International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 243–262, July 2016

A Memory-Efficient Implementation of a Plasmonics Simulation Application on SX-ACE

Raghunandan Mathur
NEC Technologies India, Noida, U.P, 201303, India

Hiroshi Matsuoka, Osamu Watanabe, Akihiro Musa
Cyberscience Center, Tohoku University, Sendai, 980-8578, Japan
NEC Corporation Tokyo, 108-8001, Japan

Ryusuke Egawa
Cyberscience Center, Tohoku University, Sendai, 980-8578, Japan

Hiroaki Kobayashi
Graduate School of Information Sciences, Tohoku University, Sendai, 980-8578, Japan
Cyberscience Center, Tohoku University, Sendai, 980-8578, Japan

Received: February 15, 2016
Revised: May 6, 2016
Accepted: July 9, 2016
Communicated by Hiroyuki Takizawa

Abstract

Since recent scientific and engineering simulations require heavy computations with large volumes of data, High-performance Computing (HPC) systems need a high computational capability with a large memory capacity. Most recent HPC systems adopt a parallel processing architecture, where the computational capability of the processors is increasing, however, the performance of the memory system is constrained. The bytes per flop (B/F), which is a ratio of the memory bandwidth to the flop/s, for the HPC systems have been reduced with the evolution of the HPC systems. To fully exploit the potential of the recent HPC systems, and to meet the increasing demand for large memory, it is necessary to optimize practical scientific and engineering applications, considering not only the parallelism of the applications, but also the limitations of the memory subsystems of the HPC systems. In this paper, we discuss a set of approaches to optimization of the memory access behavior of the applications, which enable their executions with improved performance on the recent HPC systems. Our approaches include memory optimizations through memory footprint controlling, restructuring of data structures for active elements, redundant data structure elimination through combined calculations and optimized re-calculation of data. To validate the effectiveness of our approaches, a plasmonics simulation application is evaluated on vector platforms NEC SX-ACE, NEC SX-9, and Intel Xeon based platform NEC LX 406-Re2. By applying our approaches to the implementation, the memory usage of the plasmonics simulation application can be reduced up to nearly 1/71 of the original, and its execution can be possible on a single node of a distributed parallel system with smaller memory capacity. The optimization results in 1.14 times faster execution on SX-ACE and 1.81 times faster execution on LX 406-Re2.

Keywords: Memory Management, High Performance Computing, Software Performance

1 Introduction

Scientific research is being carried out with the aid of computer simulations for many years. Over the years, the field of scientific research has observed a vast evolution of hardware and application code-developing styles. The scientific applications which have witnessed such evolution of hardware are also termed as legacy scientific applications. These scientific applications are usually written by computational scientists, not computer scientists. Hence, their code-developing styles tend to directly depict the algebraic equations and the theory of the physical phenomenon that the application simulates. They are designed to be well-structured from theoretical perspective, which enables easy code understanding. Such application designs involve heavy computations on large datasets and are also subject to heavy memory usage with excess memory operations. For some application designs, these excess memory operations may also be redundant in nature. In order to fully exploit the capability of the recent HPC systems, it is necessary for the existing scientific applications to be optimized with respect to the recent hardware trend.

According to the general trend, the mainstream of HPC systems has been dominated by the massively parallel processing systems [1]. The bytes per flop (B/F), a ratio of the memory bandwidth (Bytes/sec) to the computational performance (Flop/s) has been reduced with newer hardware designs [2]. It is shown in Figure 1 that B/F's of HPC systems began reducing around the year 2010. For example, in the case of NEC SX series between SX-9 developed in 2007 and SX-ACE in 2013, the B/F has decreased from 2.5 to 1.0 [3].

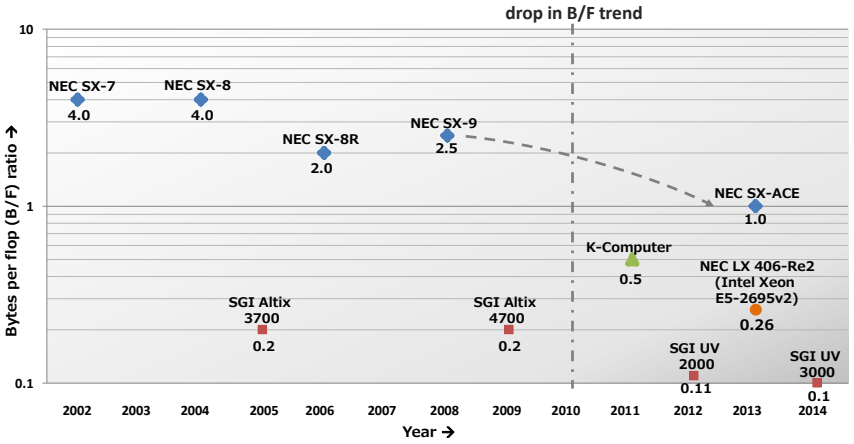

Figure 1: Trend in B/F Ratios of HPC systems.

Considering the above mentioned constraints of evolving hardware trends and established HPC application designs, we encounter a major issue in application code development. Many applications demand large working datasets not only to achieve accurate outputs from a theoretical perspective, but also to extract higher performance from HPC systems. Such applications with a requirement of large memory capacity per node are referred to as memory intensive applications. It is observed from Table 1 that the memory capacities per node of recent top 10 HPC systems vary from 16 GB to 128 GB [4]. Fat node configurations of these systems may have a larger memory of around 1 TB memory per node, however, for most recent HPC systems, the per node memory is less than several tens of gigabytes. As in the case of NEC SX-9 and NEC SX-ACE, the memory capacity of a single node has decreased from 1 TB to 64 GB [3]. Thus, many recent systems have insufficient memory capacity per node to successfully execute some memory intensive applications, such as the target application for this study. Moreover, the demand for a large memory capacity is expected to go up for many scientific applications in the future [5]. This demand can cause an impact on the power budget and implementation cost for future HPC systems. Therefore, it is necessary for scientists and engineers to adopt techniques for memory optimization in order to reduce the memory

size requirement for the scientific applications.

Table 1: Memory capacity per node for recent HPC systems.

System Name	Manufacturer	Memory capacity per node		System Name	Manufacturer	Memory capacity per node
Sunway TaihuLight [6]	NRCPC	32 GB		Tianhe 2 [7]	NUDT	88 GB
Titan [8]	Cray Inc.	38 GB		Sequoia [9]	IBM	16 GB
K-Computer [10]	Fujitsu	16 GB		Mira [11]	IBM	16 GB
Trinity [12]	Cray Inc.	128 GB		Piz Daint [13]	Cray Inc.	32 GB
Hazel Hen [14]	Cray Inc.	128 GB		Shaheen II [15]	Cray Inc.	128 GB
SX-ACE [3]	NEC	64 GB				

We explore the opportunity for memory optimization in our study and identify some approaches for effective memory optimization. Reduced memory accesses indicate that the working data set is reduced, facilitating better cache utilization. From the perspective of hardware, since memory chips are driven by high clock frequencies in the recent HPC systems, power consumption of the system also decreases with a fewer number of memory accesses.

This paper discusses a set of memory optimization approaches which can reduce the overall memory usage of the target applications, while providing an executional performance gain to the program. Through our work we also promote the idea of memory optimization as a tool for performance improvement. For evaluation of the approaches, a plasmonics simulation application is used that simulates optical responses of periodic structures. By applying the approaches to the implementation of the plasmonics simulation application, their effects on the memory usage reduction and the executional performance are examined.

