Fugaku Codesign Report

- FLAGSHIP 2020 Project Technical Report -

FLAGSHIP 2020 Project
RIKEN Center for Computational Science (R-CCS), RIKEN

March, 2022

Preface

This technical report describes the FLAGSHIP 2020 project for the development and deployment of the next-generation Japanese flagship supercomputer named as Fugaku, which was carried out by RIKEN and supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) from FY2014 to FY2020.

In the development of Fugaku, the goal is to realize a general-purpose system that can solve a wide range of problems by the world's top level performance by around 2020. RIKEN carried out the basic design of the system from October 2014 to August 2015 with Fujitsu as a vender partner. After the evaluation of the design done by the committee organized by MEXT in January 2016, the detailed design and implementation, and the software development were carried out, and it signed a manufacturing contract with Fujitsu in March 2019. From December 2019 to May 2020, the hardware was manufactured and installed, then system adjustments were made, the public service started in March 2021, and the project was finished.

In this project, the "co-design" of the system and applications is a key to making the system power efficient and high performance. In the development of exascale large-scale parallel systems, it is necessary to improve the power efficiency of the system in order to obtain the maximum performance with the power supply capacity of our facility. As well as improving the system performance, we set the target application, designed the system while taking the characteristics of the actual application into account, and optimized the application to make effective use of the system.

This report summarizes our codesign efforts in the design of systems and applications.

List of Corresponding Authors

Chapter 1. Yutaka Ishikawa (R-CCS), Mitsuhisa Sato (R-CCS), Yuetsu Kodama (R-CCS), Miwako Tsuji (R-CCS)

Chapter 2. Mitsuhisa Sato (R-CCS), Yuetsu Kodama (R-CCS)

Chapter 3. Yutaka Ishikawa (R-CCS), Atsushi Hori (R-CCS), Masamichi Takagi (R-CCS), Takahiro Ogura (R-CCS), Balazs Gerofi (R-CCS)

Chapter 4. Hitoshi Murai (R-CCS)

Chapter 5. Toshiyuki Imamura (R-CCS)

Chapter 6. Hirofumi Tomita (R-CCS), Kazunori Mikami (R-CCS)

Chapter 7. Chigusa Kobayashi (R-CCS), Jaewoon Jung (R-CCS)

Chapter 8. Yutaka Ishikawa (R-CCS), Soichiro Suzuki (R-CCS), Satoshi Ito (U. Tokyo)

Chapter 9. Kohei Fujita (U. Tokyo)

Chapter 10. Hisashi Yashiro (NIES)

Chapter 11. Kazunori Mikami (R-CCS), Michio Katouda (R-CCS)

Chapter 12. Kazuo Minami (R-CCS), Kengo Miyamoto (R-CCS)

Chapter 13. Kazuo Minami (R-CCS), Kiyoshi Kumahata (R-CCS)

Chapter 14. Yoshifumi Nakamura (R-CCS)

The affiliations in parentheses are ones at the end of the project, or at the time of participation.

Acknowledgments

We would like to thank all the people who participated in and contributed to the FLAGSHIP2020 project at RIKEN, Fujitsu, Ministry of Education, Culture, Sports, Science and Technology, and collaborations.

Notes

- In this report, the unit of memory capacity is by a power expression of 2 using the unit such as MiB, GiB, etc. Other numbers are not expressed are power expressions of 10.
- The name of the system is indicated as ”Fugaku” decided in May 2019. If it should be distinguished from the system under development, the development code name ”Post-K” is used.

Contents

List of Figures

List of Tables

Chapter 1

FLAGSHIP 2020 project and Supercomputer "Fugaku"

1.1 Project Overview

The FLAGSHIP [1] 2020 Project was launched in April 2014. The missions were defined as follows:

- Building the Japanese national flagship supercomputer, the successor to the K computer, which was tentatively named the "Post-K".
- Developing a wide range of HPC applications that will run on the Post-K in order to solve the pressing societal and scientific issues facing our country.

The RIKEN Center for Computational Science (R-CCS) is charged with the research and development of the Post-K. For the system development, we have been working with Fujitsu since October 2014 after the vendor's selection phase.

Our government committee organized and selected nine Priority Issues and projects tackling these issues. The nine Priority Issues with the institutions selected to lead each project are listed in Table 1.1.

Figure 1.1 shows the schedule of the project. The first two years (JFY2014 and JFY2015) [2] were spent on the basic design. From JFY2016 to JFY2018, detailed design and implementation were conducted. A prototype system was built in the summer of 2018 at Fujitsu. The review committee of the Japanese government reviewed the prototype system and decided to build the Post-K. From JFY2019 to JFY2020, the hardware is manufactured, and the system is installed and tuned.

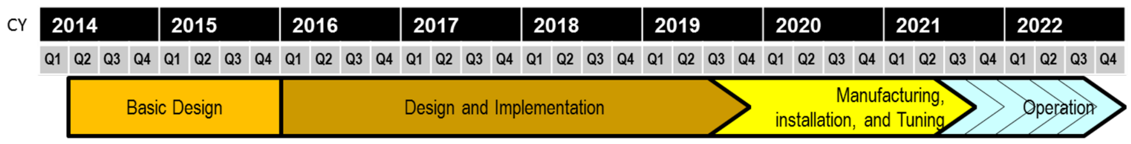


Figure 1.1: Schedule of FLAGSHIP 2020 Project

[1]FLAGSHIP: Acronym for Future LAtency core-based General-purpose Supercomputer with HIgh Productivity
[2]The Japanese fiscal year (JFY) starts from April.

Table 1.1: Nine Priority Issues with leading institutions

Health and longevity	
1. Innovative computing infrastructure for drug discovery	RIKEN Quantitative Biology Center
2. Personalized and preventive medicine using big data	Institute of Medical Science, the University of Tokyo
Disaster prevention / Environment	
3. Integrated simulation systems induced by earthquake and tsunami	Earthquake Research Institute, the University of Tokyo
4. Meteorological and global environmental prediction using big data	JAMSTEC, Center for Earth Information Science and Technology of Japan
Energy issues	
5. New technologies for energy creation, conversion / storage, and use	Institute for Molecular Science, National Institute of Neural Science
6. Accelerated development of innovative clean energy systems	School of Engineering, the University of Tokyo
Industrial competitiveness enhancement	
7. Creation of new functional devices and high-performance materials	The Institute of Solid State Physics, the University of Tokyo
8.Development of innovative design and production processes	Institute of Industrial Science, the University of Tokyo
Basic science	
9. Elucidation of the fundamental laws and evolution of the universe	Center for Computational Sciences, Tsukuba University

1.2 Project KPIs

At the beginning of the project, we defined the following three Key Performance Indicators (KPIs):

1. **Power-efficient system:** We set the maximum electric power supply capacity of our facility between 30 and 40 MW. We should maximize the system performance within this capacity. We set more than 15 GFlops/W as the target efficiency for the dgemm kernel.

2. **Effective performance of real applications:** We focus on the performance of real applications rather than that of benchmarks. We aimed at a (maximum case) speed improvement of one hundred times over the K computer for some applications. This should be accomplished through codesign of system development and target applications for the nine Priority Issues.

3. **Ease-of-use:** Since the system will be shared by a variety of users, the system should be easy to use.

We carried out the FLAGSHIP 2020 Project to develop the Japanese next-generation flagship supercomputer, the Post-K, recently named "Fugaku". We have designed an original manycore processor based on Armv8 instruction sets with the Scalable Vector Extension (SVE), an A64FX processor, as well as a system including interconnect and a storage subsystem with the industry partner, Fujitsu. In 2019, the name of the system was decided as "Fugaku". The delivery of the whole system was completed in May, 2020.

1.3 Specification of the Fugaku system

The Fugaku system is a large-scale distributed memory system with Fujitsu A64FX manycore processors connected via Tofu-D interconnect. Table1.2 describes the specification of Fugaku.

Table 1.2: Specification of Fugaku

Theoretical Peak Perf. of system (DP)	537 PF (boost mode) 488 PF (normal mode)
Number of compute nodes	158976 (24 x 23 x 24 x 12)
Network toplogy	6-dim mesh/torus (Tofu-D interconnect)
Total memory size	4.8 PiB
Storage capacity (1-st layer, SSD)	15.8 PB
Storage capactiy (2-nd layer, HDD)	150 PB
Node processor	Fujitsu A64FX
Si technology	7 nm FinFET
Proc. Freq.	2.2 GHz(boost mode) 2.0 GHz(normal mode)
Number of Proc. per node	1
Theoretical Peak Perf. of Node (DP)	3.3 TF (boost mode) 3.0 TF (normal mode)

1.4 Specifications of the A64FX Processor

The node processor is a single chip, named A64FX, which consists of 48 cores with 2 or 4 cores dedicated for OS activities, 32 GiB of HBM2 memory, TofuD Interconnect, and a PCI express controller, as shown in Table 1.3. The diagram, the photograph of die and package are shown in Figure 1.2 and 1.3.

The details of the microarchitecture are described in [2].

1.5 Specifications of the System and Storage

Each node has one A64FX chip connected with the TofuD Interconnect. The 1st-layer storage system consists of SSDs attached to SIO nodes allocated one for every 16 compute nodes. The 2nd-layer storage system is the global file system, which is a Luster-based parallel file system, FEFS, developed by Fujitsu. FEFS is connected to the system via the InfiniBand network.

The specifications of the 1st- and 2nd-layer storages are shown in Table 1.4.

A Linux kernel runs on each node. All system daemons run on two or four additional cores, called assistant cores. The CPU chip with two assistant cores is used on compute only nodes. The chip with four assistant cores is used on compute & IO nodes because such nodes service I/O functions requiring more CPU resources.

Table 1.3: Specification of the A64FX Processor

Component	Specification
Architecture	Armv8.2-A SVE (512 bits SIMD)
Core	48 cores for compute and 2/4 for OS activities
	Normal Mode: Freq: 2.0 GHz, DP: 3.072 TF, SP: 6.144 TF, HP: 12.288 TF
	Boost Mode: Freq: 2.2 GHz, DP: 3.3792TF, SP: 6.7584 TF, HP: 13.5168 TF
L1 Cache	64 KiB, 4 way, 256 GB/s(load), 128 GB/s (store) @2.0GHz
L2 Cache	8 MiB/CMG, Total 32MiB/16way, BW for Core: 128 GB/s (load), 64 GB/s (store) @ 2.0GHz
Memory	HBM2 32 GiB, BW for Chip 1024 GB/s
Interconnect	TofuD: 28 Gbps x 2 lane x 10 port, injection BW: 6.8GB/s x 6
I/O	PCIe Gen3 x 16 lane
Silicon Technology	7nm FinFET, CoWoS (Chip on Wafer on Substrate)[1] for HBM2

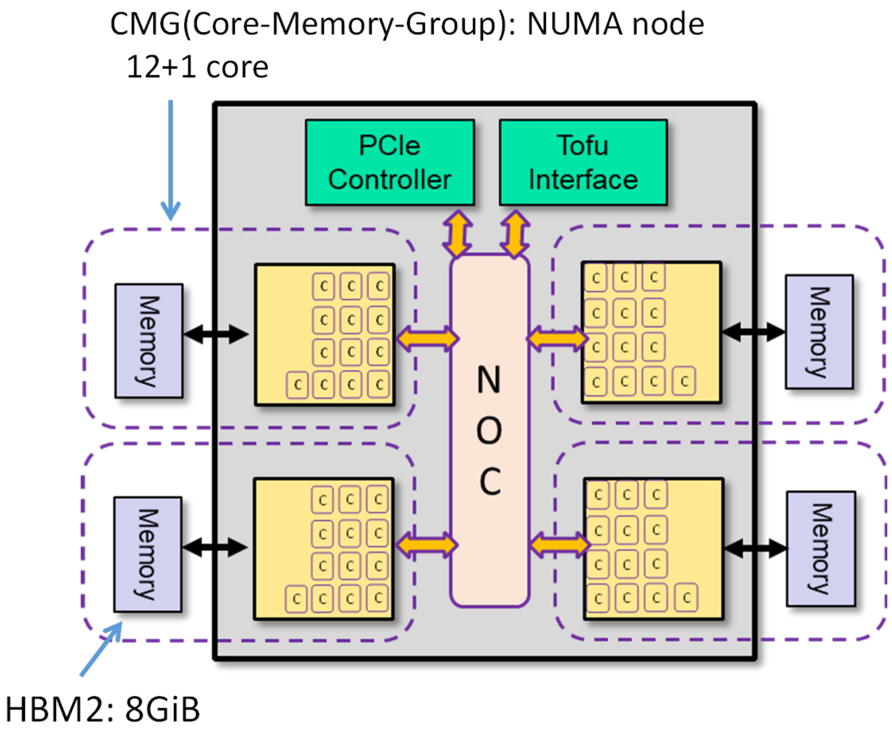


Figure 1.2: Block diagram of the A64FX processor

1.6 Benchmark Results

1.6.1 Basic Performance

We have confirmed the node performance for the basic kernels, more than 830 GB/s for the stream triad benchmark and more than 2.5 TFLOPS for the dgemm kernel with 90% efficiency.

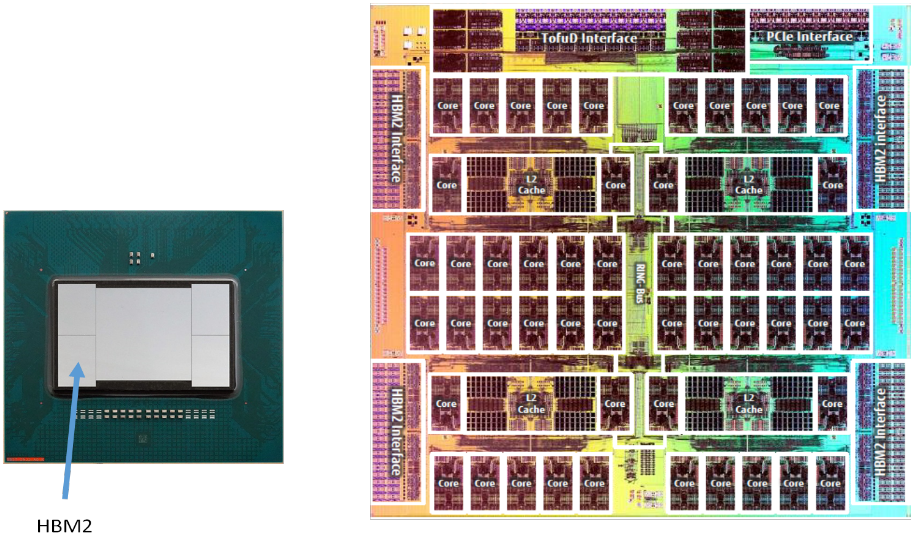


Figure 1.3: Chip package and die photograph of the A64FX processor

Table 1.4: Specifications of the 1st- and 2nd-layer Storages

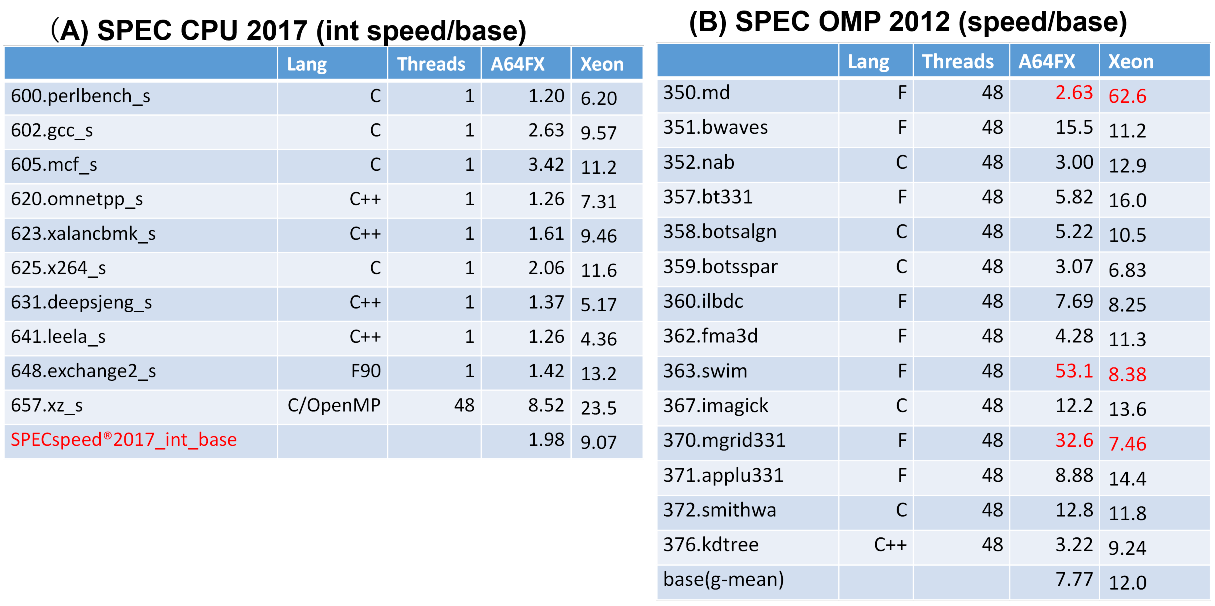
		Minimum Throughput	Measured Throughput
1st Storage	write	49 MB/s /node	125 MB/s /node
	read	113 MB/s /node	293 MB/s /node
2nd Storage	write	200 GB/s /volume	211 GB/s /volume
	read		220 GB/s /volume

Note: The 2nd storage is formed from 6 volumes.

The latency and bandwidth of the TofuD Interconnect are from 0.49 to 0.54 μs and 6.35 GB/s for a 1-MB put operation.

1.6.2 SPEC benchmark

SPEC benchmarks are well-known benchmarks suites for the evaluation of various kinds of computer systems. Figure 1.4 shows the results for SPEC CPU (int) and SPEC OMP. For our evaluation, Speed (an index for evaluating the performance of single task) and Base (metrics when the compile option common to all benchmarks) were taken. For all benchmarks, A64FX runs at 2.0GHz (normal mode). “Xeon” used for the SPEC CPU is Platinum 8168(Skylake), 2.7GHz, 24cores x 2 chip, turbo on. “Xeon” used for SPEC OMP is Platinum 8280(Cascade Lake), 2.7GHz, 28cores x 1chip, hyperthread on (56threads), turbo on. The benchmark programs of SPEC CPU (int) except 647.xz_s run by a single thread. As shown in the results, the performance of A64FX is about one-quarter of the performance of the Xeon processor. The reason for the low single thread integer performance is that the SIMD rate is low in SPEC CPU (int) and the frequency and the O3 resource are limited for the throughput-oriented architecture of A64FX. As for SPEC OMP, the performance of A64FX using 48 threads is about 65% of the performance of the Xeon processor using 56 threads (28 cores). For some programs such as 363.swim and 370.mgrid, A64FX archives extremely good performance thanks to the high memory bandwidth of the HBM2. Note that, although the performance of 350.md is very bad, the performance improvement has been confirmed by manual source code tuning such as in-lining and loop unrolling. Other SPEC results of the A64FX are published in [3].



(A) SPEC CPU 2017 (int speed/base)

	Lang	Threads	A64FX	Xeon
600.perlbench_s	C	1	1.20	6.20
602.gcc_s	C	1	2.63	9.57
605.mcf_s	C	1	3.42	11.2
620.omnetpp_s	C++	1	1.26	7.31
623.xalancbmk_s	C++	1	1.61	9.46
625.x264_s	C	1	2.06	11.6
631.deepsjeng_s	C++	1	1.37	5.17
641.leela_s	C++	1	1.26	4.36
648.exchange2_s	F90	1	1.42	13.2
657.xz_s	C/OpenMP	48	8.52	23.5
SPECspeed®2017_int_base			1.98	9.07

(B) SPEC OMP 2012 (speed/base)

	Lang	Threads	A64FX	Xeon
350.md	F	48	2.63	62.6
351.bwaves	F	48	15.5	11.2
352.nab	C	48	3.00	12.9
357.bt331	F	48	5.82	16.0
358.botsalgn	C	48	5.22	10.5
359.botsspar	C	48	3.07	6.83
360.ilbdc	F	48	7.69	8.25
362.fma3d	F	48	4.28	11.3
363.swim	F	48	53.1	8.38
367.imagick	C	48	12.2	13.6
370.mgrid331	F	48	32.6	7.46
371.applu331	F	48	8.88	14.4
372.smithwa	C	48	12.8	11.8
376.kdtree	C++	48	3.22	9.24
base(g-mean)			7.77	12.0

Figure 1.4: Results of SPEC Benchmarks

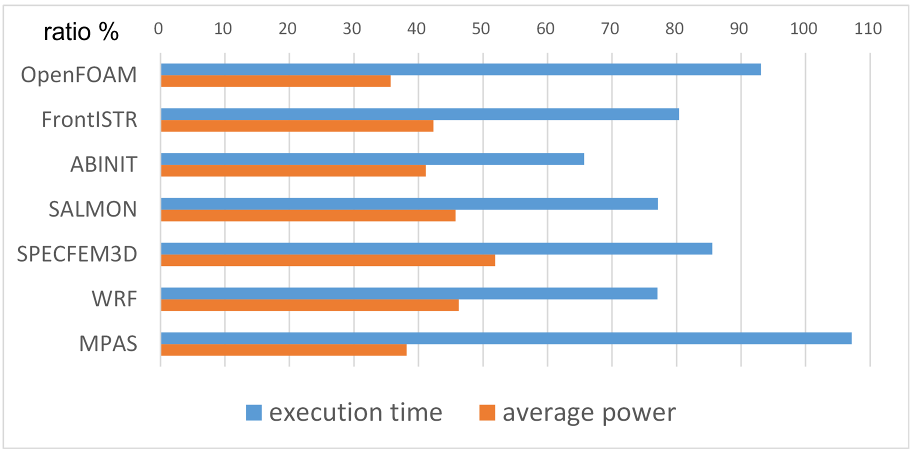


Figure 1.5: Performance and power efficiency of open-source applications (results are shown in %, relative to Intel Xeon (dual sockets))

1.6.3 Open-source HPC Applications

Several open-source scientific applications are ported and evaluated on A64FX.

Figure 1.5 shows the execution time and the average power of A64FX (2.2 GHz, single socket) relative to dual sockets of Xeon Platinum 8268 (Cascadelake, 2.90 GHz, 24 cores/socket) for several open-source scientific applications, described in Table 1.5. As the results demonstrate, the power consumption of A64FX is about half that of the Intel Xeon's in most of these applications, while the performance of A64FX is better.

Table 1.5: Descriptions and Parameters of Open-source Applications

Applications	Field	Ver.	Size	Model	Solver	MPI/ OpenMP	Measured Section
OpenFOAM [4]	CFD	1812	14 million meshes	Motor Bike	PCG	FlatMPI (48 procs)	Time integration loop
FrontISTR[5]	Structure Analysis	5	0.4 million meshes	Hinge	SSOR+CG	FlatMPI (48 procs)	Analysis section
ABINIT[6]	Material	8.10 .2	26244 FFT meshes	tmbt_3	GW/spectral method	FlatMPI (48 procs)	Analysis section
SALMON[7]	Ab-initio Light-Matter	1.2.1	1.0 million meshes	exercise _07_classic EM_lr	FDTD	FlatMPI (48 procs)	Analysis section
SPECFEM 3D[8]	Seismic	7.0.2	64 x 64	s362ani	SEM	Hybrid (4 procs x 12 threads)	Iteration loop
WRF[9]	weather	3.8.1	425 x 300 x 35	Conus 12km		Hybrid (4 procs x 12 threads)	Domain integration loop
MPAS[10]	weather	6.2	40962 meshes	The Jablonowski and Williamson baroclinic wave		Hybrid (24 procs x 2 threads)	Time integration loop

1.6.4 HPL(TOP500), HPCG, HPL-AI and Graph500

In June 2020, the Fugaku achieved HPL performance of 415.53 PFLOPS, using 396 racks (152,064 nodes, approximately 95.6% of the entire system) with a computing efficiency ratio of 80.87%, and ranked as the 1st position of the TOP500 list. It also took the first place in the ranking of HPCG, achieving 13,400 TFLOPS (360 racks, 138,240 nodes, approximately 87% of the entire system), while claiming the first position in the HPL-AI ranking with 1.421 EFLOPS using 330 racks (126,720 nodes, approximately 79.7% of the entire system). HPL-AI is a new benchmark that takes into account the capabilities of single-precision and half-precision arithmetic logic units used in artificial intelligence. It has taken the top spot on the Graph500 list, a ranking of the world's fastest supercomputers on data-intensive workloads, with the performance of a breadth-first search of a large scale graph 70,980 GTEPS, using 92,160 nodes (approximately 58% of the entire system), The achievement of remarkable records in these rankings demonstrates the overall high performance of Fugaku for a wide range of workloads.

In November 2020, the results of these benchmarks were updated by using the full system. The results are shown in Table 1.6.

Table 1.6: Benchmark Results using the full system of Fugaku

Benchmark	Measured	Peek Perf	Efficiency	(June 2020)
HPL	442.01 PF	537.21 PF	82.3%	(415.53 PF)
HPCG	16.00 PF	537.21 PF	3.0%	(13.4PF)
HPL-AI	2.00 EF	2.14 EF	93.2%	(1.42EF)
Graph500	102.95 Tteps			(70.98)

1.6.5 Green500

The successful achievement of our codesign effort is such that the prototype systems of A64FX processors took the 1st place at Green500[11] in November 2019. This performance measurement demonstrated that the A64FX processor has the highest energy efficiency, achieving 1.9995 PFLOPS for HPL compared to 2.36 PFLOPS in peak performance and 16.87 GFLOPS/W in performance per 1 watt of power consumption, exceeding the efficiency of GPUs. This corresponds to the first item in our KPIs, that is, a power-efficient system.

References

[1] *CoWoS (Chip-on-Wafer-on-Substrate) Services.* `https://www.tsmc.com/english/dedicatedFoundry/services/cowos.htm`.

[2] *A64FX Microarchtecture Manual.* `https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.0.pdf`.

[3] Yuetsu Kodama, Masaaki Kondo, and Mitsuhisa Sato. “Evaluation of SPEC CPU and SPEC OMP on the A64FX”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER).* 2021, pp. 553–561. DOI: `10.1109/Cluster48925.2021.00088`.

[4] H. Jasak, A. Jemcov, Z. Tukovic et al. *OpenFOAM:A C++ library for complex physics simulations.* International workshop on coupled methods in numerical dynamics, vol. 1000. IUC Dubrovnik, Croatia, pp.1–20, 2007.

[5] *FrontISTR : Open-Source Large-Scale Parallel FEM Program for Nonlinear Structural Analysis.* `https://gitlab.com/FrontISTR-Commons/FrontISTR`.

[6] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, et. al. *ABINIT: First-principles approach to material and nanosystem properties.* Computer Physics Communications 180, 2582 - 2615 (2009). Abinit Project home page: `https://www.abinit.org/`.

[7] *Scalable Ab-initio Light-Matter simulator for Optics and Nanoscience.* `https://salmon-tddft.jp/`.

[8] *SPECFEM3D GLOBE.* `https://geodynamics.org/cig/software/specfem3d_globe/`.

[9] *The Weather Research and Forecasting (WRF) Model.* `https://www.mmm.ucar.edu/weather-research-and-forecasting-model`.

[10] *The Model for Prediction Across Scales (MPAS).* `https://mpas-dev.github.io/`.

[11] *The Green500 Web site.* `https://www.top500.org/green500/`.

Chapter 2

Codesign of Archtecture

The "codesign" of the system and applications is a key to making it power efficient and high performance. We determined many architectural parameters by reflecting an analysis of a set of target applications provided by applications teams. In this section, the pragmatic practice of our codesign effort for "Fugaku" is described.

2.1 codesign for exascale computing

As a key strategy for this challenge, codesign, has received considerable attention in the HPC community for several years[12]. The term codesign has become popular in the development of mobile phone or embedded systems, where the two perspectives of hardware and software design are brought into a codesign process. In embedded systems, codesign sometimes includes "specialization" for particular applications. The codesign process of HPC must maximize the benefits to cover as many applications as possible. Codesign has been proposed as a methodology for the scientific application, software, and hardware communities to work together for designing future HPC systems.

In this chapter, we describe the pragmatic practice of our codesign effort in our project. According to the outcomes of feasibility study projects prior to the FLAGSHIP 2020 Project, we have chosen a huge-scale system with general-purpose manycore processors. With Fujitsu as a vendor partner, a new manycore processor supporting the Arm instruction set has been designed through codesign with the collaboration of architecture teams, software teams, and application teams. During the codesign process, we focused on not only energy efficiency, but also programmability for ease-of-use and efficient execution of real scientific applications with some compatibility with the K computer. As a result, the system has been proven to be a very power-efficient system, and it is confirmed that the performance of some target applications using the whole system is more than 100 times the performance of the K computer.

2.2 Codesign Methodology

2.2.1 Target Applications for Codesign

Under the codesign concept, R-CCS, Fujitsu, and the nine selected projects have been collaborating closely to design the architecture, system software, and applications so that the societal and scientific issues can be solved effectively and early achievements can be attained. Since increasing power consumption is a critical issue in the design of the next-generation large-scale supercomputer, it is important to make trade-offs between energy/power, cost, and performance by taking application characteristics into consideration.

In the codesign process, a set of target applications was provided from each area of the nine Priority Issues. Refer the chapter 6 for the detailed descriptions of the target applications. The reasons for choosing these applications as target applications are based on two criteria. One criterion is from the computational viewpoint, e.g., 1) the total target applications cover dense matrix operation and sparse matrix operation, and 2) they also cover the data access pattern (continuous, stride, random). The other criterion is from the viewpoint of scientific excellence. We selected each application and its problem size, discussing with members of the project for the post-K priority issues. For example, the target application in the meteorological field is a very ambitious problem, which estimates global atmospheric conditions in very high resolution (3.5 km) using a global cloud resolving model and the data assimilation technique of the localized ensemble transformed Kalman filter based on over 1000 ensembles.

System design has been performed using these applications as a target. For each application, the target performance is set relative to the K computer when using the whole system with a power budget under approximately 30 to 40 MW. The target performance is estimated by codesign tools in the basic design phase.

The target applications are also considered representatives of almost all of our applications in terms of computational methods and communication patterns relevant to designing architectural features. The table shows the points of the codesign process for each application. These target applications are used to set the performance goals of the system and are also used to verify that these goals have been achieved.

Moreover, the typical benchmarks kernels such as dgemm, HPL, and the stream benchmark were used for performance analysis.

2.2.2 Tools for Codesign

The following tools were used for the codesign process:

- **The performance estimation tool:** This tool, taking execution profile data of the latest Fujitsu supercomputer, FX100, as an input, enables the performance projection by a given set of architecture parameters. The performance projection is modeled according to the Fujitsu microarchitecture. This tool can also estimate the power consumption based on the architecture model. ‘This tool has been available since the initial phase of the codesign process.

- **Fujitsu in-house processor simulator:** We used an extended FX100 (SPARC) simulator and compiler, developed by Fujitsu, for preliminary studies in the initial phase, and an Armv8+SVE simulator and compiler afterwards. It runs at a frequency of roughly 100 kHz.

- **Hardware emulator:** The hardware emulator for the processor was used for logic design verification and accurate evaluation of performance and power consumption. It runs at a frequency of roughly 1 MHz and has been available since the later phase of the codesign process, after the circuit design was almost completed.

- **Gem5 simulator for the Post-K processor:** Gem5[13] is an open-source system-level processor simulator. The Post-K processor simulator[14] based on gem5 has been developed by R-CCS during the codesign process for architecture verification and performance tuning of software.

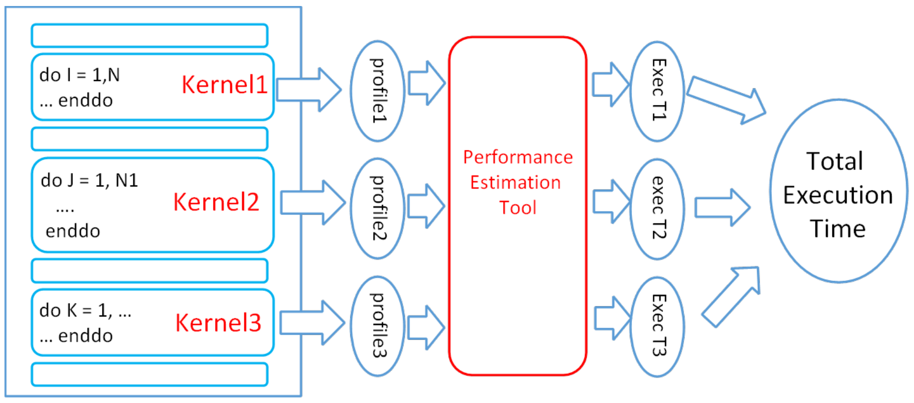


Figure 2.1: Usage of the performance estimation tool

2.2.3 Codesign Process: Design Space Exploration

As described in the next section, the basic architecture has been chosen as a manycore processor system by the feasibility studies before starting the project. Even for designing a manycore processor, there are many parameters and items to be determined in the design space exploration.

A fundamental problem is the scale of scientific applications that are expected to be run on the Post-K. Even our target applications are thousands of lines of code and are written to use complex algorithms and data structures. Although the processor simulators are capable of providing very accurate performance results at the cycle level, they are very slow and are limited to execution on a single processor without MPI communications between the nodes.

On other hand, our performance estimation tool is very useful since it enables performance analysis based on the execution profile taken from an actual run on the FX100 hardware. The FX100 hardware has a rich set of performance counters, including busy cycles for read/write memory access, busy cycles for L1/L2 cache access, busy cycles of floating-point arithmetic, and cycles for commited instruction. It enables the performance projection for a new set of hardware parameters by changing the busy cycles of functional blocks. The breakdown of the execution time (cycles) can be calculated by summing the busy cycles of each functional block in the pipeline according to the processor microarchitecture. For example, suppose the L2 cache access is a bottleneck. When doubling L2 cache bandwidth, the busy cycles of L2 cache access are halved and the execution time will be reduced if the time for L2 cache access is not hidden by other cycles in the pipeline. Since the execution time is estimated by a simple formula modeling the pipeline, it can be applied to a region of uniform behavior such as a kernel loop. Otherwise, the accuracy of the estimation would worsen. Furthermore, it is hard to take the impact of the O3 (out-of-order) resources into account in the performance estimation.

The first step of performance analysis is to identify kernels in each target application and insert the library calls to get the execution profile. As shown in Figure 2.1, the total execution time is calculated by summing the estimated execution time of each kernel. We repeated this process changing several architecture parameters for design space exploration.

Some important kernels were extracted as independent programs. These kernels can be executed by the cycle-level processor simulators for more accurate analysis. This enables analysis for a new instruction set and the effect of changing the O3 resources. These kernels were also used for the processor emulator for logic-design verification.

We used an analytical model for simple kernels, such as the HPL and dgemm kernel, and the stream benchmark. This is sometimes useful because the execution time can be calculated by a simple mathematical formula using some hardware parameters.

In the codesign process for the interconnect, the communication patterns were extracted, and we estimated the communication performance by an analytical model, such as the LogP network performance model.

2.3 Codesign of Post-K

In this section, we describe how we made several decisions on the architecture of the Post-K through our codesign process. Most codesign for the architecture was done during the basic design phase from 2014 to 2015. Since the production of the system was scheduled for 2019, the codesign process needed to be preceded by assessing what kinds of technologies for silicon fabrication and memory, SerDes, and IO interface standards would be available at the time of the production.

2.3.1 Basic Architecture Design by Feasibility Studies

Prior to the FLAGSHIP 2020 Project, the following feasibility study projects were carried out in order to investigate the basic design from 2012 to 2013:

- System study for the Next-generation "General-Purpose" Supercomputer, led by the University of Tokyo
- System study for Exascale Heterogeneous Systems with Accelerators, led by the University of Tsukuba
- System study for Memory-bandwidth Oriented Vector Architecture, led by Tohoku University
- Application study led by the RIKEN R-CCS

Based on these feasibility studies, the initial plan at the beginning (2014) was a combined system of a general purpose supercomputer and accelerators with a target performance goal of exaFLOPS. The development of the accelerator part was, however, canceled due to a budget problem. As a result, the basic architecture was decided as a large-scale system using only general-purpose manycore processors studied by the feasibility study project of the University of Tokyo.

The processor architecture suggested by the feasibility study was a manycore processor with 64 cores and two 512-bit-width SIMD arithmetic units for each core. It was reported that other configurations, such as wider SIMD arithmetic units or a dedicated small matrix multiply arithmetic, have been examined but not adopted due to low applicability and low efficacy for real applications. Since the evaluation was done by the configuration supporting one pipe of 1,024-bit-width SIMD and matrix multiply units together, the number of cores was reduced to 32 cores per chip, resulting in poor performance. For example, the execution time of the RS-DFT application by this configuration was increased by 28% compared to two pipes of 512-bit-width SIMD. Only 1,024-bit-width SIMD without matrix multiply units was not evaluated, but the difference from two pipes of 512-bit-width SIMD would be small by the estimation tool based on the profile.

2.3.2 Instruction Set Architecture and SIMD Instructions

The choice of the instruction set architecture was an important decision for architecture design. Instead of the SPARC instruction set[15] used for the K computer, Fujitsu offered the Armv8 instruction set with the Arm SIMD instruction set called the Scalable Vector Extension (SVE)[16] for the Post-K. They collaborated with Arm, contributing to the design of the SVE as a lead partner, and adopted the results in the processor architecture design. The processor is custom designed by Fujitsu using their microarchitecture as a backend of the processor core.

The Arm instruction-set architecture has been widely accepted by software developers and users not only for mobile processors, but also for HPC recently. For example, Cavium Thunder X2 is a processor designed for servers and HPC and used for several supercomputer systems, including Astra[17] and Isambard[18]. While the Intel x86 architecture is dominant in HPC, Arm processors are expected to be open to the possibilities and the diversity in the HPC community.

The SVE is an extended SIMD instruction set. The most significant feature of the SVE is uniform support for vector lengths from 128 bits to 2,048 bits. The SVE realizes Vector Length Agnostic (VLA) programming, as the name suggests, and does not depend on the vector length. We have decided to have two 512-bit-width SIMD arithmetic units, as suggested by the feasibility study. Our paper[19] using the gem5 simulator and McPAT also suggests that this configuration of SIMD provides a good balance between performance and energy consumption under the current O3 resources described later. Note that our implementation, A64FX processor, is the first processor supporting the SVE.

2.3.3 Processor Chip Configuration

Fujitsu proposed the basic structure of the manycore processor architecture. Each core has an L1 cache, and a cluster of cores shares an L2 cache and memory controller. This cluster of cores is called a Core-Memory Group (CMG). In a processor chip, CMGs are connected via a network-on-chip.

While other high-performance processors, such as those of Intel and AMD, have L1 and L2 caches in the core and share an L3 cache as a last-level cache, the core of our processor has only an L1 cache to reduce the die size for the core. In the case of the A64FX chip, the area size for the L1 cache occupies more than 10% of each core. If an additional other level of cache was implemented and required a similar area in each core, the die size would increase by at least more than 4%, estimated from the chip photograph. This impact is so large that it would be not acceptable from a cost perspective.

Based on this basic structure, we had to decide the following parameters:

- The number of cores in a CMG
- The number of CMGs in a chip
- How to connect cores to shared L2 in a CMG
- The number of ways, the size, and throughputs of the L1 and L2 caches
- The topology of network-on-chip to connect CMGs
- The die size of the chip
- The number of chips in a node

Our technology target for silicon fabrication was 7-nm FinFET technology. The die size of the chip is the most dominant factor in terms of cost. It is known that the cost of the chip

Table 2.1: Cost of Die and Package, Multi-Chip Module

#Cores x #PKG(p)/MCM(M)	#Dies	Die area	Die cost	PKG/MCM cost	Total
64 x 1p	1	1.00	0.82	0.18	1.00
32 x 2p	2	0.65	0.80	0.26	1.06
16 x 4p	4	0.38	0.73	0.30	1.03
32 x 2M	2	0.64	0.77	0.31	1.08
16 x 4M	4	0.37	0.71	0.34	1.05

Table 2.2: Kernel Performance by the 4 CMGs and 8 CMGs Configurations for 64 Cores

kernel	4 CMGs 1 proc/CMG 16threads/proc	4 CMGs 2 proc/CMG 8threads/proc	8 CMGs 1 proc/CMG 8threads/proc
Kernel (Adventure)	100%	112%	112%
Kernel (FFB)	100%	105%	110%
KernelA (NICAM)	100%	100%	100%
KernelC (NICAM)	100%	90%	90%
kernelD (NICAM)	100%	113%	122%

NOTE: The performance in this table is the average of small kernels extracted in the different ways from the kernel shown Table 2.4, 2.3

increases in proportion to the size and increases significantly beyond a certain size. Moreover, the yield of the chip becomes low as the size of the chip increases. One configuration is to use small chips and connect these chips by multi-chip module (MCM) technology. Recently, AMD has used this "chiplet" approach successfully. The advantage of this approach is that a small chip can be relatively cheap with a good yield. However, in the present case, the cost of MCM was deemed too high, and furthermore, a different kind of chip for the interconnect and IO must be made, resulting in even higher costs. Table 2.1 shows the cost of die and package, MCM, relative to the cost of 64 cores in 1 package at the time of basic design phase. The connection between chips on the MCM would also increase the power consumption.

Thus, our decision was to use a single large die containing some CMGs and the network interface for interconnect and PCIe for I/O connected by a network-on-chip. The size of the die was about $400mm^2$, which was reasonable in terms of cost for 7-nm FinFET technology.

In order to maintain a certain degree of compatibility with the K computer, we assumed that a commonly used programming model was OpenMP-MPI hybrid programming in which each MPI process multithreaded with OpenMP runs in a CMG. In order to share the L2 cache with cores in a CMG, we chose a cross-bar connection between L2 and cores.

We assumed that the total number of cores was 64 and examined the performance for 8 cores/CMG x 8 CMGs and 16 cores/CMG x 4 CMGs by running extracted kernels from the target applications. Table 2.2 shows the performance of some kernels by the 4 CMGs and 8 CMGs configurations for 64 cores. For the performance evaluation, the total L2 cache size was kept the same for both cases. The results evaluated by the performance estimation were as follows:

- For some kernels, the performance of 8 cores/CMG was better since the latency to the small L2 cache is reduced compared to the 16 cores/CMG configuration.

- We found that a few kernels ran slower on 8 cores/CMG because the cache hit ratio of the smaller cache was low.

If the number of cores in CMGs was less than 8, then the L2 cache size was too small. The number of cores in a CMG was examined by the analytical performance model of HPL.

Our conclusion was that 8 cores/CMG was better than 16 cores/CMG. When we made a decision to use the 7-nm FinFET technology of TSMC, we decided 48 cores (plus 4 cores) and 12 cores/CMG x 4 CMGs. Four CMGs provided 4 High-Bandwidth Memory (HBM) units as a main memory, as described later.

We have designed the network-on-chip similar to a ring-topology network to support the memory coherence protocol between CMGs so that a shared memory program can be executed using multiple CMGs.

2.3.4 Memory Technologies

As the peak floating-point performance of the CPU chip was expected to reach a few TFLOPS, the memory bandwidth of DDR4 was too low compared to the floating-point performance. Thus, high-speed memory technologies, such as HBM and Hybrid Memory Cube (HMC), were examined to balance the memory bandwidth and floating-point performance.

The HMC is a technology to connect the stacked memory chips and the CPU chip via high-speed serial links. Fujitsu already had adopted the HMC 1.0 for the FX100. The link speed by HMC 2.0 was 60 GB/s per link (per direction), and up to 4 links can be used. The power consumption to drive the serial links is large.

The HBM is a stacked memory chip connected via TSV on a silicon interposer. The HBM2 provides a bandwidth of 256 GB/s per module. The capacity of HBM2 is up to 8 GiB, but the cost is high because the silicon interposer is required.

As a memory technology available around 2019, HBM2 was chosen for its power efficiency and high memory bandwidth for both read and write. We decided not to use any additional DDR memory to reduce the cost. As described in the previous section, the number of HBM2 modules attached to CMGs is 4, that is, the main memory capacity is 32 GiB. Although it seems small for certain applications, we already have many scalable applications developed for the K computer. Such scalable applications can increase the problem size by increasing the number of used nodes.

2.3.5 Cache Structure

The key to designing a cache architecture is to provide a high hit rate for many applications and to prevent a bottleneck when data is supplied with full bandwidth from memory. We examined various parameters such as the line size, the number of ways, and the capacity in order to optimize the cache performance under the constraint of the size of the area on the die and the amount of power consumption.

When the capacity of data is increased and the line size is the same, the area for tags is increased as well as the area for data. The area size of the cache for the tag and data depends on the kind of available RAM macro. When the number of ways is increased, the complexity of the data path and the amount of control logic are increased even when the size of the data remains the same. We found that changing the number of ways and the line size of the L1 cache affects the area size. When the line size is changed from 256 bytes to 128 bytes, the area size is increased by 5%.

We examined the impact of the cache configuration on the performance by running some kernels extracted from target applications on the simulator for a single CMG. As shown in Table

Table 2.3: Kernel Performance by Changing L1 Cache Line Sizes (Normalized to the Estimated Performance of 256 Bytes)

Kernel (Application)	128 Byte	256 Byte
Kernel (Adventure)	99 %	100 %
Kernel (FFB)	78 %	100 %
Kernel A (NICAM)	86 %	100 %
Kernel B (NICAM)	56 %	100 %

Table 2.4: Kernel Performance by Changing L1/L2 Cache Sizes and Number of Ways (Normalized to the Estimated Performance of L1:64KiB4Way and L2:6MiB24Way)

L1	64K4W	64K8W	64K4W
Kernel (Application) L2	6MiB24W	6MiB24W	8MiB16W
Kernel (Adventure)	100 %	100 %	100 %
Kernel (FFB)	100 %	101 %	101 %
Kernel B (NICAM)	100 %	**94** %	**110** %
Kernel C (NICAM)	100 %	101 %	101 %
Kernel D (NICAM)	100 %	**98** %	101 %
Kernel (Seism3D)	100 %	100 %	99 %

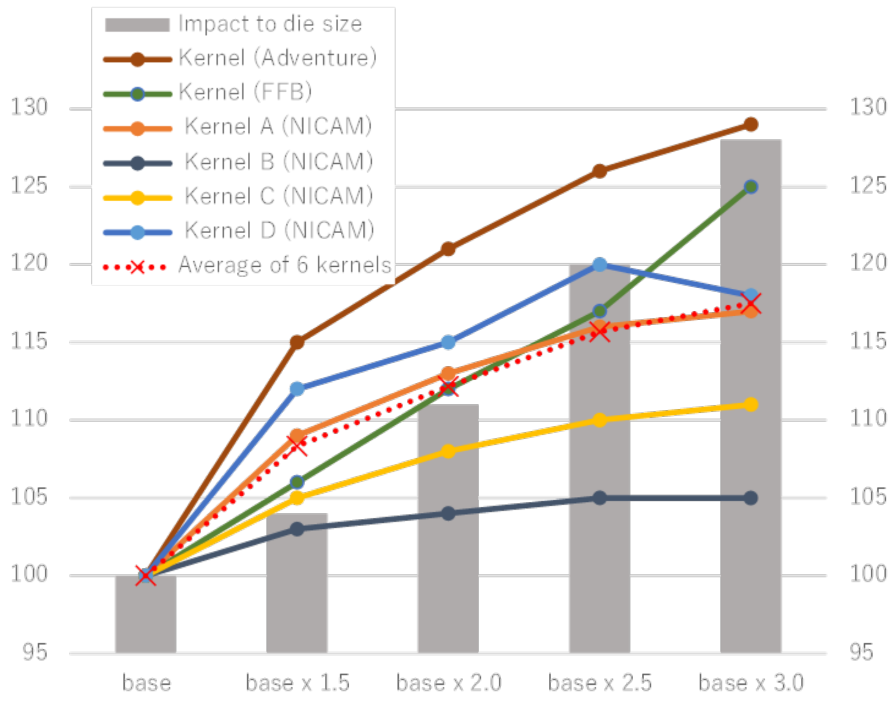
2.3, when changing the cache line size from 256 bytes to 128 bytes, a performance degradation was found in some kernels because the number of cache misses was increased. Although applications with fine-grain memory access would have the benefits of a small cache size, many HPC applications have a good performance with a 256-byte line size. No significant benefit was found by changing the cache line size to more than 256 bytes. Moreover, a wider cache line requires a wider data path, resulting in a larger area and longer latency. Table 2.4 shows the performance by changing the L1/L2 cache size and number of ways. No significant performance gain is found by changing from 4 ways to 8 ways, which has a very small impact on the power consumption. As a result, we chose a four-way L1 cache with a 256-byte line size and an 8 MiB L2 size.

We have designed the cache to save power for accessing data in a set associative cache. Data read from a way and tag search may be used in parallel to reduce the latency, but this may waste power because the data will not be used when the tag is not matched. In our design, data access is performed after a tag match. While it causes a long latency, there is less impact on the performance in the case of throughput-intensive HPC applications. This design is applied to the L1 cache for vector access and the L2 cache. We found that this design reduces the amount of power by 10% in HPL with almost no performance degradation.

2.3.6 Out-of-Order (O3) Resources

The microarchitecture is an out-of-order architecture designed by Fujitsu. The parameters for O3 resources include:

- Number of entries in the reservation station
- Number of Re-Order Buffers (ROBs)



Base Set: #RS=40, #ROB=64, renaming regs #FP=48, #GP=32, #Fetch Port=20, #Store Port=12, #write Buffer=4

Figure 2.2: Impact to area size and kernel performance by changing the amount of O3 resources (Size and performance are shown in %, relative to "Base Set")

- Number of renaming registers (general-purpose registers and floating-point/SIMD registers)
- Number of entries for the load/store queue

First, we examined several combinations of these parameters by running some kernels on the processor simulator to decide the ratio of the O3 resources. Keeping the decided ratio, the amount of the O3 resources was decided by the trade-off between the performance and the impact to the die size. Figure 2.2 shows the impact to the area size and the performance of kernels picked from a set of measured kernels and the average by changing the amount of O3 resources. We found that the impact of O3 resources to the area size is large when increasing these O3 resources, while larger resources may improve the performance.

The decisions regarding the O3 resources were some of the most difficult problems in the core design because of the following reasons:

- At the basic design phase, the compiler optimization was immature to evaluate reasonable sets of the parameters.
- Fujitsu had experience in designing O3 processors, but these instruction sets were SPARC with proprietary extensions and were completely different from Armv8.

As shown in the Figure, "base x 2.0" and "base x 2.5" can be candidates, and "base x 2.0" was chosen with respect to the impact to area size. The final decision was made as described in the microarchitecture document[2], considering the balance with the area size.

2.3.7 Enhancement for Target Applications

During the codesign process, we received feedback to enhance the architecture to improve the performance of the target applications. The chosen enhancements are as follows:

- **Combined gather operations:** For the indirect memory access in the gather instruction, when target elements are within a 128-byte aligned block for a pair of regs, requests to cache access are combined into one request, resulting in doubling the gather (indirect) load's data throughput. We found the performance improvement of the kernels containing the indirect load: 36% in "Kernel (FFB)" and 20% in "Kernel D (NICAM)".

- **Optimization for re-use of data on L2 cache:** In some applications, we found that the same data on the L2 cache is re-used frequently and increases the traffic to replace the data on the L1 cache, resulting in a performance bottleneck. We reduce the traffic for eviction by updating only the tag when it is detected that data on the L1 cache is not modified. This optimization contributes decreasing the L1 busy rate from 84% to 78% in "Kernel A (NICAM)", making room for improvement of the cache bandwidth.

2.3.8 Interconnect between Nodes

We have selected the Tofu network topology, a six-dimensional torus network, for performance compatibility with the K computer in large-scale applications.

From the viewpoint of the technology availability and maturity, we chose 28 Gbps SerDes for the link of the interconnect. While the link speed is 6.8 GB/s per link, the injection bandwidth per node is 40.8 GB/s because the number of Tofu Network Interfaces (TNIs), which are DMA engines between the network and the memory in a chip, is six.

Communication patterns were extracted from target applications, and the communication performance was estimated by the analytical model. Many target applications have neighbor communication patterns or communicate with near nodes. Therefore, the "Tofu" network with an injection bandwidth of 40.8 GB/s was concluded to be sufficient.

For all-to-all communication used in some applications, we investigated the benefits or feasibility of an additional dedicated all-to-all network, but it was not adopted due to cost. Instead, we decided to enhance the reduction operation in the interconnect to support the reduction of three double-precision floating-point numbers for QCD applications.

The new version of the interconnect is named "TofuD". The details of TofuD are described in [20].

2.3.9 Co-Design for Low Power

One of the most important challenges for an exascale system is to reduce the power consumption. We investigated several power control mechanisms for saving power. Our fundamental policy was to increase the power efficiency by reducing unnecessary power without degrading performance. To meet this goal, we prepared multiple power control mechanisms called *power knobs*, which can be turned them on and off according to the characteristics of applications.

The initial candidates for the power knobs were: the clock frequency, the SIMD width, the number of pipelines for the floating-point unit and the integer unit, the number of ways for the L2 cache, the number of instruction issuances, the O3 resources such as reservation station entries and rename registers, and memory access throttling. In order to determine which power knob should be implemented, we first estimated the power consumption by the performance estimation tool, followed by evaluation using a cycle-level simulator for more accurate analysis. We used the DGEMM kernel as a compute-intensive workload and the Stream benchmark as a

Table 2.5: Evaluation of Power Knobs for DGEMM and Stream

Power Knob	Default	Setting	DGEMM		Stream	
			Perf.	Power	Perf.	Power
Frequency	2.0GHz	1.6GHz	80%	82%	98%	89%
Inst. issuances	4 inst.	2 inst.	59%	95%	100%	98%
FPU	2 pipes	1 pipe	52%	84%	100%	100%
EXU	2 pipes	1 pipe	96%	100%	100%	100%
Memory throttling	100%	50%	100%	100%	62%	84%

Table 2.6: Power Mode for Target Applications (Performance and Power is Relative to Normal Mode (N))

Application	Power mode	B		N+E		B+E	
		perf.	pow.	perf.	pow.	perf.	pow.
GENESIS	B	1.09	1.20	1.00	0.80	1.09	0.96
Genomon	B	1.10	1.17	-	-	-	-
GAMERA	B+E	1.06	(1.14)	1.00	0.81	1.06	0.89
NICAM+LETKF	B+E	1.07	(1.18)	0.97	0.79	1.04	0.91
NTChem	B	1.08	1.21	0.57	0.69	0.62	0.83
ADVENTURE	N	1.07	(1.21)	0.90	0.85	0.98	1.00
RSDFT	B	1.06	1.20	0.71	0.80	0.77	0.90
FFB	B+E	1.10	(1.17)	1.00	0.80	1.10	0.94
LQCD	B+E	1.05	1.17	1.00	0.74	1.05	0.83

memory-intensive workload, as well as some kernels from target applications. As a result, we decided not to implement the control related to O3 resources because they were not so effective, except for the number of instruction issuances. As described in the section on the cache design, the cache is well designed for the lower power, and the control of the number of ways in the L2 cache was found to be no longer effective in reducing power consumption. Table 2.5 shows a part of the results of evaluation of power knobs using DGEMM and Stream. For example, when changing the frequency from 2.0 GHz to 1.6 GHz, the performance of DGEMM is 20% reduced and power is 18% reduced, while when limiting the FPU pipeline to one pipe, the performance of DGEMM is 48% reduced but power is only 16% reduced.

Memory access throttling was expected to be beneficial for compute-intensive applications. However, in the case of HBM, unlike HMC, the benefit by changing the frequency to access the memory is small since the power consumption to access the HBM is small enough if the memory is not accessed frequently.

For memory-intensive applications, the control of the arithmetic pipeline is expected to be beneficial for saving power because the utilization of the arithmetic unit is low. Actually, we found that there is no reduction of power by controlling the arithmetic unit for Stream as in Table 2.5. This was because the mechanism for stable operations against power fluctuation consumed a certain amount of power, even when the arithmetic unit was not working. We defined a new mode called *eco mode* in which only one arithmetic unit is active with the mechanism for stable operation for another unit inactive.

In order to improve the power efficiency, it is important to lower the power supply voltage as much as possible while increasing the frequency. Another demand is to improve the maximum performance even if the power consumption is slightly increased in order to pursue the performance. For this demand, we provide *boost mode* in which the power supply voltage is increased to increase the maximum frequency beyond that in the normal mode. As described in Chapter 2 “overview Fugaku”, the clock frequency of the boost mode is 2.2 GHz, increased from 2.0GHz

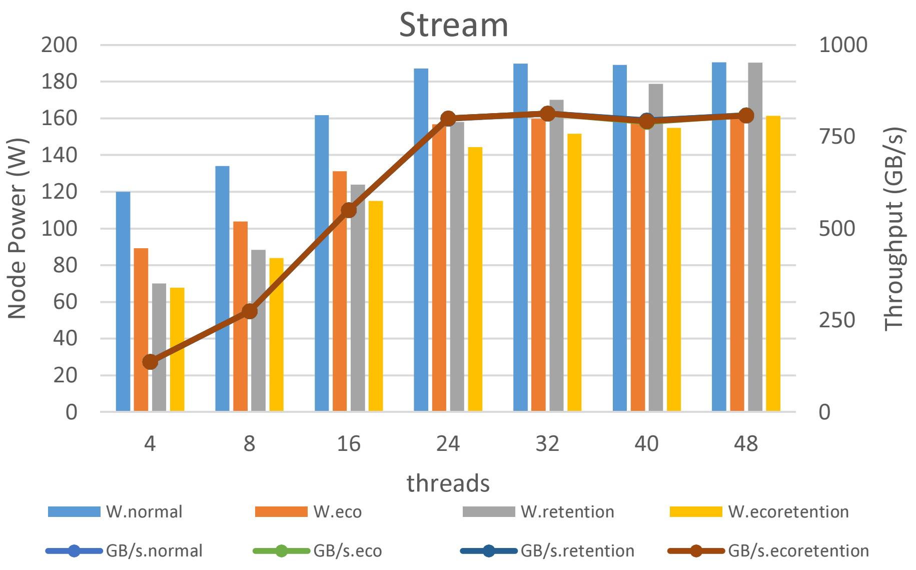


Figure 2.3: Effects of eco mode and core retention on the Stream benchmark

of the normal mode.

Since the eco mode can also be active in the boost mode, the four modes of 'boost' (B), 'boost+eco' (B+E), 'normal' (N), and 'normal+eco' (N+E) can be selected. We have evaluated several kernels taken from target applications using a performance estimation tool, and estimated the modes that could achieve the maximum performance within the capacity of the maximum power. Table 2.6 shows the results. The values are the ratio based on each normal performance and power. The value in parentheses indicates that the maximum power has been exceeded. It should be noted that four applications out of nine target applications choose the 'boost eco mode' for maximum performance, where it improves performance while reducing power in comparison with normal mode.

The power knobs can be controlled within user programs using the Sandia Power API [21], which is now maintained as the Power API Community Specification [22]. Moreover, this API allows acquiring the measured power of the node and the estimated power calculated from the performance counters.

In a large-scale system, the power consumption at the idle state is also important because the number of nodes is huge and the total power consumption of the system reaches several megawatts. To save power in the idle state, we defined a new CPU power state called *node retention*, in which the power consumption is reduced to 50% of the CPU standby state. In the node retention, all cores except one assistant core are in the *core retention* state. The transition to core retention can be controlled on a core-by-core basis. When not using all the cores in the chip, power saving can be expected by using core retention.

Figure 2.3 illustrates the result for Stream benchmark while scaling the number of threads. We compared 4 power modes; normal mode, eco mode, retention mode and eco retention (ecoret) mode. In normal mode, two FPU is active, while in eco mode, only one FPU is active. In normal mode, inactive threads is just in idle state, while in retention mode, inactive threads is in core retention state. The ecoret mode is the combination of eco mode and retention mode.

The figure shows total throughput in each power mode by line graph, but they are almost the same and they are overlaid. The throughput is increased up to 24 threads in proportion to the number of threads. The peak throughput is about 800GB/s, this is about 80% of theoretical memory throughput.

Bar graphs show the measured power consumption of node in each power mode. When the

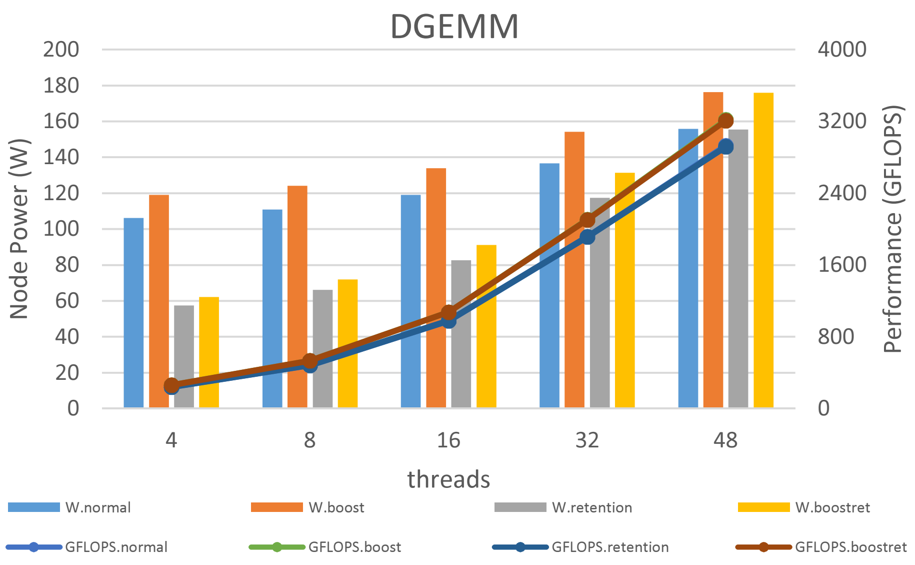


Figure 2.4: Effects of boost mode and core retention on the DGEMM benchmark

number of threads is 4, the power in normal mode is about 120W and the power in eco mode is about 90W, which is 25% lower keeping throughput. In retention mode, 44 cores are in the core retention state and only 4 cores are in the active state. Therefore, the power consumption is about 70W, which is 40% lower. As the number of threads increases, the number of retained cores decreases, so the effect of core retention decreases. The throughput is saturated with 24 threads, and at this point the power consumption of normal mode is about 190W. In eco retention mode, the power is about 145W, which is 23% lower.

Figure 2.4 shows the evaluation results of DGEMM execution. We compared 4 power modes; normal mode, boost mode, retention mode and boost retention (boostret) mode. In normal mode, program runs at 2.0GHz, while in boost mode, it runs on 2.2 GHz. The boostret mode is the combination of boost mode and retention mode.

The Line graphs show the performance by GFLOPS in each mode, but they are almost the same whether or not core retention is set, and we can only see two lines. One is the performance of normal mode, and the other is the performance of boost mode, which is 10% higher than normal mode as same as the frequency increase. Both performance increase in proportion to the number of threads. The maximum performance in boost mode is about 3200 GFLOPS, which is 95% of theoretical peak performance.

Bar graphs show the measured power consumption of node in each power mode. When the number of threads is 4, the power of normal mode is 106W while the the power of boost mode is 119W, which is 12% larger than normal mode. On the other hand, the power in the retention mode is 58W, which is 46% lower. This is because that 44 cores are in the core retention state, and only 4 cores are in the active state. When the number of threads increase, the effects of retention becomes small. When the number of threads is 32, the power of normal mode is 137W while the power of boostret is 132W, which is 4% smaller than normal mode but the performance is 10% higher. In this way, if inactive cores exist, by using active cores in boost mode and keeping the rest in the core retention state, it is possible to get a 10% performance improvement by boost mode while the power consumption remains the same or less by core retention.

More detailed analysis of the processor power consumption is described in [23].

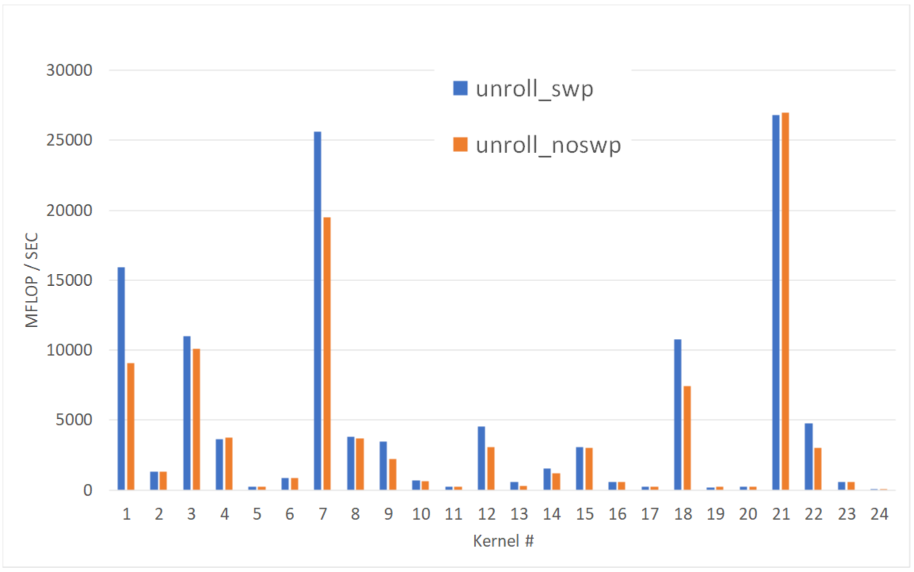


Figure 2.5: Performance improvement of Livermore Loops by software pipelining optimization

2.4 Codesign of Compiler

Since October 2019, the test system of Fugaku has been available for users. As well as the performance evaluation of several programs, including the target applications and benchmark programs, the codesign effort must move in the direction from architecture to software, including compiler optimization and performance turning of application programs.

2.4.1 Compiler Optimization for A64FX Processor

Through the codesign process for the A64FX processor, the architecture has been designed as an HPC-oriented processor. This means that the latencies to execute floating-point instructions are relatively long because many kinds of floating-point arithmetic operations are executed in nine cycles by using a Fused Multiply Add (FMA) arithmetic unit. This unit contributes to reducing the die size, but may cause a long latency. The cache structure having only an L1 cache within each core may increase the latency to access the data, while other server-class processors have L1 and L2 caches inside the core. Many HPC applications can be executed efficiently with throughput-oriented operations, such as loops for vector operations, but this may cause a problem if there are a lot of operations affected by long latency. This latency problem may cause another problem whereby the O3 resources tend to be depleted, resulting in a performance degradation.

To mitigate the problem, we found that the software pipelining technique is effective to improve the performance of some types of loops. This is because the software pipelining optimization may help to make the live period of values in O3 execution short so that it enables O3 resources to be used efficiently. Figure 2.5 shows the performance improvement of each loop in the Livermore benchmark[24]. Note that each loop is executed by a single core.

Another optimization is loop fission. As the amount of O3 resources is small compared to the latency of floating-point instructions and load/store instructions, a loop with a long body may cause a shortage of O3 resources. In addition, such a loop causes a lot of register spills, resulting in low performance. The Fujitsu compiler supports the function of automatic loop fission. Moreover, this compiler facilitates software pipelining for split loops and expects overlap in operations.

References

[2] *A64FX Microarchtecture Manual.* `https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.0.pdf`.

[12] Richard F. Barrett, Shekhar Borkar, Sudip S. Dosanjh, Simon D. Hammond, Michael A. Heroux, X. Sharon Hu, Justin Luitjens, Steven G. Parker, John Shalf, Li Tang. *On the Role of Co-design in High Performance Computing.* Volume 24: Transition of HPC Towards Exascale Computing, Advances in Parallel Computing, IOS Press, DOI: 10.3233/978-1-61499-324-7-141, pp. 141 - 155, 2013.

[13] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill and D.A. Wood. *The gem5 Simulator.* ACM SIGARCH Computer Architecture News, Vol.29, Issue 2, May 2011.

[14] Y. Kodama, T. Odajima, A. Asato and M. Sato. *Accuracy improvement of memory system simulation for modern shared memory processor.* Proc. of the HPCAsia2020, pp.142-149, Jan. 2020.

[15] *SPARC64 VIIIfx Extensions.* `http://www.fujitsu.com/downloads/JP/archive/imgjp/jhpc/sparc64viiifx-extensions.pdf`.

[16] Nigel Stephens et al. *The ARM scalable vector extension.* IEEE Micro 37.2 (2017), pp. 26–39. doi: 10.1109/MM.2017.35.

[17] *VanGuard Astra - Sandia National Laboratories.* `https://vanguard.sandia.gov/astra/`.

[18] Simon McIntosh-Smith, James Price, Tom Deakin and Andrei Poenaru. *A performance analysis of the First generation of HPC-optimized Arm processors.* Concurrency and Computation: Practice and Experience 31.16 (2019), e5110. doi: 10.1002/cpe.5110.

[19] Tetsuya Odajima, Yuetsu Kodama, Mitsuhisa Sato. *Power performance analysis of ARM scalable vector extension.* COOL CHIPS 2018: pp. 1-3, 2018.

[20] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida and T. Inoue. *The Tofu Interconnect D.* Proc. of the IEEE Cluster 2018, pp.646-654, Sep. 2018.

[21] R. E. Grant, M. Levenhagen, S. L. Olivier, D. DeBonis, K. T. Pedretti and J. H. Laros III. *Standardizing Power Monitoring and Control at Exascale.* Computer, vol. 49, no. 10, pp. 38-46, Oct. 2016.

[22] *Power API — A reference implementation for the Power API specification.* `http://github.com/pwrapi/`.

[23] Yuetsu Kodama, Tetsuya Odajima, Eishi Arima and Mitsuhisa Sato. *Evaluation of Power Controls on Supercomputer Fugaku.* Proceeding of Energy Efficient HPC State of the Practice Workshop (EE HPC SOP 2020) in conjunction with IEEE Cluster 2020, Sep. 2020.

[24] F. H. McMahon. *Livermore fortran kernels: A computer test of numerical performance range.* Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986. `https://www.netlib.org/benchmark/livermore,LivermoreFortranKernels`.

Chapter 3

Codesign of System Software

3.1 Co-Design of Operating System Kernel

3.1.1 Approach

3.1.1.1 Issues on K Computer

1. Linux distribution and continuity of providing latest Linux Kernel version
   Although the K Computer, whose CPU architecture is SPARC, adopted the Linux kernel, no major Linux distributions, such as Red Hat and SUSE, was supported at the time unlike Intel CPU architectures. Thus, users had to build programming tools, not provided by Fujitsu or RIKEN, from source code. This was sometimes difficult because such tools were not implemented for the SPARC architecture.

   The Linux kernel has meanwhile evolved, but it was also difficult to upgrade to the latest Linux kernel because K's Linux kernel was modified specifically for the K Computer.

2. Improvment of easy-of-use
   A Linux distribution mainly provides a programming environment and does not specifically support applications for supercomputers. Even if an application is available as open source, it may not be easy to install it on a supercomputer due to differences in supercomputer-specific settings. Also, the Linux kernel does not always provide system functions that are suitable to running HPC applications.

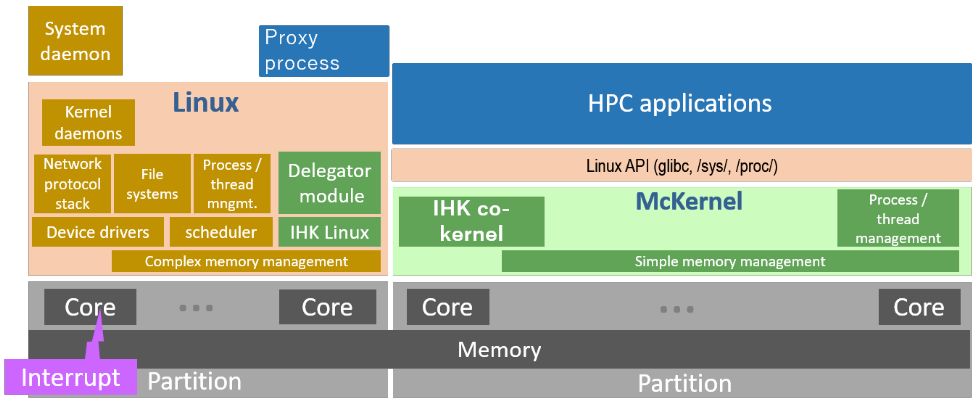


Figure 3.1: Architectural overview of IHK/McKernel.

3. Runtime environment for computer scientists
   Most of the K Computer users were computional scientists, and the number of computer scientists were limited. This is due not only to the lack of programming tools mentioned above and the lack of support for the latest Linux kernel, but also due to the lack of an experimental environment to evaluate new Linux features.

3.1.1.2 New Hardware Support

1. Reducing OS noise
   In Fugaku, which has twice the number of computing nodes and 12 times the number of CPU cores in the K Computer, OS noise reduction is required more than in the K computer.

2. Large page support
   The page table size of the Arm architecture can be flexibly configured to 4KiB, 16KiB, 2MiB, 32MiB, 64MiB, 512MiB, and 1GiB (with some restrictions on combinations). It is necessary to determine the normal page size and large page size in consideration of the balance between the TLB cache miss rate and memory utilization (using large pages could increase sub-page-sized memory regions which are hold but not used) of the target applications.

3. Many-core support
   The Fugaku CPU is a many-core architecture, equipped with 48 cores. In order to increase the efficiency of parallel processing in a node, it is desired to realize an execution environment that does not share variables like a process and operates in the same address space like a thread.

Table 3.1: Large Page Size Avaialbility

メモリ領域				McKernel拡張 THP	hugeTLBfs	
					Linux	McKernel
.text				-		
.data				2MiB – 16GiB	2MiB	
.bss				2MiB – 16GiB	2MiB	
Stack				2MiB – 16GiB	2MiB (Fujitsu Compilers only)	
Heap	malloc			2MiB – 16GiB	2MiB	
	sbrk			2MiB – 16GiB	-	
mmap	maloc			2MiB – 16GiB	2MiB	
	hugeTLBfs			2MiB		
	Anonymous	Auto		2MiB – 16GiB	-	
		Manual		2MiB – 16GiB	2MiB – 16GiB	2MiB – 16GiB
	Thread stack			2MiB – 16GiB	2MiB	
	Dynamic link library			-		
	Shared Memory	System V IPC	Auto	2MiB – 16GiB	-	
			Manual	2MiB – 16GiB	2MiB – 16GiB	2MiB – 16GiB
		POSIX		-		
		XPMEM		2MiB – 16GiB	-	

3.1.2 codesign Results

Results of codesign mentioned in the previous section are summarized.

3.1.2.1 Issues on K Computer

1. Linux distribution and contiguous support for the latest Linux Kernel version
   Fujitsu has implemented various Fugaku specific functions by adding kernel modules without modifying Linux kernel itself. Problems found during the porting of Linux to Fugaku and their fixes were reported to the Linux community and reflected in the mainstream.

