File I/O Cache Performance of Supercomputer Fugaku Using an Out-of-core Direct Numerical Simulation Code of Turbulence

Yuto Hatanaka[1], Yuki Yamane[2], Kenta Yamaguchi[2], Takashi Soga[3], Akihiro Musa[4,6], Takashi Ishihara[5][0000−0002−4520−6964], Atsuya Uno[7][0009−0002−8449−5196], Kazuhiko Komatsu[6][0000−0003−4463−8359], Hiroaki Kobayashi[8][0000−0002−3350−1413], and Mitsuo Yokokawa[1][0000−0003−3790−1243]

[1] Graduate School of System Informatics, Kobe University, Kobe, Japan
{hatanaka-m-yuto@stu,yokokawa@port}.kobe-u.ac.jp
[2] NEC Solution Innovators, Ltd., Koto-ku, Tokyo, Japan
{yamane.yuki,yamaguchi-zx}@nec.com
[3] Cybermedia Center, Osaka University, Osaka, Japan
soga.takashi.cmc@osaka-u.ac.jp
[4] NEC, Corp., Minato-ku, Tokyo, Japan
musa@tohoku.ac.jp
[5] Faculty of Environmental, Life, Natural Science and Technology, Okayama University, Okayama, Japan
takashi_ishihara@okayama-u.ac.jp
[6] Cyberscience Center, Tohoku University, Sendai, Japan
komatsu@tohoku.ac.jp
[7] National Research Institute for Earth Science and Disaster Resilience, Tsukuba, Japan
a.uno@bosai.go.jp
[8] Graduate School of Information Sciences, Tohoku University, Sendai, Japan
koba@tohoku.ac.jp

Abstract. Turbulent flows play important roles in many flow-related phenomena that appear in various fields. However, despite numerous studies on turbulence, the nature of turbulence has not yet been fully clarified. Direct numerical simulation (DNS) of incompressible homogeneous turbulence in a periodic box is currently a powerful method for studying turbulent flows. However, even modern world-class supercomputers do not have sufficient computational resources to carry out DNS at very high Reynolds number (Re). Memory capacity constraints are particularly severe. Therefore, we have developed an out-of-core DNS (ooc-DNS) code that uses storage to overcome memory limitations. The ooc-DNS code can reduce memory usage by up to a quarter and allows DNS at a higher Re, which would be impossible under normal usage due to memory limitations. When implementing the ooc-DNS code, however, it is crucial to accelerate file input/output (I/O) because the I/O time for storage accounts for a large percentage of the execution time. In this paper, we evaluate the I/O performance of the ooc-DNS code when using a file system called the Lightweight Layered I/O Accelerator of the supercomputer Fugaku. We also evaluate the impact of the I/O cache and its size on I/O performance and show that the I/O processing can be accelerated by using the cache and optimizing its size. Finally, by taking on I/O cache size when executing the ooc-DNS code with $8,192^3$ grid points, the I/O speed and overall execution speed are increased by 2.4 times and 1.9 times compared to that without the I/O cache.

Keywords: Direct numerical simulation · Turbulent flows · Out-of-core implementation · Fugaku · I/O cache.

1 Introduction

Turbulent flows are ubiquitous and play important roles in flow-associated phenomena that appear in various fields of science and technology. Despite numerous studies of turbulence, however, the nature of turbulence has not yet been fully clarified.

In turbulent flows, eddies of various spatial and temporal scales coexist. They non-linearly interact with each other to produce complex motions, so that a small difference in initial conditions can result in unpredictable different motions of individual eddies in turbulent flows. However, it is conceivable that statistically universal laws may exist in these seemingly completely unpredictable complex flows, independent of differences in boundary and initial conditions.

Direct numerical simulation (DNS) of turbulence in a periodic box is a suitable method by which to study the homogeneous isotropic equilibrium state of turbulence on small scales at sufficiently high Reynolds numbers. Direct numerical simulation of box turbulence is highly accurate under the simplest possible conditions and allows DNS of turbulence flows at higher Reynolds numbers. In fact, many DNSs have been conducted and have contributed to the development of turbulence theory [7, 8], beginning with Orzag's DNS in 1969 [12].

However, even modern world-class supercomputers do not have sufficient computational resources to carry out DNSs of box turbulence at high Re. Figure 1 shows the trend in supercomputer computational performance and memory capacity and that the computational performance has increased approximately 11,574 times in eighteen years, while the memory capacity has increased only approximately 509 times. This fact suggests that memory capacity constraints will eventually become more severe. For example, a double precision DNS with $32,768^3$ grid points using the developed code requires at least 5.6 PiB of memory. However, there is no supercomputer in Japan that has this amount of memory capacity. Therefore, we have developed an out-of-core DNS (ooc-DNS) code such that a DNS with a large number of grid points can be executed [9, 15]. The ooc-DNS code saves arrays containing variables to files on external storage devices. In fact, the code reduces memory usage by up to a quarter and allows DNS at higher Re, which would be impossible under normal usage due to memory limitations. When executing the ooc-DNS code on the supercomputer Fugaku, however, the file input/output (I/O) time accounts for a large percentage of the total execution time because the I/O speed is much slower than the computation speed.