The rest of the paper is organized as follows. Section 2 discusses the related work to our study. Section 3 presents the approaches to reducing the memory usage of scientific applications on recent HPC Systems. Section 4 presents an overview of the plasmonics simulation application and shows a case study of our memory optimization approaches using the plasmonics simulation application. Section 5 describes the evaluation of our memory optimization approaches through executions of the plasmonics simulation program on the vector parallel platforms SX-ACE, SX-9 and the Intel Xeon based scalar parallel platform LX 406-Re2. Finally, Section 6 summarizes the paper with the future work.

2 Related Work

Our work on memory reduction for existing algorithms follows some related work. The discussed memory optimization approaches are generally utilized for application development on embedded systems [16], however, our work combines several such approaches and evaluates their effects on recent HPC systems. Panda et al. provided good techniques for memory estimation which are utilized in our work for memory footprint scheduling. Their work also showed some techniques for reducing the memory usage and changing the data layout, which have inspired our techniques for redundant data structure elimination and restructuring of data structures, respectively. They provided efficient techniques for memory optimization, however, they did not discuss the performance benefits of those techniques.

Wang et al. presented memory reduction as an approach to data locality enhancement [17]. They presented algorithms for loop transformations that enabled reduced memory usage and achieved a significant gain in the application's executional performance. They presented an idea that the opportunities for memory optimization existed often because the most natural way to specify a computation task may not be the most memory-efficient. This idea is a major motivation of our

study. Their work showed that loop transformation techniques enable better cache usage and improves temporal data locality.

Miwa et al. have addressed the issue of memory wall by replacing delinquent load with lines of code that regenerate the load values [18]. Our technique for data re-calculation also works on a similar principle by leveraging the high computational power of the systems and reducing the number of accesses to the main memory. Our work utilizes the principle of re-calculation and achieves reduced memory size as well as increased performance.

Wang et al. presented the idea that memory usage reduction improves the data locality, which enables increased performance on hierarchical memory systems [19]. The idea of loop transformations has also been presented as a tool for performance enhancement through analysis of the complexity of loop fusions [19].

In our study, we have worked along a similar idea of code transformations for memory usage reduction. In contrast to the idea of loop fusion from [20] and [21], we have utilized loop exchange to improve the stride access for better cache utilization, inducing better performance. In one of our approaches, we have replaced the array references with scalar variables as Callahan et al. have proposed [22]. The performance evaluations in our study have also been done on vector capable modern HPC systems, providing a more comprehensive analysis of the performance results. With the code transformations, our study also exploits the vectorization capability of the targets for extracting more performance, as opposed to just the scalar evaluations of related studies.

3 Approaches to memory access optimizations of applications

This study targets the memory optimization of the existing scientific applications for the recent HPC systems. For effective memory optimization, we propose approaches to reduce the number of memory accesses in order to improve the performance of the target applications on recent HPC systems. To realize these approaches, the memory access patterns of the applications are examined with respect to the mathematical calculations and physical phenomena represented in the application codes. In this section, we present several approaches to effective optimization of memory.

3.1 Memory footprint control

For any given application, the duration between an allocation and a deallocation of a data structure is referred to as its memory lifetime, while the duration between its first access and last access is referred to as its memory access period. This idea is presented by Panda et al., as a memory estimation technique [16]. Generally, the memory lifetime of a data structure is longer than its memory access period. Memory footprint control is done by analyzing and rescheduling the memory lifetime of each variable, in order to adapt to its corresponding memory footprint or access period. This is done by tracing the activity of each data structure on the memory from its allocation point to its deallocation point. This activity ensures that the allocation and deallocation timing are aligned with the first and last memory access of each data structure. Timely deallocations can provide room for further allocations, thereby facilitating efficient usage of memory, especially for smaller memory systems.

3.2 Restructuring of data structures

Some applications involve data structures that store massive data on the memory, but only few elements participate in the actual calculations [23]. Furthermore, Panda et al. discuss a methodology

for live variable analysis to get the minimum memory estimate [16]. This approach is to identify the active elements of a data structure and use only them for the purpose of storage and calculation. This drastically changes the calculation flow, and eliminates redundant memory accesses and calculations. This approach benefits the application in reducing not only the size of memory allocations, but also the number of memory accesses, as well as the number of redundant calculation instructions involved in the calculations of the data structures.

3.3 Redundant data structure elimination

Panda et al. have presented a technique for reducing the memory size by eliminating temporary arrays [16]. Many HPC applications are designed in a way so as to retain redundant data structures for calculation purposes. In our approach, the idea has been adopted to eliminate not only the temporary arrays, but also the redundant data structures. This approach is to eliminate temporary and redundant work arrays and buffers that are used to store intermediate results during calculations, using techniques for statement fusion and combined calculations. This approach benefits the application in reducing the number of calculations and the number of memory accesses, providing a performance improvement.

3.4 Re-calculation of data

It is common for applications that deal with huge amounts of data to store the results of various calculations in large arrays, so that redundant calculation operations on the same data can be avoided. This is a good idea if the computational power of the target HPC systems is limited. However, recently with the high computational capability but limited memory capacity of the HPC systems, a cost for storing huge amount of calculated data is much higher than that for re-calculation on demand without keeping calculated ones. Therefore, we need to eliminate the inessential arrays which store intermediate calculation results and use up memory. Such data structures can be eliminated by repeating the calculation operations over the same data throughout the application. Miwa et al. have discussed an idea to replace cache-missed load instructions with a piece of code which regenerates the load value [18]. Similarly, this approach is to reduce the number of redundant memory accesses by re-calculation of the arithmetic operations and elimination of the data structures that store intermediate calculation results.

4 Memory optimization for a plasmonics simulation application

The above discussed approaches are implemented on the plasmonics simulation application. In this section, we discuss the characteristics of the application and provide information for implementation of the approaches for memory reduction, using a suitable example for each approach.

4.1 Overview of the plasmonics simulation application

Plasmonics is a study of the interaction between the electromagnetic field and the free electrons in a metal. The plasmonics simulation application is used to analyze the effect of optical responses of light rays on various elements under varying conditions, as samples of various elements like, Silicon (Si), Aluminum (Al), etc. are studied by measuring their transmittance and reflectance under a varying electromagnetic field. A sample three-layer stacked structure is depicted in Figure 2 below.

The plasmonics simulation application is written in Fortran90. The code structure comprises of the conventional initialization phase and the computational phase. The initialization phase per-

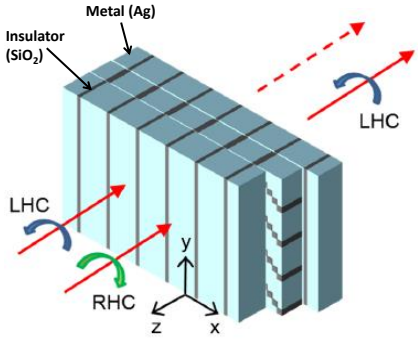

Figure 2: Sample structure for the study of plasmonics [24].

forms the setup of the data structures for input to the computational phase, where the theory is computationally realized and calculations are performed.

This application is designed such that majority of the data structures are allocated together in the initialization phase, and are deallocated together at the end of the program execution. The code of the application is constructed to contain a main loop in which the input data is read to prepare Maxwell's equations, and solve them in the Fourier space in order to resolve the scattering matrix for each periodic cycle. In this section, the algorithm of the application is described in detail.

