it)
        --call vnlocal_w_part_sum_over_lmt1
        --call add_vnlph_l_without_eko_part
        subroutine add_vnlph_l_without_eko_part()
          T-if(kimg == 1) then
            T-do ib = 1, np_e -----Loop of energy band parallelism
              T-do i = 1, lba(ik)
                V-end do
              V-end do
            --else
              T-do ib = 1, np_e -----Loop of energy band parallelism
                T-do i = 1, lba(ik)
                  V-end do
                V-end do
              V-end if
            end subroutine add_vnlph_l_without_eko_part
          V-end do
        V-end do loop_natm
      V-end do loop_ntyp
    end subroutine m_es_vnlocal_w
  
```

loop for atoms outside this loop, which also contains the heavy processing for the reciprocal lattice. The loop between this atomic number loop and the energy band parallel part was not parallelized in the original PHASE code. We found that there is a nonparallel part in section 2, so that the whole is partially parallelized.

The domain decomposition is in many cases completely parallelized. However, as in PHASE, in spite of physically being able to adopt multiple parallelization axes, if only one axis is used, nonparallel parts may remain.

3.6.3 *Modification of the Code for High Performance of PHASE*

The problem with the high parallelization of PHASE was the limited parallelization of the applications compared with the number of cores, as with RSDFT. PHASE solves the eigenvalue equation of Eq. 3.4. Basically, in the original PHASE, the parallelization was implemented for the variable i representing the energy band quantum number in the eigenvalue equation. For the Gram–Schmidt diagonalization processing, for example, parallelization through the reciprocal lattice (G) was implemented. In principle, the equations used in PHASE can be parallelized completely for all energy bands i and the reciprocal lattices G for all parts. As described in Sect. 2.7.3, enhancement of the parallelization axis allows an increase in the parallelization of the application to match the number of hardware cores. At this time, the full parallelization of two axes has been implemented for all parts concerning the energy bands and the reciprocal lattice. By thus enhancing this parallelization axis, we have extended the number of parallel paths to several tens of thousands from 1000, as shown later.

In Sect. 3.6.2, the high parallelization of PHASE showed that there was a remaining nonparallel part, which was sandwiched between the head of the atomic number loop and the energy band parallel part shown in Fig. 3.12. This part performs the calculation on the reciprocal lattice. Therefore, in addition to the energy bands, by parallelizing through the reciprocal lattice (G), this remaining nonparallel part can be parallelized.

As described in Sect. 2.8.4, it is also possible to rewrite the calculation of the non-local term and the calculation of the Gram–Schmidt orthogonalization as matrix–matrix products, as with RSDFT. This revision can reduce the required B/F value, and high performance of a single CPU can be expected. Furthermore, by using the matrix–matrix product BLAS level 3 subroutine DGEMM from the mathematical library, it becomes possible to calculate with very high performance optimized for a particular machine [5]. Because PHASE did not implement the algorithm using this matrix–matrix product, we have applied this algorithm and have evaluated the results [13].

3.6.4 Total Performance of PHASE

Table 3.7 shows the execution time and the execution efficiency while calculating the amorphous system of HfSiO_2 with 1536 atoms, as measured in the K computer using 2048 cores. The parts rewritten using the BLAS library achieved efficiencies of more than 50%, and we have achieved execution efficiency exceeding 20% throughout the SCF loop. Moreover, from measuring an SiC system with 3800 atoms using from 48 to 12,288 parallel paths, we confirmed that the scaling can be secured up to parallelization numbers much larger than the number of atoms. This could not be calculated using the conventional parallelization method because of the finer division granularity.

3.7 Examples of Single-CPU Performance Optimization

In this section, we outline the performance optimizations related to the single-CPU performance as described in Sect. 2.8, using Seism3D and FFB. For Seism3D, we describe the effective use of the cache shown in Sect. 2.8.9. The effective use of the cache is shown in Sect. 3.8.3 and the results are shown in Sect. 3.8.4 [7]. We also describe the cache validation method for applications using list access in Sect. 2.8.10 for FFB. Section 3.9.3 shows the method and Sect. 3.9.4 shows the result [7, 14, 15].

Again using FFB as the example, we describe the block coloring method for recurrence elimination, targeting the unstructured grid shown in Sect. 2.8.3. The method is shown in Sect. 3.9.3 and the results are shown in Sect. 3.9.4 [7, 14, 15]. Using Seism3D and FFB as examples, we describe the performance estimation using

Table 3.7 Performance of amorphous 1536 atomic system in 2048 processes (K computer)

Block name	Elapsed time (s)	Ratio of peak performance (%)
SCF	39.79	20.11
Block1 (FFT)	0.02	2.19
Block2 (BLAS)	6.89	53.53
Block3 (FFT)	3.99	3.56
Block4 (BLAS)	1.88	64.76
Block5 (BLAS, comm.)	2.53	17.32
Block6 (FFT)	1.24	5.15
Block8 (FFT, BLAS)	4.16	16.56
Block9 (BLAS, ScaLAPACK)	12.31	3.88
Block10 (BLAS)	1.89	64.30
Block11 (FFT)	5.05	4.78

the roofline model described in Sect. 2.8.8. The results for Seism3D are shown in Sect. 3.8.2 and those for FFB in Sect. 3.9.2 [7].

3.8 Single-CPU Performance Optimization of Seism3D

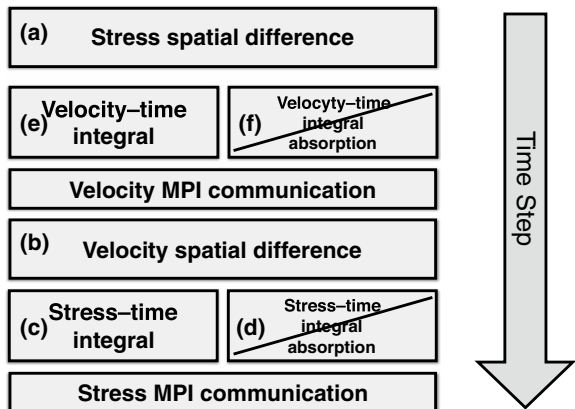
3.8.1 Overview of Seism3D

Seism3D [3, 4] is a large-scale parallelization program that solves earthquake propagation and tsunamis in conjunction with each other by the time evolution of the viscoelastic equation using the finite difference method (staggered lattice difference method). Seism3D is composed of the following six calculation kernels:

- (a) Stress spatial difference calculation
- (b) Velocity spatial difference calculation
- (c) Stress–time integral calculation
- (d) Stress–time integral absorption calculation
- (e) Velocity–time integral calculation
- (f) Velocity–time integral absorption calculation.

There are two versions of Seism3D. The initial version did not implement steps (d) and (f) and used 3D domain decomposition for the parallelization. This version only dealt with earthquake simulation. Figure 3.13 shows the calculation flow of the initial version. The full version was then developed by implementing the physical properties of water on top of the mesh to simulate tsunamis as well as earthquakes. In this version, to overcome the imbalance of calculation times between earthquake and tsunami, a 2D domain decomposition in the horizontal direction was adopted, and the processing of (d) and (f) was included. Calculations in this version required large B/F values, and the communications were only in the halo region in the horizontal direction.

Fig. 3.13 Calculation flow of Seism3D



The results of the investigation of the program flow for the original version of Seism3D are described below. Seism3D can explicitly obtain the propagation of the seismic waves in a heterogeneous underground structure from the equations of motion (stress–equilibrium equation) and the constituent equations of stress–distortion by using the difference calculation method (time: second-order accuracy, space: fourth-order accuracy). The equations of motion are

$$\rho \ddot{u}_x = \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + \frac{\partial \sigma_{xz}}{\partial z} + f_x, \quad (3.5)$$

$$\rho \ddot{u}_y = \frac{\partial \sigma_{yx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{yz}}{\partial z} + f_y, \quad (3.6)$$

$$\rho \ddot{u}_z = \frac{\partial \sigma_{zx}}{\partial x} + \frac{\partial \sigma_{zy}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} + f_z. \quad (3.7)$$

Here, \ddot{u} is an acceleration, σ is stress, ρ is a density, and f is an external force. The stress–strain constitutive equation (Hooke’s law) is

$$\sigma_{pq} = \lambda(e_{xx} + e_{yy} + e_{zz})\delta_{pq} + 2\mu e_{pq}. \quad (3.8)$$

Here, e is a distortion, λ and μ are constants of Lamé, δ is the Kronecker delta, and the distortion can be found from

$$e_{xx} = \frac{\partial u_x}{\partial x}, e_y = \frac{\partial u_y}{\partial y}, e_z = \frac{\partial u_z}{\partial z}, \quad (3.9)$$

$$e_{xy} = \frac{1}{2} \left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right), e_{yz} = \frac{1}{2} \left(\frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y} \right), e_{zx} = \frac{1}{2} \left(\frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} \right). \quad (3.10)$$

3.8.2 Evaluation of Single-CPU Performance of Seism3D

We mentioned in Sect. 2.8.9 that the sparse matrix–vector product often appears when a partial differential equation is discretized. It appears in Seism3D. The sparse matrix is a type of scalar value and the vector is represented as a 3D array. Because only a few elements appear in each row of the sparse matrix, the reusability of the vector elements is limited to the same number.

The performance prediction from the roofline model shown in Sect. 2.8.8 is described. Among the six calculation blocks shown in Sect. 3.8.1, the same calculations are performed for the spatial differential calculations of (a) and (b). As our example, we use the coding of the velocity difference term in the Z-direction in (a) and (b) of Fig. 3.14. This coding is the sparse matrix–vector product and the

Fig. 3.14 Velocity finite difference calculation of Z-direction

```

do J = 1, NY
  do I = 1, NX
    do K = 3, NZ-1
      DZV (k,I,J) = (V(k,I,J) -V(k-1,I,J))*R40 &
        - (V(k+1,I,J)-V(k-2,I,J))*R41
    end do
  end do
end do

```

sparse matrix is the scalar type. In this loop, thread parallelization through the block division is implemented in the J loop of the outermost loop. Here, $NZ = 4000$, $NX = 60$, and $NY = 80$. This program is coded in single precision. Therefore, the size of the entire array of V and DZV is 76.8 MB, the size of the K axis \times I axis is 960 KB, and the size of the K axis is 16 KB. The entire array is too large to store in the cache, though one occurrence of the K axis \times I axis is small enough to store in the L2 cache, and the K axis can be stored in the L1 cache. In this example, the FLOP value is 5,³ and the B/F value required is obtained from the coding as follows. First, we calculate the required byte value. $V(K - 2, I, J)$ is loaded from the memory first. Assuming that $V(K - 1, I, J)$, $V(K, I, J)$, and $V(K + 1, I, J)$ are in the same cache line at this time, these data do not need to be loaded separately from the memory but can be loaded from the L1 cache because they are stored together in the L1 cache. $DZV(K, I, J)$ is loaded once from memory and then stored in memory. Therefore, the number of loads and stores from memory is 3.⁴ The number of required FLOPS is 5 when counting the number of operations from Fig. 3.14. In the roofline model, only memory accesses are considered without considering cache accesses. Therefore, the byte value required from the coding in Fig. 3.14 is 12 bytes in 3×4 bytes, and the required B/F value is $12/5 = 2.4$. In the roofline model, the estimated performance is obtained by dividing the required B/F value by the hardware B/F value. The B/F value of the hardware of K computer can be obtained by dividing 64 GB/s by 128 GFLOPS. Using this value, $0.5/2.4 = 0.208$, so 20.8% is the predicted performance.

Performance prediction of Seism3D

We found that the estimation accuracy can be improved by using the effective peak performance without using the theoretical peak performance of the memory bandwidth. The effective memory bandwidth between the CPU and memory of K computer is 46 GB/s; the B/F value of the hardware (0.5) is multiplied by the ratio, 0.72 ($=46/64$), between the theoretical memory bandwidth (64 GB/s) and the effective memory bandwidth (46 GB/s); and the effective B/F value of the hardware is obtained as 0.36. Therefore, the predicted performance value is $0.36/2.4 = 0.15$, and it can be predicted that the performance of 15% of the peak performance ratio is the maximum performance of this coding.

³In this case, we consider “add” as 3 and “multiply” as 2.

⁴In this case, we consider “store” as 1 and “load” as 2.

Next, the coding of two difference calculations in the X- and Y- direction is shown in Figs. 3.15 and 3.16. The performance prediction in the X-direction is almost the same as that for the Z-direction. The difference is that three V elements out of the four V loads are predicted, as the data are stored in the L1 cache in the Z-direction calculation, whereas in the X-direction calculation, because it is in the range of $16 \text{ KB} \times 4 = 64 \text{ KB}$, it is predicted to be stored in either the L1 cache or the L2 cache. In either case, no memory access occurs, so the required B/F value is 2.4 as in the Z-direction difference and the predicted performance is 15%.

The performance prediction in the Y-direction is rather different from the prediction in the Z or X-direction. The size of the K axis \times I axis is 960 KB, and considering that thread parallelization is applied on the J axis, none of the four elements of V will remain in the cache. Therefore, there are six load/stores from memory and the 24 bytes are required bytes, so the required B/F value becomes $24/5 = 4.8$, and the predicted performance becomes 7.5%.

Measurement results and evaluation of the space differential calculation of Seism3D

Table 3.8 summarizes the required B/F values and the predicted performance value shown in Sect. 3.8.2, together with the measured values. Table 3.8 shows that the predicted value is consistent with the measured value. From the coding shown in Sect. 3.8.2, the basic pattern is that prefetch works because the data are accessed continuously, and it seems that condition (1) in Sect. 2.6 and the effective use of

Fig. 3.15 Velocity finite difference calculation of X-direction

```

do J = 1, NY
do I = 1, NX
do K = 1, NZ
DXV (k,I,J) = (V(k,I,J) -V(k,I-1,J))*R40&
- (V(k,I+1,J)-V(k,I-2,J))*R41
end do
end do
end do
    
```

Fig. 3.16 Velocity finite difference calculation of Y-direction

```

do J = 1, NY
do I = 1, NX
do K = 1, NZ
DYV (k,I,J) = (V(k,I,J) -V(k,I,J-1))*R40 &
- (V(k,I,J+1)-V(k,I,J-2))*R41
end do
end do
end do
    
```

Table 3.8 Velocity finite difference calculation of Seism3D

	Z-direction	X-direction	Y-direction
Required B/F value	2.4	2.4	4.8
Predicted performance value (%)	15.0	15.0	7.5
Measured performance value (%)	15.3	15.1	7.6

prefetch are satisfied. Because it is a continuous access, it also satisfies condition (2), that is, the effective use of line accesses. Because these conditions are satisfied, it is reasonable that the predicted value is consistent with the measured value. We concluded that there are no performance problems with this coding.

3.8.3 Modification of the Code for Improved Performance of Seism3D

In this section, detailed examples of tuning the sparse matrix–vector product shown in Sect. 2.8.9 are described.

Cyclic division thread parallelization

As a further tuning method, we first consider the loop fusion of each loop of ZXY. Because there are references to the elements of $V(K, I, J)$ in each of the three loops, by fusing loops it may be possible to halve the load of $V(K, I, J)$ overall. Next, we can change the outermost J loop from thread parallelization by the block division method to thread parallelization by the cyclic division method. In the J -axis loop, $V(K, I, J)$ is loaded from memory. However, for the sequence $V(K, I, J - 1)$, $V(K, I, J - 2)$, and $V(K, I, J + 1)$, the thread parallelization by cyclic division of the J -axis can be expected to use the elements of V loaded into the L2 cache by both the neighboring threads. This takes advantage of the characteristics of the K computer in which the L2 cache is shared by the eight cores. Figure 3.17 shows the code after tuning.

We consider the predicted performance value of this post-tuning coding. First, the FLOP value is 15. As before, one element of V is loaded from memory. At that time, the other 11 elements of V are thought to be stored in the L1 or L2 cache. $DZV(K, I, J)$, $DXV(K, I, J)$, and $DYV(K, I, J)$ are loaded once from memory and then stored in memory. Therefore, there are seven load/stores from the memory. The coding in Fig. 3.17 requires 7×4 bytes = 28 bytes, and the required B/F value is $28/15 = 1.86$. The predicted performance value is therefore 19% from $0.36/1.86 = 0.19$.

Fig. 3.17 Coding of cyclic division thread parallelization

```
!$OMP DO SCHEDULE(static,1),PRIVATE(I,J,K)
do J = 1, NY
  do I = 1, NX
    do K = 3, NZ-1
      DZV (k,I,J) = (V(k,I,J) -V(k-1,I,J))*R40 &
        - (V(k+1,I,J)-V(k-2,I,J))*R41
      DXV (k,I,J) = (V(k,I,J) -V(k,I-1,J))*R40&
        -(V(k,I+1,J)-V(k,I-2,J))*R41
      DYV (k,I,J) = (V(k,I,J) -V(k,I,J-1))*R40 &
        - (V(k,I,J+1)-V(k,I,J-2))*R41
    end do
  end do
end do
```

Table 3.9 Result of applying cyclic division thread parallelization method

Required B/F value	$28/15 = 1.86$
Predicted performance value	$0.36/1.86 = 0.19$
Measured performance value	17.7%

Table 3.10 Result of stress time integral calculation

Required B/F value	$252/175 = 1.44$
Predicted performance value	$0.36/1.44 = 0.25$
Measured performance value	17.4%

Table 3.9 summarizes the performance predictions and the measured results for this coding. The performance of about 18% is obtained for all of ZXY, and the predictions and measurements are consistent. The tuning effort has improved the original performance.

Performance prediction and measurement of stress–time integral calculation

Among the six calculation blocks shown in Sect. 3.8.1, the performance prediction and the measurement result of the stress–time integral calculation of (c) are shown in Table 3.10. Because there is no difference calculation such as -1 , $+1$ that has been evaluated so far, the stress calculation section is an example with no vector reusability, even in the sparse matrix–vector product. All the arrays are accessed with indexes (K, I, J) as the accesses of DXV, DYV, and DZY that appeared in the ZXY difference calculation. Therefore, all the arrays are accessed from memory. Because the coding length is about 100 rows, although not described here, an evaluation similar to the previous discussion was carried out and the required B/F value and the predicted performance value were obtained. Table 3.10 shows that the measured value is much lower than the predicted value.

Tuning of stress–time integral calculation

Based on this result, we investigated whether there was a prospect of improving the performance. When we examined the profile data in detail, the measured result for the memory bandwidth was about 35 GB/s, which shows that it does not satisfy the condition (1) shown in Sect. 2.6. When looking at the prefetching situation, we found that the hardware prefetching was not used efficiently. We reduced the number of streams by fusing several sequences and reducing the number of sequences to increase the efficiency of the hardware prefetching. With this tuning, the execution performance improved from 17.4 to 21.8% and approaches the predicted performance value, and the memory bandwidth value is also 42.4 GB/s, which is close to the 46 GB/s effective value.

3.8.4 Results for the High-Performance Version of Seism3D and the Total Performance

Similar evaluations and tuning were carried out for all of (a)–(f) shown in Sect. 3.8.1. Table 3.11 summarizes the results of solving the L1 cache conflict and so on. The results show the peak performance of the original and tuned code including the communication, and that the performance has improved from 10.3% to 15.3%. For the differential/integral calculation, we show the single-CPU performance without the communications, and the performance improvement is confirmed in either case. By further tuning such as the use of the XFILL⁵ control statement for the efficiency of the data store, the single-CPU performance without communications was improved up to 17.5%.

Finally, Fig. 3.18 shows the results of measuring the performance with weak scaling from 16 nodes to 82,944 nodes. Good weak scalability up to 80,000 nodes is obtained, and the total performance of all nodes is 2.1 PFLOPS, achieving the peak performance ratio of 19.7%.

Table 3.11 Performance improvement result of Seism 3D(1)

	Original		Tuned code	
	Elapsed time (s)	Ratio of peak performance (%)	Elapsed time (s)	Ratio of peak performance (%)
Overall	75.5	10.3	52.9	15.3
Stress spatial difference calculation	13.0	10.4	9.2	14.7
Velocity spatial difference calculation	13.4	10.1	7.7	17.7
Stress–time integral calculation	12.8	17.0	11.5	21.8
Stress–time integral absorption calculation	12.5	7.6	10.3	10.2
Velocity–time integral calculation	9.7	7.5	3.0	23.0
Velocity–time integral absorption calculation	7.9	15.3	6.9	17.5

⁵Control statement for streamlining array access without loading.

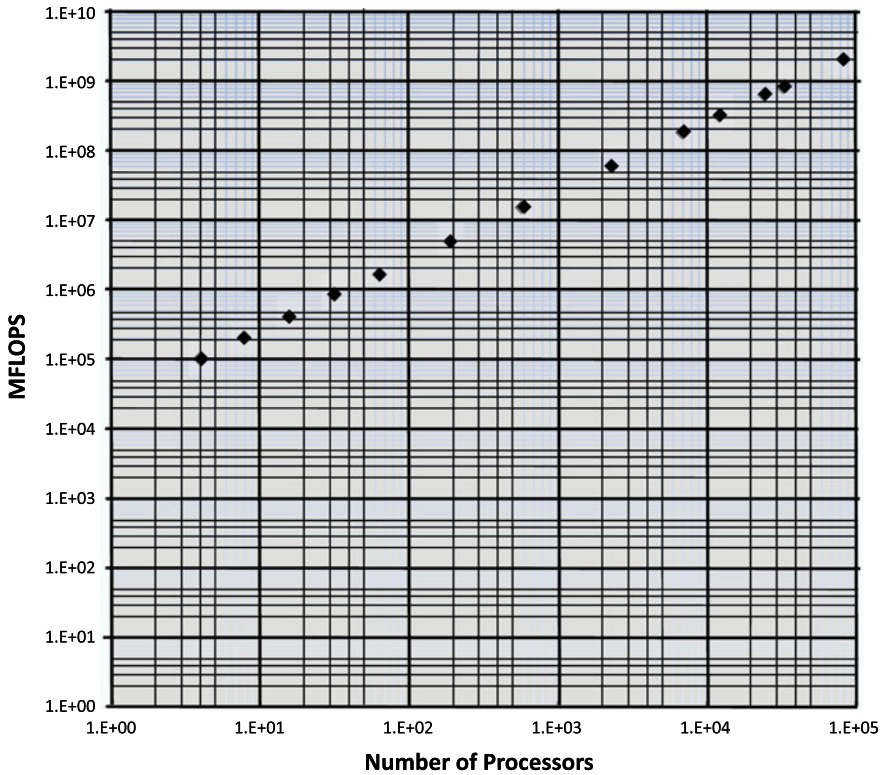


Fig. 3.18 Total performance of Seism3D

3.9 Single-CPU Performance Optimization of FFB and an Example

3.9.1 Overview of FFB

FFB [7] is a general-purpose fluid analysis application based on large eddy simulation (LES) that can predict the unsteady flow of uncompressed fluids with high accuracy. It can predict the noise generated from fluid machines such as fans or pumps by adopting discretization using the finite element method with excellent shape conformity, unsteady turbulent flow around complex shapes, and flow. There are two types of calculation methods in the finite element method. The first constructs an overall stiffness matrix and solves the matrix using the implicit method, and the other advances the calculation using only the element rigidity matrix without constructing the overall configuration matrix (element-by-element method). The new version of FFB is compatible with both the solvers.

Figure 3.19 shows the processing flow of the FFB solver. The calculation of the

velocity predictor, the calculation of pressure, and the velocity correction calculation are the main parts of this calculation. The flow of the subroutines called from the calculation of the velocity predictor is shown in Fig. 3.20. The main parts of the calculation are bcgs2x and calaxc, which calculate the matrix–vector product. The configuration of the subroutines called from the calculation of pressure is shown in Fig. 3.21. This calculation has a nodal pressure mode in which an entire stiffness matrix is constructed and solved by the implicit method, and an element pressure mode in which the calculation is advanced using the element-by-element method. The center of the calculation of the nodal pressure mode is calaxc, which is also

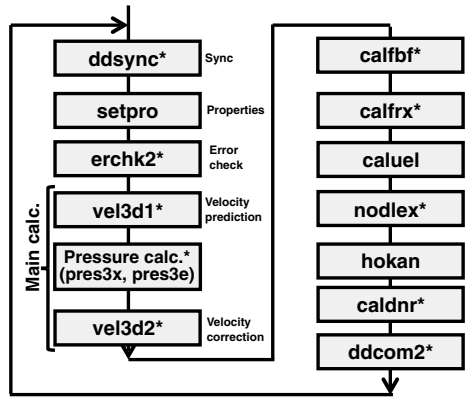


Fig. 3.19 Calculation flow of FFB

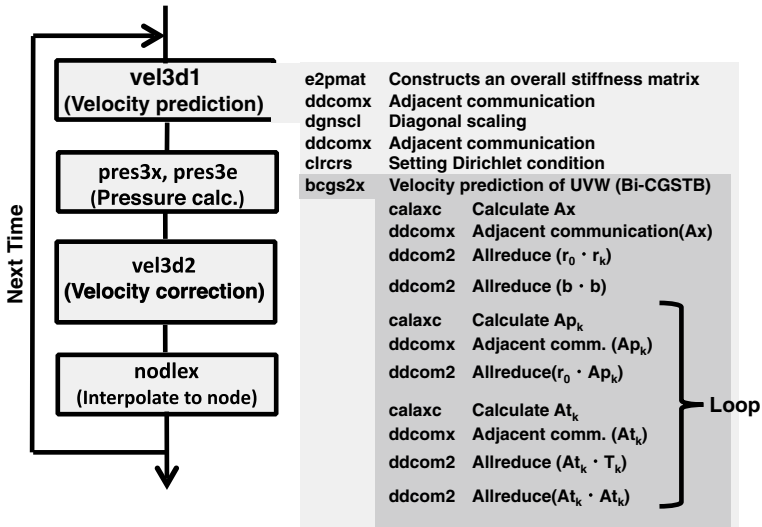


Fig. 3.20 Calculation flow of velocity prediction

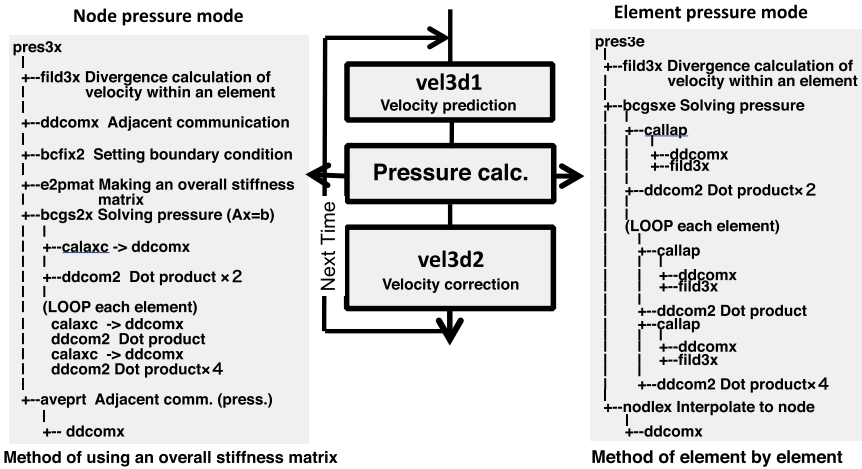


Fig. 3.21 Calculation flow of pressure calculation

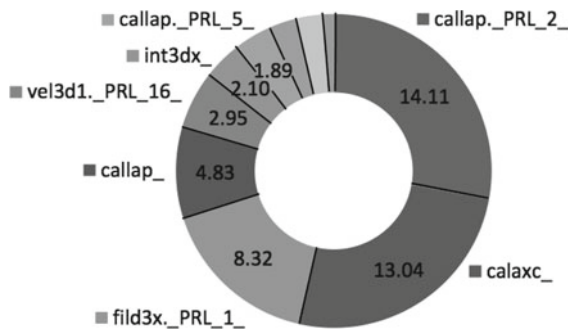
the center of the calculation of vel3d1. The center of the calculation of the element pressure mode is callap.

3.9.2 Evaluation of Single-CPU Performance of FFB

Investigation of the execution characteristics of FFB

Figure 3.22 shows the execution time distribution of the subroutines when 100,000 tetrahedron elements are calculated. The gradient calculations of the tetrahedral elements (callap) required 14.1 s (30%) and 4.8 s (10%), and the sparse matrix–vector products (calaxc) required 13.0 s (27%). As shown in Figs. 3.20 and 3.21, the main communication processing within the FFB time integration loop is of the following two types. The first performs the global communication in the subroutine ddcom2, and MPI_ALLREDUCE (MPI_SUM) is called in all MPI ranks. The second per-

Fig. 3.22 Execution time distribution of FFB



forms the adjacent communication in the subroutine `ddcomx`. The communication of physical quantities on the nodes shared with adjacent regions is performed in each region of the domain decomposition processing. The communication processing uses the following procedure:

- Packing of the communication data to the transmission buffer
- Asynchronous communication (`MPI_ISEND`, `MPI_IRECV`) call
- Waiting for the asynchronous communication (`MPI_WAITALL`)
- Adding the communication data from the receive buffer to the actual array.

From the configuration of these subroutines, we found that the center of calculation is `calaxc` and `callap`, and the center of communication is `ddcomx` and `ddcom2`.

Sparse matrix–vector product of FFB

FFB is a fluid calculation program using the finite element method. As described above, the finite element method includes two types of calculations; one builds the entire rigidity matrix, and the other is an element-by-element method that advances calculation with only the element rigidity matrix without constructing the entire configuration matrix. First, we describe the sparse matrix–vector product kernel used to build the overall rigidity matrix. The coding of the kernel of the sparse matrix–vector product of FFB is shown in Fig. 3.23. The general compressed sparse row (CSR) format⁶ is used for the matrix data storage. The vectors are accessed using a list array that stores the surrounding nodal numbers for calculating a certain node. Because the average number of elements in each row of the sparse matrix is about 30, the reusability of the vector elements is about 30.

Next, we describe the computational kernel used to calculate the element-by-element method. The coding of the kernel is shown in Fig. 3.24. In this example, the finite element approximation is

$$\nabla p = \frac{\partial p}{\partial x_i} = \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e. \quad (3.11)$$

It is calculated for the tetrahedral elements. The shape function stored in `DNX` and the others and the value of the pressure stored in `S` are continually referenced. The gradient value of the pressure represented by `FX` and the others appearing on

Fig. 3.23 Matrix–vector product coding of FFB

```

ICRS=0
DO 110 IP=1,NP
  BUF=0.0E0
  DO 100 IP=1,NP
    ICRS=ICRS+1
    IP2=IPCRS(ICRS)
    BUF=BUF+A(ICRS)*S(IP2)
100  CONTINUE
    AS(IP)=AS(IP)+BUF
110  CONTINUE

```

⁶A method for storing only the nonzero elements of a sparse matrix.

Fig. 3.24 Kernel coding of element-by-element method

```

DO IE=1,NE
  IP1=NODE(1,IE)
  IP2=NODE(2,IE)
  IP3=NODE(3,IE)
  IP4=NODE(4,IE)
  SWRK=S(IE)
  FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
  FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
  FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
  FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)
  FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
  FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
  FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
  FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)
  FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
  FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
  FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
  FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
ENDDO

```

the right side are stored in each node, and the nodal number is accessed by a list arrangement with the element number as an index.

Performance prediction of FFB

First, we describe the sparse matrix–vector product kernel used to build the overall rigidity matrix. This evaluation uses hexahedron elements with at most 27 adjacent nodal points and tetrahedral elements with at most 24 adjacent nodal points. Because in this coding we use list access to obtain the vector data, the latency of the memory access occurs when referring to the vector element and a large penalty is caused by using only one element out of one line. A remarkable performance deterioration is predicted. Even when the elements of the vector are stored in the L2 cache, they do not decrease memory accesses much, and similar penalties occur in the L2 cache. If the data of the vector used for the calculation can be stored in the L1 cache, the penalty for the latency and the line access will be eliminated and the access penalty to the vector memory can be ignored. In this case, 8 bytes are required for 2 elements \times 4 bytes and the FLOP value is 2, so the required B/F value is 4. Using the effective B/F value of 0.36, the predicted performance is $0.36/4 = 0.09$, that is 9%.

Next, we describe the computational kernel used to calculate the element-by-element method. Again, suppose that the data defined by a nodal point such as FX are stored in the L1 cache. In this case, 16 elements \times 4 bytes \times (9/4) + 1 element \times 4 bytes = 148 bytes are required, and the required FLOP value is 24, so the required B/F value is $148/24 = 6.17$. The factor (9/4) is included because nine elements are declared in the first dimension of DNX. Using the effective B/F value of 0.36, the predicted performance is $0.36/6.17 = 0.058$, that is 5.8%.

Table 3.12 Performance of matrix–vector product of FFB

	Hexahedron (ratio of 1 core peak performance)	Tetrahedron (ratio of 1 core peak performance)
Original code	5.9%	2.4%

Measurement results and evaluation of FFB kernel performance

First, we describe the sparse matrix–vector product kernel used to build the overall rigidity matrix. Initially, because the original code kernel was not thread parallelized, it was measured with one core and the results are shown in Table 3.12. The numerical value is the ratio of the peak performance to the peak performance 16 GFLOPS of one core. The result for the STREAM benchmark when occupying the memory bandwidth with one core is 20 GB/s. Therefore, the theoretical hardware B/F value is 1.25 at 20 GB/16 GFLOPS. Because the required B/F value is 4, as shown in Sect. 3.9.2, the predicted performance value is 31% at $1.25/4 = 0.31$. The value in Table 3.12 is much smaller than this predicted value. The cause is presumed to be inefficient instruction scheduling given that the iteration count of the innermost loop is at most 27, in addition to the large penalty of vector accesses described in Sect. 3.9.2.

Next, we describe the computational kernel used to calculate the element-by-element method. Because the kernel of the original code was also not thread parallelized, it was measured with one core. When predicting the performance of one core in the same way as for the sparse matrix–vector product kernel shown earlier, the predicted performance value is 20.2% at $1.25/4 = 0.202$. The measured performance ratio of the kernel was 1.6%, and the throughput of the memory was 10.4 GB/s with test data consisting of 820,000 elements. Assuming that there is no penalty for list access in the data storage on the left side because of ideal list access, the theoretical value of the L1 cache miss rate is 3.125%, but the miss rate of the result was 21.3%. The results indicate that the sparse matrix–vector product kernel is incurring a large penalty for list accesses.

Based on these evaluation results, the penalty for the large list access has occurred, which appears to be the problem limiting performance.

3.9.3 Modification of the Code for High Performance of FFB

We now discuss a detailed analysis of the sparse matrix–vector product when the required B/F value is large and list vectors are used, as shown in Sect. 2.8.10.

Data storage format with full unrolling

According to D. Guo and colleagues, it has been reported that performance improvement can be achieved by changing the matrix storage method from the CSR format to the streamed-CSR (S-CSR) method, which extends the CSR format [16]. When we tried to use it, the actual measurement result was improved as reported by Guo; how-

Fig. 3.25 Matrix–vector product fully unrolled coding of FFB

```

ICRS=0
DO 110 IP=1,NP
  BUF=0.0E0
  BUF=BUF+A(ICRS+ 1)*S(IPCRS(ICRS+ 1))
  &      +A(ICRS+ 2)*S(IPCRS(ICRS+ 2))
  . . . . . (omit) . . . . .
  &      +A(ICRS+26)*S(IPCRS(ICRS+26))
  &      +A(ICRS+27)*S(IPCRS(ICRS+27))
  ICRS=ICRS+27
  AS(IP)=AS(IP)+BUF
110 CONTINUE
    
```

ever, the performance was improved by only about 7% for the hexahedron and about 10% for the tetrahedron. As shown in Fig. 3.25, we also adopted the data storage format and coding that completely unrolled the inner loop. This method was applied to the sparse matrix–vector product kernel used in the calculation of the overall rigidity matrix. Figure 3.25 shows the code for the hexahedron; for the tetrahedron, the number of unrollings was 24. For the hexahedron with this data storage format, if the number of matrix elements is fewer than 27 for each row of the control variable IP in the DO sentence in Fig. 3.25, the rest are filled with zeroes until there are 27 to fix the expansion number. The number of elements filled with zeroes is about 2% of the total for the hexahedron and about 35% for the tetrahedron. The memory amount and the calculation amount increase; however, the calculation performance still improves.

Reordering the vector data

We modified the ordering of the nodal points to reduce the access penalty for the vector shown in Sect. 3.9.2. As shown on the left side of Fig. 3.26, the nodal point numbers were remapped to 3D space using the 3D XYZ coordinates of the nodal points. Next, the 3D space was divided in each axis direction. In Fig. 3.26, it was divided into three; this time, it was divided into 10. Basically, the ordering of nodes was changed so that the nodal points included in the divided box are consecutively stored, and the numbers of the boxes are also taken from the order of XYZ, as shown

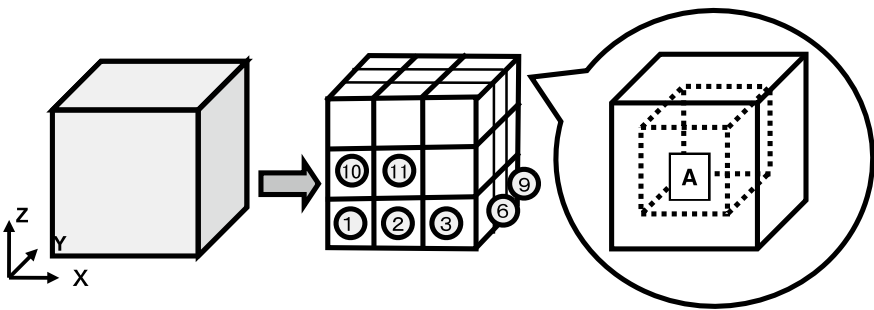


Fig. 3.26 Changing the ordering method of nodes

in the middle diagram of Fig. 3.26. The nodal points in the box were also divided into inside and outside, as shown on the right side of Fig. 3.26, and the nodal points included inside were ordered first, and then the nodal points included in the outer circumference were ordered. In this way, by changing the ordering, the physically close nodal points are assigned to close positions in the sequence of the array, and eventually, the nodal point numbers constituting one element are also close. Adjusting the size of one box with this tuning allows the data to be stored in the L1 cache for many of the list accesses of the vector, thus greatly reducing the penalty for list accesses. This method is applicable to the computational kernels for both the overall rigidity matrix and the element-by-element method.

Tuning of the element pressure kernel

The element pressure kernel holds the nodal point numbers belonging to the element as a list. In the element pressure kernel, the loop index is the element number. In the loop, the calculation result for each node in each element is added to the array of the nodal points. The neighboring elements may refer to the same nodal point number to add values; in that case, there is a recurrence and it becomes impossible to parallelize by threads. Therefore, a coloring process is performed to divide the elements in which the data dependency occurs between different groups (colors). An overview of this coloring is shown in Fig. 3.27. Because there is no recurrence within each color, the thread parallelization is performed within each color. The colors are arranged to be continuous in memory space, and the efficiency of memory access is improved. We next implemented the ordering method of the vector data described above. Through the above processing, the elements and nodal points referenced in the innermost loop are localized in terms of space and memory, and the localization of the memory access is realized.

The performance measurement at this stage revealed that the number of elements included in the small area inner color decreased because of the combination of the domain decomposition and coloring. This made the iteration count of the innermost loop insufficient, the efficiency of the computation scheduling decreased, and performance deterioration was observed. Therefore, while taking advantage of the localization of the memory access by using the domain decomposition and the nodal point renumbering, and multithread execution by means of coloring, we changed the

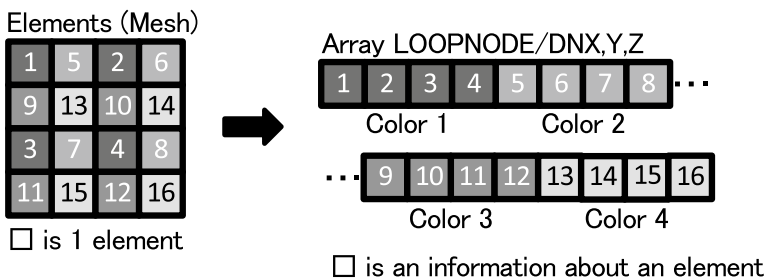


Fig. 3.27 Ordering element by coloring

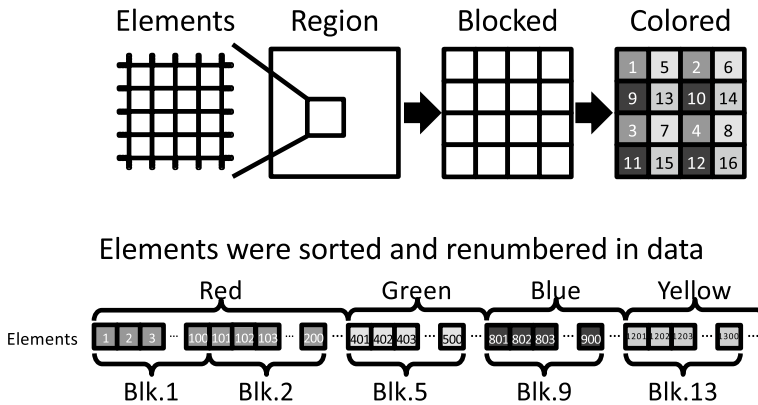


Fig. 3.28 Ordering block by coloring

object to be colored to avoid the small iteration count of the innermost loop. In other words, as shown in Fig. 3.28, we changed the coloring target from the element to the block, which is a set of elements [15]. In the previous version, there was no data dependency for the operation of the innermost loop and SIMD vectorization (a.k.a. simdization) and software pipelining were applied by the compiler. In this improved version, coloring was for small area units, and data dependencies occur in the calculation of the innermost loop, so the SIMD vectorization and the software pipelining were not applied.

3.9.4 Results of Performance Improvement of FFB

Performance of the nodal pressure kernel

Table 3.13 shows the measurement results after tuning the nodal pressure kernel. The numbers in the figure are the ratio of peak performance to 16 GFLOPS, the peak performance of one core, and the ratio of peak performance to 128 GFLOPS for

Table 3.13 Performance of sparse matrix–vector product kernel

	Hexahedron (ratio of peak performance) (%)	Tetrahedron (ratio of peak performance) (%)
Original code (1 core)	5.9	2.4
Full unroll code (1 core)	10.8	4.2
Full unroll code (8 core)	5.4	3.0
Full unroll + Reordering (1 core)	10.2	10.2
Full unroll + Reordering (8 core)	8.1	7.7

eight cores. With full unrolling, a performance improvement of about two times for the tetrahedrons and the hexahedrons is obtained in the one core measurements. The performance value close to 9%, which is the theoretical performance value when the vector shown in Sect. 3.9.2 is ideally stored in the L1 cache, is obtained with eight cores by changing the full unrolling and the nodal ordering. This shows that the tuning method implemented here is effective.

The performance of the element pressure kernel

Figure 3.29 shows the source code after tuning the performance of the element pressure kernel. When the coloring and the reordering of the elements were included and the thread parallelization was performed, the peak performance ratio of 1.6% was obtained. By changing the object of the coloring to block coloring, the peak performance ratio of 3.78% was achieved. By applying the array fusion technique, the performance upgraded to the peak performance ratio of 4.41%. By improving the balance of the number of elements between the blocks, the peak performance ratio of 4.58% and memory throughput of 41.2 GB/s were achieved. The list access for storing data does not reach the theoretical performance value because the penalty for cache misses is large.

