CPU/GPU Code Acceleration on Heterogeneous Systems and Code Verification for CFD Applications

Weicheng Xue

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Aerospace Engineering

Dec. 12th, 2020

Blacksburg, Virginia

Keywords: MPI, OpenACC, Code acceleration, Code verification, CFD

CPU/GPU Code Acceleration on Heterogeneous Systems and Code Verification for CFD Applications

Weicheng Xue

(ABSTRACT)

Computational Fluid Dynamics (CFD) applications usually involve intensive computations, which can be accelerated through using open accelerators, especially GPUs due to their common use in the scientific computing community. In addition to code acceleration, it is important to ensure that the code and algorithm are implemented numerically correctly, which is called code verification. This dissertation focuses on accelerating research CFD codes on multi-CPUs/GPUs using MPI and OpenACC, as well as the code verification for turbulence model implementation using the method of manufactured solutions and code-to-code comparisons. First, a variety of performance optimizations both agnostic and specific to applications and platforms are developed in order to 1) improve the heterogeneous CPU/GPU compute utilization; 2) improve the memory bandwidth to the main memory; 3) reduce communication overhead between the CPU host and the GPU accelerator; and 4) reduce the tedious manual tuning work for GPU scheduling. Both finite difference and finite volume CFD codes and multiple platforms with different architectures are utilized to evaluate the performance optimizations used. A maximum speedup of over 70 is achieved on 16 V100 GPUs over 16 Xeon E5-2680v4 CPUs for multi-block test cases. In addition, systematic studies of code verification are performed for a second-order accurate finite volume research CFD code. Cross-term sinusoidal manufactured solutions are applied to verify the Spalart-Allmaras and $k - omega$ SST model implementation, both in 2D and 3D. This dissertation shows that the spatial and temporal schemes are implemented numerically correctly.

CPU/GPU Code Acceleration on Heterogeneous Systems and Code Verification for CFD Applications

Weicheng Xue

(GENERAL AUDIENCE ABSTRACT)

Computational Fluid Dynamics (CFD) is a numerical method to solve fluid problems, which usually requires a large amount of computations. A large CFD problem can be decomposed into smaller sub-problems which are stored in discrete memory locations and accelerated by a large number of compute units. In addition to code acceleration, it is important to ensure that the code and algorithm are implemented correctly, which is called code verification. This dissertation focuses on the CFD code acceleration as well as the code verification for turbulence model implementation. In this dissertation, multiple Graphic Processing Units (GPUs) are utilized to accelerate two CFD codes, considering that the GPU has high computational power and high memory bandwidth. A variety of optimizations are developed and applied to improve the performance of CFD codes on different parallel computing systems. The program execution time can be reduced significantly especially when multiple GPUs are used. In addition, code-to-code comparisons with some NASA CFD codes and the method of manufactured solutions are utilized to verify the correctness of a research CFD code.

CPU/GPU Code Acceleration on Heterogeneous Systems and Code Verification for CFD Applications

Weicheng Xue

(GRANT INFORMATION)

The code verification work (Chapter 6 and Chapter 7) is partially based on research sponsored by the U.S. Air Force under agreement number FA865019-2-2204. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

To my parents, for their unconditional love, from the past to the future

Acknowledgments

To my advisor, Dr. Christopher Roy, without your guidance and support for the past five years, I would not have a chance to finish any of my work for my Ph.D. Thank you very much for your kindness, help and support for my Ph.D. life at Virginia Tech.

To my committee members, Dr. Eric de Sturler, Dr. Kevin Wang and Dr. Bhuvana Srinivasan, thank you very much for being my committee members and offering helpful suggestions through my Ph.D. life at Virginia Tech.

To Dr. Feng Wu, thank you very much for offering me the funding to work with you in a machine learning based autotuning project in 2017 summer.

To the AOE staff at Virginia Tech, especially Kelsey Walls and Steve Edwards, thank you very much for your help in answering Ph.D. program questions and fixing IT issues, respectively.

To Fred and Mary Kay, thank you very much for your help in teaching me how to improve my oral English.

To my friends and labmates, especially Jian Liu, Zhenzhong Zhang, Tong Xu, Zuen Ren, Yi Luo, Ning Liu, Hongyu Wang, Nan Si, Andrew McCall, Charles Jackson, William Tyson, Juntao Wang, Yang Zeng and Yuzhi Li, thank you very much for your help and support for my life, far more than research. Without you all, I cannot imagine what would happen and I would not have the courage to overcome any difficulties.

To my aunt, Xianxiu Xue, thank you very much for supporting my family since my childhood.

To my cousin and her husband, Jiaoer Xue and Ming Chen, thank you very much for taking care for my father when I am studying in US. You have already done so much for me.

To my father and mother, thank you very much for your unconditional love all the way, from the past to the future. You have already sacrificed a lot to bring me up and provide me a good education. I think it is time for me to take care of you. I love you all so much!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The General Purpose Graphic Processing Unit (GPGPU) [1] has aroused domain scientists' interest greatly in various scientific computing fields because of its advantages over the Central Processing Unit (CPU). First, the GPU usually has thousands of lightweight threads so that it is more suitable for compute-intensive work than the CPU, which has fewer threads but is better at executing complex instructions. The CPU usually serves as the host which offloads the data and computation to the GPU used as the accelerator device. Second, the GPU is intentionally designed to have much higher memory bandwidth to its main memory and higher memory clock rate and lower latency than the CPU. Finally, the GPU is optimized for performance per watt so it can be more energy efficient compared to the CPU, which is not considered in this work but is crucial for commercial applications.

Although the GPU has many advantages, programming on the GPU and obtaining high performance is not easy. CUDA [2], OpenCL [3] and OpenACC [4] are the most commonly used language extensions/directives for the GPU programming. CUDA and OpenCL are low level programming models which require a good understanding of the underlying architecture, while OpenACC is a high level library specification. With OpenACC, users only need to add annotated directives to the legacy serial code; although satisfactory speedup may still require performance optimizations. The compiler can determine the details of kernel scheduling on the GPU if the user does not specify how how it can be done, and the performance can still be fair. Also, with the advent of new architectures, it may require a large amount of code modification and tuning if a code uses CUDA or OpenCL. While for OpenACC, fewer adaptions are needed as a lot of runtime details can be decided by the compiler. OpenACC provides three different parallelisms for the computational work on the GPU, which are gang, worker and vector and can be seen in Fig 1.1. Gang, worker and vector are equivalent to the three respective levels which are block, warp and thread using CUDA.

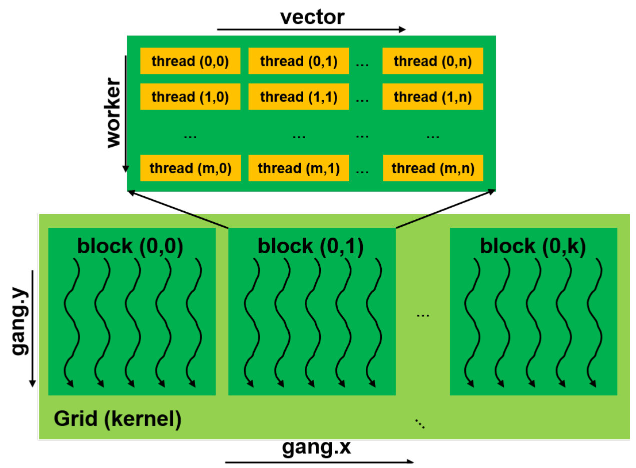


Figure 1.1: Multilevel parallelisms for GPU

A heterogeneous system is a hybrid system having multiple types of processing units, including CPU/GPU, CPU/MICS, CPU/FPGA, etc. The CPU is usually used to deal with instruction execution and another type of processing unit is used for the computations. CPU+GPU is the most commonly used heterogeneous system currently, as it can combine the advantages of the CPU and GPU and it is fit for non-uniform memory access (NUMA) architecture. However, the data transfer in heterogeneous system can be complicated and can be a very important performance bottleneck if scaling to a large number of nodes, especially for memory bound problems [5]. NVLink [6] or GPUDirect (GPUDirect is an umbrella word for multiple GPU communication technologies) can be used to improve the memory bandwidth and reduce the latency between CPU hosts and GPU devices and solely between devices [7]. Fig. 1.2 shows that there are multiple types of connections in a heterogeneous system. If multiple GPUs are used, both intra-node and inter-node communication exists, which can be optimized by using NVLink, GPUDirect, or some manual data conversions.

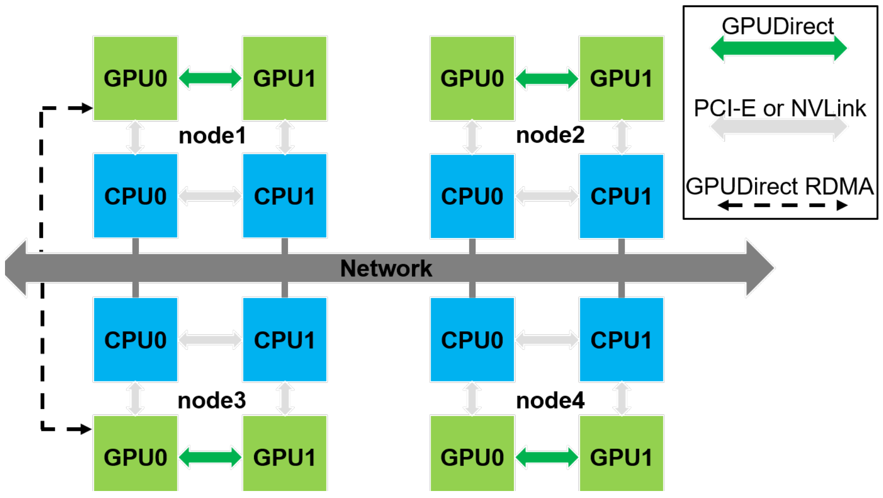


Figure 1.2: Heterogeneous CPU/GPU system

Computational fluid dynamics (CFD) is a field which involves intensive computations to solve fluid problems numerically. Major tasks for CFD include gaining execution time speedups and obtaining high accuracy solutions for fluid problems, which are the focus of this dissertation. For large problems with more than a million degrees of freedom, it may take several days or even longer to obtain a solution. Thus, it becomes necessary to accelerate such applications through high performance computing such as GPU computing [8]. If optimized properly, a single modern GPU is orders-of-magnitude faster than a single modern CPU core [8]. Higher speedup can be achieved if using multiple GPUs. In addition to accelerating a code, it is also important to prove the code is implemented correctly through rigorous code verification studies [9, 10] and careful code-to-code comparison. This dissertation will demonstrate how to verify the steady/unsteady Spalart-Allmaras [11] and $k-\omega$ SST [12, 13] RANS turbulence model implementations in a research CFD code.

1.1 Related Work

1.1.1 Code Acceleration

High performance parallel computing [14] including multi-processor computing and GPU computing enables a program to run faster. For multi-processor computing, OpenMP [15], MPI [16] and hybrid MPI+OpenMP have been widely used and their performance has also been carefully analyzed [17, 18, 19, 20, 20, 21]. MPI is fit for coarse grained parallelism while OpenMP is fit for fine grained parallelism. The hybrid MPI+OpenMP model can be applied well to modern non-uniform memory access (NUMA) architectures. MPI and OpenMP are also frequently used in GPU computing with multi-level parallelism. A lot of work has been done to leverage the power of the GPU for CFD applications. Jacobsen et al. [22] investigated MPI+CUDA and MPI+OpenMP+CUDA for the classic incompressible lid driven cavity problem and they found that the MPI+CUDA implementation performs the best. Elsen et al. [23] obtained a speedup of up to 40× on a single GPU using BrookGPU [24]. Brandvik et al. [25] applied CUDA to accelerate a 3D Euler code using a single GPU and obtained a speedup of 16×. Luo et al. [26] applied MPI+OpenACC to port a 2D incompressible Navier-Stokes solver to 32 NVIDIA C2050 GPUs and achieved a speedup of 4× over 32 CPUs. Xia et al. [27] applied OpenACC to accelerate an unstructured CFD solver based on a Discontinuous Galerkin method, and achieved a speedup of up to 24× on one GPU compared to one CPU core. Gong et al. [28] presented an optimized OpenACC version for a core kernel of an incompressible Navier-Stokes solver. Hoshino et al. [29] showed that the gap between using CUDA and OpenACC can be decreased to only 2% after careful manual optimizations. Also, it is interesting and rare to see that the OpenACC implementation can be even slightly faster than CUDA [30].

The CPU host may idle when the GPU accelerator device is doing computations. To make full use of the computational resources in nodes, the host should be utilized to do some computational work concurrently with the device. Mittal et al. [31] investigated numerous heterogeneous computing techniques at the runtime, algorithm, programming, and application level. Chandar et al. [32] developed a hybrid multi-CPU/GPU framework on unstructured overset grids using CUDA, showing that the hybrid CPU/GPU framework can outperform the pure GPU framework to some degree. Domanski et al. [33, 34] also showed that the heterogeneous CPU/GPU approach can provide performance benefits over the pure GPU implantation. Alvarez et al. [35] showed that a gain of 32% can be obtained by using the hybrid implementation compared to the GPU-only implementation. For heterogeneous computing, load imbalance and communication overhead should be carefully treated so that the performance can be improved, otherwise the performance can be slower than the pure GPU implementation.

Although the compiler can automatically determine the kernel scheduling parameters when using OpenACC, some performance gains can be obtained if the user knows how to tune the parameters properly [36]. Manual tuning requires a lot of experience and time, and is also error-prone. Also, inter-dependency may exist among different tuning parameters, which makes manual tuning rather difficult if there are too many parameters. Jia et al. [37] used a statistic tree-based approach to cluster the data according to their importance, which may require a large number of samples to achieve satisfactory accuracy. Collins et al. [38] applied a principal component approach to find important parameters. Falch et al. [39] used artificial neural networks to autotune some OpenCL kernels, but the model can be quite inconsistent as the optimal configuration can vary greatly if running multiple times. Cui et al. [40] used a pruning approach to select good samples for subsequent training. However, only one set of thread-block size (two parameters) for one kernel was considered.

1.1.2 Code Verification

The purpose of code verification is to guarantee that the code implementation and scheme are numerically correct [9, 10]. The most rigorous method is the order of accuracy (OOA) test [9]. For steady flows, there is only spatial discretization error at the converged state when the iterative residual error is driven to machine zero (There is also round-off error, but it is generally small for double precision). For unsteady flows, the total discretization error includes both the spatial and temporal discretization error components. There are a lot of factors which can affect the magnitude of the discretization errors, including mesh quality, numerical schemes, time step size (for unsteady flows), sub-iterative errors (for unsteady flows) and even implementation errors.

Exact solutions and manufactured solutions are often used in OOA studies. However, for turbulent flows, few exact solutions exist. After plugging the manufactured solutions to the governing equations, source terms can be computed. It does not matter whether the source terms are physical or not. In literature, both physical solutions [41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51] and non-physical manufactured solutions [52, 53, 54, 55, 56, 57] for Euler/laminar NS were used. This dissertation will address code verification studies for RANS turbulence models, especially using two equation models including $k - \omega$ SST.

1.2 Outline

This dissertation contains several chapters. In Chapter 2, a 3D buoyancy driven cavity code is accelerated by multiple modern GPUs using 1D, 2D and 3D domain decomposition. This code is a finite difference code using an artificial compressibility method. Several performance optimizations including the pack/unpack strategy, reducing the communication cost based on different primitive variable stencils and the use of GPUDirect are applied to improve the strong and weak scaling efficiency. MPI+OpenACC are used in the GPU code.

In Chapter 3, the performance of a multi-block CFD code called SENSEI [58, 59, 60] is optimized on multiple GPUs. Various performance optimizations are applied to improve the compute utilization, reduce the communication overhead and improve the memory bandwidth. After the optimizations, a single P100 GPU and a single v100 GPU are more than 30 $\times$ and 70 $\times$ faster than a single modern CPU core, respectively. For multiple GPUs, both the strong and weak parallel efficiencies for large enough problems are higher than 70% up to 16 GPUs for different test cases (the efficiencies can be even higher for single parent-block cases). The performance optimizations used for the complicated CFD code can be applied to other codes using MPI+OpenACC.

In Chapter 4, a CPU-GPU heterogeneous computing framework for SENSEI is developed. Using a GPU or two GPUs and a certain number of CPUs can achieve higher speedups for various cases. Some suggestions are given to improve the CPU-GPU heterogeneous computing performance. Also, feedback is given for hardware designers of the future generation CPU-GPU heterogeneous computing system.

In Chapter 5, a machine learning based autotuning technique is used to autotune fourteen parameters related to GPU kernel scheduling, including the number of thread blocks and threads in a block. Both the independent training for single type of GPU and combined training for multiple types of GPU are performed for a single fluid dynamics problem (accelerated by one GPU) on the C2075, P100 and V100 GPU. The training and the testing results indicate that using an artificial neural network has the potential to autotune a large number of parameters, while requiring a very small fraction of samples in a large search space.

In Chapter 6, the code verification for the turbulence modeling in parallel SENSEI is done. Turbulence verification cases including crossterm sinusoidal manufactured solutions and all verification cases from the turbulence modeling resources website [61] are used to justify the proper turbulence modeling implementation of the parallel SENSEI. SENSEI achieves 2nd order accuracy for the crossterm sinusoidal manufactured solutions for the Spalart-Allmaras [11] and Menter's Shear Stress Transport [12] model, in both 2D and 3D. Also, SENSEI matches very well with all the numerical benchmark solutions from CFL3D [62] and FUN3D [63] for all the variables of interest including the total lift coefficient, total drag coefficient, pressure drag coefficient and viscous drag coefficient.

In Chapter 7, the code verification is done for unsteady flow cases in SENSEI. For unsteady flows, a systematic refinement should be performed for both the spatial and temporal spacing to determine the correct overall observed order of accuracy. Since explicit time marching schemes typically require smaller time step size compared to implicit time marching schemes due to stability constraints, multiple implicit schemes such as the Singly-Diagonally Implicit Runge-Kutta multi-step scheme [64, 65] and three point backward scheme [66] are used to mitigate the stability constraints.

Bibliography

[1] Wen-Mei W Hwu. *GPU computing gems emerald edition*. Elsevier, 2011.

[2] NVIDIA. CUDA C++ Programming Guide, 2019. (last accessed on 07/24/20).

[3] Khronos OpenCL Working Group. The OpenCL Specification, 2012. (last accessed on 07/24/20).

[4] OpenACC-Standard.org. *The OpenACC Application Programming Interface.* OpenACC-Standard.org, 2018.

[5] Alex Hutcheson and Vincent Natoli. Memory bound vs. compute bound: A quantitative study of cache and memory bandwidth in high performance applications. In *Technical report, Stone Ridge Technology.* 2011.

[6] Denis Foley and John Danskin. Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[7] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[8] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[9] William L Oberkampf and Christopher J Roy. *Verification and validation in scientific computing.* Cambridge University Press, 2010.

[10] William L Oberkampf and Timothy G Trucano. Verification and validation in computational fluid dynamics. *Progress in aerospace sciences*, 38(3):209–272, 2002.

[11] Steven R Allmaras and Forrester T Johnson. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, pages 1–11, 2012.

[12] Florian R Menter. Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal*, 32(8):1598–1605, 1994.

[13] Florian R Menter, Martin Kuntz, and Robin Langtry. Ten years of industrial experience with the sst turbulence model. *Turbulence, heat and mass transfer*, 4(1):625–632, 2003.

[14] Blaise Barney. Introduction to Parallel Computing, 2020. (last accessed on 07/24/20).

[15] Blaise Barney. OpenMP, 2020. (last accessed on 07/24/20).

[16] Blaise Barney. Message Passing Interface (MPI), 2020. (last accessed on 07/24/20).

[17] N Gourdain, L Gicquel, M Montagnac, O Vermorel, M Gazaix, G Staffelbach, M Garcia, JF Boussuge, and T Poinsot. High performance parallel computing of flows in complex geometries: I. methods. *Computational Science & Discovery*, 2(1):015003, 2009.

[18] N Gourdain, L Gicquel, G Staffelbach, O Vermorel, Florent Duchaine, JF Boussuge, and Thierry Poinsot. High performance parallel computing of flows in complex geometries: Ii. applications. *Computational Science & Discovery*, 2(1):015004, 2009.

[19] Zdravko Krpic, Goran Martinovic, and Ivica Crnkovic. Green hpc: Mpi vs. openmp on a shared memory system. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 246–250. IEEE, 2012.

[20] Pablo D Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid mpi–openmp scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, 37(6-7):316–326, 2011.

[21] Yu-Yong Jiao, Qiang Zhao, Long Wang, Gang-Hai Huang, and Fei Tan. A hybrid mpi/openmp parallel computing model for spherical discontinuous deformation analysis. *Computers and Geotechnics*, 106:217–227, 2019.

[22] Dana A Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Computing*, 39(1):1–20, 2013.

[23] Erich Elsen, Patrick LeGresley, and Eric Darve. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

[24] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.

[25] Tobias Brandvik and Graham Pullan. Acceleration of a 3d euler solver using commodity graphics hardware. In *46th AIAA aerospace sciences meeting and exhibit*, page 607, 2008.

[26] Lixiang Luo, Jack R Edwards, Hong Luo, and Frank Mueller. Performance assessment of a multiblock incompressible navier-stokes solver using directive-based gpu programming in a cluster environment. In *52nd Aerospace Sciences Meeting*, 2013.

[27] Yidong Xia, Jialin Lou, Hong Luo, Jack Edwards, and Frank Mueller. Openacc acceleration of an unstructured cfd solver based on a reconstructed discontinuous galerkin method for compressible flows. *International Journal for Numerical Methods in Fluids*, 78(3):123–139, 2015.

[28] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on gpus with openacc and cuda fortran implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016.

[29] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143. IEEE, 2013.

[30] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Mpi+

openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. *Computer Physics Communications*, 2018.

[31] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[32] Dominic D Chandar, Jayanarayanan Sitaraman, and Dimitri J Mavriplis. A hybrid multi-gpu/cpu computational framework for rotorcraft flows on unstructured overset grids. In *21st AIAA Computational Fluid Dynamics Conference*, page 2855, 2013.

[33] Luke Domanski, Tomasz Bednarz, Tim E Gureyev, Lawrence Murray, Emma Huang, and John A Taylor. Applications of heterogeneous computing in computational and simulation science. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 382–389. IEEE, 2011.

[34] L Domanski, T Bednarz, P Vallotton, and J Taylor. Heterogeneous parallel 3d image deconvolution on a cluster of gpus and cpus. In *19th Int'l Congress on Modelling and Simulation, Perth, Australia,[Online, cited Aug 1, 2013] http://mssanz. org. au/modsim2011 A*, volume 8, 2013.

[35] Xavier Alvarez, Andrey Gorobets, and F Xavier Trias. Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers. In *7th European Conference on Computational Fluid Dynamics*, 2018.

[36] Brent P Pickering, Charles W Jackson, Thomas RW Scogland, Wu-Chun Feng, and Christopher J Roy. Directive-based GPU programming for computational fluid dynamics. *Computers & Fluids*, 114:242–253, 2015.

[37] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *Proceedings of the 22nd*

international conference on Parallel architectures and compilation techniques, pages 257–267. IEEE, 2013.

[38] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. Masif: Machine learning guided auto-tuning of parallel skeletons. In *20th Annual International Conference on High Performance Computing*, pages 186–195. IEEE, 2013.

[39] Thomas L Falch and Anne C Elster. Machine learning-based auto-tuning for enhanced performance portability of opencl applications. *Concurrency and Computation: Practice and Experience*, 29(8):e4029, 2017.

[40] Xuewen Cui and Wu-chun Feng. Iterml: Iterative machine learning for intelligent parameter pruning and tuning in graphics processing units. *Journal of Signal Processing Systems*, pages 1–13, 2020.

[41] Abdul-Sattar J Al-Saif and Assma J Harfash. A new approximate analytical solutions for two-and three-dimensional unsteady viscous incompressible flows by using the kinetically reduced local navier-stokes equations. *Journal of Applied Mathematics*, 2019, 2019.

[42] Seth C Spiegel, HT Huynh, and James R DeBonis. A survey of the isentropic euler vortex problem using high-order methods. In *22nd AIAA Computational Fluid Dynamics Conference*, page 2444, 2015.

[43] WH Hui. Exact solutions of the unsteady two-dimensional navier-stokes equations. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, 38(5):689–702, 1987.

[44] Abdullah Shah, Li Yuan, and Aftab Khan. Upwind compact finite difference scheme for time-accurate solution of the incompressible navier–stokes equations. *Applied Mathematics and Computation*, 215(9):3201–3213, 2010.

[45] Maurizio Tavelli and Michael Dumbser. A staggered space–time discontinuous galerkin method for the three-dimensional incompressible navier–stokes equations on unstructured tetrahedral meshes. *Journal of Computational Physics*, 319:294–323, 2016.

[46] Ronald L Panton. *Incompressible flow.* John Wiley & Sons, 2013.

[47] Frank M White and Isla Corfield. *Viscous fluid flow*, volume 3. McGraw-Hill New York, 2006.

[48] Henrik Tryggeson. *Analytical vortex solutions to Navier-Stokes equation.* PhD thesis, Växjö University Press, 2007.

[49] CY Wang. Exact solutions of the unsteady navier-stokes equations. *Applied Mechanics Reviews;(United States)*, 42(CONF-8901202–), 1989.

[50] Robert G Deissler. Unsteady viscous vortex with flow toward the center. 1965.

[51] Tapan K Sengupta, Nidhi Sharma, and Aditi Sengupta. Non-linear instability analysis of the two-dimensional navier-stokes equation: The taylor-green vortex problem. *Physics of Fluids*, 30(5):054105, 2018.

[52] Subrahmanya Pavan Kumar Veluri. *Code verification and numerical accuracy assessment for finite volume CFD codes.* PhD thesis, Virginia Tech, 2010.

[53] Kambiz Salari and Patrick Knupp. Code verification by the method of manufactured solutions. Technical report, Sandia National Labs., Albuquerque, NM (US); Sandia National Labs ..., 2000.

[54] Chris Roy, Curt Ober, and Tom Smith. Verification of a compressible cfd code using the method of manufactured solutions. In *32nd AIAA Fluid Dynamics Conference and Exhibit*, page 3110, 2002.

[55] Stephane Etienne, Andre Garon, and Dominique Pelletier. Code verification for unsteady flow simulations with high order time-stepping schemes. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 169, 2009.

[56] Michael L Minion and RI Saye. Higher-order temporal integration for the incompressible navier–stokes equations in bounded domains. *Journal of Computational Physics*, 375:797–822, 2018.

[57] Kintak Raymond Yu, Stéphane Étienne, Alexander Hay, and Dominique Pelletier. Code verification for unsteady 3-d fluid–solid interaction problems. *Theoretical and Computational Fluid Dynamics*, 29(5-6):455–471, 2015.

[58] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[59] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model implementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[60] Weicheng Xue, Hongyu Wang, and Christopher J Roy. Code verification for 3d turbulence modeling in parallel sensei accelerated with mpi. In *AIAA Scitech 2020 Forum*, page 0347, 2020.

[61] Christopher Rumsey. Turbulence modeling resources, 2019. (last accessed on 12/02/19).

[62] Christopher Lockwood Rumsey, Robert T Biedron, and James Lee Thomas. Cfl3d: Its history and some recent applications. 1997.

[63] Robert T Biedron, Jan Renee Carlson, Joseph M Derlaga, Peter A Gnoffo, Dana P Hammond, William T Jones, Bil Kleb, Elizabeth M Lee-Rausch, Eric J Nielsen, Michael A Park, et al. Fun3d manual: 13.6. 2019.

[64] Uri M Ascher, Steven J Ruuth, and Raymond J Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[65] Christopher A Kennedy and Mark H Carpenter. Diagonally implicit runge-kutta methods for ordinary differential equations. a review. 2016.

[66] JC Wu, LT Fan, and LE Erickson. Three-point backward finite-difference method for solving a system of mixed hyperbolic—parabolic partial differential equations. *Computers & chemical engineering*, 14(6):679–685, 1990.

Chapter 2

Multi-GPU Performance Optimization of a CFD Code using OpenACC

Weicheng Xue[1] and Christopher J. Roy[2]

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. All the performance optimizations were developed and implemented by the first author. All the results were collected by the first author.
- Christopher J. Roy (second author): The second author provided valuable feedback for this study and comments for this manuscript.
- This work has been published: Weicheng Xue and Christopher J. Roy, Multi-GPU Performance Optimization of a CFD Code using OpenACC, Concurrency and Com-

[1]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.

[2]Professor, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 330, Virginia Tech, 460 Old Turner St, AIAA Associate Fellow.

putation: Practice and Experience. http://dx.doi.org/10.1002/cpe.6036

Abstract

This paper investigates the multi-GPU performance of a 3D buoyancy driven cavity solver using MPI and OpenACC directives on multiple platforms. The paper shows that decomposing the total problem in different dimensions affects the strong scaling performance significantly for the GPU. Without proper performance optimizations, it is shown that 1D domain decomposition scales poorly on multiple GPUs due to the noncontiguous memory access. The performance using whatever decompositions can be benefited from a series of performance optimizations in the paper. Since the buoyancy driven cavity code is communication-bounded on the clusters examined, a series of optimizations both agnostic and tailored to the platforms are designed to reduce the communication cost and improve memory throughput between hosts and devices efficiently. First, the parallel message packing/unpacking strategy developed for noncontiguous data movement between hosts and devices improves the overall performance by about a factor of 2. Second, transferring different data based on the stencil sizes for different variables further reduces the communication overhead. These two optimizations are general enough to be beneficial to stencil computations having ghost exchanges. Third, GPUDirect is used to improve the communication on clusters which have the hardware and software support for direct communication between GPUs without staging the memory on the hosts. Finally, overlapping the communication and computations is shown to be not efficient on multi-GPUs if only using MPI or MPI+OpenACC. Although we believe our implementation has revealed enough communication and computation overlap, the actual running does not utilize the overlap well due to a lack of enough asynchronous progression.

Keywords: Multi-GPU, OpenACC, MPI, Domain Decomposition, Performance Optimization, GPUDirect

2.1 Introduction

Computational Fluid Dynamics (CFD) is a method which can be used to solve physical problems in the field of fluids, usually requiring a lot of computation. In order to obtain more accurate numerical solutions for challenging problems, researchers are using very fine meshes or high-order schemes which require much better resources such as a larger memory and faster processor. However, generally the memory cannot be infinitely large and a processor cannot hold an infinite number of transistors. These limitations may require codes to be written in a data parallel way, i.e., decomposing a big problem into small pieces and distributing these small problems to multi/many-cores or even accelerators such as GPUs. Applying high performance computing (HPC) in the CFD [1] area is necessary and has aroused CFD researchers' interest.

On multicore/manycore CPU systems or equivalent, there are three common paradigms for HPC: OpenMP, MPI, and hybrid MPI+OpenMP. OpenMP [2] is designed for shared memory systems so that the data can be shared among all threads but this comes with the risk of race conditions to exist if multiple threads are modifying the same data. Also, the scaling performance across multiple nodes or sockets may be poor on distributed memory systems (more usually used than shared memory systems). MPI [3] is a message passing standard designed for various platforms including shared and distributed memory architectures. Data can be moved between processors through sending and receiving messages. However, programming with MPI is more complicated than with OpenMP as it requires extra care to decompose the problem well and implement efficient communications. Usually, communication may be a significant bottleneck when the codes are scaled up to a lot of processors. Hybrid MPI+OpenMP methods [4] are therefore proposed to combine the advantages of MPI and OpenMP. The hybrid model is a good match with modern multicore/manycore architectures, as it can be programmed efficiently using two levels of parallelism: MPI for the inter-node/socket communication and OpenMP for the intra-node/socket computation and communication. However, hybrid MPI+OpenMP cannot be easily used on GPUs directly due to a lack of full support of OpenMP 4.0 (or later) from the compiler development.

In addition to the CPU computing, GPU computing have been gaining a lot of interest [5]. This attention is because of the GPU's high compute capability and memory bandwidth as well as their low power consumption. A single GPU have thousands of threads, therefore numerous threads can execute the same instruction simultaneously on multiple data points, known as single instruction multiple threads (SIMT). When executing a program on the GPU, the highly compute-expensive portion of a program is offloaded to the GPU. Then numerous threads on the GPU execute the code simultaneously to achieve a high speedup. File I/O, branch controls, printout, etc. remain on the CPU since the CPU has the flexibility to perform these tasks while GPUs are not as efficient for these complex tasks (even if they are possible). One important thing to mention here is that although GPUs provides higher memory bandwidth than CPUs, different memory access patterns such as array of structures (AOS) or structure of arrays (SOA) may significantly affect the actual memory throughput [6], which should be considered carefully.

Three language extensions/libraries are widely applied to port codes to GPUs [7]. They are OpenCL, CUDA and OpenACC. OpenCL and CUDA are C/C++ with some extensions while OpenACC is a compiler directive-based interface, similar to OpenMP. OpenCL and CUDA are low level programming models so that they require users to have some background in computer architecture systems. Also, programming with OpenCL or CUDA is difficult, as users need to rewrite and tune their codes on various GPUs every time. CUDA gets a strong support from NVIDIA but it cannot be compatible well with other GPUs such as AMD. OpenCL is open source and we found that it has not been commonly used in real world GPU-accelerated CFD codes, possibly due to the high complexity of programming and a lack of good ecosystem support. Different from OpenCL and CUDA, OpenACC is a high level programming model that enables users to accelerate their CFD codes more readily on various GPUs without intruding their legacy codes completely. Programmers using OpenACC [8] can be somewhat agnostic about the GPU architecture compared to using OpenCL and CUDA because compilers such as those developed by PGI (acquired by NVIDIA) can hide a lot of details and decide how to optimize the code (although it may not be optimal). Also, because of its directive-based feature and good support for portability, OpenACC can be much easier to use on various platforms compared to CUDA. We will also show this benefit because there is little code modification across platforms. However, to gain good performance across different platforms, the features of the architecture and some low level optimizations should be taken into account. Apart from the options mentioned, OpenMP can be a potential viable choice for the GPU once the development of compilers catch up in the future. Because OpenACC provides an easy way of programming [9], a good feature for portability across platforms [10] and good parallel performance if optimized enough [11], OpenACC was applied to port our CFD code to the GPU.

OpenACC has already been used for various GPU-accelerated CFD codes or related applications. Gong et al. [12] presented an optimized OpenACC version for NekBone, which is a core kernel of the incompressible Navier-Stokes solver Nek500, based on their group's prior work. They ported the optimized code to multiple GPU systems and obtained a parallel efficiency of 79.9% on 1024 GPUs. However, the code they worked on is just a kernel, not a complete CFD code. Hoshino et al. [11] found that although OpenACC is 50% slower than CUDA for a naive implementation, the gap can be decreased to only 2% after careful manual optimizations. They also pointed out that there are some intrinsic deficiencies of OpenACC, such as a lack of interface for shared memory and inter-thread communication. Searles et al. [13] studied a wavefront based mini-application for a production code for nuclear reactor modeling. It is interesting and rare to see in their work that the OpenACC implementation is even slightly faster than CUDA. Their work mainly focused on exploring complex parallel patterns in their code and exploring the scalability using MPI across different platforms. In summary, OpenACC is easy to use and also good for portability across different platforms, however to obtain good performance, careful pertinent optimizations for an application may need to be designed.

To assess the performance of a code accelerated by the CPU or GPU, weak scaling and strong scaling performance are often measured. The major difference between the two scalings is whether one keeps the total problem size fixed (strong scaling) or the problem size per processor fixed (weak scaling), while adding more processors. Obviously maintaining the strong scaling efficiency is more challenging. Commonly both scalings are investigated to satisfy different situations such as solving a fixed problem as fast as possible (strong scaling) or solving as big of a problem as possible (weak scaling). In both situations we want to max out all compute nodes or resources to gain the maximum speedup. In the CFD area, we are more interested in the weak scaling performance as we hope to solve a larger and complex problem faster, if more compute resources are available. However, there are a lot of other occasions in which small problems need to be solved if the requirement for numerical accuracy is not high. Therefore, both the weak scaling and strong scaling performance are measured in this paper.

Prior to the work presented in this paper, Pickering et al. [14] examined the process of applying OpenACC to a 2D CFD code using both single precision and double precision. They also applied OpenMP's fork/join execution model to scale the performance up to 4 NVIDIA C2070 GPUs with a strong scaling efficiency of 95%. Instead of using the OpenMP+OpenACC model, Baghapour et al. [15] switched to the MPI+OpenACC model and scaled a 3D CFD code well up to both 32 CPUs and 32 GPUs on a distributed cluster.

In their work, they used 1D domain decomposition to distribute the workload to different processors and increased the grid size in only one dimension for their weak scaling performance. Xue et al. [16] compared multi-CPU/GPU performance using 3D, 2D and 1D decompositions and gave a primitive analysis of their differences. Also, two performance optimizations including an pack/unpack method for data exchange between hosts and devices was designed, and this pack/unpack method was proved to improve the performance using 3D decomposition greatly on a platform using old GPUs (NVIDIA C2070). In the current paper, the effects of multiple factors such as the platform difference, decomposition methods, pack/unpack, strong v.s. weak scaling and GPUDirect will be investigated. Also, the limitations of overlapping communication and computation if applying MPI or MPI+OpenACC are presented.

2.2 CFD Code: Buoyancy Driven Cavity Solver

The 3D Buoyancy Driven Cavity (BDC) problem has a cubic domain, a vertical wall and its opposing wall have different temperatures, and the horizontal walls are adiabatic. A gravitational force is added to the air in the square cavity. Heat flux caused by the temperature difference leads to small density changes in the fluid (Boussinesq approximation), and the buoyancy effect (density change) causes the fluid to convect in the cavity.

The CFD code written with Fortran 2003/2008 in the paper solves the classic 3D BDC problem [15], which is a system of 3D incompressible Navier-Stokes equations. An artificial compressibility method developed by Chorin [17] is used. The artificial viscosity term makes the system of equations to be hyperbolic so that steady state solution can be obtained through time marching. The CFD code uses a first order Euler explicit scheme for temporal discretization, and a second-order central-difference scheme with an artificial dissipation term for spatial discretization. The artificial dissipation term is applied to the continuity equation to alleviate odd-even decoupling. The numerical damping term is based on the fourth-derivative of pressure and is discretized using a second-order central finite difference scheme. This BDC code is a proxy code of another complicated CFD code called Structured Euler Navier-Stokes Explicit Implicit solver (SENSEI) [16, 18, 19]. We want to know the power of leveraging the GPU to accelerate CFD codes, so we first port the BDC code to the GPU and have some performance optimizations there (Some performance optimizations such as the pack/unpack method can be generally applied to other codes). SENSEI is a finite volume code so it only focuses on the 2nd order for the spatial terms. While for this BDC code, since the spatial order is 2nd using the artifi cial viscosity method, the stencil size is fixed and cannot be changed. Therefore, only two ghost cells are required for this BDC code. The discretized form of the system of the governing equations can be written as,

$$\frac{1}{\beta^2}\frac{\partial p}{\partial t} + \rho\frac{\partial V_i}{\partial x_i} = \epsilon_i \frac{\partial^4 p}{\partial x_i^4} \tag{2.1}$$

$$\frac{\partial V_j}{\partial t} + V_i\frac{\partial V_j}{\partial x_i} = -\frac{1}{\rho}\frac{\partial p}{\partial x_j} + \nu\frac{\partial^2 V_j}{\partial x_i^2} + \sigma(T - T_\infty)g_j \tag{2.2}$$

$$\frac{\partial T}{\partial t} + V_i\frac{\partial T_j}{\partial x_i} = \alpha\frac{\partial^2 T_j}{\partial x_i^2} \tag{2.3}$$

where β is an artificial compressibility parameter calculated using the local velocity magnitude along with a reference velocity defined by the user, ϵ are numerical dissipation coefficients controlling stability, ν is the kinematic viscosity of the fluid, and α is the thermal dissipation rate.

In this paper, the size of the cavity is 0.05 m in all three dimensions. Pressure is extrapolated to the ghost cells at adiabatic walls using a one-sided second order scheme. Temperature is similarly extrapolated to the horizontal wall ghost cells using a second order scheme.

Pressure is rescaled at the center point of the cavity in every iteration. The meshes used for the BDC code are uniform and their size range from 32^3 to 1024^3. The Rayleigh number is set to be 100,000 for the convection problem. Most of the constant settings affecting the flow do not affect the parallel performance.

2.3 Implementation

2.3.1 Stencil Computation

In the BDC code studied here, since we use the fourth-derivative pressure dissipation term and a second-order central-difference scheme for all spatial derivatives, the numerical stencil size in one dimension for the pressure is 5 and for other primitive variables is 3. This can be utilized to design an optimization to reduce the data exchange across processors (*Optimized V2*, will be introduced later). Fig. 2.1 shows the stencils for a node in the computational domain. The iterative residual calculation in the explicit CFD code is intrinsically one kind of stencil computation. For each node, it needs the data in its two stencils to compute and fill in the residual array, which is later used to update the primitive variables. There should be a nested loop over all nodes in the spatial domain, and a time step loop containing all stencil computations to iterate in the pseudo time domain. When programming, data locality should be considered to use cache more efficiently. For a 3D array $A(i, j, k)$ in Fortran, i is set to be incremented fastest (the innermost loop) and k the slowest (the outermost loop). This layout is also good for the GPU, as the GPU prefers the coalesced memory access pattern in which contiguous threads in a thread block operate on the consecutive memory locations. It should be noted here that the index directions (i, j and k) are aligned with the spatial directions (x, y and z) in this problem. Therefore, the use of (i, j, k) are mixed with the use of (x, y, z) in the paper.

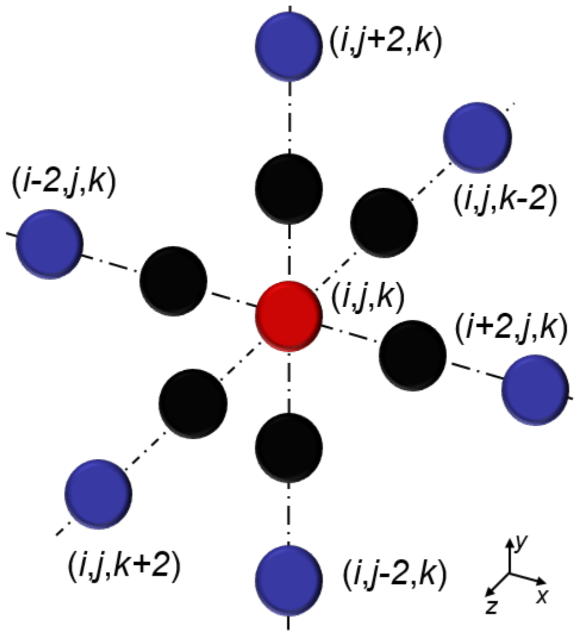


Figure 2.1: Stencil (black+red: velocity and temperature stencil, blue+black+red: pressure stencil)

This BDC code uses struct of arrays (SOA), instead of arrays of struct (AOS). The reason of using SOA is to allow efficient SIMD loads and stores and avoiding scatter-gather addressing [14].

2.3.2 Domain Decomposition

Many methods can be used to decompose a computational domain such as structured partitioning [20] and graph partitioning [21]. For a CFD problem with single-block structured grid such as a BDC problem running on pure CPUs or pure GPUs, there are three structured ways: 1D decomposition, 2D decomposition and 3D decomposition. Which way of decomposing the domain performs the best greatly depends on the application and computer architecture, i.e., the optimal decomposition can vary under different conditions. On one hand, the surface-area/volume ratio determines the total size of data transferred between processors and the total size of ghost cells, and 3D decomposition has the lowest area/volume ratio (least ghost cells). On the other hand, the frequency of data transfers between processors can greatly affect the performance, especially when the memory bandwidth or latency issue becomes important, and 1D decomposition has the least times of data transfers. Besides, for stencil computations like ours, 1D or even 2D decomposition may generate too thin slices that invalidates the spatial discretization scheme if scaling up to a large number of processors (strong scaling). Thus, it is worthwhile to investigate the effect of various domain decompositions on different platforms. Xue et al. [16] showed that 2D decomposition scales better up to 32 CPUs compared to 1D and 3D decomposition on a platform, and 3D decomposition can outperform 1D decomposition if applying optimizations. However, they did not show the comparison between 3D decomposition and 2D decomposition. Also, they only tested the code on a single platform having old GPUs. An example of the 3D decomposition adopted in this paper is shown in Fig. 2.2. Each processor is given a decomposed block, with each cutting face contacting a neighbour block. Ghost nodes are used to store decomposed boundary information transferred from neighbours. 1D and 2D decompositions are similar as 3D decomposition but have fewer decomposed directions and fewer decomposed boundaries.