The application uses Li's algorithm [25] to calculate the scattering matrix using the Rigorous Coupled-Wave Analysis (RCWA) method which relies on finding eigenmodes of Maxwell's equations [26]. The application first translates the discrete values of Maxwell's equations on special points to the summation of unit waves using the Fourier transformation. The equation is represented as below.

$$(FG + \gamma^2)\begin{pmatrix}\langle E_x^{(i)}\rangle \\ \langle E_x^{(i)}\rangle\end{pmatrix} = \overrightarrow{0}. \tag{1}$$

Here, F is a coefficient matrix in the expression of the electric field component in the magnetic field component, G is a coefficient matrix in the expression of the magnetic field component in the electric field component, γ represents the eigenvalue, $\langle E_x^{(i)}\rangle$ and $\langle E_y^{(i)}\rangle$ represent the Fourier coefficients of the electric-field, respectively.

The Maxwell equation for a periodic cycle is represented as a different layer in the Fourier space. For each periodic cycle, Equation (1) is solved to obtain its eigenmode.

$$\left\{\langle E_x^i\rangle_k, \langle E_y^i\rangle_k, \gamma_k^{(i)}\right\}_{k+1}^{2N+1}. \tag{2}$$

A scattering matrix is defined using the eigenmodes for each Fourier layer. The scattering matrix from Equation (3) represents the response against stimulants for the whole body of a generic photonic crystal. The transmittance and reflectance for the element are calculated for varying the electric field, and finally, the outputs (reflected and transmitted wave) against given input (incident wave) are calculated.

$$\begin{pmatrix}\langle \overrightarrow{E}_-^{(M+1)}\rangle \\ \langle \overrightarrow{E}_+^{(0)}\rangle\end{pmatrix} = S\begin{pmatrix}\langle \overrightarrow{E}_-^{(0)}\rangle \\ \langle \overrightarrow{0}\rangle\end{pmatrix}. \tag{3}$$

Here, $\langle \overrightarrow{E}_{\pm}^{(i)} \rangle$ represents a state vector of the electric-field, and S represents a scattering matrix.

This application has been identified as the target application for our study. The field of plasmonics is a very important research area for modern science. The target application is a well-structured scientific application written in order to support ongoing research work. Some characteristics of the code-writing style of this application resembles the code-structure of the legacy applications as described in Section 1. Therefore, this application is appropriate for evaluation of our memory optimization approaches, which are discussed in the following sections.

4.2 Memory footprint control

A simple program flow is depicted in Figure 3, where there are five data structures: *array1*, *array2*, *array3*, *array4* and *array5*, having different allocation sizes. The application assumes a sequential program flow from *sub1* to *sub11* and the vertical axis represents time. In the initial representation, these data structures are allocated together in the subroutine *sub1* and deallocated together in the subroutine *sub11*. The duration between allocation and deallocation of a data structure is referred to as its memory lifetime. Based on the algorithm, the data structures are accessed in certain subroutines. The duration between the first access and last access for each data structure is referred to as its memory access period.

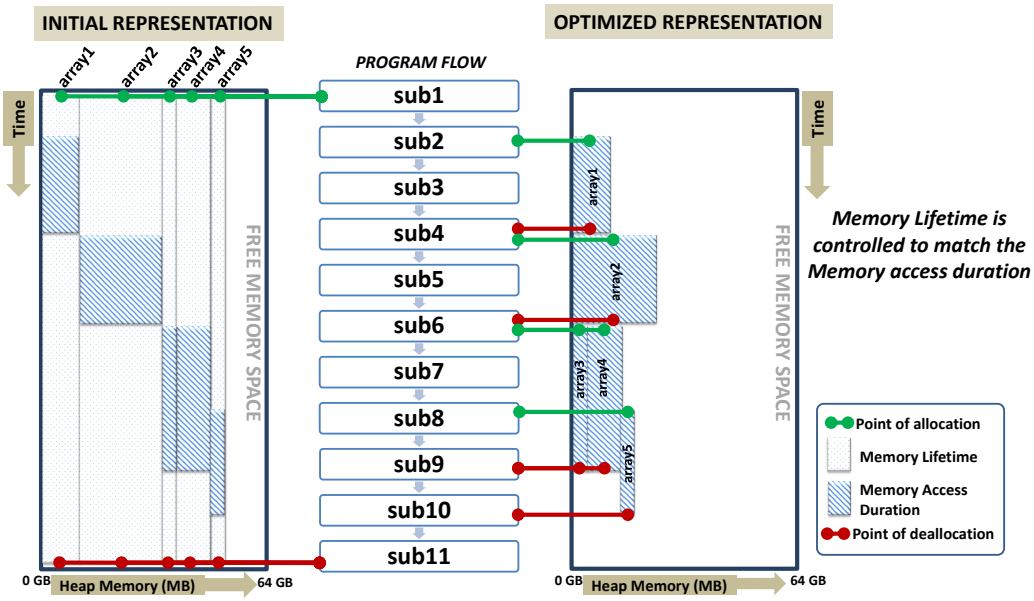

Figure 3: Controlling the memory footprint to save the memory space.

Figure 3 shows that if the memory lifetime of the data structure is larger than its memory access period, then the usage of memory space is not effective. The focus of this approach is on an adjustment of the memory lifetime of each data structure to match with its memory access period, in order to reduce its memory footprint. Therefore, we move the allocation statements for a data structure to the subroutine where the data structure is accessed for the first time. Similarly, the deallocation statements are moved to the subroutine where the data structures are accessed for the last time in the application's lifetime. In the optimized representation in Figure 3, each array is allocated in the subroutine which accesses it for the first time and deallocated in the subroutine where it is last accessed. Hence, the memory allocations are not cumulative like the initial representation. For example, *array1* is allocated in *sub2* and deallocated in *sub4*. In this way, we can ensure that the

overall memory allocation requirement of the application is controlled.

This approach does not change the data structure itself. However, allocation and deallocation timing of each variable might completely change. Since it is a basic optimization scheme for Fortran programs, it does not strictly affect the theoretical implementation. In our study, the data structures involved in all phases of the application have benefit from this approach. The allocations for the data structures for Equations (1), (2) and (3) have been controlled through this. Therefore, its effect is visible throughout the program.

4.3 Restructuring of data structures

This approach for data structure manipulations is used to reduce the calculation complexity for the participating matrices. This approach can be examined using a case of matrix multiplication where the active elements for participation in the calculations are identified.

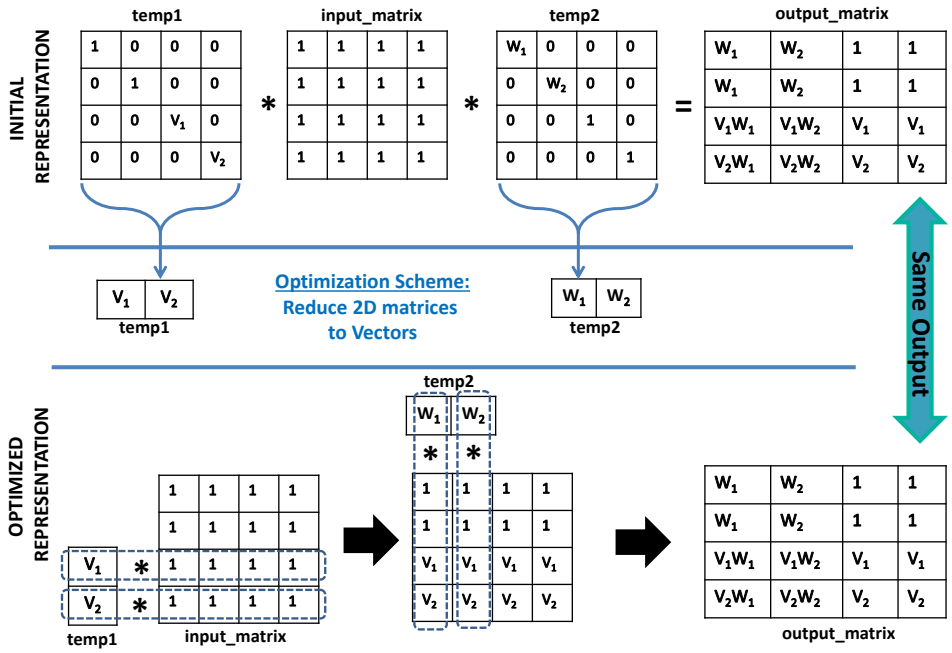

Figure 4: Representation of the approach for restructuring of data structures.