```

DO ICOLOR=1,NCOLOR(1)
  DO IGROUP=1,NGROUP(ICOLOR,1) ——Parallelize this loop
    IES=LLOOP(IGROUP,ICOLOR ,1)+1
    IEE=LLOOP(IGROUP,ICOLOR+1,1)
    DO IE=IES,IEE
      IP1=NODE(1,IE)
      IP2=NODE(2,IE)
      IP3=NODE(3,IE)
      IP4=NODE(4,IE)
      SWRK=S(IE)
      FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
      FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
      FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
      FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)
      FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
      FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
      FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
      FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)
      FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
      FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
      FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
      FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
    ENDDO
  ENDDO
ENDDO

```

Fig. 3.29 Improved kernel coding of element-by-element method

3.9.5 Total Performance of FFB

Figure 3.30 shows the results for FFB (version 7) with 40,000 and 80,000 parallel paths with weak scaling. The final overall peak performance ratio including communication is 3.16%. The MPI_ALLREDUCE communication time for scalar values for the inner product calculation originally considered as the only problem with high parallelization is represented by tddcom2 in each graph. The graph shows that tddcom2 is smaller than the total execution time. This is the performance obtained by using high-speed, one-element MPI_ALLREDUCE communication, which is available in the hardware assistance of the K computer.

Exercise

1. As shown in Table 3.8, confirm that the required B/F value matches the measured value by programming Figs. 3.14, 3.15, and 3.16 and checking it. At that time, NX, NY, and NZ should be set appropriately according to the L1 cache of the computer environment to be used and the capacity of the last-level cache so that the data for the difference term of the first dimension is placed in the L1 cache and the data for the difference term of the second dimension is placed in the last-level cache.

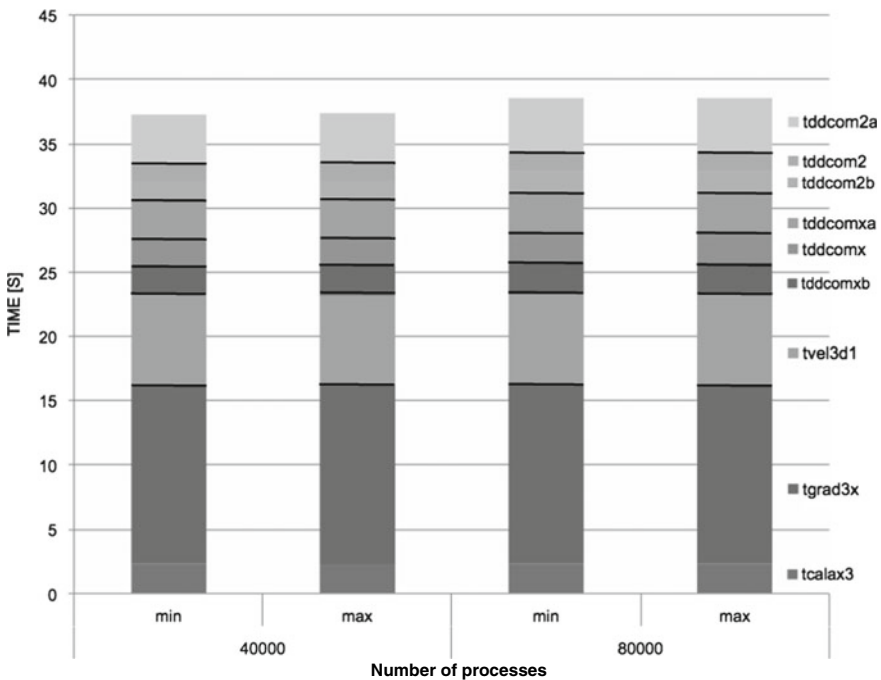


Fig. 3.30 Measurement result of scaling of FFBver7

Acknowledgements The work to summarize our achievements described from Chaps. 1 to 3 is partially supported by the Grant-in-Aid for Scientific Research(B) 16H02822, “Facilitating performance-aware programming with machine learning techniques.”

References

1. M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, T. Watanabe, in *2011 International Symposium Low Power Electronics and Design (ISLPED)*, vol. 371 (2011)
2. M. Satoh, T. Matsuno, H. Tomita, H. Miura, T. Nasuno, S. Iga, *J. Comput. Phys.* (the Special Issue on Predicting Weather, Climate and Extreme events) **227**, 3486 (2008)
3. Y. Furumura, L. Chen, *Parallel Comput.* **31**, 149 (2005)
4. T. Fukumura, “Earthquake 2” (in Japanese) **61**, S83 (2009)
5. <https://azuma.nims.go.jp/> (The software name “PHASE” changed to “PHASE/0” now.)
6. J. Iwata, D. Takahashi, A. Oshiyama, T. Boku, K. Shiraishi, S. Okada, *J. Comput. Phys.* **229**, 2339 (2010)
7. [Turbulent sound field analysis software FrontFlow/Blue] (in Japanese), http://www.ciss.iis.u-tokyo.ac.jp/rss21/theme/multi/fluid/fluid_softwareinfo.html
8. C. Kato, Y. Yamade, H. Wang, Y. Guo, M. Miyazawa, T. Takaishi, S. Yoshimura, Y. Takano, *Comput. Fluids* **36**, 53 (2007)
9. S. Aoki, K. Ishikawa, N. Ishizuka, T. Izubuchi, D. Kadoh, K. Kanaya, Y. Kuramashi, Y. Namekawa, M. Okawa, Y. Taniguchi, A. Ukawa, N. Ukita, T. Yoshie, *Phys. Rev. D* **79**, 034503 (2009)
10. FUJITSU [Feature:Supercomputer [K]] (in Japanese) **63**(3) (2012)
11. Y. Ajima, S. Sumimoto, T. Shimizu, *IEEE Comput.* **42**, 36 (2009)
12. Y. Hasegawa, J. Iwata, M. Tsuji, D. Takahashi, A. Oshiyama, K. Minami, T. Boku, F. Shoji, A. Uno, M. Kurokawa, H. Inoue, I. Miyoshi, M. Yokokawa, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11* (ACM, New York, 2011), p. 1:1
13. A. Kuroda, Y. Hasegawa, M. Terai, S. Inoue, S. Ichikawa, H. Komatsu, N. Ohi, T. Ando, T. Yamazaki, T. Ohno, K. Minami, in *Proceedings of High Performance Computing and Computational Science* (in Japanese), vol. 144 (2012)
14. K. Kumahata, S. Inoue, K. Minami, *Procedia Comput. Sci.* **18**, 2496 (2013)
15. K. Kumahata, S. Inoue, K. Minami, *IPJS Trans. Adv. Comput. Syst. (ACS)* (in Japanese) **6**, 31 (2013)
16. D. Guo, W. Gropp, *IJHPCA* **25**, 115 (2011)

Chapter 4

$O(N)$ Methods



Taisuke Ozaki

In this chapter, we will discuss $O(N)$ methods for first-principles calculations based on density functional theory (DFT) which provides a first-principles framework enabling us to efficiently calculate electronic structures of materials with quantitative accuracy, where N is the number of atoms. In general, the computational complexity of the DFT calculations scales as the third power of the number of atoms, which may hamper its applications to large-scale systems. The $O(N)$ methods discussed here extend the applicability of DFT to large-scale systems by reducing the computational complexity from $O(N^3)$ to $O(N)$.

4.1 Density Functional Theory (DFT)

Let us review history and theoretical framework of DFT as background of $O(N)$ methods before discussing theories and the practical aspects of the $O(N)$ methods. In chemistry and condensed matter physics computational methods of electronic structures of materials have been developed individually after the fundamental equations in quantum mechanics were founded by Schrödinger and Dirac. DFT is the first-principles method originated from the field of condensed matter physics, and the early prototypes were rather intuitively devised to treat condensed matters consisting of atoms whose number is in the order of Avogadro's number. Nowadays, the basic but crude idea can be seen in the Wigner-Seitz method [1], which is the one of earliest prototypes, and it is now regarded as semi-empirical method. In 1951, Slater proposed $X\alpha$ method which employs a local exchange potential obtained from an average of non-local exchange potential [2]. The basic equation in the $X\alpha$ method is nearly equivalent to an equation of Kohn and Sham which is the

T. Ozaki (✉)

The Institute for Solid State Physics, The University of Tokyo, Tokyo, Japan
e-mail: t-ozaki@issp.u-tokyo.ac.jp

© Springer Nature Singapore Pte Ltd. 2019
M. Geshi (ed.), *The Art of High Performance Computing
for Computational Science, Vol. 2*,
https://doi.org/10.1007/978-981-13-9802-5_4

basis equation in the present DFT. Though the $X\alpha$ method has been found to be a considerably accurate method in describing electronic structures of condensed matters and the success has been somewhat a surprise, the approach has not been regarded as a first-principles method because of insufficiency of the theoretical foundation. Being motivated by the success of the semi-empirical method by Slater, in 1964 Hohenberg and Kohn provided a mathematical proof which finds a theoretical justification [3]. They proved that the total energy of a ground state is expressible as a functional of electron density given by

$$E[n] = \int n(\mathbf{r})v_{\text{ex}}(\mathbf{r})d\mathbf{r} + F[n], \quad (4.1)$$

where $n(\mathbf{r})$ is the electron density of the many electron system, and $v_{\text{ex}}(\mathbf{r})$ is an external potential such as atomic nucleus potential. $F[n]$ is a density functional related to the kinetic energy and electron–electron interaction. The mathematical proof, provided by Hohenberg and Kohn, and Levy later, guarantees that a universal functional $F[n]$ exists, which can be applied to any many electron system. However, a study to find a concrete form of the functional has been left to later investigation. The second theorem proven by Hohenberg and Kohn is a variational principle with respect to electron density defined by

$$E = \min_n E[n]. \quad (4.2)$$

With the theorem the total energy for the ground state of an arbitrary system characterized by an external potential v_{ex} is obtained by minimizing the density functional of Eq. (4.1) with respect to electron density, and the electron density giving the minimum energy is found to be equivalent to the true electron density of the many electron system. Following the proof of the existence theorem of DFT, in 1965 Kohn and Sham proposed a practical method to perform calculations based on DFT [4]. The central equation in the method is nowadays known as Kohn-Sham (KS) equation. In the $O(N)$ methods the KS equation plays also a central role, and therefore let us explain the approach of Kohn and Sham. They have proposed that the total energy of a many electron system can be expressed by the following formula:

$$E[n] = T_s + E_{\text{en}}[n] + E_{\text{ee}}[n] + E_{\text{xc}}[n], \quad (4.3)$$

where T_s , E_{en} , E_{ee} , and E_{xc} are the kinetic energy of a fictitious noninteracting system $\{\psi\}$, classical Coulomb interaction energy between electrons and nuclei potential, classical Coulomb interaction energy between electrons, and exchange-correlation energy which is a quantum mechanical Coulomb interaction energy between electrons, respectively. Except for E_{xc} , they are explicitly given by

$$T_s = \sum_{\mu}^{\text{occ}} \int \psi_{\mu}^*(\mathbf{r}) \left(-\frac{1}{2}\Delta\right) \psi_{\mu}(\mathbf{r}) d\mathbf{r}, \quad (4.4)$$

$$E_{\text{en}}[n] = \int n(\mathbf{r}) v_{\text{ex}}(\mathbf{r}) d\mathbf{r}, \quad (4.5)$$

$$E_{\text{ee}}[n] = \frac{1}{2} \iint \frac{n(\mathbf{r}_1)n(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_1 d\mathbf{r}_2, \quad (4.6)$$

where the summation in Eq.(4.4) is taken over occupied states. By considering a correspondence of Eqs.(4.3) and (4.1), it turns out to be $F[n] = T_s + E_{\text{ee}}[n] + E_{\text{xc}}[n]$. If we simply follow the idea of DFT, the kinetic energy should also be expressed by a density functional. In fact density functionals of the kinetic energy have been proposed, typified by the Thomas-Fermi model which is derived from analysis of electron gas. However, the inaccuracy of such kinetic energy functionals, poor description of shell structure in atoms and large underestimation of binding energies of atoms, had already been pointed out before the proposal of Kohn and Sham in 1965. On the other hand, it had been known that the $X\alpha$ method of Slater has a considerable quantitative accuracy over the density functionals of the kinetic energy, while the theoretical foundation may not be sufficient. Therefore, Kohn and Sham have expressed the many electron wave function with a single Slater determinant by introducing the fictitious noninteracting system that the physical substance may not be clear. They expected that the majority part of the true kinetic energy is covered by the kinetic energy T_s of the noninteracting system calculated from the Slater determinant. It is also noted that all the quantum mechanical effects arose from the many electron interaction are included in E_{xc} , since E_{en} and E_{ee} are both the classical Coulomb interaction energies and T_s is the kinetic energy of noninteracting system.

In the method of Kohn and Sham, the electron density n is also calculated using the Slater determinant of the noninteracting system $\{\psi\}$ as

$$n(\mathbf{r}) = \sum_{\mu}^{\text{occ}} \psi_{\mu}^*(\mathbf{r}) \psi_{\mu}(\mathbf{r}), \quad (4.7)$$

where the summation is taken over occupied states as in Eq.(4.4). Noting from Eq.(4.7) that n is written by $\{\psi\}$, and considering $\delta E/\delta\psi^* = 0$ under the orthonormalization constraint for $\{\psi\}$, we obtain the following equations:

$$\hat{h}_{\text{KS}}|\psi_{\mu}\rangle = \varepsilon_{\mu}|\psi_{\mu}\rangle, \quad (4.8)$$

$$\hat{h}_{\text{KS}} = -\frac{1}{2}\Delta + v_{\text{eff}}(\mathbf{r}), \quad (4.9)$$

$$v_{\text{eff}}(\mathbf{r}) = v_{\text{ex}}(\mathbf{r}) + v_{\text{H}} + v_{\text{xc}}(\mathbf{r}), \quad (4.10)$$

where the Hartree potential is defined by $v_{\text{H}}(\mathbf{r}) \equiv \int \frac{n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}'$, and the exchange-correlation potential $v_{\text{xc}}(\mathbf{r}) \equiv \delta E_{\text{xc}}/\delta n(\mathbf{r})$. The KS equation is defined by Eqs.(4.8)–(4.10), and it is found that the many electron problem can be cast to a single particle

problem under an effective potential v_{eff} . Since v_{eff} consists of electron density which is originally obtained by solving Eq. (4.8), we need to solve the equations until a self-consistent condition (SCF) is achieved, i.e., an output electron density n_{out} is equal to the input electron density n_{in} . As we discussed already, the electron density n is the variational parameter in the variational principle, Eq. (4.2), of Hohenberg and Kohn. On the other hand, $\{\psi\}$ is the variational parameter in the derivation of the KS equation. Thus, it is not apparent that Eq. (4.2) is hold in the KS framework. So, let us now consider $\delta E/\delta n$. By expressing the kinetic energy of noninteracting system with Eq. (4.8) as $T_s = \sum_{\mu} \varepsilon_{\mu} - \int n(\mathbf{r})v_{\text{eff}}(\mathbf{r})d\mathbf{r}$, and considering the variation of each term in Eq. (4.3) for a small variation δn , we can obtain the following equation:

$$\delta E[n] = \int \delta n(\mathbf{r}) \left(v_{\text{ex}}(\mathbf{r}) + \int \frac{n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}' + \frac{\delta E_{\text{xc}}}{\delta n(\mathbf{r})} - v_{\text{eff}}(\mathbf{r}) \right) d\mathbf{r}. \quad (4.11)$$

If the SCF condition is reached in the iterative calculation of the KS equation, the sum of terms in the parenthesis becomes zero.¹ Thus, we see $\delta E/\delta n = 0$ leading to the fact that the variational principle is satisfied even for the electron density n . Although it is not guaranteed that an obtained electron density n gives the lowest energy of the functional even if $\delta E/\delta n = 0$ is hold, it might be generally considered that the obtained n corresponds to the true density for the ground state except for a case that there are many low-lying states. In the method of Kohn and Sham, a many body problem is transformed to a one-body problem with an effective potential by introducing the fictitious noninteracting system, and the total energy of ground state and the true electron density satisfying the variational principle $\delta E/\delta n = 0$ are obtained in a rigorous sense by solving the KS equation self-consistently. It would be understood that the method of Kohn-Sham provides an ingenious and practical theoretical framework.

So far we have proceeded our discussion without touching the details of functional form of the exchange-correlation energy E_{xc} . It is apparent that the discussion above is all valid for the case of the exact functional E_{xc} . However, it should be noted that the discussion above is valid even for the case of an approximate functional E_{xc} , since we have never asked whether E_{xc} is exact or approximate in these derivations. The fact makes DFT so useful in practical calculations. Widely used approximations are a local density approximation (LDA) and generalized gradient approximation (GGA). In LDA, the exchange-correlation functional is approximated by taking account of the local electron density via an exchange-correlation energy density of electron gas [5, 6], while in GGA the local electron density and its gradient are taken into account to calculate the approximate exchange-correlation energy [7]. Here we show a functional form of GGA:

$$E_{\text{xc}}[n_{\uparrow}, n_{\downarrow}] = \sum_{\sigma} \int n_{\sigma}(\mathbf{r})f_{\text{xc}}(n_{\uparrow}, n_{\downarrow}, \nabla n_{\uparrow}, \nabla n_{\downarrow})d\mathbf{r}. \quad (4.12)$$

¹Note that the second and third terms in the parenthesis are calculated using the output charge density n_{out} , while the fourth term is calculated using the input charge density n_{in} .

The exchange-correlation potential $v_{xc}(\mathbf{r})$ in LDA or GGA can be constructed by only the semi-local information, electron density and its gradient at each spatial point, and thereby the computational cost to construct is also marginal. In recent years many ideas developed in quantum chemistry have been employed for development of approximate exchange-correlation functionals [8], while it was pointed out in the beginning of the section that DFT has been developed initially in the field of condensed matter physics. Along with the improvement of accuracy of the approximate exchange-correlation functionals, the applicability of DFT has been widely extended beyond chemistry and condensed matter physics more than expected.

4.2 Nearsightedness of Electron

In the method of Kohn and Sham discussed in the previous section, the KS orbitals $\{\psi\}$ are used to calculate the kinetic energy T_s of noninteracting system. Thus, let us explicitly write the total energy as $E[\psi, n]$. Though the expression $E[\psi, n]$ of total energy is the starting point of conventional DFT calculations, there are other expressions of the total energy which clearly exhibits the nearsightedness of electron being important for development of $O(N)$ methods.

The KS orbitals $\{\psi\}$ for occupied states can be transformed by a unitary transformation into Wannier functions $\{w\}$ which are localized in real space:

$$|w_\nu\rangle = \frac{V}{(2\pi)^3} \int_{\text{BZ}} d\mathbf{k} \sum_{\mu}^{\text{occ}} U_{\mu\nu} |\psi_{\mathbf{k}\mu}\rangle \exp(-i\mathbf{k} \cdot \mathbf{R}), \quad (4.13)$$

where V is the volume of the primitive cell, \mathbf{k} is the \mathbf{k} -vector, and the integration is performed over the first Brillouin zone. Also, \mathbf{R} is the lattice vector. Due to the property of unitary transformation, it is noted that the subspace spanned by the Wannier functions $\{w\}$ is equivalent to that by the KS orbitals $\{\psi\}$. The feature allows us to write the total energy using $\{w\}$ instead of $\{\psi\}$, i.e., $E[\psi, n] \rightarrow E[w, n]$.

Next let us consider to expand the KS orbitals using local basis functions $\{\chi\}$ such as atomic basis functions or finite element basis functions as

$$\psi_{\mathbf{k}\mu}(\mathbf{r}) = \frac{1}{\sqrt{N_c}} \sum_{\mathbf{R}} e^{i\mathbf{R}\cdot\mathbf{k}} \sum_{i\alpha} c_{\mathbf{k}\mu, i\alpha} \chi_{i\alpha}(\mathbf{r} - \tau_i - \mathbf{R}), \quad (4.14)$$

where N_c is the number of unit cells, and note that $\chi_{i\alpha}(\mathbf{r} - \tau_i - \mathbf{R})$ is localized in real space at the center of $(\tau_i + \mathbf{R})$. Inserting Eq. (4.14) into Eq. (4.7) and considering the integration over the first Brillouin zone as well as the summation over occupied states, we obtain the following formulae:

$$n(\mathbf{r}) = \sum_{\mathbf{R}} \sum_{i\alpha, j\beta} \rho_{i\alpha j\beta}^{(\mathbf{R})} \chi_{i\alpha}(\mathbf{r} - \tau_i) \chi_{j\beta}(\mathbf{r} - \tau_j - \mathbf{R}), \quad (4.15)$$

$$\rho_{i\alpha j\beta}^{(\mathbf{R})} = \frac{1}{V_B} \int_{\text{BZ}} d\mathbf{k} \sum_{\mu}^{\text{occ}} e^{i\mathbf{R}\cdot\mathbf{k}} c_{\mathbf{k}\mu,i\alpha}^* c_{\mathbf{k}\mu,j\beta}. \quad (4.16)$$

It is found from the expression that the total energy of a system can be rewritten using ρ instead of n , since the electron density n is a function of first-order reduced density matrix ρ . In addition to this, the kinetic energy T_s of noninteracting system can be expressed by ρ and a matrix representation H_{kin} of the kinetic operator as $\text{tr}(\rho H_{\text{kin}})$. Therefore, we see $E[\psi, n] \rightarrow E[\rho]$. Though the discussion above is nothing but mathematically a transformation of variables, the change of variable describing the total energy has a significant impact physically. Recall that the Wannier functions w and the first-order reduced density matrix ρ are localized in real space [9, 10], while the KS orbitals ψ are delocalized. The localized properties of w and ρ are called nearsightedness of electron. One may be able to largely reduce the computational cost by explicitly taking account of the locality and introducing the cutoff scheme that only the matrix elements within a given cutoff radius are included in the calculation. This is a basic idea behind many $O(N)$ methods developed so far.

According to a chemical intuition that many chemists have commonly, the electronic structure, such as charge state, of a certain atom in a system is mainly determined by atomic arrangement and a kind of elements in the vicinity of the atom. Remembering that chemical properties of a functional group in molecules are pre-determined group by group, such a chemical intuition might be justified. Although it is not apparent up to here how the localized properties of w and ρ can be related to such a chemical intuition, a mathematical justification of the chemical intuition will be provided and the physical meaning of nearsightedness will become more clear when ρ is expressed using a Green function in the later discussion.

4.3 Localized Orbital Method

Compared to post Hartree-Fock methods based on many body wave functions, the mathematical structure of the KS equation is rather simple, and the one body equation only has to be solved iteratively. The feature of the KS equation enables us to address complicated systems such as surfaces, interfaces consisting of difference materials, and liquid. Nevertheless, it is not still an easy task to treat large-scale systems including more than a few thousand atoms. In general the computational complexity scales as the third power of the number of atoms, requiring 1000 times computational cost for a 10 times large system. In this section, we introduce numerical localized basis function which will play a crucial role for realization of $O(N)$ methods.

Let us first estimate the computational complexity to solve the KS equation. In Eqs. (4.8)–(4.10) we have two differential equations. One is the KS equation given by Eq. (4.8), and the other is the Poisson equation to calculate the Hartree potential. The computational complexity in solving the Poisson equation using fast Fourier transform (FFT) is $O(N \log(N))$, and the computational cost is relatively small. Thus,

we do not discuss it anymore. An expansion method using basis functions is generally employed for solution of the KS equation. It might be possible to choose either delocalized plane wave basis functions or localized basis functions. Considering the compatibility to $O(N)$ methods and large-scale parallelization on massive parallel computers, the localized basis functions will be advantageous, since the calculation can be performed locally in real space. In addition, the existence of Wannier functions infers that accurate localized basis functions can be constructed. With these considerations, we expand the KS orbitals ψ using atomic basis functions or finite element basis functions as in Eq. (4.14). Inserting Eq. (4.14) into Eq. (4.8), and rewriting it in a form of matrix equation, we have the following generalized eigenvalue problem:

$$Hc_\mu = \varepsilon_\mu S c_\mu, \quad (4.17)$$

where \mathbf{k} is dropped for simplicity of notation. c_μ is a vector consisting of linear combination coefficients. The matrix elements for H and S are given by

$$H_{i\alpha j\beta} = \langle \chi_{i\alpha} | \hat{h}_{\text{KS}} | \chi_{j\beta} \rangle, \quad (4.18)$$

$$S_{i\alpha j\beta} = \langle \chi_{i\alpha} | \chi_{j\beta} \rangle. \quad (4.19)$$

If the tail of localized orbitals becomes completely zero beyond a cutoff radius, the overlap integral $S_{i\alpha j\beta}$ is nonzero within a finite range, and thereby the number of nonzero elements in the overlap matrix is exactly proportional to the number of atoms, leading to a sparse matrix with $O(N)$ nonzero elements. As well, the KS matrix H in the case of semi-local functionals such as LDA and GGA has also the same sparse structure as in the overlap matrix S . In the case of hybrid functionals beyond the semi-local treatment, the sparse structure of the KS matrix H differs from that of the overlap matrix, and the number of nonzero elements largely increases.² Since we restrict our discussion within the semi-local functionals, the computational complexity in evaluating the matrices H and S is $O(N)$. The computational complexity to solve the generalized eigenvalue problem of Eq. (4.17) is $O(N^3)$, since the number of basis functions is proportional to that of atoms, and matrix multiplications constitute the basic operation in the computation. Considering that the electron density n is calculated by Eq. (4.15), and that terms in the summation only have to be included if the product of two basis functions $\chi_{i\alpha}\chi_{j\beta}$ is nonzero, the computational complexity is found to be $O(N)$. In short, the computational complexity in solving the KS equation is governed by the eigenvalue problem, and thereby it is $O(N^3)$ as a whole. Figure 4.1 summarizes the computational flow of SCF calculation and computational complexity of each step.

²Even if the overlap between the associated localized orbitals is zero, the two-electron exchange integral is not zero. Thus, strictly speaking all the elements are nonzero for hybrid functionals. Even in the case of hybrid functionals with a screened Coulomb interaction, the number of nonzero elements largely increases compared to semi-local functionals, while the resultant matrix will be sparse due to the limited interaction.

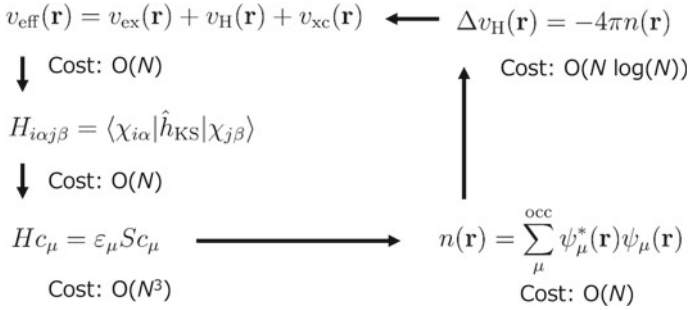


Fig. 4.1 Computational flow of SCF calculation and computational complexity of each step

The choice of basis functions is a crucial step in reducing the computational cost, since reducing the matrix size in the eigenvalue problem will directly lead to reduction of the computational cost. What kind of localized basis functions should be employed while reducing the matrix size and keeping accuracy of the calculation? In addition to Gaussian functions widely used in the field of quantum chemistry, there are a wide variety of choices for localized functions. Among these choices, numerical pseudo-atomic orbital (PAO) functions [11] are atomic like functions for pseudopotentials, which has a large degree of freedom due to its numerical form. Though finite element methods and finite difference methods are also regarded as localized basis method, the matrix size will be relatively large compared to that in the PAO functions, while the computational accuracy can be systematically controlled. Here let us introduce the PAO functions [11] which reduces the number of basis functions by making full use of the degree of freedom in the numerical form, while keeping the accuracy. The PAO function χ is defined by

$$\chi_{\alpha}(\mathbf{r}) = Y_l^m(\hat{\mathbf{r}})R_{lp}(r), \quad (4.20)$$

where Y and R are a solid spherical harmonic function and radial function, respectively, and α is a composite index which denotes a set of three indices (plm). The radial function R is generated by the following two steps: (1) primitive radial functions $R^{(p)}$ are calculated from an atomic problem under a confinement potential. (2) the radial function R is expressed by a linear combination of primitive functions $R^{(p)}$, and the contraction coefficients are optimized so that the total energy can be minimized. In the first step, an atomic DFT calculation is performed with a confinement potential, for wave functions to localize within a radius r_c , defined by

$$V_{\text{core}}(r) = \begin{cases} -\frac{Z}{r} & r \leq r_1, \\ \sum_{n=0}^3 b_n r^n & r_1 < r \leq r_c, \\ h & r_c < r, \end{cases} \quad (4.21)$$

where $b_0, b_1, b_2,$ and b_3 are determined so that the potential are continuous up to its first derivative at both r_1 and r_c . After getting the SCF solution of the atomic problem, a pseudopotential is generated with the confinement potential for each L -channel. Then, radial functions of the ground and excited states for the pseudopotential with the confinement potential are numerically calculated, which are the primitive radial functions $R^{(p)}$. If a large h is used for the calculation, the radial functions are fully localized within the confinement radius r_c in double precision even for the excited states. In Fig. 4.2a, a pseudopotential with the confinement potential and corresponding radial functions are shown for the case of $l = 0$ of a carbon atom. The radial function having no node corresponds to the ground state, and the others are related to the first, second, and higher excited states as the number of nodes increases. The primitive functions may not be optimal for molecular and solid state systems, since the functions are obtained from the atomic calculation. Thus, we expand the radial functions R using the primitive ones $R^{(p)}$, and optimize variationally contraction coefficients a so that R can be optimized for molecular and solid state systems.

$$R(r) = \sum_q a_q R_q^{(p)}(r). \quad (4.22)$$

The gradients $\partial E/\partial a$ of the total energy with respect to a can be analytically evaluated, and thereby the contraction coefficients a are optimized by the same procedure as for geometry optimization [11]. In Fig. 4.2b, we show the total energy of carbon in the diamond structure as a function of the number of basis functions. It is found that the total energy converges at a smaller number of optimized functions.

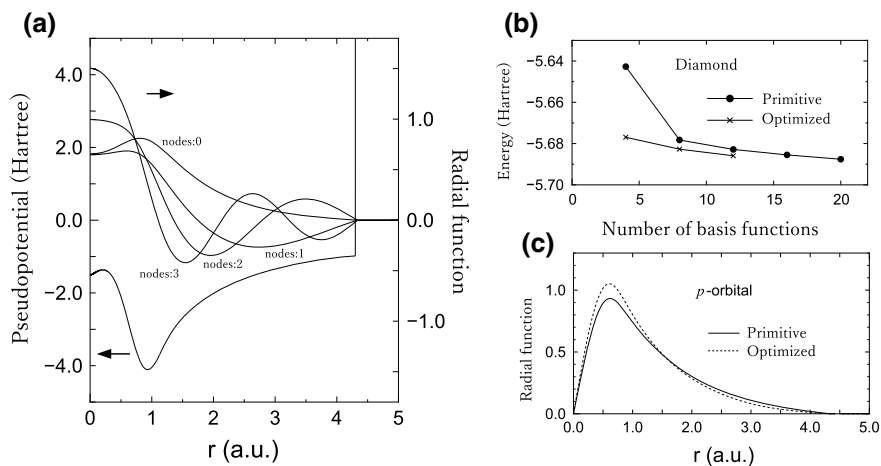


Fig. 4.2 **a** A pseudopotential with a confinement potential and corresponding radial functions for the case of $l = 0$ of a carbon atom. **b** Total energy in the carbon diamond as a function of basis functions. **c** Radial functions of p -orbitals which corresponds to the calculations of **(b)**. Reprinted figure with permission from [11]. Copyright (2017) by the American Physical Society

Figure 4.2c shows primitive and optimized radial functions of p -orbitals. The optimization method allows us to pre-optimize basis functions by properly choosing a set of reference systems, while it is also possible to optimize the contraction coefficients a for every material. After pre-optimizing basis functions, a single optimized basis set is constructed via Gram-Schmidt orthogonalization for every element. The advantage in the numerical form of basis functions can be seen in Eq. (4.22). Once the contraction coefficients a are optimized, the summation of Eq. (4.22) can be performed numerically, leading to a single numerical table. Therefore, the evaluation of matrix elements, which are calculated by the Fourier transform and a numerical quadrature method using the Fourier mesh [12], becomes much easier than the case of analytic functions which require all the evaluations of matrix elements associated with primitive functions. As the reference systems for the pre-optimization, a single atom, a dimer, and a bulk structure such as face centered cubic can be selected, which are expected to reflect a variety of chemical environments. In general the optimized orbital R tends to localize in real space as the coordination number increases, i.e., it delocalizes for a single atom, and localizes for bulks. The property of optimized functions can be understood by the quantum mechanical virial theorem. According to the virial theorem, we have a relation $\langle T \rangle = -1/2\langle V \rangle$ at an equilibrium structure, which tells us that the kinetic energy $\langle T \rangle$ becomes positively large as the potential energy $\langle V \rangle$ becomes negatively large. As the coordination number increases, the superposition of the nuclei potential leads to a deeper external potential, and as a result the kinetic energy $\langle T \rangle$ increases. As can be seen in Fig. 4.2c, the increase of kinetic energy is nothing but the localization of wave function. The optimization of radial functions reflects partly the consequence of the virial theorem. Another factor in determining the optimized shape of radial function is the charge state of atom in molecules and bulks. The optimization leads to localized and delocalized functions for a positively and negatively charged atom, respectively. The optimized basis set generated by the procedure experiences three different chemical environments at least, and it is confirmed from a series of benchmark calculations that they provide nearly convergent results for a wide variety of systems.

Here let us introduce a study concerning accuracy of the localized orbital method. In recent years many DFT codes have been developed and released for public use. Because of the difference of details in the implementation, computation results may vary depending on DFT codes even if the same exchange-correlation functional is employed. A method, which is called Δ gauge method, has been developed [13] to validate the accuracy of the implementation, and compare the accuracy among DFT codes, where the Δ gauge is defined by

$$\Delta(A, B) = \sqrt{\frac{\int_{0.94V_0}^{1.06V_0} (E_B(V) - E_A(V))^2 dV}{0.12V_0}}. \quad (4.23)$$

The total energy of an elemental bulk is calculated using two different DFT codes A and B by varying the volume within $\pm 6\%$ with respect to an equilibrium volume V_0 , and the difference between the two codes in the volume-energy curve (E_A and E_B) is

evaluated by the integration after setting the lowest energy in each case to the energy origin. The Δ gauge of GGA by Perdew et al. [7] is found to be 23.5 meV/atom on average for 58 elemental bulks by comparison between highly accurate all electron calculations and experimental data. Therefore, it is desirable that the numerical error arising from basis functions, pseudopotentials, and numerical integration is less than 10% of that arising from an exchange-correlation functional. The Δ gauge of the localized orbital method discussed in the section is 2.0 meV/atom on average for 71 elemental bulks compared to reference data calculated by a full potential linearized augmented plane wave method, which is considered to be enough accuracy for most of the purposes.

4.4 $O(N)$ Methods

4.4.1 Representation of Total Energy and Approximations

In Sect. 4.3 we argued that the computational complexity of the DFT calculations is governed by the eigenvalue problem, and scales as $O(N^3)$ for the number of atoms N . In this section, we discuss how the computational complexity can be reduced from $O(N^3)$ to $O(N)$ by introducing approximations based on the nearsightedness of electron. Remember that the Wannier functions w and the first-order reduced density matrix ρ are localized in real space as discussed in the Sect. 4.2. Therefore, let us consider to perform a calculation of the total energy using either $E[w, n]$ or $E[\rho]$ instead of $E[\psi, n]$. An $O(N)$ method based on $E[w, n]$ has been known as orbital minimization method [14, 15] in which the total energy E is minimized with respect to the Wannier functions w under a constraint that the localization range of non-orthogonal generalized Wannier functions is imposed. By ignoring the contribution beyond the localization range, the computational complexity becomes $O(N)$. Several $O(N)$ methods have been proposed based on $E[\rho]$, and among them the density matrix method has been a widely used approach [16, 17]. In the method, the total energy is minimized with respect to ρ under a constraint that the first-order reduced density matrix ρ has idempotency. Once a termination of ρ in real space is introduced, and the contribution beyond the termination range is ignored, the computational complexity becomes $O(N)$. The generalization to non-orthogonal basis functions such as finite element methods and the localized orbital method discussed in the Sect. 4.3 needs to be considered in order to take account of the overlap matrix S . Such generalizations have been already proposed for both the orbital minimization and density matrix methods, and implemented in DFT codes such as FEMTECK [18], CONQUEST [19], and ONETEP [20] with a wide variety of applications. An $O(N)$ method using Green functions is also based on $E[\rho]$ [21]. It has been known that the $O(N)$ method can be applied to not only gapped systems, but also metals. In the section, we discuss the $O(N)$ method based on the Green function.

4.4.2 Recursion Method for Orthogonal Basis Set

We start to consider a simple tight binding model in which each atom has only a single s -orbital, and the overlap matrix is set to $S = I$. Thus, the eigenvalue problem of Eq. (4.17) is simplified as

$$Hc_\mu = \varepsilon_\mu c_\mu. \quad (4.24)$$

The Green function for Eq. (4.24) is defined by

$$\begin{aligned} G(Z) &\equiv (ZI - H)^{-1}, \\ &= \sum_\mu \frac{|\psi_\mu\rangle\langle\psi_\mu|}{Z - \varepsilon_\mu}, \end{aligned} \quad (4.25)$$

where Z is a complex variable. Taking $Z = E + i0^+$, where E is a real number and 0^+ is a positive infinitesimal, for $1/(Z - \varepsilon_\mu)$ in Eq. (4.25), and considering the imaginary part, we obtain a Lorentzian function centered at ε_μ . Therefore, the reduced density matrix ρ can be calculated using the Green function as

$$\rho = -\frac{2}{\pi} \int_{-\infty}^{E_F} \text{Im}G(E + i0^+)dE, \quad (4.26)$$

where E_F is the Fermi energy, and the factor of 2 is due to spin multiplicity. The recursion method developed by Haydock et al. is an earliest $O(N)$ method to calculate the Green function [22], and it is possible to directly calculate ρ using Eq. (4.26) without diagonalizing the matrix.

In the recursion method the matrix H is tri-diagonalized by a Lanczos method, and the Green function is evaluated using a continued fraction formula. The Lanczos algorithm is derived as follows. First, let us assume that an arbitrary Hermitian matrix H can be converted by a unitary transformation into a tri-diagonalized matrix $H_{\text{TD}} = U^\dagger H U$. Then, by inversely solving $U H_{\text{TD}} = H U$, the following Lanczos algorithm is derived:

$$\alpha_n = \langle u_n | H | u_n \rangle, \quad (4.27)$$

$$|r_n\rangle = H|u_n\rangle - |u_{n-1}\rangle\beta_n - |u_n\rangle\alpha_n, \quad (4.28)$$

$$\beta_{n+1} = \langle r_n | r_n \rangle^{1/2}, \quad (4.29)$$

$$|u_{n+1}\rangle = |r_n\rangle / \beta_{n+1}. \quad (4.30)$$

Starting from an initial vector $|u_0\rangle$ at $n = 0$, and proceeding the calculation repeatedly toward the increase of n , the Hermitian matrix H is transformed to the tri-diagonalized matrix H_{TD} :

Let us illustrate the recursion method with a simple analytic model where the hopping integral is nonzero only for the nearest interaction. The Hamiltonian is given by

$$H_{ij} = \begin{cases} \epsilon & i = j, \\ t & i \text{ and } j \text{ are the nearest,} \\ 0 & \text{else.} \end{cases} \quad (4.34)$$

Starting from an arbitrary atomic site $\langle u_0 | = (\dots, 0, 0, 0, 1, 0, 0, 0, \dots)$, we have the following results:

$$\alpha_0 = \epsilon, \quad (4.35)$$

$$\langle r_0 | = (\dots, 0, 0, t, 0, t, 0, 0, \dots), \quad (4.36)$$

$$\beta_1 = \sqrt{2}t, \quad (4.37)$$

$$\langle u_1 | = \frac{1}{\sqrt{2}}(\dots, 0, 0, 1, 0, 1, 0, 0, \dots), \quad (4.38)$$

$$\alpha_1 = \epsilon, \quad (4.39)$$

$$\langle r_1 | = (\dots, 0, t, 0, 0, 0, t, 0, \dots), \quad (4.40)$$

$$\beta_2 = t, \quad (4.41)$$

$$\langle u_2 | = \frac{1}{\sqrt{2}}(\dots, 0, 1, 0, 0, 0, 1, 0, \dots). \quad (4.42)$$

We see that the calculation after Eq.(4.42) is a duplication, and that the nonzero elements in the vector u hop to outer sites from the starting site step by step as shown in Fig.4.3. Noting that $\beta_n(2 \leq n) = t$, and the same nested structure can be seen in the continued fraction for $2 \leq n$, we obtain from Eq.(4.33) the following formula:

$$G_{00}^{(\text{TD})}(Z) = \frac{1}{\sqrt{(Z - \epsilon)^2 - 4t^2}}. \quad (4.43)$$

We see the Van Hove singularity at $E = \epsilon \pm 2t$ which is a characteristic feature of a one-dimensional system.

The analysis for the chain model draws an important picture that Lanczos vectors $\{u\}$ develops toward the outer region starting from the central site. Recalling that the

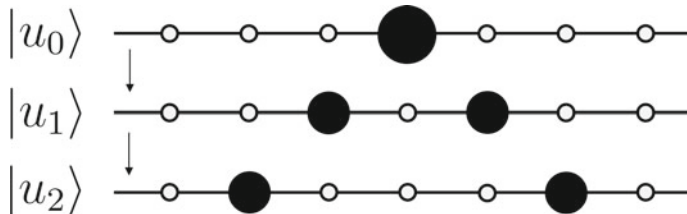


Fig. 4.3 Development of Lanczos vectors on an infinite chain

Lanczos process is terminated at a finite level even for an infinite system, we see that the information are included from the neighborhood to the distance regions step by step. The fact implies a relation to the nearsightedness of electron discussed in Sect. 4.2. So, we shall see the relation in more details. Noting that the central step in the Lanczos algorithm is the matrix-vector multiplication $H|u_n\rangle$ in Eq. (4.28), it turns out that a set of the Lanczos vectors $\{u\}$ spans the following Krylov subspace:

$$U_K = \{|u_0\rangle, H|u_0\rangle, H^2|u_0\rangle, H^3|u_0\rangle, \dots, H^q|u_0\rangle\}. \quad (4.44)$$

It is noted that calculating the Green function in the subspace U_K is equivalent to diagonalizing the matrix in the same subspace. Representing the Hamiltonian with vectors spanning the subspace U_K , we have the following equation:

$$\begin{aligned} H_{mn}^{(K)} &= \langle u_m | H | u_n \rangle, \\ &= \langle u_0 | H^{m+n+1} | u_0 \rangle, \\ &= \langle u_0 | \mu^{(m+n+1)} | u_0 \rangle, \end{aligned} \quad (4.45)$$

where $\mu^{(p)}$ is called the p -th moment, and defined by

$$\mu^{(p)} = H^p = c\varepsilon^p c^\dagger, \quad (4.46)$$

where c is a matrix whose columns consist of eigenvectors c_μ for the eigenvalue problem of Eq. (4.24), and ε is a diagonal matrix whose diagonal elements consist of the eigenvalues ε_μ . Thus, the effective Hamiltonian $H^{(K)}$ includes the first to $(2q + 1)$ -th moments when the Lanczos algorithm is iterated up to the q -th step. If we diagonalize $H^{(K)}$ and construct an approximate Green function, it is concluded that the Green function also consists of the first to $(2q + 1)$ -th moments. We shall obtain the same conclusion by rewriting Eq. (4.25). Expanding as $\frac{1}{(Z-E)} = \sum_p^\infty \frac{E^p}{Z^{p+1}}$ under a condition of $|E/Z| < 1$, and inserting it into Eq. (4.25), we obtain the following moment representation of Green function:

$$G(Z) = \sum_{p=0}^{\infty} \frac{\mu^{(p)}}{Z^{p+1}}. \quad (4.47)$$

Comparing Eq. (4.45) to Eq. (4.47), we see a one-to-one correspondence between the step in the Lanczos algorithm and the accuracy of Green function. By considering that the second and third order moments H^2 and H^3 correspond to two- and three-step hoppings, respectively, we see that the local Green function is constructed from the neighborhood to the distance regions step by step. Thus, we now understand that the nearsightedness of electron discussed in Sect. 4.2 is described by the Green function. It is also worth noting that the idea of nearsightedness can be utilized in developing model potentials [26]. Embedded atom method (EAM) potentials and Tersoff potential can derived from the second order moment.

4.4.3 Generalization to First-Principles Calculations

We next generalize the recursion method for first-principles calculations. The generalization requires us to take account of the overlap matrix S and multiple basis functions allocated to each atom. The Green function to be evaluated is modified as $G(Z) = (ZS - H)^{-1}$. A way to evaluate the Green function is to modify $G(Z)(ZS - H) = I$ as follows:

$$G(Z)S(Z - S^{-1}H) = G^{(L)}(Z)(ZI - H') = I, \quad (4.48)$$

where $G^{(L)}(Z) = G(Z)S$ and $H' = S^{-1}H$. The resultant formula looks similar to the orthogonal case. However, H' is not a Hermitian matrix H anymore, and thereby H' is transformed to a tri-diagonal matrix using a two-sided Lanczos algorithm [27]. Although the method is theoretically interesting, we do not discuss the details further, since the two-sided Lanczos algorithm tends to be seriously influenced by numerical round-off error. Therefore, let us consider an alternative method without using the tri-diagonalization of H or H' [21]. Remembering that the essential ingredient in the formalism for the orthogonal case is to generate a proper set of vectors spanning a Krylov subspace, we generate a set of vectors even for the non-orthogonal case. Note that the form of the p -th moment is modified by taking account of the overlap matrix S as

$$\begin{aligned} \mu^{(p)} &= c\varepsilon^p c^\dagger, \\ &= cc^\dagger Hcc^\dagger Hc \cdots c^\dagger Hcc^\dagger, \\ &= (S^{-1}H)^p S^{-1}, \end{aligned} \quad (4.49)$$

where the third line is obtained by using $cc^\dagger = S^{-1}$. On the other hand, the form of the moment representation for the Green function is the same as Eq.(4.47) for the orthogonal case. Therefore, for the non-orthogonal case a set of vectors spanning the Krylov subspace can be generated by multiplying $S^{-1}H$. A practical algorithm is summarized as follows:

$$\text{set } |W_0\rangle, \quad (4.50)$$

$$|R_{n+1}\rangle = S^{-1}H|W_n\rangle, \quad (4.51)$$

$$|W'_{n+1}\rangle = |R_{n+1}\rangle - \sum_{m=0}^n |W_m\rangle(W_m|S|R_{n+1}), \quad (4.52)$$

$$(\underline{B}_{n+1})^2 = (W'_{n+1}|S|W'_{n+1}), \quad (4.53)$$

$$(\underline{\lambda}_{n+1})^2 = T_{n+1}^\dagger(\underline{B}_{n+1})^2 T_{n+1}, \quad (4.54)$$

$$(\underline{B}_{n+1})^{-1} = T_{n+1}(\underline{\lambda}_{n+1})^{-1}, \quad (4.55)$$

$$|W_{n+1}\rangle = |W'_{n+1}\rangle(\underline{B}_{n+1})^{-1}. \quad (4.56)$$

It should be noted that a block algorithm is introduced to take account of multiple basis functions allocated to each atom. In the block algorithm, the initial state $|W_0\rangle$ consists of a set of vectors, e.g., all basis functions on an atom i we focus are chosen as $|W_0\rangle = (|i1\rangle, |i2\rangle, \dots, |iM_i\rangle)$. Also, the underline of \underline{B} means that the element is a matrix. A central operation in the algorithm is a multiplication of matrices by Eq. (4.51) to expand the Krylov subspace. The S -orthogonalization of a set of generated vectors to the other vectors is performed by Eq. (4.52), while Eqs. (4.53)–(4.56) correspond to the S -orthonormalization of a set of vectors generated by Eq. (4.52). Once a set of vectors spanning the Krylov subspace $\{|W_0\rangle, |W_1\rangle, |W_2\rangle, \dots, |W_q\rangle\}$ is generated, the Green function is constructed for the atom i after the KS Hamiltonian represented by the vectors is diagonalized as follows:

$$G_{i\alpha j\beta}(Z) = \sum_{\mu} \frac{|\psi_{\mu}^{(i)}\rangle \langle \psi_{\mu}^{(i)}|}{Z - \varepsilon_{\mu}^{(i)}}. \quad (4.57)$$

As well as the orthogonal case, the computational cost for each atom is found to be $O(1)$ by introducing the truncation for S and H , and the total computational cost becomes $O(N)$ by considering the contributions of all the atoms in total. Figure 4.4a shows the computational flow of the $O(N)$ Krylov subspace method. For each atom, (i) construction of truncated H and S , (ii) generation of the Krylov subspace, and (iii) diagonalization of the effective Hamiltonian are performed. After the calculations (i)–(iii) for all the atoms are completed, a common chemical potential is determined to conserve the total number of electrons in the whole system, and the calculation of density matrix follows by Eq. (4.26). In Fig. 4.4b, the convergence of error in

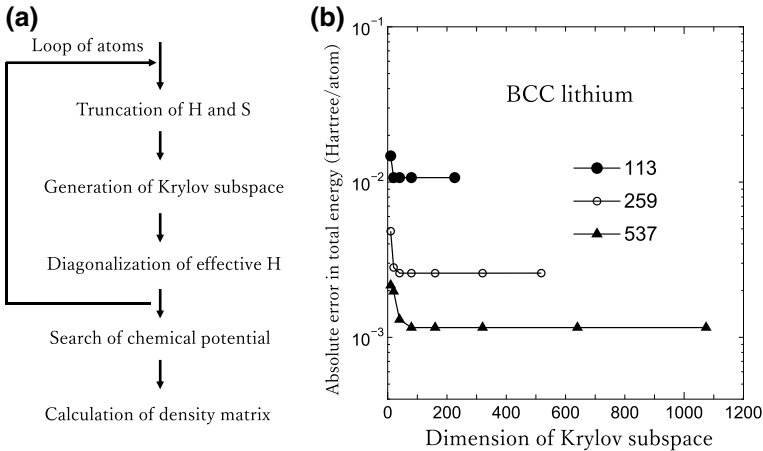


Fig. 4.4 **a** Schematic computational flow of the $O(N)$ Krylov subspace method. **b** Absolute error in the total energy of lithium in the body centered cubic structure calculated by the $O(N)$ Krylov subspace method, where the reference energy is calculated by the conventional diagonalization, and the number 113, 259, and 537 are the number of atoms in the truncated cluster. Reprinted figure with permission from [21]. Copyright (2017) by the American Physical Society

the total energy is shown for lithium in the body-centered cubic structure. We see that the computational accuracy is controlled by the size of truncated cluster and the dimension of Krylov subspace. It should be noted that the dimension of Krylov subspace to achieve sufficient convergence for each truncated cluster is much smaller than the number of basis functions in the truncated cluster. The feature of the $O(N)$ Krylov subspace method allows us to reduce the computational cost, while keeping the accuracy.

4.5 A Method of Massive Parallelization

As well as the development of efficient computational methods, it is important to make full use of parallel computers to realize more realistic simulations including more than a few thousand atoms. The calculation in the $O(N)$ Krylov subspace method discussed in the previous section can be nearly independently performed for each atom, which makes the method suitable for the parallel calculation intrinsically. However, it is not obvious how the parallel efficiency can be improved for an arbitrary system with a complicated 1D, 2D, or 3D structure. In this section, we address a general method to partition a system to a set of small fragments for improvement of efficiency in massive parallelization. The method can be applied to not only the $O(N)$ Krylov subspace method, but also other simulations.

The purpose of parallel calculations is to reduce the computational elapsed time as much as possible by dividing the calculations, and assigning them to computer processes. In order to improve the parallel efficiency, it is crucial for data in each process to be localized and for communication among processes to be minimized. The communication itself takes a considerable amount of time, and the delay of calculations is also induced by the communication. The load balancing of computational amount is also important. Once the computational elapsed times becomes non-uniform over all the processes, most of processes wait for a process which is most loaded, resulting in the reduction of parallel efficiency.

In the parallelization of the $O(N)$ Krylov subspace method, we need to consider how a system and integration meshes can be partitioned to a set of small fragments so that these requirements can be fulfilled as much as possible. Since the partitioning of integration meshes largely depends on details of implementation, here we focus on the partitioning of atoms in a system, and discuss a method based on a modified recursive bisection method and an inertia moment tensor [28]. The method discussed in the section may be applied to a wide variety of simulations other than the $O(N)$ method. The method consists of (i) a step that a given number of computational processes is mapped to a binary tree using a modified recursive bisection method, (ii) a step that atoms in a domain of atoms we focus are projected onto a principal axis which is calculated by diagonalizing an inertia moment tensor, (iii) a step that the atoms are partitioned on the principal axis using the binary tree constructed in the step (i). As shown in Fig. 4.5, it is possible to partition atoms in the system to a set of small fragments by performing the steps (ii) and (iii) recursively using the

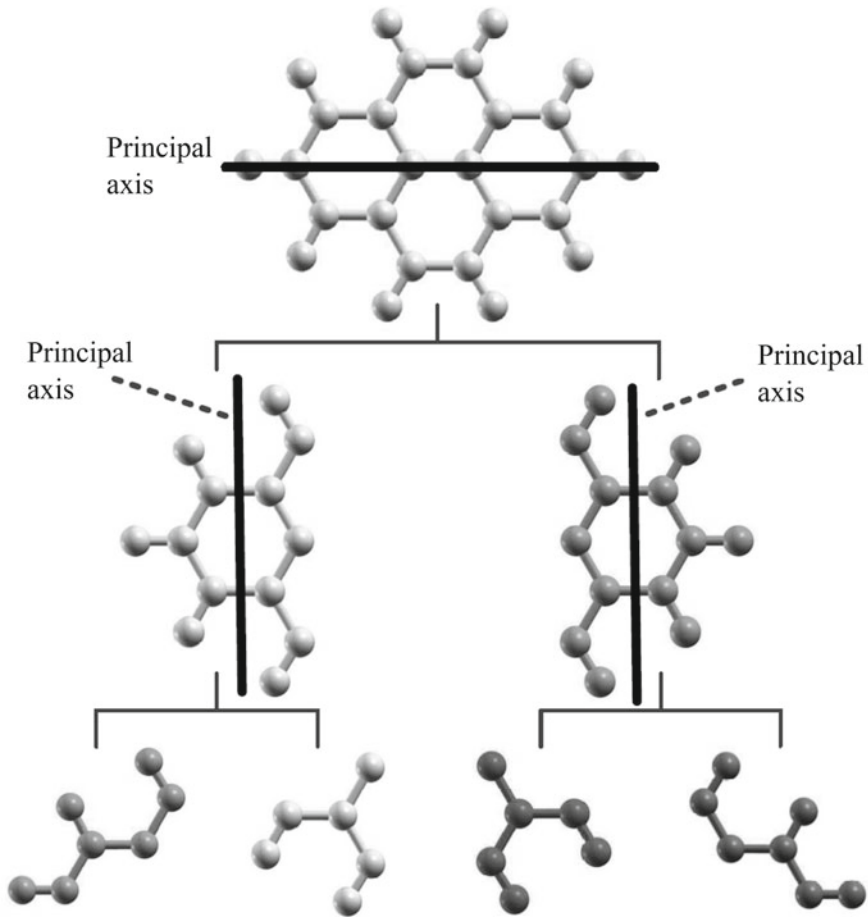


Fig. 4.5 Partitioning of a system to a set of small fragments. Reprinted figure with permission from [28]. Copyright (2017) by the Elsevier

binary tree constructed in the step (i). It is worth noting that the resultant partitioning fulfills the requirements of the locality of information and the load balancing, and the computational effort for the partitioning is negligible compared to the other parts. Let us explain details of each step below.