For a given number of processors, there may be many combinations for either 1D, 2D or 3D decomposition. For general situations, we try to decompose the domain evenly in all the available dimensions but decompose more in the slowest stride index direction, to preserve more contiguous data after decomposition. This method is designed to 1) divide the domain in the available dimensions as evenly as possible, 2) utilize enough processors, 3) decompose with a priority along the k dimension first, then j, and finally i, as Fortran is a column-majored language.

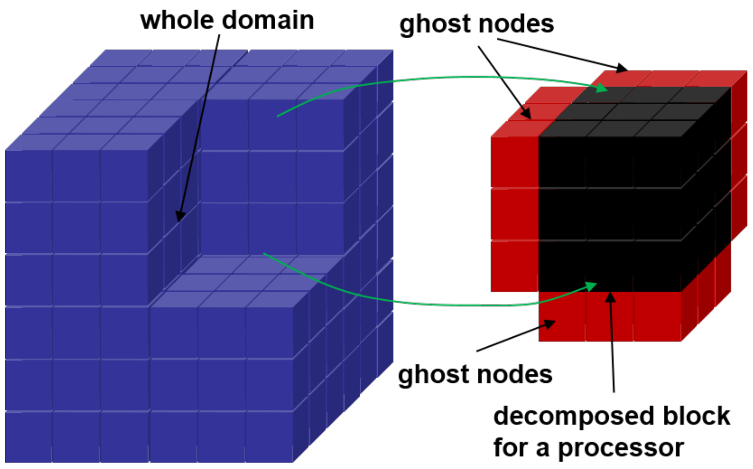


Figure 2.2: 3D domain decomposition

2.3.3 Hardware Configuration

HokieSpeed Although now decommissioned in June 2017, HokieSpeed [22] was a cluster at Virginia Tech and was previously in the list of Green500. HokieSpeed [22] had 204 nodes using a quad data rate InfiniBand interconnect. Each node was outfitted with 24GB memory, two six-core Xeon E5645 CPUs and two NVIDIA M2015/C2050 GPUs. Every GPU had 14 multiprocessors (MP) and 3GB memory. The peak bandwidth to the 3GB shared memory was 148.4GB/s. Every MP had 32 CUDA cores, 48KB shared memory and 16KB L1 cache. All the access to the global memory went through the L2 cache of size 512KB. The peak double precision performance was 513 GFLOPS. The compilers used on HokieSpeed were PGI 15.7 and Open MPI 1.10.0. A compiler optimization of -O4 was used.

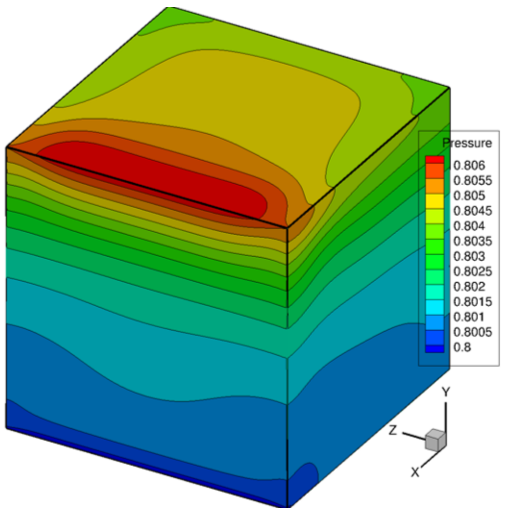
NewRiver NewRiver [23] is a cluster at Virginia Tech (VT). It has 39 GPU nodes shared by the whole VT community. On NewRiver [23], each of these nodes is equipped with two Intel Xeon E5-2680v4 (Broadwell) 2.4GHz CPU (28 cores/node in all), 512 GB memory, and two NVIDIA P100 GPUs. Each NVIDIA P100 GPU is capable of up to a theoretical 4.7 TeraFLOPS of double-precision performance. The NVIDIA P100 GPU offers much higher GFLOPS and memory bandwidth compared with the NVIDIA C2050 GPU on HokieSpeed. The modules used on NewRiver are PGI 17.5, CUDA 8.0.61 and Open MPI 2.0.0 or MVAPICH2-GDR 2.3b. It should be mentioned that MVAPICH2-GDR 2.3b is a CUDA-aware MPI wrapper compiler which supports GPUDirect (if this feature is turned on). An compiler optimization of -O4 is used. The maximum number of nodes which can be used is 12, but only 8 is used in this paper. Thus, the maximum number of GPUs used on the NewRiver cluster is 16 (when using both Open MPI 2.0.0 or MVAPICH2-GDR 2.3b). If not specified, Open MPI 2.0.0 is used.

Cascades Cascades [24] is another cluster at VT. It has 40 GPU nodes shared by the whole VT community. On Cascades [24], each of these nodes is equipped with two Intel Skylake Xeon Gold 3 Ghz CPUs (24 cores/node in all), 768 GB memory, and two NVIDIA V100 GPUs. Each NVIDIA V100 GPU is capable of up to 7.8 TeraFLOPS of double precision performance, which is 66% higher than the P100 GPU on the NewRiver cluster. The NVIDIA V100 GPU offers the highest GFLOPS and memory bandwidth among the GPUs we used. The modules used on Cascades are PGI 18.1, CUDA 8.0.61 and Open MPI 3.0.0 or MVAPICH2-GDR 2.3b. An compiler optimization of -O4 is used. Similarly, the maximum number of GPUs used on the Cascades cluster is 16 when using Open MPI 3.0.0. However, when switching to MVAPICH2-GDR 2.3b, since Cascades uses ”srun” to run MPI programs instead of using ”mpirun_rsh” (as Slurm is used on the Cascades), the maximum number of GPUs which can be used is only 8 (using more would cause the efficiency to drop to about 1%, which is not reasonable and caused by some unknown issues related to Slurm). If not specified, Open MPI 3.0.0 is used.

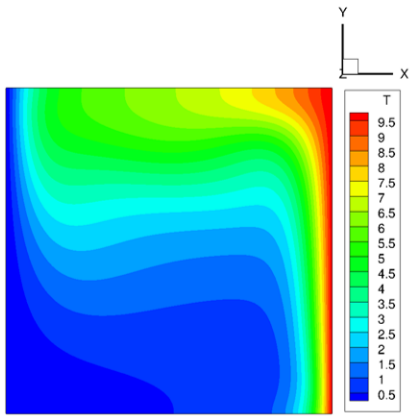
2.4 Results

2.4.1 BDC Solution

Before presenting any performance results, the first indispensable step for any parallelization or optimization is to quantify the solution difference between the serial solution and the parallel solution. An iteratively converged double precision solution using 8 GPUs on NewRiver for a 256^3 mesh is given in Fig. 2.3. All the residuals for temperature and velocities components (except for the pressure) have been compared to the relevant CPU serial residuals until converged. The relative difference of the velocity components between the serial CPU and the multi-GPU version are withing the round-off range (less than 10^{-10}), and the relative difference of the temperature is less than 10^{-9}. The difference of the serial solution and the parallel solution is caused by a accumulation of round-off errors. Note that since the BDC code uses pressure rescaling at the center of the cavity in every time step, i.e., the gauge pressure (the cavity center pressure) value is deducted from the pressure solution for every time step, the reference pressure is difficult to know so that the relative difference for the pressure is not calculated (the maximum absolute value difference between the CPU serial pressure and multi-GPU pressure is less than 10^{-8}, which is the final L_2 residual norm round-off level for the pressure). The pressure rescaling is needed in this BDC code, otherwise all boundary conditions are Neumann boundary conditions, which will make the solution indefinite. For all the optimizations presented in this paper, parallel solution correctness is always guaranteed.



(a) 3D pressure contour



(b) 2D temperature contour at y=0.025 m

Figure 2.3: 3D BDC solution

2.4.2 Scaling Performance Metrics

Two basic metrics used in this paper are parallel speedup and efficiency. Speedup denotes how much faster the parallel version is compared to the serial version of the code, while efficiency represents how efficiently the processors are used. They are defined as follows,

$$\text{speedup} = \frac{t_{serial}}{t_{parallel}} \tag{2.4}$$

$$\text{efficiency} = \frac{speedup}{np} \tag{2.5}$$

where np is the number of processors (CPUs or GPUs).

In order for the performance of the code to be measured and compared well on different platforms and for different problem sizes, the wall clock time per iteration step is converted to a ssspnt (scaled size steps per np time) value which is defined in Eq.2.6. This metric has some advantages. First, GFLOPS requires knowing the number of operations while ssspnt does not require. In most codes, it is usually difficult to know the number of operations.

Second, efficiency comparison across different platforms is not intuitive as how fast the program runs is still unknown, but ssspnt is clearer for knowing the absolute speed. Also, the benefit applies to strong and weak scaling performance comparison if using ssspnt. For example, a linear scaling problem has a constant ssspnt value but the values can be different for the strong and weak scaling. Using ssspnt, different problems, platforms, strong and weak scaling performance can be compared directly. In this paper, when gaining productive performance results, unnecessary I/O such as writing out solutions to files are turned off, which is commonly applied when testing the performance in literature.

$$\text{ssspnt} = s\frac{size \times steps}{np \times time} \tag{2.6}$$

where s is a scaling factor which scales the smallest platform ssspnt to the range of [0,1]. In this paper, s is set to be 10^{-7} for all test cases. *size* is the problem size, *steps* is the iteration steps and *time* is the program wall clock time for *steps* iterations.

For every *time*, it is measured at least three times, consecutively (so the time gap between different runs is very short, considering that the network traffic may vary a little bit in different time). Usually the difference is much less than 1% (we also make sure every run is at least longer than 120 s, but the difference for each *time* point is usually less than 1 s). We usually use the median value (if the deviation is very small, e.g., less than 0.5% or so) or the mean value (if the deviation is larger than 0.5%) as the final *time* result. Also, every time when we have a systematic run of the test case, some previously data points are randomly selected and run again, in order to make sure that all the data is consistent. In this way, reliable data are acquired as all the data points have been run and verified for more than three times.

2.4.3 Grid Growth for Weak Scaling

In the weak scaling analysis, the problem size needs to be increased accordingly when the number of processors increases. However, the way the problem scales can vary. For the BDC codes, since the problem is a 3D problem, the problem can increase in either 1D, 2D, or 3D. Therefore, we will investigate the effect of how problem size grows on the weak scaling performance. Two types of grid growth for the weak scaling are applied, seen in Table 2.1 and Table 2.2. In Table 2.1, it shows that the problem grid growth strategy is fixed (seen in the column 1 in Table 2.1), no matter whether the domain decomposition is 1D, 2D or 3D (column 2 to column 4, respectively). However, Table 2.2 scales the problem size in accordance with the way the number of processors grow, i.e., the grid size in each dimension is different for 1D, 2D and 3D domain decomposition. For example, if using 8 processors, the problem size for 3D decomposition is $512 \times 512 \times 512$, for 2D decomposition it is $256 \times 512 \times 1024$, and for 1D decomposition it is $256 \times 256 \times 2048$ or $2048 \times 256 \times 256$ (as the processor dims can be either $1 \times 1 \times 8$ or $8 \times 1 \times 1$). Both methods of grid growth are applied in this paper. If not specified, the paper uses the grid growth in Table 2.2 for the weak scaling.

Table 2.1: Grid growth type 1

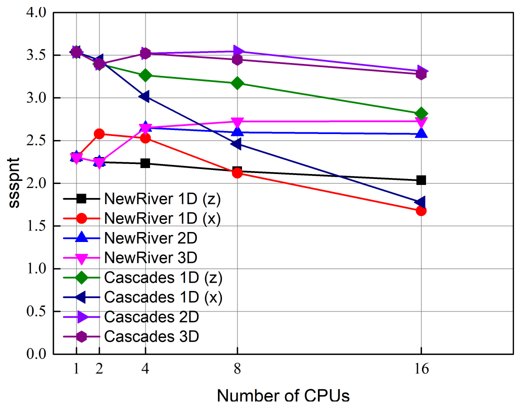
Problem size	3D decomposition	2D decomposition	1D decomposition
(256,256,256)	(1,1,1)	(1,1,1)	(1,1,1)
(256,256,512)	(1,1,2)	(1,1,2)	(1,1,2) or (2,1,1)
(256,512,512)	(1,2,2)	(1,2,2)	(1,1,4) or (4,1,1)
(512,512,512)	(2,2,2)	(1,2,4)	(1,1,8) or (8,1,1)
(512,512,1024)	(2,2,4)	(1,4,4)	(1,1,16) or (16,1,1)
(512,1024,1024)	(2,4,4)	(1,4,8)	(1,1,32) or (32,1,1)

Table 2.2: Grid growth type 2

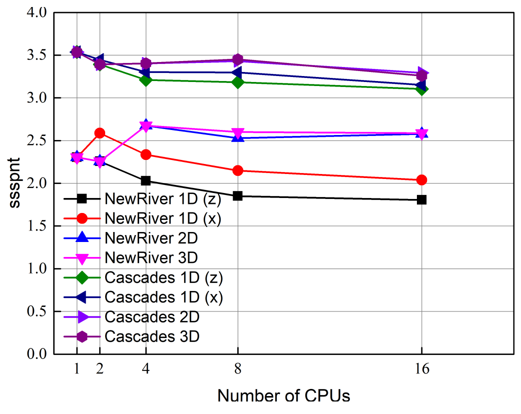
# of processors	3D decomposition	2D decomposition	1D decomposition
1	(256,256,256)	(256,256,256)	(256,256,256)
2	(256,256,512)	(256,256,512)	(256,256,512) or (512,256,256)
4	(256,512,512)	(256,512,512)	(256,256,1024) or (1024,256,256)
8	(512,512,512)	(256,512,1024)	(256,256,2048) or (2048,256,256)
16	(512,512,1024)	(256,1024,1024)	(256,256,4096) or (4096,256,256)
32	(512,1024,1024)	(256,1024,2048)	(256,256,8192) or (8192,256,256)

2.4.4 Multi-CPU Scaling Performance

A systematic multi-CPU scaling performance test is performed on all the three platforms mentioned earlier in this paper. CPU strong scaling and weak scaling performance using 1D, 2D and 3D decompositions are shown in Fig. 2.4. Here CPUs are added by sockets, i.e., adding a certain number of sockets every time (only using one CPU in each socket, similar to adding GPUs as every socket only has one GPU), or equivalently setting the processor per node (ppn) to be 2. Fig. 2.4 highlights a tradeoff that exists for the different domain decomposition techniques. Choosing a domain decomposition scheme is a balance between maintaining a small surface to volume ratio of the subdomains and minimizing the number of neighbors for each subdomain. From Fig.2.4, 1D decomposition generally performs the worst for both the strong and weak scaling on NewRiver and Cascades. 1D decomposition has only 2 neighbors but it has the highest surface to volume ratio meaning that there is a lot of data to transfer between the blocks. 2D and 3D decomposition perform the best for the strong scaling and weak scaling, respectively. 3D decomposition has the smallest surface to volume ratio but has 6 neighbors meaning it has to perform 6 communications each with their own overhead. The weak scaling performance decays more slowly than the strong scaling performance, which is reasonable as the CPU has more work to do. It is also found that decomposing in the x dimension should be avoided for the CPU in strong scaling as the performance deteriorates faster compared to other decompositions, which is especially obvious on the Cascades cluster.



(a) CPU strong scaling

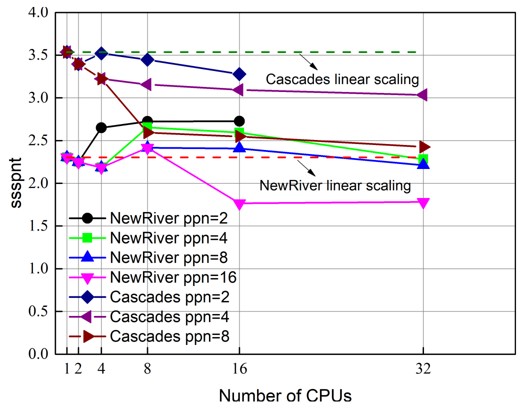


(b) CPU weak scaling

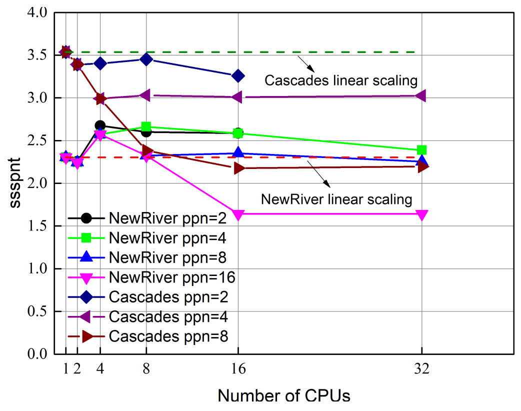
Figure 2.4: Multi-CPU scaling using different decompositions

Also, Fig. 2.4 shows that a super-linear scaling occurs on the NewRiver cluster. To investigate why there is super-linear phenomenon, the ppn value is changed. Fig. 2.5 shows the effect of ppn on the performance. For both the strong and weak scaling performance shown in Fig. 2.4, since the ppn is 2, the number of CPUs is increased along with other resources such as memory and memory bandwidth increased at the same pace (as the number of sockets increases), which may reduce the communication overhead and latency cost. To test this hypothesis, we also tried setting ppn to be higher than 2 (using less nodes) and did not observe the super-linear scaling. Also, no matter what the ppn is, the Cascades cluster does not show a super-linear scaling. It can be concluded that the super-linear phenomenon depends highly on both the platform communication system and the implementation.

Although multi-CPU implementation is not the focus of this paper, we are interested in the CPU performance comparison between different platforms, which is shown in Fig. 2.6. All



(a) CPU strong scaling



(b) CPU weak scaling

Figure 2.5: The effect of ppn (3D decomposition)

the results here use 3D domain decomposition. It can be found that the platform affects the performance most, not the number of CPUs or whether the scaling is strong or weak. The performance on Cascades is about 1.245 times faster as that on NewRiver, which is very close to the clock rate ratio of 1.25 (3Ghz/2.4Ghz). For the CPU scaling, 3D domain decomposition maintains the efficiency very well so it is recommended for pure CPU implementations.

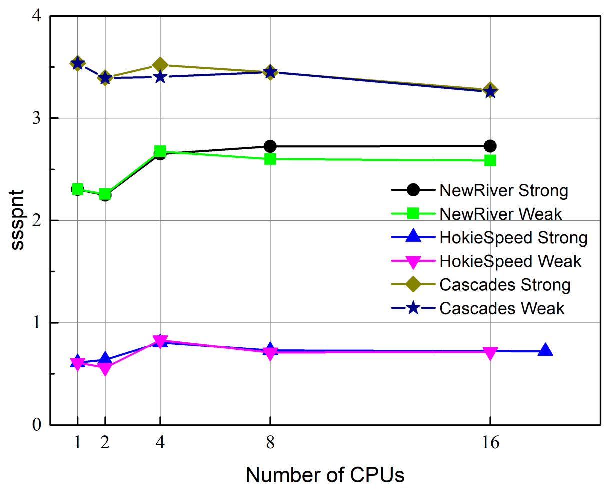


Figure 2.6: Multi-CPU performance comparison across platforms (3D domain decomposition)

2.4.5 Multi-GPU Scaling Performance

The focus of this paper is the multi-GPU implementation and its performance optimizations. There are multiple optimizations of the GPU-accelerated CFD code in this subsection: the first version is a baseline code, which can be regarded as a naive GPU implementation based on the CPU code, and the other versions are incremental optimization versions, with more optimizations based on the previous version. Actual memory bandwidth can be improved greatly as well network communication overhead can be reduced by applying these optimizations. When showing results later, the ”1D”, ”2D” and ”3D” in the legend denotes 1D, 2D and 3D decompositions, respectively. For 1D decomposition, the letter in the parentheses after ”1D” denotes which dimension to decompose. More details about these different GPU-accelerated versions are given as follows.

Baseline The multi-CPU code is directly ported to GPUs by inserting OpenACC directives in the parallel CPU code. This baseline GPU code does not use GPUDirect techniques. Therefore, data on devices need to be updated to/from hosts using !$acc update clauses. Asynchronization clauses are used to reduce some synchronization overhead between hosts and devices. The *Baseline* version uses the MPI_Type_vector function to generate user-defined type for the boundary data that needs to be communicated between processors.

Optimized V1 This version is to improve the actual memory bandwidth and reduce communication cost using 3D decomposition, as we found that the actual bandwidth between the host and the device is very small because of non-contiguous data transfer. As Fortran is a column-majored language, the first index i of a matrix $A(i, j, k)$ denotes the fastest change. If any decomposition exists in the i index direction (3D decomposition or 1D decomposition in i), the decomposition in the i index direction can generate chunks of data (at $j - k$ planes) which are highly non-contiguous. Therefore, the optimization is targeted at solving this issue by converting the non-contiguous data into a temporary contiguous array in parallel and updating this temporary array between hosts and devices using !$acc update clauses. For *Optimized V1*, temporary arrays are created only if decomposition in the direction with the fastest change (i index direction) exists, as decomposition in other index directions can still generate chunks of contiguous data. A pseudo code of how to use buffers is given in Listing. 2.1. The procedure can be summarized as follows:

1. Allocate send/recv buffers only for boundary cells on i planes on devices and hosts if decomposition happens in the i dimension, as the non-contiguous data on i planes makes data transfer very slow.

2. Pack the noncontiguous block boundary data to a buffer on the sender device side. This process can be parallelized using !$acc loop clauses and has little overhead (less than 3%). The host buffer is then updated using !$acc update clauses.

3. Have hosts transfer the data through MPI_Isend/MPI_Irecv calls (which are one-sided non-blocking calls), and block these MPI communication calls using the MPI_Waitall function to finish the data transfer.

4. Update the recv buffer on devices using OpenACC update device clauses and finally unpack the contiguous data stored in recv buffer back to noncontiguous memory on devices, which can also be parallelized using !$acc update clauses.

Listing 2.1: A pseudo code of optimization on non-contiguous data transfer between hosts and devices

```
!$acc data present(send_buffer(:,:), soln(:,:,:,:))
```

```
!Pack send_buffer(:,1) with back boundary data
!$acc parallel loop collapse(4) async(1)
do var = 0, 4
  do k= 0, k_nodes-1
    do j= 0, j_nodes-1
      !Pack up two layers of cells
      do i = 1, 2
        indx = var*k_nodes*j_nodes*2+k*j_nodes*2+j*2+i
        !Interior cell index starts at 3
        send_buffer(indx,1) = soln(i+2,j+1,k+1,var+1)

!Similar routine to pack send_buffer(:,2) with front boundary data
   async(2)

!Update send_buffer on hosts
!$acc update host(send_buffer)
!$acc wait

!Send/Recv between hosts
MPI_IRECV(recv_buffer)
MPI_ISEND(send_buffer)
MPI_WAITALL

!Update recv_buffer on devices
!$acc update device(recv_buffer)
!$acc wait
```

```
!$acc data present(recv_buffer(:,:), soln(:,:,:,:))

!Unpack the data in recv_buffer to soln ghost locations
```

Optimized V2 The use of only one stencil in the *Optimized V1* makes the communication pattern and implementation simpler but may not be efficient. Thus, *Optimized V2* is designed to reduce the amount of data exchanged. Since the pressure requires a larger stencil while other primitive variables do not, We can transfer less (40% for this BDC code) data based on their own stencil size compared to *Optimized V1*. What is more, more overlap of asynchronous communication can be achieved as a big loop is split into two asynchronous loops. A pseudo code of this optimization is given in Listing. 2.2. It should be noted that only changes based on the previous optimization are emphasized in a new Listing throughout this paper.

Listing 2.2: A pseudo code of stencil based communication optimization

```
!$acc data present(send_buffer(:,:), soln(:,:,:,:))

!Pack send_buffer(:,1) with back boundary data
!Move pressure into send_buffer(:,1)
!$acc parallel loop collapse(3) async(1)
do k= 0, k_nodes-1
  do j= 0, j_nodes-1
    do i = 1, 2
      var = 1 ! pressure only
      indx = k*j_nodes*2+j*2+i
      send_buffer(indx,1) = soln(i+2,j+1,k+1,var)
```

```
!Update starting index in send_buffer(:,1)
indx_p = k_nodes*j_nodes*2

!Move velocities & temperature into send_buffer(:,1)
!$acc parallel loop collapse(3) async(2)
do var = 2, 5
  do k= 0, k_nodes-1
    do j= 1, j_nodes
      indx = (var-2)*k_nodes*j_nodes+k*j_nodes+j
      send_buffer(indx_p+indx) = soln(3,j,k+1,var)

!Pack send_buffer(:,2) with front boundary data
!$acc parallel loop collapse(3) async(3) & async(4)

!Update send_buffer on hosts
...

!Send/Recv between hosts
...

!Update recv_buffer on devices
...

!$acc data present(recv_buffer(:,:), soln(:,:,:,:))

!Unpack recv_buffer to soln ghost locations async(1:4)
```

Optimized V3 In the *Optimized V1* and *Optimized V2*, contiguous-memory arrays are created only for the i index direction. However, if a decomposition exists in the j or k index direction, then we may also need an array in the j and k direction. It should be noted that although real cell data on k boundary faces do not need to be packed into buffers, using buffers on k faces may still be helpful considering that there are ghost cells on k boundary faces which breaks the contiguity. We found that on the HokieSpeed cluster, using such arrays improves the performance very little, but the performance can be improved significantly on the NewRiver and Cascades cluster having GPUs which are much faster. The procedure of creating arrays and parallelizing the pack/unpack process in the j and k direction is very similar to that in the i direction, so there is no need to show an pseudo code here. Readers can reference Listing. 2.1.

Based on all the optimizations mentioned above, the final structure of this multi-GPU BDC code using MPI and OpenACC can be seen in Listing. 2.3. It can been seen that buffering the boundary data, updating the solution, pressure rescaling, boundary condition enforcement, dissipation calculation and residual calculation are all parallelized either using !$acc parallel or !$acc kernel directives.

Listing 2.3: A pseudo code for the BDC code main portion (*Optimized V3*)

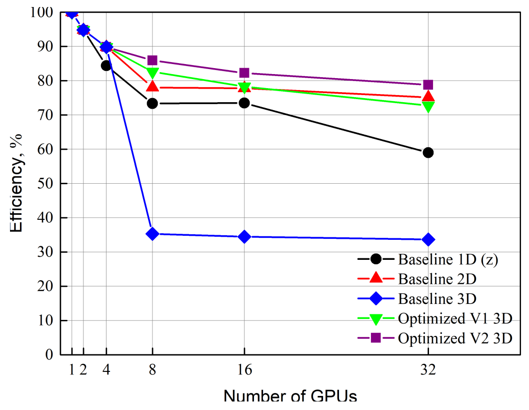
```
!!!!!!!!!!!!!!Main routine!!!!!!!!!!!!!!!
#ifdef _OPENACC
  !get the number of devices and set the GPU on each node
  ngpus=acc_get_num_devices(acc_device_nvidia)
  device_num=mod(id, ngpus)
  call acc_set_device_num(device_num, acc_device_nvidia)
#endif
```

```
   !$acc data copy(soln(:,:,:,:), residual(:,:,:,:), &
   !$acc           soln_send_x, soln_recv_x,          &
   !$acc           soln_send_y, soln_recv_y,          &
   !$acc           soln_send_z, soln_recv_z)

   !BEGIN MAIN ITERATION LOOP
   do while (iter <= max_iter)
     !zero the residual norms

#ifdef _MPI
     !pack the data to buffers
     call buffer_sendx(3,x_nodes-3)
     call buffer_sendy(3,y_nodes-3)
     call buffer_sendz(3,z_nodes-3)
     !transfer the data between neighbouring blocks
     call transfer_bc_data()
     !unpack the data to soln
     call buffer_recvx(x_nodes-1,1)
     call buffer_recvy(y_nodes-1,1)
     call buffer_recvz(z_nodes-1,1)
#endif

     !fill the residual array
     call calc_residual()

     !$acc parallel loop collapse(3) present(soln(:,:,:,:), &
     !$acc           residual(:,:,:,:))
     do k = indx_cal(5), indx_cal(6)
       do j = indx_cal(3), indx_cal(4)
         do i = indx_cal(1), indx_cal(2)
           ! update soln(i,j,k,1:5)

     !Pressure rescaling at the center point of the cavity
     call set_center_pressure()
     !$acc kernels present(soln(:,:,:,:))
     soln(:,:,:,1) = soln(:,:,:,1) - Pweightfactor

     !Reduce and print out residual norms
     call MPI_REDUCE
     call print_residual()

     iter=iter+1

   end do

 !!!!!!!!!!!!!Residual!!!!!!!!!!!!!!

 !!Boundary condition enforcement
 !Update the data in edges, corners and boundary faces first
 !Then update the dissipation terms in these boundary cells
 !use !$acc kernel or !$acc parallel directives

 !!Internal domain
```

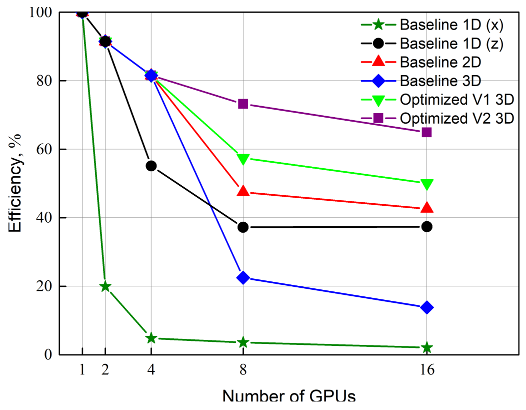
```
!Calculation the dissipation terms
#ifdef _OPENACC
   !$acc parallel loop collapse(3) async(queue_id)
#endif
   do k=3, z_nodes-2
     do j=3, y_nodes-2
        do i=3, x_nodes-2
          !calculate dissipation terms

!!Residual calculation for all needed cells
!use !$acc kernel or !$acc parallel directives
```

We will first show the benefits of applying *Optimized V1*, i.e., creating the contiguous-memory and parallelizing the process of pack/unpack. Fig. 2.7a shows the weak scaling efficiency of the different GPU code versions introduced earlier on HokieSpeed and NewRiver. The *Baseline* version using 3D decomposition performs poorly, which is very bandwidth limited and communication bounded due to noncontiguous data transfer when 3D decomposition is used. Although the baseline GPU version using 3D decomposition performs poorly, its two optimizations scales as well as the baseline using 2D decomposition or even better. This indicates that memory throughput is improved greatly and communication cost is reduced after optimization, although there is some pack/unpack overhead. Therefore, special attention should be paid to non-contiguous data movement. Also, Fig. 2.7 shows that *Optimized V2* performs better than *Optimized V1* because it transfers less data and overlaps asynchronous data transfers better.

To figure out how *Optimized V1* improves the performance significantly when partitions exist in the x dimension, profiling result comparisons of *Baseline* and *Optimized V1* are obtained



(a) Multi-GPU weak scaling performance on HokieSpeed

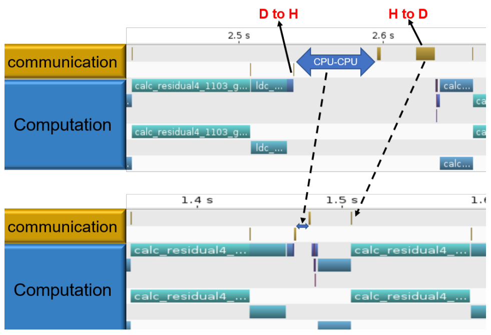


(b) Multi-GPU weak scaling performance on NewRiver

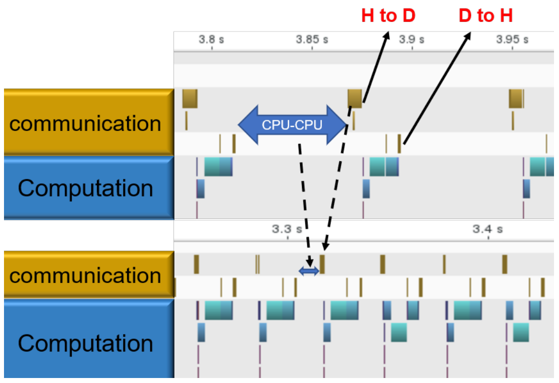
Figure 2.7: Multi-GPU weak scaling of different versions

on both platforms, which can be seen in Fig. 2.8. On both platforms, the communication time is largely reduced after applying the optimizations in *Optimized V1*. In fact, the reduction in the communication time can be divided into two aspects, the CPU-CPU communication time and the CPU-GPU (including host to device and device to host) communication time. On Hokiespeed, the CPU-CPU communication time is reduced by more than 6 times and the host to device time is reduced by more than 10 times. On NewRiver, the CPU-CPU communication time is reduced by more than 5 times and the host to device time is reduced by more than 3 times. The overhead of pack/unpack is small compared to other kernel computations, as the fraction of pack/unpack is only 1.6% of the whole kernel computations when using 8 GPUs on NewRiver (seen in Fig. 2.9).

It should be mentioned that the performance comparison of *Optimized V2* and *Optimized V3* on NewRiver and Cascades will be shown in Sec 2.4.6, as there the performance of applying different MPI compilers and GPUDirect will include such a comparison. The GPU strong scaling and weak scaling performance (*Optimized V3*) using 1D, 2D and 3D decompositions are shown in Fig. 2.10. From Fig. 2.10, 3D decomposition performs the best for the strong scaling, then 2D decomposition follows. 1D decomposition in the x or z dimension makes the



(a) 2 GPUs on HokieSpeed



(b) 8 GPUs on NewRiver

Figure 2.8: Profiling results

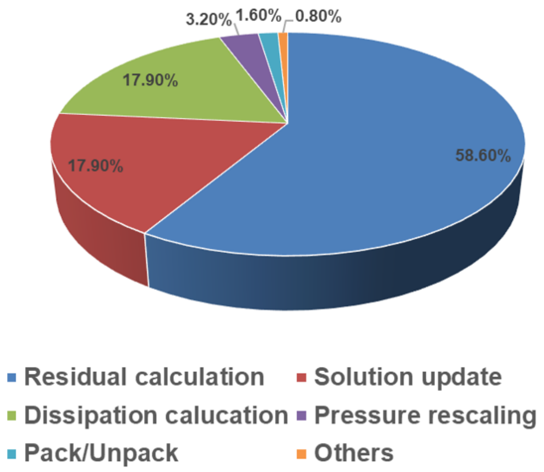
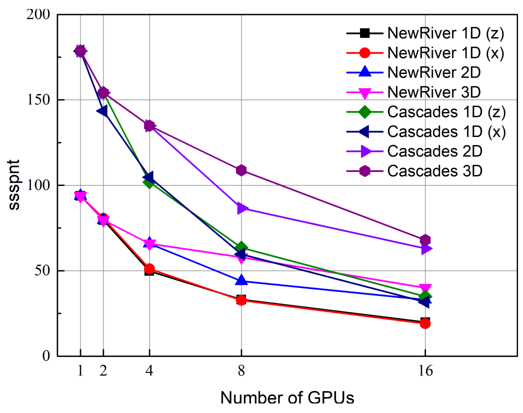


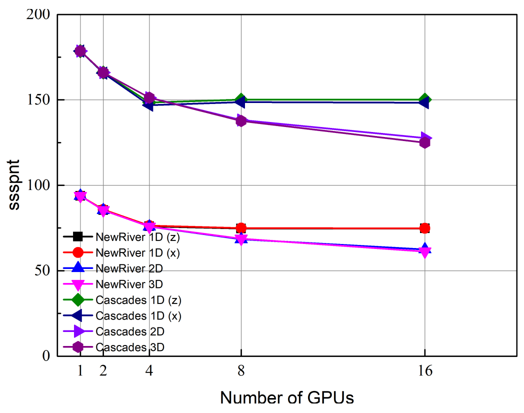
Figure 2.9: The fractions of different compute kernels

performance drop quickly, especially on Cascades, indicating that the surface to volume ratio is a more important consideration for the strong scaling. Similar to the CPU weak scaling, the weak scaling performance decays more slowly than strong scaling performance, meaning more computational work on the GPU can maintain a higher efficiency. This weak scaling applies the grid growth type 2, which is in Table 2.2. Since the problem size increases in accordance with the way *np* grows, 1D decomposition performs the best as every decomposed block has the least number of neighbours compared to 2D and 3D decomposition.

Fig. 2.11 shows the performance comparison across platforms. Different from Fig. 2.6, mul-



(a) GPU strong scaling



(b) GPU weak scaling

Figure 2.10: Multi-GPU scaling using different decompositions (*Optimized V3*)

tiple factors can affect the multi-GPU performance significantly, including the number of processors, platforms, whether a strong or weak scaling. When the number of GPUs increases, the efficiency drops significantly for both the strong and weak scaling, but the weak scaling efficiency holds a relatively higher value compared to the strong scaling. Cascades shows an about 2 times faster speedup compared to NewRiver, which is close to their theoretical double precision performance ratio, 1.66. It can be concluded that newer generation GPUs are more difficult to keep high efficiency compared to old generation GPUs, although the speed of of using newer GPUs is much higher. The reason is that the fraction of the communication time is larger on newer generation GPUs as their compute capability is higher. This also indicates that more optimizations should be made.

To investigate the effect of different grid growth methods on the weak scaling performance, some cases are tested on NewRiver and Fig. 2.12 shows such results. Since the two grid growths are the same if applying 3D decomposition, there is only one curve for 3D decomposition. It can be found that the performance is better for growth type 2, as the grid grows in all dimensions instead of only in one dimension. It should be emphasized that which decomposition should be used for the weak scaling depends on how the grid grows.

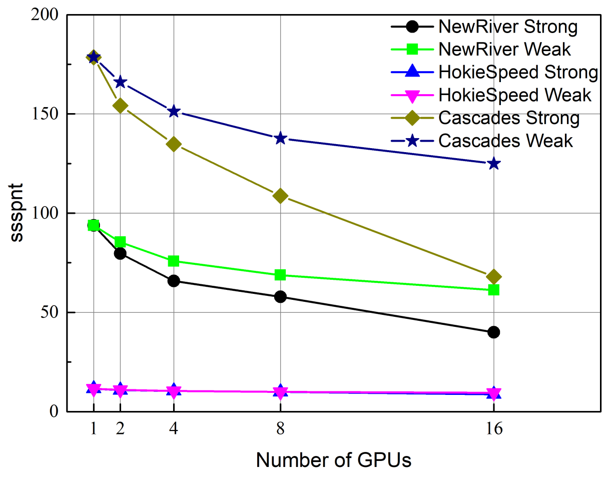


Figure 2.11: Multi-GPU performance comparison across platforms (*Optimized V3*)

2.4.6 CUDA-aware MPI and GPUDirect

The optimizations introduced in Section 2.4.5 have improved the efficiency significantly on different platforms. However, we are still interested in improving the scaling performance further on NewRiver and Cascades since they have some modern GPUs with different architectures. Thus, it became important to determine ways of reducing this communication cost by using CUDA-aware MPI and GPUDirect [25]. Later, performance comparisons will be made between using Open MPI, MVAPICH2 or MVAPICH2-GDR with GPUDirect.

HokieSpeed does not support CUDA-aware MPI. Thus, all inter-node GPU communications on HokieSpeed had to go through host memory. This staging deteriorates the performance greatly. Using CUDA-aware MPI, we only need to send GPU buffers instead of CPU buffers. CUDA-aware MPI has two performance benefits [25]. First, operations which require message transfer can be pipelined, which improves the memory throughput. Second, acceleration techniques such as GPUDirect can be utilized by the MPI library transparently to the user.

GPUDirect is an umbrella word for several GPU communication acceleration technologies. It provides high bandwidth and low latency communication between NVIDIA GPUs. There are

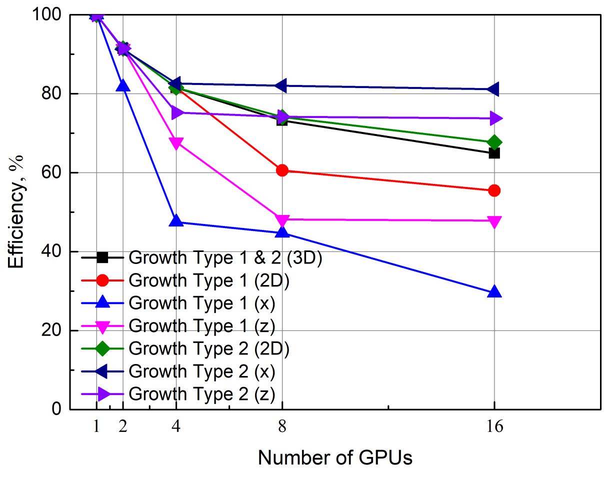


Figure 2.12: Weak scaling performance applying different grid growth methods (*Optimized V3*)

three levels of GPUDirect [26]. The first level is GPUDirect Shared Access, introduced with CUDA 3.1. This feature avoids an unnecessary memory copy within host memory between the intermediate pinned buffers of the CUDA driver and the network fabric buffer. The second level is GPUDirect Peer-to-Peer transfer (P2P transfer) and Peer-to-Peer memory access (P2P memory access), introduced with CUDA 4.0. This P2P memory access allows buffers to be copied directly between two GPUs on the same node. The last is GPU RDMA (Remote Direct Memory Access), with which buffers can be sent from the GPU memory to a network adapter without staging through host memory. The last feature is not supported on NewRiver and Cascades as it pertains to specific versions of the drivers (both from NVIDIA for the GPU and Mellanox for the Infiniband) which are not installed (other dependencies exist on NewRiver and Cascades, particularly parallel filesystems). Although GPU RDMA is not available, the other aspects of GPUDirect can be tested to determine its effect on the scaling performance using MVAPICH2-GDR.

Intra-Node Scaling Performance Results

In this subsection, we will first show the benefits of applying GPUDirect in a node. Table 2.3 shows the strong scaling performance comparison of different GPU code versions using 2 GPUs (intra-node performance) on NewRiver. The problem size is 256^3. The versions defined here are similar to the versions introduced in Section. 2.4.5, with the *Baseline* to be the non-optimized GPU version, *Optimized V3* uses the pack/unpack in all the available dimensions and the stencil-based communication method, and GPUDirect uses the P2P transfer technology applied to both of these versions of the code. Within a node, we are using GPUDirect P2P transfer between the memory of two GPUs on the same system/PCIe bus.

Table 2.3: Strong scaling comparison of different GPU code versions using 2 GPUs (on NewRiver)

GPU code versions	Decompositions	ssspnt	efficiency
Single GPU	(1,1,1)	93.8	100%
Baseline	(1,1,2)	145.9	77.8%
Baseline	(2,1,1)	22.0	11.7%
Optimized V3	(1,1,2)	167.2	89.2%
Optimized V3	(2,1,1)	169.8	90.5%
Baseline + GPUDirect	(1,1,2)	155.8	83.1%
Baseline + GPUDirect	(2,1,1)	154.7	82.5%
Optimized V3 + GPUDirect	(1,1,2)	177.9	94.9%
Optimized V3 + GPUDirect	(2,1,1)	179.4	95.7%

Using MVAPICH2, the baseline code decomposed in the i direction performs poorly, about 1/7 of that decomposed in the k direction. After a series of optimizations the efficiency changes from 11.7% to 90.5%, indicating again the importance of the coalesced memory access when doing host-device transfers. GPU direct P2P transfer on the baseline code is also able to avoid the cost of host-device transfers and is able to maintain an efficiency of 83% even though the data is not contiguous. Combining the performance optimizations with the use of GPUDirect can improve the efficiency to approximately 95% on 2 GPUs.

