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

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.

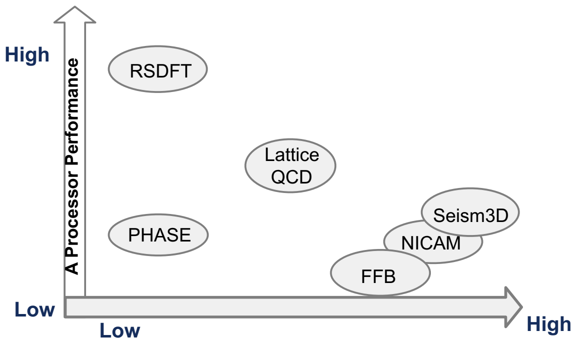


Fig. 3.1 Computer science characteristics of applications for the performance demonstration

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 $\times$ 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 (execution 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 availability[1] 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)

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

Theoretical performance: 2.3 GHz × 4 arithmetic × 4 cores × 4 sockets = 147.2 GFLOPS/node, 95.3 TFLOPS for the system
Main memory: 32 GB/node = 20.7 TB/system

(2) RICC parallel cluster

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

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

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

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

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.

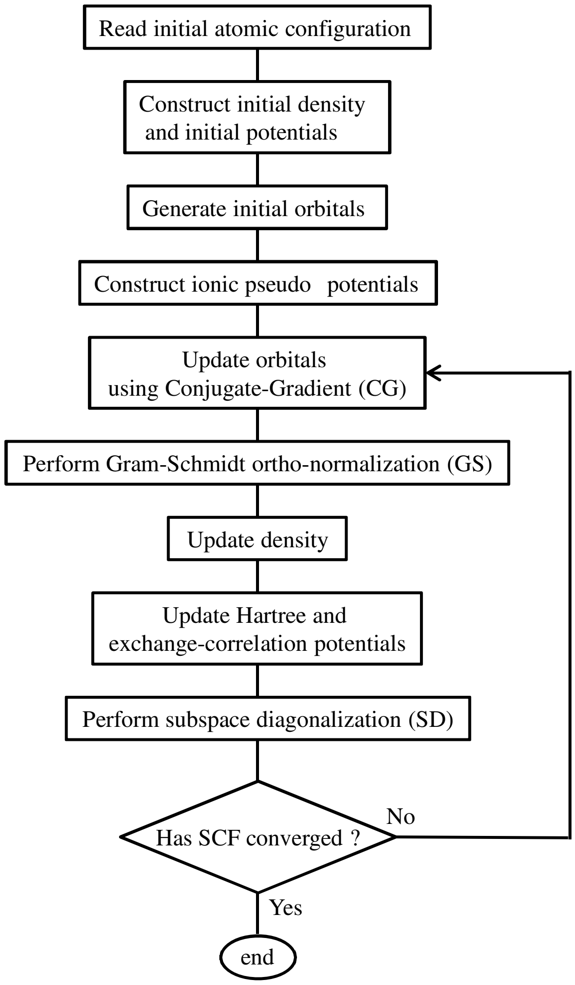

Read initial atomic configuration
Construct initial density and initial potentials
Generate initial orbitals
Construct ionic pseudo potentials
Update orbitals using Conjugate-Gradient (CG)
Perform Gram-Schmidt ortho-normalization (GS)
Update density
Update Hartree and exchange-correlation potentials
Perform subspace diagonalization (SD)
Has SCF converged ?
No
Yes
end

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*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 × 96 × 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