Figure 4 shows a case of matrix multiplication of three 2D arrays: *temp1*, *input_matrix*, and *temp2*. Their product is stored in another 2D array *output_matrix*. The contents of all the three participating matrices are analyzed to examine the least number of elements that can participate in a calculation to produce the same output. It is observed that only half-diagonals of the matrices *temp1* and *temp2* are required to produce the same output as the original calculation. Therefore, *temp1* and *temp2* are restructured to a vector. Using this approach, we are able to reduce not only the memory usage for the calculation, but also the calculation complexity.

In the plasmonics simulation application, the calculations for creation of the matrices F and G from Equation (1) can benefit from the optimization approach of restructuring of data structures. Moreover, the scattering matrix S, represented in Equation (3) can itself be restructured using this

technique.

4.4 Redundant data structure elimination

To eliminate redundant data structures, techniques for algorithm modification and statement fusion are employed. An example for algorithm modification is depicted in Figure 5 where matrices *in_mat* and *identity_mat* (multiplied by the scalar S) are participating in an arithmetic calculation to produce the matrix *out_mat*. We analyze the arithmetic operations, and eliminate the requirement for *identity_mat* by slightly modifying the calculations. As shown in Figure 5, instead of performing a matrix subtraction as depicted in the initial representation, we perform a diagonal-only scalar subtraction, to obtain the same output. This modification ensures that the redundant data structures are eliminated. Here, we are able to replace the matrix with a scalar S and hence, reduce the calculation complexity.

It is a very common practice for scientific applications to involve matrix arithmetic and use identity matrices or unit matrices. The example shown in Figure 5 can be used to further generalize the idea of eliminating identity matrices through algorithm modification for various scientific applications.

In the plasmonics simulation application, scattering matrix (S) from Equation (3) is calculated in steps by using intermediate results from the previous calculations at each step. These steps originally involve redundant calculations which use temporary data structures for storage. This optimization approach has been applied to eliminate such intermediate calculations for the creation of the scattering matrix.

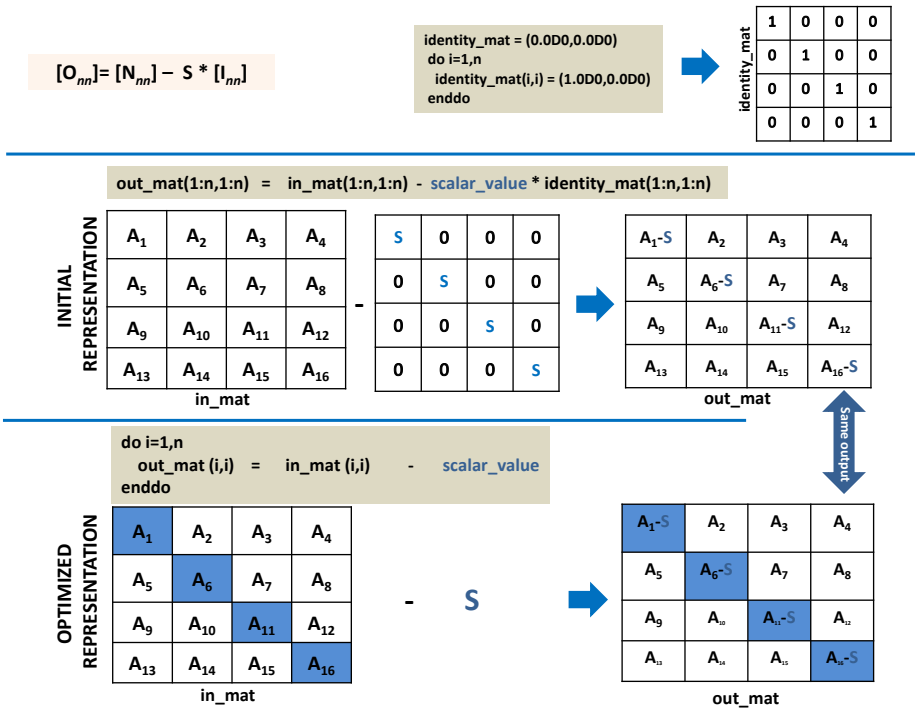

Figure 5: Representation of the approach for redundant data structure elimination.

4.5 Re-calculation of data

We reduce the number of redundant memory accesses for a data structure by on-demand re-calculation throughout the program, as opposed to calculating once and storing the results in a large data structure on the memory. This process is described using Figure 6 where a simple program flow is shown. In the initial representation *Subroutine #1* performs a calculation operation and stores the results onto an array *grd_mat*. *Subroutine #2* reads the array *grd_mat* and performs further calculations.

After applying the on-demand re-calculation approach, the requirement for calculations and storage in *Subroutine #1* is eliminated. Instead, the calculation operations for storage are moved to the *Subroutine #2* without any requirement for storage. In this way, we can eliminate the requirement for large data structures for storage in the program. This approach is utilized in the initialization phase of the plasmonics simulation application, where the matrices are being constructed. Due to the structure of the program, some of these matrices are very large in size. Elimination of these large data structures by re-calculation has a significant effect on the application's memory usage.

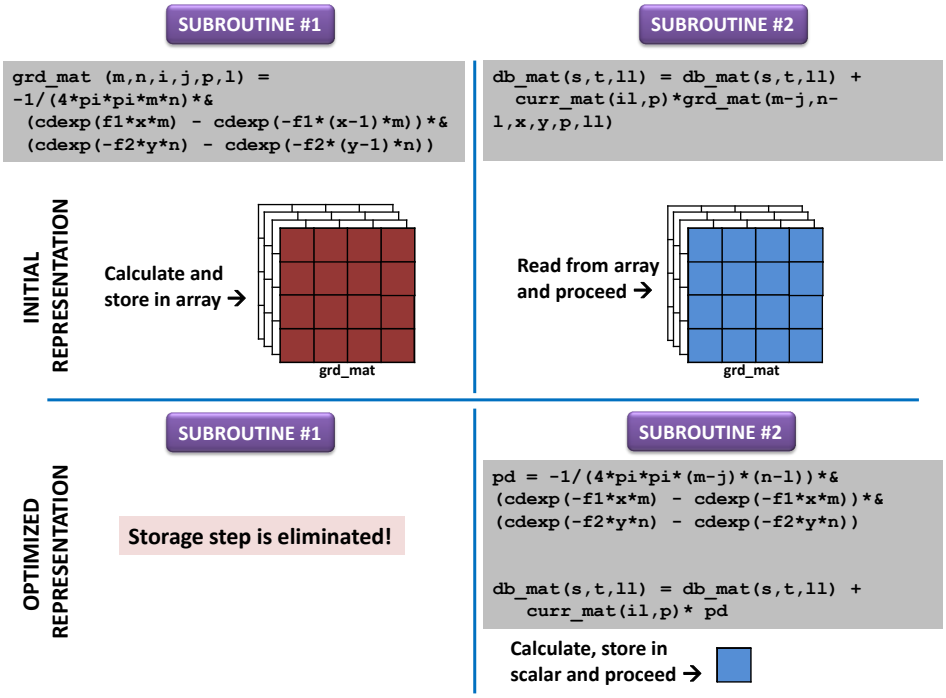

Figure 6: Representation of the approach for re-calculation of data.