   We decided to use Red Hat Enterprise Linux 8 (RHEL8) as the Linux Distribution. With RHEL's support for Arm, it will be possible in the future to update the distribution to the latest version in a timely manner because the Linux kernel itself is not modified.

2. Operating system scalability and Linux compatibility

   While the K Computer succeded in providing a scalable execution environment it suffered from the inability to follow Linux changes and thus it offered limited features for emergin applications. We have developed IHK/McKernel, that is a light-weight multi-kernel operating system designed for high-end supercomputing for addressing this issue. IHK/McKernel runs a lightweight kernel side-by-side with Linux on difference CPU cores of compute nodes with primary motivation of providing a scalable execution environment for HPC applications but to retain Linux compatibility at the same time. IHK/McKernel is open source under the GPL license and was originally developed at the University of Tokyo. Unlike K computer, Fugaku has extra CPU cores for OS activities on each node. That is, in addition of 48 CPU cores, four or two CPU cores are available on a compute & IO node or a compute node, resplectively. As shown in Figure 3.1, Linux runs on some CPU cores, called assistant cores in the case of Fugaku, and McKernel runs on the rest of CPU cores, 48 cores in the case of Fugaku. IHK/Mckerel provides an efficient memory and device management so that resource contention and data movement are minimized at the system level. It eliminates OS noise by isolating OS services in Linux and provides jitter free execution on the application cores. IHK/McKernel supports the full POSIX/Linux APIs by selectively offloading (slow-path) system calls to Linux.

   A low-level software infrastructure, called Interface for Heterogeneous Kernels (IHK), is a general framework that provides capabilities for partitioning resources in a many-core environment (e.g.,CPU cores and physical memory) and it enables management of lightweight kernels. IHK can allocate and release host resources dynamically and no reboot of the host machine is required when altering configuration. IHK also provides a low-level inter-kernel messaging infrastructure, called the Inter-Kernel Communication (IKC) layer.

   There are two OS modes on the Fugaku, the Linux-only and IHK/McKernel modes. The default is Linux-only mode in which Linux runs on all CPU cores. Because IHK/McKernel only implements basic OS functions, such as memory and process/thread management and signal handling, and the other Linux functions, such as file I/O, are offloaded to Linux, applications, whose file I/O time is dominant in the total execucion time, should run on Linux-only mode. The users may select the Linux-only or the IHK/McKernel mode.

   While the Fugaku hardware was unavailable, we develped IHK/McKernel on the Intel many-core architecture Xeon Phi. A comparative evaluation between Linux and IHK/McKernel both on the Xeon Phi based Oakforest-PACS supercomputer and on Fugaku has been published in [25].

3. Improvment of easy-of-use
   RHEL and CentOS, a free Linux distribution compatible with RHEL, are widely used in academia and industry. The introduction of RHEL gives people, who have not used supercomputers, easier access to Fugaku, contributing to improvement of usability.

   We invegstiged introducing EasyBuild or Spack as a tool to easily install open source software to Fugaku. EasyBuild is a tool developed at CERN in Europe and is actively used at CEA in France. Spack is a tool developed at Lawrence Livermore National Laboratory in the US ECP (Exascale Computing Project) and is widely used in the US. As a result of the examination, Spack was adopted to Fugaku because it can describe tool dependencies more flexibly and many open sources software already registered. However, since supporting the Arm architecture was delayed, RIKEN and Fujitsu cooperated to promote the support for the Arm architecture. As a result of the cooperation users can use as much open source software on x86 and aarch64. Table 3.2 shows a comparison of build success rate between the begging and the latest final result.

Table 3.2: Comparison of build success rate of June 2019 and Feb. 2021

Architecture	Compiler	June 2019	Feb. 2021 (Final Result)
x86_64	GCC	72.70%(2357/3242)	84.19%(4499/5344)
aarch64	GCC	67.82%(2199/3242)	81.08%(4333/5344)
aarch64	Fujitsu Compiler	34.20%(1109/3242)	74.59%(3986/5344)

4. Runtime environment for computer scientists
   KVM (Kernel-based Virtual Machine) is provided. It is possible to run a different version of Linux from the Linux kernel provided by Fugaku and experiment with new kernel functions.

3.1.2.2 New Hardware Support

1. Reducing OS noise
   Introducing assistant cores makes OS noises reduce. The number of assistant cores were determined by investigating how much CPU resource is needed to run OS daemons. Fujitsu published a paper of OS noise evaluation in Fujitsu Technical Review[26].

2. Large page support
   Large page size availability on Linux and McKernel is shown in Table 3.1. By setting the following shell environment variable, the users can use Transparent Huge Page (THP) function provided by McKernel.

```
export XOS_MMM_L_HPAGE_TYPE=thp
```

3. Many-core support

   We have researched and developed Process-in-Process (PiP) to efficiently execute parallel processing in a node[27, 28, 29]. PiP offers a new in-node parallel execution model alongside conventional multi-process (e.g. MPI) and multi-thread (e.g. OpenMP) execution model. PiP performs multiple PiP tasks (corresponding to previous processes) in the same address space. Here, the PiP task is different from the multi-thread in such a way that the static variables of each PiP task are privatized for each PiP task. Therefore, exclusive control is not required when updating the values of these variables. Also, in the case of multi-process, each process has an independent address space, so there is a drawback that the communication overhead between processes becomes large, but since PiP shares the same address space, it is simple and has little overhead. Memory access also has the advantage of enabling data exchange among PiP tasks.

PiP can be implemented if Position Independent Executable (PIE), the Linux `clone()` system call or POSIX `pthread_create()`, and the `dlmopen()` function are supported. For this reason, entire PiP implementation is user-level library and does not require a special OS kernel or dedicated compiler. Here, the `dlmopen()` function is different from `dlopen()` in that it can load libraries and programs in the new namespace. Unfortunately, in normal GLIBC, `dlmopen()` specifies that the maximum number of namespaces is 16, which is the maximum number of PiP tasks. This is completely insufficient for current manycore processors. For this reason, the PiP software package also provides a separately modified GLIBC to ease this limitation, which can be used to generate up to 300 PiP tasks.

Since the PiP tasks shares the page table, the context switching time is as fast as the thread. In the multi-process parallelism, a method of providing a shared memory area to reduce communication overhead is widely used, but in this case, a problem may occur in the memory consumption of the page table in the OS kernel. A method of mapping the memory area of another process to its own address space using XPMEM has also been proposed, but the overhead of map generation and deletion is very large. In order to reduce the overhead, it is necessary to devise an implementation such as introducing a cache-like mechanism. Also, it is difficult to map the area containing the pointers and refer to the pointers as they are, because mapping memory pages of other processes does not guarantee to be mapped to the same addresses. However, since it is not necessary to newly map the memory by using PiP, there is no overhead of area generation and deletion. Moreover, since it can be accessed with the same address, no special consideration is required for accessing the area including pointers. Therefore, it is possible to refer to the data of different PiP tasks at high speed and flexibly as needed.

The publicly available PiP software package also includes pip-gdb, which allows users to debug PiP tasks. In Pip-gdb, each PiP task is associated with the gdb's `inferior`. There is also an installer called PiP-pip to install PiP and patched GLIBC and GDB. The spack recipe for the PiP package is also registered in the spack main repository.

3.2 Co-Design of Communication Library

3.2.1 Approach

3.2.1.1 Issues on K Computer

The specifications of the MPI communication library, which is a de facto standard, are discussed at the MPI Forum and are often implemented by the open source communities, such as Open MPI and MPICH, even before the specifications are finalized. Once the specification is standardized, open source conforming to the new specification will be distributed in a short period of time. The K computer's MPI communication library is based on the Open MPI, which is an open source code, but there are many modifications to it, and Fujitsu's library could not catch up with new versions that comply with the specifications added later.

In the Fugaku development, we have cooperated with the Open MPI and MPICH communities, and did not extend them without upstreaming to or getting accepted by the communities so that the cost of catching up with a new standard can be reduced. Fujitsu continued to use Open MPI for Fujitsu MPI and RIKEN developed an MPI library based on MPICH, called MPICH-Tofu. By providing two MPI implementations, a wider range of MPI functions can be provided.

Reducing the dynamic process creation time (MPI_Comm_spawn) and the following restrictions on K computer were taken care of by Fujitsu MPI.

- Topology-aware node allocation is not possible.
- The number of processes in a node cannot be specified (inherited from the original job).
- Shell environment variables (especially related to Open MP) cannot be specified (inherited from the original job).
- More than one job cannot run on a node because only one `MPI_COMM_WORLD` is managed on each node in K computer.

The pack / unpack processing of derived data types is parallelized using Open MP to speed up such processing. MPI runtime can tell Open MP threads are available to parallelize the processing by checking if they are used for other purposes by calling the Open MPI library function `omp_in_parallel()`. The number of available Open MP threads can be obtained by using `omp_get_num_threads()` and `omp_get_max_threads()`.

Since the 1 processes-per-node (PPN) execution mode was recommended for the K computer, the inter-process communication performance within the compute node was not optimized. On the other hand, in Fugaku, the 4 PPN execution mode is recommended, so it is necessary to optimize the communication performance between the processes in the compute node. To do that, a 1 COPY implementation method is adopted in which the sender's data is directly copied to the receiving area of the receiving process using shared memory. There are several shared memory implementations in Linux kernel, e.g., KMEM, CMA, and XPMEM. Which shared memory implementation is used was determined by comparing the performance figures of those implementations.

3.2.1.2 Needs of Target Applications

It was pointed out that reducing the `Alltoall` communication latency in the 4 PPN execution mode greatly contributes to the improvement of the execution time of a target application.

At the time of K computer's development, the MPI communication library was assumed to be used by computational scientists. Most users satisfied with Fujitsu MPI performance. However, there was a need to use a low-level communication library to reduce communication latency as much as possible for those applications in which the communication latency occupy a large portion of the execution time. Computer scientists, who develop middleware and programming environments, also need to use a communication library located one layer down MPI. Thus, it was decided that Fugaku provides a low-level communication library.

3.2.1.3 New Hardware Support

Fugaku has more functionalities or a larger amount of hardware resources when compared to K computer, as shown in the followings. Using these, point-to-point communication function and collective communication algorithms were improved and the amount of memory used was reduced.

1. Assistant cores
   Fugaku is equipped with cores, called assistant cores, for executing the OS daemons and processing interrupts. Such OS functions do not run on CPU cores executing application codes.

2. Tofu ARMW (Atomic Read Modify Write) functions
   TofuD interconnect has ARMW (Atomic Read Modify Write) functions which perform swapping, addition, logical exclusive OR, logical AND and logical OR operation on 4 or 8 byte integer values.

3. Resource of network interface
   TofuD has 6 network interfaces and 32 x 6 barrier gates.

4. Increased number of processes
   We assumed that Fugaku would have about 636 K processes (4 PPN) while K computer has about 80 K processes (1 PPN). Memory usage increases in proportion to the number of processes. The amount of memory used by MPI has been reduced in the K computer as well, but it is necessary to further reduce the amount of memory used in Fugaku. The memory used by MPI consists of process information, such as the network addresses required for communication, and communication buffers. Those memory usages were reviewed and redesigned in order to reduce memory size.

3.2.2 Codesign Results

3.2.2.1 Issues on K Computer

1. Catching up with latest MPI standards
   Fujitsu MPI has adopted OpenMPI-4.0.1, which is close to the latest version of Open MPI, by not allowing to add modifications to it which are not upstreamed or not accepted by the community. This version is compatible with the latest MPI standard version 3. In addition, the Persistent Collective function, discussed in the MPI Forum as the next version feature, was implemented. Fujitsu implemented the module of the persistent collective communication scheduler by extenting the `libnbc` (non-blocking collective) library included in Open MPI, and made it available to the Open MPI development community.

2. Flexible dynamic process creation
   Open MPI uses PMIx as the process management interface. PMIx defines the interface between the execution environment daemon and the MPI process. The functions provided include exchanging information, such as the network address of the remote process required before an MPI process establishes a communication channel connected to the remote process, key-value communication between processes, synchronization, and dynamic process creation. PMIx interfaces `PMIx_Publish()`, `PMIx_Unpublish()`, `PMIx_Lookup()`, `PMIx_Connect()`, `PMIx_Disconnect()`, and `PMIx_Spawn()` are provided for dynamic process creation.

   Fujitsu designed and implemented an interface, called jTofu, for obtaining topology information that is a hint for optimizing communication and process placement. jTofu provides the following functions:

   - Inquiry about physical or logical shape of job
   - Conversion between rank and physical or logical coordinates
   - Inquiries about commuication path for communication

3. Improving Pack / Unpack Processing
   In MPI + OpenMP parallel model, when a user program calls an MPI function outside of Open MP's parallel region, the MPI runtime can use the waiting thread to parallelize the Pack / Unpack processing that occurs in the derived data type. In the Fujitsu MPI, the runtime option `opal_mt_memcpy` enables this feature.

4. Improving intra-node inter-process communication
   As a result of evaluating shared memory implementations, KMEM, CME, and XPMEM in Linux, XPMEM was chosen for implementig 1 COPY method for intra-node inter-process communication. This is because XPMEM is suited to parallelization of the communication. That is, in KMEM and CMA, both mapping pages of remote process and copying the contents are done in kernel mode. On the other hand in XPMEM, only the map part is done in kernel mode and the copy part is done in user mode, making it possible to parallelize the copy part with OpenMP without the overhead of kernel mode switching.

3.2.2.2 Target Application Needs

1. Collective communication
   As an intra-node communication method for the `Alltoall` communication, Fujitsu designed and implemented a 3-step CMG-to-CMG communication method.

2. Low-latency communication
   In addition to providing an MPI communication library, Fujitsu developed uTofu as an API to directly access the TofuD hardware to provide low-latency communication functions, and made its interface open to the public. uTofu is implemented at the user level except for privileged accesses, such as memory registration.

   RIKEN has developed another user-level communication library, called utf, using uTofu and released it to the public. The utf has a similar interface to the MPI point-point communication function, i.e., tagged message. The utf was developed in order to port the MPICH communication library, developed mainly by the Argonne National Laboratory in the United States, to Tofu, but the utf communication library can also be used in MPI application programs.

3.2.2.3 New Hardware Support

1. Improving point-to-point communication function
   Fujitsu implemented a progress engine running on assistant cores which proceeds non-blocking communication in background. In addition, the ARMW (Atomic Read Modify Write) function of TofuD was used to implement the MPI point-to-point communication function with less memory footprint.

2. Collective communication algorithms
   Fujitsu redesigned the `Allreduce` and `Bcast` collective communication algorithms to make use of the network interface resources whose number is increased from the K computer.

3. Reducing memory consumption
   Memory consumed by the process information management, such as the network address, is reduced by changing information gathering timing in OpenMPI and making use of shared memory in PMIx interface used by Open MPI. That is, communication between the execution environment daemon and MPI processes in the same node uses shared memory (since PMIx 1.2). OpenMPI added an execution mode in which the information of the

receiver's process is obtained on demand or at the first communication, instead of obtaining the information of all remote processes at the MPI initialization time.

The memory reduction of the communication buffers used by the Fujitsu MPI is described here. In the point-to-point communication by Tofu interconnect of K computer, a receive buffer is allocated for each sender process at the first communication. There are two modes, memory-saving and high-speed, depending on the size of the receive buffer, which can be selected by a run-time option. By default, the memory-saving communication mode is employed at the first communication of peers, and then the mode is changed to the high-speed mode after 16 communications. The number of receive buffers increases in proportion to the number of communication peer processes.

In Fugaku, a mechanism for sharing the receive buffer was introduced using TofuD's ARMW functions. The receive buffer is managed by 64 bit memory value in which the producer counter and the consumer counter are packed. This area can be manipulated remotely by all processes. The receive buffer area is allocated by increasing the producer counter using TofuD's remote atomic add operation at the sender side.

3.3 Codesign of File System and File I/O

3.3.1 Approach

At the basic design phase, we investigated file IO patterns of target applications. Table 3.3 summarizes file sharing relationship, lifetime, and applied file systems. In this table, the "Name Space" specifies visibility of a file, e.g., process location, in other words, the MD (meta data) server's accessibility. Temp. Local FS, Temp. Shared FS, and Global FS, represents temporary file system in a node, temporary shared file system in a job, and global file system, respectively.

As a result of this investigation, the following basic design policy was decided.

1. Three-layered storages are introduced. Solid State Disk (SSD), a parallel file system using hard disk, and an archive system using such like a tape form each layer. The first layer storage is used for a local storage and cache system for the second layer storage.

2. A file staging system operated in K is not supported in Fugaku. Instead, asynchronous file I/O processing and cache of the second layer storage on the first layer storage are introduced.

3. LLIO, a file I/O middle ware, is developed. LLIO utilizes main memory and the first layer storage for cache of the second layer storage. It realizes fast data transfer using the RDMA capability of the Tofu-D interface.

   RIKEN developed an application-level file I/O transfer framework, called DTF (Data Transfer Framework). DTF offers applications API of pNetCDF file I/O library. DTF implements data exchange between applications using the MPI API instead of file I/O API.

4. A parallel file system, based on Lustre file system, is equipped in the second storage.

5. The hardware and software configurations from the perspective of fault tolerance is reviewed.

Table 3.3: File Sharing Relationship, Lifetime, and File Systems

	Created File	Name Space (MD server)	Placement	File System
Local in Process	Discarding after job finished	Local in compute node	In file: 1st layer cache Out file: 2nd layer if exceeds 1st layer cache size	Temp. Local FS. Or Global FS
	Saving after job finished	Local in compute node or 2nd layer	In file: 1st layer cache Out file: 1st layer cache	Temp. local FS → Global FS. Or Global FS
Shared by MPI processes	Removing after job finished	Local in Job	In file: 1st layer cache Out file: 2nd layer if exceeds 1st layer cache size	Temp. Shared FS
	Saving after job finished	Local in Job or 2nd layer	In file: 1st layer cache Out file: 2nd layer if exceeds 1st layer cache size	Temp. Shared FS → Global FS. Or Global FS
Shared among MPI app. (Coupler)	Removing after job finished	—	—	DTF
	Saving after job finished	Global on 2nd layer	In file: 1st layer cache Out file: 2nd layer if exceeds 1st layer cache size	Global FS
Shared among jobs (workflow)	Removing after job finished	Global on 2nd layer	In file: 1st layer cache Out file: 1st layer cache	Global FS
	Saving after job finished	Global on 2nd layer	In file: 1st layer cache Out file: 1st layer cache	Global FS

3.3.2 Codesign Results

Table 3.4 summarizes the specification of 1^{st} and 2^{nd} layer storages. The designed file I/O performance was confirmed by investigating target applications. For detailed results, please refer to the paper[30].

Table 3.4: The Specification of 1^{st} and 2^{nd} Layer Storages

		Minimum Throughput	Measured Throughput
1st Storage	write	49 MB/s /node	125 MB/s /node
	read	113 MB/s /node	293 MB/s /node
2nd Storage	write	200 GB/s /volume	211 GB/s /volume
	read		220 GB/s /volume

Note: The 2^{nd} storage is formed from 6 volumes.

References

[25] Balazs Gerofi et al. “Linux vs. Lightweight Multi-Kernels for High Performance Computing: Experiences at Pre-Exascale”. In: *Proceedings of the International Conference*

for High Performance Computing, Networking, Storage and Analysis. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476162.

[26] Lei Zhang and Takayuki Okamoto and Shuuichirou Ishii and Kouichi Hirai and Shinji Sumimoto and Balazs Gerofi and Masamichi Takagi and Yutaka Ishikawa. *OS Enhancement in Supercomputer Fugaku*. https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article06.html. Nov. 2020.

[27] Atsushi Hori et al. "Process-in-Process: Techniques for Practical Address-Space Sharing". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '18. Tempe, Arizona: Association for Computing Machinery, 2018, pp. 131–143. ISBN: 9781450357852. DOI: 10.1145/3208040.3208045.

[28] A. Hori, B. Gerofi, and Y. Ishikawa. "An Implementation of User-Level Processes using Address Space Sharing". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 976–984.

[29] Kaiming Ouyang et al. "CAB-MPI: Exploring Interprocess Work-Stealing towards Balanced MPI Communication". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[30] Mitsuhisa Sato and Yutaka Ishikawa and Hirofumi Tomita and Yuetsu Kodama and Tetsuya Odajima and Miwako Tsuji and Hisashi Yashiro and Masaki Aoki and Naoyuki Shida and Ikuo Miyoshi and Kouichi Hirai and Atsushi Furuya and Akira Asato and Kuniki Morita and Toshiyuki Shimizu. "Co-Design for A64FX Manycore Processor and "Fugaku"". In: *SC'20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020.

Chapter 4

Codesign of Programming models and tools

This chapter describes the programming models and tools available on Fugaku: 1) a Partitioned Global Address Space Language, 2) compilers for standard programming languages, 3) a framework for developing particle simulator, and 4) programming tools. The users can efficiently develop and execute their programs on Fugaku by using them.

4.1 Introduction

In order to make the most effective use of Fugaku, the programming environment including compilers, languages, and tools also plays an important role. We have therefore designed and developed it on the basis of lessons learned from our experiences on the K computer and requests from pilot users of Fugaku. In this chapter, we describe the programming models and tools available on Fugaku.

The remainder of this chapter is organized as follows. First, Section 4.2 describes a Partitioned Global Address Space (PGAS) Language XcalableMP and its preliminary performance on Fugaku. In Section 4.3, the support of compilers for standard programming languages such as Fortran, C, and C++, are briefly explained. In Section 4.4, FDPS (Framework for Developing Particle Simulator), a universal software application for particle-based simulations, are outlined. Section 4.5 gives an overview of the programming tools such as an integrated development environment (IDE), debugger, and performance profilers.

4.2 XcalableMP Partitioned Global Address Space Language

In this section, we report our early experience and the preliminary performance of XcalableMP on Fugaku. XcalableMP is available as a parallel programming language for Fugaku, supported by R-CCS team with Fujitsu. C and Fortran are supported as base languages with XcalableMP 1.2 compliant.

We report the preliminary performance of XcalableMP program running on Fugaku [1].

We used the following versions:

- Omni XcalableMP Version: 1.3.2, Git Hash:6d23f46
- Language specification: 1.2.25

[1] The reported results were obtained on the evaluation environment in the trial phase. Note that the performance is not guaranteed at the start of its operation.

The performance of XcalableMP on Fugaku is enhanced by the manycore processor and a new Tofu-D interconnect.

4.2.1 Performance of XcalableMP Global View Programming

We executed IMPACT-3D for the evaluation of XcalableMP global view programming on Fugaku, using up to 512 nodes. The scalability on Fugaku is shown in Figure 4.1, comparing to the K computer. The program is parallelized by hybrid XMP-OpenMP parallel programming: An XMP node is assigned to a node, and 48 OpenMP threads are running within a node. The problem size is $512 \times 512 \times 512$ with 3-dimensional block distribution. The compile option is "-Kfast".

As shown in the figure, we found a good scalability on Fugaku, and the performance is better than that by MPI thanks to the optimized XMP runtime for communications in the stencil computation[31].

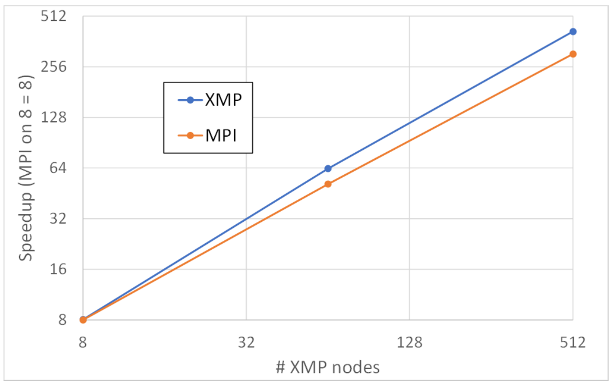


Figure 4.1: Speedup of Impact3D on Fugaku and Performance comparing to K computer

4.2.2 Performance of XcalableMP Local View Programming

Fugaku has a customized interconnection, called Tofu-D, which supports hardware-supported RDMA (Remote Direct Memory Access) operations. We implemented the XMP runtime library to make use of Tofu-D for one-sided communication for the XMP local view programming. The library is implemented by using a low-level communication layer, uTofu API [32], provided by Fujitsu.

For performance evaluation of XMP local view programming, we used CCS QCD and NTChem-MINI taken from the coarray version of Fiber Miniapp Suite [33][34].

To run CCS QCD mini-application[35], eight XMP nodes are assigned to one node, running in a flat XMP mode. The size and conditions are as follows:

- Target data: Class 2 (32 x 32 x 32 x 32) (strong scaling)
- Compiler options: -Kfast,zfill,simd=2
- Timing region: sum of "Clover + Clover_inv Performance" and "BiCGStab(CPU:double precision) Performance" of the built-in timing feature

Figure 4.2 shows the speedup on Fugaku, comparing to the performance of the K computer. The XMP version archives almost same performance of the MPI version. Note that the reason of the performance degradation of the XMP version on the K computer is the overhead of allocation for allocatable coarray used as a buffer for communication. It is improved by removing this overhead by using the uTofu communication layer.

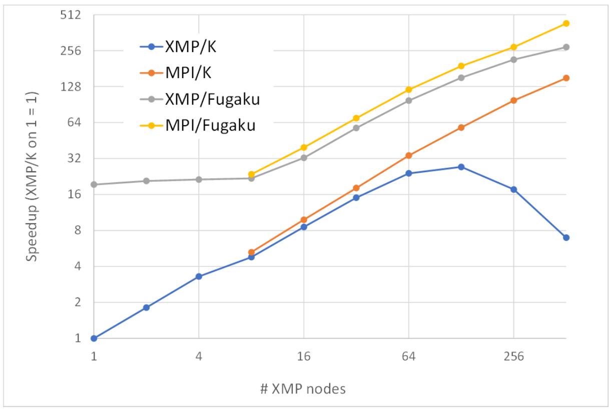


Figure 4.2: Speedup of CCS QCD on Fugaku and Performance comparing to the K computer

The NTChem-MINI is a mini-application taken from NTChem [36], a high-performance software package for molecular electronic structure calculation. One XMP node is assigned to one node, and within a node, BLAS functions are executed using 48 cores. The size and conditions are set as follows:

- Target data: taxol (strong scaling)
- Compiler options: -Kfast,simd=2
- Timing region: “ RIMP2_Driver ” of the built-in timing feature

As shown in Figure 4.3, the XMP versions archive almost the same performance of the original MPI versions.

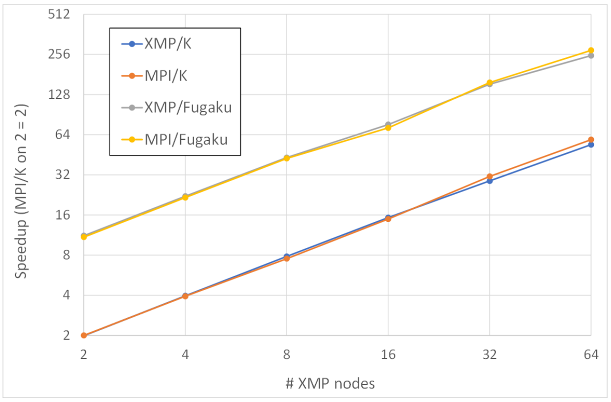


Figure 4.3: Speedup of NTChem-MINI on Fugaku and Performance comparing to the K computer

4.2.3 Global Task Parallel Programming

Recently, large-scale clusters of many-core processor such as Intel Xeon Phi have been deployed in many sites from the latest TOP500 Lists. In order to program many-core processors, OpenMP is widely used as a shared-memory programming model. Most of OpenMP programs are written using work sharing constructs for loops, which involves a global synchronization. However, especially in modern many-core processors, the global synchronization cost for work sharing becomes bigger, and the load imbalance among cores lead to the performance degradation as the number of cores on the processor increases. Task parallel programming using task dependency in OpenMP 4.0 is a promising candidate to facilitate the parallelization for such many-core processors because it enables users to avoid global synchronization by fine-grained task-to-task synchronization through user-specified data dependencies.

We are interested in extending the task parallel programming model to the PGAS model of XcalableMP for distributed memory systems. As well as removing expensive global synchronization, it is expected to enable the overlapping of communication and computation. In XMP 2.0, we propose the global task parallel programming.

In OpenMP, the task dependency in a node depends on the order of reading and writing to data based on the sequential execution. Therefore, the OpenMP multi-tasking model cannot be applied to describe the dependency between tasks running in different nodes since threads of each nodes are running in parallel. In OmpSs, interactions between nodes are described through the MPI task that is executing MPI communications. Task dependency between nodes is guaranteed by the completion of MPI point-to-point communication in tasks.

We propose new directives for communication with tasks in XMP, and they enable users to write easily the multi-tasking execution based on XMP language constructs. The tasklet directive generates a task for the associated structured block on the node specified by the on clause, and the task is scheduled and immediately executed by an arbitrary thread in the specified node if there is no task dependency. If it has any task dependencies, the task execution is postponed until all dependencies are resolved. The tasklet gmove directive copies the variable of the right-hand side (rhs) into the left-hand side (lhs) of the associated assignment statement for local or distributed data in tasks. If the variable of the rhs or the lhs is the remote data, this directive may synchronize on data dependency between nodes and execute communication. The tasklet reflect directive is a task-version of reflect operation. It updates halo regions of the array specified to array-name in tasks. In this directive, data dependency is automatically added to these tasks based on the communication data because the boundary index of the distributed data is dynamically determined by XMP runtime system.

We have designed a simple code translation algorithm from the proposed directives to XMP runtime calls with MPI and OpenMP. We have evaluated the performance using block-Cholesky Factorization Program on KNL based-system, Oakforest-PACS. Through the experiment, we confirmed the advantage of task-parallelism over the traditional loop-based data parallelism. At the same time, we found the performance problems on communication between multiple threads (MPI_THREAD_MULTIPLE). Currently, we are investigating a lower-level communication API for efficient one-sided communication of PGAS operations in multithreaded execution environment.

Details of the proposal in this chpater are decribed in [37].

4.2.3.1 OpenMP and XMP Tasklet Directive

While OpenMP originally focuses on work sharing for loops as the `parallel for` directive, OpenMP 3.0 introduces task parallelism using the `task` directive. It facilitates the parallelization where work is generated dynamically and irregularly as in recursive structures or unbounded

```
#pragma xmp tasklet [clause[, clause] ... ] [on { node-ref | template-ref } ]
  (structured-block)

#pragma xmp taskletwait [on { node-ref | template-ref } ]

#pragma xmp tasklets
  (structured-block)

  where clause is :
    {in | out | inout} (variable[, variable] ... ])
```

Figure 4.4: Syntax of the `tasklet`, `taskletwait`, and `tasklets` directives in XMP.

loops. The `depend` clause on the `task` directive is supported from OpenMP 4.0 and specifies data dependencies with dependence-type `in`, `out`, and `inout`. Task dependency can reduce the global synchronization of a thread team because it can execute fine-grained synchronization between tasks through user-specified data dependencies.

To support task parallelism in XMP as in OpenMP, the `tasklet` directive[2] is proposed in XMP 2.0. Fig. 4.4 describes the syntax of the `tasklet`, `tasklets`, and `taskletwait` directives for the multi-tasking execution in XMP. The `tasklet` directive generates a task for the associated structured block on the node specified by the `on` clause, and the task is scheduled and immediately executed by an arbitrary thread in the specified node if there is no task dependency. If it has any task dependencies, the task execution is postponed until all dependencies are resolved. These behaviors occur when these tasks are surrounded by `tasklets` directive. When these tasks are not surrounded by the `tasklets` directives, they are executed sequentially at the specified node. The `tasklet` directive supports several clauses for the description of the task dependency. The `in`, `out`, and `inout` clauses represent the task dependency in a node. When `in`, `out`, or `inout` clause presents on the `tasklet` directive, the generated task has each data dependency in a node. The behavior of these data dependencies is same as OpenMP `task depend` clause: flow, anti, and output dependencies.

The `taskletwait` directive waits on the completion of the generated tasks on each node. Since the directive does not involve the barrier synchronization, the `barrier` directive of XMP is also required in order to guarantee that all tasks of all nodes are finished at this point. There is an implicit barrier on each node at the end of the `tasklets` directive.

In OpenMP, the task dependencies are created according to the order of reading and writing to data based on the sequential execution in a node. Therefore, the OpenMP task parallel model cannot be directly applied to describe the dependency between tasks running in different nodes since threads of each nodes are running in parallel.

In OmpSs[38], interactions between nodes are described through the MPI task that is executing MPI communications. Task dependency between nodes is guaranteed by the completion of MPI point-to-point communication in tasks. While this approach can satisfy dependencies between nodes, it may cause further productivity degradation because it forces users to use a combination of two programming models that are based on different description formats. Therefore, we propose new directives for communication with tasks in XMP, and they enable users to write easily the multi-tasking execution for clusters by only using language constructs.

[2]There is the `task` directive in XMP, it is different from OpenMP 's one.

4.2.3.2 A Proposal for Global Task Parallel Programming

In order to achieve multi-tasking execution for distributed memory parallel systems, we need to perform point-to-point communication within tasks in local task dependency graphs. While XMP provides some directives for communication, many of these are performed collectively, and cause an implicit synchronization among execution nodes. This causes a performance degradation, because tasks participating in communications, such as broadcast, wait for synchronization until all tasks are completed. We propose two directives, `tasklet gmove` and `tasklet reflect`, as shown in Fig. 4.5, to describe interactions between nodes in tasks by point-to-point communication, for inter-node data dependency. These communications are only synchronized between the sender and receiver of the communication in each task.

The details of these two directives are as follows:

`tasklet gmove` directive: Although this copies the variable from the right-hand side (rhs) into the left-hand side (lhs) of the associated assignment statement for local or distributed data like the `gmove` directive, it is executed in tasks. The copy operation is basically performed on all execution nodes. However, if the distributed array is specified at the associated assignment statement, only nodes with the distributed array execute the operation in the task. The execution nodes can also be determined by the `on` clause. When the `in`, `out`, or `inout` clause is present on the `tasklet gmove` directive, the generated task has the corresponding data dependency in a node, similar to the `tasklet` directive. While the `gmove` directive also supports one-sided communication by the `in` and `out` clauses, which differ from those used for data dependencies, the `tasklet gmove` directive does not support.

`tasklet reflect` directive: Although this update halo regions of the array specified to `array-name` like the `reflect` directive, it is executed in tasks. For example, when updating one side of a halo region for a one-dimensional distributed array on two nodes, these communications are separated into four tasks: the sender of the upper element on node 1, the receiver of the upper halo region on node 1, the sender of the lower element on node 2, and the receiver of the lower halo region on node 2. In this directive, data dependency is automatically added to these generated tasks based on the communication data, because the boundary index of the distributed array is dynamically determined by the XMP runtime system. The `chunksize` clause can be added to match to the task dependency descriptions of users using the dependency generated by the `tasklet reflect` directive. When users calculate an array in block units, such as in the cache blocking technique for a node with data dependency, the user-specified task dependency and generated data dependency for halo exchange may not identically match. By specifying the `chunksize` clause, the halo region is distributed logically to equal-sized contiguous chunks, and data dependencies for the halo exchange are generated automatically by the XMP runtime system based on the specified chunk size.

Fig. 4.6 presents an example of the `tasklet gmove` directive. In this example, array *A[]* with length three is distributed onto three nodes in equal-sized contiguous blocks. This code creates three kinds of tasks. TaskA and taskC are executed on nodes specified by the `on` clause. TaskB is executed on nodes 1 and 2, because these nodes have the specified distributed array *A[0]* or *A[1]* in the associated assignment statement under the `tasklet gmove` directive. There is a flow dependency between taskA and taskB on node 1 by *A[0]*. After the execution of taskA, taskB sends *A[0]* to node 2, which is determined by the distributed array *A[1]*. In node 2, taskB receives *A[0]* from node 1 in *A[1]*. When the receive operation in taskB is finished, taskC is immediately started, because the flow dependency of *A[1]* is satisfied. TaskC sends the *A[1]* to variable B of node 3. Because the variable B is a local variable for each node, the communication destination is determined from the execution nodes specified by the `on` clause.

```
#pragma xmp tasklet gmove [clause[, clause] ... ] [on { node-ref | template-ref } ]
  (an assignment statement)

#pragma xmp tasklet reflect (array-name[, array-name] ... )
                    [blocksize (reflect-blocksize[, reflect-blocksize] ... ) ]

  where clause is :
    {in | out | inout} (variable[, variable] ... ])
```

Figure 4.5: Syntax of the `tasklet gmove` and `tasklet reflect` directives in XMP

```
int A[3], B;
#pragma xmp nodes P(3)
#pragma xmp template T(0:2)
#pragma xmp distribute T(block) onto P
#pragma xmp align A[i] with T(i)

#pragma xmp tasklets
{
#pragma xmp tasklet out(A[0]) on P(1)
   A[0] = 0;        /* taskA */
#pragma xmp tasklet gmove in(A[0]) out(A[1])
   A[1] = A[0];   /* taskB */
#pragma xmp tasklet gmove in(A[1]) out(B) on P(2:3)
   B = A[1];       /* taskC */
}
```

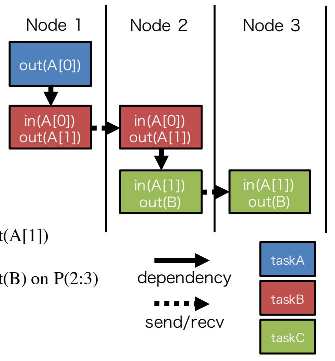


Figure 4.6: Example of the `tasklet` and `tasklet gmove` directives.

4.2.3.3 Prototype Design of Code Transformation

We have designed a simple code transformation from the code using the proposed directives to the code with XMP runtime calls using MPI and OpenMP. As for a preliminary evaluation, we have made a hand-translated MPI and OpenMP code by using the proposed transformation.

The `tasklets` directive is converted into the OpenMP `parallel` and `single` directives. The execution node is determined by the `on` clause, which is translated to an `if` statement. The `tasklet gmove` and `tasklet reflect` directives are converted into MPI_Send/Recv(), and these MPI functions are executed in OpenMP tasks with data dependency specified by users. In the case that an MPI blocking call, such as MPI_Send/Recv(), occurs in these codes, a deadlock may occur depending on the task scheduling mechanism, from the combination of MPI and OpenMP. To prevent this deadlock, in the actual implementation we used MPI asynchronous communications, such as MPI_Isend/Irecv(), MPI_Test(), and the OpenMP `taskyield` directive, which makes the current task become suspended at the time point at which it is invoked, and may result in switching to different tasks.

4.3 Compilers and Languages

4.3.1 Compilers

We developed compilers that support the languages shown in Tab. 4.1 and is capable of optimization for Fugaku.

We interviewd the developers of the target application codes to get requests about compilers and languages on Fugaku. Table 4.2 shows a part of them and the response to them.

Table 4.1: Supported Language standards/specifications

Language	Standard/Specification
Fortran	ISO/IEC 1539-1:2018 (Fortran 2018)
	ISO/IEC 1539-1:2010 (Fortran 2008)
	ISO/IEC 1539-1:2004, JIS X 3001-1:2009 (Fortran 2003)
	ISO/IEC 1539-1:1997, JIS X 3001-1:1998 (Fortrn 95)
	Fortran 90 and FORTRAN77
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)
C	ISO/IEC 9899:2011 (C11)
	ISO/IEC 9899:1999 (C99)
	ISO/IEC 9899:1990 (C89)
	GNU extensions
	Clang extensions
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)
C++	ISO/IEC 14882:2014 (C++14)
	ISO/IEC 14882:2011 (C++11)
	ISO/IEC 14882:2017 (C++17)
	GNU extensions
	Clang extensions
	OpenMP Application Program Interface Version 4.0 July 2013
	OpenMP Application Program Interface Version 4.5 November 2015 (partially)
	OpenMP Application Program Interface Version 5.0 November 2018 (partially)

Table 4.2: Excerpt of the response to requests about compilers and languages

No	Request	Response
1	inlining elemental functions	enhance the inlining feature.
2	optimization for C/C++	improve STL, more efficient compilation process, etc.
3	interprocedural optimization (IPO)	For C/C++, enhance the feature of link-time optimization. For Fortran, support inter-module optimization.
4	mandatory optimization and parallelization	support the feature for parallelization and SIMDization.
5	starategy of optimization	support the features of selecting the strategy by specifying the characteristics of the target program.
6	multi-dimensional SIMDization	supported
7	optimization of single-precision operations	enhance the optimization based on the instruction set architecture of Fugaku.
8	pragma for memory allocation	support a compiler option or pragma for memory padding.

4.3.2 Support of Half-Precision Floating-Point Type (fp16)

The FP16 feature can be used in Fortran, C, and C++. The feature supports the binary16 format of the IEEE754-2008 standard.

The supported type qualifiers are as follows:

- Fortran
  - `REAL(KIND=2)`
  - `COMPLEX(KIND=2)`
- C/C++
  - `_Float16`
  - `__fp16`

4.3.3 Other Languages

Java, Ruby, and Python are available on Fugaku. In particular, for Python, a technique for efficient execution based on mathematical libraries is provided since Python has been widely used in the field of HPC.

4.4 FDPS

FDPS (Framework for Developing Particle Simulators), [39, 40] is a framework which allows application programmers develop their own high-performance parallelized particle-based simulation code, without implementing the necessary parallelization algorithms such as domain decomposition, particle migration and some way to evaluate interactions between particles in different computational domains. We have made FDPS available on Fugaku and made a number of optimizations specific to Fugaku. Also, we have added a code generator for particle-particle interaction kernel, PIKG, which generates highly optimized particle-particle interaction kernel from high-level description of the interaction kernel. PIKG can generate optimized code for AVX2, AVX512, Cuda and A64fx SVE. Optimizations for A64fx include the use of SIMD intrinsics, loop unrolling and loop division.

4.5 Tools

4.5.1 GUI-based Integrated Development Environment

As the GUI-based Integrated Development Environment on Fugaku, we adopted Eclipse [41] with PTP, which is one of Eclipse plugins. PTP has features of editing programs, monitoring systems, and controling jobs. We enhanced or modified the second and third features to be suitable to the usage on Fugaku.

4.5.2 Features for Debugging

There are the following three features for debugging programs on Fugaku.

- Deadlock inspection
  This feature enables users to locate the point of deadlock and get the information of backtrace, memory map, and variables in the stack.

- Abnormal termination inspection
  This features enable users to get information of the backtrace, memory map, and disassembled code of the function at the abnormal termination
- Scripted debugging
  This feature enables users to execute a sequence of the debbugging operations specified in a file (i.e. script). It is possible to apply the script to all of the processes or to certain specified processes.

4.5.3 Performance Profiler

There are two performance profilers available on Fugaku: Instant Performance Profiler and Advanced Performance Profiler.

- Instant Performance Profiler

  The Instant Performance Profiler measures and outputs the statistical information of the entire program through sampling analysis. The information includes:

  - *Statistical time information*
    program elapsed time, user CPU time, and system CPU time.
  - *CPU performance characteristics*
    information related to CPU performance characteristics, such as memory throughput, the number of instructions, and the number of operations.
  - *Cost information*
    the number of sampling times during program execution as the cost for each procedure, loop, or line.
  - *Call graph information*
    procedure call traces and the cost for each procedure call trace.
  - *Source code information*
    the cost of each line of the source code.

- Advanced Peformance Profiler

  The Advanced Performance Profiler measures and outputs the execution performance information of the specified region of an application. The information includes:

  - *Statistical time information*
    the number of calls, elapsed time, user CPU time, system CPU time breakdown, etc. in the target region.
  - *MPI communication cost information*
    the number of executions of MPI functions, message length, and average, maximum, and minimum execution time, and waiting time in the target region
  - *CPU performance analysis information*
    CPU performance characteristics at the time of application execution in the target region.

References

[31] Hitoshi Murai, Mitsuhisa Sato. *An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language.* 7th International Conference on PGAS Programming Models,Edinburgh, Scotland, UK (2013, Oct.).

[32] FUJITSU Ltd. *Development Studio uTofu User's Guide.* (2020, Mar.)

[33] *Fiber Miniapp Suite.* RIKEN R-CCS, fiber-miniapp.github.io.

[34] Hitoshi Murai, Masahiro Nakao, Hidetoshi Iwashita, Mitsuhisa Sato. *Preliminary Performance Evaluation of Coarray-based Implementation of Fiber Miniapp Suite using XcalableMP PGAS Language.*

[35] *CCS QCD Solver benchmark program.* https://www.ccs.tsukuba.ac.jp/qcd/ccsqcdsolverbenchmic/.

[36] *NTChem Overview.* `https://www.r-ccs.riken.jp/software_center/software/ntchem/overview/`.

[37] Keisuke Tsugane, Jinpil Lee, Hitoshi Murai, Mitsuhisa Sato. *Multi-tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters.*

[38] D. Alejandro, A. Eduard, B. Rosa M, L. Jesus, M. Luis, M. Xavier, P. Judit. *Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures.* Parallel Processing Letters, Vol.21, pp.173–193, 2011.

[39] M. Iwasawa et al. "Implementation and performance of FDPS: a framework for developing parallel particle simulation codes". In: *Publications of the Astronomical Society of Japan* 68, 54 (Aug. 2016), p. 54. DOI: `10.1093/pasj/psw053`. arXiv: `1601.03138 [astro-ph.IM]`.

[40] M. Iwasawa et al. "Extreme-scale particle-based simulations on advanced HPC platforms". In: *CCF Transactions on High Performance Computing* 2.2 (June 2020), pp. 183–195. ISSN: 2524-4930. DOI: `10.1007/s42514-020-00020-1`.

[41] Eclipse Foundation: https://www.eclipse.org/.

Chapter 5

Codesign of Numerical library

5.1 Overview of Numerical software on Fugaku

Large-scale computing for science and technology often involves a standard problem format or uses similar computational methods, even though various physical problems can be solved. The numerical calculation functions for such general-purpose processing should be provided as a library for science and technology computing to exploit a new supercomputer's performance fully. The basic numerical functions represented by BLAS are also critical as fundamental components to be used internally from such libraries for science and technology calculations. Since there are many opportunities to be used directly by application programs, it is important to deliver high performance processing under various calling conditions through implementation that matches the underlying hardware architecture.

We have designed the numerical library stack on Fugaku for general-purpose scientific and technological computation, which is capable of fully exploiting the performance of Fugaku. The following were the three outlines of the development policy.

1. The scientific and mathematical libraries that had been provided for the K computer have been **inherited** to Fugaku to ensure satisfactory performance. In particular, the most important BLAS, LAPACK, ScaLAPACK and FFT related functions have been developed from detailed design to the implementation of the algorithms to the new architecture.

2. In the development of Fugaku, we focused on the **codesign** with application developers. We did not only enhance the functions from the viewpoint of numerical analysis, but also considered the issues involved in using Fugaku from actual application programs.

3. Based on the modernization **trends** in applied mathematics and computer science, new functionalities, algorithms of increasing importance, and implementation methodologies were considered and actively developed. We have shared the knowledge gained from tuning the libraries, and cooperated with both RIKEN and Fujitsu in porting and developing additional functions.

Following the above policies, the numerical library development working group from Fujitsu and RIKEN collaborated to establish the Fugaku numerical environment. Specifically, we pursued the software stack design with four sub-stacks; i) Fujitsu software stack, which inherits a similar numerical software environment from the K computer. ii) RIKEN software stack, which also inherits RIKEN-provided numerical software considering the conventional software's performance bottleneck. iii) codesign efforts, which are based on tight and robust interactions among application users. iv) OSS stack, which is also an inevitable piece of parallel programming.

5.1.1 Fujitsu software stack

Fujitsu's software stack provides Netlib [42]-based libraries and SSL II-based libraries[43], which have proven functions in the K computer as shown in Table 5.1.

5.1.2 RIKEN software stack

RIKEN software stack contains more advanced and highly functional numerical libraries. It has aimed to utilize and enhance the BLAS kernels tuned for A64FX by Fujitsu.Then we intended that the total numerical software stack maximizes the computational potential of Fugaku by reducing the wide variety of overheads in a highly parallel and highly threaded environment. Consequently, RIKEN has added mainly four numerical packages to provide a user with high performance numerical framework on Fugaku, as shown in Table 5.2.

5.1.3 CoDesign efforts

The CoDesign is reflectively applied to the design of the numerical libraries in response to the requirements of the users of the prioritized application codes in the basic design. In particular, we clarified the problems of the tuning target based on the results of the interviews and obtained the parameter conditions (*size*, *precision*, *number of threads*, and so on) for the performance-critical problems. In addition, during the detailed design, the functions to be tuned were determined based on the detailed evaluation results of the prioritized application codes and the information from the hot spot analysis, and the acquired information was reflected in the development.

5.1.4 OSS software stack

When we shift the current computational platforms to the new Fugaku system, OSS is used in various applications as a significant building block for parallel programming, and some of them need to be tuned for high performance by using the A64FX capabilities. Through the interviews and questionnaires in CoDesign, we have tried to understand the OSS that need to be tuned and the functions that cannot be covered by SSL II and RIKEN developments. In fact, porting many OSS is supported by the Spack framework, but Fujitsu has contributed significantly to the SVE vectorization of FFTW, which is exceptionally performance-oriented.

5.2 Netlib on Fugaku

For de facto standard software distributed by so called netlib, such as BLAS, LAPACK, and ScaLAPACK, we have tuned each function to reflect the hardware performance of Fugaku and users' demands. In particular, for the BLAS kernel designed for a single core, we conducted assembler-level programming and assembler tuning using SVE instructions in order to exploit the computational performance of the processor. In addition, DGEMM, which is highly demanded in many applications, has been tuned not only for square matrix conditions, but also for special cases such as small size and tall-skinny shapes. ARMv8 also supports IEEE754 half precision (so-called half precision, FP16). In this development, the API and the kernel code of the BLAS library that supports FP16 has been determined and has been implemented, respectively.

5.2.1 BLAS/LAPACK/ScaLAPACK

BLAS (Basic Linear Algebra Subprograms) is a set of subroutines that extract the basic operations used in linear calculations. Since these are forming the kernel part that accounts for

Table 5.1: Typical software packages included in the Fujitsu software stack

Functionality	Library name	Summary
Sequential libraries	SSL II	Covering broad range of computational fields. A numerical library containing approximately 300 types of Fortran routines.
	C-SSL II	C interface for SSL II
	BLAS, LAPACK	A de facto standard numerical linear algebra routines developed in the US, and published on the Netlib site. Approximately 80 and 400 routines are contained in BLAS and LAPACK, respectively.Some of them support fp16.
	Quad-precision library	A library that represents quadruple-precision numbers in double-double format and performs operations on them. It includes some thread parallel routines.
Thread parallel libraries	SSL II thread-enhanced version	Thread-parallel version with about 80 routines performing performance-critical functions. It provides a separate interface to the sequential version of SSL II for mixed use.
	C-SSL II thread-enhanced	Thread version of C-SSL II
	BLAS, LAPACK	The same interface as the sequential version, including PLASMA, a task-parallel version of LAPACK, also available in Python (NumPy, SciPy). Some fp16 kernels are supported.
MPI-parallel libraries	SSL II/MPI	3D FFT
	ScaLAPACK	An MPI parallelized libraries with BLAS and LAPACK functionality. About 200 kinds of routines are available.

Table 5.2: Typical software packages included in the RIKEN software stack

Functionality	Library name	Summary
Eigenvalue solvers	EigenExa	Eigenvalue library for massively parallel environments. It supports two schemes based on "tri-diagonalization" and "penta-diagonalization" routines. A new routine is adopted to improve the scalability compared to the previous version in the K computer era.
	Kevd	Eigenvalue library for thread-parallel execution, offering an API almost equivalent to LAPACK. It employs a numerical algorithm suitable for high performance on an A64FX. Mainly optimized for the problem size in the DA part of NICAM-LETKF.
Enhanced/optimized BLAS kernels	Batched BLAS	Batch-typed BLAS library, which executes multiple small-scale problems simultaneously. It provides all BLAS API's with variable- and fixed-type batch functions.
	2.5D-PDGEMM	A low communication latency parallel matrix-matrix product routine using a 2.5-dimension algorithm. It is partially compatible with PBLAS (PxGEMM), and supports various process grids and matrix shapes with some restrictions.

a high percentage of each linear calculation function's computational cost, a group of routines with particularly focused performance tuning was materialized. A thread-safe sequential version library for use in OpenMP-parallel regions or flat MPI execution and a thread-parallel version library were realized considering ILP64 and LP64 compatibility.

We have implemented a sequential version and a thread-parallel version of LAPACK (Linear Algebra PACKage), a numerical linear algebra library. In addition to the policy of improving performance through tuning and threading of the internally called BLAS, the following tuning was performed for the basic routines.

1. Block size adjustment (sequential, parallel),
2. Modification of the processing configurations (sequential, parallel),
3. Parallelization in the LAPACK layer (parallel), and
4. Reduction of the 'OMP PARALLEL' overhead (parallel)

ScaLAPACK has been implemented as a numerical linear algebra library with MPI and Hybrid parallelism. The basic development policy was to use highly-optimized BLAS and MPI for Fugaku to ensure performance, and to tune the ScaLAPACK layer as necessary.

5.2.2 FP16 extension to BLAS kernels

The following ten BLAS routines are to be enhanced to FP16. However, since the Netlib version of BLAS has no FP16 routines, it was decided to use the naming rule (BLAS function name R16) discussed for the next generation BLAS at present, with the prefix FJ (for Fortran) or fj (for cblas). The three routines, GEMV, GER, and GEMM, were tuned for assembler.

- SWAP (`FJBLAS_SWAP_R16`) $x \leftrightarrow y$
- SCAL (`FJBLAS_SCAL_R16`) $x \leftarrow \alpha x$
- COPY (`FJBLAS_COPY_R16`) $y \leftarrow x$
- AXPY (`FJBLAS_AXPY_R16`) $y \leftarrow \alpha x + y$
- DOT (`FJBLAS_DOT_R16`) $\mathrm{dot} \leftarrow x^\top y$
- ASUM (`FJBLAS_ASUM_R16`) $\mathrm{asum} \leftarrow |\mathrm{Re}(x)|_1 + |\mathrm{Im}(x)|_1$
- AMAX (`FJBLAS_AMAX_I32_R16`) $\mathrm{amax} \leftarrow \mathrm{argmax}\{x(k)\}$
- GEMV (`FJBLAS_GEMV_R16`) $y \leftarrow \alpha Ax + \beta y$ (general matrix)
- GER (`FJBLAS_GER_R16`) $A \leftarrow \alpha xy^\top + A$ (general matrix)
- GEMM (`FJBLAS_GEMM_R16`) $C \leftarrow \alpha AB + \beta C$ (general matrix)

5.2.3 Additional extensions to the BLAS routines

As a result of the codesign efforts, several function extensions that received a lot of feedback from users have been implemented.

1. The matrix copy and transpose routines:
   The library supports matrix- copy and transposition functionality similar to the `mlk_Xomatcopy` routine in Intel MKL for half-precision floating-point, single-precision real, double-precision real, single-precision complex, and double-precision complex types, and has sequential and thread-parallel versions.

2. Single-threaded API:
   A dedicated API for sequential (single-threaded calls) is provided for using both sequential and thread-parallel versions of BLAS in an application. The sequential-only version of BLAS can be managed under an alias to ensure that the sequential and parallel versions can be called separately in a single program.

3. Matrix-matrix product function for SIMD length unit real imaginary separated array for complex numbers:
   This provides a GEMM-compatible function that takes as input and output a complex matrix of data structure in which the real and imaginary parts are held consecutively for SIMD length. The data of the real part '`R`' and the imaginary part '`I`' are managed as an AOSOA (Array Of Structures Of Arrays) layout such as `RRRRRRRRRRIIIIIIIIRRRRRRRRRIIIIIIII`.

5.2.4 Preliminary performance

Figure 5.1 shows the performance of GEMM kernels, a BLAS matrix product routine. The BLAS matrix product routine performs on an A64FX processor so that the double-, single-, and half- precision arithmetic show the optimal performance according to the theoretical peak, which is equally proportional to the physical SVE unit length 8, 16, and 32 words, respectively. Sustained performance achieved roughly 710Gflop/s, 1.58Tflop/s, and 2.81Tflop/s, for DGEMM, SGEMM, HGEMM, respectively, at CPU frequency 2.0 GHz a single CMG with N=5,120 and LDA=5,184.

In addition for performance of matrix-vector product of sufficiently large size for DGEMV as a representative routine from Level 2 BLAS, 50.15 Gflops was obtained at CPU frequency of 2.0 GHz with 12 thread (1 CMG/1 node) execution. This is 6.52% of the theoretical peak performance.

5.3 SSL II on Fugaku

SSL II, which consists of about 300 routines covering 10 fields of numerical analysis, is a high-performance library for scientific and technological calculations provided by Fujitsu on the Fujitsu platform. In this project, SSL II is developed and provided on Fugaku. For example, as well as the netlib development, we have tuned the performance to the A64FX CPU architecture by using the SIMD instruction generation built-in function. In addition to porting and tuning the accumulated assets, we intend to continue functional enhancement with a strategic perspective. The entire SSL II on Fugaku supports about 300 single- and double-precision routines covering 10 major areas of numerical analysis. Various versions are available in thread parallel, C/Fortran, and MPI distributed parallel versions as shown in Table 5.1.

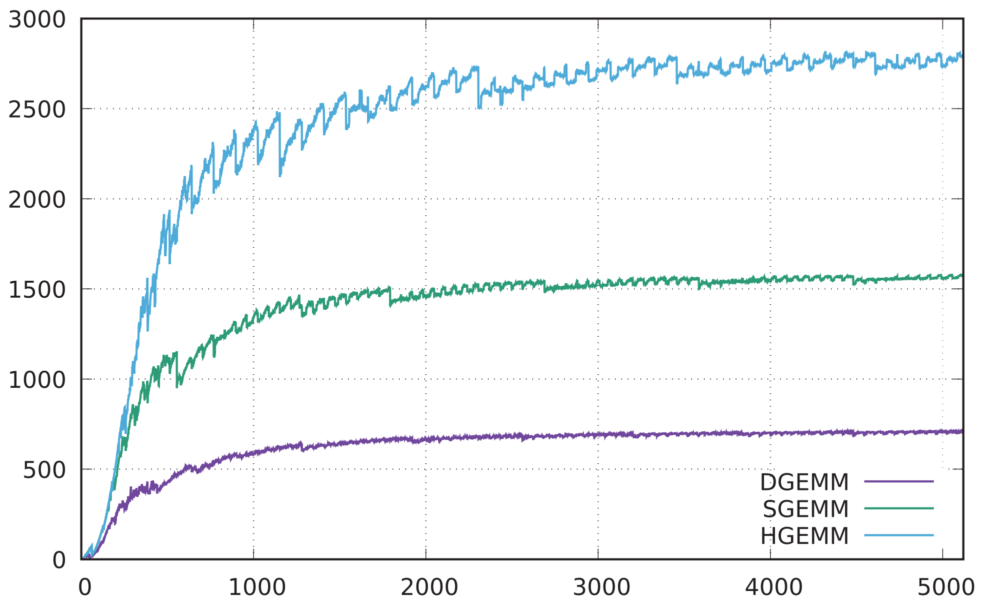


Figure 5.1: [DSH]GEM performance (Gflop/s) on a single CMG with 2.0GHz mode

5.3.1 FFT on a single core / a single CMG / pencil-decomposition 3D-FFT

The FFT function is recognized as an increasingly critical subroutine in science and technology. As a configuration method, we adopted a configuration method in which the related routines are organized by utilizing the single-core FFT kernel features tuned for performance under the condition that the used arrays can be placed in the L1 cache. In addition, the existing APIs developed for the K computer were inherited to call sequential and thread parallel functions, while the pencil-shape decomposed three dimensional MPI parallel functions were developed in response to requests of the application developers. In particular, the most important implementation point was how to realize continuous data access and uniform stride access suitable for the internal data format and the SVE instruction in the kernel that must handle real and complex numbers simultaneously.

Figure 5.2 demonstrates the achievement of excellent performance improvement by comparing naive compilation of FFTW3, FFTW3+SVE optimization, and Fujitsu's SSL II. In particular, the 1d FFT kernel in the SSL II library performs 3.3 fold faster than the naive compilation of FFTW3. It also exploits the fact that SVE provides noticeable performance improvements even for complex algorithms.

We have also developed a three-dimensional FFT with two-axis distribution in the input and output data, following the advancement of the SSLII distributed parallel FFT function. The main purpose of the implementation is to reduce the amount of communication. Both input and output data are distributed in two axes, namely pencil decomposition or pillared decomposition. The local input and output arrays of each process are stored in different shapes, which are generated by dividing the global data into columns in different directions. When the size of each dimension is not divisible by the divisor of each axis, the local shape and the amount of data held by each process are adjusted internally to balance the workload by varying the input and output. Since the `MPI_Comm_split` cost is visible in small-scale problems, the sub-communicators in each axis direction are passed as arguments.

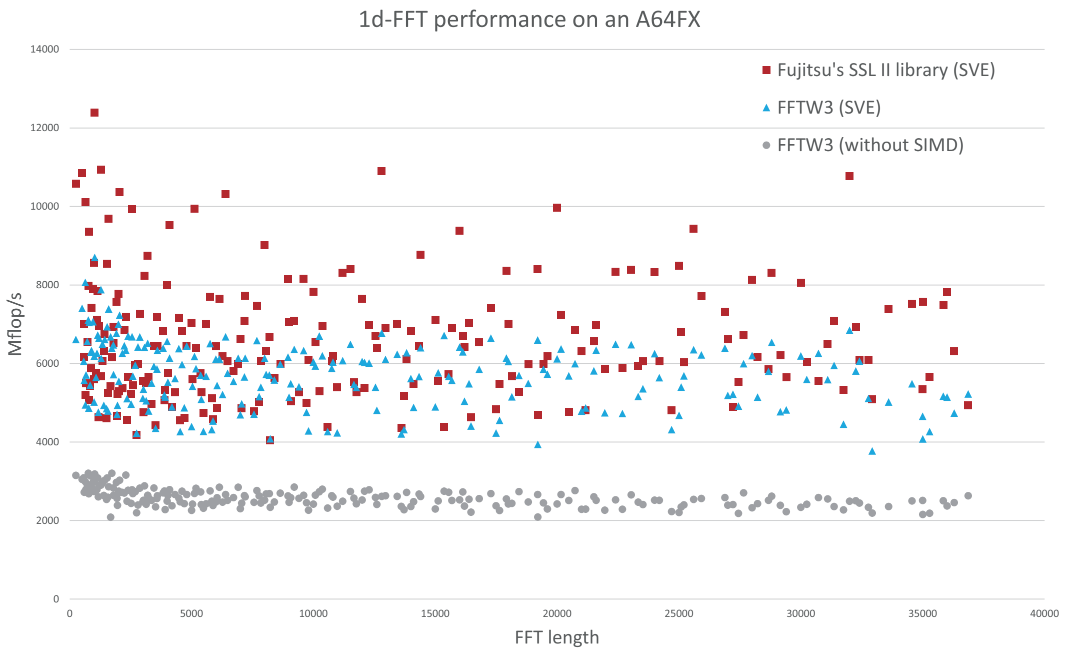


Figure 5.2: 1d-FFT kernel performance benchmark on a single core of an A64FX

5.3.2 Pseudo Random Number Generator (PRNG)

As a consequence of investigating possible ways to use each parallel hierarchy for the pseudo-random number generator, we have added a new random number routine `dm_vranu5` using MRG8 (Multiple Recursive Generator with 8 th-order full primitive polynomial) to the existing random number routine (`dvranu4/dm_vranu4`) in SSL II.

MRG8 was invented originally by Miura[44], and its principle is based on a recurrence that $x_n = a_1x_{n-1}+a_2x_{n-2}+a_3x_{n-3}+a_4x_{n-4}+a_5x_{n-5}+a_6x_{n-6}+a_7x_{n-7}+a_8x_{n-8} \mod(p)$ providing a 4.5×10^{74} period and passing the TESTU01 test program perfectly. It also generates random numbers with an arbitrary interval, and parallelization and vectorization are pretty straightforward. Another PRNG algorithm available on many platforms is Mersenne Twister developed by Matsumoto and Nishimura [45], which offers several variational PRNG packages such as MT, SFMT, dSFMT, TinyMT, and MTGP, showing a very very long period as 10^{6001}. We confirmed that some packages from them, for example, dsFMT, perform on A64FX platforms without any modifications.

5.4 Math functions linked with compiler

The following two speedups have been made for Fugaku for the built-in mathematical functions called from compiler objects.

1. Acceleration of basic mathematical functions:
   The primary basic mathematical functions have been tuned for inline expansion of built-in functions and multi-operator functions, taking advantage of SVE features (dedicated instructions and long SIMD instructions).

2. Accelerating MATMUL:
   Linking tuned BLAS matrix product routines to Fotran's MATMUL library.

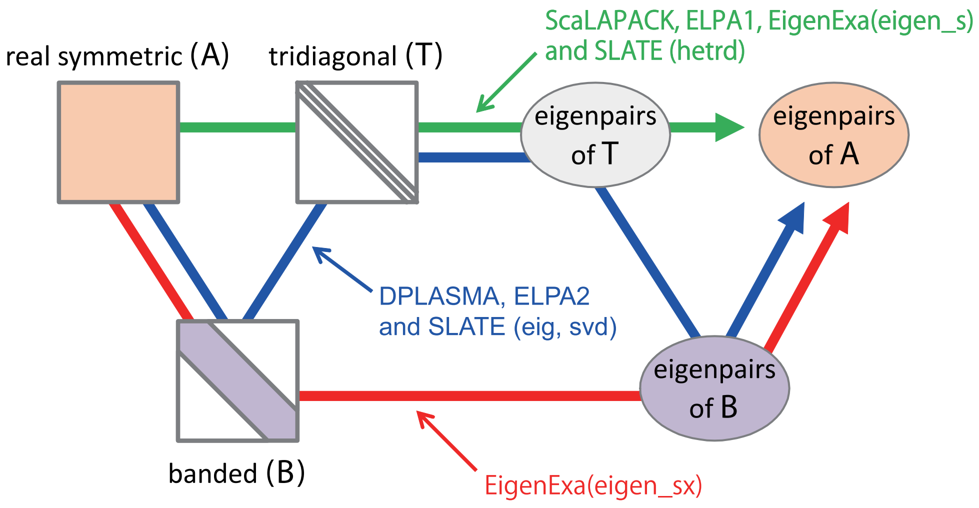


Figure 5.3:

5.5 EigenExa on Fugaku

As a result of the survey with application users and codesign with computational science community, it has been clarified that eigen- and singular- value decomposition routines are also frequently used. These routines are included in LAPACK and ScaLAPACK[46], which are the standard numerical libraries, but there are no fast implementations (in terms of realistic aspects) for thread parallelism, distributed parallelism, or hybrid parallelism in highly parallel systems. Since singular value computation can be derived from eigenvalue computation, we eventually develop highly parallel versions of both eigenvalue computation and singular value computation routines at the same time, utilizing the knowledge of EigenExa developed by RIKEN [47, 48]. As is the case with EigenExa on the K computer, MPI+X(X=OpenMP), so called hybrid parallelization, is guaranteed to perform on most of systems, and it enables to perform on an arbitrary number of processes and two-dimensional process grid.

5.5.1 Quick review, Background of State-of-the-arts engensolvers

EigenExa contains the traditional dense-matrix eigenvalue solvers, mainly Householder tridiagonalization and Cuppen's divide-and-conquer method as well as other state-of-the-art libraries, ScaLAPACK and DPLASMA [49] by ICL U. Tennessee, ELPA [50] by the German group, and SLATE [51] by Exascale Computing Project (ECP).

To be more specific, essential steps of the eigenvalue calculations are schemed as follows (Fig.5.3). At first, the conventional scheme, highlighted in green, goes through real symmetric (A) to tridiagonal (T) via Householder tridiagonalization, to eigenpairs of T via Cuppen's divide-and-conquer algorithm (or QR, bisection, MRRR, etc.), finally eigenpairs of A via Householder backward transformation. This scheme is the most conventional one in the dense eigenvalue calculation (`PDSYEVD` in ScaLAPACK, ELPA1, SLATE(`hetrd`), and `eigen_s` in EigenExa). Although it offers freedom of algorithm choice in the intermediate second stage, the divide-and-conquer method is usually chosen to consider parallel performance and numerical stability. It is endorsed by its high parallel performance and successful history. On the other hand, the bottleneck of this scheme is the Householder tridiagonalization part, where principle computing patter is to compute a vector v and rank-2k update of a matrix such that

$$v = \left(Au - \frac{1}{2}u\beta(u^\top Au)\right)\beta, \tag{5.1}$$

$$A \leftarrow A - UV^\top - VU^\top. \tag{5.2}$$

The former computational pattern is known to be bound to communication, whereas we should be strongly aware of memory- and network- intensive computing.

To overcome the memory bound challenges, modern eigenvalue solvers employ a scheme that performs a two-step transformation: dense-to-band-to-tridiagonal. This is highlighted by the blue line in Fig.5.3. Major difference between the above-mentioned conventional scheme is dense-to-band reduction (A to B) by a block Householder transformation and backward transformation (eigenparis of B to those of A). In contrast to eq. (5.1), the construction of vector v is replaced by the blocked version as,

$$\tilde{V} = \left(A\tilde{U} - \frac{1}{2}\tilde{U}C^\top(\tilde{U}^\top A\tilde{U})\right)C. \tag{5.3}$$

Although the structure is similar to that of eq. (5.1), the significant performance bottleneck caused by memory traffic is avoided by restructuring the algorithm using matrix-matrix operations. This bypass procedure is implemented as `dsytrd_2stage` in LAPACK (version 3.8 later), `heev` in SLATE (but band-tridiagonal part is only provided as a single node functionality), and `bandred_real_double` + `tridiag_band_real_double` in ELPA2. There are several algorithm at the second reduction from a band to a tridiagonal form, such as Rutishauser's method and Murata-Horikoshi s' method. These are generally called 'bulge chase' algorithms. The computational complexity is estimated as roughly $O(kN^2)$ (k refers to the semi-bandwidth of the banded matrix), which is not dominant compared to the first reduction. On the contrary to the forward transformation (dense-banded-tri), the backward transformation of the tri-to-banded phase is not a negligible part, where the computational cost accounts for $O(N^3)$, analytically equaling to that of the banded-to-dense back-transform. Meanwhile, together with the implementation difficulties, the band-tri-band intermidiate transformation scheme is inefficient under high computational loads, such as when computing all eigenvectors. Note, though, that this is not the case when computing a small number of eigenvalue modes or when sufficient machine-specific tuning is performed.

The third approach highlighted in red is our EigenExa method (`eigen_sx` kernel), which avoids the band-to-tri-to-band transformation. Since it directly calculates the eigenvalues of the band matrix, it is unnecessary to have intermediate tridiagonal format, but necessary to introduce a new solution method (divide-and-conquer method for band matrices) different from the conventional method, while essential part of the new kernel is realized as a sequential operation of the algorithm for tridiagonal matrices with the computational complexity $O(\frac{4}{3}(2k-1)N^3)$. Accordingly, the factor k included in the complexity term works negatively, but we must consider the trade-off against the speed-up achieved by blocking in the dense-to-band transformation.

5.5.2 New implementation of the divide and conquer method

The Cuppen's divide and conquer method [52] is briefed by combining the rank-one perturbation theory and recursive merging of sub-eigenproblems mapped onto a binary tree. If we split a tridiagonal matrix $T_0^{(0)} = T \in \mathbb{R}^{2^r\times 2^r}$ by the following recursive way,

$$T_i^{(j)} = T_{2i}^{(j+1)} \oplus T_{2i+1}^{(j+1)} + \rho_i^{(j)} g_i {g_i}^T, g_i = e_{2^{r-j-1}} \pm e_{2^{r-j-1}+1} \tag{5.4}$$

$$T_{i'}^{(j+1)} = T_{2i'}^{(j+2)} \oplus T_{2i'+1}^{(j+2)} + \rho_{i'}^{(j+1)} g_{i'} {g_{i'}}^T, \cdots . \tag{5.5}$$

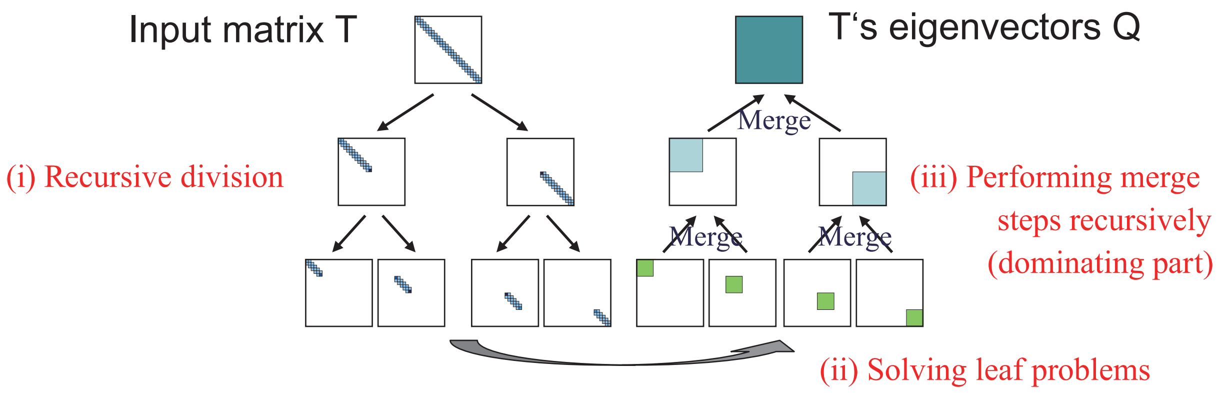

Input matrix T
T's eigenvectors Q
(i) Recursive division
Merge
Merge
Merge
(iii) Performing merge steps recursively (dominating part)
(ii) Solving leaf problems

Figure 5.4: Schematics of the Cuppen's divide and conquer method

where $T^{(j)} \in \mathbb{R}^{2^{r-j} \times 2^{r-j}}$. Eigen-decomposition such as $T_i^{(j+1)} = Q_i^{(j+1)} \Lambda_i^{(j+1)} {Q_i^{(j+1)}}^T$ yields a one-rank perturbation problem as $(Q_{2i}^{(j+1)} \oplus Q_{2i+1}^{(j+1)})^T T_i^{(j)} (Q_{2i}^{(j+1)} \oplus Q_{2i+1}^{(j+1)}) = \Lambda_{2i}^{(j+1)} \oplus \Lambda_{2i+1}^{(j+1)} + \rho_i u_i^{(j)} {u_i^{(j)}}^T = D_i^{(j)} + u_i^{(j)} {u_i^{(j)}}^T$. For a diagonal matrix with a one-rank perturnbation such as $D + \rho u u^T$, we have a trivial scheme to compute eigenvalues λ as the solutions of

$$f(\lambda) = \frac{1}{\rho} + \sum \frac{u_i^2}{d_i - \lambda} = 0, \tag{5.6}$$

and the corresponding eigenvector q can be calculated as

$$q = (D - \lambda I)^{-1} u. \tag{5.7}$$

If we solve for all the leaf problems on the same level on the binary tree, we can recursively construct $\Lambda = \{\lambda_1, \lambda_2, \cdots\}$ and $Q = [q_1, q_2, \cdots]$, which will be merged into the upper-level problem. Figure 5.4 schematics the overview of the DC method. Since this process allows full parallelism on solving individual problems, highly efficient parallel processing is anticipated by appropriate data partitioning and task assignment.