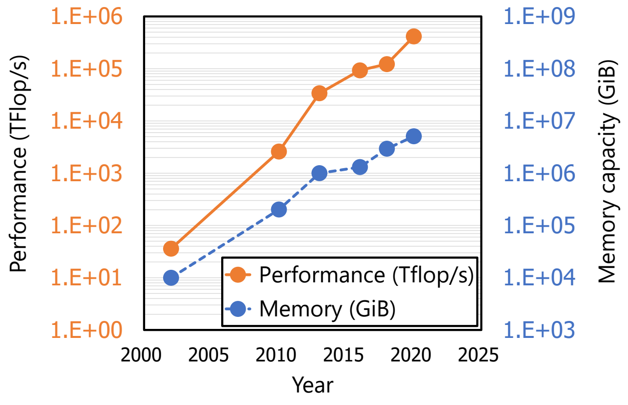


Fig. 1: Performance and memory capacity trends for computers listed in TOP500 [3]. The solid orange line represents the first-place performance in the TOP500. The blue dashed line represents the memory capacity of the computers that are ranked first in the TOP500 [2, 6, 10, 11, 13, 14].

The purpose of this paper is to accelerate I/O processing and reduce the I/O time when ooc-DNS is executed on Fugaku, and we focus on the I/O cache in compute node (CN)-cache provided by a file system called Lightweight Layered I/O Accelerator (LLIO). Specifically, we evaluate the impact of the I/O cache size on I/O performance. We also evaluate the I/O performance of the ooc-DNS code when executed on Fugaku.

In the remainder of this paper, Section 2 describes the ooc-DNS code. Section 3 describes the architecture of Fugaku. In Section 4.1, we evaluate the performance of Fugaku's CN-cache using the IOR benchmark program [1]. In Section 4.2, we evaluate the performance of the CN-cache by using a benchmark program that simulates the behavior of ooc-DNS. In Section 4.3, we evaluate the performance of the CN-cache using the ooc-DNS code with $2,048^3$ and $4,096^3$ grid points. In Section 4.4, we implement the ooc-DNS code with $8,192^3$ grid points and confirm that optimizing the CN-cache size increases the I/O speed and overall execution speed. Conclusions are given in Section 5.

2 Out-of-core Direct Numerical Simulation Code

2.1 Direct Numerical Simulation Code Implementation

We consider a cube with side length 2π and periodic boundary conditions as a computational domain. Within this computational domain, we consider homogeneous isotropic turbulence according to the Navier-Stokes equations under the incompressible condition with unit density, as follows

$$\nabla \cdot \boldsymbol{u} = 0, \tag{1}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\nabla p + \nu \nabla^2 \boldsymbol{u} + \boldsymbol{F}, \tag{2}$$

where $\boldsymbol{u} = (u_1, u_2, u_3)$, $\boldsymbol{F} = (F_1, F_2, F_3)$, p, and ν are the velocity, external force, pressure, and kinematic viscosity, respectively.

Discretizing Eqs. (1) and (2) using the Fourier spectral method on the discretized grid points that divide a cube into N equal parts in each direction leads to ordinary differential equations for the Fourier coefficient $\hat{\boldsymbol{u}}_{\boldsymbol{l}}$ of the velocities in real space, which are represented as

$$\frac{\mathrm{d}\hat{\boldsymbol{u}}_{\boldsymbol{l}}}{\mathrm{d}t} + \nu ||\boldsymbol{l}||_2^2 \hat{\boldsymbol{u}}_{\boldsymbol{l}} = -(\widehat{(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}})_{\boldsymbol{l}} + \boldsymbol{l} \cdot \frac{\boldsymbol{l} \cdot (\widehat{(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}})_{\boldsymbol{l}}}{||\boldsymbol{l}||_2^2} \qquad (1 \le l_1, l_2, l_3 \le N). \tag{3}$$

Here,

$$(\widehat{(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}})_{\boldsymbol{l}} = il_1 \sum_{\boldsymbol{l}=\boldsymbol{k}+\boldsymbol{k'}} \hat{u}_{\boldsymbol{k}} \hat{\boldsymbol{u}}_{\boldsymbol{k'}} + il_2 \sum_{\boldsymbol{l}=\boldsymbol{k}+\boldsymbol{k'}} \hat{v}_{\boldsymbol{k}} \hat{\boldsymbol{u}}_{\boldsymbol{k'}} + il_3 \sum_{\boldsymbol{l}=\boldsymbol{k}+\boldsymbol{k'}} \hat{w}_{\boldsymbol{k}} \hat{\boldsymbol{u}}_{\boldsymbol{k'}}. \tag{4}$$

where $\boldsymbol{l} = (l_1, l_2, l_3)$, $\boldsymbol{k} = (k_1, k_2, k_3)$, and $\boldsymbol{k'} = (k'_1, k'_2, k'_3)$ are the wave numbers in Fourier space, and the hat symbol denotes the Fourier coefficient. The ordinary differential equations are time evolved using the four-stage fourth-order Runge-Kutta-Gill (RKG) method.

Equations (4) are computed using a transform method based on the three-dimensional fast Fourier transform (3D-FFT). Aliasing errors introduced by the transform method are completely eliminated by the phase shift method and cutting modes larger than $\frac{\sqrt{2}N}{3}$ [5].