This approach can also improve the possibility of automatic code optimizations by compiler. Figure 7 depicts a sample code structure to illustrate this phenomenon. Figure 7(a) shows the original code and Figure 7(b) shows the optimized code with the re-calculation methodology implemented. In Figure 7(a), array *grd_mat* stores the results of previous calculations, so that the program can re-use it for further calculations. For example, the compilers of NEC SX systems attempt automatic code optimization through the method of loop interchange for best cache utilization through short strides of array references. When compared with the p-loop, the l-loop shortens the stride of array references better. Therefore, the l-loop is auto-vectorized by the SX compiler. On the other hand, in Figure 7(b), instead of using the array *grd_mat*, the values are obtained by repeated calculations in every iteration, thereby eliminating the requirement of storage array *grd_mat*. In Figure 7(b), when the SX compiler attempts automatic optimization, the loops are interchanged differently. Now

the x-loop is auto-vectorized by the SX compiler, because the x-loop shortens the stride of array references better than the p-loop. For this application, since the loop length of the x-loop is much larger than that of the l-loop, the average vector length in Figure 7(b) is longer than that in Figure 7(a) for the small dataset. Furthermore, since array *tbl* is referred by the loop index of the x-loop, the whole array *tbl* is stored on the cache because of the automatic loop interchange. Therefore, this technique also improves the data locality and facilitates for better cache utilization on the SX systems as well as the Intel Xeon system.

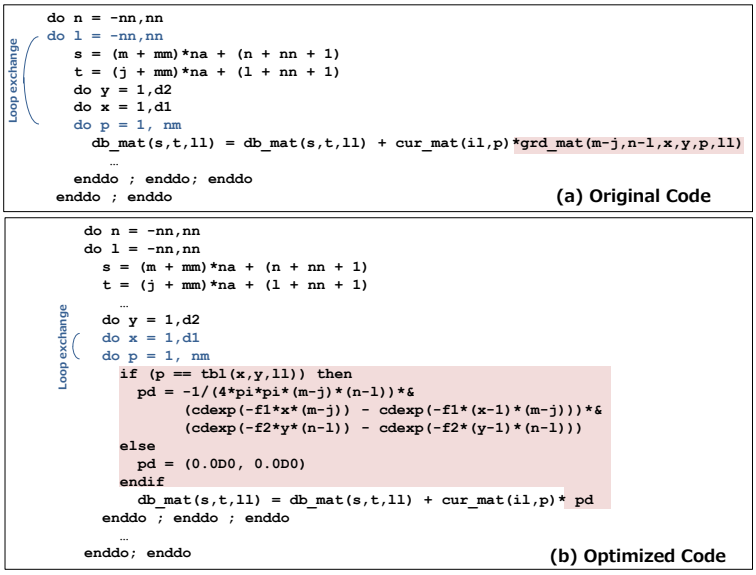

Figure 7: A kernel loop of the plasmonics simulation application: (a) Original code and (b) Recalculated code.

5 Performance Evaluation

To examine the effectiveness of our approaches, the performance of the plasmonics simulation application is evaluated in this section.

5.1 Experimental Setup

The application is executed on three computing platforms: SX-9, SX-ACE, and LX 406-Re2. We evaluate the memory consumption and its performance per-process. Table 2 shows the hardware configuration of the evaluation targets.

SX-9 and SX-ACE are vector parallel platforms, and a node of the SX-9 system contains 16 CPUs with a shared memory of 1 TB capacity [27], while a node of the SX-ACE system consists of one CPU and a main memory of 64 GB. The processor of SX-ACE has four vector architecture cores where each core has a large vector on-chip cache, named Assignable Data Buffer (ADB). The capacity of ADB is 1 MB on each core [28]. Therefore, computational performance of SX-ACE can be extracted better by effective utilization of ADB [3]. The vector length of both SX-9 and SX-ACE is 256 words [29]. The vector operation ratio and average vector length for an application are examined for the plasmonics simulation application to measure how effectively it uses the vector capability of the underlying architecture.

LX 406-Re2 is an Intel Xeon (E5-2695v2) based scalar parallel platform with two CPUs per node and 12 cores per CPU. It has a 128 GB main memory, which consists of two memory subsystems of 64 GB each. The two memory subsystems are connected through two CPUs. The capacity of L3 cache is 30 MB. This target is also used in order to examine the effectiveness of the memory optimization techniques on scalar architectures.

Table 2: Hardware Configuration.

System	Node		CPU		
	Number of CPUs	Memory Capacity (GB)	Perf. (Gflops)	Number of Cores	Cache Capacity
SX-9	16	1024	102.4	1	256 KB
SX-ACE	1	64	256	4	1 MB x 4 cores
LX 406-Re2 (E5-2695v2)	2	128	230.4	12	30 MB (L3)

As mentioned earlier, the memory usage and performance of the plasmonics simulation application are input data dependent. Table 3 shows the parameters for the experiment.

Table 3: Input Dataset parameters.

	Small Dataset	Large Dataset	Huge Dataset
Number of Grids	315 (15x21)	1225 (35x35)	3465 (45x77)
Number of Layers	5	2	5
Fourier Divisions	205 x 355	300 x 300	205 x 355
Memory Reqmnt. (Original)	36 GB	65 GB	543 GB

5.2 Memory size reduction results

In this section, we first analyze the effect of the memory optimization on the overall memory size of the application and then the effects of each approach are examined individually.

We evaluate the application's executional capability on the small memory HPC systems like SX-ACE, by approximating the pre-optimization memory usages for each input dataset. Since the large and huge datasets demand a requirement for memory sizes that are larger than the available physical memory of SX-ACE, their pre-optimization versions cannot execute on SX-ACE. Here, we observe that their original memory requirement is nearly 65 GB and 543 GB, respectively, by executing the large and huge datasets on SX-9.

The overall memory size reduction after the optimization is shown in Figure 8. The memory size has reduced to 1/71 of the original for the small dataset, 1/15 of the original for the large dataset and 1/12 of the original for the huge dataset. While executions of the small dataset achieve a remarkable reduction in memory usage, we observe that the program executes successfully on SX-ACE for the large and huge datasets with memory usages of 4.25 GB and 44.0 GB, respectively. This shows that the optimized memory usage is well fitted within the available memory capacity of the target system.

Note that each optimization approach has its individual effect on the memory size of the application. Table 4 shows the memory size benefit provided by each optimization approach, when applied individually to the original application. Memory footprint control shows a fair contribution to the overall memory reduction of the application. Restructuring of data structures and redundant data elimination approaches provide a fair contribution to the application's memory reduction when evaluated together. It is apparent that the re-calculation approach is the most effective in this eval-

