An Efficient Method for Optimizing PETSc on The Sunway TaihuLight System

Letian Kang[†], Zhi-Jie Wang[‡,#], Zhe Quan[†], Weigang Wu[‡], Song Guo[⊥], Kenli Li[†], Keqin Li[§]
[†] College of Information Science and Engineering, Hunan University, Changsha, China
[‡] School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[#] Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, China
[⊥] Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
[§] Department of Computer Science, State University of New York, United States
kangletian2014@163.com, wangzhij5@mail.sysu.edu.cn, quanzhe@gmail.com,
wuweig@mail.sysu.edu.cn, song.guo@polyu.edu.hk, lkl510@263.net, lik@newpaltz.edu

***Abstract*—High performance computing platforms can bring us great benefits on processing various ubiquitous computing tasks. The Sunway TaihuLight supercomputer is a novel high performance computing platform, which is ranked No. 1 among the TOP500 list in the world. In this paper, we focus on how to optimize the Portable and Extensible Toolkit for Scientific computation (PETSc), running on supercomputers. The main motivations for this study are twofold: (i) PETSc is widely and frequently used in many scientific research fields such as biology, fusion, artificial intelligence, geosciences, etc; and (ii) the current nuclear PETSc does not fully utilize the potential of the Sunway TaighLight system, especially its powerful processor, i.e., SW26010 processor. To achieve high efficiency of PETSc, the central idea of our optimizations is to fully promote the performance of time-consuming and frequently used computation components (e.g., matrix and vector modules). To this end, we propose (i) accelerating kernel codes with computing processing elements (CPEs), in which new compression format and targeted optimizations for vector and matrix operations are devised; and (ii) using more efficient memory access schemes. We have implemented our proposals and evaluated its effectiveness and efficiency through a real world application — Structural Finite Element Analysis (SFEA). We obtain 16∼32 times speedup for a single SW26010 processor. As an extra finding, the results also show a high scalability on over 8,000 computing nodes, i.e., 532,500 cores.**

***Index Terms*—High Performance Computing; PETSc; SW26010 processor; TanhuLight supercomputer**

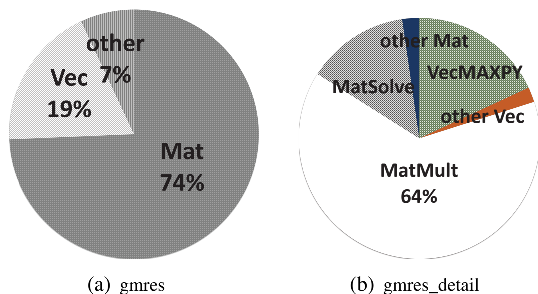
I. Introduction

Ubiquitous computing is a hot research direction in recent years and it has attracted much attention in both academia and industry [1]. Many ubiquitous computing tasks are challenging due to the real-time response requirement [2, 3]. Previous studies have shown that high performance computing platforms can bring us great benefits to processing various ubiquitous computing tasks [4–6]. This essentially indicates that promoting the performance of high performance computing platforms can directly contribute to many other application domains [7–9]. There are numerous high performance computing platforms like Tianhe-2, Sunway TaihuLight, Titan, Sequoia, etc.

Recently, as reported in [10], the Sunway TaihuLight supercomputer is ranked No. 1 among Top500 list in the world. It is one of new generation Chinese supercomputers, designed for large-scale applications in scientific computing and industrial areas [11]. Like other supercomputers, Sunway TaihuLight has also user-friendly interfaces and matched libraries, which can make it easier for programmers to write high performance applications [11, 12]. PETSc (Portable and Extensible Toolkit for Scientific computation) is such a toolkit that supplies many libraries and solvers for various application problems on high-performance computers [13–15]. Particularly, PETSc is widely used around the world and there have been more than 700 publications that used PETSc [16].

Usually, PETSc uses the message passing interface (MPI) for message passing and parallelization among distributed clusters. On general homogeneous systems, pure MPIs adopt the neighborhood collectives, which may lack scalability. In existing literature, there have been some works studying how to achieve strong scaling with PETSc, and the Hybrid MPI/OpenMP optimization has been proposed [17]. Furthermore, the core PETSc team [16] suggests a hybrid MPI-thread method that judiciously uses the MPI shared memory, and thus achieving favourable performance. In contrast, on heterogeneous systems, the performance is more depended on non-MPI elements (e.g., NVIDIA GPGPU and Intel Xeon Phi), which are main components of popular high performance computing platforms nowadays. Typically, to fit PETSc on heterogeneous systems and to achieve high efficiency, some modifications and re-factorizations are needed. It has been intensively studied on how to efficiently extend PETSc to heterogeneous systems and/or optimize the performance of PETSc (e.g., [18–21]). Nevertheless, to the best of our knowledge, little attention has focused on optimizing the performance of PETSc on the new heterogeneous system — the Sunway TanhuLight. This motivates us to study this significant and interesting issue.

