Title
Solving a trillion unknowns per second with HPGMG on Sunway TaihuLight

Journal
Cluster Computing, 23(2)

ISSN
1386-7857

Authors
Ma, Wenjing
Ao, Yulong
Yang, Chao
et al.

Publication Date
2020-06-01

DOI
10.1007/s10586-019-02938-w

Peer reviewed

Noname manuscript No.
(will be inserted by the editor)

Solving a Trillion Unknowns per Second with HPGMG on Sunway TaihuLight

Wenjing Ma · Yulong Ao · Chao Yang · Samuel Williams

the date of receipt and acceptance should be inserted later

Abstract Benchmarks for supercomputers are important tools, not only for evaluating and ranking modern supercomputers, but also for providing hints for future architecture design. As a new benchmark, HPGMG (High Performance Geometric Multigrid) solves a linear equation set with a full geometric multi-grid algorithm. It involves computation on different scales, data movement with various volumes, global communication and neighbor communication with both large and small messages, etc., and is more correlated to real world applications than traditional benchmarks such as LINPACK. Therefore, it is desirable to examine how well HPGMG can perform on leadership supercomputers such as Sunway Taihulight. Sunway Taihulight, the No. 1 supercomputer in the Top 500 list from June 2016 to June 2018, which uses a specially designed many-core architecture SW26010, is of great interest to the community of high performance computing. With careful analysis and code design, we came up with an efficient implementation of HPGMG on SW26010 processors. We not only employed traditional optimization techniques such as 2.5D partitioning, double buffering, and collective data load, but also introduced a micro-benchmark to help with the choice of optimization direction and parameter tuning. Another contribution is that we proposed a new procedure for the major operations, by granulating and reordering the smooth function and the ghost exchange operation, leading to reduced memory copy and accelerated communication process. Our optimized implementation of HPGMG on Sunway TaihuLight achieved a ground-breaking performance of 1.036×10^{12} Degrees of Freedom per second at the finest level, which is No. 1 on the HPGMG list of Nov 2017.

Keywords HPGMG · Sunway TaihuLight · performance benchmark and optimization · many-core computing

Wenjing Ma
Institute of Software & State Key Lab of Computer Science, Chinese Academy of Sciences, Beijing 100190, China.

Yulong Ao
CAPT and CCSE, School of Mathematical Sciences, Peking University, Beijing 100871, China; Peng Cheng Laboratory, Shenzhen 518052, China.

Chao Yang
CAPT and CCSE, School of Mathematical Sciences & Center for Data Science, Peking University, Beijing 100871, China; Peng Cheng Laboratory, Shenzhen 518052, China. Tel.: +86-10-62757018. E-mail: chao_yang@pku.edu.cn (Corresponding Author).

Samuel Williams
Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA.

1 Introduction

With the Top500 list [28] being updated every year, new supercomputers are emerging rapidly, armed with new architectures and techniques, which keeps changing the spectrum of compilation, imlementation, and optimization of parallel applications and libraries. To evaluate the performance of the supercomputers with various underlying architectures, developing benchmarks for high performance computing has always been an intensively studied topic. Traditionally, the systems are evaluated with the HPL (High Performance LINPACK) [16] benchmark. However, HPL has started to show the lack of capability to map its measurement of a supercomputer to performance of real world applications. For example, applications using differential equations may use sparse data structures which require indirect data access, and are more demanding in memory access and communication [25, 2, 14], and a system optimized for HPL may

not be a good fit for those applications. Therefore, other benchmarks have been proposed for evaluating supercomputers, such as Graph500 [1], HPCG (High Performance Conjugate Gradients) [15] and HPGMG (High Performance Geometric Multigrid) [2,35]. HPGMG was developed based on linear equation solvers in real world applications, which works on a hierarchy of grids, with the correction from solution on coarser levels helping the finest level to converge faster [36]. The HPGMG benchmark includes computationally intense operations on different scales, data movement with various volumes, global communication and neighbor communication with both large and small messages, etc. Such a variety of operations raise challenges for obtaining good performance and robustness on a supercomputer.

The TaihuLight supercomputer, which was No. 1 in the Top500 list from June 2016 to June 2018, provides a powerful computation platform for large scale applications, featuring its many-core SW26010 processors, high speed on-chip networks, and hierarchical fat tree network, etc. Therefore, optimizing HPGMG on TaihuLight is a task of significant value, which helps to provide optimization strategies for real world applications based on similar computation patterns or data structures, and provides hints for future hardware design as well.

Although there has been a good amount of study on optimizing numerical computation on Sunway platform [38,30,13,39,17,18], we are the first to actually investigate the optimzation and performance of HPGMG on TaihuLight. The particular architecture of the TaihuLight platform imposes several challenges to the optimization of HPGMG, including the partitioning of work on the many-core processor, efficient utilization of DMA operations, reducing the overhead of data movement in main memory, etc. In this paper, we provide a thorough solution to optimizing HPGMG on TaihuLight, leveraging the architectural features of the SW26010 processors. We carefully designed parallelization schemes for the major functions, and optimized the data movement with register communication. Furthermore, we proposed a new procedure for the major function, *smooth* with ghost exchange, by restructureing and fusing operations, which boosts performance significantly.

Our main contributions are as follow. First, we implemented and optimized HPGMG on TaihuLight, and achieved 1.036×10^{12} DOF/s (Degrees Of Freedom per second) on 131,072 processes (32,768 nodes), which is the first time to reach the order of 10^{12} in the world. Second, we provided analysis on the bandwidth with different DMA access patterns. Based on that, we designed optimized data access mechanism utilizing register communication, which is a particular feature of the SW26010 processor. Third, we granulated and restructured the operations in the major computation and communication tasks, fusing ghost area processing into the computation kernel on the CPEs, which improved performance by reducing memory copy overhead.

The paper is organized as follows. We introduce the architecture of SW26010 in Section 2 and the HPGMG benchmark in Section 3. Then, we describe our parallelization and optimization methods in Section 4. The experiment results are shown in Section 5, and related work is provided in Section 6. In Section 7, we discuss about the impact of architecture on the performance and optimization selection, as well as the comparison between HPGMG and HPCG. Section 8 concludes the paper.

2 Sunway 26010 Architecture

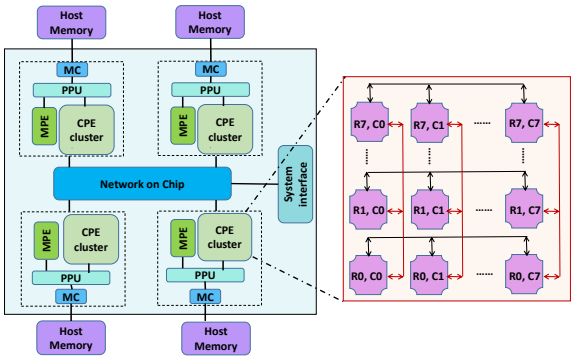


Fig. 1 SW26010 architecture.

The Sunway TaihuLight supercomputer is built with 40,960 SW26010 processors, organized as a fat tree, delivering 125 Pflops (double-precision) aggregate performance [19]. Sunway26010 is a many-core processor comprised of 4 CGs (Core Groups), as shown in Figure 1. Each CG (Core Group) has an MPE (Management Processing Element), a core cluster with 64 CPEs (Computing Processing Elements) connected by NoC (Network on Chip), a PPU (Protocol Processing Unit) and a DDR3 Memory Controller (MC). Inside a CG, the MPE is a fully functional 64-bit core, with a 256-bit vector unit. The 64 CPEs, organized in an 8*8 mesh, are reduced cores which also support 256-bit vector operations. Each CPE is equipped with a 64KB scratchpad memory, called LDM (Local Device Memory), which is software controllable. The CPEs can either access data in main memory directly, or load/store data to/from LDM using DMA. The CPEs in the same row or the same column of the mesh can exchange data in their vector registers via *register communication*.

To execute a program on the CPE cluster, a convenient approach is using OpenAcc, which provides basic code parallelization. Another way is using the Athread library designed specially for Sunway TaihuLight. With Athread, a kernel function is launched by a *spawn* operation on up to 64 threads on the CPE cluster. Then, a *join* operation will wait for all the CPEs to finish their work and then return. As *spawn* is an asynchronous function, the MPE can still work while the kernel is running on the CPEs. Tests with small and empty kernels show that the launch of an Athread kernel is only 5-10 microseconds, which is comparable to that on GPUs and with OpenMP on multi-core CPUs. The Athread library provides a set of operations that can leverage the architectural features of SW26010, such as register communication and DMA between LDM and main memory. In our work, we implemented HPGMG with Athread, taking advantage of the architectural features mentioned above.