The original parallel code was developed for two-dimensional domain decomposition with Message Passing Interface (MPI) for data distribution, and the 3D-FFT parallelized by pencil decomposition is used in the implementation. Figure 2 illustrates how the computational domain is divided into pencils.

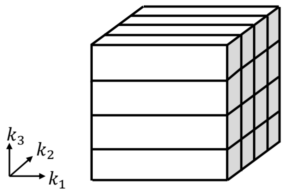


Fig. 2: Pencil domain decomposition in spectral space

2.2 Out-of-core Implementation Concept

When carrying out DNS of box turbulence using our code by the spectral and RKG methods, a total of 18 variables are required, including the velocity field in each direction in spectral space, intermediate variables in the RKG method, the velocity field in real space, and two variables for the 3D-FFT.

The ooc-DNS code divides each array containing the variables into several subarrays and stores them in separate files. In addition, the code holds only as much memory as the size of one file for each subarray. Figure 3 illustrates how arrays to be stored in storage are divided within a process. When assigning or referencing variables stored in the files, they are processed one file at a time using arrays in memory that are prepared for the size of one file. Figure 4 describes how to change the original code to the ooc-DNS code using a Fortran-like description. The rank of the array variable is 3, and the last rank is partitioned into the number of files at declaration. Then, a loop is added to repeatedly read each portion of the variables, perform calculations, and write updated values to the file.

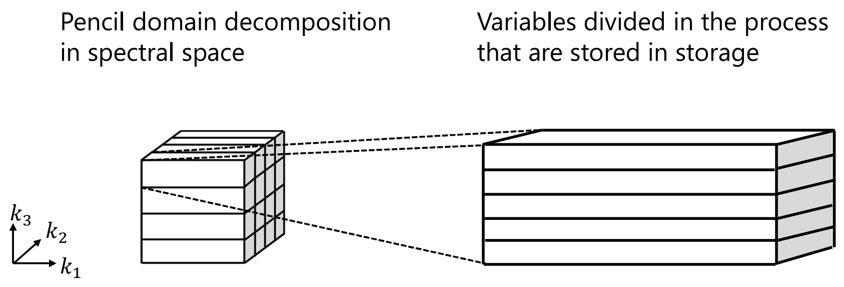
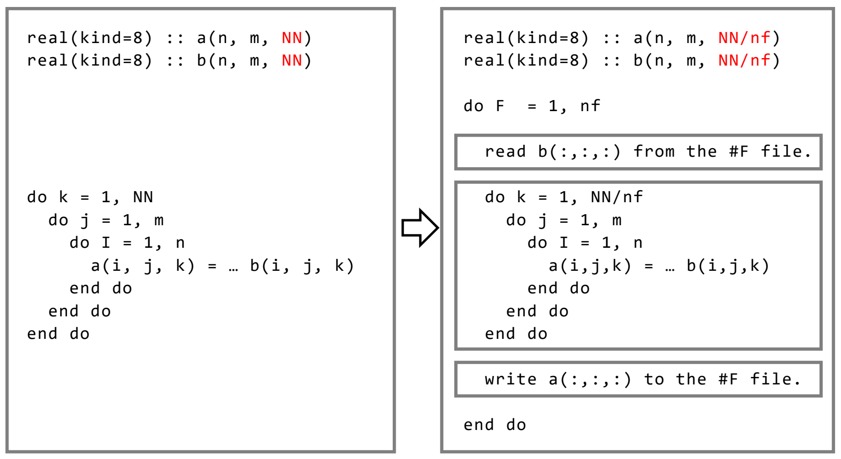

Pencil domain decomposition in spectral space
Variables divided in the process that are stored in storage
k_3
k_2
k_1

Fig. 3: Illustration of array division within the process



```
real(kind=8) :: a(n, m, NN)
real(kind=8) :: b(n, m, NN)




do k = 1, NN
  do j = 1, m
    do I = 1, n
      a(i, j, k) = … b(i, j, k)
    end do
  end do
end do
```

```
real(kind=8) :: a(n, m, NN/nf)
real(kind=8) :: b(n, m, NN/nf)

do F  = 1, nf

  read b(:,:,:) from the #F file.

  do k = 1, NN/nf
    do j = 1, m
      do I = 1, n
        a(i,j,k) = … b(i,j,k)
      end do
    end do
  end do

  write a(:,:,:) to the #F file.

end do
```

Fig. 4: Pseudo kernel of the out-of-core direct numerical simulation (ooc-DNS) code

This method processes data that is too large to fit into the physical memory of a computer and allows for easy configuration of which of the 16 variables will be stored in storage. The two variables for the 3D-FFT are all placed in memory as in the original code because the 3D-FFT is performed more frequently.

Although the additional file I/O time increases the computation time, this implementation reduces the memory required to execute the code by up to a quarter and allows for the execution of DNSs with a size that was not possible by the original code.