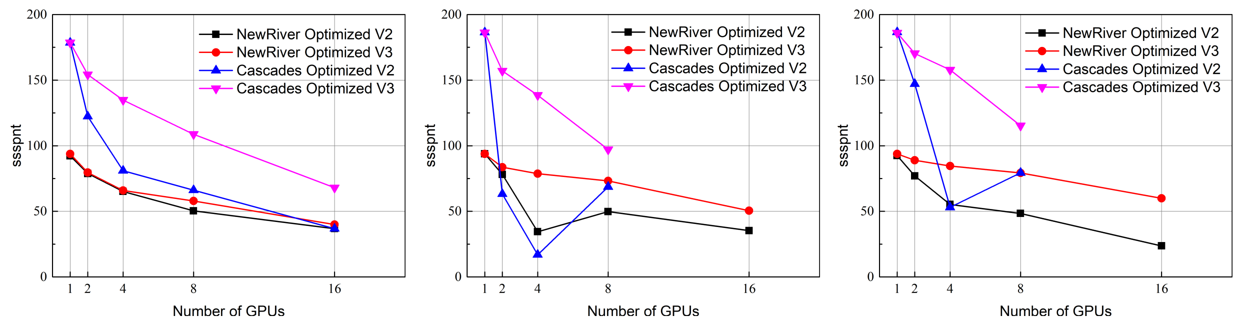
Table. 2.4 shows the weak scaling performance comparison of different GPU code versions using 2 GPUs in the intra-node mode. The result also shows either the optimizations proposed in this paper or GPUDirect (or both if applicable) should be used, if non-contiguous data transfers happen. It is also reasonable to see that the weak scaling generally performs better than the strong scaling, as more work are assigned to the GPU.

Table 2.4: Weak scaling comparison of different GPU code versions using 2 GPUs (on NewRiver)

GPU code versions	Decompositions	ssspnt	efficiency
Single GPU	(1,1,1)	93.8	100%
Baseline	(1,1,2)	161.6	86.1%
Baseline	(2,1,1)	21.4	11.4%
Optimized V3	(1,1,2)	175.3	93.5%
Optimized V3	(2,1,1)	176.3	94.0%
Baseline + GPUDirect	(1,1,2)	167.9	89.5%
Baseline + GPUDirect	(2,1,1)	154.9	82.6%
Optimized V3 + GPUDirect	(1,1,2)	180.0	96.0%
Optimized V3 + GPUDirect	(2,1,1)	180.9	96.5%

Inter-Node Scaling Performance Results

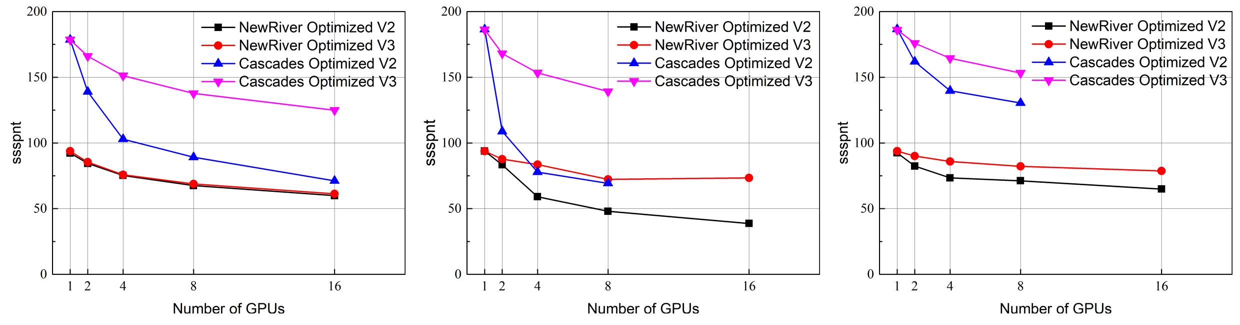
Strong Scaling Performance Results Since there are three different MPI options (Open MPI, MVAPICH2 and MVAPICH2-GDR with GPUDirect turned on) on NewRiver and Cascades, scaling performance results using the three different compilers/options are given. Fig. 2.13 shows the strong scaling performance using different MPI options, respectively. Considering 3D growth is much more common in CFD such as applying systematic mesh refinement so the 3D decomposition is of more interest. As mentioned earlier, when applying MVAPICH2 or MVAPICH2-GDR on Cascades, the performance drops to 1% if using 16 GPUs, so the maximum number of GPUs used in that occasion is 8. Since the results are for the strong scaling, we cannot expect a very high efficiency if scaling up to a large number of GPUs. Using MVAPICH2-GDR generally achieves the best performance especially when combined with *Optimized V3*. The performance drop of *Optimized V2* using 4GPUs on Cascades is caused by not using buffers in the y and z dimension. The performance curves using Open MPI on both platforms are much smoother than using MVAPICH2 and MVAPICH2-GDR. The difference is caused by that MVAPICH2 and Open MPI are different MPI compilers which have different focus (Open MPI targets more common uses and MVAPICH2 can meet special needs).



(a) Open MPI (b) MVAPICH2 (c) MVAPICH2-GDR

Figure 2.13: Strong scaling performance across platforms (3D decomposition)

Weak Scaling Performance Results When measuring the weak scaling performance, grid growth type 2 is applied here. Fig. 2.14 shows the weak scaling performance using different MPI options. For each MPI option, results of using different decompositions for different grid growth are given. MVAPICH2 and Open MPI perform equivalently but there are still some differences. MVAPICH2 performs better than Open MPI for *Optimized V3*, and generally worse for *Optimized V2*, compared to Open MPI. It is reasonable as MVAPICH2 is designed to reduce communication overhead for complicated communication patterns. It can also be seen that GPUDirect (with MVAPICH2-GDR) brings some performance benefits and performs the best for both *Optimized V2* and *Optimized V3*. Using buffers in all available decomposed dimensions (*Optimized V3*) generates the best performance.



(a) Open MPI (b) MVAPICH2 (c) MVAPICH2-GDR

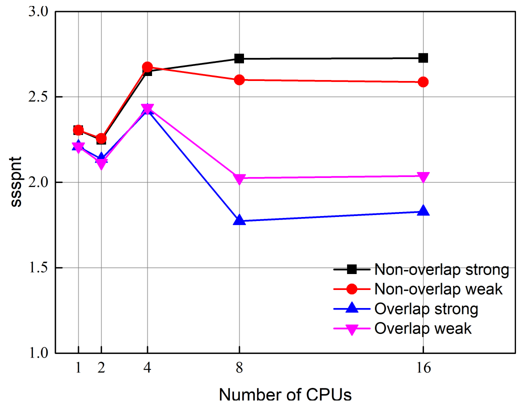
Figure 2.14: Weak scaling performance across platforms (3D decomposition)

2.4.7 Overlapping Communication and Computation

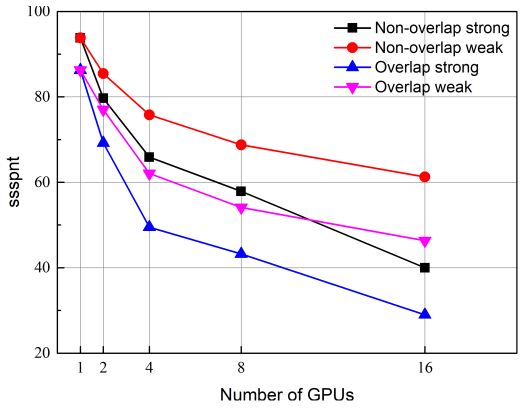
When overlapping communication and computation, every decomposed block is further separated into two components: internal and external domains. For large enough problems, the internal domain will have significantly more grid points than the external domain. These internal points do not need data from other blocks so they can compute their updates while the communication is occurring for the external portion of the block. After communication is finished, the external domain continues to finish the remaining computation. Overlapping will not reduce latency but it can hide the latency caused by inter-block communication. In this paper, overlapping communication and computation was applied to both CPUs and GPUs. Communication is always done on CPUs while computation can be performed on CPUs or GPUs.

Case studies to compare the overlap and non-overlap versions have been made on different platforms, using different decompositions and code versions, and for the strong and weak scaling performance. The overlap version performs more slowly compared to the non-overlap version. Fig. 2.15 shows the strong and weak scaling performance for both the CPU and

GPU on the NewRiver cluster, with a performance comparison between the overlap version (extended from *Optimized V3*) and the non-overlap (*Optimized V3*) version. For both the CPU and GPU, overlap performs about 20% to 30% slower than non-overlap up to 16 processors, which was out of our initial expectation. The main reason is that the asynchronous progression is not supported well, potentially caused by the MPI and the communication system used. To figure out whether the asynchronous progression engine was activated or not, we used NVIDIA Visual Profiler [27] to trace the program kernel executions on GPUs and found that the MPI used does not trigger communication until the code runs to a MPI_Waitall call, although communication is launched as early as possible. Since there is no actual overlap, and the non-overlap version only needs to setup the residual calculation kernel once while the overlap version has to do the setup multiple times (as it contains the internal domain and external domains), this overhead makes the overlap slower than the non-overlap version.



(a) CPU scaling



(b) GPU scaling

Figure 2.15: Overlap of communication and computation on NewRiver (3D decomposition)

In fact, the MPI standard [28] does not guarantee there is an actual overlap, which also means that the it may or may not be possible for communication to make progress when control has returned to the application, depending on the communication software and the underlying hardware. In Ref [29], it is also concluded that the degree of actual overlap for an application depends on the overlap potential of both the application and the underlying communication subsystem. In our case, we tested the overlap on different Virginia Tech supercomputing platforms using different MPI options and different decompositions, and none of them improved the performance using multiple GPUs. It should be mentioned that the MPI standard allows for non-blocking operations to only be progressed to completion if a proper test/wait call was made. Thus, we tried to add many MPI_Testall (dips into the MPI progression engine many times) or similar calls for the GPU code right after communication initialization. This makes overlap slightly better (observed through tracing). However, some overhead is produced due to adding these wait calls, also the degree of overlap is still not fully complete. The benefits of the performance enhancements are negligible compared with the overhead for our BDC code.

Our conclusion is that overlapping communication and computation is not a universal performance improvement for all applications and platforms, including the BDC problem using only MPI+OpenACC. Only if both the MPI compiler and the architecture supports asynchronous progression can overlap perform well and be used to hide some latency, which is difficult. An alternative way of improving the overlap is using MPI+OpenACC+OpenMP, in which OpenMP is used to generate multiple threads. These threads can work on different tasks such as computation and communication so that the actual degree of overlap can be increased [30, 31, 32, 33, 34]. In fact, there are more literature discussing how to improve the overlap performance and almost all of them use multiple threads. Therefore, developers who have an interest in the overlap version for their own codes may need to do some simple tests first and should not only depend on overlap to get high performance. Since multi-threading is not a focus in this paper, no in-depth investigation of multi-threading is applied in this paper.

2.5 Conclusions

It is shown in the paper that OpenACC directives offer a convenient way to accelerate a CFD code fast on multiple platforms. All the platforms can generally use the same code with little code intrusion, which is a big advantage over CUDA and OpenCL. Some general optimizations are examined to improve the multi-GPU code performance, such as the pack/unpack method and stencil-based communication method. The optimizations introduced are shown to be very effective for both strong scaling and weak scaling, greatly reducing communication overhead on GPUs. Further optimizations such as the overlap of communication and computation, asynchronous progression, and the use of CUDA-aware MPI and GPUDirect are also implemented and discussed. Overlapping communication and computation using only MPI+OpenACC is shown to be not an efficient way to improve the multi-GPU performance. GPUDirect is shown to be effective in a CFD application like the BDC code in this paper, as GPUDirect enables GPUs to communicate with each other directly and also increases the bandwidth between host and device. This avoids overhead between host and device and is important for communication-bound problems. Also, a combination of the use of GPUDirect and the optimizations proposed in this paper can improve both the strong and weak scaling performance substantially. 3D domain decomposition generally performs the best for the strong scaling on different platforms. For weak scaling, which decomposition performs best depends on how the grid growth is.

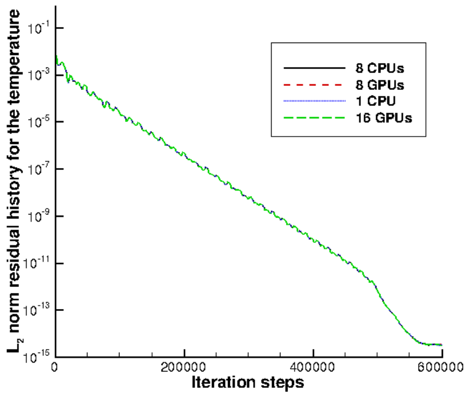
Acknowledgements

The authors would like to thank Andrew J. McCall and Behzad Baghapour for creating the original BDC code as well as giving advice, and thank Charles W. Jackson for reviewing the

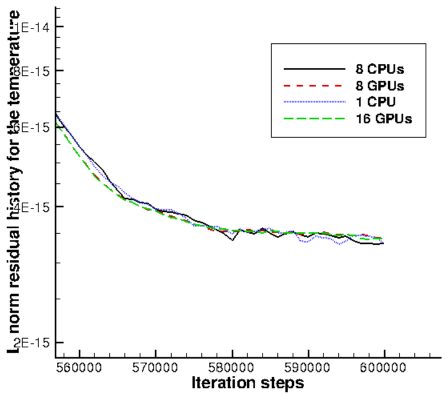
paper and participating in various helpful discussions.

2.6 Appendix

Fig. 2.16 shows the L_2 norm residual history of the temperature for different versions of the code.



(a) L2 norm residual history the temperature (whole)



(b) GPU scaling

Figure 2.16: L2 norm residual history for the temperature

Bibliography

[1] Shamoon Jamshed. *Using HPC for Computational Fluid Dynamics: A Guide to High Performance Computing for CFD Engineers.* Academic Press, 2015.

[2] Blaise Barney. OpenMP, 2018.

[3] Blaise Barney. Message Passing Interface (MPI), 2019.

[4] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI and OpenMP Parallel Programming, 2013.

[5] *GPUs for Scientific Computing*, 2009.

[6] Jiri Kraus, Michael Schlottke, Andrew Adinetz, and Dirk Pleiter. Accelerating a c++ cfd code with openacc. In *2014 first workshop on accelerator programming using directives*, pages 47–54. IEEE, 2014.

[7] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6. ACM, 2017.

[8] OpenACC-Standard.org. *The OpenACC Application Programming Interface.* OpenACC-Standard.org, 2018.

[9] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[10] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S Vetter. Evaluating performance portability of openacc. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 51–66. Springer, 2014.

[11] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143. IEEE, 2013.

[12] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on gpus with openacc and cuda fortran implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016.

[13] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. Mpi+ openacc: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems. *Computer Physics Communications*, 2018.

[14] Brent P Pickering, Charles W Jackson, Thomas RW Scogland, Wu-Chun Feng, and Christopher J Roy. Directive-based GPU programming for computational fluid dynamics. *Computers & Fluids*, 114:242–253, 2015.

[15] Behzad Baghapour, Andrew J McCall, and Christopher J Roy. Multilevel parallelism for cfd codes on heterogeneous platforms. In *46th AIAA Fluid Dynamics Conference*, page 3329, 2016.

[16] Weicheng Xue, Charles W Jackson, and Christopher J Roy. Multi-cpu/gpu parallelization, optimization and machine learning based autotuning of structured grid cfd codes. In *2018 AIAA Aerospace Sciences Meeting*, page 0362, 2018.

[17] Alexandre Joel Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of computational physics*, 135(2):118–125, 1997.

[18] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[19] Weicheng Xue, Hongyu Wang, and Christopher J Roy. Code verification for 3d turbulence modeling in parallel sensei accelerated with mpi. In *AIAA Scitech 2020 Forum*, page 0347, 2020.

[20] Anders Ytterström. A tool for partitioning structured multiblock meshes for parallel computational mechanics. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(4):336–343, 1997.

[21] Jarmo Rantakokko. Partitioning strategies for structured multiblock grids. *Parallel Computing*, 26(12):1661–1680, 2000.

[22] Hokiespeed, 2017.

[23] Newriver, 2019.

[24] Cascades, 2020.

[25] Jiri Kraus. An Introduction to CUDA-Aware MPI, 2013.

[26] NVIDIA. NVIDIA GPUDirect, 2019.

[27] NVIDIA Corporation. *Profiler User's Guide*, 2019.

[28] MPI: A message-passing interface standard, 2015.

[29] David E Bernholdt, Jarek Nieplocha, P Sadayappan, Aniruddha G Shet, and Vinod Tipparaju. Characterizing Computation-Communication Overlap in Message-Passing Systems. Technical report, The Ohio State University, 2008.

[30] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping communication and computation in mpi by multithreading. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[31] Karthikeyan Vaidyanathan, Dhiraj D Kalamkar, Kiran Pamnany, Jeff R Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. Improving concurrency and

asynchrony in multithreaded mpi applications using software offloading. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[32] Huiwei Lu, Sangmin Seo, and Pavan Balaji. Mpi+ ult: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454. IEEE, 2015.

[33] Alexandre Denis and François Trahay. Mpi overlap: Benchmark and analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 258–267. IEEE, 2016.

[34] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moreto, Martin Schulz, Ramon Beivide, Mateo Valero, and Abhinav Bhatele. Optimizing computation-communication overlap in asynchronous task-based programs. In *Proceedings of the ACM International Conference on Supercomputing*, pages 380–391, 2019.

Chapter 3

An Improved Framework of GPU Computing for CFD Applications on Structured Grids using OpenACC

Weicheng Xue[1], Charles W. Jackson[2] and Christopher J. Roy[3]

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. All the performance optimizations were developed and implemented by the first author. All the results were collected by the first author.

- Charles W. Jackson (second author): The second author provided a lot of useful suggestions when porting the CFD code called SENSEI to multiple CPUs. Also, the second author provided valuable comments for this manuscript.

[1]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.
[2]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.
[3]Professor, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 330, Virginia Tech, 460 Old Turner St, AIAA Associate Fellow.

- Christopher J. Roy (final author): The final author provided valuable feedback for this study and comments for this manuscript.

Abstract

This paper is focused on improving multi-GPU performance of a research CFD code on structured grids. MPI and OpenACC directives are used to scale the code up to 16 GPUs. This paper shows that using 16 P100 GPUs and 16 V100 GPUs can be 30× and 70× faster than 16 Xeon CPU E5-2680v4 cores for three different test cases, respectively. A series of performance issues related to the scaling for the multi-block CFD code are addressed by applying various optimizations. Performance optimizations such as the pack/unpack message method, removing temporary arrays as arguments to procedure calls, allocating global memory for limiters and connected boundary data, reordering non-blocking MPI I_send/I_recv and Wait calls, reducing unnecessary implicit derived type member data movement between the host and the device and the use of GPUDirect can improve the compute utilization, memory throughput, and asynchronous progression in the multi-block CFD code using modern programming features.

Keywords: MPI, OpenACC, Multi-GPU, CFD, Performance Optimization, Structured Grid

3.1 Introduction

Computational Fluid Dynamics (CFD) is a method to solve problems related to fluids numerically. There are numerous studies applying a variety of CFD solvers to solve different fluid problems. Usually these problems require the CFD results to be generated quickly as well as precisely. However, due to some restrictions of the CPU compute capability, system memory size or bandwidth, highly refined meshes or computationally expensive numerical methods may not be feasible. For example, it may take thousands of CPU hours to converge a 3D Navier-Stokes flow case with more than millions of degrees of freedoms. In such a circumstance, high performance parallel computing [1] enables us to solve the problem much faster. Also, parallel computing can provide more memory space (either shared or distributed) so that large problems can be solved.

Parallel computing differs from serial computing in many aspects. On the hardware side, a parallel system commonly has multi/many-core processors or even accelerators such as GPUs, which enable programs to run in parallel. Memory in a parallel system is either shared or distributed [1], with unified memory address [2] and non-unified memory address usually being used, respectively. On the software side, there are various programming models for parallel computing including OpenMP [3], MPI [4], CUDA [5], OpenCL [6] and OpenACC [7]. Different parallel applications can utilize different parallel paradigms based on a pure parallel model or even a hybrid model such as MPI+OpenMP, MPI+CUDA, MPI+OpenACC, OpenMP+OpenACC, etc.

For multi/many-core computing, OpenMP, MPI and hybrid MPI+OpenMP have been widely used and their performance has also been frequently analyzed in various areas, including CFD. Gourdain et al. [8, 9] investigated the effect of load balancing, mesh partitioning and communication overhead in their MPI implementation of a CFD code, on both structured and unstructured meshes. They achieved good speedups for various cases up to thousands of cores. Amritkar et al. [10] pointed out that OpenMP can improve data locality on a shared memory platform compared to MPI in a fluid-material application. However, Krpic et al. [11] showed that OpenMP performs worse when running large scale matrix multiplication even on shared-memory computer system when compared to MPI. Similarly, Mininni et al. [12] compared the performance of the pure MPI implementation and the hybrid MPI+OpenMP implementation of an incompressible Navier-Stokes solver, and found that the hybrid approach does not outperform the pure MPI implementation when scaling up to about 20,000 cores, which in their opinion may be caused by cache contention and memory bandwidth. In summary, it can be concluded that MPI is more suitable for massively parallel applications as it can help achieve better performance compared to OpenMP.

In addition to accelerating a code on the CPU, accelerators such as GPU [13] are becoming popular in the area of scientific computing. CUDA [5], OpenCL [6], and OpenACC [7] are the three commonly used programming models for the GPU. CUDA and OpenCL are mainly C/C++ extensions (CUDA has also been extended to Fortran) while OpenACC is a compiler directive based interface, therefore CUDA and OpenCL are more troublesome in terms of programming, requiring a lot of user intervention. CUDA is proprietary to NVIDIA and thus can only run on NVIDIA GPUs. OpenCL supports various architectures but it is a very low level API, which is not easy for domain scientists to adapt to. Also, although OpenCL has a good portability across platforms, a code may not run efficiently on various platforms without specific performance optimizations and tuning. OpenACC has some advantages over CUDA and OpenCL. Users only need to add directives in their codes to expose enough parallelisms to the compiler which determines how to accelerate the code. In such a way, a lot of low level implementation can be avoided, which provides a relatively easy way for domain scientists to accelerate their codes on the GPU. Additionally, OpenACC can perform fairly well across different platforms even without significant performance tuning. However, OpenACC may not reveal some parallelisms if there is a lack of performance optimizations. Therefore, OpenACC is usually assumed to be slower than CUDA and OpenCL, but it is still fairly fast. Even for some occasions, OpenACC can be the fastest [14], which is surprising. To program on multiple GPUs, MPI may be needed, i.e., the MPI+OpenACC hybrid model may be required. CPUs are set as hosts and GPUs are set as accelerator devices, which is referred to as the offload model, in which the most computational expensive portion of the code is offloaded to the GPU, while the CPU handles instructions of controls and file I/O.

A lot of work has been done to leverage GPUs for CFD applications. Jacobsen et al. [15] investigated multi-level parallelisms for the classic incompressible lid driven cavity problem on GPU clusters using MPI+CUDA and hybrid MPI+OpenMP+CUDA implementations. They found that the MPI+CUDA implementation performs much better than the pure CPU implementation but the hybrid performs worse than the MPI+CUDA implementation. Elsen et al. [16] ported a complex CFD code to a single GPU using BrookGPU [17] and achieved a speedup of 40$\times$ for simple geometries and 20$\times$ for complex geometries. Brandvik et al. [18] applied CUDA to accelerate a 3D Euler problem using a single GPU and got a speedup of 16$\times$. Luo et al. [19] applied MPI+OpenACC to port a 2D incompressible Navier-Stokes solver to 32 NVIDIA C2050 GPUs and achieved a speedup of 4$\times$ over 32 CPUs. They mentioned that OpenACC can increase the re-usability of the code due to OpenACC's similarity to OpenMP. Xia et al. [20] applied OpenACC to accelerate an unstructured CFD solver based on a Discontinuous Galerkin method. Their work achieved a speedup of up to 24$\times$ on one GPU compared to one CPU core. They also pointed out that using OpenACC requires the minimum code intrusion and algorithm alteration to leverage the computational power of GPU. Chandar et al. [21] developed a hybrid multi-CPU/GPU framework on unstructured overset grids using CUDA. Xue et al. [22] applied multiple GPUs for a complicated CFD code on two different platforms but the speedup was not satisfactory (only up to 4× on a NVIDIA P100 GPU), even with some performance optimizations. Also, Xue et al. [23] investigated the multi-GPU performance and its performance optimization of a 3D buoyancy-driven cavity solver using MPI and OpenACC directives. They showed that decomposing the total problem in different dimensions affects the strong scaling performance significantly when using multiple GPUs. Xue et al. [24] further applied the heterogeneous computing to accelerate a complicated CFD code on a CPU/GPU platform using MPI and OpenACC. They achieved some performance improvements for some of their test cases, and pointed out the communication and synchronization overhead between the CPU and GPU may be the performance bottleneck. Both of the works in Ref [21, 24] showed that the hybrid CPU/GPU framework can outperform the pure GPU framework to some degree, but the performance gain depends on the platform and application.

3.2 Description of the CFD code: SENSEI

SENSEI (Structured, Euler/Navier-Stokes Explicit-Implicit Solver) is our in-house 2D/3D flow solver initially developed by Derlaga et al [25], and later extended to a turbulence modeling code base through an object-oriented programming manner by Jackson et al. [26] and Xue et al. [27]. SENSEI is written in modern Fortran and is a multi-block finite volume CFD code. An important reason of why SENSEI uses structured grid is that the quality of mesh is better using a multi-block structured grid than using an unstructured grid. In addition, memory can be used more efficiently to obtain better performance since the data are stored in a structured way in memory. The governing equations can be written in weak form as

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{Q}\mathrm{d}\Omega + \oint_{\partial\Omega}(\vec{F_{i,n}} - \vec{F_{\nu,n}})\mathrm{d}s = \int_{\Omega}\vec{S}\mathrm{d}\Omega \tag{3.1}$$

where $\vec{Q}$ is the vector of conversed variable, $\vec{F_{i,n}}$ and $\vec{F_{\nu,n}}$ are the inviscid and viscous flux normal components (the dot product of the 2nd order flux tensor and the unit face normal vector), respectively, given as,

$$\vec{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_t \end{bmatrix}, \vec{F_{i,n}} = \begin{bmatrix} \rho V_n \\ \rho u V_n + n_x p \\ \rho v V_n + n_y p \\ \rho w V_n + n_z p \\ \rho h_t V_n \end{bmatrix}, \vec{F_{\nu,n}} = \begin{bmatrix} 0 \\ n_x \tau_{xx} + n_y \tau_{xy} + n_z \tau_{xz} \\ n_x \tau_{yx} + n_y \tau_{yy} + n_z \tau_{yz} \\ n_x \tau_{zx} + n_y \tau_{zy} + n_z \tau_{zz} \\ n_x \Theta_x + n_y \Theta_y + n_z \Theta_z \end{bmatrix} \tag{3.2}$$

$\vec{S}$ is the source term from either body forces, chemistry source terms, or the method of manufactured solutions [28]. ρ is the density, u, v, w are the Cartesian velocity components, e_t is the total energy, h_t is the total enthalpy, $V_n = n_x u + n_y v + n_z w$ and the n_i terms are the components of the outward-facing unit normal vector. τ_{ij} are the viscous stress components based on Stokes's hypothesis. Θ_i represents the heat conduction and work from the viscous stresses. In this paper, both the Euler and laminar Navier-Stokes solvers of SENSEI are ported to the GPU, but not for the turbulence models as the turbulence implementation involves a lot of object-oriented programming features such as overloading, polymorphism, type-bound procedures, etc. These newer features of the language are not supported well by the PGI compiler used, as they may require the GPU to jump from an address to a different address in runtime, which should be avoided when programming on GPUs.

In SENSEI, ghost cells are used for multiple purposes. First, boundary conditions can be enforced in a very straightforward way. There are different kinds of boundaries in SENSEI, such as slip wall, non-slip wall, supersonic/subsonic for inflow/outflow, farfield, etc. Second, from the perspective of parallel computing, ghost cells for connected boundaries contain data from the neighboring block used during a syncing routine so that every block can be solved independently. SENSEI uses pointers of a derived type to store the neighboring block information easily. Unless otherwise noted, all of the results presented here will be using a second-order accurate scheme. Second order accuracy is achieved using the MUSCL scheme [29], which calculates the left and right state for the primitive variables on each face of all cells. Time marching can be accomplished using an explicit M-step Runge-Kutta scheme [30] and an implicit time stepping scheme [31, 32, 33]. In this paper, only the explicit M-step Runge-Kutta scheme is used as the implicit scheme uses a completely objected-oriented way of programming which includes overloading of type-bound procedures.

Even though derived types are used frequently in SENSEI, to promote coalesced memory access and improve cache reuse, struct-of-array (SOA) instead of array-of-struct (AOS) is chosen for SENSEI. This means that, for example, the densities in each cell are stored in contiguous memory locations instead of all of the degrees of freedom for a cell being stored together. Using SOA produces a coalesced memory access pattern which performs well on GPUs and is recommended by NVIDIA [5].

SENSEI has the ability to approximate the inviscid flux with a number of different inviscid flux functions. Roe's flux difference splitting [34], Steger-Warming flux vector splitting [35], and Van Leer's flux vector splitting [36] are available. The viscous flux is calculated using a Green's theorem approach and requires more cells to be added to the inviscid stencil. For more details on the theory and background see Derlaga et at. [25], Jackson et al. [26] and Xue et al. [27].

3.3 Overview of CPU/GPU Heterogeneous System, MPI and OpenACC

3.3.1 CPU/GPU Heterogeneous System

As can be seen in Fig. 3.1, the NVIDIA GPU has more lightweight cores than the CPU, so the compute capability of the GPU is much higher than the CPU. Also, the GPU has higher memory bandwidth and lower latency to its memory. The CPU and the GPU have discrete memories so there are data movements between them, which can be realized through the PCI-E or NVLink. The offload model is commonly used for the pure GPU computing, which can be seen in Fig. 3.2. In CFD, the CPU deals with the geometry input, domain decomposition and some general settings. Then, the CPU offloads the intensive computations to the GPU. The boundary data exchange can happen either on the CPU or the GPU, depending on whether the GPUDirect is used or not. After the GPU finishes the computation, it moves the solution to the CPU. The CPU finally outputs the solution to files. To obtain good performance, there should be enough GPU threads running concurrently. Using CUDA [5] or OpenACC [7], there are three levels of tasks: grid, thread block and thread. Thread blocks can be run asynchronously in multiple streaming multiprocessors (SMs) and the communication between thread blocks is expensive. Each thread block has a number of threads. There is only lightweighted synchronization overhead for all threads in a block. All threads in a thread block can run in parallel in the Single Instruction Multiple Threads (SIMT) mode [37]. A kernel is launched as a grid of thread blocks. Several thread blocks can share a same SMs but all the resources need to be shared. Each thread block contains multiple 32 thread warps. Threads in a warp can be executed concurrently on a multiprocessor. In comparison to the CPU, which is often optimized for instruction controls and for low latency access to cached data, the GPU is optimized for data parallel and high throughput computations.

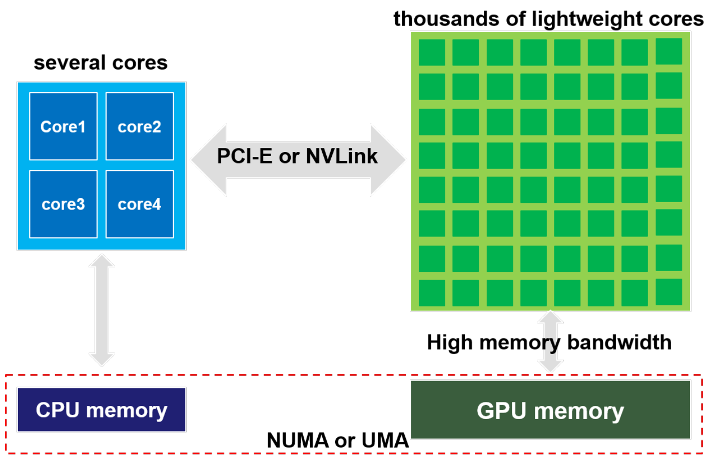


Figure 3.1: CPU and GPU

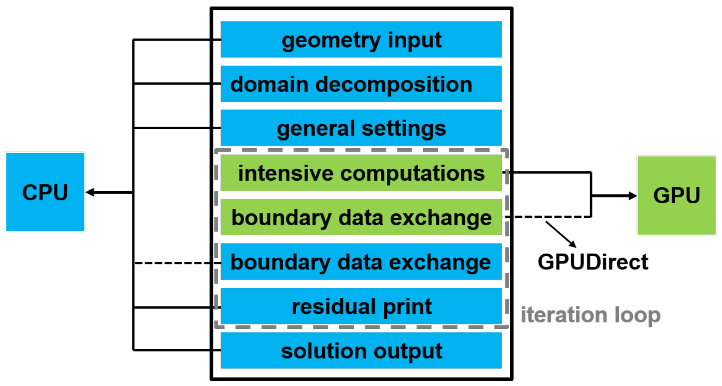


Figure 3.2: The offload model

3.3.2 MPI

MPI (Message Passing Interface) is a programming model for parallel computing [38] which enables data to be exchanged between processors via messages. It can be used on both distributed and shared systems. MPI supports point-to-point communication patterns as well as group communications. MPI also supports the customization of derived data type so transferring data between different processors is easier. It should be noted that a customized derived type may not guarantee fast data transfers. MPI supports the use of C/C++ and Fortran. There are many implementations of MPI including Open MPI [39] and MVAPICH2 [40].

3.3.3 OpenACC

OpenACC is a standard for parallel programming on heterogeneous CPU/GPU systems [7]. Very similar to OpenMP [3], OpenACC is also directive based, so it requires less code intrusion to the original code base compared with CUDA [5] or OpenCL [6]. OpenACC usually does not provide competitive performance compared to CUDA [41, 42, 43], however the performance it provides can still satisfy many needs. Compilers such as PGI [44] and GCC can support OpenACC in a way that the compiler detects the directives in a program and decides how to parallelize loops by default. The compiler also handles moving the data between discrete memory locations, but it is the users' duty to inform the compiler to do so. Users can provide more information through the OpenACC directives to attempt to optimize performance. These optimizations will be the focus of this paper.

3.4 Domain Decomposition

There are many strategies to decompose a domain, such as using Cartesian [45, 46] or graph topology [47]. Because SENSEI is a structured multi-block code, Cartesian block splitting will be used. With Cartesian block splitting, there is a tradeoff between decomposing the domain in more dimensions (e.g. 3D or 2D domain decomposition) and fewer dimensions (e.g. 1D domain decomposition). The surface area-volume ratio is larger if decomposing the domain in fewer dimensions, which means more data needs to be transferred between different processors. Also, decomposing the domain in 1D can generate slices that are too thin to support the entire stencil when decomposing the domain into many sub-blocks. However, the fewer number of dimensions being composed means that each block needs to communicate with a fewer number of neighbors, reducing the number of transfers and their corresponding latency.

By default, SENSEI uses a general 3D or 2D domain decomposition (depending on whether the problem is 3D or 2D) but can switch to 1D domain decomposition if specified. An example of the 3D decomposition is shown in Fig. 3.3. The whole domain is decomposed into a number of blocks. Each block connects to 6 neighboring blocks, one on each face. For each sub-iteration step (as the RK multi-step scheme is used), neighboring decomposed blocks need to exchange data with each other, in order to fill their own connected boundaries. Since data layout of multi-dimensional arrays in Fortran is column-majored, we always decompose the domain starting form the most non-contiguous memory dimension. For example, since the unit stride direction of a three-dimensional array $A(i, j, k)$ is the first index (i), i is the last decomposed dimension and k is the first decomposed dimension.

The 3D domain decomposition method shown in Fig. 3.3 is a processor clustered method. This method is designed for the scenarios in which the number of processors (np) is greater than the number of parent blocks (npb), i.e., the number of blocks before the domain decomposition. There are several advantages with this decomposition strategy. First, this method is an ”on the fly” approach, which is convenient to use and requires no manual operation or preprocessing of the domain decomposition. Second, it is very robust that it can handle most situations if np is greater than or equal to npb. Third, the communication overhead is small due to the simple connectivity, making the MPI communication implementation easy. The load can be balanced well if np is significantly larger than npb. Finally, some do-

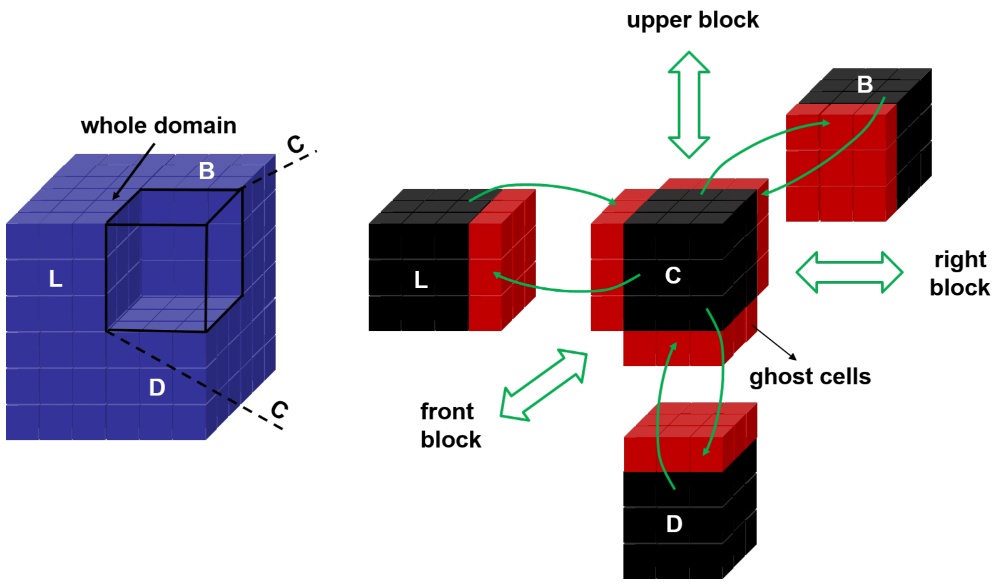


Figure 3.3: A 3D domain decomposition

main decomposition work can be done in parallel, although the degree may vary for various scenarios.

This domain decomposition method may have load imbalance issue if *np* is not obviously greater than *npb*, which can be addressed using a domain aggregation technique, similar to building blocks. A simple 2D example of how the domain aggregation works is given in Fig. 3.4. In this example, the first parent block has twice as many cells as the second parent block. If only two processors are used, the workload cannot be balanced well without over-decomposition and aggregation. With over-decomposition, the first parent block is decomposed into 4 blocks, and one of these decomposed blocks is assigned to the second processor so both processor have the same amount of work to do. It should be noted that the processor boundary length becomes longer due to the processor boundary deflection increasing the amount of communication required. Using the domain aggregation approach, any decomposed block is required to exchange data with its neighbours on the same processor but this does not require MPI communications. Only the new connected boundaries (e.g.

the red solid lines in Fig. 3.4) between neighboring processors need to be updated using MPI routines at each sub-iteration step.

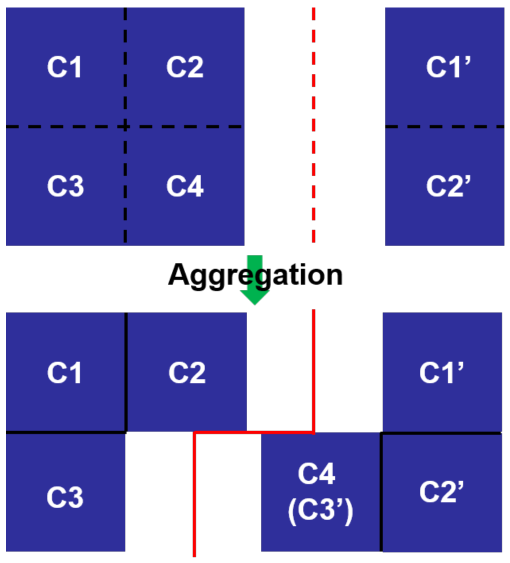


Figure 3.4: An example showing the domain aggregation

3.5 Boundary Decomposition in Parallel and Boundary Reordering

Boundaries also need to be decomposed and updated on individual processors. Initially, only the root processor has all the boundary information for all parent blocks, since root reads in the grid and boundaries. After domain decomposition, each parent block is decomposed into a number of child blocks. These child blocks need to update all the boundaries for themselves. For non-connected boundaries this update is very straightforward as each processor just needs to compare their individual block index range with the boundary index range. For interior boundaries caused by domain decomposition, a family of Cartesian MPI topology routines are used to setup communicators and make communication much less troublesome.

However, for connected parent block boundaries, the update (decomposing and re-linking these boundaries) is more difficult, as the update is completed in parallel on individual processors in SENSEI, instead of on the root processor. The parallel process can be beneficial if numerous connected boundaries exist. For every parent block connected boundary, the root processor first broadcasts the boundary to all processors within that parent block and its neighbour parent block, and then returns to deal with the next parent block connected boundary. The processors within that parent block or its neighbour parent block compare the boundary received to their block index ranges. If a processor does not contain any index range of the parent boundary, it moves forward to compare the next parent boundary. Processors in the parent block having this boundary or processors in the neighbour parent block matching part of the neighbour index range are colored but differently. These colored processors will need to update their index range for the connected boundary. To illustrate how we use MPI topology routines and inter-communicators to setup connectivity between neighbour blocks, a 2D example having 3 parent blocks and more than 3 CPUs is given in Fig. 3.5. Processors which match a parent connected boundary are included in an inter-communicator. The processor in a parent block first sends its index ranges to processors residing in the neighbour communicator. Then a processor in the neighbour communicator matching part of the index range is a neighbour while others in the neighbour communicator not matching the index range are not neighbour processors. Through looping over all the neighbour processors in the neighbour communicator, one processor sets up connectivity with all its connected neighbours. This process is performed in parallel as the root processor does not need to participate in this process except for broadcasting the parent boundary to all processors in the parent block and its neighbour parent block at the beginning. There may be special cases. The first special case is that the root is located at a parent block or its neighbour parent block. The root needs to participate in the boundary decomposition and re-linking process, as shown in Fig. 3.5. The second special case is given in the lower right square in Fig.3.5, in which a parent block partly connects to itself, which may make a decomposed block partially connect to itself.

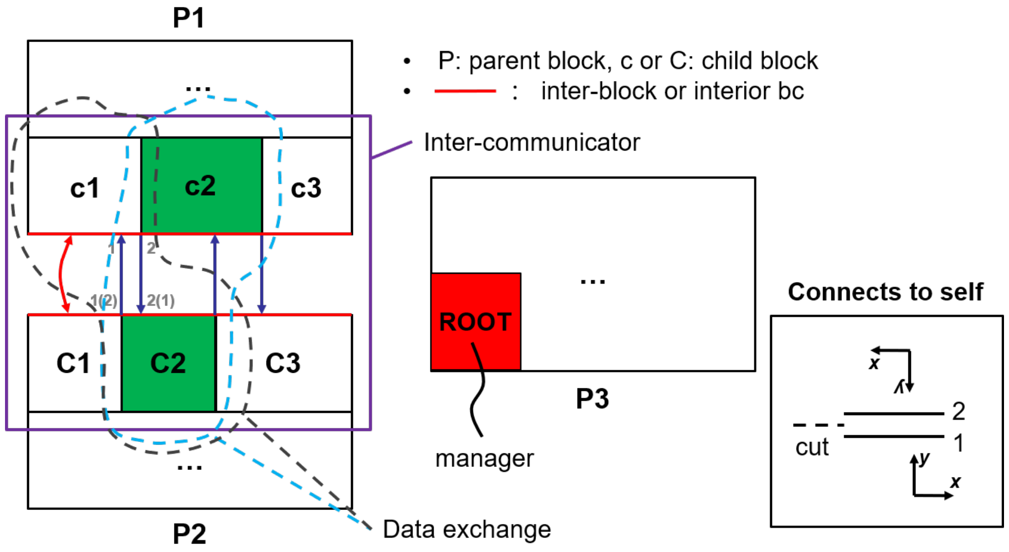


Figure 3.5: An example of using MPI inter-communicator

In SENSEI, nonblocking MPI calls instead of blocking calls are used to improve the performance. However, nonblocking MPI calls requires a blocking call such as MPI_WAIT to finish the communication, and it may cause a deadlock issue for some multi-block cases. An example of the deadlock issue is shown in Fig. 3.6. In this example, there are four processors ($PA{\sim}PD$), each with two connected boundaries ($bc1$ and $bc2$). For every processor, it needs to block a MPI_WAIT call for its $bc1$ to finish first and then for its $bc2$. However, the initial order of boundaries creates a circular dependency issue for all of the processors, and thus no communication can be completed (deadlock). This deadlock issue may happen for both the parent block connections and the child block connections after decomposition. Fig. 3.6 shows a solution to the deadlock issue, i.e. reordering boundaries. Therefore, boundary reordering is implemented in SENSEI to automatically deal with such deadlock issues.

