J Supercomput (2017) 73:2506–2524
DOI 10.1007/s11227-016-1943-0

CrossMark

Performance modeling and optimization of parallel LU-SGS on many-core processors for 3D high-order CFD simulations

Dali Li[1] · **Chuanfu Xu**[1,2] · **Bin Cheng**[1] · **Min Xiong**[1] · **Xiang Gao**[1] · **Xiaogang Deng**[3]

Published online: 16 December 2016
© Springer Science+Business Media New York 2016

Abstract As a typical Gauss–Seidel method, the inherent strong data dependency of lower-upper symmetric Gauss–Seidel (LU-SGS) poses tough challenges for shared-memory parallelization. On early multi-core processors, the pipelined parallel LU-SGS approach achieves promising scalability. However, on emerging many-core processors such as Xeon Phi, experience from our in-house high-order CFD program show that the parallel efficiency drops dramatically to less than 25%. In this paper, we model and analyze the performance of the pipelined parallel LU-SGS algorithm, present a two-level pipeline (TL-Pipeline) approach using nested OpenMP to further exploit fine-grained parallelisms and mitigate the parallel performance bottlenecks. Our TL-Pipeline approach achieves 20% performance gains for a regular problem ($256 \times 256 \times 256$) on Xeon Phi. We also discuss some practical problems including domain decomposition and algorithm parameters tuning for realistic CFD simulations. Generally, our work is applicable to the shared-memory parallelization of all Gauss–Seidel like methods with intrinsic strong data dependency.

Keywords LU-SGS · Multi-/many-core processor · Xeon Phi · Pipeline · Shared-memory parallelization · CFD

✉ Chuanfu Xu
xuchuanfu@nudt.edu.cn

[1] College of Computer Science, National University of Defense Technology, Changsha 410073, People's Republic of China

[2] State Key Laboratory of Aerodynamics, P.O. Box 211, Mianyang 621000, People's Republic of China

[3] National University of Defense Technology, Changsha 410073, People's Republic of China

1 Introduction

The lower-upper symmetric Gauss–Seidel (LU-SGS) [4,24] algorithm is a popular implicit method for solving large sparse linear equations in computational fluid dynamics (CFD) and many other PDE (partial differential equation)-based computational science areas. LU-SGS combines the LU factorization and the Gauss–Seidel relaxation, demonstrating high algorithmic efficiency with excellent convergence rate in many realistic CFD simulations. On the other hand, as other Gauss–Seidel like methods, it has strong data dependency. As Fig. 1 shows for a 3D structured grid, generally the forward (lower) sweep calculation of grid point (i, j, k) needs the data from grid points $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$, and the data dependence of backward (upper) sweep is vice versa. This characteristic poses tough challenge for shared-memory parallelization of LU-SGS algorithm using OpenMP as what we usually do for typical loops [26].

In CFD applications, researchers have proposed the hyper-plane/hyper-line approach (for 3D/2D grids) and the pipeline approach, for shared-memory parallelization of LU-SGS. The key idea is to exploit parallelism among independent grid points from different grid lines/planes. The hyper-plane/hyper-line strategy is based on the fact that grid points with the same index sum can be updated in parallel, and the pipeline strategy resolves the data dependency through thread synchronization. Experience shows that generally the pipeline approach largely outperforms the hyper-plane/hyper-line approach [25]. The same performance gap is also discovered in our in-house high-order CFD program. By carefully constructing a pipeline with each parallel thread/core acting as a pipeline stage, the LU-SGS computation on a grid is decomposed into many sub-tasks/sub-grids, and they will be scheduled to execute on the pipeline at different time according to the data dependency among them. The pipeline approach achieves promising scalability results on early multi-core processors and SMP systems. However, on latest multi-core processors, especially many-core processors with tens to hundreds of concurrent cores/threads, the pipeline LU-SGS algorithm performance is far from satisfactory [3]. Our experiences from the pipeline parallel LU-SGS kernel in our in-house high-order 3D structured CFD software HOSTA [22] show that the parallel efficiency is still over 70% on two shared-memory Xeon E5-2692 v2 (24 cores in total) using 24 OpenMP threads. However, on emerging many-core processors such as Xeon Phi 31S1P with 57 cores, the efficiency drops dramatically to less than 25% when using all 228 OpenMP threads.

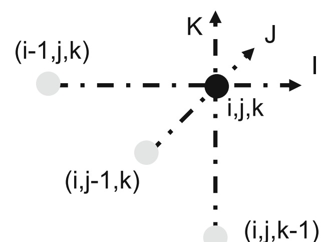

Fig. 1 Data dependence of LU-SGS forward sweep

The scalability degradation can be explained from two aspects: (1) with more pipeline stages or threads/cores (i.e., increasing the pipeline depth) on many-core processors, the overall startup and finishing overhead for the pipeline increases, for mesh block with relatively small outer k dimension, probably even there is not enough sub-tasks in the task queue to fully load the pipeline; and (2) the load balance of pipeline LU-SGS computation among tens or even hundreds of threads on many-core processors will generally be worse.

In this paper, we model and optimize the performance of the pipeline parallel LU-SGS algorithm for 3D structured grids on modern multi-/many-core processors. The main contributions of this paper are as follows:

- We present a performance model for the pipeline parallel LU-SGS on 3D grids. We use two performance metrics to evaluate the pipeline efficiency: the ratio of all pipeline stages under full-load operations (RPFO) to estimate the overall startup and finishing overheads, and the ratio of upper-/lower-bound load (RULL) between pipeline stages to estimate load imbalance. With the model, we analyze the performance behavior of the pipeline approach on Xeon CPU and Xeon Phi many-core processor, achieving similar scalability trends in line with the realistic test results.
- Guided by the model, we present a two-level pipeline (TL-Pipeline) approach for 3D problems and extend the performance model accordingly. A sub-pipeline (second-level pipeline) is implemented in each original (first-level) pipeline-stage to further exploit fine-grained parallelisms among grid dimensions (sub-planes) in a grid-plane. Particularly, the original pipeline parallel LU-SGS algorithm can be regarded as a special case of TL-Pipeline LU-SGS algorithm. The TL-Pipeline approach is implemented using nested OpenMP in the LU-SGS kernel of HOSTA. We model and analyze the performance of TL-Pipeline using a fixed problem size ($256 \times 256 \times 256$) with various pipeline depths for the second-level pipeline and the first-level pipeline. Compared to the original pipeline approach, realistic tests show that on Xeon Phi many-core processor TL-Pipeline can achieve a performance gain of 20%.
- Furthermore, we discuss how grid dimension size and the pipeline depth could impact the parallel performance, and much more impressive performance gains are obtained. We also study the feedback effects of improved two-level pipeline approach on domain decomposition. At last we give some useful suggestions in algorithm parameter configuration, making our method more practical in realistic CFD simulations. Generally, our work provides a common practice and is applicable to the shared-memory parallelization of all Gauss–Seidel like methods with intrinsic strong data dependency.

The remainder of this paper is organized as follows. We first present some related work in Sect. 2, and briefly introduces the LU-SGS algorithm and its parallel computing in Sect. 3. We model and analyze the performance of the pipelined parallel LU-SGS in Sect. 4. In Sect. 5, we detail the two-level pipeline approach. Several algorithmic problems are further discussed in Sect. 6. Finally, we conclude in Sect. 7.

