Evaluation of Flash-based Out-of-core Stencil Computation Algorithms for SSD-Equipped Clusters

Hiroko Midorikawa, Hideyuki Tan
Department of Computer and Information Science
Seikei University, JST CREST
Tokyo, Japan
midori@st.seikei.ac.jp

Abstract— **This paper proposes a new scheme for solving data size requirements for a large-scale stencil computation, which are greater than the total size of the main memories of nodes in a cluster. It utilizes distributed flash SSDs over cluster nodes as an extension to the main memory with a locality-aware algorithm. Three algorithms with a different hierarchical blocking scheme for three memory tiers, namely, flash SSD, DRAM, and cache, are proposed, and they are evaluated in different platforms and flash devices. They utilize not only highly parallel asynchronous input/output in flash SSDs, but also appropriate blocking parameters by using an auto-tuning system named Blk-Tune. They also overcome the performance degradation caused by the non-uniform memory architecture (NUMA). The optimized algorithms for single nodes are extended for multi-nodes and evaluated in a cluster with traditional SATA SSDs, as well as with state-of-the-art flash devices, such as low-power and cost-effective M.2 NVMe flash SSDs. With the use of our scheme and distributed flash devices in a cluster, large-scale stencil problems can be solved with a limited number of nodes and a moderate size of main memories.**

Keywords—Non-volatile memory; flash memory; memory hierarchy; temporal blocking; stencil; out-of-core; asynchronous I/O; access locality; NUMA; auto-tuning; SSD-equipped cluster

I. INTRODUCTION

Stencil computation is one of the most important and popular computation kernels in various scientific and engineering simulations. It performs iterative calculations on a limited data set, typically the nearest neighbor data. It sweeps the entire data, e.g., a three-dimensional (3D) physical domain data space, and updates them at each time step. Stencil computation is considered as a memory-bound type of computation, and it has been studied in numerous works with various types of techniques to speed up its execution. One typical technique is increasing data access locality, which typically causes higher cache hit, by introducing blocking techniques in spatial and temporal spaces in stencil computations [7-10].

On the other hand, scientific computation often requires significant amounts of memory for tackling large-scale problems and/or for higher resolution data analysis. One of the common solutions used to satisfy this requirement is aggregation of distributed memories over cluster nodes. This is typically accomplished by increasing the amount of DRAM per node and the number of nodes in a cluster, which eventually increases system cost and energy consumption. Moreover, there is a limit on how much DRAM can be increased in main memory because of the number of memory slots on server boards and power consumption constraints and other resource limitations. Today, various types of new memory devices including non-volatile memories, such as NVDIMM, 3D-Xpoint and Z-NAND [6], are being actively investigated. Nevertheless, NAND-flash memory is the most ubiquitous to end users at present and it is still determined as a cost-effective, power-efficient, and large-capacity type of memory behind the DRAM layer in a deep memory hierarchy. The recently developed 3D-structured flash has been improving its performance and capacity, but its access latency is still much longer, a thousand times slower, than DRAM access latency. The latency gap between DRAM and flash is much larger than that between cache and DRAM.

In this paper, three stencil algorithms using highly parallel asynchronous I/O (AIO) are proposed to overcome the latency divide between flash and DRAM. They use different hierarchical blocking schemes for flash, DRAM, and cache. The paper also investigates their performances and tradeoffs in detail, which are influenced by various factors, such as the non-uniform memory architecture (NUMA), I/O bandwidth of flash devices, amount of flash I/O, and redundant calculations.

To achieve the maximum performance, all algorithms use appropriate spatial and temporal blocking parameters chosen by a blocking-parameter auto-tuning system named Blk-Tune [3]. Blk-Tune incorporates an automatic platform hardware information retrieval and selects the best blocking parameters for a given problem setting by using this hardware information. It minimizes the amount of data transferred between the flash device and the DRAM, which is a dominant factor affecting the performance of out-of-core algorithms using flash.

The use of explicit I/O operations in a flash device together with auto-tuning allows users to easily maximize the performance for particular platforms and application settings. The algorithms also overcome the performance degradation caused by the NUMA. The optimized algorithms for single nodes are extended for multi-nodes and evaluated in a cluster with traditional SATA SSDs, as well as with state-of-the-art flash devices, such as low-power and cost-effective M.2 NVMe flash SSDs. With the use of our scheme and distributed flash devices in a cluster, large-scale stencil problems can be solved with a limited number of nodes and a moderate size of main memories. It is also a promising scheme in gaining higher performance with new memory devices, such as Z-SSD [6].

II. RELATED WORKS

Improving the performance of stencil computations by blocking techniques has been studied for a long time [7-10]. The

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.135

temporal blocking technique is not a new idea [7, 8, 10], but it has been mainly applied to cache and main memory tiers [9, 11, 12]. There are some studies that applied it to the host memory and the GPU memory [13], and to a local node and remote nodes in a cluster [12] to explore temporal locality. However, there has been no study applying the temporal blocking technique to flash memories as a main memory extension because of their long access latency. Temporal blocking was first applied to a flash and main memory tier for an out-of-core algorithm in our earlier work [1], where flash was used as a swap device under the revised swap kernel *fast-swap (OpenNVM)* [23, 24] in Linux. Moreover, we evaluated three methods, namely, 1) file mmap, 2) swap device, and 3) asynchronous I/O (AIO), that use a flash device for out-of-core stencil computations [2]. The results showed that the explicit way of using highly parallel AIO accompanied with an automatic optimal blocking size selection, which reduced the I/O traffic in flash to a minimum, was the most effective in achieving high performance [3]. The effectiveness of AIO is due to the advances in the latest I/O subsystem in Linux [25], which makes it possible to issue 64K or more I/O operations simultaneously using multiple cores in an asynchronous fashion (AIO).

Some studies on temporal blocking were conducted for fine performance tuning of the cache, the NUMA, and multicore CPUs in specific architectures [11, 12]. Others were conducted on more general performance models and/or auto-tuning mechanisms [14, 15]. Some of them focused on compile-based loop transformations and frameworks to increase access locality with optimized blocking parameters gained from theoretical analysis and auto-tuning [16-22]. Generally, more theoretical and static-based analysis is not sufficiently effective for actual usage where problem parameters and platforms are often changed or different. On the other hand, auto-tuning systems that require many preliminary executions to gather information for an actual execution have limited availability. They are not only time consuming but also unavailable for one-time executions. The automatic blocking parameter selection in Blk-Tune [3] overcomes those problems.

III. Hierarchical Temporal blocking Stencil Algorithms with Asynchronous Input/Output

The three algorithms proposed here employ temporal blocking schemes designed for DRAM and flash memory layers. Our algorithms introduce a hierarchical blocking scheme corresponding to the memory hierarchy shown in Fig. 1. Problem domain data are stored in *buffer arrays* in flash. *Block arrays* in DRAM and *iblock arrays* in cache are used for blocking schemes to extract data access locality. For the first level, a flash and DRAM tier, temporal blocking is introduced and highly parallel AIO is used to access the *buffer arrays* in flash. In the second level, a DRAM and cache tier, spatial blocking or temporal blocking is introduced.

Typical temporal blocking algorithms are categorized into two types: one with redundant calculations and the other without them. We developed both types of algorithms for flash-DRAM layers [1, 3].