We had a couple-year evaluation and internal analysis of the divide and conquer method routines in EigenExa, then have identified some problems; i) Due to the use of data distribution method fixed with the ScaLAPACK/BLACS context, some processes are idle when the processing small matrix products. ii) The independently executable processes are not executed in parallel, resulting in performance saturation immediately when the number of executing processes is increased. iii) Furthermore, some processes that are supposed to be logically neighboring may be located on physically distant processors, and the congestion on the same network routing will lead to performance degradation.

We have decided to develop a new divide-and-conquer routine that inherits the efficiency of the eigenvalue calculation library ELPA, but with a natural extension that is highly maintainable and flexible.

1. The data distribution method for eigenvectors during the merging process is a generalized form of the ELPA method, while the actual execution of the matrix product is done in parallel using one-way ring communication.

2. The partitioning order of the process grid is not fixed in either the vertical or horizontal axis, and dynamic flexibility allows the partitioning method to be adapted to the process grid information at runtime.

3. Generate concise and easy-to-maintain code that reflects the recursive structure of the merging process.

To be more specific in the implementation, whole process/design of our new parallel divide and conquer is summarized in three steps as follow.

5.5.2.1 Recursive partitioning

Perform a recursive partition so that the number of leaf problems matches the number of processes.

- For a given process grid, generate a tree structure of process rows/columns by partitioning vertically or horizontally, in short, alternatively split along the x-/y- axis, to determine the process to solve the partitioned leaf problems.
- Get the process row/column of the block-split position in the overall matrix by referring to the process id assigned to the leaf problems.
- Similarly, determine the process responsible for the split block in the overall matrix by referring to the obtained overall process row/column number.

5.5.2.2 Solve the leaf problems

Each leaf problem is solved by a local divide-and-conquer solver that works in the process. The local divide-and-conquer solver uses LAPACK's DSTEDC subroutine.

5.5.2.3 Recursive Merge

Recursive merging is done in a breadth-first manner. The eigenvectors of the subproblems that appear along the way are stored in the given storage area in the following distributed data structure

- Determine the block responsible for each process matrix in the final overall matrix shape when the recursive partitioning is performed.
- The eigenvectors of the subproblems that appear along the way are stored in the corresponding array positions in the overall matrix shape.

5.5.2.4 Matrix-products on a Ring topology

Eventually, as ELPA implemented, the high-speed matrix products based on the ring communication is significant to obtain high-performance, and it is derived from the following implementation keys.

- When computing $Q = Q_2 \times S$, ring communication is used in the process column for Q_2 and on-the-fly computation is used for the column corresponding to Q_2 for S.
- During the on-the-fly computation of S and local matrix product computation by DGEMM, asynchronous communication is performed for Q_2 corresponding to the next process column to be computed to reduce the communication time by overlapping computation and communication.

When this recursive merging is repeated, the final distributed data structure differs from the general two-dimensional block cyclic; the final redistribution into two-dimensional block cyclic or two-dimensional cyclic completes the series of processes. The merging process and the final redistribution are illustrate in Figure 5.5.

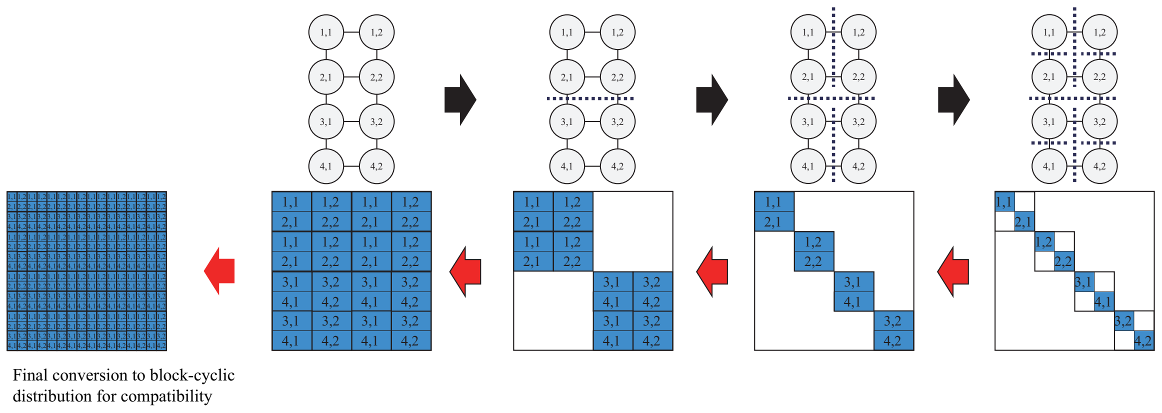

Final conversion to block-cyclic distribution for compatibility

Figure 5.5: Node topology in a 2D fashion and matrix data distribution scheme for our new DC implementation

5.5.3 Other technical considerations and development perspectives

One of the technical issues in the development of the eigensolver on Fugaku was the relative slowdown of the network performance since the K computer, in contrast to the improvement of the processor performance, which leads to the imbalance eventually. In fact, even in the small-scale supercomputers of recent years, communication has been the biggest bottleneck in the highly parallel supercomputers, and it has been recognized before the development of Fugaku that the results of various performance evaluations support the fact that this situation cannot be ignored in the scale of Fugaku. Consequently, discussions are converged into the two primary subjects as follows;

- to realize the communication avoidance technology for Fugaku in the householder tridiagonalization, and
- to adopt a new process mapping method that is attentive to improving the load balance of the divide and conquer method.

These techniques have yielded significant performance improvements in a highly parallel environment on the scale of the K computer. These are also expected to provide significant performance improvements on Fugaku for practical scale problems. Since the early stage of the development of EigenExa, it has been developed using various parallel programming languages and libraries such as MPI, OpenMP, high performance BLAS, as well as general SIMD vectorization and Fortran90 compiler technology. Along with these technical advantages, long vector technology with SVE, which was enhanced in an A64FX processor, and support for NUMA environment consisting of multiple CMGs are newly arising technical issues from Fugaku. Lastly, as an addition, we faced on a numerical difficulty of bitwise-reproducibility on an Intel Cluster environment, which takes advantage of Intel compiler and Intel MPI as well.

5.5.4 Preliminary performance benchmark

Fig. 5.6 shows the preliminary benchmark results of the overall performance using one of the beta version 2.6c, which is the new implementation for the Fugaku project and based on the latest release version EigenExa 2.7 (released on the 1st of April 2021). Figure 5.6 demonstrates the overall performance of EigenExa2.6c and PDSYEVD routine contained in ScaLAPACK using 128 to 16,384 nodes on Fugaku, on which three-dimension cases such as N=65,536, 131,072, and 262,144. However, that of PDSYEVD for the largest case was not measured due to the

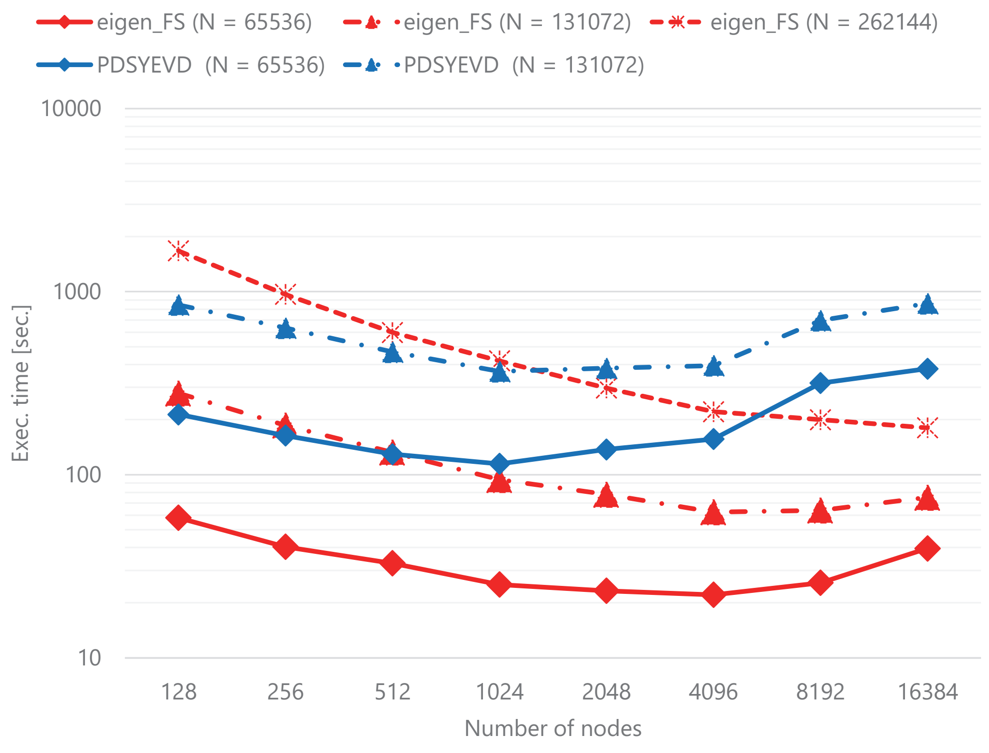


Figure 5.6: Overall performance of EigenExa2.6c on Fugaku (elapsed time unit in second)

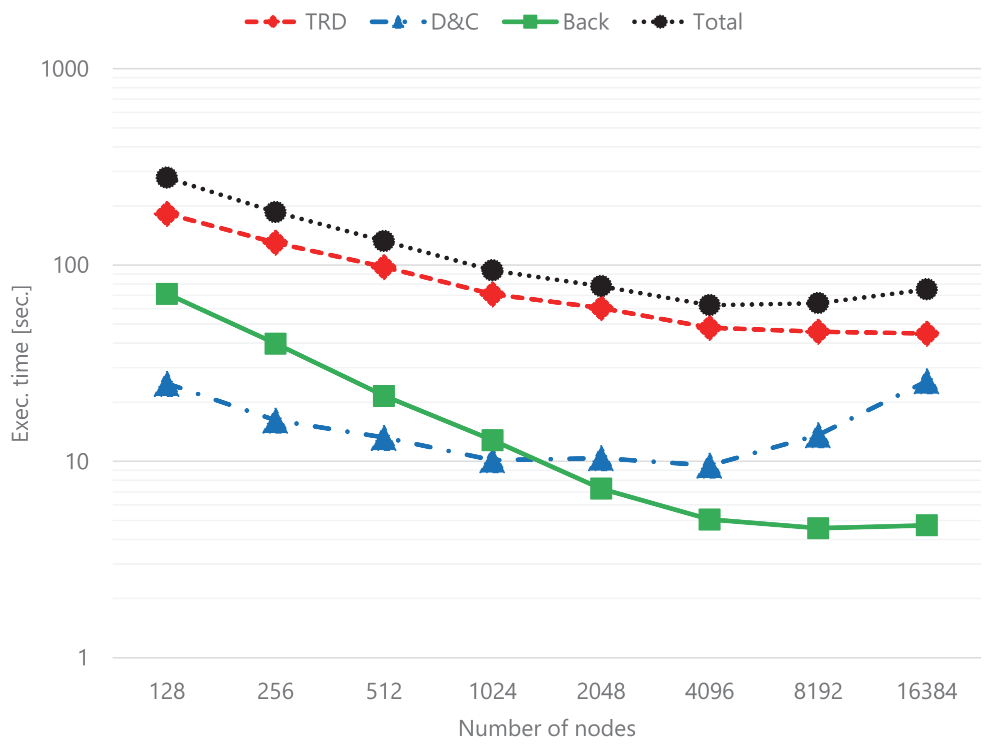


Figure 5.7: Time breakdown for three kernel routines when N=131,072

Table 5.3: Performance comparison with K computer (N=32,768 for top, and 65,536 for bottom)

Num of Nodes	128	256	512	1024	2048	4096
K (2.6)	43.56	29.84	22.18	17.32	16.65	17.24
Fugaku (2.6c) 1ppn	28.66	22.47	11.98	11.55	11.14	30.98
ratio (K/F)	1.52	1.33	1.85	1.50	1.49	0.56
Num of Nodes	128	256	512	1024	2048	4096
K (2.6)	246.31	149.32	95.32	67.79	54.66	47.02
Fugaku (2.6c) 1ppn	93.27	70.91	40.47	31.93	17.39	45.15
ratio (K/F)	2.64	2.11	2.36	2.12	3.14	1.04

inexpectation of higher performance. We can see good scalability when solving a large eigenproblem. For example, in the most significant case, when we solve a 262,144 dimension matrix, good parallel efficiency and scalability are observed from 128 to 4,096 nodes, and gradually scales down but with acceptably better parallel performance.

On the other hand, PDSYEVD yileds only 10 to 25% of the performance of EigenExa, which means that, in short, it needs four to ten folds more execution time. Also, parallel scalability is relatively poor even though that is a vendor-provide optimized routine. It is known that the other eigenvalue routines PDSYEV, PDSYEVX, and PDSYEVR have disadvantages in terms of parallel and absolute performance. Therefore, we conclude that ScaLAPACK needs a drastic restructuring if it continues to be used as the de facto standard eigenvalue solver on Fugaku.

The Figure 5.7 illustrates the time breakdown of the three main routines of EigenExa2.6c. As in the performance evaluation of the K computer, the Householder tridiagonalization is the most dominant kernel, and the backward transformation part, in which DGEMM is the primary component, requires the least computation time[53]. It can be seen that the divide-and-conquer method developed in the project has improved the absolute performance in the small to medium scale region, while the performance has not improved in the highly parallel environment as not expected initially. Further analysis is needed, but in the mapping of the binary tree partitioning between the process map and the Tofu topology, conflicts and overlaps between sub-process groups may occur in the case of high-order processes, which could be the cause of the performance degradation.

The performance comparison of the K computer should be fascinating. Table 5.3 shows the performance comparison for a small problem in Fugaku. Since the problem domain is larger in the K computer, it is not fair to make some comparisons, but roughly speaking, we can say that this domain's performance difference is a factor of two or more. This is consistent with the growth rate of the network bandwidth, and if the problem size is further increased, the performance is expected to improve up to 16 times the growth rate of the memory bandwidth.

5.6 Batched BLAS

Fugaku is primarily empowered by many-core processors an A64FX. To boost linear algebra operations in a sense of thread-level parallelism on a single node, we must adopt a more strategic way to assign a large number of small-size BLAS tasks or their internal sub-tasks into each physical core, taking up all of the core resources rather than 'single-job on single-core' scheduling. The Batched BLAS would offer a new scheduling form of linear algebra operations based on such a mechanism. The practical application of experimental Batched BLAS, part of which has already been used in AI and other applications, was eagerly awaited.

Since the Batched BLAS was not collected in the Fujitsu numerical library (SSL II), the

RIKEN development team decided to implement a full version of the Batched BLAS based on a design policy that could be easily applied to the not-yet-standardized specifications in 2017. We have made a poster presentation regarding the implementation and evaluation of Batched BLAS at ISC18 [54] to accelerate AI applications running on Fugaku. Although our implementation did not support a current standard API, we mentioned the possibility of a methodology and some scheduling techniques that would support all APIs before the standard specified all the APIs from level 1 to level 3 kernels. In July 2018, the formal standard [55] was released, and we have also responded to adjust the APIs to the formal standard (the reference implementation of NLAFET [55]), resulting in the publication of the package in February 2021. In addition, we partially support an FP16 enhancement, which has been adopted in IEEE754-2008 as half-precision or binary16.

Currently, our Batched BLAS implemenation is available on not only for Fugaku but other major systems based on many/multi-core processors such as Intel Xeon, an Intel Core, AMD RYZEN, ans so on. This software was released as OSS in February 2021 [56]. The number of available API is more than two hundred. A typical batched API for `DGEMM`, the matrix-matrix multiplication routine, has the following call formats in Fig. 5.8.

```
void blas_dgemm_batch(
    const int group_count, const int* group_size,
    const bblas_enum_t layout,
    const bblas_enum_t* transa, const bblas_enum_t* transb,
    const int* m,const int* n, const int* k,
    const double* alpha, const double** a, const int* lda,
    const double** b, const int* ldb,
    const double* beta, double** c, const int* ldc,
    int *info)
```

Figure 5.8: A typical API of Batched BLAS

The argument parameters refer to arrays with the `group_counts` described below. On the other hand, the layout parameter is the same across the board, regardless of the group as illustrated in Fig. 5.9. Since each function's core part is generated from a few templates, the Batched-X approach is extensible to any other thread-safe kernels. Preliminary performance was examined on an A64FX, where computational cores are bound in a a specific CMG by numactl command (see Figure 5.10).

5.7 2.5D-PDGEMM

On massively parallel systems such as the supercomputer Fugaku, the performance of a problem that is computation-bound on conventional systems may become communication-bound when the problem size is insufficient for the degree of parallelism. In addition, compared with the K computer, as the balance between communication and computation performance of the supercomputer Fugaku is more computation-oriented, the computation performance can easily become communication-bound. This problem can occur even in matrix multiplication, which is a fundamental kernel in numerical computing and is known as a computation-intensive task.

To address this issue, a communication-avoiding distributed matrix multiplication algorithm, the 2.5D algorithm[57], has been proposed. The algorithm replicates and stacks the 2D distributed matrices on the 3D process (2.5D distribution), in contrast to the conventional algorithm (2D algorithm) that distributes the 2D distributed matrices on the 2D process in two

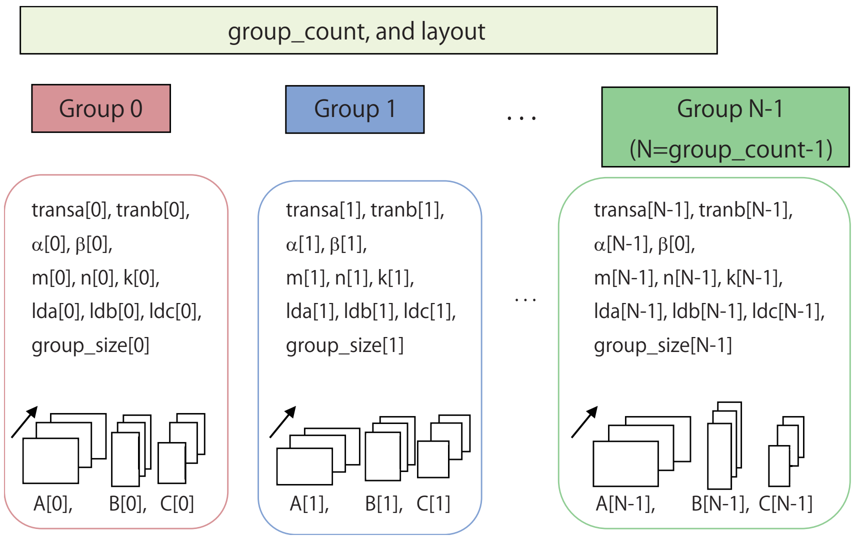

group_count, and layout
Group 0
Group 1
Group N-1
(N=group_count-1)
transa[0], tranb[0], α[0], β[0], m[0], n[0], k[0], lda[0], ldb[0], ldc[0], group_size[0]
transa[1], tranb[1], α[1], β[1], m[1], n[1], k[1], lda[1], ldb[1], ldc[1], group_size[1]
transa[N-1], tranb[N-1], α[N-1], β[0], m[N-1], n[N-1], k[N-1], lda[N-1], ldb[N-1], ldc[N-1], group_size[N-1]
A[0], B[0], C[0]
A[1], B[1], C[1]
A[N-1], B[N-1], C[N-1]

Figure 5.9: Schematic of batched arguments, layouts, and groups for Batched-GEMM

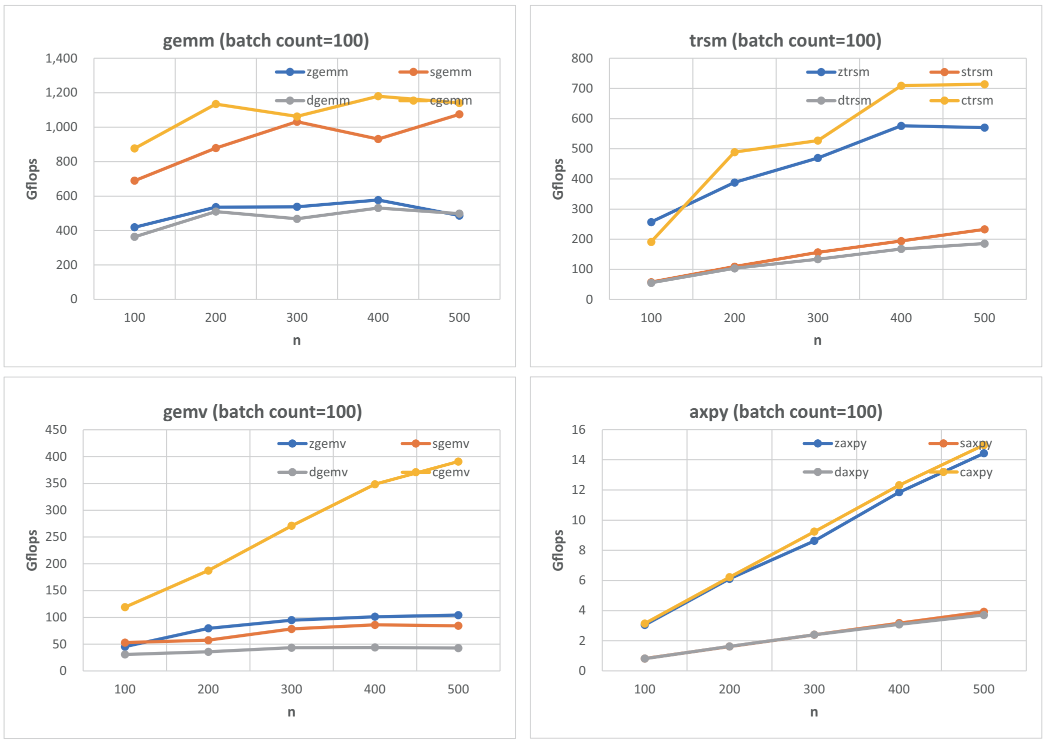


Figure 5.10: Preliminary benchmark results of Batched BLAS kernels on an AFX64 with a single CMG binding

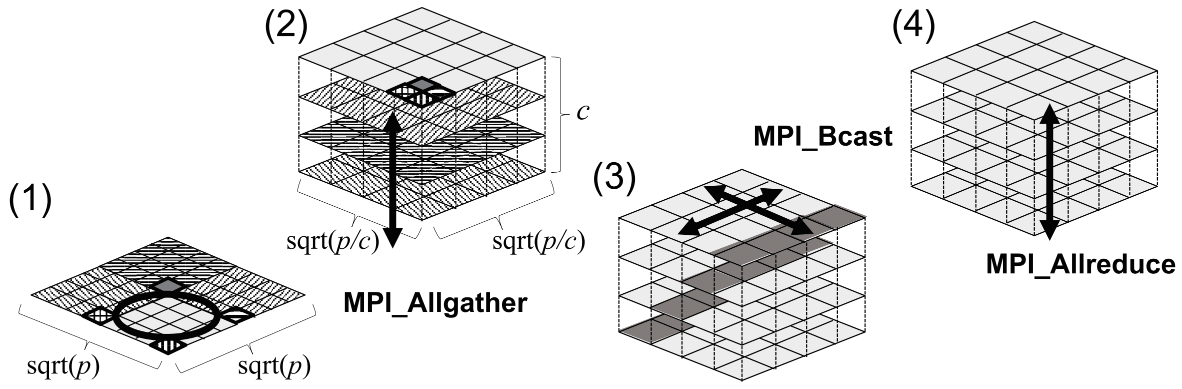


Figure 5.11: Implementation

dimensions. The conventional 2D algorithm completes the inner products in a matrix multiplication by circulating matrix elements through communication by steps. On the other hand, the 2.5D algorithm uses the same principle of the conventional 2D algorithm on each 2D process grid, but takes advantage of the data redundancy of the 2.5D distribution to reduce the number of communication steps by executing the inner product in parallel using the vertical processes corresponding to the number of stacks. For instance (see Figure 5.11), on a 2D process grid with $\sqrt{p} \times \sqrt{p}$ processes, the 2D algorithm requires $O(\sqrt{p})$ communications. On the other hand, on a 3D process grid with $\sqrt{p/c} \times \sqrt{p/c} \times c$, where c corresponds to the stack size or the vertical processes, the 2.5D algorithm takes $O(\sqrt{p/c^3})$ communications. However, the 2.5D algorithm requires c times more memory than the 2D algorithm since the matrices A and B are replicated in stacks on each 2D process grid and the intermediate results are also placed on each stack.

To use the 2.5D algorithm, the matrices need to be distributed on a 3D process grid with the 2.5D distribution. In order to substitute for ScaLAPACK's distributed parallel matrix multiplication routine (PxGEMM), which uses the conventional 2D distribution, the matrix redistribution between the 2D distribution and the 2.5D distribution on a 2D process grid is needed. Therefore, we have developed a PDGEMM routine with the 2.5D algorithm that supports the 2D distribution (2D-compatible 2.5D-PDGEMM). The 2D-compatible 2.5D-PDGEMM adopts the SUMMA algorithm [58]. Figure 5.11 shows the schematic of our implementation. It executes the following three steps. First, it creates a virtual 3D process grid on the initial 2D process grid and then redistribute the matrices from the 2D distribution to the 2.5D distribution on it using MPI_Allgather ((1)–(2) in the figure). The matrices are duplicated in stacks on each 2D grid on the 3D grid. Second, the 2.5D algorithm based on the SUMMA algorithm is executed on each stack, which consists of MPI_Bcast and DGEMM ((3) in the figure). Finally, the intermediate results on each 2D grid are reduced and redistributed using MPI_Allreduce ((4) in the figure). Currently, for simplicity, our implementation supports only the number of processes that can create a 3D process grid from a 2D process grid without any remaining processes.

Figure 5.12 shows the strong-scaling performance on the supercomputer Fugaku (it is evaluated on the pre-evaluation environment with language version 1.2.25-02 and MPI version M0514. This is the performance at the development stage, and the performance of the one available after the start of the full operation may be different). In the figure, "ScaLAPACK" is the PDGEMM routine in ScaLAPACK, and "SUMMA(c=n)" is our 2D-compatible 2.5D-PDGEMM with stack size n. "c=1" is equivalent to the 2D algorithm. "NN/NT/TN/TT" show the transpose modes for the operand matrices A and B ('N': non-transposed, 'T': transposed). You can see that the 2.5D algorithm is effective when the problem size per process is small.

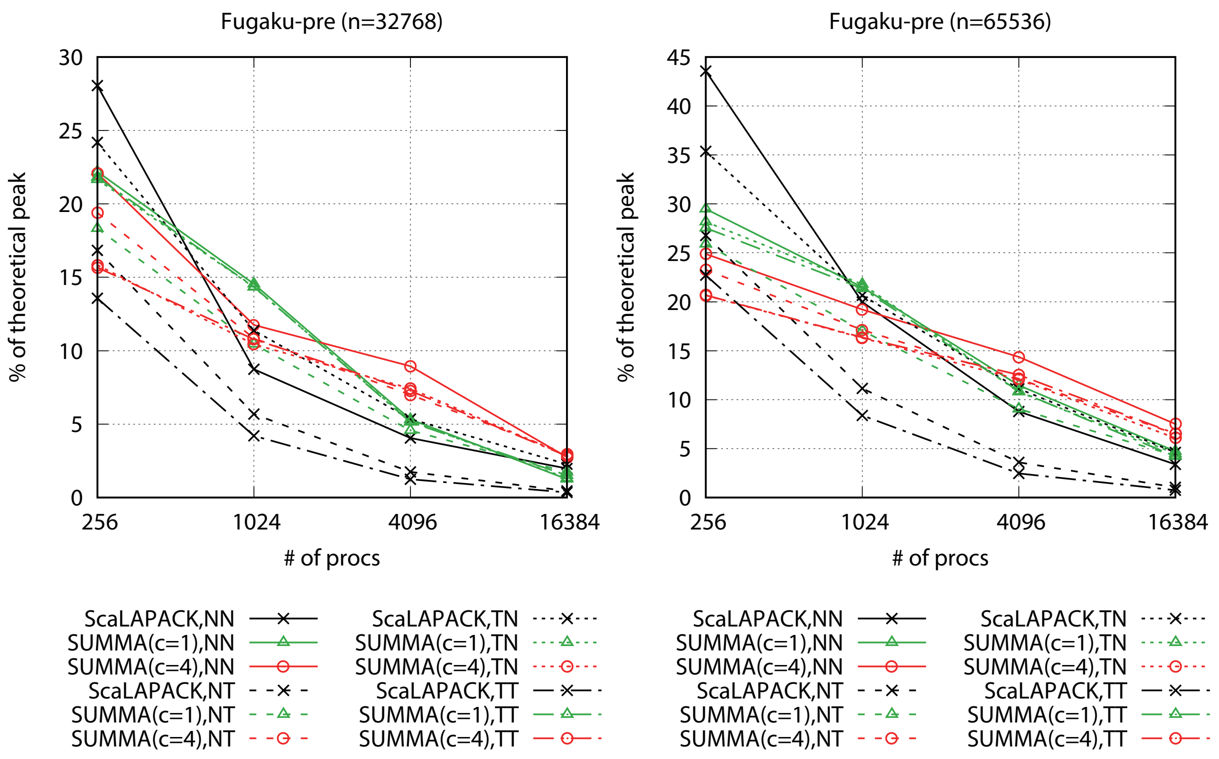


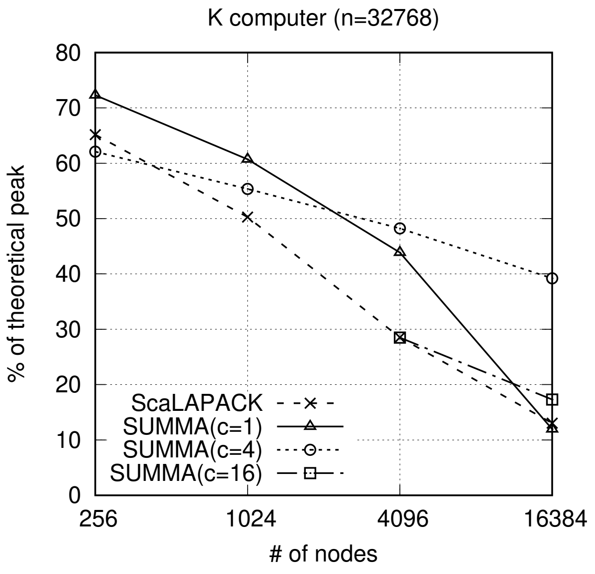
Figure 5.12: Strong scaling performance of 2D-compatible 2.5D-PDGEMM on the supercomputer Fugaku (on the pre-evaluation environment).

In the development stage, the performance has been evaluated and analyzed using the K computer in RIKEN [59] and the Oakforest-PACS system hosted by Joint Center for Advanced High Performance Computing (JCAHPC) [60]. Figure 5.13 shows the performance of "NN" mode on both systems. This result shows that the 2.5D algorithm is more effective on Oakforest-PACS than on the K computer because the Oakforest-PACS is more computation-oriented in terms of the balance between the computation performance and the communication performance. The details of the performance are analyzed in each paper [59][60].

In order to take advantage of the 2.5D algorithm in existing applications using ScaLAPACK's PDGEMM, a 2D compatible implementation such as the one we have developed will be beneficial. Our implementation will be released as open source and is expected to contribute to speeding up applications in massively parallel computing environments such as Fugaku.

5.8 Surveys, interviews, and questionnaires in Codesign

Based on the consideration for codesign with applications, the important OSS libraries are maintained by prioritizing the requests from the priority issue applications that RIKEN is developing. The selection of OSS is based on the importance of the applications' requests and the capability to large-scale computation, in which a certain amount of care for computational performance and accuracy are required. Following the priority of the individual routines, the following classification was conducted: 1) simple porting only, 2) tuning of the source code modification to improve the performance of the core part, 3) algorithm change for the part where simple tuning is not enough, and the decision on the performance improvement work was taken according to the priority.



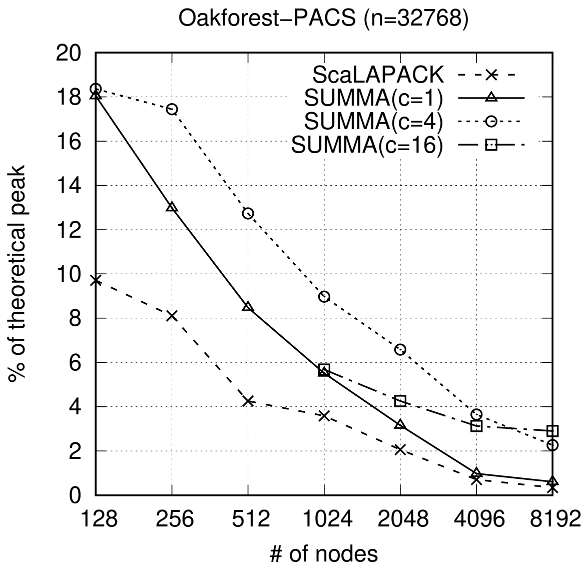


Figure 5.13: Strong scaling performance of 2D-compatible 2.5D-PDGEMM on the K computer and Oakforest-PACS (Note: in these evaluations, the performance excludes the cost for creating a virtual 3D process grid, unlike Figure 5.12)

5.8.1 Demand from nine priority issue applications

During the design phase of our numerical libraries, we had interviews among major application users and developers of prioritized application codes. What is more, we conducted a survey of the K computer users. From these preliminary investigations with, so called, 'nine priority issue application' developers, the initial demand of each application code can be summarized as Table 5.4.

Table 5.4: Review summary of nine priority issue applications

Code name (field)	Numerical algorithms	Typical computational properties and conditions	Key issues for the codesign
Genomon (Gene detection)	Parallel hash search, and parallel sorting	Few real number operations	This does not utilize a library for scientific and technological calculations.
GENESIS (Molcular dynamics)	3D-FFT with distributed parallelism is important. Eigenvalue, inverse matrix, erf function, B-spline, and PRNG	FFT is required to scale up to 128^3. Real FFT forward/backward. The correspondence between the data storage location and the frequency index is critical.	Library including a distributed parallel part, and FFT kernels with single precision SIMD support.

GAMERA (Seismic waves)	Solving a system of linear equations for a sparse and real system	The weights of single-precision calculations in the inner iterations of nested loops are greater.	It is challenging to support with generic interface solver functions.
NTChem (Quantum chemistry)	Dense eigenproblems, numercal integration, incomplete Gamma function, and random number generator	Currently, real numbers are the main numbers, but complex numbers are likely to be required. Eigenvalue problems and matrix product operations of about 20,000 orders are performed by 16 processes in a subcommunicator.	It is desirable that eigenvalue calculations scale on a medium scale. (p)dgemm performance is important Possibility to use the Mersenne twister random number, etc.
RSDFT (Density function theory)	Eigenvalue problems	Consider up to 200,000 dimensions, 20,000 parallel.	dgemm performance is critical.
FFB (CFD)	Solving a system of linear equations for a sparse and real system	There are many possible options for solvers and preprocessing, but so far no change in the methodology.	It is challenging to support with generic interface solver functions.
QCD (Quantum chromodynamics)	Mainly the determinant calculation of sparse matrices is crucial. In addition, eigenvalue calculation, inverse matrix, singular value decomposition, FFT, random numbers, etc.	Mainly complex number processing. Some multiprecision calculations are required.	pzgemm needs to be fast.
NICAM-LETKF (Weather/-climate)	Eigenvalue calculations, Inverse matrix, and Symmetric square root	NICAM works with double precision, and Diagonalization is several tens to several hundreds of dimensions.	Possibility to use the Mersenne twister random number, dgemm performance is critical.

Eventually, the following guidelines have been decided to be highly prioritized when we have developed and enhanced the numerical libraries on Fugaku.

- High speed eigensolver on a single node for NICAM-LETKF,
- Higher-precision matrix-matrix multiplication for ADVENTURE, and

- Orthogonzlization algorithm on distributed system for RSDFT.

5.8.2 Survey result on frequently used OSS

The following is a list of the numerical libraries (OSS and commercial software) that are most frequently used by users, or those implemented on the K computer.

- **OSS mentioned at the first survey**:
  PLASMA, DPLASMA, ELPA, Elemental, EigenExa, PETSc, Trilinos, ppOpen-HPC, Lis, MUMPS, SuperLU, UMFPACK, METIS, (P)ARPACK, z-Pares, BLOPEX, JADAMILU, SLEPc, FFTW, FFTE, FFTSS, OpenFFT, FFTS, FFTPACK, SFMT, exib, GNU MP, QD, FMLIB, MPACK, GSL
- **OSS mentioned at the interview**:
  EigenExa, ELPA, RANLUX (PRNG), NL2Sol (Non-linear Least Square), LibInt (functions for evaluation of two-body molecular integrals), PFAPACK (A library for numerically computing the Pfaffian of a real or complex skew-symmetric matrix), Boost (C++ library)
- **OSS ported on the K computer**:
  SuperLU, SuperLU DIST, METIS, ParMETIS, gmp, mpc, mpfr, UMFPACK, ARPACK, PETSc, SLEPc
- **OSS invented/developed in the Japanese HPC projects**:
  z-Pares, EigenExa, ppOpen-HPC, FFTE

5.8.3 Required numerical algorithms and parameters, etc

In the questionnaire for users, the required numerical algorithms and parameters, parallel computer resources, and others were investigated in association with the internal details of the users' applications. Among them, the major algorithms required by the users are as follows (the number of voters was 50).

- eigenvalue calculation: 13
- FFT: 12
- linear solver (system of linear equations): 9
- random number generator: 4
- matrix-matrix multiplication: 3
- numerical integration: 2
- special functions: 2
- matrix inversion: 1
- matrix diagonalization (eigensolver): 1
- dense matrix-multiplication: 1
- matrix-vector product: 1
- 3-dimensional FFT: 1

References

[42] *Netlib Repository at UTK and ORNL.* `http://www.netlib.org/`.

[43] *Fujitsu software, SSL II User's Guide (Scientific Subroutine Library).* J2UL-1903-01ENZ0(01). 2020.

[44] K. Miura. "Full Polynomial Multiple Recursive Generator (MRG) Revisited". In: *Proceedings of MCQC 2006, Ulm, Germany.* 2006.

[45] M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". In: *ACM Trans. on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30. DOI: `DOI:10.1145/272991.272995`.

[46] *ScaLAPACK – Scalable Linear Algebra PACKage.* `http://www.netlib.org/scalapack/`. 2019.

[47] Large-scale parallel numerical computing technology research team, RIKEN Center for Computational Science. *EigenExa.* `https://www.r-ccs.riken.jp/labs/lpnctrt/projects/eigenexa/`. 2020.

[48] Toshiyuki Imamura, Susumu Yamada, and Masahiko Machida. "Development of a High Performance Eigensolver on the Peta-Scale Next Generation Supercomputer System". In: *Progress in Nuclear Science and Technology, the Atomic Energy Society of Japan* 2 (2011), pp. 643–650.

[49] G. Bosilca et al. "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA". In: *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops).* Project homepage is `http://icl.utk.edu/dplasma/`. 2011.

[50] T. Auckenthaler et al. "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations". In: *Parallel Computing* 37 (2011), pp. 783–794. DOI: `doi:10.1016/j.parco.2011.05.002.`.

[51] A. Abdelfattah et al. *Roadmap for the Development of a Linear Algebra Library for Exascale Computing: SLATE: Software for Linear Algebra Targeting Exascale.* 2017.

[52] J. J. M. Cuppen. *A divide and conquer method for the symmetric tridiagonal eigenproblem.* 1980. DOI: `https://doi.org/10.1007/BF01396757`.

[53] Takeshi Fukaya, Toshiyuki Imamura, and Yusaku Yamamoto. "A case study on modeling the performance of dense matrix computation: Tridiagonalization in the EigenExa eigensolver on the K computer". In: *Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2018, pp. 1113–1122. DOI: `http://doi.org/10.1109/IPDPSW.2018.00171`.

[54] Yusuke Hirota, Daichi Mukunoki, and Toshiyuki Imamura. "Automatic Generation of Full-Set Batched BLAS". In: *Research Poster at International Supercomputing Conference (ISC '18).* `https://2018.isc-program.com/presentation/?id=post129&sess=sess113`. 2018.

[55] Jack Dongarra et al. *NLAFET, D7.6: Batched BLAS Specification.* `https://www.nlafet.eu/wp-content/uploads/2016/01/D7.6.pdf`. 2018.

[56] RIKEN Center for Computational Science Large-scale parallel numerical computing research team. *Batched BLAS.* `https://www.r-ccs.riken.jp/labs/lpnctrt/projects/batchedblas/`. 2021.

[57] E. Solomonik and J. Demmel. "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms". In: *Proc. 17th international conference on Parallel processing (Euro-Par 2011), Part II.* 2011, pp. 90–109.

[58] Van de Geijn, R.A., Watts, J. *SUMMA: Scalable Universal Matrix Multiplication Algorithm.* Tech. rep. Department of Computer Science, University of Texas at Austin, 1995.

[59] Daichi Mukunoki and Toshiyuki Imamura. "Implementation and Performance Analysis of 2.5D-PDGEMM on the K Computer". In: *Proc. Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science.* Vol. 10777. 2018, pp. 348–358.

[60] Daichi Mukunoki and Toshiyuki Imamura. "Performance Analysis of 2D-compatible 2.5D-PDGEMM on Knights Landing Cluster". In: *Proc. ICCS 2018, Lecture Notes in Computer Science.* Vol. 10862. 2018, pp. 853–858.

Chapter 6

Overview of Target Applications

In the development of supercomputer Fugaku, codesign effort with target applications plays an essential role. Each application leads its own codesign effort to address the performance bottleneck and to improve the computational performance of the application. The target applications cover different numerical schemes, space and time discretization, grid structure and data types. These different workload characteristics exhibit the bottlenecks in the system's different components. Because of the wide coverage of workload characteristics as a total, the combined codesign effect should help improve the the computational performance of the applications for not only the priority issues but also the wide range of high performance computing demand on Fugaku. The codesign effort is effectively the repeated mutual optimization between the application and the system in terms of performance, with power consumption and economy constraints.

6.1 Target applications

In the development of supercomputer Fugaku, a.k.a post K computer, codesign with target applications is regarded as the essential design approach. The target applications represent the nine social and scientific priority issues selected by the ministry. The list of the nine social and scientific priority issues is shown as the iconic representation in Figure 6.1 . Their corresponding Web pages are shown in the references [61] [62] [63] [64] [65] [66] [67] [68] [69].

Figure 6.1: social and scientific priority issues to be tackled by using supercomputer Fugaku

The target applications are developed by the priority issues organizations, and are used throughout the design and implementation phases of Fugaku. The list of target applications is shown in Table 6.1 , with the related Web sites shown in the reference section.

Table 6.1: priority issues and their target applications

no.	priority issue	organizations	target application
1	Innovative drug discovery infrastructure through functional control of biomolecular systems	RIKEN Center for Biosystems Dynamics Research and 6 other institutions	GENESIS
2	Integrated computational life science to support personalized and preventive medicine	The Institute of Medical Science, the University of Tokyo and 4 other institutions	Genomon
3	Development of integrated simulation systems for hazards and disasters induced by earthquakes and tsunamis	Institute, the University of Tokyo and 4 other institutions	GAMERA
4	Advancement of meteorological and global environmental predictions utilizing observational " Big Data "	Japan Agency for Marine-Earth Science and Technology (JAMSTEC) and other 5 institutions	NICAM +LETKF
5	Development of new fundamental technologies for high-efficiency energy creation, conversion/storage and use	Institute for Molecular Science, National Institute of Natural Sciences and 8 other institutions	NTChem
6	Accelerated development of innovative clean energy systems	School of Engineering, the University of Tokyo and 11 other institutions	ADVENTURE
7	Creation of new functional devices and high-performance materials to support next-generation industries (CDMSI)	The Institute for Solid State Physics, the University of Tokyo and 9 other institutions	RSDFT
8	Development of innovative design and production processes that lead the way for the manufacturing industry in the near future	Institute of Industrial Science, the University of Tokyo and 7 other institutions	FFB
9	Elucidation of the fundamental laws and evolution of the universe	Center for Computational Sciences, University of Tsukuba and 10 other institutions	LQCD

The target applications cover different numerical schemes, space and time discretization, grid structure and data types. Table 6.2 shows the numerical scheme/algorithm of the target applications. They have different workload characteristics and exhibit the bottlenecks in the system's different components, and cover typical applications workload as a total. Details of each target application and its codesign contents should be described in the following chapters. Table 6.3 shows the workload characteristics of the target applications. Addressing and relaxing these performance bottlenecks should improve the computational performance of the applications for not only the priority issues but also the wide range of high performance computing demand on Fugaku.

Table 6.2: target applications and their description (numerical scheme)

application	description (numerical scheme)
GENESIS	Molecular dynamics for proteins
Genomon	Genome data analysis
GAMERA	Earthquake simulation using hybrid grid FEM
NICAM+LETKF	Weather prediction using structured FVM + LETKF
NTChem	Electron correlation energy using molecular orbital method
ADVENTURE	Structure analysis FEM
RSDFT	Electronic-structure calculations using density functional theory
FFB	Thermal fluid flow using FEM
LQCD	Lattice QCD simulation using grid Monte Carlo method

Table 6.3: computational workload of the target applications

application	structured grid	unstructured grid	particle method	dense local matrix	dense distr. matrix	comm. latency	comm. B/W	neighbor comm.	global comm.	heavy I/O
GENESIS			○					○	○	
Genomon										○
GAMERA		○				○	○		○	
NICAM+LETKF	○			○			○	○		○
NTChem				○	○		○	○	○	
ADVENTURE		○		○	○	○		○	○	
RSDFT	○			○			○	○	○	
FFB		○						○	○	
LQCD	○					○	○	○	○	

6.2 Target problems for performance evaluation

In order to set the basis for performance evaluation, each target application has defined the target problem to be executed on Fugaku. The target problems are chosen by the priority issue organizations so that they represent the expected jobs to be run on Fugaku. By running the target problems on K computer, the elapsed time needed for completion is regarded as the baseline performance. Some target problems can be run on the existing K computer, while others can not be run because they exceed the capacity of K computer. If actual measurement is not possible, the estimed elapsed time on K computer is used as the baseline. Speed up objectives are defined for each target application, coupled with the target problem. The description of target problems is given in table 6.4

6.3 Codesign from applications view

Codesign is conducted at various levels and stages of Fugaku system design. The initially proposed post K system architecture is refined and completed through the course of codesign. From applications view, codesign means the repeated optimization of the application, interacting with system development. It can be captured from macroscopic and microscopic views.

Codesign in the macroscopic view is closely related with the overall system architecture. Major system design variables that affect applications design include the followings:

- system size, i.e. the number of nodes, memory capcity, inter node network design

Table 6.4: target problem description

application	parallel type	nodes /job	target problem description
GENESIS	multi job mix	1	For safe and efficient screening of the new pharmaceutical drug candidate compounds, execute 100,000 cases of all-atom molecular dynamics simultion. each case carries out 100 nano seconds of 100,000 atoms bonding including protein and solvent water.
Genomon	multi job mix	96	Whole genome anlysis for elucidating cancer. achieve 1000 samples/day throughput for 1,400,000,000 reads of 150 read length base pair.
GAMERA	single huge	158976	For representing the ground strain calculation of urban regeion, execute 120 second non-linear land earthquake analysis in 120,000 time steps, using 1,000,000,000,000 D.O.F. unstructured grid finite element model.
NICAM +LETKF	multi job mix + single huge	131072	For achieving highly precise prediction of localized torrential rain and tornados, execute (a) as multi case mix type job, 3.5 Km horizontal mesh 1024 member ensemble weather calculation with observational data assimilation every 3 hours for 2 months. (b) as single large case job, global 220 meter horizontal mesh 94 vertical layer atmosphere circulation simulation for 72 hours.
NTChem	multi job mix	20732	As a typical example of high precision ab-initio electron state calculation for elucidating the mechanism of photochemical reaction and screening the material candidates for light energy conversion, execute the energy calculation of 720 atom 19680 electron orbital carbon nanographene molecular complex. 20 cases should be executed.
ADVEN-TURE	multi job mix	4096	For achieving the structure design optimization of complex geometry, execute the structure alanysis cased on finite element method. non-linear response problem of thin plate region modeled using 1,650,000,000 second order tetrahedron solid elements. conduct 100 cases of 10,000 time step integration, applying 500 BDD iterations per time step.
RSDFT	multi job mix	10368	For elucidating the nanoscale interface composed of multiple composite materials, execute the silicon device structure optimization based on ab-initio quantum mechanics. run 24 cases of 110,000 atoms 220,000 bands 200 SCF cycle calculations.
FFB	single huge	158976	For the analysis of the fluid flow in complex geometry regions such as internal flow of water machinery and turbulent flow around the automobile which requires precise evaluation of heat generation, cooling/exhaust loss, cycle fluctuation, execute finite element method based flow analysis using 670,000,000,000 elements in 100,000 time steps.
LQCD	single huge	147456	For elucidating the history of the creation of matter over wide range of scales from elementary particles to the universe, compute the quark propagation function in 192^4 lattice field via iterative method.

- node configuration, i.e. the number of cores/CMG/CPU, SIMD supporting mechanism, type of memory, data transaction bandwidth from/to memory
- processor working frequency and power consumption

For applications, codesign in this macroscopic view means the decision of appropriate program model, sometimes requiring the source code refactoring for enabling the challenging computation at scale. Codesign in macroscopic view includes the following subjects:

- design of data distribution for massively parallel computation
- implementation of multiple level parallelism in view of performance and memory limit
- execution framework for applications parameter study

The macroscopic codesign can impact system level design decicion. The system level specification such as the number of nodes and the node configuration is finalized through the macroscopic codesign with emphasis on power consumption and cost.

On the other hand, codesign in the microscopic view is more related with the processor micro-architecture. Followings are some of the important micro-architecture related design variables.

- instruction scheduling, i.e. best mix of scalar/SIMD instructions, pipelining, out of order resource use, cache hit rate,
- cache design, i.e. cache size, cache lines, number of lines, data transaction bandwidth from/to cache hierarchy.
- memory access, i.e. prefetch control, indirect load/store implementation

Codesign in the microscopic view includes the following subjects:

- establishing the performance analysis and estimation methodology
- validating the various source code tuning patterns
- implementing the optimization feature into compilers
- optimized functionality in MPI libraries and numerical libraries

The application developpers identify the performance bottlenecks, typically using performance tools. Various performance tools were made available at each phase of development. Such tools include the performance prediction tools based on preceding systems hardware performance counters, the post K software simulators at cycle level precision, hardware emulator and Fugaku prototype test vehicle.

The bottlenecks can reside on both of hardware and software, and they are addressed from both sides. In the applications, the study for optimizing the apllications software is conducted. In the system software side, the effort is conducted to implement the corresponding automated optimization feature in the compilers. In the hardware component, the feasibility study for improving the design is conducted seeking for essential performance boost. Some microscopic codesign can apply general to many applications, while some are local to specific applications only.

From practical point of view, it is not feasible to mathematically formulate the general sensitivity in the systems design parameter space. Instead, experimenting the reasonable trials and choosing the local optimal options is the only realistic approach in many situations. The codesign is effectively the repeated mutual optimization between the application and the system in terms of performance, with power consumption and economy restriction. From systems view, the codesign in the early stage contributes to the design of system architecture such as the processor and memory specification and configuration, and in the later stage, it contributes to the design of the system software such as compilers and libraries.

The performance is evaluated in the contex of power control mode. Until later stages of codesign, the estimated system power consumption of some applications in the combination of boost enabled and eco disabled mode, i.e. the highest performance mode, exceeded the designed facility capacity. So all the application have been evaluated for the all power control modes of (boost enabled—disabled) x (eco enabled—disabled) in order to seek for the optimal mode which fits in the power consumption constraint [70]. The finally produced Fugaku processors performed better power efficiency, and allowed all the applications to run in the highest performance mode.

6.4 Design improvement request to the system

As the result of codesign, numerous design improvement and enhancement requests were raised from target applications to system development teams. There are more than 60 codesign requests that have been accepted and implemented on Fugaku. The list of such codesign requests is shown in Table 6.5.

Some examples of the reflected implementation are shown in Table 6.6.

Macroscopic codesign in the early stage contributed to finalizing the systems architecture such as processor and memory configuration, data transaction in cache registers. Microscopic codesign in later stage mostly contributed to improving the hardware control parameters such as hardware prefetch distance, and the system software improvement such as the compiler optimization feature, numerical library, MPI/network library

Table 6.5: codesign request from the target applications to the system development

application	codesign request
GENESIS	performance improvement of communication inside the node for distributed FFT computation
GENESIS	development of high performance sequential FFT for 2/3/5/7/mixed basis
GENESIS	wait time reduction of floating point arithmetic operations
GENESIS	high performance bit wise operations in Fortran programs
GENESIS	Optimization enhancement for loops using array pointers
Genomon	Achieve job scheduling that effectively fills processes and reduces free computing cores
Genomon	fast character string comparison using SIMD
Genomon	performance improvements of zlib and other system libraries
Genomon	maintain Python programming environment
Genomon	file I/O consideration for realizing effective Genomon2 workflow
Genomon	Eliminating delays in reading files with I/O lengths that exceed the page cache size
Genomon	Providing the feature equivalent to Grid Engine like batch job management software
GAMERA	compiler feature of loop splitting and scheduler optimization

codesign request from the target applications to the system development (continued)

application	codesign request
NICAM +LETKF	Improvement of L1 cache busy rate by suppressing the excessive indirect load instructions
common	Reduction of busy cycles in L1 cache miss
common	Optimized HW / SW prefetch coordination
NICAM +LETKF	fast single-precision exponentiation arithmetic
NICAM +LETKF	supporting the store operation that bypasses the cache
NICAM +LETKF	Optimal implementation selection and appropriate messages for OpenMP collapse with/without innermost loop
NICAM +LETKF	Eliminating hardware prefetch lost
NICAM +LETKF	preparing the compiler environment on FX100 for performance evaluation with hardware resource limit settings equivalent to post-K .
NICAM +LETKF	Evaluation of the advanced performance profiler overhead .
NTChem	Evaluation and implementation of high-performance DGEMM library
NTChem	Evaluation and implementation of high-performance SGEMM library
ADVEN-TURE	Faster short message MPI_Allgather when there are multiple processes in a node
RSDFT	Improvement of MPI_Allreduce communication algorithm for medium message size of 1KB 1MB.
RSDFT	Improving intra-node MPI_Allreduce communication for long messages
RSDFT	improving pipelined process in intra-node MPI_Reduce communication
RSDFT	implementation of intra-node collective communication by multiple representative processes
RSDFT	performance enhancement of single process DGEMM using multiple CMGs
RSDFT	Accelerate MPI_Bcast using memcpy between CMGs
RSDFT	improving the throughput of MPI_Allreduce communication by using the representative processes method and by reducing the number of memory copies
RSDFT	Applying Tofu-specific algorithm for Allreduce/Reduce communication using MPI_IN_PLACE
RSDFT	Resolving known issues with the EigenExa library
RSDFT	feature to call sequential DGEMMs from PARALLEL regions in EigenExa library
FFB	Performance improvement by issuing indirect SIMD load instructions for indirect array references
FFB	feature to support effective SIMD operations when the loop length does not match a multiple of 8 double-precision elements or 16 single-precision elements.
FFB	feature to perform rerolling (processing to restore to the original loop) for a loop that has been manually unrolled

codesign request from the target applications to the system development (continued)

application	codesign request
LQCD	capability to return the values via registers when inlining a function that returns an array of size equal to the SIMD width
LQCD	Enhancement of Tree Height reduction optimization
LQCD	Removal of temporary arrays used across multiple loops
LQCD	Reduction of integer register usage by utilizing the SIMD addressing mode
LQCD	OS development that does not degrade application performance due to OS jitter
LQCD	Providing RDMA communication mechanism
LQCD	Reduced issue time for multiple RDMA communications
LQCD	Countermeasures for the problem that communication time is significantly increased due to the influence of file I/O by other jobs
LQCD	Fast implementation of short (1-3 elements) MPI_Allreduce
common	Reduced latency for intra-node communication
common	configuring the assistant cores
common	Eliminating the decrease in L1 cache throughput when there is a cache index conflict
common	Load reduction for the access requests to L1 cache when executing indirect SIMD load instructions
common	performance enhancement of MPI_Scatter
common	performance enhancement of built-in functions
RSDFT	compiler optimization enhancement for C language functions whose arguments are pointer type
LQCD	development of GEMM variation library for complex data that takes the array argument of real part values and imaginary part values in sequence
GENESIS	error detection feature for OpenMP private directive when a pointer type is specified
common	SFI detection feature in profiler
Genomon	Providing an API to get Tofu coordinates
ADVENTURE	Combined use of sequential BLAS and thread parallel BLAS
NTChem	efficient SIMD implementation for loops in which single-precision and double-precision operations are mixed
GENESIS	New interface design for accelerated distributed parallel FFT
LQCD	Improved intra-node communication performance for distributed FFT computation
LQCD	FFT batch mode support
NICAM +LETKF	add a simple compiler option that does not affect the calculation results still providing performance
common	Improve the profiling feature to detect and report the SFI condition in CPU performance analysis report with smaller number of repetitions
common	Simplification of prefetch instruction line
common	Expansion of the number of elements that can use the Structure Load instructions

Table 6.6: examples of the reflected implementation into the system

application	codesign request	reflected implementation
common	separate services from computation	add and integrate assistant cores into CPU so that compute cores can be dedicated to computing
common	load reduction of the access requests in L1 cache when executing indirect SIMD load instructions	furnish combined gather mechanism to reduce L1 pressure
GAMERA	automatic and optimized loop fission	enhance compiler to automatically execute fission optimization when the compiler judges as effective
LQCD	OS development that does not degrade application performance due to OS jitter	develop the noiseless operating system McKernel and integrate into the system as user choosable option per job

The effect of codesign is almost obvious, impacting the value of both the target applications and Fugaku. Target applications become production ready from day 1 on Fugaku with optimized performance. Fugaku system becomes throughly optimized to typical HPC workload demand represented by target applications.

References

[61] *Priority Issue 1 : Innovative drug discovery infrastructure through functional control of biomolecular systems.* `http://www.jamstec.go.jp/pi4/en/`.

[62] *Priority Issue 2 : Integrated computational life science to support personalized and preventive medicine.* `http://en.postk.hgc.jp/home`.

[63] *Priority Issue 3 : Development of integrated simulation systems for hazards and disasters induced by earthquakes and tsunamis.* `http://www.eri.u-tokyo.ac.jp/LsETD/Post_K/bosaitop/eng/index.html`.

[64] *Priority Issue 4 : Advancement of meteorological and global environmental predictions utilizing observational "Big Data".* `http://www.jamstec.go.jp/pi4/en/`.

[65] *Priority Issue 5 : Development of new fundamental technologies for high-efficiency energy creation, conversion/storage and use.* `http://ft-energy.ims.ac.jp/ft-energy/english/index.html`.

[66] *Priority Issue 6 : Accelerated development of innovative clean energy systems.* `https://postk6.t.u-tokyo.ac.jp/en/`.

[67] *Priority Issue 7 : Creation of new functional devices and high-performance materials to support next-generation industries CDMSI.* `https://cdmsi.issp.u-tokyo.ac.jp/` (Japanese language site).

[68] *Priority Issue 8 : Development of innovative design and production processes that lead the way for the manufacturing industry in the near future.* `http://www.postk-pi8.iis.u-tokyo.ac.jp/` (Japanese language site).

[69] *Priority Issue 9 : Elucidation of the fundamental laws and evolution of the universe.* `http://www.jicfus.jp/postk9/en/`.

[70] Kazunori Mikami, Hirofumi Tomita, Kazuo Minami. *Codesign Supercomputer Fugaku and Target Applications through Performance Optimization.* `https://2020.isc-program.com/presentation/?id=post116&sess=sess325`, `https://www.youtube.com/watch?v=dThZ2vab9yQ`.

Chapter 7

Innovative computing infrastructure for drug discovery by GENESIS

7.1 Overview of GENESIS

7.1.1 Molecular Dynamics on Biological Systems

Molecular Dynamics (MD) is a method of tracking movements of atoms and molecules using numerical integration of Newton's equation of motion. In MD, coordinates and velocities of these particles are updated with a small-time step based on force evaluations. By the repeating the update procedure in many steps, the coordinates or velocities are used in investigating dynamics and statistics of the atoms and molecules. The method is widely used in chemical physics, biophysics, material science, and so on.

In the biophysics field, MD simulation is widely used to understand structures, dynamics, and functions of bio-molecules. The method can also predict binding and affinities for protein-drug complexes from free energy profiles on biochemical reactions. In the reactions, many bio-molecules and solvents are involved in the reactions and the reaction time scale order is usually greater than milliseconds. Therefore, the amount of calculation is enormous. From the first study in 1970's [71], various MD programs and algorithms have been developed to overcome computational limits of system size and simulation time. Improvements of architectures have extended to the temporal and spatial range of MD simulations. Special purpose systems for MD like MDGRAPE [72, 73] and Anton/Anton2 [74, 75] have made it possible to simulate 10-100 times faster than the computers of the time. Even without using these specialized machines, the simulation speed has been increased by using Graphics Processing Unit (GPU). Many MD software packages like GENESIS[76], AMBER[77], Gromacs[78], NAMD[79], and OpenMM[80] are making use of GPU. It is very effective for systems up to about 1 million atoms.

It is also important to expand the spatial range of simulations. The biological functions work in large biological systems such as cellular environments with proteins, nucleic acids, and metabolites. These calculations have been achieved by using massively parallel computing on supercomputers.[81, 82, 76]

However, even the specialized machines, the simulation range is on the order of milliseconds for a small protein in solution. It is quite difficult to investigate processes on biological functions. The enhanced conformational sampling algorithms such as replica exchange molecular dynamics (REMD).[83, 84] are powerful tools to solve the problem.

The method simulates multiple copies of the target system (we call the copy "replica") with different physical parameters at the same time. By exchanging the physical parameters at certain timings, the simulation allows to sample a larger conformational space. The advantage of the method is that communication between replicas is negligible, and it can be considered almost as a collection of stand-alone simulations.

7.1.2 Energy and force description in MD

In general, biological MD simulations use empirical functions called ‘force field’ to calculate the force on each particle in the simulation system. Bonded interactions involve covalent bonds between atoms, and non-bonded interactions consist of van der Waals and electrostatic interactions. For the bonded interactions, the computational cost is proportional to the number of particles (N). The non-bonded interactions should be applied to all pairs separated by three or more covalent bonds. The non-bonded interaction energy is generally described as

$$E_{\text{non-bonded}} = \frac{1}{2}\sum_{i,j=1}^{N^*}\left(\frac{a_{ij}}{r_{ij}^{12}} + \frac{b_{ij}}{r_{ij}^{6}} + \frac{q_i q_j}{r_{ij}}\right) \tag{7.1}$$

where $*$ means that we exclude bonded interactions. The computational cost of non-bonded interaction based on the above equation is $O(N^2)$. The first and second terms, showing repulsive and dispersive interactions, decay rapidly as we increase the pairwise distance, and can be evaluated with the upper threshold, named “cutoff” r_c:

$$E_{\text{vdw}} = \frac{1}{2}\sum_{i,j=1,r_{ij}<r_c}^{N^*}\left(\frac{a_{ij}}{r_{ij}^{12}} + \frac{b_{ij}}{r_{ij}^{6}}\right) \tag{7.2}$$

The electrostatic interaction has much longer interaction range than van der Waals terms, so we could not assign the same cutoff value. To reduce the computational cost in the electrostatic interaction, we usually decompose the electrostatic interaction into real- and reciprocal-space interactions. The real-space interactions are truncated at the same cutoff values as van der Waals by making use of complementary error functions. Then the real-space interaction energy becomes

$$E_{\text{real}} = \frac{1}{2}\sum_{i,j=1,r_{ij}<r_c}^{N^*}\left(\frac{a_{ij}}{r_{ij}^{12}} + \frac{b_{ij}}{r_{ij}^{6}} + \frac{\text{erfc}\left(\alpha r_{ij}\right)}{r_{ij}}\right) \tag{7.3}$$

The reciprocal-space interaction energy only includes the electrostatic ones and is are expressed as

$$E_{\text{reciprocal}} = \frac{1}{2\pi V}\sum_{\vec{m}\notin\vec{0}}\frac{e^{-\alpha^2\pi^2\vec{m}^2}}{\vec{m}^2}S(\vec{m})S(-\vec{m}) \tag{7.4}$$

By assuming (K_1, K_2, K_3) grids, $S(\vec{m})$ is

$$S(\vec{m}) = \sum_{j=1}^{N} q_j e^{2\pi i\vec{m}\cdot\vec{r}_j} \tag{7.5}$$

$$= \sum_{k_1}\sum_{k_2}\sum_{k_3} Q\left(k_1, k_2, k_3\right) e^{2\pi i\left(\frac{m_1 k_1}{K_1}+\frac{m_2 k_2}{K_2}+\frac{m_3 k_3}{K_3}\right)} \tag{7.6}$$

In Eq. (7.6), the reciprocal-space interaction requires the approximation of $Q\left(k_1, k_2, k_3\right)$ and fast Fourier transform (FFT). There are various ways to approximate $Q\left(k_1, k_2, k_3\right)$. Among them, one of the most popular schemes is the smooth particle mesh Ewald (SPME) method [85], which uses B-spline interpolations. The computational cost of $Q(k_1, k_2, k_3)$ and FFT is $O(N)$ and $O(N\log N)$, respectively.

In actual MD simulations, the real-space interactions are calculated based on a neighbor lists with a few angstroms larger than r_c. The list, called pair-list, is updated at a certain timing during MD simulation to reduce the computational cost.

7.1.3 Characteristics of a MD software package, GENESIS

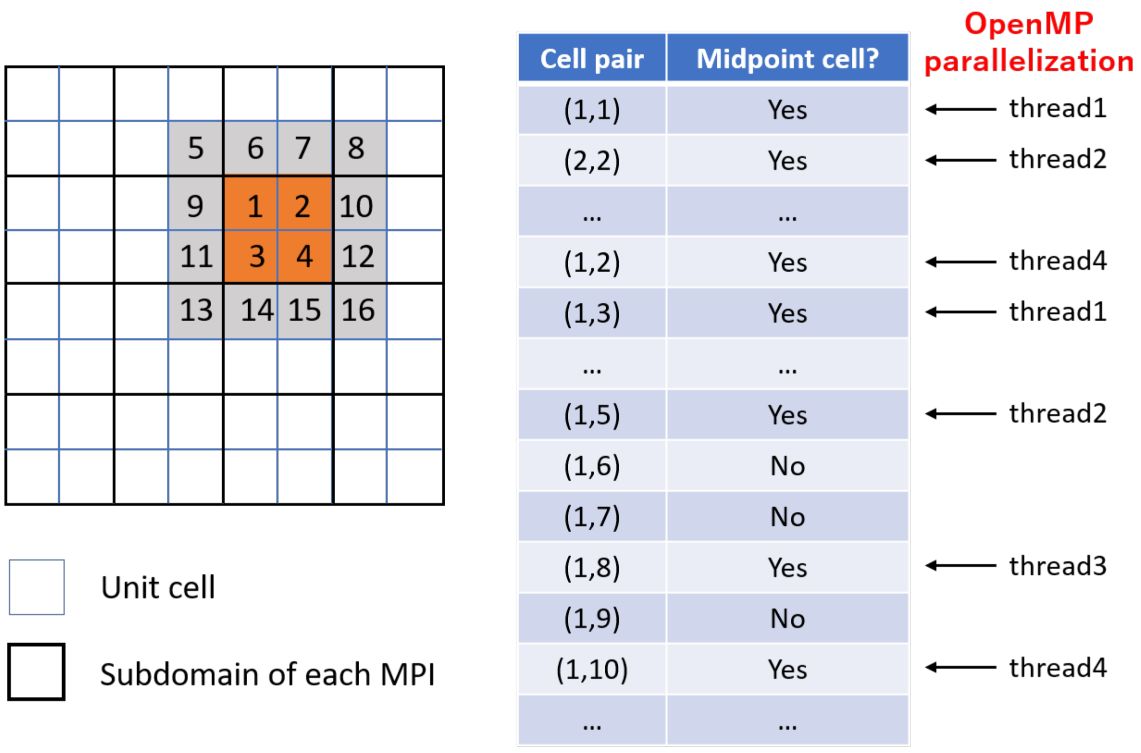
GENESIS (GENeralized-Ensemble Simulation System) [86, 87] is a MD software package developed by RIKEN and it is released under the LGPLv3 license. The package consists of two MD programs (SPDYN and ATDYN) and trajectory analysis tools. The main difference between two MD programs are parallelization (decomposition) schemes. ATDYN is parallelized with the atomic decomposition scheme. The program allows for a variety of models including quantum mechanics/molecular mechanics and coarse-grained methods as well as all-atom force fields such as CHARMM and AMBER. SPDYN is parallelized with the domain decomposition scheme as described below. The program is designed to achieve high-performance simulations with efficient parallelization while the program can use the all-atom force fields. Both ATDYN and SPDYN allow us to perform various kinds of enhanced-sampling schemes using replicas, including REMD, generalized replica exchange with solute tempering (gREST) [88, 89, 90], string method [91], and so on. From now on, we will discuss SPDYN only.

The program has its own parallelization schemes and speed-up algorithms. It has achieved high performances and parallel efficiencies on supercomputers such as K computer, TSUBAME, and KNL supercomputers. [76, 81, 92]

Taking advantage of these high performances, we have succeeded simulations of systems consisting of 100 million–1 billion atoms systems on K computer in RIKEN [81] and Trinity Phase 2 platform in Los Aramos National Laboratory. [82]

In SPDYN, the simulation system is divided into subdomains from number of MPI processes, and each subdomain is further divided into smaller cells. The forces on the particles are calculated on this cell-by-cell basis, and the interaction between the atoms in different cells are computed by the midpoint cell scheme. [93] Based on the domain decomposition scheme, each MPI process performs the calculation of forces and time evolution of positions for atoms in its own cells. Fig. 7.1 is an example of two-dimensional case with 16 MPIs. The system is divided into 16 subdomains. Each subdomain is again divided into 4 unit-cells. Each MPI has a data of corresponding subdomain (colored orange) and the buffer region consisting of adjacent cells of the subdomain (colored gray). For energy and force evaluations, the program first makes all cell pairs from the subdomain and buffer regions and checks if the midpoint cell of each cell pair is in the subdomain or not. If it is in the subdomain, it assigns the cell pairs in the corresponding MPI process. With this scheme, we can assign the same subdomain structure between real- and reciprocal-space, and can avoid communications in the evaluation of $Q(k_1, k_2, k_3)$ and its derivates. Based on the scheme, the main computational bottleneck is the real-space non-bonded interactions and the bottleneck turns into reciprocal-space interactions as we increase the number of processes and system sizes.

In this report, we will mainly discuss our various efforts (including those that failed unfortunately) in this project. The optimization schemes of GENESIS has been discussed elsewhere. [94] The recent version of GENESIS optimized on Fugaku has been released in https://github.com/genesis-release-r-ccs/genesis-2.0.



Cell pair	Midpoint cell?	OpenMP parallelization
(1,1)	Yes	thread1
(2,2)	Yes	thread2
…	…	
(1,2)	Yes	thread4
(1,3)	Yes	thread1
…	…	
(1,5)	Yes	thread2
(1,6)	No	
(1,7)	No	
(1,8)	Yes	thread3
(1,9)	No	
(1,10)	Yes	thread4
…	…	

Figure 7.1: Hybrid (MPI+OpenMP) parallelization scheme in GENESIS

7.2 Codesign procedures for GENESIS

7.2.1 Target system and performance on K computer

In the project, we've targeted MD simulations of a middle-sized system for drug discovery calculations. We selected a target system with 92,224 atoms. We consider the target problem to be capacity computing, which includes large number of computations for various drug candidates. In the actual calculations, there may be some variation in the performance due to difference of number of atoms. However, in this case, we got a computation time of a single simulation and calculated a ratio of time for two systems. The performance ratio of K and Fugaku was estimated by multiplying the scale ratio of number of nodes.

In the evaluation, we calculated 4 ps simulation and got elapse time of main loop of MD simulation. The cost of initialization such as reading files and setting variables is a constant value regardless of the number of steps and was not considered in this evaluation. We first get a computation time on 16 nodes of K computer. The numbers of MPI processes and OpenMP threads were 16 and 8, respectively. We got 115.2 seconds for the 4 ps simulation and the computation time is used as the baseline.

7.2.2 Finding bottleneck

From the evaluation, more than 50 % of the total elapse time was given from the calculation of forces from real-space non-bonded interactions. In addition, generation of pair-list costed more than 10 % although it was executed only every 10 steps. The cost of the reciprocal-space interactions was around 20 %. We selected target modules from the evaluation and have improved the performance one by one.

7.2.3 Real-space non-bonded Interaction

The main consideration to improve the performance in real-space non-bonded interaction was the efficient application of single instruction, multiple data (SIMD) and software pipelining in the source code dealing with that part. In the codesign procedure, the two main factors that prevents high performance in real-space interactions are list access with pair-list usage and lookup tables used in energy/force evaluations.

With pair-list usage, the program stores indices of atoms, which is retrieved by list access. The list access caused long waiting time of operation and load caches. To avoid the problem, we considered the scheme without using pair-list. However, it increases the operation amount more than six times and was not adopted although SIMD can be applied more easily.

The usage of lookup table was another source preventing high performance. The real-space interaction requires the evaluations of inverse square roots, switch function, and complementary error functions. By making use of lookup table instead of direct calculation of these, we can save the computational time, and almost MD programs adopt it. GENESIS also uses a lookup table with a modification from the conventionally used one. [95] Despite the lookup table reduces the computational time, it requires random array access, and increases access latency. In the early stage of the codesign, we tried an algorithm to reduce access latency by eliminating lookup table, but finally found that keeping the lookup table is better considering the overall performance. During the codesign procedure, we finally found a way to improve the performance by changing the algorithm.