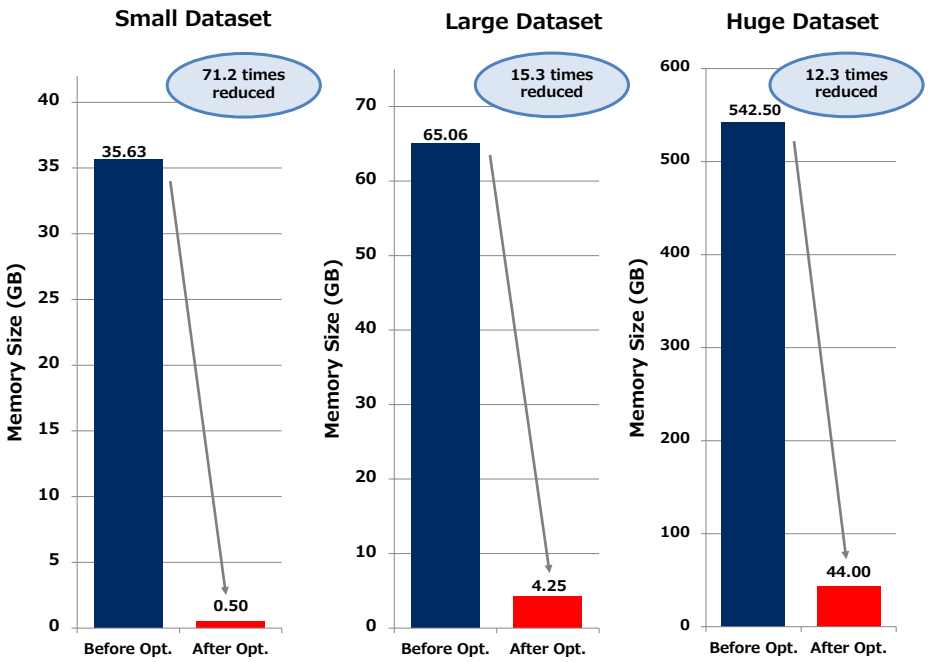

Figure 8: Memory size reduction.

Table 4: Snapshot of memory benefit from each approach.

Approach	Small Dataset	Large Dataset	Huge Dataset
Memory footprint control	0.9 GB	7.2 GB	23.0 GB
Restructuring of data structures	0.1 GB	0.2 GB	10.7 GB
Redundant data structure elimination	0.6 GB	0.3 GB	1.6 GB
Re-calculation of data	34.1 GB	53.1 GB	380.2 GB

uation, since it eliminates the arrays that consume a large portion of the application's used memory.

5.3 Experimental Results and Discussions

The memory optimization approaches result in not only the reduction in the overall memory requirement for the application, but also the improvement of the application's executional performance. We first identify the effect of each approach on the application's performance step-by-step, and later analyze the performance of the overall execution. The step-by-step evaluation follows the pattern from the evaluations in the previous section.

As discussed in Section 3, there are four optimization approaches: memory footprint control (Approach A), restructuring of data structures (Approach B), redundant data structure elimination (Approach C) and re-calculation of data (Approach D). These approaches are evaluated in three steps. First, we evaluate the Approach A individually over the original source code. Next, the Approaches B and C are applied to the original source code, and evaluated together, since there is a close dependency between both approaches in the participating data structures. Lastly, we apply the Approach D to the original source code for evaluation.

5.3.1 Performance of memory footprint control

Optimization Approach A is applied to the original application. Table 5 shows performance results in terms of the vector operation ratio, average vector length, FLOP count, vector load element count and ADB hit ratio remain unaffected by this approach. However, there is a slight degradation in speed caused by the wait time for simultaneous multiple read/write requests made to a single memory bank, known as the bank conflict time.

Table 5: Performance results of Approach A on the program execution — Vector System.

	SX-ACE — Small Dataset		SX-9 — Large Dataset	
	Original	Original + Approach A	Original	Original + Approach A
Execution Time (sec)	771.42	782.79	6000.65	6951.04
FLOP Count	2.32×10^{12}	2.32×10^{12}	2.95×10^{13}	2.95×10^{13}
V. Load Element Count	9.41×10^{11}	9.41×10^{11}	-	-
Avg. Vector Length	27.61	27.61	89.25	89.25
Vector Op. Ratio (%)	93.34	93.34	97.93	97.93
Bank Conflict (sec)	95.55	97.99	4891.62	5967.55
ADB Hit Ratio (%)	61.62	61.62	-	-
B/F Ratio	3.25	3.25	-	-

Table 6: Performance results of Approach A on the program execution — Scalar System.

	Intel Xeon — Large Dataset	
	Original	Original + Approach A
Execution Time (sec)	26085.99	24445.55
Instruction Count	4.90×10^{13}	4.91×10^{13}
Cache Hit Ratio (%)	15.18	14.78

As discussed earlier in this paper, this approach does not affect the data structures directly, but it specifically affects their allocation and deallocation timing. Hence, the FLOP count, the number of vector load elements, and the B/F ratio are almost the same as those of the original and optimized versions of this approach, as shown in Table 5. However, due to the significant change in the memory allocations, the memory access patterns change drastically. In this situation, SX-9 requires effective memory mapping, since the bank cycle time of SX-9 is longer than that of SX-ACE. The modified allocations cause the memory capacity to reduce, but the number of bank conflicts increases. Therefore, the total execution time of the program increases, with a significant performance degradation on SX-9.

This approach is also evaluated on a single-core of the Intel Xeon based scalar platform LX 406-Re2. For the large dataset, the original memory requirement of 65 GB is slightly larger than the 64 GB capacity of a single memory subsystem on a single CPU. Therefore, the execution time of the original version includes an overhead of memory latency between two CPUs on the same node. As shown in Table 6, the instruction count remains similar to the original, because there is no major change in the algorithm or execution order. Hence, the cache utilization is also similar to the original. However, due to the modification in memory allocations, the overhead of memory latency is avoided since all allocations are concentrated on a single memory subsystem with a capacity of 64 GB or lower. This is observed from the reduced execution times for this approach.

5.3.2 Performance of restructuring of data structures and redundant data elimination

Some data structures in the program are restructured by Approach B, while they are used as substitutes for the eliminated redundant data structures by Approach C. Due to such data structures,

the performance effects of both the approaches are closely related to each other. In this step, the performance results for the original program are compared with those of optimized program with those for Approaches B and C combined. These approaches directly affect the algorithm and the design of the data structures, due to restructuring and elimination. The performance effect of this approach is analyzed using the subroutines of the computational phase which have adopted this approach.

Table 7: Performance results of Approach B+C — Vector System.

	SX-ACE — Small Dataset		SX-9 — Large Dataset	
	Original	Original + Approach B+ Approach C	Original	Original + Approach B+ Approach C
Execution Time (sec)	28.91	28.10	961.52	930.49
FLOP Count	7.45×10^{11}	7.06×10^{11}	2.18×10^{13}	2.03×10^{13}
V. Load Element Count	2.77×10^{11}	2.00×10^{11}	-	-
Avg. Vector Length	213.4	201.8	241.30	239.8
Vector Op. Ratio (%)	99.24	99.18	99.57	99.60
Bank Conflict (sec)	7.07	6.29	575.33	524.57
ADB Hit Ratio (%)	48.31	84.33	-	-
B/F Ratio	2.98	2.27	-	-

Table 8: Performance results of Approach B+C — Scalar System.

	Intel Xeon — Large Dataset	
	Original	Original + Approach B+ Approach C
Execution Time (sec)	3367.02	2611.15
Instruction Count	9.93×10^{12}	9.58×10^{12}
No. of Memory Access	7.36×10^{10}	4.54×10^{10}
Cache Hit Ratio (%)	64.57	66.71

As shown in Table 7 for the small dataset on SX-ACE and large dataset on SX-9, this evaluation step shows a performance improvement in terms of the execution time, due to the reduction in the number of memory accesses. The FLOP count is reduced because the multi-dimensional arrays have either been eliminated or reduced to single-dimensional arrays or scalar variables. Due to the reduction in the array sizes, the loop indexes and loop lengths are also reduced. Therefore, there are a slight decrease in the average vector length of the program and also a slight reduction in the B/F ratio, for the small dataset.

The above factors directly affect the vector efficiency of the program. As a result, the vector operation ratio for this approach remains almost the same for all executions. Overall, Approaches B and C provide a reduction in the memory requirement of the application and improve the application's performance. This observation is also verified from the results in the case of the large dataset for SX-9.

