Hindawi
Scientific Programming
Volume 2019, Article ID 4254676, 12 pages
https://doi.org/10.1155/2019/4254676

Research Article

Implementation and Optimization of a CFD Solver Using Overlapped Meshes on Multiple MIC Coprocessors

Wenpeng Ma,[1] **Wu Yuan**,[2] **and Xiaodong Hu**[2]

[1]*College of Computer and Information Technology, Xinyang Normal University, Henan 464200, China*
[2]*Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Wenpeng Ma; mawp@xynu.edu.cn

Received 2 February 2019; Revised 27 March 2019; Accepted 23 April 2019; Published 27 May 2019

Academic Editor: Basilio B. Fraguela

Copyright © 2019 Wenpeng Ma et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In this paper, we develop and parallelize a CFD solver that supports overlapped meshes on multiple MIC architectures by using multithreaded technique. We optimize the solver through several considerations including vectorization, memory arrangement, and an asynchronous strategy for data exchange on multiple devices. Comparisons of different vectorization strategies are made, and the performances of core functions of the solver are reported. Experiments show that about 3.16x speedup can be achieved for the six core functions on a single Intel Xeon Phi 5110P MIC card, and 5.9x speedup can be achieved using two cards compared to an Intel E5-2680 processor for two ONERA M6 wings case.

1. Introduction

Computing with accelerators such as graphics processing unit (GPU) [1] and Intel many integrated core (MIC) architecture [2] has been attractive in computational fluid dynamics (CFD) areas recent years because it provides researchers with the possibility of accelerating or scaling their numerical codes by various parallel techniques. Meanwhile, the fast development of computer hardware and the emerging techniques require researches to explore suitable parallel methods for applications in engineering. Intel MIC architecture consists of processors that inherit many key features of Intel CPU cores, which makes the code migrating less expensive and become popular in the development of parallel algorithms.

Many CFD-based codes or solvers have been studied on Intel MIC architecture. Gorobets et al. [3] used various accelerators including AMD GPUs, NVIDIA GPUs, and Intel Xeon Phi coprocessors to conduct direct numerical simulation for turbulent flows and compared the results from these accelerators. Farhan et al. [4] utilized native and offload mode of MIC programming model to parallelize the flux kernel of PETSc-FUN3D, and they obtained about 3.8x speedup with offload mode and 5x speedup with native mode by exploring a series of shared memory optimization techniques. Graf et al. [5] ran their PDE codes on single Intel Xeon Phi Knights Landing (KNL) node and multiple KNL nodes respectively by using MPI + OpenMP programming model with different thread affinity types. Cai et al. [6] calculated the nonlinear dynamic problems on Intel MIC by employing offload mode and overlapped data transfer strategy, and they obtained 17x on MIC over sequential version on the host for the simulation of a bus model. Saini et al. [7] investigated various numerical codes on MIC to seek performance improvement with different host-coprocessor computing modes comparisons, and they also presented load-balancing approach for symmetric use of MIC coprocessors. Wang et al. [8] reported the large-scale computation of a high-order CFD code on Tianhe-2 supercomputer that consists of both CPU and MIC coprocessors. And other CFD-related works on Intel MIC architecture can be found in references [9–12]. Working as coprocessors, GPUs also have been popular in CFD. Many researchers [13–17] have studied GPU computing on structured meshes, which involved coalesced computation technique [13], heterogeneous algorithm [15, 17], numerical methods [16], etc. Corrigan et al. [18] investigated an Euler solver on GPU by employing unstructured grid and gained important factor of speedup over CPUs. Then, a lot of results included data structure

optimization [19, 20], numerical techniques [21], and applications [22] based on unstructured meshes on the GPU platform. For GPU simulations on overlapped (overset) meshes, Soni et al. [23] developed a steady CFD solver on unstructured overset meshes by using GPU programming model, which accelerated both the procedure of grid (mesh) assembly and the procedure of numerical calculations. Then, they extended their solver to unsteady ones [24, 25] to make their GPU implementation capable of handling dynamic overset meshes. More CFD-related computing on GPUs can be found in an overview reference [26].

However, most of the existing works either used consistent structured or unstructured meshes without mesh overlapping over the computational domain on MIC architecture or studied overlapped meshes on GPUs. A majority of CFD simulations involving overlapped meshes were implemented or developed on distributed system through message passing interface (MPI) [27] without using coprocessors in past several decades. Specifically, Djomehri and Jin [28] reported the parallel performance of an overset solver using a hybrid programming model with both MPI [27] and OpenMP [29]. Prewitt et al. [30] conducted a review of parallel implementations using overlapped mesh methods. Roget and Sitaraman [31] developed an effective overlapped mesh assembler and investigated unsteady simulations on nearly 10000 CPU cores. Zagaris et al. [32] discussed a range of problems regarding parallel computing for moving body problems using overlapped meshes and presented the preliminary performance on parallel overlapped grid assembly. Other overlapped mesh-related works can be found in [33–36]. Although the work in [7] conducted tests by using a solver that is compatible with overlapped meshes on MIC coprocessors, the way it accessed multiple MIC coprocessors is through the native mode or symmetric mode of MIC. In this paper, we focus on the use of offload mode of MIC and investigate the parallelization of a solver with overset meshes on a single host node with multiple MIC coprocessors. The contributions of this work are as follows:

(i) We parallelize an Euler solver using overlapped meshes and propose an asynchronous strategy for calculations on multiple MIC coprocessors within a single host node

(ii) We investigate the performances of core functions of the solver by employing offload mode with different thread affinity types on the MIC architecture, and we make detailed comparisons between the results obtained by Intel MIC vectorization and that obtained by Intel SSE vectorization

(iii) A speedup of 5.9x can be obtained on two MIC coprocessors over a single Intel E5-2680 processor for two M6 wings case

The remainder of the paper is as follows. We first introduce the MIC architecture and programming model. And this is followed by equations and numerical algorithms that have been implemented in the solver. In Section 4, we discuss implementation and optimization aspects including data transfer, vectorization, and asynchronous data exchange strategy on multiple MIC coprocessors. The performances of core functions by using different thread affinity types are reported, and comparisons are made in Section 5. The last section summarizes our work.

2. MIC Architecture and Programming Model

Many integrated core (MIC) architecture [2] is a processor that is capable of integrating many ×86 cores, providing the computing power of high parallelism. The architecture used in the first Intel Xeon Phi product is called Knights Corner, KNC. The KNC coprocessors can have many (up to 61) double dispatched, in-order executing ×86 computing cores. Each core has a 512 bit vector processing unit (VPU) which supports 16 single or 8 double floating point operations per cycle, and each core is able to launch 4 hardware threads. 32 KB L1 code cache, 32 KB L1 data cache, and 512 KB L2 cache are available to each core. The coprocessor used in this work is Intel Xeon Phi 5110P [37], which consists of 60 cores each of which runs at 1.05 GHz. It can launch a total of 240 threads simultaneously. And 8 GB of GDDR5 memory is available on it.

Anyone who is familiar with C, C++, or Fortran programming language can develop codes on MIC coprocessors without major revision of their source codes. MIC provides very flexible programming models, including native host mode, native MIC mode, offload mode, and symmetric mode [38]. Coprocessors are not used in native host mode, and programmers can run their codes on CPUs just like they do before the MIC architecture was introduced. By contrast, codes can be conducted only on coprocessors in native MIC mode when they are compiled with "-mmic" option. Symmetric mode allows programmers to run codes on both CPU cores and coprocessors. And offload mode is most commonly used on a single coprocessor or multiple coprocessors within a single host node. The basic use of offload mode for programmers is to write offload directives to make the code segment run on MIC coprocessors. To take full advantage of computational resources on MIC, the code segment can be conducted in parallel by employing multithreading techniques, such as OpenMP [29].

