DOI: 10.14529/jsfi190106

Performance Evaluation of Different Implementation Schemes of an Iterative Flow Solver on Modern Vector Machines

Kenta Yamaguchi[1,2]***, Takashi Soga***[2,4]***, Yoichi Shimomura***[2,4]***,***
Thorsten Reimann[3]***, Kazuhiko Komatsu***[2]***, Ryusuke Egawa***[2]***,***
Akihiro Musa[2,4]***, Hiroyuki Takizawa***[2]***, Hiroaki Kobayashi***[5]

Modern supercomputers consist of multi-core processors, and these processors have recently employed vector instructions, or so-called SIMD instructions, to improve performances. Numerical simulations need to be vectorized in order to achieve higher performance on these processors. Various legacy numerical simulation codes that have been utilized for a long time often contain two versions of source codes: a non-vectorized version and a vectorized version that is optimized for old vector supercomputers. It is important to clarify which version is better for modern supercomputers in order to achieve higher performance. In this paper, we evaluate the performances of a legacy fluid dynamics simulation code called FASTEST on modern supercomputers in order to provide a guidepost for migrating such codes to modern supercomputers. The solver has a non-vectorized version and a vectorized version, and the latter uses the hyperplane ordering method for vectorization. For the evaluation, we also implement the red-black ordering method, which is another way to vectorize the solver. Then, we examine the performance on NEC SX-ACE, SX-Aurora TSUBASA, Intel Xeon Gold, and Xeon Phi. The results show that the shortest execution times are with the red-black ordering method on SX-ACE and SX-Aurora TSUBASA, and with the non-vectorized version on Xeon Gold and Xeon Phi. Therefore, achieving a higher performance on multiple modern supercomputers potentially requires maintenance of multiple code versions. We also show that the red-black ordering method is more promising to achieve high performance on modern supercomputers.

Keywords: performance evaluation, legacy code, numerical fluid dynamics simulation, vectorization, hyperplane method, red-black method.

Introduction

Numerical simulations of fluid dynamics can solve many of the important problems the scientific and engineering field faces today. Simulations for analyzing and understanding more complex problems require higher performance supercomputers. Modern supercomputers consist of multi-core processors such as Intel Xeon and Xeon Phi processors to increase the degree of parallelism. Moreover, modern supercomputers have employed vector instructions to exploit the loop-level parallelism. Thus, in order to achieve a higher performance on these processors, it is necessary to vectorize the simulation codes. In this paper, the supercomputers composed of processors employing vector instructions are referred to as *modern vector machines*.

Many numerical simulations of fluid dynamics utilize implicit methods for stably solving large-scale models. However, the implicit methods generally have loop-carried dependencies on stencil calculations, and these simulations cannot be vectorized in a straightforward way. The hyperplane ordering method [1] was devised to vectorize them, and until recently this method was widely used for vectorizing the implicit methods on vector supercomputers. Accordingly,

[1]NEC Solution Innovators, Tokyo, Japan
[2]Cyberscience Center, Tohoku University, Sendai, Japan
[3]Technische Universität Darmstadt, Darmstadt, Germany
[4]NEC Corporation, Tokyo, Japan
[5]Graduate School of Information Science, Tohoku University, Sendai, Japan

some legacy numerical simulation codes (called old vectorized codes) often have two versions of source codes: non-vectorized version (called naive version) and vectorized version.

As the hyperplane ordering method generally increases memory loads, Soga et al. [2] utilized the red-black ordering method to vectorize the successive over-relaxation method, and then demonstrated that both of the vector supercomputer SX-9 system and Intel Xeon processor (Nehalem-EX) can achieve higher performance by using the red-black ordering method in comparison with the hyperplane ordering method. Therefore, old vectorized codes may require another vector optimization (i.e., red-black ordering) to achieve a higher performance on modern vector machines.

In this paper, we evaluate the performances of the naive version, hyperplane, and red-black ordering methods on modern vector machines, NEC SX-ACE, SX-Aurora TSUBASA (hereafter: SX-Aurora), Intel Xeon Gold, and Xeon Phi (Knights Landing), so as to clarify which method achieves higher performance. Here, the finite-volume solver FASTEST [3] serves as basis for our evaluation.

In Section 1, we present related work regarding optimization and performance evaluations of FASTEST. Section 2 explains the system architecture of four modern vector machines. Section 3 provides an overview of the FASTEST code. Section 4 presents the results of performance evaluations. We conclude in Summary section with a brief summary and mention of future work.