Similar behavior is observed on the Intel Xeon based scalar platform LX 406-Re2. As shown in Table 8, the cache-hit ratio has improved slightly. However, there are significant reductions in the number of memory accesses and the instruction count due to the reduced size of the working dataset by restructuring of the multi-dimensional arrays. Due to these reasons, the execution time is reduced considerably, resulting in an overall performance gain from this approach.

5.3.3 Performance of re-calculation approach

In this step, the experimental results obtained by Approach D are compared with those of the original. This approach has provided the largest memory reduction compared to the other approaches, and has hence shown a good improvement in performance. This is shown in Table 9 for the small dataset on SX-ACE and large dataset on SX-9. As discussed in Section 4.5, this approach has its major effect on the initialization phase of the application, and has provided a significant benefit for the memory size reduction, compared to other approaches.

Table 9: Performance results of Approach D — Vector System.

	SX-ACE — Small Dataset		SX-9 — Large Dataset	
	Original	Original + Approach D	Original	Original + Approach D
Execution Time (sec)	750.86	560.96	6245.79	3859.70
FLOP Count	1.57×10^{12}	4.22×10^{12}	7.64×10^{12}	1.53×10^{13}
V. Load Element Count	6.65×10^{11}	4.13×10^{11}	-	-
Avg. Vector Length	20.7	159.6	33.40	132.1
Vector Op. Ratio (%)	91.26	98.74	93.84	98.76
Bank Conflict (sec)	91.10	10.96	5655.07	541.99
ADB Hit Ratio (%)	68.72	99.82	-	-
B/F Ratio	3.38	0.78	-	-

Table 10: Performance results of Approach D — Scalar System.

	Intel Xeon — Large Dataset	
	Original	Original + Approach D
Execution Time (sec)	20444.54.99	12573.78
Instruction Count	2.02×10^{13}	5.33×10^{13}
No. of Memory Access	7.36×10^{11}	1.27×10^{7}
Cache Hit Ratio (%)	0.91	98.81

From Table 9 for the small dataset on SX-ACE, we confirm that this approach has a significant effect on the application's performance. Due to the elimination of large arrays, the number of memory accesses is reduced. However, due to the increase in the number of re-calculations, the FLOP count increases to nearly 2.5x compared to its original.

This approach is also effective for optimizing the memory access patterns of the program. As shown in Table 9 for the small dataset on SX-ACE, the number of vector loads is reduced and the ADB hit ratio improves to 99.82%. This shows that the approach of data re-calculation results in better cache utilization through vectorization of the loops that can offer better stride. Moreover, through better memory access patterns, there is a significant reduction in the bank conflict time.

Due to reasons mentioned in Section 4.5, there is nearly an 8x improvement in the average vector length for the initialization phase for the small dataset on SX-ACE, providing a better vector efficiency to the program. With a higher FLOP count and a higher average vector length, the high vector processing capability of SX-ACE works well, which provides a higher vector operation ratio. These reasons enable a faster execution of the program on the system. With a higher FLOP count and lesser vector loads, there is a significant drop in the B/F ratio from 3.38 to 0.78. Overall, this approach has a significant contribution of 3.5x to the performance improvement of the plasmonics simulation application. It is a beneficial approach for systems with smaller memory capacity per node. This observation is also confirmed for the large dataset on SX-9 from Table 9.

This approach is also evaluated on the Intel Xeon based scalar platform LX 406-Re2. As shown in Table 10, the execution speeds up to nearly 1.6x compared to the original. Due to the re-calculation

of data, the instruction count increases to nearly 2.6x compared to the original and the cache usage improves from nearly no utilization to nearly full utilization. The cache-hit ratio improves from 0.91% to 98.81% by this approach. This approach has proven beneficial for not only vector systems, but also scalar systems.

While this approach provides better memory access patterns and good vector efficiency, it increases the number of calculations for the CPU. With increasing the dataset sizes, the performance benefit of this approach may reduce, since the number of calculations increase the computational performance resulting in a longer execution time.

5.3.4 Performance of the overall application

The performance data of the optimized executions for the small, large and huge datasets on SX-ACE are shown in Table 11. The experimental results obtained using all datasets are represented in Figure 9. Due to the elimination of various redundancies from the original code, the memory consumption for all the datasets is reduced after the memory optimization. Performance gains are observed in terms of execution time due to improvements in vector operation ratio, average vector length and reduction in memory bank conflict time.

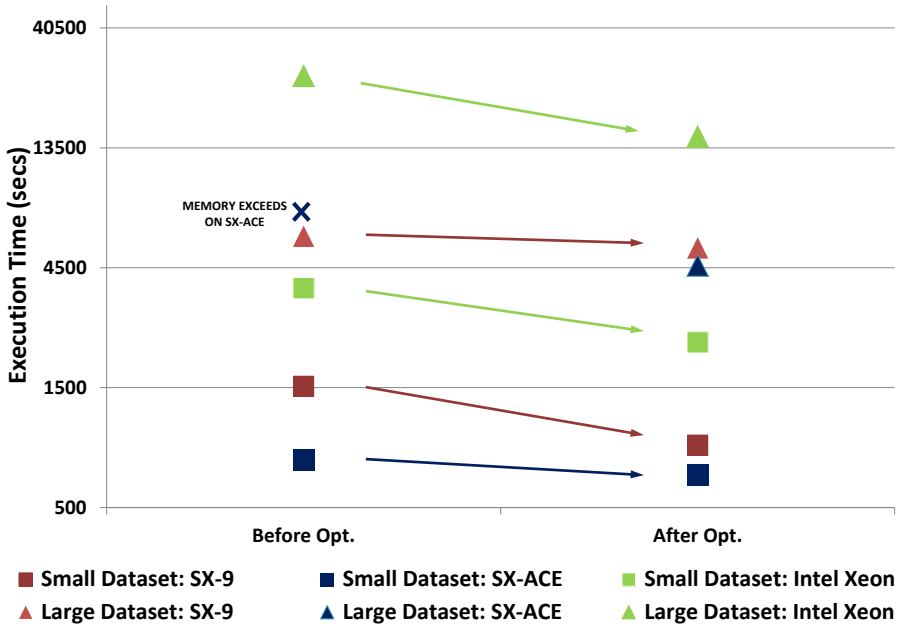

Figure 9: Performance Improvement.

As shown for the small dataset in Table 11, the memory size is reduced remarkably from nearly 34 GB to 512 MB. The memory access pattern is improved after optimization, and the ADB is being used more efficiently. The number of memory loads, represented by the vector load element count, is reduced from 9.41×10^{11} to 5.71×10^{11}, while ADB hit ratio is improved from 61.62% to 85.04%. Better memory access patterns lead to the reduced number of bank conflicts. The average vector length is also improved providing a better vector efficiency. The experimental results show that optimized memory accesses provide a gain in the computational performance of the application. Hence our approaches are effective in reducing the memory load and improving the sustained performance

Table 11: Performance Results on SX-ACE.

	Small Dataset		Large Dataset		Huge Dataset	
	Before Optimization	After Optimization	Before Optimization	After Optimization	Before Optimization	After Optimization
Execution Time (sec)	771.42	674.44		4586.37		106908.52
FLOP Count	2.32×10^{12}	4.82×10^{12}		3.91×10^{13}		1.24×10^{15}
V. Load Element Count	9.41×10^{11}	5.71×10^{11}	Memory	7.65×10^{12}	Memory	2.96×10^{14}
Comp. Performance (MFLOPS)	3005.2	7149.4	overflow	8536.1	overflow	11632.9
Avg. Vector Length	27.61	161.09	on	142.90	on	190.69
Vector Op. Ratio (%)	93.34	97.42	SX-ACE	98.33	SX-ACE	98.27
Memory Size (MB)	36480	512		4352		44800
Bank Conflict (sec)	95.95	7.22		140.47		4569.02
ADB Hit Ratio (%)	61.62	85.04		64.03		63.39
B/F Ratio	3.25	0.95		1.56		1.90

Table 12: Performance Results on LX 406-Re2.