3. Equations and Numerical Algorithms

3.1. Compressible Euler Equations. The three-dimensional time-dependent compressible Euler equations over a control volume Ω can be expressed in integral form as

$$\frac{\partial}{\partial t}\int_{\Omega} \mathbf{W}\, d\Omega + \oint_{\partial\Omega} \mathbf{F}_c dS = 0, \tag{1}$$

where $W = [\rho, \rho u, \rho v, \rho w, \rho E]^{\mathrm{T}}$ represents the vector of conservative variables and $\mathbf{F}_c = [\rho V, \rho u V + n_x p, \rho v V + n_y p, \rho w V + n_z p, \rho H V]^{\mathrm{T}}$ denotes the vector of convective fluxes with $V = n_x u + n_y v + n_z w$.

3.2. Numerical Algorithms. The flux-difference splitting (FDS) [39] technique is employed to calculate the spatial derivative of convective fluxes. In this method, the flux at the interface (for example, i direction), expressed by $\mathbf{F}_{i\pm(1/2),j,k}$,

can be computed by solving an approximate Riemann problem as

$$\mathbf{F}_{i+(1/2),j,k} = \frac{1}{2}\left[\mathbf{F}(\mathbf{W}_{\mathrm{L}}) + \mathbf{F}(\mathbf{W}_{\mathrm{R}}) - \left|\mathbf{A}_{\mathrm{inv}}\right|(\mathbf{W}_{\mathrm{R}} - \mathbf{W}_{\mathrm{L}})\right], \quad (2)$$

where the left and right state of $\mathbf{q}$, $\mathbf{q}_{\mathrm{L}}$, and $\mathbf{q}_{\mathrm{R}}$ are constructed by Monotonic Upstream-Centered Scheme for Conservation Laws (MUSCL) [40] and min-mod limiter.

Equation (1) is solved in time in this work by employing an implicit approximate-factorization method [41], which achieves first-order accuracy in steady-state simulations.

Euler wall boundary conditions (also called inviscid surface conditions), inflow/outflow boundary conditions, and symmetry plane boundary conditions are applied to equation (1) by using the ghost cell method [42].

3.3. Mesh Technique. In this work, we aim to solve 3D Euler equations on multiple MIC devices by using overlapped mesh technique. In this technique [33, 42], each entity or component is configured by a multiblock mesh system, and different multiblock mesh systems are allowed to overlap with each other. During the numerical calculations, overlapped regions need to receive interpolated information from each other by using interpolation methods [43, 44]. The process of identifying interpolation information among overlapped regions, termed as mesh (grid) assembly [31–34, 45], has to be employed before starting numerical calculations. This technique has reduced the difficulty of generating meshes for complex geometries in engineering areas because an independent mesh system with high mesh quality can be designed for a component without considering other components. However, it adds the complexity of conducting parallel calculations (via MPI [27], for example) on this kind of mesh system. As mesh blocks are distributed on separated processors where data can not be shared directly, more detailed work should be done on data exchange among mesh blocks. There are two types of data communication for overlapped mesh system when calculations are conducted on distributed computers. One is the data exchange on the shared interfaces where one block connects to another, and the other is the interpolation data exchange where one block overlaps with other blocks. And the discussion of both types of communication is going to be covered in this work.

4. Implementation

4.1. Calculation Procedure. The procedure of solving equation (1) depends mainly on the mesh technique employed. In the overlapped mesh system, the steps of calculations are organized in Listing 1. As described in Section 3.3, performing mesh assembly is a necessary step (Listing 1, line 1) before conducting numerical computing on an overlapped mesh system. Mesh assembly identifies cells which need to receive interpolation data (CRI) as well as cells which provide interpolation data (CPI) for CRIs in overlapped regions and creates a map to record where the data associating with each block should be sent to or received from. This data map stays unchanged during the steady-state procedure and is used for data exchange (Listing 1, line 8) within each iteration. Then, the mesh blocks that connect with other blocks need to share the solutions at their common interfaces. When the data from blocks that provide interpolation and from neighbouring blocks are ready, a loop (Listing 1, lines 10–13) is launched to compute fluxes and update solutions on each block one by one.

Figure 1 illustrates how we exploit multithreading technique to perform all operations in Listing 1 on a single computing node with multiple MIC coprocessors. Even though more than one MIC coprocessor can be assembled on the same node, they are unable to communicate with each other directly. They must communicate through the host node, and it is expensive for data to be moved between MIC devices and the host node because of the limited PCI-E bandwidth. That requires us to work out data transfer strategy in order to achieve better performance on MIC coprocessors as two types of communication occur in every single iteration.

Our first consideration is to locate each mesh block cluster (MBC) that is associated with a component or entity on a specific MIC coprocessor. This benefits from the nature feature of the overlapped mesh system (Section 3.3) employed in this work. Since a MBC consists of one-to-one matched blocks only, it does not involve operations of interpolation over all mesh blocks it contains. So, this way of distributing computational workload to MIC coprocessors avoids one-to-one block data exchange across MIC coprocessors, which reduces the frequency of host-device communication. However, the data transfer from one block to all the blocks it overlaps across MIC devices is inevitable. For data transfer over overlapped regions across different MIC devices, we proposed an algorithm of communication optimization which will be introduced and discussed in Section 4.4.

As shown in Figure 1, a bunch of OpenMP threads are created on the host node, and each thread is associated with a MIC coprocessor and responsible for the data movement and computation of at least one MBC. The operations of physical boundary conditions (lines 5–7), the data exchange on one-to-one block interfaces (line 9) over a MBC, and the most time consuming part (lines 10–13) in Listing 1 are conducted on the corresponding MIC device. For steady calculations in this work, the host calls the process of mesh assembly [46, 47] only once to identify the CRIs and CPIs for each mesh block and prepare the data structure in overlapped regions. At the end of each steady cycle, the updated solutions of CPIs are used to calculate, update the interpolation data, and then copied to the host memory. The host collects the data, redistributes it for CRIs, and copies the CRIs with new interpolated values back into the MIC memory. When the interpolated data are ready, a cycle of spatial and temporal calculations can be performed on mesh blocks one by one without any data dependence.

4.2. Memory Arrangement. A MIC coprocessor has its own memory space, and independent memory allocation and

(1) Conduct mesh assembly to obtain interpolation information over the mesh system (MS)
(2) Read flow configuration file and overlapped mesh data
(3) Initialization
(4) **repeat**
(5) **for** each *iblock* in MS do
(6) Apply boundary conditions to *iblock*
(7) **end for**
(8) $\mathbf{W}_s(\mathrm{CRI}) = f(\mathbf{W}_t(\mathrm{CPI}))$: exchange interpolated data between overlapped block pair (s,t), where CRI: cells receiving interpolation data and CPI: cells providing interpolation data
(9) $\mathbf{W}_i(\mathrm{ghostcells}) \longleftarrow \mathbf{W}_j$ and $\mathbf{W}_j(\mathrm{ghostcells}) \longleftarrow \mathbf{W}_i$: exchange each one-to-one block data at block interface connecting block_i and block_j
(10) **for** each *iblock* in MS do
(11) Calculate $\mathbf{F}_c^i$, $\mathbf{F}_c^j$, and $\mathbf{F}_c^k$ fluxes on *iblock*
(12) AF time advancement and update solutions on *iblock*
(13) **end for**
(14) **Until** convergence
(15) Output flow data

Listing 1: Procedure of solving equation (1) on overlapped mesh system.