1. Related Work

FASTEST is a legacy fluid dynamics code and was originally developed in the 1990s. Scheit and Becker [4] optimized the FASTEST-3D code for multi-core processors and evaluated its performance on an Intel Xeon eight-cores processor (*Sandy Bridge*) and six-cores processor (*Westmere*). They found that the code was mostly memory-bound and that the solver of Stone's strongly implicit (SIP) method [5] was the main performance bottleneck. In order to decrease the memory load, they used single-precision floating-point data, and avoided unnecessary recomputation of the incomplete LU factorization to solve the SIP method. This improved the performance by about 40 %. They also utilized non-blocking MPI functions and showed that the scalability improved, i.e. the number of nodes of the optimized code was 2.8 times more than that of the non-optimized code when each parallel efficiency was *50 %*.

Burger et al. [6] examined optimizations of memory access on the *sipsol* subroutine and found, similarly, that the solver of the SIP method was memory intensive and was the most time-consuming. In particular, the memory access on the vectorized version of the solver was inefficient due to memory accesses along a plane, called the *hyperplane*, which is a skew cutting plane across the three-dimensional space. The solver needs long-interval accesses. They packed data elements on a three-dimensional array into two two-dimensional arrays in sequential memory access and showed that the performance becomes double on an AMD Opteron twelve-cores processor.

These studies utilized an Intel Xeon processor and AMD Opteron along with the AVX instruction set, which enabled the simulation codes to be vectorized. However, there was no discussion from a view point of vectorization. Therefore, in this paper, we clarify the performance on the vectorized FASTEST codes on modern vector machines.

Table 1. Specifications of each machine used in the evaluation

		SX-ACE	SX-Aurora	Xeon Gold	Xeon Phi
Clock freq. (GHz)		1	1.4	2.6	1.3
CPU	No. of cores	4	8	16	64
	Perf. (Gflop/s)	256	2150	1331	2662
	LLC (MB)	-	16	22	32
	Memory (GB)	64	48	192	96
	(MCDRAM)				16
	Mem. BW (GB/s)	256	1200	128	102
	(MCDRAM)				409

2. Modern Vector Machines

Modern supercomputers consist of many processors and accelerators employing multi-core technologies. For example, the world's fastest supercomputer in the top 500 list in June 2018 [7] is equipped with 2,397,824 cores and achieved the theoretical peak-performance of 200.8 Pflop/s. Recently, scalar processors have provided support for vector instruction sets such as Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX), and AVX-512 instructions. Thanks to these vector instructions, numerical simulations have been accelerated not only on vector supercomputers but also on scalar supercomputers. In this section, we give overviews of modern vector machines, NEC SX-ACE, SX-Aurora TSUBASA, Intel Xeon Gold, and Xeon Phi. Table 1 lists the hardware configurations of each machine used in the evaluation.

2.1. SX-ACE

SX-ACE is a vector supercomputer launched by NEC in 2013 [8, 9]. Its processor consists of four vector cores being comprised of a vector processing unit (VPU), Assignable Data Buffer (ADB) and Miss Status Handling Register (MSHR). The VPU is a key component of SX-ACE. The vector length of SX-ACE is 256 vector elements, 8 B each, and VPU can execute 256 operations by a single vector instruction using 16 vector pipelines in 16 clock cycles. The VPU has two multiply units and two add units, which can be independently operated by different vector instructions. Thus the core can simultaneously execute 64 operations by four vector instructions in one clock cycle. As the clock frequency of the core is 1.0 GHz, the single core performance of SX-ACE is 64 Gflop/s. The total performance of a single processor reaches 256 Gflop/s with four cores. Each core is connected to a memory control unit (MCU) through the memory crossbar network at the memory bandwidth of 256 GB/s, and the bandwidth is shared by the four cores. Thanks to this architecture, a single core can use the entire bandwidth of 256 GB/s if the other three cores do not access the memory. Each core is equipped with 1 MB ADB, which is a software controllable data buffer with a directory that can be accessed by the VPU at the rate of 256 GB/s. Unlike caches on general scalar processors, ADB is controlled by manually inserting directives into the source program. Consequently, ADB can retain only reusable data, which will not be evicted by non-reusable data. MSHR is used to handle outstanding vector loads on cache misses by eliminating unnecessary vector load accesses. It can reduce redundant load requests between ADB and the main memory.