A. Algorithm 1: with redundant calculations

A temporal blocking algorithm with redundant calculations, named algorithm 1, is shown in Figs. 2 and 3. In this paper, *bt* is the temporal blocking size and (*bx, by, bz*) is the spatial blocking size. *h* is a size parameter in the stencil computation kernel, e.g., *h*=1 for a typical seven-point stencil for a 3D data domain. In algorithm 1, one block is read from a domain source (src) *buffer array* in flash and an updated destination (dst) *block* in DRAM is written back to a dst *buffer array* in flash. The calculation area in the *block arrays* in DRAM shrinks according to the current update step progress, as shown in Fig. 3. The final result area updated *bt* times is smaller than the area initially read from flash. This difference in area corresponds to the amount of redundant calculations.

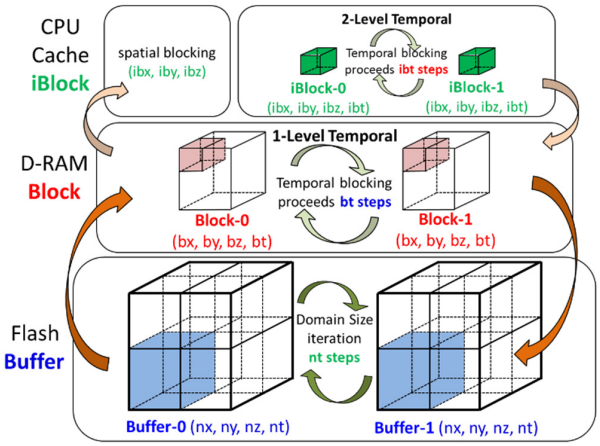

Fig.1 Three data arrays in memory layers: *buffer arrays* in flash SSD, *block arrays* in DRAM, and *iblock arrays* in cache for locality extraction.

B. Algorithm 2: without redundant calculations

A temporal blocking algorithm without redundant calculations, named algorithm 2, is shown in Figs. 4 and 5. The calculation areas in the block arrays in DRAM are shifted according to the current update step progress, as shown in Fig. 5. Unlike algorithm 1, algorithm 2 reads two block arrays from both src and dst *buffer arrays* in flash, and updated results in both src and dst *block arrays* are written back to flash after *bt*-step updates. It is necessary because each block array still includes an uncompleted calculation area, which is used and completed in the next neighbor block update procedures. This is an essential mechanism to eliminate the redundant calculations performed in algorithm 1. Thus, algorithm 2 decreases the amount of calculations, but it increases the amount of read/write data from/to the flash device.

In this paper, the following three algorithms are proposed for flash-based out-of-core stencil computations.

AL1. *One-level redundant:* applies temporal blocking with redundant calculations, algorithm 1, to a DRAM-flash tier and a spatial blocking to a DRAM-cache tier.

AL2. *Two-level redundant:* applies temporal blocking with redundant calculations, algorithm 1, to both a DRAM-flash tier and a DRAM-cache tier.

AL3. *One-level non-redundant*: applies temporal blocking without redundant calculations, algorithm 2, to a DRAM-flash tier and spatial blocking to a DRAM-cache tier.

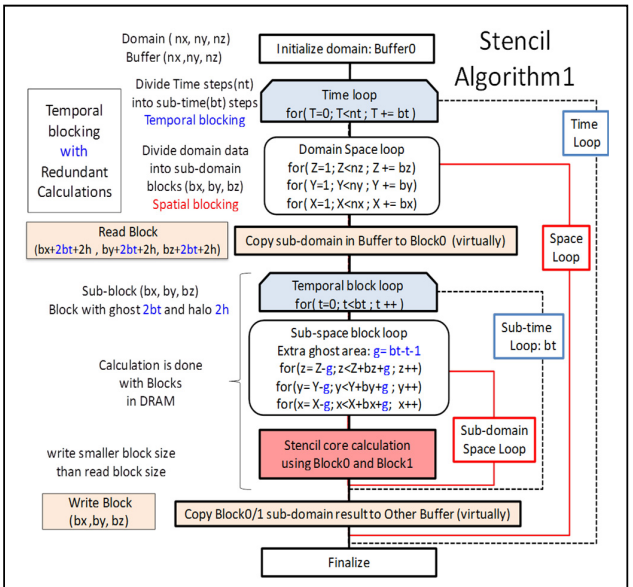

Fig. 2 Temporal blocking algorithm 1 with redundant calculation: pseudocodes for a 3D domain.

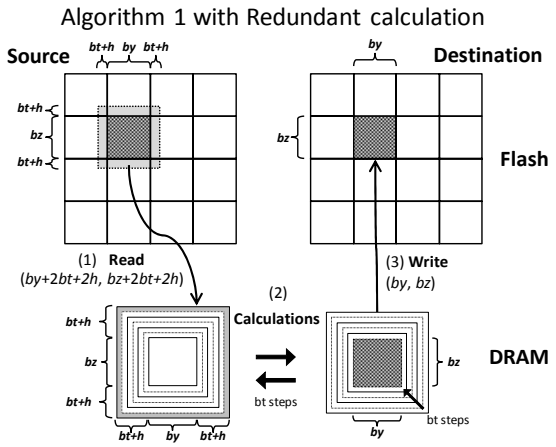

Fig. 3 Calculation of *bt* steps in block arrays in DRAM and read/write from/to domain arrays in the flash device in algorithm 1.

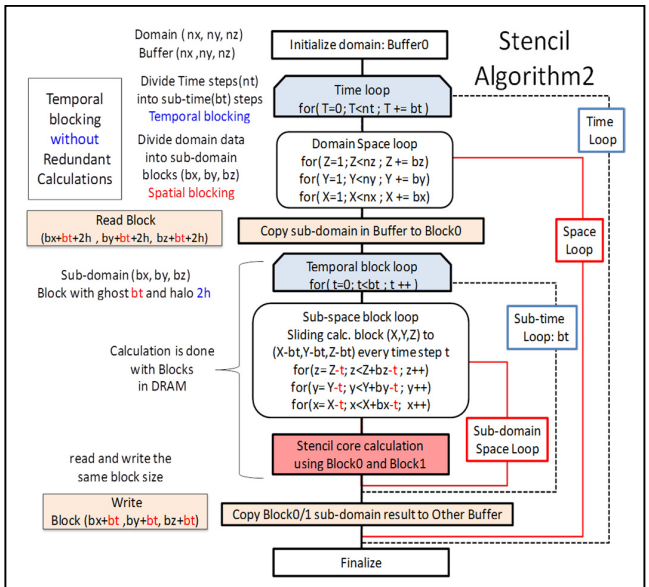

Fig. 4 Temporal blocking algorithm 2 without redundant calculation: pseudocodes for a 3D domain.

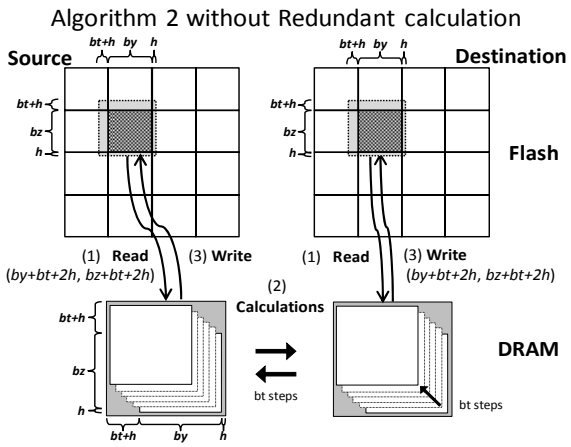

Fig. 5 Calculation of *bt* steps in block arrays in DRAM and read/write from/to domain arrays in the flash device in algorithm 2.