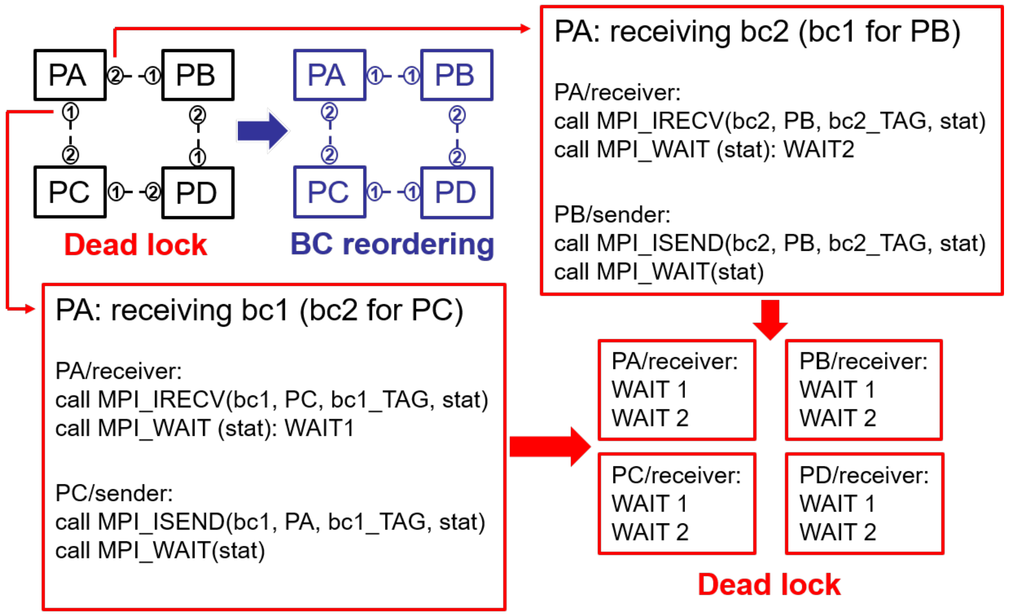

PA
PB
PC
PD
Dead lock
BC reordering
PA: receiving bc2 (bc1 for PB)
PA/receiver:
call MPI_IRECV(bc2, PB, bc2_TAG, stat)
call MPI_WAIT (stat): WAIT2
PB/sender:
call MPI_ISEND(bc2, PB, bc2_TAG, stat)
call MPI_WAIT(stat)
PA: receiving bc1 (bc2 for PC)
PA/receiver:
call MPI_IRECV(bc1, PC, bc1_TAG, stat)
call MPI_WAIT (stat): WAIT1
PC/sender:
call MPI_ISEND(bc1, PA, bc1_TAG, stat)
call MPI_WAIT(stat)
PA/receiver: WAIT 1 WAIT 2
PB/receiver: WAIT 1 WAIT 2
PC/receiver: WAIT 1 WAIT 2
PD/receiver: WAIT 1 WAIT 2
Dead lock

Figure 3.6: An example of deadlock

3.6 Platforms and Metrics

3.6.1 Platforms

Thermisto Thermisto is a workstation in our research lab. It has two NVIDIA Tesla C2075 GPUs and 32 CPU cores. The peak double precision performance is 515 GFLOPS. The compilers used on Thermisto are PGI 16.5 and Open MPI 1.10.0. An compiler optimization of -O4 is used. The GPUs on Thermisto are used mainly for testing and comparison with current generation GPUs.

NewRiver NewRiver [48] is a cluster at Virginia Tech. Each GPU node on NewRiver is equipped with two Intel Xeon E5-2680v4 (Broadwell) 2.4GHz CPUs, 512 GB memory, and two NVIDIA P100 GPUs. Each NVIDIA P100 GPU is capable of up to 4.7 TeraFLOPS of double-precision performance. The NVIDIA P100 GPU offers much higher GFLOPS compared to the NVIDIA C2075 GPU on Thermisto. The compilers used on NewRiver are PGI 17.5 and Open MPI 2.0.0 or MVAPICH2-GDR 2.3b. MVAPICH2-GDR 2.3b is a CUDA-aware MPI wrapper compiler which supports GPUDirect, also available on NewRiver. An compiler optimization of -O4 is used.

Cascades Cascades [49] is another cluster at Virginia Tech. Each GPU node on Cascades is equipped with two Intel Skylake Xeon Gold 3 GHz CPUs, 768 GB memory, and two NVIDIA V100 GPUs. Each NVIDIA V100 GPU is capable of up to 7.8 TeraFLOPS of double-precision performance. The NVIDIA V100 GPU offers the highest GFLOPS among the GPUs we used. The compilers used on Cascades are PGI 18.1 and Open MPI 3.0.0. An compiler optimization of -O4 is used.

3.6.2 Performance Metrics

To evaluate the performance of the parallel code, weak scaling and strong scaling are used. Strong scaling measures how the execution time varies when the number of processors changes for a fixed total problem size, while weak scalability measures how the execution time varies with the number of processors when the problem size on each processor is fixed. Commonly, these two scalings are valuable to be investigated together, as we care more about the weak scaling when we have enough compute resources available to run large problems, while more about the strong scaling when we only need to run small problems. In this paper, since our focus is on the acceleration of the computation and data movement in the iterative solver portion, when measuring productive performance, the timing contribution from the I/O portion (reading in grid, writing out solution) and the one-time domain decomposition is not taken into account.

Two basic metrics used in this paper are parallel speedup and efficiency. Speedup denotes how much faster the parallel version is compared with the serial version of the code, while efficiency represents how efficiently the processors are used. They are defined as follows,

$$\text{speedup} = \frac{t_{serial}}{t_{parallel}} \tag{3.3}$$

$$\text{efficiency} = \frac{\text{speedup}}{np} \tag{3.4}$$

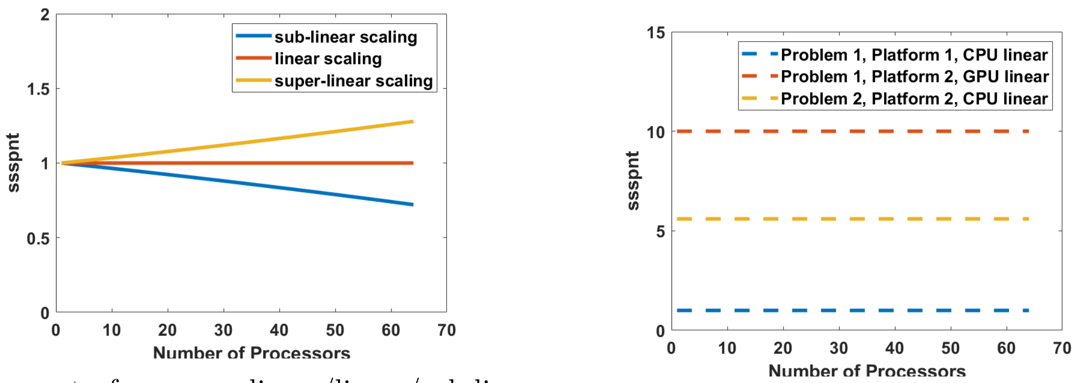
where np is the number of processors (CPUs or GPUs).

In order for the performance of the code to be compared well on different platforms and for different problem sizes, the wall clock time per iteration step is converted to a metric called ssspnt (scaled size steps per np time) which is defined in Eq.3.5.

$$\text{ssspnt} = s\frac{size \times steps}{np \times time} \tag{3.5}$$

where s is a scaling factor which scales the smallest platform ssspnt to the range of [0,1]. In this paper, s is set to be 10^{-6} for all test cases. $size$ is the problem size, $steps$ is the total iteration steps and $time$ is the program solver wall clock time for $steps$ iterations.

Using ssspnt has some advantages. First, GFLOPS requires knowing the number of operations while ssspnt does not. In most codes, especially complicated codes, it is usually difficult to know the total number of operations. The metric ssspnt is a better way of measuring the performance of a problem than the variable $time$ as $time$ may change if conditions (such as the number of iterations, problem size, etc.) change. Second, using ssspnt is clearer in terms of knowing the relative speed difference under different situations than the metric ”efficiency”. It is easy to know whether the performance is super-linear or linear or sub-linear, which is shown in Fig. 3.7a, as well as know the relative performance comparison between different scenarios, which is shown in Fig. 3.7b. Using ssspnt, different problems, platforms and different scalings can be compared more easily.



(a) ssspnt for super-linear/linear/sub-linear scaling

(b) ssspnt for different cases

Figure 3.7: An explanation of ssspn

Similar to Ref [23], every *time* in this paper is measured consecutively for at least three instances. The difference for each *time* point is smaller than 1% (usually less than 1 s out of more than 120 s). We also selected a handful of cases to run again to verify the timings were consistent day to day.

3.7 OpenACC Parallelization and Optimization

There is some general guidance for improving the performance of a program on a GPU. First, sufficient parallelism should be exposed to saturate the GPU with enough computational work, that is, the speedup for the parallel portion should compensate for the overhead of data transfers and the parallel setup. Second, the memory bandwidth between the host and the device should be improved to reduce the communication cost, which is affected by the message size and frequency (if using MPI), memory access patterns, etc. It should be noted that all performance optimizations should guarantee the correctness of the implementation. Therefore, this paper proposes and adopts various modifications to increase the speed of various CFD kernels and reduce the communication overhead while always ensuring the correct result is obtained, i.e., the results do not deviate from the serial implementation.

Load balancing, communication overhead, latency, synchronization overhead and data locality are important factors which may affect the performance. The domain decomposition and aggregation methods used in this paper can help solve the load imbalancing issue well; however, the number of dimensions that need to be decomposed may require tuning, especially when given a large number of processors. To reduce the communication overhead of data transfers between the CPU and the GPU, the data should be kept on the GPU as long as possible without being frequently moved to the CPU. Also, non-contiguous data transfer between the CPU and the GPU (large stride memory access) should be avoided to improve the memory bandwidth. To hide latency, kernel execution and data transfer should be overlapped as much as possible, which may require reordering of some portions in the program. To reduce the synchronization overhead, the number of tasks running asynchronously should be maximized. To improve data locality and increase the use of coalesced fetches, data should be loaded into cache as chunks before needed, which can make read and write more efficiently. This paper addresses some of these issues based on profiling outputs.

We should keep in mind that there are some inherent bottlenecks limiting the actual performance of a CFD code on GPUs. Some CFD codes require data exchange to communicate between partitions, which incurs some communication and synchronization overhead. Data fetching in discrete memory may cost more clock cycles than expected due to low actual memory throughput, system latency, etc. Therefore, the actual compute utilization is difficult to increase sometimes and is application dependent. Another limiter of the performance is the need for branching statements in the code. For instance, certain flux functions might execute different branches depending on the local Mach number. This causes threads in a warp to diverge reducing the peak performance possible. The actual speedup after enough performance optimization should still be smaller than the theoretical compute power the GPU can provide. The relation of the actual and theoretical speedup the GPU can provide is not covered in this paper.

3.7.1 V0: Baseline

The baseline GPU version of SENSEI was implemented by McCall [50], [51]. McCall pointed out that there are some restrictions of the PGI compiler. These restrictions mean the following features cannot be used.

1. Procedure optional arguments
2. Array-valued functions
3. Multi-dimensional array assignments
4. Temporary arrays as parameters to a procedure call
5. Reduction operations on derived type members
6. Procedure pointers within OpenACC kernels

As can be seen in [50] and [51], the 1st and 3rd restrictions do not have adverse effects on the performance. The 2nd restriction can be easily resolved by using Fortran subroutines instead of functions. The 5th restriction can be resolved by using scalar variables or arrays instead of derived type members, which has negligible effect on the performance. The 6th restriction can be easily overcome by using the select case or if statements. The 4th restriction indicates that either the compiler needs to automatically generate the temporary arrays or the user should manually create them. However, the temporary arrays deteriorate the code performance significantly. More details about these restrictions can be found in Ref [50] and [51].

Although the work in Ref [50] and [51] overcame many restrictions to port the code to the GPU, the GPU performance was not satisfactory. A NVIDIA P100 GPU was only 1.3x∼3.4x faster than a single Intel Xeon E5-2680v4 CPU core. which indicates that the GPU was not utilized efficiently. Some performance bottlenecks were fixed in Ref [22]. Profiling-driven optimizations were applied to overcome some performance bottlenecks. First, loops with small sizes were not parallelized as the launch overhead is more expensive than the benefits. As the warp size for NVIDIA GPUs is 32, the compiler may select a thread length of 128 or 256 to parallelize small loops but the loop iteration number for these small loops is less than 10. Second, the kernel of extrapolation to ghost cells was moved from the CPU to the GPU in order to improve the performance, by passing the whole array with indices as arguments. Finally, the kernel of updating corners and edges was parallelized. The eventual speedup of using a single GPU compared to a single CPU was raised to 4.1x for a 3D case on a NVIDIA P100 GPU, but no multi-GPU performance results were shown as the parallel efficiency was not satisfactory.

It should be mentioned that the relative solution differences between the CPU and the GPU code in Ref [22, 50, 51] are much larger than the round-off error mainly due to an incorrect implementation of connected boundary condition and its relevant parallelization. The solution bugs have been fixed in this paper so that the OpenACC framework is extended correctly to multi-block cases. In fact, solution debugging is troublesome using OpenACC, as intermediate results are difficult to check directly on the device. If the data on the GPU side needs to be printed outside of the parallel region, then update of the data on the host side should be made before printing. If the data needs to be known in the parallel region (when a kernel is running), a probe routine which is !$acc routine type should be inserted into the parallel region to print out the desired data. Keep in mind that both the GPU and CPU have a copy of the data with the same name but in discrete memories.

In addition, it should be mentioned that there is a caveat when updating the boundary data between the host and the device using the !$acc update directive since the ghost cells and interior cells in SENSEI are stored and addressed together, which means that the boundary data are non-contiguous in the memory. Much higher memory throughput can be obtained if the whole piece of data (including the interior cells and boundary cells) instead of array slicing is included. A 2D example can be seen in Fig. 3.8. As Fortran uses column-major storage, the memory stores the array elements by column first. However, the interior cell columns split the ghost cells in memory. For 3D or multi-dimensional arrays, the data layout is more complicated but the principle is similar. If using the method in Listing 3.1 to update the boundary data on the device, OpenACC updates the data slice by slice and there are many more invocations. The memory throughput can be about 1/100 to 1/8 of using the method in Listing 3.2, based on the profiling outputs from the NVIDIA visual profiler. In fact, the only implementation difference between the two methods is whether slicing is used or not (in Fortran, array slicing is commonly used), but the performance difference is huge. However, some applications or schemes may require avoiding updating the ghost cell values (due to concerns for solution correctness) at some temporal points when iterating the solver, then a manual data rearrangement, i.e., the pack/unpack optimization, should be applied to overcome the performance deterioration issue.

Listing 3.1: Using slicing to update

```
! IMIN face update
start_indx = 1 - n_ghost_cells(1)
end_indx   = 2
!$acc update device(
```

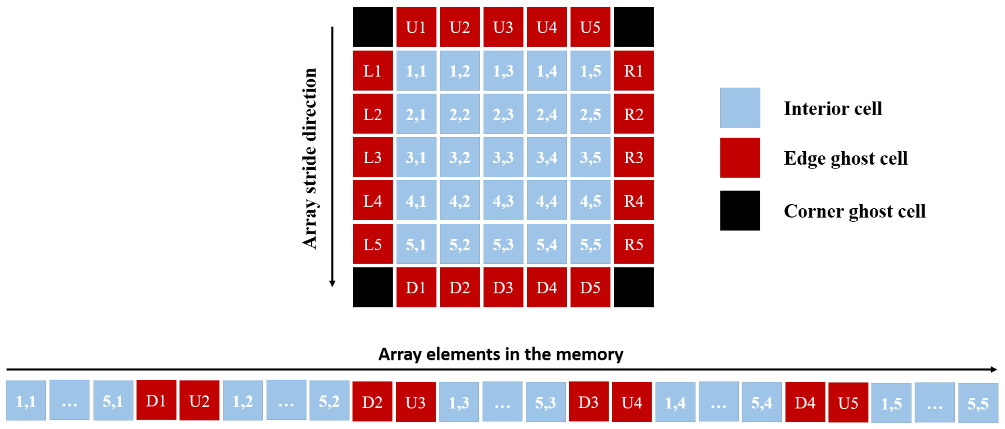


Figure 3.8: An example of showing ghost cells breaking the non-contiguity of the interior cell

```
!$acc soln%sblock(blck)%rho(start_indx:end_indx,1:jmax,1:kmax))
!$acc async(1)
```

Listing 3.2: Update including ghost cells

```
! IMIN face update
start_indx = 1 - n_ghost_cells(1)
end_indx   = 2
!$acc update device(
!$acc soln%sblock(blck)%rho(start_indx:end_indx,:,:))
!$acc async(1)
```

Table 3.1 shows the performance of some metrics for the $V00$ (using Listing 3.1) and $V0$ (using Listing 3.2) comparison on the Cascades platform. Using slicing for the !$acc update directive reduces the memory throughput greatly to about 1% for the device to host bandwidth and about 8% for the host to device bandwidth, compared to not using slicing (with ghost cell data included). Also, the total invocations of using slicing is more than 10 times higher than not using slicing. The last row in Table 3.1 is a reference (NVIDIA profiler reports different fractions of low memory throughput data transfers for different code versions) to show the more serious low memory throughput issue in $V00$. We will show some performance optimizations based on the $V0$ version next, even though $V0$ has larger solution errors than the round-off errors for some cases.

Table 3.1: Comparison of V00 and V0 performance metrics

Metrics	V00	V0
Device to Host bandwidth, GB/s	0.132	10.43
Host to Device bandwidth, GB/s	0.9	7.21
Total invocations, times	262144	20876
Compute Utilization, %	4.2	29.6
Low memory throughput	124 MB/s for 96.4% data transfers	83.88 MB/s for 11.1% data transfers

3.7.2 GPU Optimization using OpenACC

Although parallelized using the GPU in Ref [50, 51] and optimized in Ref [22], the speedup for SENSEI is still not satisfactory due to some performance issues. The NVIDIA Visual Profiler is used to detect various performance bottlenecks. The bottlenecks include low memory throughput, low GPU occupancy, inefficient data transfers, etc. Different architectures and problems may show different behaviours, which is one of our interests. Second, previously the boundary data needed to be transferred to the CPU first in order to exchange data. We will apply GPUDirect to enable data transfers directly between GPUs.

V1: Pack/Unpack The goal of this optimization is to improve the memory throughput and reduce the communication cost if the required data are not located sequentially in memory [23]. As Fortran is a column-majored language, the first index i of a matrix $A(i, j, k)$ denotes the fastest change. A decomposition in the i index direction can generate chunks of data (at $j-k$ planes) which are highly non-contiguous. Decomposing in the j index direction can also cause non-contiguous data transfers. Therefore, the optimization is targeted at solving this issue by converting the non-contiguous data into a temporary contiguous array in parallel using loop for directives and then updating this temporary array between hosts and devices using update directives. Performance gains will be obtained as the threads in a warp can access a contiguous aligned memory region, that is, coalesced memory access is deployed instead of strided memory access. The procedure can be summarized as follows:

1. Allocate send/recv buffers for boundary cells on $j-k$ planes on devices and hosts if decomposition happens in the i dimension, as the non-contiguous data on i planes make data transfer very slow.

2. Pack the noncontiguous block boundary data to the send buffer, which can be explicitly parallelized using !$acc loop directives, then update the send buffer on hosts using !$acc update directives.

3. Have hosts transfer the data through nonblocking MPI_Isend/MPI_Irecv calls and blocking MPI_Wait calls.

4. Update the recv buffer on devices using OpenACC update device directives and finally unpack the contiguous data stored in recv buffer back to noncontiguous memory on devices, which can also be parallelized.

We will show that although extra memory is required for buffers, the memory throughput can be improved to a level similar to that in $V0$ (but $V0$ has larger simulation errors due to the incorrect use of !$acc update, especially for cases having connected boundaries). Using $V1$, only the boundary data on the i boundary faces are packed/unpacked as such data are highly noncontiguous. The boundary data on the j and k plane are not buffered. The pack/unpack can be parallelized using !$acc loop directives so that the computational overhead is very small, which can be seen in Listing 3.3.

Listing 3.3: A pseudo code of showing how to pack/unpack

```
! IMIN face update
start_indx = 1 - n_ghost_cells(1)
end_indx   = interior_cells
!$acc parallel present(soln, soln%sblock,       &
!$acc                  soln%sblock(blck)%rho,  &
!$acc                  soln%sblock(blck)%vel,  &
!$acc                  soln%sblock(blck)%p,    &
!$acc                  soln%sblock(blck)%temp, &
!$acc                  rho_buffer, vel_buffer, &
!$acc                  p_buffer, temp_buffer)
!$acc loop collapse(3)
do k = k_low, k_high
  do j = j_low, j_high
    do i = start_indx, end_indx
      n = n_old + (i - start_indx) + (j - j_low) * i_count + &
          (k - k_low) * j_count * i_count
      rho_buffer(n)   = soln%sblock(blck)%rho(i,j,k)
      vel_buffer(:,n) = soln%sblock(blck)%vel(:,i,j,k)
      p_buffer(n)     = soln%sblock(blck)%p(i,j,k)
      temp_buffer(n)  = soln%sblock(blck)%temp(i,j,k)
    end do
  end do
end do
!$acc end parallel
n = n_old + i_count * j_count * k_count
!$acc update host(rho_buffer(n_old:n-1))   async(1)
```

```
!$acc update host(vel_buffer(:,n_old:n-1)) async(2)
!$acc update host(p_buffer(n_old:n-1))     async(3)
!$acc update host(temp_buffer(n_old:n-1))  async(4)
```

However, when updating the buffer arrays on either side (device or host), since the host only transfers the derived type arrays such as soln%sblock%array not the buffer arrays array_buffer, there is an extra step on the host side to pack/unpack the buffer to/from the derived type array, which can be seen in Listing 3.4. This step may not be needed for some other codes but necessary for SENSEI, as SENSEI uses derived type arrays to store primitive variables. The step adds some overhead to the host side, which will be addressed in $V5$.

Listing 3.4: An extra step to pack/unpack data to the derived type array

```
start_indx = 1 - n_ghost_cells(1)
end_indx   = interior_cells
do k = k_low, k_high
  do j = j_low, j_high
    do i = start_indx, end_indx
      soln%sblock(blck)%rho(i,j,k)    = rho_buffer(n)
      soln%sblock(blck)%vel(:,i,j,k)  = vel_buffer(:,n)
      soln%sblock(blck)%p(i,j,k)      = p_buffer(n)
      soln%sblock(blck)%temp(i,j,k)   = temp_buffer(n)
      n = n + 1
    end do
  end do
end do
```

V2: Extrapolating to ghost cells on the GPU The $V1$ version executes the kernel of extrapolating to ghost cells on the CPU. However, leaving the extrapolation on the CPU may impede further performance improvement as this portion will be the performance bottleneck for the GPU code. Therefore, $V2$ moves the kernel of extrapolating to ghost cells to the GPU. When passing an intent(out) reshaped array which is located in non-contiguous memory locations to a procedure call, the PGI compiler creates a temporary array that can be passed into the subroutine. The temporary array can reduce the performance significantly and poses a threat of cache contention if it is shared among CUDA threads. In fact, whether to support passing slices of array to a procedure call is a discussion for the NVIDIA PGI compiler group internally. To resolve this issue, manually created private temporary arrays are used to enable the GPU to parallelize the extrapolation kernel. An example of how the extrapolation works in SENSEI can be found in Listing 3.5. The data present directive notifies the compiler that the needed data are located in the GPU memory, the data copyin directive copies in the boundary information to the GPU, and the parallel loop directives parallelize the boundary loop iterations. The subroutine set_bc is a device routine which is called in the parallel region. It is difficult for the compiler to automatically know whether there are loops inside the routine, and whether there are dependencies among the loop iterations in the parallel region. The use of !$acc routine seq directive in set_bc informs the compiler such information. After using the temporary arrays such as *rho* and *vel*, each CUDA thread needs to have a copy of the arrays, which occupies a lot of SM registers and thus reduces the concurrency. As can been seen, these temporary arrays are used to store the data in the derived type in the beginning. Then they are used as arguments when invoking the set_bc subroutine. Finally the extrapolated data are copied back to the ghost cells in the original derived type soln.

Listing 3.5: Using temporary array to do the ghost cell data extrapolation

```
!$acc data present(soln, soln%rho, soln%vel, soln%p, &
!$acc               soln%temp, soln%molecular_weight, &
!$acc               grid%grid_vars%volume,            &
!$acc               grid%grid_vars%xi_n, grid, grid%grid_vars) &
!$acc      copyin(bound, bclow, bchigh, n_mmtm)

!$acc parallel
!$acc loop independent
do k = bound%indx_min(3),bound%indx_max(3)
  !$acc loop independent vector private(rho, vel, p, temp, vol)
  do j = bound%indx_min(2),bound%indx_max(2)
    rho(1:length)              = soln%rho(high+1:low:order,j,k)
    vel(1:n_mmtm,1:length)     = soln%vel(:,high+1:low:order,j,k)
    p(1:length)                = soln%p(high+1:low:order,j,k)
    temp(1:length)             = soln%temp(high+1:low:order,j,k)

    vol        = grid%grid_vars%volume(high:low:order,j,k)
    call set_bc(bound%bc_label,                                    &
                rho,                                               &
                vel,                                               &
                p,                                                 &
                temp,                                              &
                molweight,                                         &
                vol,                                               &
                grid%grid_vars%xi_n(:,i,j,k),                      &
                bclow,                                             &
                bchigh,                                            &
```

```
                    n_mmtm)


        soln%rho(high+1:low:order,j,k)              = rho(1:length)
        soln%vel(1:n_mmtm,high+1:low:order,j,k)     = vel(:,1:length)
        soln%p(high+1:low:order,j,k)                = p(1:length)
        soln%temp(high+1:low:order,j,k)             = temp(1:length)
      end do
    end do
    !$acc end parallel
    !$acc end data
```

V3: Removal of Temporary Variables Either the automatic or the manual use of temporary arrays in $V2$ can greatly deteriorate the GPU performance. Instead of passing array slices to a subroutine, the entire array was passed with the indicies of the desired slice as shown in Listing 3.6, which avoids the use of temporary arrays. This method requires many subroutines to be modified in SENSEI. However, it saves the use of shared resources and improves the concurrency.

Listing 3.6: Passing derived type data and index range

```
    !$acc data present(soln, soln%rho, soln%vel, soln%p, &
    !$acc              soln%temp, soln%molecular_weight, &
    !$acc              grid%grid_vars%volume,            &
    !$acc              grid%grid_vars%xi_n, grid, grid%grid_vars) &
    !$acc       copyin(bound, bclow, bchigh, n_mmtm)

    !$acc parallel
```

```
!$acc loop independent
do k = bound%indx_min(3),bound%indx_max(3)
  !$acc loop independent vector
  do j = bound%indx_min(2),bound%indx_max(2)
    call set_bc(bound%bc_label,                   &
                grid,                             &
                soln,                             &
                soln%rho,                         &
                soln%vel,                         &
                soln%p,                           &
                soln%temp,                        &
                molweight,                        &
                grid%grid_vars%volume,            &
                grid%grid_vars%xi_n(:,i,j,k),     &
                bclow,                            &
                bchigh,                           &
                j,                                &
                k,                                &
                n_mmtm,                           &
                boundary_lbl,                     &
                normal_lbl)

  end do
end do
!$acc end parallel
!$acc end data
```

V4: Splitting flux calculation and limiter calculation For cases which require the use of limiters, the CPU calculates the left and right limiters on a face once, as the next loop iteration can reuse two limiter values without computing them again, which can be seen in Eq. 3.6.

$$\vec{Q}^L_{i+1/2} = \vec{Q}_i + \frac{\epsilon}{4}[(1-\kappa)\Psi^+_{i-1/2}(\vec{Q}_i - \vec{Q}_{i-1}) + (1+\kappa)\Psi^-_{i+1/2}(\vec{Q}_{i+1} - \vec{Q}_i)] \tag{3.6}$$

$$\vec{Q}^R_{i+1/2} = \vec{Q}_{i+1} - \frac{\epsilon}{4}[(1+\kappa)\Psi^+_{i+1/2}(\vec{Q}_{i+1} - \vec{Q}_i) + (1-\kappa)\Psi^-_{i+3/2}(\vec{Q}_{i+2} - \vec{Q}_{i+1})] \tag{3.7}$$

where ϵ and κ are MUSCL extrapolation parameters, Ψ are limiter function values. L and R denote the left and right states, respectively.

After porting the code to the GPU, since SENSEI calculates the limiters locally for each solution state (in $V0$ through $V3$), the limiter cannot be reused as different CUDA threads have their own copies of four limiter values, otherwise thread contention may occur. To fix this issue, the total cost of the limiter calculation on the GPU is twice of that on the CPU. Also, storing the limiter locally requires the limiter calculation and flux extrapolation to be together, which is highly compute intensive. $V4$ uses global arrays to store these limiters so that the flux calculation and limiter calculation can be separated, which is given in listing 3.7. This approach will leave more room for kernel concurrency and asynchronization and also avoid thread contention.

Listing 3.7: Splitting MUSCL extrapolation and limiter calculation

```
! xi limiter
!$acc parallel
!$acc loop independent collapse(3)
do k = 1, k_cells
```

```
    do j = 1, j_cells

      do i = 1, imax-1
        call limiter_subroutine_x( sblock, gblock, i, j, k,  &
                                   sblock%limiter_xi%left,   &
                                   sblock%limiter_xi%right )
      end do
    end do
  end do
  !$acc end parallel

  ! xi flux
  !$acc parallel
  !$acc loop independent collapse(3) private(qL, qR)
  do k = 1, k_cells
    do j = 1, j_cells
      do i = 2, imax-1

        call muscl_extrapolation_xi( sblock, i, j, k, &
                  sblock%limiter_xi%left(1:neq,i-1,j,k),  &
                  sblock%limiter_xi%left(1:neq,i,j,k),    &
                  sblock%limiter_xi%right(1:neq,i,j,k),   &
                  sblock%limiter_xi%right(1:neq,i+1,j,k), &
                  qL, qR )

        call flux_function(qL, qR, &
                         gblock%grid_vars%xi_n(:,i,j,k), &
```

```
                             sblock%xi_flux(1:neq,i,j,k))

          end do

        end do
      end do
      !$acc end parallel
```

V5: Derived type for connected boundaries on the GPU The previous versions update the connected boundaries between the host and the device through using local dynamic arrays. Therefore, it is worthwhile to investigate the effect of using global derived type arrays to store the connected boundary data. It removes the extra data copies on the host side mentioned in $V1$. An example of using the global derived type is given in Listing 3.8. If there is no communication required among different CPU processors, the MPI functions are not called.

Listing 3.8: Derived type for connected boundary data

```
    !$acc update host(grid%gblock(blck)%bcs_acc(nc)%rho_send(    &
    !$acc             1:idx_max_nbor(1)-idx_min_nbor(1)+1,       &
    !$acc             1:idx_max_nbor(2)-idx_min_nbor(2)+1,       &
    !$acc             1:idx_max_nbor(3)-idx_min_nbor(3)+1))

    ! SEND and RECV derived type boundary data
    call MPI_IRECV( grid%gblock(blck)%bcs_acc(nc)%rho_recv,      &
                    scalar_count, MPI_DOUBLE_PRECISION,          &
                    bound%bound_nbor%process_id, RHO_TAG,        &
                    world_comm, req(req_count+1), ierr )
```

```
call MPI_ISEND( grid%gblock(blck)%bcs_acc(nc)%rho_send,     &
                scalar_count, MPI_DOUBLE_PRECISION,          &
                bound%bound_nbor%process_id, RHO_TAG,        &
                world_comm, req(req_count+5), ierr )

call MPI_WAITALL(req_count, req(1:req_count),                &
                 stat(:,1:req_count), ierr)

!$acc update device(grid%gblock(blck)%bcs_acc(nc)%rho_recv( &
!$acc                buff_size_self(1)*buff_size_self(2)*    &
!$acc                buff_size_self(3)))
```

V6: Change of blocking call locations Since SENSEI is a multi-block CFD code, a processor may hold multiple blocks and many connected boundaries. Using MPI non-blocking routines, there should be a place to execute the blocking call such as MPI_WAIT to complete the communications. Each Isend/Irecv call needs one MPI_WAIT, or multiple MPI_WAIT can be wrapped up into one MPI_WAITALL. The previous versions block the MPI_WAITALL call for every decomposed block. A newer way of achieving the function is moving the MPI_WAITALL calls to a new loop, so that these MPI_WAITALL calls are executed after all Isend & Irecv are executed. An example is given in Fig. 3.9. In this example, there are two blocks, each having two connected boundaries. However, $V6$ only improves the performance when multiple connected boundaries exist.

For platforms in which the asynchronous progression is supported completely (from both the software and hardware sides), this optimization may work much better. However, for

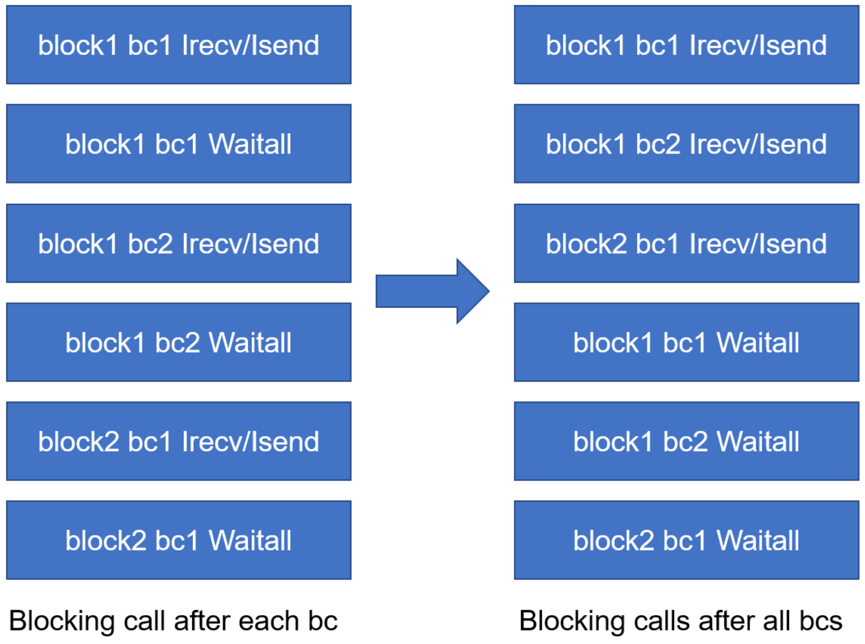


Figure 3.9: Change of blocking call position

common platforms in which the asynchronous progression is not supported fully, OpenMP may need to be used to promote the asynchronous progression [52, 53, 54, 55, 56]. Full asynchronous progression is a very complicated issue and is not covered in this paper. This paper will only apply MPI+OpenACC to accelerate the CFD code.

V7: Boundary flux optimization In SENSEI, the fluxes for the wall and farfield boundaries need to be overwritten to get more accurate estimate for the solution. These overwritten flux calculations are done after the boundary enforcement. For these two kinds of fluxes, the previous versions do not compute them very efficiently. A lot of temporary variables are allocated for each thread, which deteriorates the concurrency of using OpenACC, as registers are limited. The principle of this optimization is similar to that in $V3$. An example of the optimization is given in Listing 3.9.

Listing 3.9: Optimization of the overwritten boundary flux kernel

```
! V0 ~ V6

!$acc parallel copyin(i, bound) async(1)
!$acc loop independent
do k = bound%indx_min(3), bound%indx_max(3)
  !$acc loop independent vector private(      &
  !$acc       soln_L2, soln_L1, soln_R1,       &
  !$acc       soln_R2, qL, qR, modf,           &
  !$acc       lim_L2, lim_L1, lim_R1, lim_R2,  &
  !$acc       vel_xi, rho_xi, p_xi, temp_xi)
  do j = bound%indx_min(2), bound%indx_max(2)

! V7

!$acc parallel copyin(i, bound) async(1)
!$acc loop independent
do k = bound%indx_min(3), bound%indx_max(3)
  !$acc loop independent vector private(      &
  !$acc       qL, qR, modf)
  do j = bound%indx_min(2), bound%indx_max(2)
```

V8: Asynchronicity improvement Kernels from different streams can be overlapped so that the performance can be improved. The version is exactly the same as that in $V7$ but the environment variable "PGI_ACC_SYNCHRONOUS" is set to 0 when executing SENSEI, that is, asynchronization among some independent kernels is promoted. The !$acc wait directive makes the host wait until asynchronous accelerator activities finish, i.e., it is the synchronization on the host side.

V9: Removal of implicit data copies between the host and device The last performance optimization is essentially manual tuning work. It requires the user to modify the code through profiling. The compiler sometimes does not know what variables are to be updated between the host and the device, so for the reason of safety the compiler may update variables frequently, which may be unnecessary. Different architectures and compilers may deal with the update differently, therefore the user can optimize it based on the profiler outputs. The compiler may transfer some scalar variables, arrays with small size and even derived type data in every iteration, but they only need to be copied once. There are multiple places in SENSEI where the PGI compiler makes unnecessary copies. These extra unnecessary data transfers are usually small in size and deteriorate the memory throughput. The effect of these copies can be significant for small size problems. However, for compute-intensive computations, this optimization may not be very useful. This performance optimization is only applied for the P100 GPU and V100 GPU, with the newer version of PGI compiler. Running with $V9$ on the C2075 returns some linker errors due to the old PGI compiler.

V10: GPUDirect GPUDirect is an umbrella word for several GPU communication acceleration technologies. It provides high bandwidth and low latency communication between NVIDIA GPUs. There are three levels of GPUDirect [57]. The first level is GPUDirect Shared Access, introduced with CUDA 3.1. This feature avoids an unnecessary memory copy within host memory between the intermediate pinned buffers of the CUDA driver and the network fabric buffer. The second level is GPUDirect Peer-to-Peer transfer (P2P transfer) and Peer-to-Peer memory access (P2P memory access), introduced with CUDA 4.0. This P2P memory access allows buffers to be copied directly between two GPUs on the same node. The last is GPU RDMA (Remote Direct Memory Access), with which buffers can be sent from the GPU memory to a network adapter without staging through host memory. The last feature is not supported on NewRiver as it pertains to specific versions of the drivers (from NVIDIA and Mellanox for the GPU and the Infiniband, respectively) which are not installed (other dependencies exist, particularly parallel filesystems). Although GPU RDMA is not available, the other aspects of GPUDirect can be utilized to further improve the scaling performance on multiple GPUs.

3.8 Solution and Scaling Performance

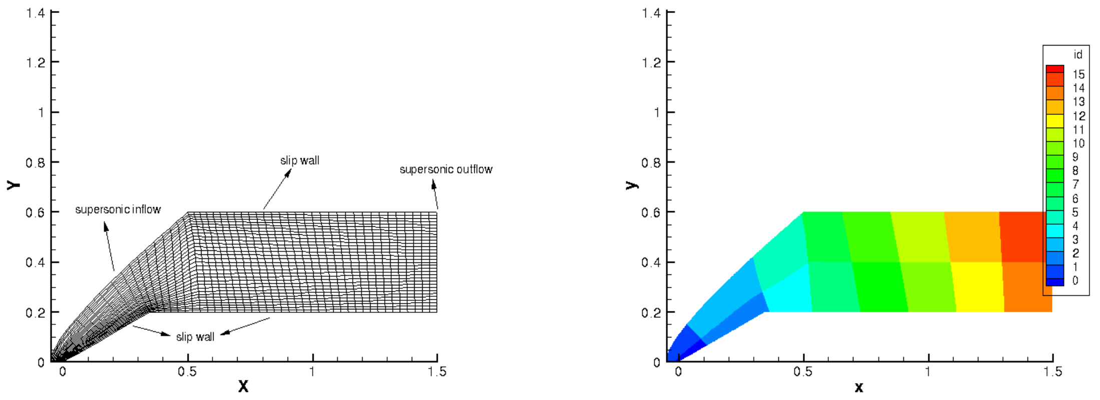
3.8.1 Supersonic Flow Through a 2D Inlet

The first test case is a simplified 2D 30 degree supersonic inlet, which has only one parent block without having connected boundaries. The inflow conditions are given in Table 3.2. There are multiple levels of grid for strong and weak scaling analysis, of which the total amount of cells range from 50k to 7 million. The parallel solution and the serial solution have been compared from the beginning to the converged state during the iterations, and the relative errors for all the primitive variables based on the inflow boundary values is within round-off error range (10^{-12}).

Table 3.2: Inlet case inflow boundary conditions

Mach number	4.0
Pressure	12270 Pa
Temperature	217 K

A very coarse level of grid for the 2D inlet flow is shown in Fig. 3.10a. The decomposition of using 16 GPUs (which is the highest number of GPUs available) on a 416x128 grid is shown in Fig. 3.10b. The decomposition is 2D, creating multiple connected boundaries between processors. Ghost cells on the face of connected boundaries are used to exchange data between neighboring processors. The device needs to communicate with the host if multiple processors are used.



(a) A coarse (52x16) grid for the 2D inlet Euler flow

(b) A domain decomposition for the 2D inlet Euler flow (using 16 GPUs)

Figure 3.10: 2D Euler supersonic inlet

The relative residual L_2 norm history is shown in Fig. 3.11. It can be seen that the iterative errors have been driven down small enough for all the primitive variables when converged. The Mach number and density solutions are shown in Fig. 3.12. There are multiple flow deflections when the flow goes through the reflected oblique shocks.

Fig. 3.13 shows the performance comparison of different optimizations using different flux options on different platforms. The grid size used in Fig. 3.13 is 416×128. The goal of making such a comparison is to investigate the effect of using various flux options, time marching schemes and various generation GPUs when applying the optimizations introduced earlier in this paper. For such a small problem which does not have any connected boundary conditions, a single P100 GPU is about 3 times faster than the a single C2075 GPU. We expect that the speedup would be higher if the problem size was larger. Another observation

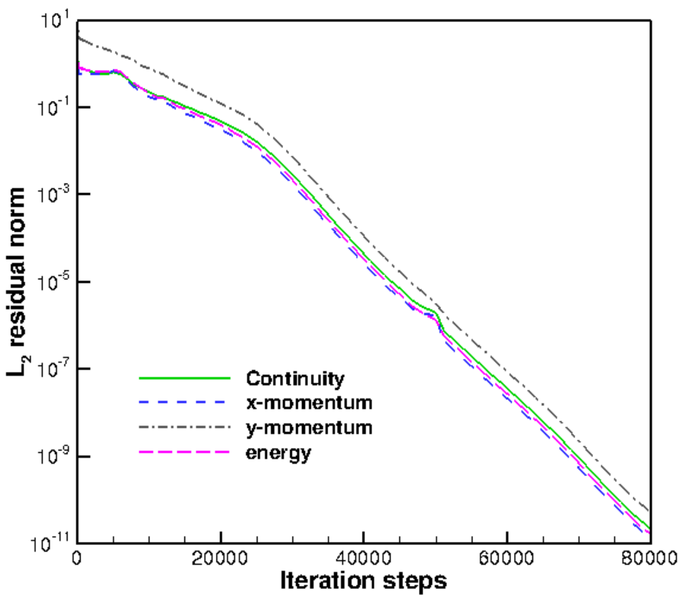
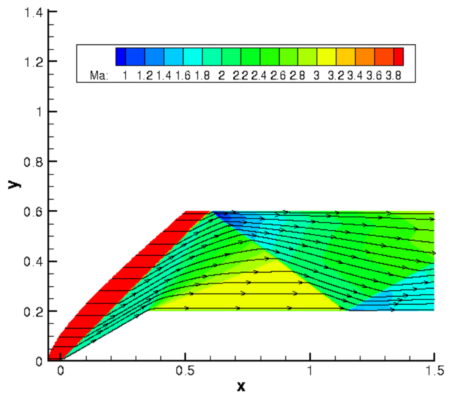


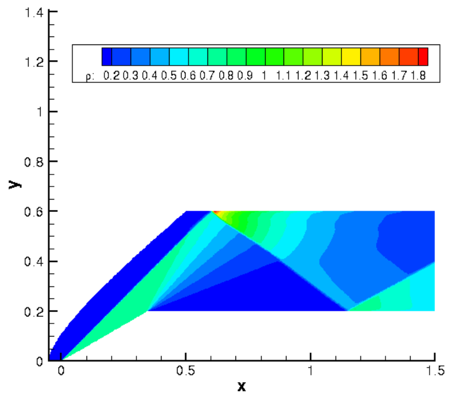
Figure 3.11: The relative iterative residual history for the inlet case

is that using the Roe flux is slightly slower than using the van Leer flux, which is reasonable as the Roe flux is a bit more expensive than the van Leer flux. It should be kept in mind that the ssspnt metric does not take the number of double precision operations for each step into account so ssspnt is not equivalent to GFLOPS. Also, the speed of RK2 and RK4 is comparable, so this paper will stick to the use of RK2 unless otherwise specified.