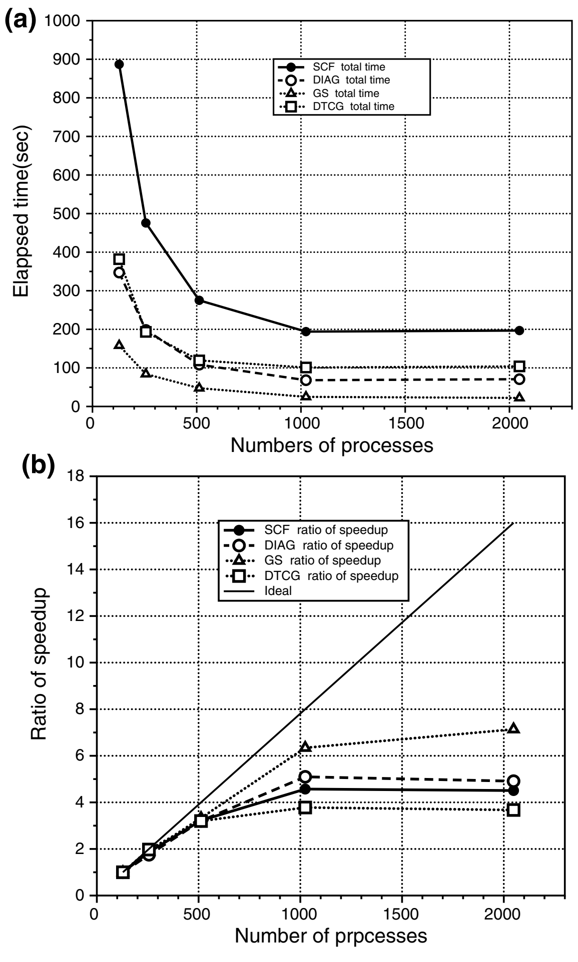


Fig. 3.3 Scalability of each calculation block of RSDFT

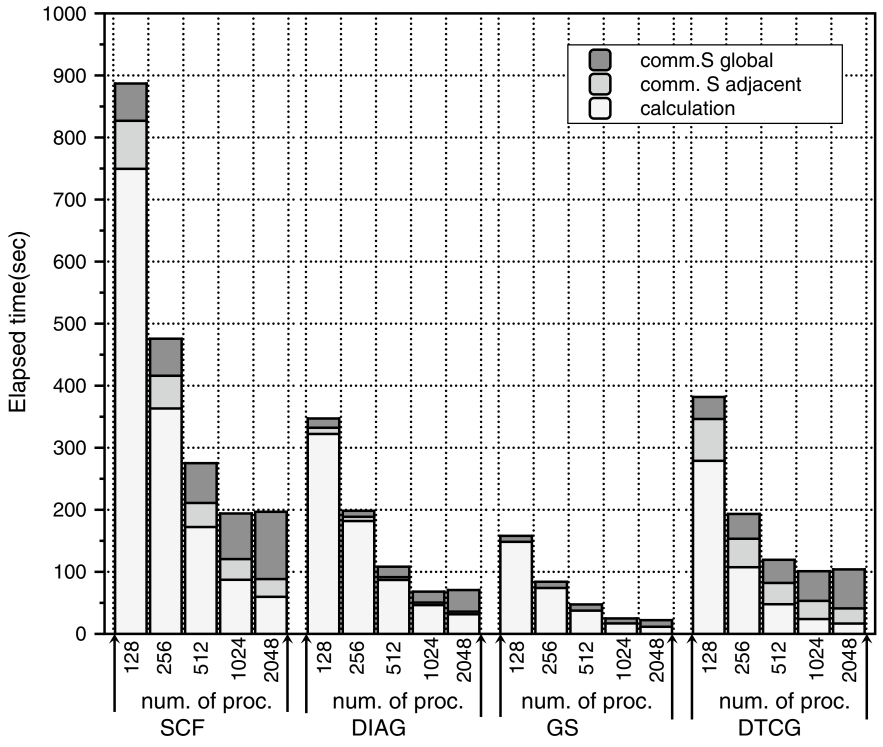


Fig. 3.4 Scalability of each calculation/communication block of RSDFT

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 DTCG, 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, DTCG 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 DTCG, 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 $\frac{\langle\psi_m\|H_{KS}\|\psi_n\rangle}{\langle\psi_m\psi_n\rangle}$	$O(ML \times ML)$ $O(N^2)$	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)	$(H_{N\times N})(\overrightarrow{c_n}) = \varepsilon(\overrightarrow{c_n})$	$O(MB^3)$ $O(N^3)$		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