Compared against other heterogeneous systems, the Sunway TaihuLight supercomputer uses the new published many-core processor — SW26010. This processor employs a Scratch Pad Memory (SPM) structure, which is generally used in embedded systems. Note that, although the SPM structure is also employed in other processors (e.g., NVIDIA GPGPU [22]). The main features of SW26010 processor are: (i) it uses pure SPM caches and shares main memory among all



(a) gmres (b) gmres_detail

Fig. 1. Time cost distribution of a widely used solver "gmres". (a) Vec:vector, Mat: maxtrix; (b) MatMult: SpMV.

cores; and (ii) it gets a custom programming language and compiling environment, normal C/C++ and FORTRAN codes can directly run on it. Current implementation of PETSc on TaihuLight supercomputer is a straightforward transplant, without fully exploiting the potential of the new system, especially the new published SW26010 processor. As indicated in our experiments, the performance based on such an implementation is not efficient enough (e.g., a MatMult operation takes about 8.41 seconds, whereas our optimization solution takes only 0.35 seconds).

To achieve high efficiency of PETSc, the central idea of our optimizations is to fully promote the performance of time-consuming and frequently used computation components. To dig out such components, we conduct extensive tests on PETSc that contains various algorithms and solvers. An interesting finding is that main computational workloads always and finally fall on the Vector and Matrix modules, and particularly the Sparse Matrix-Vector multiplication (SpMV) takes up most of time. Fig. 1 shows a representative example. The implementation and optimization of SpMV on heterogeneous platforms are challenging, due to (i) the irregularities of sparse matrices, and (ii) the parallelization of SpMV usually involves many tricky issues such as memory access and load balancing. In view of this, we present two simple yet efficient ideas: (i) we accelerate kernel codes with computing processing elements (CPEs), in which new compression format and targeted optimizations for vector and matrix operations are suggested; and (ii) we employ more efficient memory access schemes. To verify the performance of our optimization solution, we implement our proposal and evaluate it based on a real world application — structural finite element analysis (SFEA). Extensive experimental results show us that it is efficient and effective to optimize PETSc on Sunway TaihuLight based on our suggested implementation. To summarize, the main contributions of this paper are as follows.

- We suggest the efficient method to optimize PETSc on TaihuLight supercomputer. The optimization strategies employed by our method is simple yet efficient.
- We implement our method on Sunway TaihuLight supercomputer with more than 8,000 nodes, which is equivalent to 532,500 cores.
- We test the performance of PETSc through a real world application. We obtain 16x∼32x speedup for most PETSc functions in terms of a single SW26010 processor. Meanwhile, we also test its performance via more computing nodes, showing its high scalability.

The remainder of the paper is organized as follows. Section II introduces background. Section III presents our optimizations in detail. Section IV covers the experimental results, and Section V concludes the paper.

II. Background

This section introduces Sunway TaihuLight, PETSc, and SpMV, for ease of understanding the rest of the paper.

A. Sunway TaihuLight System

Sunway TaihuLight is the first supercomputer with peak performance over 100PFlops [10]. The computing system of the Sunway TaihuLight is built using a fully customized integration approach with 4 levels. The entire computing system contains 40 cabinets. Each cabinet consists of 4 super nodes. And each super node includes 256 computing nodes, which are the basic units of the computing system. A single computer node is constituted by one SW26010 processor, 32 GB memory, a node management controller, and some other necessary components.

The SW26010 processor is a brand new processor designed for the new generation of supercomputers. The processor consists of four core groups, each of them has a single management processing element (MPE), 64 computing processing elements (CPEs), one memory controller (MC) and 8 GB RAM. Both MPE and CPE cores get a 1.45 GHz frequency. Each core of the CPE has a single floating point pipeline that can perform 8 flops per cycle, and the MPE has a dual same pipeline.

Note that, the SW26010 processor gets a special feature in cache organization to fit the high peak performance. Each MPE has a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a 256 KB L2 cache for both instruction and data. Each CPE has a 16 KB L1 instruction cache and has no L2 cache. Instead, a user-controlled ScratchPad Memory (SPM) is designed for CPE. The SPM can be configured as either a fast buffer that supports precise user control, or a software-emulated cache that achieves automatic data caching.

B. The PETSc

PETSc consists of many modules, as shown in Fig. 2. All these modules can be functionally divided into two categories [16]. The upper layer mainly includes (i) the KSP module, which provides fifteen Krylov subspace methods; (ii) the PC module, which describes dozens of pre-conditioners; (iii) the TS module, which supports for solving time-dependent (nonlinear) partial differential equations, differential algebraic equations, etc; and (iv) the SNES module, which is a collection of the nonlinear solvers.