If comparing the performance of different versions in Fig. 3.13, there are two performance leaps including from $V2$ to $V3$ and from $V8$ to $V9$. Since the extrapolation to ghost cells on the GPU runs inefficiently in $V2$ due to the low compute utilization, removing the use of temporary arrays in the parallel regions reduces the overhead from CUDA threads. More concurrency in the code can therefore be utilized by the GPU. From $V8$ to $V9$, since the problem size is small (the compute fraction is not very high), removing unnecessary data movement improves the overall performance by more than 52%. For larger problems, the performance gain is not that significant, as we will show later. In the meantime, there is



(a) The Mach number and streamlines for the 2D inlet Euler flow



(b) The density solution for the 2D inlet Euler flow

Figure 3.12: 2D Euler supersonic inlet

a gradual performance improvement from $V3$ to $V4$ and $V6$ to $V8$. These optimizations should not be overlooked as the issues related to the optimizations will eventually become bottlenecks. Since this case does not have connected boundaries, there is no obvious performance change from $V4$ to $V6$. It should be mentioned that the performance optimizations proposed earlier are not for only a specific case, but for general cases with multiple blocks and connected boundaries.

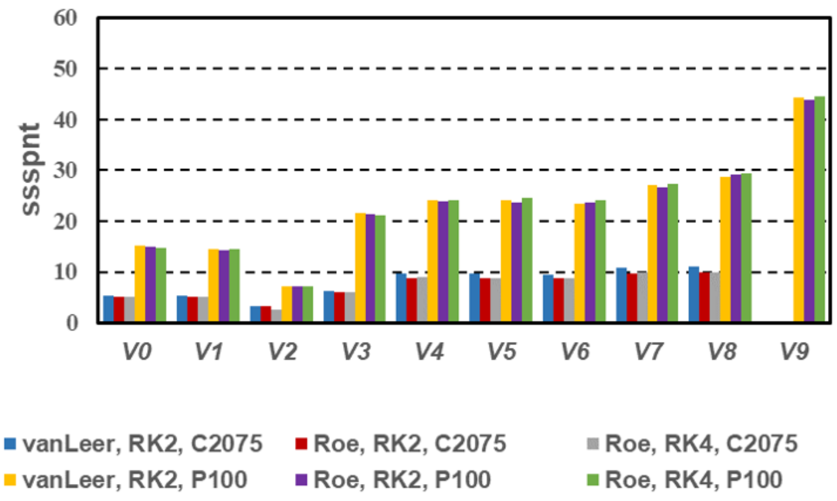
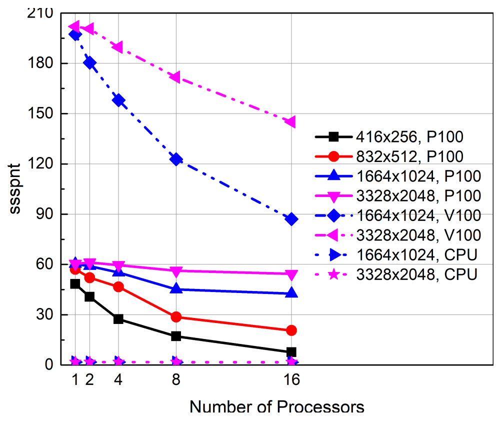
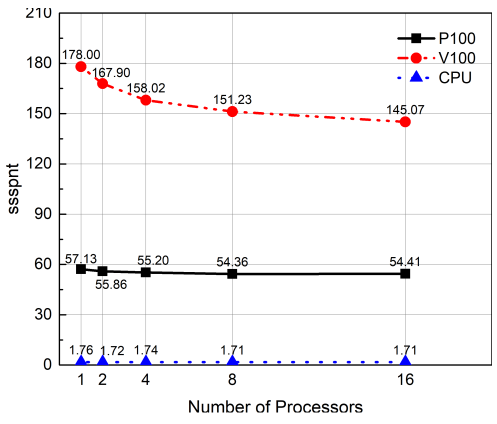


Figure 3.13: Performance comparison for the 2D inlet Euler flow

Fig. 3.14a and Fig. 3.14b show the strong and weak scaling performance for the 2D inlet Euler flow, respectively. The CPU scaling performance is also given for reference. A single P100 GPU is more than 32× faster than a single CPU, on a grid level of 416x256, which displays the compute power of the GPU. The strong scaling efficiency decays quickly for small problem sizes but not for the largest problem size in Fig. 3.14a. The parallel efficiency using 16 P100 GPUs on the 3328×2048 grid is still kept higher than 90%. While for the weak scaling, the parallel efficiency is higher (95.2% above) than the strong scaling efficiency, as there is more work to saturate the GPU. The V100 GPU shows higher speedups but lower efficiency, because the V100 GPU needs more computational work as it is faster. The boundary connections for this inlet flow case after the domain decomposition are not complicated, which is one important reason why the performance is very good.



(a) Strong scaling



(b) Weak scaling

Figure 3.14: The scaling performance for the 2D inlet case

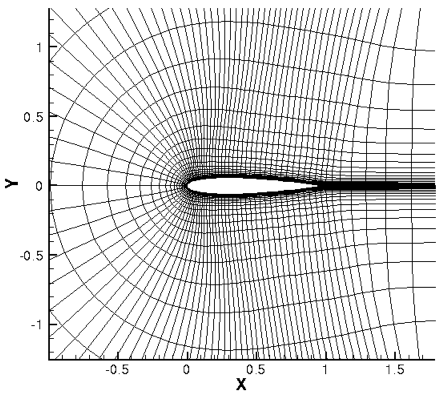
3.8.2 2D Subsonic Flow past a NACA 0012 Airfoil

The second test case in this paper is the 2D subsonic flow ($M_\infty = 0.25$) past a NACA 0012 airfoil, at an angle of attack of 5 degrees. The flow field for all the simulation runs of this case is initialized using the farfield boundary conditions which are given in Table 3.3. This case will be solved by both the Euler and laminar NS solvers in SENSEI.

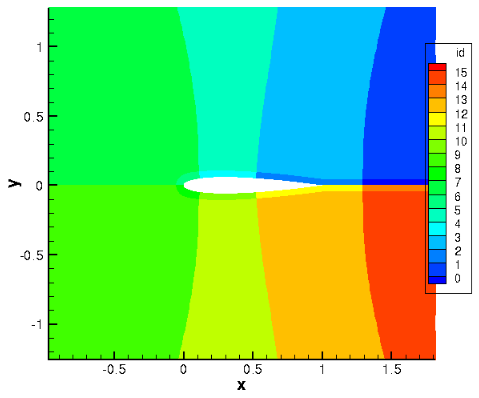
Table 3.3: NACA 0012 airfoil farfield boundary conditions

Mach number	0.25
Static pressure	84307 Pa
Temperature	300 K
Angle of attack, α	5 degrees

Although the airfoil case contains only one parent block, the grid is a C-grid, which means that the only one block connects to itself on a face through a connected boundary, which makes the airfoil case different from the 2D inlet flow case. One coarse grid of this airfoil case is shown in Fig. 3.15a. For the scaling analysis, the grid size ranges from 400k to 6 million. Also, the domain decomposition of using 16 GPUs is shown in Fig. 3.15b. Near the airfoil surface, the grid is refined locally so processors near the wall take smaller blocks, but the load is balanced.



(a) A coarse (128x48) grid for the flow past a NACA 0012 airfoil

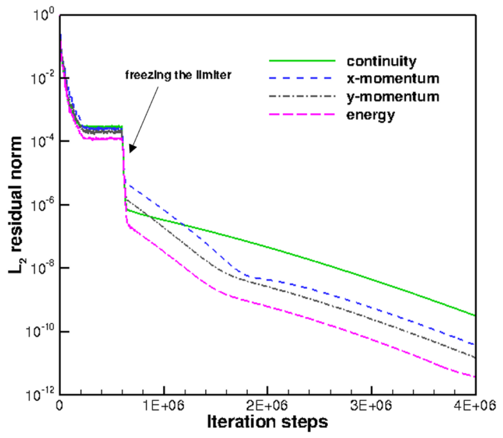


(b) The domain decomposition for the airfoil case (using 16 GPUs)

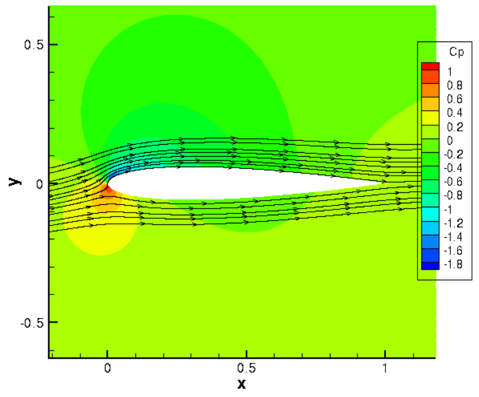
Figure 3.15: 2D NS NACA 0012 airfoil

Fig. 3.16a shows the relative iterative residual L_2 norm history for the laminar NS subsonic flow past a NACA 0012 airfoil. This case requires the most iteration steps to be converged among all the test cases considered. Leveraging the compute power of the GPU saves a lot of time. To enable the iterative residual to further go down instead of oscillation, limiter freezing is adopted at around 600k steps. After freezing the limiter, the iterative residual norms continue to reduce smoothly. The iterative errors are driven down small enough to obtain the steady state solution.

The parallel solution and the serial solution have been compared on coarse levels of grid and the relative errors for all the primitive variables based on the reference values are within round-off error range (10^{-12}). Fig.3.16b shows the pressure coefficient solution and the streamlines for the laminar NS subsonic flow past the NACA 0012 airfoil.



(a) The relative iterative residual L_2 norm history for the laminar NS subsonic flow past a NACA airfoil



(b) The pressure coefficient contour for the laminar NS subsonic flow past a NACA airfoil

Figure 3.16: 2D Laminar NS NACA 0012 airfoil

Fig. 3.17 shows the comparison of different versions for the flow past a NACA 0012 airfoil using a single P100 GPU. Laminar NS has a smaller ssspnt (about 70%) compared to using the Euler solver. From $V2$ to $V3$, the speedup is more than 2 times on different levels of grid, for both the Euler and laminar NS solver. To use globally allocated derived types to store the connected boundary data cannot improve the performance, which can be seen from the comparison of $V4$ and $V5$, if only using one processor, as there are no MPI communication calls. Although the airfoil case has a connected boundary, the data in the ghost cells for that boundary are filled directly through copying. This case only has one connected boundary, so there is no need to reorder the non-blocking MPI I_send/I_recv calls and the MPI_Wait call. Similarly to the 2D inlet case, on coarse levels of grid, there is noticeable performance improvement if applying the optimization in $V9$. On fine levels of grid, the benefit is limited.

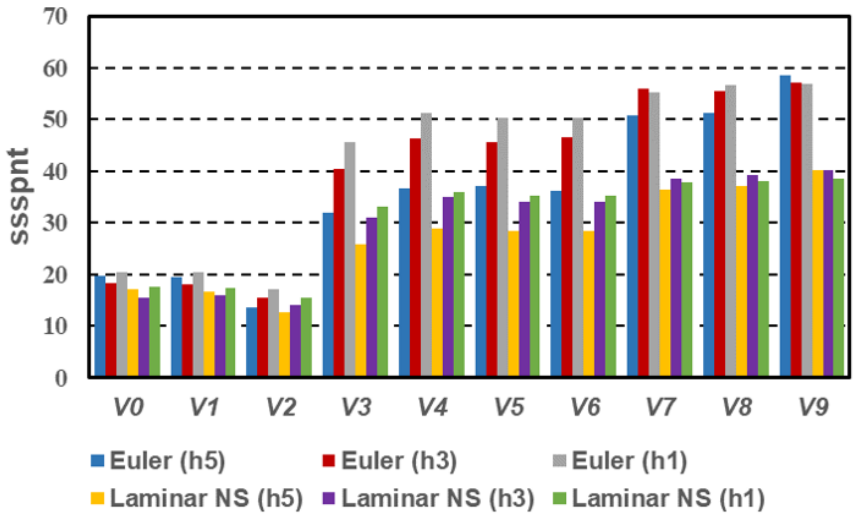


Figure 3.17: The performance of different versions for the NACA 0012 airfoil case (P100 GPU)

Since we cannot see any performance gain from $V5$ to $V6$ using single GPU, multiple GPUs are used to show the benefits. For all the runs shown in Fig. 3.18, $V6$ (the red bars) outperforms $V5$ (the blue bars) by 4% to 50%, depending on the solver type, grid level and number of GPUs used. After applying multiple GPUs, multiple connected boundaries are created, which creates margin for the reordering of I_send/I_recv and Wait to work. Intrinsically, this ordering is to propel more asynchronous progression on the implementation side. The actual overlap degree still highly depends on the communication system, which is out of the scope of this paper. Readers who are interested in more overlap and better asynchronous progression may try the combination of MPI+OpenACC+OpenMP.

Fig. 3.19 and Fig. 3.20 show the strong and weak scaling performance of this subsonic flow

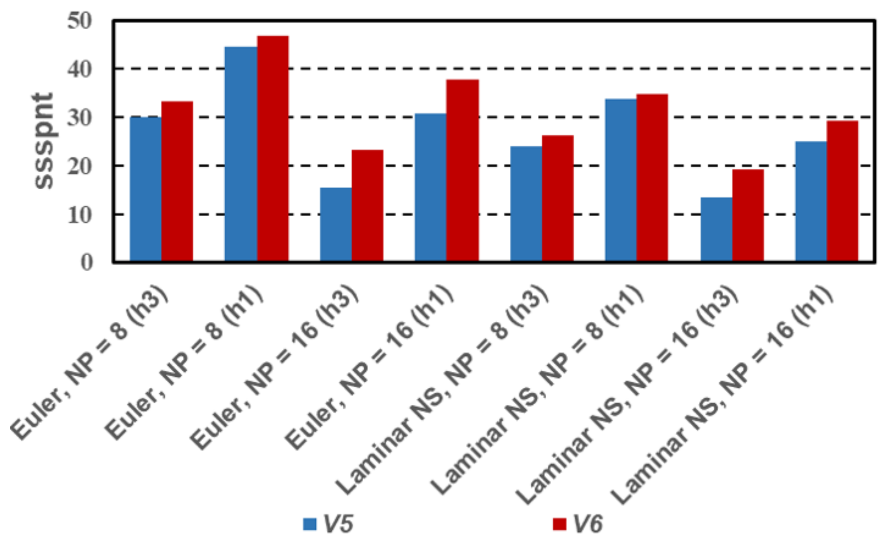
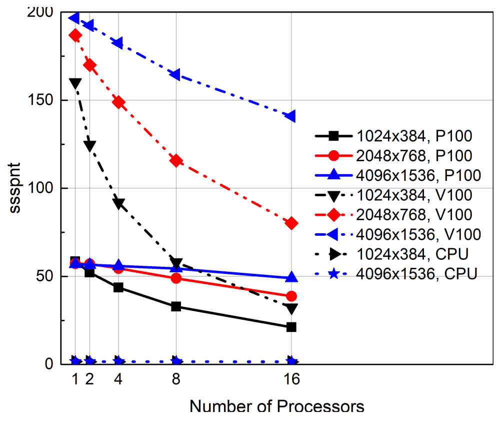


Figure 3.18: Performance comparison between $V5$ and $V6$ for the NACA 0012 airfoil case (P100 GPU)

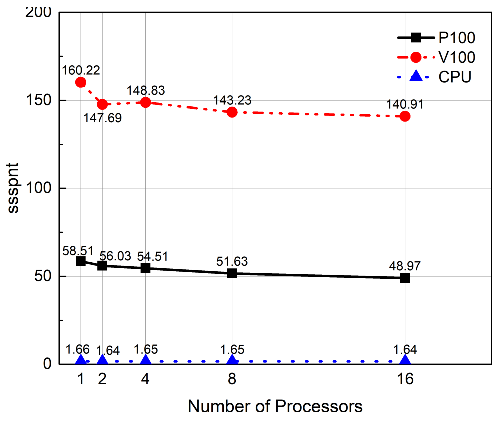
past a NACA 0012 airfoil solved by the Euler and laminar NS solver on P100 and V100 GPUs, respectively. They show very similar behaviours with the only difference in the scales. Overall, the laminar ssspnt is about 0.7 of the Euler ssspnt using multiple GPUs. The strong parallel efficiency on the 4096×1536 grid using 16 P100 GPUs for the Euler and laminar NS solver is about 87% and 90%, respectively. The weak scaling efficiency is generally higher as there is more work to do for the GPU. The efficiencies using V100 GPUs are lower than those using P100 GPUs, which indicates that faster GPUs may need more computational work to hold high efficiency.

3.8.3 3D Transonic Flow Past an ONERA M6 Wing

The final case tested in this paper is the 3D transonic flow ($M_\infty = 0.839$) past an ONERA M6 wing, at an angle of attack of 3.06 degrees [58]. The flow field is initialized using the farfield boundary conditions which are given in Table 3.4. Both the Euler and laminar NS solvers in SENSEI are used to solve this problem. Different from the previous two 2D

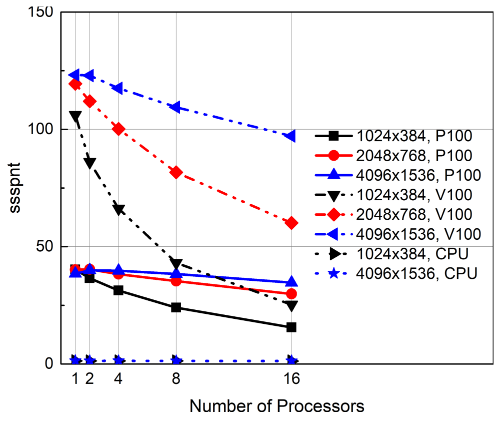


(a) Strong scaling (Euler)

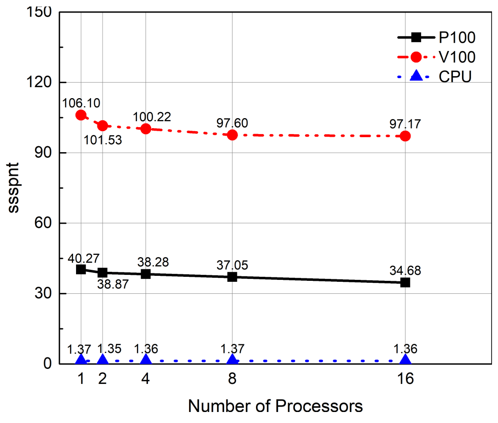


(b) Weak scaling (Euler)

Figure 3.19: The scaling performance for the 2D Euler flow past a NACA 0012 airfoil



(a) Strong scaling (Laminar NS)



(b) Weak scaling (Laminar NS)

Figure 3.20: The scaling performance for the 2D laminar NS flow past a NACA 0012 airfoil

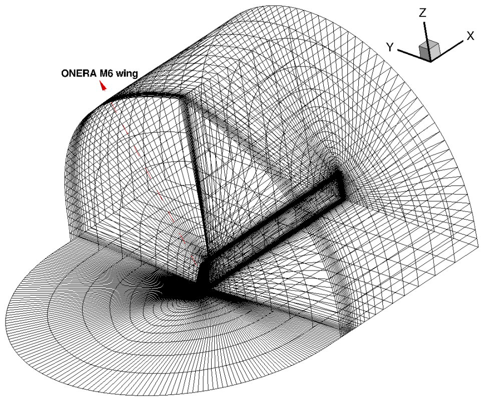
problems, this 3D case has 4 parent blocks with various sizes. Under some conditions (when using 2 and 4 processors in this paper), domain aggregation is needed to balance the load on different processors. This 3D wing case has a total grid size ranging from 300k to 5 million.

The parallel solution and the serial solution of the wing case have been compared to each other on a coarse mesh and the relative errors for primitive variables based on the farfield boundary values is within round-off error (10^{-12}).A coarse level of grid and the domain decomposition of using 16 GPUs are given in Fig. 3.21a and Fig. 3.21b, respectively. The relative iterative residual L_2 norm history and the pressure coefficient (C_p) contour using

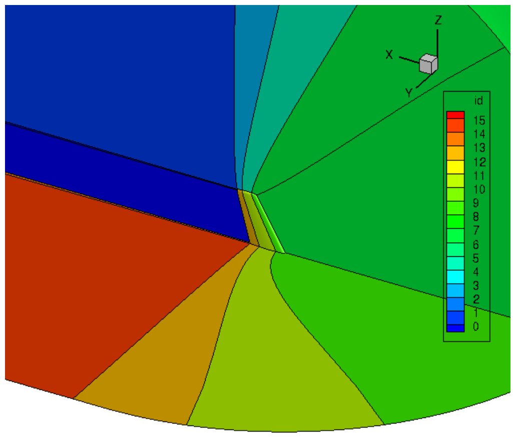
Table 3.4: ONERA M6 wing farfield boundary conditions

Mach number, M_∞	0.8395
Temperature, T_∞	255.556 K
Pressure, p_∞	315979.763 Pa
Angle of attack, α	3.06 degrees

the laminar NS solver in SENSEI are given in Fig. 3.22a and Fig.3.22b, respectively. From Fig. 3.22a, it can be seen that the iterative errors have been driven down small enough.



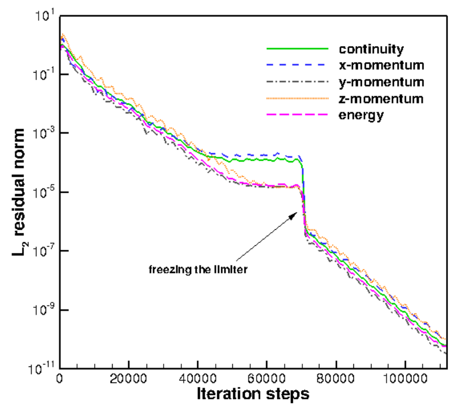
(a) A grid for the ONERA M6 wing



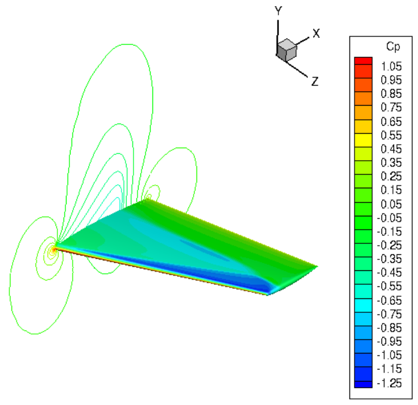
(b) The domain decomposition for the ONERA M6 wing case using 16 GPUs

Figure 3.21: Grid and domain decomposition for ONERA M6 wing

Since this wing case is 3D and has multiple parent blocks, we are interested in whether the performance optimizations introduced earlier can improve the performance of this wing case. Fig. 3.23 shows the performance of different versions for the ONERA M6 wing case. From the grid level of $h5$ to $h1$, the grid refinement factor is 2 (refined in z, y and x cyclically). $V2$ runs slower than $V1$ for all levels of grid, indicating that the extrapolation to ghost cells on the GPU is not as efficient as that on the CPU, although it is parallelized. With proper optimization, $V3$ is about 3 to 4 times faster than $V2$, which is similar to the previous two 2D cases. From $V3$ to $V4$, there is a performance drop for almost all runs, no matter what the grid level and the solver is. Splitting one kernel into two kernels for this case incurs



(a) The relative residual norm history for ONERA M6 wing



(b) The Laminar NS pressure coefficient contour for ONERA M6 wing

Figure 3.22: Residual history and solution for ONERA M6 wing

some overhead and reduces the compute utilization a bit. There is a slight performance improvement from $V4$ to $V5$ when using the derived type to buffer the boundary data for connected boundaries. The data will be allocated in the main memory of the GPU before needed, which outperforms the use of dynamic data to buffer the boundary data. For a single GPU, $V5$ and $V6$ perform equivalently fast. Further performance optimization on the boundary flux calculation can improve the performance significantly, which can be seen from $V6$ to $V7$. Carefully moving the data between the host and the device can improve the performance on coarse levels of grid, but not on very fine levels of grid, as the computation becomes more dominant when refining the grid.

Although the wing case has multiple parent blocks, there is no MPI communication if using only a single GPU. Therefore, there is only negligible difference between $V5$ and $V6$. Similar to the NACA 0012 airfoil case, multiple GPUs are used to show the effect of reordering the non-blocking MPI I_send/I_recv calls and the MPI_Wait calls. Fig. 3.24 shows that there are some performance gains for some runs but not all. $V6$ accelerates the code by 14% to 18% when np is equal to 8. If using 16 GPUs, more connected boundaries are

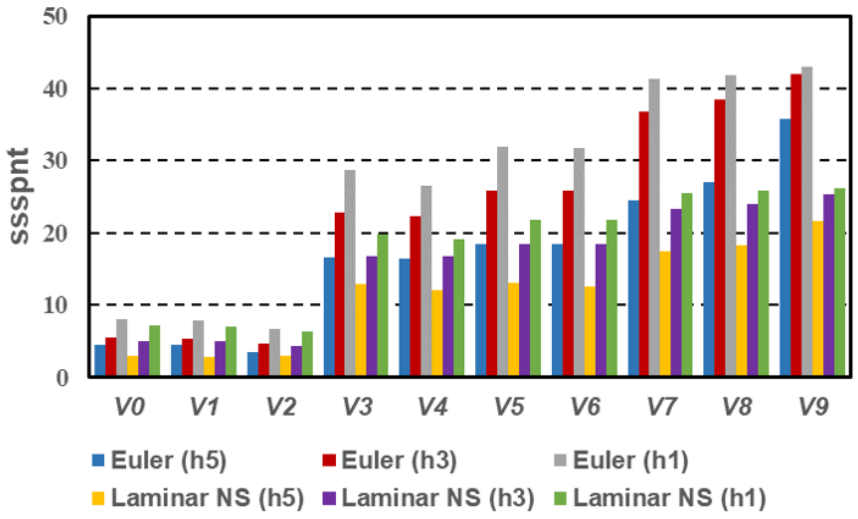


Figure 3.23: The single P100 GPU performance of different versions for ONERA M6 wing

created, and it impedes the performance improvement. A possible reason for this may be that although the implementation from $V5$ to $V6$ exposes more asynchronous progression on the implementation side, the platform communication system does not support that very well when too many communication calls are invoked. This issue may be resolved if switching to the MPI+OpenACC+OpenMP model, which is not covered in this paper. However, it can be seen that the performance degradation using 16 GPUs is only 0.8% to 3%, which is small.

Fig. 3.25 and Fig. 3.26 show the strong and weak scaling performance using Euler and laminar NS solvers, respectively. Some different behaviours show as in this case some processors need to hold multiple blocks, which is different from the 2D inlet and 2D NACA 0012 case. A single GPU is about 33 times faster than a single CPU on the $h5$ level grid. The weak scaling of the GPU keeps good efficiency over the whole np range shown in Fig. 3.25b and Fig. 3.26b.

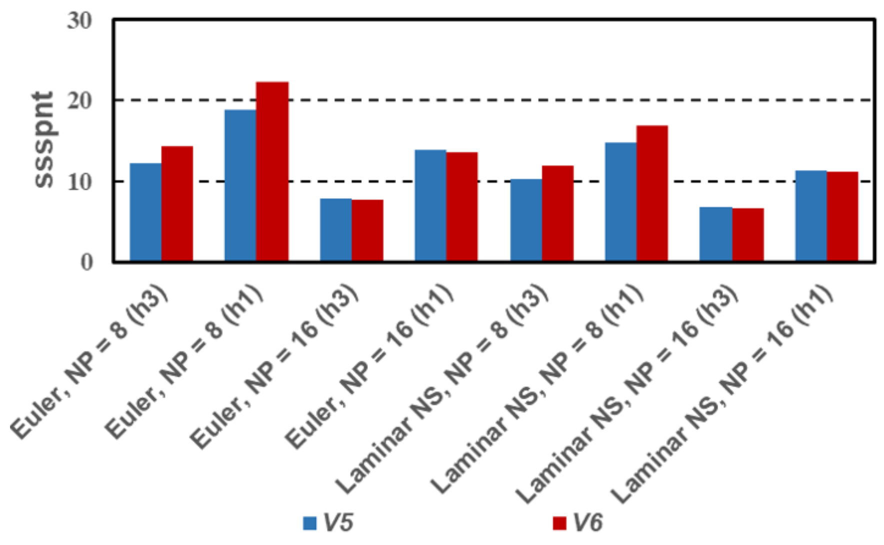
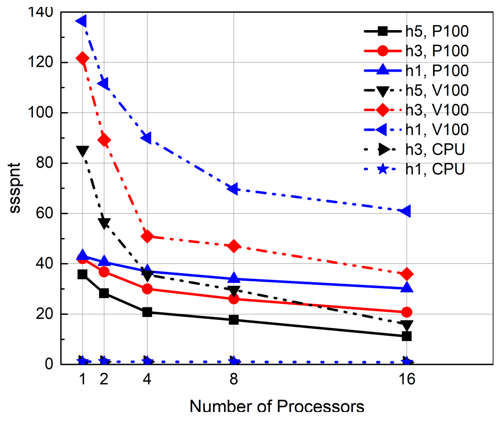


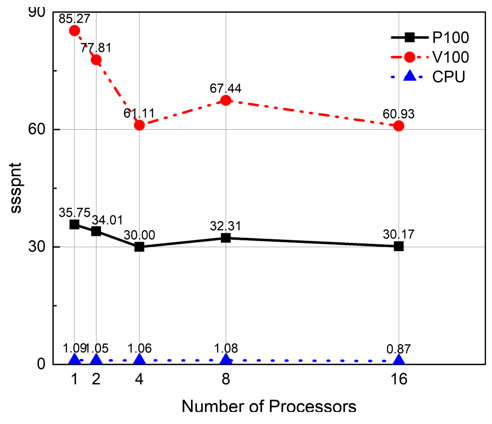
Figure 3.24: Performance comparison between $V5$ and $V6$ for ONERA M6 wing

3.8.4 GPUDirect

Since GPUDirect is not a general performance optimization, as it requires some support from both the compiler side and the communication system side, a comparison of $V9$ and $V10$ is made at the end to give readers more insights of the effect of GPUDirect. GPUDirect was applied to the 2D Euler/laminar flow past the NACA 0012 airfoil and the transonic flow over the 3D ONERA M6 wing. It should be noted that there is no guarantee that using GPUDirect can improve the performance substantially without the hardware support like using NVLink (however both the NewRiver and the Cascades cluster does not have NVLink so the memory bandwidth is still not high enough). It can be found that the two cases show different behaviours when applying GPUDirect, seen in Fig. 3.27. For the NACA 0012 case, generally $V10$ is slower than $V9$, which means that GPUDirect makes the code to run slower. However, for the ONERA wing case, using GPUDirect improves the performance by 4% to 14%. Whether there is a performance gain or not depends on the problem and number of communications. Commonly if high memory bandwidth NVLink is available, GPUDirect

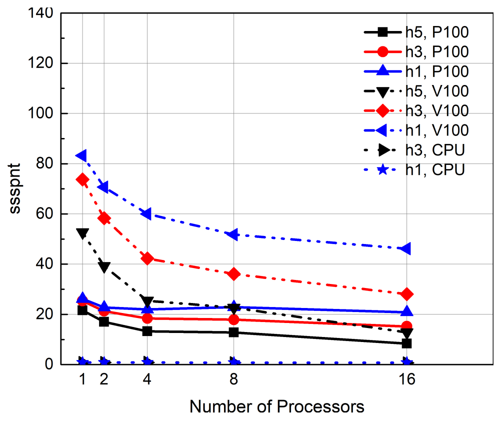


(a) Strong scaling (Euler)

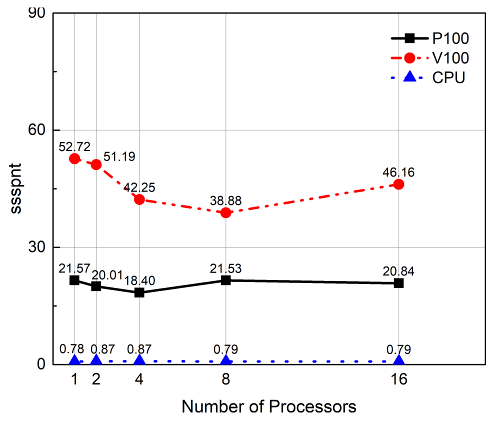


(b) Weak scaling (Euler)

Figure 3.25: The scaling performance for the 3D Euler ONERA M6 wing case



(a) Strong scaling (Laminar NS)



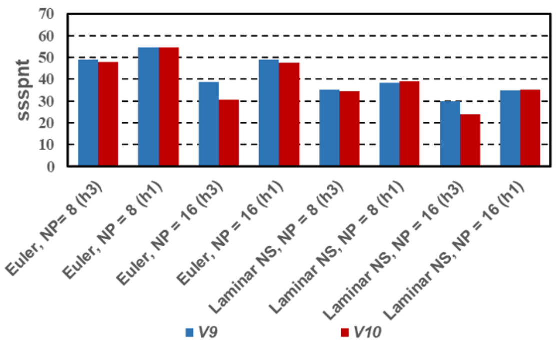
(b) Weak scaling (Laminar NS)

Figure 3.26: The scaling performance for the 3D Laminar NS ONERA M6 wing case

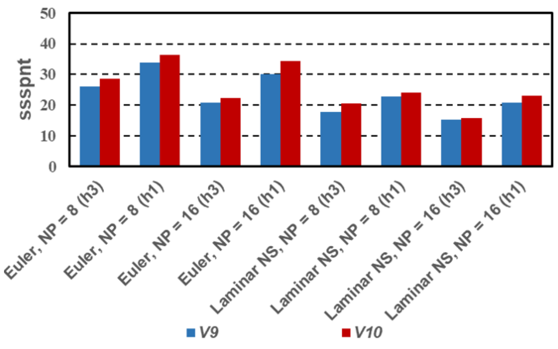
should be more beneficial to the performance.

3.9 Conclusions & Future Work

An improved framework using MPI+OpenACC is developed to accelerate a CFD code on multi-block structured grids. OpenACC has some advantages in terms of the ease of programming, the good portability and the fair performance. A processor-clustered domain decomposition and a block-clustered domain aggregation method are used to balance the



(a) Subsonic flow past a NACA 0012 airfoil



(b) Transonic flow past an ONERA M6 wing

Figure 3.27: Performance comparison between $V9$ and $V10$

workload among processors. Also, the communication overhead is not high using the domain decomposition and aggregation methods. A parallel boundary decomposition method is also proposed with the use of the MPI inter-communicator functions. The boundary reordering for multi-block cases is addressed to avoid the dead lock issue when sending and receiving messages. A number of performance optimizations are examined, such as using the global derived type to buffer the connected boundary data, removing temporary arrays when making procedure calls, reordering of blocking calls for non-blocking MPI communications for multi-block cases, using GPUDirect, etc. These performance optimizations have been demonstrated to improve single GPU performance more than up to 5 times compared to the baseline GPU version. More importantly, all the three test cases show good strong and weak scaling up to 16 GPUs, with a good parallel efficiency if the problem is large enough.

For the future work, more complicated 3D multi-block cases on more GPUs can be tested. Also, CPU can be assigned some computational work (instead of just I/O and some control instructions) so that SENSEI can be accelerated further.

Bibliography

[1] Blaise Barney. Introduction to Parallel Computing, 2020. (last accessed on 07/24/20).

[2] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. An Investigation of Unified Memory Access Performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Waltham, MA, US, 2014. IEEE.

[3] Blaise Barney. OpenMP, 2020. (last accessed on 07/24/20).

[4] Blaise Barney. Message Passing Interface (MPI), 2020. (last accessed on 07/24/20).

[5] NVIDIA. CUDA C++ Programming Guide, 2019. (last accessed on 07/24/20).

[6] Khronos OpenCL Working Group. The OpenCL Specification, 2012. (last accessed on 07/24/20).

[7] OpenACC-Standard.org. *The OpenACC Application Programming Interface.* OpenACC-Standard.org, 2018.

[8] N Gourdain, L Gicquel, M Montagnac, O Vermorel, M Gazaix, G Staffelbach, M Garcia, JF Boussuge, and T Poinsot. High performance parallel computing of flows in complex geometries: I. methods. *Computational Science & Discovery*, 2(1):015003, 2009.

[9] N Gourdain, L Gicquel, G Staffelbach, O Vermorel, Florent Duchaine, JF Boussuge, and Thierry Poinsot. High performance parallel computing of flows in complex geometries: Ii. applications. *Computational Science & Discovery*, 2(1):015004, 2009.

[10] Amit Amritkar, Surya Deb, and Danesh Tafti. Efficient parallel cfd-dem simulations using openmp. *Journal of Computational Physics*, 256:501–519, 2014.

[11] Zdravko Krpic, Goran Martinovic, and Ivica Crnkovic. Green hpc: Mpi vs. openmp on a shared memory system. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 246–250. IEEE, 2012.

[12] Pablo D Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid mpi–openmp scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, 37(6-7):316–326, 2011.

[13] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[14] JA Herdman, WP Gaudin, Simon McIntosh-Smith, Michael Boulton, David A Beckingsale, AC Mallinson, and Stephen A Jarvis. Accelerating hydrocodes with openacc, opencl and cuda. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 465–471. IEEE, 2012.

[15] Dana A Jacobsen and Inanc Senocak. Multi-level parallelism for incompressible flow computations on gpu clusters. *Parallel Computing*, 39(1):1–20, 2013.

[16] Erich Elsen, Patrick LeGresley, and Eric Darve. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

[17] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 23(3):777–786, 2004.

[18] Tobias Brandvik and Graham Pullan. Acceleration of a 3d euler solver using commodity graphics hardware. In *46th AIAA aerospace sciences meeting and exhibit*, page 607, 2008.

[19] Lixiang Luo, Jack R Edwards, Hong Luo, and Frank Mueller. Performance assessment of a multiblock incompressible navier-stokes solver using directive-based gpu programming in a cluster environment. In *52nd Aerospace Sciences Meeting*, 2013.

[20] Yidong Xia, Jialin Lou, Hong Luo, Jack Edwards, and Frank Mueller. Openacc acceleration of an unstructured cfd solver based on a reconstructed discontinuous galerkin method for compressible flows. *International Journal for Numerical Methods in Fluids*, 78(3):123–139, 2015.

[21] Dominic D Chandar, Jayanarayanan Sitaraman, and Dimitri J Mavriplis. A hybrid multi-gpu/cpu computational framework for rotorcraft flows on unstructured overset grids. In *21st AIAA Computational Fluid Dynamics Conference*, page 2855, 2013.

[22] Weicheng Xue, Charles W Jackson, and Christopher J Roy. Multi-cpu/gpu parallelization, optimization and machine learning based autotuning of structured grid cfd codes. In *2018 AIAA Aerospace Sciences Meeting*, page 0362, 2018.

[23] Weicheng Xue and Christopher J Roy. Multi-gpu performance optimization of a cfd code using openacc on different platforms. *arXiv preprint arXiv:2006.02602*, 2020.

[24] Weicheng Xue and Christopher J Roy. Heterogeneous computing of cfd applications on cpu-gpu platforms using openacc directives. In *AIAA Scitech 2020 Forum*, page 1046, 2020.

[25] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[26] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model im-

plementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[27] Weicheng Xue, Hongyu Wang, and Christopher J Roy. Code verification for 3d turbulence modeling in parallel sensei accelerated with mpi. In *AIAA Scitech 2020 Forum*, page 0347, 2020.

[28] William L Oberkampf and Christopher J Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[29] Bram Van Leer. Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method. *Journal of computational Physics*, 32(1):101–136, 1979.

[30] Antony Jameson, Wolfgang Schmidt, and Eli Turkel. Numerical solution of the euler equations by finite volume methods using runge kutta time stepping schemes. In *14th fluid and plasma dynamics conference*, page 1259, 1981.

[31] Uri M Ascher, Steven J Ruuth, and Raymond J Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[32] Christopher A Kennedy and Mark H Carpenter. Diagonally implicit runge-kutta methods for ordinary differential equations. a review. 2016.

[33] JC Wu, LT Fan, and LE Erickson. Three-point backward finite-difference method for solving a system of mixed hyperbolic—parabolic partial differential equations. *Computers & chemical engineering*, 14(6):679–685, 1990.

[34] Philip L Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[35] Joseph L Steger and RF Warming. Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods. *Journal of computational physics*, 40(2):263–293, 1981.

[36] Bram Van Leer. Flux-vector splitting for the euler equation. In *Upwind and High-Resolution Schemes*, pages 80–89. Springer, 1997.

[37] Single instruction, multiple threads, 2020. (last accessed on 05/10/20).

[38] Blaise Barney. Message Passing Interface (MPI), 2019.

[39] Open mpi documentation, 2020. (last accessed on 05/10/20).

[40] Mvapich: Mpi over infiniband, omni-path, ethernet/iwarp, and roce, 2020. (last accessed on 05/10/20).

[41] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143. IEEE, 2013.

[42] Furqan Baig, Chao Gao, Dejun Teng, Jun Kong, and Fusheng Wang. Accelerating spatial cross-matching on cpu-gpu hybrid platform with cuda and openacc. *Frontiers Big Data*, 3:14, 2020.

[43] Victor Artigues, Katharina Kormann, Markus Rampp, and Klaus Reuter. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurrency and Computation: Practice and Experience*, 32(11):e5640, 2020.

[44] Pgi compiler user's guide, 2019. (last accessed on 05/10/20).

[45] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[46] Rob Farber. *Parallel programming with OpenACC*. Newnes, 2016.

[47] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[48] Newriver, 2019.

[49] Cascades, 2020.

[50] Andrew James McCall. *Multi-level Parallelism with MPI and OpenACC for CFD Applications*. PhD thesis, Virginia Tech, 2017.

[51] Andrew J McCall and Christopher J Roy. A multilevel parallelism approach with mpi and openacc for complex cfd codes. In *23rd AIAA Computational Fluid Dynamics Conference*, page 3293, 2017.

[52] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping communication and computation in mpi by multithreading. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006.

[53] Karthikeyan Vaidyanathan, Dhiraj D Kalamkar, Kiran Pamnany, Jeff R Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. Improving concurrency and asynchrony in multithreaded mpi applications using software offloading. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[54] Huiwei Lu, Sangmin Seo, and Pavan Balaji. Mpi+ ult: Overlapping communication and computation with user-level threads. In *2015 IEEE 17th International Conference*

on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pages 444–454. IEEE, 2015.

[55] Alexandre Denis and François Trahay. Mpi overlap: Benchmark and analysis. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 258–267. IEEE, 2016.

[56] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moreto, Martin Schulz, Ramon Beivide, Mateo Valero, and Abhinav Bhatele. Optimizing computation-communication overlap in asynchronous task-based programs. In *Proceedings of the ACM International Conference on Supercomputing*, pages 380–391, 2019.

[57] NVIDIA. NVIDIA GPUDirect, 2019.