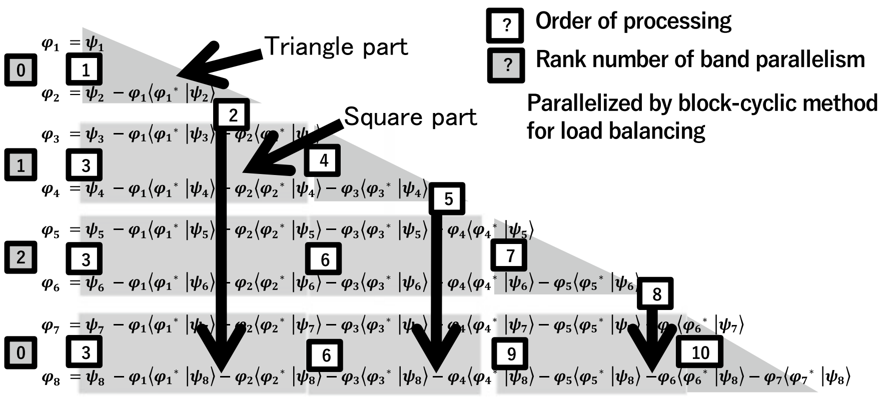

Triangle part
Square part
? Order of processing
? Rank number of band parallelism
Parallelized by block-cyclic method for load balancing
$\varphi_1 = \psi_1$
$\varphi_2 = \psi_2 - \varphi_1\langle\varphi_1^*|\psi_2\rangle$
$\varphi_3 = \psi_3 - \varphi_1\langle\varphi_1^*|\psi_3\rangle - \varphi_2\langle\varphi_2^*|\psi_3\rangle$
$\varphi_4 = \psi_4 - \varphi_1\langle\varphi_1^*|\psi_4\rangle - \varphi_2\langle\varphi_2^*|\psi_4\rangle - \varphi_3\langle\varphi_3^*|\psi_4\rangle$
$\varphi_5 = \psi_5 - \varphi_1\langle\varphi_1^*|\psi_5\rangle - \varphi_2\langle\varphi_2^*|\psi_5\rangle - \varphi_3\langle\varphi_3^*|\psi_5\rangle - \varphi_4\langle\varphi_4^*|\psi_5\rangle$
$\varphi_6 = \psi_6 - \varphi_1\langle\varphi_1^*|\psi_6\rangle - \varphi_2\langle\varphi_2^*|\psi_6\rangle - \varphi_3\langle\varphi_3^*|\psi_6\rangle - \varphi_4\langle\varphi_4^*|\psi_6\rangle - \varphi_5\langle\varphi_5^*|\psi_6\rangle$
$\varphi_7 = \psi_7 - \varphi_1\langle\varphi_1^*|\psi_7\rangle - \varphi_2\langle\varphi_2^*|\psi_7\rangle - \varphi_3\langle\varphi_3^*|\psi_7\rangle - \varphi_4\langle\varphi_4^*|\psi_7\rangle - \varphi_5\langle\varphi_5^*|\psi_7\rangle - \varphi_6\langle\varphi_6^*|\psi_7\rangle$
$\varphi_8 = \psi_8 - \varphi_1\langle\varphi_1^*|\psi_8\rangle - \varphi_2\langle\varphi_2^*|\psi_8\rangle - \varphi_3\langle\varphi_3^*|\psi_8\rangle - \varphi_4\langle\varphi_4^*|\psi_8\rangle - \varphi_5\langle\varphi_5^*|\psi_8\rangle - \varphi_6\langle\varphi_6^*|\psi_8\rangle - \varphi_7\langle\varphi_7^*|\psi_8\rangle$
0 1 2 3 4 5 6 7 8 9 10
0 1 2 0

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 ~ (NBKL1+1) * (NBLK1+1)	MB/NBLK * MBLK/NPB + Int(log(NBLK/NBLK1)) * (MB/NBLK/NPB)
	mpi_allreduce	mpi_real8	NBLK1~1	NBLK1 * (MB/NBLK/NPB)
	mpi_allreduce	mpi_real8	1	MB/NBLK * MB/NBLK/NPB + 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 × MB matrix, *NBSIZE* Column block size of MB × 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 × 48 × 48	19,200	512 (4 × 4 × 4 × 8)
Pattern2	1000	60 × 60 × 60	19,200	1000 (5 × 5 × 5 × 8)
Pattern3	1728	72 × 72 × 72	19,200	1728 (6 × 6 × 6 × 8)
Pattern4	4096	96 × 96 × 96	19,200	4096 (8 × 8 × 8 × 8)