2.2. SX-Aurora TSUBASA

SX-Aurora TSUBASA was released in 2017 [10, 11]. An SX-Aurora TSUBASA system is comprised of a vector host (VH) and a vector engine (VE). A VE is implemented as a PCI Express (PCIe) card equipped with a vector processor, i.e., the card is connected to the VH via PCIe. One VH can control eight VEs. The vector processor consists of eight vector cores, a 16 MB last-level cache (LLC), and six High Bandwidth Memory 2 (HBM2) memory modules. The SX-Aurora TSUBASA's VPU has three fused multiply add (VFMA) units and each VFMA unit has 32 vector pipelines. The vector length of SX-Aurora TSUBASA is 256, which is the same as that of SX-ACE. Thus, a VPU can execute 256 operations by a single vector instruction in eight clock cycles. As the clock frequency of the VE is 1.4 GHz, a single core provides 268.8 Gflop/s (32 operations in one clock cycle $\times$ two floating-point operations (add and multiply) by VFMA $\times$ 3 VFMA units $\times$ 1.4 GHz). Hence, a vector processor, which consists of eight cores, achieves 2.15 Tflop/s. The LLC is directly connected to the vector registers of each core, and shared by eight cores. The six HBM2 memory modules deliver the high memory bandwidth of 1.288 TB/s in total. This memory architecture enables a high sustained performance especially in executing memory-intensive applications.

2.3. Intel Xeon Gold

Intel Xeon Gold is marketed as a 6th-generation Intel core. In this paper, we evaluate the performance of the application using a Xeon Gold 6142 processor, which consists of 16 cores. On the previous generations of Intel Xeon processor families, cores, last-level cache (LLC), memory controller, IO controller, and inter-socket Intel QuickPath Interconnect (Intel QPI) ports are connected together using a ring architecture. In contrast, the 6-th generation Xeon Gold processor introduces a mesh architecture that encompasses an array of vertical and horizontal communication paths. This architecture allows traversal from one core to another through the shortest path. The processor interconnect is Intel Ultra Path Interconnect (Intel UPI) which replaced the Intel QPI. Modern scalar processors such as the Xeon Gold processor introduce SIMD instructions. For example, AVX-512 instructions, which are supported by Xeon Gold, provide 512-bit-wide vector instructions, 32 logical registers, eight mask registers, and indirect vector access via gathers and scatters. A single AVX-512 instruction can execute 32 single-precision or 16 double-precision floating-point operations per cycle. AVX-512 instructions are classified into five categories: foundation instructions (AVX-512F), which are the base 512-bit extensions; conflict-detection instructions (AVX-512CD); doubleword and quadword instructions (AVX-512DQ); byte and word instructions (AVX-512BW); and vector length extensions (AVX-512VL) [12].

2.4. Intel Xeon Phi

We use the second-generation Intel Xeon Phi 7210, so called Knights Landing (KNL), for performance evaluation. The processor consists of 32 active physical tiles, and each tile is comprised of two cores and two vector processing units (VPU) per core. A 1 MB level-2 (L2) cache which is shared by two cores is also included in each tile. Multi-channel DRAM (MCDRAM) and double data rate (DDR) memory are used in the Xeon Phi. The total memory capacities of MCDRAM and DDR are 16 GB and 96 GB, respectively. MCDRAM, which is 3D-stacked DRAM integrated on-package, provides high bandwidth of 409 GB/s. The two types of memory

```
do k=2,nkblk-1
   lkk=(k-1)*nijblk
   do i=2,niblk-1
      lki=ijksblk+lkk+(i-1)*njblk
      do ijk=lki+2,lki+njblk-1
        res(ijk)=ae(ijk)*fi(ijk+njblk)+aw(ijk)*fi(ijk-njblk)+an(ijk)*fi(ijk+1)+as(ijk)*fi(ijk-1) &
                 +at(ijk)*fi(ijk+nijblk)+ab(ijk)*fi(ijk-nijblk)+su(ijk)-ap(ijk)*fi(ijk)
        rhelp(m)=rhelp(m)+abs(res(ijk))
        res(ijk)=(res(ijk)-bb(ijk)*res(ijk-nijblk)-bw(ijk)*res(ijk-njblk)-bs(ijk)*res(ijk-1))*bp(ijk)
end do; end do; end do
```

Figure 1. Source code of the naive version

provide three memory modes, which are selectable through the BIOS setting at boot time. One mode is the cache mode in which MCDRAM is used as a cache for the DDR memory. Another mode is the flat mode, which handles both MCDRAM and DDR in the same way to organize one memory space. The other is the hybrid mode, in which either 25 % or 50 % of MCDRAM is used as cache and the rest is used as memory. In this work, the flat mode is selected for the evaluation. As with Xeon Gold, the Xeon Phi's VPU also provides support for AVX-512 instructions. Xeon Phi's AVX-512 instructions fall into four categories. AVX-512F and AVX-512CD are the same as those in Xeon Gold. Two additional categories – exponential and reciprocal instructions (AVX-512ER) and prefetch instructions (AVX-PF) – are only provided by Xeon Phi [13].