In contrast, the lower layer includes (i) the index sets (IS) module, which is responsible for indexing, permutations,

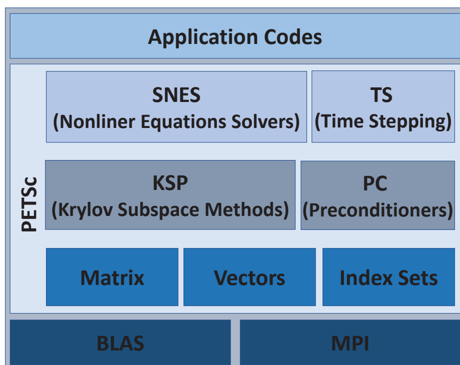


Fig. 2. Organization of the PETSc Libraries

renumbering, etc; (ii) the vectors (Vec) module; and (iii) the matrices (Mat) module. Note that, the Vec and Mat modules provide the structures of vectors and matrices for distributed cluster, data managements of distributed memory architecture, and basic operations of vectors and matrices.

The above hierarchy effectively distinguishes the abstract algebra from computational implementations. The upper layer provides interface and algorithms, while the communication and calculation workloads are handed over to the lower layer, for which some basic libraries like MPI and BLAS are used to assist the execution of communication and computing operations.

C. The Sparse Matrix-Vector Multiplication

Given a sparse matrix A and a vector x, SpMV performs

$$Ax = b. \tag{1}$$

where b denotes the result, which is a new generated vector. The Sparse Matrix-Vector multiplication (SpMV) is a very common operation in high performance computing. In past years many storage formats for sparse matrix is presented to improve the computing performance of SpMV.

The compressed sparse row (CSR) format could be the most popular and general-purpose representation for sparse matrix. The CSR explicitly stores column indices and non-zero values in arrays *indices* and *data*, and the third array *ptr* (i.e., row pointer) allows the CSR format to represent rows of varying length. As an example, consider a sparse matrix A shown in Eq. 2, its CSR format is shown in Eq. 3.

$$A = \begin{bmatrix} 2 & 4 & 0 & 0 \\ 0 & 4 & 0 & 5 \\ 2 & 0 & 1 & 3 \\ 0 & 0 & 5 & 6 \end{bmatrix}, \tag{2}$$

$$\begin{aligned} ptr &= [0, 2, 4, 7, 9], \\ indices &= [0, 1, 1, 3, 0, 2, 3, 2, 3], \\ data &= [2, 4, 4, 5, 2, 1, 3, 5, 6] \end{aligned} \tag{3}$$

Another well-known format is the ELLPACK (ELL). It compresses the non-zero elements along each row, the numbers of elements are made to be same via zero-filling, in order to store the matrix data in column major ordering; and the array for row pointers is omitted. The ELL format is suitable if the rows have similar numbers of non-zero elements. Otherwise, it is inefficient since the number of zero-fillings in this case is much larger. For the matrix A above, Eq. 4 shows its ELL format.

$$data = \begin{bmatrix} 2 & 4 & * \\ 4 & 5 & * \\ 2 & 1 & 3 \\ 5 & 6 & * \end{bmatrix}, offsets = \begin{bmatrix} 0 & 1 & * \\ 1 & 3 & * \\ 0 & 2 & 3 \\ 2 & 3 & * \end{bmatrix} \tag{4}$$

Another intuitive storage scheme is the coordinate (COO) format. It uses *row*, *indices*, and *data* to store the row indices, column indices, and values of the non-zero entries, respectively. Eq. 5 shows the example.

$$\begin{aligned} row &= [0, 0, 1, 1, 2, 2, 2, 3, 3], \\ indices &= [0, 1, 1, 3, 0, 2, 3, 2, 3], \\ data &= [2, 4, 4, 5, 2, 1, 3, 5, 6]. \end{aligned} \tag{5}$$

III. Our Method

In this section, we first discuss our method at a high level and then cover our optimization strategies and implementation in detail.

A. Overview

As we mentioned earlier, although some prior work used PETSc on TaihuLight supercomputer, none of prior work focused on optimizing the performance of PETSc on TaihuLight supercomputer. In this case, when the libraries and/or functions of PETSc is called, they are run on MPE, instead of CPE. It is inefficient, as we will show later (in Table I). This motivates us to re-examine the PETSc and develop the optimization method tailored for the new generation supercomputer. The central idea of our method is to fully promote the performance of time-consuming and frequently used computation components. Our method (or optimization) is mainly based on the following two observations: (i) our preliminary study shows us that main computational workloads always and finally fall on the Vec and Mat modules (recall Fig. 1), and so it allows us to focus more of our attention on optimizing these parts; and (ii) we observe that the computing processing elements (CPEs) in SW16010 processor provide most performance of the computing node.

At a high level, we improve the PETSc on TaihuLight supercomputer from three points of view. First, we accelerate vector operations with CPEs. Notice that, matrix operations are essentially also accelerated, since matrix consists of vectors. Second, we suggest a new compression format for sparse matrix. Third, we employ more efficient memory access scheme. In what follows, we present the details of our method.