7.2.4 Long-range interactions

In GENESIS MD software, reciprocal-space electrostatic interaction consists for the following five steps:

- step1: charge grid data before forward FFT
- step2: forward FFT calculation
- step3: energy and virial calculation
- step4: backward FFT calculation
- step5: force evaluation

In these five steps, codesign observed the performance in two categories: computational performance of step 1 and 5 and communicational costs in step 2 and 4. In particular, in step 2 and 4, we tested the communicational costs of various types of MPI_alltoall communications.

7.2.5 Updates of integrator for large time steps

In early stage, we evaluated the performance under constant particle number, volume, and energy (NVE) conditions without temperature and pressure control. In the past in the biophysical field, the NVE condition was widely used in the evaluation, but in the actual drug discovery calculations, NPT (constant particle number, pressure, and temperature) condition is more realistic. Therefore, researches in the priority issue 1 requested us to evaluate under NPT conditions corresponding to the actual drug candidate calculation. In response to this request, the GENESIS development team developed a more accurate numerical integration scheme under NPT conditions. [96]

7.2.6 Requests to hardware and system (MPI, libraries) working groups

Through this project, we have made the following requests to other working groups; 1) improvement of compiler that causes long waiting time for floating-point operations, 2) improvement of compiler for efficient software pipelining, 3) adoption of a high-speed method of intra-node communication, 4) optimized FFT library with small number of grids on Fugaku. The first three have already been implemented and shown to be effective. The last part is under development and the performance improvement in the development version was confirmed.

7.3 Optimization of GENESIS

7.3.1 Real-space non-bonded interaction

One of the features we found is that there are non-negligible operand and cache wait whenever we confront the most inner do loop. Here, wait time means the number of cycles (unit intervals) in which all instructions are not completed. The wait time depends on the frequency of the most inner loop and the most inner loop-length. Even the same numbers of instructions, we found that we can reduce the wait time by increasing the most inner loop in the case of Fugaku. The GENESIS program originally calculates these interactions based on the cell pairs. Let us assume the two cell indices are i and j, respectively, for a given cell pair. Let us also assume that the atom indices in i-th and j-th cells are ia and ja. Then, GENESIS calculates the non-bonded interactions by making do loops of ja for each given ia. The loop-length is given by the number of atoms in j-th cells within pair-list cutoff distance from ia-th atom. For the optimized performance on Fugaku, we accumulate the lists of ja obtained in different cell pairs for a given ia. For example, in Fig. 7.2, we accumulate 6 different j_a lists for a i_a-th atom in cell 1 to make a longer loop-lengths in the most inner do loops. By doing this way, we could increase the average loop-lengths by a factor of 10 for the target system (Fig. 7.3. The detailed explanation with pseudo code is shown in [94].

In addition to the new algorithm increasing the most inner do-loop length, we further optimized by applying the "CONTIGUOUS" attribute to the arrays and by applying L1 cache prefetch for related arrays. Like the previous section, the detailed explanation is shown in [94]

7.3.2 Optimization of reciprocal-space interaction

In step1 in 7.2.4, charge grid data, $Q\left(k_1, k_2, k_3\right)$ in Eq. (7.6) is obtained by B-spline interpolation. In step5, we need to evaluate the derivate of $Q\left(k_1, k_2, k_3\right)$ with respect to the atomic coordinates. If the B-spline order is n, the computational cost of these becomes $O\left(Nn^3\right)$, and for small number of processes, this could be the main bottleneck in reciprocal-space interactions. In previous version of GENESIS, each MPI process obtain a part of $Q\left(k_1, k_2, k_3\right)$ from the subdomain and its buffer regions (Fig. 7.4). In this algorithm, there are two problems. First, each MPI sometimes generates unnecessary charge data that are not in the corresponding subdomain, which increases the overall operation amount. Second, to minimize the operation by discarding unnecessary calculation, the "conditional" statement is used, which again prevents SIMD. However, it has an advantage that we do not need any communication before starting forward FFT and after finishing backward FFT.

To circumvent the performance loss in the previous FFT scheme, we suggest a new algorithm. Buffer region of each subdomain is not considered in charge grid data generation. We can increase the performance by reducing operations and by excluding conditional statements. The generated charge grid data then communicated to the neighboring processes to complete the data in each subdomain. This has the weakness that it requires communications. In the updated version,

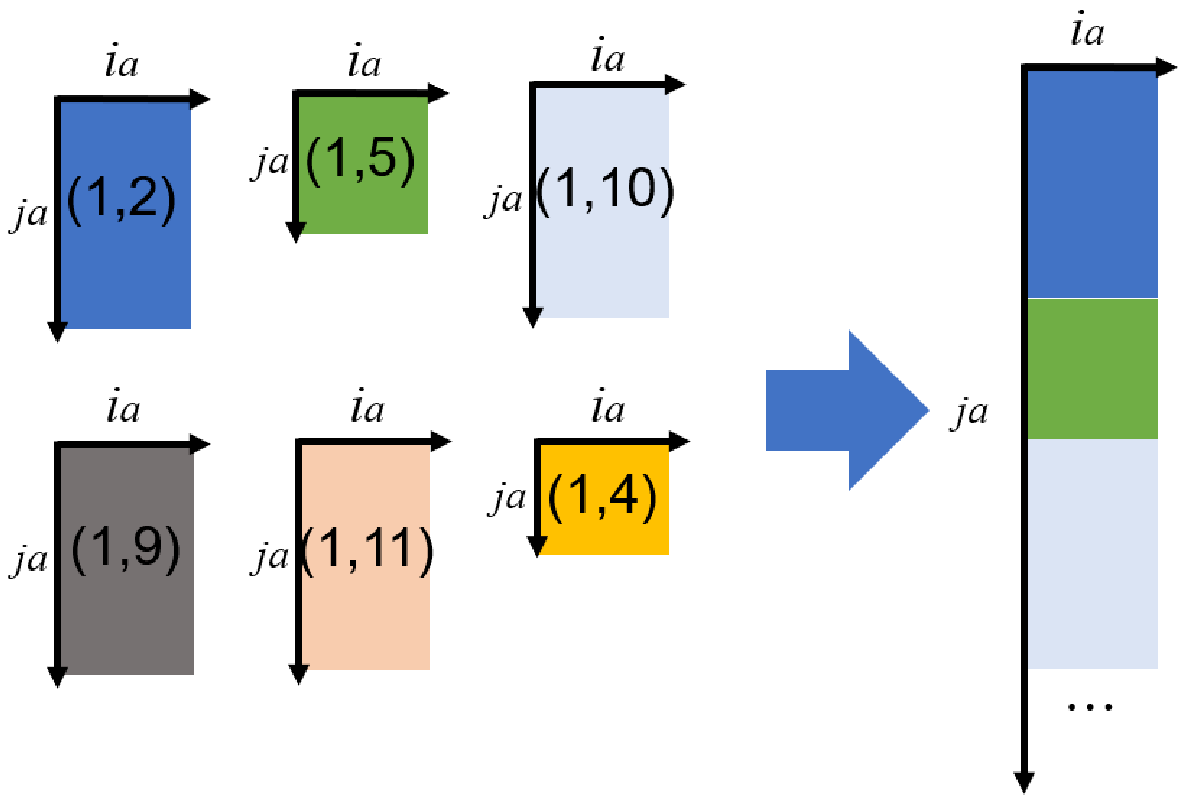


Figure 7.2: The original non-bonded schemes in GENESIS (left) and its modification for Fugaku. In the left figure, we assume that particles in cell 1 have interactions with particles in cells 2, 4, 5, 9, 10, and 11. Based on this, we generate cell pair-lists (1,2), (1,4), (1,5), (1,9), (1,10), and (1,11). For each cell pair (1, j), we consider the do loop of j_a-th particle in cell j for a given i_a-th particle in cell 1. In the updated version for Fugaku, we accumulate all the j_a lists interacting i_a-th particle, and make the do loop calculation for the accumulated lists of j_a.

we make a routine to compare these schemes before starting MD simulations to obtain the best performance condition. More detailed explanation is described in [94].

7.3.3 Minimizing communication from intra- and inter-node communication patterns

On Fugaku, we optimized the performance by minimizing communication costs. Generally, communication on intra-node is faster than that on inter-node, and we need to optimize the performance by considering the intra- and inter-node communicational costs. With P_x, P_y, and P_z subdomains in x, y, and z dimension respectively, the total number of processes, P is same as the total number of subdomains, i.e., $P = P_x \times P_y \times P_z$. By defining the MPI rank in each dimension, R_x, R_y, and R_z, we can obtain the relationship between the MPI rank and those in each dimension using

$$R = R_x + R_y \times P_x + R_z \times P_x \times P_y \tag{7.7}$$

In GENESIS, there are three time-consuming communications in FFT:

- Alltoall_x: Alltoall communication between processes having the same R_y and R_z.

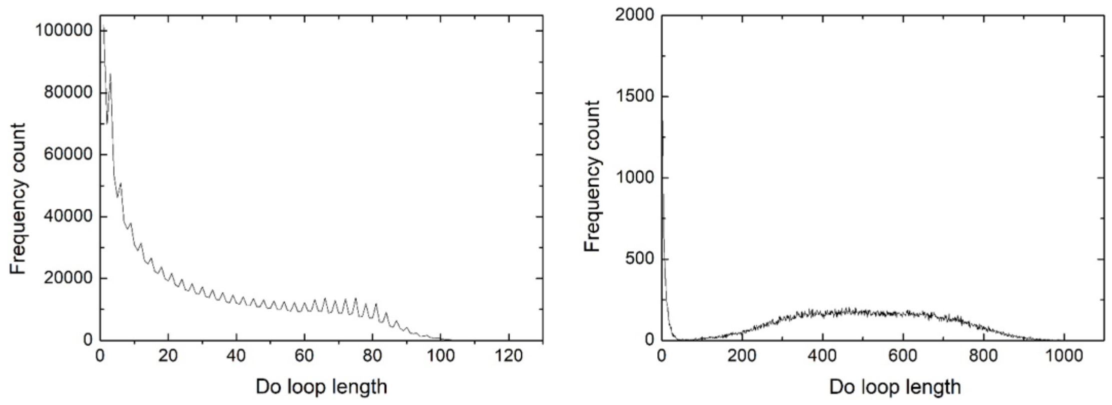


Figure 7.3: Do loop length distribution using the original (left) and new (right) schemes. In the original scheme, the do loop length of the most inner loop is less than 100. On the other hand, in the modified one, the longest do loop length becomes larger than 800.

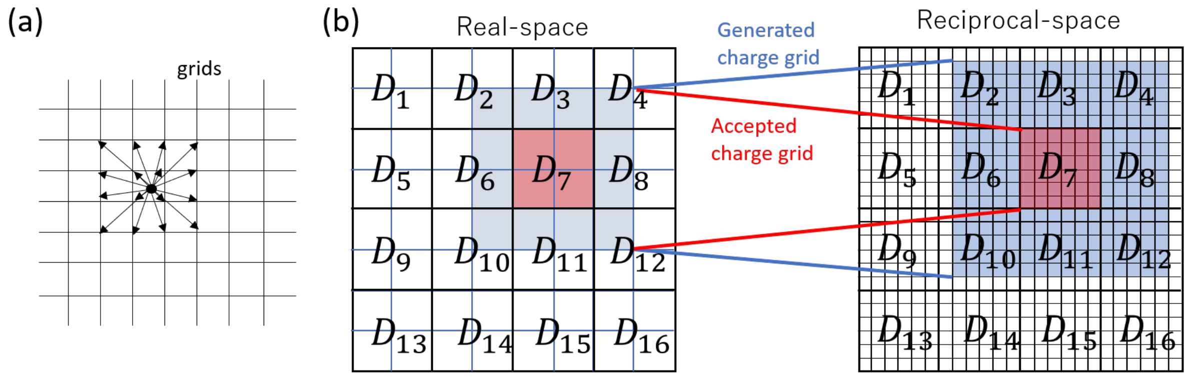


Figure 7.4: (a) Charge data on grids from a charge in a real-space. The amount of effect on grid points depends on the spline order. (b) Charge data generation on grids with the original scheme in GENESIS. The MPI process corresponding to the subdomain D_7 and its buffer generates grids points in the region colored in blue. From them, we only select the data in the subdomain region colored in red.

- Alltoall_y: Alltoall communication between processes having the same R_x and R_z.
- Alltoall_z: Alltoall communication between processes having the same R_x and R_y.
- Alltoall_xy: Alltoall communication between processes having the same R_z.

Generally, given the same amount of process and communication data, Alltoall_x is less time-consuming than others. Therefore, we considered FFT with more frequent Alltoall_x than Alltoall_z.

7.3.4 Optimization of bonded interactions

Bonded interaction is less time-consuming than non-bonded interactions, but we could slightly increase the performance by optimizing this part, too. First, "no simd" directive is assigned for do loops with short loop-length. Without the directive, it takes additional computational time for scheduling optimization in each do loop. If the loop-length is short, the time for scheduling can be larger than that for the computation of the loop itself. Second, we increased OpenMP threads balances in each do loop by dynamic scheduling of the do loops.

7.3.5 Optimization of the performance by an integration scheme with a large time step

To maximize the performance, we have developed MD integration schemes enabling a larger time step. First, we have developed ways to evaluate temperature and pressure in more accurate way during MD simulations. Second, we have designed a group-approach for temperature and pressure evaluations. This way accelerates MD integration speed by removing iterations required in pressure and temperature evaluations. These developments allowed us to perform MD simulations with 3.5 fs time step for real-space and 7.0 fs time step for reciprocal-space forces by combining the accurate temperature/pressure evaluates with hydrogen mass repartitioning (HMR) schemes. [97]. Even without HMR, the developments allowed stable MD simulations with 2.5 to 3.0 fs time step for real-space and 5.0 to 6.0 fs time step for reciprocal-space forces.

7.3.6 Performance of real-space non-bonded interaction

The improvement from the real-space optimization has been discussed by comparing it with a conventional scheme without these updates for 1.58 million atoms in the previous paper. [94] In the paper, we showed that the new scheme improves the performances of both force calculation and generation of pair-list more than twice of the conventional one. The reason is reduction of the waiting times before starting floating-point operations and accesses to the L1D cache by increasing the most inner do loop length.

In this section, we show the improvements by applying the "CONTIGUOUS" attribute and L1 cache prefetch using the target system. We used a single node of Fugaku with 2.0 GHz CPU clock and assigned 16 MPI processes with 3 threads. The performance is given by 0-th core memory group (CMG). We performed 4 ps MD simulation by assigned 1600 steps and 2.5 fs time step. In the force calculation, the calculation no "CONTIGUOUS" attribute, no prefetch, and SoA coordinate and force arrays reduces the performance by low memory throughput rate (Table 7.1). In particular, it is found that the "CONTIGUOUS" attribute makes an important role in performance. If we see it in more detail, we measured the floating-point cache access wait time and floating-point operation wait time in Table 7.2 . Low performance from no "CONTIGUOUS" attribute is mainly from large operation wait time. Performance reduction from no prefetch and SoA arrays are mainly due to L2 cache access wait time.

7.3.7 Performances on Fugaku

We evaluated computer times of 4 ps MD simulation on a single node of Fugaku. We also compared the times using different clocks of CPU and "eco" modes. The version of development environment on Fugaku is lang/tcds-1.2.26b. These performances are shown in Table 7.3. Finally, we got 131.1 times speed-up of K computer on 2.2GHz clock CPU on Fugaku. We note that the version is optimized to use on three OMP threads on a single node and the performance would be decreased for more than 6 OMP threads. Therefore, the released version has been updated for use of large numbers of OMP threads.

Table 7.1: Performance statistic summary

Schemes	GFlops	MT[a] (GB/s)	SIMD (%)
Optimized	57.14	11.92	85.09
No contiguous	39.41	11.67	75.39
No prefetch	44.30	7.87	88.14
SoA array	48.45	11.32	85.39

[a] Memory throughput

Table 7.2: Floating-point cache access wait and operation wait

Schemes	WT_L1D[a]	WT_L2D[b]	Operation[c]
Optimized	6.67	1.40	1.72
No contiguous	5.60	4.50	1.55
No prefetch	5.53	1.43	7.37
SoA array	6.58	3.57	1.85

[a] Floating-point and integer load L1D cache access wait
[b] Floating-point load L2 cache access wait
[c] Floating-point operation wait

Table 7.3: Performance on K and Fugaku

System	K	Fugaku			
		w/o eco		w eco	
Mode	Base	2.2GHz	2.0GHz	2.2GHz	2.0GHz
Computation time (sec)	115.2	26.9	29.6	28.1	30.6
Ratio of performance (a job)	1.0	4.3	3.9	4.1	3.8
Number of nodes (a job)	16	1	1	1	1
Number of nodes (system)	82944	158976	158976	158976	158976
Ratio of performance (system)	1.0	131.1	119.4	125.6	115.4

7.4 Summary

In this project, we have developed new algorithms and MD integration scheme to optimize GENESIS on Fugaku. Since GENESIS is able to operate with fewer nodes (even a single node), it was able to make many trials possible on trial machine and Fugaku. Therefore, we can increase the performances of most modules, not just the bottleneck modules. Although the target system is less than 100,000 atoms, these updates also speed up the simulations in more realistic biological system on Fugaku. [94] Finally, we would like to thank all the people who were involved in the GENESIS working group of the project.

References

[71] J. A. McCammon, B. R. Gelin, and M. Karplus. "Dynamics of folded proteins". In: *Nature* 267.5612 (1977), pp. 585–590. ISSN: 1476-4687. DOI: `10.1038/267585a0`.

[72] T. Narumi et al. "A 55 TFLOPS Simulation of Amyloid-Forming Peptides from Yeast Prion Sup35 with the Special-Purpose Computer System MDGRAPE-3". In: SC '06 (2006), 49–es. DOI: `10.1145/1188455.1188506`.

[73] I. Ohmura et al. "MDGRAPE-4: a special-purpose computer system for molecular dynamics simulations". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2021 (2014), p. 20130387. DOI: `10.1098/rsta.2013.0387`.

[74] D. E. Shaw et al. "Millisecond-Scale Molecular Dynamics Simulations on Anton". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: Association for Computing Machinery, 2009. ISBN: 9781605587448. DOI: `10.1145/1654059.1654126`.

[75] D. E. Shaw et al. "Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer". In: SC '14 (2014), pp. 41–53. DOI: `10.1109/SC.2014.9`.

[76] J. Jung et al. "Graphics Processing Unit Acceleration and Parallelization of GENESIS for Large-Scale Molecular Dynamics Simulations". In: *Journal of Chemical Theory and Computation* 12.10 (2016), pp. 4947–4958. ISSN: 1549-9618. DOI: `10.1021/acs.jctc.6b00241`.

[77] R. Salomon-Ferrer et al. "Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald". In: *Journal of Chemical Theory and Computation* 9.9 (2013), pp. 3878–3888. ISSN: 1549-9618. DOI: `10.1021/ct400314y`.

[78] C. Kutzner et al. "More bang for your buck: Improved use of GPU nodes for GROMACS 2018". In: *Journal of Computational Chemistry* 40.27 (2019), pp. 2418–2431. DOI: `https://doi.org/10.1002/jcc.26011`.

[79] J. C. Phillips et al. "Scalable molecular dynamics on CPU and GPU architectures with NAMD". In: *The Journal of Chemical Physics* 153.4 (2020), p. 044130. DOI: `10.1063/5.0014475`.

[80] P. Eastman et al. "OpenMM 7: Rapid development of high performance algorithms for molecular dynamics". In: *PLOS Computational Biology* 13.7 (July 2017), pp. 1–17. DOI: `10.1371/journal.pcbi.1005659`.

[81] I. Yu et al. “Biomolecular interactions modulate macromolecular structure and dynamics in atomistic model of a bacterial cytoplasm”. In: *Elife* 5 (2016), e19274. ISSN: 2050-084X. DOI: `ARTNe1927410.7554/eLife.19274`.

[82] J. Jung et al. “Scaling molecular dynamics beyond 100,000 processor cores for large-scale biophysical simulations”. In: *Journal of Computational Chemistry* 40.21 (2019), pp. 1919–1930. ISSN: 0192-8651. DOI: `10.1002/jcc.25840`.

[83] Y. Sugita and Y. Okamoto. “Replica-exchange molecular dynamics method for protein folding”. In: *Chemical Physics Letters* 314.1 (1999), pp. 141–151. ISSN: 0009-2614. DOI: `https://doi.org/10.1016/S0009-2614(99)01123-9`.

[84] Y. Sugita, A. Kitao, and Y. Okamoto. “Multidimensional replica-exchange method for free-energy calculations”. In: *The Journal of Chemical Physics* 113.15 (2000), pp. 6042–6051. DOI: `10.1063/1.1308516`. eprint: `https://doi.org/10.1063/1.1308516`.

[85] U. Essmann et al. “A Smooth Particle Mesh Ewald Method”. In: *Journal of Chemical Physics* 103.19 (1995), pp. 8577–8593. ISSN: 0021-9606. DOI: `Doi10.1063/1.470117`.

[86] J. Jung et al. “GENESIS: a hybrid-parallel and multi-scale molecular dynamics simulator with enhanced sampling algorithms for biomolecular and cellular simulations”. In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 5.4 (2015), pp. 310–323. ISSN: 1759-0876. DOI: `10.1002/wcms.1220`.

[87] C. Kobayashi et al. “GENESIS 1.1: A hybrid-parallel molecular dynamics simulator with enhanced sampling algorithms on multiple computational platforms”. In: *Journal of Computational Chemistry* 38.25 (2017), pp. 2193–2206. DOI: `10.1002/jcc.24874`.

[88] P. Liu et al. “Replica exchange with solute tempering: A method for sampling biological systems in explicit water”. In: *Proceedings of the National Academy of Sciences* 102.39 (2005), pp. 13749–13754. ISSN: 0027-8424. DOI: `10.1073/pnas.0506346102`. eprint: `https://www.pnas.org/content/102/39/13749.full.pdf`.

[89] L. Wang, R. A. Friesner, and B. J. Berne. “Replica Exchange with Solute Scaling: A More Efficient Version of Replica Exchange with Solute Tempering (REST2)”. In: *The Journal of Physical Chemistry B* 115.30 (2011). PMID: 21714551, pp. 9431–9438. DOI: `10.1021/jp204407d`. eprint: `https://doi.org/10.1021/jp204407d`.

[90] M. Kamiya and Y. Sugita. “Flexible selection of the solute region in replica exchange with solute tempering: Application to protein-folding simulations”. In: *The Journal of Chemical Physics* 149.7 (2018), p. 072304. DOI: `10.1063/1.5016222`. eprint: `https://doi.org/10.1063/1.5016222`.

[91] L. Maragliano et al. “String method in collective variables: Minimum free energy paths and isocommittor surfaces”. In: *The Journal of Chemical Physics* 125.2 (2006), p. 024106. DOI: `10.1063/1.2212942`. eprint: `https://doi.org/10.1063/1.2212942`.

[92] J. Jung et al. “Parallel implementation of 3D FFT with volumetric decomposition schemes for efficient molecular dynamics simulations”. In: *Computer Physics Communications* 200 (2016), pp. 57–65. ISSN: 0010-4655.

[93] J. Jung, T. Mori, and Y. Sugita. “Midpoint Cell Method for Hybrid (MPI+OPENMP) Parallelization of Molecular Dynamics Simulations”. In: *Journal of Computational Chemistry* 35.14 (2014), pp. 1064–1072. ISSN: 0192-8651. DOI: `Doi10.1002/Jcc.23591`.

[94] J. Jung et al. “New parallel computing algorithm of molecular dynamics for extremely huge scale biological systems”. In: *Journal of Computational Chemistry* 42.4 (2021), pp. 231–241. DOI: `https://doi.org/10.1002/jcc.26450`.

[95] J. Jung, T. Mori, and Y. Sugita. “Efficient Lookup Table Using a Linear Function of Inverse Distance Squared”. In: *Journal of Computational Chemistry* 34.28 (2013), pp. 2412–2420. ISSN: 0192-8651. DOI: `Doi10.1002/Jcc.23404`.

[96] J. Jung, C. Kobayashi, and Y. Sugita. “Kinetic energy definition in velocity Verlet integration for accurate pressure evaluation”. In: *J Chem Phys* 148.16 (2018), p. 164109. ISSN: 1089-7690 (Electronic) 0021-9606 (Linking). DOI: `10.1063/1.5008438`.

[97] “Group-based evaluation of temperature and pressure for molecular dynamics simulation with a large time step”. In: *J Chem Phys* 153.23 (2020), p. 234115. DOI: `10.1063/5.0027873`.

Chapter 8

Codesign of Genomon

8.1 Co-Design of Genomon

8.1.1 Application Features

Genomon is a whole genome sequencing (WGS) data analysis pipeline for human cancer developed in the Human Genome Center at the Institute of Medical Science, the University of Tokyo. In the basic design phase, Genomon-fusion for K (GFK), Genomon-fusion running on K computer, was taken into account as a target application. It analyzes RNA sequences utilizing the Bowtie and BLAT alignment tools, and detects mutations of fusion genes that cause cancer. After the completion of the basic design phase, the target application was replaced with Genomon2. Genmon2 employs the BWA (Burrows-Wheeler Aligner) tool for alignments of WGS data. The target problem size was decided as follows, based on the prediction of high-througput genome sequencer's performance in 2017.

Before 2017 read counts: 3.9M read, read length: 200 base pair

After 2017 read counts: 1.4M read,read length: 150 base pair

The WGS variant callers are a kind of workflow applications performed by connecting small programs that process using input data and output data for the next program in a pipeline way. Programs running in each pipeline stage are called a task in this paper. Also, Genomon2 is referred to Genomon in this paper.

Genomon workflow and job scheduling are controlled by Ruffus[1] and Grid Engine (Sun Grid Engine/Univa Grid Engine). The overall workflow of the DNA analysis pipeline is shown in Figure 8.1. The area in the red frame is used for the Fugaku evaluation.

In order to carry out high-throughput ensemble computing for genome analysis, such as Genomon, on a large-scale supercomputer, the following common features of such applications must be paid attention:

- Many of tasks in pipeline are not thread-parallelized or cannot be parallelized.
- Node-level parallelisms in each task differs than other tasks.
  - The alignment computation task can be parallelized with any number of nodes by dividing the input data. Its load-balance can be well obtained.
  - On the other hand, node-level parallelisms in other tasks are limited and its load-balance cannot be well obtained.

[1] `http://www.ruffus.org.uk/`

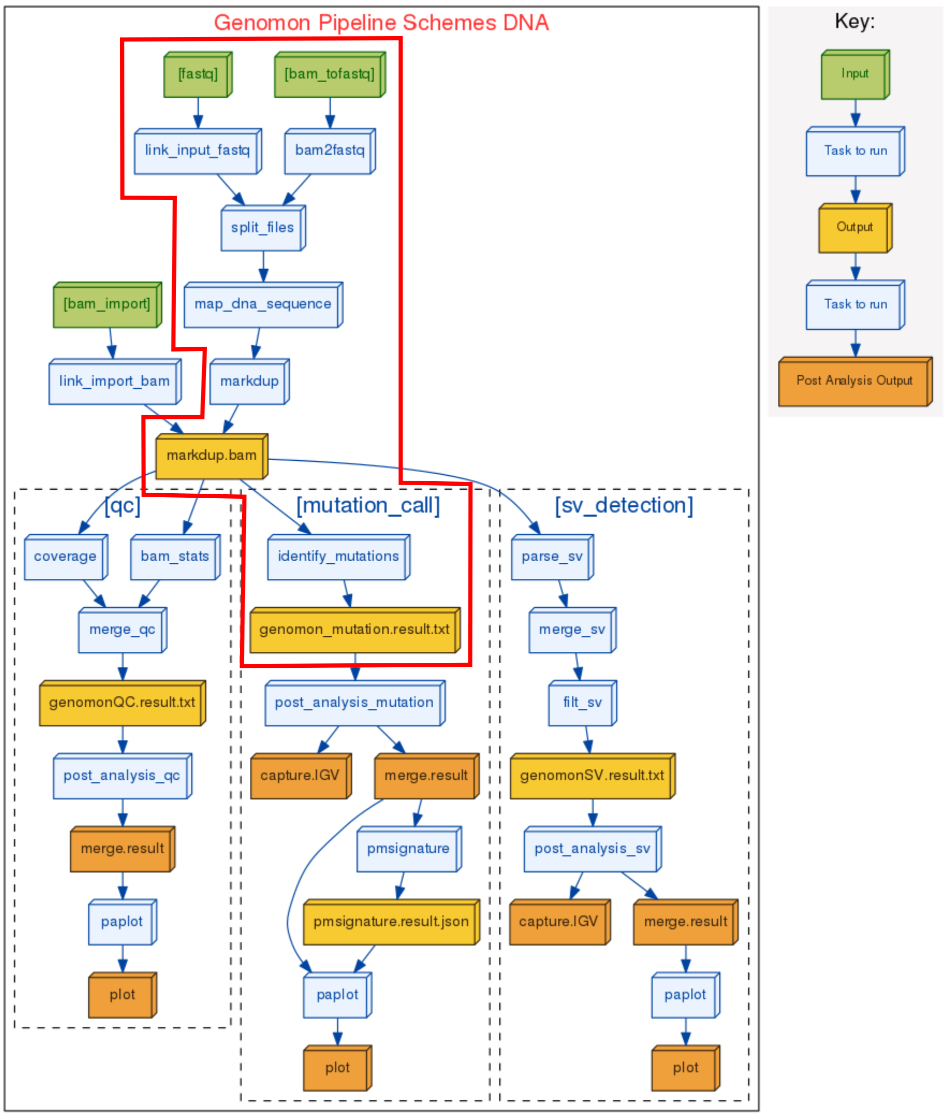


Figure 8.1: DNA analysis workflow in Genomon

– It cannot be ignored that some tasks are serialized.

Genomon has following characteristics from the view of codesign methodology: It is an integer arithmetic intensive application for string operations. Some program's execution times are dominated by file I/O processing. Several processes may access about tera-byte files at the same time. Thus, codesign of Genomon is not only for the Genomon application, but also contributes to applications whose performance is dominated by integer arithmetic and file I/O processing.

Since Genomon utilizes open source codes and its workflow is written by python, we have not optimized such codes except the BWA tool, which SIMD paralleization was applied to. This report focuses on the workflow runtime environment and file I/O which are key contributions.

8.1.2 Workflow Runtime

8.1.2.1 User-level Job Scheduler

Many large-scale bioinformatics applications, such as Genomon, are described using the array job feature provided by a batch job system, such as the Grid Engine. An array job is a collection of jobs each of which uses the same application code, but different input parameters. Especially, Genomon has been provided with the assumption of Grid Engine.

Genomon-fusion for K was running on VGE (Virtual Grid Engine), developed in the Human Genome Center. VGE is an MPI application that realizes the array job feature and other Grid Engine features needed by Genomon. Though the same function of Grid Engine is provided in Fugaku, it is not the same API and Genomon must be modified to adopt it to the Fugaku environment. Thus, the Human Genome Center has ported VGE to Fugaku.

Though Genomon does not require Grid Engine features in Fugaku, the following requirements were summarized and requested to Fujitsu:

- A batch job can be submitted on compute nodes.
- The smallest job unit should be node or CMG.
- The limit on the number of jobs in a bulk job is relaxed. A bulk job is the similar function of array job in K and Fugaku, but only 15 jobs can be submitted in K.

8.1.3 File I/O

8.1.3.1 Investigation of File I/O Patterns

Genomon consists of open source codes, not developed by the Genomon developers, and thus all intermediate files generated by those codes are not known. In order to reveal such file I/O patterns and file sizes, a file I/O profiler was developed and applied to Genomon. The profiler is an extension of Darshan[2], that enables an MPI application to capture I/O behavior. Extended features are listed below: i) recording I/O stream functions, such as fwrite, ii) recording stack trace at an I/O function call, iii) recording the maximum file size if the same file is multiply opened and closed, and iv) recording file I/O by a program invoked by the multiprocessing module in python. The extended Darshan I/O profiler is available in `https://github.com/yutaka-ishikawa/darshan-mod/`.

The placements of files used in applications, the 1^{st} or 2^{nd} layer storage, were decided based on a result of measuring file I/O patterns and file sizes by the extended Darshan I/O profiler.

8.1.3.2 Utilization of the 1^{st} layer Storage

In Fugaku, SSDs as the 1^{st} layer storage are utilized by dividing it into the following areas for each purpose.

(1) Temporary files within node
(2) Shared files among nodes
(3) Copy of files on the 2^{nd} layer
(4) Cache for files on the 2^{nd} layer

Files used in Genomon are classified into input files, output files, and temporary files and those are ideally placed in the 1^{st} layer storage as shown below.

[2] `https://www.mcs.anl.gov/research/projects/darshan/`

- Input files

  Accessed only by programs on a single node → (4)
  Others → (3)

- Output files → (4)

- Temporary files

  Temporally accessed by a program → (1)
  Temporally accessed in a task → (1)
  Temporally accessed across tasks → (2)

Genomon uses 96 nodes for one sample analysis in the target problem size. Since 1,600 GB 1st layer storage are equipped in each 16 nodes, the total amount of 1st layer storage is 9,600 GB on 96 nodes, and 100 GB can be used per node.

The requirement of total file sizes in most tasks except one task does not exceed 9,600 GB, but one task exceeds this limitation because it generates 157.1 GB temporary file in each node. This file is allocated in a shared temporary area so that storage size requirements of each node is reduced.

8.1.4 Results

8.1.4.1 Computational Performance in 100 base pairs

Because we could not obtain sample data whose base pair size is 150, sample data of 100 base pairs are used to measure the computational performance. Experimental job parameters are summarized as follows:

- The node allocation option is "4x3x8:strict-io"
- Stripe size and counts in the 2nd layer storage are 2 MiB and 32, respectively
- Stripe size and counts in the 1st layer storage are 2 MiB and 6, respectively.

Because it is not enough capacity in the 1st storage to hold all input files, those are accessed via cache provided by the 1st storage. The access latency of the 2nd storage depends on the load of the 2nd storage that are shared by other jobs. Thus, the performance result is the fastest execution time of multiple executions.

The measurement was performed for four cases of the combination of normal or boost mode and eco-disabled or enabled mode. The result is shown in Table 8.1. In this table, average power consumption is calculated by the average of all "AVG POWER CONSUMPTION OF NODE (MEASURED)" values of nodes, reported in the statistic information generated by the batch job system. The maximum power consumption is also calculated using the "MAX POWER CONSUMPTION OF NODE (MEASURED)" values of nodes.

In Genomon, the variation in execution time due to external factors is larger than the effect of eco-setting. In this experience, the eco-enabled boost mode was slightly faster than the eco-disabled boost mode.

As shown in Figure 8.2, the power consumption varies greatly depending on the type of node in Genomon. In this figure, the left and right graphs shows average and maximum power consummations, respectively. CN, BIO, SIO, and GIO represents compute only node, compute node with boot IO device, compute node with SSD, compute node with I/O network. ALL represents all nodes.

Table 8.1: Computational Performance in 100 base pairs

	Eco off		Eco on	
	Normal	Boost	Normal	Boost
exec. time (s)	8918	8552	9038	8532
Max memory/Node(GiB)	24.8	24.8	24.7	24.8
Avg. power cons./Node(W)	103	116	76	84
Max power cons./Node(W)	110	123	83	92

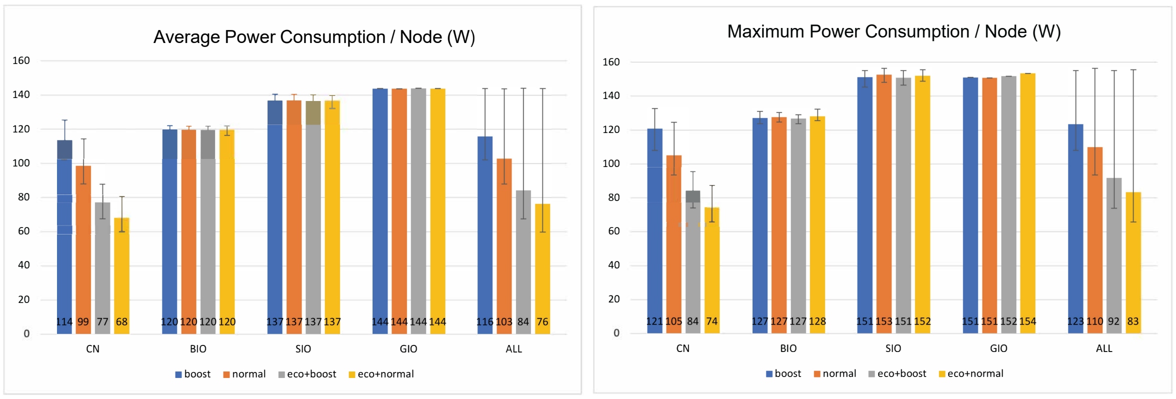


Figure 8.2: Power Consumption in Genomon

8.1.4.2 Prediction of Computational Performance in 150 base pairs

The computational performance in 150 base pairs is predicted using the result of 100 base pairs. The result is shown in Table 8.2. Note that power consumption is not the predicted one, but the result of 100 base pairs.

Table 8.2: Predicted performance using 100 base pairs

	K	Fugaku			
	Baseline	Eco off		Eco on	
		Normal	Boost	Noramal	Boost
# node used	36	96	96	96	96
Analyzed samples/Day	261	5888	6046	5732	6091
Time for 1 analysis (day)	8.84	0.28	0.27	0.29	0.27
Improved perf. ratio	1	22.6	23.2	22.0	23.4
Max memory/Node(GiB)	-	24.8	24.8	24.7	24.8
Avg. power cons./node(W)	-	103	116	76	84
Max power cons./node(W)	-	110	123	83	92

Chapter 9

Codesign of GAMERA

9.1 Overview of target application

GAMERA is an implicit unstructured finite element analysis method for seismic ground motion simulation originally designed for the K computer (Ichimura at al. 2014 (SC14)). This program combines mixed-precision arithmetic and multi-grid method in the preconditioner of a conjugate gradient method for solving large linear system of equations at each time step, and the Element-by-Element (EBE) method is used for matrix-vector products in the conjugate gradient solver. In standard finite-element solvers, sparse matrices

$$K = \sum_i P_i K_i P_i^T, \tag{9.1}$$

is often stored in memory and read each time a matrix-vector product is called as

$$f \Leftarrow Ku. \tag{9.2}$$

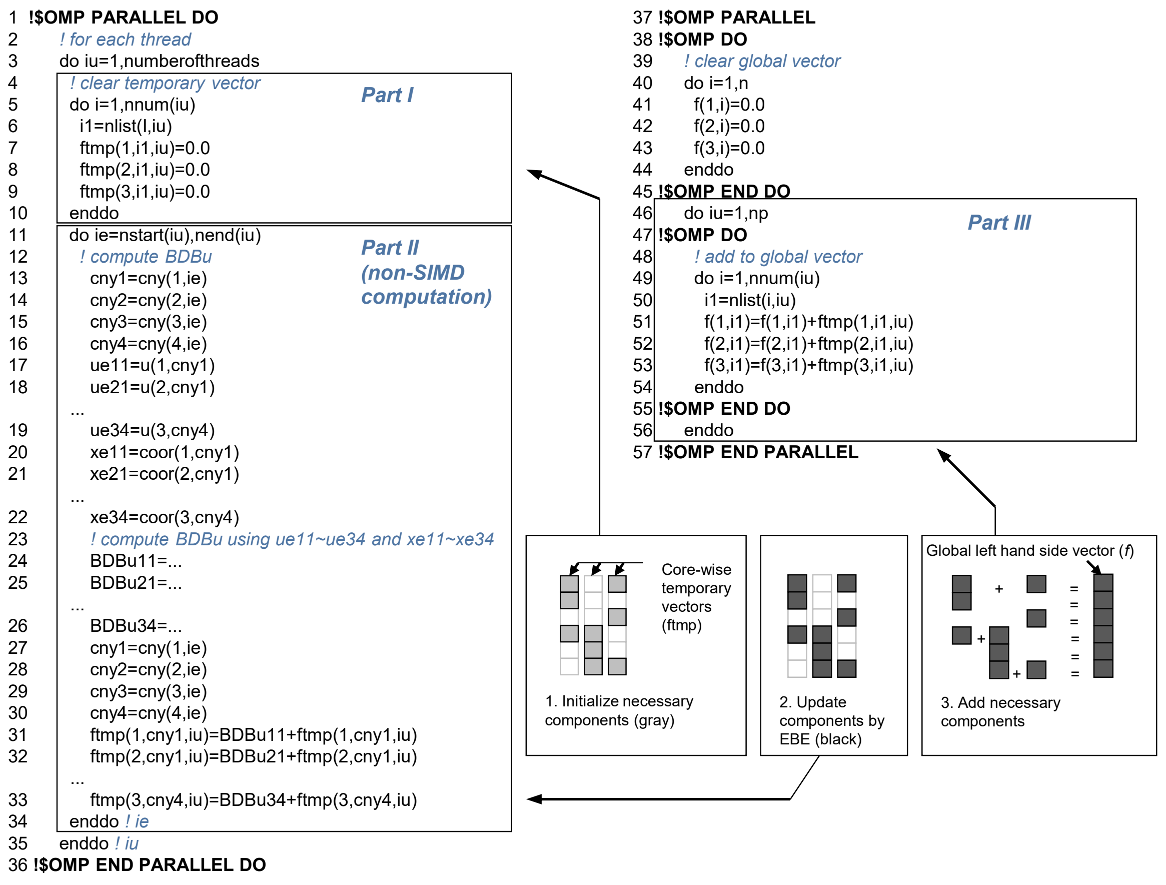
On the other hand, matrix-vector products are computed as

$$f \Leftarrow \sum_i (P_i(K_i(P_i^T u))). \tag{9.3}$$

in the EBE method. Here, the subscript indicates the value related to the element number i, K_i is the element matrix, and P_i is the mapping matrix from the global node number to the element node number. By using Eq. (9.3) for the matrix-vector product, the amount of data read from memory to compute K_i and P_i is reduced from the amount of data read from memory when using the global matrix K in Eq. (9.2). On the other hand, on-cache computation is increased for generating K_i at each matrix-vector product, and random load of u and random write to f are required. As the single-precision EBE kernel in the second-order tetrahedral element mesh used in the multi-grid preconditioner becomes the most costly kernel of the total application, we conducted codesign of single-precision EBE kernel in GAMERA for Fugaku. In this way, GAMERA contributes to codesign of applications with on-cache unstructured computation, that requires random access performance such as indirect SIMD load (gather load) and low cache latency.

9.2 EBE kernel for K computer

Figure 9.1 shows the overview of the EBE kernel before codesign. When parallelizing Eq. (9.3) in the element direction (i), data recurrence occurs in f for nodes that are shared between elements. Thus, in the kernel for K computer, temporary buffer `ftmp` is generated (Part I in Fig. 9.1), and each thread adds results $(P_i(K_i(P_i^T u)))$ into `ftmp` (Part II in Fig. 9.1), and thread wise results are added into the final vector `f` (Part III in Fig. 9.1). As the parts accessed by each thread is localized, only the parts updated is initialized to 0 in Part I and added in Part III for reducing thread parallelization cost. On the other hand, as data recurrence occurs in `ftmp`, SIMD cannot be applied to the main computation loop (Part II).



```
!$OMP PARALLEL DO
  ! for each thread
  do iu=1,numberofthreads
   ! clear temporary vector
   do i=1,nnum(iu)
     i1=nlist(I,iu)
     ftmp(1,i1,iu)=0.0
     ftmp(2,i1,iu)=0.0
     ftmp(3,i1,iu)=0.0
   enddo
   do ie=nstart(iu),nend(iu)
     ! compute BDBu
       cny1=cny(1,ie)
       cny2=cny(2,ie)
       cny3=cny(3,ie)
       cny4=cny(4,ie)
       ue11=u(1,cny1)
       ue21=u(2,cny1)
   ...
       ue34=u(3,cny4)
       xe11=coor(1,cny1)
       xe21=coor(2,cny1)
   ...
       xe34=coor(3,cny4)
       ! compute BDBu using ue11~ue34 and xe11~xe34
       BDBu11=...
       BDBu21=...
   ...
       BDBu34=...
       cny1=cny(1,ie)
       cny2=cny(2,ie)
       cny3=cny(3,ie)
       cny4=cny(4,ie)
       ftmp(1,cny1,iu)=BDBu11+ftmp(1,cny1,iu)
       ftmp(2,cny1,iu)=BDBu21+ftmp(2,cny1,iu)
   ...
       ftmp(3,cny4,iu)=BDBu34+ftmp(3,cny4,iu)
   enddo ! ie
  enddo ! iu
!$OMP END PARALLEL DO
!$OMP PARALLEL
!$OMP DO
  ! clear global vector
  do i=1,n
    f(1,i)=0.0
    f(2,i)=0.0
    f(3,i)=0.0
  enddo
!$OMP END DO
  do iu=1,np
!$OMP DO
    ! add to global vector
    do i=1,nnum(iu)
      i1=nlist(i,iu)
      f(1,i1)=f(1,i1)+ftmp(1,i1,iu)
      f(2,i1)=f(2,i1)+ftmp(2,i1,iu)
      f(3,i1)=f(3,i1)+ftmp(3,i1,iu)
    enddo
!$OMP END DO
  enddo
!$OMP END PARALLEL
```

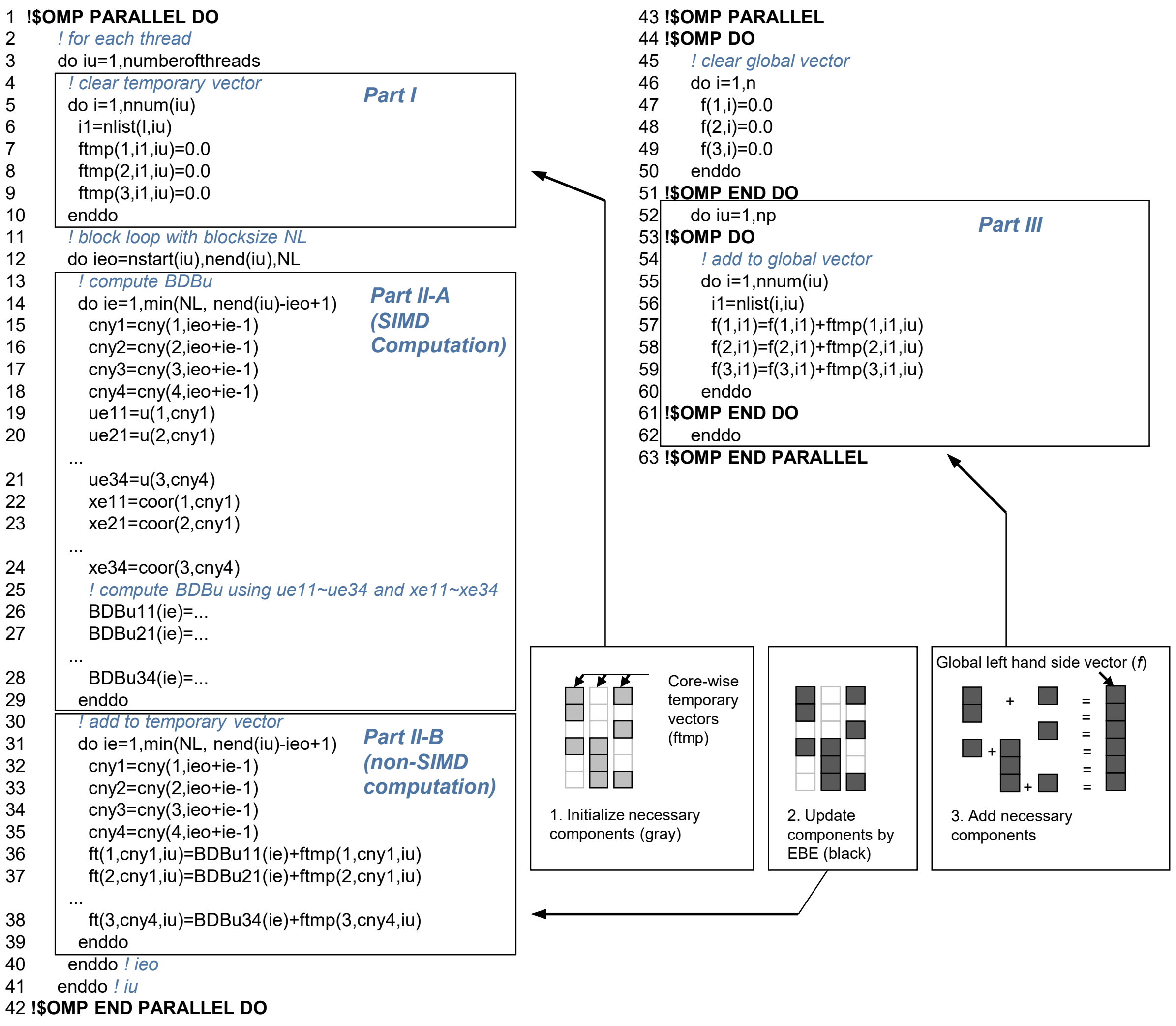
Part I
Part II (non-SIMD computation)
Part III
Core-wise temporary vectors (ftmp)
1. Initialize necessary components (gray)
2. Update components by EBE (black)
Global left hand side vector (f)
3. Add necessary components

Figure 9.1: EBE kernel before codesign

9.3 Codesign of EBE kernel for Fugaku

9.3.1 Use of SIMD in EBE kernel

We first worked on the parallelization of SIMD in Part II, keeping in mind that the SIMD width increases in Fugaku. Here, we first split Part II into two parts: the first half without data recurrence and the other with data recurrence (Fig. 9.2 Part II-A and II-B). By such loop splitting, parallel computation by SIMD is utilized in the first half. Since computation of $(P_i(K_i(P_i^T u)))$, which is the main computation of the EBE kernel, is performed in Part II-A, most of the floating-point computation of the kernel can be performed by SIMD arithmetic units. Temporary buffers (`BDBu11`∼`BDBu34`) for passing variables between Part II-A and II-B is required to divide Part II into two; however, by using loop blocking (block length `NL`=SIMD width), we can suppress the size of the temporary buffers, and the SIMD calculation of the main computation part can be conducted on L1 cache.



```
!$OMP PARALLEL DO
  ! for each thread
  do iu=1,numberofthreads
   ! clear temporary vector
   do i=1,nnum(iu)
    i1=nlist(I,iu)
    ftmp(1,i1,iu)=0.0
    ftmp(2,i1,iu)=0.0
    ftmp(3,i1,iu)=0.0
   enddo
   ! block loop with blocksize NL
   do ieo=nstart(iu),nend(iu),NL
    ! compute BDBu
    do ie=1,min(NL, nend(iu)-ieo+1)
     cny1=cny(1,ieo+ie-1)
     cny2=cny(2,ieo+ie-1)
     cny3=cny(3,ieo+ie-1)
     cny4=cny(4,ieo+ie-1)
     ue11=u(1,cny1)
     ue21=u(2,cny1)
   ...
     ue34=u(3,cny4)
     xe11=coor(1,cny1)
     xe21=coor(2,cny1)
   ...
     xe34=coor(3,cny4)
     ! compute BDBu using ue11~ue34 and xe11~xe34
     BDBu11(ie)=...
     BDBu21(ie)=...
   ...
     BDBu34(ie)=...
    enddo
    ! add to temporary vector
    do ie=1,min(NL, nend(iu)-ieo+1)
     cny1=cny(1,ieo+ie-1)
     cny2=cny(2,ieo+ie-1)
     cny3=cny(3,ieo+ie-1)
     cny4=cny(4,ieo+ie-1)
     ft(1,cny1,iu)=BDBu11(ie)+ftmp(1,cny1,iu)
     ft(2,cny1,iu)=BDBu21(ie)+ftmp(2,cny1,iu)
   ...
     ft(3,cny4,iu)=BDBu34(ie)+ftmp(3,cny4,iu)
    enddo
   enddo ! ieo
  enddo ! iu
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL
!$OMP DO
  ! clear global vector
  do i=1,n
   f(1,i)=0.0
   f(2,i)=0.0
   f(3,i)=0.0
  enddo
!$OMP END DO
  do iu=1,np
!$OMP DO
   ! add to global vector
   do i=1,nnum(iu)
    i1=nlist(i,iu)
    f(1,i1)=f(1,i1)+ftmp(1,i1,iu)
    f(2,i1)=f(2,i1)+ftmp(2,i1,iu)
    f(3,i1)=f(3,i1)+ftmp(3,i1,iu)
   enddo
!$OMP END DO
  enddo
!$OMP END PARALLEL
```

Part I
Part II-A (SIMD Computation)
Part II-B (non-SIMD computation)
Part III
Core-wise temporary vectors (ftmp)
1. Initialize necessary components (gray)
2. Update components by EBE (black)
Global left hand side vector (f)
3. Add necessary components

Figure 9.2: EBE kernel computed using SIMD arithmetic units

9.3.2 Coloring in EBE kernel for efficient multi-core computation

Since Part II became faster due to the use of SIMD, the random access costs in Part I and Part III became relatively large. Especially in Fugaku, the cost of these random accesses increases becomes significant as the SIMD width increases compared to the K computer. Therefore, as a codesign for Fugaku, we carried out coloring of the element loop to reduce the random access costs involved in Part I and III (Fig. 9.3). In normal element coloring, a subset of elements that do not share a node is extracted and used as one color (Fig. 9.4). This method allows the nodal values calculated in each thread to be added directly to f, eliminating the use of temporary vector `ftmp`. On the other hand values of u and f cannot be reused on the cache. In order to circumvent this situation, we developed a graph partitioning-based method to partition the target mesh into the number of threads (Fig. 9.5). As a result, u and f can be reused on the cache in each thread, and Parts I and III can be omitted.

9.3.3 Reduction of register spills by loop splitting of the EBE kernel

Part II-A (Fig. 9.3), which is the main calculation part of the EBE kernel, has a large loop body of 500 lines or more in the source code. In the K computer, high-speed calculation with reduced cache access was possible by utilizing a large number of registers, but in Fugaku, it was expected that spills would occur frequently due to the limitation in the number of registers. Indeed, it turned out that compiling that part of the code produces many register spills. Therefore, the relevant loop was subdivided into 18 loops. As a result, the number of spills in the assembly code could be reduced by 5-fold (from 6692 to 1149).

9.4 Performance of GAMERA on Fugaku

As a result of the codesign, the developed solver attained 63-fold speedup from that of the as is solver on full K computer. The details of the measurement results are shown below. First, in order to confirm the scalability of the application, we measured the performance on weak scaling model set A-1 to A-9 by setting the problem scale per compute node close to the target problem (Table 9.1). The smallest model (A-1) at 288 nodes could run at 4.13 s, while the target model A-Fugaku (1.08 trillion degrees of freedom (DOF)) at 147456 nodes was 5.54 s (49.6 TFLOPS, corresponding to 9.97% of FP64 peak performance) (Fig. 9.6). In this way, it was confirmed that high weak scaling performance was achieved from 288 nodes to 147456 nodes. Figure 9.6 also shows the performance when the as is code for K computer before codesign is measured on Fugaku. It can be seen that a significant speedup has been achieved by codesign in each weak scaling model, such as achieving 7.0 times faster speed in the target problem. It is considered that the efficient calculation method of random access-based calculation developed in this study can be applied to other applications, and it is expected that the performance of those applications will be improved.

Details of the development and measurements are reported in: Kohei Fujita, Kentaro Koyama, Kazuo Minami, Hikaru Inoue, Seiya Nishizawa, Miwako Tsuji, Tatsuo Nishiki, Tsuyoshi Ichimura, Muneo Hori, Lalith Maddegedara, High-fidelity nonlinear low-order unstructured implicit finite-element seismic simulation of important structures by accelerated element-by-element method, Journal of Computational Science, 2020, https://doi.org/10.1016/j.jocs.2020.1012.

Table 9.1: Model set used for weak scaling of GAMERA. A-Fugaku and A-K indicate the target problem on Fugaku and K computer, respectively.

Model	# of nodes	# of MPI processes	DOF	DOF per process
A-1	288	1152	1888114923	1638988
A-2	576	2304	3775253451	1638564
A-3	1152	4608	7548554667	1638141
A-4	2304	9216	15095157099	1637929
A-5	4608	18432	30186410283	1637718
A-6	9216	36864	60368916651	1637612
A-7	18432	73728	120730026027	1637505
A-8	36864	147456	241452244779	1637452
A-9	73728	294912	482888875563	1637399
A-Fugaku	147456	589824	1086476549163	1842035
A-K	82944	82944	1086476549163	13098916

```
!$OMP PARALLEL
!$OMP DO
    ! clear global vector
    do i=1,n
      f(1,i1)=0.0
      f(2,i1)=0.0
      f(3,i1)=0.0
    enddo
!$OMP END DO
    do icolor=1,ncolor
!$OMP DO
    ! for each thread
    do iu=1,numberofthreads
    ! block loop with blocksize NL
      do ieo=nstart(iu,icolor),nend(iu,icolor),NL
        ! compute BDBu
        do ie=1,min(NL, nend(iu)-ieo+1)    Part II-A
          cny1=cny(1,ieo+ie-1)             (SIMD
          cny2=cny(2,ieo+ie-1)             computation)
          cny3=cny(3,ieo+ie-1)
          cny4=cny(4,ieo+ie-1)
          ue11=u(1,cny1)
          ue21=u(2,cny1)
    ...
          ue34=u(3,cny4)
          xe11=coor(1,cny1)
          xe21=coor(2,cny1)
    ...
          xe34=coor(3,cny4)
          ! compute BDBu using ue11~ue34 and xe11~xe34
          BDBu11(ie)=...
          BDBu21(ie)=...
    ...
          BDBu34(ie)=...
        enddo
        ! add to temporary vector
        do ie=1,min(NL, nend(iu)-ieo+1)    Part II-B
          cny1=cny(1,ieo+ie-1)             (non-SIMD
          cny2=cny(2,ieo+ie-1)             computation)
          cny3=cny(3,ieo+ie-1)
          cny4=cny(4,ieo+ie-1)
          ft(1,cny1)=BDBu11(ie)+f(1,cny1)
          ft(2,cny1)=BDBu21(ie)+f(2,cny1)
    ...
          ft(3,cny4)=BDBu34(ie)+f(3,cny4)
        enddo
      enddo ! ieo
    enddo ! iu
!$OMP END DO
  enddo ! icolor
!$OMP END PARALLEL
```

Figure 9.3: Reduction of random access in EBE kernel by element coloring

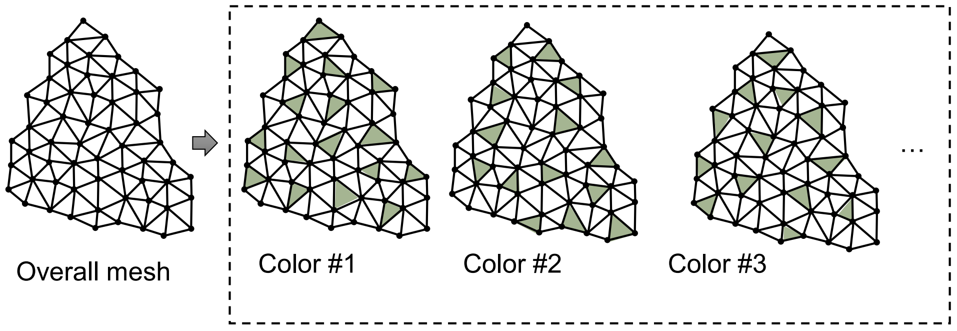


Figure 9.4: Standard coloring method

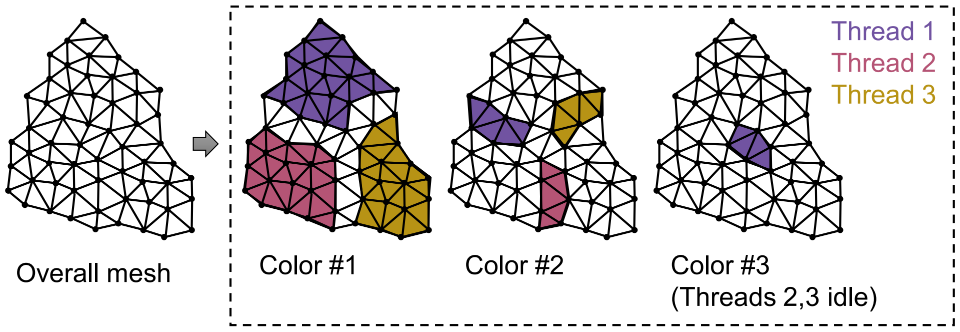


Figure 9.5: Developed coloring method

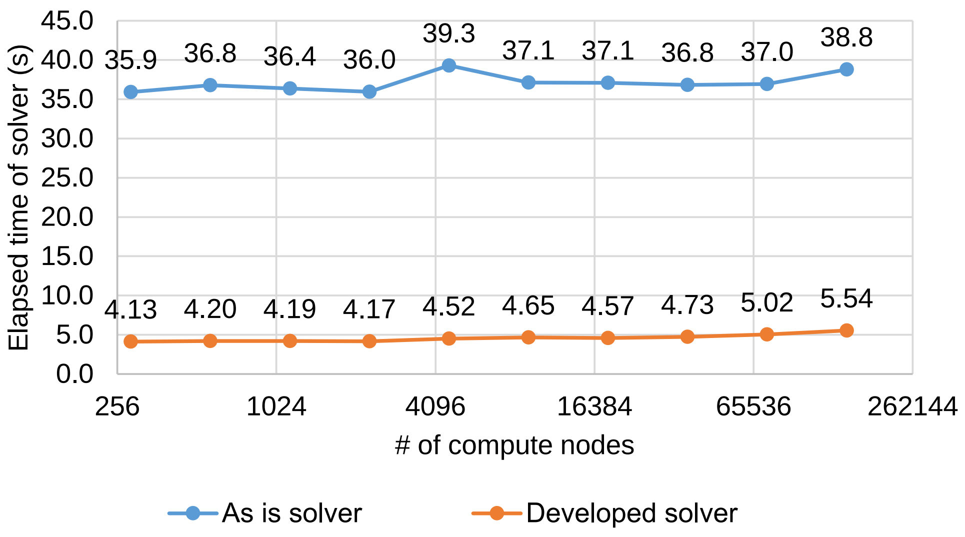


Figure 9.6: Weak scaling of GAMERA on Fugaku

Chapter 10

Codesign of NICAM+LETKF

10.1 Application Features and Codesign target

10.1.1 Weather/Climate applications

'Priority Issue 4, "Advancement of meteorological and global environmental predictions utilizing observational 'Big Data,' " aimed to develop the technological basis for accurate prediction of meteorological disasters that threaten human life and property. In the challenge to get a longer "lead time", which is the time from a forecast of weather disaster to its occurrence, we chose both the global weather simulation model and the data assimilation (DA) system as the target application to codesign with the supercomputer system.

Weather/climate application software has a long history and is a multidisciplinary community code with many components. The total number of lines of source code is in the hundreds of thousands, and the type of algorithm and degree of optimization of each component varies. Since the computational section uses many state variables at 3D grid points, the software tends to be bounded by the memory performance. In most cases, these simulation models do not have so-called "hot spot" sections that account for the most computational amount and elapsed time [98, 99, 100]. The cost ranking by the performance profiler shows low peak and long tail, and we call this the "flat profile." The part that solves the hydrodynamics and tracer advection is called the dynamical core or the "dycore" and accounts for about half of the computation time. The other half is the physical processes that calculate, for example, the phase change of clouds and radiative transfer in the atmosphere.

Commonly used in weather DA systems are the variational method and the ensemble Kalman filter. The computational characteristics of these applications are very different from those of simulation models. They have sections that read in model simulation results and observation data, perform spatial interpolation, and perform matrix and eigenvalue calculations. The variational method has a component called the adjoint code that performs the inverse of the simulation. The ensemble Kalman filter does not require such an adjoint code, but it is necessary to perform multiple simulations (ensemble) with slightly different initial or boundary values.

10.1.2 NICAM+LETKF

We selected NICAM+LETKF [101] as our target application. NICAM+LETKF is a complex application that consists of a Nonhydrostatic ICosahedral Atmospheric Model (NICAM) [102, 103, 104] and the ensemble DA system based on the Local Ensemble Transform Kalman Filter (LETKF) [105, 106, 107]. This software solves the time evolution of global weather using the ensemble simulation of the atmosphere and applies analytical process with Earth observations.

The weather forecast requires both an accurate prediction model and precise initial condition. NICAM is developed to improve the reproducibility of atmospheric phenomena with higher spatial resolution. Since the computational amount increases explosively when the grid spacing of the model is made finer, efficient computation is essential factor. NICAM is a next-generation atmospheric model that discretizes non-hydrostatic equations on icosahedral grid points and employs a software design suitable for vectorization and massively parallel computing and realized the first atmospheric simulation with a horizontal resolution less than 1 km on the K computer [108]. In the codesign process, NICAM mainly contributes to evaluating memory and cache transfer performance and latency, thread parallelism, scheduling, and out-of-order execution, as a representative application of stencil computation using a structured grid system. Furthermore, the non-stencil part of NICAM is expected to improve compiler performance by accumulating knowledge on optimization methods for program codes with big loop bodies and frequent conditional branches.

Another application, DA systems, is as important to weather forecasting as simulation models. Increasing the number of ensembles is a way to increase the accuracy of estimating initial state values for forecasting using ensemble-based DA. NICAM+LETKF has been used to perform a low-resolution calculation with 10240 members on the K computer [109]. The experimental results show that the ensemble size, which is more than an order of magnitude larger than the conventional one, contributes to the use of more spatially distant observational information. The DA system mainly contributes to evaluating large-sized file input/output (I/O), global communication, and eigenvalue solver in the numerical library.

10.1.3 Benchmark Setting

We set our target problem directly relevant to future operational weather forecasting; the horizontal resolution was a global 3.5 km mesh, the number of vertical layers was 94, and the ensemble size was 1024. We made this decision after considering both the total memory capacity and the computation time. The Japan Meteorological Agency performs DA and 72-hour forecasts every 6 hours. Therefore, we selected a horizontal resolution to be able to compute to meet this schedule using the entire Fugaku system. The memory capacity was not enough to run 1024 members simultaneously, so We divided into four runs with 256 members. We determined the number of members based on previous 10240-member experiments, which showed that at least 1024 members are necessary to achieve a significant improvement in forecast performance. Our target problem size is more than 500 times larger than that used by the weather centers around the world for ensemble DA in terms of the combination of horizontal resolution and the number of ensemble members. We set the number of the time steps to 2700, equivalent to 4.5 hours with a time interval of 8 seconds. After the simulation part, we calculate the DA part using the simulation result and hundreds of thousands of observation data. We defined the combination of one simulation part and one DA part as one assimilation cycle. The time required to execute 480 cycles (corresponding to the two months in the simulation) was used as the criterion for performance evaluation.

The execution time of single NICAM members on the K computer was estimated from the measured times of 10-nodes experiment, assuming perfect weak scaling. The total elapsed time of the simulation part was estimated by dividing it into 32-times runs with 32 members. Regarding the DA part, we measured the kernel code with the same problem size per process. The elapsed times of communication parts for both NICAM and DA were estimated from the measurement of the communication kernel using 81,920 nodes. The file I/O parts were also estimated from the measurement using the 2560 node of the K computer. By combining these results, we estimated that it would take 15 days per cycle and 20 years for 480 cycles.

Since the target problem could only be actually measured in Fugaku, each interval of elapsed time was estimated in a different way. For the NICAM part, 256-member computation was substituted by one member. In addition, the measurements were weak-scaled from 2560 MPI processes of 3.5 km mesh to 10 processes of 56 km mesh with the same problem size per process. In the middle of the project, the number of processes per member was changed from 2560 (10) to 2048 (8). For the DA system, the file I/O and global communication part were estimated based on the amount of data movement, and the computational core part of LETKF was measured by one process calculation using the same problem size.

It was difficult to run the full application on the emulator and simulator of Fugaku. Therefore, we extracted specific sections from the NICAM and DA system as kernels. These kernels have computational patterns that characterize the application. However, as mentioned above, NICAM has a computational performance feature called "flat profile", and the total computation time of these kernels is less than half of the total computation time. Therefore, to estimate the total computation time, we used the performance mapping method described in later section.

10.2 Detail of codesign

10.2.1 Kernel Extraction and Optimization

In this section, we mainly discuss the kernels extracted from NICAM. In order to estimate the computational part in NICAM, six kernels were extracted from the stencil kernels with different characteristics such as data dependency and conditional branching, and three kernels were extracted from the physical processes.

In optimizing the kernels, we had to be aware of two trade-offs. One is to ensure data locality and parallelism. For the K computer and Fugaku, the SIMD width increased from 2 to 8 in double precision, and from 2 to 16 in single precision. In order to process data continuously with this SIMD width, the size of the innermost loop and array must be large enough. In addition, the length of the innermost loop needs to be longer in order for software pipelining to be effective. However, the memory footprint corresponding to this degree of parallelism cannot stay in the cache for long. Since weather/climate simulation models use three-dimensional state variables of the atmosphere such as wind speed, temperature, and water vapor content in their calculations, there are many large-size arrays to be used in the loop. This further reduces the reusability of the cache.

The changes in the typical loop structure of NICAM due to the optimization strategies are shown in Figure 10.1. First, we coded in a style that relied on memory performance to ensure a continuous supply of data. From the point of view of continuous reading of data, stencil calculations did not show good performance with hardware prefetching. This is because when calculating 2-D or 3-D grid points in sequence, the halo grid points are skipped without calculation, which may be judged as a cache miss. To solve the above problem, we serialized the horizontal 2-D data and calculated the halo grid points as a dummy. This serialization is a method having used in NICAM to secure the vector length in vector computers. In our optimization for Fugaku, we found that stencil serialization can easily take advantage of hardware prefetching, long SIMD width, and SWP, rather than using 2-D or 3-D cache tiling, which is a common method to accelerate stencil computation.

Next, in order to make the memory footprint utilized as small as possible, we fused loops manually to reduce the dimensions and sizes of intermediate arrays. Finally, the innermost loop size was split and moved to the outermost loop to enhance the effect of cache blocking. For this optimization, we did not change the size and dimension of the arrays. For thread parallelization, we used the vertical axis of the 3-D grid points as the parallelization axis. However, in the case of loops with data dependency in the vertical direction, thread parallelization needs to be performed on other axes. In this case, we parallelized the horizontal loop by dividing it by the number of threads and fixing the loop range that each thread was responsible for.

For some arrays, even though the size of some dimensions was fixed at compiling time, the optimization considering the size was not applied in some cases. This happens in the case of Fortran modular arrays: even if an allocatable modular array is allocated at runtime with a fixed size, the subroutine that uses the array cannot know the size of the array at compiling time. On the other hand, if a subroutine is called with the array and array size as arguments, optimization can proceed concerning some fixed array size.

10.2.2 Register Spill and Loop Fission Studies

The other of the two optimization tradeoffs is to ensure data locality and avoid register spilling. The optimization described above has combined many loops and reduced the number of intermediate arrays. However, the number of instructions in one loop body has increased, and register spilling had become a frequent problem. To avoid this, we had to split the innermost loop again and create intermediate arrays. We first manually searched for the optimal number of loop divisions. In particular, for the cloud microphysics process, which has thousands of numbers of lines in the main loop, we split the innermost loop into 54. The division enabled software pipelining first, and then solved the register shortage. Further splitting degraded the performance due to memory access waiting. The optimal number of partitions depends on the parameters of each CPU, so if we adopt the Fugaku-specific number of partitions we surveyed here, computational performances on the other supercomputers may degrade. In order to solve this problem in an advanced way, we enhanced the automatic loop partitioning function by the compiler and evaluated it using the NICAM kernel. This is one of the major achievements of Fugaku's system software codesign.

In addition to these, improvements in compiler addressing were also applied, and in some cases, speedup was achieved by using the option of scheduling assuming that the number of registers was 10% larger.

10.2.3 Evaluation of Mixed-precision Calculation

The aggressive utilization of less-precision floating-point numbers is coming to be applied to many climate models and DA systems worldwide [110, 111]. There are two major advantages of using single-precision floating-point numbers for speeding up calculations. The first is that the required Byte/FLOP ratio is reduced, which is expected to speed up computation, especially in the sections that are limited by memory transfer performance; since the algorithms of NICAM have low arithmetic densities and many sections are limited by memory bandwidth, this effect was expected to be significant. The other point is that Fugaku can perform single-precision SIMD operations with up to 16. This can be expected to increase the speed in the sections that are slowed down by waiting for operations or where there is still room for the memory cache busy rate. The average speedup ratio of single-precision calculations to double-precision calculations was about 1.7 times for the six kernels. Kernels with a memory-bounded type were about 1.7 times faster, while kernels with a cache-bounded type achieved a speedup of about 2 times.