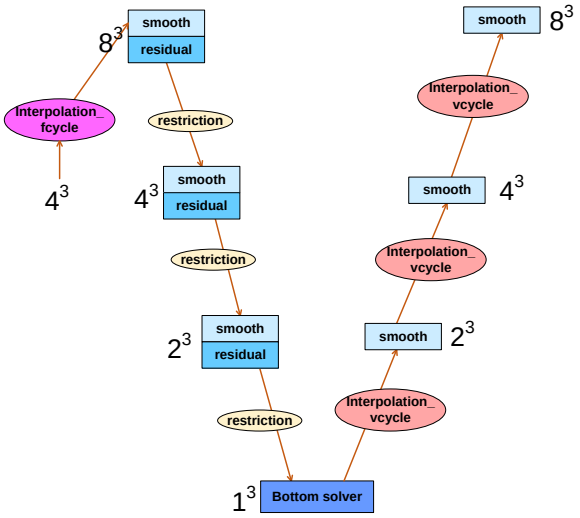


Fig. 2 v-cycle multigrid in the HPGMG computation procedure [35]. This procedure is executed with increasing top level size, until reaching the finest grid. *Smooth* is applied before every *restriction* and *interpolation*.

3 HPGMG Benchmark

The HPGMG (Finite Volume) Benchmark solves a linear equation set with a full geometric multi-grid algorithm, which operates on a cubical Cartesian domain. The domain is organized as a hierarchy of grids, which consists of several levels. The first (finest) level has the whole domain of M^3 elements, where M is the size of each dimension at the finest resolution. Then, every next level gets coarser, reducing the size in each dimension by half. For example, if the finest level has 2048^3 elements, then the second level is of size 1024^3, and the next level has 512^3 elements. The same trend goes on until the coarsest level, which has a small number (cube of 2 or a small odd number) of elements.

The elements in each level (grid) are organized as boxes, each of which has k elements on every dimension. Therefore, a level with grid dimension of m would have $(m/k)^3$ boxes. The boxes in each level are distributed to the processes evenly before the computation starts. In the first few levels which have large boxes, a coarser level is constructed by reducing the size of each box in the finer level. When the box size is small enough, the coarsening of the grid is done by reducing the number of boxes, instead of reducing the box size.

Based on the grid hierarchy, the computation is accomplished by an "f-cycle" multigrid (full geometric multi-grid), which is demonstrated in Algorithm 1, where u is the unknowns to resolve, f is the right hand side, and h is the spacing between elements in each dimension of the grid. The f-cycle starts from the coarsest level (Line 2 of Algorithm 1), and u^h is the initial values of u at the coarest level. Then in the while loop, the spacing of grid elements h decreases in every iteration (Line 5), until reaching the finest grid. Line 4 generates the finer grid by high order interpolation on the coarser grid. "v-cycle" multigrid in Line 6 is the major computation of this algorithm, as defined in Algorithm 2 in a recursive way. It is illustrated by Figure 2 in a more descriptive view, where the numbers denote the number of elements in each level. In a v-cycle, *smooth* (the light blue rectangles) calculates u with Gauss Seidel Red Black (GSRB) (Line 5 of Algorithm 2). Then *residual* (the dark blue rectangles) calculates the residual $f-Au$, as Line 6 in Algorithm 2. After that, *restriction*s (the yellow ovals on the left column) are applied for coarsening each level. The three procedures, *smooth*, *residual*, and *restriction* are invoked repeated until the coarsest level, where the bottom solver is applied (Line 3 in Algorithm 2). Starting from the bottom level, lower order interpolation, *interpolation_vcycle* (the red ovals on the right column), is applied to each coarser level, and followed by a *smooth* operation to generate u on the finer level grid.

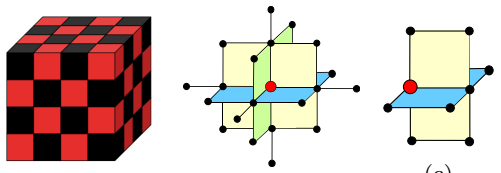
Among all the major operations, *smooth* is the most time consuming one, followed by *residual*, which uses the same stencil operation. Therefore, we focus on the optimization of *smooth* in our work. *smooth* leverages a GSRB (Gauss Seidel Red Black) method, which involves several iterations of updating the correction value of u, using a stencil computation. The out-of-place GSRB

Algorithm 1: f-cycle-multigrid

```
h ← h0;
u^h ← A_h^{-1} f_h /* coarsest solve */
while h>0 do
    u^{h/2} ← I_h^{h/2} u^h /* FMG interpolation */
    h ← h/2
    u^h ← V-cycle(A_h, u_h, f_h)
    e_h ← error(u^h) /* error for convergance test */
end
```

Algorithm 2: v-cycle

```
Function v-cycle-multigrid(A_h, u_0^h, f_h)
if h==h0 then
    return u^h ← A_h^{-1} f_h;
end
u^h ← smooth(A_h, u_0^h, f_h)
r^h ← f_h − A_h u^h
r^{2h} ← I_h^{2h} r^h
u_0^{2h} ← 0
δ^{2h} ← V-cycle(A_{2h}, u_2^{2h}, r^{2h})
u^h ← u^h + I_{2h}^h δ^{2h}
u^h ← smooth(A_h, u^h, f_h)
return u^h
```



(a) (b) (c)

Fig. 3 *smooth* with stencil computation: (a) the red and black cells. (b) elements of u used for one point. (c) elements of *beta_i* used for one point.

operation is done on different colors in each iteration as Figure 3(a) shows. In the first iteration, the red cells in Figure 3(a) are updated (each cell represents one element in the domain), and the new values are put in a temporary vector. Then, in the second iteration, the black cells in Figure 3(a) are updated with the data obtained from the first iteration, and the updated cells are put back to the original vector. The whole procedure of *smooth* requires a series of such iterations, in the interleaved updating fashion mentioned above. The elements in u required for computing one output element is shown in Figure 3(b). From the figure, we can see that one element requires two elements from its neighbor in each dimension and each direction. It implies that the computation of one box requires two layers of data from other boxes on each face. So the overlapped areas between two boxes, called "ghost areas", have a depth of 2 in this algorithm, and thus the boxes are "enlarged" into size $(k + 2 * 2)^3$. The stencil computation also requires the three β parameters, with the pattern of β_i shown in Figure 3(c). β_j and β_k have the same shape, but in different dimensions. The right hand side value and A^{-1} require no ghost areas. The updates are applied on the whole domain, implying that the computation of the boundary of each box would require the latest value of u in neighboring boxes. Therefore, a boundary exchange and building process is required to update the data in the ghost area of each box before every iteration. The official specification of the benchmark requires 6 such iterations in one invocation of *smooth*, which means it sweeps the entire domain and builds the ghost areas of each box for 6 times, leading to the demand for enormous computation resource and memory bandwidth. In the following text, we are going to show how we optimize *smooth*, which is essential for gaining high performance on HPGMG.

Another 3 major functions, *restriction*, *interpolation_fcycle* and *interpolation_vcycle* are also stencil-like operations, with different patterns. *interpolation_fcycle* uses a 125 point stencil to generate 8 points, and *interpolation_vcycle* uses 27 points to generate 8 points. *restriction* writes the average of 8 neighboring elements into one output point. Since those 3 functions occupy a very small portion of the running time, and their optimization is similar to that of *smooth*, we are not going to discuss them in detail. The *bottom solver*, which works on the coarsest level, only takes a negligible time, thus is not discussed either.

4 Implementation and Optimization of HPGMG on Sunway TaihuLight

Based on the architectural features of Sunway TaihuLight, we carefully designed the parallelization scheme and proposed several optimization methods. By applying them to the major functions in the HPGMG benchmark, we were able to harness a good amount of the computing power of the SW26010 processors.