	Intel Xeon — Large Dataset	
	Before Optimization	After Optimization
Execution Time (sec)	28831.45	14961.90
Instruction Count	4.90×10^{13}	9.79×10^{13}
Cache Hit Ratio (%)	15.18	59.48

of the application.

As shown in Table 12 for the large dataset on the Intel Xeon based scalar platform LX 406-Re2, the instruction count is nearly doubled after applying the memory optimization approaches. The cache-hit ratio also shows a drastic improvement, which proves the importance of memory optimization for the system's cache utilization. The performance in terms of execution time improves by nearly 2x compared to the original. These results suggest that our approaches work well for scalar machines as well as vector machines.

6 Conclusions

The bytes per flop (B/F) of the modern HPC systems have been decreasing with newer hardware designs. On the other hand, the memory requirement of the existing and future scientific applications is expected to increase in terms of memory usage and the number of memory accesses. This situation leads to a major issue in their deployment on recent HPC systems. To solve this issue, we have presented approaches to the memory usage reduction of the applications. The proposed approaches to memory optimization have been discussed based on memory footprint control, restructuring of data structures for active elements, redundant data structure elimination by combined calculations and re-calculation of temporary data. From our evaluations, we have confirmed that our proposed approaches are able to not only reduce the memory usage of the application, but also provide a gain in computational performance on recent HPC systems. Moreover, we have also proved that these approaches are beneficial for not only vector platforms NEC SX-9, NEC SX-ACE, but also for the Intel Xeon based scalar platform LX 406-Re2.

Some of the proposed optimizations drastically change the code structure of the applications, however, the code transformation itself is simple and can be automated. Using the discussed optimization approaches, it is possible to improve the readability and maintainability of the source code. The automated source code transformation of the scientific applications and validation of our approaches using other applications are addressed as the future work.

Acknowledgments

The authors would like to thank Mr. Sourav Saha of NEC Technologies India for his continuous efforts for the memory optimization and application evaluation. We would also like to thank Mr. Kenryou Kataumi of NEC Solution Innovators for providing his guidance and expertise required for the investigation. We would like to extend our gratitude to Dr. Masanobu Iwanaga of National Institute for Materials Science (NIMS) for sharing his research to enable our investigation.

References

[1] J. Dongarra, and A. van der Steen,. High-performance computing systems: Status and outlook. *Acta Numerica*, 21:379–474, May 2012.

[2] P. M. Kogge, and T. J. Dysart. Using the TOP500 to trace and project technology and architecture trends. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Nov 2011.

[3] R.Egawa, S.Momose, K.Komatsu, Y.Isobe, A.Musa, H.Takizawa, and H.Kobayashi. Early evaluation of the sx-ace processor. *The poster presentation at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, 2014.

[4] H. Meuer, J. Dongarra, E. Strohmaier, and H. Simon. TOP 500 Lists June 2016. TOP 500 Supercomputer Sites. `http://top500.org`.

[5] Application Working Group. Computational Science Roadmap - Overview. on Feasibility Study on Future HPC Infrastructures, `http://hpci-aplfs.aics.riken.jp/document/roadmap/roadmap_e_1405.pdf`, May 2014.

[6] J. Dongarra. Report on the Sunway TaihuLight System, June 2016.

[7] J. Dongarra. A report on Tianhe-2. `www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf`, June 2013.

[8] Oak Ridge National Laboratory. Introducing Titan. Advancing the era of advanced computing. `https://www.olcf.ornl.gov/titan/`.

[9] Lawrence Livermore National Laboratory. Sequoia. Machine Catalog. `http://computation.llnl.gov/computers/sequoia`.

[10] RIKEN Advanced Institute for Computational Science. Two-page summary on hardware and software of the K computer. system handout. `http://www.aics.riken.jp/en/wp-content/uploads/system_handout.pdf`.

[11] K.Kumaran. Introduction to Mira. `https://www.alcf.anl.gov/files/bgq-perfengr.pdf`.

[12] M.Vigil, D.Doerfler. Trinity Advanced Technology System Overview. `www.lanl.gov/projects/trinity/_assets/docs/trinity-overview-for-web.pdf`.

[13] CSCS Swiss National Supercomputing Centre. Piz Daint. Specifications. `http://user.cscs.ch/computing_systems/piz_daint/index.html`.

[14] HLRS High-Performance Computing Centre. Cray XC40 (Hazel Hen). Technical Description. `https://www.hlrs.de/en/systems/cray-xc40-hazel-hen/`.

[15] Shaheen Supercomputing Laboratory. Shaheen II Get Started. Shaheen II Spec. `https://www.hpc.kaust.edu.sa/sites/default/files/files/public/GetStartedFlyer.pdf`.

[16] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. VANDERCAPPELLE and P. G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, April 2001.

[17] C. Wang, Y. Song, R. Xu, and Z. Li. Data locality enhancement by memory reduction. *ICS '01 Proceedings of the 15th international conference on Supercomputing*, pages 50–64, 2001.

[18] H. Miwa, Y. Dougo, V. M. G.Ferreira, K. Inoue, and K. Murakami. Preliminary Evaluation of the Load Data Re-Computation Method for Delinquent Loads. In *Proceedings of the International Conference on Systems Engineering (ICSEng'05)*, Aug. 2005.

[19] C. Wang, Z. Li, Y. Song, and R. Xu. Improving Data Locality by Array Contraction. *IEEE Transactions on Computers*, 53(9):1073–1084, 2004.

[20] A. Darte. On the Complexity of Loop Fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 149–157, Washington, DC, USA, 1999. IEEE Computer Society.

[21] C. Ding. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. *Doctoral thesis, Rice University*, 2000. `http://hdl.handle.net/1911/19488`.

[22] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[23] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 11, Optimizing for Parallelism and Locality. Addison Wesley, 1986.

[24] M. Iwanaga. Photonic metamaterials a new class of materials for manipulating light waves. *Science and Technology of Advanced Materials*, 13(5):053002, 2012.

[25] L. Li. Formulation and comparison of two recursive matrix algorithms for modeling layered diffraction gratings. *Journal of the Optical Soceity of America A: Optics and Image Science, and Vision*, 13(5):1024–1035, 1996.

[26] M. Iwanaga, and B.Choi. Heteroplasmon Hybridization in Stacked Complementary Plasmo-Photonic Crystals. *National Institute for Materials Science (NIMS)*, 15(3):1904–1910, 2015.

[27] S. Nakazato, S. Tagaya, N. Nakagome, T. Watai, and A. Sawamura. Hardware Technology of the SX-9 (1) - Main System. *NEC Techncical Journal*, 3(4):15–18, Dec. 2008.

[28] S. Momose. *M. Resch et al. editors, Sustained Simulation Performance 2014*, chapter SX-ACE, Brand-New Vector Supercomputer for Higher Sustained Performance I, pages 57–67. Springer International Publishing, 2015.

[29] T. Soga, A. Musa, Y. Shimomura, R. Egawa, K. Itakura, H. Takizawa, K. Okabe, and H. Kobayashi. Performance evaluation NEC SX-9 using real science and engineering applications. *In Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC09)*, 2009.