B. Accelerating Vector Operations with CPEs

The vector operations are essential part of all solvers. To accelerate vector operations with CPEs, we should connect vector operations to CPEs. There are two ways. (i) For functions created manually, we need to use the interface,

such as `athread_spawn(·)`, in `athread` library to active it; here the `athread` library is specially designed for CPE programming. (ii) For PETSc operations that can exactly match the standard interfaces in the "BLAS (or swBLAS)" library provided by the compile environment, we can directly use the standard interfaces to active them. For example, "VecAXPY"operation is a vector operation (in PETSc) that performs $y \leftarrow Ax + y$, and it exactly matches the BLASaxpy interface. In this case, we can accelerate the vector operation "VecAXPY" with CPEs by using a simple call. It is worth noting that, although some vector operations (like VecAXPY) in PETSc exactly match the standard interfaces, and the optimization like the above is easy to implement. Yet, such an optimization does not make much improvement in terms of the overall performance. This is because these operations like VecAXPY take little part of total running time, as we will show later (in Table I of Section IV).

To get much more improvement, we analyze the vector operations and available library interfaces in a deeper level, and optimize some operations that are time-consuming, and approximately but not exactly match the interfaces in BLAS (or swBLAS) library. Generally, there are two common cases: (i) the PETSc operation changes input(s) and/or output(s) of the standard interface; and (ii) the PETSc operation repeatedly calls some standard interface. We next use two representative examples to illustrate how we achieve the optimizations for these time-consuming operations.

Algorithm 1 VecMAXPY with swBLAS

Input: Parameter array a; arrays $x[][]$; parameter M; array y; length N.
Output: Array y.
1: **for** $i \leftarrow 1$ to M **do**
2: call BLASaxpy(a[i],x[i],y,N);
3: **return** Array y.

Case (i). One of examples is the VecAYPX operation, which performs $y \leftarrow Ay + x$. In situations like this, using standard libraries needs to finish a disassembly of the operation, as shown in Algorithm 2. This leads to the extra memory migration (see Lines 2 and 4). The memory migration takes about half of the total time of VecAYPX operation. (Other operations like VecWAXPY and VecAXPBYPCZ have the similar time cost).

Algorithm 2 VecAYPX with swBLAS

Input: vector yin; coefficient α; vector xin; length of vector n.
Output: vector yin.
1: alloc array $ztemp$;
2: copy data from xin to $ztemp$;
3: call BLASaxpy(α,yin,$ztemp$,n);
4: copy result from $ztemp$ to yin;
5: free array $ztemp$;
6: **return**

To alleviate this dilemma, "manual" optimization is proposed to further improve the performance. Note that, here manual optimization refers to that, we do not still use the existing interfaces (or functions) like "BLASaxpy" (see Line 3 in Algorithm 2); instead, we develop new interfaces (or functions) in order to make full use of the computation performance of CPEs. Algorithm 3 shows our manual optimization, in which $blength$ refers to the buffer length which is determined by the SPE size (Lines 4-7, 10). The general idea of this optimization is first to distribute workloads among CPEs (Lines 2-3), and then use single instruction multiple data (SIMD) instructions to perform calculations (Line 8).

Algorithm 3 VecAYPX manual

Input: VecAYPX_Arg para.
Output: vector yin.
1: $xin, yin, \alpha, n \leftarrow para$;
2: $mystart = n * getThreadId/threadNum$;
3: $myend = n * (getThreadId + 1)/threadNum$;
4: **for** $i = mystart; i < myend; i+ = blength$ **do**
5: Load $xin[i : i + blength]$;
6: Load $yin[i : i + blength]$;
7: **for** $j = i; j < i + blength; j+ = sizeofsimd$ **do**
8: $yin[i] = simd_vmas(a, yin[i], xin[i])$;
9: Write back yin[i:i+blength]
10: **return**

Algorithm 4 VecMAXPY manual

Input: VecMAXPY_Arg para.
Output: Array y.
1: **for** $i \leftarrow 1$ to N, $i+ = Blocklen$ **do**
2: read $y[i : i + Blocklen]$ to $ty[1 : Blocklen]$;
3: **for** $j \leftarrow 1$ to M **do**
4: read $x[j][i : i + Blocklen]$ to $tx[1 : Blocklen]$;
5: read $a[j]$ to ta;
6: **for** $j \leftarrow 1$ to $Blocklen$ **do**
7: $ty[i] \leftarrow ya * tx[i] + ty[i]$;
8: store $ty[1 : Blocklen]$ to $y[i : i + Blocklen]$;
9: **return** Array y.

Case (ii). One of examples is the VecMAXPY operation, which performs $y \leftarrow A_i x_i + y, i \in [0, M)$. Similarly, we can also optimize it with swBLAS, as shown in Algorithm 1. Nevertheless, we observe that this method invokes "BLASaxpy" interface (or function) M times (Lines 1-2); this is not efficient enough. To further improve the performance, we thus suggest a manual implementation to VecMAXPY. Algorithm 4 shows our method in which $Blocklen$ refers to the size of each "piece". Note that, the basic idea of this method is to divide the array y into pieces which can be loaded in SPM. This way, it can effectively avoid redundant memory accesses. (Other operations like VecMDOT can be manually optimized in the same way.)