3 Fugaku Architecture

The Fugaku is installed at RIKEN Center for Computational Science (R-CCS) in Kobe, Japan [11]. The system is built on the A64FX ARM v8.2-A, which uses scalable vector extension instructions with a 512-bit implementation. The A64FX processor is a many-core ARM CPU with 48 compute cores and two or four assistant cores used by the operating system. The memory capacity of one node is 32 GiB, of which approximately 5 GiB is reserved for the system. Table 1 provides a further hardware breakdown. There are a total of 158,976 compute nodes, with one compute node for every 16 compute nodes serving as both a storage I/O (SIO) node and a compute node. The SIO node is connected to a first-layer storage and performs file I/O for this storage (Fig. 5). The group of 16 compute nodes is referred to as the SIO group. Each SIO group has a non-volatile memory express solid state drive (NVMe SSD) with a capacity of approximately 1.6 TiB.

Table 1: Specifications of supercomputer Fugaku

Total peak performance		488 PFlops
Total memory		4.85 PiB
Number of nodes		158,976
node	Interconnect	Tofu Interconnect D
	CPU	FUJITSU Processor A64FX
	Performance	3.072 TFlops
	Number of cores	48
	Memory	32 GiB
	Memory bandwidth	1,024 GB/s
	L1D/core cache	64 KiB, 4 way 256 GB/s (load), 128 GB/s (store)
	L2/CMG cache	8 MiB, 16 way
	L2/node cache	4 TB/s (load), 2 TB/s (store)
	L2/core cache	128 GB/s (load), 64 GB/s (store)
	I/O	PCIe Gen3×16

Figure 6 shows that the storage system consists of three primary layers. The first-layer storage consists of NVMe SSDs controlled by LLIO. The second-

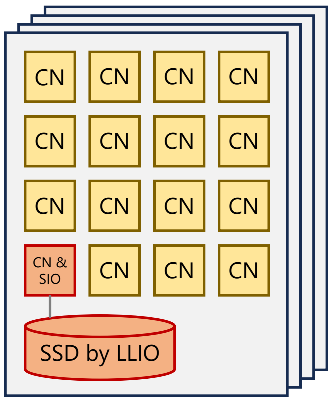


Fig. 5: Lightweight Layered I/O Accelerator (LLIO) configuration diagram. Here, CN indicates a compute node and CN&SIO indicates a compute node with a storage I/O function.

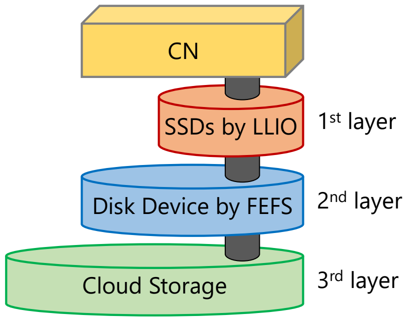


Fig. 6: Layered storage in supercomputer Fugaku

layer storage consists of multiple hard disk drives controlled by a Lustre-based global file system called the Fujitsu Exabyte File System (FEFS). The total capacities of the first- and second-layer storages are approximately 15.9 PiB and 150 PiB, respectively. The third-layer storage consists of commercial cloud storage services.

The LLIO provides three storage areas in the first-layer storage: a file cache area for the second-layer storage, a shared temporary area for compute nodes assigned a job, and a local temporary area for each compute node. The local temporary area has the largest bandwidth of these three areas [4].

The LLIO also provides an I/O cache function in compute nodes in order to accelerate the I/O from/to the first-layer storage. This function uses a portion of the compute node memory as an I/O cache, called the CN-cache. The CN-cache area is allocated in the size specified by the user in the memory area, and the size on one compute node that can be specified ranges from 4 MiB to 32 GiB. When this size is not specified explicitly, it is the default size of 128 MiB. Therefore, if there is remaining memory capacity out of the capacity that is used by a user program, then the CN-cache size can be increased accordingly. For example, if a program that uses 25 GiB of memory per compute node, considering that the system uses approximately 5 GiB of memory as previously stated, the remaining 2 GiB can be allocated as the CN-cache. Figure 7 illustrates the memory usage for one compute node in this example.

The CN-cache can be enabled and used without any modifications to the code or compiling the code again against specific libraries that overload read and write posix calls. The CN-cache size can just be set as an environment variable when executing the program.

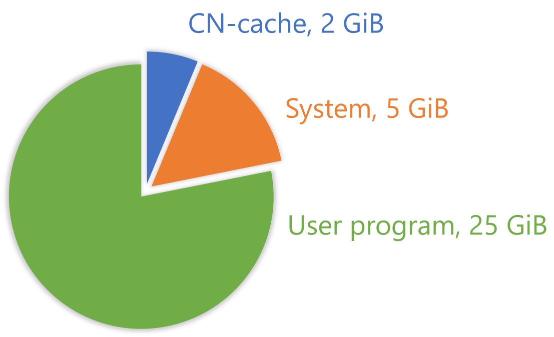


Fig. 7: Example of memory usage for one compute node on supercomputer Fugaku

The CN-cache function allows data to be cached during read and write operations. Hereinafter, caching during read operations is referred to as "read cache" and caching during write operations is referred to as "write cache".