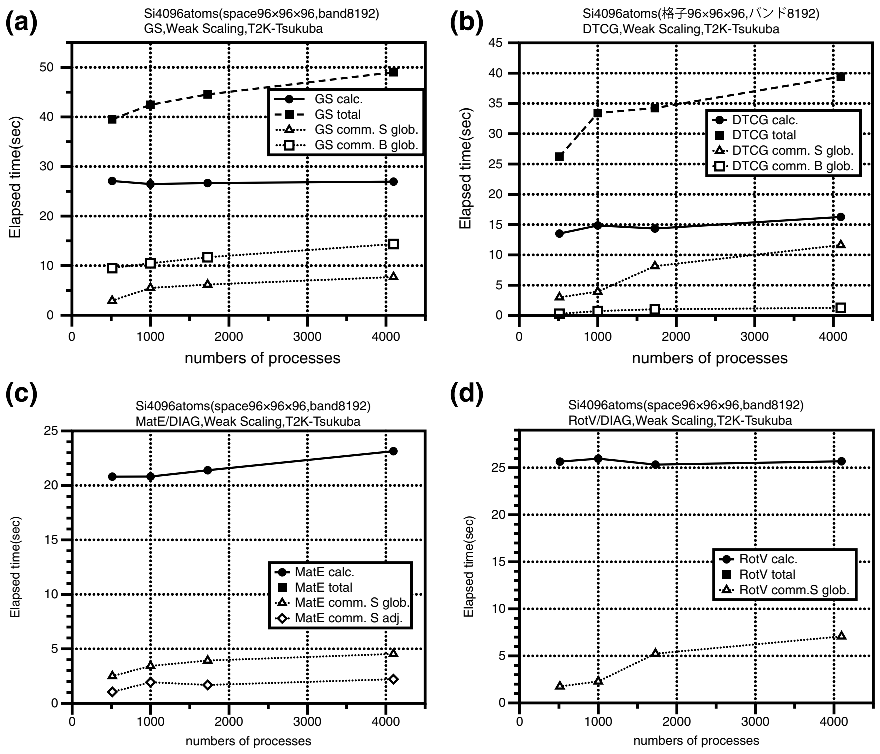


Fig. 3.6 Evaluation result of effect expanding the parallelization axis

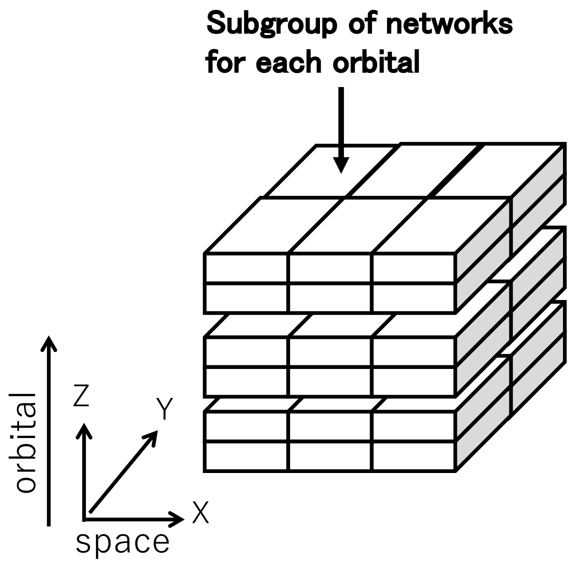


Fig. 3.7 Process mapping using for RSDFT on Tofu topology

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 file[2] 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

[2]This 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.

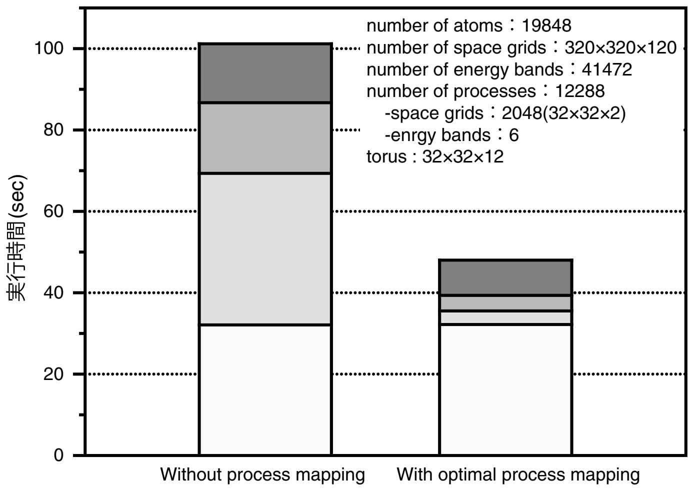


Fig. 3.8 Process mapping effect using for RSDFT on Tofu topology

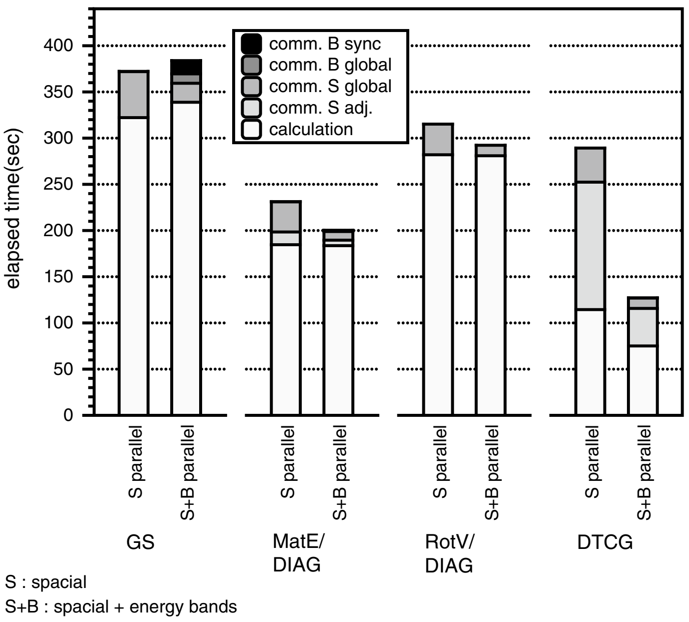


Fig. 3.9 Improving effect of communication performance by expanding the parallelization axis

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.

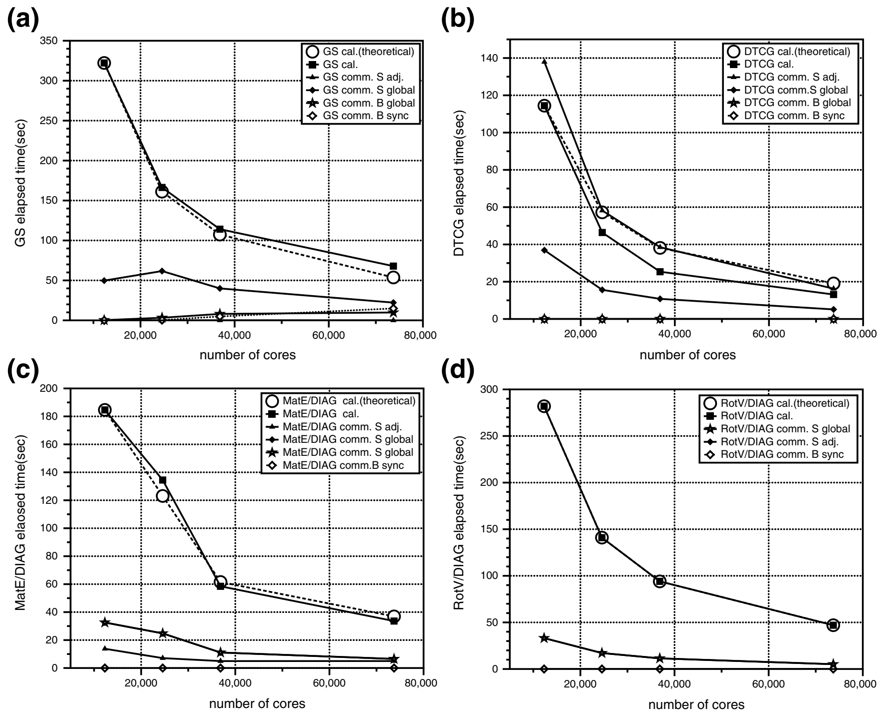


Fig. 3.10 Strong scaling performance of RSDFT with two-axes parallelization

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:

$$\mathrm{H}\psi_{ik}\left(\overrightarrow{G}\right) = \varepsilon_i \psi_{ik}\left(\overrightarrow{G}\right). \tag{3.4}$$

Here, ψ_{ik} represents the wave function defined in the periodic boundary condition, i represents the quantum number of the energy band, $\overrightarrow{G}$ represents the reciprocal lattice vector, and k represents the kth 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 $\overrightarrow{G}$. 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

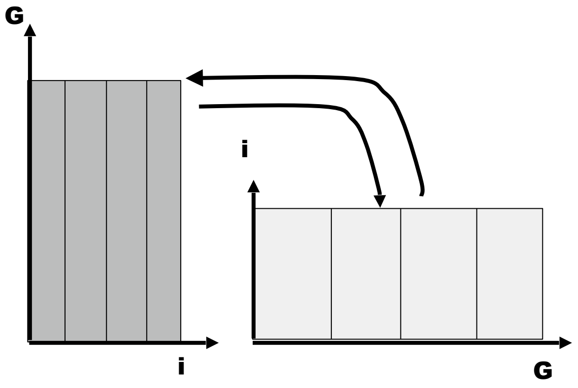
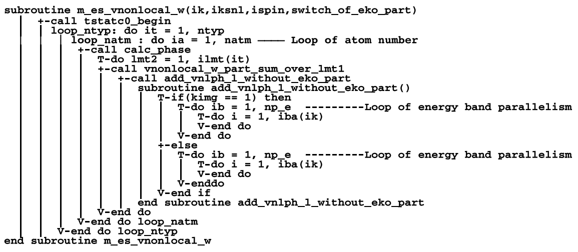


Fig. 3.11 Transpose operation in PHASE



```
subroutine m_es_vnonlocal_w(ik,iksnl,ispin,switch_of_eko_part)
 |   +-call tstatc0_begin
 |   |  loop_ntyp: do it = 1, ntyp
 |   |   |  loop_natm : do ia = 1, natm ---- Loop of atom number
 |   |   |   +-call calc_phase
 |   |   |   |  T-do lmt2 = 1, ilmt(it)
 |   |   |   |  +-call vnonlocal_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, iba(ik)
 |   |   |   |  |   |   |   |   |  V-end do
 |   |   |   |  |   |   |   |  V-end do
 |   |   |   |  |   |   |  +-else
 |   |   |   |  |   |   |   |  T-do ib = 1, np_e  ---------Loop of energy band parallelism
 |   |   |   |  |   |   |   |   |  T-do i = 1, iba(ik)
 |   |   |   |  |   |   |   |   |  V-end do
 |   |   |   |  |   |   |   |  V-enddo
 |   |   |   |  |   |   |  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_vnonlocal_w
```

Fig. 3.12 Loop structure of PHASE block 2

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 nonlocal 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

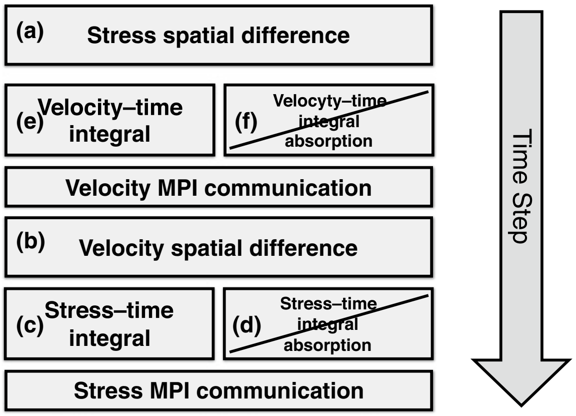
3.8.1 Overview of Seism3D

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


(a) Stress spatial difference
(e) Velocity–time integral
(f) ~~Velocyty–time integral absorption~~
Velocity MPI communication
(b) Velocity spatial difference
(c) Stress–time integral
(d) ~~Stress–time integral absorption~~
Stress MPI communication
Time Step

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

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

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

$$\rho\ddot{u}_z = \frac{\partial\sigma_{zx}}{\partial x} + \frac{\partial\sigma_{zy}}{\partial y} + \frac{\partial\sigma_{zz}}{\partial z} + f_{z.} \tag{3.7}$$

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

$$\sigma_{pq} = \lambda\big(e_{xx} + e_{yy+} + e_{zz}\big)\delta e_{pq} + 2\mu e_{pq}). \tag{3.8}$$