C. New Compression Format to Sparse Matrix

As discussed earlier, SpVM (i.e., MatMult) operation is time consuming. An inappropriate compression format easily incurs irregular memory accesses, damaging the performance. In the context of new system, it is urgently important to study efficient compression format for sparse matrix. To fully leverage the new feature of the SW16010 processor, we suggest a new compression format for sparse matrix. In short, the new compression format is a variant of the classic ELL format (recall Section II). Yet, it is different from ELL in the following points at least:

- We compress the non-zero elements along each *column*; we call it Column-ELL (CE for short).
- We use a hybrid manner to store matrix, i.e., one of parts is stored using CE, and the other is stored using another existing format "COO" (see Section II); we call it Column-ELL-COO (CEC for short).

Compared against ELL, the main benefit of Column-ELL is that, the accesses of input matrix is continuous and every row of data matrix corresponds to one element in input vector. Consider again the matrix A in Eq. 2, the Column-ELL format is as follows:

$$data = \begin{bmatrix} 2 & 2 & * \\ 4 & 4 & * \\ 1 & 5 & * \\ 5 & 3 & 6 \end{bmatrix}, offsets = \begin{bmatrix} 0 & 2 & * \\ 0 & 1 & * \\ 2 & 3 & * \\ 1 & 2 & 3 \end{bmatrix}. \quad (6)$$

Nevertheless, the Column-ELL has a flaw. That is, when there are some columns (or even only one) which have (has) more elements than others, there will be redundant zero-filling. To fix this, we thus present Column-ELL-COO (CEC) format mentioned before. More specifically, given a threshold K, the part exceeding K non-zeros in a row is extracted to be stored by COO, and the other part is stored by Column-ELL in order to minimize zero-padding. Consider again the matrix A in Eq. 2, and we set $K = 2$ for simplicity. Then, the CEC format is as follows:

$$COO: \begin{cases} row & = & [3] \\ indices & = & [3] \\ data & = & [6] \end{cases} \quad (7)$$

$$C-ELL: \begin{cases} data = \begin{bmatrix} 2 & 2 \\ 4 & 4 \\ 1 & 5 \\ 5 & 3 \end{bmatrix}, offsets = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 2 & 3 \\ 1 & 2 \end{bmatrix}. \end{cases} \quad (8)$$

With the above compression format CEC, we can easily partition matrix and distribute the sub-matrices to MPEs and/or CPEs. There are two ways.

Manner 1. It partitions matrix and distributes all sub-matrices among MPEs. All of the sub-matrices are stored in CEC format. The CPEs will averagely get a share of the sub-matrix from MPE. The size of sub-matrix is n rows, where n is determined by the threshold K and the size of SPM. For example, if each SPE gets mB SPM, and each number takes fB, then every sub-matrix will get $n = m \times K/(f * 2)$. Then, each CPE will maintain a sub-part of array b and all these sub-bs will be added up finally.

Manner 2. It partitions matrix among MPEs, and then distributes sub-matrices among CPEs. All of these sub-sub-matrixes are also stored in CEC format. The CPEs will uniformly get their own "sub-sub-matrices". The size of sub-sub-matrix is same to that of sub-matrix in Manner 1.

Algorithm 5 Memory_Access_Optimization

```
1: for i ranges from start to end do
2:     Start pre-fetch data for first iteration with DMA;
3:     for j ranges from 0 to bufferSize do
4:         Start pre-fetch data for next iteration with DMA;
5:         Synchronize this iteration with DMA;
6:         Calculate with pre-fetched data;
7:     Synchronize the last iteration with DMA;
8:     Calculate with final data;
9: return
```

D. Memory Access Optimization

As we mentioned in Section II-A, the CPE in the SW16010 processor has no L2 level cache. Instead, it uses a scratchpad memory (SPM) to replace it. The benefit of this design is that, it gives users more chance (or greater authority) to mange the tasks like *memory access*, *data prefetching*, and so on. Note that, for the general cache these tasks are automatically managed by operation system. In our method, we shall fully utilize this feature to further improve the performance.

In brief, our main idea is to use a loop prefetching strategy to hide the data access latency, since we observe that most of operations in PETSc are single instruction multiple data (SIMD) operations. The rationale of the loop prefetching strategy is described as follows. Assume that there are n iterations in solving the equations. Then, in the process of computation of the ith iteration ($i \in [1, n-1)$), we prefetch the data which shall be used in the $(i+1)$th iteration. Note that, it is trivial to know the data to be used in the next iteration, since the calculations for matrix and vector are intuitive and regular.

To implement our method, we adopt the so-called double-buffering mechanism, which is a technique designed to hide memory access latency. Algorithm 5 shows the pseudo-codes, in which DMA is used to support data transmission between SPM and main memory.