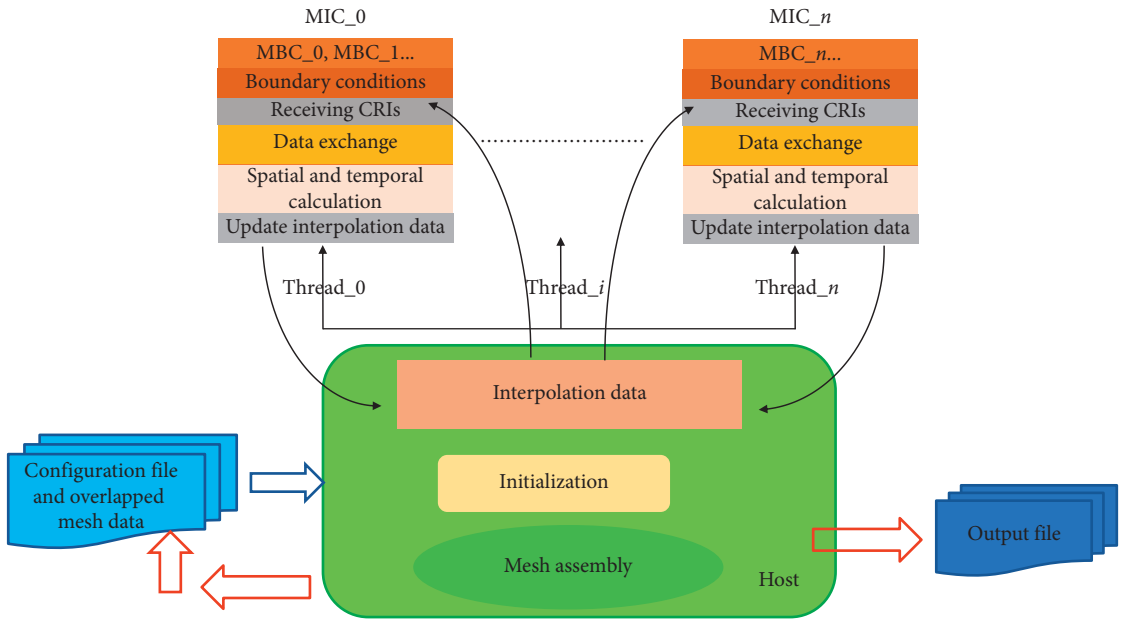


Figure 1: Flow chart of calculations on multiple MIC coprocessors.

arrangement have to be performed for the computations that are offloaded to a MIC device. More importantly, manipulating memory frequently and dynamically might have a negative effect on the overall efficiency, and it is the programmer's responsibility to control and optimize the memory usage.

In the computational procedure of solving equation (1), the core calculations including boundary conditions, spatial discretizations, and advancements in time just involve updating the values in arrays that can be used in different subroutines during the repeated cycles. Therefore, the whole procedure conducts calculations on allocated variables during the cycle and then outputs them at the end of the process. This fact inspires us to use !*dec$ offload begin target(mic:i)* and !*dec$ end offload* to include all the variables that need to be allocated on MIC or copied from host before the cycle starts. The clause of *in(varname:length(len))* is used with *alloc_if(.true.) free_if(.false.)* options to allocate memory for a variable and initialize it with the values on the host by memory copy. However, there are a lot of variables that are used as temporary spaces and do not have to be initialized. In this case, we use the keyword of *nocopy*(*nocopy(tmp:length(len_tmp) alloc_if(.true.) free_if(.false.))*) to avoid extra memory copies between the host and a MIC coprocessor. When all the calculations are completed, *out* clause is declared with *alloc_if(.false.) free_if(.true.)* options and placed between *offload begin* and *end offload* regions to copy the newest conservative flow variables to the host memory and free the variables in the device memory. The memory arrangement described above is illustrated in Listing 2.

```
(1) !dec$ offload begin target(mic:n) in(mbc_n:length(len) alloc_if(.true) free_if(.false.)) ...
(2) !dec$ end offload
(3) repeat
(4)    ..... .
(5)    !dec$ offload target(mic:n) nocopy(mbc_n:length(len) alloc_if(.false.) free_if(.false.))
(6)    !$omp parallel do private (ik, i, k)
(7)    do ik = 1, ik_total
(8)       i = (ik − 1)/(kd − 1) + 1
(9)       k = mod(ik − 1, kd − 1) + 1
(10)      call Fluxj_mic(mbc_n, i, k, ..... .)
(11)   end do
(12)   !$omp end parallel do
(13)   ..... .
(14) until convergence
(15) !dec$ offload begin target(mic:n) nocopy(mbc_n:length(len) alloc_if(.false.) free_if(.false.))
(16) !dec$ end offload
```

Listing 2: Memory arrangement on MIC.

Inside the cycle, many MIC kernels are launched by the host, and the *nocopy* keyword is used again to keep the data declared out of the cycle persistent across offloading procedure. For example, as shown in Listing 2, *Fluxj_mic* is declared as a function that can be called on MIC directly, and it reuses the array *mbc_n* which is declared and allocated outside the cycle without any extra memory operations during the cycle. And *mbc_n* can either be copied out to host memory or just deallocated on the coprocessor at the end point of the cycle. This concept of memory arrangement, which has been applied in other applications [6, 7], prevents frequent data copies inside the cycle from affecting the overall efficiency.

4.3. Vectorization. Vectorization is an effective way to improve computing speed on CPU by using Streaming SIMD (single instruction, multiple data) Extension (SSE). Similarly, MIC coprocessors support 512 bit wide Knights Corner instructions which allow 16 single or 8 double floating point operations at the same time. In this section, we explore vector optimization to make full use of MIC computational resources.

Generally, there are two different ways to implement the flux calculations. The first method is to perform a nested loop over all mesh cells (or mesh nodes). It calculates all the fluxes that go in or out of a cell's interfaces and accumulates them for the cell. And the vector operations can be applied to the innermost loop by using Intel autovectorization or manual SIMD directives. However, this algorithm involves redundant computations, in which the flux of an interface which is shared by two cells is computed twice. Literatures [18] have shown that redundant computing was not harmful to GPU implementation because it can hide the latency of global memory accesses on GPUs. An alternative way of flux computation, which is more popular for CPU computing and employed in this work, is to evaluate the flux of every edge only once by performing three nested loops along three directions. And then, the fluxes of edges are accumulated into the cells that contain them. However, there are still different coding strategies for the later technique. We investigate two of them and compare the effects on vectorizations.

One of the coding strategies (CS-1) is to conduct flux computing along different directions. For example, each thread is responsible for visiting a piece of the array of solution along i direction, denoted as $w(j, k, 1 : id, n)$, when F_i is considered. As discontinuous memory accesses occur when higher dimensions of w are accessed, loading the piece of w into a one-dimensional temporary array at the beginning is an effective way to reduce cache misses in the following process. And the remaining flux calculations can be concentrated on the temporary array completely. Then, two directives, *dir$ ivdep* or *dir$ simd*, can be declared to vectorize all the loops regarding the temporary array. Figure 2(a) shows the pseudocode of this strategy. The advantage of this consideration is to keep the code of flux computation along different directions almost same and reduce the workload of redevelopment. Another method (CS-2) refers to performing F_j, F_k, and F_i fluxes along the lowest dimension of w. Fluxes along different directions are computed by the nested loops with the same innermost loop to guarantee that the accesses of continuous memory hold, no matter which direction is under consideration. However, the body codes inside the nested loops vary from one direction to another. In this circumstance, vectorization directives are applied to the innermost (the lowest dimension) loop and the two nested loops outside are unpacked manually to increase the parallelism [7], or applied to the two inner loops by merging two loops to make full use of the VPU on MIC. The method merging the two inner loops is employed in the present paper, and the pseudocode is shown in Figure 2(b). It should be noted that the Knights Corner instructions [38], which can be applied to perform vectorization more elaborately, are not involved in this work, because this assembly-level optimization may result in poor code readability and make the overhead of code maintenance increase.