IV. Extended Hierarchical Temporal Blocking Stencil Algorithms for a Cluster

Three algorithms for single nodes with a local flash SSD are extended to the multi-node version with distributed flash SSDs by introducing another temporal blocking scheme among multiple nodes in a cluster. We believe that this is the first study on hierarchical temporal blocking computations utilizing distributed flash SSDs over a cluster that tackles large-size data requirements beyond the total amount of DRAMs in cluster nodes. An overview of the hierarchical temporal blocking algorithm for a cluster is shown in Fig. 6. The *domain buffer arrays* at the bottom of Fig. 6 correspond to problem data domain arrays, which are actually distributed over multiple flash SSDs in a cluster. Divided sub-domain data are stored in a local flash SSD as *sub-buffer arrays*, as shown in Fig. 6. Problem data are distributed initially when a stencil program starts, and result data are collected when the program finishes. The size of *sub-buffer arrays* is larger than the size of the main memory in a local node, but is smaller than the capacity of the local flash SSD. In the current version, a two-dimensional cluster node configuration is assumed and users specify the node configuration as a mesh size, e.g., 1 × 1, 1 × 2, or 2 × 2. When the problem domain size is (*Nx, Ny, Nz*) in a three-dimensional grid and the cluster-node configuration is specified as a mesh (*Py* × *Pz*), the *sub-buffer array* size becomes ($Nx + 2 * h$, $Ny/Py + 2 * (bt + h)$, $Nz / Pz + 2 * (bt + h)$). The sleeve area (halo

area) calculations in the *sub-buffer-array* correspond to the redundant calculations in algorithm 1 for multiple nodes.

In each *bt*-step update in a local node, the sleeve area data are read from the *sub-buffer array* in the local flash SSD and then exchanged between neighboring nodes in both the *z*-axis and the *y*-axis, as shown in Fig. 7. After new sleeve area data from the neighboring nodes are received, the data are written back to the *sub-buffer array* in the local flash SSD. In this experiment, the temporal blocking size *bt* for multi-nodes was set to the same value as the temporal blocking size used in DRAM and local flash layers.

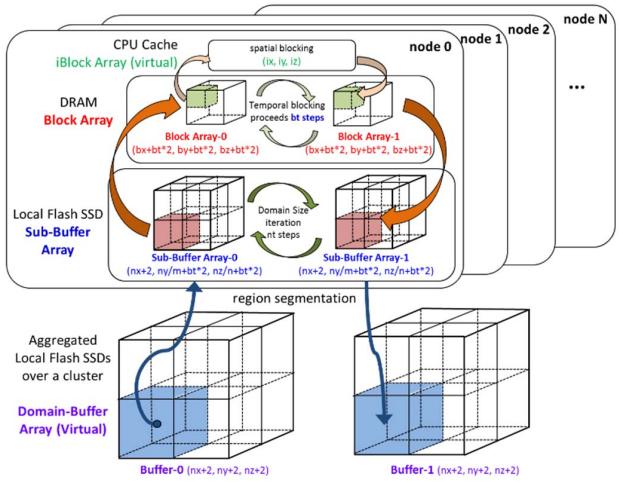

Fig. 6 Vertical memory layers in a local node and a horizontal domain-data distribution in a hierarchical temporal blocking computation for a cluster.

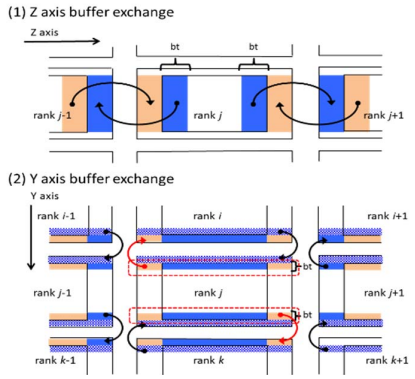

Fig. 7 Data exchange scheme in a hierarchical temporal blocking computation for a cluster.

V. Performance Tuning of the Algorithms

A. Block array memory layout and NUMA-aware tuning

Several memory layouts of *block arrays* for highly parallel AIO in flash SSDs were investigated in Ref. [2]. The efficient layout of block arrays with paddings was implemented with a z-dimensional pointer array and multiple xy-planes pointed to by a pointer in the z-dimensional array, as shown in Fig. 8. The start address and the size of each xy-plane were aligned on the device block size boundary of the flash SSD, which was 4KB in our case. Asynchronous I/O parameters, such as the start address, offset, and size, must be aligned on the device block size boundary. Each *xy*-plane is a single I/O unit of AIOs by multiple threads in parallel. The flash device is opened with O_DIRECT to eliminate page cache maintenance in the OS kernel.

The performance degradation caused by the NUMA is a significant issue in high-performance computing. In a NUMA system with *n* CPU sockets, *block arrays* are virtually divided, along the z-dimension, into *n sub-blocks* that are calculated in each CPU socket with local cores. It is accomplished by either using a memory-bind and CPU-bind combination or using a *sections* statement in OpenMP with CPU-bind. In the latter case, the *sections* statement includes the same number of *section* statements as the number of CPU sockets. In each *section* statement, the same number of threads as the number of local cores are used for flash I/O and for the calculation of the allocated *sub-block array*. Every time when new data are read from the flash SSD to the *block arrays* after *bt*-step updates, *block arrays* are newly reallocated by the *free()* and *malloc()* functions. Actual data placement, i.e., physical page binding, is done when the data are first accessed by a thread (core) and the data (page) are placed in the local memory in the same socket to which the thread (core) belongs. In each section, generated threads are fixed to the local cores. This NUMA-aware tuning is done by treating each CPU socket as an independent computing node with distinct memory. It maximizes local memory accesses and minimizes remote ones by local cores.

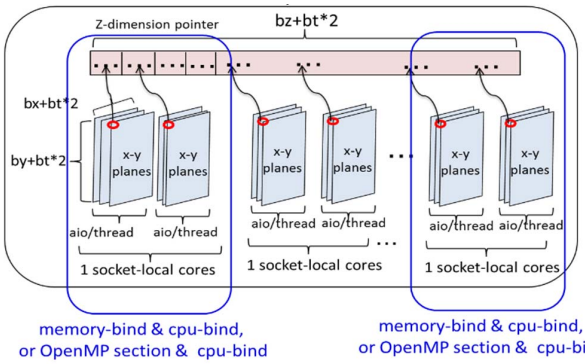

Fig. 8 Efficient memory layout of block arrays for AIO and NUMA-aware tuning.

B. Blocking parameter tuning by the Blk-Tune system

Finding the best spatial and temporal blocking parameter combination is very difficult but essential to achieve the highest performance. We proposed a blocking-parameter auto-tuning system named Blk-Tune [3]. In this paper, the extended-version of Blk-Tune is used. Fig. 9 shows the system diagram of Blk-Tune. Blk-Tune retrieves platform hardware information automatically by using the Portable Hardware Locality library [4] and selects the best spatial and temporal blocking parameter set for a given problem setting and a platform. It searches for a globally optimal parameter set to minimize the amount of data transferred between the flash device and the DRAM, which is the dominant factor affecting the performance of out-of-core algorithms using flash. Blk-Tune can be used for both online tuning of the application runtime and offline tuning in search of optimal blocking parameters. Blk-Tune is available for algorithm 1, algorithm 2, and their variants for single nodes.