3. Incompressible Flow Solver, FASTEST

FASTEST is a three-dimensional incompressible flow solver [6]. Several academic developers have independently enhanced it to simulate various flow phenomena and multi-physics couplings. In this paper, we use the code developed at Technische Universität Darmstadt, whose codes are parallelized by using MPI and OpenMP.

The solver is based on the Semi-Implicit method for the Pressure Linked Equations (SIMPLE) method [14], which iteratively solves the momentum and pressure-correction equations. These equations are discretized using a second-order finite-volume scheme on a block-structured grid. The resulting linear equation system is solved using the SIP method, which is based on an incomplete LU factorization. The subroutine containing the SIP method (called *sipsol* subroutine) is the main performance bottleneck and holds two different implementations [6]. The first, called the naive version, is shown in Fig. 1. This code calculates the residual calculations after the LU decomposition in the *sipsol* subroutine. Each data element in the figure is saved in a linearized one-dimensional array. All elements are calculated along the three coordinate axes via a nested loop using variables *lkk* and *lki*. The variable *lkk* indicates the start points of *k*. Similarly, the variable *lki* also indicates the start points of *i*. Here, a loop-carried dependency exists between the left-hand array *res(ijk)* and the right-hand arrays *res(ijk-nijblk)*, *res(ijk-njblk)*, and *res(ijk-1)* in the second line from the bottom in Fig. 1. Hence, this *do* loop cannot be vectorized.

The second one, called the HP version, utilizes the hyperplane ordering method. The first *do* loop in Fig. 2 converts a triple-nested *do* loop in Fig. 1 into a single *do* loop. The second nested *do* loop is a loop of the hyperplane. Array *ijkdia* constructs planes fulfilling the condition $i + j +$

```
do ijk=ijksblk+nijblk+njblk+2,ijksblk+(nkblk-2)*nijblk+njblk*(niblk-2)+njblk-1
  res(ijk)=ae(ijk)*fi(ijk+njblk)+aw(ijk)*fi(ijk-njblk)+an(ijk)*fi(ijk+1)+as(ijk)*fi(ijk-1) &
       +at(ijk)*fi(ijk+nijblk)+ab(ijk)*fi(ijk-nijblk)+su(ijk)-ap(ijk)*fi(ijk)
  rhelp(m)=rhelp(m)+abs(res(ijk))
end do
do ndia=1,numdia(ngr,m)
  do npoi=npsta(ndia,ngr,m)+1,npsta(ndia+1,ngr,m)
  ijk=ijkdia(npoi)
   res(ijk)=(res(ijk)-bb(npoi)*res(ijk-nijblk)-bw(npoi)*res(ijk-njblk)-bs(npoi)*res(ijk-1))*bp(npoi)
end do; end do
```

Figure 2. Source code of the HP version

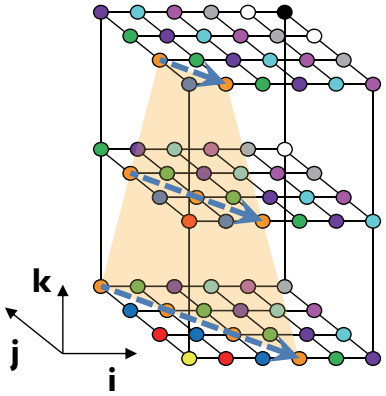

Figure 3. Diagram of a hyperplane

k = *constant*. The plane is a skew cutting plane across the three-dimensional space. The orange plane in Fig. 3 is one of these hyperplanes that consist of the same color across the grid in three-dimensional space. Then, the inner *do* loop calculates the data elements on the planes. Since this calculation does not have any loop-carried dependency, the code of the *sipsol* subroutine is then vectorized. However, the hyperplane ordering method generally increases the number of memory loads. This is because the memory accesses of array *res(ijk)* are stride accesses due to access along the planes, and they have a long access latency. Moreover, the length of loop *npoi* is a variable and the efficiency of vectorized calculation becomes low when *npoi* is small.