Except for the content of the code inside the loops, the process of time advancement has the similar structure of

```
jk_total = (jd − 1) ∗ (kd − 1)
do jk = 1, jk_total
    k = (jk − 1)/(jd − 1) + 1
    j = mod(jk − 1, jd − 1) + 1
!dir$ simd
    tmp(:) = w(j, k, :, n)
!dir$ ivdep
    operation_1(tmp(:))
!dir$ ivdep
    operation_2(tmp(:))
    ......
end do
```

(a)

```
do k = 1, (kd − 1)

!dir$ ivdep
    tmp(:) = w(:, k, :, n)
!dir$ ivdep
    operation_1(tmp(:))
!dir$ ivdep
    operation_2(tmp(:))
    ......


end do
```

(b)

Figure 2: Two different vectorization implementations.

nested loops with fluxes computing. Therefore, we merge the two loops inside to provide larger data set for vectorization on MIC.

4.4. Host-Device Communication Optimization. As described in Section 4.1, our task allocation strategy is to distribute workload by MBCs, which avoids the data exchange on one-to-one interfaces across MIC coprocessors but inevitably leads to communication across MIC coprocessors because of interpolation among MBCs.

Although workloads which are located on different MIC devices are performed in parallel, they still are dispatched one by one when they are on the same coprocessor. As a MBC corresponds to a specific MIC device, mesh blocks in the MBC are calculated and updated one by one. Therefore, a mesh block that contains CPIs does not have to wait until the computation of other mesh blocks has completed before it needs to copy data to the host. More specifically, when the solution of a mesh block has been updated successfully during a block loop (Listing 1, line 10), it can start a communication request to copy interpolation data provided by CPIs to the host immediately. And the overhead of copying data from a mesh block to the host can be hidden by the flow computing of other mesh blocks in the same device. To implement this idea, we take advantage of the asynchronous transfer of the offload mode. Because overlapped mesh that consists of 2 MBCs is the most common case, we address the pseudocode of our algorithm about communication optimization in Algorithm 1 by using 2 MBCs and 2 MICs.

Outside the main procedure of numerical calculations, a loop over all MIC devices is performed. In order to make each device responsible for a part of the whole computation, we use OpenMP directive *omp parallel do* before the loop to start a number of threads, each of which maps to a device. According to the memory arrangement discussed in Section 4.2, memory allocation and initialization have been done on devices before the computation starts. Therefore, by using this thread-device mapping, each device can access its own device memory, start computing, and use shared host memory for necessary data exchange. At the end point of time advancement, the clause *offload_transfer* is used with the *signal* option to put a request of data transfer from the device to the host, and it returns immediately and allows the device to keep running the next MIC kernels. So, the data transfer and the block computing can be conducted at the same time. Once all the MIC kernels in a steady step have finished, the CPU is required to check the signals by using *offload_wait* to make sure all the transfers have been done before executing the process of exchanging interpolation data. Then, the CPU uses the master thread to distribute the interpolation data from CPIs to the corresponding positions of CRIs. Vectorization can be employed in this part because the operations involve assignment operations only. After that, the CPU starts data transfer requests to copy data back to each device respectively. At the same time, the next cycle begins, and the MIC kernel of *set_boundary_condition* is expected to run on MIC overlapping with the overhead of transferring the interpolation data back to MIC devices. In some cases, the last mesh block residing on a MIC device may have the largest number of CPIs, and it is difficult to overlap the data transfer from device to the host with computation because there are no more blocks that need to be computed. As the overlapped region involves only a part of mesh blocks, we sort the mesh blocks in the descending order in accordance with the number of CPIs to avoid transferring data with insufficient overlapped computation.

5. Experiments

5.1. Testing Environment. Our experiments were conducted on the YUAN cluster at the Computer Network Information Center at the Chinese Academy of Sciences. The cluster is of hybrid architecture that consists of both MIC and GPU nodes. The configuration of MIC nodes is that each node has two Intel E5-2680 V2 (Ivy Bridge, 2.8 GHz, 10 cores) CPUs and two Intel Xeon Phi 5110P MIC coprocessors. The memory capacity for the host and coprocessors is 64 GB and 8 GB, respectively. Two environmental variables, MIC_OMP_NUM_THREADS and MIC_KMP_AFFINITY, were declared to investigate the performances under different number of OpenMP threads and affinity types. Intel Fortran compiler (version 2013_sp1.0.080) was used with the highest optimization -O3 and -openmp options to compile the solver.

5.2. Results. We used two ONERA M6 wings (shown in Figure 3), each of which was configured with four $129 \times 113 \times 105$ subblocks. The lower wing and its mesh system were formed by making a translation of the upper wing down along Y-axis by the length of the wing, and then, the two mesh systems overlapped with each other. Figure 4 shows a closer view of the overlapped mesh system at the symmetry plane before the mesh assembly. Figure 5 shows a closer view of the mesh system after the mesh assembly was performed. The mesh assembly procedure automatically formed an optimized region where the interpolation relationship between the two mesh systems can be identified, which is shown in the middle of the upper and lower wings. The initial condition for this case is as follows: Mach number $M = 0.84$, angle of attack $\alpha = 3.06^{\circ}$, and slip angle $\beta = 0^{\circ}$. As the host node has two Intel MIC coprocessors, we made each

```
(1) !$omp parallel do private(idev, ib, ......)
(2) do idev = 0, 1
(3) repeat
(4)    offload target(mic:idev): set_boundary_condition for each block
(5)    if(icycle > 1) offload target(mic:idev) wait(sgr(idev)): set_CRI_to_domain
(6)    offload target(mic:idev) exchange_interface_data
(7)    do ib = 1, nb(idev)
(8)       offload target(mic:idev): spatial_step
(9)       offload target(mic:idev): temporal_step
(10)      offload target(mic:idev): compute_CPI
(11)      offload_transfer target(mic:idev) out(CPI_ib) signal(sgp_idev(ib))
(12)   end do
(13)   master thread: offload_wait all sgp_idev related to each device
(14)   master thread on CPU: exchange_interpolation_data
(15)   master thread: offload_transfer target(mic:idev) in(CRI_imbc) signal(sgr(idev))
(16) until convergence
(17) !$omp end parallel do
```

Algorithm 1: Communication optimization algorithm.

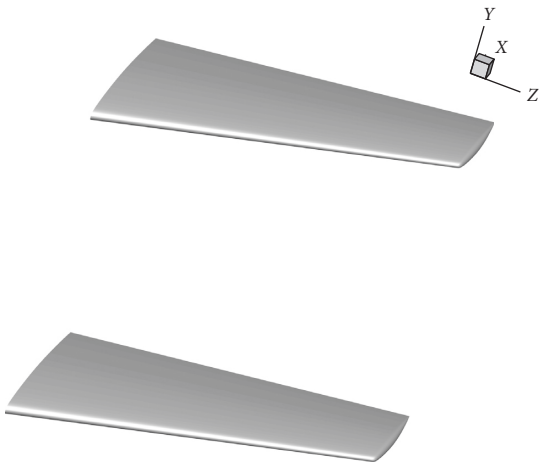


Figure 3: Two M6 wings.