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

$$\boldsymbol{e_{xx}} = \frac{\partial \boldsymbol{u_x}}{\partial x}, \boldsymbol{e_y} = \frac{\partial \boldsymbol{u_y}}{\partial y}, \boldsymbol{e_z} = \frac{\partial \boldsymbol{u}_z}{\partial z}, \tag{3.9}$$

$$\boldsymbol{e_{xy}} = \frac{1}{2}\left(\frac{\partial \boldsymbol{u_x}}{\partial y} + \frac{\partial \sigma_{\boldsymbol{y}}}{\partial x}\right), \boldsymbol{e_{yz}} = \frac{1}{2}\left(\frac{\partial \boldsymbol{u_y}}{\partial z} + \frac{\partial u_z}{\partial y}\right), \ \boldsymbol{e_{zx}} = \frac{1}{2}\left(\frac{\partial \boldsymbol{u}_z}{\partial x} + \frac{\partial u_{\boldsymbol{x}}}{\partial z}\right). \tag{3.10}$$

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 $\times$ 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.

[3]In this case, we consider "add" as 3 and "multiply" as 2.

[4]In 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 KB $\times$ 4 = 64 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 XFILL[5] 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

[5]Control statement for streamlining array access without loading.

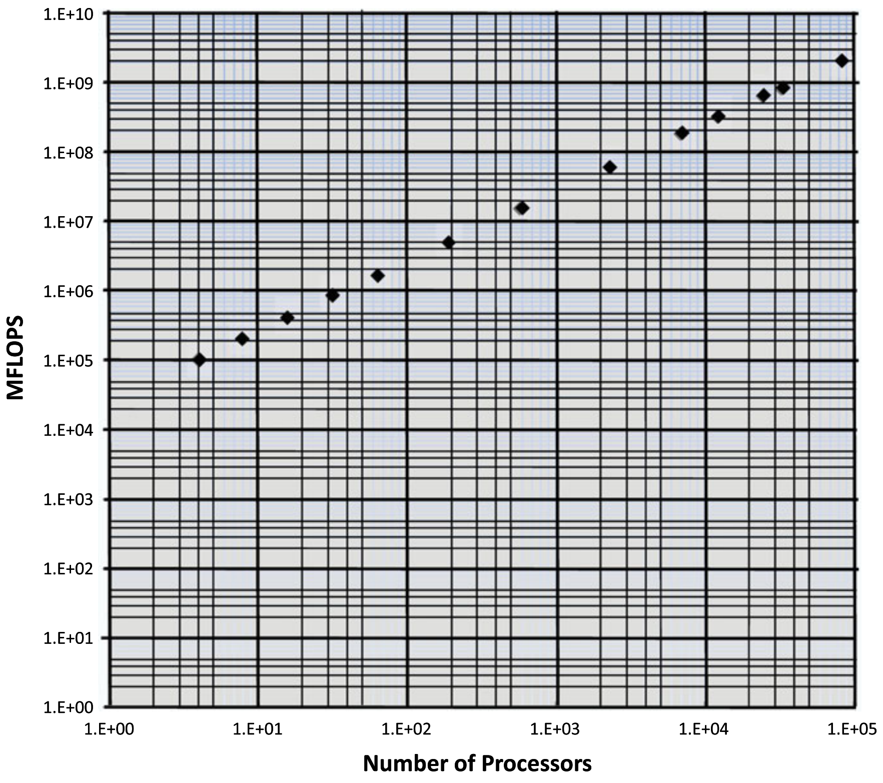


Fig. 3.18 Total performance of Seism3D

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

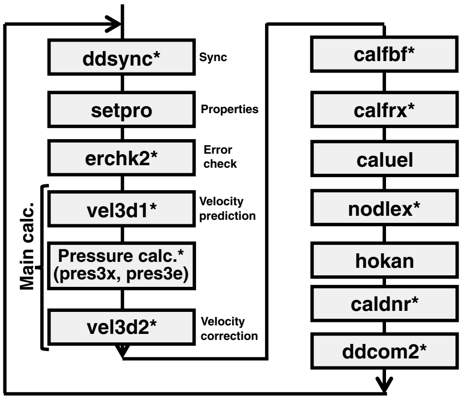


Fig. 3.19 Calculation flow of FFB

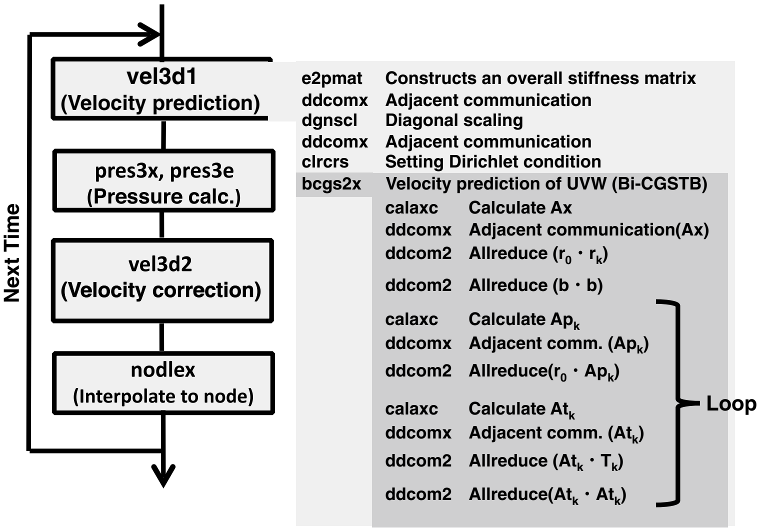

Next Time
vel3d1 (Velocity prediction)
pres3x, pres3e (Pressure calc.)
vel3d2 (Velocity correction)
nodlex (Interpolate to node)
e2pmat Constructs an overall stiffness matrix
ddcomx Adjacent communication
dgnscl Diagonal scaling