2 Related work

Due to its algorithmic efficiency and convergence rate, LU-SGS has been very popular in CFD since it was first proposed by Seokkwan Yoon et al. [24]. Seokkwan Yoon presented a vectorizable and unconditionally stable LU-SGS method, achieving a 30% speedup with respect to the LU implicit scheme for transonic flow simulations. For specific CFD application problems, researchers also developed some improved LU-SGS variants. For example, Chen et al. [4] developed a block LU-SGS (BLU-SGS) for unstructured meshes, and at the cost of 20–30% more memory usage, BLU-SGS increases converges many times faster than conventional LU-SGS approaches in the simulations of transonic flows over the NACA 0012 airfoil and ONERA M6 wing, as well as supersonic flows over a 3D forebody. Yuzhi Sun et al. [20] developed an implicit nonlinear LU-SGS solver for high-order spectral difference Navier–Stokes problems, and achieved a speedups of 1 to 2 orders of magnitude over a Runge–Kutta scheme for inviscid flow and steady viscous flow simulations. Gang Wang et al. [12] proposed an improved LU-SGS scheme to meet the needs of high Reynolds number problems, and gained a significantly efficiency increase in the simulations of turbulent flows around the NACA 0012 airfoil, RAE 2822 airfoil and LANN wing. Besides, LU-SGS is also used as an efficient preconditioner for Krylov sub-space iterative algorithms [9,15,16,19] and multigrid iterative methods [14,17,18]. Generally, our parallel approach is applicable to shared-memory parallelization of LU-SGS and all its variants in various CFD applications.

To enable MPI/OpenMP hybrid parallelization of implicit CFD codes based on LU-SGS on large-scale multi-core HPC systems, researchers have proposed the hyper-plane/hyper-line approach and the pipeline approach for multithreading LU-SGS. Djomehri et al. [8] implemented both the hyper-plane and the pipeline strategies for hybrid MPI+OpenMP CFD simulations on Cray SX6, IBM Power3 and Power4, and SGI origin3000. In [25], Seokkwan Yoon investigated the performance of the hyper-plane and the pipeline strategies for real gas flow simulations, and the results show that the pipeline approach achieves better scalabilities on SMP platforms. Rupak Biswas et al. [1,2] implemented the LU-SGS linear solver of OVERFLOW-D using the pipeline approach on the Columbia supercomputer. Satoru Yamamoto et al. [23] developed a parallel "Numerical Turbine" to simulate 3D multistage stator-rotor cascade flows, using a pipelined parallel LU-SGS for implicit time-integration. All of the above works are implemented and evaluated on early multi-core processors and SMP systems.

With the development and complexity of both CFD applications and computer architectures, especially the shift from multi-core technology to many-core technology in HPC systems, it is essential to understand the behavior of the pipelined LU-SGS on modern multi-core HPC processors or even many-core processors such as GPU and MIC. Recently, Yonggang Che et al. [3] implemented a pipelined parallel LU-SGS method for supersonic combustion simulations on an compute node of Intel Xeon CPUs, observing a severe drop of parallel efficiency of 33.8% with 24 threads, and the poor scalability may be attributed to their relatively small test problem (only 812835 cells) according to our performance model. Due to the strong data dependency of LU-SGS, researchers tend to adopt more parallel-friendly time advancing methods on many-core platforms, e.g., Runge–Kutta method [21] and Jacobi iterative method

[22], and we seldom see researchers choose LU-SGS when porting their codes onto GPUs. To our knowledge, this is the first paper that presents a performance model and an improved parallel implementation for LU-SGS on Xeon Phi.

3 LU-SGS and its parallel computing

3.1 The LU-SGS algorithm

In CFD, after discretization and linearization, we need to solve a large equation system,

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

where A is the left-hand-side (LHS) matrix, b is the right-hand-side (RHS) vector, and x is the solution vector. In LU-SGS, a lower-upper splitting or decomposition is applied on A, and in each iteration the solution is separated into a forward sweep and a backward sweep. For example, in the lower-upper decomposition method, the LHS matrix A is decomposed and approximated as follows:

$$\begin{aligned} A &= D + L + U = D(I + D^{-1}L + D^{-1}U) \\ &= D\left(I + D^{-1}L\right)\left(I + D^{-1}U\right) + LD^{-1}U \\ &\approx D\left(I + D^{-1}L\right)\left(I + D^{-1}U\right) \\ &= (D + L)\, D^{-1}(D + U), \end{aligned} \tag{2}$$

where D, L and U are the diagonal part, the lower triangular part and the upper triangular part of A, respectively. Thus the Eq. (1) can be rewritten as

$$(D + L)D^{-1}(D + U)x = b. \tag{3}$$

An intermediate variable y is introduced into Eq. (3) to form two symmetric Gauss–Seidel sweep stages, i.e., a forward sweep and a backward sweep.

$$\begin{aligned} (D + L)\, y &= b \\ (D + U)\, x &= Dy, \end{aligned} \tag{4}$$

where y and x can be solved by back substitution.

In CFD, the entries of LHS matrix A (also called the Jacobian matrix) are usually approximated using grid points and their neighboring grid points. In structured CFD applications, those grid points form a regular computational stencil. Typically, the 7-point stencil is used in three-dimensional cases and the matrix A can be illustrated as follows:

$$\begin{bmatrix} \ddots & & \ddots & & \ddots & \ddots & \ddots & & \ddots & & \ddots & & \\ \mathbf{0} & l_{k-1} & \mathbf{0} & l_{j-1} & \mathbf{0} & l_{i-1} & d_{ijk} & u_{i+1} & \mathbf{0} & u_{j+1} & \mathbf{0} & u_{k+1} & \mathbf{0} \\ & & \ddots & & \ddots & & \ddots & \ddots & \ddots & & \ddots & & \ddots \end{bmatrix}, \quad (5)$$

where i, j, and k represent the indexes of three spatial dimensions, respectively. Elements of l and u in the same line are neighbors of d in physical and logical space, but in memory space with linear storage feature, they are of different distances to d. All entries of A are zero, except the entries at the diagonal and the six symmetric off-diagonals.

3.2 The pipeline approach

In CFD, other researches [25] and our experience show that the pipeline approach generally outperforms the hyper-plane/hyper-line approach for shared-memory parallelization of LU-SGS. Thus, we focus on the parallel performance modeling and optimization of the pipeline approach. Algorithm 1 presents the pseudo-code skeleton of forward LU-SGS sweep with a pipelined OpenMP parallel implementation. Due to the data dependency, we cannot simply add some OpenMP directives on the $I/J/K$ loop for multithreading. The key idea of the pipelined parallel LU-SGS is to construct a pipeline on the K dimension, with each parallel thread acting as a pipeline-stage, and on the J dimension, the LU-SGS computation is statically decomposed into many sub-tasks (i.e., $I - J$ grid sub-planes) and scheduled to the threads, the variable $flag$ is used for thread synchronization.