(i) Partitioning of processes by a binary tree

Suppose that the number of computational processes is 19, then we can construct a binary tree as shown in Fig. 4.6 using the modified recursive bisection method. Note that a binary tree structure can be certainly constructed by a recursive procedure regardless of an even, odd, or prime number. Considering the usability of code, it would be better for a code to be able to deal with any number of processes. This is the step (i).

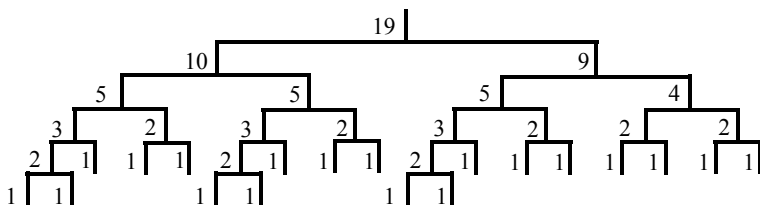


Fig. 4.6 Binary tree structure of computational processes by a modified recursive bisection method

(ii) Determination of a principal axis

A principal axis for a domain which includes N_a atoms is given by

$$\mathbf{r} = \mathbf{C} + \mathbf{a}t, \quad (4.58)$$

where $\mathbf{r} = (x, y, z)$, $\mathbf{a} = (a_x, a_y, a_z)$, $\mathbf{C} = (C_x, C_y, C_z)$. The center of inertia \mathbf{C} is calculated using atomic coordinates \mathbf{r}_i in the domain we focus as follows:

$$\mathbf{C} = \frac{1}{N_a} \sum_{i=1}^{N_a} \mathbf{r}_i w_i. \quad (4.59)$$

Note that the weight w is not the atomic mass, and will be discussed later on. Because of the fact that atoms distribute three-dimensionally in general, how to partition the atoms may not be straightforward. Thus, we consider a projection of atoms onto the principal axis, and transform the three-dimensional distribution of atoms to a one-dimensional one. An equation for the projection is given by

$$t_i = a_x(x_i - C_x) + a_y(y_i - C_y) + a_z(z_i - C_z). \quad (4.60)$$

How should we determine \mathbf{a} in Eq. (4.60)? A proper \mathbf{a} can be obtained by maximizing the following function F with the Lagrange multiplier method.

$$F = \sum_i w_i t_i^2 - \lambda(|\mathbf{a}|^2 - 1). \quad (4.61)$$

Considering $\partial F / \partial \mathbf{a} = 0$, we have the following eigenvalue problem:

$$T\mathbf{a} = \lambda\mathbf{a} \quad (4.62)$$

$$T = \begin{pmatrix} \sum_i w_i (Y_i^2 + Z_i^2) & -\sum_i w_i X_i Y_i & -\sum_i w_i X_i Z_i \\ -\sum_i w_i Y_i X_i & \sum_i w_i (X_i^2 + Z_i^2) & -\sum_i w_i Y_i Z_i \\ -\sum_i w_i Z_i X_i & -\sum_i w_i Z_i Y_i & \sum_i w_i (X_i^2 + Y_i^2) \end{pmatrix}, \quad (4.63)$$

where $(X_i, Y_i, Z_i) = (x_i - C_x, y_i - C_y, z_i - C_z)$. If we choose the atomic mass for the weight w , T is nothing but the inertia tensor of the domain. By diagonalizing

Eq. (4.63), and choosing an eigenstate which maximizes the function F , we have the sparsest distribution of atoms on the axis.

(iii) Partitioning of atoms

The one-dimensionally distributed atoms on the axis are partitioned using the binary tree structure constructed by the step (i) so that the following condition:

$$\sum_i^{\text{left side}} w_i : \sum_i^{\text{right side}} w_i = \# \text{ in the left side} : \# \text{ in the right side} \quad (4.64)$$

can be fulfilled as much as possible, where $\#$ means the number of processes. In the case of Fig. 4.6, the first partitioning is performed with the ratio of 10 : 9 by referring the top stage of the binary tree. Summing up weights w starting from both the edges, and finding a point which is closest to the ratio of 10 : 9, we can partition the domain into two sub-domains. Furthermore, the same procedure is applied to each of the two sub-domains. We repeat the procedure recursively until we reach at the lowest stage of the binary tree. Here we return to the issue how the weight w should be chosen. In order to improve the load balance, we should choose the computational elapsed time for each atom. The computational elapsed time of each atom can be monitored in geometry optimization and molecular dynamics (MD) simulations at every MD step. Thus, the load balance might be improved by employing the elapsed time obtained at the previous MD step as the weight w . Also, note that atoms with small and large w tend to be located around the center of inertia \mathbf{C} and far from \mathbf{C} , respectively, because of the definition of Eq. (4.61). Since the atoms with small w gather around the point at which the partitioning is performed, the condition by Eq. (4.61) might be well satisfied. Figure 4.7 shows partitioning of diamond structure by 19 processes. In spite of the use of 19 processes, we see that the system is properly partitioned in such a way to keep the locality.

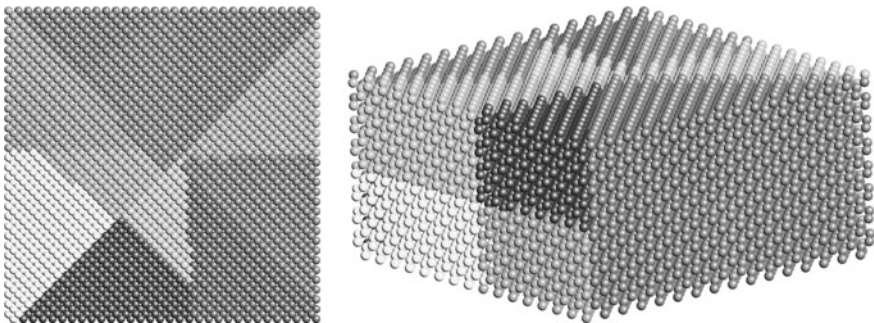


Fig. 4.7 Partitioning of diamond structure by 19 processes (Left: a top view, Right: bird-eye view). Atoms with the same color are allocated to a same process. Reprinted figure with permission from [28]. Copyright (2017) by the Elsevier

4.6 Applications of the $O(N)$ Method

$O(N)$ methods have been applied to a wide variety of materials such as interfaces, electronic devices, electrodes in lithium-ion batteries, and liquids [10]. Here we introduce an application related to a structural material using the $O(N)$ Krylov subspace method [29].

Figure 4.8 shows an optimized interface structure in a semi-coherent precipitation consisting of body-centered cubic (BCC) iron and NbC. The precipitation of carbides in steel is one of the important means to control the strength of steel. Experimentally it is known that the strength of steel depends on the diameter of precipitating carbide. A possible cause might be attributed to change of the interface structure between the BCC iron and carbide. The coherent and semi-coherent interface structures are found experimentally for a small and large diameter of precipitating carbide, respectively, and the interface structure is considered to be determined by a subtle balance between the interface energy and a strain energy induced in the BCC iron being the matrix. If the interface energy and strain energy can be accurately evaluated, it might be possible to estimate how the critical radius, at which the transition from the coherent to semi-coherent interface structure occurs, varies depending on a kind of carbides. However, a large supercell is required for calculations of the semi-coherent interface, e.g., a cell including 1463 atoms, consisting of 14 layers (7 iron layers and 7 NbC

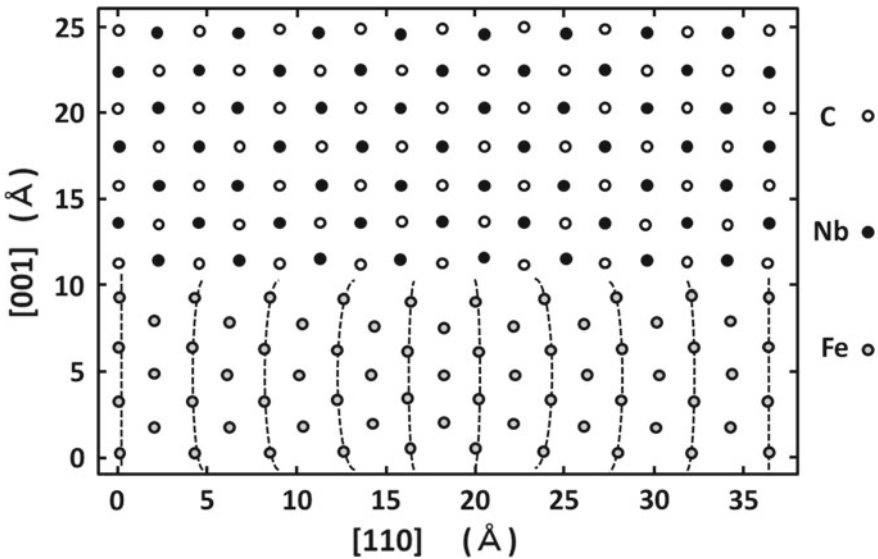


Fig. 4.8 Optimized interface structure in a semi-coherent precipitation consisting of body-centered cubic (BCC) iron and NbC calculated by the $O(N)$ Krylov subspace method. In the interface structure, the BCC iron (100) and NbC(100) in the NaCl structure forms semi-coherent interface structure in the Baker-Nutting relation: $[010]_{\text{NbC}}//[011]_{\text{Fe}}$, $[001]_{\text{NbC}}//[0\bar{1}1]_{\text{Fe}}$. Reprinted figure with permission from [29]. Copyright (2017) by the IOP Publishing

layers), has to be considered even for a carbide of NbC with a large lattice mismatch for the BCC iron. It is found from the optimized structure shown in Fig. 4.8 that iron atoms approach to carbon atoms by a strong interaction between carbon and iron atoms, resulting in large structural distortion in the BCC iron. The transition from the coherent to semi-coherent interface structure is estimated to 1.6 nm by a series of calculations [29], which is well compared to an experimental value. The example demonstrates that such a rather macroscopic problem, the strength of steel, has started to be addressed from an electronic structure level, suggesting that its future progress is highly expected.

4.7 Numerically Exact Low-Order Scaling Methods

In Sect. 4.3, it was argued that the computational cost of the KS equation is mainly governed by the eigenvalue problem, and the complexity is $O(N^3)$. Also, we discussed in Sect. 4.4 that the computational cost can be reduced to $O(N)$ if approximations are introduced based on the nearsightedness of electron. Then, it would be natural to consider whether a numerically exact low-order scaling method can be developed. In fact, such methods have been proposed [30, 31], suggesting a new possibility toward accurate large-scale DFT calculations. In this section, let us introduce a numerically exact low-order scaling method [31] based on a contour integration of Green function and a nested dissection method.

The method is composed of two ideas. The first idea is to directly evaluate necessary elements of the density matrix using the contour integration of Green function, and the second idea is to calculate only necessary elements of Green function using a set of recurrence formulae derived from a nested dissection method. The recurrence formulae are derived by recursively applying the LDL^t decomposition to a hierarchical structured matrix which is obtained by a nested dissection method. The computational complexity of the method is found to be $O(N(\log_2 N)^2)$, $O(N^2)$, and $O(N^{7/3})$ for 1D, 2D, and 3D systems, respectively, without introducing approximations. One of the characteristic features of the method is that not only gapped systems, but also metals can be treated by the method on the same footing.³ Each of the two ideas will be explained below.

(i) Contour integration of the Green function

Recalling that only elements of the density matrix ρ corresponding to nonzero overlap between basis functions contribute to the electron density n due to Eq. (4.15), we only have to calculate the necessary elements of ρ . Thus, the number of elements which need to be calculated is $O(N)$. The necessary elements of ρ is directly calculated by the Green function using Eq. (4.26). The integral of Eq. (4.26) might be efficiently evaluated using a contour integration with the following continued fraction representation of the Fermi function derived from a hypergeometric function [32]:

³The formulation does not rely on the band gap of a systems.

$$\frac{1}{1 + \exp(x)} = \frac{1}{2} - \frac{x}{4} \left(\frac{1}{1 + \frac{(\frac{x}{2})^2}{3 + \frac{(\frac{x}{2})^2}{5 + \frac{(\frac{x}{2})^2}{\dots}}}} \right) \quad (4.65)$$

Although the continued fraction of Eq. (4.65) is similar to the Matsubara summation in a sense that it has poles on the imaginary axis, the interval between adjacent poles becomes sparse as the distance between a pole and the real axis increases. Because of the special pole structure, the contour integration is significantly accelerated. In addition to this, it is proven that the number of poles required for the convergence is independent of the number of atoms N . Thus, the computational complexity is determined by the calculation of the Green function.

(ii) Nested dissection and LDL^t factorization

Any sparse matrix can be transformed to a hierarchical structured matrix using a nested dissection method. As an example of the case, we show in Fig. 4.9 how a Hamiltonian matrix for a finite chain of 10 atoms can be transformed to a hierarchical structured matrix. When the index is assigned from the left-hand side, the Hamiltonian matrix becomes a tri-diagonal form as shown in Fig. 4.9a. In a case that the index of 10 is assigned for a central atom, the Hamiltonian matrix is transformed in such a way that the 5 atoms in the left-hand side interact with the 4 atoms in the right-hand side via the central atom with the index of 10 as shown in Fig. 4.9b. By applying the procedure repeatedly, we obtain the last case as shown in Fig. 4.9c. We see that the interaction in the last case is mapped to the hierarchical binary tree as shown in Fig. 4.9d. Remembering that the Green function is the inverse of $(ZS - H)$, and that $(ZS - H)$ is a sparse matrix, we first transform $(ZS - H)$ to a hierarchical structured matrix by the nested dissection method. Then, we can obtain a set of recurrence formulae by applying repeatedly the LDL^t factorization from the bottom of the matrix in upward direction. Here we show a part of the recurrence formulae.

$$V_{p,m+1,n}^T = \begin{pmatrix} V_{p,m,2n}^T \\ V_{p,m,2n+1}^T \\ 0 \end{pmatrix} + \begin{pmatrix} L_{m,2n}^T \\ L_{m,2n+1}^T \\ -I \end{pmatrix} Q_{p,m+1,n}^T \quad (4.66)$$

Note that it is possible to calculate only the necessary elements of ρ using the recurrence formulae. The computational complexity might be reduced, since the set of recurrence formulae is derived by making use of the sparseness of the Hamiltonian matrix. In fact, the detailed analysis results in the computational complexity of $O(N(\log_2 N)^2)$, $O(N^2)$, and $O(N^{7/3})$ for 1D, 2D, and 3D systems, respectively [31]. The numerically exact low-order scaling method is a promising direction, and thereby the further investigation would be highly expected.

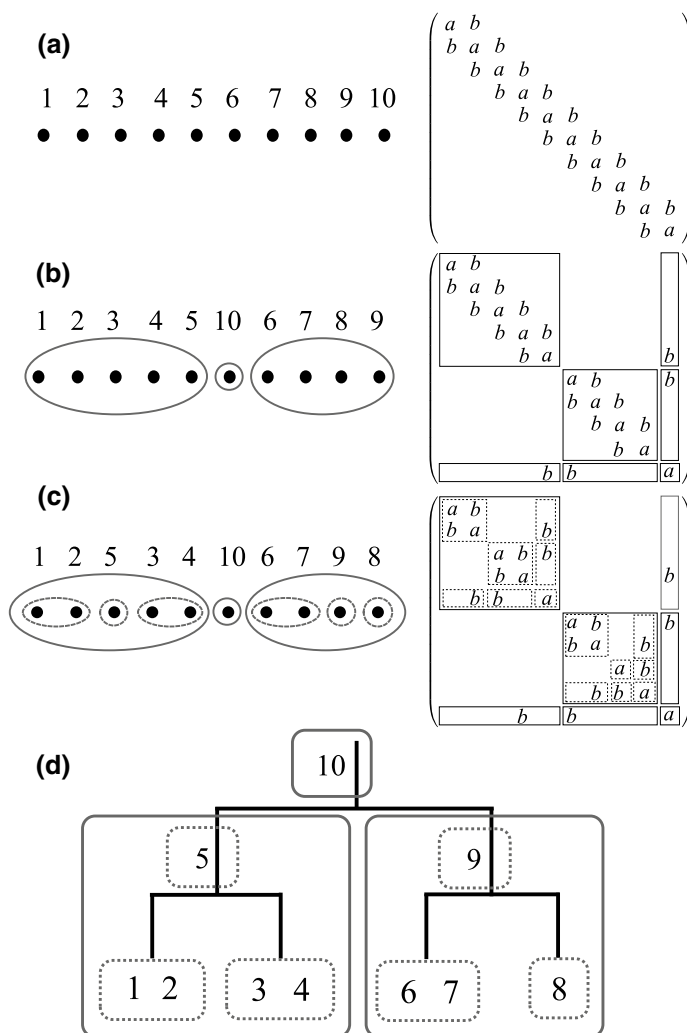


Fig. 4.9 Idea of the nested dissection. Reprinted figure with permission from [31]. Copyright (2017) by the American Physical Society

4.8 Remarks

The usefulness of DFT has been widely recognized, and applications for more challenging subjects have been ongoing. The $O(N)$ methods discussed in the chapter will extend the frontier of challenging DFT simulations. The underlying idea behind the development of $O(N)$ methods is the nearsightedness of electron. The idea plays an important role for not only the development of the eigenvalue problem, but also an

accurate exchange functional [33]. Though the developments of $O(N)$ methods have been active since the early of 1990s, there would be still a plenty of room for the improvement from the theoretical aspect. It would be nice if the chapter provides a hint toward further development.

4.9 Exercises

1. Derive the KS equation of (4.8)–(4.10).
2. Show that the solution of KS equation satisfies the variational principle $\delta E/\delta n$ for electron density.
3. Derive the expression (4.15) for electron density n . Also, show that the calculation of n needs only $O(N)$ elements of the density matrix ρ .
4. Derive the continued fraction formula (4.33) of the Green function.
5. Derive the Green function (4.43) of the infinite chain model.
6. Discuss the one-to-one correspondence between the step in the Lanczos algorithm and the accuracy of Green function using moments.
7. Explain the relationship between the local electronic structure of an atom and $O(N)$ methods.
8. Derive the eigenvalue equation of (4.63) for determining the principal axis.

References

1. E. Wigner, F. Seitz, Phys. Rev. **43**, 804 (1933)
2. J.C. Slater, Phys. Rev. **81**, 385 (1951)
3. P. Hohenberg, W. Kohn, Phys. Rev. **136**, B864 (1964)
4. W. Kohn, L.J. Sham, Phys. Rev. **140**, A1133 (1965)
5. D.M. Ceperley, B.J. Alder, Phys. Rev. Lett. **45**, 566 (1980)
6. J.P. Perdew, A. Zunger, Phys. Rev. B **23**, 5048 (1981)
7. J.P. Perdew, K. Burke, M. Ernzerhof, Phys. Rev. Lett. **77**, 3865 (1996)
8. C.E. Dykstra, G. Frenking, K.S. Kim, G.E. Scuseria (ed.), *Theory and Applications of Computational Chemistry: The First 40 Years* (Elsevier, Amsterdam, 2005)
9. S. Goedecker, Rev. Mod. Phys. **71**, 1085 (1999)
10. D.R. Bowler, T. Miyazaki, Rep. Prog. Phys. **75**, 036503 (2012)
11. T. Ozaki, Phys. Rev. B **67**, 155108 (2003)
12. T. Ozaki, H. Kino, Phys. Rev. B **72**, 045121 (2005)
13. K. Lejaeghere et al., Science **351**, aad3000 (2016)
14. F. Mauri, G. Galli, R. Car, Phys. Rev. B **47**, 9973 (1993)
15. P. Ordejón, D.A. Drabold, M.P. Grumbach, R.M. Martin, Phys. Rev. B **48**, 14646 (1993)
16. X.P. Li, R.W. Nunes, D. Vanderbilt, Phys. Rev. B **47**, 10891 (1993)
17. M.S. Daw, Phys. Rev. B **47**, 10895 (1993)
18. E. Tsuchida, M. Tsukada, Phys. Rev. B **54**, 7602 (1996)
19. M.J. Gillan, D.R. Bowler, A.S. Torralba, T. Miyazaki, Comput. Phys. Commun. **177**, 14 (2007)
20. C.-K. Skylaris et al., J. Chem. Phys. **122**, 084119 (2005)
21. T. Ozaki, Phys. Rev. B **74**, 245101 (2006)

22. R. Haydock, V. Heine, M.J. Kelly, J. Phys. C **5**, 2845 (1972); **8**, 2591 (1975)
23. M. Aoki, Phys. Rev. Lett. **71**, 3842 (1993)
24. T. Ozaki, M. Aoki, D.G. Pettifor, Phys. Rev. B **61**, 7972 (2000)
25. W.T. Yang, Phys. Rev. Lett. **66**, 1438 (1991)
26. D.G. Pettifor, *Bonding and Structure of Molecules and Solids*, 1st edn. (Clarendon Press, 1995)
27. T. Ozaki, Phys. Rev. B **64**, 195126 (2001)
28. T.V.T. Duy, T. Ozaki, Comput. Phys. Commun. **185**, 777 (2014)
29. H. Sawada, S. Taniguchi, K. Kawakami, T. Ozaki, Model. Simul. Mater. Sci. Eng. **21**, 045012 (2013)
30. L. Lin, J. Lu, L. Ying, R. Car, E. Weinan, Commun. Math. Sci. **7**, 755 (2009)
31. T. Ozaki, Phys. Rev. B **82**, 075131 (2010)
32. T. Ozaki, Phys. Rev. B **75**, 035123 (2007)
33. M. Toyoda, T. Ozaki, Phys. Rev. A **83**, 032515 (2011)

Chapter 5

Acceleration of Classical Molecular Dynamics Simulations



Y. Andoh, N. Yoshii, J. Jung and Y. Sugita

Abstract In this chapter, we describe the acceleration and parallelization in classical molecular dynamics simulations. As electrostatic interactions are computationally intensive, the importance of the particle mesh Ewald (PME) method and the fast multipole method (FMM) will increase. These methods will be described here. In addition, general techniques for hierarchical parallelization on the latest general-purpose supercomputers (especially connected by a three-dimensional torus network), together with the critical importance of the data array structure, are explained. We show the optimization and benchmark results in the parallel environments of the molecular dynamics calculation programs, MODYLAS and GENESIS.

5.1 Classical Molecular Dynamics Simulations

Molecular dynamics (MD) calculation is a method to investigate the thermodynamic properties, structures, and dynamics of molecular assemblies, such as liquids, solids, gases, glasses, polymers, and biomolecules. By numerically solving the equations of motion of all atoms and molecules in the system, we directly obtain their trajectories. When the motion of the atoms follows the equation of motion as described by classical mechanics (Newtonian mechanics), it is called classical MD calculation.

To solve the equations of motion of atoms, it is necessary to obtain the force acting on each atom. When the force is sufficiently accurate and a numerical solution of the equation of motion is obtained with sufficient precision, it can be regarded that the

Y. Andoh · N. Yoshii (✉)

Center for Computational Science, Graduate School of Engineering, Nagoya University,
Furo-cho, Chikusa-ku, Nagoya, Aichi 464-8603, Japan
e-mail: yoshii@ccs.engg.nagoya-u.ac.jp

N. Yoshii

Department of Materials Chemistry, Nagoya University, Furo-cho, Chikusa-ku, Nagoya,
Aichi 464-8603, Japan

J. Jung · Y. Sugita

RIKEN Advanced Institute for Computational Science, Kobe, Japan

trajectory of the atom is equivalent to that of the real system. Various thermodynamic quantities and statistical mechanical functions can be obtained from the trajectories of the atoms. We can discuss the physics and chemistry of the target system.

Due to the recent widespread use of highly parallel computers, target systems for MD calculations become larger. MD calculations are now used in various fields, including materials science and bioscience. The development of high-precision general-purpose potential functions, such as AMBER [1], CHARMM [2], OPLS [3], and the development of high performance and multifunctional free software, such as GROMACS [4], NAMD [5], LAMMPS [6], has made it easier to perform high-quality MD calculations. However, using massively parallel machines with more than 1,000 nodes, it is difficult to efficiently execute these programs because of the reduction in parallelization efficiency caused by internode communication. It is necessary to choose algorithms and optimize data structures in programs to maintain high parallel efficiency in highly parallel environments. By applying various techniques, we can use the high performance of a massively parallel computer. In this chapter, we will explain the various algorithms and schemes used in the MD calculation software, MODYLAS [7], and GENESIS [8] developed by the authors, suitable for the massively parallel computers.

5.1.1 *Molecular Dynamics Calculation Flow*

Here, we outline the MD calculation flow (Fig. 5.1). First, we set the initial conditions. The coordinates and velocities of atoms are input as an initial condition. When atomic coordinates are obtained experimentally, their structures should be used as the initial coordinates. As a preparation for MD calculation, addition of the solvent molecules and energy optimization is applied, if necessary. In the simulation of liquids, we first prepare a lattice structure as an initial configuration. We obtain the equilibrated liquid structure by performing a high-temperature MD simulation. The initial velocity of each atom is assigned to reproduce a Maxwell distribution corresponding to the target temperature. In addition, we must set up the calculation conditions for each MD calculation, such as the parameters for mass and intermolecular interaction of each atom, statistical ensembles concerning temperature and pressure control, number of MD steps, time steps, and so on.

Next, we calculate the forces acting on each atom, which are assigned to intramolecular interactions acting through chemical bonds within the molecule and intermolecular interactions acting between atoms in different molecules or distant atoms in the same molecule. This intramolecular interaction consists of stretching potential which is a two-body interaction acting between bonded atoms, bending potential which is a three-body interaction, torsional potential which is a four-body interaction, and so on. The computational complexity of these interactions is $O(N)$, where N is the number of atoms in the system. However, the calculation amount for intermolecular interactions, such as electrostatic and van der Waals interactions, is $O(N^2)$. As a system becomes large, the computational amount increases drastically.

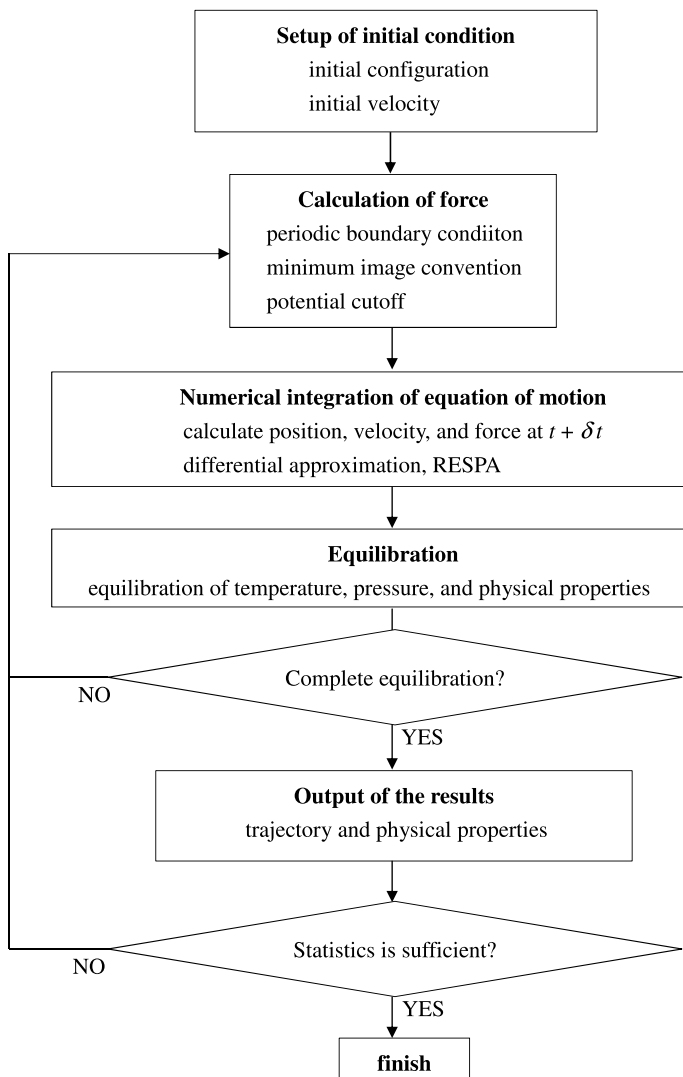


Fig. 5.1 MD calculation flow

The intermolecular interaction is the important factor in determining the size of the target system. Various techniques have been developed to accelerate MD calculations. This chapter focuses on the acceleration of intermolecular interaction calculations.

The numerical integration of the equation of motion is performed using the intermolecular force at time t , and the new velocity and position of the atoms at time $t + \Delta t$ are obtained. Difference approximation and predictor-corrector methods have been often used as numerical integration algorithms for the equation of motion [9, 10]. In

Sect. 5.1.4, we discuss the reference system propagator algorithm (RESPA) method, which enables multiple time steps and efficient numerical integration.

In the numerical integration of the equation of motion, rapid intramolecular degrees of freedom are often removed. For example, stretching and bending motions in water molecules are not so important, and they are often treated as rigid molecules that eliminate intramolecular degrees of freedom. The motion of the water molecules can be decomposed as the translational motion of the center of mass and rotational motion around the center of mass. Each motion can be obtained by numerically integrating Newton's equation of motion and Euler's equation [9, 10]. Alternatively, constraint dynamics may be used where the equations of motion are numerically solved with distance- or angle-constraints among atoms. By these methods, it is possible to introduce a larger time step Δt , which greatly improves the efficiency of the numerical integration of equations of motion in MD simulations. In Sect. 5.1.5, we discuss the constraint dynamics.

The physical quantity can be calculated using the position and velocity obtained from MD calculations. If the MD calculations reach an equilibrium state where the physical quantity of interest fluctuates around a certain value, the equilibration from the initial configuration is completed. The MD calculations are continued and we output the molecular trajectory and physical quantity at each step. If sufficient statistics are obtained, the MD calculations are terminated.

5.1.2 Algorithms to Reduce the Calculation Amount for Nonbonded Interactions

In general, the amount of calculation for nonbonded interactions is considerably larger than that for bonded interactions. The Lennard-Jones (LJ) interaction is composed of $(1/r_{ij})^{12}$ and $(1/r_{ij})^6$, where both converge rapidly to zero with r_{ij} and r_{ij} is the distance between the atom i and j . The LJ interaction can be truncated if r_{ij} is greater than a cutoff distance r_{cut} . Typically, r_{cut} is chosen to be $3\sigma - 4\sigma$, where σ is the LJ parameter that corresponds to the diameter of the atom.

By using a cutoff technique, most of the candidate j atoms located in a region distant from the i atom can be omitted without calculating the r_{ij} values. There are two established methods for this purpose: i.e., the Verlet neighbor list and linked-list cell (LLC) methods [11].

In the Verlet neighbor list method, a list for candidate j atoms is stored for each i atom by using a longer cutoff radius, $r'_{\text{cut}} = r_{\text{cut}} + \Delta r_{\text{cut}}$, than the actual r_{cut} . The list is updated per tens of MD steps. In the LLC method, a calculation unit cell is divided into subregions (subcells). Each atom is assigned to one of the subcells according to its position. The subcells where the candidate j atoms may be included are listed using the distance between subcell centers, which can reduce the search range of the candidate j atoms. The interaction is calculated between i and j atoms by applying a cutoff. These two methods are widely used in MD calculation programs

in which the LLC method is sometimes used in preprocessing to create the neighbor list effectively.

However, the electrostatic interaction is a function of $(1/r_{ij})$, which converges very slowly to zero with r_{ij} . Thus, the cutoff technique causes serious artifacts in the calculation results [12], even though a relatively large r_{cut} value is chosen. To date, some algorithms to reduce the calculation cost of the electrostatic interaction have been suggested without losing the accuracy of the calculated potential energy and forces. We will explain some of these algorithms below.

5.1.3 Acceleration of Electrostatic Interactions

In most MD calculations including all biomolecular systems, the electrostatic interaction calculation is the largest hot spot. When the electrostatic interaction is calculated directly, the calculation complexity is $O(N^2)$. If a cutoff is used, the computational cost becomes $O(N)$, but a discontinuous and large change of force and potential appears at the cutoff length, and a significant artifact occurs in the calculation result. Therefore, it is not permissible to use the cutoff.

The Ewald method has been used for a long time to evaluate the electrostatic interaction under periodic boundary conditions efficiently [9, 11]. In this method, the electrostatic potential $1/r$ is multiplied by the error and complementary error functions. The sum of these two functions becomes 1. With increasing x , the error function, $\text{erf}(x)$, attenuates slowly and the complementary error function, $\text{erfc}(x)$, attenuates quickly. The cutoff can be used for the latter. For the former, the function can be expressed as a sum of wavenumber vectors by a Fourier series. The sum of this Fourier series is obtained through fast convergence and does not require so many terms. However, the calculation amount increases as $O(N^{2/3}) \sim O(N^2)$. Therefore, it is difficult to apply to large-scale systems. Currently, the particle mesh Ewald (PME) method, which efficiently performs Fourier series, partly using the fast Fourier transform (FFT), is the mainstream technique [13, 14]. In the PME method, the point charges distributed in the MD cell are allocated lattice points using the interpolation technique to reproduce the electrostatic potential in the system. The contribution of the reciprocal-space of the Ewald method can be evaluated by the computational complexity $O(N \log N)$ by using the FFT for the charge on this lattice. However, care must be taken when executing the FFT in a highly parallel environment. Generally, the FFT requires all-to-all communication. In massively parallel computers of 1,000 nodes or more, all-to-all communication is often a major hurdle. This problem is successfully overcome by the GENESIS software described later, which achieves high performance [8]. On the other hand, the FMM [15] and multilevel summation method [16] were developed as non-FFT algorithms. The FMM with a periodic boundary condition is described in detail in Appendix.

5.1.4 Acceleration by Multiple Time Steps

In the numerical integration of the equation of motion, the time span for MD steps is determined by the fastest atomic motion in the system. Slow degrees of freedom must also be integrated by small time steps, which is quite inefficient. It is desirable to separate the degrees of freedom according to the rapidity of the molecular motion and integrate the equations of motion separately according to their suitable time steps. Multiple time steps enable this calculation.

First, the RESPA method [10], which is a numerical integration method that enables multiple time steps, is described. m_i , \mathbf{r}_i , and \mathbf{p}_i are the mass, coordinate, and momentum of atom i , respectively. $V(\mathbf{r}^N)$ is the potential energy of the system. When the Hamiltonian is given by

$$H(\mathbf{r}^N, \mathbf{p}^N) = \sum_{i=1}^N \frac{\mathbf{p}_i^2}{2m_i} + V(\mathbf{r}^N), \quad (5.1)$$

the Liouville operator corresponding to Eq. (5.1) is

$$iL = \sum_{i=1}^N \left(\frac{\mathbf{p}_i}{m_i} \frac{\partial}{\partial \mathbf{r}_i} + \mathbf{F}_i \frac{\partial}{\partial \mathbf{p}_i} \right). \quad (5.2)$$

The time evolution operator for time step Δt is given by $e^{iL\Delta t}$ [10]. The above Liouville operators are composed of two noncommutative operators $iL_1 = \sum_{i=1}^N \frac{\mathbf{p}_i}{m_i} \frac{\partial}{\partial \mathbf{r}_i}$ and $iL_2 = \sum_{i=1}^N \mathbf{F}_i \frac{\partial}{\partial \mathbf{p}_i}$. When the time evolution operator $e^{iL_1\Delta t + iL_2\Delta t}$ is symmetrically decomposed by Suzuki-Trotter expansion, it can be approximated as

$$e^{(iL_1+iL_2)\Delta t} = e^{iL_2 \frac{\Delta t}{2}} e^{iL_1\Delta t} e^{iL_2 \frac{\Delta t}{2}} + O(\Delta t^3). \quad (5.3)$$

By applying this to $\mathbf{r}_i(0)$ and $\mathbf{p}_i(0)$ at time $t = 0$, the well-known velocity Verlet method,

$$\mathbf{r}_i(\Delta t) = \mathbf{r}_i(0) + \Delta t \frac{\mathbf{p}_i(0)}{m_i} + \Delta t^2 \frac{\mathbf{F}_i(0)}{m_i} \quad (5.4)$$

$$\mathbf{p}_i(\Delta t) = \mathbf{p}_i(0) + \Delta t \frac{\mathbf{F}_i(0) + \mathbf{F}_i(\Delta t)}{m_i} \quad (5.5)$$

is obtained. For Hamiltonian dynamics systems, the numerical solution as described above becomes a symplectic integrator. The coordinates and the momentum are the canonical variables, and the discretized expression (5.4) has the conservative quantity called shadow Hamiltonian. Numerical integration is known to be stable over a long period of time [9, 10].

Next, iL is divided into fast iL_{fast} and slow degrees of freedom iL_{slow} . Then, the time evolution operator $e^{iL\Delta t}$ in Eq. (5.4) can be divided as

$$\begin{aligned}
e^{(iL_1+iL_2)\Delta t} &\approx e^{iL_{\text{fast}}\frac{\Delta t}{2}} e^{iL_{\text{slow}}\Delta t} e^{iL_{\text{fast}}\frac{\Delta t}{2}} \\
&= \left(e^{iL_{\text{fast}}\frac{\delta t}{2}} \right)^n e^{iL_{\text{slow}}\Delta t} \left(e^{iL_{\text{fast}}\frac{\delta t}{2}} \right)^n
\end{aligned} \tag{5.6}$$

where $\Delta t = n\delta t$, δt is the time step for the fast degree of freedom, and Δt is the time step for the slow degree of freedom. In Eq. (5.6), on the one hand, the fast part is divided into n so that we calculate the fast motion in a small time step. On the other hand, the part that moves slowly does not calculate the interaction. In general, the calculation cost of the slow part is higher; therefore, multiple time steps are very efficient. With respect to $e^{iL_{\text{fast}}\frac{\Delta t}{2}}$ and $e^{iL_{\text{slow}}\Delta t}$ in Eq. (5.6), \mathbf{r}_i and \mathbf{p}_i are updated according to the formula of the velocity Verlet method represented by Eq. (5.4).

Here, division of the Liouville operator can be done for each degree of freedom. However, there is no restriction on this division. For example, it is possible to separate total force acting on an atom into fast changing force caused by stretching, bending, and torsional motions, and a slowly changing one by LJ and electrostatic interactions.

In multiple time steps, a force acts on the atom for every δt and another force acts for each Δt . A certain node in the system may resonate with the force acting on every Δt . Some atomic motions may become unstable or the trajectory may greatly differ from the original trajectory [17]. In recent years, various schemes have been proposed to avoid this phenomenon [18]. It is necessary to pay attention to whether such things have happened.

5.1.5 Constraint Dynamics

The degrees of freedom in the molecule are often removed and the molecule is treated as a rigid body. For example, water molecules have three intramolecular degrees of freedom. By fixing the distance between oxygen and hydrogen atoms, the OH stretching motion can be removed. Furthermore, when the distance between the two hydrogen atoms is fixed, we can remove the bending motion of $\angle\text{HOH}$. By constraining these degrees of freedom, Δt can be increased and efficient calculation can be performed.

There are two methods to constrain the intramolecular degree of freedom. One is to handle a molecule as a rigid rotor model and the other is to use constraint dynamics. Here we describe constraint dynamics, which is applicable to various constraints and widely used in general-purpose MD calculation software.

Now, we consider that there is a chemical bond in a molecule and its bond length is fixed. In constraint dynamics, a constraint condition, i.e., the distance between atoms i and j is constant, is imposed. The binding force between atoms is considered to act on atoms to satisfy the constraint condition,

$$g = (\mathbf{r}_i - \mathbf{r}_j)^2 - d_{ij}^2 = 0. \tag{5.7}$$

The constraint condition expressed by Eq. (5.7) is called a holonomic constraint. The equation of motion of the atom is

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = \mathbf{F}_i + \lambda \frac{\partial g}{\partial \mathbf{r}_i}, \quad (5.8)$$

which includes the constraint force $\lambda \frac{\partial g}{\partial \mathbf{r}_i}$ by the constraint condition g . Here, λ is an undetermined Lagrange multiplier. When this equation of motion is numerically integrated according to the velocity Verlet method, the position coordinate at time $t = \Delta t$ is

$$\begin{aligned} \mathbf{r}_i(\Delta t) &= \mathbf{r}_i(0) + \Delta t \frac{\mathbf{p}_i(0)}{m_i} + \frac{\Delta t^2}{2m_i} \left\{ \mathbf{F}_i(0) + \lambda \frac{\partial g}{\partial \mathbf{r}_i} \right\} \\ &= \mathbf{r}'_i(\Delta t) + \frac{\Delta t^2}{2m_i} \lambda \frac{\partial g}{\partial \mathbf{r}_i} \end{aligned} \quad (5.9)$$

where $\mathbf{r}'_i(\Delta t) = \mathbf{r}_i(0) + \Delta t \frac{\mathbf{p}_i(0)}{m_i} + \frac{\Delta t^2}{2m_i} \mathbf{F}_i(0)$. The new position $\mathbf{r}_i(\Delta t)$ must satisfy the constraint condition g . By substituting $\mathbf{r}_i(\Delta t)$ into Eq. (5.7), we have

$$g = \left[\left\{ \mathbf{r}'_i(\Delta t) + \frac{\Delta t^2}{2m_i} \lambda \frac{\partial g}{\partial \mathbf{r}_i} \right\} - \left\{ \mathbf{r}'_j(\Delta t) + \frac{\Delta t^2}{2m_j} \lambda \frac{\partial g}{\partial \mathbf{r}_j} \right\} \right]^2 - d_{ij}^2 = 0. \quad (5.10)$$

Since this is a quadratic equation for λ , we can evaluate the constraint force $\lambda \frac{\partial g}{\partial \mathbf{r}_i}$ by solving for λ .

This is a simple case where the constraint on each atom is one. In the case of two or more constraints on one atom, the Eq. (5.10) is replaced by simultaneous quadratic equations including the same number of undetermined multipliers λ as the number of constraints. When the target molecule becomes large, such as macromolecules and proteins, the constrained atoms are connected to each other; therefore, the simultaneous equations are increased and the calculation cost for obtaining λ becomes very high. Thus, in general, Eq. (5.10) is linearized by ignoring the λ^2 term. Iterative calculations are performed until the constraint condition g is satisfied. This is the SHAKE method [19], which is widely used in general-purpose MD software. Various schemes to accelerate the convergence of iterative calculations in the SHAKE method have been proposed [20].

In addition, the velocity constraint was also obtained by differentiating Eq. (5.7) with respect to time. A constraint algorithm, the RATTLE method [21] for velocity similarly to the SHAKE method, was developed to satisfy the constraint condition.

Accelerated algorithms for constraint dynamics designed for water molecules, such as LINCS [22] and SETTLE [23], have been developed and are used widely.

These techniques can be introduced separately to the site to be constrained. It is not necessary to introduce new coordinates (e.g., Euler angle, quaternion), which are necessary for a rigid rotor model. Therefore, it is possible to keep the structure of the program simple; it is widely used for many general-purpose MD software. In

contrast, a rigid rotor model can increase the time step Δt in MD calculations [24]. As described above, several methods for constraining the intramolecular degree of freedom are available; therefore, we must choose an effective method for the target system.

5.1.6 Developments of Classical MD Simulations Related with HPC

5.1.6.1 Development of Parallelization

The main bottleneck in MD is the evaluation of nonbonded interactions, i.e., electrostatic and van der Waals interactions. Fortunately, the van der Waals interaction decreases rapidly as the interparticle distance increases. Therefore, if a sufficiently large cut-off distance is assigned, energy and force values in the van der Waals interaction can be regarded as zero when the interparticle distance is greater than the defined cutoff distance. Using Ewald summation description for electrostatic energy/force, interactions can be split into those in real- and reciprocal-space, and cutoff value can be assigned like van der Waals case [13, 14]. To deal with interactions with long-range distance, we perform energy and force calculation using fast Fourier transform (FFT) in the reciprocal-space. Therefore, it is necessary to consider parallelization of nonbonded interactions in both real- and reciprocal-spaces. Basically, there are three parallelization schemes for the real-space interactions: atomic, force, and spatial decompositions. Parallelization based on the atomic decomposition is a relatively simple algorithm, and historically introduced earliest in MD programs. The algorithm of parallelization based on spatial decomposition is more complicated than the atomic decomposition, but it is currently introduced in most MD programs because of high parallel efficiency. Detailed parallelization schemes based on spatial decomposition are slightly different in each program. NAMD applies a parallelization method combining spatial and force decomposition in the real-space interactions, and furthermore it schedules interactions between subdomains by CHARMM++ [5]. In addition, pencil (two-dimensional) decomposition is used to parallelize the FFT calculation that is required in the reciprocal-space calculation. In DESMOND, spatial decomposition based on the midpoint method is applied for parallelization in the real-space interaction. In addition, all-to-all communications are replaced by butterfly communications in one-dimensional FFT for the reciprocal-space interaction [25, 26]. Parallelization scheme of the real-space interaction in BLUE MATTER is similar to DESMOND, but the volumetric decomposition (three-dimensional decomposition) scheme is used for parallelizing the reciprocal-space interaction [27]. In GENESIS, the midpoint cell method [8] was introduced to improve the performance from the midpoint method, and FFT is parallelized using volumetric decomposition with one-dimensional all-to-all communications [28]. In addition, hybrid parallelization scheme combining OpenMP and MPI is introduced. In CHARMM and GRO-

MACS, a spatial decomposition method based on the eighth-shell scheme is adopted and non-bonded interactions in the real-space and reciprocal-space are performed in different CPU nodes [29, 30]. CHARMM only performs MPI communication, but GROMACS can make use of a hybrid parallelization combining MPI and OpenMP. In some programs, PME is replaced by another long-range force calculation method to increase the parallel efficiency. MODYLAS showed excellent parallelization efficiency on the K computer by introducing the fast multipole method (FMM) [7]. Recently, NAMD introduces a new scheme called multilevel summation method to increase parallelization efficiency [14].

5.1.6.2 Development of MD in Specialized Platforms

In recent years, dedicated computers have become a great influence for speeding up the MD energy/force calculation. MDGRAPE-3 accelerates MD by using specialized hardware in computing non-bonded interactions [31]. In ANTON [32], developed by D. E. Shaw Research, specialized hardware is used for the entire MD simulation. Reciprocal-space interactions are accelerated by using the Gaussian-split Ewald method. By such a contrivance, ANTON achieved 100 times faster performance. However, MD calculation using ANTON has limitations on the size of the molecular system. It is difficult to calculate a large-scale molecular system exceeding 1 million atoms with ANTON. In the latest ANTON 2, this restriction was relaxed, and high-speed calculation was possible for 2 million atoms system [33].

5.1.6.3 Development of MD for GPUs

An important development in MD from the viewpoint of hardware utilization is the use of a graphics processing unit (GPU). GPUs were originally developed as arithmetic chips dedicated to graphics, but today they have been used for various scientific and technical calculations. MD program using GPUs is roughly divided into two categories. The first one is to send all necessary information to GPU and perform all operations with GPUs. This method is adopted in AMBER [34], ACEMD [35], OpenMM [36], MOIL [37], DESMOND, and so on. Despite this method is very efficient when single node is used, there are restrictions in the system size that can be calculated due to the memory limitation of the GPU. It is also difficult to parallelize on multiple computers. The other method is to divide the overall calculation into a computation-intensive part, that is, calculation of the real-space non-bonded interactions and a communication-intensive part including reciprocal-space interaction with FFT. The former is mainly computed by GPUs and the latter is computed by CPUs. In this case, communication between the CPU and the GPU is required at every integration step. NAMD [38], GROMACS [39], and GENESIS [40] adopt this method. In this method, parallelization across multiple computers can be performed, so it is possible to handle a large molecular system. On the other hand, when using a

single node, the calculation speed of the latter is slower than computing using only GPUs.

5.2 Hierarchical Parallelization in the Latest Supercomputers

5.2.1 Hierarchical Hardware Structure

The latest supercomputers have a hierarchical structure for not only their calculation units, but also their devices to store calculation data.

Calculation units are composed of thousands or tens of thousands of calculation nodes, which are connected by a high-speed internode network. A calculation node sometimes has plenty of CPUs. The independent arithmetic units inside the CPUs are called the cores. In each core, units to execute various kinds of arithmetic operations are installed. Modern CPUs are also equipped with vector operation units (the same arithmetic operations on inputted multiple data), which work by single-instruction-multiple-data (SIMD) processing. A coprocessor to accelerate a part of the calculation is optionally installed onto the PCI-e bus at the same level as the CPUs, which is beyond the scope of this section.

As data storage devices, the main memory or random-access memory (RAM) devices (with tens of gigabytes (GB) of memory) are installed on each calculation node and accessed by the CPU(s). When one node has more than two CPUs, the access speed from each CPU to each main memory is mostly nonuniform; this is called nonuniform memory access (NUMA) architecture. In each CPU, a small but high-speed data storage device called cache memory is shared by several of the cores. These caches have levels: i.e., the level 1 cache with tens of kilobytes (kB) is the nearest cache from the arithmetic unit, and the level 2 cache with hundreds of kB is the second nearest one. The lowest data storage module is a processor register, which connects directly to the arithmetic units in the CPU, such as floating-point arithmetic unit (FPU). The data access speed of this module is the fastest, but it only stores a fixed amount of one integer or floating-point data. Vector registers can treat a large amount of data for SIMD operations.

Any parallelized software aiming at high parallelization efficiency must be designed considering the hierarchical structures of both calculation units and data storage devices in modern supercomputer systems.

5.2.2 Parallelization Strategy

Parallel execution between calculation nodes involves a distribution of calculation data onto the main memory devices on each calculation node. In standard operations,

such a parallelized execution is realized by the message-passing interface (MPI) by inserting additional MPI functions onto the original code. The unit of parallelization by the MPI is called the MPI process. In the case of several CPUs in one node or one CPU with the NUMA structure, allocating one MPI process onto one CPU or one NUMA unit often provides better parallelization efficiency.

Next, in parallel execution between the cores, the required calculation data on the cache(s) can be shared between the cores. This style of parallelized execution is usually realized by the OpenMP language extension, which is a standard component of modern Fortran/C language compilers. By adding a set of OpenMP directives before and after a DO loop (Fortran) or for loop (C), the arithmetic operations in the loop are distributed among the parallelized units (threads). Typically, one thread is assigned to one physical core.

Finally, when the CPU equips vector arithmetic devices, parallel execution by the SIMD instruction is possible for the deepest loops. For highest parallelization efficiency, it is essential to code a series of arithmetic operations at hot spots by vectorized assembler language or in the intrinsic instructions, although this requires high coding skills. A second-best method is to use the compiler's automatic SIMD parallelization function, in addition to the vectorized messages provided by the compiler. From these messages, we can observe which loops are vectorized, and if not vectorized, what elements inhibit SIMD parallelization of the loop. Better SIMD efficiency could be obtained by modifying the loop structure of problematic loops repeatedly.

Consider that hierarchical parallelization starts from the lowest MPI level, followed by the thread level, to the SIMD processing because the efficiency of the higher level parallelization is strongly restricted by the manner of implementation of the lower level parallelization. In particular, the parallelization efficiency of the code is dominated not by arithmetic efficiency (i.e., equal partitioning of tasks among parallelized units), but by the data transfer efficiency between hierarchical data devices. If the access time to data on a register is 1, it is 10 for data on the cache(s), and 100 on the main memory. Therefore, without establishing a smoothed data-flow path from the main memory to the register, an ideal parallelization efficiency could not be obtained at higher levels. Arithmetic devices waste time waiting for data to be calculated.

Section 5.3.2 describes an example of a sophisticated data structure that enables a smoothed data-flow path for MPI/OpenMP/SIMD hierarchical parallelization on supercomputer systems with a three-dimensional (3D) torus network.

5.2.3 Parallelization Techniques Based on Torus Network Characteristics

5.2.3.1 A Torus Network

A torus network is a network to connect calculation nodes in series and circularly along one direction. A 3D torus network is comprised of calculation nodes placed at 3D lattice points connected by a torus network along the x , y , and z axes. There can be more than three dimensions of network connections; e.g., the Tofu interconnect developed by Fujitsu Co. Ltd. provides a six-dimensional torus network for supercomputer systems with high fault tolerance to network troubles. A torus network is superior to other network structures, such as fat-tree networks from the following perspectives.

- High-speed and low-latency data transfer to first neighbor nodes along each axis.
- High affinity to MPI parallelization in spatial domain decomposition style.

However, it also has the following disadvantages:

- Need for relay nodes for communication with nodes other than the nearest neighbors.
- Increase of communication count, if software requires wide range communication.
- Frequent occurrence of collisions between two data transfers in opposite directions, if without a communication procedure design.

As schematically shown in Fig. 5.2a, a direct communication with calculation nodes on the diagonal line can be made possible by a two-step communication; i.e., the first communication along the x -axis followed by y -axis communication with a relay node. The situation becomes more severe when the target node is more distant from

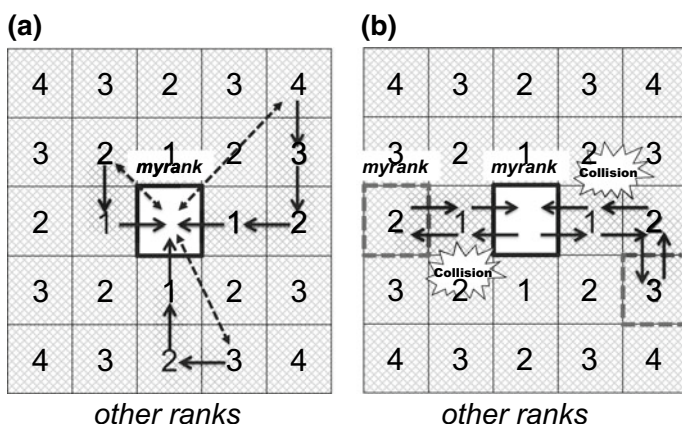


Fig. 5.2 Problems that can occur in a torus network. **a** Increase in communication count and **b** occurrence of collisions between two data transfers in opposite directions

the source node. The integer numbers in Fig. 5.2a indicates a communication count required for data transfer to the centered myrank from each rank. In total, 60 counts of communication are necessary to gather data from the surrounding hashed region to myrank. On the other hand, as shown in Fig. 5.2b, if each communication is not scheduled, collisions of two data transfers in opposite directions occur frequently everywhere. The occurrence of a collision event delays communication time significantly because one communication started after the other communication ended.

For parallelized calculations aiming at strong scalability, such as MD calculations, these delays in communication due to collisions are fatal. Thus, it is strongly desirable to implement MPI parallelization algorithms to minimize the communication counts without any collision events on the 3D torus network.

5.2.3.2 MPI Function Suited to a Torus Network

The `MPI_SENDRECV` is a MPI function to hide the disadvantages of a torus network as described above and to realize highly effective parallelization by the MPI.