In algorithm 2, the optimal blocking parameters can be determined by calculating only the amount of flash I/O. The greatest feature of Blk-Tune is the *just-in-time selection* of a globally optimal blocking parameter set during an actual execution, without any preliminary trial executions with various sets of parameters for problems and platforms to gather the information used for the actual executions, which are typical in ordinary auto-tuning systems. Blk-Tune can find an optimal blocking parameter set with minimum cost by a search technique using cost functions [3], rather than by tuning using trial-and-error methods. Although in algorithm 1 there is a tradeoff between increase in redundant calculation cost and decrease in flash I/O cost, Blk-Tune requires a CPU performance index to be compared with the flash device I/O bandwidth in the platform in use to achieve precise parameter selection. For algorithm 1, Blk-Tune requires a small piece of test-kernel execution to measure CPU performance and flash I/O performance indexes, which can be done at the start of the actual execution or in the preliminary offline execution. Moreover, Blk-Tune is available to determine an optimal temporal blocking size between multi-nodes in a cluster, as well as in a DRAM-flash layer and a cache-DRAM layer in single nodes. The details are omitted in this paper.

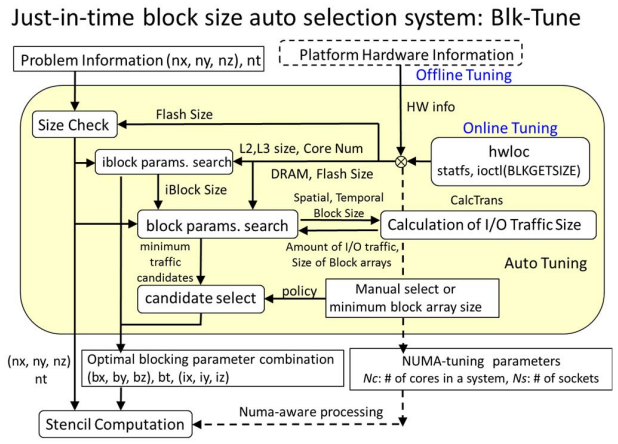

Fig. 9 Blk-Tune: A blocking parameter auto-tuning system for out-of-core stencil computations using flash, DRAM, and cache.

VI. Performance Evaluation Settings

A. Computing platforms and flash devices

The computation platforms used for the evaluations are shown in Table I. Table II shows various flash devices in a wide range of cost, form, power consumption, and performance. All flash devices except the fourth device (TSUBAME2.5 local SATA SSD) are attached via a PCIe interface to the computing node. Half of them are configured as RAID0 using two devices.

Fig. 10 shows the read/write performances (MB/s) of the individual cases, from no. 1 to no. 8 in Table II. They are measured on the crest4 platform shown in Table I, with a typical I/O size and a total data size used in the 64-GiB-size stencil computation problem in our algorithms. In this paper, four flash devices, namely, ioDrive2 (no. 1), SATA SSD (no. 4), M.2 AHCI flash (no. 5), and M.2 NVMe flash SW-RAID0 (no. 8), were used for the evaluations.

TABLE I. Computing platforms

	Hetero cluster 1-node crest4	Haswell cluster 1-node crest7	TSUBAME2.5 cluster S96 node
No. of nodes	1 - 4	1 - 4	1 - 32
Network	Infiniband single FDRx4(56Gbps)		Infiniband dual QDRx4(80Gbps)
Node CPU	Xeon E5-2687W (3.10GHz) 2 CPU x 8 Core	Huswell Xeon E5-2687W v3 (3.10GHz) 2 CPU x 10 Core	Westmere Xeon EP X5670 (2.93GHz) 2 CPU × 6 Core
Node Memory	DDR3-1600 8GiB x 8 (64 GiB)	DDR4-2133 16GiB x 8 (128 GiB)	DDR3 96 GiB (use 64 GiB)
L1 cache	32 KiB	32 KiB	32 KiB
L2 cache	256 KiB	256 KiB	256 KiB
L3 cache	20 MiB	25 MiB	12 MiB
Local Flash Storage	Fusion-io ioDrive2 (785 GB, or 1.2 TiB), or Samsung XP941 M.2 512 GB	Samsung 950 Pro M.2 MZ-V5P512B/IT, NVMe1.1, 512 GBx2 (1 TiB)	SATA-SSD (HP MK0120EAVDT 120GB x2), 160 GB
	No.1, No.2, No.5	No.8	No.4
OS	CentOS 7.1.1503 (x86_64), kernel 3.19.5	CentOS 7.1.1503 (x86_64), kernel 3.19.5	SUSE Linux Enterprise Server 11 SP3
Compiler	gcc version 4.8.3 -O3	gcc version 4.8.3 -O3	gcc 4.3.4 -O3

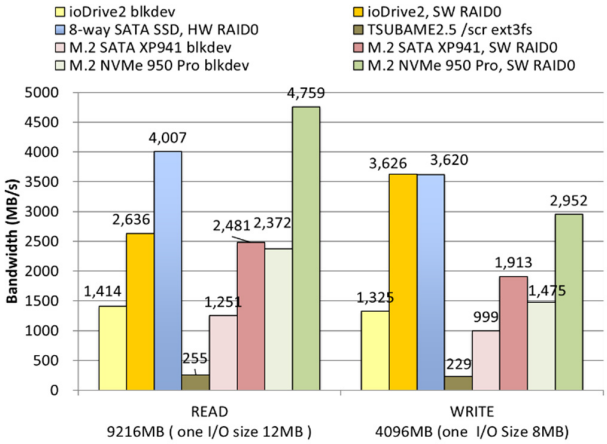

Fig. 10 Comparison of flash device performances: read/write throughput with the total size and the I/O size used in the 64-GiB-size (2K × 2K × 1K) stencil computation problem. Parallel asynchronous I/O with multiple threads was used.

TABLE II. Flash devices

No.	PCI-Connected Flash Device, Size	RAID	Thread	open() target
1	Fusion-io ioDrive2 MLC (1.2 TB) , PCIe x4 (Gen2)	-	16	block device
2	Fusion-io ioDrive2 MLC (1.2 TB & 785 GB), PCIe x4 (Gen2)	SW RAID0 (chunk: 1MB)	16	block device
3	Crucial M550 CT512M550SSD1 MLC 512GB x8 (4 TB), Adaptec RAID 81605ZQ, PCIe x8 (Gen3)	HW RAID0 (chunk: 256KB)	16	block device
4	TSUBAME2.5 Local Scratch HP MK0120EAVDT 120 GB SATA2.0 x2	SW RAID0 (chunk: 32KB)	12	file on ext3fs
5	Samsung XP941 MZHPU512HCGL-00004 M.2 AHCI, 512 GB, PCIe x4 (Gen2) adapter	-	16	block device
6	Samsung XP941 MZHPU512HCGL-00004 M.2 AHCI, 512 GB x2 (1 TiB), PCIe x4 (Gen2) adapter	SW RAID0 (chunk: 1MB)	16	block device
7	Samsung 950 Pro M.2 MZ-V5P512B/IT NVMe1.1, 512 GB, PCIe x4 (Gen3) adapter	-	16	block device
8	Samsung 950 Pro M.2 MZ-V5P512B/IT NVMe1.1, 512 GB x2 (1 TB), PCIe x4 (Gen3)	SW RAID0 (chunk: 1MB)	16	block device

VII. Performance Evaluation in a single node

The performance evaluations in this paper employed a seven-point stencil computation for a 3D data grid as a general case, which is the most difficult case to gain high performance when using slow memory such as flash because it has the simplest calculation compared to memory accesses. The proposed algorithms are widely available for different types of stencil computations that update data arrays using a double-buffer scheme. The programs used here are executed as different calculation kernels, e.g., 19-point and 27-point stencil computations, by defining the parameters. In this paper, the multi-node stencil algorithm and its implementation in Section IV are used even for single-node evaluations, unlike the ones in earlier works [1-3]. By specifying a node configuration as 1 × 1 in mesh size, we can realize single-node stencil computation without using Message Passing Interface (MPI) communications. In this case, *sub-buffer arrays* have no redundant area for temporal blocking for multi-nodes.