To perform simulations using mixed precision, it is necessary to show that the simulation results do not degrade significantly even when using single-precision floating-point numbers. To evaluate the simulation results, we used the baroclinic wave test case, which is widely used to evaluate the performance of the dycore component of the atmospheric model. Experiments were conducted using the K computer on meshes ranging from 224 km to 56 km, and results were compared between single and double precision. 224 km mesh experiments showed little difference

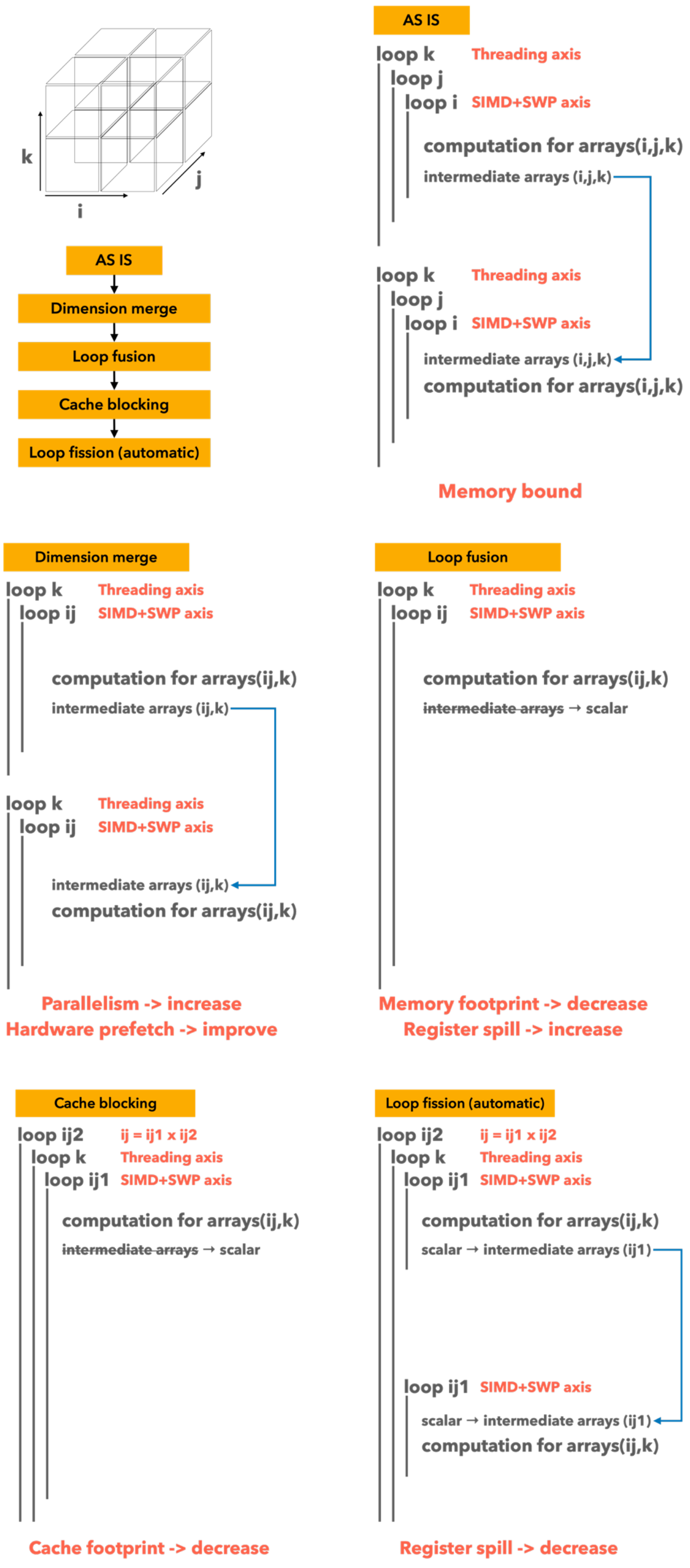

k
i
j
AS IS
Dimension merge
Loop fusion
Cache blocking
Loop fission (automatic)
AS IS
loop k Threading axis
loop j
loop i SIMD+SWP axis
computation for arrays(i,j,k)
intermediate arrays (i,j,k)
loop k Threading axis
loop j
loop i SIMD+SWP axis
intermediate arrays (i,j,k)
computation for arrays(i,j,k)
Memory bound
Dimension merge
loop k Threading axis
loop ij SIMD+SWP axis
computation for arrays(ij,k)
intermediate arrays (ij,k)
loop k Threading axis
loop ij SIMD+SWP axis
intermediate arrays (ij,k)
computation for arrays(ij,k)
Parallelism -> increase
Hardware prefetch -> improve
Loop fusion
loop k Threading axis
loop ij SIMD+SWP axis
computation for arrays(ij,k)
~~intermediate arrays~~ → scalar
Memory footprint -> decrease
Register spill -> increase
Cache blocking
loop ij2 ij = ij1 x ij2
loop k Threading axis
loop ij1 SIMD+SWP axis
computation for arrays(ij,k)
~~intermediate arrays~~ → scalar
Cache footprint -> decrease
Loop fission (automatic)
loop ij2 ij = ij1 x ij2
loop k Threading axis
loop ij1 SIMD+SWP axis
computation for arrays(ij,k)
scalar → intermediate arrays (ij1)
loop ij1 SIMD+SWP axis
scalar → intermediate arrays (ij1)
computation for arrays(ij,k)
Register spill -> decrease

Figure 10.1: The optimization strategies for the typical loop structure of NICAM

between all double-precision and all single-precision results, while 56 km mesh resolution results clearly showed unnatural flow fields. To identify the sections that significantly impact the calculation results, we conducted experiments using double-precision in a partial part of the simulation model. We found that the reason for the degradation of the calculation results was the inaccuracy of the coefficient values used in the stencil operator. The accuracy of the distance calculation between two points on the sphere and the metrics term calculation was affected by the single precision calculation. The calculation for the coefficient preparation is done once only in the initialization part. By keeping the accuracy of the variables used in the calculation at this time high, the results were not degraded even if the coefficients themselves were single precision.

When the experiment was conducted up to the target resolution of 3.5 km horizontal mesh, the difference between the double- and single-precision results of the same model showed enough small value of norms from the reference experiment [112]. Those were smaller than one order of magnitude smaller than the difference between the results for multiple atmospheric models reported in previous studies. However, at the highest resolution of 3.5 km mesh, as the calculation proceeds, the differences between double and single precision results, which were not visible by 7 km, emerge as patterns along the atmospheric disturbance. The results are shown in Figure 10.2. From top to bottom, the results are for the control experiment, the case where the pressure gradient force is calculated with double-precision, the case where the numerical viscosity is calculated with double-precision, and the case where the single-precision rounding filter is applied every time step. To investigate the generation factors of the noise pattern in more detail, the experiments were conducted by partially increasing the calculation sections of the pressure gradient force and numerical viscosity to double precision. However, We found that the simulation accuracy was not improved by doubling the accuracy of these sections. On the other hand, when we reduce the floating-point precision of the time-evolving variable to single-precision at the end of the main loop and then immediately return to double-precision, the simulation result showed larger noise. We concluded that the noise was generated by the fluid dynamics solver while maintaining the mechanical balance and that it was difficult to remove.

We also modified the three main components of the physical process to single-precision, and each of them achieved a speed-up of about 1.6 times. However, there are additional times for casting arrays between the single-precision components and double-precision components. In the atmospheric radiation process, we fixed some scalar variables to double precision because some operations have the "loss of significance" with the combination of exponential calculation and subtraction.

10.2.4 Evaluation of Eigenvalue Solver

In the LEKTF method, we have to obtain the eigenvalues and eigenvector of the real symmetric square matrix. The size of the target matrix is determined by the number of ensemble members, which is quite small in the research field of HPC. However, we have to execute the solver as much as the number of grid points. For this purpose, we used an eigenvalue solver named Kevd [113]. Kevd is lightweight and highly optimized for small matrix sizes. Kevd takes advantage of a particular matrix data layout (AoSoA) for better cache controls. For the Fugaku, it is further improved with new features, including fine-grained thread controls and single-precision floating-point supports, as well as vectorization on SVE. As a result, Kevd has more than 2x better throughput on a single CMG of the Fugaku compared with a single CPU of the K.

10.2.5 Data-centric Design in Application Coupling

Usually, the performances of the weather simulation models and DA systems have been optimized individually. However, we need to maximize the total performance of the DA cycles. In the

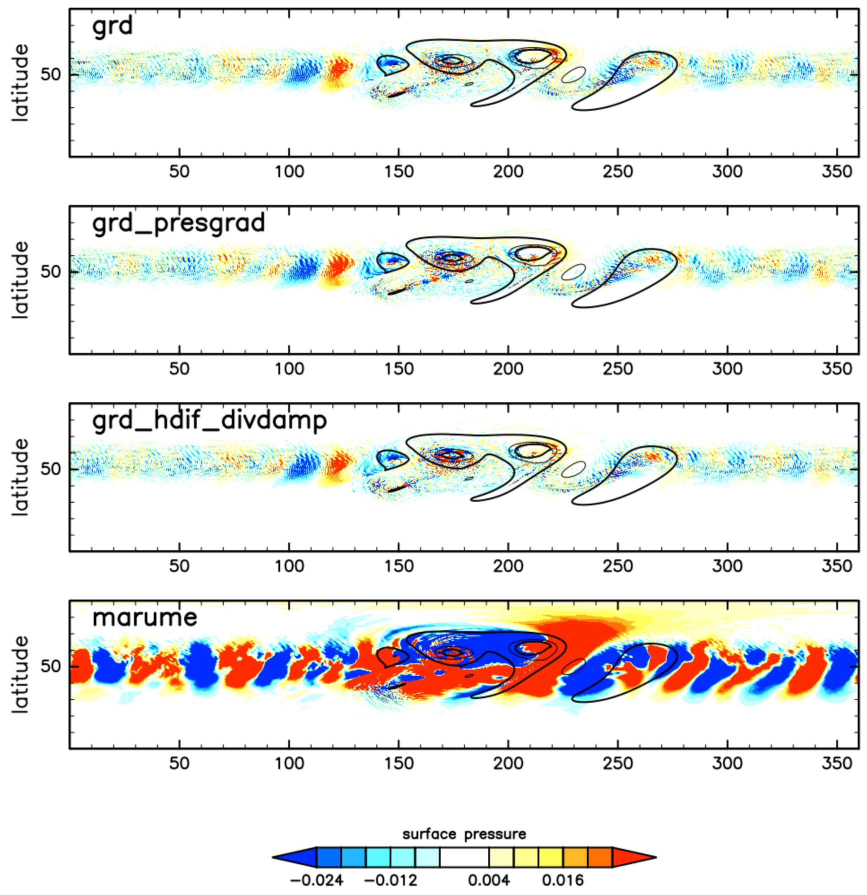


Figure 10.2: Difference between single and double precision experiments in the 3.5km-mesh baroclinic wave test case

simulation part, each ensemble member is allocated to the individual process group. On the other hand, in the DA part, each process requires the output of all ensemble members of the same grid point. Therefore we have to transpose the data between simulation and DA. We focused on file I/O and all-to-all communications. We redesigned the application to maximize the total throughput of reads and writes and reduce data movement as much as possible [114]. In the simulation part, each process independently outputs files to local SSD storage. In the DA part, the processes read them and perform group communication in small groups. In particular, the fully independent and distributed file IO provided sufficient scalability to speed up the total elapsed time. In the DA benchmark results described later, we achieved a data throughput of 0.14TiB/s (0.27GiB/s per node) in the case with 512 nodes and 30TiB/s (0.23GiB/s per node) in the case with 131,072 nodes.

10.3 Estimations and Results

10.3.1 Performance Estimation and Mapping

As mentioned above, meteorological and climate applications have a computing performance feature called "flat-profile". We extracted each characteristic section (subroutine) from dycore as a kernel, optimized it, executed it in a CPU simulator, and predicted its performance. However, these six kernels account for only about 10% of the total calculation time of NICAM. Therefore, we reflected the performance prediction results using the kernel in the entire application, that is, "performance mapping".

In the performance mapping, first, the performance of NICAM was measured using FX100. We have measured the elapsed time, memory transfer rate, and floating-point operations for the entire section of the program code. Next, the measurement sections were subdivided into smaller sections that made it easy to understand what type of calculation was being performed. The subsections are categorized as computation-bound calculations, cache-bound calculations, memory-bound calculations, branch processing, communication, I/O, etc. We fit a roofline model to each computational subsection and evaluated whether the expected computational performance or memory transfer rate was achieved. The results were summarized and prioritized for optimization. In particular, we worked to reduce non-computational operations and latencies -for example, data copy, thread imbalance, and register spill. This is similar to the way we review our household spending in detail and increase our savings. That's why we call this method the "household account method".

What we learned from the detailed measurements was that many of the computational sections were memory-controlled. We have tried to predict the performance of the non-kernelized section by substituting the results of a representative kernel with memory-determining characteristics. Initially, *OPRT3D_divdamp*, a dycore kernel, was used as the representative kernel, but as a result of single precision and optimization, the kernel characteristics changed from memory-controlled to cache-controlled. We evaluated the performance of the six optimized kernels, reselected *horizontal_adv_flux* as the representative kernel, and performed performance mapping according to the following rules.

- The kernelized section itself uses the kernel performance estimation result.
- For non-kernelized sections, use the performance estimation results of the *horizontal_adv_flux* kernel.
- Estimate the execution time of each non-kernelized section using the data transfer amount

as a feature value and the relationship between the data transfer amount of the representative kernel and the execution time.

Using the measured data transfer amount of each section and the performance of the kernel in FX100, the mapped estimated elapsed time and the measured time were compared (The bottom panel of Figure 10.3). The predicted time was about 27% longer than the actual measurement.

This discrepancy was due to the presence of code in the non-kernelized section that was less computationally dense than the representative kernel and resembled the STREAM-Triad benchmark. Such sections are not as fast as the optimized representative kernel, as there is little room for optimization. Therefore, referring to the NICAM source code, we added a *streamlike* kernel with data access and calculation patterns that represent sections with low calculation density, and optimized and measured it. Using this result, the second mapping rule was modified as follows.

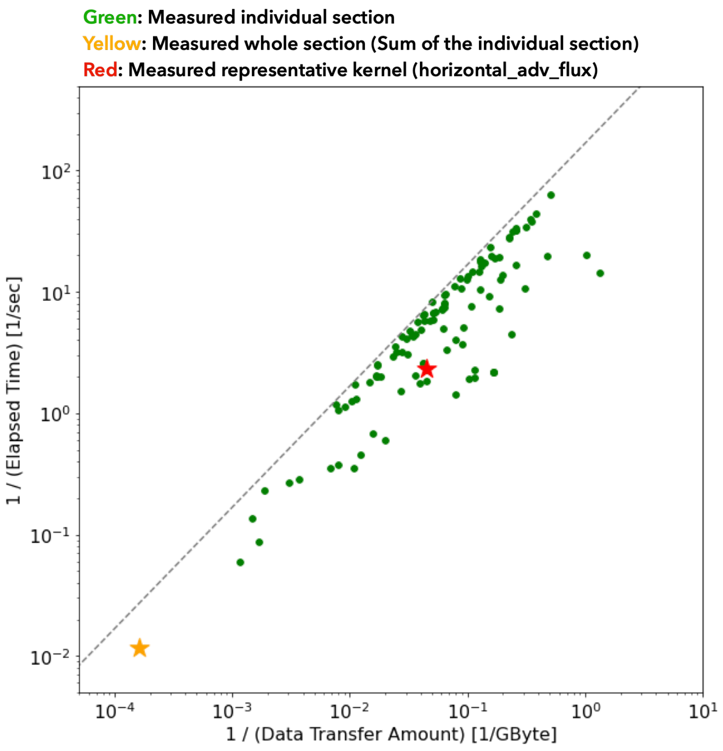
- The non-kerneled section is judged based on the memory transfer performance when measured with double precision on FX100.
- For sections with memory transfer performance of 120GB/s or more, use the performance estimation results of the *streamlike* kernel.
- For sections with memory transfer performance of 120GB/s or less, use the performance estimation results of the *horizontal_adv_flux* kernel.

The reason why 120GB/s was adopted as the threshold value for switching the kernel to be applied is that the calculation density of each subdivided section of NICAM measured on FX100 tended to change at this value. The measurement time estimated using the modified mapping method was 1.7% longer than the actual measurement, and a more accurate estimate was obtained (Figure 10.4).

Table 10.1 shows the transition of NICAM's calculation time estimation breakdown at each evaluation stage. Note that in Phases 3 to 4, the problem size per process increases due to reviewing the number of nodes. The dycore has been gradually accelerated by the result of refactoring reflecting the knowledge of the kernel. The estimation system has also improved due to improvements in performance mapping. The elapsed time of the cloud microphysics component was short in the initial estimates in 2014. We initially assumed that we would halve the number of calls in the substep of the cloud microphysics part. However, in the subsequent study, we considered the simulation accuracy, and the above assumption was discarded for estimation. The elapsed time and amount of computation of the atmospheric radiation part increased due to the sophistication of the component simultaneously as optimizing it. The boundary layer component and other physical processes have significantly increased time in phases 2 to 3. This is a result of improved performance mapping in phase 3, which has improved the accuracy of the predicted time. We overestimated for speed-up until phase 2. The boundary layer component was kernelized from phase 4 and measured directly. As a result of measurement under the condition of simultaneous execution of 256 member ensembles in phase 6, the calculation time increased in some sections, reflecting the different amount of calculation for each member.

10.3.2 Benchmark on Fugaku

Figure 10.5 shows the weak scaling performance evaluation of single member of NICAM on Fugaku. The number of grid points per node is kept constant, and the horizontal resolution is changed from 56km to 28km, 14km, 7km, and 3.5km. At this time, the number of using nodes was increased by 4 times from 2 nodes (8 processes) to 512 nodes (2048 processes). The



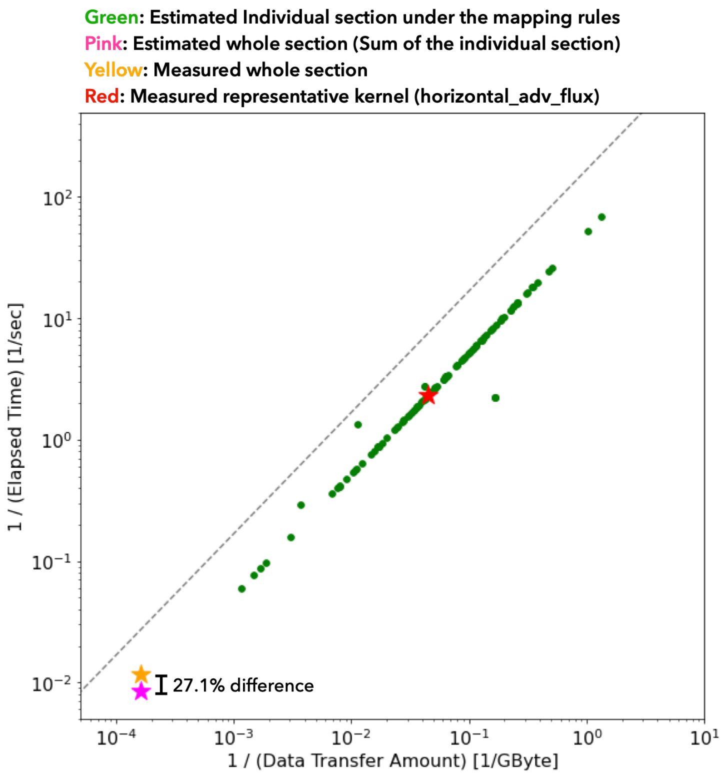


Figure 10.3: Scatterplot of the elapsed time of NICAM against the measured data transfer amount

	on K	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	Phase 6
# of time steps	2,700	2,700	2,700	2,700	2,700	2,700	2,700
# of grid point per PE	16,900	16,900	16,900	16,900	21,780	21,780	21,780
# of ens. members per set	32	256	256	256	256	256	256
# of sets	32	4	4	4	4	4	4
Elapsed time [sec]							
Dycore - Pre_Post	10,661			230	201	188	231
Dycore - Large_Step	40,475	7,702	5,194	595	548	566	845
Dycore - Small_Step	105,979			2,470	1,884	2,173	2,362
Dycore - Tracer_Advection	52,338	2,143	1,548	1,296	1,040	933	1,073
Phys - Cloud_Microphysics	35,887	771	816	1,557	1,354	1,483	1,257
Phys - Radiation	38,705	1,249	1,354	1,605	1,591	1,100	1,158
Phys - Boundary_Layer	31,152	2,779	232	1,133	754	728	706
Phys - Others	60,666		470	623	395	926	1,449
Dycore total	209,454	9,845	6,742	4,592	3,673	3,860	4,511
Physics part total	1,66,409	4,799	2,872	4,918	4,094	4,237	4,570
MPI communications	1,243	283	291	114	229	116	-
NICAM Total	377,106	14,927	9,905	9,510	7,995	8,213	9,081

Table 10.1: Change of NICAM elapsed time estimation result

time interval was 6 seconds in all experiments, and 50-step calculations were performed with double- and mixed-precision, respectively. The results show good weak scaling performance in both two precision settings. The calculation part is almost constant on the node-averaged elapsed time. In the communication section, the elapsed time tended to increase as the number of nodes increased. This factor is due to the imbalance of both communication and calculation time between nodes. The amount of calculation for, such as cloud phase changes, varies greatly depending on the point on the Earth's atmosphere. As a result of increasing the horizontal resolution in weak scaling, the contrast between the grid point group with many clouds (heavy computational load) and that with few clouds (low computational load) makes a large time imbalance. The time for file I/O scaled very well. In this experiment, we used local SSDs, a first-tier file system. When we used a second-tier lustre-based file system configured with HDD, a large variation in I/O time occurred between nodes, and a significant waiting time appeared in communication. Although not shown here, we also evaluated the time when 256 members are executed simultaneously. Since all members are calculated from a different initial value, the computation time varies. Therefore, the elapsed time increased by about 10% compared to that for single-member.

Figure 10.6 shows the performance evaluation results of the DA part, including LETKF. We calculated with different ensemble sizes, horizontal resolutions, and floating-point precisions. When the ensemble size is increased, the amount of computation per process increases. In particular, the computation amount of eigenvalue decomposition performed on the array of $\langle$ number of ensemble members $\rangle \times \langle$ number of ensemble members $\rangle$ increases. This is also clear in the breakdown of the elapsed time in the case of the 14km mesh (Fig. 10.7)). When we use double-precision, the elapsed time of the eigenvalue calculation part increases non-linearly with the increase in the number of ensemble members. On the other hand, when we use mixed-precision,

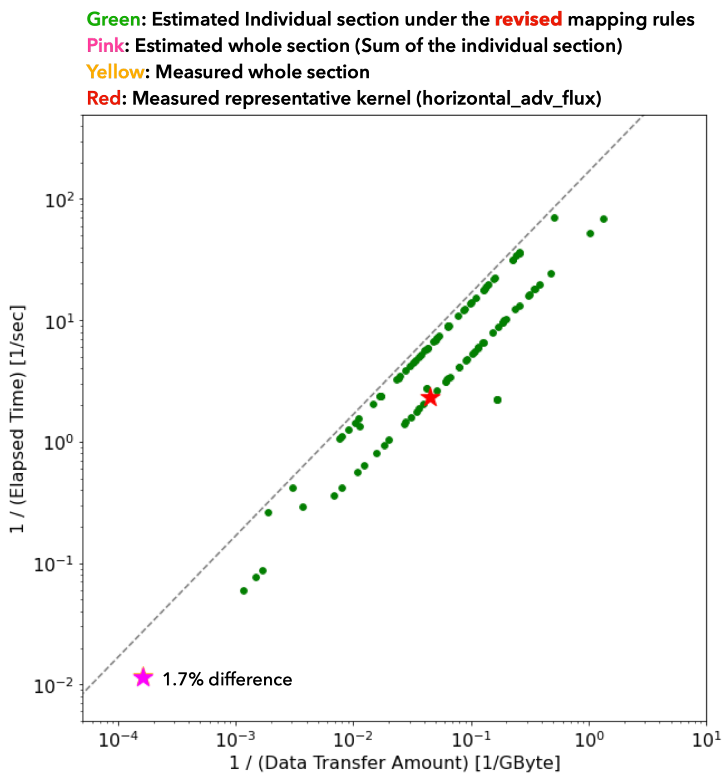


Figure 10.4: Scatterplot of the elapsed time of NICAM against the measured data transfer amount (cont.)

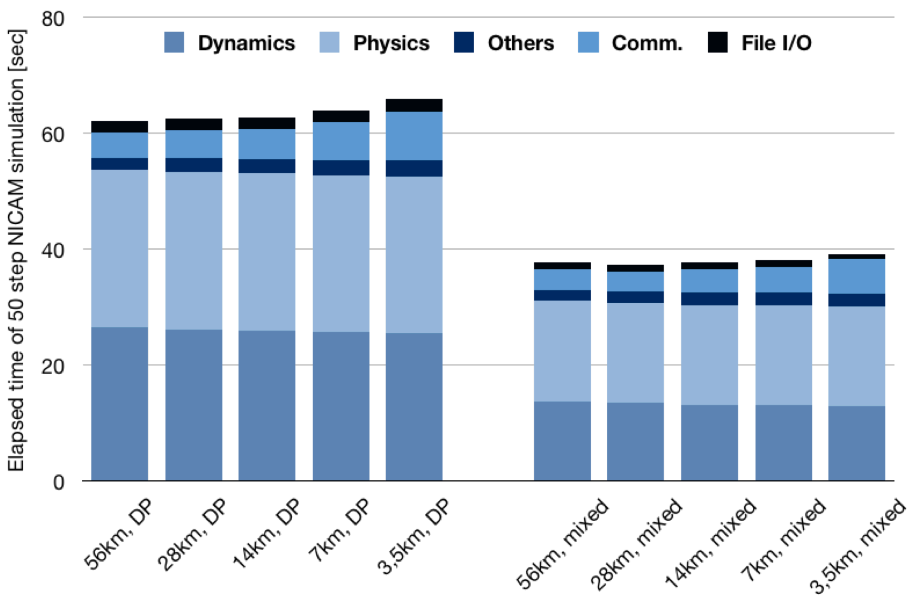


Figure 10.5: Weak scaling test result of NICAM single run on Fugaku

the eigenvalue calculation is performed in single precision, and the elapsed time increases almost linearly with increasing ensemble size. The factor that suppressed the increase in elapsed time is the effect of deflation when using single-precision calculation. When increasing the horizontal resolution to 56km, 14km, 3.5km, we increased the number of nodes so that the number of grid points per process was the same. Therefore, for the same ensemble size, the difference in horizontal resolution is related to weak scaling. In this case, the growth of elapsed time was small, and the results showed good scaling performance.

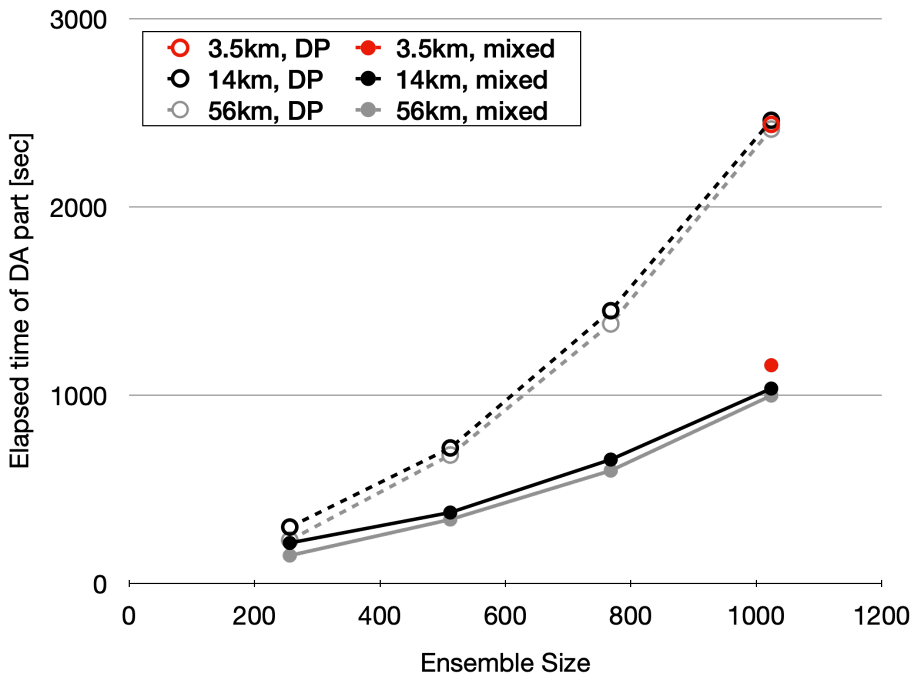


Figure 10.6: Benchmark results of DA part on Fugaku

Of particular note is the IO time. In Figure 10.6, we observed no delay even when the used nodes were increased from 512 to 131072 by changing the horizontal resolution. In Figure 10.7, there is a slight increase in IO time for an increase in ensemble size at the same horizontal resolution. Short I/O time in all cases results from achieving scalable file I/O with the redesigned NICAM+LETKF workflow. In this data-centric workflow, file I/O can be performed using only the local SSD, significantly improving data throughput.

The following procedure carried out the elapsed time estimation of the target problem on Fugaku.

- Estimate the elapsed time of 480 DA cycles from the execution time for one DA cycle with mixed-precision.
- Estimate or measure the execution time of the simulation part and DA part individually and total them.
- For the simulation part, we run 256 members simultaneously as one set and use 512 nodes per member, for a total of 131072 nodes. However, we scale the results by reducing the number of sets to measure from 4 to 1 and the number of time steps from 2700 to 50 steps.
- For the DA part, we measure the total time using 131072 nodes. Both parts' elapsed time includes file I/O time and uses the tier-1 file system (local SSD).

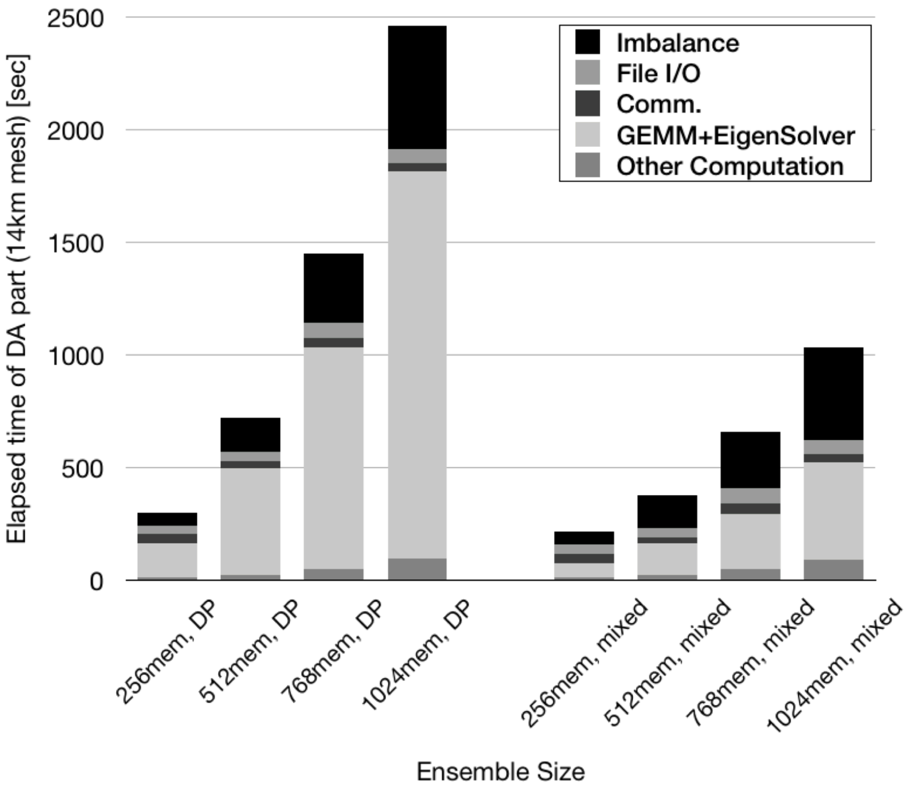


Figure 10.7: Benchmark results of DA part on Fugaku for the 14km-mesh case

- The execution mode of Fugaku is set to boost-mode and non-eco-mode, and the node mapping pattern is not specified in any measurement.

The achieved performance is summerized in Table 10.2. We used 82% of the total node of Fugaku for the execution. Total time of one DA cycle is 2.8 hours, and the estimated time of 480 DA cycle is 57 days, which is x127 faster than the time estimated on the K computer.

Table 10.2: Achieved performance on Fugaku

	Elapsed Time [sec]
One DA Cycle total	10,239
Simulation part total (estimated from set1 x4)	9,081
NICAM set1 (estimated from short time steps)	2,270
DA part total	1,158
StoO	196
LETKF	961

In addition, as an additional experiment, we further extended the time interval between simulation steps from the target problem to 8 seconds and calculated up to 450 steps for the one set. We achieved 29 PFLOPS and 79 PFLOPS of effective performance for the simulation and the DA part, corresponding to 7% and 18% of the peak performance, respectively. The paper summarizing the results of this benchmark was selected as a finalist for the 2020 ACM Gordon Bell Awards [115].

10.3.3 Summary

We have achieved more than x100 speedup of ensemble weather data assimilation on Fugaku through the codesign in FS2020 project. This codesign activity shows that weather/climate applications have specific performance characteristics named ”flat-profile” and need more improvement by focusing on the data movement. Finally, we would like to thank all the people who were involved in the NICAM+LETKF working group of the project.

10.4 Author/Co–Authors

10.4.0.0.1 Author

- Hisashi Yashiro (National Institute for Environmental Studies)

10.4.0.0.2 Co–Authors (in alphabet order)

- Yuta Kawai (RIKEN)
- Chihiro Kodama (JAMSTEC)
- Kazunori Mikami (RIKEN)
- Kazuo Minami (RIKEN)
- Takemasa Miyoshi (RIKEN)
- Masuo Nakano (JAMSTEC)
- Koji Terasaki (RIKEN)
- Hirofumi Tomita (RIKEN)

References

[98] Hisashi Yashiro et al. “Performance Analysis and Optimization of Nonhydrostatic ICosahedral Atmospheric Model (NICAM) on the K Computer and TSUBAME2.5”. In: New York, New York, USA: ACM, June 2016, pp. 1–8. ISBN: 9781450341264. DOI: 10.1145/2929908.2929911.

[99] Haohuan Fu et al. “Redesigning CAM-SE for peta-scale climate modeling performance and ultra-high resolution on Sunway TaihuLight”. In: (Jan. 2017), pp. 1–12. DOI: 10.1145/3126908.3126909.

[100] Bryan N Lawrence et al. “Crossing the chasm: how to develop weather and climate models for next generation computers?” In: *Geoscientific Model Development* 11.5 (2018), pp. 1799–1821. ISSN: 1991-959X. DOI: 10.5194/gmd-11-1799-2018.

[101] Koji Terasaki, Masahiro Sawada, and Takemasa Miyoshi. "Local Ensemble Transform Kalman Filter Experiments with the Nonhydrostatic Icosahedral Atmospheric Model NICAM". In: *SOLA* 11.0 (Jan. 2015), pp. 23–26. ISSN: 1349-6476. DOI: 10.2151/sola.2015-006.

[102] Hirofumi Tomita and Masaki Satoh. "A new dynamical framework of nonhydrostatic global model using the icosahedral grid". In: *Fluid Dynamics Research* 34.6 (Jan. 2004), pp. 357–400. ISSN: 0169-5983. DOI: 10.1016/j.fluiddyn.2004.03.003.

[103] M Satoh et al. "Nonhydrostatic icosahedral atmospheric model (NICAM) for global cloud resolving simulations". In: *Journal of Computational Physics* 227.7 (Jan. 2008), pp. 3486–3514. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2007.02.006.

[104] Masaki Satoh et al. "The Non-hydrostatic Icosahedral Atmospheric Model: description and development". In: *Progress in Earth and Planetary Science* 1.1 (Oct. 2014), p. 18. DOI: 10.1186/s40645-014-0018-1.

[105] Brian R Hunt, Eric J Kostelich, and Istvan Szunyogh. "Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter". In: *Physica D: Nonlinear Phenomena* 230.1-2 (Jan. 2007), pp. 112–126. ISSN: 0167-2789. DOI: 10.1016/j.physd.2006.11.008.

[106] Takemasa Miyoshi and Shozo Yamane. "Local Ensemble Transform Kalman Filtering with an AGCM at a T159/L48 Resolution". In: *Monthly Weather Review* 135.11 (Jan. 2007), pp. 3841–3861. ISSN: 0027-0644. DOI: 10.1175/2007mwr1873.1.

[107] Takemasa Miyoshi, Yoshiaki Sato, and Takashi Kadowaki. "Ensemble Kalman Filter and 4D-Var Intercomparison with the Japanese Operational Global Analysis and Prediction System". In: *Monthly Weather Review* 138.7 (Jan. 2010), pp. 2846–2866. ISSN: 0027-0644. DOI: 10.1175/2010mwr3209.1.

[108] Yoshiaki Miyamoto et al. "Deep moist atmospheric convection in a subkilometer global simulation". In: *Geophysical Research Letters* 40.18 (Sept. 2013), pp. 4922–4926. ISSN: 0094-8276. DOI: 10.1002/grl.50944.

[109] Takemasa Miyoshi, Keiichi Kondo, and Toshiyuki Imamura. "The 10,240-member ensemble Kalman filtering with an intermediate AGCM: 10240-MEMBER ENKF WITH AN AGCM". In: *Geophysical Research Letters* 41.14 (July 2014), pp. 5264–5271. ISSN: 0094-8276. DOI: 10.1002/2014gl060863.

[110] Peter D Düben, Hugh McNamara, and T N Palmer. "The use of imprecise processing to improve accuracy in weather & climate prediction". In: *Journal of Computational Physics* 271 (2014), pp. 2–18. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2013.10.042.

[111] Sam Hatfield et al. "Improving Weather Forecast Skill through Reduced-Precision Data Assimilation". In: *Monthly Weather Review* 146.1 (2018), pp. 49–62. ISSN: 0027-0644. DOI: 10.1175/mwr-d-17-0132.1.

[112] Masuo Nakano et al. "Single Precision in the Dynamical Core of a Nonhydrostatic Global Atmospheric Model: Evaluation Using a Baroclinic Wave Test Case". In: *Monthly Weather Review* 146.2 (Jan. 2018), pp. 409–416. ISSN: 0027-0644. DOI: 10.1175/mwr-d-17-0257.1.

[113] Shuhei Kudo and Toshiyuki Imamura. "Cache-Efficient Implementation and Batching of Tridiagonalization on Manycore CPUs". In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia 2019. Guangzhou, China: Association for Computing Machinery, 2019, pp. 71–80. ISBN: 9781450366328. DOI: 10.1145/3293320.3293329.

[114] Hisashi Yashiro et al. "Performance evaluation of a throughput-aware framework for ensemble data assimilation: the case of NICAM-LETKF". In: *Geoscientific Model Development* 9.7 (Jan. 2016), pp. 2293–2300. ISSN: 1991-959X. DOI: 10.5194/gmd-9-2293-2016.

[115] Hisashi Yashiro et al. "A 1024-Member Ensemble Data Assimilation with 3.5-Km Mesh Global Weather Simulations". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), pp. 1–10.

Chapter 11

Codesign of NTChem

In Priority Issue 5, there were three research branches driving the research activities toward the goal of "Development of new fundamental technologies for high-efficiency energy creation, conversion/storage and use". Priority Issue 5 chose NTChem/RI-MP2 module as the codesign platform. NTChem is actively developed in order to be used for the solar battery simulator and/or the chemical reaction simulator in the condensed phases. Such simulations require the calculation based on the highly accurate quantitative electronic interaction theory. NTChem/RI-MP2 module employs the 2nd order Møller-Plesset (MP2) pertubation method based on the Resolution-of-Identity (RI) approximation, which was the economic and optimized algorithm for the K computer when starting codesign, and was regarded as the appropriate choice for codesign. The codesign contribution from NTChem/RI-MP2 to Fugaku is expected from the viewpoint of highly dense computing coupled with heavy communication. This chapter describes the contents of codesign in NTChem/RI-MP2 from computational science perspective.

11.1 NTChem/RI-MP2 overview

NTChem [116] is a general purpose software package for computing the molecular electronic structure. It has been developed by R-CCS and related institutes with strong attention to state of the art supercomputer such as Fugaku. NTChem enables the ab initio computational studies for large and complicated molecular systems. Some of the major computational methodologies implemented in NTChem includes:

- Calculation of molecular electron structure in ground state and excited state, based on the Hartree-Fock (HF) method and density functional theory (DFT).
- Highly accurate calculation of the electron structure in ground state and excited states, based on the Møller-Plesset 2nd order pertubation (MP2) method, the coupled cluster (CC) method, the quantum Monte Carlo (QMC) method.

In Priority Issue 5, NTChem was actively developed in order to be used for the solar battery simulator and the chemical reaction simulator in the condensed phases. Such simulations require the calculation based on the highly accurate quantitative electronic correlation theory. Assuming the number of atmic orbitals as N, the computational cost of the total calculation becomes $O(N^5)$ or more, which is very expensive. With this background, Priority Issue 5 pours the effort to implement the sofisticated high precision electronic correlation theory, and to codesign Fugaku using NTChem as the performance evaluation.

Priority Issue 5 chose NTChem/RI-MP2 module, i.e. Resolution-of-Identity MP2 (RI-MP2) method, as the codesign platform. NTChem/RI-MP2 module employed the most economic and optimized algorithm on the K computer at the time, and was regarded as the appropriate choice for evaluating the electron-correlation effect in large size molecular systems.

NTChem/RI-MP2 module executes matrix element computations and dense matrix computations such as matrix-matrix multiplication, and its computing density is high, i.e. low Byte/Flop (B/F) ratio in total. It utilizes systems linear algebra library. NTChem/RI-MP2 requires frequent MPI collective data communication and point-to-point data communication among processes to update matrix data. The communicatin data volume is fairly large.

The codesign contribution from NTChem/RI-MP2 is expected from these perspectives, i.e. highly dense computing coupled with heavy communication. The actual codesign activity is lead by the NTChem/RI-MP2 working group (NTChem WG) composed of the members from the Priority Issue 5 organizations as well as the members from Riken R-CCS and from Fujitsu.

The main contents of the codesign in NTChem/RI-MP2 from computational science point of view is described in the following sections.

11.2 NTChem/RI-MP2 baseline code

The version of NTChem/RI-MP2 module before starting codesign is reffered to as the baseline code. The major computing routines and the communication routines in baseline code are listed in Table 11.1 and Table 11.2.

Table 11.1: major computing routines in baseline code

routines	brief description	computing cost
RIMDInt3c_calc	3-center integration (McD kernels)	$O(N^3)$
MOInt3c_tran_1	3-center integration 1st transformation (calls DGEMM)	$O(N^4)$
MOInt3c_tran_2	3-center integration 2nd transformation (calls DGEMM)	$O(N^4)$
AOInt2c_calc	2-center integration (McD kernels)	$O(N^2)$
Inv2c_DPOTRF	2-center int. Cholesky decomposition (calls DPOTRF)	$O(N^3)$
Inv2c_DTRTRI	2-center int. Inverse upper triangle (calls DTRTRI)	$O(N^3)$
MOInt3c_tran_3	3-center integration 3rd transformation (calls DGEMM)	$O(N^4)$
MOInt4c_calc	4-center integration (calls DGEMM)	$O(N^5)$
MP2Energy_calc	MP2 correlation energy (global summ kernel)	$O(N^4)$

Table 11.2: major communication routines in baseline code

section	brief description	data length	# of calls
MO_comm	MPI_Bcast in molecular orbital coefficient matrix	$O(N^2)$	1
RIMDInt3c_comm	MPI_Allreduce in 3-center integration	$O(N^3/N_{\mathbf{procsMO}})$	$N_{\mathbf{procsMO}}$
MOInt3c2_comm	MPI_Allreduce in 3-center transformation	$O(N^3/N_{\mathbf{procsMO}})$	$N_{\mathbf{procsMO}}$
NBF_RI_rank_comm	MPI_Allgather in 3-center shell aux. basis function	$O(N)$	1
MOInt3c_comm_1a	MPI_ISend,MPI_IRecv in 3-center integration	$O(N^3/N_{\mathbf{procsMO}})$	$N_{\mathbf{procsMO}}$
MOInt3c_comm_1b	MPI_ISend,MPI_IRecv in 3-center integration	$O(N^3/N_{\mathbf{procsMO}})$	$N_{\mathbf{procsMO}}$
MOInt3c_comm_2	MPI_ISend,MPI_IRecv in 3-center integration	$O(N^3)$	$N_{\mathbf{procsMO}}$
MOInt3c_comm_3	MPI_ISend,MPI_IRecv in 3-center integration	$O(N^3)$	$N_{\mathbf{procsMO}}/2$
MOInt4c_comm	MPI_Allreduce in 4-center integration	$O(N^4/N_{\mathbf{procsMO}})$	$N_{\mathbf{procsMO}}/2$
MP2Energy_comm	MPI_Allreduce in MP2 correlation energy	1	1

11.3 massively parallel implementation of RI-MP2 calculation

The original parallel hybrid algorithm of MPI and OpenMP applied to NTChem, scaled well up to few thousands of processors on the K computer. Its parallel implementation was to divide the index of virtual molecular orbitals for the four-center electron repulsion integration into MPI process space [117]. The number of MPI processes for dividing the virtual molecular orbitals is lableded $N_{\mathbf{procsMO}}$ hereafter. This algorithm shows better scalability than the other one which divides the index of occupied molecular orbitals. However, the degree of parallelism $N_{\mathbf{procsMO}}$ is limited by the number of virtual molecular orbitals in single dimension, and can not be extended. Increasing the degree of parallelism to more than tens of thoughsands of processes is needed to achieve the highly parallel computation of the large molecular systems.

In order to extend the degree of parallelism, we developed a new algorithm that adopted two-dimensional layered parallel structure using the local MPI communicator [118]. The essence of this algorithm is to apply the parallel processing in two different parameter spaces. In addition to dividing the outmost loop of virtual orbital index as the first parallel axis, we added the second parallel axis in the matrix data space, so that the integration and matrix computation inside of the divided orbital loop would maintain the second level parallelism. Figures 11.1 and 11.2 show the scheme.

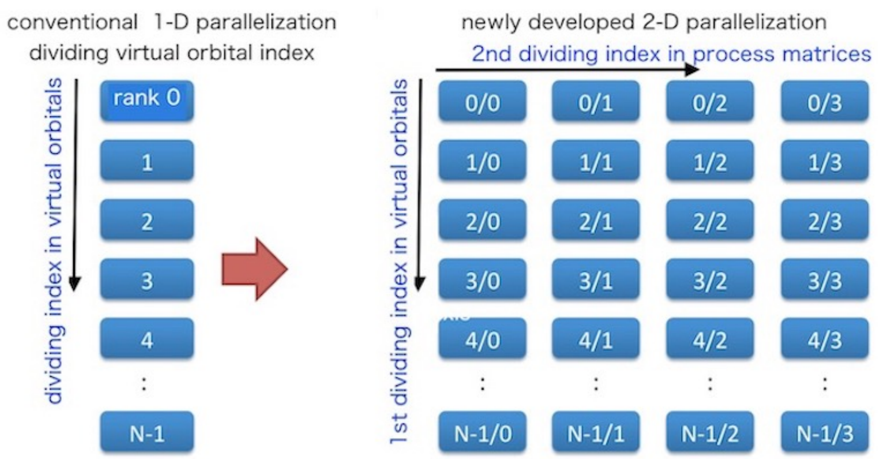
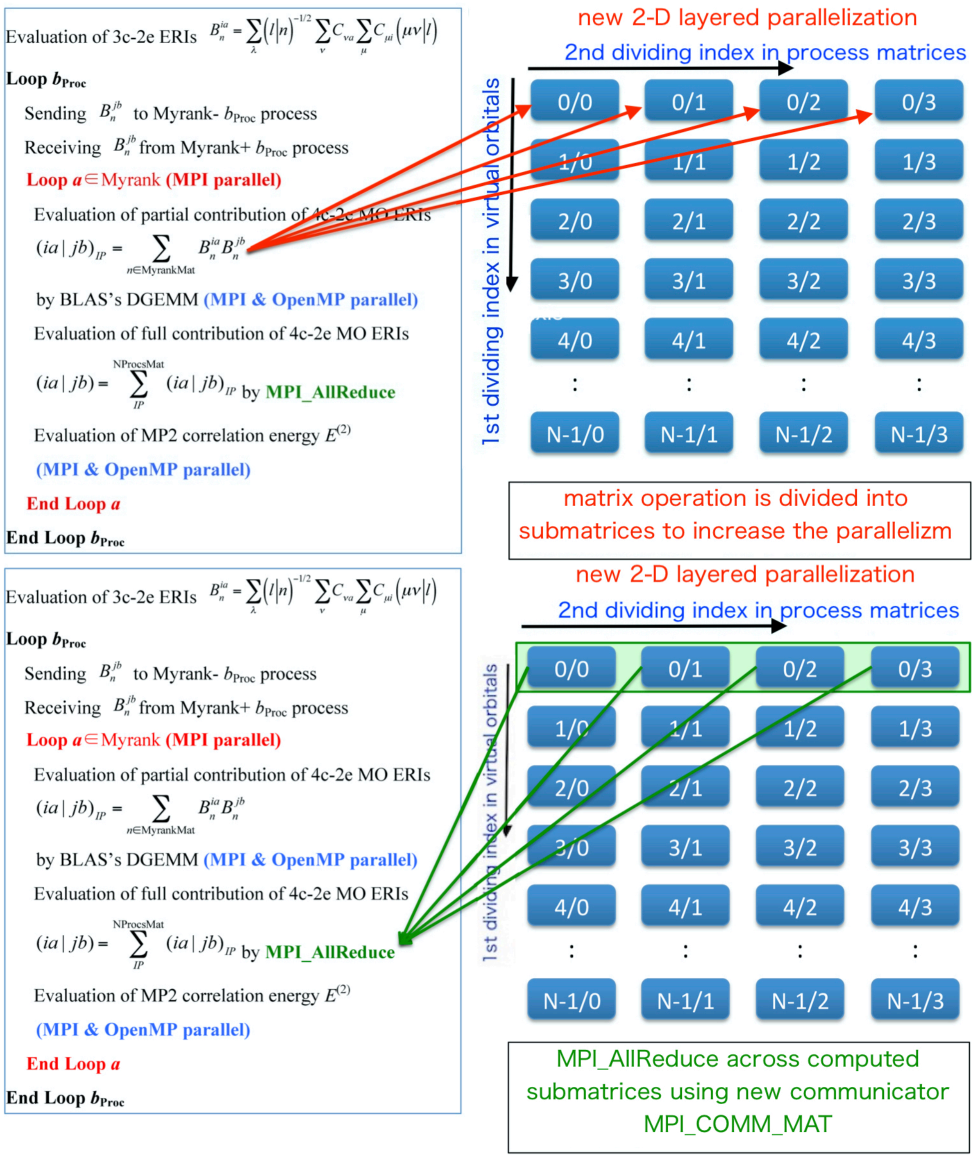

conventional 1-D parallelization
dividing virtual orbital index
dividing index in virtual orbitals
rank 0
1
2
3
4
:
N-1
newly developed 2-D parallelization
2nd dividing index in process matrices
1st dividing index in virtual orbitals
0/0
0/1
0/2
0/3
1/0
1/1
1/2
1/3
2/0
2/1
2/2
2/3
3/0
3/1
3/2
3/3
4/0
4/1
4/2
4/3
N-1/0
N-1/1
N-1/2
N-1/3

Figure 11.1: conventional 1-D v.s. new 2-D layered parallellization

In this development of two-dimensional layered parallel algorithm, we also modified the existing communication pattern in the three-center integration, which resulted in the improvement of efficiency in point-to-point (P2P) communications. Previous implementation of P2P communication in the three-center integration was to communicate from an originating computing process to all the related processes that required the data from the original process. While conducting the large size test jobs on the K computer, it became apparent that the above P2P implementation suffered from significant communication performance degradation on Torus topology network, because of the different number of hops among processes. After the detail study of communicating data movement, we developed the ring communication. The ring communication deploys the progressive data communication, basically send/receive the communicating data in P2P pair wise and then form the next P2P pair toward the sending neighbour in ring manner. These parallel algorithms are illustrated in Figure 11.3.

To verify the performance effect of this ring communication algorithm, we conducted the benchmark test of the RI-MP2/cc-pVTZ calculation using a graphene dimer $(C_{96}H_{24})_2$ on the K computer using 2048 nodes. The benchmark data was consisted of 240 atmos, 6432 atomic



```
Evaluation of 3c-2e ERIs  $B_n^{ia} = \sum_\lambda (l|n)^{-1/2} \sum_\nu C_{\nu a} \sum_\mu C_{\mu i} (\mu\nu|l)$
Loop $b_{\mathrm{Proc}}$
  Sending $B_n^{jb}$ to Myrank- $b_{\mathrm{Proc}}$ process
  Receiving $B_n^{jb}$ from Myrank+ $b_{\mathrm{Proc}}$ process
  Loop $a \in$ Myrank (MPI parallel)
    Evaluation of partial contribution of 4c-2e MO ERIs
    $(ia|jb)_{IP} = \sum_{n \in \mathrm{MyrankMat}} B_n^{ia} B_n^{jb}$
    by BLAS's DGEMM (MPI & OpenMP parallel)
    Evaluation of full contribution of 4c-2e MO ERIs
    $(ia|jb) = \sum_{IP}^{\mathrm{NProcsMat}} (ia|jb)_{IP}$ by MPI_AllReduce
    Evaluation of MP2 correlation energy $E^{(2)}$
    (MPI & OpenMP parallel)
  End Loop $a$
End Loop $b_{\mathrm{Proc}}$
```

```
Evaluation of 3c-2e ERIs  $B_n^{ia} = \sum_\lambda (l|n)^{-1/2} \sum_\nu C_{\nu a} \sum_\mu C_{\mu i} (\mu\nu|l)$
Loop $b_{\mathrm{Proc}}$
  Sending $B_n^{jb}$ to Myrank- $b_{\mathrm{Proc}}$ process
  Receiving $B_n^{jb}$ from Myrank+ $b_{\mathrm{Proc}}$ process
  Loop $a \in$ Myrank (MPI parallel)
    Evaluation of partial contribution of 4c-2e MO ERIs
    $(ia|jb)_{IP} = \sum_{n \in \mathrm{MyrankMat}} B_n^{ia} B_n^{jb}$
    by BLAS's DGEMM (MPI & OpenMP parallel)
    Evaluation of full contribution of 4c-2e MO ERIs
    $(ia|jb) = \sum_{IP}^{\mathrm{NProcsMat}} (ia|jb)_{IP}$ by MPI_AllReduce
    Evaluation of MP2 correlation energy $E^{(2)}$
    (MPI & OpenMP parallel)
  End Loop $a$
End Loop $b_{\mathrm{Proc}}$
```

new 2-D layered parallelization
2nd dividing index in process matrices
1st dividing index in virtual orbitals
0/0 0/1 0/2 0/3
1/0 1/1 1/2 1/3
2/0 2/1 2/2 2/3
3/0 3/1 3/2 3/3
4/0 4/1 4/2 4/3
: : : :
N-1/0 N-1/1 N-1/2 N-1/3
matrix operation is divided into submatrices to increase the parallelizm
new 2-D layered parallelization
2nd dividing index in process matrices
1st dividing index in virtual orbitals
MPI_AllReduce across computed submatrices using new communicator MPI_COMM_MAT

Figure 11.2: blocked matrix data distribution and communication pattern in new 2-D layered parallel

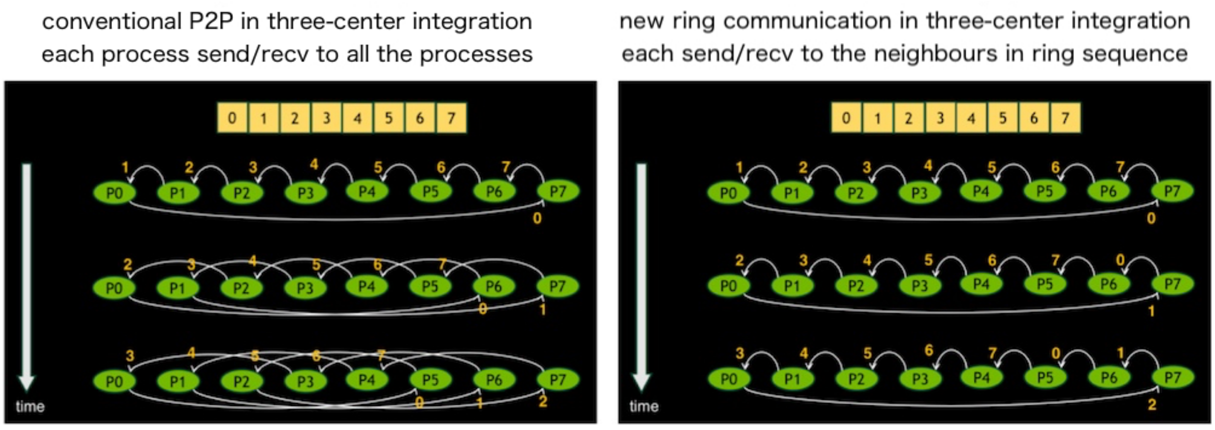

conventional P2P in three-center integration
each process send/recv to all the processes
new ring communication in three-center integration
each send/recv to the neighbours in ring sequence
0 1 2 3 4 5 6 7
P0 P1 P2 P3 P4 P5 P6 P7
time

Figure 11.3: P2P communication in three-center integration, conventional all pairs versus newly developed progressive ring pairs

orbitals, 600 occupied orbitals, 5832 virtual orbitals, 16992 auxiliary basis functions. In order to isolate the effect of ring communication from other effects, the test was run using the previous 1-D index dividing communication algorithm, The result of the benchmark test is shown in Figure 11.4, with the corresponding section colored red. Compared to the conventional all pairs, the new ring communication reduced the wait time in the three-center integration to half. This ring communication was incorporated into the new two-dimensional layered parallel algorithm.

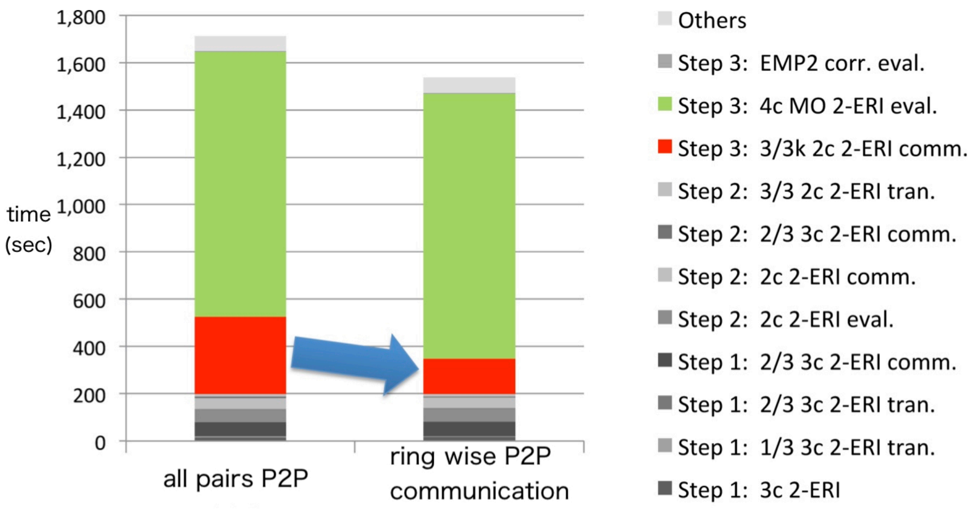


Figure 11.4: communication wait time reduction in three-center integration

Next, to verify the total performance effect of the two-dimensional layered parallelization algorithm, we conducted the benchmark test of RI-MP2/cc-pVTZ calculation using the same graphene dimer $(C_{96}H_{24})_2$ with varied number of MPI processes over the first axis and the second axis. The test results are shown in Table 11.3. The value of $N_{\mathbf{procsMat}}$ shows the number of MPI processes in the second axis. The leftmost column of $N_{\mathbf{procsMat}}=1$ corresponds to the previous algorithm, and it shows the limited scalability only upto 2048 processes. By increasing $N_{\mathbf{procsMat}}$ from 1 to 16, the strong scalability was improved, and the effectiveness of this two-dimensional layered parallel algorithm became evident.

The breakdown of the computing time and the communication time for this benchmark test using 2048 nodes is shown in Figure 11.5. By increasing the value of $N_{\mathbf{procsMat}}$, i.e. the number

Table 11.3: execution time on the K computer for a graphene dimer $(C_{96}H_{24})_2$

	execution time (sec) for $N_{\mathbf{procsMat}}$ value of				
number of nodes	1	2	4	8	16
1024	2366	2289	2211	2346	-
2048	1491	1292	1291	1315	1581
4096	1451	856	743	773	895
8192		826	532	490	541
16384			543	366	391
32768				359	298
65536					282

of MPI processes in the second axis, the time to compute the four-center integration, which is the hot spot and colored green, is reduced. The communication wait time in the three-center integration, which is colored red, is also reduced since the number of processes and the total amount of communicated data in the three-center integration is reduced. On the other hand, the total communicated data size for the matrix block collective communication within the local MPI communicator increases according to $N_{\mathbf{procsMat}}$. And the wait time for the local communicator in the second axis becomes longer, which is colored blue. Figure 11.5 indicates that the balance of computing time and communication time depends on the number of processes in the first axis and the second axis, and that the appropriate parallel layout and distribution should be studied for achieving the highly efficient massively parallel computation.

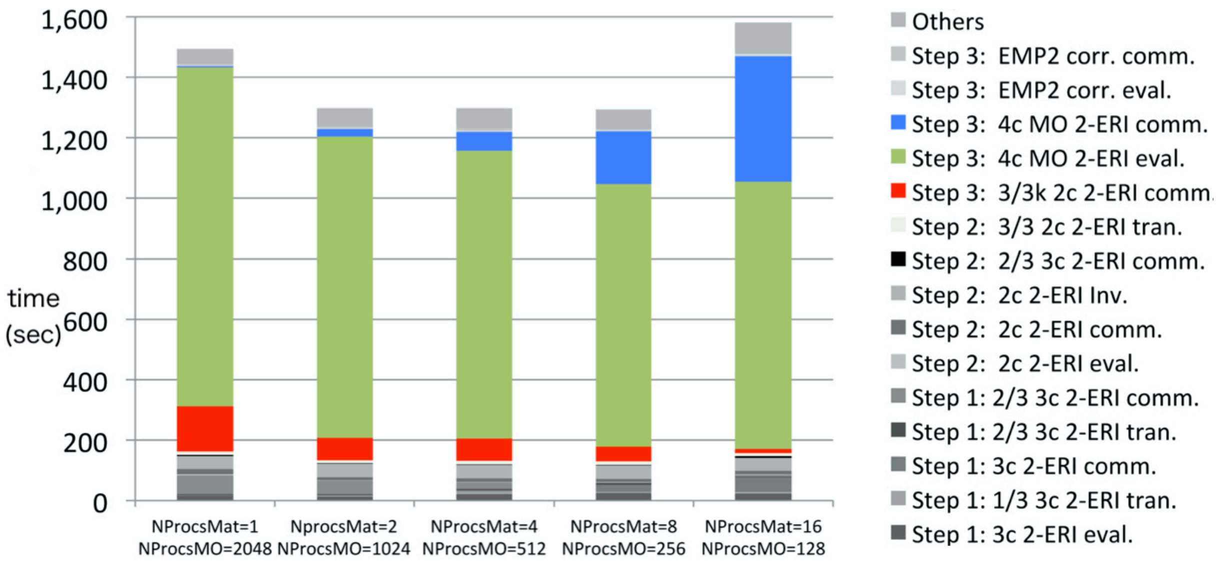


Figure 11.5: breakdown of the execution time for 2048 node jobs on the K computer for a graphene dimer $(C_{96}H_{24})_2$

The benchmark test results for a larger molecule model are also shown as a reference. A graphene dimer $(C_{150}H_{30})_2$ model with 360 atoms and 9840 atomic orbitals is tested on the K computer using from 8911 to 80199 compute nodes, whose results are shown in Table 11.4 . The RI-MP2/cc-pVTZ tests using this new two-dimensional layered parallel algorithm gained speed up well beyond 8911 processes. Since the number of virtual orbitals for this graphene dimer $(C_{150}H_{30})_2$ model is 8910, the original single axis parallel algorithm would not show any speed up beyond 8910 processes, for comparison.

Table 11.4: execution time on the K computer for a graphene dimer $(C_{150}H_{30})_2$

number of nodes	number of cores	execution time (sec)	Peta Flops	peak/sustained ratio (%)
8911	71288	2692	0.7	62
17822	142576	1634	1.2	54
35644	285152	1095	2.0	44
62377	499016	817	2.8	36
80199	641592	759	3.1	30

11.4 memory data distribution - phase 1 implementation

The actual code with the massively parallel implementation explained in previous 11.3 adopted the memory data layout that utilize the matrix data replica.

As we made progress in macroscopic codesign and code refactoring, both the accurate estimated value of memory requirement to run the target job and the actually available memory size became clear. Since the actual orbital data to run the target job was not available at the time, the estimation was based on the theoretical study and the extrapolation from smaller data. The memory requirement to run the target job was estimated to be 31.6 GB per process using 80000 processes. From our point of view at that time, such requirement was judged not realistic from process layout view and from scalability view. Our design of the target job at the time was to launch 4 process on a compute node, running 1 process on each CMG, i.e. core-memory-group, so that the job would run on 20000 nodes. The estimated memory requirement was almost 4 times larger than the physical memory size. The number of required nodes would be 80000 instead of 20000. We preferred to use less number of nodes considering the proportion of computing and communication. So we started to investigate the algorithm that would enable further memory reduction and less node requirement.

During the basic design phases, we spent time to redesign the implementation of the matrix data replica, and to fully distribute the matrix data. During the detail design phases, we carefully refactored the code for multiple purposes. The actual implementation was conducted in the following three algorithm blocks.

1. two-center two electron integration, matrix element computation
2. two-center two electron integration, upper triangle matrix Cholesky decomposition and inversion
3. three-center integration

For these blocks, electron integration is conducted per each shell block that depends on the total angular momentum of the basis functions, and the actual implementation was based on the non-uniform division of the matrix column blocks.

The first block (1) divides the shells in matrix column wise blocks, and executes the matrix element computation of each block in MPI parallel way. Because of the relationship between the shells and the basis functions, the dimension of the basis function which corresponds to the divided dimension of the blocked matrix becomes non-uniform.

The second block (2) uses the block matrix data obtained in above (1), and utilizes the LAPACK Cholesky decomposition routine DPOTRF, inversion routine DTRTRI and BLAS level

3 matrix-matrix multiplication routine DGEMM. The column blocked matrix data communication is implemented using MPI_BCAST within the subcommunicator MPI_COMM_MAT as shown in Figures 11.6 and 11.7 .

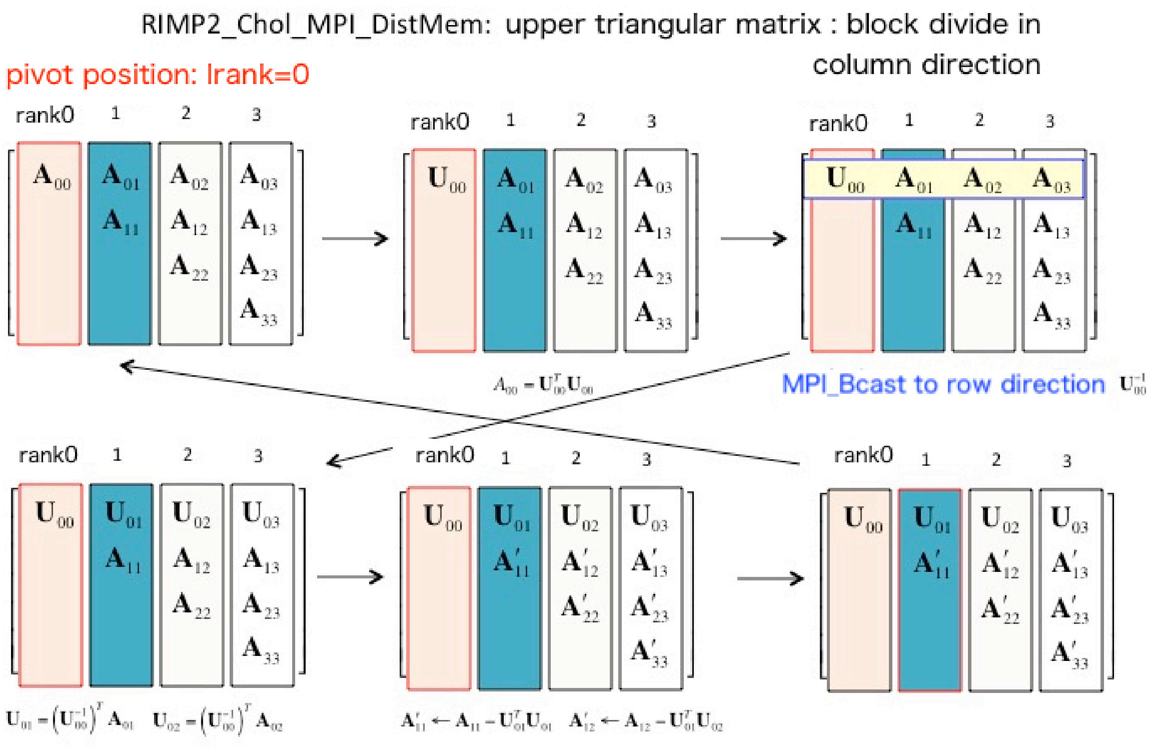

RIMP2_Chol_MPI_DistMem: upper triangular matrix : block divide in column direction
pivot position: Irank=0
rank0 1 2 3
A_{00} A_{01} A_{11} A_{02} A_{12} A_{22} A_{03} A_{13} A_{23} A_{33}
U_{00}
$A_{00} = U_{00}^T U_{00}$
MPI_Bcast to row direction U_{00}^{-1}
U_{01} U_{02} U_{03}
A'_{11} A'_{12} A'_{22} A'_{13} A'_{23} A'_{33}
$U_{01} = (U_{00}^{-1})^T A_{01}$ $U_{02} = (U_{00}^{-1})^T A_{02}$
$A'_{11} \leftarrow A_{11} - U_{01}^T U_{01}$ $A'_{12} \leftarrow A_{12} - U_{01}^T U_{02}$

Figure 11.6: distributed parallel implementation of Cholesky decomposition

The third block (3) divides the shells in matrix column wise blocks, and executes the three-center integration of each block in MPI parallel way. Similarly to (1), the divided dimension of the blocked column matrix becomes non-uniform.. The matrix-matrix multiplication for operating the coefficient matrix after the three-center integration is implemented using DGEMM in each block, and the partial sum of the matrix data is implemented using MPI_ALLREDUCE within the subcommunicator MPI_COMM_MAT.

Although the main purpose of this implementation is to reduce the memory requirement, there is some side effect of speed up brought by (a) the MPI prallelization of Cholesky decomposition and inversion and (b) the reduction of MPI_ALLREDUCE communication needed for the sum of matrix elements kept in the matrix replica.

11.5 memory data distribution - phase 2 implementation

After the memory data distribution explained in the previous section 11.4 is completed, the actual input orbital data to run the target problem is generated, and the memory usage to run the target problem was verified on the K computer using 82944 nodes. The measured maximum memory size per process was 11.1 GB. Compared to the originally estimated 31.6 GB, this reduction was significant. However, our planned process layout for the target job was to launch

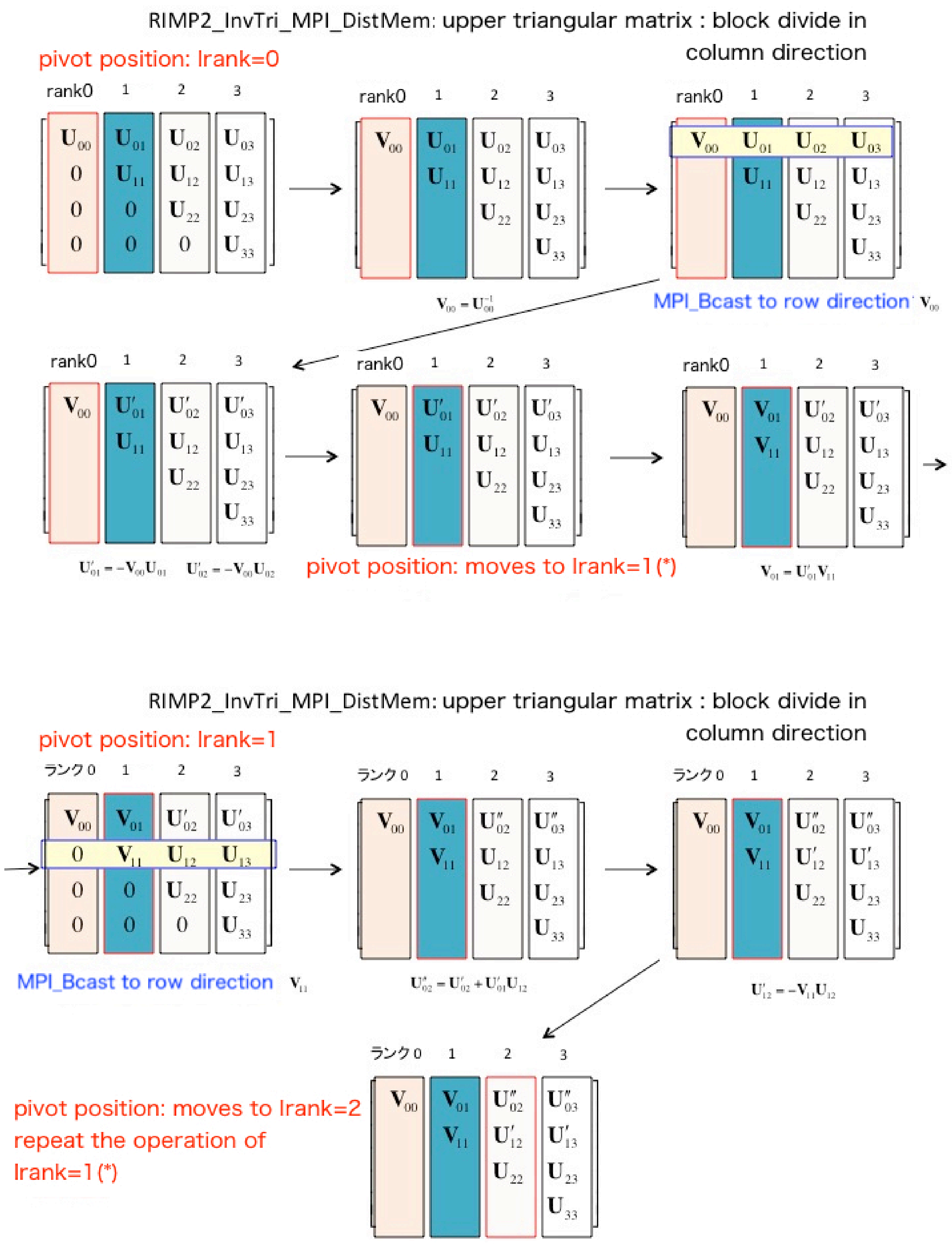

RIMP2_InvTri_MPI_DistMem: upper triangular matrix : block divide in column direction
pivot position: Irank=0
rank0 1 2 3
$V_{00} = U_{00}^{-1}$
MPI_Bcast to row direction V_{00}
$U'_{01} = -V_{00}U_{01}$
$U'_{02} = -V_{00}U_{02}$
pivot position: moves to Irank=1 (*)
$V_{01} = U'_{01}V_{11}$
RIMP2_InvTri_MPI_DistMem: upper triangular matrix : block divide in column direction
pivot position: Irank=1
ランク0 1 2 3
MPI_Bcast to row direction V_{11}
$U''_{02} = U'_{02} + U'_{01}U_{12}$
$U'_{12} = -V_{11}U_{12}$
pivot position: moves to Irank=2
repeat the operation of
Irank=1 (*)

Figure 11.7: distributed parallel implementation of upper triangular matrix inversion

4 processes per compute node, and the memory size per process needed to be 8 GB or less. We decided to pour further effort to reduce the memory requirement to achieve this.

The additional effort to reduce the memory requirement is conducted on top of 11.4 . The strategy for this further reduction is to introduce the next level of block division, and to take advantage of the mixed precision algorithm.