IV. Performance Study

A. Experimental Settings

In our experiments, we run SW26010 processor with the CG private mode, which is a general used mode. In this mode, each of the four CGs takes an independent address space. It is appropriate to use this mode, since 8 GB memory space is enough for almost all operations of PETSc. To test the performance, we use a real world application, structural finite element analysis (SFEA), as the test case. The SFEA example

TABLE I
A GENERAL VIEW OF OUR RESULTS

		swBLAS		manual	
	B	T	S	T	S
VecAXPY	0.47	0.03	14	-	-
VecDot	0.18	0.01	16	-	-
VecAXPBYCZ	3.11	0.20	15	0.10	30
VecWAXPY	2.70	0.16	17	0.08	32
VecMAXPY	8.51	0.80	11	0.43	20
VecMDOT	4.51	1.07	4.22	0.43	20
MatMult	8.14	8.14	1	0.35	23.26

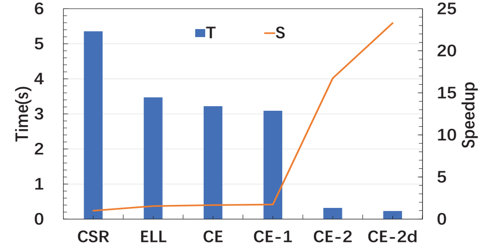


Fig. 3. Performance results in a deeper viewpoint. Six methods are compared: CSR, ELL, CE, CEC-1, CEC-2, CEC-2d.

used in our experiments is extracted from the automotive body design engineering. The coefficient matrix of finite element equations is a sparse symmetric positive definite matrix. We choose the general gmres and bcgs methods, which fit the coefficient matrix best, to solve equations.

Generally, the SFEA consists of five steps: (i) forming the stiffness matrix and the equivalent nodal load array of every unit; (ii) integrating the stiffness matrix and the equivalent nodal load array of each unit to form the overall stiffness matrix and integral nodal load array; (iii) introducing the given displacement boundary condition; (iv) solving the finite element equations; and (v) calculating the unit strain and stress. In our experiments, we intercept the time for Step (iv) to evaluate the performance.

For short, we use the notations "T" and "S" to denote time and speedup, respectively. And the baseline method is shortened as B. The baseline method refers to the current implementation (which does not use any optimizations and/or manually modifications), unless stated otherwise. In our results, all the reported time is in *second*. It is obtained by running 100 times, and then we calculate the average value.

In our experiments we conduct single-node tests and multi-node tests. For single-node tests, the number of CPEs used in our experiments is [2,4,8,16,32,64]. For multi-node tests, the number of nodes is [32,218,512,1024,2048,4096,8192], and for each node all 64 CPEs are used. In addition, we use the median of row lengths of matrix as the threshold K mentioned in Section III-C.

B. Performance Results for Single-node Tests

Table I shows a general view of the running results on a single node, in which the most time-consuming operations are picked out, and some upper operations, which are composed of these basic operations, are omitted. For the baseline method, these typical operations (e.g., those shown in the table) do not use optimizations suggested in the paper.

It can be seen that we gain about $14\times \sim 16\times$ speedup for operations like VecAXPY and VecDot, once we use the swBLAS library, which is a well optimized library designed specially for the SW26010 processor. In addition, the operations, like VecAXPBYCZ, VecWAXPY and VecMAXPY, can also benefit from the use of swBLAS. Particularly, a deeper manually optimization provides us an extra (about $2\times$) speedup. These results essentially demonstrate the effectiveness of our optimizations designed for the vector module.

Interestingly, the operation MatMult seemingly cannot benefit from the use of swBLAS. This is mainly because this operation performs SpVM, in which the optimizations designed for the vector module cannot contribute to SpVM. Nevertheless, a deeper manually optimization provides us much improvement, it gets about $23.26\times$ speedup. This basically demonstrates the effectiveness of optimizations designed for the matrix module.

It is necessary to investigate the performance of the MatMult operation in a deeper viewpoint, since it allows us to observe more clearly the effectiveness of our optimization designed for matrix module. In this set of experiments, we use CSR and ELL provided by PETSc as the baselines. To validate the effectiveness of our optimizations clearly. We implement Column-ELL (CE), Column-ELL-COO (CEC). Particularly, for the CEC format, we implement three versions: CEC format with partition manner 1 (CEC-1), CEC format with partition manner 2 (CEC-2), CEC-2 with double-buffer optimization (CEC-2d). Fig. 3 shows the performance results of these methods. From this figure it can be seen that the proposed method, CE, outperforms two baselines. It demonstrates that the Column-ELL format is more suitable for sparse matrix on the Sunway TaihuLight System. In addition, we observe that three CEC methods are much better than CE. This essentially demonstrates that it is effective to use a hybrid strategy. Nevertheless, it is easy to see that the significant improvement happens when partition manner 2 is used. This implies that irregular memory access significantly limits the performance of the CEC-1, while CEC-2 (i.e., with partition manner 2) can efficiently alleviate this limitation. Our final method (i.e., CEC-2d) further improves CEC-2, and achieves $23.11\times$ speedup (compared against the CSR), demonstrating the superiorities of our method.