[58] M Mani, J Ladd, A Cain, R Bush, M Mani, J Ladd, A Cain, and R Bush. An assessment of one-and two-equation turbulence models for internal and external flows. In *28th Fluid Dynamics Conference*, page 2010, 1997.

Chapter 4

Heterogeneous Computing of CFD Applications on a CPU-GPU Platform using MPI and OpenACC

Weicheng Xue[1] and Christopher J. Roy[2]

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. The heterogeneous CPU/GPU framework in SENSEI was implemented by the first author. All the results were collected and analyzed by the first author.

- Christopher J. Roy (second author): The second author provided valuable feedback for this study and comments for this manuscript.

[1]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.

[2]Professor, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 330, Virginia Tech, 460 Old Turner St, AIAA Associate Fellow.

Abstract

This paper deals with the CPU-GPU heterogeneous computing of a multi-block structured grid CFD code using multiple CPUs and GPUs. First, a brief description on how the heterogeneous computing framework is implemented using MPI and OpenACC is given. OpenACC directives are added for GPUs so most of the CFD code base is the same for CPUs and GPUs except for some specific performance optimizations for the GPU which have no significant effect on the CPU performance. Then, different domain decomposition strategies and a domain aggregation method along with their advantages tailored for our research CFD code, our applications, and the accelerator used are presented. The speedup and scaling performance of the CPU-GPU heterogeneous computing and the comparison with multi-CPU/GPU performance for several test cases including a supersonic inlet, a NACA 0012 airfoil and an ONERA M6 wing are given. Using a GPU or two GPUs and a certain number of CPUs together, the program solver time can be reduced compared with using pure CPUs or GPUs. Finally, conclusions are drawn to provide 1) suggestions for programmers who have an interest in developing their own CPU-GPU heterogeneous computing framework and 2) feedback for hardware researchers who are working on the design of the future CPU-GPU heterogeneous systems.

Keywords: Multi-GPU, OpenACC, MPI, Domain Decomposition, Performance Optimization, GPUDirect

4.1 Introduction

Heterogeneous computing has been receiving a lot of attention in recent years. As of November 2019 (the latest release), a majority of the top 10 supercomputers in the Green500 list [1] use heterogeneous processing units (CPU & accelerators). Similarly, most of the top 10 in the Top500 list [2] use accelerators (specifically GPUs), including Summit and Sierra, both of which went online in 2018. This displays the popularity of heterogeneous systems in the High Performance Computing world. There are several reasons behind their popularity. The first reason is heterogeneous systems are more energy-efficient compared with conventional pure-CPU systems. Power is not the leading factor considered in this paper but it is among the most important ones when designing a supercomputer. The second reason is that heterogeneous systems can provide heterogeneity for different needs in various applications. To simplify, one application such as Computational Fluid Dynamics (CFD) is highly compute-intensive when using a lot of cells or nodes, and it has the feature of Single Instruction Multiple Data (SIMD). In the CPU-GPU heterogeneous model, the compute-intensive components can be offloaded to the GPUs, which are suitable for SIMD. A modern GPU usually has thousands of processing units (PUs) so one instruction on the data can be executed in parallel, thus reducing about an order of magnitude of execution time or more. However, the model just described is not a pure heterogeneous model, as when GPUs are doing computation work, CPUs may just idle, although many researchers have been using the word ”heterogeneous” frequently even if their applications are not purely heterogeneous in this sense. In a more rigorous heterogeneous mode which is the focus of this paper, part of the computation is done on CPUs and the rest is done on GPUs. In this way, we can make full use of the computational resources in nodes and in the meantime get better performance compared with using single type PUs to do the computation work. Mittal et al. [3] surveyed a lot of heterogeneous computing techniques at the runtime, algorithm, programming, and application level. Heterogeneous systems can have CPUs, GPUs, MICs, FPGAs, etc. They reviewed both discrete and fused CPU-GPU systems carefully as CPU-GPU systems are the most popular system. In discrete systems, hosts and devices have different memory addresses so data transfer happens over the PCIe bus, which may incur large overhead and serious latency issues. They pointed out that matching algorithmic requirements to features of PUs is very important.

Running applications on heterogeneous systems may bring some performance benefits and this technique has aroused a lot of interest in different fields, not just limited to the CFD area. Domanski et al. [4, 5] concluded that the CPU-GPU heterogeneous approach can provide performance benefits over either pure-CPU or pure GPU system alone, and the heterogeneous implementation produces even better performance when the ratio of CPU cores over GPUs in a node is higher. Alvarez et al. [6] presented an algebra-based framework with a heterogeneous MPI+OpenMP+OpenCL implementation. The heterogeneous implementation shows a gain of 32% compared to the GPU-only implementation. Sometimes in a heterogeneous system, different PUs can do different kinds of tasks to display their advantages, because of the fact that GPUs are rich in compute capacity but poor in memory capacity and the CPUs do the opposite. In [7], the branch instruction task such as boundary condition enforcement and data transfer are done on the CPU, while the compute-intensive part such as flux calculation are done on the GPU. Their work was only for one CPU-GPU pair and they also did not compare pure GPU performance with their heterogeneous CPU-GPU performance, so there are limitations in their conclusions. There are other heterogeneous systems tailored for other needs or constraints besides just using a CPU-GPU system. Wang et al. [8] implemented an "MPI+OpenMP+offload" heterogeneous framework for large-scale CFD flow field simulations on heterogeneous CPU-MIC system, with lots of CPUs and MIC coprocessors being employed. In this framework, each process can offload some share of its work to the corresponding MIC devices and it scales up to half of the Tianhe-2 supercomputer system. Because the GPU usually has a high latency when communicating to the global memory, and the specific application requires an ultra-low latency, Liu et al. [9] used an FPGA-based approach. The characteristics of FPGAs enables them to optimize the heterogeneous nature of their implementation. However, there are some researchers who hold the view that the heterogeneous implementations are slower. Nikolic [10] evaluated general systems of differential and algebraic equations on different architecture systems and found that the heterogeneous approach dose not provide performance benefits compared with one single GPU. They attributed this contradictory result to the fact that the code being used was not optimized very well.

Also, since the heterogeneous systems have different PUs with different architectures, memory hierarchy, compute capabilities, memory access preference, execution models (SIMD, MIMD, etc.), programming models, different bandwidth requirement, load imbalance issue, extra imported communication overhead due to the use of heterogeneity etc., programming on heterogeneous systems is troublesome. Therefore, specifically for the CPU-GPU heterogeneous system, which is also an emphasis in this paper, the characteristics of CPU and GPU need to be considered carefully.

The two key issues for heterogeneous computing investigated in this paper are load imbalance and communication overhead. They are non-trivial issues for heterogeneous computing and require careful attention and suitable optimizations. Domanski et al. [4] used GPUs and multi-core CPUs ranging from a single workstation to large GPU clusters for solving some computational problems. They suggested that it was good to keep all data on GPUs, thus reducing the communication overhead between CPUs and GPUs. Since their application does not require communication between CPUs and CPUs, or GPUs and GPUs, it is infeasible to extend the work to other areas, especially for CFD applications which usually require communication when iterating. Kovac et al. [11] found that fully utilizing the resources of both CPUs and GPUs requires a carefully balanced heterogeneous approach. The heterogeneous and asynchronous approach in their work achieves a higher speedup compared with pure GPUs. Liu et al. [12] used a hybrid linear solver for a CFD application which is the classical Lid Driven Cavity on CPU-GPU heterogeneous platforms, and proposed a method of using functional performance models to distribute work between heterogeneous processors units. In [8], an efficient parallelization and optimization method is investigated. The communication between CPUs and MICs is implemented using directive statements and good load balancing is obtained by adjusting the ratio of the workload of CPUs/MICs. In [6], it is found that the scalability drops faster for the heterogeneous system than the GPU-only implementation. The reason for this is the computational load over communication cost ratio drops on the heterogeneous system. Therefore, to obtain high efficiency, the load on devices should be sufficiently high. For multi CPU-GPU systems, load balancing and communication overhead should be balanced well. For example, a good conventional dynamic decomposition algorithm may schedule the workload very well but generate large communication overhead due to inter-block boundaries created. It is valuable to assess which one of the two factors is the more important bottleneck, on different systems, for different applications, etc. For some problems in which the communication overhead is difficult to reduce, GPUDirect [13] may be a viable option as it allows peer-to-peer GPU communication. However, the communication overhead between CPUs may also be an important bottleneck to the performance.

Apart from the difficulties mentioned above, a good metric for heterogeneous computing performance should also be investigated. For homogeneous system, scalability analysis is usually adopted to assess whether an implementation scales well or not by adding more homogeneous cores. However, for the CPU-GPU heterogeneous system specifically, how the scalability should be measured when adding more CPUs, or GPUs, or using a different task ratio between different PUs remains a question. Moreover, the scaling performance depends on the domain decomposition strategies. Overall, the scalability analysis becomes more complicated due to the heterogeneity of the system.

As with the programming model used in this paper, OpenACC and MPI are selected. OpenACC is a high-level programming standard which enables users to program readily on heterogeneous systems, including multicore/manycore CPU or GPU accelerators. It provides three different levels of parallelism for loops: gang, worker and vector, which is very similar to the relevant parallelism provided by CUDA or OpenCL. Since OpenACC cannot automatically divide loops across CPUs and GPUs at the same time due to the difficulty of managing the data between discrete memories [14], we need to build up two versions of routines, most part of which are the same except for that the GPU version has some OpenACC directives the CPU version does not have. The PGI company whose compiler supports the OpenACC standard is working on making it automatic. Maybe in the future it is much easier to use OpenACC to implement on the heterogeneous systems. MPI is a message passing library standard based on the consensus of the MPI Forum [15]. MPI is used to port our research CFD code to multiple CPUs. Combining MPI with OpenACC can enable us to run our code on the CPU-GPU heterogeneous system and port the code to different platforms readily.

4.2 Description of the CFD Code: SENSEI

SENSEI (Structured, Euler/Navier-Stokes Explicit-Implicit Solver) is our in-house 2D/3D flow solver originally developed by Derlaga et al. [16], and later extended to a turbulence modeling code base through an object-oriented programming manner by Jackson et al. [17].

SENSEI is written in modern Fortran and is a multi-block finite volume CFD code. The governing equations of a general Navier-Stokes CFD problem are given in Eq. 4.1.

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{Q}\mathrm{d}\Omega + \oint_{\partial\Omega}(\vec{F_{i,n}} - \vec{F_{\nu,n}})\mathrm{d}s = \int_{\Omega}\vec{S}\mathrm{d}\Omega \tag{4.1}$$

where $\vec{Q}$ is the vector of conversed variable, $\vec{F_{i,n}}$ and $\vec{F_{\nu,n}}$ are the inviscid and viscid flux normal components (the dot product of the 2nd order flux tensor and the unit face normal vector), respectively, given as,

$$\vec{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_t \end{bmatrix}, \vec{F_i} = \begin{bmatrix} \rho V_n \\ \rho u V_n + n_x p \\ \rho v V_n + n_y p \\ \rho w V_n + n_z p \\ \rho h_t V_n \end{bmatrix}, \vec{F_i} = \begin{bmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\Theta_x + n_y\Theta_y + n_z\Theta_z \end{bmatrix} \tag{4.2}$$

$\vec{S}$ is the source term from either body forces, chemistry source terms, or the method of manufactured solutions. ρ is density, u, v, w are the Cartesian velocity components, e_t is the total energy, h_t is the total enthalpy, $V_n = n_x u + n_y v + n_z w$ and the n_i terms are the components of the outward-facing normal unit vector. τ_{ij} are the viscous stress components based on Stokes's hypothesis. Θ_i represents the heat conduction and work from the viscous stresses. In this paper, only the Euler solver of SENSEI is used as a lot of GPU optimizations are finished for it, but not for the viscous flux calculation (the viscous flux calculation and turbulence modeling portion uses a lot of type-bound procedures and overloading, which are not supported well by OpenACC). This simplicity should not basically affect the conclusion of applying heterogeneous computing in a CFD code, as Laminar NS solver or turbulence modeling just requires more computation and communication.

In SENSEI, ghost cells are used for multiple purposes. First, boundary conditions can be enforced in a very straitforward way using ghost cells. There are different kinds of boundaries in SENSEI, such as slip wall, non-slip wall, supersonic/subsonic for inflow/outflow, farfield, etc. Second, from the view of parallel computing, ghost cells on a face of a block can hold data exchanged from its neighbour blocks, which makes parallel implementation easier. Users can readily set the number of ghost cells in SENSEI for different accuracy requirements. In most of our work, second order accuracy is used, and it is achieved using the MUSCL scheme, which calculates the left and right state for the primitive variables on each face of all cells. The stencil size is 13 for 3D Euler cases and 9 for 2D Euler cases. This also shows how much data need to be exchanged when running SENSEI on multiple PUs. Time marching can be accomplished using an explicit M-step Runge-Kutta scheme and an implicit time stepping scheme. In this paper, only the explicit M-step Runge-Kutta scheme is used as the implicit scheme uses a completely objected-oriented way of programming which includes overloading of type-bound procedures. Using a lot of objected-oriented programming may deteriorate the GPU performance greatly as it may require the GPU to jump from an address to a different address (trampoline), which should be avoided when programming on GPUs.

Provisions are made to approximate the inviscid flux with a number of different inviscid flux functions. Roe's flux difference splitting [18], Steger-Warming flux vector splitting [19], and Van Leer's flux vector splitting [20] are available. The viscous flux is calculated using a Green's theorem approach. The viscous fluxes requires more cells to be added to the inviscid stencil, but it is not considered in this paper as all the cases in this paper are inviscid. For more details on the implementation, see Derlag et at. [16]

4.3 Domain Decomposition and Aggregation Methods

A processor-cluster based domain decomposition method is used in this paper to deal with situations in which the number of processors (np) is significantly greater (>2 times) than or equal to the number of subgrids (nsg). Note that a subgrid is a parent block before domain decomposition. A subgrid can be decomposed into multiple blocks after domain decomposition. Thus, the word ”subgrid” is different from ”block” in this paper to avoid any misunderstanding. There are several advantages and disadvantages for this decomposition strategy. In terms of the advantages, firstly it is an ”on the fly” approach, which is convenient and requires no manual operation or preprocessing of the domain decomposition. Second, it is very robust in that it can handle any cases if np is greater than or equal to nsg. Third, the communication overhead is small and load imbalance is good as each processor holds only one block in a given subgrid. Finally, Processors in subgrids have the structured topology, which has less intra-subgrid communication overhead and makes communication easier if using MPI topology functions. In this way, the overhead of decomposing and linking boundaries can be reduced as this work can be done partly in parallel. However there are some disadvantages. First, this processor-cluster based approach may have load imbalance issue if np is not obviously greater than nsg, although the communication overhead still remains small. Second, it is a static method which requires enough inputs related to architectures, applications, etc. to be effective. For this issue, we devised a grid-cluster based aggregation method which will be introduced later.

For the CPU-GPU heterogeneous system, it is not good to use the processor-cluster domain decomposition method directly for multi-subgrid cases, potentially due to serious load imbalancing issues. However, 1D domain decomposition is feasible for single subgrid cases as we can use a variable to control the load ratio between the GPU and CPU. Moreover, the process of updating connected boundaries and the communication pattern are simple. We can decompose the grid only in one dimension of a grid and thus get many chunks of blocks in that dimension. Considering that the CPU and GPU have different compute capability, we assign more cells to GPU, by defining a variable *ratio_gc* to represent the disparity of the compute capabilities for one GPU and one CPU. The definition of *ratio_gc* is defined as a single GPU workload over a single CPU workload in a CPU+GPU heterogeneous computing case. This variable is used to make CPUs/GPUs run at approximately the same pace and thus reduce the synchronization overhead. The variable *ratio_gc* can be defined by calculating the speedup of one GPU over one CPU in the serial mode, although it is different from the actual speedup of the GPU over the CPU in the heterogeneous mode, as there are other factors such as synchronization, communication, etc. which are not considered beforehand.

Although the 1D domain decomposition is very convenient to use, we want to use a general 3D domain decomposition for general multi-subgrid cases to improve the scope and scalability of our code. A grid-cluster based domain aggregation method is also implemented in SENSEI to deal with situation in which the number of processors (np) is less than or equal to the number of subgrids (nsg), or the number of decomposed blocks generated using the processor-cluster based domain decomposition. The grid-cluster based domain aggregation method can only be used after applying the processor-cluster based domain decomposition, because we are applying a partially parallel domain decomposition method as mentioned earlier, which means that some domain decomposition tasks are done on different processors instead of only on the ROOT processor. The grid-cluster based domain aggregation method is fully parallel. The aggregation step is not necessary if there is no serious load balancing issue at the decomposition step. The aim of the aggregation step is to provide a way to aggregate blocks to balance loads on different processors, such as building blocks. Users can assign a certain number of blocks from the domain decomposition step to a processor and thus keep the load approximately balanced for general multi-grid cases. For the CPU-GPU heterogeneous computing, obviously the GPU should take more blocks or larger blocks in order to run at the same pace as the CPU does. One issue of applying the aggregation step is that the GPU still needs to exchange data with its host (the CPU) if there are connections among the blocks residing on the same processor. The extra communication overhead may make the heterogeneous computing slower than not considering the overhead (however the communication overhead exists).

4.4 Results

4.4.1 Hardware Configuration

Thermisto Thermisto is a workstation in our research lab. It has two Nvidia Tesla C2075 GPUs and 32 CPU cores. Every GPU has 14 MPs, each with 32 CUDA cores. The peak bandwidth is 144GB/s and the peak double precision performance is 515 GFLOPS. The compilers used on Thermisto are PGI 16.5 and OpenMPI/1.10.0. An compiler optimization of -O4 is used.

4.4.2 Performance Metrics

Two important metrics used in this paper are parallel speedup and efficiency. Speedup denotes how much faster the parallel version is compared with the serial version of the code, while efficiency represents how efficiently the processors are used. They are defined as follows,

$$speedup = \frac{t_{serial}}{t_{parallel}} \tag{4.3}$$

$$efficiency = \frac{speedup}{np} \tag{4.4}$$

where t_{serial} and $t_{parallel}$ are the serial execution time and parallel execution time, respectively, and np is the number of processors (CPUs or GPUs).

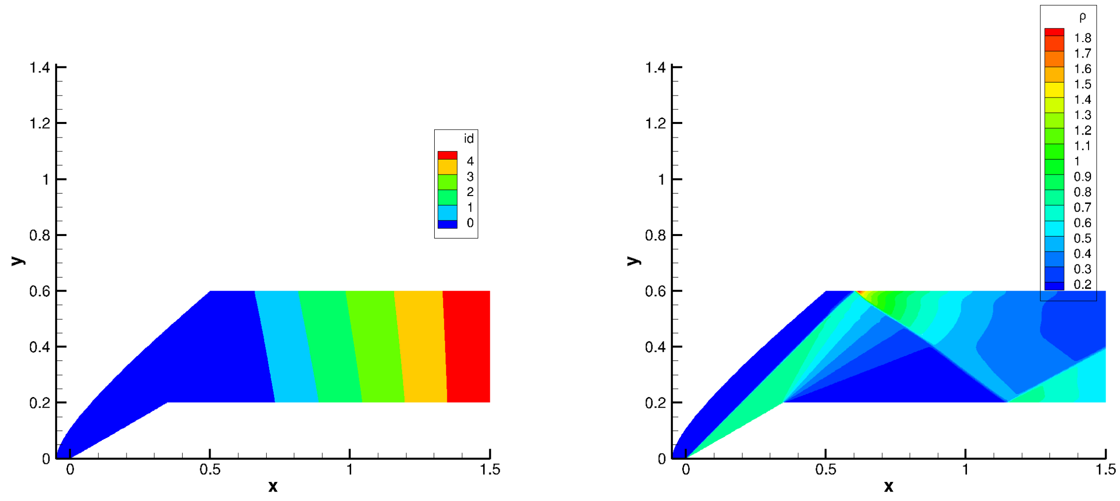
4.4.3 Supersonic Inlet Case

The first case tested in this paper is a simplified 2D 30 degree supersonic inlet, which has only one subgrid. The inflow conditions are given in Table 4.1. There are two grid sizes for this inlet case tested including a 417x129 grid and a 833x257 grid, having a total number of 53,248 and 106,496 cells, respectively. Different problem sizes are chosen to investigate the effect of problem size on the GPU performance in both the pure GPU and the heterogeneous CPU-GPU computing mode. The problem sizes tested are small so that it does not take long to get the converged solution using an explicit scheme on single GPU. A 1D decomposition for the inlet case is shown in Fig. 4.1a if using 1GPU+5CPUs with the *ratio_gc* being 5. The density contour obtained using this decomposition is shown in Fig. 4.1b. The solution has been checked with the serial (single CPU) solution and the relative errors for all primitive variables based on the reference values is within round-off error range (10^{-14}).

Table 4.1: Inlet case inflow boundary conditions

Mach number	4.0
Pressure	12270 Pa
Temperature	217 K

Fig. 4.2a shows the scaling performance of using pure CPUs and pure GPUs on Thermisto. A speedup of 11.75 (based on the single CPU performance) with a parallel efficiency of 73.44% can be obtained using 16 CPUs for this case. The efficiency drop when adding more processors is due to more communication overhead. For this case, the efficiency when using

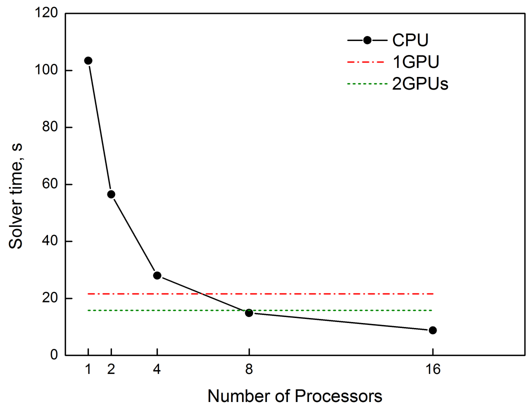


(a) 1D Decomposition (b) Pressure Coefficient

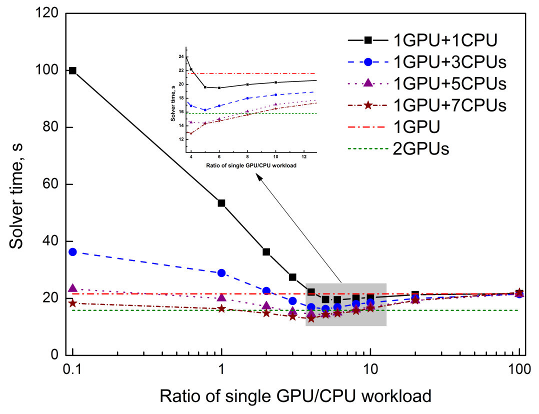
Figure 4.1: Inlet case using 1GPU+5CPUs

8 CPUs is about 86.74% which is fairly good. A single GPU has a speedup of 4.79 over a single CPU and two GPUs is about 6.52 (with an efficiency of 68.35%) as fast as the single CPU performance. One reason of why the efficiency drops faster on the GPU than the CPU is that the GPU needs more work to be saturated, otherwise the communication dominates the whole runtime. When using 2 GPUs, the parallel efficiency is 68.35% for the inlet case.

From Fig. 4.2b, it can be seen that the CPU-GPU heterogeneous computing performance is better than the pure CPU or the pure GPU performance when given a good task ratio between GPU and CPU. All the curves with symbols shown use 1 GPU plus a certain number of CPUs (1, 3, 5 or 7). The best workload ratio *ratio_gc* is between 4 and 5, which is consistent with single GPU over single CPU speedup shown in Fig. 4.2a. Also, using more CPUs gives better performance as the workload on each processor is reduced. Therefore, we can conclude that the performance is better on the heterogeneous system than that on pure GPU or pure CPU systems. Moreover, the effect of the single GPU/CPU workload ratio *ratio_gc* is more obvious when using a higher number of CPUs, because when the task ratio is greater than the optimal value the performance becomes flatter and better for the case of using only one CPU. It is interesting to see that all the four curves reach an asymptotic solver time of about 21.6 s if the ratio *ratio_gc* is large enough, which is the single GPU solver time shown in Fig. 4.2a. The reason is that if we use a very large *ratio_gc*, which means GPU takes a large portion of the total work, the solver time is dominated by the execution on GPU, although there is some communication overhead due to more ranks. Similarly, CPUs become dominant if *ratio_gc* is very small. Also, both 1GPU+5CPUs and 1GPU+7CPUs can outperform 2GPUs for this small size problem.



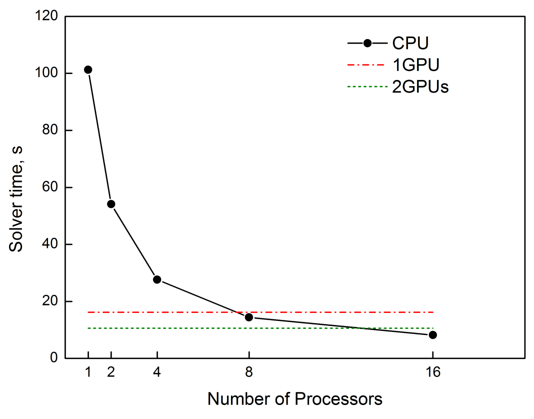
(a) Pure CPU or GPU performance



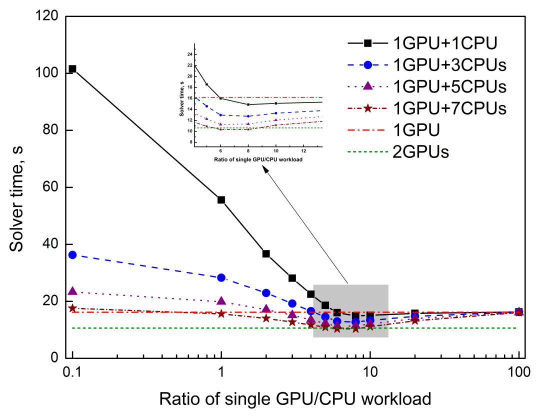
(b) CPU-GPU heterogeneous performance

Figure 4.2: Grid size: 417x129 (on Thermisto)

Fig. 4.3a shows the pure CPU and GPU performance of the 833x257 grid case on Thermisto. Since the task size is 4 times larger than the earlier grid, the larger problem should take more iterations. The efficiencies of the pure CPU and pure GPU are higher than the 417x129 case. For example, the efficiency is 77.2% using on 16 CPUs and 76.2% on 2 GPUs, which indicates that saturating the GPU with enough work is important to the efficiency. Using 2 GPUs is faster than using 8 CPUs for this case. Similarly, the CPU-GPU heterogeneous performance is better than that on the coarser grid, which is given in Fig. 4.3b. The bottoms of the curves with symbols (1GPU plus a certain number of CPUs) inf Fig. 4.3b become flatter than the 417x129 case, which means that the optimal ratio *ratio_gc* range is wider. Also, 1GPU+7CPUs can surpass the 2GPU performance but not for 1GPU+5CPUs.



(a) Pure CPU or GPU performance



(b) CPU-GPU heterogeneous performance

Figure 4.3: Grid size: 833x257 (on Thermisto)

Table 4.2 shows the effect of using 2GPUs in the CPU-GPU heterogeneous mode for the 833x257 grid case. Recall that the single GPU and 2GPUs take 16.2 s and 10.63 s to finish, respectively. Obviously using 2GPUs plus enough CPUs outperform the pure GPU performance. The 2GPU+10CPUs is about 16.6% faster than the 2GPU performance.

Table 4.2: The performance comparison between using 1GPU and 2GPUs for the inlet case in the CPU-GPU heterogeneous mode

ratio_gc	1GPU+5CPUs	1GPU+7CPUs	2GPUs+2CPUs	2GPUs+6CPUs	2GPUs+10CPUs
4	13.5 s	11.75 s	12.55 s	10.14 s	9.15 s
5	12.27 s	10.92 s	11.47 s	10.05 s	9.12 s

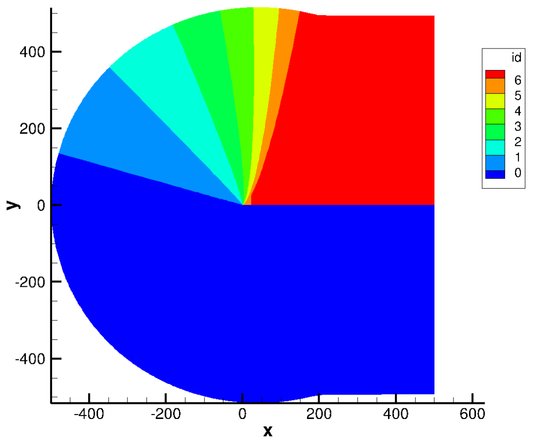
4.4.4 NACA 0012 Airfoil

The second case tested in this paper is a subsonic flow ($M_\infty = 0.25$) past a NACA 0012 airfoil, at an angle of attack of 5 degrees. The farfield conditions are given in Table. 4.3. The case has a subgrid with a size of 897x257 nodes, or a total number of 229,376 cells. The grid for this NACA 0012 airfoil in this paper is a C-type grid, which means the grid connects to itself at the same face. Thus, we are interested in knowing whether the heterogeneous implementation in SENSEI can be used to accelerate such a case having connected boundaries.

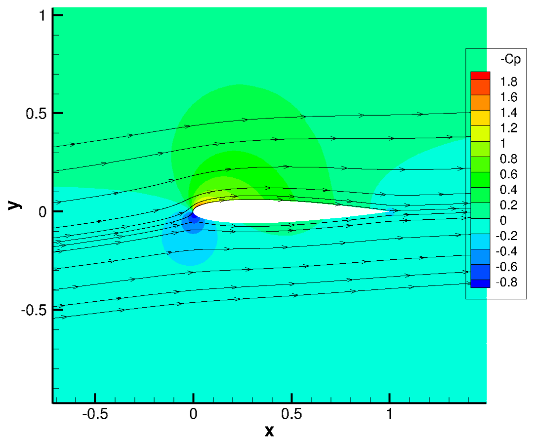
Table 4.3: NACA 0012 airfoil farfield boundary conditions

Mach number	0.25
Static pressure	84307 Pa
Temperature	300 K
Angle of attack, α	5 degrees

Fig. 4.4a gives the 1D domain decomposition (using 1GPU+7CPUs) for heterogeneous CPU-GPU computing of the NACA 0012 case. It is clearly seen that the GPU (with an id of 0) takes more cells (about 8 times) than a single CPU, due to its higher compute capability. Fig. 4.4b shows the steady state pressure coefficient contour calculated by 1GPU+7CPUs near the airfoil under the boundary conditions mentioned above. The solution has been checked with the serial (single CPU) solution and the relative errors for all primitive variables based on the reference values is within round-off error range (10^{-14}).



(a) 1D Decomposition



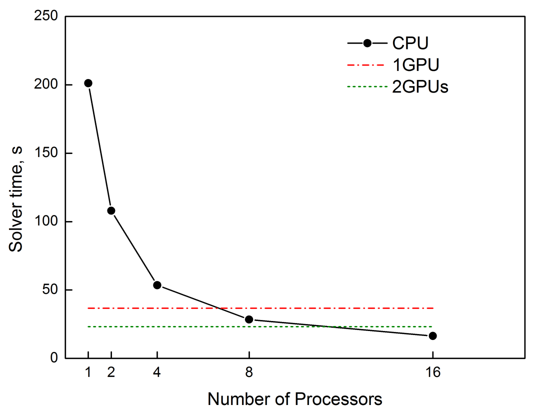
(b) Pressure Coefficient

Figure 4.4: NACA 0012 case using 1GPU+7CPUs

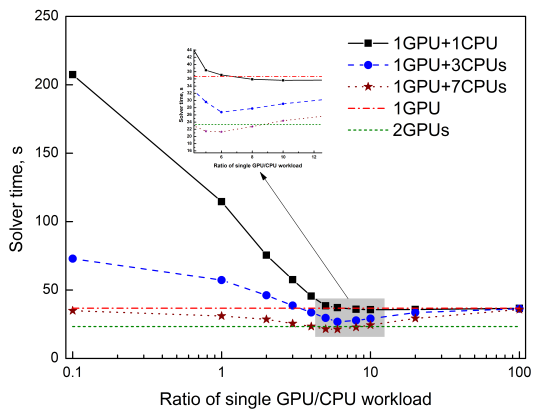
Fig. 4.5a shows the scaling performance of using multiple CPUs and up to 2 GPUs for the NACA 0012 case on Thermisto. A speedup of 12.27 with a parallel efficiency of 76.68% can be obtained using 16 CPUs, which is slightly lower than the inlet 833x257 grid case speedup. Single GPU and 2GPUs have a speedup of 5.48 and 8.67 over single CPU, respectively. The speedup and efficiency are higher than the inlet 833x257 grid case, as this airfoil case has a connected boundary (although to itself). Even using a single processor, there is some overhead associated with this connected boundary enforcement, so that decomposing this grid and running the case in parallel brings less extra overhead compared with the inlet case. The airfoil case is more general than the single subgrid inlet case as it contains a connected boundary, which is more often seen in a structured grid CFD code.

Similar to Fig. 4.3b, Fig. 4.5b shows the change of solver time as a function of *ratio_gc* of using the CPU-GPU heterogeneous computing on the NACA 0012 airfoil case with an connected boundary. When *ratio_gc* is small, the performance is very close to the pure CPU performance. If *ratio_gc* is increased to about 6 or 7, the CPU-GPU heterogeneous performance using different number of CPUs reaches its maximum. The optimal *ratio_gc* is greater than the single GPU over single CPU speedup, i.e., 5.48. Actually it is very difficult to get the optimal *ratio_gc* very accurately as the solver time changes slowly near the optimal regions. Also, it is good to have a range of optimal *ratio_gc* because it is easy for us to obtain the approximately optimal performance (there is no need to get the exactly best performance in this paper, otherwise requiring some manual tuning). For this airfoil case, an optimal *ratio_gc* should be around 6. If *ratio_gc* is greater than 8, the performance is slightly weakened as the GPU takes more work than the optimal, and CPUs may be idling during the parallel execution.

Table 4.4 shows the effect of using 2GPUs in the CPU-GPU heterogeneous mode for the airfoil case. Recall that the single GPU and 2GPUs take 36.7 s and 23.3 s to finish, respectively. The 2GPU+10CPUs (at *ratio_gc* = 5) is about 29.8% faster than the 2GPU performance. This also indicates that the CPU-GPU heterogeneous computing is faster than the pure GPU performance given a good *ratio_gc*.



(a) Pure CPU or GPU performance



(b) CPU-GPU heterogeneous performance

Figure 4.5: Grid size: 897x257 (on Thermisto)

Table 4.4: The performance comparison between using 1GPU and 2GPUs for the NACA 0012 airfoil case in the CPU-GPU heterogeneous mode

ratio_gc	1GPU+7CPUs	2GPUs+2CPUs	2GPUs+6CPUs	2GPUs+10CPUs
4	23.35	24.53	19.66	18.12
5	21.46	21.89	18.83	17.95
6	21.28	22.82	19.85	19.11

4.4.5 ONERA M6 Wing

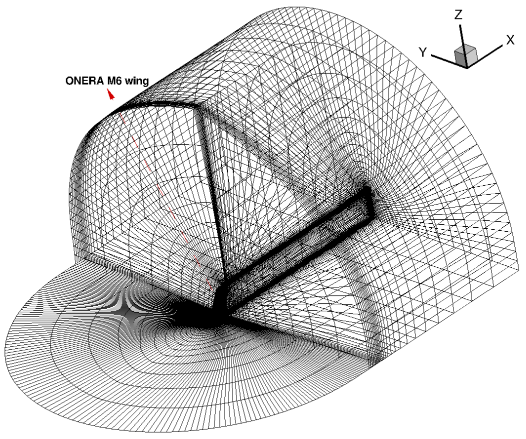
The third case tested in this paper is a flow over a transonic wing, at an angle of attack of 3.06 degrees. The farfield boundary conditions are given in Table. 4.5.

Table 4.5: ONERA M6 wing farfield boundary conditions

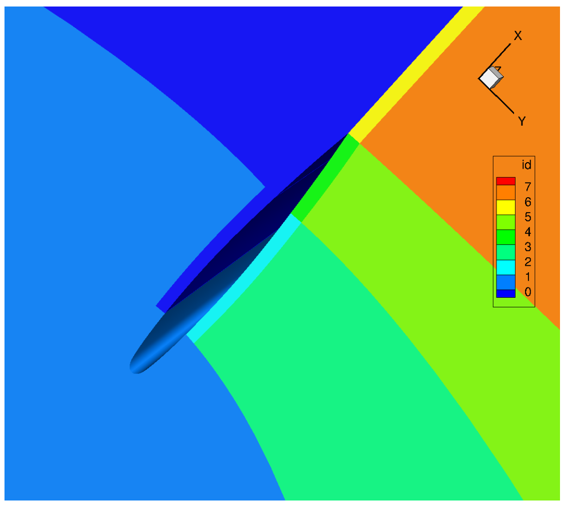
Mach number	0.8395
Temperature	255.556 K
Pressure	315979.763 Pa
Angle of attack, α	3.06 degrees

Fig. 4.6a shows the grid of the ONERA M6 wing case. The grid has 4 subgrids (having 25x49x33, 73x49x33, 73x49x33 and 25x49x33 nodes, respectively), or equivalently a total number of 294,912 cells. Fig. 4.6b shows the 3D domain decomposition (using 2GPUs+6CPUs) of the ONERA M6 wing case for the CPU-GPU heterogeneous computing. The two GPUs

(with the id of 0 and 1) have 5 times more cells than a single CPU (with the id from 2 to 7).



(a) ONERA M6 wing grid (with 2 subgrids not shown)



(b) 3D Domain Decomposition

Figure 4.6: ONERA M6 wing case using 2GPUs+6CPUs

Fig. 4.7 shows the converged surface pressure coefficient contour of the ONERA M6 wing using 16 CPUs on Termisto. Fig. 4.8a shows the scaling performance of multiple CPUs and up to 2 GPUs of the ONERA wing case. Curves labeled with "_16blocks" mean that the performance is for the decomposed blocks, that is, the original 4 subgrids are decomposed into 16 blocks with an equal size. The reason for doing so is that we want to test the heterogeneous implementation on a case with more grid blocks. It is noted that the performance drops significantly for the pure GPU after domain decomposition, as there is more communication overhead. After combining the domain decomposition step and domain aggregation step, we can balance loads well on all the processors for this case, although bringing some communication overhead. Before domain decomposition, the speedup and parallel efficiency of using 16 CPUs (based on the single CPU performance) are 10.63 and 66.45%, respectively. The single GPU and 2GPUs have a speedup of 4.51 and 6.94 over single CPU, respectively. The speedup drop compared with the earlier cases (the inlet and NACA 0012 case) is mainly caused by connected boundary enforcement (communication) between processors. After domain decomposition (now the total number of blocks is 16), the single GPU and 2GPUs are only 3.25 and 4.76 times faster than the single CPU performance. Having connected boundaries is general for multi-block cases so that the speedup here is closer to the practical applications. Boundary enforcement is intrinsically irregular computation which requires special treatments and it also requires inter-processor and host-to-device communication. This case has multiple connections on each processor so it is a good general case to test the performance of the CPU-GPU heterogeneous computing.

Fig. 4.8b shows the performance comparison of using different numbers of CPUs and GPUs. The heterogeneous performance is better than the pure GPU performance if using a small number of CPUs. However, the performance is out of our expectation if there are a higher number of ranks, which is equal to the total number of CPUs and GPUs. More ranks means more communication and synchronization overhead, which are not considered well although the load is balanced well. Also, the communication overhead may make the performance deteriorate when increasing the ratio $ratio\_gc$ when using 1GPU+7CPUs and 2GPUs+6CPUs. These issues will be investigated in the future.

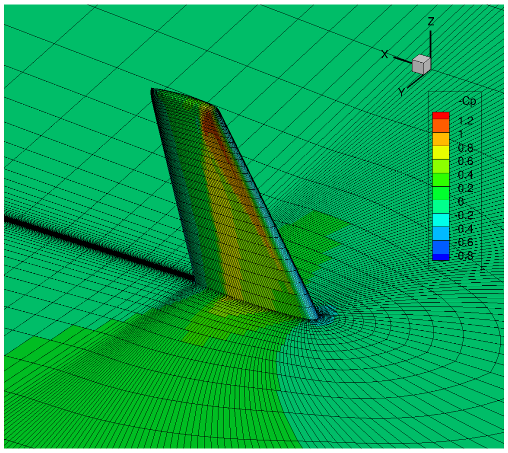
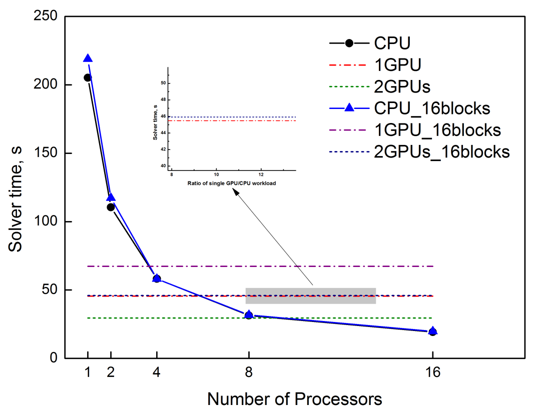
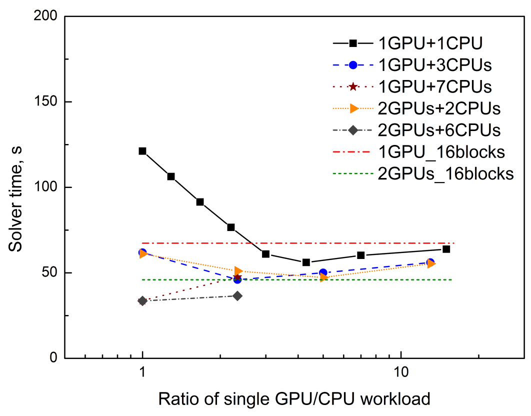


Figure 4.7: Pressure contour of the ONERA M6 wing



(a) Pure CPU/GPU scaling performance (Thermisto)



(b) CPU-GPU heterogeneous performance (Thermisto)

Figure 4.8: The ONERA M6 wing case performance

4.5 Conclusions

From the inlet case and airfoil case presented in this paper, it is obvious that the CPU-GPU heterogeneous computing brings some performance gain if the task assigned to CPUs and GPUs is balanced well. For the inlet case, 1GPU+5CPU has an about 1.3 to 1.5 speedup over the single GPU performance, based on the grid size. This is good as more computational resources are used to maximize the power of parallel computing, since many CPUs may just idle in the homogeneous GPU computing mode. Using 2 GPUs in the heterogeneous mode can generate more speedup than only using 1 GPU. Similar conclusions can be drawn from the airfoil case. 1GPU+7CPU is as 1.52 times faster than the single GPU performance. 2 GPUs + 10 CPUs (at $ratio\_gc = 5$) is about 30% faster than the 2 GPU performance. For the 16-block (the original 4 subgrids are decomposed into 16 blocks) ONERA M6 wing case, the heterogeneous performance is still better than the pure GPU performance, but not very much if using more ranks. However, more ranks also means more communication and synchronization overhead, which are not considered well enough in this paper as the focus in this paper is load balancing. The effect of $ratio\_gc$ is also out of our expectation if more CPUs are added in the heterogeneous mode, possibly caused by the communication overhead.

Different domain decompositions (1D & 3D) are applied in the paper. We developed a processor-cluster based decomposition step which can handle the situations in which the number of processors is obviously larger than the number of subgrids. Also, we implemented a grid-cluster based aggregation step so that we can deal with the situations in which the number of processors is less than the number of subgrids. After combining the decomposition and aggregation step, a processor may hold several blocks so that the load can be balanced on all the processors. However, the inter-block communication overhead on a GPU would still be expensive as the GPU needs to exchange the data frequently with its host (the CPU).

In terms of what can be improved for applying the CPU-GPU heterogeneous computing in the CFD area, we think the communication subsystem and the memory bandwidth between the CPU-CPU, CPU-GPU, and even GPU-GPU if applicable, should be considered carefully when designing a heterogeneous system for various scenarios (e.g., single or multi-subgrid cases), in order to reduce the communication and synchronization overhead. The reason is the communication is usually an important bottleneck for memory-bounded problems such as in CFD. Users can balance load easily through using a variable such as *ratio_gc* in this paper, but accounting for communication overhead is more difficult.