The purpose of the read cache is to store data that has been read from the first-layer storage into the CN-cache. This results in the acceleration of the input processing. The decision to use the read cache can be made by the user through a specified parameter. Similarly, the write cache is used to store data that is to be written to the first-layer storage in the CN-cache, resulting in the acceleration of the output processing. A threshold value can be specified for the write cache to determine whether it should be used. The write cache is used when the data size to be written is less than or equal to the specified threshold. It is important to note that the read cache is always enabled in this paper, and the write cache threshold is consistently set to the same size as the CN-cache size, unless otherwise stated.

In this paper, we use the first-layer storage for storing temporary files, measure the I/O time when the size of the CN-cache is changed, and evaluate its performance.

4 Evaluation of Compute Node (CN)-cache Performance

In this section, we evaluate the CN-cache performance from three perspectives. First, we evaluate the performance of the CN-cache using IOR [1], a benchmark program for measuring parallel file system performance. Next, we evaluate the performance using a simple program similar to the I/O kernel of the ooc-DNS code. Finally, we evaluate the performance using the ooc-DNS code.

The CN-cache performance is evaluated by varying the CN-cache size. When the CN-cache performance of 0 MiB is measured, the read cache is set to disabled and the write cache threshold is to 0 MiB.

4.1 Performance Evaluation of the I/O Cache with IOR

There are more than 30 options that can be specified when executing an IOR program. Considering the I/O kernel behavior of the ooc-DNS code, we specified those options as shown in Table 2. Just as the ooc-DNS code divides a 512 MiB variable into 32 segments, the IOR code divides a 512 MiB block-sized file into 32 segments. However, the parameter "`fsyncPerWrite`" is set to 1 when the CN-cache size is 0 MiB. In contrast, "`fsyncPerWrite`" is set to 0 when the CN-cache size is not 0 MiB.

Table 2: Description of options and contents specified in IOR

Parameter	Value	Description
`repetition`	10	number of repetitions of test
`fsync`	1	performs fsync upon POSIX file close
`SegmentCount`	32	number of segments
`intraTestBarriers`	1	uses barriers between open, write/read, and close
`blockSize`	512 MiB	-
`transferSize`	16 MiB	-
`fsyncPerWrite`	0 or 1	performs fsync after each POSIX write

The IOR program was executed using 4,608 MPI processes on 1,152 compute nodes, with four processes per node. The sizes of the CN-cache were set to 0, 4, 16, 32, 64, 128, 192, 256, and 1,024 MiB. Figure 8 shows the bandwidth of first-layer storage in one SIO group, calculated based on the results of the IOR execution. The read and write bandwidths are depicted in orange and blue, respectively. The read and write bandwidths for the CN-cache size of 0 are depicted by the symbol "■" in the same figure.

It is found that the actual read and write bandwidths for the first-layer storage in one SIO group are approximately 4,500 MiB/s and 1,900 MiB/s, respectively. The values are nearly identical to those measured by Akimoto et al. [4]

The highest write bandwidth with the CN-cache size of 1,024 MiB is obtained because the data is transferred together in larger sizes. Since the CN-cache of 4 MiB is not used for transferring data of 16 MiB, the bandwidths with CN-cache sizes of 0 MiB and 4 MiB are approximately the same.

The read and write bandwidths are lowest when the CN-cache size is between 16 MiB and 32 MiB because the CN-cache size is too small to transfer the data continuously. However, the bandwidths gradually increase as the CN-cache size increases. Processing takes longer due to the lack of free CN-cache area, and increasing the CN-cache size improves the bandwidths.

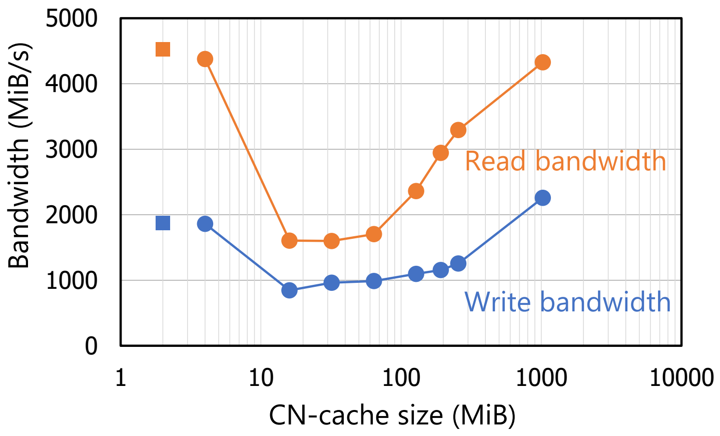


Fig. 8: Measurements of bandwidth per the storage I/O (SIO) group, the first-layer storage, using IOR

4.2 Performance Evaluation of the I/O Cache with a Simple Program Similar to the Ooc-DNS Code

In the ooc-DNS code, if variables in files are referenced and assigned, then the sequence of reading values from the file, calculating them, and writing them to the file is repeated until all subarrays are calculated. The benchmark program shown in Fig. 9 was generated to perform similar operations as the ooc-DNS code with three arrays: `a`, `b`, and `c`. Each array is divided into 32 subarrays, each of which is stored into separate files. The size of arrays `a`, `b`, and `c` in each process is 512 MiB, so that the size of the subarrays is 16 MiB. The input and output times were measured with barrier synchronization after each operation. The variable `F` in Fig. 9 is the number of subarrays. In the same way, the benchmark program was also generated with six arrays: `a`, `b`, `c`, `d`, `e`, and `f`.