4.1 Parallelization of Stencil Operations

For the parallelization of the major functions, we adopt the z-morton order data distribution scheme provided by the benchmark, which distributes boxes evenly to the processes. With each process mapped to one CG, a box is processed by all the 64 CPEs in the CG [1]. For data partitioning on the CPE cluster, we adopt the widely used 2.5D partition [5,40,26] for the box data

[1] Since one thread runs on one CPE, we use CPE and thread interchangeably in the following text.

and leveraged the double buffering mechanism [22,5, 34]. As shown in Figure 4, the data on an ij plane are distributed onto the CPE cluster, with each CPE processing one sub-block [5]. The entire k dimension is processed in a loop by each CPE. The pseudo code of the kernel function for *smooth* is shown in Algorithm 3, which loads the required data of each sub-block into LDM, then conducts the stencil computation, and stores the result in LDM to main memory. Since the stencil computation of *smooth* has ghost depth of 2 for u, computing one plane requires 5 planes of u, which are shown as the green planes in Figure 4. Among the 5 planes used for the current output plane, 4 of them can be reused in the computation of the next plane. In every iteration, as Line 6 and Line 10 of Algorithm 3 show, one more plane (Plane $k+3$) is being loaded, while the computation is done for the current plane (Plane k), enabling overlapping of memory access and computation, as indicated in the figure. Double buffering is facilitated by using two descriptors, $d1$ and $d2$ in Algorithm 3.

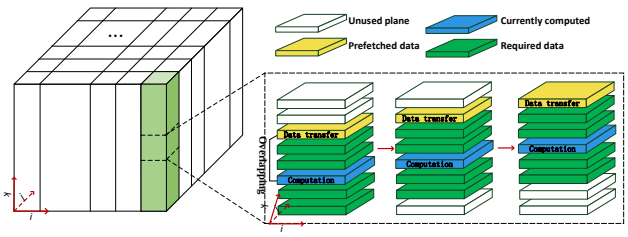


Fig. 4 Data partitioning and double buffering for stencil computation of ghost depth 2. The blue plane is being computed in each step, using the green planes. The yellow plane is the prefetched one, whose transfer is overlapped with the computation.

We let each CPE deal with a 16×8 tile on the ij plane for *smooth* and *residual*. With this tile size, we are already using about 38KB LDM space, which is nearly 60% of the total 64KB (though $beta_i$, $beta_j$ and $beta_k$ requires a ghost depth of 1, we allocate LDM space for them with a ghost depth of 2 on i and j dimensions, for the requirement of collective data loading, which will be explained in the next subsection). Adding the other data and stack space in the LDM, about 65% is used, and doubling the tile size on any dimension will result in overflow of LDM. Therefore, 16×8 is the largest tile size we use. Moreover, a "more square" tile implies less redundant data, thus, we did not use a "less square" tile, such as 32×4. For the other functions, the tile sizes are also chosen in such a way that enables more contiguous data access and less redundant data access.

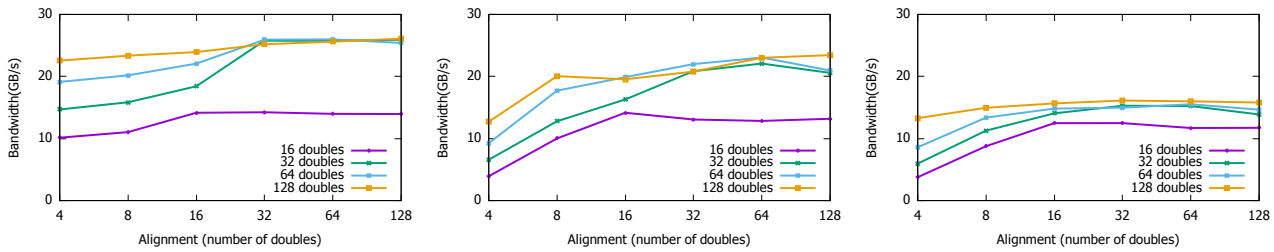
Algorithm 3: *smooth* on a CPE

```
1  dma_desc d1, d2;
2  for each sub block do
3      Prefetch the first 5 planes of x and other arrays
       to LDM;
4      for k in 0 to box_dim do
5          if k&1 == 0 then
6              load Plane k+3 of x and other arrays to
               LDM with d1;
7              dma_wait(d2);
8          end
9          else
10             load Plane k+3 of x and other arrays to
               LDM with d2;
11             dma_wait(d1);
12         end
13         for j in j_low to j_high do
14             for i in i_low to i_high do
15                 x_np = stencil(x, beta_i,
                     beta_j,beta_k, rhs, Dinv, b, h2inv);
16             end
17         end
18         Write x_np of Plane k to main memory.
19     end
20 end
```

The above tile size is used on levels with box size larger than 64^3. For boxes of size 64^3 and 32^3, we use a tile size of 8×8 (instead of 16×8) to make full use of the cores in the mesh. Levels with box size smaller than 32^3 are processed by the MPE.

4.2 Bandwidth Oriented Optimization

As a stencil operation with 6 input arrays, *smooth* is a typical memory bound computation. With the naive data access approach, each CPE loads data required by the sub-block it processes. This approach suffers from two deficiencies. First, each CPE has to load a relatively large ghost area. Second, as a 16×8 sub-block spreads on 8 rows, the data are loaded in short stanza. An important approach to alleviate the memory access overhead is using collective data loading [5]. To facilitate collective data access, several threads form a "thread group". Every CPE loads a few sections in one row, and exchange the data they load with other CPEs in the group, to ensure the same final status as with the naive method. Figure 6 shows this mechanism with a thread group of 4 threads. Each thread loads consecutive sections in the main memory, and then, after replicating boundaries and exchanging among threads, every thread gets their required data. This method helps to reduce the redundant data loaded by the CPEs in a



(a) read (b) write (c) read+write

Fig. 5 Bandwidth of DMA transfer between main memory and LDM of a single CG on a 256^3 box. Each line denotes a different access length. The performance with read+write is below read, and the lines flatten with large enough access length and alignment.

group, and data are loaded in a more contiguous pattern by each CPE.

To determine the size of the thread group, and conduct a systematic analysis on the memory accessing behavior, we designed a micro-benchmark for testing bandwidth, which reads or writes a box of 256^3 double-precision floating-point numbers (without ghost areas) in the main memory through DMA operations. We use 3 parameters to control the DMA operations between LDM and the main memory, namely *access_length*, *alignment*, and *access_pattern*. *access_length* is the number of consecutive double-precision numbers to be loaded or stored in each DMA operation. *alignment* refers to the number of double-precision numbers each DMA operation is aligned to, which can be adjusted by setting the length of j dimension in the box. The starting position of the box is aligned to 1KBs, to ensure that the alignment of data access is determined by the length of j dimension. *access_pattern* is set as one of 3 patterns, namely *read*, *write*, and *read* + *write*. *read* and *write* sweeps the whole box once, leading to memory access of 256^3 double-precision numbers. *read* + *write* is done by issuing one DMA write after one DMA read operation, implying that $256^3 \times 2$ double-precision numbers are accessed in total. Figure 5 shows the bandwidth obtained with various DMA configurations. It is observed that the performance of *write* is a little worse than *read*, and *read* + *write* is even worse than *write*. As the *read* + *write* pattern is unavoidable in the implementation of stencil computation, we look into other aspects for optimization of memory access. Then, we have another important observation, that the bandwidth is better with larger *alignment* and larger *access_length* until reaching a certain threshold. Therefore, we optimize the DMA operations in two directions, based on the performance trend under the influence of those two parameters.

One direction is increasing *access_length*, which can be accomplished by collective memory access, as described above. For *smooth*, 4 arrays are read with ghost areas, and 2 arrays are read without ghost areas. The outputs are written without ghost areas. We adopt the collective access pattern for all the 7 arrays. The collective reading for data without ghost area is similar to Figure 6, while the augmentation is eliminated. Since we use a tile(sub-block) size of 16×8, with 4 CPEs forming one group, to load the entire sub-block, each CPE needs to read and exchange twice. With a different configuration of grouping, the performance would vary. For example, when exchanging data among 8 threads, each thread would load one row with 8 segments. Therefore every CPE reads only one long section for loading a sub-block. However, this configuration for collective loading requires more register communication operations. In the experiments, we test different schemes and select the best one.