We implement the red-black ordering method (called the RB version), the memory loads of which are lower than those of the hyperplane ordering method, as shown in Fig. 4. However, the number of iterations required for convergence is generally increased on the red-black ordering method. Thus there is a performance trade-off with increasing the number of iterations. The *do* loop in the second line from the top in Fig. 4 is a switch of *Red* and *Black* executions using the *if* statement in the fourth line from the top in Fig. 4. Array *itbl* is a mask table to skip elements with the color that are not calculated at each iteration, and then the loop is vectorized. This implementation also increases the number of operations compared to that of the naive version because the number of iterations generally increases and inserting loop *n2* doubles the total iteration count within the loop nest. Although the *if* statement is used to skip the computation in the loop body, SX-ACE executes the computation of both *true* and *false* and then the results are discarded by the so-called masking capability [2]. The benefit of this implementation is to

```
do m2=ijkgit(ngr,m)+1,ijkgit(ngr,m)+njblk*niblk*nkblk,iblk
 do n2=1,2
  do ijk=m2,min(m2+iblk-1,ijkgit(ngr,m)+njblk*niblk*nkblk)
    if(itbl(ijk-ijkgit(ngr,m),1,1,n2,m).ne.1) cycle
    res(ijk)=ae(ijk)*fi(ijk+njblk)+aw(ijk)*fi(ijk-njblk)+an(ijk)*fi(ijk+1)+as(ijk)*fi(ijk-1) &
          +at(ijk)*fi(ijk+nijblk)+ab(ijk)*fi(ijk-nijblk)+su(ijk)-ap(ijk)*fi(ijk)
    rhelp(m)=rhelp(m)+abs(res(ijk))
    res(ijk)=(res(ijk)-bb(ijk)*res(ijk-nijblk)-bw(ijk)*res(ijk-njblk)-bs(ijk)*res(ijk+rb_jm1))*bp(ijk)
end do; end do; end do
```

Figure 4. Source code of the RB version

increase the length of the innermost loop. This code also uses cache blocking to improve the performance of memory accesses. This is possible because the data loaded at iteration *n2=1 (Red)* can be reused at iteration *n2=2 (Black)*. Here, the variable *iblk* is the size of a cache block.

4. Performance Evaluation

4.1. Experimental Setup

The performance of FASTEST is evaluated by using four modern vector machines: NEC SX-ACE, SX-Aurora TSUBASA, Intel Xeon Gold, and Xeon Phi. The evaluated codes are the naive version, the HP version, and the RB version, which are parallelized with OpenMP. Here, we evaluate performances of a single processor on each machine because the performance per processor is a key factor to achieve high performance on a large-scale parallel simulation. Then, since modern HPC applications are usually written with considering the cache memory, we use the model size of $(32 \times 32 \times 53)$ per processor so that the data potentially fit in the cache memory of each processor. Here, the maximum degree of parallelism using OpenMP is 16 determined from the model size. Thus, the code is executed on four cores of SX-ACE, eight cores of SX-Aurora, 16 cores of Xeon Gold, and 16 cores of Xeon Phi. The memory mode of Xeon Phi is set to the flat mode.

Table 2 shows the versions and options of each machine's Fortran compiler. The options are high-level optimizations and inlining functions and subroutines. The codes are parallelized by using OpenMP. Table 3 lists the compiler directives for each code on each machine. The *sipsol* subroutine is vectorized by using the compiler directives: *nodep*, *ivdep*, and *simd*. SX-ACE and SX-Aurora use the optimization directives: *vovertake*, *gthreorder*, and *gather_reorder*, which optimize the operation sequences of memory accesses. Here, the codes are executed with double precision floating-point operations.

4.2. Experimental Results and Discussion

FASTEST iteratively computes the momentum and pressure-correction equations from initial values of pressure and velocity at each point to their converged values. The number of iterations varies with optimized and vectorized methods. We measure the execution time of the *sipsol* subroutine per iteration, and Figure 5 shows the performance ratios of HP and RB ver-

Table 2. Versions and options of each compiler

Machine	Versions	Options
SX-ACE	FORTRAN90/SX Rev.534	-Popenmp -Chopt -pi
SX-Aurora	NEC Fortran 1.5.1	-fopenmp -O3 -msched-block
		-finline-functions
Xeon Gold	Intel Fortran 18.0.2.199	-qopenmp -O3 -xCORE-AVX512
		-ipo -no-prec-div -fp-model fast=2
Xeon Phi	Intel Fortran 18.0.2.199	-qopenmp -O3 -xMIC-AVX512
		-ipo -no-prec-div -fp-model fast=2