Introducing the next level of block division, for example, is to apply block division to the already distributed data arrays, and accomplish the computation in nested loops, instead of long un-nested loop. Then the size of the block divided distributed data arrays become variable, as opposed to the constant values defined by the input atoms and orbitals.

The mixed precision algorithm will be explained in sections 11.6, 11.8, and 11.9 in detail. Taking advantage of this mixed precision algorithm, some appropriate variables are kept in single precision memory storage. Also the selective storage of the virtual orbitals contributed to reduce the size of memory requirement.

With all these efforts, further reduced memory requirement enabled the computation of the target problem on Fugaku. The final memory requirement to run the target problem on Fugaku is summarized in section 11.12.

11.6 research of mixed precision algorithm and performance effect

The most computationally demanding section in RI-MP2 is the four-center integration section.

$$(ia|jb) = \sum_n B_n^{ia} B_n^{jb}$$

The kernel of this computation uses the multiplication of double precision matrices. The actual electron correlation energy value obtained from the MP2 method or from the high precision coupled cluster method is 4 to 5 orders of magnitude smaller than the HF energy value or the DFT energy value. The acceptable significant figures in the typical application are 3 digits after the decimal point, which correspond to mHartree order in atomic unit.

The design of Fugaku provides the single precision floating point arithmetic whose peak performance is twice the performance of the double precision arithmetic. With this background, we conducted the feasibility study to apply single precision arithmetic in the matrix or tensor calculation.

The first approach is purely mathematic, which is based on the data sorting and precision separation algorithm as below.

1. The original double precision matrix multiplication is written as:
   $(C) = (A)\,(B)$
2. Separate the original double precision matrix into two matrices of double precision elements only and single precision elements only, based on the value of the elements and certain cut-off criteria δ.
   $(A) = \left(A^{\mathbf{double}} + A^{\mathbf{single}}\right), (B) = \left(B^{\mathbf{double}} + B^{\mathbf{single}}\right)$
3. The original multiplication is now:
   $(C) = (A)\left(B^{\mathbf{double}}\right) + \left(A^{\mathbf{double}}\right)\left(B^{\mathbf{single}}\right) + \left(A^{\mathbf{single}}\right)\left(B^{\mathbf{single}}\right)$

4. Use single precision matrix multiplication SGEMM if both the matrices single.
   $\left(A^{\mathbf{single}}\right)\left(B^{\mathbf{single}}\right)$

5. Use double precision multiplication if either of the matrices is double. Although the element value distribution is not known at this point, we keep the double precision elements kept in CSR sparse matrix format, and the multiplication is done using sparse matrix library routines. If the performance gain is to be expected at all, the ratio of double precision elelements should be low, and sparse handling can be judged reasonable in such cases.
   $\left(A\right)\left(B^{\mathbf{double}}\right),\left(A^{\mathbf{double}}\right)\left(B^{\mathbf{single}}\right)$

6. The performance boost by applying the mixed precision algorithm obviously depends on the ratio of the number of single elements versus double, i.e. the performance becomes high if the single/double ratio is high.

The performance estimation formula for above algorithm is built assuming the microarchitecture parameters emplyed at the time, and the numerical study for Fugaku is conducted using artificially generated data, comparing the performance with and without this algorithm. The obtained conclusion is that the performance gain can be expected if the percentage of the double precision elements is lower than 5% of the total elements.

We then conducted the data analysis using small molecules. Figure 11.8 shows the value distribution of the computed matrix elements from the RI-MP2/def2-SVP calculation of a coronene dimer $(C_{24}H_{12})_2$ (72 atoms, 794 atmic orbitals, 158 occupied orbitals, 636 virtual orbitals, 2640 auxiliary basis functions). In this case, setting the cut-off criteria δ as 10^{-8}, will allow all the matrix elements storage and computation can be done in single precision. Setting the cut-off criteria δ as $\delta = 10^{-9}$ will yield only 0.2% of the total elements as double precision, and almost all of the matrix computation can be done in single precision, and significant performance improvement can be expected.

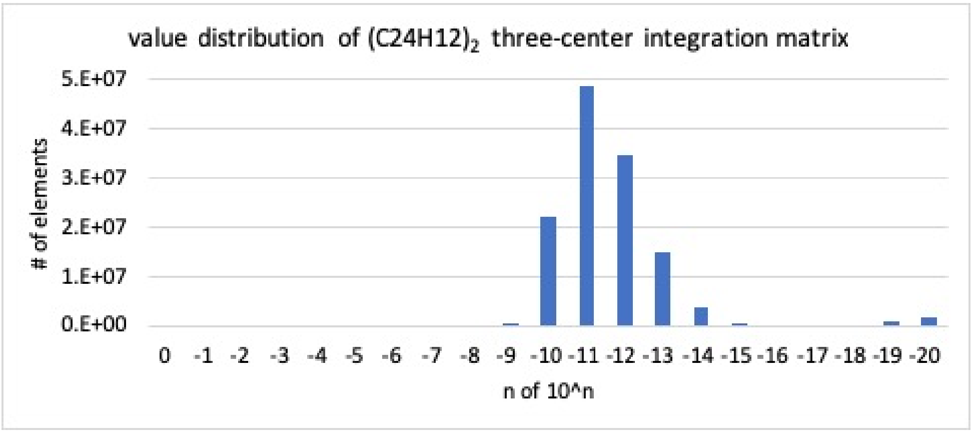


Figure 11.8: element value distribution of the three-center integration matrix for a coronene dimer $(C_{24}H_{12})_2$

11.7 network communicated data reduction

Through the feasibility study of the mixed precision algorithm, the effect to reduce the network communicated data volume by using single precision data also became evident. A simple estimation formula of the network communication performance was built based on the assumed interconnect design of Fugaku, and a quick estimation was done to find out the criteria for reducing the communication wait time . The above first algorithm would enjoy the shorter communication wait time if the ratio of the double precision elements to the total elements is lower than 80%, which is quite likely. If the ratio is lower than 5%, the P2P communication wait time can be reduced to almost half.

11.8 implementation of mixed precision algorithm and performance evaluation

The first approach explained in 11.6 provides some interesting insight into the usefulness of the mixed precision algorithm by separating the matrix according to a certain element value criteria. However, in the applications using various molecular models, the criteria depends on the discrete nature of the molecular model being analyzed. In order to implement the first approach, the criteria must be generally formulated and must be valid not only from mathematical view but also from quantum chemistry view. We perceive that general formulation of such criteria is difficult, and we opt to seek for an alternative approach in quantum chemistry orientation.

This section describes this second approach which is finally adopted and implemented in NTChem/RI-MP2. Our strategy in the second approach of mixed precision algorithm is to put more weight on quantum chemistry algorithm view by classifying the sequence of electron repulsion integration steps and find out which calculation really tolerates single precision arithmetic still providing accurate enough energy value.

For each of computing steps in NTChem/RI-MP2 integration sequence, we prepared both double precision and single precision version of the procedure each having discrete storage and arithmetic precision. We carefully analyzed the effect of applying precision and resulting effect in backward manner using many molecular models, trying to find the appropriate location for applying single precision arithmetic from both views of precision and performance.

We started the analysis using the prototype single precision implementation applyed to matrix-matrix multiplication in the four-center integration, coupled with the single precision pairwaise communication using MPI_ISend and MPI_IRecv.

Table 11.5: mixed precision test results of typical molecular models on Intel cluster

molecular model	$C_{24}H_{12}$	Taxol	$C_{54}H_{18}$	$(C_{54}H_{18})_2$	C_{60}@$C_{60}H_{28}$
energy value error (Hartree)	6.0E-7	8.4E-7	-3.5E-8	-1.3E-7	6.1E-6
mixed precision time (sec)	5.9	46.6	96.7	1986.5	2717.9
double precision time (sec)	8.6	78.5	172.5	3821.9	5356.7

Table 11.5 shows the RI-MP2/cc-pVTZ test results of this prototype using a handful of test molecular models. The tests were run on Intel Xeon Haswell cluster configured as: Xeon E5-2697 v4, 18 cores/CPU, 2CPUs/node, DDR4 256GB memory/node, 4 nodes/system, InfiniBand FDR network. All the test cases show that the energy value difference between fully double precision version and mixed precision version is less than 1.0E-6 Hartree and is judged accurate enough for application usage. The performance effect of this mixed precision version ranges from 1.5x to nearly 2x which is satisfactory.

The double precision execution time breakdown of major routines for the ritht most molecular model C_{60}@$C_{60}H_{28}$, also known as buckycatcher, in above Table 11.5 is shown in Figure 11.6 for reference. The four-center integration and transformation routines named MOInt4c_* spends most of the execution time and their effect of applying single precision dominates the overall performance.

Table 11.6: execution time in double precision for buckycatcher $C_{60}@C_{60}H_{28}$ per routine base

routine	time (sec)
RIMDInt3c_calc	49.89
MOInt3c_tran_1 (DGEMM)	28.79
MOInt3c_tran_2 (DGEMM)	27.33
MOInt3c_comm_1a,b (MPI_ISend)	15.12
AOInt2c_calc	0.17
Inv2c_DPOTRF	2.53
Inv2c_DTRTRI	2.28
MOInt3c_comm_2 (MPI_ISend)	8.27
MOInt3c_tran_3 (DGEMM)	70.43
MOInt3c_comm_3 (MPI_ISend)	0.04
MOInt4c_calc (DGEMM)	3708.24
MOInt4c_comm (MPI_AllReduce)	1228.72
MP2Energy_calc	56.03
MP2Energy_comm (MPI_AllRecude)	6.30
Total time	5356.70

This prototype implementation provided satisfactory speed up while maintaining the required computational accuracy. So we proceeded to the backward steps into the three-center integration and transformation, and prepared several implementation variations in order to check if precision relaxation can be applied to further range of NTChem/RI-MP2 algorithm. To distinguish these variations, we tentatively name precision implementation model as Table 11.7.

The routines named MOInt4c_* are related with the four-center integration and transformation, and MOInt3c_* are related with the three-center integration and transformation. For example, MOInt3c_tran_3 calculates the third transformation in the three-center integration.

Table 11.7: mixed precision implementation variation and corresponding matrix routines

implementation	MOInt3c_tran_1	MOInt3c_tran_2	MOInt3c_tran_3	MOInt4c_calc
variation	gemm	gemm	gemm	gemm
SP123a	SGEMM	SGEMM	SGEMM	SGEMM
SP123b	DGEMM	SGEMM	SGEMM	SGEMM
SP23	DGEMM	DGEMM	SGEMM	SGEMM
SP3	DGEMM	DGEMM	DGEMM	SGEMM
DP	DGEMM	DGEMM	DGEMM	DGEMM

The accuracy of the computed results and the computing performance of each variation is verified using nanographene $C_{24}H_{12}$ and $C_{54}H_{18}$ molecular models on the same Intel Xeon Haswell cluster. Table 11.8 shows the test results. Regarding the accuracy of the computed results, SP23 and SP3 provided accurate enough Hartree value compared with DP, whereas SP123b and SP123a resulted in larger numerical error which is inadequate for real application usage. Regarding the computing performance, SP3 provided significant speed-up, whereas SP123b, SP123a and SP23 provided only marginal speed-up on top of SP3.

Table 11.8: mixed precision implementation variation for $C_{24}H_{12}$ and $C_{54}H_{18}$

	$C_{24}H_{12}$	$C_{54}H_{18}$
variation	error (Hartree)	error (Hartree)
SP123a	-4.9.E-03	-8.2.E-01
SP123b	-6.7.E-04	-9.1.E-02
SP23	6.0.E-07	-3.5.E-08
SP3	8.1.E-08	3.4.E-07
DP	0.0	0.0
variation	time (sec)	time (sec)
SP123a	5.9	97.0
SP123b	5.6	96.2
SP23	5.9	96.7
SP3	6.2	101.0
DP	8.6	172.5
variation	speed-up (X)	speed-up (X)
SP123a	1.5	1.8
SP123b	1.5	1.8
SP23	1.5	1.8
SP3	1.4	1.7
DP	1.0	1.0

Considering these test results, SP23 and SP3 implementation variations are suggested as good choices in the balance of accuracy and performance.

On the other hand, the verification of the computational accuracy should be carried out using various molecular systems in order to convince ourselves of the validity of this mixed precision algorithm. Such broad and systematic verification is explained in the next section.

11.9 Further study and numerical verification of mixed precision algorithm

In order to evaluate the effect of the mixed precision algorithm in comprehensive and systematic way, we conducted numbers of numerical tests of various molecular systems using the implementation variation in Table 11.7. We selected multiple data sets that handle many weak interactions of molecules. The datasets are explained later in this section.

The molecular systems contained in the dataset are supramolecules in which two molecules are close to each other and gather in a weak interaction that is not a covalent bond. The intermolecular interaction energy $E_{\mathbf{bind}}$ of supramolecule (1-2) formed in molecule (1) and molecule (2) is computed by the following formula,

$$E_{\mathbf{bind}} = E_{(\mathbf{supramolecule\ 1\text{-}2})} - E_{(\mathbf{molecule\ 1})} - E_{(\mathbf{molecule\ 2})}$$

using the energy E of supramolecule(1-2), molecule(1) and molecule(2) each computed by RI-MP2 method.

Table 11.7 shows the summary of the numerical accuracy tests of RI-MP2 with cc-pVTZ basis function using intermolecular interaction datasets.

The value of the intermolecular interaction energy $E_{\mathbf{bind}}$ computed with each of SP3/SP23/SP123b/SP123a implementation variation is compared to that of the original double precision (DP) implementation, and the difference, i.e. numerical error, is shown in Table 11.7 in average absolute error unit of (kcal/mol).

Table 11.9: numerical verification of implementation variation using intermolecular interaction datasets

molecule	implementation variation			
dataset	SP3	SP23	SP123b	SP123a
S66x8	2.16E-05	2.09E-04	4.29E-04	6.51E-04
Water Cluster	1.80E-05	4.97E-04	5.49E-04	6.13E-04
X40	1.58E-05	1.17E-04	3.32E-04	5.62E-04
L7	8.98E-04	1.60E-03	1.51E+00	7.05E+02

Firstly, the numerical verification is conducted using the S66x8 dataset [119], which deals with 66 small molecular systems and changes the intermolecular distance in 8 ways. By changing the intermolecular distance, even a slight change in the intermolecular interaction energy can be investigated, enabling more precise verification. The results are shown in Table 11.7. The average absolute error of the intermolecular interaction energy in comparison to DP is less than 0.0001 kcal/mol for all of the implementation variations. This is accurate enough for practical applications which require the computational accuracy in 0.1 kcal/mol order.

Secondly, the verification is conducted using the Water Cluster dataset [120], which deals with 38 types of aggregate molecular systems of various water molecules. This dataset deals with aggregate molecules containing up to 2-10 water molecules, including molecules larger than S66x8. The average absolute error is, again, less than 0.0001 kcal/mol for all of the implementation variations.

Thirdly, the same verification is conducted using the X40 dataset [121], which deals with 40 types of molecular systems containing organic halides, halohydrides, and halogen molecules. The average absolute error is, again, less than 0.0001 kcal/mol for all of the implementation variations. It was confirmed that the implementation variations can be adopted to practical applications for the calculation of weak intermolecular interactions in small molecular systems.

Lastly, the same verification is conducted using the L7 dataset [122], which deals with relatively large molecular systems such as the coronene dimer or the guanine trimer, etc.

The average absolute errors for SP3 and SP23 are 0.0009 and 0.0016 respectively. The errors for SP3 and SP23 are 0.0009 and 0.0016 respectively. SP123b and SP123a becomes larger than 1 kcal/mol. Through the series of verification, it is obvious now that the single precision calculation applied to larger molecular systems tends to accumulate the numerical error beyond acceptable criteria unless we carefully choose the right implementation variation for mixed precision.

From computing cost view, the most demanding part is the four-center integration section whose major routine (MOInt4c_calc) executes matrix multiplication. It occupies the major portion of the computing time of RI-MP2 calculation. The next demanding part is the third transformation in the three-center integration section, and the corresponding routine (MOInt3c_tran_3) spends the second most portion of the computing time. The upstream first and second transformation routines in the three-center integration section (MOInt3c_tran_1 and MOInt3c_tran_2) spends less computing time.

Based on the facts obtained through the numerical verification, we decided to focus the implementation effort of mixed precision algorithm at the time onto SP23 and SP3, and proceeded to the numerical verification using the target problem which computes a large molecule system which is explained in 11.12.

In the later phase of detail design, the NTChem/RI-MP2 codesign module furnished with various memory reduction scheme and the target problem $(C_{150}H_{30})_4$ molecule input data files were made ready. We were able to run the numerical verification using the target problem on the K computer before it was decommissioned. The accuracy results are summarized in Table 11.10. The variation SP23 yielded 3.0 kcal/mol and SP3 yielded 0.041 kcal/mol. Remark that the unit in the right most absolute error value is in kcal/mol, not hatree. SP3 has been judged adequate for the calculation of intermolecular interactions. SP23 has been judged inadequate for the calculation of intermolecular interactions. large size molecular systems. We finally selected SP3 mixed precision model as the default algorithm of NTChem/RI-MP2 module, which provides efficient computational performance and precise accuracy for very large molecule systems such as the target problem $(C_{150}H_{30})_4$.

Table 11.10: RI-MP2 correlation energy (a.u.) of the target problem and the relative error compared to the fully double precision algorithm(DP)

variation	correlation energy (a.u.)	relative error (kcal/mol)
DP	-105.2921859	0.0
SP3	-105.2921208	0.041
SP23	-105.2970498	3.0

11.10 performance improvement in the three-center integration

In the NTChem/RI-MP2 calculation of the target problem, the computing time of the three-center integration section (electronic repulsion integration calculation) is much lower than that of the four-center integration.

But, in smaller molecular systems, the portion of the three-center computing time becomes relatively high in the total computing time. The computational performance of the three-center integration section of NTChem/RI-MP2 is not good because of the rather naive implementation of the classic algorithm. Since the three-center integration is widely used in the other applications of Priority Issue 5 as well, we generally agreed that the performance improvement of the three-center integration section should be addressed.

The original three-center integration routine was based on the naive implementation of McMurchie-Davidson(McD) algorithm [123]. We investigated other computational method for the three-center integration, and picked up an open source quantum chemistry package called SMASH developed by Kazuya Ishimura. SMASH includes the integral calculation library optimized for high performance computing, and it has the program structure that can be incorporated into NTChem/RI-MP2. We incorporated SMASH integration library into NTChem/RI-MP2 as routines, so that it can be selected at run time.

In three-center integration, the computing cost of the integral based on the high angular momentum function is particularly high. SMASH integral kernel routines reduce the computing cost of the basis integral by using different algorithms for low angular momentum and for high angular momentum. For integrals with low angular momentum, SMASH uses the Ishimura (IN) algorithm [124] which combines the Pople-Hehre (PH) algorithm [125] and the McD algorithm. For high angular momentum, SMASH uses the algorithm based on Rys quadrature method (DRK) [126].

After incorporating the open source SMASH library into NTChem/RI-MP2, we applied some source code tuning to that library trying to to reduce the computing wait time and cache access wait time through better instruction scheduling.

The benchmark test results of the three-center integration using original McD, SMASH open source, SMASH tuning are shown in Table 11.11. The table shows the execution time and some hardware performance counter (HWPC) statistics of interest. All the tests are run on Fujitsu FX100 system.

Table 11.11: performance improvement of the three-center integration

routine version	original McD	SMASH asis	SMASH tuned
execution time (sec)	1.91	0.47	0.41
sustained/peak rate (%)	0.28%	0.98%	1.13%
L1 busy rate (%)	27%	24%	27%
L2 busy rate(%)	6%	4%	5%
memory busy rate(%)	0%	0%	0%
L2 throughput (GB/s)	36.51	25.56	27.87
memory throughput (GB/s)	0.50	0.17	0.03

11.11 selective file format for input data

The size of the molecular orbital coefficient file is 9.4 GB in text format. On the K computer, it takes 589 seconds for reading. By changing the file format to binary, the file size is reduced to 1/3, and the time for reading is reduced to 1/7. The program specification and the input file specification is revised so that NTChem/RI-MP2 can read both types of file format.

11.12 the target problem and the process layout

The target problem is defined as a typical example of high precision ab initio electronic state calculation for elucidating the mechanism of photochemical reaction and screening the material candidates for light energy conversion, The target problem executes the RI-MP2 electron correlation energy calculation of 720 atom 19680 electron orbital carbon nanographene molecular complex $(C_{150}H_{30})_4$.

As explained previously, the NTChem/RI-MP2 job is distributed over the process space in two dimensional $N_{\mathbf{procsMO}} \times N_{\mathbf{procsMat}}$ process layout. The design of such process layout affects the computing performance as well as the required amount of memory space. The optimum configuration for the target problem is searched considering the conditions such as the number of used nodes, the required memory size per process, the number of threads, the size of the matrices, the aspect ratio of the matrices, etc. We prepared a small subset of NTChem/RI-MP2 in order to conduct this parameter study, and conducted the exhaustive search to find the reasonable process layout patterns for the target problem. We first searched through the process layouts based on the system recommended 4 processes $\times$ 12 threads per node patterns. This standard 4 process/node layout is labled as "Std". We also searched through the fat process thread layouts of 2×24 and 1×48 . These 2 process/node and 1 process/node layout patterns are labeled as "Fat". Table 11.12 shows some of the process layout patterns and their attributes.

The final configuration we chose for running the target problem is shown in the last row, i.e. Fat 17820x1 layout.

Altough the total computing volume stays the same across these layout patterns, the computational performance and communication pattern varies a lot across the layout patterns. The sustained/peak ratio for computing matrices often depends on the size and shape of the matrices, and the communication wait time varies according to the length of messages, communicator

Table 11.12: process layout candidate for the target problem

layout pattern	$N_{\mathbf{procs}}$	threads /process	$N_{\mathbf{procs}}$ Mat	$N_{\mathbf{procs}}$ MO	mem.GB /process	mem.GB /node	matrix m	matrix n	matrix k
Std 82928	82928	12	16	5183	7.50	30.0	7440	3263	7440
Std 71280	71280	12	8	8910	6.08	24.3	1860	1860	6525
Std 71288	71288	12	8	8911	6.86	27.4	3720	3720	6525
Fat 35640x2	35640	24	8	4455	10.12	20.25	7440	7440	6525
Fat 20376x1	20376	48	8	2547	14.17	14.17	13020	13020	6526
Fat 17820x1	17820	48	4	4455	13.68	13.68	7440	7440	13049

range and frequency communication. Table 11.13 shows a couple of layout patterns and their comminication statistics of the selected sections for target job execution, as an example. Note that they apply different precision model of DP and SP3.

Table 11.13: NTChem/RI-MP2 communication statistics for the target problem

layout pattern	routine	called MPI routine	message length (Bytes)	# of calling times	total data volume (GB)
Std	MOInt3c_comm_2 +	MPI_ISend/IRecv	194451840	5184	1008.0
82928	MOInt3c_comm_3				
(DP)	MOInt4c_comm	MPI_Allreduce	442828800	2271	1112.3
	MOInt4c_comm	MPI_Allreduce	332121600	321	
Std	MOInt3c_comm_2 +	MPI_ISend/IRecv	48546000	35642	1730.3
71280	MOInt3c_comm_3				
(SP3)	MOInt4c_comm	MPI_Allreduce	13838400	17821	246.6

11.13 explicit promotion of overlapping the computation and the communication

In order to take advantage of the assistant cores available on Fugaku CPU whose feature include the support of overlapping non-blocking MPI communication with arithmetic operations on compute cores, explicit API for promoting the non-blocking MPI communication is added to NTChem/RI-MP2. The overlapping performance of non-blocking MPI depends on systems implementation and often does not meet the callers expectation. The Fugaku local APIs named FJMPI_Progress_start() and FJMPI_Progress_stop(), as the extension from the standard OpenMPI 4.0, explicitly promotes the overlapping data transfer and help reduce the MPI wait time. The effect of this explicit overlapping promotion is apparent in the target problem run in the final round.

11.14 early test on prototype vehicle

A year before the massive production of Fugaku begins, several prototype vehicles are made ready for our early tests. These prototype vehicles are mainly regarded as the testing environment for verifying the functionality of hardware and software. The early tests using small size test data indicated the performance of Fugaku over the K computer is 5.8X faster using 1/4 number of nodes. This is a quite encouraging speed-up at the time, and we continued the effort to complete the codesign subjects explained in the previous sections.

11.15 incorporating PMlib performance monitor library

In order to accurately measure the computational performance of the distributed processes, PMlib performance monitor library [127, 128] is incorporated into NTChem/RI-MP2. Detail statistics from Fugaku hardware performance counters (HWPC) are abtained and sorted according to the user specified performance categories through environment variable. Supported report categories and the corresponding HWPC statistics on Fugaku are shown in Table 11.14.

The measured HWPC statistics and wait time distribution of the specified sections across the processes is useful when analyzing the load imbalance and/or computing and communication overlapping effectiveness. The finally measured target jobs achieved quite high average FLOPs, as shown in the later section 11.17.2, Since PMlib is an open source software, such installation will help users conduct the performance analysis of the application across different systems.

Table 11.14: PMlib report variation on Fugaku

PMlib category	reported HWPC statistics
FLOPS	floating point operations in single precision and double precision FLOPS and peak % (sustained over peak performance)
BANDWIDTH	CMG local memory read and write counts, bytes and bandwidth
VECTOR	floating point operations in single and double precision by SVE and scalar/armv8 instructions percentage of SIMD (vectorized) floating point operations
CACHE	memory load and store instructions L1 data cache hits and misses L2 cache misses data access hit(%) in L1 cache and in L2 cache
LOADSTORE	memory load and store instructions grouped into: scalar instructions SVE and Advanced SIMD load and store instructions Advanced SIMD multiple vector contiguous structure load/store instructions SVE non-contiguous gather-load and scatter-store instructions percentage of SVE load/store instructions over all load/store instructions
CYCLE	total cycles and instructions percentage of fused multiply+add (FMA) instructions over all f.p. instructions performed instructions per machine clock cycle
USER	User provided argument values (Arithmetic Workload) are accumulated

11.16 implementation request to Fugaku system through codesign

NTChem WG requested the folowing design enhancements to Fugaku system through codesign.

- implement DGEMM double precision matrix library making it provide high performance multiplication for various input matrix sizes

- implement SGEMM single precision matrix library making it provide high performance multiplication for various input matrix sizes
- optimize the loop containing the formula using single and double precision variables by utilizing the 16 element single precision SIMD instructions.

As previously explained, the most demanding and time consuming part of the NTChem/RI-MP2 calculation is the dense matrix multiplication in four-center integration. The size of such matrix for the target problem can be seen in Table 11.12. For example, the first "Std 82928" layout requires the following matrix multiplication.

$$\big(C\big) = \big(A\big)\,\big(B\big) \ \text{ where } \ \big(A\big) \equiv A(7440, 3263) \text{ and } \ \big(B\big) \equiv B(3263, 7440)$$

From the experience of running HPL(Linpack) on various platforms, it is well known that the computing performance of matrix multiplication (GEMM) can be very high if the input matrices are appropriately blocked (tiled) so as to improve the data locality and reusability at L1/L2 cache layers. Such blocking implementation in GEMM, especially DGEMM, is usually optimized for HPL size of problem for good reasons.

However, the matrix size per process of the NTChem/RI-MP2 target problem is an order of magnitude smaller than such HPL matrix size. There will many cases of electronic interaction calculations by Priority Issue 5 and the related researchers using different molecular systems as well. In order to effectively run NTChem/RI-MP2 using different matrices with parallel combination of processes and threads, the provisioning of the high performance multiplication for various matrix size is very important. Especially, the mixed precision algorightm heavily depends on the performance of SGEMM in its usefulness as described in sections 11.6, 11.8 and 11.9. With this backgrond NTChem WG requested the design enhancement of both DGEMM and SGEMM to system side with discrete values of sustained/peak ratio as must criteria. The request has been reflected to DGEMM and SGEMM routines in Fugaku numerical library as SSL II, achieving the requested performance criteria for each matrix size.

Regarding the loop optimization that contains both of single precision variables and double precision variables, the general implementation using the 16 element SIMD instructions by converting double precision variables into single precision turned out too expensive because of the type conversion and register packing. Instead, a small set of optimization feature was implemented into the compiler to support the mixed precision reduction using the 16 element SIMD instructions in the form of

$$Y_{\mathbf{double}} = Y_{\mathbf{double}} \text{ .op. } X_{\mathbf{double}}$$

where .op. represents one of $+ - \times$ reduction operators.

11.17 performance evaluation of Fugaku

The system wise performance measurement comparing the baseline performance on the K computer and the performance on Fugaku using the latest NTChem/RI-MP2 module is conducted. Such comparison is of interest from the viewpoint of overall codesign achievement as the combined effect of the new system capability and the application improvement. The latest NTChem/RI-MP2 module with various codesign subjects explained in the previous sections is reffered to as NTChem/RI-MP2 codesign code.

11.17.1 baseline performance on the K computer

The performance on the K computer using baseline code is reffered to as baseline performance. At the time of starting codesign, it was not possible run the target problem using NTChem/RI-MP2 module because of the total memory requirement, thus the baseline performance is estimated with the following assumptions:

- the performance characteristics is represented by a set of kernel routines and their weight
- the computing volume is extrapolated from the number of atmos and atmic orbitals.
- the execution time is decoupled into computing time and communication time
- the preliminary process layout for the target problem is assumed

At a later stage of codesign, the actual measurement is conducted with run time parameters set as close to that of baseline code and its condition, to confirm the adequacy of above assumption. Table 11.15 shows the comparison of these estimated and measured baseline time.

Table 11.15: estimated and measured baseline performance on the K computer

	total time (sec)	computing time (sec)	communication time (sec)	I/O wait time (sec)
estimated K baseline	7725	7609	116	-
measured K baseline	10938	9073	1265	600

The measured execution time in computing sections turns out 20% shorter than the estimation, and the time in communication sections turns out an order of magnitude longer. The excessive wait time in communication on the K computer is problematic, but it should be addressed in other context. The wait time is reduced to 1/5 by explicitly assigning the appropriate MPI map file on the K computer, for readers interest. On Fugaku, the measured communication time is well reduced by applying the appropriate process layout without needing such map file.

11.17.2 achieved performance on Fugaku

Table 11.16: achieved performance on Fugaku

total time (sec)	computing time (sec)	communication time (sec)	I/O wait time (sec)	speed up /job	speed up /system	sustained /peak (%)
990	744	235	11	7.8	70	55

Table 11.16 shows the finally measured execution time of the target problem on Fugaku. The measurement job was run on Fugaku for Fat 17820x1 layout at boost mode. The job achieved the computing performance of 3.38 TFLOPS / node in average which is 55% of the peak single precision performance of Fugaku compute node. With Fugaku's 158976 compute nodes, more than 8 copies of the similar jobs can be executed simultaneously on Fugaku, providing almost 70 times more powerful computing throughput compared to the K computer. This measurement reflects the overall achievements of codesign, including the application improvement and the system hardware/software enhancement.

Through the effort of codesign, not only the computational performance improvement, but also the code modernization has been realized. The codesign effort is judged quite fruitful resulting in production ready NTChem/RI-MP2 module on the supercomputer Fugaku, as well as the valuable insight and knowledge obtained through codesign, which has been described in detail as above.

11.18 Authors

The authors and the contributors of this chapter are listed below. The affiliations in the parentheses are the ones at the time of participation.

11.18.0.0.1 Authors

- Kazunori Mikami (RIKEN)
- Michio Katouda (RIKEN)

11.18.0.0.2 Co-Authors in alphabetical order

- Yoshimichi Andoh (Nagoya U.)
- Kouji Inagaki (Osaka U.)
- Kazuya Ishimura (IMS)
- Muneaki Kamiya (Gifu U.)
- Yukio Kawashima (RIKEN)
- Kazuo Minami (RIKEN)
- Takahito Nakajima (RIKEN)
- Yutaka Nakatuka (Gifu U.)
- Yu-ya Ohnishi (Kobe U.)
- Susumu Okazaki (IMS)
- Keisuke Sawada (RIKEN)
- Takashi Tsuchimochi (Kobe U.)
- Motoyuki Uejima (Kobe U.)

References

[116] T. Nakajima et al. “NTChem: A High-Performance Software Package for Quantum Molecular Simulation”. In: *Int. J. Quantum Chem.* 115.5 (2015), pp. 349–359.

[117] M.Katouda and T. Nakajima. “MPI/OpenMP Hybrid Parallel Algorithm of Resolution of Identity Second-Order Moller-Plesset Perturbation Calculation for Massively Parallel Multicore Supercomputers”. In: *J. Chem. Theory Comput.* 9.12 (2013), pp. 5373–5378.

[118] M.Katouda et al. “Massively parallel algorithm and implementation of RI-MP2 energy calculation for peta-scale many-core supercomputers”. In: *J. Comput. Chem.* 37.12 (2013), pp. 5373–5378.

[119] J. Řezáč, K. E. Riley, and P. Hobza. "S66: A Well-balanced Database of Benchmark Interaction Energies Relevant to Biomolecular Structures". In: *J. Chem. Theory Comput.* 7.8 (2011), pp. 2427–2438.

[120] B. Temelso, K. A. Archer, and G. C. Shields. "Benchmark Structures and Binding Energies of Small Water Clusters with Anharmonicity Corrections". In: *J. Phys. Chem. A* 115.43 (2011), pp. 12034–12046.

[121] J. Řezáč, K. E. Riley, and P. Hobza. "Benchmark Calculations of Noncovalent Interactions of Halogenated Molecules". In: *J. Chem. Theory Comput.* 8.11 (2010), pp. 4285–4292.

[122] R. Sedlak et al. "Accuracy of Quantum Chemical Methods for Large Noncovalent Complexes". In: *J. Chem. Theory Comput.* 8.9 (2013), pp. 3364–3374.

[123] L. E. McMurchie and E. R. Davidson. "One- and Two-Electron Integrals over Cartesian Gaussian Functions". In: *J. Comput. Phys.* 26.2 (1978), pp. 218–231.

[124] K. Ishimura and S. Nagase. "A new algorithm of two-electron repulsion integral calculations: a combination of Pople–Hehre and McMurchie–Davidson methods". In: *Theo. Chem. Acc.* 120.1-3 (2007), pp. 185–189.

[125] J. A. Pople and W. J. Hehre. "Computation of Electron Repulsion Integrals Involving Contracted Gaussian Basis Functions". In: *J. Comput. Phys.* 27.2 (1978), pp. 161–168.

[126] M. Dupuis, J. Rys, and H. F. King. "Evaluation of molecular integrals over Gaussian basis functions". In: *J. Chem. Phys.* 65.1 (1976), pp. 111–116.

[127] K. Ono and K. Mikami. *PMlib - performance monitor library.* `https://github.com/avr-aics-riken/PMlib`.

[128] K. Mikami, K. Ono, and J. Nonaka. "Performance evaluation and visualization of scientific applications using PMlib". In: *6th International Workshop on Large-scale HPC Application Modernization at CANDAR 2018.*

Chapter 12

Codesign of ADVENTURE

12.1 Application features

In the domain decomposition method (DDM) structural analysis program ADVENTURE, the analysis target is divided into multiple subregions that do not overlap each other, the simultaneous linear equations (interface problem) regarding the displacement of the nodes on the boundary between the subregions are calculated by the conjugate gradient method. The iterative solver part by the CG method, mentioned here, is the main processing of the ADVENTURE. That is, in the main processing unit, the reaction force at the boundary between the partial regions is calculated from the temporary forced displacement at the boundary between the partial regions, and the temporary forced displacement between the partial regions is updated from the total reaction force. This is repeated until the sum of the reaction forces at the boundary between the partial regions is sufficiently close to zero. Further, in order to reduce the number of repetitions of this CG iteration, a preprocessing consisting of two stages, the diagonal scaling for each partial region and the course grid correction is also performed for each CG iteration.This preprocessing is called BDD (Balancing Domain Decomposition) -diag preprocessing, and this CG iteration with BDD-diag preprocessing is also called BDD iteration.

The main process for calculating the reaction force at the boundary between subregions from the temporary forced displacement at the boundary between subregions is the Finite Element Method (FEM) calculation (DomainFEM) for each subregion. In DomainFEM, the matrix vector product of a matrix of local Schur complement and a vector of interface degree of freedom is performed. The matrix of local Schur complement have one side length of the interface degree of freedom and the matrix vector product is performed in function of MatVec_produc.

The interface degrees of freedom of each sub-region is the total number of degrees of freedom of nodes on the interface with the adjacent sub-region in each sub-region. The degree of freedom per node is 3 in displacement only. Here, both the local Schur complement matrix and the interface degree of freedom vector are held in double precision, and the operation related to MatVec _product, which is the product of the two, is also performed in double precision. Process parallelization in DomainFEM is performed by allocating the same number of subregions to each process, and the source program of MatVec_product itself is not process parallelized. That is, in each process, MatVec_product is executed several times in the subregion allocated to that process.

By using the results of DomainFEM, for the purpose of to evaluate the balance of reaction forces on the boundaries between subregions, it is performed that is the boundary surface communication (Exchange: mpi_isend & mpi_irecv) between adjacent subregions to exchange the node data on the interface with adjacent subregions. Similarly, also the reduced communication for summation (Allreduce) to calculate for two work variables (double precision type) is performed. In addition, to determine if the total reaction force is sufficiently close to zero and to update the tentative forced displacement on the inter-regional interface from the total reaction force, reduction communication (Allreduce:double precision type) for sum calculation related to one working variable is performed.

In the course grid modification, which is the latter half of the BDD-diag preprocessing, residual calculation and course solver solving are performed. Similar to DomainFEM, the main processing of residual calculation in course grid modification is matrix-vector product oparation(MatVec_product) of a double-precision symmetric dense matrix having one side length of interface degree of freedom and a double-precision vector having length of interface degree of freedom. And also, inter-adjacent sub-region boundary surface communication (Exchange) for exchanging node data (double precision type) on the interface with adjacent sub-regions is performed.

In the course solver solution in the course grid modification, the sparse matrix (course matrix) of the degree of freedom of the course problem and the inverse matrix is generated in advance before entering the BDD iteration. During BDD iteration, a method (inverse matrix approach) is adopted in which the inverse matrix is multiplied by a vector (course degree of freedom vector) in which the iteration vector of the CG method is mapped on the course grid.

Therefore, the main processing of the course solver solution in the course grid modification during BDD iteration is the dense matrix vector product(CoarseMatVec) of the inverse matrix and the course degrees of freedom vector. Here, the course degree of freedom vector is a double precision type, but the inverse matrix is held in a double precision format, and CoarseMatVec, which is an operation of multiplying the matrix and vector, is also performed in the double precision format. Process parallelization in CoarseMatVec is done by assigning the same number of inverse matrix rows to each process.

That is, in each process, the dot product calculation in double-double-precision format of double-precision and double-double-precision vector whose length having the degree of freedom of the course problem is performed, as many as the row number of the inverse matrix assigned to that each process.

At this time, since all processes need to hold the entire course degree of freedom vector (double precision type), the course degree of freedom vector aggregation communication (Allgather) is performed immediately before CoaseMatVec. Also, since the double-double-precision vector obtained as a result of CoarseMatVec is converted to double-precision type immediately after CoarseMatVec, ADVENTURE handles variables and operations in double-double-precision format only for CoarseMatVec.

After the course grid modification, the adjacent subregion boundary surface communication (Exchange) is performed in order to reflect the result of the course grid modification (double precision type) to the displacement(double precision type) of the boundary between the subregions.

12.2 Codesign policy

When estimating an execution time of Adventure, the main processing of Adventure is a time step iteration and processing within BDD iteration, and the time taken to it is accounted the execution time of Adventure. The BDD iteration is divided into eight operation intervals and three communication intervals as follows, estimating execution time of each section, and estimating the execution time of adventure by summing them. Within Adventure's BDD iteration, the heavy processing of the load is a dense row vector product (DomainFem, CoarseMatvec) and a short consolidation communication (allgather) of the message length.

12.3 Target problem for performance estimation

Here, I will show the assumed problem about ADVENTURE on the Fugaku computer. One partial area is a cube and $256 \times 256 = 65536$ pieces of partial problems with a flat plate-like area are a system to be analyzed and this non-linear response problem is made the assumed problem when estimating an execution time. Expected Problem 10000 Time step iteration is performed per one case, and 500 BDD repetitions per one step iteration are performed. In other words, it is assumed that 5 million BDD repetitions per one case is performed. Estimate the execution time of ADVENTURE in Fugaku, using the time required to process 100 cases of the assumed problem as the execution time of ADVENTURE. In Fugaku, hypothetical problem 1 case shall be handled by 16384 processes on 4096 nodes. Therefore, it will be in charge of 4 subregions per process.

In the assumption problem, one side of the partial area is divided into 10 elements. Also, since it uses a 3D solid tetrahedral quadratic element, it has an intermediate node for each element side. Therefore, the number of nodes per side of one subregion is calculated as $(10 \times 2 + 1) = 21$. Here, in a system in which a plurality of subregions are arranged in a plane, the number of nodes per side in the direction along the plane of the system is not equal to the number of subregions times 21. Since it is necessary to avoid double counting of nodes on the boundary between subregions, if two subregions are arranged side by side, the number of nodes per side in the horizontal direction is calculated as $(2 \times 10 \times 2 + 1) = 41$. Based on this, the total number of nodes in the system to be analyzed is calculated as $(256 \times 10 \times 2 + 1)^2 \times (1 \times 10 \times 2 + 1) = 550717461$. The number of degrees of freedom per node is only displacement 3, and the total number of degrees of freedom of the system to be analyzed is about 1.65 billion, which is three times the total number of nodes. The interface degrees of freedom of each subregion is the total number of degrees of freedom of the nodes on the interface with the adjacent subregion in each subregion, and in the system to be analyzed, It is calculated as $(21 \times 21 - 19 \times 19) \times 21 \times 3 = 5040$. Here you need to be careful that is assumed that all the sub-regions are arranged in a plane, there are no sub-regions adjacent to each other on the top and bottom, and the number of boundary surfaces with the adjacent sub-regions is 4. The degree of freedom per node is 3 of displacement only. The degree of freedom of the course problem is that each sub-region is represented by one node per sub-region, and the degree of freedom per this node is displacement 3+ rotation 3= total 6. It is 6 times the number of regions. And it is calculated as $65536 \times 6 = 393216$.

12.4 Tuning of DomainFEM

In the kernel program of one process that cuts out the operation interval DomainFEM, the double-precision product (MatVec _product) of a double-precision symmetric dense matrix with a side length of 5040 and a double-precision vector with a length of 5040 is performed multiple times. In this kernel program, when only the lower triangle of the symmetric matrix is held, and when it is used as the upper triangle, optimization to be used and used is adopted. The data structure is a block-by-step type format that divides the lower triangular matrix into blocks of 6 rows and six columns. The data structure in the block of each sixth row of six rows is also a row priority format. When the loop structure is shown from the outside, there is a block-based row loop as first and a block-based column loop as second. Therefore, in order from the beginning of the lower triangular matrix, the calculation load per row increases as the loop rotation of the row in block units move on. Inside this double loop, processing is performed for blocks with 6 rows and 6 columns. This kernel program repeats the processing in units of blocks of 6 rows and 6 columns, and in the original source program, the processing for blocks of 6 rows and 6 columns was fully unrolled, so the loop of columns blocks was the axis for SIMD. In other words, the

SIMD axis was in the direction of straddling the blocks of 6 rows and 6 columns. For this reason, coupled with full unrolling, the continuity of data access have tends to be interrupted. As a result, hardware prefetching tends to fail, which has been a factor in performance degradation in the original source program. Therefore, the performance was improved by issuing a software prefetch instruction using the built-in prefetch function.

In the original source program, thread parallelization is performed by dynamic scheduling with chunk size 1 in a loop of rows in block units, and the low continuity of memory access due to the small chunk size of 1 and there was concern about the overhead associating with the dynamic scheduling. Therefore, the source program was modified so that the schedule method for looping rows in block units would be static scheduling in descending order folding cyclic method with chunk size 12, and the performance was improved.

In the original source program, the processing for blocks of 6 rows and 6 columns was fully unrolled, so the loop of columns in block units was the axis of SIMD. In other words, the SIMD axis was in the direction of straddling the blocks of 6 rows and 6 columns. At this time, the stride is $6 \times 6 = 36$, so it is an indirect SIMD access. In this case, data from multiple cache lines was required within the SIMD length range, and the busy rate of the L1D cache was high. In addition, many address calculation instructions for indirect SIMD access have been issued, which has been a factor in increasing the number of instructions.

Therefore, by rerolling the column loop (innermost loop) of the processing for the 6-by-6 block and using it as the SIMD axis, SIMD access was made continuous and the busy rate of the L1D cache was reduced. As a result, most of the fully unrolled operations are confined in the SIMD instruction, so the effect of significantly reducing the required number of registers can be expected. However, the loop length of the column here is 6, which is not divisible by the SIMD length of 4 of FX100 or the SIMD length of 8 assumed by Fugaku. Therefore, by specifying the optimization instruction line and using the masked SIMD instruction, SIMD processing including the fractional part was performed. If you reroll the column loop (innermost loop) of processing for a block of 6 rows and 6 columns and use it as the SIMD axis, the inner product calculation of the vector of length 6 will be SIMD processed, and the reduction operation within SIMD will be performed. This SIMD reduction operation requires more instructions than normal processing. Therefore, by changing the source program so that the result of multiplication during the inner product calculation is added to the working array, the SIMD reduction operation was reduced and the increase in the number of instructions was suppressed.

Even though the number of rotations of the innermost loop is not divisible by the SIMD length, by specifying the optimization instruction line, by using the masked SIMD instruction, SIMD processing including the fractional part was made. At this time, due to this masked part, when accessing the real memory in the i-th store and the i + 1-th load, the phenomenon called the excessive SFI (Store-Fetch-Interlock) that store oparation prevents load oparation has occurred. There is no problem in the logical movement of the program if load operation overtake store operation, from the viewpoint of safety, the hardware perform a sequential operation, which is also a commonly used method. As a result, pipeline operation is disrupted and performance is degraded. Therefore, a margin considering the SIMD width is added to the first dimension of the work array to avoid the excessive SFI, thereby avoiding performance deterioration.

Due to the improvement of the source program mentioned above, when the kernel program of one process that cut out DomainFEM is executed using 12 cores of FX100, the execution time per MatVec _product is reduced from 1.902×10^{-3} seconds to 1.091×10^{-3} seconds.

12.5 Tuning of CoarseMatVec

In the one-process kernel program that be cuted out the operation interval CoarseMatVec, the product of the double-double-precision format matrix of 24 rows and 393216 columns and the double-precision vector of length 393216 is repeatedly performed many times. Here, one double-double-precision format number is represented by two double-precision types, one of these two double-precision types is the double-precision type itself, and the other is the value using for he precision extension.

The original source program used a function to calculate the double-double-precision form product of a double-double-precision form matrix and a double-double-precision form vector. The matrix data structure was in a row-priority format, and the loop configuration was a matrix row loop from the outside, followed by a matrix column loop.

Inside this double loop, so the double precision form multiplication of the two double precision form numbers and the double precision form addition to the work variable for sum calculation of the result were performed sequentially, the inner most loop as a column of a matrix was not SIMDized. The loop of the column of the matrix is divided equally into the number of threads and thread parallelization that each calculation is assigned to each thread is performed. Therefore, after the processing in each thread is completed, it had necessary to calculate the sum of a work variables of double-double-precision form for each thread.

In addition, since the matrix row loop was unrolled in three stages, L1D cache was prone to cache line conflicts. Furthermore, since the vector which is originally a double-precision type is formalized in double-gouble-precision and then the product with the matrix is calculated, the number of data streams increases by one, and cache line conflict is more likely to occur.

Therefore, the performance of this arithmetic kernel program was improved by changing the program structure. First, by stopping the unrolling of the loop in the row of the outermost matrix, we alleviated the situation where cache line contention is likely to occur in the L1D cache. By making the loop of the rows of the matrix a thread parallelization loop, it is no longer necessary to calculate the sum of the work variables of the double-double-precision form for sum calculation for each thread later. By calling a subroutine that performs the double-double-precision format inner product of the double-doublke-precision format vector and the double-precision type vector inside the loop of the rows of the matrix, the double-double-precision format expansion of the double-precision vector, which was originally unnecessary is stopped and reduced the number of data streams by one to mitigate situations where cash line conflicts are likely to occur. In order to utilize the SIMD arithmetic unit to perform the processing related to the double-double-precision format inner product of the double-double-precision format vector and the double-precision vector, the inside of the subroutine make be divided into a multiplication part and a summation part. In the multiplication part, only the double-double-precision format multiplication of the elements having the same element numbers of the double-double-precision format vector and the double-precision type vector is performed, and the result is stored in a work array formating double-double-precision. This process can be performed by SIMD.

In the summation section, the double-double-precision form sum of all the elements of the work array for sum calculation of this double-double-precision form is calculated by the first half second half two-division vector sum repetition method. Here, the first half second half two-division vector sum repetition method is one of the methods for calculating the sum of vector elements utilizing the SIMD calculator. When calculating the sum of the vectors of one element number 2N, the first half element numbers 1 to N and the second half element numbers N + 1 to 2N are regarded as one independent vector, and these two Calculate the sum of the vectors. The calculation of the sum of these two vectors can be SIMD processed. As a result, one vector with the number of elements N is obtained. Subsequently, the first half element numbers 1 to

N / 2 and the second half element numbers N / 2 + 1 to N, are regarded as one independent vector. Calculate the sum of these two vectors. The calculation of the sum of these two vectors can also be SIMD processed. As a result, one vector having N / 2 elements is obtained. The above process is repeated until the number of vector elements becomes 1. This is a method called the first half second half two-division vector sum repetition method, which calculates the sum of vector elements utilizing a SIMD calculator. However, if the program is written using a normal loop statement such as the "do" statement without using the "do concurrent" statement newly defined in the Fortran 2008 standard, it is necessary to prepare two working arrays in double-double-precision format. In that case, the summation part can be SIMD-processable, but there is a weakness that both the amount of data and the number of data streams are increased with respect to loading. Since Fugaku conforms to the Fortran 2008 standard, by adopting the "do concurrent" statement, it was nolonger need to prepare two arrays for summation calculation in double-double-precision format even when writing the source program according to the language. Then, by rewriting the source program to use only one double-double-precision format, and reducing both the amount of data and number of data streams to be loaded, and reduced execution time. Furthermore, in order to keep the working array of the double-double-precision format for summention in the cache, the strip mining optimization was applied to the entire inner product calculation part including both the multiplication part and the sum part. In addition, we also searched for the strip length that minimizes the execution time when executing the kernel program of one process that cut out CoarseMatVec using 12 cores of FX100. Due to the improvement of the above source program, when the kernel program of one process that cut out CoarseMatVec is executed using 12 cores of FX100, the execution time of CoarseMatVec is reduced from 22.31×10^{-3} seconds to 2.41×10^{-3} seconds. This improved source program is used for post-K computer performance estimation in the detailed design (3).

In the detailed design (3), the source program of CoarseMatVec was further improved. Specifically, in the above, the processing related to the double-double-precision format inner product of the double-double-precision format vector and the double-precision vector was divided into the multiplication part and the summation part, but this should be stopped and the following processing should be performed. First, for the first B elements of each of the double-double-precision form vector and the double-precision type vector for which the inner product is to be calculated, the double-double-precision form product of the elements having the same element number is calculated and store the result to work array having length B. Next, for the next B elements of each of the two vectors for which the inner product is to be calculated, the double-double-precision formal product of the elements having the same element number is calculated and added the result to the work array having length B. After that, similarly, for the two vectors for which the inner product is to be taken, the result of the product of the vector elements having the same element number is added to the work array for sum calculation. This processes repeat until reach the end of the two vectors. Finally, the sum of all the elements of the work array of length B is calculated by the above-mentioned first half and second half two-division vector sum repetition method. In the detailed design (3), the size of B is tentatively set to 896. As a result of improving the source program of CoarseMatVec by detailed design (3), when the kernel program of one process that cut out CoarseMatVec is executed using 12 cores of FX100, the execution time per CoarseMatVec is reduced to 1.95×10^{-3} seconds.

12.6 Base line measurement on K

We measured the execution time of one case of the target problem that was processed by 32768 process on 32768 node of K. The measurement section in BDD iteration is divided into eight calculation sections and three communication sections. The results are shown in table 12.1.

Table 12.1: Execution time using 32768node(32768process) on K

section name	Execution time of 1BDD iteration
DomainFEM	8.36972msec
CoarseMatVec	17.8402msec
CollectRhs.P3	0.26833msec
QuadMatVec.P1	1.04247msec
Exec.BlancingL	0.76123msec
Exch.Buffering	0.52835msec
DomainFem.P1	0.31601msec
Calc._Others	0.24571msec
Exchange	0.27125msec
Allreduce	0.20164msec
Allgather	3.84919msec
Total	33.69414msec

When calculate the performance improvement ratio fromK to Fugaku, we have used the total of these 11 measurements as the measurement value of the ADVENTURE execution time(K baseline value) at K.

This K computer's baseline value (33.69414 ms per BDD iteration) was roughly the same as 33 ms per 1 BDD iteration that was measured by the ADVENTURE developer on 32768 nodes in K prior to the start of the Flagship 2020 project.

12.7 Performance measurement on Fugaku

12.7.1 Tuning of MulMatVec

The new code provided in March 2020 allocates a row block pair when the lower triangular matrix is folded back only once at the center row number to the threads, and equalizes the processing amount per thread. In the symmetric matrix storage method, the lower triangular matrix excluding the diagonal elements is stored in the skyline, and the diagonal elements are stored in another array. These two points are different from the code used so far. The following tuning was performed on this code.

(1) :Equalize memory access costs by performing loopback cyclic allocation on a block-by-block basis

(2) :Change the number of unrolls to 6

Next, the result of CPU analysis report when the tuning code is executed 100 times is shown in the figure 12.1. The access wait time and instruction commit cycle are almost even among threads, which is exactly what we were aiming for. The calculation wait time is a little noticeable, but according to the compile list, it is perfomed software pipelining and not occurring spill. Also, the memory throughput is close to the upper limit.

12.7.2 Tuning of MatVec_Rect_Quad

In Regions 1 and 2 that perform quadruple precision arithmetic matrix vector product of the new code provided in March 2020, the addition calculation part in the inner product calculation

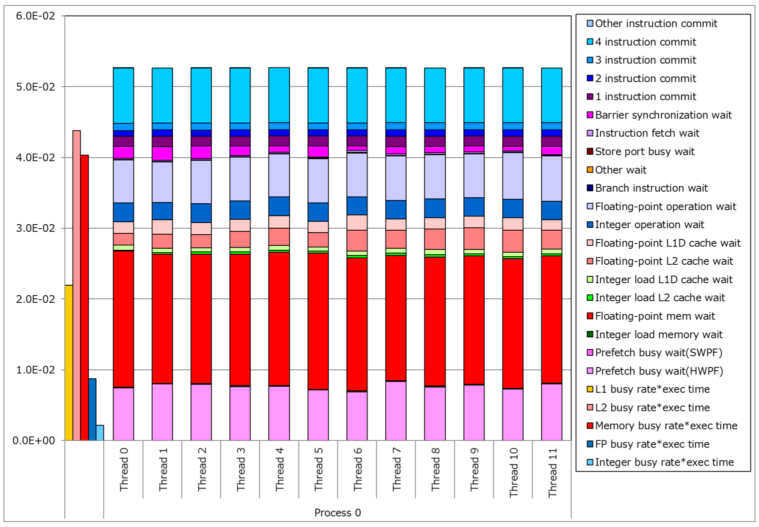


Figure 12.1: Result of CPU performance report of MulMatVec

is the code that manually unrolled the innermost loop. In the same code, the inner product calculation is blocked, and the multiplication result is passed from the multiplication part to the addition calculation part to reduce memory access.

After abolishing the manual unroll of the innermost loop for this code, The blocking method has been improved as follows.

(1) Store the multiplication result in the first block in the working array.

(2) Multiply in the i-th block, and add the result of the i-1st block and the elements. Repeat this for the number of blocks-1 time.

(3) In-block reduction addition calculation by Recursive Halving is performed for the block of the final result.

12.7.3 Other tuning

As another tuning, the processing corresponding to 4 and 5 in the figure 12.2 described later was moved before the BDD iteration. In addition, the processes 11 and 12 were converted to omp, and the processes corresponding to 23 were deleted.

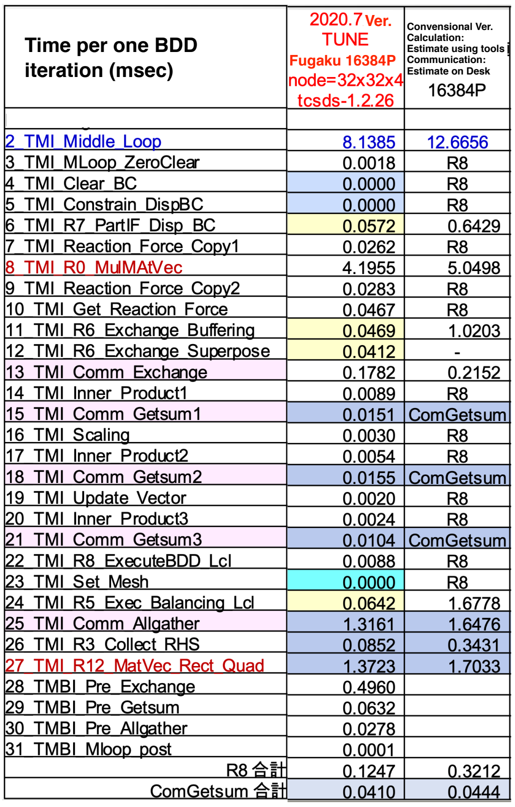
12.7.4 Performance measurement result on Fugaku

A comparing result of the performance measurement results at Fugaku and the performance estimation results at Fugaku indicate in the figure 12.2.

The correspondence between the section name shown in the table 12.1 and the section number shown in the figure 12.2 indicate in the table 12.2.

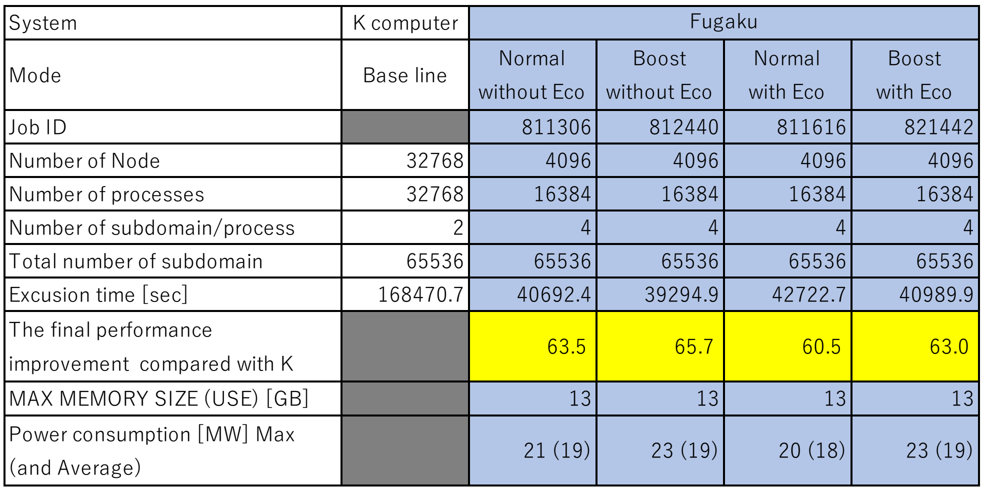
It can be seen that the speed is increased by the tuning of MulMatVec, the tuning of MatVec_Rect_Quad, and other tuning effects.

Figure 12.3 shows the final performance improvement and power consumption compared with K. In the end, 63 times performance improvement is obtained normal and without eco.



Time per one BDD iteration (msec)	2020.7 Ver. TUNE Fugaku 16384P node=32x32x4 tcsds-1.2.26	Convensional Ver. Calculation: Estimate using tools Communication: Estimate on Desk 16384P
2_TMI_Middle_Loop	8.1385	12.6656
3_TMI_MLoop_ZeroClear	0.0018	R8
4_TMI_Clear_BC	0.0000	R8
5_TMI_Constrain_DispBC	0.0000	R8
6_TMI_R7_PartIF_Disp_BC	0.0572	0.6429
7_TMI_Reaction_Force_Copy1	0.0262	R8
8_TMI_R0_MulMAtVec	4.1955	5.0498
9_TMI_Reaction_Force_Copy2	0.0283	R8
10_TMI_Get_Reaction_Force	0.0467	R8
11_TMI_R6_Exchange_Buffering	0.0469	1.0203
12_TMI_R6_Exchange_Superpose	0.0412	-
13_TMI_Comm_Exchange	0.1782	0.2152
14_TMI_Inner_Product1	0.0089	R8
15_TMI_Comm_Getsum1	0.0151	ComGetsum
16_TMI_Scaling	0.0030	R8
17_TMI_Inner_Product2	0.0054	R8
18_TMI_Comm_Getsum2	0.0155	ComGetsum
19_TMI_Update_Vector	0.0020	R8
20_TMI_Inner_Product3	0.0024	R8
21_TMI_Comm_Getsum3	0.0104	ComGetsum
22_TMI_R8_ExecuteBDD_Lcl	0.0088	R8
23_TMI_Set_Mesh	0.0000	R8
24_TMI_R5_Exec_Balancing_Lcl	0.0642	1.6778
25_TMI_Comm_Allgather	1.3161	1.6476
26_TMI_R3_Collect_RHS	0.0852	0.3431
27_TMI_R12_MatVec_Rect_Quad	1.3723	1.7033
28_TMBI_Pre_Exchange	0.4960	
29_TMBI_Pre_Getsum	0.0632	
30_TMBI_Pre_Allgather	0.0278	
31_TMBI_Mloop_post	0.0001	
R8 合計	0.1247	0.3212
ComGetsum 合計	0.0410	0.0444

Figure 12.2: FugakuResult of Performance measurement and estimation



System	K computer	Fugaku			
Mode	Base line	Normal without Eco	Boost without Eco	Normal with Eco	Boost with Eco
Job ID		811306	812440	811616	821442
Number of Node	32768	4096	4096	4096	4096
Number of processes	32768	16384	16384	16384	16384
Number of subdomain/process	2	4	4	4	4
Total number of subdomain	65536	65536	65536	65536	65536
Excusion time [sec]	168470.7	40692.4	39294.9	42722.7	40989.9
The final performance improvement compared with K		63.5	65.7	60.5	63.0
MAX MEMORY SIZE (USE) [GB]		13	13	13	13
Power consumption [MW] Max (and Average)		21 (19)	23 (19)	20 (18)	23 (19)

Figure 12.3: the final performance improvement on Fugaku compared with K

12.8 Authors

The authors and the contributors of this chapter are listed below. The affiliations in the parentheses are the ones at the time of participation.

Table 12.2: Correspondence between the section at the time of performance estimation and the section at the time of performance measurement

Section name of performance estimation	Section name of performance measurment
DomainFEM	8
CoarseMatVec	27
CollectRhs.P3	26
QuadMatVec.P1	26
Exec.BlancingL	24
Exch.Buffering	11
DomainFem.P1	6
Calc._Others	R8 total
Exchange	13
Allreduce	ComGetsum total
Allgather	25

12.8.0.0.1 Authors

- Kazuo Minami (RIKEN)
- Kengo Miyamoto (RIKEN)

12.8.0.0.2 Co-Authors in alphabetical order

- Hiroshi Kawai (Toyo U.)
- Tomonori Yamada (U. Tokyo)
- Shinobu Yoshimura (U. Tokyo)

Chapter 13

Codesign of FrontFlow/blue

13.1 Codesign overview

This priority issue 8 aims at the development of computational science and technology that will bring about changes in Japan's future manufacturing. It includes changes in the performance of the product itself by computer simulation, changes such as streamlining the product development process, and dissemination to actual manufacturing sites such as industry. In the field of manufacturing, all applications must handle various shapes as analysis targets, and many applications use unstructured grids. When the unstructured grid is used, the position of the data to be referred to in the memory space cannot be obtained from a simple rule. In order to find the position of the data you want to refer to, you need to refer to the list that describes the adjacency, which tends to increase the amount of memory transfer. Therefore, the required B/F is high, and the processing of the core part of the FEM application is similar. FrontFlow/blue (or FFB) is positioned as an application that represents a group of the FEM applications that use unstructured grids and request high B/F value for indirect access via lists which frequently occurs. It contributes to codesign from the viewpoint of improving the performance of calculations that require high and high memory throughput. If FrontFlow/blue can operate at high speed on Fugaku, it will be easy to horizontally deploy the technology to other applications.

So far, FFB has a track record of executing flow analysis on a scale of up to 32 billion elements using about 40,000 nodes in K. If the calculation of hundreds of billions of elements can be realized using Fugaku, it will be possible to perform quasi-direct numerical calculation of turbulence with Reynolds number of 10^7, and to perform the high-precision fluid calculation (the internal flow of hydraulic machinery and the aerodynamic disturbance of the actual vehicle, etc) with the same accuracy as loop test and wind tunnel experiment. In addition, using this highly accurate analysis result as reference data, it will be possible to execute multipurpose design optimization calculation for 10,000 case (100 case $\times$ 100 generation) in a few days with Fugaku. It is possible to realize a wide range of multipurpose design optimization technologies with higher precision than before, and it can contribute to the sophistication of related manufacturing designs such as turbomachinery. In addition, since the scale of this problem is the largest in the priority task 8, it is possible to carry out the target problem in other themes in the task. FrontFlow/blue is a fluid simulation code based on the finite element method. The matrix is a sparse matrix and an unstructured grid is used and the memory access pattern is random access that differs depending on the input data. The calculation of the flow field in FrontFlow/Blue is performed by obtaining the velocity predictor by calculating the convection/diffusion term, obtaining the pressure from the velocity predictor, and determining the velocity by modifying the velocity predictor from the pressure. Velocity predictor and pressure solving are implicit.

The conjugate gradient (CG) method is used as the implicit solver, and the main operation is the product of a sparse matrix and a vector. Even if it is kernels that are not implemented so that the processing content can be clearly identified as a sparse matrix vector product, such as the gradient calculation routine (GRAD3X) and divergence calculation routine (FLD3X2), and the motion equation routine (VEL3D1), although the whole stiffness matrix is not generated from the matrix defined in element units, the value corresponding to the whole matrix is calculated using the adjacency information of the elements and the small matrix defined in each element, so-called Element by Element. Since the method of abobe mentioned is used, it can be said that all arithmetic kernels are basically the product of sparse matrices and vectors. The product of a sparse matrix and a vector is a process that mainly depends on the memory throughput. Because the memory access is discontinuous due to the data structure that the finite element method presupposes, not only the matrix value but also the connectivity information must also be loaded from the memory. Therefore, the main purpose of improving performance is to improve memory throughput. With this in mind, we conducted a codesign study on each of the following items. The main items of the specific contents of each of the above items are shown in the following sections. Unless otherwise specified, the values for shortening the processing time by codesign shown below are all the measured values when the test questions are executed with K at present.

- Gradient Calculation Routine (GRAD3X) Main Loop
- Divergence calculation routine (FLD3X2) main loop
- Equation of Motion Routine (VEL3D1) Main Loop
- Sparse matrix vector product routine (CALAX) main loop
- Copy of array
- Zero clear of array
- Non-threaded parallel routine(CRSCVA, CLRCRS, NODLEX, DGNSCL)
- List vectorization of loops containing if
- Eliminating type conversion loops

13.2 Gradient calculation routine (GRAD3X) SIMDization of main loop, software pipelining, algorithm change for the purpose of reducing the number of load stores

GRAD3X calculates the value on the node from the gradient (∇p) of the pressure value defined in the center of the element. A loop is rotated by the elements, and the result of multiplying the pressure value of each element by a coefficient is added to the node referenced by the element. Figure 13.1 is a schematic diagram of this process. The square frame in the figure represents the element, and the black circle indicates the node. Each element has a centrally defined pressure P_n and factor value $C_{n,v}$ depending on the number of vertices. For example, element 1 refers to four nodes, nodes 1,2,12,11 as vertices and has four coefficients $C_{1,1}$,$C_{1,2}$,$C_{1,3}$,$C_{1,4}$. The arrow means the process of adding the value obtained by multiplying the pressure P_n and the coefficient $C_{n,v}$ to the corresponding node. Since processing is performed for each element, addition to a specific node occurs multiple times at irregular intervals. In the example of this figure, the addition to the node 12 circled in red in the figure occurs a total of four times when processing elements 1, 2, 11, and 12. The number of additions and the interval of occurrence are random depending on the input data. Since the value of the same node is updated from multiple elements, there is a data dependency and the thread parallelization, the SIMDization, and the software pipelininning cannot be performed as it is.

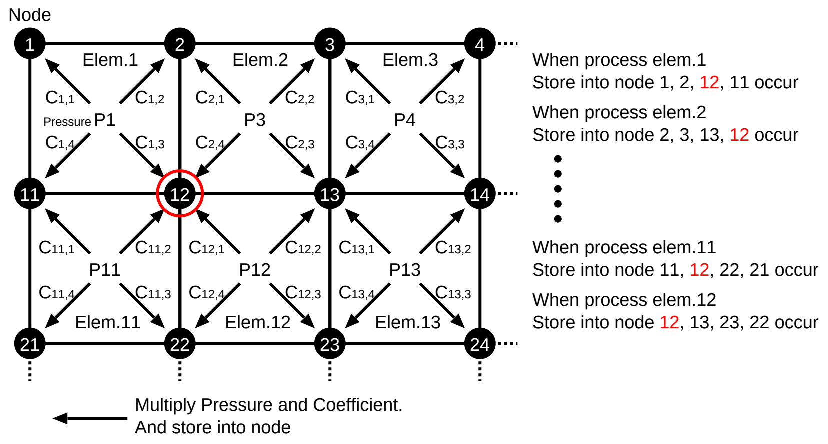
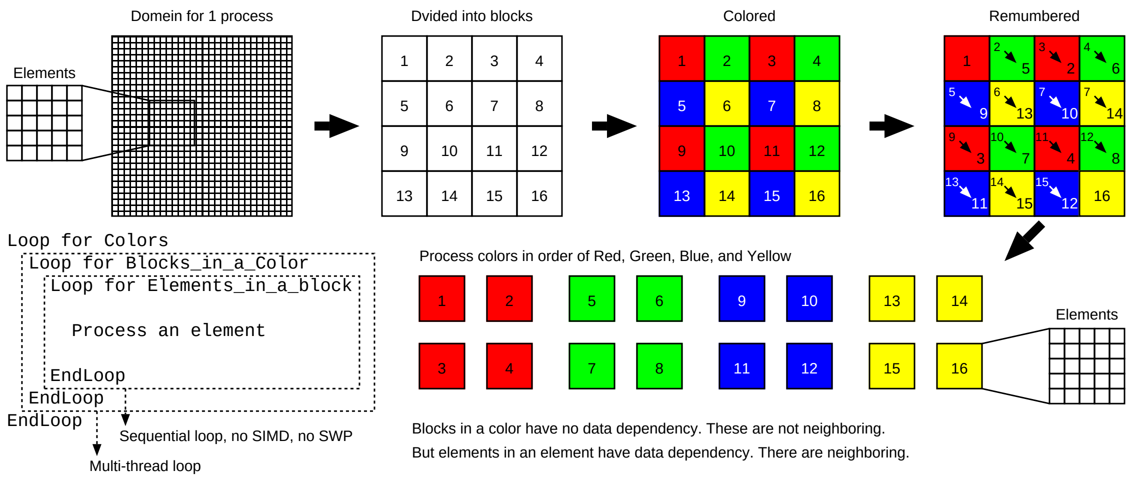

Node
Elem.1
Elem.2
Elem.3
Pressure P1
P3
P4
P11
P12
P13
Elem.11
Elem.12
Elem.13
When process elem.1
Store into node 1, 2, 12, 11 occur
When process elem.2
Store into node 2, 3, 13, 12 occur
When process elem.11
Store into node 11, 12, 22, 21 occur
When process elem.12
Store into node 12, 13, 23, 22 occur
Multiply Pressure and Coefficient.
And store into node

Figure 13.1: Processing contents and situation of store process of GRAD3X

Therefore, in this implementation, in addition to the domain decomposition for process parallelism, the calculation area is divided into subdomain within the process to avoid data dependency. Figure 13.2 is a schematic diagram of this process. The area processed by each process is divided into multiple blocks, each with approximately the same number of elements, coloring is performed so that blocks that are not adjacent to each other are not included in the same color, and the numbers are arranged so that the block numbers are continuous in each color. Although not shown in the figure, the element numbers are also sorted so as to be continuous within each block. The lower part of the figure is the final data structure. The blocks in each color are not adjacent to each other so there is no data dependency, but the elements in each block are adjacent to each other so that the elements has data dependency . The loop structure is a triple loop, the outermost loop is a loop of the collar, the middle loop goes is a loop of the block in the collar, and the innermost loop is a loop of the elements inside the block. Thread parallelism is done in an intermediate loop that runs in blocks that are not adjacent to each other and have no data dependencies. Since the innermost loop is a loop that rotates around elements adjacent to each other, it is data-dependent, and SIMDization and software pipelining cannot be applied.

In an implementation that directly coloring elements without using such block division, the loop structure has a double loop structure of a color's loop and a element's loop, and there is no data dependency in the innermost loop. Therefore, thread parallelization, SIMDization, and software pipelining can be applied in the innermost loop. But the locality of memory access is poor and the time density for reusing reusable data on the cache is coarse and the performance was not good. Since the processing of this loop strongly depends on the memory throughput, it was decided that it is better to improve the memory throughput than to apply SIMDization and software pipelinining, so it was implemented now. Even so, the memory throughput was not sufficient, so I have futher improved this kernel.