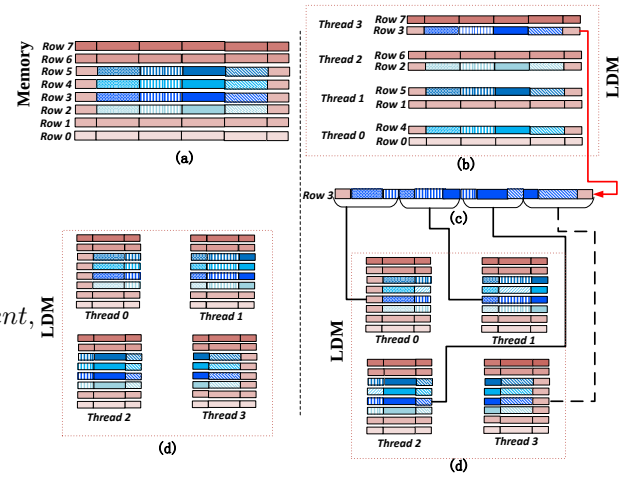


Fig. 6 Exchanging data within a group of 4 threads. The red areas are ghost areas. The left part shows the naive approach, in which each thread loads their own sub-block independently. The right part shows the collective method in three steps. First, from (a) to (b), each thread loads one long stanza. Second, from (b) to (c), Row 3 is used as an example to show how the loaded data in LDM are augmented to add ghost areas. Third, from (c) to (d), we demonstrate the distribution of data in Row 3 within the 4 threads.

Another direction is using large *alignment* by setting a large enough j dimension. For the data that do not require ghost area, the boxes are aligned for the first byte of the inner area. For the three *beta* arrays, the alignment is done for the first byte of the enlarged box, since the computation requires ghost areas in those arrays. As for the correction u itself, we align it for the first element of the inner area. This is because it is both read and written, and from Figure 5, we can see that writing is more sensitive to the alignment. Therefore, we align u according to the requirement of its writing operation, which does not involve ghost areas.

4.3 Fusion of Boundary Processing

Before every sweep of the whole domain in *smooth*, the ghost areas of each box need to be built, either filled with data from neighboring boxes or calculated with data inside the domain. As an invocation of *smooth* includes 6 iterations, the ghost area processing turns out to be a big overhead. We analyzed the performance bottleneck of the boundary exchange procedure, and redesigned *smooth* in a transformed sequence, utilizing a fused procedure on CPEs.

Before going to the optimization method, we explain what happens in the ghost area processing of *smooth*, including two functions, *exchange_boundary* and *boundary_calculation*. For every box, the ghost areas include 6 faces and 12 edges, with corners not required, and we call each ghost area a "ghost block". *exchange_boundary* fills in the ghost blocks with data from other boxes, either local boxes that is owned by the same process, or remote boxes that require inter-process communication. And *boundary_calculation* calculates the ghost blocks with data inside the box. Figure 7(a) illustrates the operations in two iterations of *smooth*. *exchange_boundary* accomplishes the exchange of ghost blocks in 3 steps. In the first step, the ghost blocks to be sent to other processes are copied to "send buffer"s, corresponding to the "pack" operation in Figure 7(a), and asynchronous MPI *send* is invoked. Asynchronous MPI *receive*s are also launched to accept messages in the "receive buffer"s. In the second step, while waiting for the ghost blocks from other processes, ghost blocks from local boxes are copied to the appropriate ghost areas of the local destination boxes, which is the "local copy" operation in Figure 7(a). Finally, after all the remote ghost blocks have arrived at the receive buffers, the ghost blocks are copied to their corresponding ghost areas in the local boxes, referred to as the "unpack" operation in Figure 7(a). *boundary_calculation* is done only for boxes on the boundary of the whole domain, which builds the ghost blocks using data inside the domain.

In this process, we found that most of the time in boundary exchange is spent on "pack", "unpack", and "local copy", which copy data between communication buffers (receive buffers and send buffers) and the box, as well as between local boxes. A simple way to reduce this overhead is implementing *exchange_boundary* and *boundary_calculation* on the CPEs, and invoking them before launching the kernel function of *smooth*. This approach helps to reduce the time of data copy, because the memory bandwidth is higher for CPE with the DMA data transfer. However, it still involves data movement between the communication buffers and the boxes in main memory. To further reduce the overhead of memory copy, we design a new execution sequence for *smooth*, fusing *exchange_boundary* and *boundary_calculation* into the *smooth* kernel function.

The restructured *smooth* with boundary fusion is shown in Figure 7, where the original procedure in Figure 7(a) is transformed to Figure 7(b), showing two iterations in *smooth*. This transformation requires two major rearrangements of operations in every iteration. One is moving the "pack" operation to the kernel of the previous iteration, which is done by writing the data in ghost blocks of the current sub-block into the send buffer with a DMA operation, after the computation of one sub-block in a plane. This is illustrated by the red stars in the figure, where the star in Figure 7(a) denotes packing operation in iteration $N + 1$. In the transformed code, the "pack" operation is accomplished by the kernel in Iteration N, which copies the data to send buffers in the computation loop, shown as the star in Figure 7(b). Another rearrangement is moving "unpack" and "local copy" out of *exchange_boundary*, and inserting them in the kernel which runs on the CPEs. Those operations are also fused into the computation of each sub-block, right after the collective loading of the box data, and before the stencil computation. This is demonstrated in Figure 8 conceptually. In the original execution process in Figure 8(a), all of the ghost blocks are copied to the box in main memory before the kernel. In Figure 8(b), the ghost blocks are copied from receive buffers to the LDM in small pieces by each CPE. *boundary_calculation* is also fused into the CPE kernel, as it requires the data loaded from the receive buffer in some cases, and cannot be done before the fused kernel.

4.4 Other Optimization

Because the tile size and register communication patterns may be different for boxes at different scales, we

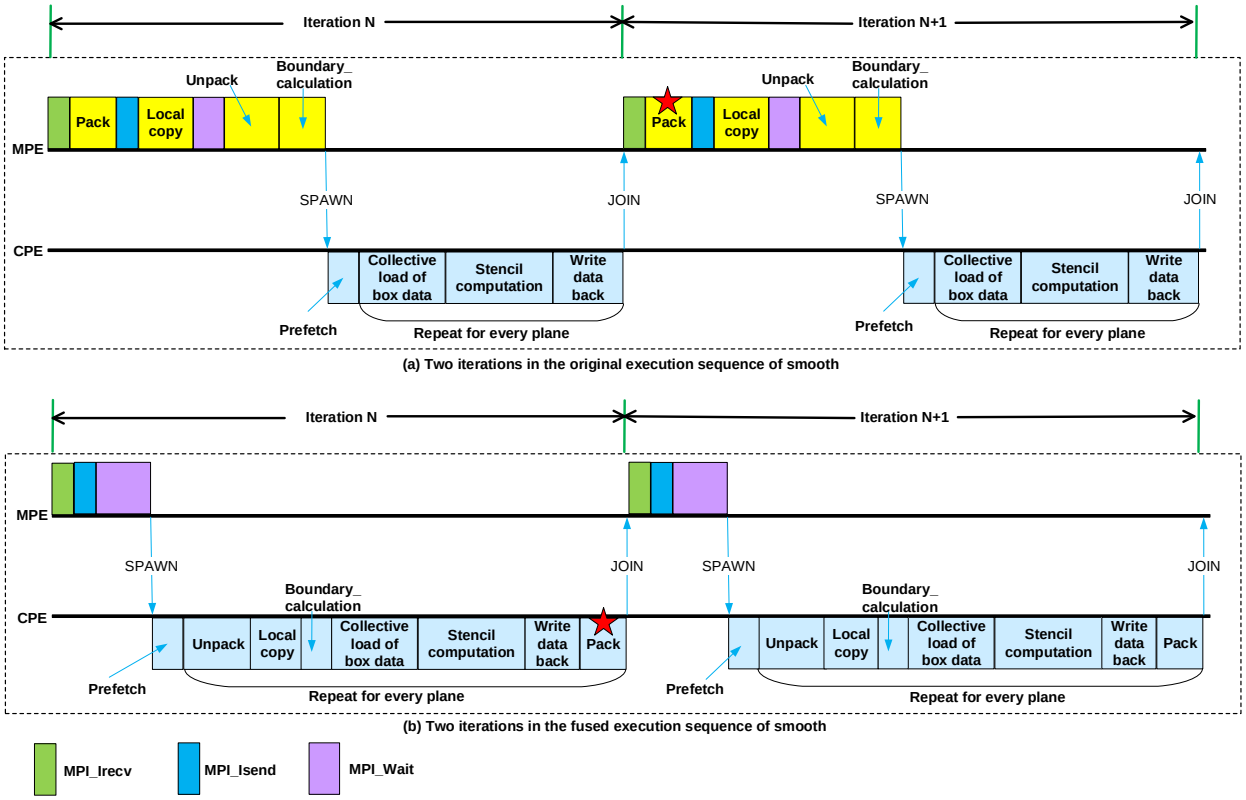


Fig. 7 The execution procedure of smooth with and without fusion. Iteration N and $N+1$ are shown in the figures, with yellow blocks showing operations on the MPE, and light blue blocks showing those on the CPEs. The green, blue and purple blocks are communication operations. The stars imply the same "Pack" operation, but done in different iterations in the two approaches.