Table 3. Compiler directives of each code

Machine	Naive	HP	RB
SX-ACE	-	nodep, vovertake, gthreorder	nodep, vovertake
SX-Aurora	-	ivdep, vovertake, gather_reorder	ivdep, vovertake
Xeon Gold	-	ivdep, simd	ivdep, simd
Xeon Phi	-	ivdep, simd	ivdep, simd

sions to the naive version (the performance ratio = the execution time of naive version ÷ the execution time of HP (RB) version). Here, the execution times of the naive version are 0.156 seconds on SX-ACE, 0.053 seconds on SX-Aurora TSUBASA, 0.017 seconds on Xeon Gold, and 0.039 seconds on Xeon Phi. HP(non-vectorized) and RB(non-vectorized) indicate that the codes are not vectorized, and hence the performance differences between HP/RB and HP(non-vectorized)/RB(non-vectorized) indicate the performance gain by vectorization in Xeon Gold and Xeon Phi.

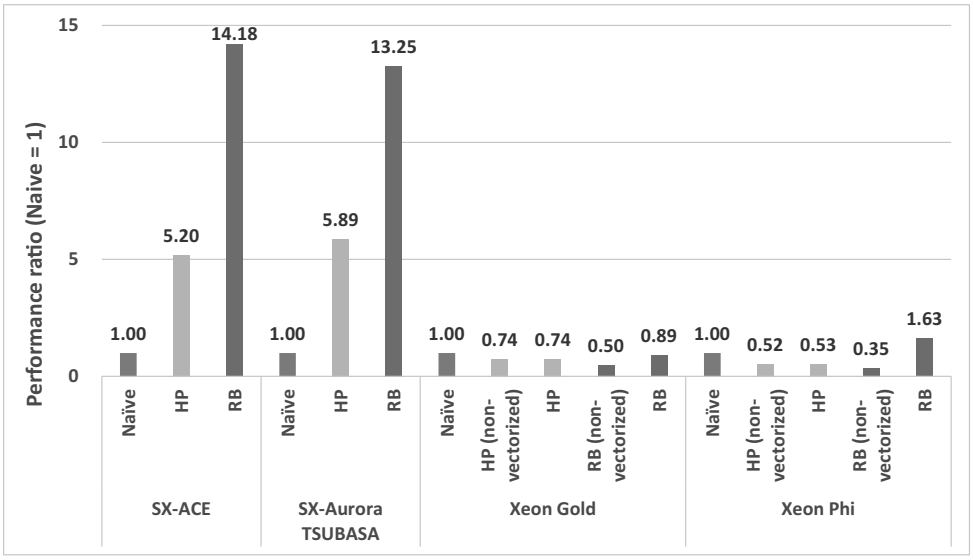

Figure 5. Performance ratio of HP and RB versions to naive version of the *sipsol* subroutine with an iteration on each machine

SX-ACE and SX-Aurora have the same performance characteristics. The vectorized codes, HP version and RB version, can achieve a higher performance than the naive version. In the case of SX-Aurora, the HP and RB versions are 5.89 and 13.25 times faster than the naive version, respectively. In other words, SX-ACE and SX-Aurora cannot achieve high performance unless codes are vectorized. Meanwhile, although both the HP and RB versions are vectorized, the performances of the HP version on SX-ACE and SX-Aurora are lower than that of the RB version, respectively. This is because the HP version needs indirect memory access and its memory access latency is longer than that of 2-stride memory access used in the RB version. Moreover, the efficiency of vectorized calculations becomes lower as the length of *npoi* is small.

Table 4 lists the ratios of stall time of the processor to total execution time on SX-ACE, SX-Aurora and Xeon Gold. Here, the execution statistics of all the versions on Xeon Phi and the naive version on SX-ACE and SX-Aurora cannot be measured by performance tools. This table shows that the stall time of the HP version is larger than that of the others, and then the HP version is inefficient on these machines.

Table 4. Ratio of stall time of processor on SX-ACE and Xeon Gold

	Naive	HP	RB
SX-ACE	-	76.6 %	56.5 %
SX-Aurora	-	72.7 %	67.9 %
Xeon Gold	11.7 %	63.3 %	31.4 %

In Xeon Gold and Xeon Phi, vectorization does not improve the performance of the HP version. The execution time is almost unchanged by enabling vectorization. In contrast, the vectorized RB version is 1.78 and 4.66 times faster than its non-vectorized version in Xeon Gold and Xeon Phi, respectively. Especially, the vectorized RB version on Xeon Phi is faster than the naive version. Meanwhile, as the clock frequency of Xeon Gold decreases, the performance gain of Xeon Gold by the vectorization is not high. However, this demonstrates that vectorization is important to achieve high performance even on Xeon Gold and Xeon Phi processors.