A. One-level redundant (AL1) vs. two-level redundant (AL2)

In this section, two redundant algorithms, AL1 (one-level redundant) and AL2 (two-level redundant), are compared in various-size problems with/without *section* statements for NUMA tuning. Figs. 11 and 12 show a comparison of the execution times between the two algorithms. They also show the problem settings, size (GB) and time step (nt), and spatial and temporal blocking parameters for a flash and DRAM tier, which were selected by Blk-Tune for each problem and the crest7 platform.

Fig. 11 shows the case without *section* statements for NUMA tuning, and Fig. 12 shows the one with them. All cases without *section* statements in Fig. 11 took a longer time compared to the corresponding cases in Fig. 12. In one-level redundant, the times were reduced to 50-57% by NUMA tuning. In two-level redundant, the times were reduced to 85-87%. Thus, using *section* statements in OpenMP is very effective to achieve better performance.

Thereafter, AL1 and AL2 were compared with and without NUMA tuning. AL2 performed better than AL1 in the case without NUMA tuning, as shown in Fig.11, but it performed worse than AL1 when using NUMA tuning, as shown in Fig. 12. The difference between AL1 and AL2 is in the schemes used for the top memory tier (cache and DRAM tier) shown in Fig. 1. AL1 employs spatial blocking, whereas AL2 employs temporal blocking. Generally, AL2 seems to have an advantage because it explores more localities in data accesses compared to AL1, which only uses spatial blocking. Actually, AL2 with temporal blocking reduces execution times in 67-80% of the times compared to AL1, as shown in Fig. 11.

The results are due to the data structure in the top memory tier. AL2 allocates small *iblock arrays* besides *block arrays* in DRAM for two-level temporal blocking. The *iblock arrays* are allocated in local memory in the same CPU socket. In AL2, read/write operations between *block arrays* and *iblock arrays* occur every *ibt* step for temporal blocking. In the case without NUMA tuning, the data of *block arrays* are spread over remote and local memories. Thus, AL2 has an advantage in increasing data access locality by copying *block array* data in remote memory to *iblock arrays* in local memory.

On the other hand, AL1 has no extra *iblock arrays* in DRAM. Instead, it accesses *block arrays* directly in an ordinary spatial blocking fashion, without the redundant memory copying and calculations performed by AL2. In the case with NUMA tuning, where most of the *block array* data are in the local memory, AL2's advantage in increasing local access and its disadvantage in memory copying and redundant calculations change the situation. Besides this tradeoff, the performance of AL2 is influenced more by the blocking parameters for iblock and by the cache behavior in the platform. Blk-Tune can select a near-optimal spatial blocking size for AL1 [3], but it is not easy to choose an optimal spatial and temporal blocking size for AL2.

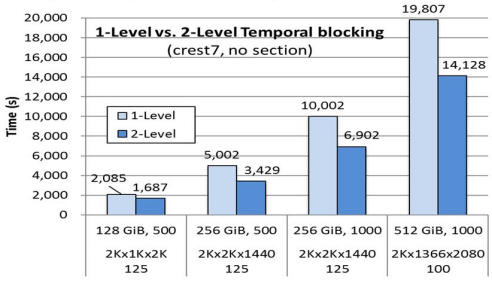

Fig. 11 Comparison between one-level redundant and two-level redundant in terms of execution times without NUMA tuning.

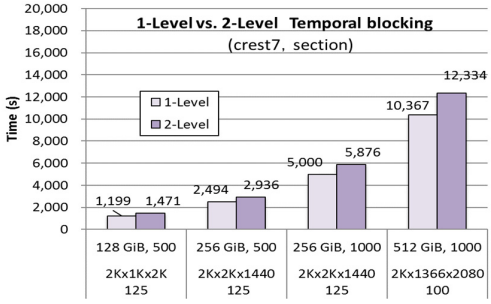

Fig. 12 Comparison between one-level redundant and two-level redundant in terms of execution times with NUMA tuning.

B. One-level redundant (AL1) vs. one-level non-redundant (AL3)

The performances of the three algorithms, AL1, AL2, and AL3, with/without *section* statements for NUMA tuning were compared. Figs. 13 and 14 show the execution times and effective Mflops for a 256-GB problem (domain grid size: 2048 × 2048 × 4096, 500 time steps) in the crest7 platform (128 GiB DRAM). Among these algorithms, one-level non-redundant (AL3) achieved the best performance and one-level redundant (AL1) was the second best when NUMA tuning was applied. The same results were observed in the crest4 platform (64 GiB DRAM).

Figs. 15 and 16 show a comparison between the best and the second best algorithms, AL3 and AL1, respectively, in execution times and effective Mflops for various problems. Two algorithms were evaluated for the 64-GB to 512-GiB problems with 500 or 1000 update steps on the crest7 platform. The leftmost bar in each graph corresponds to the results using only DRAM without flash access. The relative performance degradation is also shown in Fig. 16. For the 512-GiB problem,

with four times larger size of main memory capacity in crest7, AL3 and AL1 respectively achieved 85% and 72% of the performance gained by AL3 in the 64-GiB problem execution using only DRAM. It is a remarkable result because 75% of the data in the problem existed in flash, whose access latency is a thousand times slower than DRAM access latency.

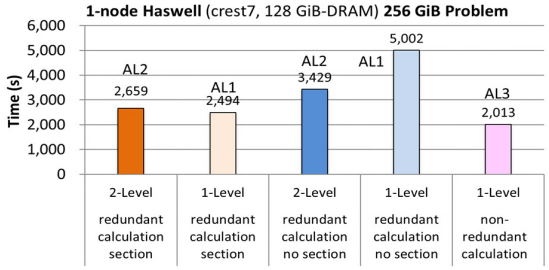

Fig. 13 Comparison of the three algorithms with/without NUMA tuning in terms of execution times for the 256-GIB problem (500 steps).

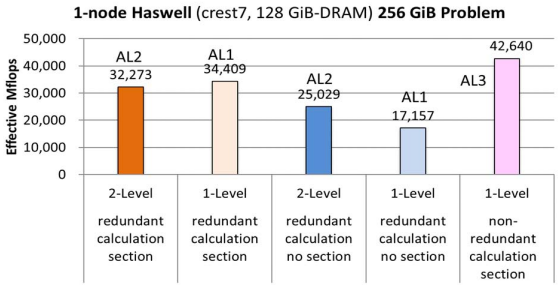

Fig. 14 Comparison of the three algorithms with/without NUMA tuning in terms of Mflops for the 256-GIB problem (500 steps).