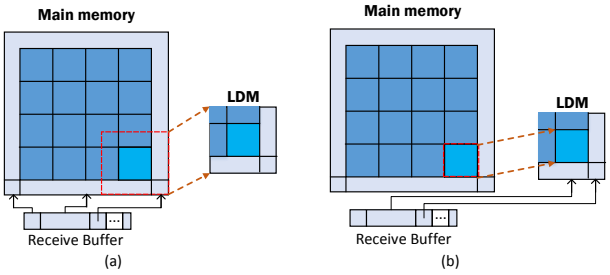


Fig. 8 Fusing boundary processing into the computation kernel. In the fused code, the boundary is loaded from receive buffer in small pieces when they are required by the computation of each sub-block.

keep various versions of kernel functions for different levels. It is better than putting all the cases in a single kernel function, which imposes more branches and degrades the performance. We also interleaved the DMA operations and the other statements in some code segments, which gives the compiler more opportunities to arrange the instructions in a way that overlaps floating-point computation and DMA operations. Additionally, in the collective loading of the box data, loading n rows for each sub-block (in our implementation n is 2 or 3) is done by using n DMA operations, instead of one strided DMA operation. This is a choice based on experiments which show that the latter approach yields better performance.

5 Performance Evaluation

In this section, we show the performance improvement obtained by our optimization techniques, and how the code performs on 8 million cores on TaihuLight. To minimize the overhead of ghost area processing and communication, we make the box size as big as possible. Since the 4 CGs in a node share the 32GB memory, each Core Group can use up to 8GB memory. As we map one process to one CG, according to the data requirement, the largest box size that can be accommodated in one process is 256^3. Therefore, in the following tests, the boxes in the finest level are set to be of size 256^3, and each CG can process up to 4 boxes.

5.1 Performance Gain of Optimization Schemes

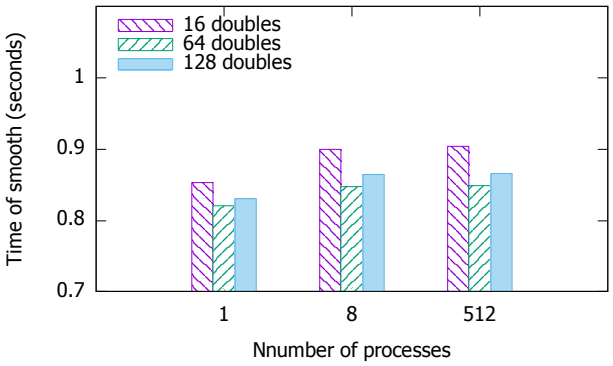


Fig. 9 Execution time of *smooth* with different data exchange schemes for data without ghost areas, showing the tradeoff between larger access_length and more register communication overhead.

We first test the bandwidth-oriented optimizations. In terms of *alignment*, we simply set the j dimension aligned to 128 double-precision numbers, which is large enough for maintaining good performance. We test the different collective access patterns, since there is the tradeoff between larger *access_length* and more register communication overhead. As mentioned above, with the 16×8 sub-block size, a naive loading by each CPE results in an *access_length* of 16 doubles. Collective loading with 4 CPEs and 8 CPEs leads to an *access_length* of 64 doubles and 128 doubles respectively, with more register communication required (*access_length* of 64 doubles requires each thread communicate 96 doubles for one sub-block, and *access_length* of 128 doubles requires communication of 112 doubles). We tested the three *access_length*s and show the results in Figure 9. The code is based on the most optimized version, and the variation is done on the input data without ghost areas. The time of *smooth* is measured by summing all the *smooth* kernel execution time of the finest level in one f-cycle. The legend denotes the *access_length* of the three collective data loading patterns. We can see that the performance gain from 16 doubles to 64 doubles is obvious, which is consistent with the bandwidth tests in Figure 5. The performance gets a little worse from 64 doubles to 128 doubles, because the bandwidths between those two access lengths are almost the same, while the latter has extra overhead of register communication.

In Figure 5, we showed that though the DMA read operation can deliver bandwidth of 26GB/s, the performance with mixed read and write can only achieve up to 16GB/s bandwidth. In our test with the *smooth* code, the running time of one iteration on a single CPE got a bandwidth of about 13GB/s, which means the bandwidth utilization is about 81%. Considering the extra data loaded by the overlapped sub-blocks among CPEs, this bandwidth utilization is very close to that of the code without computation, implying that we have achieved good overlapping of DMA operation and computation.

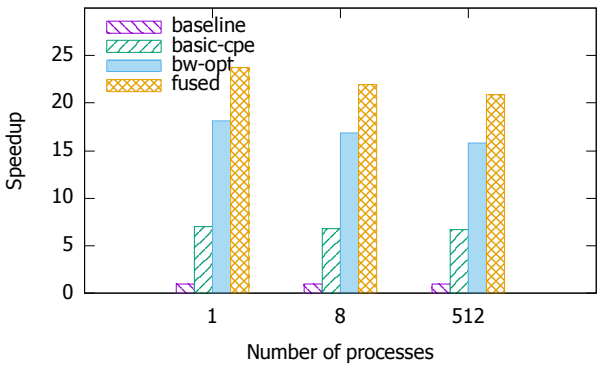


Fig. 10 Speedup of smooth with different optimization techniques applied, compared to the *baseline* version on MPE.

Next, we test the code with different optimizations applied, and show the results in Figure 10. The *baseline* version is a single thread execution running only on the MPE. The *basic-cpe* version uses the CPE in a very simple fashion, which partitions the data on the CPE mesh, and use the LDM for buffering input and output data. The 2.5D partitioning and double buffering is used for this implementation. The third one, *bw-opt*, uses collective data exchange and enlarged j dimension. The fourth one, $fused$, is the version that fuses *exchange_boundary* and *boundary_calculation* into *smooth* and *residual*, which is the most optimized code. The experiments are conducted on 1, 8, and 512 processes, with 1 box per process, using a box size of 256^3. The time is the sum of all the invocations of *smooth* on the finest level, including time for ghost exchange and boundary building. In the three cases (1, 8, and 512 processes), *basic-cpe* got a speedup of 6.6-7× using the simple blocking mechanism on the CPE cluster. With collective and aligned memory access, the speedup of *bw-opt* over *baseline* is 18.1× on one process, and 15.8× on 512 processes. The most optimized version $fused$ achieved a speedup of 23.7× on 1 process, and 20.9× on 512 processes over the *baseline* version. In Figure 10, we can see the speedup of the optimized versions goes down with more processes. This is because the overhead of communication (let us call it $Tcomm$) is higher on more processes, and our optimization is mainly for re-

ducing computation time $Tcomp$. Since the total time of smooth is $Tcomm$+$Tcomp$, a larger $Tcomm$ amortizes the improvement of $Tcomp$. From 1 process to 512 processes, $Tcomm$ grows from 0 to 0.121s, thus the percentage of $Tcomp$ is smaller on 512 processes. However, this trend will be more flattened with more processes. Because the percentage of $Tcomm$ does not grow much on more than 512 processes. For example, $Tcomm$ on 110,596 processes is only 0.133s, just a 10% increase from that on 512 processes. Therefore, the percentage of $Tcomp$ on a huge number of processes has not much difference from that on 512 processes, while the improvement of $Tcomp$ stays the same, implying that the speedup over baseline version on large scale tests would not be much different from that on 512 processes.

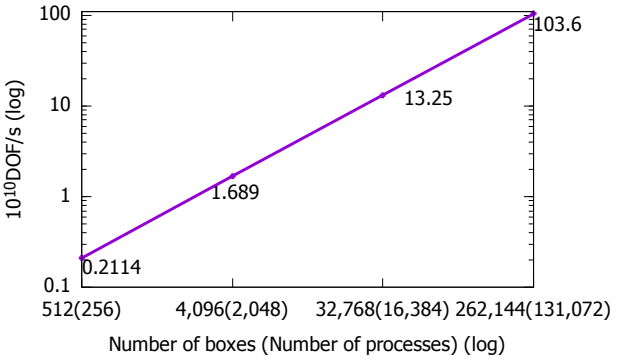


Fig. 11 Weak scaling test, shown as performance of the optimized HPGMG on different problem sizes, with 2 boxes on each process. We demonstrate linear scalability (Note the log-log plot).