ddcomx Adjacent communication
clrcrs Setting Dirichlet condition
bcgs2x Velocity prediction of UVW (Bi-CGSTB)
calaxc Calculate Ax
ddcomx Adjacent communication(Ax)
ddcom2 Allreduce (r0 · rk)
ddcom2 Allreduce (b · b)
calaxc Calculate Apk
ddcomx Adjacent comm. (Apk)
ddcom2 Allreduce(r0 · Apk)
calaxc Calculate Atk
ddcomx Adjacent comm. (Atk)
ddcom2 Allreduce (Atk · Tk)
ddcom2 Allreduce(Atk · Atk)
Loop

Fig. 3.20 Calculation flow of velocity prediction

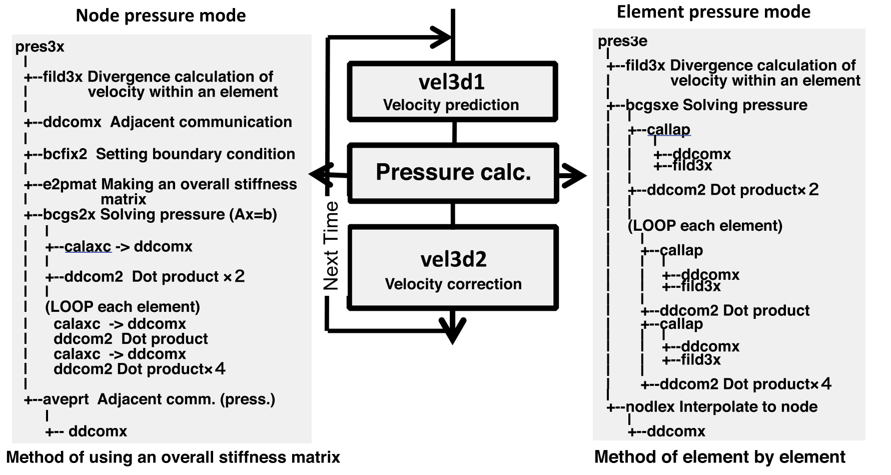

Node pressure mode
pres3x
+--fild3x Divergence calculation of velocity within an element
+--ddcomx Adjacent communication
+--bcfix2 Setting boundary condition
+--e2pmat Making an overall stiffness matrix
+--bcgs2x Solving pressure (Ax=b)
+--calaxc -> ddcomx
+--ddcom2 Dot product × 2
(LOOP each element)
calaxc -> ddcomx
ddcom2 Dot product
calaxc -> ddcomx
ddcom2 Dot product×4
+--aveprt Adjacent comm. (press.)
+-- ddcomx
Method of using an overall stiffness matrix
vel3d1
Velocity prediction
Pressure calc.
vel3d2
Velocity correction
Next Time
Element pressure mode
pres3e
+--fild3x Divergence calculation of velocity within an element
+--bcgsxe Solving pressure
+--callap
+--ddcomx
+--fild3x
+--ddcom2 Dot product×2
(LOOP each element)
+--callap
+--ddcomx
+--fild3x
+--ddcom2 Dot product
+--callap
+--ddcomx
+--fild3x
+--ddcom2 Dot product×4
+--nodlex Interpolate to node
+--ddcomx
Method of element by element

Fig. 3.21 Calculation flow of pressure calculation

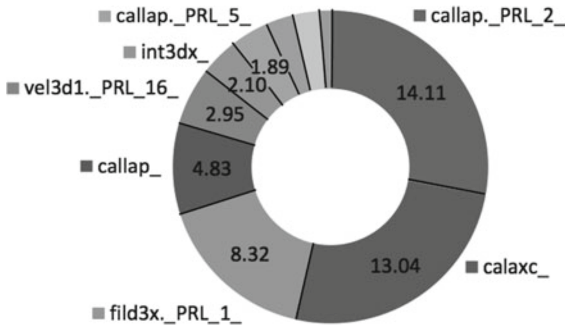
the center of the calculation of vel3d1. The center of the calculation of the element pressure mode is callap.