These programs were executed using 192 MPI processes on 48 compute nodes, with four processes per node. The sizes of the CN-cache size were set to 0, 16, 32, 64, 96, 128, 192, 256, and 512 MiB. Figures 10 and 11 show the execution results when using three arrays and six arrays, respectively. For each configuration, we repeat the code execution 11 times and the figures show the average time of the last 10 iterations. The read and write times are depicted in orange and blue, respectively. The times for the CN-cache size of 0 MiB are depicted in the same figure using the symbol "■."

Figure 10 shows that when the CN-cache size is 0 MiB, the I/O time is longest and there is not much difference in the read time and write time. When the CN-cache size is 16 MiB or larger, the write time is approximately twice as long as the read time. The write time is shortest when the CN-cache size is 128

```
do F = 1, 32
    call read_a_file(F)
    call read_b_file(F)
    call read_c_file(F)

    call calculation(a,b,c)

    call write_a_file(F)
    call write_b_file(F)
    call write_c_file(F)
end do
```

```
subroutine calculation(a,b,c)
    a = a*04 + a*0.3 + a*0.2
    b = b*04 + b*0.3 + b*0.2
    c = c*04 + c*0.3 + c*0.2
end subroutine calculation
```

Fig. 9: Simple program created to evaluate CN-cache performance

MiB. On the other hand, the read time is shortest when the CN-cache size is 96 MiB.

Figure 11 shows that the I/O time is longest when the CN-cache size is 0 MiB and that there is no difference in the read time and write time when the CN-cache size is 0 MiB. The write time is shortest when the CN-cache size is 128 MiB, and increases monotonically when the CN-cache size is 128 MiB or larger. On the other hand, the read times for the CN-cache size of 128 MiB and 192 MiB are approximately 2 and 1.8 times longer than for that of 96 MiB, respectively. However, no such significant difference in read time was observed when only reading and calculating were repeated without writing. Therefore, these spikes for 128 MiB and 192 MiB are caused by the sequence of reading and writing instructions. More specifically, when the CN-cache size is between 128 MiB and 192 MiB and the transfer data is 16 MiB, consecutive reads and writes will degrade the speed of reading data.

It is found that I/O time was reduced by using the CN-cache for both cases of three and six arrays. However, it is also found that there are spikes in read time, as shown in Fig. 11, and that increasing the CN-cache size results in longer I/O time in some cases. Therefore, optimizing the CN-cache size is important to further reduce I/O time.

4.3 Performance Evaluation of I/O Cache with the Ooc-DNS Code

We carried out the ooc-DNS code with two problem sizes, $N^3 = 2,048^3$ and $4,096^3$.

The number of nodes used in the execution with $N^3 = 2,048^3$ and $4,096^3$ are 32 and 1,024, respectively, with four MPI processes allocated per node. The DNSs with $N^3 = 2,048^3$ and $4,096^3$, however, require approximately 1,281 GiB and 10,256 GiB memory capacities, respectively, and it is necessary to store variables to files in storage considering that one node has only a 32-GiB memory capacity. We changed the number of variables to be stored in files on each problem size

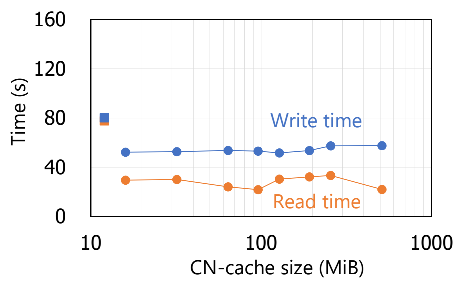


Fig. 10: Measured performance of CN-cache in simple program similar to ooc-DNS code with three arrays

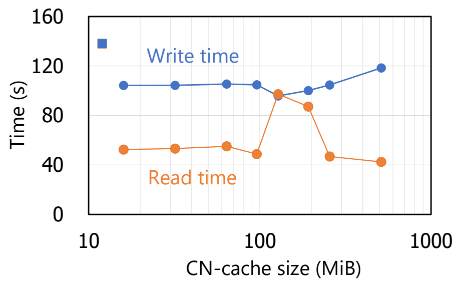


Fig. 11: Measured performance of CN-cache in simple program similar to ooc-DNS code with six arrays

so that as few variables as possible are stored in files. In fact, 11 or 12 out of 16 variables are divided into 32 subarrays for the cases of $N^3 = 2,048^3$ and $4,096^3$, respectively. Then, the size of one file storing a subarray is 16 MiB, and the memory usage is approximately 0.52 times for N=2,048 and approximately 0.45 times for N=4,096. Barrier synchronization is performed at the end of the interval for which time measurements were taken, and the I/O time required to proceed a time step is measured. The numbers of read and write operations in the ooc-DNS code are 4,069 and 3,008, respectively, for $N^3 = 2,048^3$, for one time step. In contrast, the numbers of read and write operations in the ooc-DNS code are 4,704 and 3,648, respectively, for $N^3 = 4,096^3$.