Figs. 17 and 18 show the execution times and effective Mflops in crest4 for the 32-GiB to 1-TiB problems by the AL1 algorithm. The performance of the flash device in crest4 (no. 1) was lower than that of the flash device in crest7 (no. 8). Its bandwidth was less than half of that in crest7, as shown in Fig. 10. Fig. 17 shows the time components, flash write (blk2buf), flash read (buf2blk), and calculation. In the 1-TiB problem, the flash read and write components were 17.5% and 13.0% of the total time, respectively. Fig. 18 also shows the relative performance of various-size problems normalized by the Mflops in the 32-GiB problem, which runs only on DRAM without flash access. Even when using the lower performance flash device in crest4, AL1 achieved 63% (for the 1-TiB problem in the rightmost part of Fig. 18) of the performance gained in the 32-GiB problem, which runs on DRAM without flash access. The 1-TiB problem corresponds to 16 times larger size of main memory capacity in crest4. In other words, performance degradation by using flash as a main memory extension was only 37%, even if 93.75% (=15/16) of the problem data were in the flash device. Moreover, the performance degradation according to the increase in problem size in the same platform was limited, i.e., 0.72-0.63 for the 128-GiB to 1-TiB problems in Fig. 18. The reason is that the available block array size in DRAM, which is constant in the same platform, is a key component in its performance.

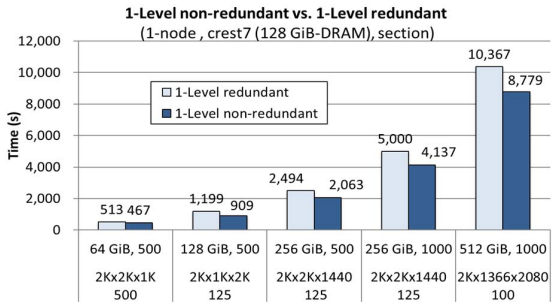

Fig. 15 Execution times for various-size problems in the one-level redundant algorithm and one-level non-redundant algorithm in crest7.

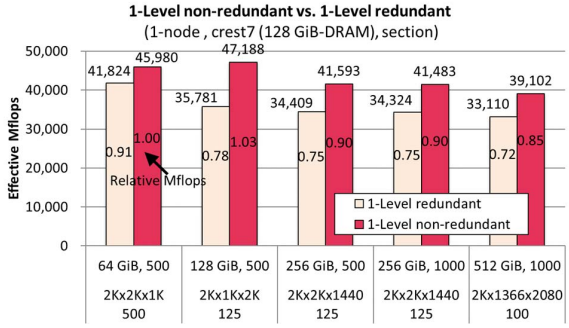

Fig. 16 Effective Mflops and relative Mflops for various-size problems in the one-level redundant algorithm and one-level non-redundant algorithm in crest7.

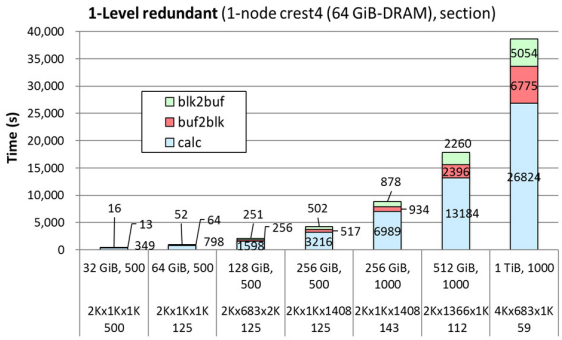

Fig. 17 Time components for various-size problems in the one-level redundant algorithm in crest4 (64 GiB DRAM).

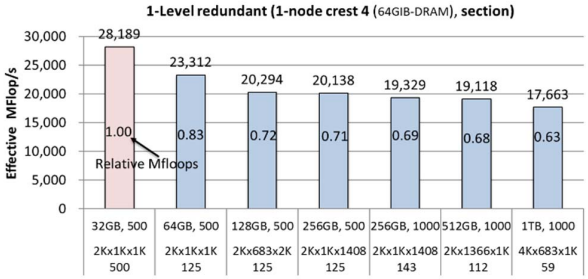

Fig. 18 Effective Mflops and relative Mflops for various-size problems in the one-level redundant algorithm in crest4 (64GiB-DRAM).

VIII. Performance Evaluation in Multi-node

A. Performance in the latest cluster with NVMe-flash devices

The latest Haswell cluster was used for the multi-node evaluation of the three algorithms. Its nodes had lower-power, cost-effective, and small-shape 1 TiB NVMe flash devices. The local flash was configured by software-RAID0 using two M.2-form 512 GiB devices with a PCIe (gen3) × 4 adapter. In the multi-node version, another redundant temporal blocking technique was introduced in multi-nodes, which influenced the performance.

Figs. 19 and 20 compare the three algorithms in terms of times and Mflops for the 512-GiB and 1-TiB problems on a four-node cluster. The difference between the three algorithms in single nodes (Fig. 13) became smaller in the multi-node case in Fig. 19. In this evaluation, AL1 became the best and AL3 dropped to third place. AL1 achieved 3.71 times higher performance in the four-node cluster than in the one-node cluster for the 512-GiB problem, whereas AL3 gained 2.93 times higher performance. When comparing the performances in the 512-GiB and 1-TiB problems with a fixed capacity of DRAM (128 GiB), the performance degradations were 4.4% in AL1 and 11.7% in AL3. AL1 achieved highly efficient performance. Fig. 21 shows the time components for the three algorithms. Flash write time (blk2buf) in AL3 was 1.4 times longer than those in AL2 and AL1. As described in Section II, AL3 performs more flash accesses compared to AL1 and AL2. Moreover, the *sub-buffer array* size became larger in multi-nodes than in single nodes because it was expanded with an extra sleeve area, which was proportional to the temporal blocking size, *bt,* for multi-nodes. It is also important to choose an appropriate temporal blocking size in multi-nodes. Data exchange costs, including flash read/write and MPI communications, were small in the four-node cluster, which were 1-3% of the total time, as shown in Fig. 22.

The left part of Fig. 23 shows the performance of multi-nodes when the processing array size per node was fixed to 256 GiB. When using one node, there were no data exchange cost and no redundant sleeve area for the *sub-buffer arrays*. In the multi-node case, there were both. However, the two-node case had half the cost of data exchange compared to the four-node case. According to the right part of Fig. 23, the performances in the four-node cluster were 1.79 times higher than those in the two-node cluster.

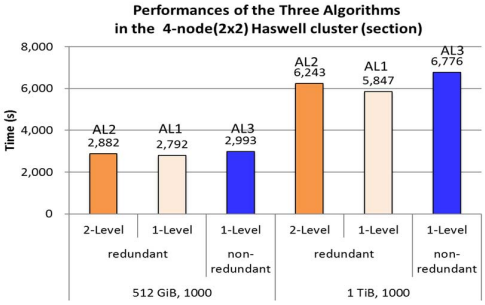

Fig. 19 Execution times of the three algorithms (AL2, AL1, and AL3 with NUMA tuning) for the 512-GB and 1-TB problems in the four-node Haswell cluster.

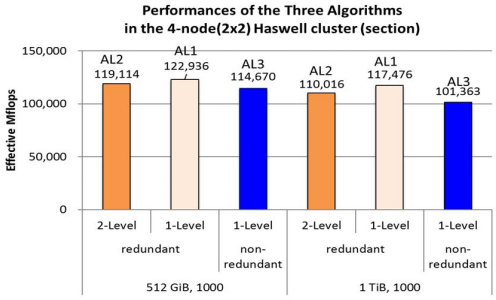

Fig. 20 Mflops of the three algorithms (AL2, AL1, and AL3 with NUMA tuning) for the 512-GB and 1-TB problems in the four-node Haswell cluster.