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) format[6] 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

$$\nabla \mathrm{p} = \frac{\partial p}{\partial x_i} = \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e. \tag{3.11}$$

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
```

[6]A 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

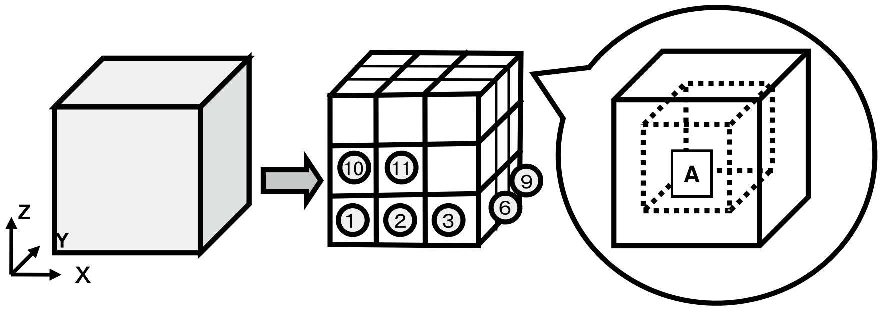


Fig. 3.26 Changing the ordering method of nodes

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

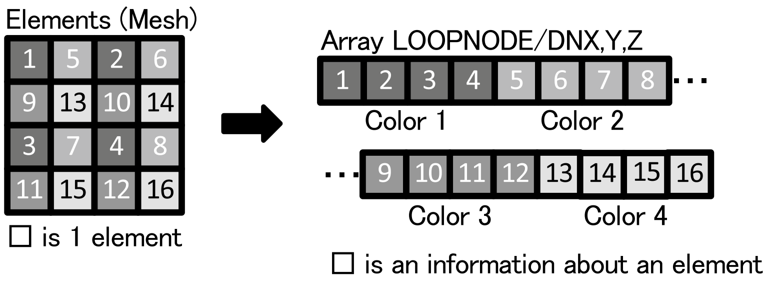


Fig. 3.27 Ordering element by coloring

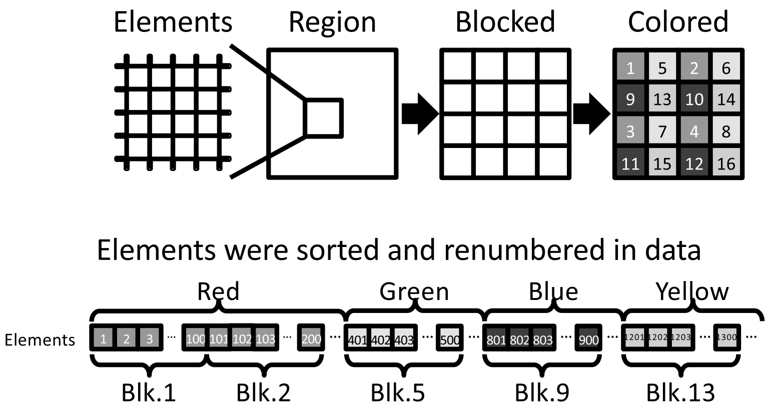


Fig. 3.28 Ordering block by coloring

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.

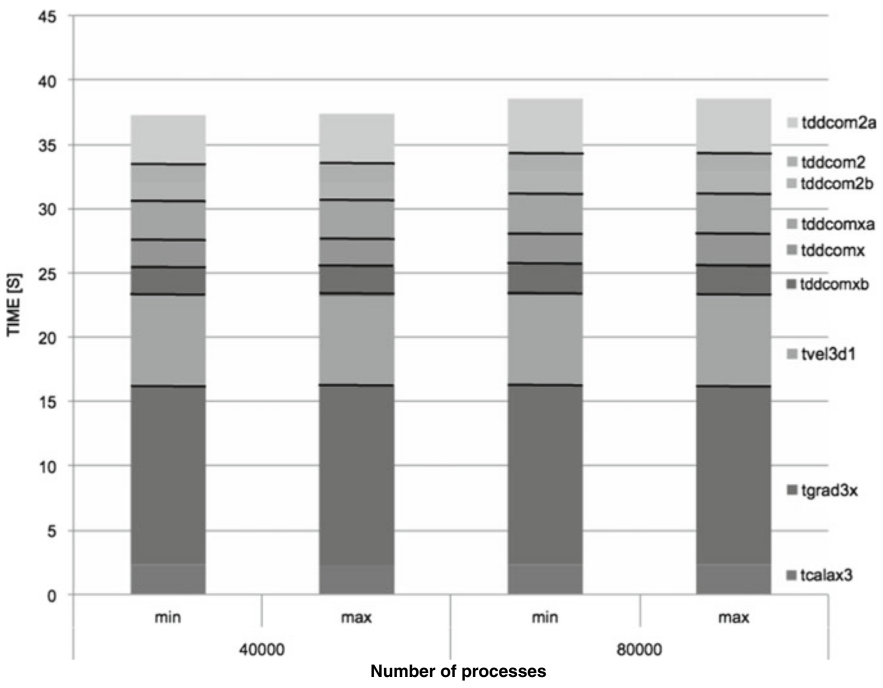


Fig. 3.30 Measurement result of scaling of FFBver7

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, IPSJ Trans. Adv. Comput. Syst. (ACS) (in Japanese) **6**, 31 (2013)
16. D. Guo, W. Gropp, IJHPCA **25**, 115 (2011)