5.2 Performance on 8 Million Cores

With the optimized code, we tested the performance of HPGMG on Taihulight. As required by the benchmark, the number of boxes is a cubic number N, with N=$C \times 2^r$, where C is an odd number less than 12. And as mentioned above, using a box of size 256^3, each process can hold up to 4 boxes. Thus, on TaihuLight, a system with 160,000 Core Groups, the biggest number of processes we can use is 131,072, when each process keeps two boxes, which means 64^3=262,144 boxes are processed in total. Furthermore, using more cores would incur load imbalance, without yielding higher throughput. We conducted weak scaling test of our optimized code[2] on 64^3, 32^3, 16^3, and 8^3 boxes, and show the results in DOF/s (Degrees of Freedom per second) in Figure 11. From the figure, it can be seen that the performance goes up linearly with more processes. This is because the communication is mostly with neighboring processes, which is efficient on TaihuLight, and our fusion strategy for boundary processing helps to reduce the time on waiting for the packing of data, which accelerates the whole communication procedure. In addition, the global reduction on TaihuLight has excellent performance, ensuring good scalability of HPGMG. With 2 boxes per process, on 131,072 processes, we attain 1.036×10^{12} DOF/s, which is the highest performance achieved on TaihuLight, and is Number 1 in the HPGMG ranking in Nov 2017, followed by 8.59×10^{11} DOF/s on Cori, and 5×10^{11} DOF/s on Mira [3]. The breakdown of execution time is shown in Figure 12. *smooth* is still the most time consuming function, followed by *residual*. Note that *ghost exchange* takes a larger percetage on coarser levels (boxes of size 128^3 and 64^3), but remains to be small compared to the total time. Strong scaling is also tested, on 32^3 boxes, and

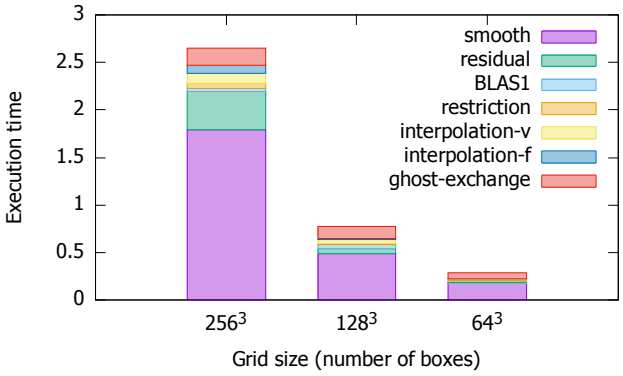


Fig. 12 Fraction of each function in total running time on 262,144 boxes. The boxes sizes denote different levels of the grid.

the results are shown in Figure 13. The tests are done on three scales (8,192 processes, 16,384 processes, and 32,768 processes), with each process dealing with 4, 2, and 1 boxes respectively. The chart shows performance on three levels, with box size of 256^3, 128^3, and 64^3. Though the coarser levels (box size of 128^3 and 64^3) have sub-linear performance, the finest level still shows almost linear scalability. The reason for the sub-linear performance on smaller boxes is that, the overhead of communication for ghost area occupies a relatively large portion. At the finer levels with more computation, the

[2] Some of the optimization is added after the testing on the whole system, therefore the results shown in this subsection is a little lower than the performance we can achieve with our latest code

[3] http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/results/results-201711/

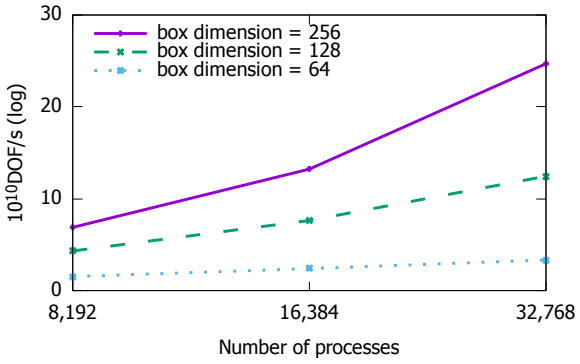


Fig. 13 Strong scaling test, shown as normalized performance of 32^3 boxes on different number of nodes. Each line plots the performance at various grid levels. Strong scaling is observed on each line, as the same amount of work is done by different number of processes. The coarser levels have sub-linear performance because the time is more dominated by communication.

communication overhead is amortized by the computation time.

6 Related work

Benchmarks for supercomputers are important measurements that not only provide basic evaluation standards for ranking, but also help to direct the design and optimization of future Exascale hardware and software. HPL (High Performance LINPACK) [16] has been the traditional benchmark for evaluating supercomputers, which solves linear equations with Gaussian elimination, using mainly dense matrix operations. As complements and alternatives to HPL, new benchmarks are being proposed, such as HPCG and HPGMG. HPCG (High Performance Conjugate Gradient) includes computation patterns that are closer to the computational applications, including sparse matrix vector multiplication, symmetric Gauss-Seidel relaxation, etc. [15,25]. HPGMG, as a new benchmark, is a solver which could be utilized in real world applications. Evaluation of HPGMG has been done on a variety of platforms, including those with multi-core CPUs and many-core GPUs [4,25]. For instance, researchers have done thorough tests and analysis on Blue Waters, a Cray XE6/XK7 hybrid system, to evaluate the performance of HPGMG in different configurations. They have made an effort to make use of both the multi-core CPUs and the many-core GPUs, by running multiple processes on each GPU and varying the box size and other parameters to find the sweet spot for the best performance. It provides an important basis for optimization and tuning of applications on hybrid systems [25]. Efforts are made on compilation to improve the performance of geometric multi-grid, leveraging function fusion [7,8]. Optimization of geometric multigrid on modern GPUs has also been studied, leveraging auto-tuning technique to find the best thread block configuration and loop tiling [9], and using the unified memory and nvlink to reduce the overhead of communication [24,23,32,31]. For TaihuLight, we reduce the memory copy overhead with the ghost area by fusing the copy for boundary into the kernel, as introduced in Section 4.3.

Stencil computation, which is the main body of HPGMG, is a hot spot in the research of high performance computing. Many works have been done to optimize stencil operations on GPUs, utilizing various techniques such as reuse of data in shared memory [40,26,29,27,21]. Aldinucci et al. built a parallel pattern for a large class of applications with the LOOP-OF-STENCIL-REDUCE within the FastFlow framework, providing optimized device memory manipulation for the iterative computation [3]. Cao et al. provided a thorough solution to optimization of high-order stencil computation in a cluster with GPUs, including optimization to GPU kernels, work load balancing between GPUs and CPU cores, and pipelining of communication, packing and computation among processes, which is similar to the work flow of HPGMG [10]. Though the SW26010 CPU is also a many-core platform, it is still quite different from GPUs. Therefore, we adopt the same 2.5D partition, but use register communication for data sharing, as the LDM is private to each CPE. Computation reuse has been investigated for multi-core CPUs and many-core GPUs, both in optimizing hand written stencil code [12, 33] and in optimization of automatic code generation [6, 26,20]. However, it is not applicable in our code, because *smooth* is a stencil computation involving 6 arrays, which is not "constant coefficient". Stencil computation has also been implemented and tuned on the IBM Cell processors, which assembles the architecture of SW26010. On both simulators and the real architectures, the blocking strategy, alignment for stanza data access, and the double buffering mechanism are effective for memory bandwidth optimization [37,11]. In our implementation, we adapt similar methodology, tailored for SW26010, as it is a larger processor mesh which supports communication among cores. Stencil operations on Sunway platform have been optimized with collective data access [5,38]. We adapted similar register communication schemes, after systematic analysis of the DMA data access behaviors, and tailored them in a way that favors the HPGMG code. Vectorization is also an important optimization for stencil computation on Sunway platform [22,5]. We did not use vector operations for *smooth* in HPGMG, as such a complicated stencil involving strided updates (the red or black el-

ements are not contiguous) and a large number of parameters would use up the vector registers, and result in too much overhead on loading and shuffling the data.

Function fusion is a widely used optimization for stencil computation [8,7], which reduces data communication significantly, at the cost of computing with deeper ghost depth. However, we did not adopt this strategy in our code, for the following reason. Fusing two iterations implies computation with a deeper ghost area in the *smooth* kernel, which requires more data to be stored in the LDM, not only for u, but also for the three *beta* arrays. As the current implementation is already making almost full usage of LDM in each CPE, we cannot fit all the data required for the fused kernel with deep ghost depth into LDM. Therefore, function fusion is not applied in our code.