Algorithm 1 The pseudo-code skeleton of forward sweep of pipelined LU-SGS

```
01 !$OMP PARALLEL NUM_THREADS(d_p)
02 idt=omp_get_thread_num();
03 flag(idt) = 0;
04 !$OMP BARRIER
05 DO k = 1, nk
06    IF(idt > 0) THEN
07       Wait until flag(idt - 1) = 1;
08       flag(idt - 1) = 0;
09    END IF
10 !$OMP DO SCHEDULE(STATIC)
11    DO j = 1, nj
12    DO i = 1, ni
13       v_{i,j,k} = F(v_{i,j,k}, v_{i-1,j,k}, v_{i,j-1,k}, v_{i,j,k-1});
14    END DO
15    END DO
16 !$OMP END DO NOWAIT
17    IF(idt < d_p - 1) flag(idt) = 1;
18 END DO
19 !$OMP END PARALLEL
```

Figure 2 shows a schematic illustration for both task scheduling and the execution timeline of the pipeline with 4 threads (pipeline stages). For the mth layer (i.e., $k = m$),

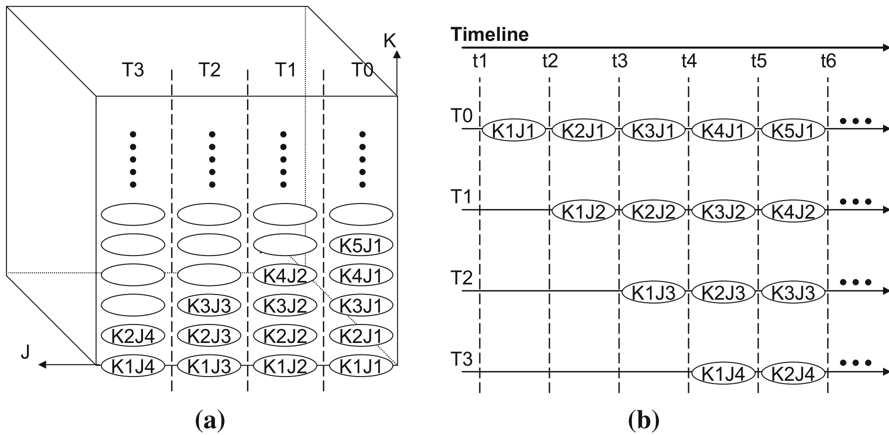

Fig. 2 Schematic illustration of the forward sweep stage of pipelined LU-SGS. **a** Workload partitioning and task scheduling. **b** Execution timeline of sub-tasks

the $I-J$ plane is divided into n ($n = 4$ for this example) sub-planes/sub-tasks (denoted as $K_m J_n$). Obviously, the sub-task $K_m J_n$ depends on $K_{m-1} J_n$ and $K_m J_{n-1}$ in the task queue, due to the data dependency. Those sub-tasks are carefully scheduled to threads in the pipeline, and in each sub-task/thread the grid points are calculated along the I dimension serially. Figure 2b shows the execution timeline, in this way, dependent sub-tasks from different K and J are pipelined with multithreading and parallelisms along $K - J$ dimensions are exploited. As we will analyze in the following sections, the pipeline efficiency will vary dramatically for different problems with various K/J and pipeline depth (i.e., the number of threads).

3.3 Implementation and experimental setup

The traditional pipeline approach and our two-level pipeline approach are implemented in the LU-SGS kernel from our in-house high-order 3D structured software HOSTA [22]. HOSTA solves the Navier–Stokes equations of fluid flow in compressible and incompressible forms with a self-developed weighted compact nonlinear scheme (WCNS) [5–7], and it has been used extensively for aerodynamic research and design optimization of realistic aircrafts such as China's civil large airplane C919. In HOSTA, LU-SGS is a very popular solving method with both favorable algorithmic efficiency and convergence rate.

The test platform we use contains on Intel many integrated cores (MIC) coprocessor, specifically Intel Xeon Phi Knight's Corner (KNC) 31S1P [10,11,13], and two shared memory Xeon E5-2692 v2 (Ivy Bridge) CPUs. The Xeon Phi 31S1P has 57 cores, with 4 hardware threads per core (i.e., supporting 228 concurrent threads in total), delivering 1 TFLOPS peak performance in double precision. The coprocessor has an on-chip memory of 8 GB, with a peak memory bandwidth of 352GB/s. Each CPU has 12 cores and the two CPUs share a 64 GB memory with a peak memory bandwidth of 204.8 GB/s, delivering 422 GFLOPS peak performance in double precision. Both the

CPUs and the Xeon Phi have extended math unit (EMU) and vector processing unit (VPU) with 256 and 512 bit vector width, respectively. We use the native programming model on Xeon Phi and compile our code using Intel *ifort* with *-mmic* option. The baseline 3D problem size is $(n_i, n_j, n_k) = (256, 256, 256)$, and other problem sizes are evaluated by altering $I/J/K$ dimensions arbitrarily.

4 Performance modeling of pipelined LU-SGS

4.1 Performance metrics

It is clearly stated in [25] that the pipeline efficiency is limited by the startup and finishing procedures. We define the ratio of all pipeline stages under full-load operations (RPFO) to evaluate the overhead of the startup and finishing. For a 3D problem with (n_i, n_j, n_k) grid points on a pipeline with d_p pipeline stages, suppose all sub-tasks are well balanced on every pipeline stages, and RPFO is defined as

$$\text{RPFO} = \begin{cases} \dfrac{(n_k - d_p + 1)}{(n_k + d_p - 1)} \times 100\% & , n_k \geq d_p \\ 0\% & , n_k < d_p, \end{cases} \tag{6}$$

where $n_k + d_p - 1$ is the number of pipeline cycles including the startup and finishing, and $n_k - d_p + 1$ is the cycles when the whole pipeline is fully occupied. If there are not enough sub-tasks in the task queue (i.e., $n_k < d_p$), the pipeline cannot be under full-load (i.e., RPFO = 0).

As we can see from Eq. (6), on early multi-core processors with only a few cores, d_p is generally much less than n_k, and correspondingly RPFO will be close to 100%, indicating a high pipeline efficiency. However, on modern multi-/many-core processors, the number of threads is likely comparable to n_k, and RPFO will decrease dramatically.

With the increase of the number of threads, the load balance among threads (pipeline stages) could be worse. Since we assume a static workload allocation among the J dimension of threads, we define the ratio of upper-/lower-bound load (RULL) between pipeline stages to estimate load imbalance as follows:

$$\text{RULL} = \left\lfloor \frac{n_j}{d_p} \right\rfloor / \left\lceil \frac{n_j}{d_p} \right\rceil \times 100\%, \tag{7}$$

where n_j/d_p is the ideal workload size for each thread, and "$\lfloor\ \rfloor$" and "$\lceil\ \rceil$" represent rounding to the floor and ceiling, respectively. Assume $n_j = d_p \times t + r (0 \leq r < dp)$, i.e., for r threads their workload size is $t + 1$, and for the other $d_p - r$ threads, their workload size is t. When the pipeline depth d_p is much less than n_j or n_j can be divided by d_p with no remainder, RULL is nearly or equals to 100%. However, when d_p is comparable to n_j, and n_j cannot be exactly divided by d_p, RULL becomes worse. An extreme case is that r equals to 1 and t is very small, the total performance would be limited by the single thread with a workload size $t + 1$, indicating a poor load balance.