```
call MPI_SENDRECV (sendbuf,scount,stype,dest,stag,
  recvbuf,rcount,rtype,source,rtag,comm,status,ierr)
sendbuf      : initial address of send buffer
scount       : element number in sendbuf
stype, stag  : type of elements in sendbuf, send tag
recvbuf      : initial address of receive buffer
rcount       : element number in recvbuf
rtype, rtag  : type of elements in recvbuf, receive tag
dest, source : rank of destination, rank of source
comm, status, ierr : communicator, status code, error status
```

This function executes `MPI_SEND` and `MPI_RECV` in one call; i.e., myrank sends `scount` elements of `sendbuf` with `stype` to `dest` rank, while it receives `rcount` elements of `recvbuf` with `rtype` from `source` rank. By calling this subroutine from every MPI process along one axis on a torus network, a circularly shifted communication is realized.

Figure 5.3 shows how the `MPI_SENDRECV` function can hide the disadvantages of a torus network in a two-dimensional (2D) case. Assuming that data on ranks 0–24, except for rank 12 (myrank), are transferred to the myrank. First, by calling the `MPI_SENDRECV` function, data on each rank are shifted along the $+x$ direction. Integer numbers below `s` and `d` in squares are values for rank of source and rank of destination as set in subroutine arguments, respectively. In this process, the function is called from all ranks simultaneously. As a result of these collective MPI communications, any collision events do not occur. Second, after the communication, the transferred data are merged with the originally owned data. The merged data are then shifted again along the $+x$ direction by calling the `MPI_SENDRECV` function. By applying the same communications along the $-x$ direction, distributed data on the ranks along the x -axis are gathered to the centered ranks (ranks 2, 7, 12, 17, and 22). Third, the merged data within each x line are shifted twice along the $+y$

direction by calling the `MPI_SENDRECV` function twice with setting source and destination numbers as shown in the right panel of Fig. 5.3. Finally, by shifting the merged data along the $-y$ direction twice in the same manner, all data on ranks 0–24 are transferred to `myrank` without any collision events.

In this technique, the communication count required to gather the data is only 8, which is 13% of the count for a series of fundamental one-to-one communications as shown in Fig. 5.2a. In a 3D case, the degree of reduction is more remarkable, i.e., from 450 to 12, which is a reduction rate of 97%. Furthermore, with this style of collective communication, all involved MPI processes gather the same range of data at once, which is in most cases a favorable situation for the subsequent arithmetic operations. If the torus network equipment has bidirectional communication architecture, such as the Tofu interconnect, the communication count can be further reduced to half.

For example, `MODYLAS` [7] software implements this kind of collective communication routine to gather the data of atom coordinates and coefficients of multipole expansions required to calculate potential energy and forces. The same type of communication algorithm is available in a process to send data back to the original processes, such as a scattering of reaction forces by the nonbonded two-body interactions.

The communication technique described here is widely applicable to any MPI parallelized software on a 3D torus network.

5.2.4 *Necessity of Sophisticated Data Structure for Highly Efficient Parallelization*

As described briefly in Sect. 5.2.2, the parallelization efficiency of hybrid parallelization is determined by the smoothness of data transfers between memory storage devices. First, an element controlling the smoothness of data flow is the data array structure, followed by proper coding at each parallelization level. An ideal data structure for hierarchical parallelization should have the following properties simultaneously:

1. Data are sequentially accessed at the time of arithmetic operation.
2. Data are localized, compartmentalized by small areas, and a data-blocking technique can be applied without using temporary arrays.
3. No data sorting and copying operations are required through a series of MPI communications.

Property 1 is important for the working efficiency of arithmetic units by smoothed data transfer from the cache(s) to registers. If not, software pipelining cannot be applied; therefore, the units waste time waiting for input data. If vector units are assumed, data sequentiality is more important to realize highly efficient vectorized operations. Data sequentiality is not normally achieved in particle-based calculations, such as MD calculations. For example, when the Verlet neighbor list method is adopted for pairwise additive LJ interaction calculations, the access to coordinate

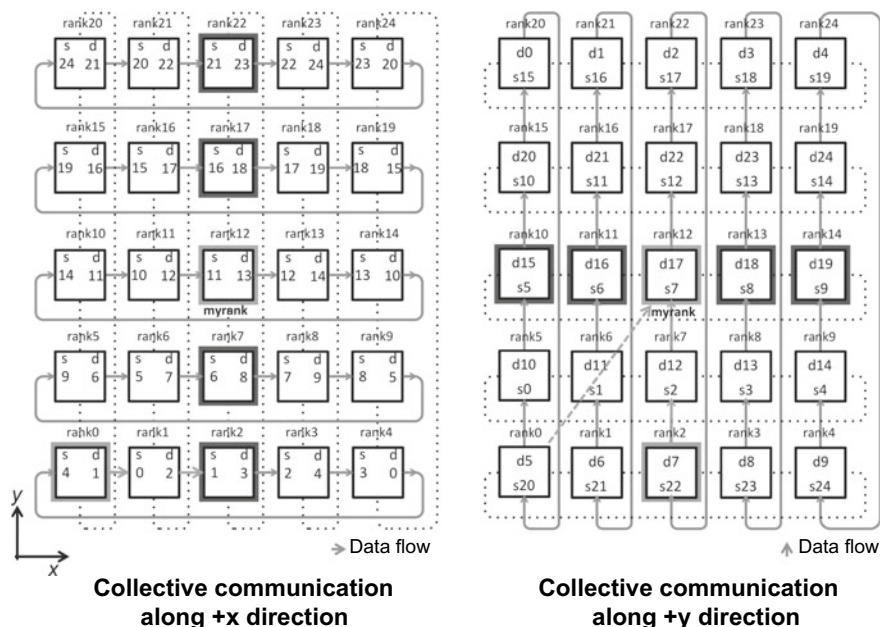


Fig. 5.3 An example of a circularly shifted communication on a 2D torus network realized by the MPI_SENDRECV function. First collective communication along the $\pm x$ -axis (left) and successive collective communications along the $\pm y$ -axis (right). In each stage, data is shifted by twice to plus or minus direction. Consequently, data on ranks 0–24 are transferred to rank 12 with 4 counts of communication

data based on a list of candidate j atoms becomes random in general, as shown in Fig. 5.4. This is because an arrangement of coordinate data on memory devices does not consider the relative distance between atoms. Much effort is needed to realize sequential access to coordinate data with the Verlet neighbor list method. In contrast, when the cell-link method is used, sequential access on data can be realized relatively easily with a well-designed data structure as described in Sect. 5.3.2.

Property 2 is important from the perspective of effective utilization of data on the cache(s) without reloading data from the main memory. In a calculation hot spot, it is desirable to load the data required for a series of arithmetic operations just once from the main memory to the cache. An established method, e.g., the data-blocking technique, is usually adopted for this purpose. The data-blocking technique assumes that a whole data set, such as atom coordinates stored in myrank, is subdivided into smaller data sets. The manner of division is chosen so that accesses to data in one data set are concentrated and data loading is performed with a unit of each data set. Unnecessary data reloading can be reduced by applying some modifications to the nested loop structures.

Property 3 is important from the perspective of strong scaling at process-level parallelization by omitting essentially unnecessary data sorting and duplication opera-

tions. Some types of parallelized calculations, such as MD calculations, in their scalability to MPI process number with a fixed problem size (strong scalability) is more important than the scalability by making problem size per process constant (weak scalability). Decreasing the overall elapsed time with large MPI process numbers, an accumulation of small times costed for pre- and post-processing of interprocess communications by the MPI functions could become a rate-determining process in parallelized calculations. This extra wasted time comes from the implementation style, which can be removed by devising the data structure appropriately.

Various data structures could have all of the features listed above, depending on the degree of hierarchy of hardware, kind of interconnections between calculation nodes, and whether coprocessors are used or not. Section 5.3.2 shows an example of the data structure suitable for MD calculations, which is hierarchically parallelized on multicore CPU nodes connected by a 3D torus network.

5.3 MODYLAS

This section introduces general techniques of a hybrid parallelization of large-scale calculations by MPI, OpenMP, and SIMD suited for supercomputers connected by a 3D torus network, with practical examples of implementation of the techniques on MD calculations that adopts the FMM (see Appendix) to calculate the long-range electrostatic interaction under a 3D periodic boundary condition. With a sophisticated data structure, it becomes possible to perform massively parallelized calculations with excellent parallelization and arithmetic efficiency [7].

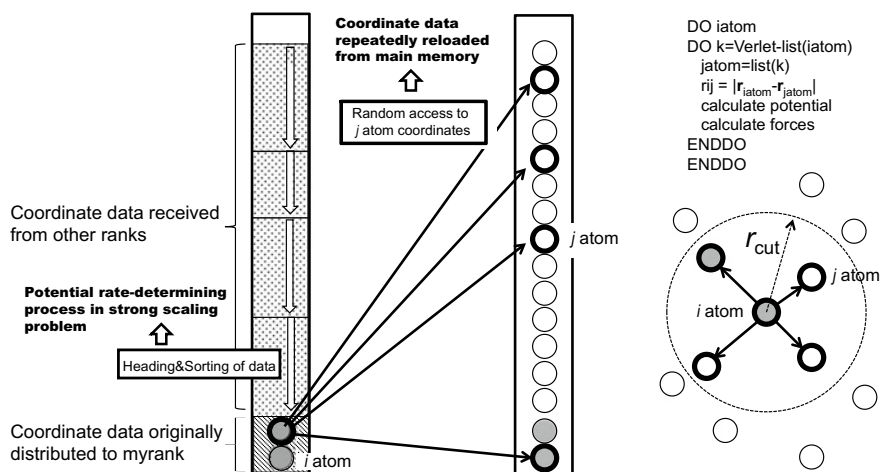


Fig. 5.4 General issues to inhibit efficient hierarchical parallelization with standard data structure for atom coordinates

5.3.1 Parallelization Characteristics of the FMM

From the perspective of massive parallelization, the FMM is superior to the PME method in the following aspects.

First, the data required to calculate both short- and long-range interaction are spatially localized at each level by its nested structure. Thus, adjacent communications can be used by the MPI instead of all-to-all type communications. In particular, if a 3D torus network is installed onto a supercomputer, the techniques described in Sect. 5.2.3 are directly applicable by minimizing a number of hops and avoiding data-transfer collisions. Second, the boundary of two kinds of potential energy calculations (P2P and L2P operations) is not spherical as in the Ewald method; therefore, access to r_j does not require a list of candidates of r_{js} . List access could become a barrier to achieving highly efficient vectorized arithmetic operations with SIMD instructions. Third, the FMM assumes a partitioning of a calculation unit cell into subcells; thus, the established LLC method (Sect. 5.1.2) can be used for the P2P operation. Furthermore, because atoms have metadata of the relative position of their belonging subcells, continuous access to data of atom coordinates r_{js} is possible with their well-designed data structure as shown in Sect. 5.2.4. Finally, since its calculation order is $O(N)$, there is a greater possibility to treat a much larger number of atoms in MD calculations using over 10,000 calculation nodes of exascale supercomputers [41] without losing calculation accuracy. The techniques described in this section can be the basis for such massively parallelized calculations in the near future.

5.3.2 A Metadata Structure of Coordinates and Multipoles

The interatomic interactions treated in MD calculations are distance dependent and reduce to zero with increasing distance between the source and destination atoms. Most of these interactions become completely zero at ten and a few angstroms, except for the electrostatic interaction. Therefore, it is beneficial to add supplementary information about spatial decomposition onto the atom coordinate data, which is one of the *metadata* sets. These metadata are also useful for other primary data used in electrostatic interaction calculations, such as multipoles in the FMM and grid charges in the PME method.

Figure 5.5 shows the data structure of atom coordinates `meta_xyz` with metadata of the relative distances between each subcell. Relative indices rather than absolute indices of each subcell are more favorable as metadata for homogeneous coding among different myranks. The data are represented in two ways: i.e., a one-dimensional representation as a copy of the original array (left panel of the figure) and a multidimensional representation emphasizing the metadata of the relative address of each subcell (right panel of the figure). The data originally stored by myrank are placed in a continuous manner not at the top of the array, but at the center of the array. The blanks arranged above and below the data are storage regions to receive

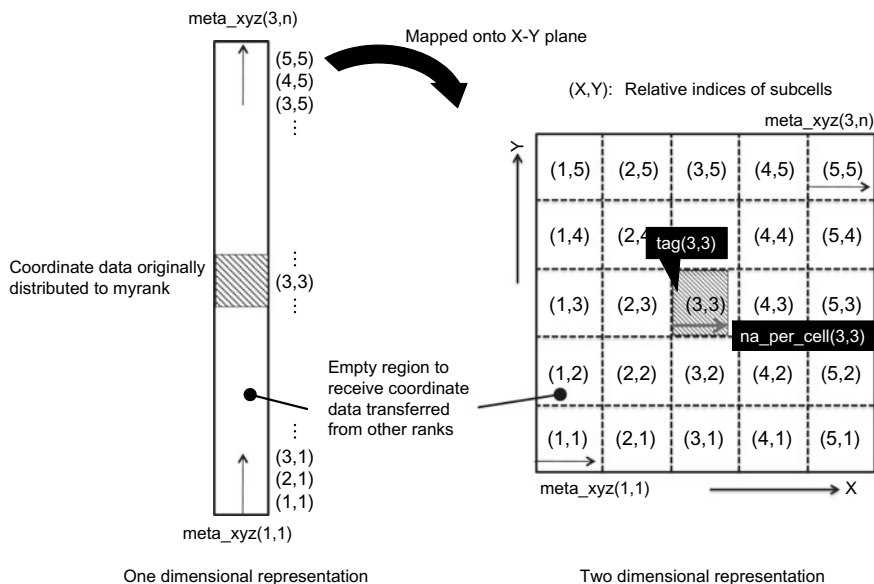


Fig. 5.5 Two kind of data array representations for atom coordinates with metadata structure

coordinates transferred from other surrounding MPI processes. In the multidimensional representation, blanks form a square halo in a 2D representation and a cubic halo in a 3D representation.

Access to the data is always performed using metadata by preparing two additional arrays. The first is a tag array, which stores initial addresses of a series of coordinate data included in each subcell (**tag** in Fig. 5.5), and the second is a counter array, which stores the number of atoms packed in each subcell (**na_per_cell** in Fig. 5.5). The dimensions of these arrays are the same as the metadata information; e.g., $(5,5)$ in Fig. 5.5. With these arrays, the data originally distributed to myrank with an address of $(3,3)$ are accessed by

```
DO i0=tag(3,3), tag(3,3)+na_per_cell(3,3)-1
  meta_xyz(1:3, i0)
ENDDO
```

Data to be transferred by a series of MPI communications are stored as shown in Fig. 5.6. These data are spatially localized around the original myrank data according to the following series of MPI communications. First, data transferred along the $\pm x$ -axis are placed at both sides of the original data by connecting its left ($+x$) and right ($-x$) end to the original data. As a result, data blocks are laid along the x -axis in which all coordinate data are arranged serially. Next, data transferred along the $\pm y$ -axis with data-block units from $(1,3)$ to $(5,3)$ are placed at the top and bottom of the line at address 3, where voids at both sides of the block are also transferred to avoid any heading and sorting of data in each communication process. With **tag**

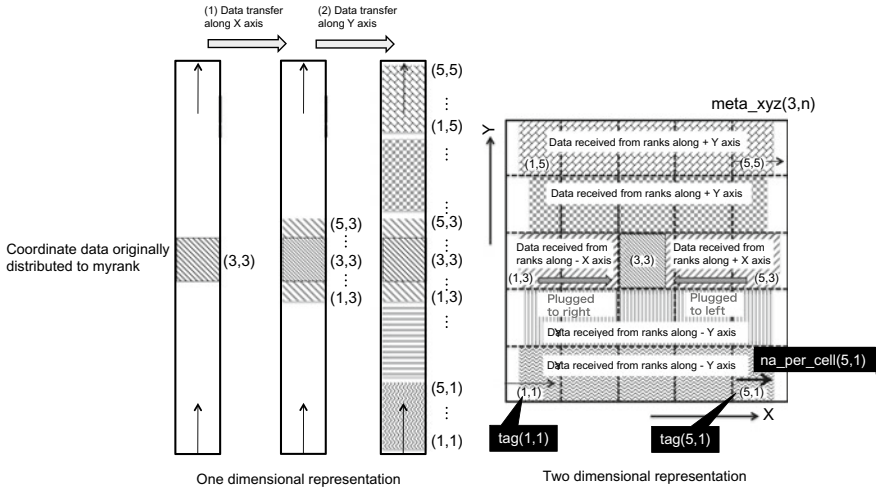


Fig. 5.6 Storage style for coordinate data transferred from other ranks

and **na_per_cell** reconstructed after the communications, any data stored in the halo region can be accessed in the same way as a method to access data on (3,3). For example, the coordinate data packed in a set of five subcells at line address 1 are accessed by

```
DO i0=tag(1,1), tag(5,1)+na_per_cell(5,1)-1
  meta_xyz(1:3, i0)
ENDDO
```

with a completely serial access favorable for the following vectorized arithmetic operations. In addition, a set of five subcells designated according to **tag** and **na_per_cell** with the start and end address of subcell block can be a data-blocking unit without introducing any temporary arrays.

Generally, several subcells are distributed to each rank, depending on the given total subcell number and MPI process. In such a case, the coordinate data are separately stored with a unit of a line along one axis, which is chosen as same as the first axis in a series of MPI communications optimized for a 3D torus network described in Sect. 5.2.3.2 and Fig. 5.3.

For the multipole data, a similar data structure with metadata of relative indices of subcells or supercell addresses is considered.

The M2M operation refers to multipoles on eight subcells or supercells with the same parent cell. The M2L operation requires multipoles on surrounding $875 (=10^3 - 2^3)$ subcells or supercells for each level where interactions with centered 2^3 cells are calculated at the lower level. Within this range of multipole information, the other operation is available. In the torus network and collective communications described in Sect. 5.2.3.2, data not only on target ranks, but also on intermediate ranks, are automatically gathered to myrank. Therefore, it is more efficient to receive

Multipole data originally distributed to myrank