4.6 Future Work

Test cases with more subgrids will be tested in the future. At present, only strong scaling performance results are given. We are also interested in the weak scaling performance as running a very large problem on many processors is more meaningful.

Also, more advanced GPUs should be used. At present, we just test our implementation on a workstation with a total number of 32 CPU cores (with hyper-threading) and 2 GPUs. We are interested in testing the heterogeneous implementation on a cluster with more cores and GPUs, and with different architectures. The communication systems on different systems are different, so investigating the heterogeneous computing implementation on different platforms should be meaningful.

Finally, we will investigate ways to reduce the communication overhead when adding more processors. Most of our applications are communication bounded so that reducing communication overhead and synchronization is very important for the overall performance.

Bibliography

[1] Green500 list for november 2019, 2019. (last accessed on 12/02/19).

[2] TOP500.org. Top 10 sites for november 2019, 2019. (last accessed on 12/02/19).

[3] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[4] Luke Domanski, Tomasz Bednarz, Tim E Gureyev, Lawrence Murray, Emma Huang, and John A Taylor. Applications of heterogeneous computing in computational and simulation science. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 382–389. IEEE, 2011.

[5] L Domanski, T Bednarz, P Vallotton, and J Taylor. Heterogeneous parallel 3d image deconvolution on a cluster of gpus and cpus. In *19th Int'l Congress on Modelling and Simulation, Perth, Australia,[Online, cited Aug 1, 2013] http://mssanz. org. au/modsim2011 A*, volume 8, 2013.

[6] Xavier Alvarez, Andrey Gorobets, and F Xavier Trias. Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers. In *7th European Conference on Computational Fluid Dynamics*, 2018.

[7] Jianqi Lai, Zhengyu Tian, Hua Li, and Sha Pan. A cfd heterogeneous parallel solver based on collaborating cpu and gpu. In *IOP Conference Series: Materials Science and Engineering*, volume 326, page 012012. IOP Publishing, 2018.

[8] Yong-Xian Wang, Li-Lun Zhang, Wei Liu, Xing-Hua Cheng, Yu Zhuang, and Anthony T Chronopoulos. Performance optimizations for scalable cfd applications on hybrid cpu+ mic heterogeneous computing system with millions of cores. *Computers & Fluids*, 2018.

[9] Isaac Liu, Edward A Lee, Matthew Viele, Guoqiang Wang, and Hugo Andrade. A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 125–132. IEEE, 2012.

[10] Dragan D Nikolić. Parallelisation of equation-based simulation programs on heterogeneous computing systems. *PeerJ Computer Science*, 4:e160, 2018.

[11] Thomas Kovac, Tom Haber, Frank Van Reeth, and Niel Hens. Heterogeneous computing for epidemiological model fitting and simulation. *BMC bioinformatics*, 19(1):101, 2018.

[12] Xiaocheng Liu, Ziming Zhong, and Kai Xu. A hybrid solution method for cfd applications on gpu-accelerated hybrid hpc platforms. *Future Generation Computer Systems*, 56:759–765, 2016.

[13] NVIDIA. NVIDIA GPUDirect, 2019.

[14] Openacccourse, 2019. (last accessed on 12/02/19).

[15] Blaise Barney. Message Passing Interface (MPI), 2019.

[16] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[17] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model implementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[18] Philip L Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[19] Joseph L Steger and RF Warming. Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods. *Journal of computational physics*, 40(2):263–293, 1981.

[20] Bram Van Leer. Flux-vector splitting for the euler equation. In *Upwind and High-Resolution Schemes*, pages 80–89. Springer, 1997.

Chapter 5

Machine Learning based Autotuning of a GPU-accelerated Computational Fluid Dynamics Code

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. The machine learning based autotuning work was implemented by the first author. All the results were collected and analyzed by the first author.

- Christopher J. Roy (second author): The second author provided valuable feedback for this study and comments for this manuscript.

Abstract

A machine learning based autotuning technique is used to autotune fourteen parameters related to GPU kernel scheduling, including the number of thread blocks and threads in a block. Both the independent training for single type of GPU and combined training for multiple types of GPU are performed for a single fluid dynamics problem (accelerated by one GPU) on the C2075, P100 and V100 GPU. The training and the testing results indicate that using an artificial neural network has the potential to autotune a large number of parameters, while requiring a very small fraction of samples in a large search space.

Keywords: OpenACC, Machine Learning, Autotuning

5.1 Introduction

General purpose graphic processing unit (GPGPU) [1] has been gaining interests in the area of scientific computing due to its higher compute capability and higher memory throughput compared to central processing unit (CPU). CPUs are usually used as hosts which deal with general settings and controls, while GPGPUs are used as accelerator devices which run intensive computations. Since the GPU has thousands of lighweight cores so the computation can run faster and in parallel on the device. After the device finishes the computations, the device moves the solution back to the the host. Therefore, data movements exist between the host and device as the host and the device have discrete memories. The host and device can be connected through the PCI-E or NVLink [2].

GPGPU has multi-level parallelisms including block level and thread level, which can be seen in Fig. 5.1. It should be noted that different GPU parallel standards such as CUDA [3], OpenCL [4] or OpenACC [5] has different names for the scheduling structures but they are basically similar. In order to obtain high performance across different platforms or for other different environments such as different problem sizes, numerical schemes, etc., users may need to tune their GPU-accelerated code frequently. A program may have more than 10 or even 100 kernels, with each kernel having multiple tuning parameters, and with each parameter having a possible range, so the whole search space can be extremely large. It can be difficult for domain scientists to tune the parameters properly as they may not have such an expertise. In addition, even though a compiler may set the tuning parameters by default but the default setting is usually not the optimal configuration.

In this work, OpenACC is used for a computational fluid dynamics code acceleration on the GPU. OpenACC is a library specification for GPU programming. It was initially created to extend OpenMP to support GPUs. Since the realization of OpenMP 4.0 [6] or higher has

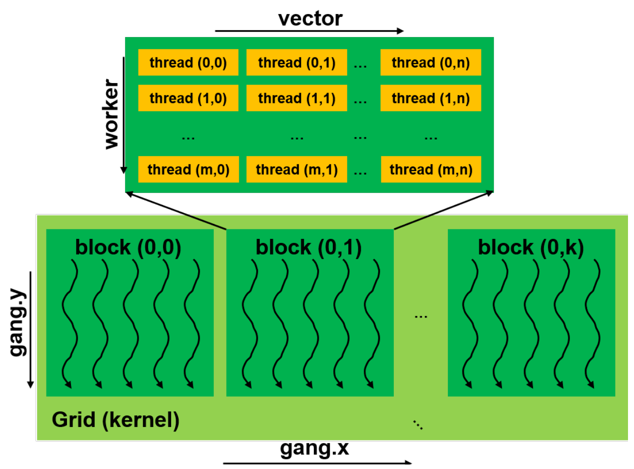


Figure 5.1: Multilevel parallelisms for GPU.

been too slow, the development of OpenACC has become quite independent. OpenACC can be used to simplify the programming on heterogeneous CPU/GPU computing systems, and it is an important reason why OpenACC is used in this work.

Using OpenACC, there are many parameters which need to be tuned to obtain good performance, such as gang size and thread length. On one hand, enough threads and gangs need to be used to saturate the GPGPU with enough work so that the utilization and concurrency is higher. On the other hand, since GPU streaming multiprocessor registers are shared by all the threads, the number of threads running concurrently cannot be unlimited large. Fig. 5.2 shows a two-parameter manual tuning of the thread-block size for a lid-driven cavity code. The compiler can decide how to map the loop iterations to different levels of parallelism on the device automatically, but the scheduling work can be done better through a well designed autotuning. In addition, a good autotuning method may have the potential to improve the performance of a code on next generation GPGPUs. Also, the performance tuning greatly depends on the application. For example, A high latency problem may require higher occupation to hide the latency and also to increase the concurrent accesses, and a low latency problem may not have the same requirement. The compiler is very likely to choose very similar settings for quite different problems, which is not ideal.

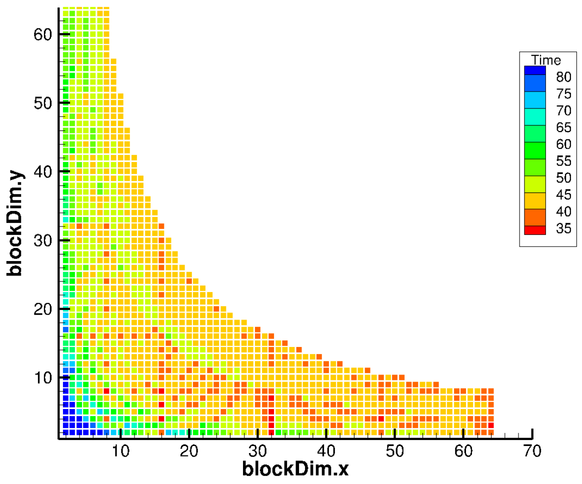


Figure 5.2: Two-parameter manual tuning for a lid-driven cavity code

Artificial neural network may be a promising method for autotuning. An artificial neural network has a number of layers of neurons, with each neuron to be a set of arithmetic operations. The arithmetic operations mainly contain a dot product of weights and inputs. Activation functions are used to add non-linearity to each layer. The output of each layer is propagated to the next layer, which is called forward propagation. The loss function is calculated by comparing the predicted data and the ground truth. Usually a penalty function is added to the loss function to make trained model more generalized. Based on the jacobians of the loss with respect to the weights for each layer, these weights can be updated using different types of optimizer schemes, which is called backward propagation. The whole procedures can be seen in Fig. 5.3.

In this work, tuning parameters such as gang sizes and vector lengths for different kernels are used as features. The program solver runtime of our GPU-accelerated computational fluid dynamics code is used as the target. Since the execution time is continuous, this problem is supervised regression learning. A moderate number of samples are generated by running the

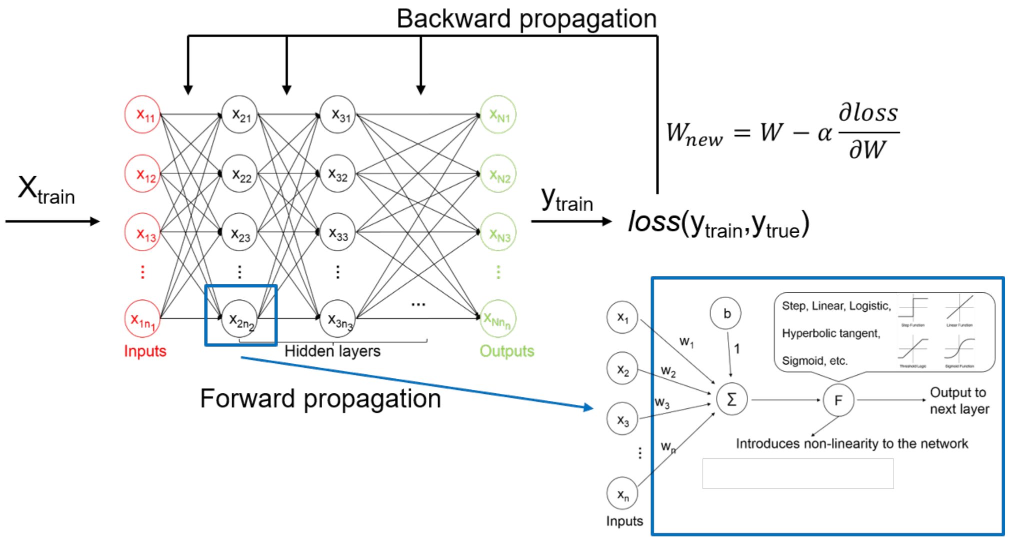


Figure 5.3: Artificial Neural Network

computational fluid dynamics code. After the input layer, there are several hidden layers, each with a number of neurons. ReLU is used as the activation function for each layer. The number of layers, number of neurons for each layer, learning rated, etc. are hyperparameters which need to be tuned to obtain a well trained model. Finally the model is evaluated through testing on some unseen data.

5.2 Related Work

There are different approaches in dealing with the autotuning of a code. Pickering et al. [7] exhaustively searched the entire space of 2D thread-block dimensions for a GPU-accelerated code and they found that the compiler default was non-optimal. Manual tuning requires a lot of experience and time, and is also error-prone. Jia et al. [8] used a statistic tree-based approach to cluster the data according to their importance, which may require a large number of samples to achieve satisfactory accuracy. Collins et al. [9] applied a machine learning based method embedded with principal components analysis to reduce the search space greatly. However, the accuracy may not be very good due to dimension collapsing. Falch et al. [10] used neural networks to autotune some OpenCL kernel applications. They can obtain good prediction for some benchmark codes. However their autotuning framework was not very robust as the optimal configuration can vary greatly if running for several times. Cui et al. [11] developed an iterative machine learning method looking for potentially better samples in subsequent iterations based on samples from one iteration. However, all the results shown only focus on the GPU thread-block size of one single kernel, which contains only two parameters.

5.3 Data Collection

5.3.1 Platforms

C2075 GPU The peak float precision performance is 1028 GFLOPS, which is used as a feature in the combined training (all the data on different GPUs are combined and trained together). There are other factors affecting the performance of a GPU, including memory bandwidth, number of streaming multiprocessors, etc. This work only uses the theoretical performance as a feature, due to the fact that not too many types of GPUs are available.

P100 GPU The peak float precision performance is 9526 GFLOPS, which is used as a feature in the combined training.

V100 GPU The peak float precision performance is 14130 GFLOPS, which is used as a feature in the combined training, similar to 2075 and P100 GPUs.

5.3.2 The Computational Fluid Dynamics Code Base: SENSEI

SENSEI (Structured, Euler/Navier-Stokes Explicit-Implicit Solver) is our in-house 2D/3D flow solver [12, 13, 14]. SENSEI is written in modern Fortran and is a multi-block finite volume computational fluid dynamics code. SENSEI can be accelerated on multiple GPUs and can achieve a speedup of up to 70× on 16 V100 GPUs over 16 Xeon CPU E5-2680v4 cores [15]. SENSEI has not yet been tuned well due to the fact that there are a lot of kernels inside, each with multiple tuning parameters, including gangs and vectors which can be set as features. The search space constructed by multiple parameters can be very large, and there may be inter-dependency between discrete parameters [16]. A random sampling method will be used to generate the samples in this work. 75% of the samples will be used for training, and the remaining 25% samples which are unseen will be for used for testing.

5.3.3 A Test Case

A test case used in this work is a simplified 2D 30 degree supersonic inlet. the problem size is also fixed to have 3328×1024 cells, for simplicity. The parallel solution using 16 P100 GPUs is shown in Fig. 5.4.

5.3.4 Settings for Neural Network

The Scikit-learn [17] framework is used in this work. Scikit-learn provides a metric R^2 to evaluate the trained model, which is defined in Eq. 5.1. It can be seen that the model is more accurate if R^2 is closer to 1. The adaptive moment estimation (Adam) optimization algorithm is chosen. After conducting a number of tests, the parameters for the neural network are determined and given in Table 5.1.

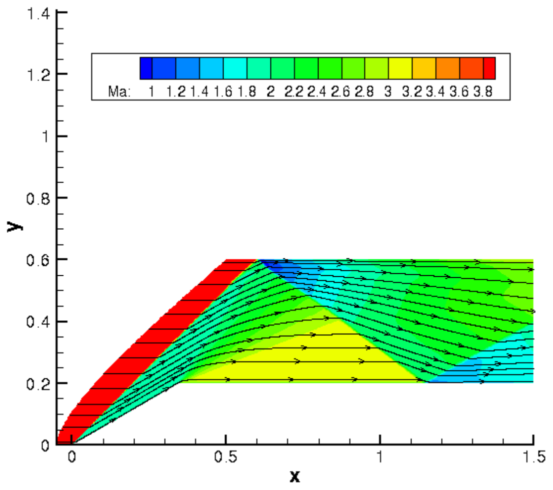


Figure 5.4: The Mach number and streamlines for 2D inlet Euler flow.

$$R^2 = 1 - \frac{\Sigma_{i=1}^{N}(y_{actual,i} - y_{pred,i})^2}{\Sigma_{i=1}^{N}(y_{actual,i} - y_{actual,mean})^2} \tag{5.1}$$

Table 5.1: Parameter settings for the neural network

Parameters	Settings
L2 regularization coefficient, α	0.0001
The 1st moment exponential decay factor, β_1	0.95
The 2nd moment exponential decay factor, β_2	0.90
Activation function solver	Relu
Learning rate	Adaptive
Initial learning rate	0.0009
Maximum epoch number	200
Batch size	200
Tolerance	10^{-6}
Numerical stability factor	10^{-9}

5.3.5 Tuning Parameters

Seven kernels which are the most time consuming in SENSEI are chosen to be tuned. Each kernel can have two tuning parameters including the gang size and vector length. Therefore, a total number of 14 parameters are used as features, which can be seen in Table 5.2. The whole search space has $10^7 \times 12^7 = 3.58 \times 10^{14}$ configurations. In this work, only 7500 samples are used for training.

Table 5.2: Tuning Parameters

Kernels	Range
xi limiter gang	100, 200, ..., 1000
xi limier vector	32, 64, ..., 384
eta limiter gang	100, 200, ..., 1000
eta limiter vector	32, 64, ..., 384
xi flux gang	100, 200, ..., 1000
xi flux vector	32, 64, ..., 384
eta flux gang	100, 200, ..., 1000
eta flux vector	32, 64, ..., 384
source term gang	100, 200, ..., 1000
source term vector	32, 64, ..., 384
right hand side gang	100, 200, ..., 1000
right hand side vector	32, 64, ..., 384
update solution gang	100, 200, ..., 1000
update solution vector	32, 64, ..., 384

5.3.6 Feature Centering and Scaling

Feature centering and scaling (also called mean removal and variance scaling) is usually applied in machine learning based applications. One advantage of applying centering and scaling is to make different features equivalently important so that different weights have the same order of magnitude. Without centering and scaling, the training may perform poorly. Scikit-learn provides different kinds of scaling including standardscaler, minmaxscaler, maxabsscaler, etc. The standardscalar is used in this work as it is one of the most straightforward scalers. Although the standard scaler is designed to be appropriate for Gaussian distribution. However, in a lot of occasions, the distribution of the data is unknown but the standard scaling still helps improve the accuracy.

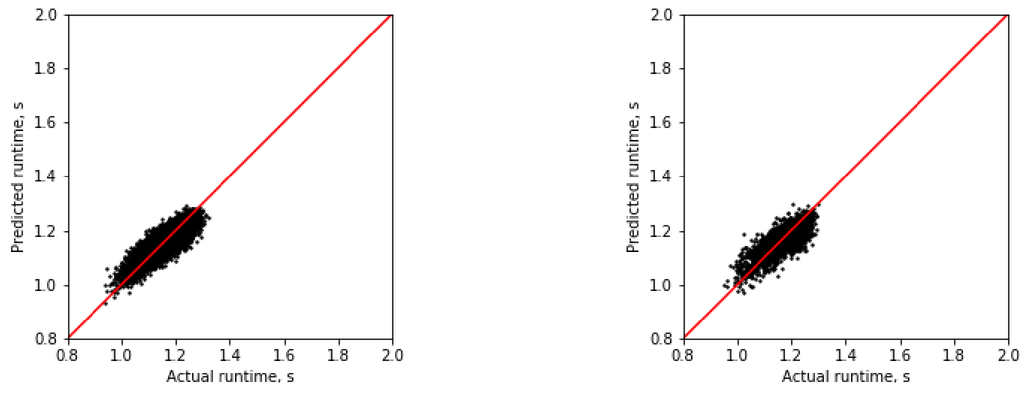
5.4 Results

In this work, the data on each type of GPU can be trained independently, or they can be combined and trained together. The latter may be more promising as including the GPU type as a feature may enable the trained model to predict the performance on unseen GPUs. One drawback of the combined training is that the number of GPU types is still small so the accuracy on unseen GPUs may not be satisfactory. Since the three GPUs have different compute capabilities, the program solver execution time on different GPUs are scaled to approximately the same range. In other words, the inlet case runs for 1 iteration step on the C2075 GPU, 5 iteration steps on the P100 GPU and 15 iteration steps on the V100 GPU. The solver time of all samples is in the range of [0.8 s, 2.0 s]. Considering that there is some time spent in the launch of code and initial and final data movement (which should not be considered since more iteration steps can compensate the cost in real running), 10000 samples actually cost about 24 hours to finish.

5.4.1 Training on Single Platform

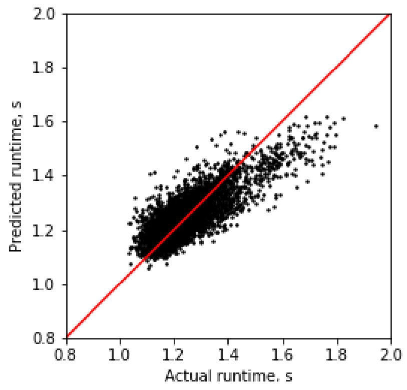
Firstly, training is performed on one single GPU, including C2075, P100 and V100 GPU. The reason why training on single platform first is to see whether the artificial neural network technique can be applied to auto-tune a real application, instead of some simple kernels or benchmarks.

Fig. 5.5, Fig. 5.6 and Fig. 5.7 show the training and testing results on a C2075, V100 and P100 GPU, respectively. Each GPU has 10,000 samples, with 7,500 samples used for training and the remaining 2,500 unseen samples for testing. The horizontal axis denotes the actual runtime and the vertical axis denotes the predicted runtime. The red line with a slope of 1 denotes the perfect prediction for all samples. It can be seen that for all GPUs tested, this neural network works well (especially on slower GPUs), as most data points cluster near the perfect prediction line. In addition, the predicted sample having the lowest runtime is very close to the actual sample having the lowest runtime, which means that using the best predicted sample is a good option. It should be noted that finding the global optimal is not required as long as a sub-optimal does not cost far more time than the global optimal. There are some large prediction errors for configurations having longer actual runtime. The model tends to under-predict the runtime for these cases, which may suggest that the machine learning code has some unknown bias errors inherently due to not being well designed, or the data itself are biased. However, this work cares more about finding configurations having shorter actual runtime correctly.

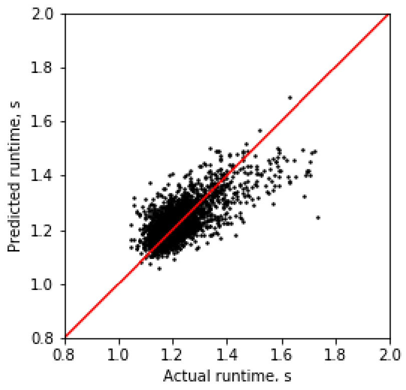


(a) Training (b) Test samples

Figure 5.5: Training and testing on C2075 GPU

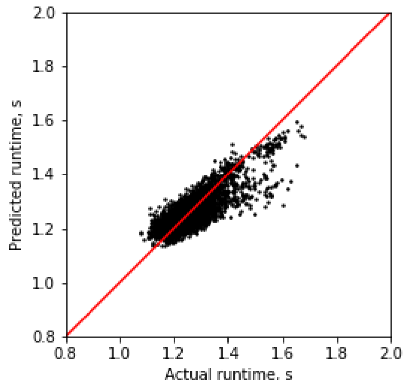


(a) Training

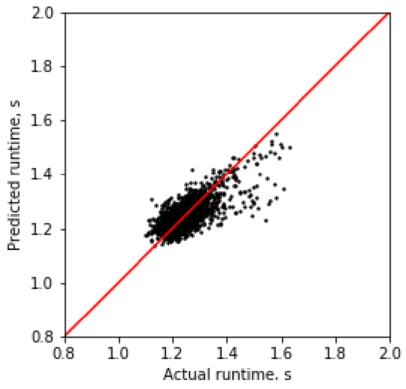


(b) Test samples

Figure 5.6: Training and testing on V100 GPU



(a) Training



(b) Test samples

Figure 5.7: Training and testing on P100 GPU

5.4.2 Combined Training

Now, all the data on different GPUs are combined and trained together. The GPU type is used as a feature to include the impact of different types, especially their compute capability difference. The trained model can be used to predict the performance on unseen GPUs. Different problems, problem sizes, domain decompositions and numerical scheme are not considered in this work.

Fig. 5.8 shows the comparison of loss history using independent training and combined training. It can be seen that the combined training loss drops faster than the independent training in the beginning, possibly due to a better centering and scaling (since more data are used for data preprocessing). Also, it should be noted that the combined training has 22,500 samples for training (each independent GPU has 7,500 samples) so the training may be more stable and accurate. Fig. 5.9 shows the comparison of the R^2 score using independent training and combined training. It is interesting to see that a slower GPU has higher R^2 score than a faster GPU. However, after combining all the GPU data, the scores for the training and testing can be improved.

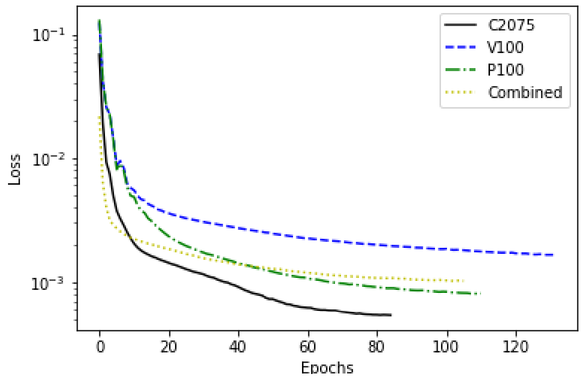


Figure 5.8: Loss history.

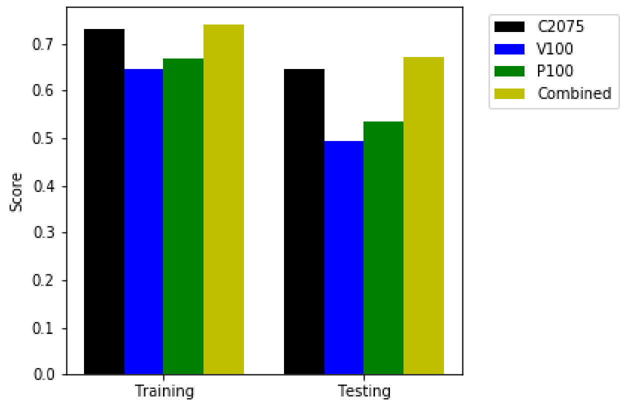
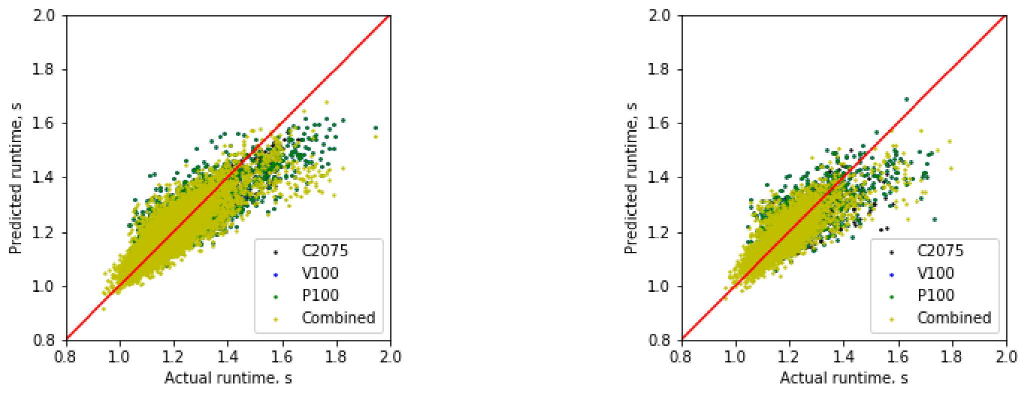


Figure 5.9: R^2 score.

Fig. 5.10 shows the combined training and testing results for the 2D inlet case. After combining all the data points from three different types of GPUs, a good model can still be obtained. This model can be used to predict the performance on unseen GPUs.



(a) Training (b) Test samples

Figure 5.10: Training and testing on combined dataset

5.5 Conclusions

In this work, the approach of artificial neural network is used to auto-tune fourteen GPU kernel scheduling parameters in a GPU-accelerated research computational fluid dynamics code. Both the independent and combined training show the potential of applying the machine learning technique to auto-tune a code. Although there is only one problem and the size is fixed, data on different types of GPU may be utilized to predict the performance on unseen GPUs.

Broader Impact

General programs which require tuning but have a large search space may benefit from this work. Also, compiler development can be designed to be smarter by including a similar machine learning based toolkit on the backend, so that users do not need to manually tune a lot of parameters, which require expertises. To make this research more meaningful and practical, multiple factors including different types of problems, problem sizes, numerical schemes, etc. should be considered in the future. In addition, this work can be more useful if extended to multiple GPUs. Other methods such as reinforcement learning is worthwhile to be tried.

Bibliography

[1] Wen-Mei W Hwu. *GPU computing gems emerald edition.* Elsevier, 2011.

[2] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[3] NVIDIA. CUDA C++ Programming Guide, 2019. (last accessed on 07/24/20).

[4] Khronos OpenCL Working Group. The OpenCL Specification, 2012. (last accessed on 07/24/20).

[5] OpenACC-Standard.org. *The OpenACC Application Programming Interface.* OpenACC-Standard.org, 2018.

[6] Nawrin Sultana, Alexander Calvert, Jeffrey L Overbey, and Galen Arnold. From openacc to openmp 4: toward automatic translation. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, pages 1–8, 2016.

[7] Brent P Pickering, Charles W Jackson, Thomas RW Scogland, Wu-Chun Feng, and

Christopher J Roy. Directive-based GPU programming for computational fluid dynamics. *Computers & Fluids*, 114:242–253, 2015.

[8] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 257–267. IEEE, 2013.

[9] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. Masif: Machine learning guided auto-tuning of parallel skeletons. In *20th Annual International Conference on High Performance Computing*, pages 186–195. IEEE, 2013.

[10] Thomas L Falch and Anne C Elster. Machine learning-based auto-tuning for enhanced performance portability of opencl applications. *Concurrency and Computation: Practice and Experience*, 29(8):e4029, 2017.

[11] Xuewen Cui and Wu-chun Feng. Iterml: Iterative machine learning for intelligent parameter pruning and tuning in graphics processing units. *Journal of Signal Processing Systems*, pages 1–13, 2020.

[12] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[13] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model implementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[14] Weicheng Xue, Hongyu Wang, and Christopher J Roy. Code verification for 3d turbu-

lence modeling in parallel sensei accelerated with mpi. In *AIAA Scitech 2020 Forum*, page 0347, 2020.

[15] Weicheng Xue, Charles W Jackson, and Christoper J Roy. An improved framework of gpu computing for cfd applications on structured grids using openacc. *arXiv preprint arXiv:2012.02925*, 2020.

[16] Wu-Chun Feng. A deep learning approach towards auto tuning cfd codes. Technical report, Virginia Polytechnic Institute And State University Blacksburg United States, 2018.

[17] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

Chapter 6

Code Verification for Turbulence Modeling in Parallel SENSEI Accelerated with MPI

Weicheng Xue[1], Hongyu Wang[2] and Christopher J. Roy[3]

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. The first author implemented half of the Jacobians for the 3D implicit solver, half of the Spalart-Allmaras and $k-\omega$ SST turbulence models in SENSEI. In addition, all the results were collected by the first author.

- Hongyu Wang (second author): The second author contributed equally as the first author to the implementation of the Jacobians for the 3D implicit solver, the Spalart-

[1]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.

[2]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.

[3]Professor, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 330, Virginia Tech, 460 Old Turner St, AIAA Associate Fellow.

Allmaras and $k-\omega$ SST turbulence models in SENSEI. Also, the second author provided a lot of useful advice when implementing the turbulence models in SENSEI.

- Christopher J. Roy (final author): The final author provided valuable feedback for this study and comments for this manuscript. For the turbulence modeling implementation in SENSEI and stability issues, the final author offered a lot of useful insights and guidance.

Abstract

This paper mainly deals with code verification for turbulence modeling in our parallel research CFD code SENSEI. SENSEI is now being accelerated with the use of MPI. Some turbulence modeling verification cases including cross term sinusoidal manufactured solutions and all verification cases from the turbulence modeling resources website are used to justify the proper turbulence modeling implementation of the parallel SENSEI. For the cross term sinusoidal manufactured solutions, the observed order of accuracy is close to two for the SA and $k-\omega$ SST model, in both 2D and 3D. Also, SENSEI matches very well with all the numerical benchmark solutions from both CFL3D and FUN3D for all the quantities of interest including the total lift coefficient, total drag coefficient, pressure drag coefficient and viscous drag coefficient if the flux limiters are turned off.

6.1 Introduction

To model turbulent flows such as flow past an object (a vehicle, building, etc.), jet flow, and atmospheric motion, researchers have proposed methods including Reynolds-Averaged Navier-Stokes (RANS) models, Large Eddy Simulation (LES) and Direct Numerical Simulation (DNS) [1]. LES and DNS are not the focus of this paper as RANS is the cheapest in terms of computational cost but still has a reasonable accuracy for a variety of turbulent flow problems. RANS models can be categorized as zero equation models (Prandtl's mixing length model for an example), one equation models (.eg, Spalart-Allmaras [2]), two equation model ($k-\epsilon$ [3], $k-\omega$ [4], $k-\omega$ SST [5], etc.), etc. Since there is no analytic expression for the Reynolds Stresses which are introduced by ensemble averaging over the Navier-Stokes equations, the Boussinesq approximation is introduced to close the RANS equations. The Boussinesq approximation relates the Reynolds stresses with mean flow velocities through the use of eddy viscosity. Different RANS models solve the eddy viscosity in different ways. The Spalart-Allmaras (SA) model solves eddy viscosity through a working variable, and other models calculate it by solving additional transport equations. The SA model is cheaper in terms of computational cost so it is good for problems with large grid size, and it is often used in flows with just slight separation such as transonic flow past airfoils, etc. The $k-\epsilon$ model which is a two equation model performs well for free shear flows. The $k-\omega$ model (two equation model) is a good option for wall-bounded flows. The $k-\omega$ SST model (also known as the Menter's SST) is aimed at combining the advantages of $k-\epsilon$ and $k-\omega$ model. It is devised in a way so that in the inner region of the boundary layer $k-\omega$ is used, and in the free shear flow $k-\epsilon$ is used.

6.2 The CFD Code Base: SENSEI

6.2.1 Overview of SENSEI

SENSEI (Structured, Euler/Navier-Stokes Explicit-Implicit Solver) is our in-house flow solver originally developed by Derlaga et al. [6]. The original code was written in a very structured programming style, although with many modern Fortran features such as derived types, pointers, PASS attributes, implied shape array, etc. being used. Jackson et al. [7] followed up their work by implementing the negative Spalart-Allmaras turbulence model into SENSEI and doing tests on four 2D verification test cases (2D Zero pressure gradient flat plate case, 2D Coflowing jet case, 2D Bump-in-channel case, and 2D Airfoil near-wake case [8]). They also refactored SENSEI into an object-oriented programming style so that new capabilities including new turbulence models, solvers, and boundary conditions can be easily added due to the improvement of SENSEI's modularity. SENSEI is a multi-block structured finite volume code and is embedded with several flux options. An important reason for why SENSEI uses structured grid is that the quality of grid is usually better using a multi-block structured grid than that provided on an unstructured grid. Also, memory can be used more efficiently, since data are ordered in a structured way located in the memory, which is fit for cache reuse when running SENSEI in parallel. It also saves some memory as no connectivity information needs to be stored for every cell.

Second order spatial accuracy for the inviscid flux is achieved by using MUSCL extrapolation, which uses a thirteen point stencil per cell for a 3D problem and a nine point stencil per cell for a 2D problem. Provisions are made to approximate the inviscid flux with a number of different inviscid flux options. Roe's flux difference splitting [9], Steger-Warming flux vector splitting [10], and Van Leer's flux vector splitting [11] are available. The viscous flux and the turbulence flux are calculated through a central flux scheme using the Green-Gauss theorem. The viscous flux and the turbulence flux require more cells to be added to the inviscid stencil, especially for 3D cases.

Time marching can be accomplished using an explicit M-step Runge-Kutta scheme and a backward Euler implicit time stepping scheme with preconditioning (mainly for steady-state flow problems). Ghost cells are used to transfer interblock boundary data between neighbour blocks explicitly (for both explicit and implicit solvers) and SENSEI requires point matching between neighbour blocks. The default number of ghost cells is 2, but it can be changed accordingly if different order of accuracy is required in the future.

Different kinds of boundary conditions including slip wall, non-slip wall, farfield, pressure inflow, supersonic inflow/outflow, subsonic inflow/outflow, symmetry and interblock boundaries are implemented in SENSEI using an object-oriented programming approach. A derived type is used to store all information of a block such as grid locations, cell volumes, boundaries, etc. An interblock boundary includes a neighbour derived type storing its neighbour information containing block id, index range, face label, etc. Ghost cells on interblock boundaries contain data from the adjoining block during a syncing routine so that every block can be solved independently. Different boundaries are allocated when reading from the boundary file, and are re-allocated and updated when decomposing domain if running SENSEI on multiple processors.

The SA-neg turbulence model and its linearization for 2D cases are implemented through the use of object-oriented programming [7]. Multiple 2D turbulence verification cases have been used to compare results from CFL3D and FUN3D to verify that the SA-neg turbulence model in SENSEI is implemented correctly for 2D cases. Some differences in the results are attributed to differences in implementation and discretization order of boundary conditions and turbulence models. However, these differences do not have a significant effect on quantifies of interest such as pressure coefficient, viscous drag coefficient, etc. for all the 2D turbulence cases te. For more details on the theory, background and 2D case tests see Derlag et al. [6] and Jackson et al. [7].

6.2.2 Turbulence Models into the FANS Equations in SENSEI

The governing equations used in the code can be generalized into a weak form as

$$\frac{\partial}{\partial t}\int_{\Omega}\vec{U}\mathrm{d}\Omega + \oint_{\partial\Omega}(\vec{F_{i,n}} - \vec{F_{\nu,n}})\mathrm{d}s = \int_{\Omega}\vec{S}\mathrm{d}\Omega, \tag{6.1}$$

where $\vec{U}$ is the vector of conversed variables, $\vec{F_{i,n}}$ and $\vec{F_{\nu,n}}$ are the inviscid and viscous flux normal components respectively (can be understood as the dot product of the second order flux tensor and the normal vector of the surface) and $\vec{S}$ is the source term from either body forces, chemistry source terms, or the method of manufactured solutions. The Favre-Averaged Navier-Stokes equations (FANS) can be given in the framework shown in Eq. 6.1 as

$$\vec{U} = \begin{bmatrix} \bar{\rho} \\ \bar{\rho}\tilde{u} \\ \bar{\rho}\tilde{v} \\ \bar{\rho}\tilde{w} \\ \bar{\rho}e_t \end{bmatrix}, \vec{F_{i,n}} = \begin{bmatrix} \bar{\rho}\tilde{V}_n \\ \bar{\rho}\tilde{u}\tilde{V}_n + \hat{n}_x p \\ \bar{\rho}\tilde{v}\tilde{V}_n + \hat{n}_y p \\ \bar{\rho}\tilde{w}\tilde{V}_n + \hat{n}_z p \\ \bar{\rho}h_t\tilde{V}_n \end{bmatrix}, \vec{F_{\nu,n}} = \begin{bmatrix} 0 \\ \hat{n}_x\tilde{\tau}_{xx} + \hat{n}_y\tilde{\tau}_{xy} + \hat{n}_z\tilde{\tau}_{xz} \\ \hat{n}_x\tilde{\tau}_{yx} + \hat{n}_y\tilde{\tau}_{yy} + \hat{n}_z\tilde{\tau}_{yz} \\ \hat{n}_x\tilde{\tau}_{zx} + \hat{n}_y\tilde{\tau}_{zy} + \hat{n}_z\tilde{\tau}_{zz} \\ \hat{n}_x\tilde{\Theta}_x + \hat{n}_y\tilde{\Theta}_y + \hat{n}_z\tilde{\Theta}_z \end{bmatrix}, \vec{S} = \vec{0} \tag{6.2}$$

where $\bar{\rho}$ is density, $\tilde{u}$, $\tilde{v}$, $\tilde{w}$ are the Cartesian velocity components, e_t is the total energy, h_t is the total enthalpy, $\tilde{V}_n = \hat{n}_x\tilde{u} + \hat{n}_y\tilde{v} + \hat{n}_z\tilde{w}$ and the $\hat{n}_i$ terms are the components of the outward-facing normal unit vector. The laminar and turbulent effects are combined in the definition for the viscous stresses which gives

$$\begin{aligned}
\tilde{\tau}_{xx} &= 2\mu_{eff}\left(S_{xx} - \frac{1}{3}\left(\frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} + \frac{\partial \tilde{w}}{\partial z}\right)\right) - \frac{2}{3}\bar{\rho}k,\\
\tilde{\tau}_{yy} &= 2\mu_{eff}\left(S_{yy} - \frac{1}{3}\left(\frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} + \frac{\partial \tilde{w}}{\partial z}\right)\right) - \frac{2}{3}\bar{\rho}k,\\
\tilde{\tau}_{zz} &= 2\mu_{eff}\left(S_{zz} - \frac{1}{3}\left(\frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} + \frac{\partial \tilde{w}}{\partial z}\right)\right) - \frac{2}{3}\bar{\rho}k,\\
\tilde{\tau}_{xy} &= 2\mu_{eff}S_{xy},\\
\tilde{\tau}_{yz} &= 2\mu_{eff}S_{yz},\\
\tilde{\tau}_{zx} &= 2\mu_{eff}S_{zx},
\end{aligned} \tag{6.3}$$

where S_{xx}, S_{xy}, ..., etc. are the mean strain rates, k is the turbulent kinetic energy and the effective viscosity μ_{eff} is defined as

$$\mu_{eff} = \mu + \mu_T, \tag{6.4}$$

where μ is the dynamic viscosity and μ_T is the turbulent eddy viscosity. $\tilde{}$ represents the heat conduction, work from the viscous stresses and contribution from turbulent effects. Its components are given as

$$\begin{aligned}
\tilde{\Theta}_x &= \tilde{u}\tilde{\tau}_{xx} + \tilde{v}\tilde{\tau}_{xy} + \tilde{w}\tilde{\tau}_{xz} + k_{eff}\frac{\partial \tilde{T}}{\partial x} + MDTT_x,\\
\tilde{\Theta}_y &= \tilde{u}\tilde{\tau}_{xy} + \tilde{v}\tilde{\tau}_{yy} + \tilde{w}\tilde{\tau}_{yz} + k_{eff}\frac{\partial \tilde{T}}{\partial y} + MDTT_y,\\
\tilde{\Theta}_z &= \tilde{u}\tilde{\tau}_{xz} + \tilde{v}\tilde{\tau}_{yz} + \tilde{w}\tilde{\tau}_{zz} + k_{eff}\frac{\partial \tilde{T}}{\partial z} + MDTT_z,
\end{aligned} \tag{6.5}$$

where $MDTT$ is the lumped term associated with molecular diffusion and turbulent transport in the energy equation of a given turbulence model and k_{eff} is effective thermal conductivity which is given by

$$k_{eff} = C_p\left(\frac{\mu}{Pr_L} + \frac{\mu_T}{Pr_T}\right). \tag{6.6}$$

In Eq. 6.6, Pr_L is the (laminar) Prandtl number and Pr_T is the turbulent Prandtl number. In order to close the system, a turbulence model is needed to model the turbulent eddy viscosity μ_T and the turbulent kinetic energy k.