7 Discussion

In this section, we conduct some discussion on the interaction of optimization methods and architectural features, and compare the optimization methods as well as the performance of HPCG and HPGMG on Sunway TaihuLight.

7.1 Optimization of Iterative Computation and Communication

As mentioned in Section 4.3, in the optimized code of *smooth*, we leave the inter-process communication out of the computation kernel, but fuse the memory copy (packing, unpacking, local copy) into the computation kernel.

Another approach for dealing with the ghost areas is separating the inner area and edge areas [5]. With this approach, the inner data are processed first, by the CPEs, while the communication of the ghost areas is being conducted by the MPE. The edge areas, which require ghost data, are processed when the communication is completed. Thus, communication and computation can be overlapped. Table 1 lists the computation time for different areas with this approach in the first two rows, where T_{ij} is the time of computing top and bottom faces, T_{ik} is the time for computing south and north faces, and T_{jk} is the time for computing left and right faces. The last two rows list the time of the approach in this paper, with T_{comp} representing the kernel computation time and T_{comm} standing for the communication time. The difference among T_{ij}, T_{ik}, and T_{jk} is mainly caused by different amounts of redundant data and various DMA access stanzas. Obviously, the benefit of avoiding inter-process communication does not overweigh the overhead of extra time spent on processing the edge areas, and that is why we did not use this approach for HPGMG on TaihuLight. The next question is, when should we switch to the overlapped approach? Let us assume the system is described with a set of parameters B_D, B_M, L, where B_D is the DMA bandwidth, B_M is the MPE memory bandwidth, and L is the network latency. Examining Table 1, we can find that T_{inner} is only slightly smaller than T_{comp}, because the amount of work on the inner area is comparable to that on the whole box, especially with collective memory access. As the total time of the two approaches are approximately $T_{comp}+T_{comm}$ and $T_{inner}+T_{ij}+T_{ik}+T_{jk}$ respectively, the switch happens when $T_{ij}+T_{ik}+T_{jk}$ is smaller than T_{comm}. Since the computation is bound by the DMA bandwidth B_D, and the communication is determined by the network latency L, the selection of the two approaches can be roughly decided with the following rule. Let f $= (T_{ij}+T_{ik}+T_{jk})/T_{comm}$, with the current system setting. For a new system with parameter set $\{B'_D, B'_M, L'\}$, we would switch to the overlapping approach when $(B'_D * L)/(B_D * L') > f$. If L is unchanged, then the crossover point is $B'_D/B_D > f$, which means B_D is improved by f times (for example, if HBM or HMC is used, or more/faster DMA channels are added).

Table 1 Processing time breakdown of the smooth operation with 512 processes.

overlapped	T_{inner}	T_{ij}	T_{ik}	T_{jk}
	0.835	0.0397	0.0738	0.192
fused	T_{comp}	T_{comm}		
	0.853	0.092		

7.2 Comparison between HPGMG and HPCG

As another important benchmark, HPCG has also been optimized on various supercomputers. In this section, we conduct a brief comparison on the computation patterns, optimization methods, and performance between HPGMG and HPCG.

The two benchmarks have a lot in common, as both solve an elliptic equation, using a numerical method on 3D grids. However, there are many differences between the methods used for the two benchmarks, listed as Table 2. An important difference between the two benchmarks is that HPCG is an iterative solver, and the benchmark terminates when achieving the same level of residuals as the reference run with 50 iterations, while HPGMG is a non-iterative solver which

solves the equation after one f-cycle. For both of them, the most time consuming function is the *smooth* operation, but as the data structure, data requirement, and the algorithm are different on the two benchmarks, the optimization approaches are also different, as shown in Table 3. One thing to notice is that the multi-color parallelization of HPCG changes the semantic of the original code, therefore changing the number of iterations, but as the metric for HPCG is Pflops, the overhead introduced by extra iterations caused by parallelization is not accurately counted in the performance metric. The performance of the two benchmarks on TaihuLight is listed in Table 4. In terms of system utilization, since we use large boxes, and each processes only 2 boxes, we were only using 32,768 nodes (131,072 CGs) in the system (using more processes would lead to imbalanced work load which would not reduce the total time), while HPCG can make use of all the 40,000 nodes (160,000 CGs). In terms of the number of grid levels, HPCG only works on 4 levels, while HPGMG needs to process $\log N$ levels, where N is the size of each dimension of the domain. This implies that HPGMG has a more complicated communication pattern.

Table 2 General comparison of HPGMG and HPCG.

HPGMG	HPCG
Solve Equation $-\nabla\cdot\ \beta(\mathrm{x})\nabla u(x) = f(x)$	Solve Equation $\nabla^2 u = f$
Matrix free	sparse matrix
Full geometric multi-grid	Conjugate gradient with multi-grid preconditioner
Stencil computation	Sparse matrix computation
Gauss Seidel Red-Black smoother	Symmetric Gauss Seidel smoother

Table 3 Comparing optimization methodologies of HPGMG and HPCG on Sunway TaihuLight.

	HPGMG	HPCG
Parallelization	Decomposition based on boxes	Multi-coloring
Blocking	2D on XY-plane with pipelining on Z-axis	3D
On-chip data movement	Collective memory access in groups	All-to-all data exchange
Communication optimization	Fusion of Pack/Unpack and smooth	Pack/Unpack on CPEs

8 Conclusion and Future Work

We provide a high performance implementation for HPGMG on Taihulight, the fastest supercomputer in the world, using SW26010 processors. With our optimization strategies including 2.5D blocking with tuned tile size, bandwidth oriented optimization using register communication, and a transformed execution sequence fusing the boundary processing into the computation kernel, we achieved 1.036×10^{12} DOF/s on 8.5 million cores. In the future, we are planning on two directions of expansion based on this work. One is more systematic investigation on the instruction level parallelism, which requires schedule adjustment of the assembly code. The other one is to apply the optimization techniques to real world applications with similar computation patterns.

Table 4 Comparing performance of HPGMG and HPCG on Sunway TaihuLight.

	HPGMG	HPCG
Grid size	4.398×10^{12}	5.033×10^{11}
#processes	131,072	160,000
Performance	1.036×10^{12} DOF/s (1.2Pflops)	0.481Pflops

Acknowledgements The authors would like to thank the anonymous reviewers for helping improve the quality of the paper. This work was supported in part by National Key R&D Plan of China (grant# 2016YFB0200603) and Beijing Natural Science Foundation (grant# JQ18001). Dr. Williams was supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.

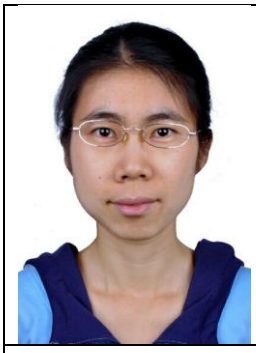
References

1. https://graph500.org (2017)
2. Adams, M.F., Brown, J., Shalf, J., Straalen, B.V., Strohmaier, E., Williams, S.: HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems (2014)
3. Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Misale, C., Peretti Pezzi, G., Torquati, M.: A parallel pattern for iterative stencil + reduce. The Journal of Supercomputing **74**(11), 5690–5705 (2018). DOI 10.1007/s11227-016-1871-z. URL `https://doi.org/10.1007/s11227-016-1871-z`
4. Ao, Y., Liu, Y., Yang, C., Liu, F., Zhang, P., Lu, Y., Du, Y.: "Performance Evaluation of HPGMG on Tianhe-2: Early Experience", pp. 230–243. Springer International Publishing, Cham (2015)
5. Ao, Y., Yang, C., Wang, X., Xue, W., Fu, H., Liu, F., Gan, L., Xu, P., Ma, W.: 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. In: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017, pp. 535–544 (2017)
6. Basu, P., Hall, M., Williams, S., Straalen, B.V., Oliker, L., Colella, P.: In: 2015 IEEE International Parallel and Distributed Processing Symposium
7. Basu, P., Hall, M., Williams, S., Van Straalen, B., Oliker, L.: Converting stencils to accumulations for communication-avoiding optimization in geometric multigrid, pp. 9–16. Association for Computing Machinery, Inc (2014)