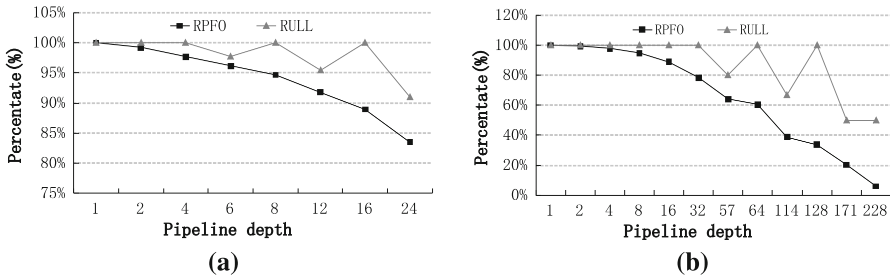

Fig. 3 Performance metrics of the pipelined LU-SGS algorithm. **a** Xeon CPUs. **b** Xeon Phi

Figure 3a, b shows the variation of RPFO and RULL on the Xeon CPUs and Xeon Phi coprocessor, respectively, for a (256, 256, 256) problem. Both metrics drop with the increase of the number of threads (pipeline depth). On the two Xeon CPUs, RPFO drops from 100% to less than 85%, and RULL drops from 100% to about 90%. On Xeon Phi, the two metrics drops significantly after the pipeline depth scales beyond 32. When all 228 threads are used (i.e. $d_p = 228$), RPFO drops sharply to less than 10% and RULL drops to 50%. Consequently, we estimate a severe parallel scalability loss for the pipelined parallel LU-SGS algorithm on many-core processors, such as Xeon Phi, with tens or even hundreds of threads according to our performance metrics.

4.2 Performance issues

Suppose the computational cost of LU-SGS for each grid point is an equal unit, then for a given (n_i, n_j, n_k) problem, the wall-time cost for a serial LU-SGS calculation WT_s is $n_i \times n_j \times n_k$, and for a pipelined implementation, the wall-time cost WT_{pp} is $(n_k + d_p - 1) \times \lceil n_j/d_p \rceil \times n_i$. We derive the speedup of the pipelined parallel LU-SGS S_{pp} as follows:

$$S_{pp} = \frac{\mathrm{WT}_s}{\mathrm{WT}_{pp}} = \frac{n_k \times n_j \times n_i}{(n_k + d_p - 1) \times \left\lceil \frac{n_j}{d_p} \right\rceil \times n_i} = \frac{n_k \times n_j}{(n_k + d_p - 1) \times \left\lceil \frac{n_j}{d_p} \right\rceil}. \quad (8)$$

This performance model combines the aforementioned two performance metrics and ignores the impact of memory bandwidth and cache efficiency for simplicity.

We compare the predicted speedup based on Eq. (8) with the test results and the ideal/linear speedup to validate our performance model. Figure 4 shows the comparison on Xeon CPUs and Xeon Phi coprocessor for a (256, 256, 256) problem.

As Fig. 4 indicates, the predicted speedup (S_{pp}) is smaller than the ideal speedup (S^*_{pp}), this is due to the overhead of pipeline mechanism. The performance gap between S_{pp} and test speedup ($\hat{S}_{pp}$) may be attributed to the memory bandwidth limitation and other potential hardware limitations. On Xeon CPUs (Fig. 4a), the three speedups are fairly close when the number of threads (pipeline depth) is relatively small (6 or less). A detaching point between S_{pp} and $\hat{S}_{pp}$(DP_2) occurs at 6 threads, and another detaching

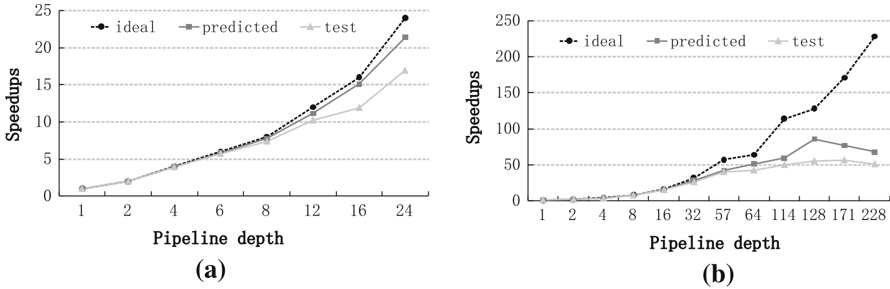

Fig. 4 The contrast of ideal, predicted and test speedups of the pipelined LU-SGS algorithm. **a** On Xeon CPUs node. **b** On Xeon Phi

point between S^*_{pp} and $S_{pp}(\text{DP}_1)$ occurs at 8 threads. In Fig. 4a, the gap between S^*_{pp} and S_{pp} is much smaller than the gap between S_{pp} and $\hat{S}_{pp}$, indicating that the pipeline approach is almost optimal on Xeon CPUs, and the parallel performance is limited by the memory bandwidth and other hardware limitations.

On Xeon Phi coprocessor (Fig. 4b) DP_1 and DP_2 appear at 16 and 57 threads, respectively. Unlike the results in Fig. 4a, the gap between S^*_{pp} and S_{pp} is much larger than the gap between S_{pp} and $\hat{S}_{pp}$. This means that the pipeline approach has a severe adaptability problem on the Xeon Phi coprocessor. Because of the relatively larger memory bandwidth and lower core/thread average floating performance, the DP_1 and DP_2 on Xeon Phi appears later than on Xeon CPU node.

5 The two-level pipeline LU-SGS approach

5.1 Fundamental ideas and implementation

As both the predicted and test results show in Sect. 3, although we could utilize more available cores for pipelined parallelization on many-core processors, the speedup drops significantly due to a long pipeline depth. Based on an in-depth analysis, we propose a two-level pipeline (TL-pipeline) approach for multithreading LU-SGS. The fundamental idea is to further exploit parallelism in each 2D sub-task/sub-plane for 3D LU-SGS problems. This is accomplished by transforming a long deep original pipeline (with depth d_p) into a relatively short first-level pipeline (with depth d_{p1}), with each pipeline-stage containing a second-level sub-pipeline (with depth d_{p2}). Obviously, $d_p = d_{p1} \times d_{p2}$ holds. The sub-pipeline is constructed in the J dimension, and each sub-task is statically decomposed along the I grid line. Figure 5 shows the task schedule for TL-Pipeline with both $d_{p1} = 4$ and $d_{p2} = 4$. In this case, $K_2 J_4$ is decomposed into some more fine-grained sub-tasks/grid-lines $J_m I_n$ to be scheduled on the sub-pipeline. Similarly, Fig. 6 presents the execution timeline for the TL-Pipeline model. The TL-Pipeline approach is implemented using nested OpenMP. For each level of OpenMP threads, we need a separate *flag* array for synchronization. We customize the number of threads of each pipeline level (d_{p1} and d_{p2}) using *num_threads()* directive clause.