The times for read and write operations are measured by varying the CN-cache size as 0, 4, 16, 32, 64, 96, 128, 256, and 512 MiB. The times for the read and write operations and their sums for $N^3 = 2,048^3$ and $N^3 = 4,096^3$ are plotted in Figs. 12 and 13, respectively. The times for read, write, and their sum are depicted in orange, blue, and gray, respectively. The times for the CN-cache size of 0 MiB are plotted by the "■" symbol.

The difference in I/O time between the DNS sizes of $2,048^3$ and $4,096^3$ is due to the different number of variables stored in first-layer storage, and, regardless of CN-cache size, the I/O total time is shorter for $N^3 = 2,048^3$.

Figure 12 shows that the write performance when the CN-cache size is 96 MiB or larger is worse than when the CN-cache size is 0 MiB. Fig. 13 shows that the write performance when the CN-cache size is 192 MiB or larger is similarly worse. These results indicate that the write performance is better with a small CN-cache size, despite the large number of write operations. The reasons for these are thought to be that the CN-cache improves the write speed, and that there is too much data in the CN-cache for the LLIO to manage.

Figures 12 and 13 also show that spikes are found for read performance when the CN-cache size is 128 MiB for the two cases. In Fig. 12, the read time at 128 MiB is approximately 1.5 times longer than that at 96 MiB. On the other hand, in Fig. 13, the read time at 128 MiB is approximately 1.8 times longer than that at 96 MiB. These behaviors are also confirmed in Fig. 11 in Section 4.2.

We found that the I/O total time for the CN-cache size of 0 MiB is the longest and that the CN-cache improved the I/O total speed and reduce I/O total time for practical applications.

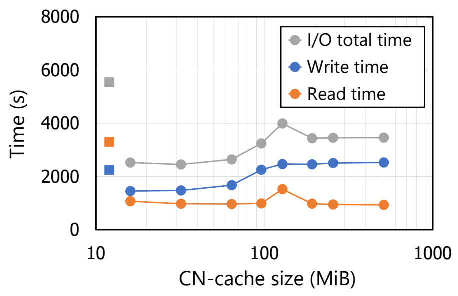


Fig. 12: Performance for $2,048^3$ grid points

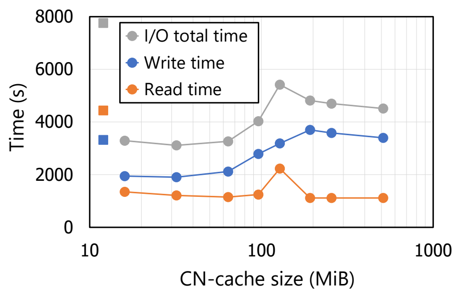


Fig. 13: Performance for $4,096^3$ grid points

4.4 Execution of Ooc-DNS Code with $8,192^3$ Grid Points

The DNS with $N^3 = 8,192^3$ was executed using 8,192 processes on 2,048 nodes with four processes per node. The DNS with $N^3 = 8,192^3$ requires a memory capacity of approximately 80-TiB, and it is necessary to store variables to files in storage. Twelve out of 16 variables are divided into 32 subarrays and sent to storage. Then, the size of one file storing a subarray is 16 MiB, and the memory usage is approximately 0.45 times as large as that without the out-of-core implementation. Barrier synchronization is performed at the end of the interval for which time measurements were taken, and the I/O time and the other time required to proceed a time step are measured. The numbers of read and write operations in the ooc-DNS code are 4,704 and 3,648, respectively, for one time step.

The times for reading, writing, and computation are measured at two CN-cache sizes of 0 and 32 MiB. These times are plotted in Fig. 14. The times for read, write, and other computation are depicted in orange, blue, black, respectively. It is found that there is little difference in computation time between the CN-caches of 0 MiB and 32 MiB. In addition, when the CN-cache size is set to 32 MiB, a two-fold speed up in overall execution time is achieved compared to the CN-cache size of 0 MiB.

5 Conclusions

Modern world-class supercomputers do not have sufficient memory capacity to carry out DNS at very high Reynolds number. Therefore, we implemented the