MIC coprocessor responsible for the workload of one mesh system. The results presented below start with the performances of six kernels (three in spatial and three in temporal) running on MIC.

Tables 1–3 show both the wall clock times for one cycle obtained by different combinations of number of threads and thread affinity types and that obtained by CPUs. The wall clock times for one cycle were evaluated by averaging the times of first 100 cycles. And the second coding strategy for vectorization of flux computing was employed in this test. In each table, the first column lists the number of threads from 4 to 118, which is horizontally followed by two three-column blocks. Each block corresponds to a specified MIC kernel and lists wall clock times in different thread affinity types. A column in such a block represents the wall clock times on MIC varying from the number of threads under a declared thread affinity type. In order to observe the scalability of MIC kernels as the number of threads increases, we demonstrate the relative speedup as t_4/t_p, where t_4 is the wall clock time estimated by using 4 threads and t_p is that estimated by using p threads. The relative speedups are also filled in each column on the right of wall clock times. Furthermore, to compare the performance on the MIC architecture with that on CPUs, we also show the corresponding full vectorization optimized CPU time for each kernel at the last row of each table and calculate the absolute speedup through dividing the CPU time by the minimum value in all three values obtained by 118 threads under different affinity types. The results in all three tables show that the wall clock times obtained by scatter and balanced modes have an advantage over that obtained by compact mode. This can be explained by the fact that the compact mode distributes adjacent threads to the same physical processor as much as possible in order to make maximum utilization of L2 cache when adjacent threads need to share data with each other but this can result in load imbalance problems among physical cores. Our implementations have made each thread load independent data into a temporary array and operate the array locally, which is more suitable for both scatter and balanced modes because the local operations in each thread make better use of L1 cache on a processor without any intervention from other threads.

As the balanced mode distributes threads to processors in the same way as the scatter mode does when NT ≤ 59, the results obtained from these two modes are supposed to be same. However, there exist slight differences between them through Tables 1–3 because of time collecting errors coming from different runs. When NT = 118, each coprocessor took responsibility of two threads in both the balanced and scatter modes, but they differed because the balanced mode kept two adjacent threads on the same coprocessor whereas the scatter mode kept them on different coprocessor. However, the results obtained by these two modes did not show obvious gap in this case. This might be caused by the cache race among threads on a single MIC processor when it undertook more than one thread.

It is noticed that except F_j the wall clock times of other five MIC kernels no longer decrease effectively when more

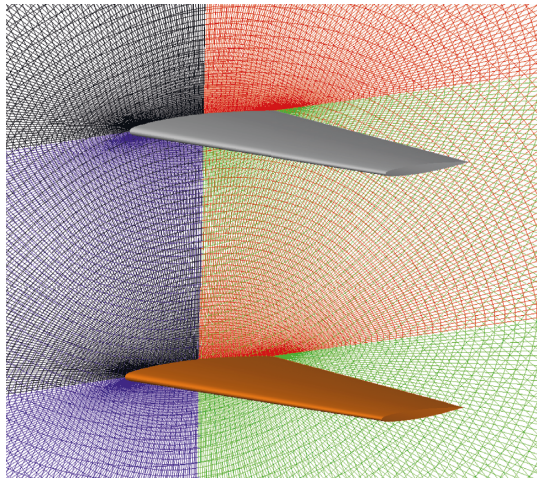


Figure 4: Closer view of two overlapped mesh systems before mesh assembly.

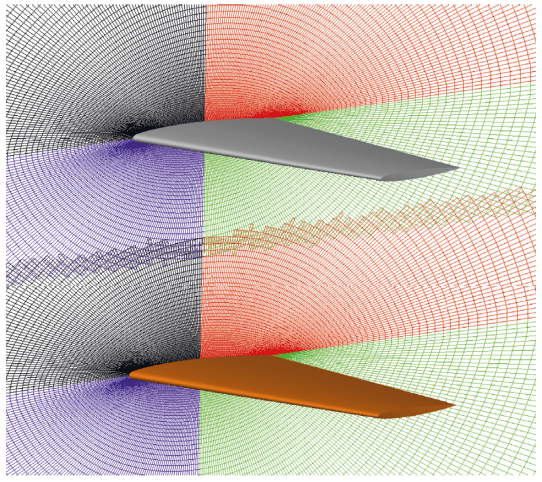


Figure 5: Closer view of two overlapped mesh systems after mesh assembly.

Table 1: Wall clock times for F_j and F_k (in seconds).

NT	F_j			F_k		
	Scatter	Balanced	Compact	Scatter	Balanced	Compact
4	1.80 (1.0x)	1.81 (1.0x)	3.256 (1.0x)	0.680 (1.0x)	0.680 (1.0x)	1.668 (1.0x)
8	0.932 (1.93x)	0.935 (1.94x)	1.680 (1.94x)	0.352 (1.93x)	0.352 (1.93x)	0.856 (1.95x)
16	0.520 (3.46x)	0.519 (3.49x)	0.880 (3.7x)	0.192 (3.54x)	0.194 (3.51x)	0.468 (3.56x)
32	0.288 (6.25x)	0.288 (6.28x)	0.488 (6.67x)	0.160 (4.25x)	0.161 (4.22x)	0.28 (5.96x)
59	0.196 (9.18x)	0.196 (9.23x)	0.296 (11.0x)	0.144 (4.72x)	0.144 (4.72x)	0.224 (7.45x)
118	0.144 (12.5x)	0.136 (13.3x)	0.160 (20.35x)	0.148 (4.59x)	0.148 (4.59x)	0.196 (8.51x)
CPU time		0.52 (3.82x)			0.54 (3.64x)	

NT : number of threads.

than 59 threads were used in both balanced and scatter modes. This can be explained by the fact that the OpenMP directives in these kernels worked at the outmost loop and the maximum number of workload pieces equals to 128 (or 112, 104); so, it was difficult for each thread to balance the workload on it when 118 threads were launched. And for F_j, we vectorized it along the innermost direction and manually combined the two nested looped into one loop to increase the parallelism by slightly modifying the codes. Therefore, it has been showed that F_j scaled better than F_k and F_i did as the number of threads increased, and F_k and F_i have an advantage in wall clock time due to maximizing the use of VPU on MIC but at the expense of scalability.

A speedup of 2.5x was obtained on the MIC coprocessor for TA_j, which is the lowest speedup among the three kernels. That was mainly caused by the cache misses and poor vectorization when each thread tried to load the $j-k$ plane of the solution array $(w(j,k,i,n))$ into the temporary

Table 2: Wall clock times for F_i and TA_j (in seconds).

NT	F_i			TA_j		
	Scatter	Balanced	Compact	Scatter	Balanced	Compact
4	1.22 (1.0x)	1.22 (1.0x)	2.288 (1.0x)	2.132 (1.0x)	2.132 (1.0x)	6.80 (1.0x)
8	0.636 (1.91x)	0.636 (1.91x)	1.196 (1.91x)	1.160 (1.83x)	1.161 (1.84x)	3.44 (1.98x)
16	0.352 (3.47x)	0.353 (3.46x)	0.660 (3.47x)	0.632 (3.37x)	0.630 (3.38x)	1.832 (3.71x)
32	0.268 (4.55x)	0.266 (4.59x)	0.432 (5.29x)	0.360 (5.92x)	0.362 (5.89x)	0.960 (7.08x)
59	0.232 (5.26x)	0.231 (5.28x)	0.272 (8.41x)	0.296 (7.2x)	0.296 (7.2x)	0.684 (9.94x)
118	0.216 (5.65x)	0.212 (5.75x)	0.260 (8.8x)	0.296 (7.2x)	0.288 (7.4x)	0.48 (14.17x)
CPU time		0.676 (3.19x)			0.72 (2.5x)	

NT : number of threads.