Table 5. Number of iterations on each method

	Naive	HP	RB
Number of iterations	10,004	10,004	17,516

Table 5 shows the number of iterations required by each method for convergence. The RB version requires 1.8 times more iterations than the naive and HP versions. The total execution times of the *sipsol* subroutine increase. Figure 6 shows the performance ratio of HP and RB versions to naive version (the performance ratio = the execution time of naive version ÷ the execution time of HP (RB) version). Here, the execution times of the naive version are 1565.5 seconds on SX-ACE, 530.8 seconds on SX-Aurora TSUBASA, 171.4 seconds on Xeon Gold, and 386.0 seconds on Xeon Phi.

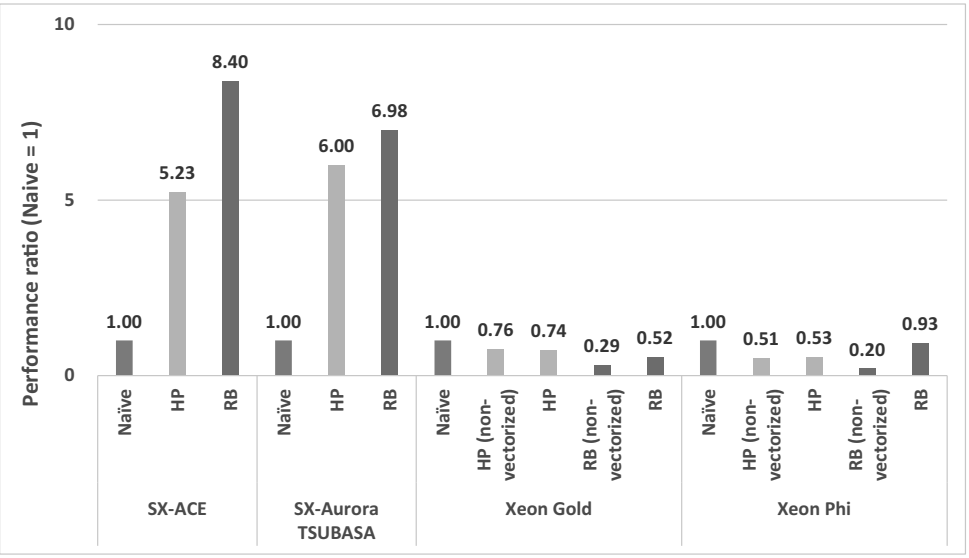

Figure 6. Performance ratio of HP and RB versions to naive version of the *sipsol* subroutine with the total execution times on each machine

In SX-ACE and SX-Aurora, the RB version can achieve the shortest execution time, in spite of an increase in the number of iterations. These results indicate that the red-black method is suitable to vectorize FASTET for SX-ACE and SX-Aurora. On the other hand, the naive version achieves the shortest execution time on Xeon Gold and Xeon Phi. These results show that, in case of the RB version, there is a trade-off between the performance degradation by needing more iterations and the performance improvement by using vectorization. Therefore, achieving a higher performance on multiple modern vector machines will require maintenance of multiple code versions. As a result, it will be more important to consider how to manage the code complexity in a systematic way in the future. We have been working on this problem [15, 16], and will further discuss such a methodology in our future work.

Summary

Processors of modern supercomputers have employed vector instructions to exploit loop-level parallelism efficiently, and the vector lengths are increasing, resulting in increasing the importance of vectorization to fully exploit the system performance. Various legacy numerical simulation codes have been vectorized considering old vector machines in mind, and hence the simulations have two different code versions of their kernel codes: non-vectorized version and vectorized version. In this paper, we evaluate the performances of such a legacy code called FASTEST, which is vectorized by the hyperplane and red-black ordering methods, using four modern vector machines (SX-ACE, SX-Aurora, Xeon Gold, and Xeon Phi) in order to provide which version is suitable for the modern vector machines.

Experimental results show that the red-black ordering method can achieve the shortest execution time on SX-ACE and SX-Aurora, while the naive version achieves the shortest execution time on Xeon Gold and Xeon Phi. This demonstrates that achieving higher performance on multiple modern vector machines will require maintenance of multiple code versions, namely, the naive version and the RB version. Overall, these results indicate that the red-black ordering method has the potential to achieve high performance on the modern vector machine, and that vectorization is a key optimization method of the modern vector machine.