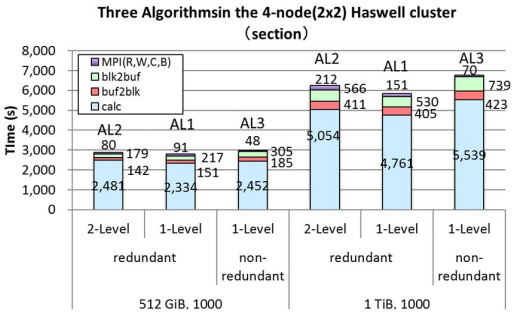

Fig. 21 Time components in the three algorithms (AL2, AL1, and AL3) for the 512-GB and 1-TB problems in the four-node Haswell cluster.

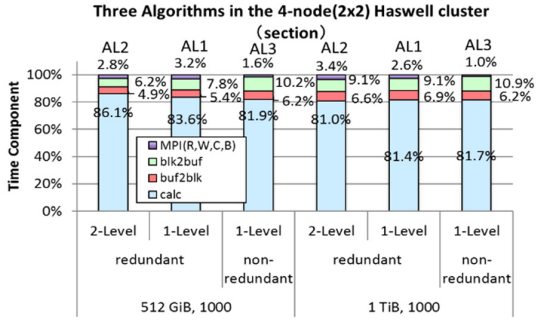

Fig. 22 Time components in the three algorithms (AL2, AL1, and AL3) for the 512-GB and 1-TB problems in the four-node Haswell cluster.

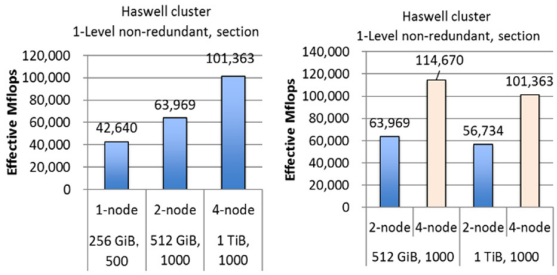

Fig. 23 Mflops using the one-level non-redundant algorithm (AL3) in the Haswell cluster.

B. Performance in the cluster with PCIe-based flashes

To show the scalability of AL1 in multiple nodes, we show the evaluation results in the four-node hetero-cluster, which

has three different (heterogeneous) flash devices, in Figs. 24 and 25. Fig. 24 shows the performances of a fixed-size problem (1 TiB size, 256 steps) on clusters with one, two, and four nodes (2 × 2 mesh and 1 × 4 mesh). Fig. 25 shows the performances when the processing size per node was fixed to 256 GiB with 256 steps.

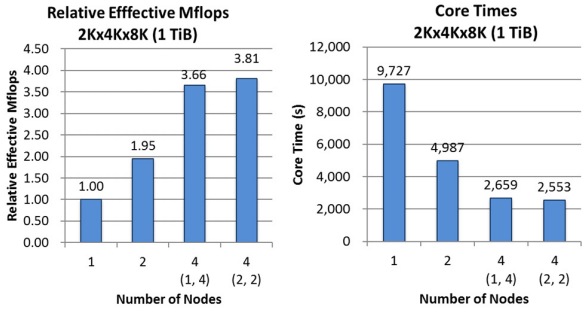

Fig. 24 Relative Mflops and times using the one-level redundant algorithm (AL1) in the Hetero-cluster. 1-TiB problem (2K × 4X × 8K, 256 steps).

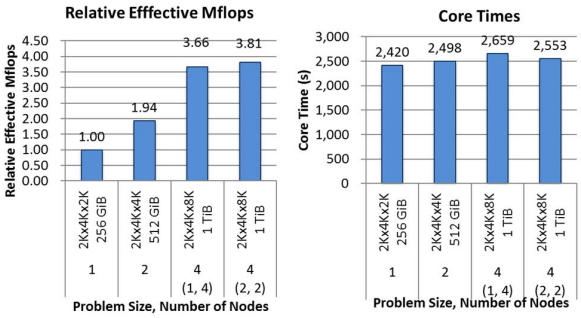

Fig. 25 Relative Mflops and times using the one-level redundant algorithm (AL1) in the Hetero-cluster. The processing size per node was fixed to 256 GiB with 256 steps.

C. *Performance in the cluster with SATA SSDs*

To evaluate the algorithms with more than four nodes, we used the TSUBAME2.5 cluster [5]. Unfortunately, there was no available cluster equipped with a local flash SSD of sufficiently large capacity compared to the node memory capacity. The available size of the local SSD in TSUBAME2.5 was limited to 120 GiB (device capacity of 140 GiB), which is relatively small compared to the node memory size, 96 GiB, for evaluating out-of-core computations. Moreover, the number of available nodes was limited to 32. We used only 64 GiB of DRAM and a SATA flash SSD (120 GiB) for one node for the evaluations. The performance of the local SSD in TSUBAME2.5 was the lowest, as shown in Fig. 10. The local SSD's read and write bandwidths correspond respectively to 5.3% and 7.8% of the read and write bandwidths of the latest NVMe flash used in the Haswell cluster.

First, AL1 and AL3 were compared in terms of execution times and amount of data I/O in flash SSD, as shown in Fig. 26. The results showed that AL3 had a longer read/write time compared to AL1. AL3 generated two times larger the amount of data transfer in flash and consumed six times longer flash write time compared to AL1, the redundant algorithm. As shown in Fig. 5, the non-redundant algorithm reduced the calculation but generated more flash read/write operations naturally. The performance gap between the two algorithms was amplified by the low performance of the local SATA SSD in TSUBAME2.5. Thus, redundant algorithm is better than non-redundant one when using low-specification flash devices.

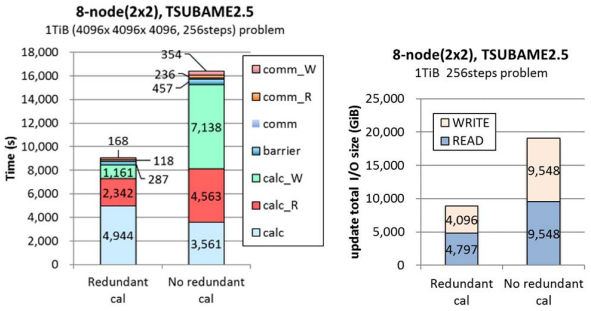

Fig. 26 Comparison between the one-level redundant algorithm (AL1) and the one-level non-redundant algorithm (AL3) in the TSUBAME2.5 cluster.

Fig. 27 shows the relative Mflops and times of the fixed-size problem (1 TiB size, 256 steps) on clusters with 8, 16, and 32 nodes (2 × 4, 4 × 4, and 4 × 8 in mesh). Owing to the local flash capacity limitation, it is impossible to run on less than eight nodes for the 1-TiB problem. Fig. 28 shows the performance when the processing size per node was fixed to 128 GiB with 256 steps. Although the execution times were not small compared to the results in the other two clusters, AL1's scalability over nodes was good. Its parallel efficiency reached 97% in 16 nodes and 91% in 32 nodes, when the performance in the eight-node cluster was used as the basis.

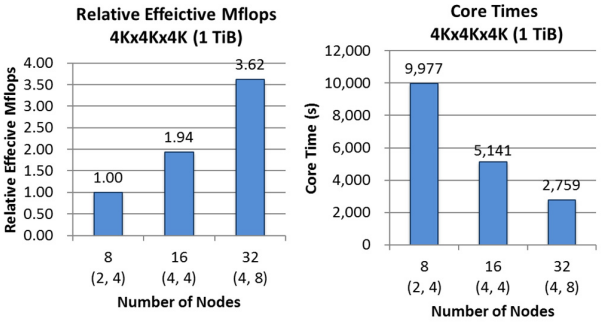

Fig. 27 Relative Mflops and times using the one-level redundant algorithm (AL1) in the TSUBAME2.5 cluster. 1-TiB problem (4K × 4X × 4K, 256 steps).