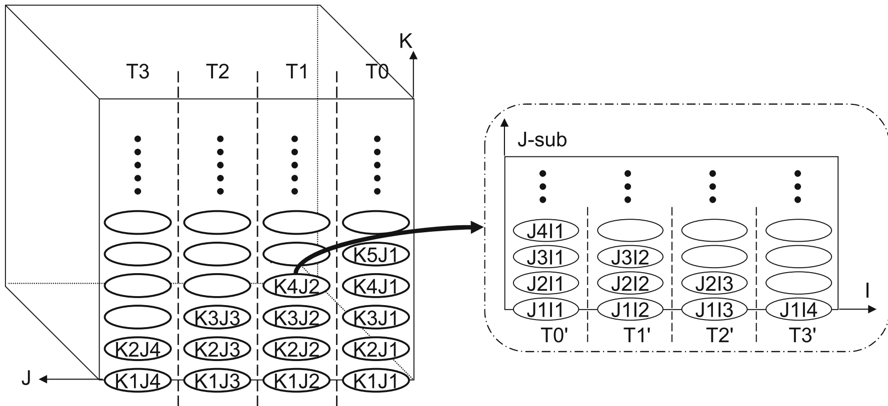

Fig. 5 Workload dispatching of TL-Pipeline LU-SGS algorithm

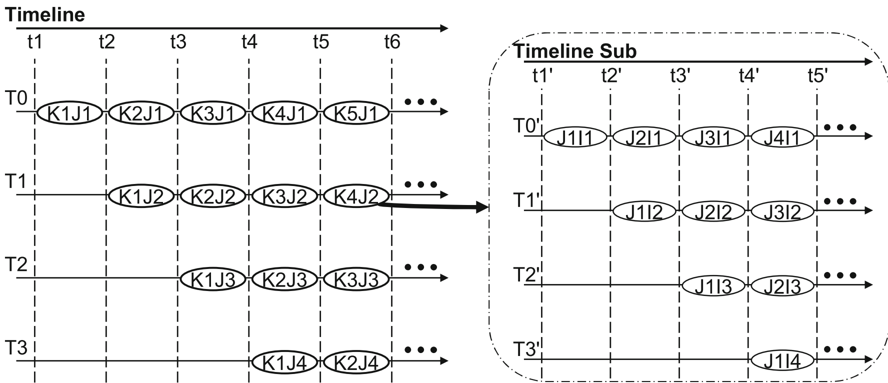

Fig. 6 Execute procedure of TL-Pipeline LU-SGS algorithm

5.2 Performance evaluation

We extend the performance model in Sect. 3 for our TL-pipeline approach. The two performance metrics, RPFO and RULL, are evaluated according to Table 1. For the first-level pipeline, $RPFO_1$ and $RULL_1$ are calculated in the same way as Eq. (8) with a pipeline depth d_{p1}. On a sub-pipeline with depth d_{p2}, sub-tasks from a $I - J$ sub-plane of $(n_i, n_j/d_{p1})$ are scheduled, thus the $RPFO_2$ and $RULL_2$ are slightly different.

Figure 7 shows the variation of the two metrics for our TL-Pipeline on Xeon CPUs and Xeon Phi for a (256, 256, 256) problem. On Xeon CPUs, appropriate d_{p1} and d_{p2} configuration $(d_{p1} \times d_{p2})$ can enhance both RPFO and RULL of the two pipelines to a nearly ideal level (i.e., 100%). For example, for a $d_{p1} \times d_{p2} = 4 \times 6$ case on Xeon CPUs with 24 cores, all metrics except $RPFO_2$ are all between 95 and 100%. The superiority of TL-Pipeline is particularly ' outstanding on Xeon Phi coprocessor with hundreds of threads. $RPFO_1$ can be increased from under 10% to nearly 100%. Although $RPFO_2$ decreases rapidly, the overall performance is a combined effect of both pipelines and its impact is limited to a certain extent.

Table 1 Performance metrics of the TL-pipeline LU-SGS algorithm

Level	RPFO	RULL
1	$\begin{cases} \frac{n_k - d_{p1} + 1}{n_k + d_{p1} - 1} \times 100\% & , n_k \geq d_{p1} \\ 0\% & , n_j < d_{p1} \end{cases}$	$\left\lfloor \frac{n_j}{d_{p1}} \right\rfloor / \left\lceil \frac{n_j}{d_{p1}} \right\rceil \times 100\%$
2	$\begin{cases} \frac{\left\lceil \frac{n_j}{d_{p1}} \right\rceil - d_{p2} + 1}{\left\lceil \frac{n_j}{d_{p1}} \right\rceil + d_{p2} - 1} \times 100\% & , \left\lceil \frac{n_j}{d_{p1}} \right\rceil \geq d_p \\ 0\% & , \left\lceil \frac{n_j}{d_{p1}} \right\rceil < d_p \end{cases}$	$\left\lfloor \frac{n_i}{d_{p2}} \right\rfloor / \left\lceil \frac{n_i}{d_{p2}} \right\rceil \times 100\%$

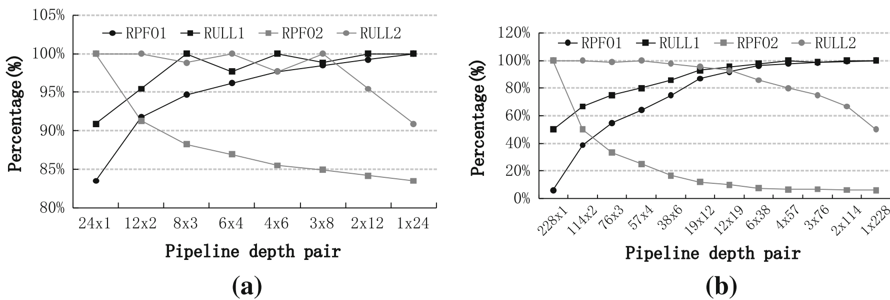

Fig. 7 Performance metrics of the TL-Pipeline LU-SGS algorithm. **a** Xeon CPUs. **b** Xeon Phi

Similar to Eq. (8), the speedup of our TL-Pipeline approach compared to a serial LU-SGS, S_{tlp}, can be evaluated as follows:

$$S_{tlp} = \frac{\mathrm{WT}_s}{WT_{tlp}} = \frac{n_k \times n_j \times n_i}{(n_k + d_{p1} - 1) \times \left(\left\lceil \frac{n_j}{d_{p1}} \right\rceil + d_{p2} - 1 \right) \times \left\lceil \frac{n_i}{d_{p2}} \right\rceil}. \quad (9)$$

Equation (9) is the same as Eq. (8) if $d_{p2} = 1$, which indicates that the original pipeline approach is a special case of our TL-Pipeline approach.

Figure 8 shows the predicted speedups (S_{tlp}) and test speedups ($\hat{S}_{tlp}$) of the TL-Pipeline approach with different $d_{p1} \times d_{p2}$ configurations for a (256, 256, 256) problem. In Fig. 8a, both the S_{tlp} and $\hat{S}_{tlp}$ vary slightly with $d_{p1} \times d_{p2}$ on Xeon CPUs. Note that the test speedup for the 1×24 case drops a lot, this is mainly due to a worse data locality and cache efficiency when decomposing the I dimension with large numbers of threads ($d_{p2} = 24$). Although for this regular problem with the same size of $n_i/n_j/n_k$, TL-Pipeline has little performance advantage over the original pipeline approach (i.e., 24×1), for other problem with various $n_i/n_j/n_k$, the benefit of TL-Pipeline is certain, as we will discuss in Sect. 5. In Fig. 8b, the S_{tlp} has a maximum improvement of nearly 70% ($d_{p1} \times d_{p2} = 6 \times 38$) compared to the original approach on Xeon Phi coprocessor, while the $\hat{S}_{tlp}$ has a much more moderate improvement of

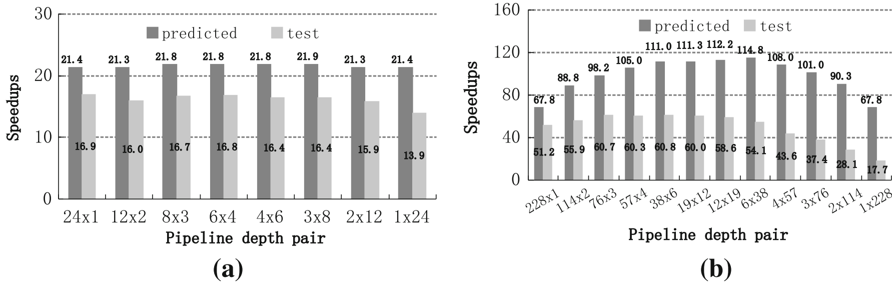

Fig. 8 The contrast of predicted and test speedups of TL-Pipeline LU-SGS. **a** Xeon CPUs. **b** Xeon Phi

nearly 20% (76×3, 57×4 and 38×6), because of memory bandwidth and other hardware limitations. Both the S_{tlp} and the $\hat{S}_{tlp}$ start to drop when $d_{p2} > 38$, and this is mainly caused by the increase of the overhead of the sub-pipeline (RULL_2) as well as the performance problems such as cache efficiency and nested OpenMP costs.