Table 3: Wall clock times for TA_k and TA_i (in seconds).

NT	TA_k			TA_i		
	Scatter	Balanced	Compact	Scatter	Balanced	Compact
4	0.988 (1.0x)	0.988 (1.0x)	2.424 (1.0x)	1.404 (1.0x)	1.404 (1.0x)	2.736 (1.0x)
8	0.508 (1.94x)	0.508 (1.94x)	1.256 (1.93x)	0.716 (1.96x)	0.714 (1.97x)	1.436 (1.91x)
16	0.280 (3.53x)	0.282 (3.50x)	0.664 (3.65x)	0.408 (3.44x)	0.407 (3.45x)	0.804 (3.4x)
32	0.164 (6.02x)	0.166 (5.95x)	0.368 (6.59x)	0.260 (5.4x)	0.264 (5.32x)	0.464 (5.90x)
59	0.140 (7.06x)	0.139 (7.11x)	0.232 (10.4x)	0.200 (7.02x)	0.202 (6.95x)	0.283 (9.67x)
118	0.156 (6.33x)	0.152 (6.5x)	0.196 (12.4x)	0.200 (7.02x)	0.199 (7.06x)	0.228 (12.0x)
CPU time		0.56 (3.68x)			0.572 (2.87x)	

NT : number of threads.

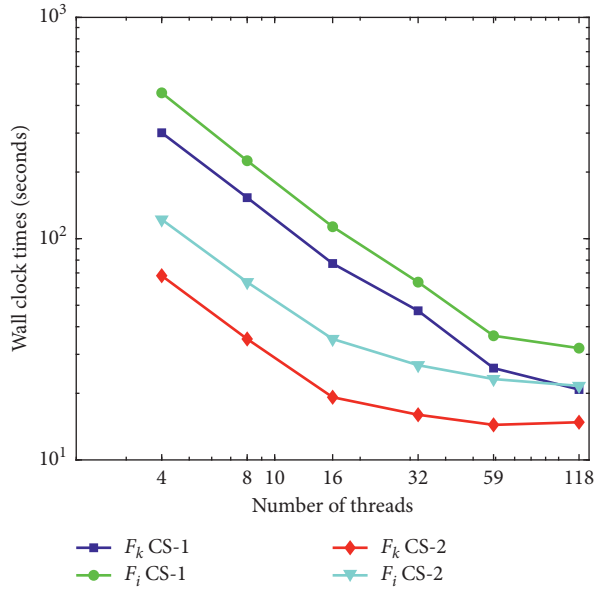


Figure 6: Comparisons among different implementations in vectorization.

space and reshaped it along k direction. The same reason can explain the results shown in Figure 6, where the comparisons between two code strategies (CS) for flux computation of F_k and F_i by using balanced mode have been made. The wall clock times for the four kernels were evaluated by performing the first 100 steps of the solver. CS-1 conducted 10 more discontinuous array loadings than CS-2 did, which made vectorizations far from efficient on MIC coprocessors. Although the discontinuous memory accesses were vectorized by *simd* clause compulsively, they turned out less effective on MIC coprocessors. Also, we can observe clear efficiency drops when more than 16 threads were used in CS-1, because we have done 2D vectorization in CS-1 which made it hard for MIC to keep balanced workload among threads.

Then, we split each of the original block into 4 subblocks along both i and j directions and form a mesh system of 16 blocks for each wing and report the wall clock times for 500 steps by running the case on CPU and two MIC devices, respectively. Figure 7 shows wall clock time comparisons in

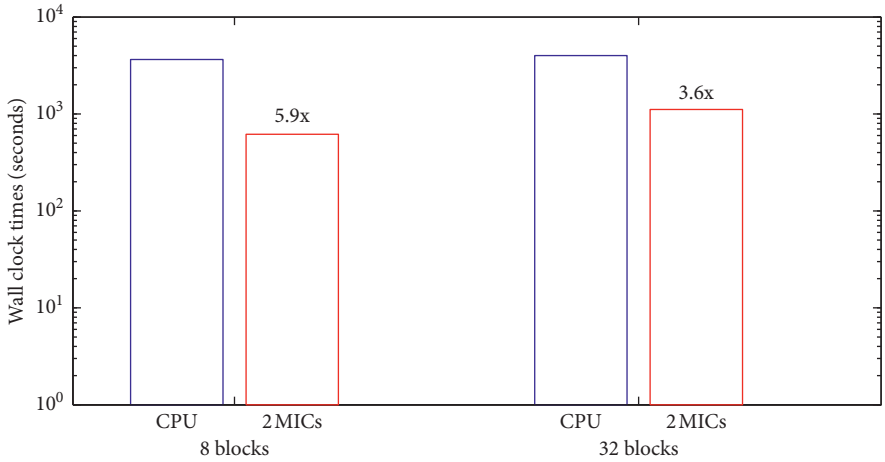


Figure 7: Wall clock times for 500 steps on CPU and MIC.

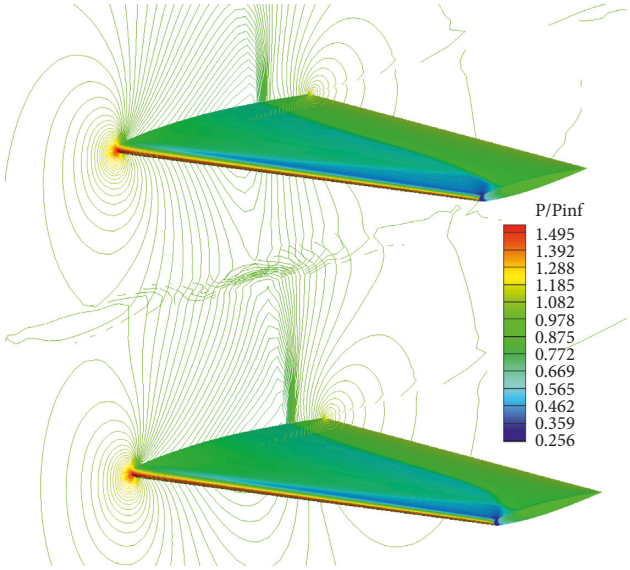


Figure 8: Pressure contours at the symmetry plane.