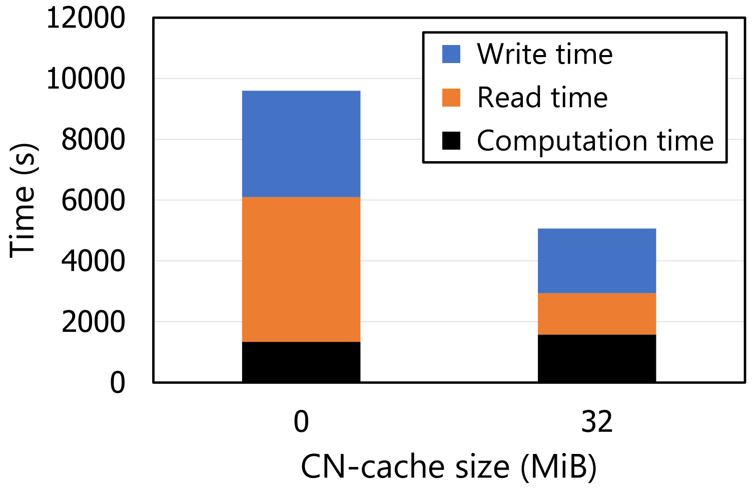


Fig. 14: Execution time for ooc-DNS with $N^3 = 8,192^3$

ooc-DNS code. When we ran the ooc-DNS code on Fugaku, the I/O time accounted for the majority of the execution time. In order to accelerate the I/O speed when executing the ooc-DNS code, we have focused on the CN-cache function for optimizing the I/O cache size on Fugaku and evaluated the CN-cache performance from three perspectives. First, the basic evaluation of the performance of the CN-cache was conducted using the IOR benchmark program. Next, we evaluated the performance with a simple program that is similar to the ooc-DNS code. The perfowermance was then evaluated with the ooc-DNS code. Finally, we optimized the CN-cache size in order to accelerate the ooc-DNS code. As a result, it was found that specifying a CN-cache size larger than 512 MiB does not reduce the I/O time in the execution of the ooc-DNS code due to performance degradation of the write cache. In addition, the read performance degrades drastically without explicit specification of the CN-cache size. By specifying an appropriate CN-cache size when executing the ooc-DNS code with $N^3 = 8,192^3$, the I/O speed and the overall execution speed was increased by 2.4 times and 1.9 times, respectively, as compared to the CN-cache size of 0 MiB.

Acknowledgments. This study used the computational resources of the Supercomputer Fugaku provided by the RIKEN Center for Computational Science through the HPCI System Research project (Project ID No. hp230143). This work was partially conducted and funded at Joint-Research Division of High-Performance Computing (NEC) of Cyberscience Center at Tohoku University.

References

1. Github: IOR. `https://github.com/hpc/ior`

2. System Overview (ES). `https://www.jamstec.go.jp/es/jp/es1/system/system.html`, last accessed 12 December 2023
3. Performance development. `https://www.top500.org/statistics/perfdevel/` (2023), last accessed 12 December 2023
4. Akimoto, H., Okamoto, T., Kagami, T., Seki, K., Sakai, K., Imade, H., et al.: File system and power management enhanced for supercomputer Fugaku. Fujitsu Technical Review (3), 2020–03 (2020)
5. Canuto, C., Hussaini, M.Y., Quarteroni, A., Zang, T.A.: Spectral Methods in Fluid Dynamics. Springer Berlin, Heidelberg (1988)
6. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., et al.: The Sunway TaihuLight supercomputer: system and applications. Science China Information Sciences **59**(7) (Jun 2016). https://doi.org/10.1007/s11432-016-5588-7
7. Ishihara, T., Kaneda, Y., Morishita, K., Yokokawa, M., Uno, A.: Second-order velocity structure functions in direct numerical simulations of turbulence with R_λ up to 2250. Phys. Rev. Fluids **5**, 104608 (Oct 2020). https://doi.org/10.1103/PhysRevFluids.5.104608
8. Kaneda, Y., Ishihara, T., Yokokawa, M., Itakura, K., Uno, A.: Energy dissipation rate and energy spectrum in high resolution direct numerical simulations of turbulence in a periodic box. Physics of Fluids **15**(2), L21–L24 (01 2003). https://doi.org/10.1063/1.1539855
9. Komatsu, K., Momose, S., Isobe, Y., Watanabe, O., Musa, A., Yokokawa, M., et al.: Performance evaluation of a vector supercomputer sx-aurora tsubasa. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 685–696 (2018). https://doi.org/10.1109/SC.2018.00057
10. Liao, X., Xiao, L., Yang, C., Lu, Y.: MilkyWay-2 supercomputer: system and application. Frontiers of Computer Science **8**(3), 345–356 (May 2014). https://doi.org/10.1007/s11704-014-3501-3
11. Mitsuhisa, S., Yutaka, I., Hirofumi, T., Yuetsu, K., Tetsuya, O., Miwako, T., et al.: Co-Design for A64FX Manycore Processor and "Fugaku". SC20: International Conference for High Performance Computing, Networking, Storage and Analysis p. 1 (11 2020). https://doi.org/10.1109/sc41405.2020.00051, `https://cir.nii.ac.jp/crid/1360013173149649280`
12. Orszag, S.A.: Numerical Methods for the Simulation of Turbulence. The Physics of Fluids **12**(12), II–250–II–257 (12 1969). https://doi.org/10.1063/1.1692445
13. Vazhkudai, S.S., De Supinski, B.R., Bland, A.S., Geist, A., Sexton, J., Kahle, J., et al.: The design, deployment, and evaluation of the CORAL pre-exascale systems. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 661–672. IEEE (2018)
14. Yang, X.J., Liao, X.K., Lu, K., Hu, Q.F., Song, J.Q., Su, J.S.: The TianHe-1A Supercomputer: Its Hardware and Software. Journal of Computer Science and Technology **26**(3), 344–351 (May 2011). https://doi.org/10.1007/s02011-011-1137-8
15. Yokokawa, M., Yamane, Y., Yamaguchi, K., Soga, T., Matsumoto, T., et al.: I/O Performance Evaluation of a Memory-Saving DNS Code on SX-Aurora TSUBASA. In: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 692–696 (2023). https://doi.org/10.1109/IPDPSW59300.2023.00117