6 Further analyses and discussions

As we can see in the previous sections, the problem size (n_i, n_j, n_k) and the depth of the two pipelines (d_{p1} and d_{p2}) have an direct impact on the performance of TL-Pipeline. This section presents further discussions and suggestions helpful to achieve optimal performance in practical CFD applications.

6.1 The impact of each dimension size

Previous results are obtained using a regular problem size (i.e., $n_i = n_j = n_k = 256$); however in practice, both the problem size $(n_i \times n_j \times n_k)$ and the (n_i, n_j, n_k) dimension size vary significantly for complex multi-block structured grids. For example, a 3D delta wing grid used in our daily CFD simulations has 30 grid blocks with 16 million grid points in total. Figure 9 presents the dimension sizes and the geometric average dimension size ($\sqrt[3]{n_i \times n_j \times n_k}$) for each grid block. We can find that the dimension sizes of each block vary in a large range. In realistic applications with various (n_i, n_j, n_k) sizes on different multi-/many-core processors, if it is not convenient to decide an optimal $d_{p1} \times d_{p2}$ configuration for each grid block, users can achieve a roughly optimal configuration according to our performance model.

We change on dimension in the range of $0.1 \times n_c$ to $10 \times n_c$ (n_c is the number of cores/threads of a given processor), with the other two dimensions fixed to 256, to analyze its independent performance effect on TL-Pipeline. Figures 10 and 11 present the predicted and test performance increases for different scale ratios (from 0.1 to 10) compared to the traditional pipeline approach on Xeon CPUs and Xeon Phi, with each dimension in a sub-figure. For the K dimension on Xeon CPUs (Fig. 10a), we can see that the predicted increase is remarkable (up to 1000%) for small-scale ratios, and drops to less than 10% with large-scale ration. The test results show a similar increase, indicating that our TL-Pipeline is much more superior for the case with

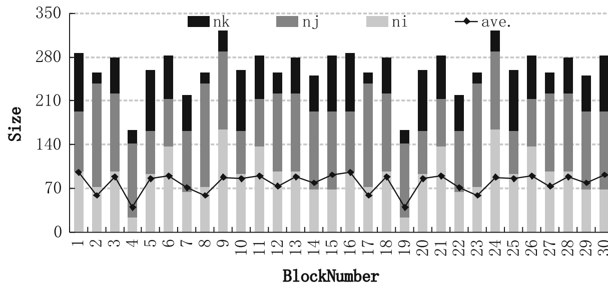

Fig. 9 Dimension sizes of a Delta Wing mesh with 30 blocks

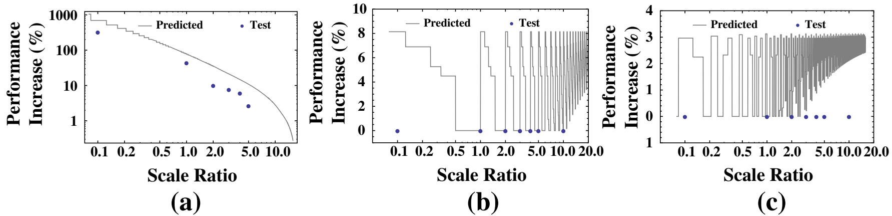

Fig. 10 The performance impact of each dimension size on Xeon CPUs node. **a** n_k, **b** n_j, **c** n_i

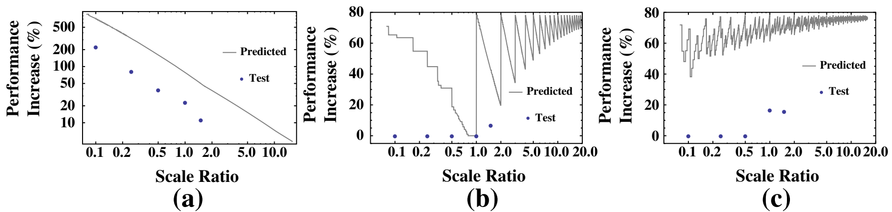

Fig. 11 The performance impact of each dimension size on Xeon Phi. **a** n_k, **b** n_j, **c** n_i

smaller K dimension. On the other hand, for the J and I dimension (Fig. 10b, c), the variation only has slight impact on performance: less than 8 and 3%, respectively, for the predicted performance and no improvement observed for the test performance. On Xeon Phi, we observe a similar variation as that of the result on Xeon CPUs: for the K dimension (Fig. 11a), both the predicted and test results have a significant increase up to 200 and 500%; for the other two dimensions (Fig. 11b, c), the predicted increases indicate a up to 80% increase, and the test result show a up to 20 and 40% increase. This is reasonable since our TL-Pipeline approach is much more superior with more thread numbers on Xeon Phi. Limited by its relatively small memory space, further tests with larger scale ratios on Xeon Phi are unavailable.

6.2 The impact of workload shape

Solving realistic CFD problems on HPC systems often involves partitioning the original grid blocks into many sub-blocks and mapping them to large-scale paral-

Table 2 The relative speedups of TL-pipeline for different workload shapes on Xeon CPUs node

	1/16	1/8	1/4	1/2	1	2	4	8	16 (α)
(a) Predicted values									
1/16	–	–	–	–	0.728	0.724	0.716	0.701	**0.672**
1/8	–	–	–	0.73	0.728	0.726	0.721	0.712	0.694
1/4	–	–	0.971	0.966	0.955	0.934	0.896	0.833	0.806
1/2	–	0.974	0.971	0.966	0.956	0.949	0.926	0.924	0.929
1	1.059	1.053	1.042	1.019	**1**	0.996	1	1	1.006
2	1.059	1.054	1.049	1.045	1.037	1.029	1.043	1.043	–
4	1.072	1.07	1.066	1.058	1.042	1.042	1.066	–	–
8	1.082	1.078	1.07	1.054	1.053	1.053	–	–	–
16 (β)	**1.084**	1.075	1.059	1.028	1.059	–	–	–	–
(b) Test results									
1/16	–	–	–	–	0.839	0.75	0.809	0.72	0.739
1/8	–	–	–	0.849	0.951	0.83	0.813	0.808	0.653
1/4	–	–	1.181	0.868	1.19	1.046	0.97	0.613	0.74
1/2	–	1.034	0.872	1.043	1.039	1.015	0.954	**0.607**	0.76
1	1.002	1.194	**1.219**	1.09	**1**	0.998	0.857	0.782	0.832
2	0.95	1.037	1.059	1.03	0.983	0.845	0.848	0.783	–
4	0.938	0.975	0.991	0.921	0.831	0.826	0.79	–	–
8	0.807	0.942	0.652	0.624	0.763	0.766	–	–	–
16 (β)	0.81	0.756	0.805	0.718	0.761	–	–	–	–

Bold values highlight relative speedups of the baseline workload shape, the best performance workload shape, and the worst performance workload shape

lel processes. The (n_i, n_j, n_k) size of sub-blocks is determined according to the average workload of each process. As we can see in Sect. 5.1, the performance of TL-Pipeline varies a lot with different (n_i, n_j, n_k) sizes. In this subsection, given a fixed overall workload $n_i \times n_j \times n_k$, we discuss some guidelines to partition optimal workload/sub-blocks for TL-Pipeline. We use the cubic grid block (256, 256, 256) as our baseline workload and compare the performance of TL-Pipeline for various grid blocks with the same workload and with scale factors α, β and γ for n_i, n_j and n_k, i.e., $(\alpha \times n_i) \times (\beta \times n_j) \times (\gamma \times n_k) = 256 \times 256 \times 256$ and $\alpha \times \beta \times \gamma = 1$. Tables 2 and 3 present both the predicted result and test result on Xeon CPUs and Xeon Phi, respectively, assuming α, β and γ ranging from 1/16 to 16 and γ determined by α and β according to the identical equation.