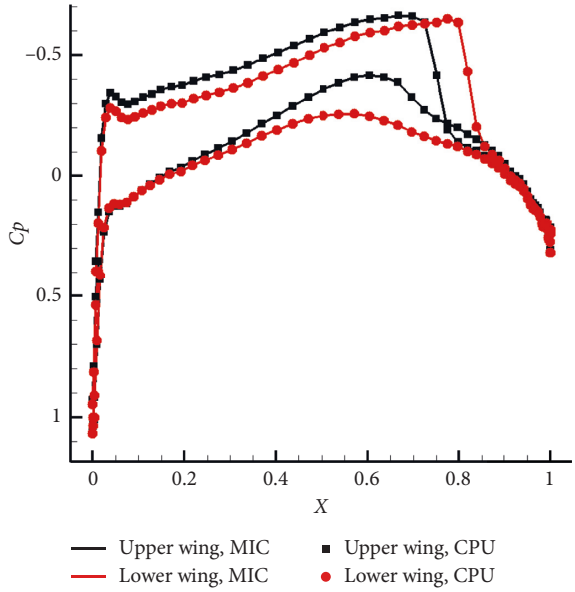


Figure 9: Pressure coefficients comparisons on the airfoils at the symmetry plane.

different block sizes. Solving equation (1) on 8 overlapped mesh blocks using two Intel Xeon Phi 5110P MIC cards can achieve 5.9x speedup compared to sequential calculations with full optimizations, whereas solving the same problem on 32 overlapped mesh blocks using two MIC cards only achieves 3.6x. This is to be expected, because threads are unlikely to keep busy and balanced workload when the dimensions of blocks are relatively small as stated above, and the six core functions can achieve only about 2x speedup on a single MIC device. Furthermore, it is observed that sequential calculations using 32 mesh blocks spent about 10% more time than that using 8 mesh blocks. This can be explained by the fact that larger mesh block makes better use of Intel SSE vectorization. To show the correctness and accuracy of the parallel solver on the MIC architecture, we plot the pressure contours at the symmetry plane in Figure 8 and compare the corresponding pressure coefficients on the airfoils calculated by MICs with that calculated by CPUs in Figure 9. We can see clearly from Figure 9 that our solver running on MICs can capture each shockwave which is expected on each airfoil for this transonic problem and produce pressure coefficients in agreement with that calculated by CPUs.

6. Summary

We have developed and optimized an overlapped mesh-supported CFD solver on multiple MIC coprocessors. We demonstrated and compared different code strategies for vectorizations by using a two M6 wings case and analysed the results in detail. An asynchronous method has been employed in order to keep the data exchange from interfering the overall efficiency. Calculations on the case achieved 5.9x speedup using two MIC devices compared to the case using an Intel E5-2680 processor. Our future work includes extending this solver to cover unsteady flow

calculations which involve relative motion among overlapped mesh blocks on multiple MIC devices.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by a grant from the National Natural Science Foundation of China (nos. 61702438 and 11502267), the Key Research Project of Institutions of Higher Education of Henan Province (no. 17B520034), and the Nanhu Scholar Program of XYNU.

References

[1] NVIDIA, *NVIDIA Tesla V100 GPU Architecture*, NVIDIA, Santa Clara, CA, USA, 2017.

[2] https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[3] A. Gorobets, F. Trias, R. Borrell, G. Oyarzun, and A. Oliva, "Direct numerical simulation of turbulent flows with parallel algorithms for various computing architectures," in *Proceedings of the 6th European Conference on Computational Fluid Dynamics*, Barcelona, Spain, July 2014.

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

[5] J. S. Graf, M. K. Gobbert, and S. Khuvis, "Long-time simulations with complex code using multiple nodes of Intel Xeon Phi knights landing," *Journal of Computational and Applied Mathematics*, vol. 337, pp. 18–36, 2018.

[6] Y. Cai, G. Li, and W. Liu, "Parallelized implementation of an explicit finite element method in many integrated core (MIC) architecture," *Advances in Engineering Software*, vol. 116, pp. 50–59, 2018.

[7] S. Saini, H. Jin, D. Jespersen et al., "Early multi-node performance evaluation of a knights corner (KNC) based NASA supercomputer," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshop*, Chicago, FL, USA, May 2015.

[8] Y. X. Wang, L. L. Zhang, W. Liu, X. H. Cheng, Y. Zhuang, and A. T. Chronopoulos, "Performance optimizations for scalable CFD applications on hybrid CPU+MIC heterogeneous computing system with millions of cores," *Computers & Fluids*, vol. 173, pp. 226–236, 2018.

[9] K. Banaś, F. Krużel, and J. Bielański, "Finite element numerical integration for first order approximations on multi- and many-core architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 305, pp. 827–848, 2016.

[10] W. C. Schneck, E. D. Gregory, and C. A. C. Leckey, "Optimization of elastodynamic finite integration technique on Intel Xeon Phi knights landing processors," *Journal of Computational Physics*, vol. 374, pp. 550–562, 2018.

[11] J. M. Cebrián, J. M. Cecilia, M. Hernández, and J. M. García, "Code modernization strategies to 3-D stencil-based applications on Intel Xeon Phi: KNC and KNL," *Computers & Mathematics with Applications*, vol. 74, no. 10, pp. 2557–2571, 2017.

[12] M. Lukas, Z. Jan, M. Michal et al., "Evaluation of the Intel Xeon Phi offload runtimes for domain decomposition solvers," *Advances in Engineering Software*, vol. 125, pp. 46–154, 2018.

[13] S. M. I. Gohari, V. Esfahanian, and H. Moqtaderi, "Coalesced computations of the incompressible Navier–Stokes equations over an airfoil using graphics processing units," *Computers & Fluids*, vol. 80, no. 1, pp. 102–115, 2013.

[14] L. Fu, K. Z. Gao, and F. Xu, "A multi-block viscous flow solver based on GPU parallel methodology," *Computers & Fluids*, vol. 95, pp. 19–39, 2014.

[15] W. Cao, C. F. Xu, Z. H. Wang, H. Y. Liu, and H. Y. Liu, "CPU/GPU computing for a multi-block structured grid based high-order flow solver on a large heterogeneous system," *Cluster Computing*, vol. 17, no. 2, pp. 255–270, 2014.

[16] M. Aissa, T. Verstraete, and C. Vuik, "Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes," *Computers & Mathematics with Applications*, vol. 74, no. 1, pp. 201–217, 2017.

[17] C. Xu, X. Deng, L. Zhang et al., "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer," *Journal of Computational Physics*, vol. 278, pp. 275–297, 2014.

[18] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, "Running unstructured grid-based CFD solvers on modern graphics hardware," *International Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221–229, 2011.

[19] A. Lacasta, M. Morales-Hernández, J. Murillo, and P. García-Navarro, "An optimized GPU implementation of a 2D free surface simulation model on unstructured meshes," *Advances in Engineering Software*, vol. 78, pp. 1–15, 2014.

[20] P. Barrio, C. Carreras, R. Robles, A. L. Juan, R. Jevtic, and R. Sierra, "Memory optimization in FPGA-accelerated scientific codes based on unstructured meshes," *Journal of Systems Architecture*, vol. 60, no. 7, pp. 579–591, 2014.

[21] Y. Xia, H. Luo, M. Frisbey, and R. Nourgaliev, "A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids," in *Proceedings of the 7th AIAA Theoretical Fluid Mechanics Conference*, Atlanta, GA, USA, 2014.

[22] J. Langguth, N. Wu, J. Chai, and X. Cai, "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 120–131, 2015.

[23] K. Soni, D. D. J. Chandar, and J. Sitaraman, "Development of an overset grid computational fluid dynamics solver on graphical processing units," *Computers & Fluids*, vol. 58, pp. 1–14, 2012.

[24] D. D. J. Chandar, J. Sitaraman, and D. Mavriplis, "GPU parallelization of an unstructured overset grid incompressible Navier–Stokes solver for moving bodies," in *Proceedings of the 50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Nashville, TN, USA, January 2012.

[25] D. Chandar, J. Sitaraman, and D. Mavriplis, "Dynamic overset grid computations for CFD applications on graphics processing units," in *Proceedings of the Seventh International Conference on Computational Fluid Dynamics*, Big Island, Hawaii, July 2012.

[26] K. E. Niemeyer and C.-J. Sung, "Recent progress and challenges in exploiting graphics processors in computational fluid dynamics," *Journal of Supercomputing*, vol. 67, no. 2, pp. 528–564, 2014.