This process accounted for 95 seconds of the total execution time of 394 seconds, and memory access-related waits accounted for a large proportion of that time. Examining the breakdown of execution instructions, there were many the store operation. This was due to the above-



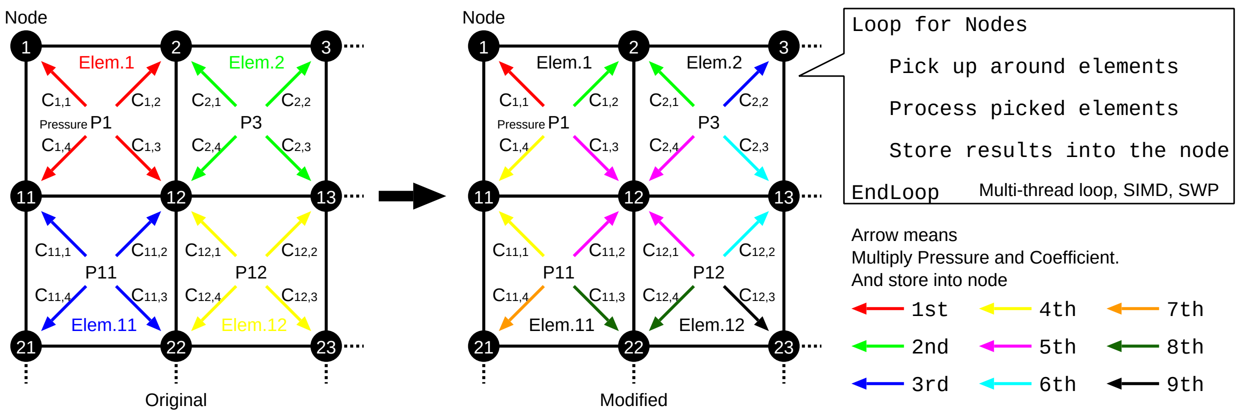
```
Loop for Colors
 Loop for Blocks_in_a_Color
  Loop for Elements_in_a_block

   Process an element

  EndLoop
 EndLoop
EndLoop
```

Domein for 1 process
Elements
Dvided into blocks
Colored
Remumbered
Sequential loop, no SIMD, no SWP
Multi-thread loop
Process colors in order of Red, Green, Blue, and Yellow
Elements
Blocks in a color have no data dependency. These are not neighboring.
But elements in an element have data dependency. There are neighboring.

Figure 13.2: Loop structure of GRAD3X

mentioned characteristic that if a store to a node referenced by multiple elements is processed in a loop that goes around the element, a store to the same node is required many times.

Therefore, the loop structure was changed to rotate at the node. In this implementation, the contributions from all the elements that refer to that node are calculated and aggregated at once for each node, so that each node can be stored only once.



```
Loop for Nodes

  Pick up around elements

  Process picked elements

  Store results into the node

EndLoop        Multi-thread loop, SIMD, SWP
```

Figure 13.3: Situation of store process of GRAD3X before and after the loop structure change

Figure 13.3 shows a schematic diagram of the state of the store of before and after the loop structure change and the source code after the change. Before the change, the processing is performed in the order of the elements, so when processing element 1, a store to nodes 1,2,11,12 occurs, and when processing element 2, a store to nodes 2,3,13,12 occurs.

After the change, the processing is performed in the order of nodes. For example, when processing node 2, as shown by the green arrow in the figure, the contributions from the two elements using node 2 to node 2 are collectively calculated and stored in node 2. After that, the value of node 2 is not updated to node. In addition, since the store order is sequential and there is no need for a complicated structure for thread parallelization because there is no data dependency. Futhermore, SIMDization and software pipelining can be applied to the loop at the same time. As a result, the number of load/store instructions was reduced by 42%, the number of store instructions was reduced by 87%, and the execution time was improved from 95 seconds to 41 seconds.

The memory throughput at this time improved from 23.2GB/s to 47.2GB/s that is almost the effective upper limit for K. Fugaku It is estimated that the same process can be executed in about 14 seconds or so in the measurement using the actual machine. At this time, the memory throughput at Fugaku was 146.3 GB/s. This value indicates that there is still room for improvement compared to the effective memory throughput upper limit of 205 GB/s for Fugaku 1CMG. In the implementation of this kernel, in the innermost loop, the array whose size is 3 is accessed 3 times like as A (1, I, J), A (2, I, J), A (3, I, J). It is a continuous access in the memory space, but the compiler cannot grasp that it is continuous at compile time, and issues an inappropriate load instruction (the gather load instruction) instead of a continuous SIMD load instruction.

Therefore, here, by changing the shape of the array from A (1, I, J) to A (I, J, IP), the continuous SIMD load instruction is issued. As a result, memory throughput has increased to 170GB/s.

(*)Currently, the compiler has been improved so that it can issue appropriate load instructions (Structure load instructions) even with access patterns such as A (1, I, J), A (2, I, J), and A (3, I, J).

13.3 Divergence calculation routine (FLD3X2),apply of SIMDization of main loop, stream number adjustment, use of sector cache

This application is a code that has been used for a long time, and as an implementation to avoid bank conflicts on vector supercomputers, it was originally declared that the size of the first dimension of a multidimensional array should be 9, so 9 changed to 8. With such an implementation, since the data is transferred in units of cache lines, the 9th array component that is not actually used is also transferred, so 1/9 of the memory bandwidth is wasted. Also, in FrontFlow/blue, all the arrays handled in the subroutine use the memory area allocated in the main routine, and the array is passed to the subroutine together with the variable indicating its size. Since this subroutine receives a two-dimensional array whose first-dimensional size is 8, the size is already fixed at 8 at compile time, but since the array size is specified by a variable, the compiler has recognized the first-dimensional size as variable. Due to this recognition of this compiler, efficient instructions were not generated. Therefore, I have gave the compiler information that the array size is fixed at 8. The main loop of this subroutine has a double loop structure with an innermost loop that rotates 8 times in the most basic implementation that realizes the algorithm, but in reality it was implemented single loop structure unrolled the innermost loop into 8 oparation. It is desirable that the innermost load instruction can be executed by the SIMD instruction, but because of the unroll implementation, the array index is specified as an immediate value in the source code, and the SIMD load instruction cannot be applied continuous access by the compiler. Therefore, it was re-implemented in a loop coding again to promote SIMD loading.

In addition, the number of arrays in the loop was adjusted. Since the cache capacity is smaller than the memory capacity, it often happens that data in different memory areas are placed in the same cache line. The cache is managed in units called ways together with the line number. For example, 2-way cache can hold data from two different addresses on the same line number. However, since the number of ways is generally small, as the number of arrays referenced in the loop increases, the array data on the same line increases, and as a result, the cache tends to overflow. In this loop, multiple arrays are referenced, but in particular, three two-dimensional arrays are referenced with the same access pattern, and cache line contention is likely to occur. Therefore, taking advantage of the feature that the access pattern is the same, we reimplemented three 2D arrays as one 3D array and improved the cache utilization efficiency.

Furthermore, the parameter of the sector cache was adjusted and the execution time improved from 61 seconds to 48 seconds. The memory throughput at this time improved from 40.3GB/s to 42.5GB/s. This is a value that is almost equal to the effective upper limit of K.

When this tuning version code was executed on Fugaku and the CPU performance analysis report was collected, imbalance between threads occurred. From the viewpoint of calculation amount, imbalance between threads does not occur, but the cause is access to the array fgp in the list-accessed array, and the cache miss rate changes for each thread.

Since the values in the list depend on the input data, as a workaround for this phenomenon, a directive that promotes prefetching of the array fgp was inserted, and an appropriate prefetch distance was determined by a parameter study. This eliminated the imbalance between threads. As a result, it is estimated that the same process can be executed in about 18 seconds in Fugaku. At this time, the memory throughput at Fugaku was 125GB/s. This value indicates that there is still room for improvement compared to the effective memory throughput upper limit of 205GB/s for Fugaku 1CMG.

13.4 Equation of Motion Routine (VEL3D1) Examination of performance improving of the main loop

The equation of motion routine (VEL3D1) that remained unimproved has a target calculation time of less than 10 seconds from the viewpoint of memory throughput, but it is currently about 30 seconds. We are continuing to consider improvements. Compared to other major loops, this loop has a large and complicated loop body, and has a large number of referenced arrays and temporary variables, so register spills occur. Occurrence of the register spill is a serious concern in Fugaku because it has fewer registers than K, and we investigated spill reduction using the currently available compilers for Fugaku for various implementation patterns. In addition, the loop structure of this routine is the same as the gradient calculation routine before improvement. Therefore, we examined the improvement of changing the loop structure shown in 13.2 to a nodal loop. As a result, it took 32 seconds to execute, but it improved to 12 seconds. Memory throughput is still low, so there is room for further improvement. The memory throughput at K at this time improved from 11.3GB/s to 34.3GB/s. The value is still smaller than the effective upper limit of K, but unlike other kernels, the ratio of operations is large for this kernel, and the B/F ratio is not so large about 1 so it is staying to this level. Fugaku It is estimated that the same process can be performed in a little less than 7 seconds in the measurement using the actual machine. At this time, the memory throughput at Fugaku was 74.6GB/s.

13.5 Sparse matrix vector product routine (CALAX) main loop

Since this kernel performs the product of a sparse matrix and a vector, it is a double loop in which the outer loop is a loop of row and the inner loop is a loop of a non-zero element in each row. The rotation speed of the innermost loop is at most 30. In the implementation for K, the innermost loop that rotates 30 times is unrolled, the index when accessing the array is not specified by the loop variable, and it was specified as an offset value from the loop variable such as i+1, i+2, i+3. Originally, it was a continuous access on the memory space, but the compiler could not grasp it and issued an inappropriate load instruction (the gather load instruction) instead of a continuous SIMD load instruction, and the memory throughput was low (about 94 GB/s).

Therefore, the unrolled loops by 30 were re-looped again. As a result, the innermost loop becomes a loop that rotates 30 times, and the immediate value does not appear in the index of array access, and the compiler grasps that it is continuous access, and a continuous SIMD load instruction is issued. As a result, the process that used to take about 5.3 seconds on Fugaku can now be completed in about 2.8 seconds. The memory throughput at this time was greatly improved from 94GB/s to about 182GB/s. This is a sufficient value compared to the effective upper limit of 205GB/s of Fugaku.

13.6 List vectorization of loops containing if statement

This is the second loop of the gradient calculation routine (GRAD3X) described above. In most input data, the number of upper elements of a certain node is about 8 at the maximum, but when the shape is partially complicated, the number of upper elements is 8 or more. In order to improve execution efficiency, GRAD3X divides the main calculation loop into a loop that calculates the contribution from the 8th higher-level element and a loop that calculates the contribution from the 9th and subsequent higher-level elements. This outside loop is the node loop, and there is an if statement that determines whether the number of upper elements is 8 or more for each node. When it is only for nodes with 8 or more upper elements, execute a loop that calculates the contribution of the 9th and subsequent higher-level elements to the gradient value. This innermost loop is SIMDized and software-pipelined by the compiler. Since the test problem does not include nodes with 8 or more high-order elements, this loop should be a ”do nothing loop”, but in reality, it took about 8 seconds that is about 2% of the total execution time of 394 seconds. As a result of the investigation, it was found that the waiting for the branch instruction occupies most of the processing time of this loop. This is because the actual operation of this loop is not ”do nothing”, but it is determined whether the number of upper elements is 8 or more for all nodes, and the loop that turns at the node is the innermost loop. This is because SIMD and software pipelines are not applied and instruction execution efficiency is poor. In such a code that includes if in the loop, if the distribution where if is true does not change every time when it is executed repeatedly, it is effective to create a list (list vector) that collects only the indexes for which if is true in advance, and perform list vectorization that processes only the objects included in the list vector. This loop was repeatedly executed 11880 times in the test problem, and the number of high-order elements at each node did not change throughout the program execution, so list vectorization was applicable to this loop. As a result, in the test problem, the upper list vector became empty, so the execution time improved from about 8 seconds to almost 0 seconds. It should be noted that the time of the list vector creation process that collects only those whose if is true, which is executed only once during program execution, can be regarded as almost 0 seconds.

13.7 Baseline measurement on K

13.7.1 Terget problem

Here, considering large-scale analysis of turbomachinery, flow around the hull, aerodynamics of the actual vehicle, etc., assuming a large-scale single problem type calculation that uses the entire computer to calculate one problem about as 674.8 billion elements 10,000 time steps. In addition, since FrontFlow/blue is a general-purpose application, it is necessary to handle various input data, but in this evaluation, the simple shape data creating by a program that automatically generates input data using only 2 parameters(the number of elements and processes) is used.

Since FrontFlow/blue focuses on large-scale single-processing calculations, we estimated the time required to solve a terget problem using the entire K system. The problem scale assumed in Fugaku is a calculation of 100,000 time steps dealing with about 674.8 billion elements in the entire system. The scale of this problem is determined by the upper limit of the amount of memory that can be used by the application in each process on Fugaku, which is about 1.06 million elements per process on Fugaku. FrontFlow/blue is a general-purpose application, and even if the problem scale is the same, the amount of calculation and communication load change depending on the element type (hexa element, tetra element, etc.) and the aspect of domain division. In this study, the number of elements per process and the shape generated by the input data automatic generation program with the total number of processes as parameters are cubic shapes consisting of only hexa elements, and the amount of calculation for each process is equal. Each process is assumed to have a shape in which rank numbers are adjacent only to adjacent processes, except for the last two processes. To calculate a problem of about 674.8 billion elements in K, it is necessary to handle about 8.14 million elements per process, but such a problem size is not feasible in K due to lack of memory. Therefore, in this study, the calculation of 1.06 million elements per process, which is the problem size that can be executed in K, is executed in 100 time steps in 24 processes. And, the calculation time in 8.14 million elements per process is estimated from these data. Hereinafter, execution of 1.06 million elements per process, 24 processes in total, and 100 time steps is referred to as test execution.

13.7.2 Result

The execution time and the breakdown when the problem of about 674.8 billion elements, which is the assumed problem in Fugaku, is executed in K are shown. The value here is the linear extrapolation of the time of 100 time steps to 100,000 time steps. In the target problem, the convergence test conditions for the iterative calculation are strictly set, so the iterative calculation always goes up to the upper limit specified in advance. Therefore, the amount of calculation is constant in every time step, and the calculation time for 100,000 time steps is 1000 times that of 100 time steps.

Table 13.1: Base line execution time of FrontFlow/blue on K (100000time step)

Problem size	time (h)		
	total	calculation	communication
about 674.8Billion	838	824	14.2

13.8 Performance measurement on Fugaku

13.8.1 Tuning the gradient calculation routine (GRAD3X) on Fugaku

Improvement of to hide the cache access latency by ocl prefetch instruction line for list access reference was performed. The CPU performance analysis report before and after tuning is shown in Figure 13.4 and Figure 13.5. The execution time of the relevant section decreased from 6.52 seconds to 5.38 seconds. In addition, the elapsed time is approaching the memory busy time, and the performance up to the upper limit of the memory bandwidth is obtained.

Further tuning was performed on other parts of GRAD3X. In pre-tuning code, instead of multiple structures load, store instructions, the gather load and scatter store instructions were

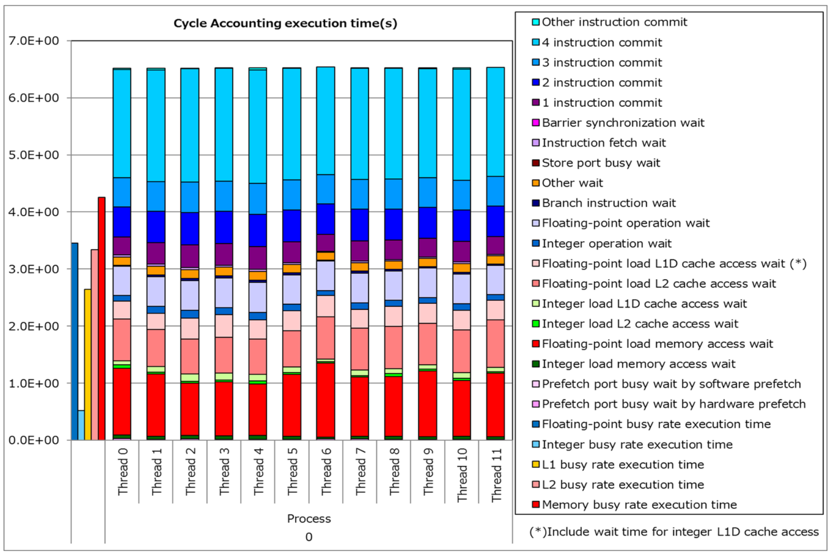


Figure 13.4: Performance of GRAD3X of before tuning

issued. As a result of investigation, the cause was that it referred to the same an array as the definition. Therefore, the performance is improved by referring to another array called FXYZ _tmp and changing it to a form that can issue multiple structures load and store instructions.

The CPU performance analysis report before and after tuning is shown in Figure 13.6 and Figure 13.7. The execution time of the relevant section decreased from 1.75 seconds to 1.24 seconds. In addition, the elapsed time is approaching the memory busy time, and the performance up to the upper limit of the memory bandwidth is obtained.

13.8.2 Tuning the divergence calculation routine(FLD3X2) on Fugaku

The prefetch operation made to apply to the innermost loop by the ocl instruction line, and the scaler prefetch instruction was changed to the gather prefetch instruction to improve performance. It also has the effect of changing the list access array:NODE from non-SIMD access to SIMD access. The CPU performance analysis report before and after tuning is shown in Figure 13.8 and Figure 13.9. Due to the change of the prefetch instruction to the gather prefetch instruction and the SIMD conversion of the list access array: NODE, the execution time of the corresponding section was reduced from 9.15 seconds to 6.67 seconds. Non-SIMD load instructions decreased from $6.06E + 10$ to $1.02E + 10$, and Single vector contiguous load instructions increased from 2.55E10 to 3.18E10. The total number of instructions and the 4-instruction commit time decreased, and the L1 busy rate increased to 74.5%.

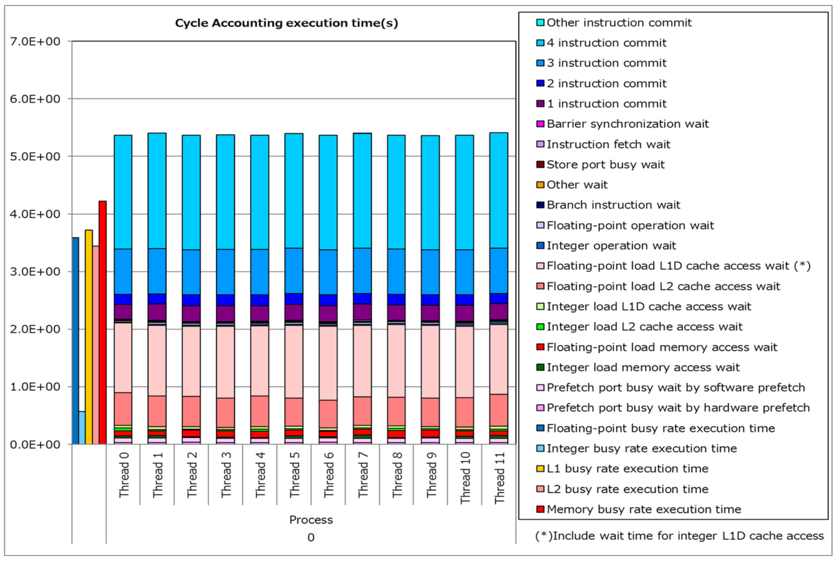


Figure 13.5: Performance of GRAD3X of after tuning

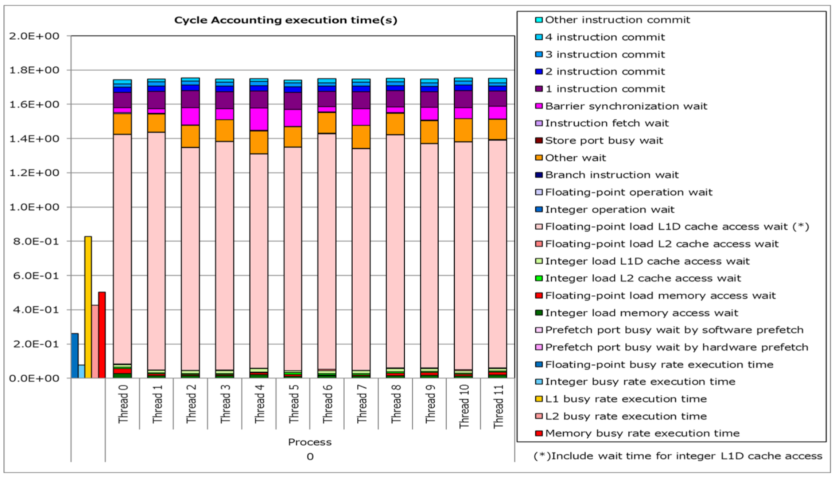


Figure 13.6: Performance of GRAD3X of before tuning(2)

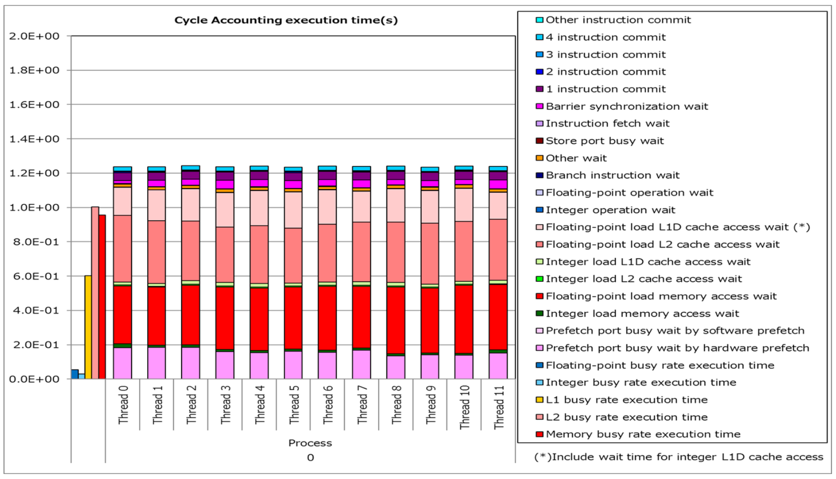


Figure 13.7: Performance of GRAD3X of after tuning(2)

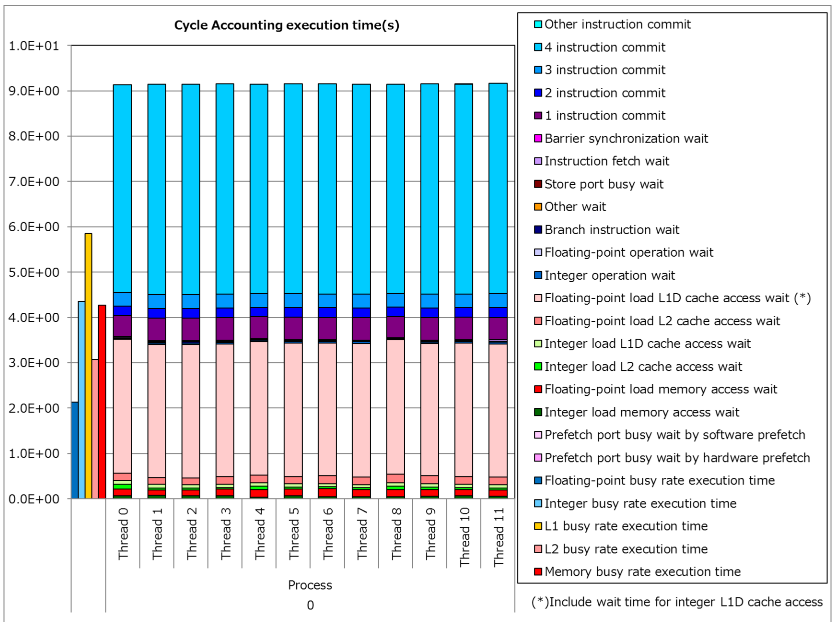


Figure 13.8: Performance of FIELD3X of before tuning

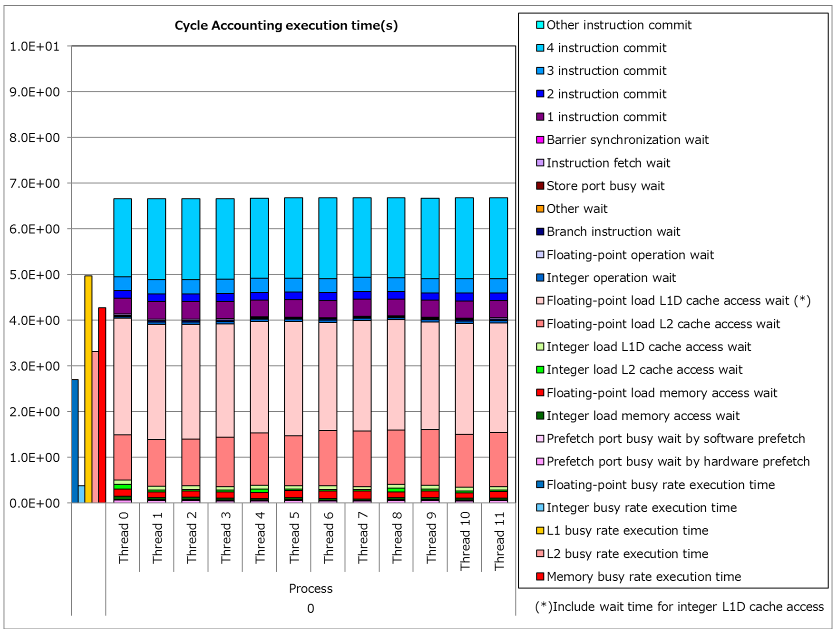


Figure 13.9: Performance of FIELD3X of after tuning

13.8.3 Performance measurement result on Fugaku

The final single performance results of the main kernel are shown in Figure 13.10 and Figure 13.11. The parallel performance is shown in Figure 13.12.

It can be seen that the three kernels, which are the memory bandwidth necks, have almost reached the limit of the effective memory bandwidth, and the highest peak performance ratio is obtained for the one kernel, which is rich in calculation. It can be seen that even better parallel performance is obtained.

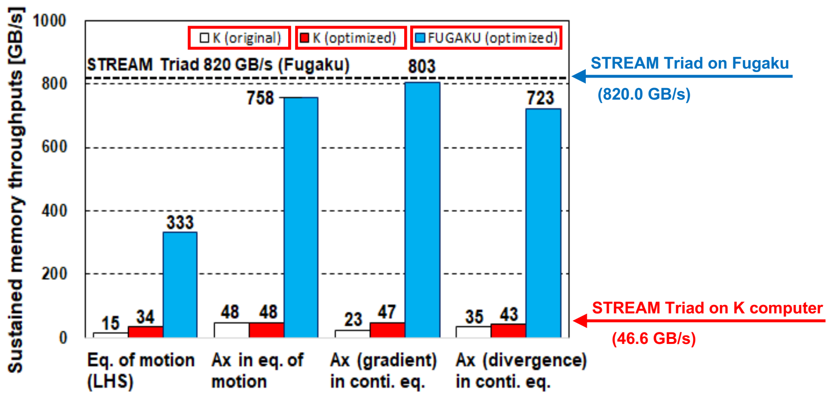


Figure 13.10: Memory bandwidth value (node) of FFB main kernel on Fugaku

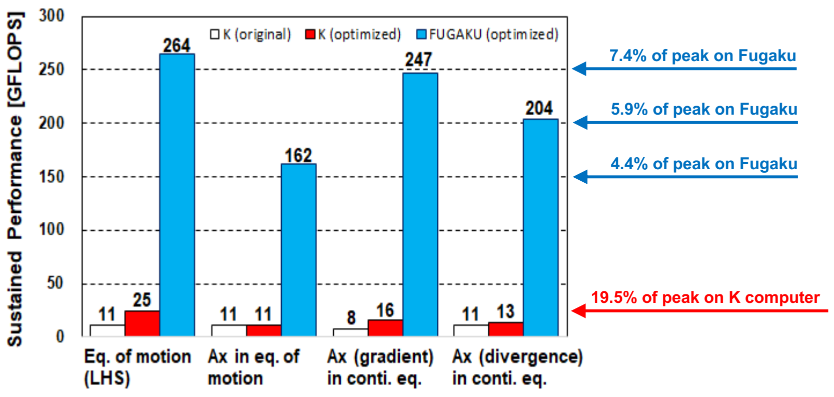


Figure 13.11: Performance (node) of FFB main kernel of Fugaku

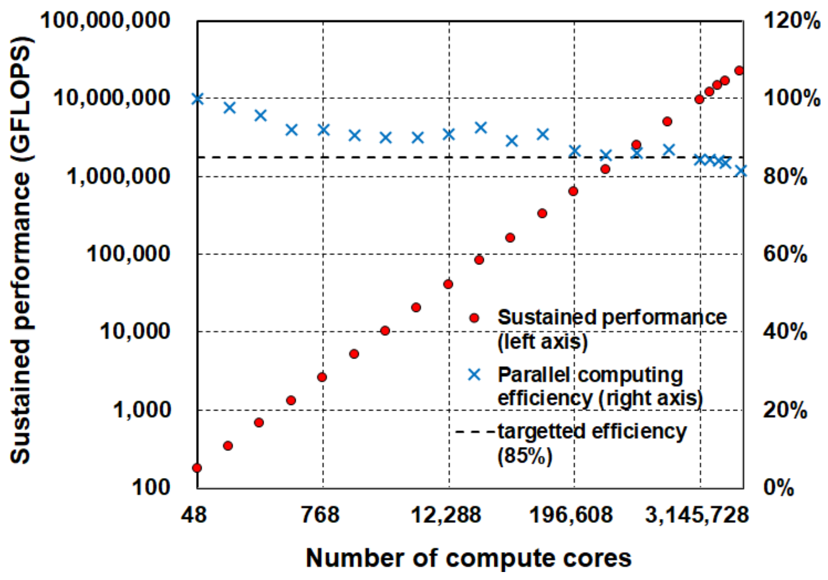
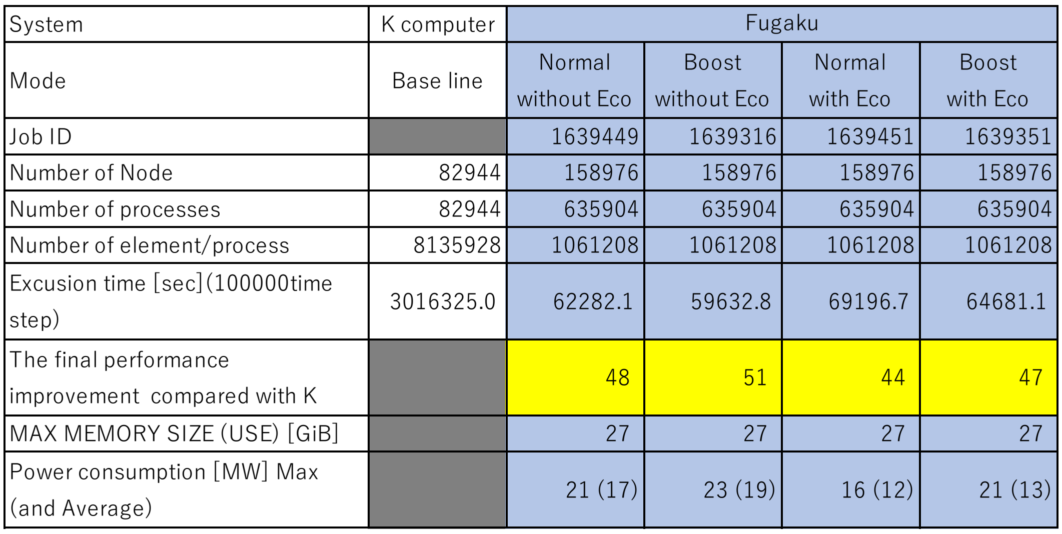


Figure 13.12: Parallel performance of FFB on Fugaku

Figure 13.13 shows the final performance improvement and power consumption compared with K. In the end, 51 times performance improvement is obtained boost and without eco.



System	K computer	Fugaku			
Mode	Base line	Normal without Eco	Boost without Eco	Normal with Eco	Boost with Eco
Job ID		1639449	1639316	1639451	1639351
Number of Node	82944	158976	158976	158976	158976
Number of processes	82944	635904	635904	635904	635904
Number of element/process	8135928	1061208	1061208	1061208	1061208
Excusion time [sec] (100000time step)	3016325.0	62282.1	59632.8	69196.7	64681.1
The final performance improvement compared with K		48	51	44	47
MAX MEMORY SIZE (USE) [GiB]		27	27	27	27
Power consumption [MW] Max (and Average)		21 (17)	23 (19)	16 (12)	21 (13)

Figure 13.13: the final performance improvement compared with K

13.9 Authors

The authors and the contributors of this chapter are listed below. The affiliations in the parentheses are the ones at the time of participation.

13.9.0.0.1 Authors

- Kazuo Minami (RIKEN)
- Kiyoshi Kumahata (RIKEN)

13.9.0.0.2 Co-Authors in alphabetical order

- Chisachi Kato (U. Tokyo)
- Yoshinobu Yamade (Mizuho Research & Technologies)

Chapter 14

Codesign of LQCD

Lattice quantum chromodynamics (LQCD) is an application to solve problems in elementary particle physics. In LQCD, we compute the quantum chromodynamics (QCD) theory of quarks and gluons on the four dimensional regular lattice. Solver of the Dirac equations, so-called “quark solver”, which would be solved by iterative methods consumes CPU time in this application. In quark solver computation on the supercomputer Fugaku, we achieve a factor 38 time speedup from the supercomputer K, with 102 PFLOPS, 10% floating-point operation efficiency against single precision floating-point operation peak.

14.1 Application detail and its codesign

Lattice quantum chromodynamics (LQCD) is an application to solve problems in elementary particle physics. In LQCD, we compute the quantum chromodynamics (QCD) theory of quarks and gluons on the four-dimensional (4D) regular lattice. Solver of the Dirac equations, so-called “quark solver”, which would be solved by iterative methods is the most time consuming part in this application.

The coefficient matrix of quark solver using Wilson type fermions is a large stencil sparse matrix. The 4D space and time is discretized to a 4D square lattice. Quark field of 12 complex numbers is put at the site and gauge field of 9 complex numbers is put at the link on the lattice. Computaional speed is usually limited by main memory bandwidth and network bandwidth, although required memory size is relatively small compared to other applications. Therefore achieving nice strong scaling is very challenging in LQCD.

LQCD contributes to the codesign as an application requiring high memory bandwidth and network bandwidth, the communication mechanism of direct memory access, low cache latency, low network latency, no OS jitter, and enough registers to maximize performance of floating-point arithmetic unit by out-of-order execution.

To get high performance for computing quark solver on the supercomputer Fugaku (hereinafter referred to as Fugaku), we have developed “Lattice quantum chromodynamics simulation library for Fugaku and computers with wide SIMD” (QWS [129]) and compared the performance on the entire system of Fugaku to the the performance on the entire system of the supercomputer K (hereinafter referred to as K) using highly optimized code for K [130]. QWS is an open source software on github. It provides a set of functions related to the Wilson-clover quark solver as a library of LQCD and has been used by actual applications of LQCD [131].

14.2 Optimization for quark solver

We optimize and evaluate the region of "the single precision BiCGStab for a Wilson-clover Dirac matrix with Schwarz Alternating Procedure domain decomposition preconditioning [132] using Jacobi iteration for the local domain matrix inversion (hereinafter referred to as QCDJDD)" in quark solver. The lattice size of the target problem is 192^4. QCDJDD is divided to several computation and communication regions as in Table 14.1. Performance evaluation using "performance and power estimation tool (hereinafter referred to as ppptool)"[1] and using "a prototype of Fugaku (hereinafter referred to as prototype)" differ in how to divide a region for the computation region.

Table 14.1: Regions of LQCD

Computation region for ppptool	
`jinv_ddd_in_s_`	static solver in domain
`ddd_in_s_`	matrix vector multiplication in domain
`ddd_out_pre_s_`	preprocess for interdomain matrix vector multiplication
`ddd_out_pos_s_`	postprocess for interdomain matrix vector multiplication
`other_calc`	other calculation in iteration
Computation region for prototype	
`all_calc`	all calculation
`overlapped`	region overlapped by communication
Communication region	
`comlib_irecv_all_c`	start receiving for neighboring communication
`comlib_isend_all_c`	start sending for neighboring communication
`comlib_recv_wait_all_c`	wait receiving for neighboring communication
`comlib_send_wait_all_c`	wait sending for neighboring communication
`s_drbicgstab_dd_hpc_iter_reduc1_`	Allreduce for one float
`s_drbicgstab_dd_hpc_iter_reduc2_`	Allreduce for two floats
`s_drbicgstab_dd_hpc_iter_reduc3_`	two times of Allreduce for three floats

14.2.1 Avoiding main memory bandwidth bottleneck

It is known that mixed precision iterative solver is efficient for large linear systems [133, 134, 135]. In mixed precision iterative solver, main calculations are done in single precision and double precision solution accuracy is obtained by iterative refinement. In applications that performance is limited by main memory bandwidth bottleneck as LQCD, required main memory bandwidth and/or low level cache capacity can be decreased by half by using mixed precision algorithms. We use a double precision BiCGStab with single precision preconditioning which is QCDJDD. Memory size for QCDJDD in the target problem size 192^4 is about 2TB which is enough smaller than about 5 TB of L2 cache capacity of Fugaku. We perform LQCD on almost full nodes, 147456 nodes (589824 processes), to avoid main memory bandwidth bottleneck in QCDJDD. In codesign, we have removed unnecessary temporal array and zero-fill for memory size reduction and performance improvement.

14.2.2 Utilizing wide SIMD

Main computations in LQCD are complex arithmetic. The original code used for Fugaku codesign was tuned for narrow SIMD width which was used in HPC architectures in early 2000s. On K, the data layout for complex number, (Real-Imag)-(Real-Imag)-..., was used since the SIMD width was 128 bits and operations for complex numbers were supported. In the case of wide SIMD width in recent HPC architectures, a different optimization like for vector computers in

[1]ppptool estimates performance on Fugaku's CPU by using Performance analysis (PA) information obtained on FX100.

1990s is needed. We have considered simple data layout that sequential access continues to use wide SIMD width effectively in LQCD. The SIMD width on Fugaku is 512 bits. It is for 8 double precision floating-point numbers, 16 single precision floating-point numbers, or 32 half precision floating-point numbers. Operations for complex numbers are available on Fugaku. Therefore we have compared two data layouts, an array of complex numbers and a layout that real part and imaginary part is separated. Resulting performance of the real-imaginary separated layout is better than the complex number layout. Data layouts we adopt for fermion field, for example, can be written in C language as follows,

$$\begin{aligned}
\text{K} &: [n_t][n_z][n_y][n_x][3][4][2] \\
\text{Fugaku (double)} &: [n_t][n_z][n_y][n_x/8][3][4][2][8] \\
\text{Fugaku (single)} &: [n_t][n_z][n_y][n_x/16][3][4][2][16] \\
\text{Fugaku (half)} &: [n_t][n_z][n_y][n_x/32][3][4][2][32]
\end{aligned}$$

where n_x, n_y, n_z, n_t are 4D lattice size per process. n_x is divided by SIMD length on Fugaku for consecutive access.

14.2.3 Eliminating thread imbalance by loop splitting and changing the thread parallelization axis

When 192^4 lattices is divided by 589824 MPI processes, lattice size per process would be $32 \times 3 \times 6 \times 4$, $32 \times 6 \times 6 \times 2$ if $n_x = 32$. In the detailed design (1), $32 \times 3 \times 6 \times 4$ was used because neighboring communications can be minimized. But there was a load imbalance between threads in regions `ddd_in_s_` and `ddd_out_pos_s_` when number of threads is 12 and we handle accessing data outside process in a triple loop of $3 \times 6 \times 4$ for YZT directions[2].

To avoid the load imbalance, we considered using $32 \times 6 \times 6 \times 2$ in the detailed design (2). In this local size, the load imbalance can be solved if we change a triple loop for $6 \times 6 \times 2$ to the following.

```
For($n_z, n_t$ loop (6x2=12) OpenMP parallel)
  For($n_x, n_y$ loop )
    X-direction difference computation
    Y-direction difference computation
  EndFor
EndFor
For($n_y, n_t$ loop (6x2=12) OpenMP parallel)
  For($n_x, n_z$ loop)
    Z-direction difference computation
    T-direction difference computation
  EndFor
EndFor
```

Hereinafter this optimization is referred to as thread-balancing. We can expect further optimization by the compiler because loop bodies become simpler than the original code. On the other hand, dividing accumulation of difference computations for each direction into two loops may cause a performance reduction due to cache missing and additional add operations. Therefore we compared the new and original codes and found that the new one showed a better performance on single process. We also found that the elapsed time of the new code on $32 \times 6 \times 6 \times 2$ (see

[2]Loop iteration of the innermost loop for X-direction is one since 32 is divided by SIMD length 16 and 2 of division number for domain decomposition in process.

subsec. 14.2.17) lattice per process is shorter than that on $32 \times 3 \times 6 \times 4$ used in the detailed design (2).

In the detailed design (3), we disused this loop fission tuning and reverted to the simple quadruple loop. Further tuning on `jinv_ddd_in_s_` and `ddd_in_s_` such as inline expansion of functions, optimization arrestation of common subexpression elimination reduced the number of spill/fill operations and showed a better performance on "Fugaku CPU performance simulator (hereinafter referred to as simulator)"[3]. Load balancing had been still applied in `ddd_out_pre_s_` and `ddd_out_pos_s_` due to lack of time for tuning. The performance was estimated on $32 \times 6 \times 6 \times 2$ lattice per process, which was still faster than on $32 \times 3 \times 6 \times 4$ case.

In the detailed design (4), we disused the loop fission in `jinv_ddd_in_s_` and `ddd_in_s_` as well and applied the similar tuning to them. Resulting performance was improved although there was possibility of performance reduction due to load balancing in threads. So we estimated performance with prototype on $32 \times 6 \times 4 \times 3$. We, however, estimated performance with ppptool on $32 \times 6 \times 6 \times 2$ because performance on FX100 decreased significantly due to special optimizations for ARM CPU.

14.2.4 Unequal OpenMP parallelization for reducing L1I missing

Quadruple loop for n_x, n_y, n_z, and n_t is allocated to all 12 threads in original `ddd_out_pre_s_`. The number of the loop iterations is small and the ratio of L1I miss count is high when lattice size per process is small. So we had tried unequal OpenMP parallelization that we allocate four threads for n_x, n_t loops and two threads for n_y, n_z loops, respectively. By this change, L1I missing was reduced and performance increased. In the detailed design (4), `ddd_out_pre_s_` was reverted to the simple quadruple loop and optimized as `ddd_in_s_`.

14.2.5 Manual insertion of software prefetch

We manually insert software prefetch requests to reduce access wait time for floating-point load from L2 cache. We prefetch data which will be used at one iteration ahead in the innermost loop for in `ddd_in_s_`, and so on. For `*in` of `union scs_t` defined as following, we prefetch data every 256 Byte in the case of VLENS = 8. We also prefetch data which will be used in the next loop at the end of loop.

```
typedef union {
  float   c[3][4][2][VLENS];
  float   c_prefetch[24*VLENS];
  rvecs_t cv[3][4][2];
  rvecs_t cs[12][2];
  rvecs_t ccs[24];
} scs_t;

    __builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[0]),0,1);
    __builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[64]) ,0,1);
    __builtin_prefetch(&((( in+i0+_PFI)->c_prefetch)[128]),0,1);
```

The compiler options and run time environments on FX100 are as follows.

Environments	XOS_MMM_L_ARENA_FREE=2
	OMP_NUM_THREADS=12

Options	-Kfast,restp=all,ocl,preex,openmp,noswp,noprefetch,unroll=100

The weak prefetch is the default setting in prefetch oprations with `__builtin_prefetch` function. We changed it to the strong prefetch by modifiyng the assemble code in the makefle as follow

[3]It simulates Fugaku's CPU performance.

```
ddd_in_s_strp.s:ddd_in_s.s
        sed -e "s/\(prefetch\t.*,\)0$$/\120/g" \
            -e "s/\(prefetch\t.*,\)2$$/\122/g" $< > ddd_in_s_strp.s
```

By manual insertion of software prefetch, “L1D miss demand rate” and “Floating-point load L2 cache access wait” decreased and the performance was improved in the all computation regions. Gather prefetch that we tested in the detailed design (3) as a tuning for ARM64 and SVE was ineffective. In the detailed design (4), we applied manual prefetch in the quadruple loops to XYZT direction differential calculations.

14.2.6 OpenMP parallel region expansion

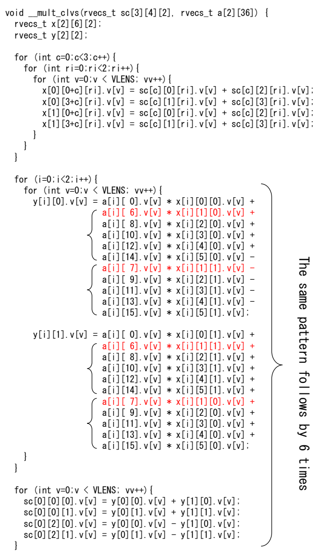
Because making omp parallel region is costly, we put “omp parallel” on higher level caller routines. “omp parallel” in the regions, `ddd_in_s_`, `ddd_out_pre_s_`, `ddd_out_pos_s_`, `other_calc`, and `jinv_ddd_in_s_` were moved to the outside of isap loop of caller routine: `prec_ddd_s_`. This gave that “Instruction fetch wait” by “L1I miss rate” reduction, “Floating-point load L2 cache access wait” by “L1D miss demand rate” reduction, “Barrier synchronization wait”, “Integer load L1D cache access wait” by “Integer instruction” reduction, and “Integer load L2 cache access wait” decreased for each region.

14.2.7 Tuning `__mult_clvs` by reordering arithmetic

A function `__mult_clvs` is used in `ddd_in_s_`, `jinv_ddd_in_s_`, and `ddd_out_s_` for clover term multiplication. We found by using simulator that the original code of `__mult_clvs` was inefficient due to load and store operations because of register spill/fill. Fig. 14.1 is the outline of the original code with which the computational latency can not be hidden because the red lined arithmetic can not be performed before completing the previous arithmetic. A way to hide the latency is to do independent arithmetic operations sequentially. On Fugaku, L1D Byte/FLOP ratio in single precision is 2 and there are two floating-point operation pipelines. If the arithmetic intensity is one FMA per one load (4 bytes per 2 operations) and there are $9+9=18$ arithmetic streams to hide 9 cycles of arithmetic latency for 2 arithmetic pipelines, one can obtain the peak performance by round robin manner. This region almost satisfies the above conditions with 32 registers, even if there are load operations for reuse of input date. Specifically, there are 16 arithmetic streams, with an arithmetic intensity of 4 loads and 4 FMAs. The order of the operations and the code after the aligning the streams are shown in Fig. 14.2. We prevented undesired compiler optimization, which is common subexpression elimination for long-distance reuse, by splitting blocks with if statements. We also specified the compiler flags that suppress instruction scheduling (`-Knosch_post_ra -Knosch_pre_ra -Knoeval`). We confirmed that these optimizations reduced the number of spills from the original 512 to 14.

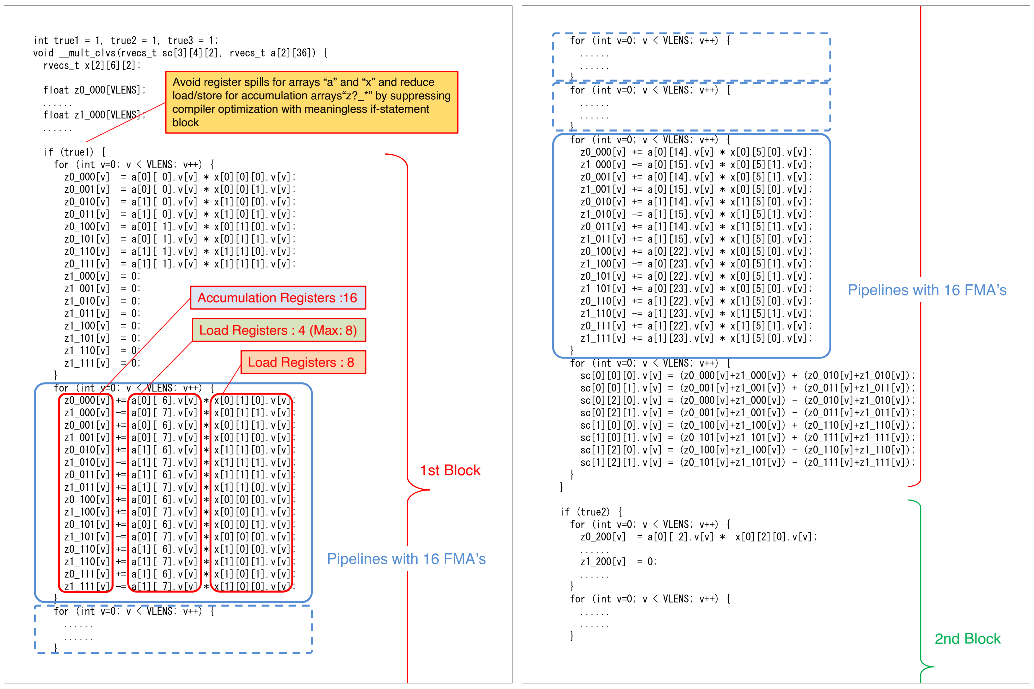
14.2.8 Optimization with ARM C Language Extensions (ACLE)

In the detailed design (3), the process of extracting the components that appear in the stencil difference calculation in the X-direction, the SIMD vectorization direction, from the two vector components into a single SIMD vector was implemented using the ACLE function for SVE. As a result, the extraction of array elements for the difference calculation in the X-direction was completed on the SIMD registers, the number of load/store instructions for temporary arrays was reduced, and the execution efficiency greatly increased. Fig. 14.3 shows the conceptual diagram of this optimization.



```
void __mult_clvs(rvecs_t sc[3][4][2], rvecs_t a[2][36]) {
  rvecs_t x[2][6][2];
  rvecs_t y[2][2];

  for (int c=0;c<3;c++){
    for (int ri=0;ri<2;ri++){
      for (int v=0;v < VLENS; vv++){
        x[0][0+c][ri].v[v] = sc[c][0][ri].v[v] + sc[c][2][ri].v[v];
        x[0][3+c][ri].v[v] = sc[c][1][ri].v[v] + sc[c][3][ri].v[v];
        x[1][0+c][ri].v[v] = sc[c][0][ri].v[v] + sc[c][2][ri].v[v];
        x[1][3+c][ri].v[v] = sc[c][1][ri].v[v] + sc[c][3][ri].v[v];
      }
    }
  }

  for (i=0;i<2;i++){
    for (int v=0;v < VLENS; vv++){
      y[i][0].v[v] = a[i][ 0].v[v] * x[i][0][0].v[v] +
                     a[i][ 6].v[v] * x[i][1][0].v[v] +
                     a[i][ 8].v[v] * x[i][2][0].v[v] +
                     a[i][10].v[v] * x[i][3][0].v[v] +
                     a[i][12].v[v] * x[i][4][0].v[v] +
                     a[i][14].v[v] * x[i][5][0].v[v] -
                     a[i][ 7].v[v] * x[i][1][1].v[v] -
                     a[i][ 9].v[v] * x[i][2][1].v[v] -
                     a[i][11].v[v] * x[i][3][1].v[v] -
                     a[i][13].v[v] * x[i][4][1].v[v] -
                     a[i][15].v[v] * x[i][5][1].v[v];

      y[i][1].v[v] = a[i][ 0].v[v] * x[i][0][1].v[v] +
                     a[i][ 6].v[v] * x[i][1][1].v[v] +
                     a[i][ 8].v[v] * x[i][2][1].v[v] +
                     a[i][10].v[v] * x[i][3][1].v[v] +
                     a[i][12].v[v] * x[i][4][1].v[v] +
                     a[i][14].v[v] * x[i][5][1].v[v] +
                     a[i][ 7].v[v] * x[i][1][0].v[v] +
                     a[i][ 9].v[v] * x[i][2][0].v[v] +
                     a[i][11].v[v] * x[i][3][0].v[v] +
                     a[i][13].v[v] * x[i][4][0].v[v] +
                     a[i][15].v[v] * x[i][5][0].v[v];
    }
  }

  for (int v=0;v < VLENS; vv++){
    sc[0][0][0].v[v] = y[0][0].v[v] + y[1][0].v[v];
    sc[0][0][1].v[v] = y[0][1].v[v] + y[1][1].v[v];
    sc[0][2][0].v[v] = y[0][0].v[v] - y[1][0].v[v];
    sc[0][2][1].v[v] = y[0][1].v[v] - y[1][1].v[v];
  }
```

Figure 14.1: `__mult_clvs` before optimizations.



```
int true1 = 1, true2 = 1, true3 = 1;
void __mult_clvs(rvecs_t sc[3][4][2], rvecs_t a[2][36]) {
  rvecs_t x[2][6][2];

  float z0_000[VLENS];
  ......
  float z1_000[VLENS];
  ......

  if (true1) {
    for (int v=0; v < VLENS; v++) {
      z0_000[v]  = a[0][ 0].v[v] * x[0][0][0].v[v];
      z0_001[v]  = a[0][ 0].v[v] * x[0][0][1].v[v];
      z0_010[v]  = a[1][ 0].v[v] * x[1][0][0].v[v];
      z0_011[v]  = a[1][ 0].v[v] * x[1][0][1].v[v];
      z0_100[v]  = a[0][ 1].v[v] * x[0][1][0].v[v];
      z0_101[v]  = a[0][ 1].v[v] * x[0][1][1].v[v];
      z0_110[v]  = a[1][ 1].v[v] * x[1][1][0].v[v];
      z0_111[v]  = a[1][ 1].v[v] * x[1][1][1].v[v];
      z1_000[v]  = 0;
      z1_001[v]  = 0;
      z1_010[v]  = 0;
      z1_011[v]  = 0;
      z1_100[v]  = 0;
      z1_101[v]  = 0;
      z1_110[v]  = 0;
      z1_111[v]  = 0;
    }
    for (int v=0; v < VLENS; v++) {
      z0_000[v] += a[0][ 6].v[v] * x[0][1][0].v[v];
      z1_000[v] -= a[0][ 7].v[v] * x[0][1][1].v[v];
      z0_001[v] += a[0][ 6].v[v] * x[0][1][1].v[v];
      z1_001[v] += a[0][ 7].v[v] * x[0][1][0].v[v];
      z0_010[v] += a[1][ 6].v[v] * x[1][1][0].v[v];
      z1_010[v] -= a[1][ 7].v[v] * x[1][1][1].v[v];
      z0_011[v] += a[1][ 6].v[v] * x[1][1][1].v[v];
      z1_011[v] += a[1][ 7].v[v] * x[1][1][0].v[v];
      z0_100[v] += a[0][ 6].v[v] * x[0][0][0].v[v];
      z1_100[v] += a[0][ 7].v[v] * x[0][0][1].v[v];
      z0_101[v] += a[0][ 6].v[v] * x[0][0][1].v[v];
      z1_101[v] -= a[0][ 7].v[v] * x[0][0][0].v[v];
      z0_110[v] += a[1][ 6].v[v] * x[1][0][0].v[v];
      z1_110[v] += a[1][ 7].v[v] * x[1][0][1].v[v];
      z0_111[v] += a[1][ 6].v[v] * x[1][0][1].v[v];
      z1_111[v] -= a[1][ 7].v[v] * x[1][0][0].v[v];
    }
    for (int v=0; v < VLENS; v++) {
      ......
      ......
    }
```

```
    for (int v=0; v < VLENS; v++) {
      ......
      ......
    }
    for (int v=0; v < VLENS; v++) {
      ......
      ......
    }
    for (int v=0; v < VLENS; v++) {
      z0_000[v] += a[0][14].v[v] * x[0][5][0].v[v];
      z1_000[v] -= a[0][15].v[v] * x[0][5][1].v[v];
      z0_001[v] += a[0][14].v[v] * x[0][5][1].v[v];
      z1_001[v] += a[0][15].v[v] * x[0][5][0].v[v];
      z0_010[v] += a[1][14].v[v] * x[1][5][0].v[v];
      z1_010[v] -= a[1][15].v[v] * x[1][5][1].v[v];
      z0_011[v] += a[1][14].v[v] * x[1][5][1].v[v];
      z1_011[v] += a[1][15].v[v] * x[1][5][0].v[v];
      z0_100[v] += a[0][22].v[v] * x[0][5][0].v[v];
      z1_100[v] -= a[0][23].v[v] * x[0][5][1].v[v];
      z0_101[v] += a[0][22].v[v] * x[0][5][1].v[v];
      z1_101[v] += a[0][23].v[v] * x[0][5][0].v[v];
      z0_110[v] += a[1][22].v[v] * x[1][5][0].v[v];
      z1_110[v] -= a[1][23].v[v] * x[1][5][1].v[v];
      z0_111[v] += a[1][22].v[v] * x[1][5][1].v[v];
      z1_111[v] += a[1][23].v[v] * x[1][5][0].v[v];
    }
    for (int v=0; v < VLENS; v++) {
      sc[0][0][0].v[v] = (z0_000[v]+z1_000[v]) + (z0_010[v]+z1_010[v]);
      sc[0][0][1].v[v] = (z0_001[v]+z1_001[v]) + (z0_011[v]+z1_011[v]);
      sc[0][2][0].v[v] = (z0_000[v]+z1_000[v]) - (z0_010[v]+z1_010[v]);
      sc[0][2][1].v[v] = (z0_001[v]+z1_001[v]) - (z0_011[v]+z1_011[v]);
      sc[1][0][0].v[v] = (z0_100[v]+z1_100[v]) + (z0_110[v]+z1_110[v]);
      sc[1][0][1].v[v] = (z0_101[v]+z1_101[v]) + (z0_111[v]+z1_111[v]);
      sc[1][2][0].v[v] = (z0_100[v]+z1_100[v]) - (z0_110[v]+z1_110[v]);
      sc[1][2][1].v[v] = (z0_101[v]+z1_101[v]) - (z0_111[v]+z1_111[v]);
    }
  }

  if (true2) {
    for (int v=0; v < VLENS; v++) {
      z0_200[v]  = a[0][ 2].v[v] *  x[0][2][0].v[v];
      ......
      z1_200[v]  = 0;
      ......
    }
    for (int v=0; v < VLENS; v++) {
      ......
      ......
    }
```

Avoid register spills for arrays “a” and “x” and reduce load/store for accumulation arrays“z?_*” by suppressing compiler optimization with meaningless if-statement block
Accumulation Registers :16
Load Registers : 4 (Max: 8)
Load Registers : 8
1st Block
Pipelines with 16 FMA’s
Pipelines with 16 FMA’s
2nd Block

Figure 14.2: `__mult_clvs` after optimizations.

14.2.9 Integer register spill/fill suppression

In the detailed design (3), the loop was simplified by eliminating thread-balancing in subsec. 14.2.3. Furthermore, inline expansion of `__mult_clvs` was performed to eliminate the overhead of function calls. The compiler generates register spills/fills for other variables in order to maintain registers that hold the results of common expressions. In order to avoid this behav-

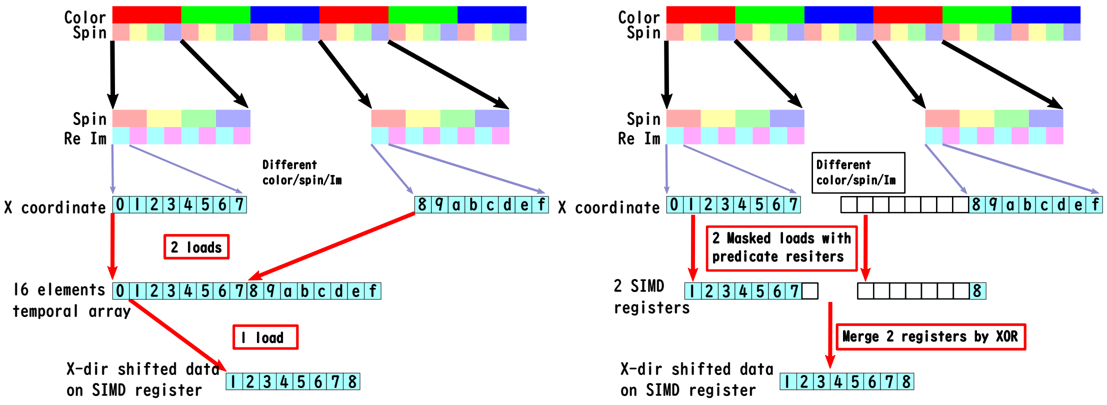
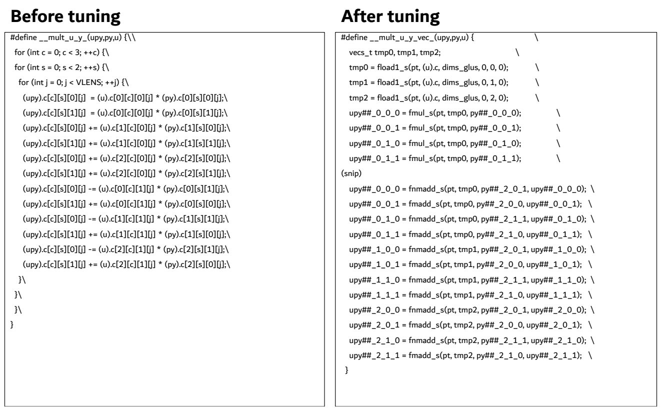


Figure 14.3: Conceptual diagrams of the X-direction data load for difference calculation before (left) and after (right) optimization.

ior, we disabled the common expression by rewriting the base variable as `(asm("":"+r"(x)))`. In addition, we explicitly separated the base and immediate values in the source code to use addressing with common base registers and immediate values as much as possible. As a result, spill/fill of integer registers was suppressed by the compiler, and execution efficiency was improved. Since this optimization was better than thread-balancing, the latest version of the code does not adopt thread-balancing.

14.2.10 Scheduling operations on the source code to increase instruction-level parallelism

A suitable source-code-level scheduling of operations can increase instruction-level parallelism. Fig. 14.4 shows the source code before and after this tuning. Hereinafter this optimization is referred to as hand-scheduling.


Before tuning

```
#define __mult_u_y_(upy,py,u) {\\
 for (int c = 0; c < 3; ++c) {\
 for (int s = 0; s < 2; ++s) {\
  for (int j = 0; j < VLENS; ++j) {\
   (upy).c[c][s][0][j]  = (u).c[0][c][0][j] * (py).c[0][s][0][j];\
   (upy).c[c][s][1][j]  = (u).c[0][c][0][j] * (py).c[0][s][1][j];\
   (upy).c[c][s][0][j] += (u).c[1][c][0][j] * (py).c[1][s][0][j];\
   (upy).c[c][s][1][j] += (u).c[1][c][0][j] * (py).c[1][s][1][j];\
   (upy).c[c][s][0][j] += (u).c[2][c][0][j] * (py).c[2][s][0][j];\
   (upy).c[c][s][1][j] += (u).c[2][c][0][j] * (py).c[2][s][1][j];\
   (upy).c[c][s][0][j] -= (u).c[0][c][1][j] * (py).c[0][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[0][c][1][j] * (py).c[0][s][0][j];\
   (upy).c[c][s][0][j] -= (u).c[1][c][1][j] * (py).c[1][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[1][c][1][j] * (py).c[1][s][0][j];\
   (upy).c[c][s][0][j] -= (u).c[2][c][1][j] * (py).c[2][s][1][j];\
   (upy).c[c][s][1][j] += (u).c[2][c][1][j] * (py).c[2][s][0][j];\
  }\
 }\
 }\
}
```

After tuning

```
#define __mult_u_y_vec_(upy,py,u) {                        \
  vecs_t tmp0, tmp1, tmp2;                                 \
  tmp0 = fload1_s(pt, (u).c, dims_glus, 0, 0, 0);          \
  tmp1 = fload1_s(pt, (u).c, dims_glus, 0, 1, 0);          \
  tmp2 = fload1_s(pt, (u).c, dims_glus, 0, 2, 0);          \
  upy##_0_0_0 = fmul_s(pt, tmp0, py##_0_0_0);              \
  upy##_0_0_1 = fmul_s(pt, tmp0, py##_0_0_1);              \
  upy##_0_1_0 = fmul_s(pt, tmp0, py##_0_1_0);              \
  upy##_0_1_1 = fmul_s(pt, tmp0, py##_0_1_1);              \
(snip)
  upy##_0_0_0 = fnmadd_s(pt, tmp0, py##_2_0_1, upy##_0_0_0); \
  upy##_0_0_1 = fmadd_s(pt, tmp0, py##_2_0_0, upy##_0_0_1);  \
  upy##_0_1_0 = fnmadd_s(pt, tmp0, py##_2_1_1, upy##_0_1_0); \
  upy##_0_1_1 = fmadd_s(pt, tmp0, py##_2_1_0, upy##_0_1_1);  \
  upy##_1_0_0 = fnmadd_s(pt, tmp1, py##_2_0_1, upy##_1_0_0); \
  upy##_1_0_1 = fmadd_s(pt, tmp1, py##_2_0_0, upy##_1_0_1);  \
  upy##_1_1_0 = fnmadd_s(pt, tmp1, py##_2_1_1, upy##_1_1_0); \
  upy##_1_1_1 = fmadd_s(pt, tmp1, py##_2_1_0, upy##_1_1_1);  \
  upy##_2_0_0 = fnmadd_s(pt, tmp2, py##_2_0_1, upy##_2_0_0); \
  upy##_2_0_1 = fmadd_s(pt, tmp2, py##_2_0_0, upy##_2_0_1);  \
  upy##_2_1_0 = fnmadd_s(pt, tmp2, py##_2_1_1, upy##_2_1_0); \
  upy##_2_1_1 = fmadd_s(pt, tmp2, py##_2_1_0, upy##_2_1_1);  \
 }
```

Figure 14.4: Code before and after tuning to increase instruction-level parallelism.

14.2.11 Statical set of the problem size at compile time

By giving the problem size as a constant at compile time, the efficiency was slightly improved.

14.2.12 Improving efficiency of reduction computation in multiple loops

The following multiple loops with reduction computation appears in the iterative method.

Multiple loops with reduction

```
for(i=0; i<n; i++)
 for(j=0; j<m; j++)
  s += a[i][j];
```

This code was translated to use vector reductions in the outer loop as "Before optimization". We instead directly wrote a loop using the vector built-in function with the following loop structure as "After optimization". Hereinafter this optimization is referred to as effective-reduction.

Before optimization

```
for(i=0; i<n; i++) {
 vector = {0,0, ...};
 for(j=0; j<m/VLEN; j+=VLEN) {
  vector += {a[i][j], a[i][j+1], ...};
 }s += sum(vector);
}
```

After optimization

```
vector = {0,0, ...};
for(i=0; i<n; i++) {
 for(j=0; j<m/VLEN; j+=VLEN) {
  vector += {a[i][j], a[i][j+1], ...};
 }
}
s = sum(vector);
```

14.2.13 Summary of tuning for each computation region

The following is a summary of the tuning of each computation region.

- `ddd_in_s_`
  - Optimization with ACLE
  - Integer register spill/fill suppression
  - Hand-scheduling
  - Manual prefetch insertion
  - Deleting work array *tmp*
- `jinv_ddd_in_s_`
  - Same tuning as `ddd_in_s_`
  - Manual prefetch insertion into loops other than `ddd_in_s_`
- `ddd_out_pre_s_`
  - Similar tuning as `ddd_in_s_`
- `ddd_out_pos_s_`
  - Similar tuning as `ddd_in_s_`
- `other_calc`
  - Manual prefetch insertion
  - Effective-reduction
- `__mult_clvs`
  - Hiding operation latency by rearranging the order of mutually independent operations

  - Register spill/fill suppression by rearranging the order of operations
- Common to all computation regions
  - Aligning updated array
  - OpenMP parallel region expansion
  - Statical set of the problem size at compile time

14.2.14 Utilizing FP16

The QCD quark solver uses an iterative algorithm to solve the large scale linear equation obtained from the equation of motion of quarks on lattice. To obtain double precision solution for the equation efficiently, we employ a mixed precision scheme, in which single precision (FP32) arithmetic is primary used, in the solver algorithm. Using the following FP32 arithmetic properties, high efficiency on the cash and memory bandwidth utilization per FLOP and doubled peak FLOP performance than that of double precision (FP64), we can improve the performance of the solver. Fugaku supports half precision (FP16) arithmetic and its peak FLOP performance is doubled than that of FP32. We expect a more performance gain by using FP16 arithmetic in the mixed precision solver algorithm.

In the detailed design (4), we discussed a mixed precision algorithm incorporating FP16, FP32, and FP64. In general, the total iteration counts tends to be large to obtain a FP64 solution with the mixed precision scheme. It could spoil the gain from the high peak performance of lower precision arithmetic. Therefore, we investigate the convergence behavior of the mixed precision algorithm with FP16. Because it is difficult to maintain the numerical precision of vector norm and inner product computation with FP16 arithmetic, we employ FP16 in a fixed iterative solver involved in the quark solver.

In the FP64-FP32 mixed precision algorithm used in the quark solver, we utilize FP32 arithmetic in the multiplication of a preconditioning matrix on a vector. The matrix-vector multiplication of the preconditioning matrix is computed by solving roughly the corresponding linear equation with an iterative algorithm entirely in FP32. In the FP32 solver algorithm, one more fixed iterative solver is involved as the preconditioner for the FP32 linear equation. We change the precision of the fixed iterative preconditioning to FP16 and investigate the whole convergence behavior to obtain the FP64 solution. In this study, as the compiler set and the CPU of Fugaku were in developing and not available, we employ a C++ class library `half-1.12.0` (`http://half.sourceforge.net/`), which is compliant to IEEE-754, in implementing the FP16 iterative solver.

We investigate the convergence behavior of the quark solver at five quark masses corresponding $\kappa = 0.1324, 0.1333, 0.1342, 0.1348, 0.1350$ on a configuration generated at $\beta = 6.00$ in pure Wilson gauge theory, where the corresponding pseudo-scalar meson masses aM_{PS} are $aM_{\mathrm{PS}} \simeq 0.100, 0.090, 0.065, 0.035, 0.010$, respectively. Figure 14.5 plots the residual norm of the linear equation against each the FP32 solver iteration count. The left panels are the results with the FP64-FP32-PF16 mixed precision solver, and the right ones are with the FP64-FP32 mixed precision one. The panels are vertically aligned according to the quark mass from the top panels (heaviest quark mass) to the bottom ones (lightest). The total iteration counts for the convergence becomes larger as we decrease the quark mass, and the count of the FP64-FP32-PF16 mixed precision is $\sim 17\%$ ($\sim 26\%$) longer than that of the FP64-FP32 mixed precision for $\kappa = 0.1348$ ($\kappa = 0.1350$). We could expect a performance elapse time gain with FP16 when the computational performance gain with FP16 overcomes the increase of the computational amount for the convergence with FP16.

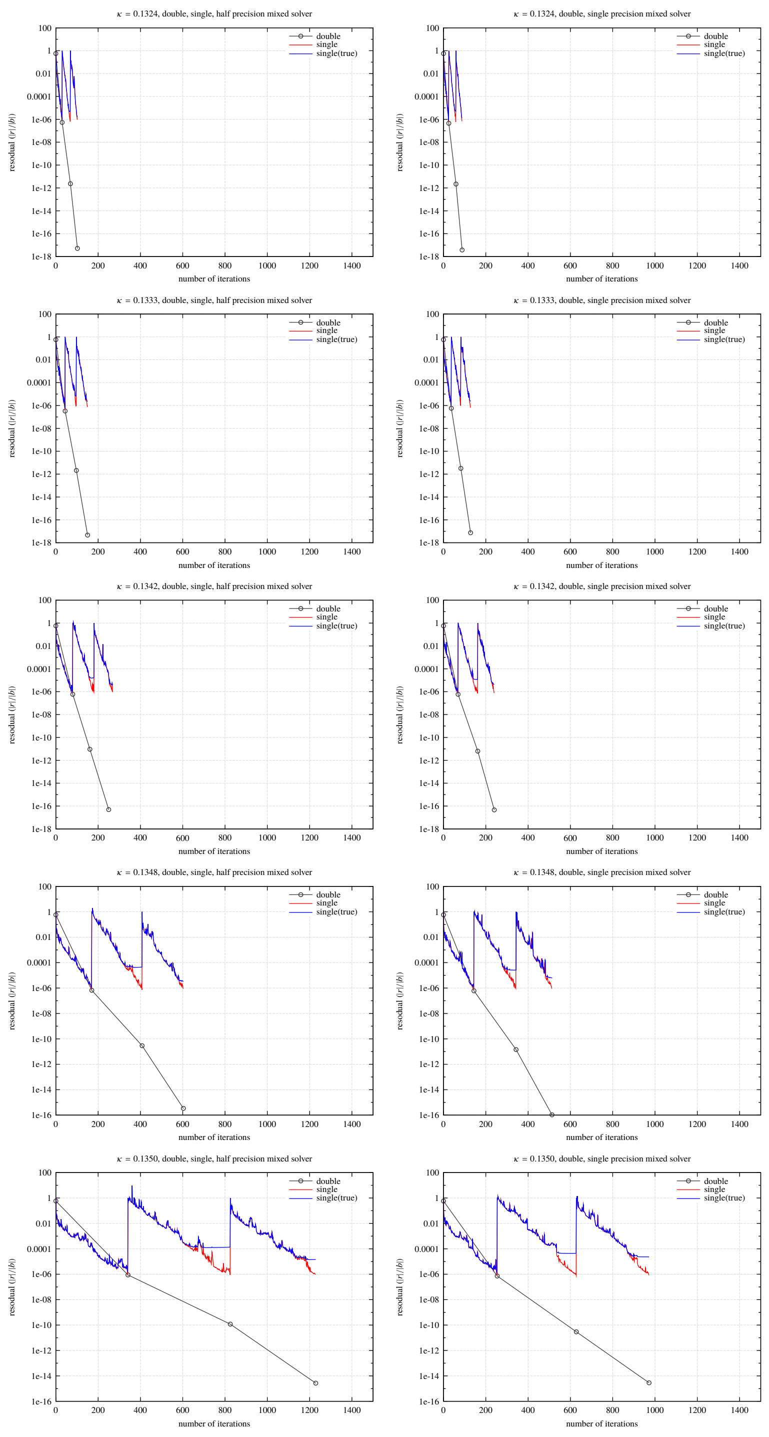


Figure 14.5: Residual norm v.s. the FP32 solver iteration count.

In tuning period (1) of the project, we tested the program of the FP16-FP32-F64 mixed precision algorithm on prototype. As the C/C++ compiler in the trad mode does not support the native FP16 arithmetic, we use the compiler in the clang mode which supports the native FP16. We confirmed that the native FP16 mnemonics were cast by the compiler. However, they were not vector mnemonics but scalar ones. Since the compiler in the clang mode is currently developing and the optimization ability is not complete yet, we cannot conclude the LQCD code performance with FP16 at this stage and it remains for future studies.

We note that the longer SIMD vector length of FP16 cannot fit the vector length of the LQCD solver code. In our target problem, the lattice size is 192^4 and it is divided to small local-lattices for MPI parallelization. The local-lattice extent in the X-direction is 16 and this length is identical to the FP32 SIMD vector length. The longer vector length 32 of FP16 exceeds the local-lattice extent 16 to be vectorized so that half of the FP16 SIMD vector functionality does not work. In order to fully utilize FP16 vector we need a reconstruction of preconditioning method or parallelization scheme of the quark solver algorithm. For the small sized problem with a small number of MPI processes, on the other hand, we can increase the local-lattice size so that the FP16 SIMD vector can fully function. We will prepare the fixed iteration solver with FP16 functional for these cases.