```

meta_wm0(1:(nmax+1)2,5,5)
meta_wm0(1:(nmax+1)2,6,5)
meta_wm0(1:(nmax+1)2,5,6)
meta_wm0(1:(nmax+1)2,6,6)

```

- (1) Data transfer along X axis
- (2) Data transfer along Y axis



Multipole data received from other myranks

```

meta_wm0(1:(nmax+1)2,1:10,9:10)
meta_wm0(1:(nmax+1)2,1:10,7:8)
meta_wm0(1:(nmax+1)2,1:10,5:6)
meta_wm0(1:(nmax+1)2,1:10,3:4)
meta_wm0(1:(nmax+1)2,1:10,1:2)

```

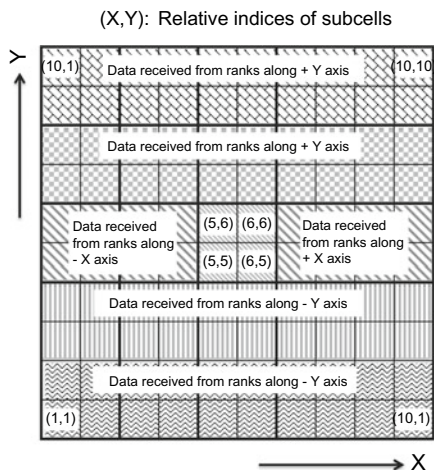


Fig. 5.7 Two-dimensional representations of data array for multipoles with metadata structure

redundant multipoles on surrounding 10^3 cells than to extract multipoles on 875 (for M2L) and 8 (for M2M) cells explicitly.

Figure 5.7 shows the data structure of multipoles at level 0 (**meta_wm0**) in a 2D representation where multipoles on four subcells at (5,5), (6,5), (5,6), and (6,6) addresses are distributed to myrank. The FMM is based on a hierarchical partitioning of a calculation unit cell where metadata of relative distance among subcells or supercells are equipped originally. Unlike the case of atomic number, the number of elements for each cell is constant, which corresponds to the number of expansion terms $(nmax + 1)^2$. Thus, it is not necessary to prepare additional arrays to store the metadata of data numbers for each cell.

In **meta_wm0** array, data are serially packed along the X -axis in each line. Data transferred from other ranks along the $\pm X$ -axis by MPI communications are placed at both sides of data in myrank. A data block in each line, e.g., $meta_wm0(1:(nmax + 1)^2, 1:10, 5)$ is sent to the adjacent rank by a MPI communication along the $+Y$ -axis, receiving a data block $meta_wm0(1:(nmax + 1)^2, 1:10, 4)$ from the adjacent rank in $-Y$ direction. By repeating send/receive communications, the empty region in the halo area is filled with multipole data necessary for the M2M and M2L operations.

5.3.3 Techniques to Realize Highly Effective Arithmetic Operations

5.3.3.1 Data Blocking

Data blocking is a general method to utilize the data on the cache(s). The data to be calculated are partitioned into a group of patches, which are small enough to be placed on the cache, especially level 1 cache (a few 10 kBs). It is desirable for calculation efficiency that the same data are loaded from the main memory to the cache only once. In other words, the data loaded on the cache are consumed thoroughly without flowing back.

In MD calculations, a data-blocking target is the coordinate data in the pairwise additive interaction calculations. If the FMM is adopted for the calculation method of the long-range electrostatic interaction, the multipole data on each subcell or supercell and partial elements of the transformation matrix in the M2L operation are to be blocked.

An issue in the pairwise additive interaction calculation by the LLC method is summarized in Fig. 5.8a. In the usual implementations of the LLC method, DO loop for i -cells owned by myrank is located at the outermost position, followed by j -cells loop. Inside these cell loops, loops for i atoms in the i -cell and j atoms in one specified j -cell are included. This loop structure is easy to understand, although there is a large amount of wasted time loading the coordinate data of j atoms; i.e., the range of access is determined separately by the position of each i -cell and most of them overlap each other. Thus, the same data of the j atom coordinates are repeatedly loaded from the main memory to the caches.

On the other hand, Fig. 5.8b shows a DO loop structure considering the data blocking of j atom coordinates. The outermost DO loop orders a shift of blocked data, i.e., **jcell_line** in the figure. A length of the **jcell_line** is five subcells in the direction where data are stored serially. The second inner loop orders a shift of i -cells in a direction vertical to the **jcell_line** within the decomposed domain (**icell_line**). The third DO loop selects i atoms in each subcell, followed by the fourth DO loop, which selects j atoms in each **jcell_line**. With this DO loop structure, coordinate data in each **jcell_line** are loaded only once, and all relevant pairwise interactions are calculated properly.

In typical biological systems, if a side length of subcell is about 6–7 Å, the number of atoms in each subcell is nearly 40. Thus, the data size of atom coordinates in each **jcell_line** is about 5 kB, which is small enough to be placed in the level 1 cache (typically, 32 kB in modern CPUs). For calculations of the LJ and electrostatic interactions, the force field parameters (σ , ε , and q) are also required. In general, the same parameters are reused for different atoms; therefore, the required data size for these is relatively small. The data size for i atoms is one fifth of j atoms, which costs only a few kBs of data.

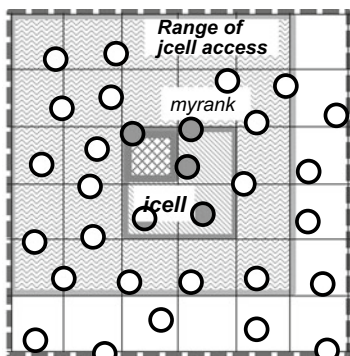
In the case of M2L operation, the same loop structure is adopted for a blocked arithmetic operation, as shown in Fig. 5.9. In the conventional loop structure, desti-

(a)

```

do icell(myrank)
do jcell
do iatom=tag(icell),
    tag(icell)+na_per_cell(icell)-1
do jatom=tag(jcell),
    tag(jcell)+na_per_cell(jcell)-1
    Calculate potential and forces
    between iatom and jatom
enddo
enddo
enddo
enddo

```



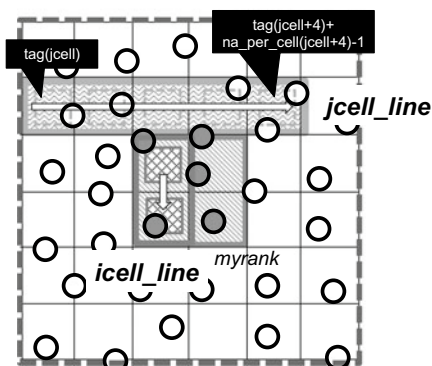
Region where coordinate data were transferred

(b)

```

do jcell_line Move jcell blok loop to outermost
do icell(myrank) [shift icell along icell_line]
do iatom=tag(icell),
    tag(icell)+na_per_cell(icell)-1
do jatom=tag(jcell),
    tag(jcell+4)+na_per_cell(jcell+4)-1
    Calculate potential and forces
    between iatom and jatom
enddo
enddo
enddo
enddo

```



Region where coordinate data were transferred

Fig. 5.8 An example of the data blocking in the pairwise additive calculation of interatomic interactions

nation cells are shifted by the outermost DO loop, followed by a shift of source cells. For each selected cell pair, matrix (M2L transformation matrix) times vector (multipole elements) arithmetic operations are executed. However, in the data-blocking arithmetic operations, the multipole data on the surrounding 875 subcells or supercells are partitioned into a set of small patches. Similarly to the blocked pairwise additive calculation, the outermost DO loop shifts a patch of source cells. The second inner DO loop shifts a destination cell within the domain distributed to myrank. After selecting source and destination cell pairs, a matrix-times-vector operation is executed. The optimal size of each block depends on the truncated degree of multipole expansions n_{\max} . The expansion coefficient and translation matrix elements are complex numbers (16 bytes (B) in double precision), and the required data size for blocks is an integer multiple of $16 \text{ B} \times (n_{\max} + 1)^2$, which should be less than the cache size on the hardware used.

When the data-blocking technique is adopted, the data in each block are also required to be serially packed on the main memory to reduce the frequency of data transfer to the cache and for the following vectorized arithmetic operations by SIMD instructions. An easy way is to prepare a temporary array for blocked arithmetic operations by copying the original data to the temporary array with partitions between each block. However, as described below, the time consumed in a data copy operation from the original array to the temporary array could become a rate-determining process when the elapsed time is greatly reduced by hierarchical parallelization of a code. With the metadata structure of coordinate data as described in Sect. 5.3.2, the temporary array and copy operation are omitted completely because the coordinate data are already partitioned based on small units (subcells) and data are serially packed along one axis by a series of MPI communications. Consequently, the atom coordinates in an arbitrary number of subcells in one direction can be serially accessed by a combination of **tag** and **na_per_cell**.

In conclusion, the metadata structure is excellent from the perspectives of not only low latency MPI communications, but also the data-blocking technique for the following arithmetic operations.

5.3.3.2 Thread-Level Parallelization

Thread-level parallelization is realized by the OpenMP language extension, which is activated by adding the OpenMP directives. Since the latest hardware development trend is toward many cores and wide SIMD architectures, it becomes critically important to apply efficient thread-level parallelization at every hot spot.

Briefly, the range of thread-level parallelization in a Fortran code is defined by a pair of `!$omp parallel` and `!$omp end parallel` directives (`#pragma omp parallel` and `#pragma omp end parallel` in a C code). In a defined parallelized region, any DO loops sandwiched by `!$omp DO` and `!$omp end DO` directives are the target of thread-level parallelization (`#pragma omp for` in a C code, without end statement). In addition, a proper compile option to a compiler (e.g., `-fopenmp` for gfortran, or `-qopenmp` for ifort, `-mp` for pgfortran) is required to interpret the inserted directives. Modern compilers are equipped with the auto thread-level parallelization function, which is activated by the defined option (`-parallel` for ifort, `-Mconcur` for pgfortran). In practice, however, it is only applicable to simple codes, such as setting of initial array values because the compiler assigns priority to keep the auto parallelized code accurate. Therefore, an explicit thread-level parallelization of calculation hot spots by inserting the OpenMP directives is required.

A general point to be noted for efficient thread-level parallelization is the uniform load balancing between threads. The easiest but effective way is to elongate the length of the target loop for thread-level parallelization compared with a given thread number. By default, a block of a whole loop length divided by N_t is assigned to each thread. Thus, the longer the loop, the smaller the load imbalance between threads becomes. Frequently, a fusion technique of two or more loops is used to elongate the loop length. The collapse clause in the OpenMP is prepared for this purpose; i.e.,

```

!$omp parallel
!$omp do
do iblk = 1, nblock
do icy = icyblkst ( iblk ), icyblkend ( iblk )
do icx = icxblkst ( iblk ), icxblkend ( iblk )
do icell ( myrank )
if ( icx, icy is within 2 nearest neighbors ) cycle
do m1 = 1, ( nmax + 1 )2
do m2 = 1, ( nmax + 1 )2
w ( m1 ) = w ( m1 ) + m2l ( m1, m2 ) * wm ( m2 )
enddo
enddo
enddo
enddo
enddo
!$omp end do
!$omp end parallel

```

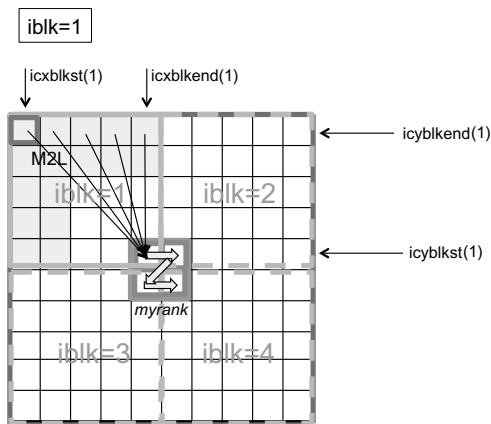


Fig. 5.9 Blocked arithmetic operations parallelized with the OpenMP directives in the M2L operation

multiple loops are merged into one loop in the compiling process and thread-level parallelization is performed on the merged loop.

The next method is to design the size of loop blocks assigned to each thread explicitly, if possible, considering blocked arithmetic operations. In addition, it is desirable that the number of blocks is integer times the number of threads from the perspective of load balancing of parallelized calculations between threads. This method achieves excellent load balancing, especially when the amount of calculations between each pair of source block and destination patch is uniform.

Figure 5.9 shows an example of the blocked arithmetic operations parallelized with the OpenMP directives in the M2L operation. The outermost loop with DO variable **iblk** shifts the block of source cells with a total nblock number (nblock = 4 in this case). If given thread number N_i is 4 (determined by an environmental variable of operation system OMP_NUM_THREADS when a code is executed), one block composed of 25 cells is assigned to each thread. The block shape is controlled by the index arrays icxblkst, icxblkend, icyblkst, and icyblkend with an argument, **iblk**. In a 3D case, an additional two index arrays are introduced, iczblkst and iczblkend.

When the amount of calculations is not uniform for each block, which is the general case for pairwise additive interaction calculations, a more sophisticated thread-level parallelization algorithm is required to achieve high parallelization efficiency with good load balancing.

5.3.3.3 Vectorized Arithmetic Operations

For operations within the innermost loop, vectorized arithmetic operations could be applied by the SIMD instructions on modern CPU architectures. Its prerequisite is

that the same kinds of operations are performed on a series of input data. In MD calculations, e.g., pairwise additive calculations of the LJ and electrostatic interactions fully satisfy this prerequisite. That is, the data of atom coordinates and force field parameters for j atoms in the `jcell_line` and i atoms in i -cell are loaded, and then the potential energy and force are calculated by the equations in the same functional form. The M2L operation is also a good candidate for vectorized arithmetic operations because multiplications of the coefficient matrix and multipole element row are repeated for each pair of source and destination subcells or supercells.

There are several ways to implement vectorized arithmetic operations. The easiest way is to use the compilers' auto-SIMD optimization or autovectorized functions, which are activated by adding specific compiler options (e.g., `-msse` for `gfortran`, `-vec` for `ifort`, or `-fastsse` for `pgfortran`). Compilers are also equipped with an option to output vectorized reports of a code, sometimes with a function annotating the elements to inhibit the generation of vectorized instructions (e.g., `-qopt-report` for `ifort`). Performance of vectorized arithmetic operations in a calculation hot spot could be improved step by step, modifying the code being more suited to vectorized arithmetic operations based on a vectorized report. Directives for SIMD optimization (`!$OMP SIMD` [42]) are now prepared from the OpenMP 4.0, which are activated in another compiler option (e.g., `-qopenmp-simd` for `ifort`). With these SIMD directives, more detailed information to enhance autovectorization can be added to the compiler.

The second, but highly professional way is to use SIMD's intrinsic functions [43]. An optimized set of vectorized codes for basic mathematical operations are prepared for each vector technology (e.g., SSE, AVX, AVX-512) and precision. However, intrinsic coding does not always result in highly effective vectorization of a hot spot. It is only valid when the code is rate-limited by arithmetic operations themselves: If not, vectorized arithmetic units waste most of the time waiting for input data. Therefore, in advance of optimization for vectorized arithmetic operations, hierarchical coding based on a sophisticated data structure must be realized to achieve excellent total parallelization efficiency of the code. For example, our proposed metadata structures for coordinates and multipoles satisfy the condition for well-vectorized arithmetic operations. The blocked coordinate data for j atoms in `jcell_line` or multipoles on source cells are placed serially in the level 1 cache without delays in loading time for data from the main memory.

We outline the general considerations for highly efficient vectorized operations as follows. First, the serial dependency between elements in data arrays should be removed by modifying a code or the algorithm itself. In general, backward or forward reference to data, such as

```
DO
  a[i]=a[i-1]+b[i]
ENDDO
```

or

```
DO
  a[i]=a[i+1]+b[i]
```


ENDDO

where loop index i is difficult to be vectorized because the present arithmetic operation for $a[i]$ depends on the past $a[i-1]$ or future $a[i+1]$ data with the possibility of being updated. It is also difficult to vectorize a code effectively when a loop contains an indirect access to a data array element by introducing an index list array, such as

```
DO
  a[list(i)]=a[list(i)]+b[i]
ENDDO
```

with loop index i . There are some established tricks to overcome these difficulties, whereas the best way is to adopt an alternative algorithm without backward or forward reference to data.

Second, the type of arithmetic operation to input data in the target loop is as uniform as possible; e.g., in the calculation of the LJ interactions in MD calculations, a cutoff technique (Sect. 5.1.2) is usually adopted. Its execution depends on the distance between i and j atoms. Usually, an IF sentence may be introduced into the target loop of vectorized arithmetic operations, which makes arithmetic operations in the loop nonuniform. Another example in MD calculations is an exclusive calculation of interatomic interactions in the same molecule. Since interatomic interactions between atoms separated by one chemical bond (the so-called 1–2 interaction) and by two chemical bonds (the 1–3 interaction) are evaluated by bonded potential terms and angled potential terms, they must be eliminated from interatomic LJ and electrostatic interaction calculations to avoid a duplicated estimation of interactions. In addition, a scaled calculation (or calculation with special parameters) of an interatomic interaction between atoms separated by three chemical bonds (the 1–4 interaction) in the same molecule should be considered in longer molecules, since the interaction is partially evaluated by dihedral potential terms. In the former, the code can be made uniform by introducing a cutoff variable with a value of 0 (if $r_{ij} > r_{\text{cut}}$) or 1 (else) multiplied by the calculated potential and forces. In the latter, it may be possible to use a redundant calculation and removal technique that calculates all pair interactions once, and then eliminates the pair interactions from the total. In most cases, the removal process code is complicated and does not suit vectorization. However, since the calculation amount is much higher in the uniform calculation than the latter removal calculation, this method greatly improves the total performance. An issue is that a large round off could produce errors when subtracting repulsive 12-power terms in the LJ interaction. The error is acceptable with a double-precision floating-point calculation, while it is not acceptable in the case of a single-precision calculation. In such a case, another technique, such as mask processing to avoid the 1–2 and 1–3 calculations should be implemented to keep the MD calculations accurate.

Third, the loop length is long enough to conceal the pre- and post-processing time for vectorized arithmetic operations and enhance the software pipeline. It is also desirable from the perspective of load balancing between vector lines. An easy way to elongate a loop length is a fusion of the loop with upper loop(s) as in the case for thread-level parallelization by the OpenMP directives described above. However,


```

!$omp parallel
do jcell_line
do icell [along icell_line]
!$omp do
do iatom=tag(icell),
tag(icell)+na_per_cell(icell)-1
do jatom=tag(jcell),
tag(icell+4)+na_per_cell(icell+4)-1
rij=rjij(ri,rj)
if(rij>rcut) LJ_epsilon=0d0
phi_nonbond=phi_nonbond+phi_ij
f(i)=f(i)+Fi
f(j)=f(j)+Fj
enddo
enddo
!$omp end do
enddo
!$omp end parallel

```

Target do loop for vectorized arithmetics

This "if sentence" is replaced by a mask processing by Fujitsu compiler

Calculate once the potential energy and force for the all pairs without distinguishing 1-2, -3, -4, and others

```

!$omp parallel
do icell [along icell_line]
!$omp do
do iatom=tag(icell),
tag(icell)+na_per_cell(icell)-1
do jatom=1,voidpair123(iatom)
rij=rjij(ri,rj)
phi_nonbond=phi_nonbond-phi_ij
f(i)=f(i)-Fi
f(j)=f(j)-Fj
enddo
do jatom=1,scalepair14(iatom)
rij=rjij(ri,rj)
x=1-s
phi_nonbond=phi_nonbond-x*phi_ij
f(i)=f(i)-xFi
f(j)=f(j)-xFj
enddo
enddo
!$omp end do
enddo
!$omp end parallel

```

Removal of 1-2 and 1-3 interactions to be omitted

Removal of scaled 1-4 interactions with scaling factor s

Fig. 5.10 Vectorized arithmetic operations of the pairwise additive LJ and electrostatic interactions for a molecular system, which includes the 1–2 and 1–3 interactions to be omitted and the 1–4 interactions to be scaled in the same molecule

it usually involves an introduction of an index list that relates the loop variable of a fused new loop to the loop variables in the original loops. As stated above, an indirect access by an index list makes it difficult to create an optimized vectorized code using a compiler because the data dependency between loops becomes opaque for a compiler. In any case, the best way is to elongate the target loop in the original code from the perspective of calculation algorithms.

Figure 5.10 shows an example of the SIMD-optimized code for pairwise additive calculations of the LJ and electrostatic interactions, including thread-level parallelization by the OpenMP directives adopted by the MODYLAS [7] software. The left block of the program is executed first, followed by the right block. The third DO loop with DO variable **iatom** in the left block is the target of thread-level parallelization. The innermost DO loop with DO variable **jatom** is the target of vectorized arithmetic operations; its loop length is determined as the [average number of atoms in each subcell (about 40)] × [number of subcells in jcell_block (5)] ≈ 200 for typical biological systems. The cutoff processing of the LJ interaction is performed by the IF sentence, which is replaced by mask processing in the compiling process by Fujitsu's Fortran compiler. The potential energy and forces are removed from their total by the third DO loop in the right block, which detects the atoms of the 1–2 and 1–3 interaction sources for each iatom. Similarly, the fourth DO loop in the right block detects the 1–4 interaction source for each iatom in which the calculated interaction multiplied by $x = 1 - s$ (s : original scaling factor) is removed from the total.

By a performance measurement on the K computer, which has a 128-bit SIMD with for double-precision floating-point calculations and eight cores (eight threads

execution) in each CPU, the left program block gives over a 95% SIMD instruction fraction to floating-point arithmetic operations with a very low cache miss rate at less than 1% and nearly 0% for the level 1 and 2 caches. In addition, the distributed memory parallelization by the MPI keeps scalability up to 65,536 (2^{16}) processes with an input of a 10 million atom system when the long-range electrostatic interactions are calculated by the FMM [7].

5.3.4 Future Prospects

In the exascale era, general-purpose supercomputers with many cores and vectorized units with wider SIMD widths will be installed. The quality of not only thread-level parallelization, but also vectorized arithmetic operations becomes more critical in performing highly effective parallelized calculations on such supercomputers. Obviously, points to note for achieving highly effective parallelization are a good load balance and the uniformity of arithmetic operations between parallelized units. These are realized by temporarily applying some established methods, although the best way is to modify the calculation algorithm at the hot spot itself as we recently proposed for the P2P operation [44]. New algorithms for parallelization should be continuously invented to adopt to future supercomputer architectures.

5.4 GENESIS

5.4.1 Main Features of GENESIS for Optimization and Parallelization

To extend the simulation system size and time is a great challenge in MD due to small integration time step and required long simulation time. GENESIS (Generalized-Ensemble Simulation System) is implemented to address these challenges by optimizing the program and developing efficient algorithms for ensemble generations [8]. In this section, we focus on the developments in GENESIS from the point of view of optimization. Because the electrostatic interaction in GENESIS is based on particle mesh Ewald (PME), we will discuss optimization on two points: efficient parallelization of three-dimensional Fast Fourier Transform (FFT) and evaluations of finite-range interactions in the real-space.

5.4.1.1 Inverse Lookup Table [45]

In MD, the major bottleneck is the calculation of non-bonded interactions including van der Waals or electrostatics. Especially in the PME method, much of the opera-

tion cost is spent on the square root function and the error function evaluations. MD software, therefore, usually employs a calculation method that avoids calling these mathematical functions. Instead, a numerical lookup table obtained by performing numerical interpolation is generated in advance and used for non-bonded interactions. The feature of GENESIS's lookup table is that the numerical interpolation is performed evenly in proportion to the reciprocal of the distance squared. Using this lookup table, it is possible to accurately calculate the interaction energy and the force at short distance since many interpolation points are used. As for the large distance interaction, it is possible to increase the performance by using a small number of interpolation points. Given the interaction distance r and the cutoff distance r_v , respectively, it is possible to define the interpolation point coordinates L using the interpolation point density D ,

$$L = \text{INT} \left(D \times \frac{r_v^2}{r^2} \right) \quad (5.11)$$

Then, the energy function $E(r)$ using the lookup table can be written as

$$E(r) = E_{\text{tab}}(L) + t(E_{\text{tab}}(L+1) - E_{\text{tab}}(L)) \quad (5.12)$$

$$t = D \times \frac{r_v^2}{r^2} - L \quad (5.13)$$

By using the lookup table scheme with Eqs. (5.11)–(5.13), which we named inverse lookup table, CPU cache memory can be used efficiently, so high-speed nonbonded interaction evaluation can be performed. Even using small number of interpolation points, it provides accurate energy and force calculation.

5.4.1.2 Spatial Decomposition Scheme in GENESIS

In GENESIS, there are two MD simulators: ATDYN (ATomic decomposition DYNamics) and SPDYN (SPatial decomposition DYNamics). ATDYN is parallelized based on the atomic decomposition, and each MPI processor has all information of the system, such as coordinates, velocities, charges, and so on. Parallelization of SPDYN is based on the spatial decomposition scheme. SPDYN is particularly designed for large-scale MD simulations suitable for recent multicore CPUs. Therefore, in this section, we introduce the parallelization method of GENESIS, limited to SPDYN. In SPDYN, the simulation space is divided into the same subdomain as the number of MPI processors. Subdomain is further divided into smaller unit domains called cell. The length of cell in each dimension is limited to a size greater than half of the cutoff distance. Particle data like coordinates or velocities are grouped according to cell indices. Pairwise interactions are grouped by cell pairs which are distributed over OpenMP threads by shared memory parallelization scheme using OpenMP.

(1) *Midpoint cell method* [8]

The midpoint cell method is an extension of the existing midpoint method for hybrid parallelization of MPI and OpenMP. In the midpoint method, the non-bonded interaction between a particle pair is evaluated in the subdomain containing the midpoint of the particle pair. Therefore, a particle pair interaction is often performed in a subdomain where none of the particles exist. In the case of using the cut-off distance or pairlist cutoff distance, r_v , it is sufficient to import coordinate data within a distance of $r_v/2$ from each subdomain using the midpoint scheme. In the midpoint cell method, particle data are grouped cell-wise and the interaction subdomain is not decided from the midpoint of each particle pair but from the midpoint cell of each cell pair in which each particle resides. The midpoint cell is sometimes not uniquely defined. For example, in Fig. 5.11, the midpoint cell of the cell pair a and b is uniquely determined. On the other hand, there are two possibilities for the midpoint cell of the cell pair c and d. In such a case, the midpoint cell is determined considering the load balance in the actual operation. In the case of using the midpoint cell method, it is only necessary to communicate from/to the adjacent cells of each subdomain. In both midpoint and midpoint cell methods, communication cost is reduced by increasing the number of processors, enabling high parallel efficiency. In the existing midpoint method, it is necessary to determine the subdomains including the midpoints for all particle pairs every step or whenever pairlist is rewritten. On the other hand, in the midpoint cell method, midpoint cells are decided before running MD simulation.

(2) *Volumetric decomposition FFT* [28]

By using the PME method, the amount of computation required for the non-bonded interaction decreases from $O(N^2)$ to $O(N \log N)$ when the total number of particles is N . However, since the FFT calculation in PME requires global all-to-all communications, FFT becomes a main bottleneck in MD when using very large number of processors. In NAMD and GROMACS, slab (one-dimensional) or pencil (two-dimensional) decomposition schemes are used, whereas GENESIS uses a parallelization method based on volumetric (three-dimensional) decomposition. Volumetric decomposition FFT requires more frequent all-to-all communication than the case of using slab or pencil decompositions, but the number of processors involved in each communication is minimized. For this reason, FFT calculation using volumetric decomposition is generally suitable for MD simulations of big systems using many processors. Furthermore, by applying the same node topology as volumetric decomposition, volumetric decomposition can be useful for supercomputers equipped by torus network type like K. When the midpoint cell method and volumetric decomposition FFT are combined to each other, all-to-all communication of charge information can be skipped. For example, in Fig. 5.12a, the information of the charge in the reciprocal-space held by the processor 10 can be obtained without any communication from the information held by the same processor in the real-space. If different decomposition scheme is done between the real- and reciprocal-spaces, shown in

Fig. 5.11 Spatial decomposition scheme in GENESIS SPDYN

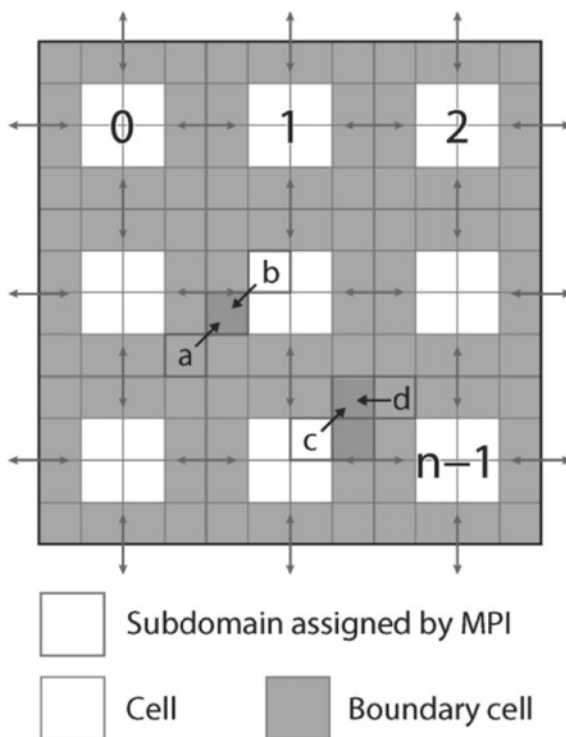


Fig. 5.12b, the charge information in the reciprocal-space in the processor 10 can be obtained by communication among processors 3, 7, 11, and 15.

5.4.2 Benchmark Performance of GENESIS

Here we introduce the benchmark results of GENESIS using PC clusters and the K computer. As for the PC clusters, we used 32 nodes, each of which consists of two Intel Xeon E5-2670 CPUs. Since the number of cores included in this CPU is 8, the number of cores in a node is 16, so the PC cluster consists of 512 CPU cores. We used three molecular systems for the benchmark, which are listed as followings:

- (1) Main porin from *Mycobacterium smegmatis* (MSPA, 216, 726 atoms in $126.933 \times 126.933 \times 131.063 \text{ \AA}^3$ box)
- (2) Satellite tobacco mosaic virus (STMV, 1,066,628 atoms in $216.83 \times 216.83 \times 216.83 \text{ \AA}^3$ box)
- (3) 27 STMV (28, 798, 956 atoms in $650.49 \times 650.49 \times 650.49 \text{ \AA}^3$ box)

The third molecular system is obtained by magnifying three times in each dimension from (2). The number of FFT grids are (144, 144, 144), (256, 256, 256), and (512, 512,

not decrease up to 16,384 nodes, and the maximum speeds using the velocity Verlet and RESPA MTS integrations are 14.60 ns/day and 17.51 ns/day, respectively.

Our benchmark results show excellent parallelization efficiency of GENESIS. Despite executing FFT calculation including all-to-all communication every step, high parallelization efficiency is maintained both in PC cluster and K. The performance of GENESIS shows even better values using RESPA MTS integration scheme.

Exercises

1. **Process number calculation:** MD calculations are often performed under the three-dimensional periodic boundary condition. In a circularly shifted communication using the MPI_SENDRECV function described in Sect. 5.2.3.2, it is necessary to determine process numbers **dest** and **source** with taking into account of the periodic boundary condition. Given that the total number of MPI processes is $NPROCS = 2^n$ ($n \geq 1$), and the number of processes assigned along the $x, y,$

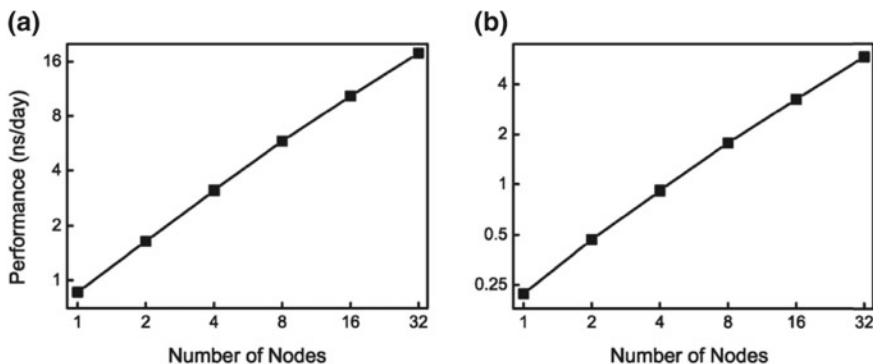


Fig. 5.13 Performance of a MSPA and b STMV systems on K computer

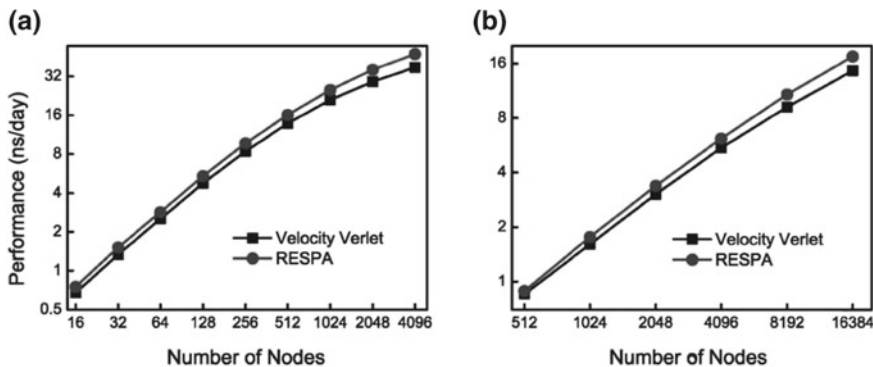


Fig. 5.14 Performance of a STMV and b 27 STMV systems on K computer

and z axis are 2^{n_x} , 2^{n_y} and 2^{n_z} (i.e., $n = n_x + n_y + n_z$). Then, derive an equation to calculate **dest** and **source** for any myrank.

2. **Reduction of array:** Code a program that takes a reduction of array parallelized by threads, without using the reduction clause prepared in OpenMP language extension (i.e., !omp reduction (+:a)).
3. **Performance measurement of vectorized operation:** (1) Specify an option to enable automatic SIMD vectorization, by applying “man” linux command to a compiler. Its name depends on a kind of fortran compiler. (2) Evaluate rate of acceleration by vectorized SIMD operations, comparing elapsed time of a code by vectorized operation with that by normal (non-vectorized) operation, by using “time” linux command. With adding “-O0” option for compilers, completely non-vectorized executable is created. Note that the latest compilers do optimization of a code to some extent at default, and that “-O0” option disables such hidden optimizations.

Appendix: FMM

The principle of the fast multipole method (FMM) is briefly explained in this Appendix. See Ref. [15] for exact mathematical treatment of the transformation of multipole moment and local expansion coefficients, such as M2M, M2L, and L2L.

Fundamentals of Multipole Expansion

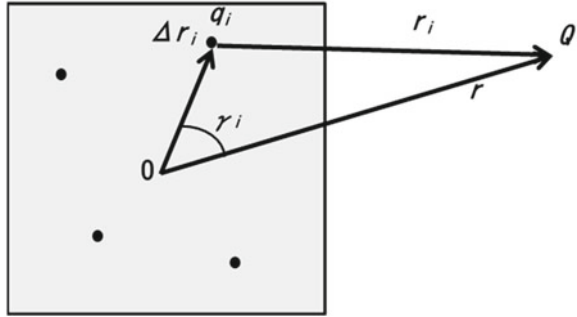
As shown in Fig. 5.15, the atom i of the charge q_i in a certain area forms the potential at a distant point Q . The reciprocal $1/r_i$ of the distance between two points is calculated using the distance Δr_i from the origin O to the atom i in the region, the distance r from the origin O to the point Q , and the angle γ_i formed by the points Δr_i and r with the point O . This can be expressed as

$$\frac{1}{r_i} = \frac{1}{\sqrt{r^2 - 2r\Delta r_i \cos \gamma_i + \Delta r_i^2}} = \frac{1}{r\sqrt{1 - 2\frac{\Delta r_i}{r} \cos \gamma_i + \left(\frac{\Delta r_i}{r}\right)^2}} \quad (5.14)$$

from the cosine theorem. When this equation is rewritten by Taylor expansion, it becomes

$$\begin{aligned} \frac{1}{r_i} &= \frac{1}{r} + \frac{\Delta r_i}{r^2} \cos \gamma_i + \frac{\Delta r_i^2}{2r^3} (3 \cos^2 \gamma_i - 1) + \frac{\Delta r_i^3}{2r^4} (5 \cos^3 \gamma_i - 3 \cos \gamma_i) + \dots \\ &= \frac{1}{r} \sum_n P_n(\cos \gamma_i) \left(\frac{\Delta r_i}{r}\right)^n, \end{aligned} \quad (5.15)$$

Fig. 5.15 Multipole expansion of electrostatic interaction



where $P_n(\cos \gamma_i)$ is a Legendre polynomial. This Eq. (5.15) is called multipole expansion. $P_n(\cos \gamma_i)$ is written as

$$P_n(\cos \gamma_i) = \sum_{m=-n}^n Y_n^{-m}(\theta_i, \phi_i) Y_n^m(\theta, \phi) \quad (5.16)$$

using the spherical harmonics, $Y_n^m(\theta, \phi)$ for spherical coordinates $(\Delta r_i, \theta_i, \phi_i)$ and (r, θ, ϕ) of the two vectors Δr_i and r . In Eq. (5.16), the coordinate variables of atom i and point Q are completely separated into two factors. Therefore, the sum with respect to the charges in a region can be represented independently of the position of the point Q . The sum

$$M_n^m = \sum_i q_i Y_n^{-m}(\theta_i, \phi_i) \Delta r_i^n \quad (5.17)$$

is the multipole moment formed by the charge in the region. The electrostatic field formed by all charges in the region can be expressed as

$$\sum_i \frac{q_i}{r_i} = \frac{1}{r^{n+1}} \sum_{n=0}^{\infty} \sum_{m=-n}^n M_n^m Y_n^m(\theta, \phi). \quad (5.18)$$

Division of Unit Cells

In FMM, unit cells are divided into small spaces (subcells). As described in the next section, the interaction calculation is performed using these subcells. There are several ways to divide unit cells into subcells; however, an octree structure is commonly used, in which each side of the unit cell is divided into two equal parts and the whole is divided into eight subcells (Fig. 5.16). This division is repeated until the subcell reaches the desired size of small subcells, i.e., level 0 for the unit cell and level 1 after

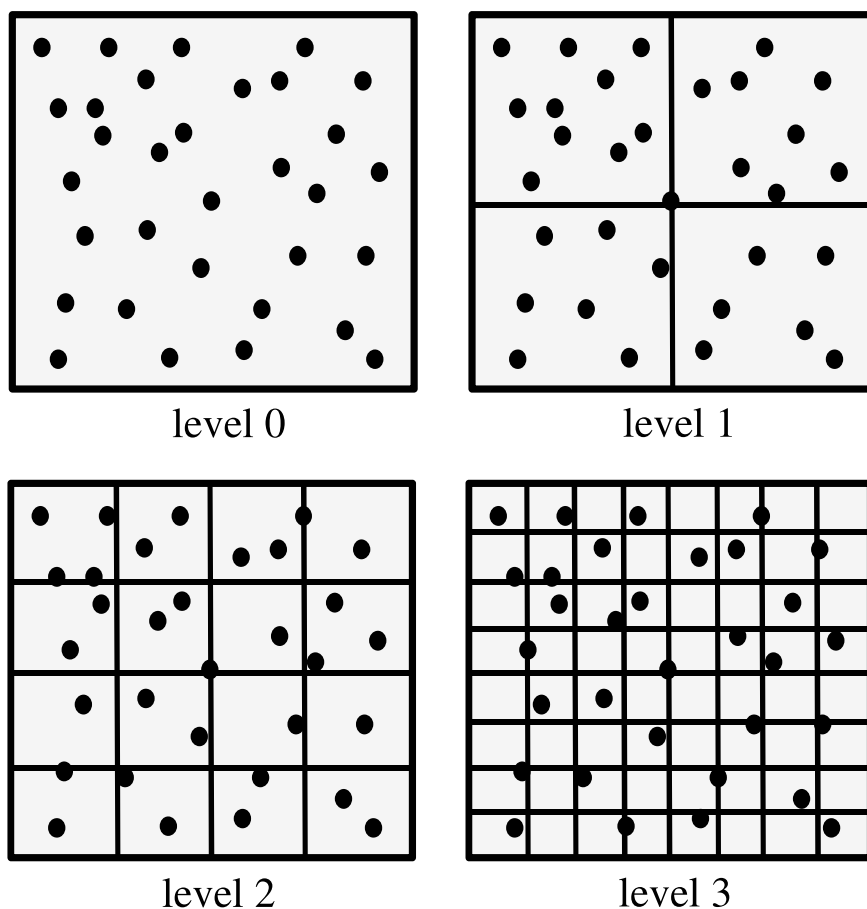


Fig. 5.16 Division of unit cell

dividing once. The number of subcells of level n is 8^n . In practical use, tens of atoms are assigned to the smallest subcell. This smallest subcell is a unit of regional division. We assign subcells to compute nodes and cores to perform parallel calculations.

Interaction Calculation by FMM

We assume that the unit cell is divided to level 4, as shown in Fig. 5.17. We now calculate the potential of an atom in region A. First, we evaluate the electrostatic interaction directly up to the second nearest neighbor (B) of region A of level 4. For

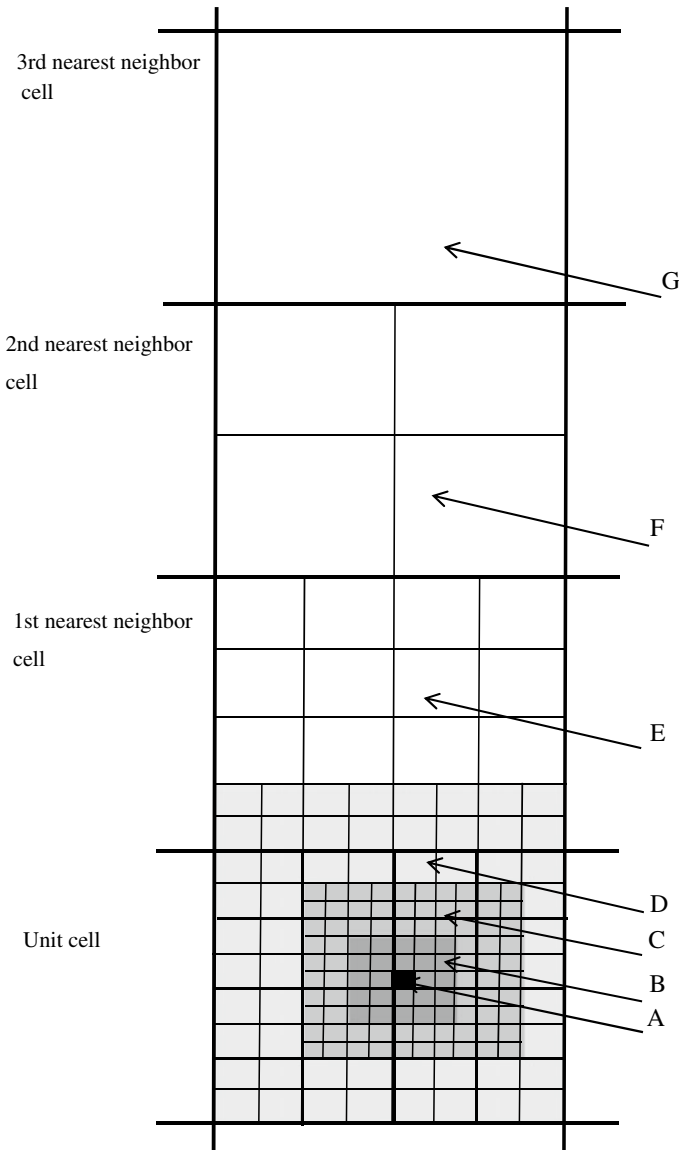


Fig. 5.17 Interactions between atoms and subcell

subcells beyond region B, the interaction calculation is performed using multipole moments.

Region C is within the second nearest neighbor of the subcell of level 3, but excluding region B. The interaction calculation with region C is performed using the multipole moment of level 4. Region D is within the second nearest neighbor

of the subcell of level 2, but excluding region C. The interaction calculation with region B is performed using the multipole moment of level 3. Hereinafter, regions E and F are determined in the same manner. Subcells beyond region F (image cell of third or further nearest neighbor of the unit cell) are treated as those of level 0. Thus, the interactions can be calculated for each region without excess or deficiency. By using small subcells for interactions with neighboring regions and large subcells for interactions with distant regions, it is possible to efficiently perform interaction calculations without decreasing accuracy.

Multipole Expansion and Local Expansion

The procedure for determining the potential acting on the i atom in region A is shown (Fig. 5.18). First, the contributions from regions A and B is obtained by direct calculation. This procedure is called particle to particle (P2P) in FMM. For region C, the multipole moment is calculated in level 4 subcells according to Eq. (5.17). This operation is called particle to multipole (P2M). Next, the center of the multipole moment is moved to the center of subcell A. The expression obtained by this transformation is the form of Taylor expansion at the center of subcell A, which is called local expansion. This shift operation of the expansion center is called multipole to local (M2L). By using the local expansion coefficient and the relative position from the center subcell of atom i , its potential, force, and virial can be calculated, which is called local to particle (L2P). For the further subcell D, the multipole moment of the level 4 subcell is obtained by P2M similarly to C. The expansion center of the multipole moment is shifted to the center position of the level 3 subcell (i.e., multipole to multipole (M2M)). Similarly, with respect to the other seven subcells, the center of expansion is shifted by M2M. The obtained multipole moments are summed to obtain a multipole moment of level 3, which is transformed to the local expansion coefficient of the level 3 subcell, including subcell A, by M2L operation. The local expansion coefficient is shifted to the center of the subcell of level A (i.e., local to local (L2L)). The contributions from the previously obtained subcell C and the subcells D are summed and then the interaction is evaluated by L2P. The contribution from image cells more distant than region F in Fig. 5.17 can be included by applying Ewald's method to the multipole moment of image cells [46].

Accuracy of FMM

FMM is a strictly formulated analytical method that includes error evaluation. The error depends on the expansion order of the multipole moment; i.e., as the order increases, the calculation accuracy monotonically increases. Table 5.1 shows the benchmark results of potential and force in a system consisting of 10,125,000 atoms using MODYLAS software. The accuracy of FMM was evaluated by comparison

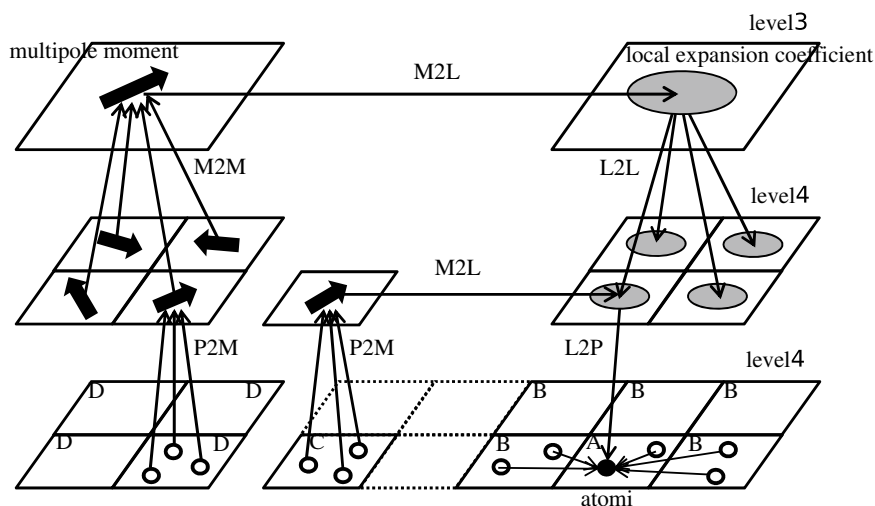


Fig. 5.18 Interaction between atoms and subcell

Table 5.1 Potential and force obtained by FMM

Expansion degree	$V/10^{-13}$ J	$F_x/10^{-10}$ N	$F_y/10^{-10}$ N	$F_z/10^{-10}$ N
Exact	-218,735,559	-4.357177	6.380146	-1.477348
4th	-218,735,742	-4.356659	6.380802	-1.475152
8th	-218,735,559	-4.357201	6.380145	-1.477340

with the result of the PME method with parameters giving a sufficiently high accuracy. The values obtained at the 4th to 8th expansion order coincide with the exact value in the range of 4–9 digits. There is also an accuracy of 4–7 digits for force; i.e., enough accuracy can be obtained for MD calculations even in expansions up to the 4th order.

References

1. W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz Jr., D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell, P.A. Kollman, *J. Am. Chem. Soc.* **117**, 51795197 (1995)
2. A.D. MacKerell Jr., M. Feig, C.L. Brooks III, *J. Comput. Chem.* **25**, 14001415 (2004)
3. W.L. Jorgensen, D.S. Maxwell, J. Tirado-Rives, *J. Am. Chem. Soc.* **118**, 1122511236 (1996)
4. H.J.C. Berendsen, D. van der Spoel, R. van Drunen, *Comput. Phys. Commun.* **91**, 43–56 (1995)
5. J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, *J. Comput. Chem.* **26**, 1781–1802 (2005)
6. S. Plimpton, *J. Comput. Phys.* **117**, 1–19 (1995)
7. Y. Andoh et al., *J. Chem. Theory Comput.* **9**, 3201 (2013)
8. J. Jung, T. Mori, C. Kobayashi, Y. Matsunaga, T. Yoda, M. Feig, Y. Sugita, *WIREs Comput. Mol. Sci.* **5**, 310 (2015)
9. D. Frenkel, B. Smit, *Understanding Molecular Simulation* (Academic Press, San Diego, 1996)

10. M.E. Tuckerman, *Statistical Mechanics: Theory and Molecular Simulation* (Oxford University Press, New York, 2010)
11. M.P. Allen, D.J. Tildesley, *Computer Simulation of Liquids* (Oxford University Press, 1989)
12. J. Wong-ekkabut, M. Karttunen, *Biochim. Biophys. Acta* **1858**, 2529 (2016)
13. T. Darden, D. York, L. Pedersen, *J. Chem. Phys.* **98**, 10089 (1993)
14. U. Essmann, L. Perera, M.L. Berkowitz, T. Darden, H. Lee, L.G. Pedersen, *J. Chem. Phys.* **103**, 8577 (1995)
15. L. Greengard, V. Rokhlin, *J. Comput. Phys.* **73**, 325 (1987)
16. D.J. Hardy, Z. Wu, J.C. Phillips, J.E. Stone, R.D. Skeel, K. Schulten, *J. Chem. Theory Comput.* **11**, 766 (2015)
17. J.J. Biesiadecki, R.D. Skeel, *J. Comput. Phys.* **109**, 318–328 (1993)
18. P. Minari, M.E. Tuckerman, G.J. Martyna, *Phys. Rev. Lett.* **93**, 150201 (2004)
19. J.P. Ryckaert, G. Ciccotti, H.J.C. Berendsen, *J. Comput. Phys.* **23**, 327341 (1977)
20. P. Gonnet, *J. Comput. Phys.* **220**, 740 (2007)
21. H.C. Andersen, *J. Comput. Phys.* **52**, 2434 (1983)
22. B. Hess, H. Bekker, H.J.C. Berendsen, J.G.E.M. Fraaije, *J. Comput. Chem.* **18**, 14631472 (1997)
23. S. Miyamoto, P.A. Kollman, *J. Comput. Chem.* **13**, 952962 (1992)
24. H. Okumura, S.G. Itoh, Y. Okamoto, *J. Chem. Phys.* **126**, 084103 (2007)
25. K.J. Bowers et al., in *ACM/IEEE Conference on Supercomputing (SC06)* (IEEE, Tampa, Florida, 2006)
26. K.J. Bowers, R.O. Dror, D.E. Shaw, *J. Chem. Phys.* **124**, 184109 (2006)
27. B.G. Fitch, A. Rayshubskiy, M. Eleftheriou, T.J.C. Ward, M. Giampapa, M.C. Pitman, J. Pitera, W.C. Swope, R.S. Germain, *Comput. Model. Membr. Bilayers* **60**, 159 (2008)
28. J. Jung, C. Kobayashi, T. Imamura, Y. Sugita, *Comput. Phys. Commun.* **200**, 57 (2016)
29. A.P. Hynninen, M.F. Crowley, *J. Comput. Chem.* **35**, 406 (2014)
30. B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, *J. Chem. Theory Comput.* **4**, 435 (2008)
31. T. Narumi et al., in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (ACM, 2006), p. 49
32. D.E. Shaw et al., in *34th Annual International Symposium on Computer Architecture (ISCA '07)* (ACM, 2007), p. 910
33. D.E. Shaw et al., in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Press, 2014), p. 41
34. R. Salomon-Ferrer, A.W. Gotz, D. Poole, S. Le Grand, R.C. Walker, *J. Chem. Theory Comput.* **9**, 3878 (2013)
35. M.J. Harvey, G. Giupponi, G. De Fabritiis, *J. Chem. Theory Comput.* **5**, 1632 (2009)
36. P. Eastman et al., *J. Chem. Theory Comput.* **9**, 461 (2013)
37. A.P. Ruymgaart, A.E. Cardenas, R. Elber, *J. Chem. Theory Comput.* **7**, 3072 (2011)
38. J.C. Phillips, Y. Sun, N. Jain, E.J. Bohm, L.V. Kal, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Press, 2014), p. 81
39. M.J. Abraham, T. Murtola, R. Schulz, S. Pali, J.C. Smith, B. Hess, E. Lindahl, *Softw. X* **1–2**, 19 (2015)
40. Jung et al., *J. Chem. Theory Comput.* **12**, 4947 (2016)
41. W.J. Harrod, A journey to exascale computing, in *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC12* (2012), https://science.energy.gov/~media/ascr/ascac/pdf/reports/2013/SC12_Harrod.pdf
42. <https://software.intel.com/en-us/articles/explicit-vector-programming-in-fortran>
43. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
44. Y. Andoh, S. Suzuki, S. Ohshima, T. Sakashita, M. Ogino, T. Katagiri, N. Yoshii, S. Okazaki, *J. Supercomput.* **74**, 2449 (2018)
45. J. Jung, T. Mori, Y. Sugita, *J. Comput. Chem.* **34**, 2412 (2013)
46. T. Amisaki, *J. Comput. Chem.* **21**, 10751087 (2000)

Chapter 6

Large-Scale Quantum Chemical Calculation



Kazuya Ishimura and Masato Kobayashi

Abstract Quantum chemical calculations give electronic structures, energies, geometries, and properties of molecular systems, and then their reactivities and functions are analyzed and predicted. Approximations, acceleration, parallelization, and the features of calculation methods from the viewpoint of computer use are introduced. In addition, the application of quantum chemical methods to huge systems based on the fragmentation methods will be overviewed.

Quantum chemical calculations give electronic structures, energies, geometries, and properties of molecular systems, and then their reactivities and functions are analyzed and predicted. They are widely applied to the analysis and control of chemical reactions, the elucidation of the functions for chemical substances and catalysts, the design of new materials, and drug developments. With the development of theories and computational methods, and the performance improvement of computers, it is becoming possible to calculate nano-sized molecules using supercomputers. By enriching software including visualization, not only theoretical and computational researchers, but experimental researchers can also easily perform quantum chemical calculations. In this chapter, approximations, acceleration, parallelization, and the features of calculation methods from the viewpoint of computer use are introduced. In addition, the application of quantum chemical methods to huge systems based on the fragmentation methods will be overviewed.

K. Ishimura
Department of Theoretical and Computational Molecular Science,
Institute for Molecular Science, Okazaki, Japan
e-mail: ishimura.smash@gmail.com

M. Kobayashi (✉)
Faculty of Science, Hokkaido University, Sapporo, Japan
e-mail: k-masato@sci.hokudai.ac.jp

© Springer Nature Singapore Pte Ltd. 2019
M. Geshi (ed.), *The Art of High Performance Computing
for Computational Science*, Vol. 2,
https://doi.org/10.1007/978-981-13-9802-5_6

6.1 Purpose of Quantum Chemical Calculation

The quantum chemistry is a research field, in which, as its name suggests, the quantum mechanics that governs the material science is applied to the chemical phenomena. The targets of the material science are molecules, molecular clusters, and solids, which consist of atomic nuclei and electrons. Their quantum behavior determines all the physical properties of the material, although the electrons are only treated in the quantum mechanical manner usually because the mass of each nucleus is more than thousands times larger than that of the electron. The purpose of the quantum chemical calculation is to numerically obtain the quantum behavior of electrons.

The first-principles calculation, or *ab initio* calculation is a method to evaluate molecular energy, electron distribution, and the other properties only from the nuclear coordinates and the electron number as the input. Additionally, the computational method and the basis set (see below) may have to be specified explicitly because various levels of approximate computational methods are implemented in the actual computational package. Furthermore, the evaluations of energy derivatives with respect to various fields are frequently performed because a physical property obtained from an actual measurement is the response to a field or the energy balance. Especially, the derivative calculation with respect to the nuclear coordinates is routinely performed to obtain the equilibrium and transition-state structures of the molecule. In this chapter, however, we only focus on the energy calculation of the ground state.

6.2 Quantum Chemical Calculation Software Packages

The software packages to perform the quantum chemical calculation are available in many formats. Typical software packages are summarized in Table 6.1. Gaussian is the most popular package and is commonly used even by experimental researchers, of which several handbooks are also available [1]. NTCHEM is developed for the use with the K Computer, and is available in the computer resources in Research Center for Computational Science (RCCS) of the National Institute of Natural Sciences, Japan (former Computer Center, Institute for Molecular Science, Japan) and in Foundation for Computational Science (FOCUS), Japan, as well as in the K Computer. Although CONQUEST and OpenMX are not usually regarded as the quantum chemical calculation packages, they share several similarities with the quantum chemical calculation such as the use of atom-centered basis functions. The purpose of so-called “first-principles calculation software” is almost the same except mainly using plane-wave basis functions, although not listed in the table.

Here, we show the input file of SMASH program, which is the open-source quantum chemical package developed by one of the authors, Ishimura (Fig. 6.1). The Cartesian coordinates of nuclei are given below `geom`. By putting nuclei as specified, the structure of bent water molecule appears on the *yz* plane. Many programs