[27] G. Edgar, E. F. Graham, B. George et al., "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users? Group Meeting*, pp. 97–104, Budapest, Hungary, September 2004, http://www.open-mpi.org.

[28] M. J. Djomehri and H. Jin, "Hybrid MPI+OpenMP programming of an overset CFD solver and performance investigations," NASA Technical Report, NASA Ames Research Center, Moffett Field, CA, USA, 2002.

[29] B. Chapman, G. Jost, and R. van der Pass, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, Cambridge, MA, USA, 2007.

[30] N. C. Prewitt, D. M. Belk, and W. Shyy, "Parallel computing of overset grids for aerodynamic problems with moving objects," *Progress in Aerospace Sciences*, vol. 36, no. 2, pp. 117–172, 2000.

[31] B. Roget and J. Sitaraman, "Robust and efficient overset grid assembly for partitioned unstructured meshes," *Journal of Computational Physics*, vol. 260, pp. 1–24, 2014.

[32] G. Zagaris, M. T. Campbell, D. J. Bodony et al., "A toolkit for parallel overset grid assembly targeting large-scale moving body aerodynamic simulations," in *Proceedings of the 19th International Meshing Roundtable*, pp. 385–401, Springer, Berlin, Heidelberg, October 2010.

[33] J. Cai, F. Tsai, and F. Liu, "A parallel viscous flow solver on multi-block overset grids," *Computers & Fluids*, vol. 35, no. 10, pp. 1290–1301, 2006.

[34] B. Landmann and M. Montagnac, "A highly automated parallel Chimera method for overset grids based on the implicit hole cutting technique," *International Journal for Numerical Methods in Fluids*, vol. 66, no. 6, pp. 778–804, 2011.

[35] W. D. Henshaw, "Solving fluid flow problems on moving and adaptive overlapping grids," in *Proceedings of the International Conference on Parallel Computational Fluid Dynamics*, Washington, DC, USA, May 2005.

[36] W. Liao, J. Cai, and H. M. Tsai, "A multigrid overset grid flow solver with implicit hole cutting method," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 9–12, pp. 1701–1715, 2007.

[37] https://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1-053-GHz-60-core-.

[38] E. Wang, Q. Zhang, B. Shen et al., *High-Performance Computing on the Intel® Xeon Phi™, How to Fully Exploit MIC Architectures*, Springer, Berlin, Germany, 2014.

[39] P. L. Roe, "Approximate Riemann solvers, parameter vectors, and difference schemes," *Journal of Computational Physics*, vol. 43, no. 2, pp. 357–372, 1981.

[40] A. Jameson, W. Schmidt, and E. Trukel, "Numerical solutions of the Euler equations by finite volume methods using Runge–Kutta time-stepping schemes," in *Proceedings of the 14th Fluid and Plasma Dynamics Conference AIAA Paper*, Palo Alto, CA, USA, 1981.

[41] T. H. Pulliam and D. S. Chaussee, "A diagonal form of an implicit approximate-factorization algorithm," *Journal of Computational Physics*, vol. 39, no. 2, pp. 347–363, 1981.

[42] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*, Elsevier, Amsterdam, Netherlands, 2nd edition, 2005.

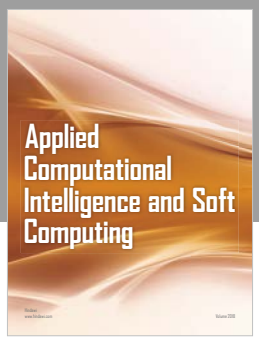
[43] Z. Wang, N. Hariharan, and R. Chen, "Recent developments on the conservation property of chimera," in *Proceedings of the 36th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, USA, January 1998.

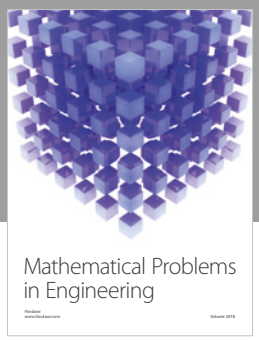
[44] R. L. Meakin, "On the spatial and temporal accuracy of overset grid methods for moving body problems," in *Proceedings of the 12th Applied Aerodynamics Conference AIAA Paper 1994-1925*, Colorado Springs, CO, USA, June 1994.

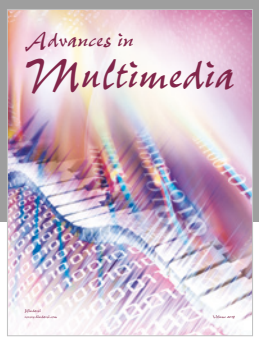
[45] S. E. Rogers, N. E. Suhs, and W. E. Dietz, "PEGASUS 5: an automated preprocessor for overset-grid computational fluid dynamics," *AIAA Journal*, vol. 41, no. 6, pp. 1037–1045, 2003.

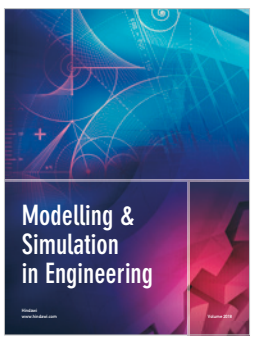
[46] W. Ma, X. Hu, and X. Liu, "Parallel multibody separation simulation using MPI and OpenMP with communication optimization," *Journal of Algorithms & Computational Technology*, vol. 13, pp. 1–17, 2018.

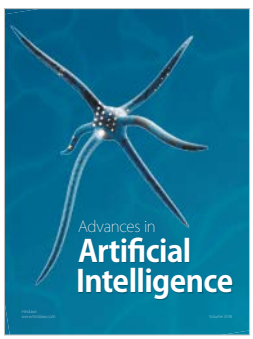
[47] Y. Wu, "Numerical simulation and Aerodynamic effect research for multi-warhead projection," *Journal of System Simulation*, vol. 28, no. 7, pp. 1552–1560, 2016, in Chinese.

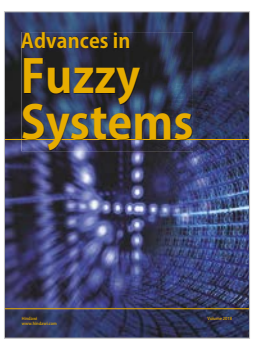


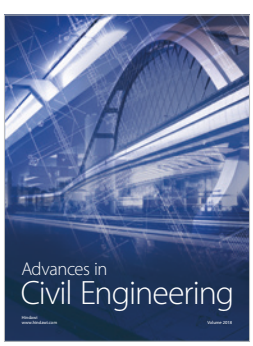


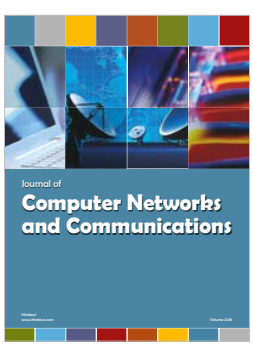


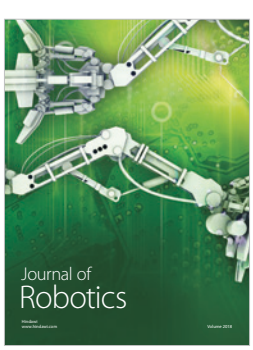


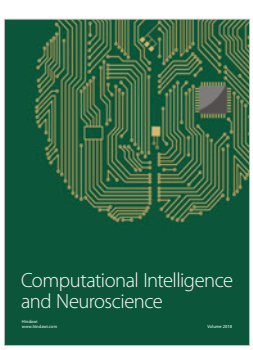












Hindawi

Submit your manuscripts at
www.hindawi.com