Furthermore, we test the parallel efficiency by varying the number of CPEs in the SW16010 processor. In this set of experiments, we also show the performance differences before and after the memory access optimization is used. Fig. 4 shows the experimental results. Theoretically, the ideal parallelization should get $64\times$ speedup when 64 CPEs are used. Yet, the results show us that, for the method without the memory access optimization, its parallelization reaches the upper bound when the number of CPEs is added to 16. Fortunately, when the memory access optimization is used, the performance is

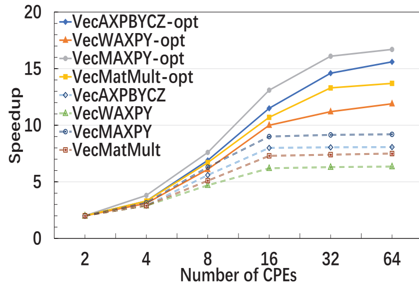


Fig. 4. Performance results before (dotted lines) and after (solid lines) the memory access optimization is used.

TABLE II
RESULTS OF WEAK SCALING

NodesNum	32	128	512	2048	8192
VecMDot	0.82	0.83	0.94	1.00	1.11
VecMAXPY	2.19	2.20	2.20	2.25	2.25
MatMult	3.39	3.54	3.24	3.34	3.39

improved significantly. This verifies the effectiveness of the memory access optimization. It is worth noting that, there is still a gap between ideal parallelization even if our memory access optimization is used. It is interesting and challenging to further improve the parallization, and this could be another independent work, we leave it as the future work.

C. Performance Results for multi-node Tests

Usually, when multiple nodes are used, the scalability would be the focus of attention. In the context of high performance computing there are two common notions of scalability: (i) strong scaling, which is defined as how the solution time varies with the number of processors for a fixed total problem size; (ii) weak scaling, which is defined as how the solution time varies with the number of processors for a fixed problem size per processor. In our experiments, we will examine both of them. All the results reported in this subsection refer to our preferred method (i.e., the proposed optimizations are fully used).

First of all, we discuss weak scaling among nodes. In this set of tests, the input matrix for 32 nodes is a compressed $(10^6 \times 10^6)$ sparse matrix, and when the number of nodes increases, we keep the calculation load for each node unchanged by changing the matrix size. Table II shows the results of PETSc on Sunway TaihuLight System; the most time-consuming operations, VecMDot, VecMAXPY and MatMult are chosen to be listed. Ideally, a perfect weak scaling should be a constant (or fixed) time to the solute problem, independent from the size of nodes. From this table it can be seen that, for the VecMAXPY (or MatMult) operation, the time cost does not change much with the increase of nodes. It basically demonstrates that our method has good weak scaling. It is worth noting that the weak scaling of the VecMDot operation is not as good as other two operations. This is mainly because the VecMDot operation inherently calls function “VecMDot_MPI”, in which an “MPI_Allreduce” operation is triggered. Note that, the “MPI_Allreduce” operation usually incurs that the traffic cost rise with the increase of nodes.

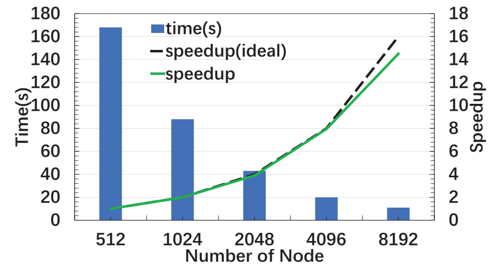


Fig. 5. Results of strong scaling: time and speedup

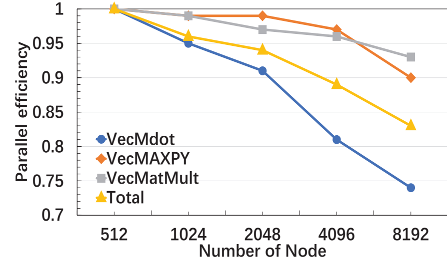


Fig. 6. Results of strong scaling (parallel efficiency)

Fig. 5 shows the running time, and also the speedup in terms of the strong scaling. Here the input matrix is a compressed $(256 \times 10^6, 256 \times 10^6)$ sparse matrix, regardless of the number of nodes. It can be seen that the time cost basically keeps being halved as the number of nodes doubled. Note that, the performance is almost near to the ideal performance in terms of speedup (see the dotted and solid lines). It demosntrates the good scalability of our method from anther point of view.

Furthermore, we also examine the parallel performance based on the strong scaling metric. Note that, here the parallel efficiency is measured by the ration of speedup and the number of nodes. Fig. 6 shows the parallel efficiency by varying the number of nodes. It can be seen that, regarding the parallel efficency, VecMAXPY and MatMult keep beyond 90%, regardless of the number of nodes. Yet, VecMDot drags down the parallel efficiency especially when the number of nodes is large (e.g., 8192). This could be mainly due to the increase of the communication cost, since VecMDot triggers the “MPI_Allreduce” operation (as stated previously). Nevertheless, the overall parallel efficiency is good, since it keeps over 80% (see the yellow line with triangle).