In this work, we showed that the maintenance of multiple code versions is required to achieve higher performance on multiple modern vector machines. We will pursue our methodology of maintenance of multiple code versions. Moreover, since the number of cores on modern machines has increased year by year, our future work will investigate performance characteristics of thread-level parallelism such as OpenMP and OpenACC in order to achieve high performance on legacy numerical simulation codes.

Acknowledgements

This research was conducted as a joint project of Tohoku University and NEC Corporation. This research is partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems", DFG SPPEXA ExaFSA project, and Grant-in-Aid for Scientific Research(B) #16H02822. We are grateful for the use of NEC SX-ACE at Cyberscience Center at Tohoku University.

References

1. Fujino, S., Mori, M., Takeuchi, T.: Performance of Hyperplane ordering on vector computers, Journal of Computational and Applied Mathematics 38, 125–136 (1991), DOI: 10.1016/0377-0427(91)90165-G

2. Soga, T., Musa, A., Okabe, K., Komatsu, K., Egawa, R., Takizawa, H., Kobayashi, H., Takahashi, S., Sasaki, D., Nakahashi, K.: Performance of SOR Methods on Modern Vector and Scalar Processors, Journal of the Computers & Fluids 45(1), 215–221 (2011), DOI: 10.1016/j.compfluid.2010.12.024

3. Project Site for Fastest at Technische Universität Darmsadt. `https://www.fnb.tu-darmstadt.de/forschung_fnb/software_fnb/software_fnb.en.jsp`, accessed: 2019-01-21

4. Scheit, C., Becker, S., Hager, G., Treibig, J., Wellein, G.: Optimization of FASTEST-3D for Modern Multicore System. `https://arxiv.org/pdf/1303.4538`, accessed: 2019-01-21

5. Stone, H.L.: Iterative solution of implicit approximations of multidimensional partial differential equations, SIAM Journal on Numerical Analysis 5(3), 530–558 (1968), DOI: 10.1137/0705044

6. Burger, M., Bischof, C.: Optimizing the memory access performance of FASTEST's sipsol routine. In: 6th European Conference on Computational Fluid Dynamics, ECFD VI, July 2014, Barcelona, Spain. DOI: 10.13140/RG.2.2.32568.14089

7. Top 500 supercomputers sites. `https://www.top500.org/`, accessed: 2019-01-21

8. Momose, S.: Next generation vector supercomputer for providing higher sustained performance. In: Proceedings of IEEE Symposium on Low-Power and High-Speed Chips and Systems XVI, COOLChips 19, Yokohama, Japan, April 17–19, 2013

9. Egawa, R., Komatsu, K., Momose, S., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Potential of a Modern Vector Supercomputer for Practical Applications - Performance Evaluation of SX-ACE , The Journal of Supercomputing 73(9), 3948–3976 (2017), DOI: 10.1007/s11227-017-1993-y

10. Yamada Y., Momose, S.: Vector Engine Processor of NEC's Brand-New Supercomputer SX-Aurora TSUBASA. In: Proceedings of A Symposium on High Performance Chips, Hot Chips 30, Cupertino, California, USA, August 19–21, 2018

11. Komatsu, K., Momose, S., Isobe, Y., Sato, M., Musa, A., Kobayashi, H.: Early Evaluation of a New Vector Processor SX-Aurora TSUBASA. In: Research Poster of ISC Higher Performance 2018, ISC 18, Frankfurt, Germany, June 24–28, 2018

12. Mulnix, D.: Intel Xeon Processor Scalable Family Technical Overview. `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`, accessed: 2019-01-21

13. Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.-C.: Knight Landing: Second-Generation Intel Xeon Phi Product, IEEE Micro 36(2), 34–46 (2016), DOI: 10.1109/MM.2016.25

14. Patankar, S.V.: Numerical heat transfer and fluid flow, Hemisphere Publishing Corporation (1980)

15. Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R., Kobayashi, H.: Xevolver: An XML-based code translation framework for supporting HPC application migration. In: Proceedings of IEEE International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17–20, 2014, vol. 1, pp. 1–11 (2014)

16. Suda, R., Takizawa, H., Hirasawa, S.: Xevtgen: Fortran code transformer generator for high performance scientific codes, International Journal of Networking and Computing 6(2), 263–289 (2016)