On Xeon CPUs (Table 2a, b), compared to the baseline with ($\alpha = 1, \beta = 1, \gamma = 1$), we observe a speedup of up to $1.084\times$ ($\alpha = 1/16, \beta = 16, \gamma = 1$) for predicted results, and a speedup of up to $1.219\times$ ($\alpha = 1/4, \beta = 1, \gamma = 4$) for test results. On the other hand, both results show cases with significant performance degradation: $0.672\times$ for the predicted performance and $0.607\times$ for the test performance, respectively. The results have a strong implication for grid partition strategies in CFD simulations for achieving an optimal parallel performance using TL-Pipeline. On Xeon Phi, the variation of speedup is even more significant. In Table 3a, the max-

Table 3 The relative speedups of TL-pipeline for different workload shapes on Xeon Phi

	1/16	1/8	1/4	1/2	1	2	4	8	16 (α)
(a) Predicted values									
1/16	–	–	–	–	0.138	0.137	0.136	0.134	**0.131**
1/8	–	–	–	0.275	0.272	0.267	0.26	0.245	0.245
1/4	–	–	0.543	0.534	0.516	0.485	0.436	0.436	0.436
1/2	–	1.057	1.004	0.913	0.773	0.713	0.711	0.715	0.715
1	1.10	1.085	1.057	1.008	**1**	1.033	1.037	1.051	1.051
2	1.447	1.41	1.34	1.282	1.276	1.317	1.334	1.373	–
4	1.611	1.532	1.417	1.444	1.455	1.487	1.528	–	–
8	**1.650**	1.501	1.483	1.473	1.489	1.53	–	–	–
16 (β)	1.562	1.526	1.537	1.492	1.425	–	–	–	–
(b) Test results									
1/16	–	–	–	–	0.187	0.181	0.239	0.238	**0.22**
1/8	–	–	–	0.351	0.356	0.369	0.402	0.373	0.327
1/4	–	–	0.656	0.564	0.66	0.631	0.598	0.43	0.509
1/2	–	0.208	0.932	1.086	1.02	0.868	0.761	0.65	0.743
1	0.184	0.808	1.102	1.036	**1**	0.97	0.984	0.97	0.973
2	0.408	**1.383**	1.272	1.163	1.129	1.074	1.155	1.24	–
4	1.032	1.294	1.246	1.206	1.108	1.151	1.297	–	–
8	1.323	1.174	1.097	1.058	1.119	1.178	–	–	–
16 (β)	1.105	1.104	1.025	1.062	1.088	–	–	–	–

Bold values highlight relative speedups of the baseline workload shape, the best performance workload shape, and the worst performance workload shape

imum speedup is 1.650 $\times$ ($\alpha = 1/16, \beta = 8, \gamma = 2$) and the minimum speedup is 0.131 $\times$ ($\alpha = 16, \beta = 1/16, \gamma = 1$) for the predicted performance; in Table 3b, the maximum speedup is 1.383 $\times$ ($\alpha = 1/16, \beta = 8, \gamma = 2$) and the minimum speedup is 0.181 $\times$ ($\alpha = 2, \beta = 1/16, \gamma = 8$) for the test performance.

According to the above results, generally we prefer bigger n_k and smaller n_i for optimal performance using TL-Pipeline with a fixed workload. The conclusion has another implication beneficial to structured CFD: the stripe of the LHS matrix A equals to the product of the middle and inner dimension sizes (i.e., $n_i \times n_j$ in our case) and primitive flow variable number for 3D problems. Since TL-Pipeline tends to perform better with large n_k and small n_i, this will result in a small stripe and consequently, a relatively higher memory access efficiency and a better convergence rate for iterative methods.

6.3 The configuration of d_{p1} and d_{p2}

As we can see from the performance model and realistic test results, the configuration of d_{p1} and d_{p2} in TL-Pipeline also has a complicated impact on the performance. In

practical CFD simulations with varying number of threads and grid sizes, choosing appropriate d_{p1} and d_{p2} is definitely a non-trivial problem. It is not possible to provide a universal optimal configuration of d_{p1} and d_{p2} for all cases. According to previous test results and our performance model, we suggest the following guidelines to configure d_{p1} and d_{p2}:

- $d_{p1} \times d_{p2}$ should be equal to or at least as close as possible to n_c (the number of cores/threads), utilizing as more available cores/threads as possible;
- d_{p1} should be much smaller than the K dimension size n_k;
- d_{p1} should be a factor of the J dimension size n_j, and d_{p2} should be much smaller than n_j/d_{p1};
- d_{p2} should be a factor of the I dimension size n_i, and n_i/d_{p2} should be rationally large enough for better cache efficiency;
- Generally d_{p1} should be larger than d_{p2}, unless the K dimension size n_k is extremely small.

Besides, since CFD simulation often involves numerous time steps of running, we can also profile simulations in the first several time steps with all possible $d_{p1} \times d_{p2}$ combinations, and use an optimal one in the following time steps.

7 Conclusion and future work

In this paper, we first discuss the strong data-dependent feature of LU-SGS algorithm and its tough challenges for parallel computing. After that, we introduce two existing parallel LU-SGS algorithms (hyper-plane and pipeline) on shared-memory platforms, and compare the merits and drawbacks of them. Then, we analyze the performance factors of pipeline LU-SGS algorithm, which usually has better performance than hyper-plane, extract two performance metrics RPFO and RULL to reflect the performance problems, and build a performance model of naïve parallel LU-SGS algorithm. Through these analyses, we discover that on latest multi-/many-core processors the pipeline depth (number of cores/threads) is commensurate with the sizes of realistic workload dimensions. This would cause worse performance metrics of the original pipeline parallel algorithm and become the main performance bottleneck.

In order to alleviate the performance problems of pipeline LU-SGS algorithm on latest multi-core especially many-core processors, we propose a novel Two-Level Pipeline LU-SGS (TL-Pipeline LU-SGS) algorithm. The TL-Pipeline LU-SGS algorithm further exploits the fine-grained parallelism of 3-dimensional workloads and organizes the cores/threads hierarchically in nested two pipeline layers. We further evaluate the performance metrics of TL-Pipeline LU-SGS algorithm on Xeon CPU node and Xeon Phi with the given workload ($256 \times 256 \times 256$), and build the performance model of TL-Pipeline LU-SGS algorithm as well. Emphatically, the basic idea of TL-Pipeline LU-SGS algorithm is not limited to the specific LU-SGS algorithm, and it can be easily extended to other strong data-dependent algorithms in various 3-dimensional applications, including the whole Gauss–Seidel algorithm family.