8. Basu, P., Venkat, A., Hall, M., Williams, S., Van Straalen, B., Oliker, L.: Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. IEEE Computer Society (2013)
9. Basu, P., Williams, S., Van Straalen, B., Oliker, L., Colella, P., Hall, M.: Compiler-based Code Generation and Autotuning for Geometric Multigrid on GPU-accelerated Supercomputers. Parallel Comput. **64**(C), 50–64 (2017)
10. Cao, W., Xu, C.f., Wang, Z.h., Yao, L., Liu, H.y.: Cpu/gpu computing for a multi-block structured grid based high-order flow solver on a large heterogeneous system. Cluster Computing **17**(2), 255–270 (2014). DOI 10.1007/s10586-013-0332-1. URL `https://doi.org/10.1007/s10586-013-0332-1`
11. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, pp. 4:1–4:12. IEEE Press, Piscataway, NJ, USA (2008). URL `http://dl.acm.org/citation.cfm?id=1413370.1413375`
12. Datta, K., Williams, S., Volkov, V., Carter, J., Oliker, L., Shalf, J., Yelick, K.: Auto-tuning Stencil Computations on Multicore and Accelerators. CRC Press (2010)
13. Dong, W., Kang, L., Quan, Z., Li, K., Li, K., Hao, Z., Xie, X.H.: Implementing Molecular Dynamics Simulation on Sunway TaihuLight System. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 443–450 (2016). DOI 10.1109/HPCC-SmartCity-DSS.2016.0070
14. Dongarra, J.: Confessions of an accidental benchmarker. http://sc13.supercomputing.org/ sites/default-/files/WorkshopsArchive/pdfs/wp156s1.pdf
15. Dongarra, J., Heroux, M.A., Luszczek, P.: High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. International Journal of High Performance Computing Applications p. 1094342015593158 (2015). DOI 10.1177/1094342015593158
16. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency Computat.: Pract. Exper. **15**, 803–820 (2003). DOI 10.1002/cpe.728
17. Fu, H., He, C., Chen, B., Yin, Z., Zhang, Z., Zhang, W., Zhang, T., Xue, W., Liu, W., Yin, W., Yang, G., Chen, X.: 18.9Pflopss Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-meter Scenarios. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pp. 2:1–2:12. ACM, New York, NY, USA (2017)
18. Fu, H., Liao, J., Ding, N., Duan, X., Gan, L., Liang, Y., Wang, X., Yang, J., Zheng, Y., Liu, W., Wang, L., Yang, G.: Redesigning CAM-SE for Peta-scale Climate Modeling Performance and Ultra-high Resolution on Sunway TaihuLight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, pp. 1:1–1:12. ACM, New York, NY, USA (2017)
19. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight supercomputer: system and applications. Sci. China Inf. Sci. pp. 1–16 (2016). DOI 10.1007/s11432-016-5588-7
20. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: CGO, pp. 100–112. ACM (2018)
21. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance Code Generation for Stencil Computations on GPU Architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, pp. 311–320. ACM, New York, NY, USA (2012)
22. Jiang, L., Yang, C., Ao, Y., Ma, W.: Towards Highly Efficient DGEMM on the Emerging SW26010 Many-core Processor. In: The 46th International Conference on Parallel Processing' (2017)
23. Köstler, H., Feichtinger, C., Rüde, U., Aoki, T.: "A Geometric Multigrid Solver on Tsubame 2.0", pp. 155–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
24. Köstler, H., Ritter, D., Feichtinger, C.: "A Geometric Multigrid Solver on GPU Clusters", pp. 407–422. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
25. Kwack, J., Bauer, G.H.: HPCG and HPGMG benchmark tests on Multiple Program, Multiple Data (MPMD) mode on Blue Waters - a Cray XE6/XK7 hybrid system. https://cug.org/proceedings/cug2017_proceedings/includes/files/pap118s2-file1.pdf (2017)
26. Ma, W., Gao, K., Long, G.: Highly Optimized Code Generation for Stencil Codes with Computation Reuse for GPUs. J. Comput. Sci. Technol. **31**(6), 1262–1274 (2016)
27. Maruyama, N., Aoki, T.: Optimizing stencil computations for nvidia kepler gpus (2014)
28. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., Martin, M.: Top 500 Supercomputer Lists (2016). URL `http://www.top500.org`
29. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In: 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–13 (2010)
30. Qiao, F., Zhao, W., Yin, X., Huang, X., Liu, X., Shu, Q., Wang, G., Song, Z., Li, X., Liu, H., Yang, G., Yuan, Y.: A Highly Effective Global Surface Wave Numerical Simulation with Ultra-high Resolution. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, pp. 5:1–5:11. IEEE Press, Piscataway, NJ, USA (2016). URL `http://dl.acm.org/citation.cfm?id=3014904.3014911`
31. Sakharnykh, N.: https://github.com/e-ago/hpgmg-cuda-async (2016)
32. Sakharnykh, N.: Beyond GPU Memory Limits with Unified Memory on Pascal. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/ (2016)
33. Stock, K., Kong, M., Grosser, T., Pouchet, L.N., Rastello, F., Ramanujam, J., Sadayappan, P.: A Framework for Enhancing Data Reuse via Associative Reordering. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pp. 65–76. ACM, New York, NY, USA (2014)
34. Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y., Sun, N.: Fast implementation of DGEMM on Fermi GPU. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 35. ACM (2011)
35. Williams, S.: Hpgmg. https://crd.lbl.gov/assets/pubs_presos/HPGMG-FV-FF2-Proxy-App.pdf

36. Williams, S., Kalamkar, D.D., Singh, A., Deshpande, A.M., Straalen, B.V., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., Oliker, L.: Optimization of geometric multigrid for emerging multi- and manycore processors. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, pp. 1–11 (2012)
37. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The Potential of the Cell Processor for Scientific Computing. In: Proceedings of the 3rd Conference on Computing Frontiers, CF '06, pp. 9–20. ACM, New York, NY, USA (2006)
38. Yang, C., Xue, W., Fu, H., You, H., Wang, X., Ao, Y., Liu, F., Gan, L., Xu, P., Wang, L., Yang, G., Zheng, W.: 10M-core Scalable Fully-implicit Solver for Nonhydrostatic Atmospheric Dynamics. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, pp. 6:1–6:12. IEEE Press, Piscataway, NJ, USA (2016)
39. Zhang, J., Zhou, C., Wang, Y., Ju, L., Du, Q., Chi, X., Xu, D., Chen, D., Liu, Y., Liu, Z.: Extreme-Scale Phase Field Simulations of Coarsening Dynamics on the Sunway TaihuLight Supercomputer. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 34–45 (2016)
40. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012, pp. 155–164 (2012)

Author Photograph

			
Wenjing Ma	Yulong Ao	Chao Yang	Sam Williams

Wenjing Ma is an associate professor at the Institute of Software, Chinese Academy of Sciences. She got her Bachelor's degree in computer science and technology from Nankai University in 2004, and her Ph.D.'s degree in computer science and engineering from The Ohio State University in 2011. Her research focus is high performance computing and parallel computing, code generation and optimization.

Yulong Ao is a postdoctoral researcher at the Peking University. He received his BS of software engineering in Jilin University in 2012 and earned his PhD from University of Chinese Academy of Sciences in 2017. His research interests include high-performance and parallel computing in scientific and engineering applications as well as artificial intelligent applications, especially on large-scale supercomputing systems and heterogeneous platforms.

Chao Yang is a professor at Peking University. He received his BS in mathematics from University of Science and Technology of China in 2002 and earned his PhD from Institute of Software, Chinese Academy Sciences in 2007. His research interests include numerical analysis and modeling, large-scale scientific computing, and parallel numerical software. He has received the 2016 ACM Gordon Bell Prize, the 2017 CAS Outstanding Science and Technology Achievement Prize, and the 2017 CCF-IEEE CS Young Computer Scientist Award. He is a member of IEEE, ACM and SIAM.

Sam Williams is a staff scientist in the Performance and Algorithms Research Group at the Lawrence Berkeley National Laboratory (LBNL). His research interests include high-performance computing, auto-tuning, performance modeling, computer architecture, and hardware/software co-design. Dr. Williams received his Ph.D. in Computer Science from the University of California at Berkeley (UCB) in December of 2008. During this period, his doctoral research focused on multicore architectures and automated performance tuning. Previously, within the IRAM project, he implemented the RTL for the integer and floating-point datapaths, verified the simulators and all RTL, floorplanned the entire VIRAM1 chip, and performed all necessary place-and-route (PnR) work.