14.2.15 Double-buffering

Our target parallelism is fine grained, the lattice size per process is rather small so that the performance of the halo exchange communication, which is nearest neighbor communication in the 4D lattice, could affect the total performance of the quark solver. In order to reduce the impact of the nearest neighbor communication to the total performance, we overlap the halo exchange communication and the matrix stencil application on the interior sites in the process. We further divide the lattice volume of $n_x \times n_y \times n_z \times n_t$ into two subvolumes with each $(n_x/2) \times n_y \times n_z \times n_t$. Accordingly the stencil matrix-vector multiplication is divided into the following three operations; (STEP A) stencil matrix-vector multiplication within a subvolume, (STEP B) stencil matrix-vector multiplication between two subvolumes, and (C) halo exchange communication. The operation (STEP B) is further divided into two parts; (STEP B-1) packing data to send for halo exchange, (STEP B-2) unpacking data received and computing remaining stencil on the surface sites. The operation (STEP C) performs the MPI communication and can be overlapped to the operation (STEP A) by using non-blocking MPI send/receive procedures. The full stencil multiplication goes as (STEP B-1) $\to$ (STEP A)+(STEP C) $\to$ (STEP B-2).

We have implemented the communication-computation overlap scheme in the quark solver as described above. The communication performance estimated for Fugaku, however, shows that testing the completion of data sending and receiving can be a large performance bottleneck for the total performance. As these tests are called in (STEP B-1) and (STEP B-2), we cannot overlap them to (STEP A) or (STEP C). In order to reduce the overhead by the tests, as shown in Fig. 14.6, we evaluated the performance of a double-buffering (DB) method by which these tests can be delayed to overlap.

The boundary condition of the equation of motion of quarks is periodic in the 4D space-time, the nearest neighbor halo exchange in each direction is required. Usually a pair of buffers for data-send and data-receive is assigned in each direction and `MPI_isend/irecv` uses these buffers for halo exchange. To reduce the latency furthermore, we planned to combine a RDMA (one-sided) communication instead of the point-to-point communication and DB.

The DB scheme is implemented as follows. Each MPI process has one send-buffer and two receive-buffers in each communication link. There are eight communication links in a process, which correspond to eight directions in the 4D periodic boundary condition. The receive-buffer has its own parity attribute, 0 and 1, respectively. Each MPI process has state P which is a set of parity states for eight links. As the halo exchange communication is accomplished by all the MPI processes simultaneously for each matrix-vector multiplication, the state P is identical among all the MPI processes. With these setups the DB scheme in a link proceeds as

1. Copy the data to send into the send-buffer. An additional element, the receive-buffer-occupy state flag, is attached as the last element of the send-buffer. The receive-buffer-occupy state flag is set to “occupied”. For the **uTofu** implementation discussed later, the

receive-buffer-occupy state flag is changed to another value different from each communication instead of “occupied”.

2. Transfer the data in the send-buffer using the RDMA put function. The RDMA put function is asynchronous and non-blocking. The data will be transferred into one of the receive-buffers with corresponding parity P. In order to test the completion of data transfer on the receiver side quickly, we employ a mode that the completion is guaranteed when the last element arrives in the receive-buffer.

3. Do stencil computation on the interior sites.

4. Wait for the data arrival by inspecting the receive-buffer-occupy state flag in the last element of the receive-buffer with parity P. When the occupy state flag is changed to “occupied” state, it is guaranteed that the new data are ready in the receive-buffer. For the **uTofu** implementation, wait for the occupy state flag to change to a state different from the previous communication.

5. Unpack the data from the receive-buffer, and do remaining stencil computation using the data.

6. Set the occupy state flag to “empty” in the receive-buffer. For the **uTofu** implementation, this step has to be omitted.

7. Wait for the completion of the RDMA put function. After the completion, flip the parity P state of the process.

In our previous version of the DB algorithm, we had a step asking the destination MPI rank for the readiness of the receive-buffer by using RDMA Get communication before the first step. However, this step is not required in the DB algorithm and omitted here. In order to verify our DB algorithm, we implemented the algorithm in the Jacobi iterative solver for the two-dimensional (2D) Laplace equation, which is also a stencil type application similar to the LQCD quark solver. We have verified:

- The inquiry step for the receive-buffer can be omitted,
- The parity state is appropriately flipped to work the algorithm.

We implemented several versions for the verification program and executed them on FX100. We search for the most efficient version by comparing the computation time and communication bandwidth among them. The details of the versions, in which we also show the **uTofu** version for later convenience, are as follows:

1. Two versions, single-buffering (SB) and DB versions using MPI-1 persistent point-to-point communication functions. When Fujitsu MPI extension `FJMPI` are available for the persistent communication, we employ the corresponding `FJMPI` functions. The arrival signal polling for the receive-buffer, we employ `MPI_Waitall` function so that the receive-buffer-occupy state flag is not attached.

   In these versions, the following functions are used: `FJMPI_Prequest_recv_init()`, `FJMPI_Prequest_send_init()`, `FJMPI_Prequest_startall()`, and `MPI_Waitall()`.

2. SB and DB versions using MPI-2 one-sided communication functions. Some MPI-3 functions are used for synchronization of buffers.

Table 14.2: Communication functions used in DB

Implementation	STEP A	STEP B	STEP C
1	`FJMPI_Prequest_startall`	`MPI_Waitall`	`MPI_Waitall`
2	`MPI_Put`	Check `MPI_Win_sync` and recieve buffer	`MPI_Win_flush_local`
3	`FJMPI_Rdma_put`	Check recieve buffer	`FJMPI_Rdma_poll_cq`
4	`utofu_put`	Check recieve buffer	`utofu_poll_tcq`

The receive-buffer is allocated with `MPI_Win_allocate` and exposed to the sender rank with `MPI_Win_lock`. For the synchronization between the buffer and memory, `MPI_Win_sync` is used. The completion of the `MPI_Put` is checked with `MPI_Win_flush_local`. Because we have to call `MPI_Win_sync` later than the call to `MPI_Win_flush_local`, we have to put "Put completed" before "Check local receive buffer" in Fig. 14.6. The timing of parity flip is not changed.

Functions used are : `MPI_Win_allocate()`, `MPI_Win_lock()` (used with `MPI_LOCK_EXCLUSIVE` mode),
`MPI_Put()`, `MPI_Win_flush_local()`, `MPI_Win_sync()`

3. DB version with Fujitsu's RDMA functions. A wrapper library `rdma_comlib`, which utilizes Fujitsu's RDMA for the quark solver in K-computer, is employed by modifying it capable of the DB algorithm.

   Functions used are : `FJMPI_Rdma_reg_mem()`, `FJMPI_Rdma_put()`, `FJMPI_Rdma_poll_cq()`

4. DB version with **uTofu**. This version is based on the above Fujitsu's RDMA version. (The polling method for the data arrival completion is changed as described later.)

   Functions used are : `utofu_create_vcq()`, `utofu_query_vcq_id()`, `utofu_reg_mem()`, `utofu_dereg_mem()`, `utofu_poll_tcq()`, `utofu_poll_mrq()`, `utofu_free_vcq()`, `utofu_put()`

Table 14.2 summarizes the communication functions used in the SB and DB algorithms and the location of these functions in the algorithms (see also Fig. 14.6 for the location and functions used).

According to the above implementations and tests on FX100, we have verified the algorithm for all versions. In the test we observed that the version 2 (MPI-2 put version) could not fully utilize the network bandwidth. On the other hand, other versions could utilize the network bandwidth by which the total performance is limited. Comparing the SB and DB versions, no significant performance differences are observed. The best performance was achieved by Fujitsu's RDMA version (version 3) on FX100. Because the full functionality of **uTofu** was not available and emulated on FX100 at that stage, we postponed the verification with **uTofu**.

After verifying the DB algorithm, we incorporate the version 1 (MPI-1 persistent) and version 3 (Fujitsu's RDMA) into qws-0.1.8 (Version 0.1.8 of QWS). We further replace the version 3 (Fujitsu's RDMA) to the version 4 (**uTofu**) for Fugaku after qws-0.2.2 version.

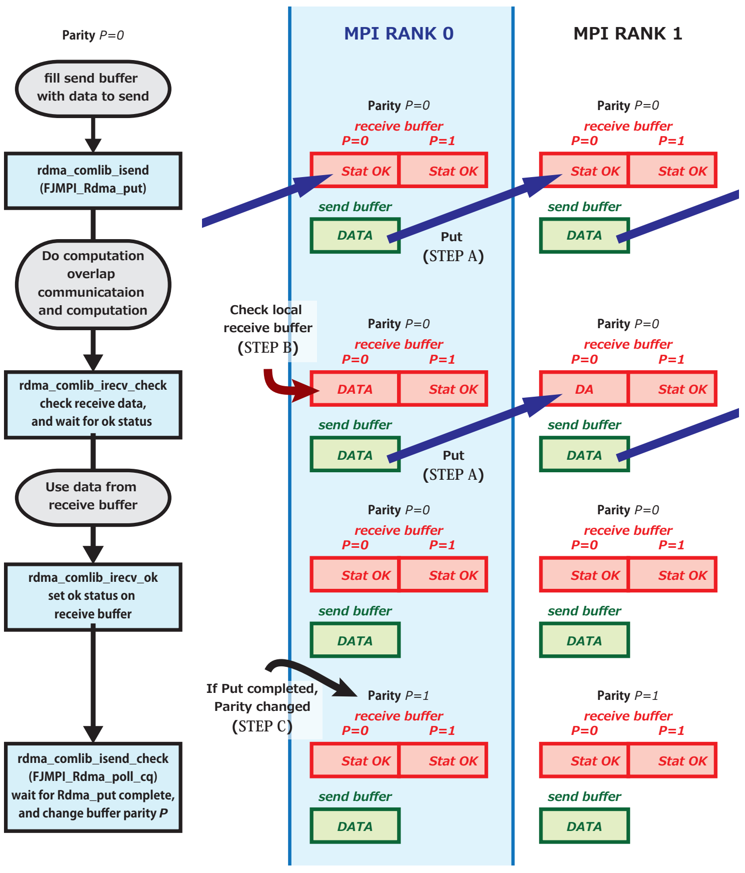

Parity P=0
fill send buffer with data to send
rdma_comlib_isend (FJMPI_Rdma_put)
Do computation overlap communicataion and computation
rdma_comlib_irecv_check check receive data, and wait for ok status
Use data from receive buffer
rdma_comlib_irecv_ok set ok status on receive buffer
rdma_comlib_isend_check (FJMPI_Rdma_poll_cq) wait for Rdma_put complete, and change buffer parity P
MPI RANK 0
MPI RANK 1
Parity P=0
receive buffer
P=0
P=1
Stat OK
send buffer
DATA
Put (STEP A)
Check local receive buffer (STEP B)
DA
If Put completed, Parity changed (STEP C)
Parity P=1

Figure 14.6: DB algorithm with Fujitsu's RDMA functions.

14.2.16 uTofu

4D lattice used in LQCD requires at most 8 directions to communicate, and in the domain decomposition scheme, 7 of them should take place simultaneously. Since **uTofu** provides an interface which enables simultaneous communication with those directions, it should be best suited for bringing the performance of Fugaku. It is important to verify and compare various implementation on the actual machine. The use of **uTofu** is also related to following subsections 14.2.17 and 14.2.18. In this subsection, we describe the implementation in qws-0.2.4 on the evaluation environment in the trial phase of Fugaku and the overview of the measured performance.

Usage of vcq Commutation instructions must be given through Virtual Control Queue (vcq). We prepare one vcq for each direction, 8 vcqs in total. A vcq transferring data to one direction also receives the data from the same direction. We use MPI communication to notify vcq id to the logical neighboring ranks in the application. All vcqs are constructed as thread safe with `UTOFU_VCQ_FLAG_THREAD_SAFE` flag so that maximally 8 put instructions can be simultaneously issued in thread parallel. With this thread parallelization, data transfer from memory to TNI (with domain decomposed algorithm, data copy between domains inside the rank as well) are accelerated.

Polling of the queue Notifications related to transferring (TCQ) and receiving (MRQ) piled in the vcqs must be removed by occasional pollings. Polling TCQ to confirm data transfer completion automatically removes other notifications related to the transfer. Notifications in MRQ are removed by additional pollings at the beginning of waiting for data receiving.

Specification change of the occupy state flag We have changed the specification of the occupy state flag in the **uTofu** version. This is to use the cache injection. The received data is written to memory by default but the cache injection directly writes them into the cache instead[4]. In order for cache injection to work, in addition to having a suitable memory alignment of the receive-buffer, no additional writes than receiving should be made to the buffer. For this reason, we changed from the receiver to the sender that updates the state of the flag in the data: the receiver does not change the state to "empty" but the sender updates the value of the flag before each transfer. The receiver holds the precious value and update of the value in the buffer is used to detect the completion of data receiving.

Process mapping We use **uTofu** to notify the process mapping (rankmap) in qws-0.2.4. We have changed this implementation to use MPI communication afterwards, however, what follows is the original implementation with **uTofu**.

Each rank needs to know the rank number of the neighbors in 4D LQCD coordinates. It is easy to calculate the six-dimensional (6D) Tofu coordinates of neighbors from own Tofu coordinate, however, it is non-trivial to obtain the neighbor's rank numbers. In qws-0.2.4, we use Atomic Read Modify Write (ARMW) of **uTofu** to exchange them. The ARMW communication requires vcq id of the neighboring ranks, which is calculated by using `utofu_create_vcq_with_cmp_id()` function. One further needs to know the value of control queue (cq) used in `utofu_create_vcq_with_cmp_id()` function to create a vcq, however, a way to obtain the cq is not given in the **uTofu** specification. We guessed it from the behavior of `utofu_create_vcq_with_cmp_id()`[5].

We used the Jacobi solver for 2D Laplace equation in the previous section to verify the behavior of DB scheme implemented with **uTofu**. We confirmed that the theoretical communication band width is well saturated and that a very good weak scaling up to three racks on the real machine. The environment was the evaluation environment in the trial phase with the compiler as of February, 2020. On the same environment, the MPI version roughly weak scaled but the throughput was about a half of the **uTofu** version.

We also verified that **uTofu** version of qws-0.2.4 shows roughly a weak scaling up to 36 racks on the evaluation environment in the trial phase and the compiler as of December 2019. The performance up to three racks was almost the same as that on a single node, but became about a half with more racks[6]. On the same environment, MPI version without rankmap did not show weak scaling in general. By combining the rankmap, it showed a scaling up 16 racks but slower than **uTofu** version by 40%, and did not scale at all with 36 racks.

14.2.17 Optimization of process mapping and TNI allocation

This subsection describes a method for searching for process mapping and TNI allocation related to the execution time of adjacent communication. This method searches a configuration that

[4]In order to verify this change, the cache injection is disabled by a flag to the put instruction in qws-0.2.4.

[5] This guess failed in the language environment lang/tcsds-1.2.27b so we stopped using ARMW and switched to use MPI communication instead. After that, a table between MPI ranks and Tofu coordinates are sent to all ranks with `MPI_Allreduce`.

[6] It turned out later that it was caused by the OS jitters.

minimizes the transfer message length of the Tofu link or TNI. In the following explanation, the space of process allocation is expressed as $\{T_1, T_2, \ldots, T_n\}$,

$$\begin{aligned}
&T_i = (t_i, s_i, o_i, h_i)\,, \\
&t_i \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\
&s_i \in \mathbb{N}\,, \quad o_i \in \{\mathrm{torus}, \mathrm{mesh}\}\,, \quad h_i \in \mathbb{N}\,,
\end{aligned} \tag{14.1}$$

where T_i is the respective axis of the network, t_i denotes the physical axis of Tofu (when multiple processes are allocated to one node, the axis is expressed as IN_NODE), s_i is the length of T_i, o_i is torus if the coordinates at both ends of T_i are adjacent to each other, otherwise it is mesh, and h_i is the distance between the adjacent coordinates in the T_i axis as viewed in the t_i axis. For example, a physical X-axis of length 24 (24 being the entire system, it is torus) is represented as (TX, 24, torus, 1).

The physical X-axis of length 24 is hypothetically divided into the axes TXc of length 2 and TXd of length 12, and the coordinate x of TX corresponds to the coordinate $[x \bmod 2]$ of TXc and the coordinate $[x/2]$ of TXd. they are (TX, 2,mesh, 1) and (TX, 12, torus, 2), respectively.

Step 1 For the input Tofu physical shape, enumerate the combinations of at most two divisions for each physical axis. In other words, for all s_1 and s_2 satisfying $s = s_1 \times s_2$ for a physical axis t of length s and tortuosity o, enumerate $\{(t, s_1, \mathrm{mesh}, 1), (t, s_2, o, s_1)\}$, where if either s_1 or s_2 is 1, then $\{(t, s, o, 1)\}$.

Step 2 For each physical axis, select one from those listed in Step 1 and combine them. Then, list all the combinations as P_2. For example, when the lengths of the physical X, Y, and Z axes are 24, 22, and 24, respectively, an example of $p_2 \in P_2$ is {(TX, 24, torus, 1), (TY, 11, mesh, 2), (TY, 2, mesh,1), (TZ, 8, torus, 3), (TZ, 3, mesh, 1), (TA, 2, mesh, 1), (TB, 3, torus, 1), (TC, 2, mesh, 1), (IN_NODE, 4, mesh,1)}.

Step 3 If all the elements of P_2 have already been processed, terminate the algorithm. For $p_2 \in P_2$, which has not yet been selected, select and combine axes to make a torus from p_2, and list the possible combinations assigned to each axis of the process partition. That is, when the lengths of the four axes of the process partition are (n_1, n_2, n_3, n_4) and we assign the following axesC_i, enumerate all combinations of (C_1, C_2, C_3, C_4).

$$C_i = \{(t_{i,1}, s_{i,1}, o_{i,1}, h_{i,1}), \ldots, (t_{i,n}, s_{i,n}, o_{i,n}, h_{i,n})\} \subset p_2\,, \tag{14.2}$$

where C_i satisfies $n_i \leq \prod_j s_{i,j}$, and two or more $t_{i,j}$ exist except for IN_NODE or $o_{i,j}$ = torus exists. We assume that the torus passing through n_i vertices can be constructed even if n_i is even and $\prod_j s_{i,j}$ (except j for which $t_{i,j}$=IN_NODE) is odd. We can also exclude combinations where there are j, k such that $t_{i,j} = t_{i,k}$, since they are equivalent to combinations that do not split that physical axis. For example, for an axis with process number 6, we can assign {(TY, 2,mesh, 1), (TZ, 3,mesh, 1)}. Let P_3 be the result of the enumeration.

Step 4 If all elements of P_3 have already been processed, return to Step 3. For $p_3 \in P_3$ that has not yet been selected, compute the corresponding F of the message length to be transferred for each combination of the Tofu axis and the LQCD axis of process division. The elements of F are

$$\begin{aligned}
&(t, q, d, m) \in F\,, \quad t \in \{\mathrm{TX}, \mathrm{TY}, \mathrm{TZ}, \mathrm{TA}, \mathrm{TB}, \mathrm{TC}, \mathrm{IN\_NODE}\}\,, \\
&q \in \{\mathrm{QX}, \mathrm{QY}, \mathrm{QZ}, \mathrm{QT}\}\,, \quad d \in \{\mathrm{plus}, \mathrm{minus}, \mathrm{both}\}\,, \quad m \in \mathbb{N}\,,
\end{aligned} \tag{14.3}$$

where t is the physical axis, q is the axis of process division of LQCD, d is the direction of transmission as seen in the axis of process division, and m is the message length. The following substeps are used to calculate F for the next assignment to the process partition axis q.

$$\{(t_1, s_1, o_1, h_1), \ldots, (t_n, s_n, o_n, h_n)\} \tag{14.4}$$

Step 4.1 On the q-axis, find the number of inter-node communication processes n_{inter} and the number of intra-node communication processes n_{intra} per node. Set the overall number of processes per node as n. If $t_i = \text{IN\_NODE}$, $n_q = s_i$; otherwise, $n_q = 1$. In this case

$$n_{\text{inter}} = \frac{n}{n_q}, \quad n_{\text{intra}} = \frac{(n_q - 1) \times n}{n_q}. \tag{14.5}$$

Step 4.2 Assign the length of the message sent per process at the q-axis to m_q. When $t_i \neq \text{IN\_NODE}$, let $(t_i, q, \text{both}, n_{\text{inter}} \times m_q \times h_i) \in F$. Multiply by h_i to include transfers to other nodes due to hops. If $t_i = \text{IN\_NODE}$, then $(\text{IN\_NODE}, q, \text{both}, n_{\text{intra}} \times m_q)$.

Step 5 For F obtained in Step 4, calculate the total transfer message length for each Tofu physical axis, where IN_NODE is excluded. The total transfer message length for physical axis t is the sum of m for all $(t, q, d, m) \in F$. If the best value among them exceeds the best value from other assignments obtained so far, return to step 4.

Step 6 For F obtained in Step 4, list the possible combinations of the six TNI assignments. For a physical axis t, when $(t, q_i, \text{both}, m_i) \in F$, we can use either Step 6a or 6b to assign TNI a, b ($a = b$ is also acceptable). These methods satisfy the restriction that, from the point of view of a Tofu link, there is only one TNI that uses the Tofu link. Let I_a, I_b be the set of communications assigned to a, b, respectively. Since there is no need to distinguish each of the six TNI, we can express $K = \{(I'_1, n_1), \ldots, (I'_m, n_m)\}$ instead of $(I_1, \ldots, I_6)$, where n_i is the number of j for which $I'_i = I_j$. In this way, when choosing the TNI to assign, we can reduce it to $|K|$ ways.

Step 6a When t is IN_NODE, or when there is exactly one axis of q_i that communicates in both directions in one adjacent communication (i.e., when there is one i such that $q_i \in \{\text{QY,QZ,QT}\}$), assign $(t, q_i, \text{ plus, } m_i) \in I_a$, $(t, q_i, \text{ minus, } m_i) \in I_b$ or $(t, q_i, \text{ minus, } m_i) \in I_a$, $(t, q_i, \text{ plus, } m_i) \in I_b$ to each different q_i. This means that in the former case, I_a is used for communication in the positive direction of q_i, regardless of the direction of t, and I_b is used for communication in the negative direction of q_i, regardless of the direction of t. When $q_i = \text{QX}$ and $a = b$, the communication of q_i never occurs in both directions at the same time, so only one of plus or minus shall be assigned. For example, when $\{(t, \text{QX}, \text{both}, m_x), (t, \text{QY}, \text{both}, m_y)\} \subset F$, the following assignments can be made respectively.

A) $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_a, \{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_b$

B) $\{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_a, \{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_b$

C) $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_a, \{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_b$

D) $\{(t, \text{QX}, \text{minus}, m_x), (t, \text{QY}, \text{minus}, m_y)\} \subset I_a, \{(t, \text{QX}, \text{plus}, m_x), (t, \text{QY}, \text{plus}, m_y)\} \subset I_b$

In this case, "A and D" and "B and C" are the same if a, b are swapped, respectively, so one of them can be omitted.

Step 6b When t is not an IN_NODE, assign $(t, q_i, \text{ both, } m_i) \in I_a$ for all i. This means that if t has only a one-way link (either TA or TC), the communication through t uses a regardless of the direction of q_i. If t has links in both directions, we further assign (t, q_i, both, m_i) $\in I_b$. This means that communication through one direction, positive or negative of t, will use a regardless of the direction of q_i, and communication through the other direction will use b regardless of the direction of q_i. Since we do not need to distinguish between the positive and negative values of T until the end, we assign the same value. When $a = b$, in order to express the existence of both directions, we further use ($\{t\}$, q_i, both, m_i) $\in I_a$.

Step 7 For the allocations listed in Step 6, calculate the transfer message length of the bottleneck TNI. Calculate the transfer message length of the TNI for the allocation $I = \{(t_1, q_1, d_1, m_1), \ldots, (t_n, q_n, d_n, m_n)\}$ in Step 7.1 and Step 7.2. If the transfer message length of the bottleneck TNI is smaller than the best value so far, update the best value. After completing Step 7, return to Step 4. The calculation of the transfer message length of the TNI by step 7 can be performed in the allocation by step 6, and if it exceeds the best value, the steps for the allocation can be terminated at that point.

Step 7.1 For each i for which t_i = IN_NODE, add m_i.

Step 7.2 In I', excluding those processed in Step 7.1, for each $q \in$ {QX,QY,QZ,QT}, divide it into $I' = \cup_q I_q$, where $I_q = \{(t_{q,1}, q, d_{q,1}, m_{q,1}), \ldots, (t_{q,n_q}, q, d_{q,n_q}, m_{q,n_q})\}$. For each I_q, add the largest $m_{q,i}$ of i that is $d_{q,i} \in$ {plus, both}. Furthermore, add the largest $m_{q,i}$ among i that is $d_{q,i} \in$ {minus, both}. In addition, add the largest $m_{q,i}$ of i that is $d_{q,i} \in$ {minus, both}. The reason why only the largest value is added is that only one link is used for communication in the same direction on the same axis.

14.2.18 Estimation and verification of the efficient process mapping and wait time of the nearest neighbor communication completion

Different lattices per process are used by ppptool and prototype to estimate the performance. They are $32 \times 6 \times 6 \times 2$ and $32 \times 6 \times 4 \times 3$. For each case, the wait time in unit of millisecond; [ms] of the nearest neighbor communication completion is shown in the following table. We state how to estimate in the following subsections.

lattice size per process	wait time [ms]	
	each	total
$32 \times 6 \times 6 \times 2$	0.018	0.36
$32 \times 6 \times 4 \times 3$	0.012	0.24

14.2.18.1 Estimate of the neighboring-communication timing for lattice size per process $32 \times 6 \times 6 \times 2$

The process division of 192^4 lattice by $32 \times 6 \times 6 \times 2$ lattice per process is $6 \times 32 \times 32 \times 96$. The amount of data transfer for each direction is

$$\begin{aligned}
\text{data transfer to QX}_\pm \text{ direction} &= 3456 \text{ bytes},\\
\text{data transfer to QY}_\pm \text{ direction} &= 9216 \text{ bytes},\\
\text{data transfer to QZ}_\pm \text{ direction} &= 9216 \text{ bytes},\\
\text{data transfer to QT}_\pm \text{ direction} &= 27648 \text{ bytes}.
\end{aligned}$$

The subscript $\pm$ indicates data transfer to forward and backward direction, respectively. For QX direction, transfer to only forward or backward direction is used in one set of neighboring communications. For other directions, transfer to both directions are needed.

By using the process mapping and TNI allocation search algorithm described in subsec. 14.2.17, one finds that the process map in Table 14.3 is the best, which uses a 6D $24 \times 22 \times 24 \times 2 \times 3 \times 2$

physical node shape. The map virtually divides each of TY and TZ axis into two axes. The axes TYc/TYd and TZc/TZd in Table 14.3 are the names of virtual axis made from TY and TZ axis, respectively.

Table 14.3: LQCD and Tofu axis (lattice size per process: $32 \times 6 \times 6 \times 2$).

QCD axis	number of process	Tofu axis	number of process per node	number of process w/ intra-node comm.	number of process w/ inter node comm.
QX	6	TYc,TZc (2×3)	1	0	4
QY	32	TYd,TB (11×3)	1	0	4
QZ	32	TZd (8)	4	3	1
QT	96	TA,TC,TX ($2 \times 2 \times 24$)	1	0	4

The TY axis divided into TYc axis with length 2 and TYd axis with length 11, and the TY coordinate y is associated with $y \bmod 2$ and $\left[\frac{y}{2}\right]$, respectively. Therefore, a neighbor in the TYc axis is also a neighbor in TY while a neighbor in the TYd axis is two-hop away in the TY axis. Similarly, the TZ axis divided into TZc axis with length 3 and TZd axis with length 8, and the TZ coordinate z is associated with $z \bmod 3$ and $\left[\frac{z}{3}\right]$, respectively. A neighbor in the TZc axis is also a neighbor in TZ while a neighbor in the TZd axis is three-hop away in the TZ axis. Since the TZ axis has a torus geometry, TZd axis becomes a torus as well. Therefore, we can assign the TZd axis to the QZ axis. The TNI assignment and message size for this process assignment is in Table 14.4. Each node has 6 TNIs and physical 6D Tofu coordinate (we denote

Table 14.4: TNI assignment and message size (lattice size per process: $32 \times 6 \times 6 \times 2$).

Direction in the QCD coordinate	Direction in the Tofu coordinate	TNI#	Message length per process (Byte)	Number of communication process	Link Multiplicity (number of hops)	Amount of data transfer to link/intra-node (Byte)
QX	TY±	0/1	3456	4	1	13824
	TZ±	2		4	1	13824
QY-	TB±	0	9216	4	1	36864
	TY±	0		4	2	73728
QY+	TB±	1	9216	4	1	36864
	TY±	1		4	2	73728
QZ-	TZ±	2	9216	1	3	27648
	LB	3		3	-	27648
QZ+	TZ±	2	9216	1	3	27648
	LB	3		3	-	27648
QT-	TA	4	27648	4	1	110592
	TC	4		4	1	110592
	TX±	4		4	1	110592
QT+	TA	5	27648	4	1	110592
	TC	5		4	1	110592
	TX±	5		4	1	110592

the path for the communication by link) has 10 directions. We assign TNIs such that each link corresponds to a single TNI. This constraint is to avoid conflicts in TNI-link data transfers caused by simultaneous using of a link by multiple TNIs. We use the same link and TNI for both sending and receiving, which makes sending and receiving symmetric. Together with the fact that the sending and receiving band widths are the same for TNIs and links, the arguments bellow apply to both sending and receiving. In the table, LB denotes a loop-back to the sender node.

The amount of the data transfer to link/intra-node is a sum of data transfer from all processes in the node and pass through data due to the multiple hoppings. The amount is obtained as a product of the message length, the number of communication processes and the link multiplicities. The number of communication processes are listed in Table 14.4. The link multiplicity is 2 for the communication in TYd direction and 3 in TZd direction.

Some rows in the Tables have “0/1” in the TNI assignment. Since the communication in QX direction dose not use TY- and TY+ simultaneously, it uses the same TNI as one of QY- and QY+ that shares the same link. For example, if communication to QX direction and QY- direction use the same link, QX uses TNI#0.

By using Table 14.4, we can calculate the amount of data transfer for each link and TNI. Since the QCD axes are mapped to a torus geometry, no communication to forward and backward directions of the same QCD axis shares the same link. To calculate the amount of data transferred through the link, we therefore add the data only to one direction. Furthermore, no communication uses multiple links simultaneously in sending data to one direction. To obtain amount of data transfer for each TNI, we add only the maximum one among the data transfer to links from the TNI. We show the obtained amount of the data transfer for each link and TNI in Table 14.5 and 14.6, respectively.

Table 14.5: Message size in a link (lattice size per process: $32 \times 6 \times 6 \times 2$).

Tofu coord. dirs.	QCD corrd. dirs.	TNI#	Injectgion size to a link (Byte)	Sum’d to Total	Total (Byte)
TA	QT-	4	110592	✓	110592
	QT+	5	110592		
TB-	QY-	0	36864	✓	36864
	QY+	1	36864		
TB+	QY-	0	36864	✓	36864
	QY+	1	36864		
TC	QT-	4	110592	✓	110592
	QT+	5	110592		
TX-	QT-	4	110592	✓	110592
	QT+	5	110592		
TX+	QT-	4	110592	✓	110592
	QT+	5	110592		
TY-	QX	0/1	13824	✓	87552
	QY-	0	73728	✓	
	QY+	1	73728		
TY+	QX	0/1	13824	✓	87552
	QY-	0	73728	✓	
	QY+	1	73728		
TZ-	QX	2	13824	✓	41472
	QZ-	2	27648	✓	
	QZ+	2	27684		
TZ+	QX	2	13824	✓	41472
	QZ-	2	27648	✓	
	QZ+	2	27684		
LB	QZ-	3	27648	-	-
	QZ+	3	27648	-	-

From Tables 14.5 and 14.6, one finds that the bottleneck is TA, TC, TX+, TX- and TNI#4, 5, of which amount of data transfer is 110,592 bytes and 110,592B/(6.12GB/s)=0.01807 [ms] is the waiting time for each set of neighboring communication. QCDJDD has 20 sets of neighboring communication in one iteration. Our estimation of waiting time for neighboring communication in the whole evaluation region is 0.36 [ms].

14.2.18.2 Verification of the neighboring-communication timing for lattice size per process $32 \times 6 \times 6 \times 2$

We have theoretically estimated the performance of the nearest neighbor communication region `comlib_recv_wait_all_c` in subsubsec. 14.2.18.1. In this subsection we examine the estimation method by actually performing benchmark run on K. To simulate Fugaku’s communication environment on K, we shrink the parallelization size to be contained in a bunch of nodes of K, while the communication pattern per process is identical to Fugaku. Although the process

Table 14.6: Message size in a TNI (lattice size per process: $32 \times 6 \times 6 \times 2$).

TNI#	QCD coord. dirs.	Tofu coord. dirs.	Injection size to a link (Byte)	Summ'd to Total	Total (Byte)
0	QX	TY-	13824	✓	87552
		TY+	13824		
	QY-	TB-	36864		
		TB+	36864		
		TY-	73728	✓	
		TY+	73728		
1	QX	TY-	13824	✓	87552
		TY+	13824		
	QY+	TB-	36864		
		TB+	36864		
		TY-	73728	✓	
		TY+	73728		
2	QX	TZ-	13824	✓	69120
		TZ+	13824		
	QZ-	TZ-	27648	✓	
		TZ+	27648		
	QZ+	TZ-	27648	✓	
		TZ+	27648		
3	QZ-	LB	27648	✓	55296
	QZ+	LB	27648	✓	
4	QT-	TA	110592	✓	110592
		TC	110592		
		TX-	110592		
		TX+	110592		
5	QT+	TA	110592	✓	110592
		TC	110592		
		TX-	110592		
		TX+	110592		

and TNI mapping pattern for Fugaku described in subsubsec. 14.2.18.1 cannot be completely simulated on K, we can verify the algorithm of process mapping and estimation of communication message length on the simulated benchmark on K as they are the same as those of Fugaku. We can examine the performance bottlenecks by eliminating the intra-node commutation and the TNI assignment on K as K only has four TNI's.

We employ the same parameters as those in subsubsec. 14.2.18.1 for the benchmark on K. The process mapping and the TNI assignment are shown in Tables 14.7 and 14.8, respectively. In this case, the message size per single link and TNI is shown in Tables 14.9 and 14.10, respectively. The estimation methods and meanings in the tables are the same as those in subsubsec. 14.2.18.1. From these estimate, we find that the performance bottleneck is in the $\mathrm{TZ}_{\pm}$ links and the message length is 124416 byte in this assignment.

Table 14.7: LQCD and Tofu axis for verification benchmark on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# o procs. w inter-node-comm.
QX	6	TYc,TZc (2×3)	1	0	4
QY	32	TYd,TB (11×3)	1	0	4
QZ	32	TZd,TA,TC ($8 \times 2 \times 2$)	1	0	4
QT	96	TX (24)	4	3	1

We perform a benchmark to examine these process mapping and TIN assignment with a reduced problem size on K. We employ the process topology $6 \times 6 \times 12 \times 3$ for the reduced problem size and the number of nodes is $2 \times 6 \times 9 \times 2 \times 3 \times 2 = 1296$. The process mapping is shown in Table 14.11, where we assign single process on a node and enlarge the message size by a factor four to simulate the load of a TNI link of Fugaku instead of placing four processes. By

Table 14.8: TNI assignment and message size for verification benchmark on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

QCD coord. dirs.	Tofu coord. dirs.	TNI#	Message length per procs. (Byte)	# of procs. communicating	Link multiplicity (# of hops)	Message size to link/intra-node (Byte)
QX	TY±	0/1	3456	4	1	13824
	TZ±	2/3		4	1	13824
QY−	TB±	0	9216	4	1	36864
	TY±	0		4	2	73728
QY+	TB±	1	9216	4	1	36864
	TY±	1		4	2	73728
QZ−	TA	2	9216	4	1	36864
	TC	2		4	1	36864
	TZ±	2		4	3	110592
QZ+	TA	3	9216	4	1	36864
	TC	3		4	1	36864
	TZ±	3		4	3	110592
QZ−	TX±	0	27648	1	1	27648
	LB	4		3	-	82944
QZ+	TX±	1	27648	1	1	27648
	LB	5		3	-	82944

Table 14.9: Message size in a link for verification benchmark on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

Tofu coord. dirs.	QCD corrd. dirs.	TNI#	Injection size to a link (Byte)	Sum'd to Total	Total (Byte)
TA	QZ−	2	36864	✓	36864
	QZ+	3	36864		
TB−	QY−	0	36864	✓	36864
	QY+	1	36864		
TB+	QZ−	2	36864	✓	36864
	QZ+	3	36864		
TC	QY−	0	36864	✓	36864
	QY+	1	36864		
TX−	QT−	0	27648	✓	27648
	QT+	1	27648		
TX+	QT−	0	27648	✓	27648
	QT+	1	27648		
TY−	QX	0/1	13824	✓	87552
	QY−	0	73728	✓	
	QY+	1	73728		
TY+	QX	0/1	13824	✓	87552
	QY−	0	73728	✓	
	QY+	1	73728		
TZ−	QX	2/3	13824	✓	124416
	QZ−	2	110592	✓	
	QZ+	3	110592		
TZ+	QX	2/3	13824	✓	124416
	QZ−	2	110592	✓	
	QZ+	3	110592		
LB	QT−	4	82944	-	-
	QT+	5	82944	-	-

this process mapping we can reserve the number of TNIs for a process and omit the intra-node communication. The TNI#4 and #5 assignments in Table 14.10 are eliminated and the load of TNI can be emulated.

When a Tofu axis is not decomposed for constructing QCD axis, the minimum length of the Tofu axis is two to emulate the Fugaku's communication pattern on K. When a Tofu axis is decomposed for constructing QCD axis, for ex. TY or TZ, the length of (TYc, TZc), which constructs nearest neighbor process coordinate in both of QCD and Tofu axis, must be the same length as that of Fugaku, while that of (TYd, TZd), which is non-nearest neighbor pro-

Table 14.10: Message size in a TNI for verification benchmark on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

TNI#	QCD coord. dirs.	Tofu coord. dirs.	Injection size to a link (Byte)	Summ'd to Total	Total (Byte)
0	QX	TY−	13824	✓	115200
		TY+	13824		
	QY−	TB−	36864		
		TB+	36864		
		TY−	73728	✓	
		TY+	73728		
	QT−	TX−	27648	✓	
		TX+	27648		
1	QX	TY−	13824	✓	115200
		TY+	13824		
	QY+	TB−	36864		
		TB+	36864		
		TY−	73728	✓	
		TY+	73728		
	QT+	TX−	27648	✓	
		TX+	27648		
2	QX	TZ−	13824	✓	124416
		TZ+	13824		
	QZ−	TA	36864		
		TC	36864		
		TZ−	110592	✓	
		TZ+	110592		
3	QX	TZ−	13824	✓	124416
		TZ+	13824		
	QZ+	TA	36864		
		TC	36864		
		TZ−	110592	✓	
		TZ+	110592		
4	QT−	LB	82944	✓	82944
5	QT+	LB	82944	✓	82944

cess coordinate in Tofu axis, must be three or longer than that of Fugaku. With this length assignment, load multiplicity of the links on Fugaku can be emulated on K. Table 14.11 shows the minimum node size satisfying the above requirement on K. With the node assignment, the message size of links and TNIs is the same as that of Tables 14.9 and 14.10, except that TNI#4 and #5 are not used as explained above and TB is used instead of TX for QT axis to construct 4D torus, in verification benchmark on K. In order to measure the actual message size and

Table 14.11: LQCD and Tofu axis for reduced size verification benchmark on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

QCD axis	# of procs.	Tofu axis	# of procs. in a node	Message length multiply factor per procs.
QX	6	TYc,TZc (2x3)	1	$\times$ 4
QY	6	TYd,TX (2x3)	1	$\times$ 4
QZ	12	TZd,TA,TC (3x2x2)	1	$\times$ 4
QT	3	TB (3)	1	$\times$ 1

timing on K, we employ a program which only performs the nearest neighbor communication of LQCD. Table 14.12 shows the message size in each link measured with the benchmark on K. The discrepancy to the theoretical estimate is about +3%, which comes from the overhead costs of packet header and receive-acknowledge packets. Our theoretical estimate for the message size is validated with this benchmark test on K.

Table 14.13 shows the timing for the nearest neighbor communication measured on K. The timing is for the region waiting for the completion of receiving data and evaluated from the total nearest neighbor communication time by subtracting the time to start data sending. The

Table 14.12: Message size in a link, measured on K (lattice size per process : $32 \times 6 \times 6 \times 2$).

Link	Theory(Byte)	Measured(Byte)	(Meas./Theo.)×100%
A	36864	37832	+2.6%
B+	27648	28512	+3.1%
B−	27648	28512	+3.1%
C	36864	37864	+2.7%
X+	36864	37832	+2.6%
X−	36864	37832	+2.6%
Y+	87552	89896	+2.7%
Y−	87552	89896	+2.7%
Z+	124416	127792	+2.7%
Z−	124416	127792	+2.7%

theoretical timing is estimated as 27.65 [μs] from the message size, 124416 byte, which is the largest and the bottleneck, and the effective bandwidth of Tofu network of K, 4.5 [GB/s].

Table 14.13: Measured nearest neighbor communication time on K (lattice size per process : $32 \times 6 \times 6 \times 2$) .

Region	Theory (μsec)	Measured(μsec)	(Meas./Theo.)×100 %
Wait for completion of data receiving	27.65	30.83	+11.5%
Start data sending	-	3.71	-
Total	-	34.54	-

The timing measured is larger than the theoretical estimation by 11.5%. The discrepancy could be explained by the reduced efficiency in links where traffic congestion occurs due to the multiplicity of QCD axis in Tofu axis or performance saturation of a TNI receiving data from several QCD axes simultaneously. It is difficult to precisely evaluate these performance bottlenecks as these occur and change depending on the execution timing. We adopt theoretically estimated timing value for the QCD nearest neighbor communication performance on Fugaku, because it causes only 3.6% increase in the total timing even with 11.5% increase in the time waiting for data receiving.

14.2.18.3 Estimate of the neighboring-communication timing for lattice size per process $32 \times 6 \times 4 \times 3$

The process mapping shown in Table 14.14 is also valid and appropriate for the case that the lattice size of a process is $32 \times 6 \times 4 \times 3$ as shown in subsubsec. 14.2.18.1. The message size and TNI assignment are shown in Table 14.15. Tables 14.16 and 14.17 show that the communication bottlenecks are in the directions, TB−, TB+, TY−, TY+, with assignments TNI#2, #3, #4, #5, and the message size is 73,728 byte. Thus we estimate the time for completion of receiving data as 73728B/(6.12GB/s) = 0.01205 [ms]. Because the nearest neighbor communication occurs 20 times in one iteration of QCDJDD, the time for completion of receiving data in the whole estimation region is estimated to be 0.24 [ms]. This estimate has been verified to be almost correct by benchmark tests on Fugaku (see Table 14.30).

Table 14.14: LQCD and Tofu axis (lattice size per process : $32 \times 6 \times 4 \times 3$),

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# of procs. w inter-node-comm.
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	48	TX (24)	2	2	2
QT	64	TY,TB (22×3)	1	0	4

Table 14.15: TNI assignment and message size (lattice size per process : $32 \times 6 \times 4 \times 3$).

QCD coord. dirs.	Tofu corrd. dirs.	TNI#	Message length per procs. (Byte)	# of procs. communicating	Link multiplicity (# of hops)	Message size to link/intra-node (Byte)
QX	TA	0	3456	4	1	13824
	TZ±	0/1		4	1	13824
QY−	TC	0	9216	2	1	18432
	TZ+	0		2	3	55296
	LB	3		2	-	18432
QY+	TC	1	9216	2	1	18432
	TZ−	1		2	3	55296
	LB	2		2	-	18432
QZ−	TX	3	13824	2	1	27648
	LB	3		2	-	27648
QZ+	TX	2	13824	2	1	27648
	LB	2		2	-	27648
QT−	TB±	5	18432	4	1	73728
	TY±	5		4	1	73728
QT+	TB±	4	18432	4	1	73728
	TY±	4		4	1	73728

Table 14.16: Message size in a link (lattice size per process : $32 \times 6 \times 4 \times 3$).

Tofu coord. dirs.	QCD coord. dirs.	TNI#	Injection size to a link(Byte)	Summ'd to Total	Total (Byte)
TA	QX	0	13842	✓	13842
TB−	QT∓	5/4	73728	✓	73728
TB+	QT±	4/5	73728	✓	73728
TC	QY∓	0/1	18432	✓	18432
TX−	QZ−	3	27648	✓	27648
TX+	QZ+	2	27648	✓	27648
TY−	QT∓	5/4	73728	✓	73728
TY+	QT±	4/5	73728	✓	73728
TZ−	QX	1	13824	✓	69120
	QY+	1	55296	✓	
TZ+	QX	0	13824	✓	69120
	QY−	0	55296	✓	
LB	QY−	3	18432	-	-
	QZ−	3	27648	-	-
	QY+	2	18432	-	-
	QZ+	2	27648	-	-

Table 14.17: Message size in a TNI (lattice size per process : $32 \times 6 \times 4 \times 3$).

TNI#	QCD coord. dirs.	Tofu coord. dirs.	Injection size to a link (Byte)	Summ'd to Total	Total (Byte)
0	QX	TA	13824		69120
	QX+	TZ±	13824	✓	
	QY−	TC	18432		
		TZ+	55296	✓	
1	QX−	TZ∓	13824	✓	69120
	QY+	TC	18432		
		TZ−	55296	✓	
2	QZ+	TX+	27648	✓	73728
		LB	27648	✓	
	QY+	LB	18432	✓	
3	QZ−	TX−	27648	✓	73728
		LB	27648	✓	
	QY−	LB	18432	✓	
4	QT+	TB±	73728	✓	73728
		TY±	73728		
5	QT−	TB∓	73728	✓	73728
		TY∓	73728		

14.2.18.4 Verification of the neighboring-communication timing for lattice size per process $32 \times 6 \times 4 \times 3$

We discussed how to verify the elapsed time for the `comlib_recv_wait_all_c` region that waits the nearest neighboring communication is finished in the case of lattice size per process $32 \times 6 \times 4 \times 3$ in software arrangement (1). In the same way as in subsubsec. 14.2.18.2, the situation of communication can be reproduced with small number of nodes on Fugaku. Without change of the Tofu axes, as shown in Table 14.18, by reducing size of TY to (TX $\times$ TA) $\times$ (TY $\times$ TB) $\times$ (TZ $\times$ TC) $= (24 \times 2) \times (2 \times 3) \times (24 \times 2)$ on 36 racks, the communication pattern is reproduced with the same links and TNI assignment as in the target system size. For the number of racks less than 36, the torus geometry in X-axis and Z-axis is lost. In such cases, by exchanging X-axis and Z-axis, the torus is composed in X-axis and Y-axis of Tofu. For example, in the case of 3 racks with (TX $\times$ TA) $\times$ (TY $\times$ TB) $\times$ (TZ $\times$ TC) $= (2 \times 2) \times (2 \times 3) \times (24 \times 2)$, the communication pattern is reproduced by assigning the LQCD axes to the Tofu axes as in Table 14.19. In this case one should note that TNI assignment for the X-axis and B-axis are exchanged. The TZd axis is not needed to be torus since the QY axis is composed of the TZd axis and TC axis. Therefore, without requiring the Tofu Z-axis being torus, the communication pattern can be reproduced. The minimum construction is realized with 432 nodes as (TX $\times$ TA) $\times$ (TY $\times$ TB) $\times$ (TZ $\times$ TC) $= (2 \times 2) \times (2 \times 3) \times (9 \times 2)$, as listed in Table 14.20.

Table 14.18: LQCD and Tofu axis for verification benchmark with 36 racks of Fugaku (lattice size per process : $32 \times 6 \times 4 \times 3$).

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# of procs. w inter-node-comm.
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	48	TX (24)	2	2	2
QT	6	TY,TB (2×3)	1	0	4

Table 14.19: Same table as the Table 14.18, but with 3 racks.

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# of procs. w inter-node-comm.
QX	6	TA,TZc (2×3)	1	0	4
QY	32	TZd,TC (8×2)	2	2	2
QZ	6	TB (3)	2	2	2
QT	4	TY,TX (2×2)	1	0	4

Table 14.20: Same table as the Table 14.18, but wit 432 nodes.

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# of procs. w inter-node-comm.
QX	6	TA,TZc (2×3)	1	0	4
QY	12	TZd,TC (3×2)	2	2	2
QZ	6	TB (3)	2	2	2
QT	4	TY,TX (2×2)	1	0	4

14.2.18.5 Considering execution with 1 MPI process on 4 CMG

To avoid the bottleneck in TNI, we are examining the execution with 1 MPI process on 4 CMGs. Parallelizing a 192^4 lattice with 147456 processes, the minimum communication surface is achieved by the lattice size per process of $n_x \times n_y \times n_z \times n_t = 32 \times 6 \times 6 \times 8$ by process division $6 \times 32 \times 32 \times 24$. In this case the communication size in each direction reads as follows.

transferred data size in $QX_\pm$ direction =	13824 bytes
transferred data size in $QY_\pm$ direction =	36864 bytes
transferred data size in $QZ_\pm$ direction =	36864 bytes
transferred data size in $QT_\pm$ direction =	27648 bytes

Similarly to the discussion about the efficient process mapping, each of the TX axis and TY axis is virtually divided into two axes. The TX axis is divided into the TXc axis of length 4 and the TXd axis of length 6, and the TX coordinate x corresponds to $x \bmod 4$ and $\left[\frac{x}{4}\right]$, respectively. The TY axis is divided into the TYc axis of length 2 and the TYd axis of length 12, and the TY coordinate z corresponds to $z \bmod 2$ and $\left[\frac{z}{2}\right]$, respectively. Table 14.21 displays the assignment of the QCD axes to the Tofu axes.

Table 14.21: LQCD and Tofu axis (1 MPI process on 4 CMGs).

QCD axis	# of procs.	Tofu axis	# of procs. in a node	# of procs. w intra-node-comm.	# of procs. w inter-node-comm.
QX	6	TXd	1	0	1
QY	32	TB×TYd	1	0	1
QZ	32	TA×TC×TXc×TYc	1	0	1
QT	24	TZ	1	0	1

The message size in each link in this case is summarized in Table 14.22. If one assigns the links and TNI as in Table 14.23, TNI#2,3,5 become bottlenecks so that the waiting time for receive in each nearest neighbor communication consumes 110592B/(6.12GB/s)=0.01807 [ms]. This amounts to the same elapsed time discussed in subsubsec. 14.2.18.1 and is longer than the elapsed time discussed in subsubsec. 14.2.18.3.

Table 14.22: Message size in a link (1 MPI process on 4 CMGs).

Link	Message size	Calculation
$TX_\pm$	92160 Byte	$QX_\pm$-dir. size × 4 hops + $QZ_\pm$-dir. size
$TY_\pm$	110592 Byte	$QZ_\pm$-dir. size + $QY_\pm$-dir size × 2 hops
$TZ_\pm$	27648 Byte	$QT_\pm$-dir. size
TA	36864 Byte	$QZ_\pm$-dir. size
$TB_\pm$	36864 Byte	$QY_\pm$-dir. size
TC	36864 Byte	$QZ_\pm$-dir. size

Table 14.23: Message size in a TNI (1 MPI process on 4 CMGs).

TNI No.	Link	Message size	Calculation
0	TX_+	92160 Byte	TX
1	TX_-	92160 Byte	TX
2	TY_+	110592 Byte	TY
3	TY_-	110592 Byte	TY
4	TZ_+, TZ_-, TA	92160 Byte	$TZ_+ + TZ_- +$ TA
5	TB_+, TB_-, TC	110592 Byte	$TB_+ + TB_- +$ TC

14.3 Five-dimensional fermion matrix code

In LQCD, not only the Wilson-type fermion described so far but also variety of fermion matrices are practically used. As an example, we consider the five-dimensional (5D) formulation that improves the symmetry of light quarks. Since the 4D structure corresponding to space-time is the same as the Wilson-type matrix, by the LQCD kernel library developed above can be efficiently employed. On the other hand, for exploiting the cache effect it may deserve to treat the fifth-dimensional lattice sites (typically of 10 to 100) as one of on-site degrees of freedom. We compare these two implementations and thereby generalize the tuning techniques established in the development of the LQCD library.

We have implemented the codes of 5D fermion matrix in these two approaches. In the first approach, we employ an existing general-purpose code set and modify its 5D fermion matrix so as to call the LQCD library. In the second approach, we directly developed the 5D fermion matrix code by referring the implementation of the LQCD library. Although in the preliminary performance measurement the latter code shows better performance, this may be caused partially by less efficient implementation of linear algebraic functions in the former as well as the parameter setup was not that the LQCD library exhibits its best performance. There is still room to improve the performance of both the codes and continuous improvement is still underway.

14.4 Study on improvement of nuclear force code

On 192^4 lattice, the memory size needed to keep a single quark propagator is 16 Byte $\times 3 \times 4 \times 3 \times 4 \times 192^3 = 16.3$ GByte. In order to calculate the spin-orbit forces (LS forces) and the anti-symmetric LS forces in the coupled channel formalism including nuclear and hyperon forces in the odd parity sector in 2+1 flavor QCD, it is necessary to keep seven propagators for up and down quarks corresponding to the directions of momentum of quarks and seven propagators for strange quark. The total memory size amounts to 228 GByte. Since this exceeds the memory capacity of a single node of Fugaku, quark propagators have to be spatially divided.

On the other hand, 3D FFT is used for an efficient calculation of four point functions of baryon fields. The current code performs the calculations by generating Wick contractions of four point functions of baryon fields (semi-)automatically both for 115 channels of all the combinations of two hyperons and for 7 directions corresponding to the directions of momentum of two baryons. This requires 82,800,000 3D FFTs. Due to the situation where the quark propagators are spatially divided, the 3D FFT is carried out with MPI communication, which is difficult to perform. As a result, a very complicated algorithm and coding is needed to use the momentum wall source to calculate nuclear and hyperon forces in the negative parity sector. However, we have developed a method of dividing the four point functions of baryon fields into parts and storing them after 3D FFT is over. We are currently investigating a better method.

14.5 Items requested from codesign to the system implementation

The implementation requests and improvement items that were fed back to the system through this codesign are listed below.

1. MPI_Allreduce up to 3 elements

   The most time-consuming part of LQCD is the quark solver by the iterative method. In the target problem, the execution time per iteration is very short, approximately one millisecond, and the execution time of MPI_Allreduce is non-negligible, thus we decided to have a fast processing mechanism of MPI_Allreduce up to three elements.

   MPI_Allreduce of 1 to 3 elements on Fugaku was realized by performing reduction and broadcast processing in a tree structure using a dedicated hardware resource called TBI (Tofu Barrier Interface), similar to the 1-element MPI_Allreduce in K.

2. Zero OS jitter

Since the execution time per iteration is very short (about 1 millisecond) in the target problem, the effect of OS jitter cannot be ignored. Therefore, Fugaku was equipped with an assistant core to prevent OS jitter from occurring in the computational core.

3. RDMA communication mechanism

   In quark solver, the sparse matrix-vector product is the main part of the operation. In the target problem, the problem size per process is so small that all the data used in the iterative method can be placed in the L2 cache and the time required for the out-of-process data transfer for the sparse matrix-vector product and the execution time for the in-process closed sparse matrix-vector product operation are comparable. This concluded that a DMA communication mechanism for overlapping adjacent communications and operations was also necessary for LQCD. And RDMA was implemented on Fugaku following K.

4. Reducing issue time for multiple RDMA communications

   LQCD needs to issue 8 RDMA communications in a single sleeve area communication in 4D stencil calculation. As the overhead caused by issuing RDMA communication is large in K, we examined the API for issuing multiple RDMA communication and its time-saving effect. The use of uTofu made it possible to issue multiple RDMA communications.

5. Countermeasure for the problem of a significant increase in communication time due to the impact of other jobs

   In K, there are cases that the communication time increases significantly due to the influence of other jobs. The cause of this phenomenon is largely due to the "I/O communication of other jobs" that share the communication path. Since LQCD has a relatively large adjacent communication time, we sought to counter it on Fugaku. This problem can be solved by considering the symmetry (X, Y, Z)=(2, 2, 8) for both SIO and GIO, assigning jobs to the starting coordinates of the rack, and fixing the placement of I/O nodes (SIO, GIO) in the node geometry. Specifically, set "4lx6mx16n:strict-io" for the "node" factor of the job submission command as follows.

   pjsub -L node=4lx6mx16n:strict-io job.sh ($l, m, n = 1, 2, 3\dots$)

6. The list of implementation requests for the system, including other compiler optimization improvements and library issues

   (a) When inline expanding a function that returns an array of size equal to the SIMD width, the value should be returned via a register

      Developed as a mechanism to reduce the number of instructions related to value return during function inline expansion.

   (b) Enhanced Tree Height Reduction optimization

      Added the ability to select an algorithm for Tree Height Reduction optimization.

   (c) Removal of temporary arrays used across multiple loops

      After applying the Tree Height Reduction optimization, the existing optimization functions made it possible.

(d) Reduction of the number of integer registers used by utilizing the addressing mode during SIMD

Developed as an addressing mode selection function for Fugaku.

(e) GEMM library that takes as its argument an array in which the real and imaginary parts of complex numbers are stored consecutively and alternately for the number of SIMD elements

Matrix matrix product function for arrays with separated real and imaginary parts in SIMD element units has been added.

(f) Matrix transposition function

Developed matrix transposition function as a BLAS extension.

(g) Batch mode support for FFT

Consider supporting this with FFTW.

14.6 Baseline performance measured on K

We run the quark solver on the target problem 192^4 using 8 OpenMP threads per node on 82944 nodes of K. The process decomposition of QCDJDD on the virtual 4D node-shape and its layout on the physical 6D node-shape (Tofu) are as follows:

$$\begin{aligned}&\mathrm{TX} \times \mathrm{TY} \times \mathrm{TZ} \times \mathrm{TA} \times \mathrm{TB} \times \mathrm{TC} = 24 \times 18 \times 16 \times 2 \times 3 \times 2\\ &\mathrm{QX} \times \mathrm{QY} \times \mathrm{QZ} \times \mathrm{QT} = 24 \times 24 \times 24 \times 6\end{aligned}$$

The lattice size per node is $8 \times 8 \times 8 \times 32$ with this 4D process decomposition. This physical 6D node-shape are chosen for a better communication throughput and a shorter latency on the virtual 4D process layout. The mapping between the Tofu network and virtual 4D network is as follows.

$$\begin{aligned}&\mathrm{QX} = \mathrm{TX} = 24\\ &\mathrm{QY} = \mathrm{TYc} \times \mathrm{TZd} = 3 \times 8 = 24\\ &\mathrm{QZ} = \mathrm{TZc} \times \mathrm{TA} \times \mathrm{TB} \times \mathrm{TC} = 2 \times 2 \times 3 \times 2 = 24\\ &\mathrm{QT} = \mathrm{TYd} = 6\end{aligned}$$

where we decompose the Tofu axes TY and TZ as (TY = 18 = TYd$\times$TYc, TYd=6, TYc=3), (TZ = 16 = TZd$\times$TZc, TZd=8, TZc=2), and combine them to have QY, QZ, and QT appropriately. With this mapping, the processes in the QX direction and QZ direction are nearest neighboring in the TX direction and the TZ direction, respectively. They are nearest neighboring in the TY direction or next-to-nearest (two-hops) in the TZ direction for the QY direction, and are next-to-next-to-nearest (three-hops) in the TY direction for the QT direction.

The measured performance is shown in Table 14.24 for single iteration of QCDJDD on K. The computation efficiency is estimated based on the peak double precision performance as 1. The performance on the regions `ddd_in_s_` and `comlib_recv_wait_all_c` are measured with communication-computation overlapping on K, while it will be estimated theoretically with the ideal overlapping by taking the longest estimated time from the two regions in the Fugaku performance estimate.

Table 14.24: LQCD baseline performace measured on K.

		Exec. Time [ms]	Comp. Eff. (%)
Total time for computationl regions		27.99	34.8
Region name	`jinv_ddd_in_s_`	14.09	41.9
	`ddd_in_s_`	6.52	44.3
	`ddd_out_pre_s_`	0.95	12.6
	`ddd_out_pos_s_`	3.84	16.9
	`other_calc`	2.58	7.1
Total time communication regions		2.66	
Region name	`comlib_irecv_all_c`	0.45	
	`comlib_isend_all_c`	0.17	
	`comlib_recv_wait_all_c`	0.16	
	`comlib_send_wait_all_c`	0.17	
	`s_drbicgstab_dd_hpc_iter_reduc1_`	0.18	
	`s_drbicgstab_dd_hpc_iter_reduc2_`	0.85	
	`s_drbicgstab_dd_hpc_iter_reduc3_`	0.67	
Time for overlapping		-	
Total time		30.65	31.8

14.7 Performance estimation for Fugaku

In this section, we describe the performance estimation method using ppptool and prototype before the actual Fugaku became available. However, the results obtained by ppptool are omitted due to NDA. In the performance estimation with ppptool and prototype, time for each communication region has been estimated as follows. The result is shown in Table 14.25.

- `s_drbicgstab_dd_hpc_iter_reduc[123]_`
  A high speed version of MPI_Allreduce, which can process three individual reductions simultaneously, will be used.
- `comlib_irecv_all_c`
  It is estimated as zero second as the region can be eliminated by using the DB algorithm.
- `comlib_isend_all_c`
  It is estimated from the execution time of the code only involving the communication regions.
  - We measure the execution time on K with the ppptool, and use it as the estimation on Fugaku.
  - For the estimate using prototype, we measure the time on prototype actually for better estimate.
- `comlib_send_wait_all_c`
  It is assumed that the timing is the same as K, wihch was measured with a code that involves only the communication regions.
- `comlib_recv_wait_all_c`
  It is estimated from the optimal waiting time for the completion of the nearest neighbor communication (see also subsec. 14.2.18).

14.7.1 Performance estimate with prototype.

Table 14.26 shows the performance results measured on the two nodes of prototype. qws-0.2.2+patch_200107 was used for the measurement. The lattice size per process is $32 \times 6 \times 4 \times 3$. CE was not measured and PC is a tentative reference value.

Table 14.25: Estimated Time in [ms] for the communication regions.

Region name	Lattice size per process							
	32 × 6 × 6 × 2 (For estim. with ppptool)				32 × 6 × 4 × 3 (For estim. with prototype)			
	Eco mode OFF		Eco mode ON		Eco mode OFF		Eco mode ON	
	Normal	Boost	Normal	Boost	Normal	Boost	Normal	Boost
comlib_irecv_all_c	0	0	0	0	0	0	0	0
comlib_isend_all_c	0.13	0.13	0.13	0.13	0.081	0.073	0.082	0.074
comlib_recv_wait_all_c	0.36	0.36	0.36	0.36	0.241	0.241	0.241	0.241
comlib_send_wait_all_c	0.11	0.11	0.11	0.11	0.106	0.106	0.106	0.106
s_drbicgstab_dd_hpc_iter_reduc1_	0.02	0.02	0.02	0.02	0.020	0.020	0.020	0.020
s_drbicgstab_dd_hpc_iter_reduc2_	0.02	0.02	0.02	0.02	0.021	0.021	0.021	0.021
s_drbicgstab_dd_hpc_iter_reduc3_	0.04	0.04	0.04	0.04	0.043	0.043	0.043	0.043
in Total	0.68	0.68	0.68	0.68	0.512	0.504	0.513	0.505

Table 14.26: Performance estimate with prototype.

Mode	Eco mode OFF						Eco mode ON					
	Normal mode			Boost mode			Normal mode			Boost mode		
Region name	ET [ms]	CE [%]	PC [W]	ET [ms]	CE [%]	PC [W]	ET [ms]	CE [%]	PC [W]	ET [ms]	CE [%]	PC [W]
all_calc	0.452	-	-	0.412	-	-	0.509	-	-	0.465	-	-
overlapped	0.138	-	-	0.126	-	-	0.159	-	-	0.145	-	-
Computation region	0.590	-	-	0.538	-	-	0.668	-	-	0.610	-	-
Communication region	0.512	-	-	0.504	-	-	0.513	-	-	0.505	-	-
Time for overlapping	0.138	-	-	0.126	-	-	0.159	-	-	0.145	-	-
Total	0.964	-	117	0.916	-	138	1.022	-	85	0.970	-	99

14.8 Benchmark tests on Fugaku

Benchmark tests are performed on the boost mode 2.2 GHz without using Eco modes that one of two floating-point arithmetic pipelines is limited. Elapse time and performance of 500 iterations of QCDJDD and ddd_in_s_ region measured on two MPI processes using two CMGs for several problem sizes per MPI process are listed in Table 14.27 and Table 14.28. FLOP indicates a floating-point operation count calculated theoretically. Efficiency indicates floating-point operation efficiency against single precision floating-point operation peak. We see that performance for $32 \times 6 \times 4 \times 6$ is the best in the tests if the communications are not taken into consideration.

Table 14.27: Elapse time and performance for 500 iterations of QCDJDD.

Size	Elapse [s]	TFLOPS	Efficiency	FLOP
32 × 6 × 4 × 3	0.334867	0.8272	12.24%	69254421000
32 × 6 × 4 × 6	0.515010	1.0839	16.04%	139560981000
32 × 6 × 8 × 6	1.145754	0.9786	14.48%	280304661000
32 × 6 × 8 × 12	2.606202	0.8616	12.75%	561369621000
32 × 12 × 8 × 12	5.778703	0.7773	11.50%	1122981141000

We use a fast Allreduce using the Tofu barrier in quark solver. The Allreduce up to three elements for MPI_DOUBLE and MPI_FLOAT can be performed on the Tofu barrier. In Table 14.29, we show Allreduce benchmark results on 72 racks, 27648 nodes of $48 \times 12 \times 48$ node shape by using "Intel(R) MPI Benchmarks 2019 Update 6, MPI 1 part (IMB-MPI1)" with and without Tofu barrier. Minimum (min), maximum (max), and average (avg) time for repetition number, 10000 are shown. The number of bytes (byte) is a message length to be reduced per one MPI_Allreuce call. And the number of counts (count) is a number of elements. The data type of MPI_FLOAT is reduced as default of IMB-MPI1. We see that Allreduce up to three elements with the Tofu barrier is about six times faster than one without the Tofu barrier and it is faster to split MPI_Allreduce for 15 elements into five MPI_Allreduce for three elements.

Table 14.28: Same as Table 14.27, but for region of ddd_in_s_ during 500 iterations of QCDJDD.

Size	Elapse [s]	TFLOPS	Efficiency	FLOP
$32 \times 6 \times 4 \times 3$	0.068043	1.1208	16.58%	19065600000
$32 \times 6 \times 4 \times 6$	0.119455	1.3170	19.49%	39329280000
$32 \times 6 \times 8 \times 6$	0.219403	1.4693	21.74%	80593920000
$32 \times 6 \times 8 \times 12$	0.559297	1.1699	17.31%	163584000000
$32 \times 12 \times 8 \times 12$	1.192644	1.1146	16.49%	332328960000

Table 14.29: Allreduce benchmark by IMB-MPI1.

		with Tofu barrier			without Tofu barrier		
byte	count	min [μs]	max [μs]	avg [μs]	min [μs]	max [μs]	avg [μs]
0	0	0.09	0.14	0.10	0.10	0.16	0.12
4	1	7.60	11.33	9.46	55.69	69.05	62.83
8	2	8.25	10.79	9.50	55.79	68.93	62.91
12	3	8.25	10.93	9.57	55.89	69.02	62.94
16	4	58.99	66.95	62.68	56.42	69.71	63.51
32	8	61.50	72.34	66.32	78.24	97.57	88.14
64	16	61.61	72.38	66.31	78.63	97.84	88.42
128	32	63.70	74.45	68.43	80.46	99.56	90.10

We show a weak scaling plot of the evaluation region in Fig. 14.7. The vertical line at 147456 nodes denotes the number of nodes used in the benchmark for the target problem size while 158976 nodes is a total nodes of Fugaku. We see a nice weak scaling from 432 nodes to 147456 nodes of the target nodes with a few exceptions caused by OS jitters. The elapse time increases 0.5 [ms] (about 7 %) from 432 nodes to 147456 nodes due to the time for Allreduce. The elapse times of five benchmark tests on 147456 nodes are 0.8000, 0.7998, 0.7982, 0.7989, and 0.7978 [ms], respectively. These are about 38.3 times faster than the elapse time, 30.65 [ms], for same problem setup on the full system of K. Performance is 102 PFLOPS, 10% floating-point operation efficiency against single precision floating-point operation peak. Averaged power is about 20 MW. The power efficiency is 5 GFLOPS/W.

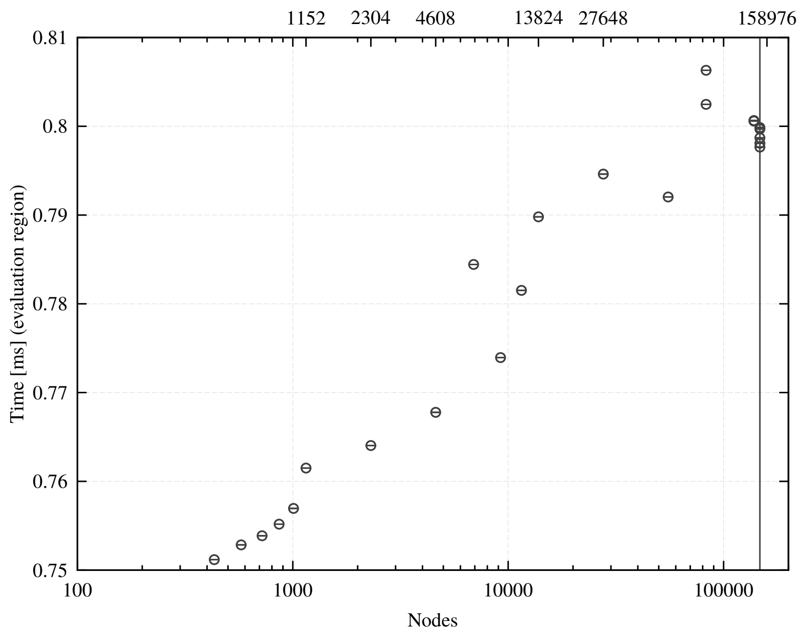


Figure 14.7: Weak scaling of the evaluation region.

Table 14.30 shows a breakdown of the total elapse time for the target problem size on 147456 nodes. The total time is divided into calculation time (all_calc) and communication time (all_comm). all_comm is divided into three parts for neighboring communication and three parts for Allreduce. Summed total elapse time in the table is slightly longer than 0.8000 [ms] which is measured in peak performance tests because there is a non-negligible overhead to measure elapse times for each region. We see half of time is spent for communication. In usual production runs, we may better use a smaller number of nodes.

Table 14.30: Elapse time breakdown.

region	Elapse time [ms]
all_calc	0.400
all_comm	0.407
comlib_isend_all_c	0.029
comlib_recv_wait_all_c	0.254
comlib_send_wait_all_c	0.062
s_drbicgstab_dd_hpc_iter_reduc1_	0.015
s_drbicgstab_dd_hpc_iter_reduc1_	0.016
s_drbicgstab_dd_hpc_iter_reduc1_	0.031
total	0.807

14.9 Summary

We have achieved 102 PFLOPS, 10% floating-point operation efficiency against single precision floating-point operation peak, of Clover–Wilson quark solver on 192^4 lattice on Fugaku through the codesign in FS2020 project. From this codesign activity, several feed backs especially on communication was send to the system implementation. All the benchmark results on Fugaku have been obtained on the evaluation environment in the trial phase. It does not guarantee the performance, power and other attributes of Fugaku at the start of its public use operation.

Since this chapter was written in terms of the codesign of LQCD in the FS2020 project, it also described optimizations that were rejected during the codesign process. The details of the algorithms and optimizations that performed best in the codesign of LQCD are discussed more professionally in our paper [136]. Finally, we would like to thank all the people who were involved in the LQCD working group of the project.

14.10 Author/Co–Authors

14.10.0.0.1 Author

- Yoshifumi Nakamura (R-CCS, RIKEN)

14.10.0.0.2 Co–Authors (in alphabet order)

- Ken-Ichi Ishikawa (GSASE, Hiroshima U.)
- Issaku Kanamori (R-CCS, RIKEN)
- Hideo Matsufuru (Computing Research Center, KEK)
- Keigo Nitadori (R-CCS, RIKEN)
- Miwako Tsuji (R-CCS, RIKEN)

References

[129] Y. Nakamura, Y. Mukai, K.-I. Ishikawa, I. Kanamori. *Lattice quantum chromodynamics simulation library for Fugaku and computers with wide SIMD.* (https://github.com/RIKEN-LQCD/qws).

[130] T. Boku et al. "Multi-block/multi-core SSOR preconditioner for the QCD quark solver for K computer". In: *PoS* LATTICE2012 (2012). Ed. by Derek Leinweber et al., p. 188. DOI: `10.22323/1.164.0188`. arXiv: `1210.7398 [hep-lat]`.

[131] Ken-Ichi Ishikawa, Issaku Kanamori, and Hideo Matsufuru. *Multigrid Solver on Fugaku.* 2021. arXiv: `2112.00501 [hep-lat]`.

[132] M. Lüscher. *Lattice QCD and the Schwarz alternating procedure.* JHEP 0305, 052 (2003); Comput. Phys. Commun. 165, (2005) 119.

[133] Alfredo Buttari et al. "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance While Achieving 64-Bit Accuracy". In: *ACM Trans. Math. Softw.* 34.4 (July 2008). ISSN: 0098-3500. DOI: `10.1145/1377596.1377597`.

[134] Youcef Saad. "A Flexible Inner-Outer Preconditioned GMRES Algorithm". In: *SIAM Journal on Scientific Computing* 14.2 (1993), pp. 461–469. DOI: `10.1137/0914028`. eprint: `https://doi.org/10.1137/0914028`.

[135] Judith A. Vogel. "Flexible BiCG and flexible Bi-CGSTAB for nonsymmetric linear systems". In: *Applied Mathematics and Computation* 188.1 (2007), pp. 226–233. ISSN: 0096-3003. DOI: `https://doi.org/10.1016/j.amc.2006.09.116`.

[136] Ken-Ichi Ishikawa et al. "102 PFLOPS Lattice QCD quark solver on Fugaku". In: (Sept. 2021). arXiv: `2109.10687 [hep-lat]`.