V. CONCLUSIONS

In this paper we present the efficient method to optimize the PETSc on the Sunway TanhuLight System. The central idea of our method is to improve the performance of time-consuming and frequently used computation components. We take full consideration into the features of the new system, and optimize the PETSc from three points of view: (i) accelerate vector operations with CPEs; (ii) suggest a new compression format

for sparse matrix; and (iii) employ more efficient memory access scheme. We implement our method and validate its efficiency and effectiveness through extensive experiments.

Acknowledgment

This work was supported by Grant-in-aid for scientific research from the International (Regional) Cooperation and Exchange Program of National Natural Science Foundation of China (No. 61661146006), the National Key R&D Program of China (No. SQ2018YFB020061), the National Key R&D Program of China (No. 2016YFB0200201), and the National Natural Science Foundation of China (No. 61472453, U1401256, U1501252, U1611264, U1711261, U1711262, U1711263), the National Natural Science Foundation for the Youth of China (No. 61602166).

References

[1] J. Kaye and M. W. Steenson, "Theme issue on histories of ubicomp," *Pers. & Ubiq. Comp.*, no. 3, pp. 553–555, 2017.

[2] H. Gu, K. Kunze, M. Takatani, and K. Minamizawa, "Towards performance feedback through tactile displays to improve learning archery," in *UbiComp/ISWC*, 2015, pp. 141–144.

[3] H. Robert, K. Kise, and O. Augereau, "Real-time wordometer demonstration using commercial eog glasses," in *UbiComp/ISWC*, 2017, pp. 277–280.

[4] D. J. Lillethun, D. Hilley, S. Horrigan, and et al., "MB++: an integrated architecture for pervasive computing and high-performance computing," in *RTCSA*, 2007, pp. 241–248.

[5] C. A. Ardagna, K. Ariyapala, M. Conti, C. M. Pinotti, and J. Stefa, "Anonymous end-to-end communications in adversarial mobile clouds," *Pervasive and Mobile Computing*, vol. 36, pp. 57–67, 2017.

[6] J. B. Abdo and J. Demerjian, "Evaluation of mobile cloud architectures," *Pervasive and Mobile Computing*, vol. 39, pp. 284–303, 2017.

[7] P. Shivam, S. Babu, and J. Chase, "Active and accelerated learning of cost models for optimizing scientific applications," in *VLDB*, 2006, pp. 535–546.

[8] J. Tao, M. Blazewicz, and S. R. Brandt, "Using gpu's to accelerate stencil-based computation kernels for the development of large scale scientific applications on heterogeneous systems," in *ACM SIGPLAN Notices*, vol. 47, no. 8, 2012, pp. 287–288.

[11] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao *et al.*, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[9] J. Carretero, J. Garcia-Blas, and M. G. Neytcheva, "Introduction to the special section on optimization of parallel scientific applications with accelerated high-performance computers," *Computers and Electrical Engineering*, no. 46, pp. 78–80, 2015.

[10] "The top 500 list," https://www.top500.org/lists/2016/06/.

[12] W. Wang, O. Kolditz, and T. Nagel, "A parallel fem scheme for the simulation of large scale thermochemical energy storage with complex geometries using petsc routines," *Energy Procedia*, vol. 75, pp. 2080–2086, 2015.

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, 1997, pp. 163–202.

[14] P. D. Hovland and L. C. McInnes, "Parallel simulation of compressible flow using automatic differentiation and petsc," *Parallel Computing*, vol. 27, no. 4, pp. 503–519, 2001.

[15] L. Carracciuolo, L. D'Amore, and V. Mele, "Toward a fully parallel multigrid in time algorithm in petsc environment: A case study in ocean models," in *HPCS*, 2015, pp. 595–598.

[16] S. Balay, S. Abhyankar, M. F. Adams, and et al., "PETSc Web page," 2017. [Online]. Available: http://www.mcs.anl.gov/petsc

[17] M. Lange, G. Gorman, M. Weiland, L. Mitchell, and J. Southern, "Achieving efficient strong scaling with petsc using hybrid mpi/openmp optimisation," in *ISC*, 2013, pp. 97–108.

[18] D. A. May, P. Sanan, K. Rupp, M. G. Knepley, and B. F. Smith, "Extreme-scale multigrid components within petsc," in *PASC*, 2016, p. 5.

[19] S. Cuomo, A. Galletti, G. Giunta, and L. Marcellino, "Toward a multi-level parallel framework on gpu cluster with petsc-cuda for pde-based optical flow computation," *Procedia Computer Science*, vol. 51, pp. 170–179, 2015.

[20] M. A. Al Farhan, D. K. Kaushik, and D. E. Keyes, "Unstructured computational aerodynamics on many integrated core architecture," *Parallel Computing*, vol. 59, pp. 97–118, 2016.

[21] V. Minden, B. Smith, and M. G. Knepley, "Preliminary implementation of petsc using gpus," in *GPU Solutions to Multi-scale Problems in Science and Engineering*, 2013, pp. 131–140.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.