```

job runtime=energy method=B3LYP basis=cc-pVDZ
geom
O  0.000   0.000   0.122
H  0.000   0.793  -0.487
H  0.000  -0.793  -0.487

```

Fig. 6.1 Input file of the SMASH program for water molecule

also accept the internal coordinate input known as the Z-matrix instead of the Cartesian coordinate input. The electron number, which is the other required information, is usually set automatically for the neutral and lowest spin multiplicity molecule. At the first line started from `job`, the calculation method (DFT calculation with B3LYP functional), the basis set (cc-pVDZ), and the type of the calculation (energy only calculation) are specified.

6.3 Computer Environment in the Field of Molecular Science

Because the computational cost for quantum chemical calculations increases in proportion to more than the cubic of the number of atoms, which depends on the computational method, supercomputers have been utilized in this field for more than 40 years. Table 6.2 shows the history of the computer performance in RCCS [2]. Since the first introduction in 1979, they have been regularly updated, and the performance continues to improve at a pace of several hundred times in 10 years. The peak performance in 2000 is 515 GFLOPS, which is equivalent to 1 PC in 2019. It is now possible to secure the computing capacity of supercomputers 20 years ago in a laboratory. The peak performance in 2015 is 492 TFLOPS, which is equivalent to about 1/20 of the K Computer (10.6 PFLOPS). The total number of CPU cores is more than 15,000, and thus parallel computing is necessary for taking advantage of performance. If the performance improvement of computers continues, it is predicted that computers with the performance of several tenths of the K computer are available in a laboratory in the late 2020s and the number of cores is enormous. Therefore, the

Table 6.1 Typical quantum chemical calculation software packages

Commercial		Gaussian16, Q-Chem, MOLPRO, Molcas, Turbomole, ADF
Free	Provided binary only	NTChem, ORCA, Firefly, CONQUEST
	Source code available	GAMESS, DIRAC, SIESTA
	Open source	NWChem, ACES III, SMASH, OpenMX

Table 6.2 History of computer performance in Research Center for Computational Science (RCCS), National Institutes of Natural Sciences [2]

Year	Machine	Peak performance (GFLOPS)
1979	HITACHI M-180 (2 machines)	0.036
1988	HITACHI M-680H	0.016
	HITACHI S-820/80	2
	Total	2
2000	IBM SP2 (Wide 24 machines)	7
	IBM SP2 (Thin 24 machines)	3
	NEC SX-5 (8 CPUs)	64
	Fujitsu VPP5000 (30 PEs)	288
	SGI SGI2800 (256 CPUs)	153
	Total	515
2015	Fujitsu PRIMERGY RX300S7 (5472 cores)	126,950
	(+ NVIDIA Tesla M2090 32 boards)	21,280
	Fujitsu PRIMEHPC FX10 (1536 cores)	20,152
	SGI UV2000 (1024 cores)	21,299
	Fujitsu PRIMERGY CX2550M1 (7280 cores)	302,848
	Total	492,530
(Ref.)	Earth Simulator (2002)	35,860
	K Computer (2011)	10,510,000

effort to master the use of current supercomputers has become the foundation for future daily researches.

6.4 Types of Quantum Chemical Calculations

There are various types of quantum chemical calculation methods. Here, the typical calculation methods and their computational bottlenecks are briefly summarized.

The Hartree–Fock (HF) method is the nonempirical mean-field theory of the electronic ground state, which is the basic of the quantum chemical calculation. The computational bottlenecks of the HF calculation is the computation of the electron repulsion integrals (ERIs) and the diagonalization of dense symmetric matrix.

Currently, the HF method is hardly adopted as the final calculation method. The Kohn–Sham (KS) density functional theory (DFT), which is formally almost equivalent to the HF method and usually yields more accurate results than the HF method, is the current mainstream method. However, compared to the post-HF methods described below, which offer the systematic hierarchy to improve the accuracy, the current DFT has no way for certain and systematic improvement. For the theoretical aspects of DFT, please refer to the Ref. [3]. In the DFT calculation, the spatial

numerical integration of the density functional is required in addition to the ERI computation and matrix diagonalization.

There is another class of calculations known as the post-HF methods that provide more accurate results than the HF and possibly DFT. In these methods, the energy correction by considering the explicit correlation between electrons (called correlation energy), which is not included in the HF wave function, is evaluated after the HF calculation. The procedure for these methods is completely different from that of the HF or DFT calculation. Specifically, the main computational processes of the Møller–Plesset perturbation (MP) and coupled cluster (CC) methods are the integral transformation introduced below and the tensor contractions. In the configuration interaction (CI) method, the diagonalization of huge sparse matrix is required in addition to the integral transformation. For the theoretical details of the individual post-HF methods, please refer to the textbook of the quantum chemistry such as [4].

6.4.1 Theory of Hartree–Fock Method

At first, the theoretical aspects of the HF method, the basic of the quantum chemistry, will be overviewed briefly.¹ The motion of N_e electrons in a molecule is governed by the Schrödinger equation, the fundamental equation of the quantum mechanics:

$$\hat{H}\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_{N_e}) = E_e\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_{N_e}). \quad (6.1)$$

This is an eigenvalue equation, where the electronic energy E_e and the N_e -electron wave function $\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_{N_e})$ are determined from the Hamiltonian \hat{H} that is the energy operator. The electronic Hamiltonian of an isolated molecule is given by²

$$\hat{H} = -\frac{1}{2} \sum_{i=1}^{N_e} \nabla_i^2 - \sum_{i=1}^{N_e} \sum_{I=1}^{N_n} \frac{Z_I}{|\vec{r}_i - \vec{R}_I|} + \sum_{i=1}^{N_e} \sum_{j=1}^{i-1} \frac{1}{|\vec{r}_i - \vec{r}_j|}. \quad (6.2)$$

From this equation, it can be seen that the external parameters independent of the wave function Ψ are only the coordinates $\{\vec{R}_I\}$ and charges $\{Z_I\}$ of the N_n nuclei.

It is difficult to obtain the wave function Ψ by directly solving Eq. (6.1) because it includes the coordinates of N_e electrons. Then, the HF approximation is introduced, where each electron moves independently in the mean field (\hat{H}_{eff}) created by N_e electrons. The N_e -electron wave function Ψ is approximated by the product of N_e

¹In this chapter, the description concerning spin is omitted, and only the formulas for the closed-shell system are given. There are descriptions that can be pointed out to be partially incorrect, e.g., the coordinates of electrons must include the spin coordinate. Their details are given in the other textbook [5].

²In this chapter, the atomic unit system is used without notice.

one-electron wave functions $\{\psi_p(\vec{r})\}$, each of which is called the molecular orbital (MO),

$$\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_{N_e}) \approx \frac{1}{\sqrt{N_e!}} |\psi_1(\vec{r}_1)\psi_2(\vec{r}_2) \cdots \psi_{N_e}(\vec{r}_{N_e})|, \quad (6.3)$$

and the MOs $\{\psi_p(\vec{r})\}$ are determined by solving the following equation:

$$\hat{H}_{\text{eff}}\psi_p(\vec{r}) = \varepsilon_p\psi_p(\vec{r}). \quad (6.4)$$

$|\cdot|$ on the right-hand side of Eq. (6.3) denotes a determinant (Slater determinant), which expresses the behavior of electrons as the Fermi particle obeying the Pauli's exclusion principle. Because the effective Hamiltonian \hat{H}_{eff} depends on N_e MOs $\{\psi_i(\vec{r})\}$ occupied by electrons, it has to be solved by an iterative method. Furthermore, instead of analytically solving MOs, the Roothaan method, where the MOs are expanded by given functions $\{\phi_\mu(\vec{r})\}$ (called basis functions), is generally adopted:

$$\psi_p(\vec{r}) = \sum_{\mu} C_{\mu p}\phi_{\mu}(\vec{r}). \quad (6.5)$$

It replaces Eq. (6.4) to the following algebraic equation:

$$FC = SC\varepsilon \quad (6.6)$$

$F_{\mu\nu} = \int d\vec{r} \phi_{\mu}(\vec{r})\hat{H}_{\text{eff}}\phi_{\nu}(\vec{r})$ is the matrix corresponding to the effective Hamiltonian (called the Fock matrix) and $S_{\mu\nu} = \int d\vec{r} \phi_{\mu}(\vec{r})\phi_{\nu}(\vec{r})$ is the overlap integral matrix of the basis functions, which appears when adopting non-orthogonal basis functions. In summary, the HF calculation is the method to obtain MOs from the nuclear coordinates and given basis functions.

6.4.2 Basis Functions of Quantum Chemical Calculation

As the basis functions, uniform systematic functions such as finite element functions or plane waves are first thought. Actually, in the first-principles calculation program of solid-state physics, these functions are adopted in many cases. However, in many quantum chemical calculation programs, the functions localized at one atom are adopted as the basis functions. These functions are occasionally called the atomic orbitals (AOs). For the same chemical elements, the same set of AOs (basis set) different only in the central coordinates is often assigned. This not only coincides with the chemist's intuitive picture that "a molecule is a collection of atoms", but also contributes to a drastic reduction in the number of basis functions. In the case of the hydrogen molecule H_2 as an extreme example, it is possible to obtain a qualitatively

and symmetrically correct wave function only with two basis functions, each of which is located on each H atom.

The first thing that comes up as an AO is the Slater type function, the radial function of which is expressed as $\exp(-\zeta|\vec{r} - \vec{R}|)$ because it is the exact wave function of the hydrogen atom. However, since the computation of the ERIs to be introduced later becomes extremely demanding with the Slater type functions, the program adopting the Slater type functions is limited to only a few packages such as ADF. The standard AO used in the quantum chemistry is the Gauss-type function, the radial function of which is expressed as $\exp(-\alpha|\vec{r} - \vec{R}|^2)$. Although it does not look much different from the Slater type function, the computation of the integrals can be simplified by using the fact that the product of the Gauss-type functions become another Gauss-type function. Various types of basis sets are implemented in the packages, one of which may be specified in the calculation depending on the required accuracy and application. You can also download them from the website below:

<https://www.basissetexchange.org/>

<http://sapporo.center.ims.ac.jp/sapporo/>

In this chapter, we will consider quantum chemical calculations with the Gauss-type basis functions.

6.4.3 Procedure of Quantum Chemical Calculation

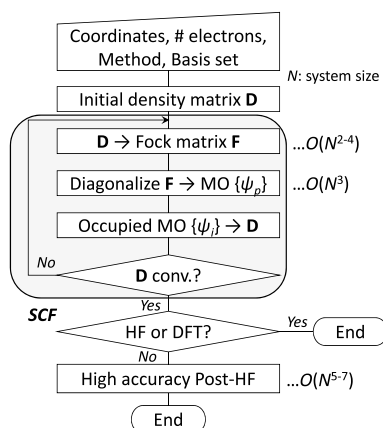
The procedure of the quantum chemical calculation is summarized in Fig. 6.2. At first, the initial guess of the electron density (density matrix \mathbf{D}) is obtained with an easy method such as a semiempirical MO calculation. Using the initial \mathbf{D} matrix, the Fock matrix³ \mathbf{F} is constructed, where the computational bottlenecks of the ERI calculation and the spatial numerical integration (in DFT case) is included. Next, the other bottleneck of the diagonalization of \mathbf{F} is performed to obtain MOs $\{\psi_p\}$ of Eq. (6.5). The density matrix \mathbf{D} is evaluated with the occupied MOs as:

$$D_{\mu\nu} = 2 \sum_i^{\text{occ}} C_{\mu i} C_{\nu i}. \quad (6.7)$$

The process to iterate abovementioned procedure until \mathbf{D} converges is called the self-consistent field (SCF) calculation. In the case of HF or DFT calculation, the computation is completed when the SCF converges. In contrast, when performing highly accurate post-HF calculation, the largest computational bottleneck exists after SCF.

³In DFT, it is called the KS Hamiltonian matrix.

Fig. 6.2 Procedure of quantum chemical calculation



In the following sections, the computational details of each method will be explained, focusing on these bottlenecks, and their rapid computational algorithms will be illustrated.

6.5 Components of SCF Calculation

6.5.1 Fock Matrix Construction

The Fock matrix F of the HF method is expressed as the sum of the D -independent term, H^{core} , and D -dependent two-electron term, G , where D is the density matrix expressing the electron density:

$$F = H^{\text{core}} + G. \quad (6.8)$$

The total energy is given in the HF method as

$$E = \frac{1}{2} \text{Tr}[D(H^{\text{core}} + F)] + \sum_{A < B}^{N_n} \frac{Z_A Z_B}{|\vec{R}_A - \vec{R}_B|}. \quad (6.9)$$

H^{core} is normally stored on the memory during the SCF calculation. The explicit form of G is expressed as

$$G_{\mu\nu} = \sum_{\lambda\sigma}^{\text{AO}} D_{\lambda\sigma} [2\Gamma_{\mu\nu,\sigma\lambda} - \Gamma_{\mu\lambda,\sigma\nu}]. \quad (6.10)$$

The first term is derived from the classical electrostatic repulsion among electrons and is known as the Coulomb term J . The second term is derived from the Pauli's exclusion principle and is known as the exchange term K .

$$\Gamma_{\mu\nu,\sigma\lambda} = \iint d\vec{r}_1 d\vec{r}_2 \phi_\mu(\vec{r}_1) \phi_\nu(\vec{r}_1) |\vec{r}_1 - \vec{r}_2|^{-1} \phi_\sigma(\vec{r}_2) \phi_\lambda(\vec{r}_2) \quad (6.11)$$

is the rank-4 tensor known as the two-electron integral or the electron repulsion integral, and has the following high symmetry:

$$\begin{aligned} \Gamma_{\mu\nu,\lambda\sigma} &= \Gamma_{\mu\nu,\sigma\lambda} = \Gamma_{\nu\mu,\lambda\sigma} = \Gamma_{\nu\mu,\sigma\lambda} \\ &= \Gamma_{\lambda\sigma,\mu\nu} = \Gamma_{\sigma\lambda,\mu\nu} = \Gamma_{\lambda\sigma,\nu\mu} = \Gamma_{\sigma\lambda,\nu\mu}, \end{aligned} \quad (6.12)$$

when using the real basis functions. This symmetry can reduce the number of integrals to be stored to $O(N^4/8)$. However, let us think about a medium-size example where the number of basis functions N is 1000 and 10% of them are nonzero. In this case, the scratch volume required to store the integral value increases to approximately 200 GB.⁴ Therefore, it immediately becomes difficult to store them in the memory as the size of the system increases. There are the following two directions for the treatment of ERIs:

- Conventional SCF
To store ERIs on the external storage (disk).
- Direct SCF
To calculate ERIs every time and discard them.

Due to the advances in the computer performance, the computation of the integrals now can be more efficient in the actual computational time than the excessive disk access for reading them. Recently, the direct SCF algorithm or its effective combination with the memory-based scheme, called the semi in-core algorithm, is commonly used.

Figure 6.3 shows the pseudocode for the Fock matrix construction with the direct SCF algorithm. The AO indices are processed with the unit composed of several basis functions of the same center, which is called the shell. The number of AO shells $\mu, \nu, \lambda, \sigma$, running at the loop of Lines 2–16, is expected to be more than 100, or even higher than 1000 for large molecules, for each index. The number of shell quartets increases to tens of millions or more even if the integral symmetry described above is used. Then, this procedure can be parallelized with the MPI/OpenMP hybrid, though the MPI parallelization is only used in the Figure, and the inside of this loop may not be parallelized.

The ERIs Γ is computed at Line 5, which is a unique part in quantum chemical calculations and will be explained in detail later. The loop of Lines 6–13 is for the integrals in the shell quartet, and the loop length of each index is fairly short (approximately 1–10). At Lines 7–12, Fock matrix elements are incremented utilizing

⁴Normally, 64-bit variables are used respectively to store the integral value and the index set ($\mu\nu\lambda\sigma$).