We implement the TL-Pipeline LU-SGS algorithm in a domestic in-house high-order accuracy CFD program, and we evaluate and contrast the performances of TL-Pipeline LU-SGS versus naïve pipeline LU-SGS algorithm on both Xeon CPU node

and Xeon Phi. Theoretically, for the given workload (256 × 256 × 256), the TL-Pipeline LU-SGS algorithm has 2 and 70% performance increases on Xeon CPU node and Xeon Phi, respectively. Our program test results draw a similar conclusion as model predicted, our TL-Pipeline LU-SGS algorithm has a performance gain of moderately 20% on Xeon Phi. Afterwards, we analyze the effects of workload sizes on algorithm performance, and discover that the k dimension size n_k has the most significant effect on algorithm performance, and the performance gain of TL-Pipeline LU-SGS algorithm on Xeon Phi is more promising than that on Xeon CPU node. We also discuss the algorithm feedback effects on domain decomposition to generate performance-friendly workloads. And finally we offer two useful strategies for the optimal or nearly optimal (d_{p1}, d_{p2}) pair determination.

Both Xeon CPU node and Xeon Phi have powerful wide vector processing ability, but the strong data-dependent feature causes complicated vector dependency and makes it difficult to exploit the performance potential of vectorization (SIMD). If this problem can be settled, the performance benefits would be impressive on these processors. Memory access and cache efficiency are two constant key points to performance optimization, especially for high memory bandwidth bounded stencil computing applications. Our TL-Pipeline LU-SGS algorithm can greatly enlarge the potential performance increases space of memory and cache optimizations.

Acknowledgements This paper was supported by the Basic Research Program of National University of Defense Technology under Grant No. ZDYYJCYJ20140101, the Open Research Program of China State Key Laboratory of Aerodynamics under Grant No. SKLA20160104, the Defense Industrial Technology Development Program under Grant No. C1520110002, and the National Science Foundation of China under Grant Nos. 11502296 and 61561146395.

References

1. Aftosmis M, Berger M, Biswas R, Djomehri MJ, Hood R, Jin H, Kiris C (2006) A detailed performance characterization of columbia using aeronautics benchmarks and applications. In: Proc. 44th AIAA Aerospace Sciences Meeting & Exhibit
2. Biswas R, Djomehri MJ, Hood R, Jin H, Kiris C, Saini S (2005) An application-based performance characterization of the columbia supercluster. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, p 26. IEEE Computer Society
3. Che Y, Cheng X, Xu C, Zhu X, Wang Z (2015) Performance engineering of a supersonic combustion simulator on heterogeneous platforms. In: Proceedings of 27th International Conference on Parallel Computational Fluid Dynamics
4. Chen R, Wang Z (2000) Fast, block lower-upper symmetric gauss-seidel scheme for arbitrary grids. AIAA j 38(12):2238–2245
5. Deng X, Mao M (1997) Weighted compact high-order nonlinear schemes for the euler equations. AIAA paper, pp 97–1941
6. Deng X, Mao M, Jiang Y, Liu H (2011) New high-order hybrid cell-edge and cell-node weighted compact nonlinear schemes. AIAA Pap 3857:2011
7. Deng X, Zhang H (2000) Developing high-order weighted compact nonlinear schemes. J Comput Phys 165(1):22–44
8. Djomehri MJ, Jin HH, Biegel B (2002) Hybrid mpi+ openmp programming of an overset cfd solver and performance investigations. Tech. rep., NASA Ames Research Center, NAS Technical Report, NAS-02-002
9. Economon TD, Palacios F, Alonso JJ, Bansal G, Mudigere D, Deshpande A, Heinecke A, Smelyanskiy M (2015) Towards high-performance optimizations of the unstructured open-source su2 suite. AIAA SciTech AIAA Pap 1949:2015

10. Fang J (2014) Towards a Systematic Exploration of the Optimization Space for Many-Core Processors. Delft University of Technology, Delft
11. Fang J, Sips H, Zhang L, Xu C, Che Y, Varbanescu AL (2014) Test-driving intel xeon phi. In: Proceedings of the 5th ACM/SPEC international conference on Performance engineering. ACM, pp 137–148
12. Gang W, Jiang Y, Zhengyin Y (2012) An improved lu-sgs implicit scheme for high reynolds number flow computations on hybrid unstructured mesh. Chin J Aeronaut 25(1):33–41
13. Li D, Xu C, Wang Y, Song Z, Xiong M, Gao X, Deng X (2015) Parallelizing and optimizing large-scale 3d multi-phase flow simulations on the tianhe-2 supercomputer. Practice and Experience, Concurrency and Computation
14. Li R, Wang X, Zhao W (2008) A multigrid block lu-sgs algorithm for euler equations on unstructured grids. Numer Math Theory Methods Appl 1:92–112
15. Liu W, Zhang L, Zhong Y, Wang Y, Che Y, Xu C, Cheng X (2015) Cfd high-order accurate scheme jacobian-free newton krylov method. Comput Fluids 110:43–47
16. Luo H, Sharov D, Baum JD, Löhner R (2003) Parallel unstructured grid gmres+ lu-sgs method for turbulent flows. AIAA Pap 273:2003
17. Otero E, Eliasson P (2011) Convergence acceleration of the cfd code edge by lu-sgs. In: 3rd CEAS European Air & Space Conference. CEAS/AIDAA, pp 606–611
18. Parsani M, Van den Abeele K, Lacor C (2007) Implicit lu-sgs time integration algorithm for high-order spectral volume method with p-multigrid strategy. In: West-East High-Speed Flow Field Conference, Moscow, Russia
19. Sharov D, Luo H, Baum JD, Löhner R (2000) Implementation of unstructured grid gmres+ lu-sgs method on shared-memory, cache-based parallel computers. AIAA Pap 927:2000
20. Sun Y, Wang Z, Liu Y (2009) Efficient implicit non-linear lu-sgs approach for compressible flow computation using high-order spectral difference method. commun. Comput Phys 5(2–4):760–778
21. Wang YX, Zhang LL, Che YG, Xu CF, Liu W, Cheng XH (2015) Efficient parallel computing and performance tuning for multi-block structured grid cfd applications on tianhe supercomputer. Tien Tzu Hsueh Pao/acta Electronica Sinica 43(1):36–44
22. Xu C, Deng X, Zhang L, Fang J, Wang G, Jiang Y, Cao W, Che Y, Wang Y, Wang Z et al (2014) Collaborating cpu and gpu for large-scale high-order cfd simulations with complex grids on the tianhe-1a supercomputer. J Comput Phys 278:275–297
23. Yamamoto S, Sasao Y, Sato S, Sano K (2007) Parallel-implicit computation of three-dimensional multistage stator-rotor cascade flows with condensation. In: Proc. 18th AIAA Computational Fluid Dynamics Conference, AIAA Paper, vol 4460, p 2007
24. Yoon S, Jameson A (1988) Lower-upper symmetric-gauss-seidel method for the euler and navier-stokes equations. AIAA J 26(9):1025–1026
25. Yoon S, Jost G, Chang S (2005) Parallelization of gauss-seidel relaxation for real gas flow. Tech. rep., NAS Technical Report, NAS-05-011
26. Zhang L, Wang Z (2004) A block lu-sgs implicit dual time-stepping algorithm for hybrid dynamic meshes. Comput Fluids 33(7):891–916