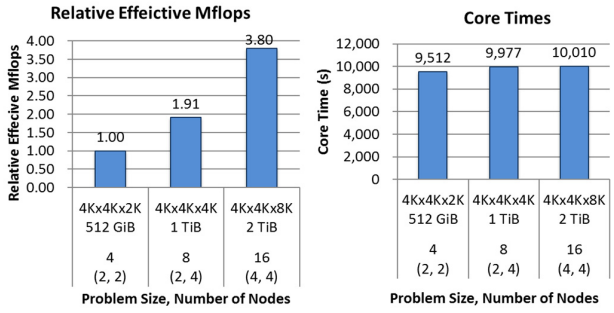

Fig. 28 Relative Mflops and times using the one-level redundant algorithm (AL1) in the TSUBAME2.5 cluster. The processing size per node was fixed to 128 GiB with 256 steps.

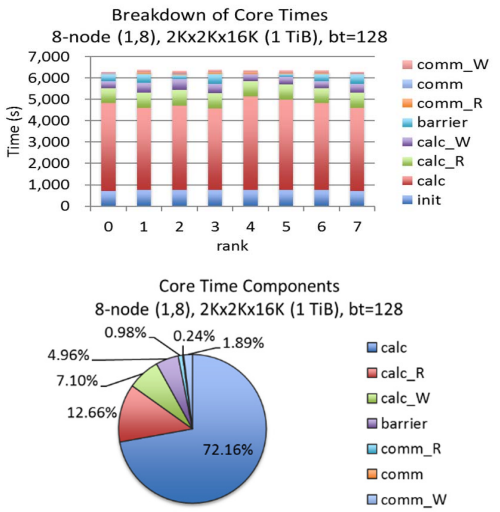

Fig. 29 Time components in the 1-TiB problem (2K × 2K × 16K, 256 steps) on eight nodes in the Tsubame2.5 cluster.

The time components in the 1-TiB problem (2K × 2K × 16K, 256 steps) on eight nodes are shown in Fig. 29. When using many nodes, the barrier time for synchronizing the nodes before the data exchange grew to 4.96% of the total time in the eight nodes. The data exchange time component was 8.07% in total. Although the MPI send/receive time was still small, 0.24%, the flash write and read times for the data exchange increased compared to those in the Haswell cluster because of the low performance of the SATA SSD. Even when using such a traditional flash device, SATA SSD, our algorithms are applicable to a larger-scale problem which requires greater data size beyond the total size of the main memories in a cluster.

IX. Conclusion and Future Work

This paper proposes a new scheme to solve data size requirements for large-scale stencil computation problems, which are greater than the total size of the main memories of nodes in a cluster. It utilized distributed flash SSDs over cluster nodes as an extension to the main memory with a locality-aware algorithm. A multi-level blocking scheme for cache, main memory, local flash, and remote node was introduced and evaluated on a cluster with state-of-the-art flash devices as well as with traditional SSDs. The results showed that large-scale stencil problems can be solved with a limited number of nodes and a moderate size of main memories, which reduces system cost and energy consumption.

Our current work includes the development of an automatic optimal blocking parameter selection for every memory hierarchy in a multi-node system.

References

[1] H. Midorikawa, H. Tan, and T. Endo, "An Evaluation of the Potential of Flash SSD as Large and Slow Memory for Stencil Computations," Proc. 2014 Int. Conf. on High Performance Computing and Simulation IEEE-HPCS2014, Jul. 2014, pp. 268-277.

[2] H. Midorikawa and H. Tan, "Locality-Aware Stencil Computations Using Flash SSDs as Main Memory Extension," Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing CCGrid2015, May 2015, pp. 1163-1168.

[3] H. Midorikawa: "Blk-Tune: Blocking Parameter Auto-Tuning to Minimize Input-Output Traffic for Flash-Based Out-of-Core Stencil Computations", The 11th Int. workshop on Automatic Performance Tuning, iWAPT2016, IEEE Int. Parallel and Distributed Process. Symp. workshops (DOI 10.1109/IPDPSW.2016.48), pp.1516-1526, May 2016

[4] Portable Hardware Locality (hwloc). https://www.open-mpi.org/projects/hwloc/

[5] TSUBAME2.5 http://tsubame.gsic.titech.ac.jp/en/node/1072.

[6] Proc. of Flash Memory Summit 2016, Aug. 9-11, 2016, Santa Clara. http://www.flashmemorysummit.com/English/Collaterals/Documents/FlashMemorySummit_Preview_Program.pdf

[7] M. Wolfe, "More Iteration Space Tiling", Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 655-664, 1989.

[8] F. Bassetti, K. Davis, and D. Quinlan, "Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures", ISCOPE'98, LNCS 1505, pp107-118, 1998.

[9] L. Renganaranaya et al., "Toward Optimal Multi-level Tiling for Stencil Computations", IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007).

[10] D. Wonnacott, "Using Time Skewing to Eliminate Idle Time Due to Memory Bandwidth and Network Limitations", International Parallel and Distributed Processing Symposium (IPDPS 2000).

[11] G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske, "Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization", Computer Software and Applications Conference, vol. 1, pp. 579-586, 2009.

[12] M. Wittmann, G. Hager, and G. Wellein, "Multicore-Aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory", Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7 pages, April 2010.

[13] G. Jin, T. Endo, and S. Matsuoka, "A Parallel Optimization Method for Stencil Computation on the Domain That Is Bigger Than Memory Capacity of GPUs", IEEE Cluster2013, 2013.

[14] K. Datta et al. "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures", Proc. SC2008.

[15] M. Frigo and V. Strumpen, "Cache Oblivious Stencil Computations", Proc. ICS 2005.

[16] D. G. Wonnacott and M. M. Strout, "On the Scalability of Loop Tiling Techniques", IMPACT2013, pp. 3-11.

[17] Y. Zhang and F. Mueller, "Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters", IEEE Trans. Parallel and Distributed Syst., vol. 24, pp. 417-427, May 2012.

[18] Y. Luo, G. Tan, Z. Mo, and N. Sun, "FAST: A Fast Stencil Autotuning Framework Based on an Optimal-Solution Space Model," ICS '15 Proc. 29th ACM Int. Conf. on Supercomputing, 2015, pp. 187-196

[19] J. D. Garvey and T. S. Abdelrahman, "Automatic Performance Tuning of Stencil Computations on GPUs," Parallel Processing (ICPP), 44th Int. Conf., 2015, pp. 300-309

[20] P. Basu, M. Hall, and S. Williams, "Compiler-Directed Transformation for Higher-Order Stencils," IEEE Int. Parallel and Distributed Process. Symp. (IPDPS), 2015, pp. 313-323,

[21] V. Bandishti, I. Pananilath, U. Bondhugula, "Tiling Stencil Computations to Maximize Parallelism", IEEE SC12, 2012

[22] A. Acharya, U. Bondhugula, "PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality", ACM PPoPP 2015, pp.54-64

[23] Improve Linux SWAP for High Speed Flash Storage. http://events.linuxfoundation.org/sites/events/files/lcjpcojp13_shaohua.pdf.

[24] OpenNVM, FusionIO, http://opennvm.github.io.

[25] M Bjørling, J Axboe, D Nellans, P Bonnet, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems", Proc. of the 6th International Systems and Storage Conference (SYSTOR '13)