```

1: F ← 0
2: Loop over μνλσsh (μsh ≥ νsh, λsh ≥ σsh, [μν]sh ≥ [λσ]sh)
3:   If (MOD(μνλσsh, NPROC) == MYRANK) Then
4:     If (Estimate_Magnitude(μνλσsh, D) >= thresh) Then
5:       Call ConstructGamma(gamma, μνλσsh)
6:       Loop over μνλσ
7:         F(μν) ← F(μν) + 4*D(λσ)*gamma(μνλσ)
8:         F(λσ) ← F(λσ) + 4*D(μν)*gamma(μνλσ)
9:         F(μλ) ← F(μλ) - D(νσ)*gamma(μνλσ)
10:        F(μσ) ← F(μσ) - D(νλ)*gamma(μνλσ)
11:        F(νλ) ← F(νλ) - D(μσ)*gamma(μνλσ)
12:        F(νσ) ← F(νσ) - D(μλ)*gamma(μνλσ)
13:       End Loop
14:     End If
15:   End If
16: End Loop
17: Call MPI_AllReduce(F)
18: F ← F + HCORE

```

Fig. 6.3 Pseudocode for the Fock matrix construction

the abovementioned symmetry of the integral. Due to this symmetry usage, the access to F is not sequential by this algorithm. When utilizing the reduction clause of the OpenMP parallelization, different arrays for F are prepared for different threads and they are finally gathered.

At Line 4, the density matrix is used to screen the integral set as well as the AO set of $\{\mu\nu\lambda\sigma\}$. Specifically,

$$|\Gamma_{\mu\nu,\lambda\sigma}| \leq \sqrt{\Gamma_{\mu\nu,\mu\nu}} \cdot \sqrt{\Gamma_{\lambda\sigma,\lambda\sigma}}, \quad (6.13)$$

derived from the Cauchy–Schwarz inequality, is used to evaluate the upper limit of the 4-index integral value by the product of two 2-index integral values computed in advance. Furthermore, the maximum absolute value of the density matrix elements that are multiplied by $\Gamma_{\mu\nu,\lambda\sigma}$ when evaluating the energy,

$$D_{\max} = \text{Max}(4|D_{\mu\nu}|, 4|D_{\lambda\sigma}|, |D_{\mu\lambda}|, |D_{\mu\sigma}|, |D_{\nu\lambda}|, |D_{\nu\sigma}|), \quad (6.14)$$

is used and the ERIs that are expected to have the contribution more than a certain threshold $\varepsilon_{\text{thresh}}$ to the energy

$$D_{\max} \sqrt{\Gamma_{\mu\nu,\mu\nu}} \cdot \sqrt{\Gamma_{\lambda\sigma,\lambda\sigma}} \geq \varepsilon_{\text{thresh}} \quad (6.15)$$

are only computed. In the actual implementation, a set of integrals are evaluated at once by processing with the shell indices. Furthermore, because F is linear with respect to D , this screening can also be adopted to the difference from the previous cycle, ΔD , if the previous F is stored elsewhere (except for the exchange–correlation term in DFT calculations). As the magnitude of ΔD decreases rapidly as the convergence approaches, this screening becomes more effective as the SCF calculation proceeds.

Even if this screening is adopted, the ideal order of the computational cost is $O(N^2)$ because the ERI evaluation is performed when both of $\Gamma_{\mu\nu,\mu\nu}$ and $\Gamma_{\lambda\sigma,\lambda\sigma}$ are large. Many programs equip the reduced-order computational code that, for example, adopts the fast multipole method (FMM), used in the classical MD calculations, for the potential derived by the continuous electron cloud. The implementation of the FMM in the classical calculation is reviewed in detail in an appendix of Chap. 5 of this book. Here, we just cite important articles for the extension to the continuous electrons [6, 7]. Although this method cannot be used directly to the exchange term appeared only in the quantum chemical calculations, approximate $O(N)$ calculation is possible by adopting the cut-off technique, because the exchange term decays exponentially with respect to the distance for insulators [8].

6.5.2 Generalized Eigenproblem

The second component of the SCF calculation is the solution of the generalized eigenproblem (Eq. (6.6)), in other words, the diagonalization. The typical size of the eigenproblem in the SCF calculation is not so large, i.e., thousands by thousands. However, because the number of the roots to construct the density matrix (=the number of occupied orbitals) is comparable in magnitude with the number of the basis functions, the algorithm to obtain all the roots will be usually selected. The iron rule to do this is to use the existing linear algebra library.

When using non-orthogonal basis functions, the linear dependency of the basis functions often becomes a problem. To avoid this problem, it is typically used to cut-off the linearly dependent components by solving the eigenproblem of the overlap matrix S ,

$$SU = Us. \quad (6.16)$$

Then, it is possible to orthogonalize the basis functions using $X = Us^{-1/2}$. When constructing this matrix, it is possible to avoid the linear dependence by eliminating the components with small eigenvalues s . Because X does not depend on the density and MOs, it is possible to construct before SCF calculation. For the matrix orthogonalized by X , $F' = X^T F X$, the standard eigenproblem is solved, ($F' C' = C' \epsilon$). Then, the matrix is backtransformed as $C = X C'$.

6.5.3 Spatial Numerical Integration of Density Functional

The third component of the SCF calculation appears only in the DFT calculation, that is, the integration of the density functional. The DFT energy by the KS method is given by,

$$E = \frac{1}{2} \text{Tr} [\mathbf{D}(2\mathbf{H}^{\text{core}} + 2\mathbf{J})] + E_{\text{xc}}[\rho(\vec{r})] + \sum_{A < B}^{N_n} \frac{Z_A Z_B}{|\vec{R}_A - \vec{R}_B|}. \quad (6.17)$$

Comparing with the HF energy equation (6.9), the functional of the density known as the exchange-correlation functional, E_{xc} , appears instead of the exchange term of \mathbf{K} . KS Hamiltonian is expressed as,

$$\mathbf{H}^{\text{KS}} = \mathbf{H}^{\text{core}} + 2\mathbf{J} + \mathbf{V}_{\text{xc}}, \quad (6.18)$$

where

$$(\mathbf{V}_{\text{xc}})_{\mu\nu} = \int d\vec{r} \phi_{\mu}(\vec{r}) \frac{\delta E_{\text{xc}}}{\delta \rho} \phi_{\nu}(\vec{r}). \quad (6.19)$$

Because E_{xc} is nonlinear with respect to the density, its explicit differential form is required. E_{xc} and \mathbf{V}_{xc} must be evaluated by three-dimensional spatial numerical integration.

In quantum chemical calculations, the atom-centered polar coordinate grid is typically used [9]. This utilizes the fact that the electrons are dense around nuclei and its density decreases with respect to the distance from the nucleus. Due to the atom-centered grid, the overlap problem cannot be avoided. To avoid the overlapped counting, the partition function $p(\vec{r})$ is introduced,

$$\int d\vec{r} V_{\text{xc}}(\vec{r}) \approx \sum_A^{\text{atom}} \sum_g^{\text{grid}} \omega_g p_A(\vec{r}_g) V_{\text{xc}}(\vec{r}_g). \quad (6.20)$$

ω_g represents the weight of the grid g , which is systematically defined to each quadrature point of each atom, whereas the partition function $p(\vec{r})$ is determined so that it is normalized for each grid point,

$$\sum_A^{\text{atom}} p_A(\vec{r}_g) = 1, \quad (6.21)$$

and the weight becomes larger for closer atom. For partitioning, a gentle function such as the Fermi function is used instead of the step function. The increase in the number of grid points is $O(N)$, but if there is no ingenuity in calculating the electron density, the cost increases by $O(N^3)$. Parallelization of this procedure is easy because the number of grid points are tens of thousands for each atom.

6.5.4 Reduction of SCF Cycles by DIIS Method

Because the SCF calculation is an iterative procedure, it is important to remember that the computational time is the one-cycle time \times the number of iterations. Therefore, although it is not essential for SCF calculation, we describe the DIIS (direct inversion in the iterative subspace) method [10, 11] as one of the methods of reducing the number of iteration, or a method to improve the convergence.

In the DIIS method, the Fock matrix is predicted as the linear combination of several prior Fock matrices,

$$\mathbf{F}^{n,\text{DIIS}} = \sum_{i=1}^n x_i \mathbf{F}^i \left(\sum_{i=1}^n x_i = 1 \right). \quad (6.22)$$

The condition in the bracket is derived from the fact that the number of electrons to be considered is constant. To obtain the appropriate coefficients of the linear combination $\{x_i\}$, the error vector \mathbf{e} that represents the distance of each \mathbf{F} from its exact solution is introduced.

There are several definitions of the error vector. If the error can be linearly approximated around the convergence, the following error vector can be considered:

$$\mathbf{e}^i = \mathbf{F}^i - \mathbf{F}^{i-1}. \quad (6.23)$$

This definition can be easily applicable to general nonlinear optimization problem, which is also known as the Anderson mixing. As the characteristic definition for the SCF calculation,

$$\mathbf{e}^i = \mathbf{F}^i \mathbf{D}^i \mathbf{S} - \mathbf{S} \mathbf{D}^i \mathbf{F}^i \quad (6.24)$$

is usually more useful, which utilizes the fact that \mathbf{F} commutes with \mathbf{S} at the converged solution.

$\{x_i\}$ will be determined to reduce the error by the least square method with the Lagrange multiplier constraint, i.e., for $B_{ij} = \mathbf{e}^{iT} \cdot \mathbf{e}^j$, the following linear system of equations will be solved:

$$\begin{pmatrix} 0 & -1 & -1 & \cdots & -1 \\ -1 & B_{11} & B_{12} & \cdots & B_{1n} \\ \vdots & & & \ddots & \vdots \\ -1 & B_{n1} & B_{n2} & \cdots & B_{nn} \end{pmatrix} \begin{pmatrix} -\lambda \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \quad (6.25)$$

This is the DIIS method.

6.6 Components of Post-HF Calculation

6.6.1 Component of the MP2 Method: Integral Transformation

Let us introduce the bottlenecks of the more accurate post-HF calculations. The simplest post-HF calculation is the second-order MP (MP2) method, of which the correlation energy is given by:

$$\Delta E_{\text{MP2}} = \sum_{ij}^{\text{occ}} \sum_{ab}^{\text{vir}} \frac{(ia|jb)[2(ia|jb) - (ib|ja)]}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b}. \quad (6.26)$$

Here, the computation of the molecular integral

$$(ia|jb) = \sum_{\mu\nu\lambda\sigma} C_{\mu i} C_{\nu a} C_{\lambda j} C_{\sigma b} \Gamma_{\mu\nu,\lambda\sigma} \quad (6.27)$$

is the bottleneck and costs $O(N^5)$ if the product sum is taken stepwise.

In the actual implementation, it is necessary to balance the computational cost with the memory and/or scratch capacity by considering the general relationship of the number of AOs ($\mu\nu$) > the number of unoccupied orbitals (ab) > the number of occupied orbitals (ij).

6.6.2 Component of the CI Method: Diagonalization of Large Sparse Matrix

The second topic of this section is the diagonalization of huge matrix, which is required in such as the CI method. At first, the formulation of the CI method will be described.

The general CI wave function is given by,

$$\Psi_{\text{CI}} = t_0 \Phi_0 + \sum_i^{\text{occ}} \sum_a^{\text{vir}} t_i^a \Phi_i^a + \sum_{ij}^{\text{occ}} \sum_{ab}^{\text{vir}} t_{ij}^{ab} \Phi_{ij}^{ab} + \dots, \quad (6.28)$$

where Φ_0 represents the HF wave function, Φ_i^a represents the configuration function where one electron of i in the HF configuration is excited to orbital a , Φ_{ij}^{ab} represents the configuration function where two electrons of i and j in the HF configuration are excited to a and b . In the ground-state CI method, the following equation is variationally solved,

$$\mathbf{H}t = tE \quad (6.29)$$

to obtain the lowest eigenenergy and the linear combination coefficients, t . Here, the Hamiltonian matrix, \mathbf{H} , has the following structure:

$$\mathbf{H} = \begin{pmatrix} \langle \Phi_0 | \hat{H} | \Phi_0 \rangle & \boxed{0} & \langle \Phi_0 | \hat{H} | \Phi_{kl}^{cd} \rangle & \cdots \\ \boxed{0} & \langle \Phi_i^a | \hat{H} | \Phi_k^c \rangle & \langle \Phi_i^a | \hat{H} | \Phi_{kl}^{cd} \rangle & \\ \langle \Phi_{ij}^{ab} | \hat{H} | \Phi_0 \rangle & \langle \Phi_{ij}^{ab} | \hat{H} | \Phi_k^c \rangle & \langle \Phi_{ij}^{ab} | \hat{H} | \Phi_{kl}^{cd} \rangle & \\ \vdots & & & \ddots \end{pmatrix}, \quad (6.30)$$

where each piece in the square box represents all matrix elements for all i, j, \dots and/or a, b, \dots .⁵ Therefore, the actual size of the Hamiltonian is significantly large, although the number of the required roots is only a few with the lowest eigenvalues or even one when only calculating the ground state.

An algorithm to obtain a few of the lowest eigensolutions is the Davidson method, which is the iterative procedure with trial vectors [12]. First, the trial vectors $\{\mathbf{b}_k; k = 1, \dots, M\}$ are prepared, where M is about twice of the number of required roots. Because the CI Hamiltonian is diagonal dominant, the basis vectors for the smallest diagonal matrix elements can be adopted as the initial trial vectors. The $M \times M$ matrix that is constructed by projecting the Hamiltonian with these trial vectors, $\mathbf{H}' = \mathbf{b}^\dagger \mathbf{H} \mathbf{b}$, is diagonalized to obtain the eigenvalues $\{\lambda_k\}$ and eigenvectors $\{\mathbf{v}_k\}$ approximated in the subspace. The residual for the target state i is calculated as the following:

$$\mathbf{t}_i = (\lambda_i \mathbf{I} - \mathbf{D}_H)^{-1} (\mathbf{H} - \lambda_i \mathbf{I}) \mathbf{b} \mathbf{v}_i, \quad (6.31)$$

where \mathbf{D}_H is the diagonal element matrix of \mathbf{H} . The components of \mathbf{t}_i orthogonal to the subspace $\{\mathbf{b}_k\}$ are then added to the subspace and this procedure is iterated until convergence. This algorithm achieves not only the reduction of the computational cost but also the avoidance of the huge \mathbf{H} storage on the memory for the diagonalization. Furthermore, using the fact that \mathbf{H} is a sparse matrix of which the elements are the linear combination of the molecular integrals, the storage of \mathbf{H} on the scratch can be altered by that of molecular integrals by directly constructing the σ vector, $\sigma = \mathbf{H} \mathbf{b}$ for the molecular integrals read from the scratch. The Davidson method is famous as the diagonalization method of huge matrix born in the field of the quantum chemistry and is often used even in the other field of studies.

⁵ $\boxed{0}$ is due to Brillouin's theorem $\langle \Phi_0 | \hat{H} | \Phi_i^a \rangle = 0$.

6.7 Acceleration

6.7.1 *Reduction in Operation Amount*

The most important approach for acceleration is to reduce the amount of computation. This should be tried prior to tuning source codes in order to obtain better computer performance. One of the basic methods is to develop new algorithms of AO ERIs and electron correlation calculations. These operation amounts depend on the timing for the contraction of basis functions or the transformation order from AOs to MOs. Various algorithms continue to be proposed at present. Commonly used techniques are the cut-off and symmetry of AO ERIs described in Sect. 6.5.1. Furthermore, if a molecule has space symmetry, electron configurations that have different symmetries do not mix in electron correlation calculations, and AO ERIs with the same value are obtained by symmetry operations such as rotations and reflections. The computational cost can be decreased by treating only configurations with the same symmetry or symmetry-unique ERIs.

6.7.2 *Reduction in Iteration Cycles*

Iteration procedures are often performed in quantum chemical calculations, and the reduction in the number of iteration cycles lead to the reduction in the computational time. The DIIS method describe in Sect. 6.5.4 is mostly used in SCF calculations. The quadratically convergent SCF method [13] is also used especially for complicated electronic structure systems containing transition metals although the computational cost is very high. In the case of a small energy gap between electron-occupied and unoccupied orbitals (the so-called HOMO-LUMO gap), the total molecular energy sometimes oscillates. To widen the gap artificially, the level shift method [14] is useful to prevent changing electron-occupied orbitals.

In geometry optimization calculations, stable geometries are found with the quasi-Newton–Raphson method [15], in which the first derivatives are calculated analytically and the second derivatives are usually updated by the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method [16]. Initial second derivatives are calculated with force field parameters on the redundant coordinate system that utilizes molecular bond lengths, angles, and torsions [17], while the unit matrix is employed on the Cartesian coordinate system. The initial values are improved on the redundant system, and then the number of geometry optimization cycles can be greatly reduced.

6.7.3 *Improvement of Single-Core Performance*

Computational speed may vary depending on how you write a source code even though an algorithm or an equation is the same. It is difficult to make full use of the performance of CPUs and memory unless you understand the current computing mechanism. In single instruction multiple data (SIMD) operations, since multiple data are processed simultaneously by one operation instruction, the performance can be drawn by using “do” or “for” loops a lot. If there are data dependencies or if-clauses in a loop, it is difficult to simply run the loop, so it is necessary to simplify the operations in the loop. If the loop length does not depend on the input data, writing it with a specific numerical value or parameter variable (a value unchanged in the program) rather than a variable makes it easier for a compiler to optimize. In the future, the SIMD width (amount of data that can be processed by single instruction) will be even larger, and it is considered that the use of SIMD instructions becomes more and more important. In electron correlation calculations, the amount of their intermediates is huge, and three- or four-dimensional arrays are often handled. When a CPU fetches data from the memory, continuous data on the memory are transferred together. If the number of accesses to the memory is reduced by fetching efficiently, the data transfer time can be shortened. Therefore, cache misses can be reduced by devising the arrangement of multidimensional arrays and multiple-loop structures to consecutive data access, enabling efficient computation of the CPU.

It is possible to use CPUs much more effectively by using mathematical BLAS and LAPACK routines than writing matrix multiplication and diagonalization operations by hand. In particular, BLAS level 3 routines are indispensable for dense matrix–matrix multiplications of electronic correlation calculations. In many libraries, the effective utilization of caches is taken into consideration and thread parallelization is implemented, and thus intra-node matrix–matrix product calculations are greatly accelerated.

6.8 Parallelization

6.8.1 *Parallelization Method*

The way of parallelization depends on what range and what to distribute, and by combining them, it is possible to efficiently utilize large numbers of nodes and cores simultaneously. It is the mainstream to adopt the MPI standard for internode parallelization and the OpenMP standard for intra-node, and using thread-parallelized BLAS and LAPACK libraries enables more efficient development. The MPI is available for inter- and intra-node parallelization. Obtaining performance is relatively easy, but starting to use is a bit difficult. The OpenMP is available for intra-node parallelization and is easy to start using. In many cases, it is difficult to obtain high performance unless you firmly consider how to store data and how to distribute

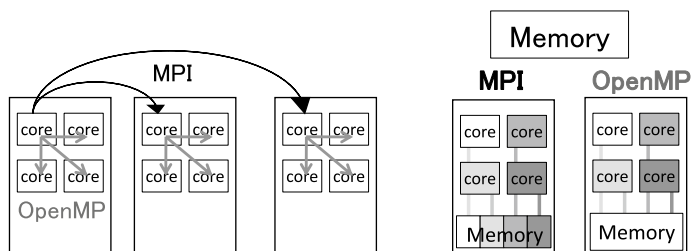


Fig. 6.4 Hybrid MPI/OpenMP parallelization (left) communication (right) memory distribution

indices in a node. The importance of OpenMP is increasing more and more because it is expected that the number of CPU cores per node will further increase in the future.

The feature of the hybrid MPI/OpenMP parallelization is shown in Fig. 6.4. One of the great merits in the field of quantum chemistry is the effective use of memory. If distributed only with MPI, each core allocates memory space as shown in Fig. 6.4b, and the amount of memory that can be used per MPI process becomes small, which makes it difficult to assign a large array. On a node using OpenMP, since the memory area can be shared by the cores, the array size can be increased. Another great merit is the improvement of parallelization efficiency. In hybrid parallelization, the number of MPI processes can be reduced from the total number of CPU cores to the total number of nodes, contributing to the improvement of high load balancing and the reduction of communication volume. Furthermore, it is possible to dynamically distribute computational tasks with OpenMP in a node. As a disadvantage, algorithms and programs become complicated and the development cost increases, but this approach is quite significant for large-scale calculations using present and future computers.

6.8.2 Optimization of MPI Communication

The primary factor of the MPI communication time depends on the data size, and the countermeasure will change accordingly. In the case of small data, the latency time accounts for the majority, and it is important to reduce the number of communications. A solution is to gather data on one array and send them only once, not repeatedly as shown in Fig. 6.5. In the case of large data, the bandwidth is dominant, and an algorithm with small amount of communications should be developed.

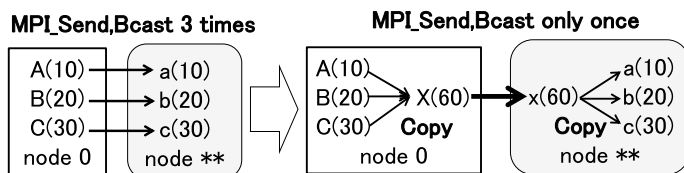


Fig. 6.5 Procedure for reducing the number of communications

6.8.3 Optimization of OpenMP Parallelization

In an OpenMP parallel region, all variables are classified into shared variables, which are shared among threads of one process, and private ones, in which each thread has a different value. When introducing OpenMP into an existing code, pay attention to bugs due to forgetting to specify variables to be private. The cost of “!\$OMP parallel” or “!\$OMP do” (especially, `schedule(dynamic)`) cannot be ignored if the calculation amount of the OpenMP region is small. We should design as many calculations as possible in one OpenMP parallel region, and parallelize the outer loop of a multiple loop to reduce the extra cost as much as possible.

When we heavily use exclusive `critical` and/or `atomic` directives, the wait time increases and the efficiency decreases. We have to develop loop and data structures that do not overwrite data among threads, or prepare variables for each thread with `private` or `reduction` directives.

6.8.4 Importance and Difficulty of Acceleration and Parallelization

It is expected that the numbers of nodes and CPU cores per node will increase in supercomputers and PC clusters, and the SIMD width will be larger. Acceleration and parallelization are becoming indispensable not only for special large-scale calculations but also for routine calculations. In the field of quantum chemistry, the improvement of existing programs is often complicated to obtain good performance. It is also required to rethink from the derivation of equations and algorithms. Sometimes acceleration and parallelization cannot be progressed independently, and it may be necessary to consider them simultaneously. Consequently, the development cost has been increasing. We are now at the stage of thinking about cost reductions throughout the field through open-source program sharing.

6.9 Cases of Acceleration and Parallelization

6.9.1 Algorithm of Atomic Orbital Electron Repulsion Integral Calculation

The AO ERI is a very important term, which is required for most quantum-chemical calculations and is expressed as Eq. (6.11). Compared with the classical point charge treatment, the computation is complicated since an electron is described by a set of Gauss-type functions. Thus many algorithms have been proposed so far, for example, the Rys polynomial method [18], the Pople–Hehre (PH) method which rotates the xyz axes to reduce the operation amount [19], the Obara–Saika [20], McMurchie–Davidson (MD) [21], and Head–Gordon–Pople [22] methods which use recurrence relations to efficiently increase angular momenta, the Accompanying Coordinate Expansion (ACE) method [23], and so on.

An AO (basis function) is described as a linear combination of Gauss-type functions. For instance, the STO-3G basis set is the contraction of three Gauss-type functions. The calculation cost of an ERI increases in proportion to the fourth, second, and zeroth powers of the number of the contraction and changes at the timing of the contraction. The combination of the PH and MD methods is introduced here [24].

The PH method rotates the coordinate axes as shown in Fig. 6.6 in order to make the x and y components of AB and the y component of CD be zero and some other components be constant values. The MD method increases the orbital angular momenta of the integrals from the $(ss|ss)$ type using recurrence relations (Fig. 6.7).

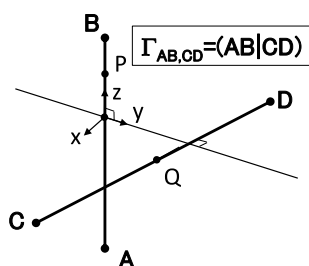


Fig. 6.6 Axes rotation in Pople–Hehre method

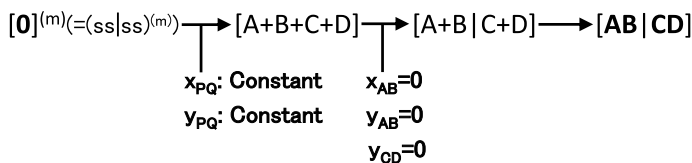


Fig. 6.7 Procedure for Pople–Hehre + McMurchie–Davidson method

Table 6.3 Operation amount of $(sp, sp|sp, sp)$ integral ($=xK^4 + yK^2 + z$, K : contraction number of basis functions)

Method	PH	PH+MD
x	220	180
y	2300	1100
z	4000	5330

Table 6.4 Operation amount of $(sp, sp|sp, sp)$ integral for each contraction number of basis functions (K)

K	PH	PH+MD
1	6,520	6,583
2	16,720	12,490
3	42,520	29,535

Table 6.5 Fock matrix calculation time (s)

Molecule	Taxol (C ₄₇ H ₅₁ NO ₁₄)		Luciferin (C ₁₁ H ₈ N ₂ O ₃ S ₂)
Basis set	STO-3G (361 dimensions)	6-31G(d) (1032 dimensions)	aug-cc-pVDZ (550 dimensions)
Original GAMESS (PH)	85.7	2015.2	2014.9
PH+MD	69.9	1361.8	1154.5

In the combination of these methods, the axes are rotated, the angular momenta of the integrals are increased using the recurrence relations, and then finally the axes are re-rotated to the original position. The cost proportional to the fourth power of the contraction can be reduced by the feature that several components are constant values at the first recurrence relation (Table 6.3). Where the angular momenta are distributed to the bra and ket, since several components are 0, the amount of the calculation proportional to the square can be reduced. In the case of the $(sp, sp|sp, sp)$ integral, the operation amount of the PH+MD method can be reduced by about 30% compared with the PH method for the STO-3G basis set (the number of the contraction $K = 3$) as shown in Table 6.4. It demonstrates good performance for moderately contracted basis sets such as the 6-31G(d) and cc-pVDZ basis sets.

Using this algorithm, the code of 21 integral routines from $(ss|ss)$ to $(dd|dd)$ were generated. About 20,000 lines of the code were automatically generated by the program writing them, reducing the development cost including debugging. After integrating these routines into the GAMESS program [25], the computational time for the Fock assembly was measured on a Pentium 4 machine (Table 6.5). The time of the PH+MD method is 20–40% faster than that of the PH method. This method has been used as a default ERI routine of the GAMESS since 2005.

6.9.2 MPI Parallelization of the Second-Order Perturbation (MP2) Calculation

The MP2 method is the simplest electron correlation calculation, and the energy is obtained using MO energies and MO ERIs evaluated by multiplied AO ERIs ($\mu\nu|\lambda\sigma$) by MO coefficients $C_{\mu a}$ four times as described in Eqs. (6.26) and (6.27). Many parallel algorithms that distribute AO or MO indices have been developed. In the distribution of AOs, it is necessary to sum partial MO ERIs on multiple processes, and thus the total communication amount depends on the number of processes. On the other hand, in the distribution of MOs, the same AO ERIs have to be evaluated in multiple processes. The parallel efficiency decreases as the number of processes increases.

Baker and Pulay proposed a parallel algorithm to distribute AOs in the first half transformation and MOs in the second half in 2002 [26]. Data sorting for communication takes some time, but all calculations can be distributed, and the total traffic volume is almost constant irrespective of the number of processes. Since the data amount of the ERIs after the first half transformation increases in proportion to the fourth power of the number of basis functions, it is roughly in TB units for molecules of hundreds of atoms. Therefore, intermediate data were stored on disk at the time.

Figure 6.8 shows an algorithm to distribute AOs until the third quarter transformation and MOs at the fourth transformation and the energy calculation [27] based on the algorithm above. Simple data sorting is achieved while maintaining the features of load balancing and communication. In the second to fourth transformations, using the BLAS-3 matrix–matrix multiplication routine DGEMM, the calculation and rearrangement of data of multidimensional arrays are performed simultaneously. The data after the third transformation are saved on disk.

When calculating the product of $\mathbf{A}(M,K)$ and $\mathbf{B}(K,N)$ with the DGEMM routine, the obtained matrix \mathbf{C} can have two types of dimensions (Fig. 6.9). If \mathbf{A} and \mathbf{B} are multiplied without transposing, \mathbf{C} becomes (M,N) , and if \mathbf{B} and \mathbf{A} are multiplied with transposing, \mathbf{C} becomes (N,M) . Rearranging data without an extra cost can be done with an efficient library.

Fig. 6.8 MPI Parallel algorithm for MP2 energy calculation

```

 $\mu$  (AO index) Distributed to processes
do  $\lambda, \sigma$ 
  AO ERI Calc. ( $\mu\nu|\lambda\sigma$ ) [ $v, \mu\lambda\sigma$ ] (all  $v$ )
  First Trans. ( $\mu i|\lambda\sigma$ ) [ $i, \mu\lambda\sigma$ ] (all  $i$ )
  Second Trans. ( $\mu i|\lambda j$ ) [ $i j, \lambda, \mu$ ] (all  $i \geq j$ )
end do  $\lambda, \sigma$ 
Third Trans. ( $\mu i|bj$ ) [ $b, ij$ ] (all  $b$ )
Store ( $\mu i|bj$ ) on Disk [ $b, ij, \mu$ ]
end do  $\mu$ 
ij (MO index pair) Distributed to processes
Read ( $\mu i|bj$ ) from Disk + MPI_isend,irecv
Fourth Trans. ( $a i|bj$ ) [ $b, a$ ] (all  $a, b$ )
MP2 Energy Calc.
end do ij
  
```

Fig. 6.9 Matrices obtained by DGEMM routine

•Without transpose: DGEMM('N', 'N' ,...,A,...,B,...,C,...)

$$\begin{array}{c} \text{N} \\ \boxed{C_1} \end{array} = \begin{array}{c} \boxed{A} \end{array} \begin{array}{c} \boxed{B} \end{array}$$

•With transpose: DGEMM('T', 'T' ,...,B,...,A,...,C,...)

$$\begin{array}{c} \text{M} \\ \boxed{C_2} \end{array} = \begin{array}{c} \boxed{B^\dagger} \end{array} \begin{array}{c} \boxed{A^\dagger} \end{array}$$

Table 6.6 MP2 energy calculation time (h) and speedup

Number of CPUs	1	2	4	8	16
<i>6-31G(d) (1032 dimensions)</i>					
Time	10.2	5.08	2.54	1.31	0.64
Speedup	1.0	2.0	4.0	7.8	15.8
<i>6-311G(d, p) (1484 dimensions)</i>					
Time	31.6	16.3	8.06	4.05	2.05
Speedup	1.0	1.9	3.9	7.8	15.4

This algorithm was implemented into the GAMESS program, and benchmark calculations were performed using a Pentium 4 3.0GHz cluster connected by a Gigabit Ethernet network. Results of Taxol ($C_{47}H_{51}NO_{14}$) molecule with the 6-31G(d) and 6-311G(d, p) basis sets (1032 and 1484 dimensions, respectively) are summarized in Table 6.6. The total amount of the intermediate data for the 6-311G(d, p) was 202 GB. In both cases, the speedup is almost the same as the number of CPUs used, indicating high parallel performance.

6.9.3 Hybrid MPI/OpenMP Parallelization for Hartree–Fock Calculation

The most time-consuming step in the HF calculation is the evaluation of AO ERIs and their addition to the Fock matrix. An MPI/OpenMP parallel algorithm of this step [28] is shown in Fig. 6.10. The outermost loop of the quadruple loop is parallelized by OpenMP and the third loop by MPI. After the quadruple loop, the Fock matrix elements are added within a process by “reduction” of OpenMP and among processes by “MPI_allreduce”.

Since the OpenMP parallelization is done in the outermost loop, extra costs such as thread creation and distribution are decreased as much as possible. High load balancing is achieved due to the dynamic distribution from indices with a large amount of computation. The MPI parallelization is performed by the “mod” (remainder)

Fig. 6.10 MPI/OpenMP parallel algorithm of Fock matrix calculation

```

!$OMP parallel do schedule(dynamic,1) reduction(+:Fock)
do  $\mu = \text{nbasis}, 1, -1$   $\leftarrow$  Distributed by OpenMP
  do  $\nu = 1, \mu$ 
     $\mu\nu = \mu(\mu+1)/2 + \nu$ 
     $\lambda\text{start} = \text{mod}(\mu\nu + \text{mpi\_rank}, \text{nproc}) + 1$ 
    do  $\lambda = \lambda\text{start}, \mu, \text{nproc}$   $\leftarrow$  Distributed by MPI
      do  $\sigma = 1, \lambda$ 
        ERI ( $\mu\nu|\lambda\sigma$ ) Calc.+ Addition to Fock Matrix
      enddo
    enddo
  enddo
enddo
!$OMP end parallel do
call mpi_allreduce(Fock)

```

calculation without using an if-clause and a counter. Because all processes execute the same calculations up to the distribution point by MPI ranks, the redundant part is reduced as much as possible. The MPI communication is located outside the OpenMP region, which is a simple communication.

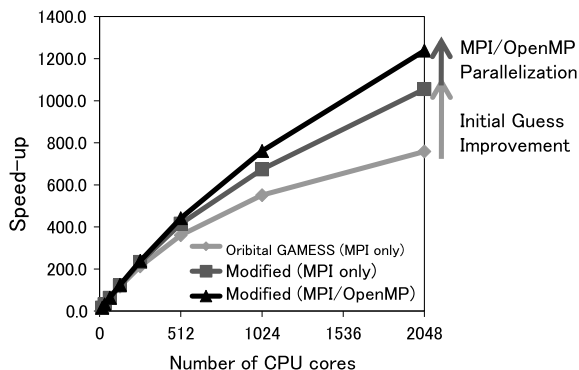
This algorithm was implemented into the GAMESS program in which only process parallelization was introduced. In addition to the Fock matrix computation, the initial guess calculation was also accelerated and parallelized. In the stage of projecting MOs obtained by the extended Hückel method to MOs of the basis used in the SCF calculation, many matrix–matrix multiplications are adopted. Furthermore, in the middle of the SCF calculation, an approximate second-order SCF method [29] based on the Newton–Raphson method is introduced. The Fock matrix is diagonalized only at the first and last two times.

Benchmark calculations were performed using a TiO₂ cluster (T₃₅O₇₀, 6-31G (1645 dimensions), 30 SCF cycles), and Cray XT5 (Opteron 2.4 GHz, 8 cores/node) 2048 cores. The speedup of the total time is shown in Fig. 6.11. The speedup of the new algorithm with only MPI is improved compared with that of original GAMESS as the number of cores increases, and the speedup of the hybrid MPI/OpenMP parallelization is further improved.

In the original calculation, the fraction of the initial guess calculation time to the total calculation is less than 1% on 16 cores, but it accounts for about 40% on 2048 cores (Table 6.7). Due to the improvement of this part, it becomes less than 7% even on 2048 cores, and the overall speedup increased drastically. Although the Fock matrix calculation time (Table 6.8) is almost the same in any calculation on 16 cores, the time of hybrid parallelization is the shortest on 2048 cores, and this also contributes to shortening the calculation time overall and improving the speedup.

Hybrid parallelization becomes more effective as the number of used cores increases, and it is significant in future many-core systems. The analysis of the initial guess calculation time demonstrates that it is necessary to accelerate and parallelize all steps in massively parallel computation.

When introducing OpenMP into the GAMESS program, it changed to a specific program because the source code was rewritten considerably. All the variables in the

Fig. 6.11 Speedup of Hartree-Fock calculation**Table 6.7** Initial guess calculation time (s) and its percentage of total calculation (in parentheses)

Number of CPU cores		16	256	1024	2048
Original GAMESS	MPI only	166.2 (0.9%)	143.6 (10.5%)	143.6 (27.2%)	143.8 (37.5%)
Modified	MPI only	20.2 (0.1%)	18.6 (1.5%)	18.9 (4.4%)	19.2 (7.0%)
Modified	MPI/OpenMP	18.6 (0.1%)	13.2 (1.1%)	13.6 (3.6%)	13.8 (5.9%)

Table 6.8 Fock matrix calculation time (s) and speedup (in parentheses)


Number of CPU cores		16	256	1024	2048
Original GAMESS	MPI only	17881.8 (16.0)	1175.2 (243.5)	334.0 (856.6)	188.6 (1517.0)
Modified	MPI only	17953.5 (16.0)	1175.2 (244.4)	360.0 (797.9)	203.1 (1414.4)
Modified	MPI/OpenMP	17777.6 (16.0)	1150.4 (247.3)	316.4 (899.0)	174.8 (1627.2)

OpenMP region are classified as private or shared ones, the common variables to be made private is changed to the arguments of subroutines, and scalar variables written in arguments are rearranged into an array in order to reduce the number of variables. It is often difficult to introduce OpenMP to an existing program and to achieve high execution and parallel performance. The development with data and loop structures considering parallelization is now required from the design stage of a program.

6.10 Approximations of Quantum Chemical Methods for Large Systems

Here, we summarize the orders of the computational time and data amounts, and the accuracy of the typical quantum chemical methods (Table 6.9). Even in the low-accuracy methods, the required computational time scales as $O(N^3)$, which is derived

Table 6.9 Computational time and accuracy of the typical quantum chemical methods

Method	Hartree-Fock	DFT	MP2	CCSD	CCSD(T)
Time	$O(N^3)$	$O(N^3)$	$O(N^5)$	$O(N^6)$	$O(N^7)$
Data amount	$O(N^2)$	$O(N^2)$	$O(N^4)$	$O(N^4)$	$O(N^4)$
Approximation	← Mean-field theory →		← Electron correlation theory →		
w/ × 1000 faster comp.	× 10	× 10	× 4.0	× 3.2	× 2.7
Accuracy	Qualitative  Accurate				

from the direct diagonalization of dense matrix. Therefore, even if a 1000 times faster computer is provided, the computable system size only increases by a factor of 10. Furthermore, the computational time increases rapidly as the accuracy is improved by treating the electron correlation.

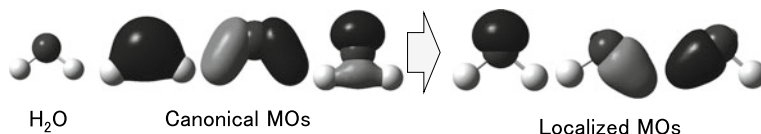
There are many approximations aiming at the treatment of large systems. In the ONIOM (our own N -layered integrated molecular orbital and molecular mechanics) method [30], a molecular system is divided into two or three layers, and the layers are calculated with different kinds of methods. High level (ab initio) methods are applied to important layers, and low level (semiempirical or molecular mechanics) methods are applied to the others. The results are combined to predict the high-level result of the whole system.

The quantum mechanics/molecular mechanics (QM/MM) method [31] is an approach to divide a system and calculate the important part with a QM method and the other with MM. The hybrid method can be applied to chemical reactions with solvent or protein environments, etc.

Since valence electrons play a significant role in chemical bonds, effective core potentials (ECPs) [32] are often used to replace core electrons of heavy atoms by potentials. The number of electrons treated explicitly can be reduced. Major potentials are LANL2 (Hay–Wadt) [33], Stuttgart [34], and SBKJC [35].

Another approach to restrict the range of calculations is the frozen core approximation [36] which ignores excitations from core electrons. This is used in most electron correlation calculations.

In general, MOs obtained by SCF calculations spread throughout a molecule. In the localized molecular orbital (LMO) method (Fig. 6.12) [37], MOs are localized in atoms or bonds by an appropriate transformation. The cost for electron correlation calculations can be reduced by considering only excitations among near LMOs.

**Fig. 6.12** Localized molecular orbital method

Several approximations for the ERI evaluation has been developed. In the density fitting or resolution of the identity (RI) method [38], four-center ERIs are approximated by the products of two- and three-center integrals including auxiliary functions. The amounts of data and operations become small especially in electron correlation calculations. There is also the FMM method which uses the multipole expansion for long-range Coulomb interactions described in Sect. 6.5.1.

Recently, the linear-scaling methods, or equivalently the $O(N)$ theories have rapidly been developed to treat huge systems. In Chap. 4 of this book, an $O(N)$ method using the Krylov subspace is introduced mainly for the Green's function method, where the Hamiltonian is approximated based on the cut-off distance. In this section, we describe the fragment-based methods that achieve $O(N)$ computation by combining the results of several subsystem calculations.

6.10.1 Fragment Molecular Orbital Method

Fragment molecular orbital (FMO) method proposed by Kitaura and coworkers is one of the pioneering fragment-based methods that orient toward large-scale calculation. First, this FMO method will be introduced [39].

The FMO method is mainly used in calculations of molecular clusters and biomolecular systems such as proteins. When treating a huge single molecule, the FMO method requires to cut chemical bonds to construct fragment systems. As an example, Fig. 6.13 shows the recommended fragmentation scheme in the FMO calculations of peptide systems. A cut chemical bond is described using the hybridized orbital by either of two fragments sharing the bond. For example, because the left-most C–C bond in the Figure is included in the left-side fragment, the hybridized orbital of the right-side C atom that contributes to the C–C bond is included into the left-side fragment instead of the right-side one. This scheme is called as the hybridization orbital projection (HOP).

Various properties such as the energy are expressed by the many-body expansion formulas. The simplest form expressed by the sum of the one-body (monomer) energies $\{E_I\}$ is called as the FMO1,

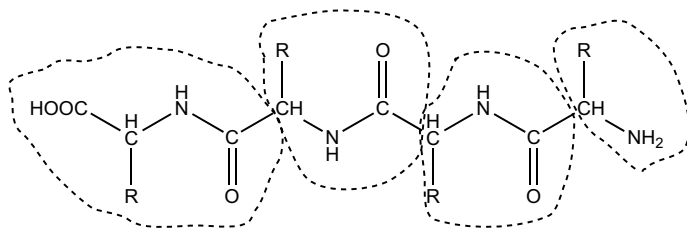


Fig. 6.13 Recommended fragmentation scheme in FMO calculation of peptides

$$E_{\text{FMO1}} = \sum_I E_I. \quad (6.32)$$

The FMO2 and FMO3 include the two-body (dimer) and three-body (trimer) energy corrections,

$$E_{\text{FMO2}} = E_{\text{FMO1}} + \sum_{I>J} (E_{IJ} - E_I - E_J), \quad (6.33)$$

$$E_{\text{FMO3}} = E_{\text{FMO2}} + \sum_{I>J>K} (E_{IJK} - E_{IJ} - E_{JK} - E_{IK} + E_I + E_J + E_K), \quad (6.34)$$

where E_{IJ} and E_{IJK} represent the dimer and trimer energies, respectively. In the FMO method, the computational time can be drastically reduced by truncating this many-body expansion.

Let us go into a bit more detail about the FMO method. In the calculation of fragment X (X can be either a monomer, dimer, or trimer), the following effective Hamiltonian is used:

$$\tilde{H}_{\mu\nu}^X = H_{\mu\nu}^X + V_{\mu\nu}^X + P_{\mu\nu}^X \quad (\mu\nu \in X), \quad (6.35)$$

where \mathbf{H}^X represents the Hamiltonian of the electrons and nuclei in the fragment X itself, and \mathbf{V}^X includes the electrostatic potential from the outside electrons and nuclei of the fragment X :

$$V_{\mu\nu}^X = \sum_{K \notin X} \left[\sum_{A \in K} \left\langle \mu \left| -\frac{Z_A}{|\vec{r} - \vec{R}_A|} \right| \nu \right\rangle + \sum_{\lambda\sigma \in K} D_{\lambda\sigma}^K \Gamma_{\mu\nu,\lambda\sigma} \right]. \quad (6.36)$$

Here, the electron density of the fragment K that is not included in X is evaluated from the monomer density matrix \mathbf{D}^K . \mathbf{P}^X is derived from the bond separation by the hybridization orbital, which is used to project out the components of the hybridization orbitals that does not belong to X . For $\tilde{\mathbf{H}}$ of Eq. (6.35), the HF or KS equation is solved:

$$\tilde{\mathbf{H}}^X \mathbf{C}^X = \mathbf{S}^X \mathbf{C}^X \boldsymbol{\epsilon}^X. \quad (6.37)$$

The density matrix of the fragment X is expressed in the similar manner as the standard calculation as

$$D_{\mu\nu}^X = 2 \sum_i^{\text{occ}(X)} C_{\mu i}^X C_{\nu i}^X. \quad (6.38)$$

The electron number (and the spin multiplicity in case of open-shell system) of each fragment should be specified a priori. Because the monomer density matrices $\{\mathbf{D}^K\}$ are included in the Hamiltonians of the other fragment through \mathbf{V}^X , these density matrices have to be determined in the self-consistent manner. In actual fact, because the density matrix is approximately treated by the electronic charge, this procedure is called as the self-consistent charge (SCC) method.

If no further approximation than the truncation of the many-body expansion is adopted, the computational time for the FMO2 calculation scales as $O(N^2)$ with respect to the entire system size N . For example, in the actual implementation in GAMESS program, the following approximations based on the cut-off distance is introduced.

- RESPPC
Intra-fragment electrostatic potential is approximated with the Mulliken charges instead of explicitly evaluating with ERIs.
- RESDIM
Avoiding explicit dimer calculations by approximating with the electrostatic interaction.
- RCORSD (in case of electron correlation calculation)
Dimer electron correlation is neglected.

If the time for the electrostatic field calculation with the Mulliken charges are neglected, the computational time of the monomer SCC calculation scales $O(N)$ by adopting RESPPC. RESPPC also replaces the monomer density matrix with the Mulliken charges, which can drastically reduce the communication volume in the parallel calculation. RESDIM further reduces the computational scaling of the dimer calculation to $O(N)$.

Various applications of the FMO method can be found in the Ref. [39]. Recent advances in the fragmentation scheme and the generalization to the FMO3 or FMO4 have extended the target systems of the FMO method.

6.10.2 Divide-and-Conquer Method

Here, we would like to introduce another $O(N)$ method, namely the divide-and-conquer (DC) quantum chemical method. The DC method as well as the FMO method are fragment-based large-scale calculation methods in the sense that the entire system is divided into fragments (called subsystems in the DC method). Instead of the dimer corrections in the FMO method, the buffer region is added to each disjoint subsystem (called the central region) and the interaction with the surrounding environment of the subsystem is explicitly considered (Fig. 6.14). The size of the buffer region is the computational parameter that highly correlates with the accuracy of the DC method.

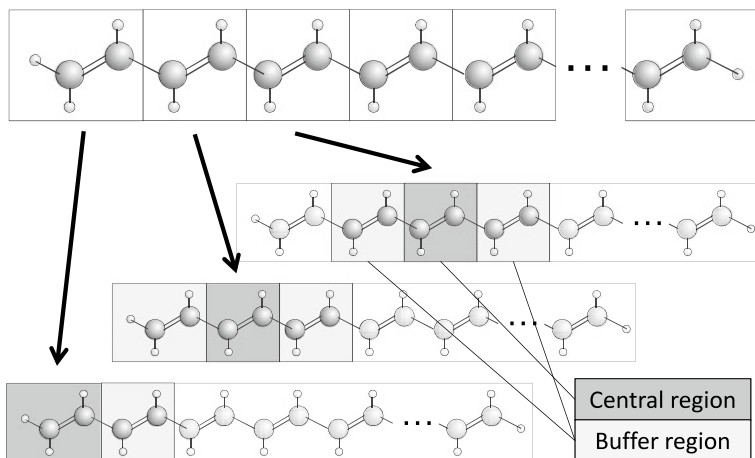


Fig. 6.14 The division scheme of the DC method and schematics of the central and buffer regions

The treatment of the buffer region in the DC-SCF calculation is different from that in the DC post-HF electron correlation calculation. The detailed procedures of these DC calculations are introduced below.

6.10.2.1 DC-SCF Method

The DC-SCF calculation can be considered as a method to construct the density matrix of the entire system from many density matrices evaluated in the subsystem calculations, namely,

$$D_{\mu\nu}^{\text{DC}} = \sum_{\alpha}^{\text{subsystem}} P_{\mu\nu}^{\alpha} D_{\mu\nu}^{\alpha}. \quad (6.39)$$

The scheme to compute Hamiltonian matrix and energy is the same as the standard SCF calculation except using Eq. (6.39) for the density matrix. The density matrix of the subsystem α , $D_{\mu\nu}^{\alpha}$, is constructed with the subsystem MOs. Here, for technical reasons to be explained later, this matrix is evaluated for the ensemble at finite temperature $\beta = (k_{\text{B}}T)^{-1}$:

$$D_{\mu\nu}^{\alpha} \approx 2 \sum_p^{\text{MO}(\alpha)} f_{\beta}(\varepsilon_{\text{F}} - \varepsilon_p^{\alpha}) C_{\mu p}^{\alpha} C_{\nu p}^{\alpha}, \quad (6.40)$$

where ε_{F} represents the Fermi level, $f_{\beta}(x) = [\exp(\beta x) + 1]^{-1}$ is the Fermi distribution function. MOs of the subsystem α is expanded with the basis functions in the

central region $\mathcal{S}(\alpha)$ and those in the buffer region $\mathcal{B}(\alpha)$, and is determined by solving the following subsystem HF (or KS) equation:

$$\mathbf{F}^\alpha \mathbf{C}^\alpha = \mathbf{S}^\alpha \mathbf{C}^\alpha \boldsymbol{\varepsilon}^\alpha. \quad (6.41)$$

The partition matrix \mathbf{P} in Eq. (6.39) defined by

$$P_{\mu\nu}^\alpha = \begin{cases} 1 & (\mu \in \mathcal{S}(\alpha) \wedge \nu \in \mathcal{S}(\alpha)) \\ 1/2 & (\mu \in \mathcal{S}(\alpha) \wedge \nu \in \mathcal{B}(\alpha) \text{ or } \textit{vice versa}) \\ 0 & (\textit{otherwise}) \end{cases} \quad (6.42)$$

is used to normalize the contributions from the overlapping buffer region. Although the Fermi level ε_F still remains as an unknown variable, it is determined from the following equation so that the electron number of the entire system is preserved:

$$N_e = \text{Tr}(\mathbf{D}^{\text{DC}} \mathbf{S}) = 2 \sum_{\alpha} \sum_p^{\text{MO}(\alpha)} f_{\beta}(\varepsilon_F - \varepsilon_p^\alpha) \sum_{\mu\nu}^{\text{AO}(\alpha)} P_{\mu\nu}^\alpha C_{\mu p}^\alpha C_{\nu p}^\alpha S_{\nu\mu}^\alpha. \quad (6.43)$$

The greatest incentive to introduce a finite temperature is to give a solution for this nonlinear equation. In the DC method, the electron number (and the spin multiplicity in an open-shell case) in each subsystem is automatically determined from Eq. (6.43), while it has to be specified before the calculation in the other fragment-based methods such as the FMO method. Due to this feature, the DC method can handle the delocalized electronic structures, which is generally considered difficult to describe by the fragment-based methods.

Let us return the story to the procedure. The subsystem Fock matrix \mathbf{F}^α is evaluated using the same formula to the standard calculation, i.e., Eq. (6.8). Because the computational cost for constructing Fock matrix is large, it is common to compute the entire Fock matrix \mathbf{F} and then take its submatrix corresponding to each subsystem in the subsystem calculation. However, the strategy to directly construct \mathbf{F}^α for each subsystem should also be considered especially in case of using semiempirical Hamiltonian.

Lastly, let us consider this procedure from the viewpoint of the computational cost. Although the calculations of Eq. (6.41) require high computational cost, the time increases as $O(N)$ for large systems because the diagonalization within each subsystem ($O(N^0)$) is performed for every subsystem, the number of which is $O(N)$. The other part requiring high computational cost is the construction of the Fock matrix. Because the DC method makes \mathbf{D} sparse, the time increases as $O(N^2)$ even if only the Schwarz screening is adopted. If FMM is further applied, the $O(N)$ computation can be achieved. Figure 6.15 shows the computational time for the HF/6-31G** calculation of polyene system, C_nH_{n+2} . Obviously, both two parts become faster in the DC method than in the standard method, especially for larger system.

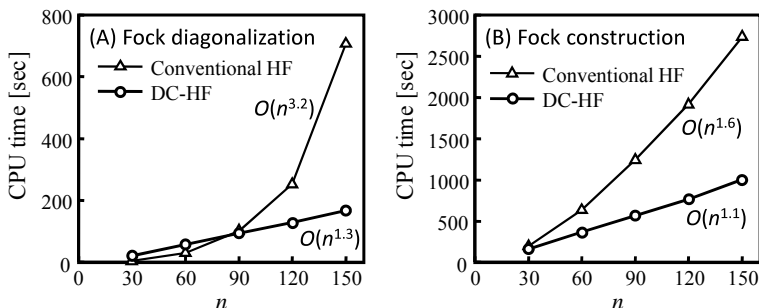


Fig. 6.15 Computational times for **A** diagonalizing Fock matrix and **B** constructing Fock matrix in the HF and DC-HF calculations. The FMM option is used to construct Fock matrix [40]

6.10.2.2 DC-DFTB Method and Determination of Fermi Level

Here, let us think about the electron number conservation condition (Eq. (6.43)) that appears only in the DC method. It is the nonlinear equation to obtain ε_F . Defining the weight of the subsystem MO as

$$w_p^\alpha = \sum_{\mu\nu}^{\text{AO}(\alpha)} P_{\mu\nu}^\alpha C_{\mu p}^\alpha C_{\nu p}^\alpha S_{\nu\mu}^\alpha \quad (6.44)$$

is useful because the electron number can be calculated as sum of the orbitals (p) of all subsystems (α). Because the Fermi function, $f_\beta(\varepsilon_F - \varepsilon_p^\alpha)$, gives almost 1 or 0 except around $\varepsilon_p^\alpha = \varepsilon_F$, the range of ε_p^α that must be evaluated explicitly is determined from the threshold value in advance of the calculation. Then, ε_F can be obtained with the standard iterative root-finding method such as the Brent method [41] by appropriately setting the initial upper and lower limits.⁶ Because the number of the variables regarding all the subsystems (ε_p^α and w_p^α) is $O(N)$, the computational time for the determination of ε_F becomes $O(N)$. However, it is normally adopted to gather all required information to the master process for determining ε_F , since its straightforward implementation requires the communication for every iteration. This does not deteriorate the performance in the DC-HF and DC-DFT calculations because the computational time for the other part is significantly long. However, it becomes problematic if a semiempirical Hamiltonian such as the density functional tight-binding (DFTB) method, which will be introduced below, is adopted.

The DFTB method is a semiempirical computational method based on DFT [42]. In DFT, the total energy is expressed as the functional of the electron density $\rho(\vec{r})$ as Eq. (6.17). In the DFTB method, the density is expressed as sum of the density of the reference state ρ_0 and its change $\delta\rho$ using the tight binding approximation. Because the exchange-correlation functional is nonlinear to the electron density, it is

⁶Either is set to be 0 or the prior convergent value and the other is set to be crossed over the root.

approximated by the second-order Taylor expansion. Furthermore, for the density-dependent terms, the electric charge Δq_A (namely Mulliken charge is used in usual) is approximately used instead of the explicit electron density. Then, Eq. (6.17) can be expressed as the sum of the charge-independent, charge-dependent, and nuclear repulsion terms as

$$E_2^{\text{TB}} = \sum_i^{\text{occ}} 2 \left\langle \psi_i \left| \hat{H}_0 \right| \psi_i \right\rangle + \frac{1}{2} \sum_{A,B} \Delta q_A \Delta q_B \gamma_{AB} + \sum_{A < B} E_{\text{rep}}(R_{AB}). \quad (6.45)$$

The first and third terms and γ_{AB} are parametrized from *a priori* DFT calculations. The Hamiltonian matrix can be expressed as

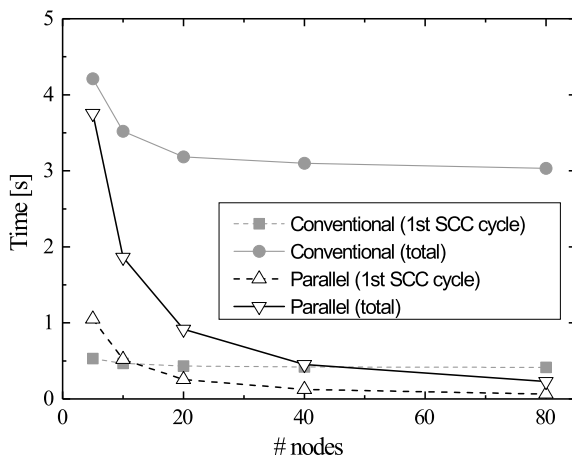
$$H_{\mu\nu} = H_{\mu\nu}^0 + S_{\mu\nu} \sum_C \frac{1}{2} (\gamma_{AC} + \gamma_{BC}) \Delta q_C \quad (\mu \in A, \nu \in B), \quad (6.46)$$

which does not require the ERI evaluation, leading much faster computation. Especially, the method to self-consistently determine the Mulliken charges are known as the SCC-DFTB or DFTB2 method [43].

Although this DFTB method even requires the $O(N^3)$ computational time due to the diagonalization of the Hamiltonian, its combination with the DC method is straightforward and easily achieves near $O(N)$ computation. Its massive parallel calculation, however, significantly deteriorates the performance if the sequential Fermi-level determination scheme is adopted. Then we have proposed a novel algorithm to determine the Fermi level that realizes to reduce the number of iterations with the interpolation method and to parallelize the computation [44]. For each tentative Fermi level given by equally dividing the estimated range of the Fermi level, which has to be specified in advance, the number of electrons included in the subsystems allocated for each node is calculated. Its reduction operation yields the number of electrons in the entire system for each tentative Fermi level, and we can obtain the upper and lower limits of the proper Fermi level within the accuracy of energy interval. The Fermi level is estimated from the interpolation between the upper and lower limits. If sufficient accuracy can not be obtained, this process is repeated once again. It is possible to reduce the number of iterations (=the number of reduction operations) within 2 times and to shorten the computational time by computing for several energy levels simultaneously, although a nonlinear iteration calculation is not avoided.

Figure 6.16 shows the computational times for Fermi level determination of a system containing 4000 water molecules with the conventional and the abovementioned parallel interpolation methods. The time for the first SCC cycle and the accumulated time measured with the K Computer are shown, where the horizontal axis shows the number of nodes used. In the first SCC cycle, the time with the conventional

Fig. 6.16 Dependence of the computational time for Fermi-level determination on the number of nodes. The time was measured with the K Computer for a system with 4000 water molecules [44]



method is shorter than that with the parallel interpolation method when using a small number of nodes. However, it hardly decreases with the increase of the number of nodes. On the contrary, the time with the interpolation method decreases ideally as the number of nodes increases. Comparing the accumulated time, the time with the interpolation method becomes shorter than that with the conventional one even with a small number of nodes. This is because the Fermi level hardly changes for the late SCC cycles, and sufficient accuracy can be obtained by only one interpolation process.

We have developed a super-parallel DC-DFTB program, called DC-DFTB-K, which includes the techniques mentioned above, and implemented into the K Computer. Table 6.10 shows the computational time for each DC-DFTB procedure when performing the DC-DFTB calculation of a system containing 256,000 water molecules with the DC-DFTB-K program. N_{node} represents the number of nodes used in the calculation. The last row (efficiency) represents the parallel efficiency with 5,120 nodes against the time with 640 nodes. The parallel efficiency is higher than 80% except for the one-electron integral (\mathbf{H}^0 and \mathbf{S}) evaluation with the short computational time. Because an $O(N)$ Coulomb calculation method such as FMM is not implemented, the computational scalings for γ and gradient calculations are $O(N^2)$. However, the parallel efficiency of these procedures are especially high.

Table 6.10 Computational time and parallel efficiency of each procedure in DC-DFTB calculation [44]

N_{node}	E_{rep}	\mathbf{H}^0, \mathbf{S}	γ	SCC	Grad.	Total
640	1.81	0.42	27.95	165.66	37.33	233.16
1280	0.92	0.22	13.96	85.81	18.78	119.69
2560	0.48	0.13	7.11	47.87	9.58	65.17
5120	0.26	0.07	3.55	25.18	5.05	34.12
Efficiency (%)	86	74	98	82	92	85

6.10.2.3 DC-Based Post-HF Electron Correlation Calculation

Let us move on to the post-HF electron correlation calculation based on the DC method. The DC-based post-HF calculation adopts a strategy to obtain the electron correlation energy of the entire system by accumulating the subsystem correlation energies that is expressed with the subsystem MOs. The contribution from the overlapping buffer region is removed using the analogy to the energy density analysis [45].

According to the Nesbet's theorem [46], the correlation energy ΔE can be expressed with the occupied and unoccupied MOs of the HF method as

$$\Delta E = \sum_{ij}^{\text{occ}} \sum_{ab}^{\text{vir}} (ia|jb)[2\tilde{t}_{ia,jb} - \tilde{t}_{ib,ja}]. \quad (6.47)$$

Here, $\tilde{t}_{ia,jb}$ represents a coefficient called the amplitude, and the expression differs depending on the method adopted. To obtain the correlation energy of a subsystem, the MOs in this equation are replaced with the subsystem MOs. Here, the Energy Density Analysis (EDA) is used to obtain only the contribution from the central region. Using the following 3-index transformed ERIs, $(\mu a^\alpha | j^\alpha b^\alpha) = \sum_{\nu\lambda\sigma} C_{\nu a}^\alpha C_{\lambda j}^\alpha C_{\sigma b}^\alpha \Gamma_{\mu\nu,\lambda\sigma}$, the summation of the last integral transformation is limited to the AOs of the central region of the subsystem α , $\mathcal{S}(\alpha)$,

$$\Delta E^\alpha = \sum_{ij}^{\text{occ}(\alpha)} \sum_{ab}^{\text{vir}(\alpha)} \sum_{\mu \in \mathcal{S}(\alpha)} C_{\mu i}^\alpha (\mu a^\alpha | j^\alpha b^\alpha) [2\tilde{t}_{ia,jb}^\alpha - \tilde{t}_{ib,ja}^\alpha]. \quad (6.48)$$

It makes it possible to estimate the contribution of the correlation energy only from the central region. The amplitudes are evaluated with the subsystem MOs, e.g., in case of MP2 method,

$$\tilde{t}_{ia,jb}^\alpha = \frac{(i^\alpha a^\alpha | j^\alpha b^\alpha)}{\varepsilon_i^\alpha + \varepsilon_j^\alpha - \varepsilon_a^\alpha - \varepsilon_b^\alpha}. \quad (6.49)$$

The total correlation energy is evaluated by summing up all the subsystem correlation energies:

$$\Delta E \approx \sum_{\alpha} \Delta E^\alpha. \quad (6.50)$$

This procedure not only achieves $O(N)$ computation time but also realizes the required memory size independence for N . This feature is essential especially in the highly accurate post-HF electron correlation calculations such as the coupled cluster method, which requires huge memory capacity.

In the DC-based post-HF method, because the computation to evaluate a subsystem correlation energy is completely independent of the other subsystems, the process can be parallelized straightforwardly. Furthermore, a standard highly parallelized MP2 algorithm can be adopted for each subsystem MP2 energy calculation. It provides a two-level parallelization scheme where the fine-grained parallel computation of each subsystem MP2 calculation is performed inside the course-grained parallelization over the subsystem. An example to implement this algorithm to GAMESS will be introduced.

GAMESS provides the proprietary interface for multilayer parallel processing, called GDDI (generalized distributed data interface). This is first introduced for the layered parallel computation for the FMO method.

GDDI controls the communicating processes using three kinds of scopes (corresponding to communicators in MPI). The first scope is DDI_WORLD, which communicates among all processes as the same as MPI_COMM_WORLD in MPI. The second one is DDI_GROUP, where the processes are divided into several groups and the communications are limited within each group. The last one is DDI_MASTERS, which communicates among masters of all groups and is responsible for the inter-group parallelization.

In the two-level parallel algorithm, each subsystem energy calculation is associated with a group and the computation is parallelized within each group with the DDI_GROUP scope. After all subsystem energy calculations, the energy is summed up among the masters of all groups with the DDI_MASTERS scope.

Figure 6.17 shows the pseudocode for the above procedure. For better load balancing, the subsystems are sorted with the number of basis functions before the calculation. Line 7 judges whether the subsystem should be calculated in my group. Each MP2 calculation can be parallelized with the standard MPI or MPI/OpenMP hybrid parallel algorithm within each group.

This two-level parallel DC-based post-HF method was implemented for the MP2 calculation and its efficiency was measured using T2K-Tsukuba computer (Fig. 6.18).

```

1: Sort subsystems by # basis functions in the subsystem in descending order
2: Call DDI_SCOPE (DDI_GROUP)
3: Call GDDICOUNT (-1, MYJOB)
4: EMP2TOT ← 0
5: Loop isub=1, nsub ; GDDI-parallelized loop over sorted subsystems
6:   Call GDDICOUNT (0, MYJOB)
7:   If (MYJOB=TRUE) Then
8:     EMP2 ← [MP2 correlation energy of isub subsystem]
              (applying fine-grained parallelization)
9:     EMP2TOT ← EMP2TOT + EMP2
10:   End If
11: End Loop
12: Call GDDICOUNT (1, MYJOB)
13: Call DDI_SCOPE (DDI_MASTERS)
14: Call DDI_GSUMF (EMP2TOT)
15: Call DDI_SCOPE (DDI_WORLD)

```

Fig. 6.17 Pseudocode for the two-level parallel DC-MP2 calculation with GDDI [47]

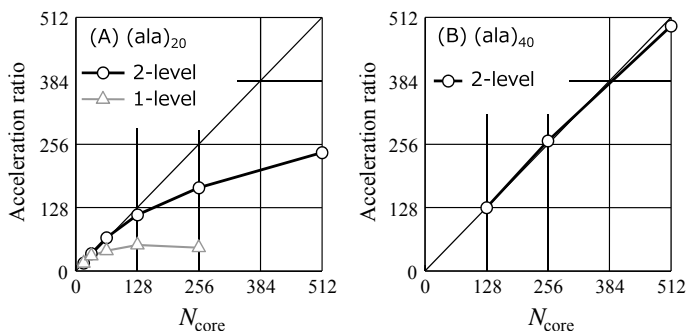


Fig. 6.18 Parallel efficiency of the DC-MP2 calculation. The results for **A** β -strand alanine 20 mer and **B** 40 mer. The times were measured with T2K-Tsukuba computer

Looking at the left chart showing the results for alanine 20 mer, the saturation of the efficiency with respect to the number of processes obviously become later by adopting the two-level parallel algorithm. For larger systems, 40 mer shown in the right chart, much higher efficiency can be confirmed.

6.11 Development of New Quantum Chemical Calculation Program

6.11.1 Outline of SMASH Program

On the basis of the experiences described in Sect. 6.9, a newly developed program is SMASH (Scalable Molecular Analysis Solver for High-performance computing) [48]. It has been distributed under the Apache 2.0 open-source license since September 2014, and the latest version is 2.2.0 as of 2019. Energy and geometry optimization calculations of the HF, DFT, MP2 methods are available. Computers equipped with scalar processors from PC clusters to supercomputers including the K Computer are supported.

It was developed with the Fortran 90/95 language considering hybrid MPI/OpenMP parallelization from the design stage. All the large data of electronic correlation calculations are distributed and stored in memory, not on disk. Frequently used calculation routines such as ERIs are modularized, making it possible to reduce development costs and making it easy to port to other programs.

• int2elec(twoeri, exijkl, coijkl, xyzijkl, nprimijkl, nangijkl, nbfiijkl, maxdim, mxprsh, threshex)	
twoeri	Two-electron repulsion integrals (Output)
exijkl	Exponents of basis functions (Input)
coijkl	Coefficients of basis functions
xyzijkl	Atom coordinates
nprimijkl	Numbers of primitive functions
nangijkl	Degrees of angular momenta (s=0, p=1, d=2,...)
nbfiijkl	Numbers of basis functions (s=1, p=3, d=5or6,...)
maxdim	Dimension of two-electron integral array
mxprsh	Dimension of primitive function array
threshex	Threshold of exponential calculation ($\exp(-x^2)$)

Fig. 6.19 Atomic orbital electron repulsion integral routine

6.11.2 Atomic Orbital Electron Repulsion Integral Calculation Routine

As shown in Fig. 6.19, the ERI calculation routine is structured to pass all the data as arguments. Since the derivative of an ERI is a linear combination of ERIs with different coefficients and angular momenta, the routine is reusable in derivative calculations. The necessary procedure is to pass appropriate coefficients and angular momenta as the arguments and to add the obtained values to appropriate elements of the gradient array. For example, the derivatives of $(p_x s | s s)$ are

$$\begin{aligned}
 \partial(p_x s | s s) / \partial X_a &= -2\alpha_a(\mathbf{d}_{xx} s | s s) + (s s | s s) \\
 \partial(p_x s | s s) / \partial Y_a &= -2\alpha_a(\mathbf{d}_{xy} s | s s) \\
 \partial(p_x s | s s) / \partial Z_a &= -2\alpha_a(\mathbf{d}_{xz} s | s s)
 \end{aligned}
 \tag{6.51}$$

and then what to do is to calculate $(d s | s s)$ and $(s s | s s)$.

6.11.3 Hybrid MPI/OpenMP Parallelization

The parallel algorithm described in Sect. 6.9.3 is adopted for HF and DFT calculations in the SMASH program, and a new algorithm for MP2 calculations based on the idea in Sect. 6.9.2 is developed considering hybrid parallelization and memory storage for intermediate data (Fig. 6.20). AO indices are distributed among processes up to the second transformation, and MO indices are distributed after the third transformation. All calculations within a process are also parallelized by OpenMP or the BLAS library. The huge data after the second transformation are distributed and stored in memory.

Fig. 6.20 Hybrid MPI/OpenMP parallel algorithm for MP2 energy calculation

```

do  $\mu\lambda$  (AO index pair) Distributed by MPI
!$OMP parallel do schedule(dynamic,1)
do  $\sigma$ 
ERI Calc. ( $\mu\nu|\lambda\sigma$ ) (all  $\nu$ )
First Trans. ( $\mu i|\lambda\sigma$ ) (all  $i$ )
enddo
!$OMP end parallel do
Second Trans.(dgemm) ( $\mu i|\lambda j$ ) (all  $i \geq j$ )
end do  $\mu\lambda$ 
do ij (MO index pair) Distributed by MPI
MPI_sendrecv ( $\mu i|\lambda j$ )
Third Trans.(dgemm) ( $\mu i|bj$ ) (all  $b$ )
Fourth Trans.(dgemm) ( $ai|bj$ ) (all  $a, b$ )
MP2 Energy Calc.
end do ij
call mpi_reduce(MP2 Energy)

```

6.11.4 Performance of SMASH

Several benchmark calculations of the SMASH program were performed using the K Computer (2.0GHz, 8 cores/node). The DFT (B3LYP) energy calculation of $(C_{150}H_{30})_2$ (cc-pVDZ basis set (4500 dimensions), 16 SCF cycles, Fig. 6.21) achieved a speedup of 50,000 and a CPU efficiency of 13% on 100,000 cores, which took 154 s [49]. The diagonalization (LAPACK, DSYEVD routine) is parallelized only within a node and is performed for one overlap matrix of the basis functions and two Fock matrices, and the time is 35 s on any number of cores. This is a main factor of relatively low parallel efficiency on 50,000 and 100,000 cores. It is necessary to further improve the parallel performance by introducing libraries that are parallelized among processes such as ScaLAPACK and EigenExa [50]. On the other hand, since a DFT energy gradient calculation does not include a matrix diagonalization calculation, even if the number of cores increases, the parallelization efficiency of the B3LYP

Fig. 6.21 Speedup of B3LYP energy calculation on K computer

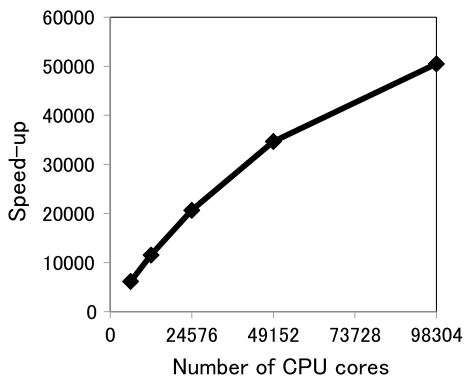


Table 6.11 B3LYP energy gradient calculation time (s) and speedup on K computer

Number of cores	1024	4096	8192	16,384
Time	402.0	101.2	50.8	25.5
Speedup	1024.0	4067.7	8103.3	16143.1

Table 6.12 MP2 energy calculation time (s) and speedup

Number of cores	4608	6912	9216	18,432
Time	152.5	105.7	83.4	46.9
Speedup	4608.0	6648.2	8425.9	14983.4

Table 6.13 Number of B3LYP/cc-pVDZ geometry optimization cycles (initial geometry: HF/STO-3G)

	Cartesian	Redundant
Luciferin (C ₁₁ H ₈ N ₂ O ₃ S ₂)	63	11
Taxol (C ₄₇ H ₅₁ NO ₁₄)	203	40

energy gradient calculation for C₁₅₀H₃₀ (with the cc-pVDZ basis (2250 dimensions), shown in Table 6.11) is almost 100%.

The intermediate data volume of the MP2 energy calculation for C₁₅₀H₃₀ (with the 6-31G(d) basis set (2160 dimensions), shown in Table 6.12) is huge as 1.1 TB, but it can be executed by distributedly storing them in memory of many nodes. Despite the total amount of 1.1 TB MPI_Sendrecv communication, the calculation achieved a speedup of 15,000 and an execution time of 47 seconds on 18,400 cores.

In geometry optimization calculations, the reduction of the number of optimization cycles are significant as well as the acceleration of energy gradient calculations. The number of optimization cycles can be reduced to 1/5 or 1/6 by the introduction of the redundant coordinate that uses molecular bond length, angle, and torsion information compared with the simple Cartesian coordinate, and thus whole geometry optimization calculations are drastically accelerated (Table 6.13).

6.12 Summary

Many researchers have been able to easily perform quantum chemical calculations due to the developments of new theories, calculation methods, and software, and the improvement of computer performance. It became possible to handle real systems and it became the stage where sufficient accuracy was obtained in comparison with the experiment and analysis. In the future, the need to calculate larger molecules at high speed with high accuracy will be expected to become larger and larger. In order

to make full use of the performance of the computer in the future, though it takes time and effort, we have to not only tune source codes but redo from fundamental things such as new formula derivations and algorithm developments with understanding the computer system.

We also focused on the quantum chemical calculation method for large systems based on the fragmentation and discussed the algorithms used therein. Due to the computational cost for the direct diagonalization of Fock matrix scaling $O(N^3)$, it was found that some ingenuity is necessary for treating huge systems, beyond the use of supercomputers such as the K Computer. In addition, a non-parallelized algorithm may become a problem even if the computational time is very short, as in the case of Fermi level determination in the DC-DFTB method. In such a case, a seemingly inefficient method may work efficiently when the computation is parallelized. Furthermore, it was shown that the hierarchical parallel computation algorithms such as the DC-MP2 method with GDDI are very effective especially in the fragment-based methods.

Efforts to reduce development costs in this field such as the development of common foundation libraries for quantum chemical calculations will be a major challenge from now on.

6.13 Exercise

1. Using the symmetry of Γ described in Eq. (6.12), the storage of Γ for 100 AOs can be reduced from $100 \times 100 \times 100 \times 100 = 100$ million element array to $\sim 1/8$ order of magnitudes. Calculate the correct number of unique elements in the array when storing without waste, and store the value of $\mu + \nu + \lambda + \sigma$ to such array GAMMA. Make a program to get and show any specified element of GAMMA.
2. Both the FMO and DC methods are implemented in GAMESS. Download GAMESS from the following webpage and compile it.
<https://www.msg.chem.iastate.edu/GAMESS/>
exam37.inp and exam44.inp in the test calculations are the sample inputs for the FMO calculation of water trimer and DC calculation of hydrogen fluoride hexamer, respectively. Perform them and confirm their computational results. Compute the hydrogen fluoride hexamer with the FMO method and compare the result with that of the DC method (DC calculation of water trimer will be meaningless because the entire system will be included in subsystems). Confirm the buffer-size dependence of the DC method by changing the size of buffer region (BUFRAD in \$DANDC and RBUFCR in \$DCCORR).
3. Download the SMASH source code from the web site, and execute “make” to compile the source code.
4. Execute the SMASH program with the different numbers of MPI processes and threads, and compare their computational times.

References

1. J.B. Foresman, Æ. Frisch, *Exploring Chemistry with Electronic Structure Methods*, 3rd ed. (Gaussian Inc., 2016)
2. <https://ccportal.ims.ac.jp/>
3. T. Tsuneda, *Density Functional Theory in Quantum Chemistry* (Springer, 2014)
4. T. Helgaker, P. Jørgensen, J. Olsen, *Molecular Electronic-Structure Theory* (Wiley, 2000)
5. A. Szabo, N.S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory* (Dover, 1996)
6. C.A. White, M. Head-Gordon, *J. Chem. Phys.* **105**, 5061 (1994)
7. M.C. Strain, G.E. Scuseria, M.J. Frisch, *Science* **271**, 51 (1996)
8. E. Schwegler, M. Challacombe, *J. Chem. Phys.* **105**, 2726 (1996)
9. A.D. Becke, *J. Chem. Phys.* **88**, 2547 (1988)
10. P. Pulay, *Chem. Phys. Lett.* **73**, 393 (1980)
11. P. Pulay, *J. Comput. Chem.* **3**, 556 (1982)
12. E.R. Davidson, *J. Comput. Phys.* **17**, 87 (1975)
13. G.B. Bacskay, *Chem. Phys.* **61**, 385–404 (1981)
14. D.H. Sleeman, *Thoret. Chim. Acta* **11**, 135–144 (1968)
15. H.B. Schlegel, *J. Comput. Chem.* **3**, 214–218 (1982)
16. (a) C.G. Broyden, *J. Inst. Math. Appl.* **6**, 76–90 (1970); (b) R. Fletcher, *Comput. J.* **13**, 317–322 (1970); (c) D. Goldfarb, *Math. Comput.* **24**, 23–26 (1970); (d) D.F. Shanno, *Math. Comput.* **24**, 647–656 (1970)
17. P. Pulay, G. Fogarasi, *J. Chem. Phys.* **96**, 2856–2860 (1992)
18. M. Dupuis, J. Rys, H.F. King, *J. Chem. Phys.* **65**, 111–116 (1976)
19. J.A. Pople, W.J. Hehre, *J. Comput. Phys.* **27**, 161–168 (1978)
20. S. Obara, A. Saika, *J. Chem. Phys.* **84**, 3963–3974 (1986)
21. L.E. McMurchie, E.R. Davidson, *J. Comput. Phys.* **65**, 218–231 (1978)
22. M. Head-Gordon, J.A. Pople, *J. Chem. Phys.* **89**, 5777–5786 (1988)
23. K. Ishida, *Int. J. Quant. Chem.* **59**, 209–218 (1996)
24. K. Ishimura, S. Nagase, *Theor. Chem. Acc.* **120**, 185–189 (2008)
25. M.W. Schmidt, K.K. Baldrige, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S.J. Su, T.L. Windus, M. Dupuis, J.A. Montgomery, *J. Comput. Chem.* **14**, 1347–1363 (1993)
26. J. Baker, P. Pulay, *J. Comput. Chem.* **23**, 1150–1156 (2002)
27. K. Ishimura, P. Pulay, S. Nagase, *J. Comput. Chem.* **27**, 407–413 (2006)
28. K. Ishimura, K. Kuramoto, Y. Ikuta, S. Hyodo, *J. Chem. Theory Comput.* **6**, 1075–1080 (2010)
29. G. Chaban, M.W. Schmidt, M.S. Gordon, *Theor. Chem. Acc.* **97**, 88–95 (1997)
30. M. Svensson, S. Humbel, R.D.J. Froese, T. Matsubara, S. Sieber, K. Morokuma, *J. Phys. Chem.* **100**, 19357–19363 (1996)
31. A. Warshel, M. Karplus, *J. Am. Chem. Soc.* **94**, 5612–5625 (1972)
32. (a) Y.S. Lee, W.C. Ermler, K.S. Pitzer, *J. Chem. Phys.* **67**, 5861–5876 (1977); (b) Y. Ishikawa, G. Malli, *J. Chem. Phys.* **75**, 5423–5431 (1981)
33. (a) P.J. Hay, W.R. Wadt, *J. Chem. Phys.* **82**, 270–283 (1985); (b) W.R. Wadt, P.J. Hay, *J. Chem. Phys.* **82**, 284–298 (1985); (c) P.J. Hay, W.R. Wadt, *J. Chem. Phys.* **82**, 299–310 (1985)
34. P. Fuentealba, H. Preuss, H. Stoll, L.V. Szentpaly, *Chem. Phys. Lett.* **89**, 418–422 (1982)
35. W.J. Stevens, H. Basch, M. Krauss, *J. Chem. Phys.* **81**, 6026–6033 (1984)
36. R.P. Hosteny, T.H. Dunning, R.R. Gilman, A. Pipano, I. Shavitt, *J. Chem. Phys.* **62**, 4764–4779 (1975)
37. J.M. Foster, S.F. Boys, *Rev. Mod. Phys.* **32**, 300–302 (1960)
38. J.L. Whitten, *J. Chem. Phys.* **58**, 4496–4501 (1973)
39. D.G. Fedorov, K. Kitaura (ed.), *The Fragment Molecular Orbital Method: Practical Applications to Large Molecular Systems* (CRC Press, 2009)
40. M. Kobayashi, H. Nakai, *Linear-Scaling Techniques in Computational Chemistry and Physics* (Springer, 2011), pp. 97–127

41. R.P. Brent, in *Algorithms for Minimization Without Derivatives*, chap. 4 (Prentice-Hall, 1973)
42. D. Porezag, Th Frauenheim, Th Köhler, R. Kaschner, *Phys. Rev. B* **51**, 12947 (1995)
43. M. Elstner, D. Porezag, G. Jungnickel, J. Elsner, M. Haugk, T. Frauenheim, S. Suhai, G. Seifert, *Phys. Rev. B* **58**, 7260 (1998)
44. H. Nishizawa, Y. Nishimura, M. Kobayashi, S. Irle, H. Nakai, *J. Comput. Chem.* **37**, 1983 (2016)
45. H. Nakai, *Chem. Phys. Lett.* **363**, 73 (2002)
46. R.K. Nesbet, *Adv. Chem. Phys.* **14**, 1 (1969)
47. M. Katouda, M. Kobayashi, H. Nakai, S. Nagase, *J. Comput. Chem.* **32**, 2756 (2011)
48. K. Ishimura, SMASH, <http://smash-qc.sourceforge.net/>
49. K. Ishimura, in *AIP Conference Proceedings*, vol. 1702, 090053 (2015)
50. <https://www.r-ccs.riken.jp/labs/lpnctr/projects/eigenexa/>

Index

A

- Ab initio calculation, 160
- Accompanying Coordinate Expansion (ACE) method, 178
- Adjacent communication, 6, 12, 14–20, 27, 45, 53, 54, 57–59, 63, 80, 133
- Amdahl's law, 7, 19
- Anderson mixing, 171
- Approximate second-order SCF method, 182
- Atomic Orbital (AO), 96, 164, 165, 167, 168, 172, 174, 178, 180, 181, 193, 196, 199
- Auxiliary function, 185

B

- Bandwidth, 26, 31, 35–37, 42, 45, 46, 72, 75, 82, 176
- Basis function, 66, 93–99, 101, 104–106, 111, 160, 164, 165, 167, 169, 174, 178–180, 188, 194, 197
- Basis set, 98, 100, 160, 161, 164, 165, 178, 179, 181, 197, 198
- B/F value, 8, 9, 21, 22, 30–37, 42, 46, 68, 71–75, 81, 82, 87
- Binary tree structure, 107–109
- BLAS, 32, 68, 69, 175, 180, 196
- Block division, 53, 64, 72, 74
- Bond-order potential method, 101
- Brent method, 190
- Brillouin's theorem, 173
- Broyden–Fletcher–Goldfarb–Shanno (BFGS) method, 174

C

- Cache, 4, 5, 8, 9, 21, 22, 27–29, 31, 33–38, 42–46, 69, 72–74, 76, 81, 82, 84, 86, 87, 127, 128, 132, 137–139, 142, 144, 146, 175
- Cache blocking, 5, 21, 29, 33
- Cache miss, 82, 86
- Calculation block, 12–14, 53, 71, 75
- Calculation unit cell, 120, 134, 136
- Catch miss, 144, 175
- Cauchy–Schwarz inequality, 168
- Coloring, 31, 32, 69, 84–86
- Communication bandwidth, 26
- Communication block, 12–14
- Configuration Interaction (CI) method, 163, 172
- CONQUEST, 160
- Continued fraction representation, 111
- Core electron, 184
- Coulomb interaction, 21, 90, 91, 95, 185
- Coulomb term, 167
- Coupled Cluster (CC) method, 163, 193
- Critical directive, 177
- Cut off, 94, 95, 120, 121, 125, 143, 145, 146, 149, 169, 174, 185, 187
- Cyclic division method, 74

D

- Data structure, 21, 118, 128, 132–137, 139, 140, 142, 177
- Davidson method, 173, 178
- DC-DFTB-K, 192
- DC-DFTB method, 190, 199
- Debugging, 179
- Dense matrix, 175, 184

Density fitting method, 185
Density Functional Theory (DFT), 21, 43, 44, 50, 66, 89–91, 93, 96, 98, 99, 111, 113, 161–163, 165, 168, 169, 190, 195–197
Density Functional Tight-Binding (DFTB) method, 190, 191
Density matrix, 94, 99, 100, 105, 111, 165, 166, 168, 169, 186–188
Density-matrix method, 99
DGEMM, 21, 32, 46, 58, 59, 68, 180, 181
Diagonalization, 51, 66–68, 104, 105, 162, 163, 165, 169, 172, 173, 175, 184, 189, 191, 197, 199
Direct Inversion of the Iterative Subspace (DIIS), 171
Direct Inversion of the Iterative Subspace (DIIS) method, 171, 174
Divide-and-Conquer (DC) method, 187–191, 193, 199
Domain decomposition, 6, 7, 24, 42, 45, 68, 70, 80, 84, 129

E

Effective Core Potential (ECP), 184
EigenExa, 197
Electron correlation calculation, 174, 175, 180, 184, 185, 187, 188, 193
Electron correlation energy, 193
Electron Repulsion Integral (ERI), 162, 163, 165, 167, 169, 178, 179, 185, 191, 196
Electrostatic interaction, 117, 121, 123, 133, 134, 137, 142–145, 152, 187
Element-by-element method, 77, 78, 80–82, 84, 86
Elongation method, 140, 143, 144
Energy density analysis, 193
Ewald method, 121, 126, 134
Exchange-correlation energy, 90, 92
Exchange-correlation functional, 92, 98, 99, 170, 190
Exchange-correlation potential, 91, 93
Exchange term, 167, 169, 170

F

Fast Fourier Transform (FFT), 66, 67, 69, 94, 121, 125, 126, 145, 147, 149, 150
Fast Multipole Method (FMM), 117, 121, 126, 133, 134, 136, 137, 144, 151, 154, 155, 156, 169, 185, 189, 190, 192

FEMTECK, 99
Finite element method, 21, 22, 77, 80, 96
First-order reduced density matrix, 94, 99
First-principles calculation, 160, 164
Floating-point registers, 29, 30, 46
Fock matrix, 164–168, 171, 179, 181–183, 189, 190, 199
Force field parameter, 138, 139, 141, 174
Fragment Molecular Orbital (FMO) method, 185–187, 189, 194, 199
FrontFlow/Blue (FFB), 38, 42, 44, 45, 69, 70, 77–82, 85, 87
Frozen core approximation, 184

G

GAMESS, 161, 179, 181, 182, 187, 194, 199
Gaussian, 96, 126, 160
Gauss-type function, 165
Generalized eigenproblem, 169
Generalized Gradient Approximation (GGA), 92, 93, 95, 99
Global communication, 6, 12–14, 16–20, 24–26, 45, 49, 50, 53–55, 57, 58, 61–63, 67, 79
Gram-Schmidt orthogonalization, 32, 33, 51, 52, 56, 68, 98
Graphics Processing Unit (GPU), 126
Green function, 94, 99–101, 103–105, 111, 112

H

Hamiltonian, 52, 56, 66, 102, 103, 105, 112, 122, 163–165, 170, 173, 185, 186, 188–191
Hardware barrier, 46, 47
Hartree–Fock, 94, 163, 181
Hartree–Fock approximation, 163
Hartree–Fock method, 162, 163
Hartree potential, 91, 94
Head–Gordon–Pople method, 178
Hot spot, 121, 128, 132, 140, 142, 145
Hybrid parallelization, 125, 126, 131, 133, 146, 176, 182, 196

I

Implicit method, 77, 78
Integral transformation, 163, 172, 193

K

Kohn-Sham (KS) equation, 66, 67, 90–92, 94, 95, 111, 186

- Kohn-Sham (KS) orbital, 93–95
 Krylov subspace, 103–106, 110, 185
- L**
- Lanczos algorithm, 100, 101, 103, 104
 LAPACK, 175, 197
 Large Eddy Simulation (LES), 44, 77
 Latency, 27, 28, 81, 129, 140, 176
 LatticeQCD, 42, 44
 LDL¹ decomposition, 111
 Lennard-Jones (LJ) interaction, 120
 Level 1 (L1) cache, 28, 29, 31, 46, 72, 73, 76, 81, 82, 84, 86, 87, 127, 137, 139, 142
 Level 2 (L2) cache, 28, 31, 46, 72–74, 81, 127
 Level shift method, 174
 Linear-scaling method, *see* $O(N)$ method
 LINPACK benchmark, 47
 List access, 21, 22, 33, 44, 45, 69, 81, 82, 84, 86, 134
 Load balancing, 180
 Load imbalance, 6, 14, 19, 20, 27, 49, 140
 Local Density Approximation (LDA), 50, 92, 93, 95
 Localized Molecular Orbital (LMO) method, 184
 Local orbital, 95, 98, 99
 Low-order scaling method, 111, 112
- M**
- Many-core, 182
 Matrix–matrix multiplication (Multiplication of matrix), 180
 Matrix–matrix products, 8, 9, 21, 28, 32, 33, 42–44, 66, 68, 175
 Matrix–vector product, 9, 32, 33, 37, 71, 72, 74, 75, 78–83
 McMurchie–Davidson (MD) method, 178, 179
 Memory bandwidth, 31, 35–37, 42, 45, 72, 75, 82
 Memory bank, 3, 4
 Memory wall problem, 1, 3–5, 9
 Message Passing Interface (MPI), 38, 48, 57–59, 61, 67, 80, 87, 109, 125–137, 139, 140, 144, 146, 150, 167, 175, 176, 180–182, 194–197, 199
 Metadata, 134–137, 139, 140, 142
 Molecular Dynamics (MD), 21, 27, 43, 44, 109, 117–120, 122–126, 129, 132–134, 137, 141, 143, 145, 147, 150, 169, 178, 179
 Molecular Dynamics (MD) calculation, 27, 117–121, 123, 125, 126, 129, 132–134, 137, 141–143, 150, 156, 169
 Molecular integral, 172, 173
 Molecular Orbital (MO), 164, 184
 Møller–Plesset perturbation (MP) method, 163
 MPI_Allreduce, 57–59, 61, 80, 87, 181
 MPI process, 127, 130, 133, 134, 136, 146, 150, 176, 199
 MPI rank, 80, 182
 MPI_Sendrecv, 130, 131, 150, 198
 MP2 method, 172, 180, 193, 195
 Multipole expansion, 131, 139, 151, 152, 154
 Multipole to Local (M2L), 136–141, 151, 155
- N**
- Nearsightedness of electron, 93, 94, 99, 103, 111, 113
 Nesbet’s theorem, 193
 NICAM, 42, 43
 Nonuniform Memory Access (NUMA), 127
 NTChem, 160, 161
- O**
- Obara–Saika method, 178
 ONETEP, 99
 ONIOM method, 184
 $O(N)$ Krylov subspace method, 105, 106, 110
 OpenMP, 38, 125, 126, 128, 133, 140–143, 146, 150, 167, 168, 175–177, 181–183, 194–197
 OpenMX, 160, 161
 Order- N method $O(N)$ method, 89, 90, 93–95, 99, 100, 106, 110, 113, 114, 185, 187
 Overlap integral, 95, 164
- P**
- Parallel efficiency, 19, 106, 118, 125, 126, 147, 180, 192, 195, 197
 Parallelization efficiency, 7, 19, 38, 118, 126–128, 131, 141, 142, 149, 150, 176, 197
 Parallelization ratio, 19

Particle Mesh Ewald (PME) method, 145
 Pauli's exclusion principle, 164, 167
 Peak performance, 14, 30, 36, 42–45, 64, 69,
 72, 76, 82, 85–87, 161, 162
 Perturbation method, 163, 180
 PHASE, 42, 43, 45, 49, 64, 66–69
 Poisson equation, 94
 Pople–Hehre (PH) method, 178, 179
 Post-HF method, 162, 163, 172, 194
 Prefetch, 5, 27, 28, 46, 73, 74
 Private, 177, 183
 Private directive, 140, 177
 Processing block, 12, 14, 26, 59, 62, 63, 66
 Pseudopotential, 96, 97, 99

Q

Quadratically convergent SCF method, 174
 Quantum chemical calculation, 21,
 159–162, 164–167, 169, 170, 174,
 178, 195, 198, 199
 Quantum chemistry, 93, 96, 160, 163, 165,
 173, 176, 177
 Quantum Mechanics/Molecular Mechanics
 (QM/MM) method, 184
 Quasi Newton–Raphson method, 174

R

Ratio of peak performance, 69, 76, 85
 Recurrence, 4, 5, 69, 84, 101, 111, 112, 178,
 179
 Recursion method, 100–102, 104
 Recursive bisection method, 106–108
 Reduction directive, 177
 Redundant coordinate, 174, 198
 Register, 29, 30, 34, 37, 46, 127, 128, 132
 Register blocking, 29
 Resolution of Identity (RI) method, 185
 RISC, 3, 5
 Roof-line model, 36
 Roothaan method, 164
 RSDFT, 42, 44, 45, 49–56, 58, 59, 61, 63–68
 Rys polynomial method, 178

S

ScaLAPACK, 56, 59, 69, 197
 Scalar architecture, 3–5, 45
 SCF calculation, *see* self-consistent field calculation
 Schrödinger equation, 163
 Second-order MP (MP2) method, 172, 180,
 193, 195, 199

Seism3D, 42, 43, 45, 69–74, 76
 Self-Consistent Charge (SCC), 187, 191,
 192
 Self-Consistent Field (SCF), 50–52, 64, 65,
 69, 92, 95–97, 165–169, 171, 174,
 182, 184, 188, 197
 Self-consistent field calculation, 165
 Shared, 46, 47, 74, 80, 127, 128, 146, 176,
 177, 183
 Single Instruction Multiple Data (SIMD), 3,
 5, 30, 31, 46, 85, 175
 Slater determinant, 91, 164
 Slater type function, 165
 SMASH, 160, 161, 195–197, 199
 Software pipelining, 5, 85, 132
 Sparse matrix, 37, 71, 72, 74, 75, 79–83, 85,
 95, 112, 163, 172, 173
 STREAM benchmark, 47, 82
 Strong scaling, 15–17, 53, 63, 64, 133
 Subcell, 120, 134–137, 139, 141, 143,
 152–155
 Superscalar, 3

T

Thread parallelization, 27, 31, 72–75, 84, 86,
 175
 Tofu interconnect, 47, 48, 128, 130
 Tofu Network Interfaces (TNI), 47
 Tofu Network Router (TNR), 47
 Torus network, 47, 48, 62, 117, 128–131,
 133, 136, 148
 Tri-diagonal, 104, 112
 Two-electron integral, 167

V

Variational principle, 90, 92, 114
 Vector architecture, 3, 4
 Vectorization, 5, 85, 142, 143, 150
 Vector pipeline, 3
 Velocity Verlet method, 122–124
 Virial theorem, 98

W

Wannier function, 93–95, 99
 Weak scaling, 15–19, 58, 59, 76, 87

Z

Z-matrix, 161