6.2.3 The Negative Spalart-Allmaras Turbulent Model

The implementation of the original Spalart-Allmaras (SA) turbulence model can be given in the form shown in Eq. 6.1 as

$$\vec{U} = [\bar{\rho}\tilde{\nu}],\ \vec{F}_{i,n} = [\bar{\rho}\tilde{\nu}\tilde{V}_n],\ \vec{F}_{\nu,n} = [\hat{n}_x\tilde{\Upsilon}_x + \hat{n}_y\tilde{\Upsilon}_y + \hat{n}_z\tilde{\Upsilon}_z], \tag{6.7}$$

where $\tilde{\nu}$ is the turbulence working variable which is related to the kinematic eddy viscosity ν_T by

$$\nu_T = \tilde{\nu} f_{\nu 1}, \quad f_{\nu 1} = \frac{\chi^3}{\chi^3 + c_{\nu 1}^3}, \quad \chi = \frac{\tilde{\nu}}{\nu}, \tag{6.8}$$

and

$$\tilde{\Upsilon}_x = \frac{\bar{\rho}}{\sigma}(\nu + \tilde{\nu})\frac{\partial\tilde{\nu}}{\partial x},\ \tilde{\Upsilon}_y = \frac{\bar{\rho}}{\sigma}(\nu + \tilde{\nu})\frac{\partial\tilde{\nu}}{\partial y},\ \tilde{\Upsilon}_z = \frac{\bar{\rho}}{\sigma}(\nu + \tilde{\nu})\frac{\partial\tilde{\nu}}{\partial z}, \tag{6.9}$$

where ν is the kinematic viscosity. The source term for the SA model is given as

$$\vec{S} = [\bar{\rho}(P-D) + \frac{c_{b2}}{\sigma}\bar{\rho}\left[\left(\frac{\partial\tilde{\nu}}{\partial x}\right)^2 + \left(\frac{\partial\tilde{\nu}}{\partial y}\right)^2 + \left(\frac{\partial\tilde{\nu}}{\partial z}\right)^2\right] - \frac{1}{\sigma}(\nu+\tilde{\nu})\left[\frac{\partial\bar{\rho}}{\partial x}\frac{\partial\tilde{\nu}}{\partial x} + \frac{\partial\bar{\rho}}{\partial y}\frac{\partial\tilde{\nu}}{\partial y} + \frac{\partial\bar{\rho}}{\partial z}\frac{\partial\tilde{\nu}}{\partial z}\right]], \tag{6.10}$$

where P is a production term and D is a destruction term given by

$$P = c_{b1}(1 - f_{t2})\tilde{S}\tilde{\nu}, \qquad D = (c_{w1}f_w - \frac{c_{b1}}{\kappa^2}f_{t2})\left[\frac{\tilde{\nu}}{d}\right]^2. \tag{6.11}$$

The term d, defined as the distance to the nearest wall, and the modified vorticity $\tilde{S}$, are given by

$$\tilde{S} = \Omega + \frac{\tilde{\nu}}{\kappa^2 d^2} f_{\nu 2}, \quad f_{\nu 2} = 1 - \frac{\chi}{1 + \chi f_{\nu 1}}, \tag{6.12}$$

where Ω is the magnitude of vorticity. The remaining terms are given as

$$f_{t2} = c_{t3} exp(-c_{t4}\chi^2), \quad f_w = g\left[\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right]^{1/6}, \quad g = r + c_{w2}(r^6 - r), \quad r = min\left(\frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}, r_{lim}\right). \tag{6.13}$$

The coefficients used in the model are given as

$c_{b1} = 0.1355$	$\sigma = 2/3$	$c_{b2} = 0.622$	$\kappa = 0.41$
$c_{w1} = \dfrac{c_{b1}}{\kappa^2} + \dfrac{1 + c_{b2}}{\sigma}$	$c_{w2} = 0.3$	$c_{w3} = 2$	$c_{\nu 1} = 7.1$
$c_{t1} = 1$	$c_{t2} = 2$	$c_{t3} = 1.2$	$c_{t4} = 0.5$
$r_{lim} = 10$			

The $MDTT$ term for the SA model is zero. Since the model does not model the turbulent kinetic energy, k, all terms related to k in Eq. 6.2 are dropped for the SA model.

In the original SA model, the working variable, $\tilde{\nu}$, is not allowed to be negative. The initial transients in the solution can drive $\tilde{\nu}$ to negative especially near the edge of a wake. The SA model ran into many such issues in our initial testing. The negative Spalart-Allmaras (SA-neg) model [2] is a modified version of the SA model which allows $\tilde{\nu}$ to be negative. The SA-neg model is the same as the SA model when $\tilde{\nu}$ is greater than or equal to zero. The modification is only applied when $\tilde{\nu}$ is negative. μ_T is set to zero for negative $\tilde{\nu}$ and the definitions for Υ_x, Υ_y and Υ_z are modified to be

$$\tilde{\Upsilon}_x = \frac{\bar{\rho}}{\sigma}(\nu + f_n\tilde{\nu})\frac{\partial \tilde{\nu}}{\partial x}, \ \tilde{\Upsilon}_y = \frac{\bar{\rho}}{\sigma}(\nu + f_n\tilde{\nu})\frac{\partial \tilde{\nu}}{\partial y}, \ \tilde{\Upsilon}_z = \frac{\bar{\rho}}{\sigma}(\nu + f_n\tilde{\nu})\frac{\partial \tilde{\nu}}{\partial z}. \tag{6.14}$$

The source term is also modified is given by

$$\vec{S} = [\bar{\rho}(P_n - D_n) + \frac{c_{b2}}{\sigma}\bar{\rho}\Big[\Big(\frac{\partial\tilde{\nu}}{\partial x}\Big)^2 + \Big(\frac{\partial\tilde{\nu}}{\partial y}\Big)^2 + \Big(\frac{\partial\tilde{\nu}}{\partial z}\Big)^2\Big] - \frac{1}{\sigma}(\nu + f_n\tilde{\nu})\Big[\frac{\partial\bar{\rho}}{\partial x}\frac{\partial\tilde{\nu}}{\partial x} + \frac{\partial\bar{\rho}}{\partial y}\frac{\partial\tilde{\nu}}{\partial y} + \frac{\partial\bar{\rho}}{\partial z}\frac{\partial\tilde{\nu}}{\partial z}\Big]], \tag{6.15}$$

where the modified production and destruction term P_n and D_n are given as

$$P_n = c_{b1}(1 - c_{t3})S\tilde{\nu}, \quad D_n = -c_{w1}\Big(\frac{\tilde{\nu}}{d}\Big)^2. \tag{6.16}$$

The coefficient f_n that is introduced in the modification is given by

$$f_n = \frac{c_{n1} + \chi^3}{c_{n1} - \chi^3}, c_{n1} = 16. \tag{6.17}$$

6.2.4 The Menter's Shear Stress Transport Turbulence Model

The governing equations for the Menter's Shear Stress Transport turbulence ($k - \omega$ SST) model can also be cast into the form given in Eq. 6.1 as

$$\vec{U} = \begin{bmatrix} \bar{\rho}k \\ \bar{\rho}\omega \end{bmatrix}, \vec{F}_{i,n} = \begin{bmatrix} \bar{\rho}k\tilde{V}_n \\ \bar{\rho}\omega\tilde{V}_n \end{bmatrix}, \vec{F}_{\nu,n} = \hat{n}_x\vec{\Xi}_x + \hat{n}_y\vec{\Xi}_y + \hat{n}_z\vec{\Xi}_z, \tag{6.18}$$

where ω is the specific rate of dissipation and

$$\vec{\Xi}_x = \begin{bmatrix} (\mu + \sigma_k\mu_T)\frac{\partial k}{\partial x} \\ (\mu + \sigma_\omega\mu_T)\frac{\partial \omega}{\partial x} \end{bmatrix}, \quad \vec{\Xi}_y = \begin{bmatrix} (\mu + \sigma_k\mu_T)\frac{\partial k}{\partial y} \\ (\mu + \sigma_\omega\mu_T)\frac{\partial \omega}{\partial y} \end{bmatrix}, \quad \vec{\Xi}_z = \begin{bmatrix} (\mu + \sigma_k\mu_T)\frac{\partial k}{\partial z} \\ (\mu + \sigma_\omega\mu_T)\frac{\partial \omega}{\partial z} \end{bmatrix}. \tag{6.19}$$

For the Menter's SST model, the source term is given by

$$\vec{S} = \begin{bmatrix} P_s - \beta^* \bar{\rho}\omega k \\ \dfrac{\gamma}{\nu_T} P_s - \beta\bar{\rho}\omega^2 + 2(1-F_1)\dfrac{\bar{\rho}\sigma_{\omega 2}}{\omega}\left(\dfrac{\partial k}{\partial x}\dfrac{\partial \omega}{\partial x} + \dfrac{\partial k}{\partial y}\dfrac{\partial \omega}{\partial y} + \dfrac{\partial k}{\partial z}\dfrac{\partial \omega}{\partial z}\right) \end{bmatrix}, \tag{6.20}$$

where $\nu_T = \mu_T/\bar{\rho}$ and the term P_s is a production term which is given by

$$P_s = \tilde{\tau}_{xx}\frac{\partial \tilde{u}}{\partial x} + \tilde{\tau}_{xy}\frac{\partial \tilde{u}}{\partial y} + \tilde{\tau}_{xz}\frac{\partial \tilde{u}}{\partial z} + \tilde{\tau}_{xy}\frac{\partial \tilde{v}}{\partial x} + \tilde{\tau}_{yy}\frac{\partial \tilde{v}}{\partial y} + \tilde{\tau}_{yz}\frac{\partial \tilde{v}}{\partial z} + \tilde{\tau}_{xz}\frac{\partial \tilde{w}}{\partial x} + \tilde{\tau}_{yz}\frac{\partial \tilde{w}}{\partial y} + \tilde{\tau}_{zz}\frac{\partial \tilde{w}}{\partial z} \tag{6.21}$$

Here the $\tilde{\tau}$ terms do not take the dynamic viscosity into consideration, i.e., they only include the effect of the turbulent eddy viscosity, which is calculated as

$$\mu_T = \frac{\bar{\rho} a_1 k}{max(a_1\omega, \Omega F_2)}. \tag{6.22}$$

where Ω is the vorticity magnitude. The terms F_1 and F_2 are used to blend an inner (expressed with subscript 1) and outer (expressed with subscript 2) coefficients, which are given as

$$F_1 = tanh(arg_1^4), \quad arg_1 = min\left[max\left(\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right), \frac{4\bar{\rho}\sigma_{\omega 2}k}{CD_{k\omega}d^2}\right], \tag{6.23}$$

where

$$CD_{k\omega} = max\left[2\bar{\rho}\sigma_{\omega 2}\frac{1}{\omega}\left(\frac{\partial k}{\partial x}\frac{\partial \omega}{\partial x} + \frac{\partial k}{\partial y}\frac{\partial \omega}{\partial y} + \frac{\partial k}{\partial z}\frac{\partial \omega}{\partial z}\right), 10^{-20}\right], \tag{6.24}$$

and

$$F_2 = tanh(arg_2^2), \quad arg_2 = max\left(2\frac{\sqrt{k}}{\beta^*\omega d}, \frac{500\nu}{d^2\omega}\right). \tag{6.25}$$

The blending process is performed by

$$\phi = F_1\phi_1 + (1-F_1)\phi_2, \tag{6.26}$$

where ϕ can be any of the coefficients. A production limiter [12] is applied, which replaces the P_s term in the k-equation by $min(P_s, 20\beta^*\bar{\rho}\omega k)$. The coefficients in the model are given as

$$\gamma_1 = \frac{\beta_1}{\beta^*} - \frac{\sigma_{\omega_1}\kappa^2}{\sqrt{\beta^*}} \qquad \gamma_2 = \frac{\beta_2}{\beta^*} - \frac{\sigma_{\omega_2}\kappa^2}{\sqrt{\beta^*}}$$

$\sigma_{k1} = 0.85$	$\sigma_{\omega 1} = 0.5$	$\beta_1 = 0.075$
$\sigma_{k2} = 1.0$	$\sigma_{\omega 2} = 0.856$	$\beta_2 = 0.0828$
$\beta^* = 0.09$	$\kappa = 0.41$	$a_1 = 0.31$

The $MDTT$ terms for the Menter's SST model is computed as

$$MDTT_x = \left(\mu + \frac{\mu_T}{\sigma_k}\right)\frac{\partial k}{\partial x}, \quad MDTT_y = \left(\mu + \frac{\mu_T}{\sigma_k}\right)\frac{\partial k}{\partial y}, \quad MDTT_z = \left(\mu + \frac{\mu_T}{\sigma_k}\right)\frac{\partial k}{\partial z}, \tag{6.27}$$

6.2.5 The Menter's Shear Stress Transport Turbulence Model with Vorticity Source Term

There are many variants of the Menter's SST model, such as the Menter's SST model with vorticity source term (SST-V). The SST-V model is slightly different from the standard SST model shown earlier in that the SST-V uses the vorticity magnitude to compute the production term instead of using the shear stresses. The production term given in Eq. 6.21 is replaced with

$$P_s = \mu_T\Omega^2 - \frac{2}{3}\bar{\rho}k\left(\frac{\partial\tilde{u}}{\partial x} + \frac{\partial\tilde{v}}{\partial y} + \frac{\partial\tilde{w}}{\partial z}\right) \tag{6.28}$$

The production limiter used in the standard Menter's SST model is still employed in the k-equation.

6.2.6 Finite-Volume Discretization

For finite-volume discretization, the domain of interest, Ω, is partitioned into a sequence of non-overlapping control volumes, Ω_i, such that $\Omega = \bigcup_{i=1}^{N_\nu} \Omega_i$. The weak form in Eq. 6.1 can be rewritten for each control volume as

$$\frac{\partial}{\partial t}\int_{\Omega_i} \vec{U}\mathrm{d}\Omega + \oint_{\partial\Omega_i} (\vec{F_{i,n}} - \vec{F_{\nu,n}})\mathrm{d}s = \int_{\Omega_i} \vec{S}\mathrm{d}\Omega. \tag{6.29}$$

Denote the discrete solution of finite-volume method as $\vec{U}_h$ which is assumed to be constant in each control volume and approximates the control volume average of the exact solution. With the discrete steady state residual given as $\vec{R}_h$, the discrete version of Eq. 6.29 can be given in a semi-discrete form as

$$|\Omega_i|\frac{\partial}{\partial t}\vec{U}_h + \vec{R}_h = \vec{0}, \tag{6.30}$$

where $|\Omega_i|$ is the volume of Ω_i. Any time marching scheme can be applied to this semi-discrete form given a reasonable initial condition.

6.2.7 Focus of this Paper

Based on the prior work by Derlaga et al. [6] and Jackson et al. [7], this paper is mainly focused on the following aspects. First, we extended the SA-neg turbulence model implementation and its linearization into 3D. We compared the results of a 3D turbulence case with those computed using CFL3D and FUN3D. Second, we included the $k - \omega$ SST turbulence model into SENSEI and finished implementing the nondimensional form and the linearization of $k - \omega$ SST. We ran all the 2D and 3D verification cases in Ref [13] and also compared them with published results using CFL3D and FUN3D. Third, we applied cross term sinusoidal (CTS) manufactured solutions and did order of accuracy tests, and verified that SENSEI is 2nd order accurate for the SA and $k-\omega$ SST models, both in 2D and 3D. Finally, we finished parallelizing SENSEI through the use of MPI, so that a lot of tasks including wall distance calculation, solving a linear system of equations, and part of domain decomposition work can be done in parallel on multiple processors. This brings significant speedup when solving large problems either implicitly or explicitly.

6.3 MPI Implementation into SENSEI

6.3.1 An Introduction of Message Passing Interface (MPI)

Message Passing Interface (MPI) is a message passing library standard based on the consensus of the MPI Forum [14]. The goal of MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. MPI is the ”industry standard” for writing message passing programs on HPC platforms. It defines a lot of specifications for C and Fortran language bindings. MPI can run on shared memory, distributed memory or hybrid platforms. There are multiple MPI implementations such as MVAPICH, OpenMPI (used in this paper), Intel MPI, etc. All of them offer various point-to-point communication routines as well as a lot of group communication routines, which satisfy different purposes.

6.3.2 Domain Decomposition in SENSEI

A processor-clustered decomposition method is implemented into SENSEI. This decomposition method is proper when the number of processors (NP) is greater than or equal to the number of parent blocks (PB) given a grid. The method has two modules, the block decomposition module dealing with decomposition for parent blocks and the number decomposition module dealing with decomposition within a block which may have different number of cells in different dimensions. The algorithms of the two modules are detailed in Fig. 6.1 and Fig. 6.2. It is an "on the fly" approach which requires no preprocessing of the grid or boundaries. The communication overhead is also small with loads on different processors balanced well, because the grid is decomposed in a structured way. The structured decomposition makes programming and communication much easier as there are some built-in topology routines in MPI which have good supports for point-to-point or group communication. Although this decomposition method is in a manager-slave mode, some tasks such as linking interblock boundaries after domain decomposition can be done partly in parallel. It helps to reduce some decomposition overhead, especially when NP becomes very large. For all the cases running in parallel in this paper, the decomposition can be performed in a short amount of time. The processor-clustered approach may have load imbalance issue if NP is not obviously greater than PB and PB is greater than 1. We may devise a grid-clustered decomposition method in the future, however this is not a focus in this paper.

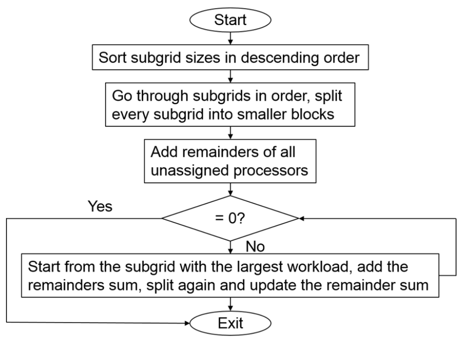

Start
Sort subgrid sizes in descending order
Go through subgrids in order, split every subgrid into smaller blocks
Add remainders of all unassigned processors
= 0?
Yes
No
Start from the subgrid with the largest workload, add the remainders sum, split again and update the remainder sum
Exit

Figure 6.1: Block decomposition module

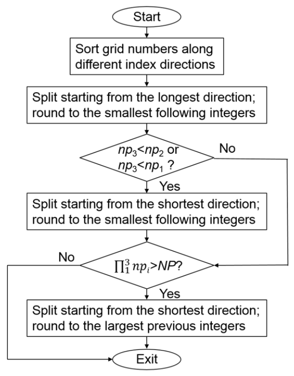

Start
Sort grid numbers along different index directions
Split starting from the longest direction; round to the smallest following integers
$np_3<np_2$ or $np_3<np_1$?
No
Yes
Split starting from the shortest direction; round to the smallest following integers
No
$\prod_1^3 np_i > NP$?
Yes
Split starting from the shortest direction; round to the largest previous integers
Exit

Figure 6.2: Integer Factorization module

Boundaries also need to be decomposed and updated after domain decomposition. Initially, only the ROOT processor has all the boundary information for all parent blocks, since ROOT reads the grid and boundaries. After domain decomposition, each parent block is decomposed into a number of child blocks. These child blocks need to update all the boundaries for themselves, synchronously in SENSEI. For non-connected boundaries this update is very straightforward as each processor just needs to compare their individual block global index range with the boundary global range. For interior boundaries caused by domain decomposition, a family of Cartesian MPI topology routines are used to setup communicators and make communication much less troublesome. However for connected parent block boundaries, the updates (decomposing and re-linking these boundaries) are more difficult as this process is mainly completed on individual processors, instead of on the ROOT processor. In this way, this procedure can be done in parallel so it is much faster, and the ROOT processor does not need to do the entire domain decomposition work for all other blocks. For every parent block connected boundary, the ROOT processor first broadcasts the boundary to all processors within that parent and its neighbour parent block, and then returns to deal with the next parent block connected boundary. The processors within that parent block or its neighbour parent block compare the boundary received and their block index ranges. If a processor does not contain any index range of the parent boundary, it moves forward to compare the next parent boundary. Also, processors in the parent block having this boundary or processors in the neighbour parent block matching part of the neighbour index range are colored valid but differently. To illustrate how we use MPI topology routines and inter-communicators to setup connectivity between neighbour blocks, a 2D example having 3 parent blocks and more than 3 CPUs is given in Fig. 6.3. Processors which match a parent connected boundary are included in an inter-communicator. One processor in a parent block then sends its index ranges to processors residing in the neighbour communicator. Then a processor in the neighbour communicator matching part of the index range is a neighbour while others in the neighbour communicator not matching the index range are not neighbour processors. Through looping over all the neighbour processors in the neighbour communicator, one processor sets up connectivity with all its connected neighbours. This process is in parallel as the ROOT processor does not need to participate in this process except for broadcasting the parent boundary to all processors in the parent block and its neighbour parent block at the beginning. There may be special cases. The first special case is that the ROOT is located at a parent block or its neighbour parent block. The ROOT needs to participate in the boundary decomposition and re-linking process, as shown in Fig. 6.3. The second special case is given in the lower right square in Fig.6.3, in which a parent block connects to itself completely. For this case, an inter-communicator is not needed as this processor knows all the information to setup the boundary. However, if a parent block connected boundary is decomposed in a way that some processors connects to themselves partly but also connects to other processors partly on this same parent connected boundary, then it becomes complicated. However, the use of the inter-communicator can still get rid of a lot difficulties for the communication setup for special cases like this.

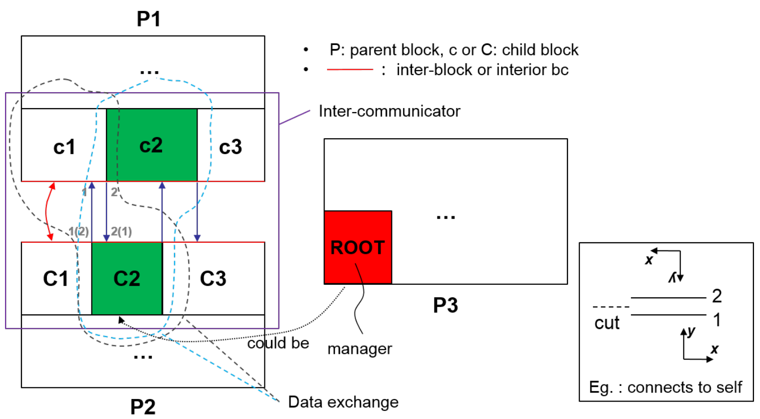


Figure 6.3: An example of using MPI inter-communicator

6.3.3 Wall Distance Calculation in SENSEI

For a case with a moderate grid size (number of cells below 1 million), wall distances for most cases can be computed on one processor in just seconds. However, for a large problem with many cells near walls, the wall distance calculation can take a very long time. For example, for a wind tunnel bump case (this is an application case having four face wall boundaries and 3.5 million cells in total, not shown in the paper), it took the serial SENSEI 3.5 days to finish the wall distance calculation on a workstation. If running the same case using the parallel SENSEI and distributing the wall distance calculation to 24 processors (every processor needs to store all the wall node coordinates for all parent blocks), then this one-time processing is reduced to 3.5 hours, which is a linear speedup. To get a higher speedup or to run larger problems, more processors should be used.

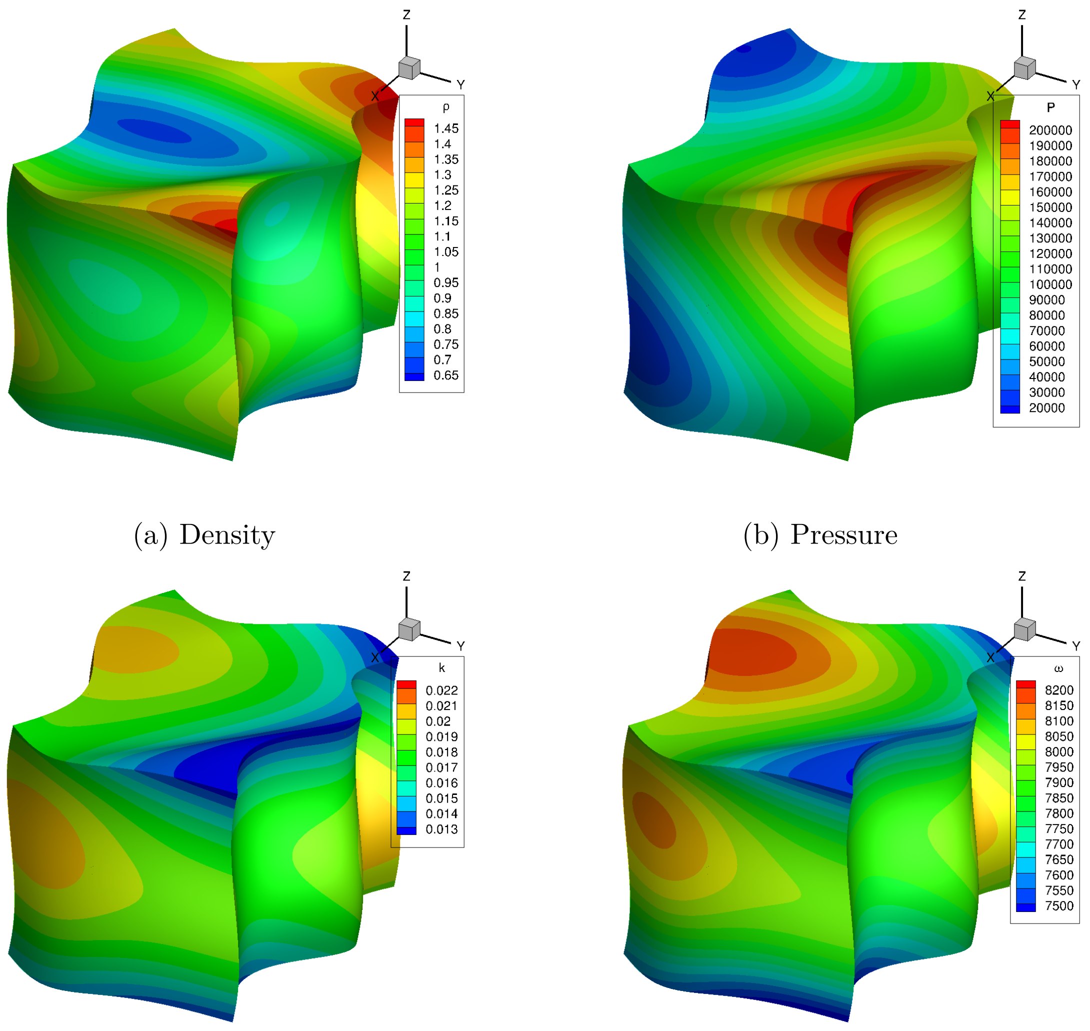
6.4 Order of Accuracy Test Results for the CTS Manufactured Solutions

Since the most rigorous way of verifying a code is the order of accuracy test, this paper uses manufactured solutions which are non-physical but spatially smooth. It should be emphasized that physical solutions are not required as the verification process only deals with the mathematics of a problem and thus can be used to determine the correctness of the implementation. After applying the governing equations to the manufactured solutions, SENSEI calculates the manufactured source terms numerically. The source terms have the wall distance variable inside so we need to specify the wall distance for different cells. For the source terms computed in this paper, we set the wall distance to be a very large value, i.e., 1000 m (we also tried a very small value 10^{-6} m and the results are very similar, which can be seen in [15] but not shown in this paper due to the limit of scope). The manufactured solutions are three sinusoidals in each direction and cross term ones as well. The equation takes the general form

$$\begin{aligned} f = a_1 + a_2 sin(a_3\pi x/l + a_4\pi) + a_5 sin(a_6\pi y/l + a_7\pi) + a_8 sin(a_9\pi z/l + a_{10}\pi) \\ + a_{11} sin(a_{12}\pi xy/l^2 + a_{13}\pi) + a_{14} sin(a_{15}\pi yz/l^2 + a_{16}\pi) + a_{17} sin(a_{18}\pi xz/l^2 + a_{19}\pi) \end{aligned} \tag{6.31}$$

where a_i are the coefficients and can be found in [15] and l is a reference length which is set to be 1.

There are $n_{mean} + n_{turb}$ equations for the mean flow and turbulence flow equations in total. For different equations, the coefficients are set differently. The 3D manufactured solutions for some of the primitive variables in the Menter's SST model are shown in Fig. 6.4. The Reynolds number for this CTS MMS case is about 78 million (low Reynolds numbers range from 5000 to 7000 are also tried and the results can be seen in [15]). If a lower dimension problem is solved then the relevant coefficients are set to 0. The grid size in the paper ranges from 4^d to 256^d (d is the dimension, i.e., 2 for 2D and 3 for 3D).

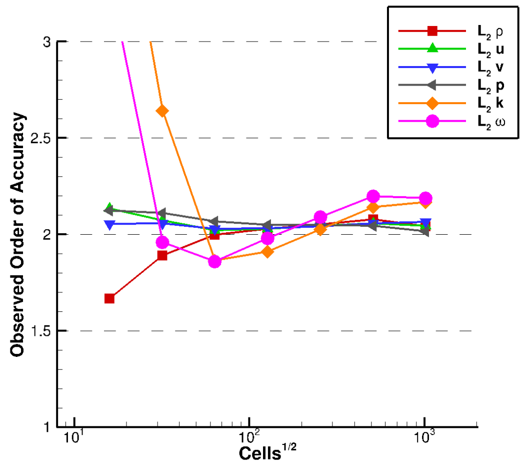


(a) Density (b) Pressure

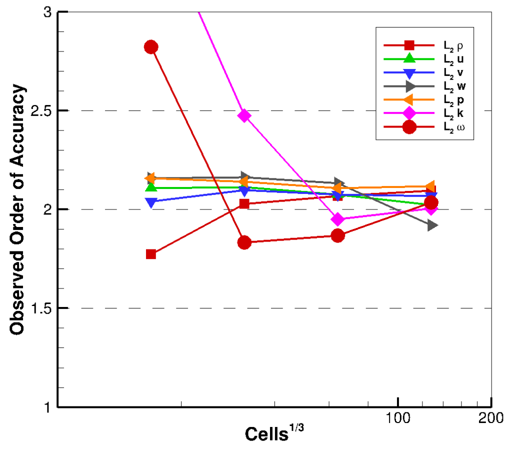
(c) Turbulent kinetic energy (d) Eddy dissipation rate

Figure 6.4: 3D cross term sinusoidal manufactured solutions ($Re = 78$ *million*)

Through applying the CTS manufactured solutions, we te both the 2D and 3D SA implementation in SENSEI and found that the observed order of accuracy is close to two if the grid is refined enough. The CTS manufactured solutions using the SA model are not shown in the paper but can be found in [15]. The observed order of accuracy results for the 2D and 3D Menter's SST models are given in Fig. 6.5. For both the 2D and 3D CTS MMS cases, the values for the turbulent kinetic energy and turbulent eddy dissipation rate are chosen carefully so that the turbulent eddy viscosity has the same order O(1) as the viscous kinetic viscosity. Although the chosen value of the turbulent eddy viscosity and the viscous kinetic viscosity is $0.01m^2/s$, which is much higher than the actual value (about $1.8 \times 10^{-5}m^2/s$), the purpose here is to make the importance of the viscous and turbulence approximately equal so that both the mean flow and turbulent flow systems can be verified together. Note that it is only for the code verification purpose. Actually, it is easier to converge the MMS cases using a physical value (e.g. $1.8 \times 10^{-5}m^2/s$) for the viscosity than using a non-physical value (e.g. $0.01m^2/s$). Also, the observed order of accuracy using the physical value for the viscosity is still near 2 on the 2D and 3D CTS MMS cases. The Reynolds number for the 2D case and the 3D case shown are about 5600 and 7000, respectively. For the 3D Menter's SST, the grid size of 256^3 is not applicable as the computer cannot allocate such a big memory on the ROOT processor. The MMS solution case cannot currently be run in parallel but efforts are underway to add this capability.



(a) 2D curvilinear CTS (Re=5600)



(b) 3D curvilinear CTS (Re=7000)

Figure 6.5: CTS MMS: Observed order of accuracy

6.5 Grid Convergence Test Results for the Turbulence Modeling Verification Cases

6.5.1 Some Considerations

Before running any turbulence modeling verification test cases in Ref [13], we ran unit tests for all the cases by comparing the residuals and Jacobians with a finite difference approach. All the verification test cases in this paper passed the unit test.

All the 2D and 3D verification cases in Ref [13], including the 2D zero pressure gradient flat plate, 2D coflowing jet, 2D bump-in-channel, 2D airfoil near-wake (2DANW) and 3D bump-in-channel (3DB) are test using the SA and Menter's SST models in SENSEI. To avoid including all the results and making similar analysis in the paper, only the results of the SA model for 3D bump, Menter's SST model for 2DANW and the SST-V model for 3D bump are shown in this paper. Other verification tests can be found in [15] due to the limit of the scope.

For all the verification cases in Ref [13], we turned off the limiters for the mean flow equations. As we found that using limiters for the mean flow equations makes the converged primitive variables less than 1% different from not using limiters which can be seen in Fig. 6.6, but the pressure drag is much higher when limiters are used. For example, for the 2DANW SA model, on the second coarsest grid (85x113), the comparison of forces between using a Michalak/Olivier-Gooch limiter [16] and no limiter is seen in Table 6.1. The differences exist and are not negligible even if the grid is refined, although smaller. Also, this happens to other verification cases and the Menter's SST model. Usually the pressure drag is off the most and then the total drag. The reason for the difference is that the dissipation caused by using the limiters causes some difference in the pressure, which cannot be neglected when computing the total drag coefficient and the pressure drag coefficient. This also happened in Ref [17] and the difference between using a limiter and no limiter can be more than 2.5 times in the total drag (the example shown in Fig. 6.6 is about 2 times). Moreover, Ref [13] turns off the limiters for all the verification cases so limiters are turned off in this paper to provide a better comparison.

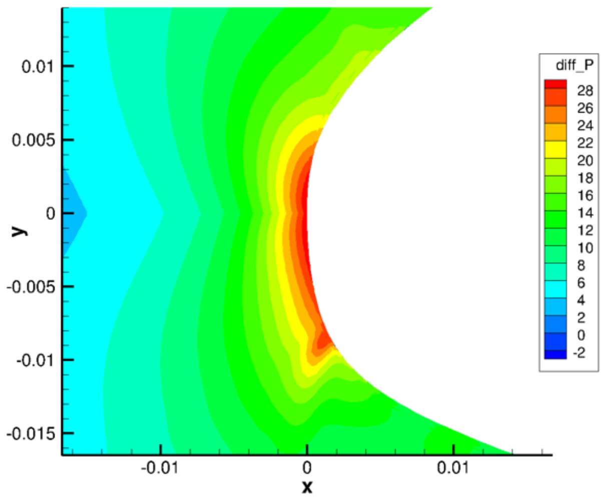


Figure 6.6: Pressure difference around the LE of the airfoil between turning on and turning off a flux limiter

Table 6.1: Comparison of forces between using a limiter and no limiter

	No limiter		using a limiter	
	Axial	Normal	Axial	Normal
Pressure force, N	1.799	56.01	6.292	55.10
Viscous force, N	2.851	2.996×10^{-2}	3.035	2.663×10^{-2}
Total force, N	4.65	56.03	9.327	55.13

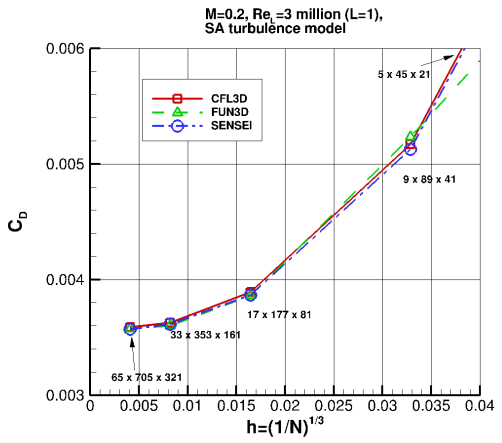
6.5.2 Non-physical Check and Update

For some cases such as the 3D bump, it is normal to encounter non-physical (negative) values for the density, pressure, k and ω during the initial transients in the Menter's SST model (not for the turbulence variable in the SA model since we use the SA-negative model). Sometimes this would make the program crash immediately, especially using a second order primal solver. Flux limiters are necessary for problems with large gradients for the turbulence variables, however using flux limiters may have an adverse effect on the verification tests, as shown above. In the early stage of running the verification cases, we had some issues converging some cases and tried different techniques in Ref [18, 19, 20] but none of them worked well for SENSEI. Although previously SENSEI has a technique to alleviate the effect of the issue, through reducing the CFL number if non-physical values encountered but it is very easy to hit the minimum CFL tolerance in some occasions. We later used a simple technique to stabilize SENSEI. SENSEI now checks whether there is any non-physical value for some primitive variables at each step. If SENSEI detects one non-physical value in a cell at one step, there is no update for that variable in that cell (only at that step), i.e, freezing the relevant variable for that cell if a non-physical value emerges at that step. For the next step, SENSEI will still update the values if there is no non-physical phenomenon. Although this technique seems very simple but it is very useful as it makes all of our tests stable.

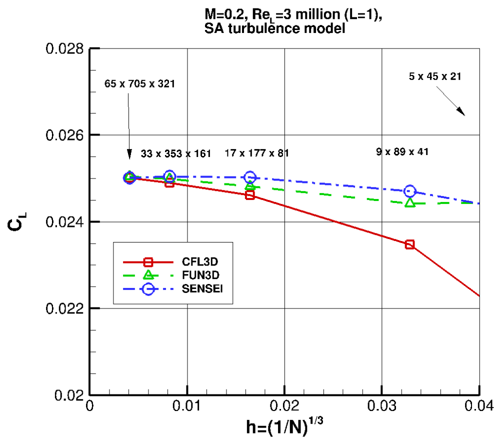
6.5.3 SA model for 3D Bump

This is a 3D bump (3DB) case which has been numerically solved using CFL3D and FUN3D in Ref [13]. For SENSEI in this paper, 36 CPUs are used in parallel for the finest grid (65x705x321) with more than 14 million cells. To be consistent with CFL3D and FUN3D, SENSEI uses Roe's flux difference splitting, MUSCL extrapolation with kappa=0.3333, and first order upwinding for the advective terms of the SA model. All the absolute iterative residuals including the mean flow and turbulence equations are driven down below 10^{-12} on all grids except for the finest grid (for the finest grid, the iterative residual norms for the mean flow equations are below 10^{-12} while the turbulence variable equation residual norm is only under 10^{-10}). The residual history plot can be found in [15].

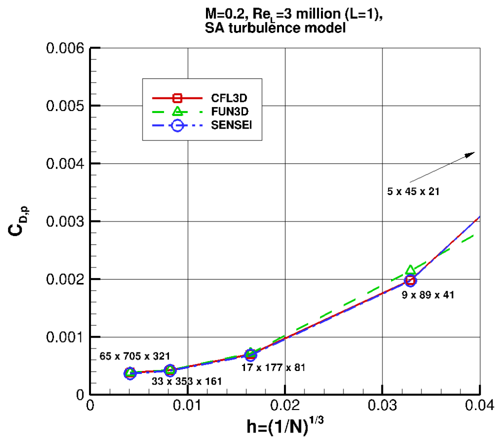
There are four quantities of interest for the 3D bump case including the total drag coefficient C_D, total lift coefficient C_L, pressure drag coefficient $C_{D,p}$ and viscous drag coefficient $C_{D,\nu}$. Their values on different levels of grid are shown in Fig. 6.7 and the observed order of accuracy is given in Table 6.2. SENSEI matches very well with CFL3D and FUN3D for all quantities of interest if the grid is refined enough (especially with CFL3D, possibly because CFL3D and SENSEI are both cell centered structured codes). Contour such as surface pressure coefficient and eddy viscosity are very similar to the CFL3D and FUN3D results so that there are not presented here.



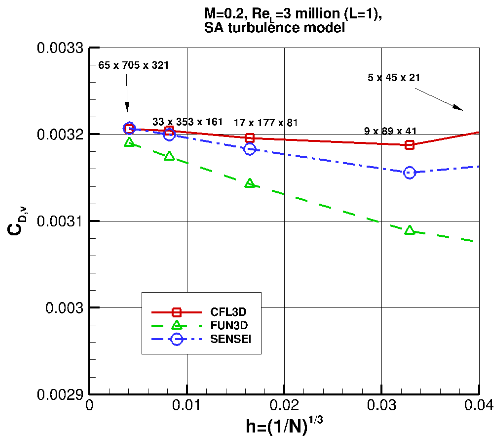
(a) Total drag coefficient



(b) Total lift coefficient



(c) Pressure drag coefficient



(d) Viscous drag coefficient

Figure 6.7: 3DB SA: Grid convergence of various quantities

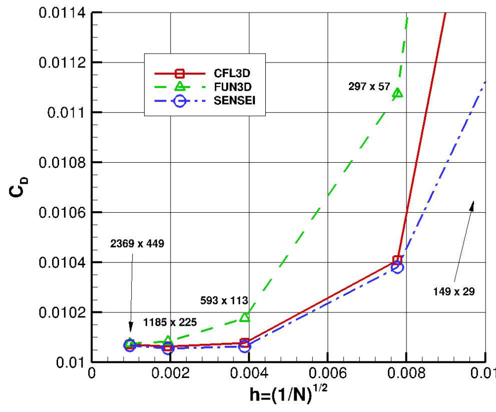
Table 6.2: Observed order of accuracy on the three finest grid for the 3DB SA

Quantity	CFL3D	FUN3D	SENSEI
C_L	1.34	1.45	oscillatory
C_D	2.76	3.25	2.71
C_{Dp}	2.75	2.74	2.55
C_{Dv}	2.45	0.93	1.20

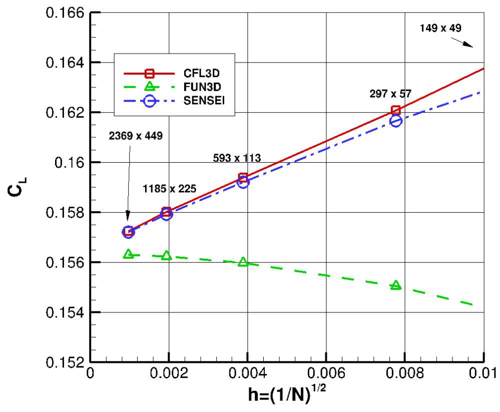
6.5.4 Menter's SST model for 2DANW

This DSMA661(MODEL A) airfoil case can be found in Ref [13]. Since this case is only 2D, 2 CPUs are used for all levels of grid. Roe's flux difference splitting, MUSCL extrapolation with kappa=0.3333, and first order upwinding for the advective terms of the Menter's SST model are used in SENSEI. All the absolute iterative residuals including the mean flow and turbulence variables are driven down below 10^{-11}. The residual history plot can be found in [15].

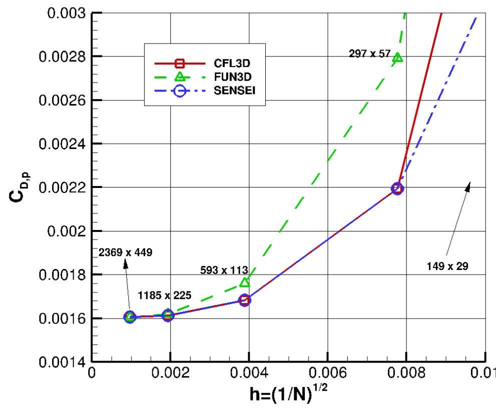
The grid convergence study for the total lift coefficient, total drag coefficient, pressure drag coefficient and viscous drag coefficient are shown in Fig. 6.8 and the observed order of accuracy for these quantities is given in Table 6.3. Similar to the 3DB using the SA model, all the quantities of interest from SENSEI show indiscernible difference compared to CFL3D and FUN3D if the grid is refined enough. A major difference between SENSEI and the other two codes is that the viscous drag coefficient is monotonic on all levels of grid, even on a coarse grid, while CFL3D and FUN3D are not.



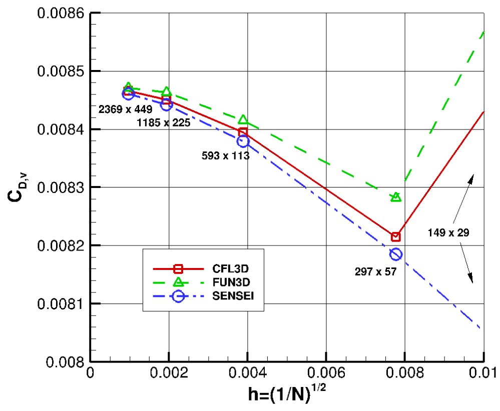
(a) Total drag coefficient



(b) Total lift coefficient



(c) Pressure drag coefficient



(d) Viscous drag coefficient

Figure 6.8: 2DANW Menter's SST: Grid convergence of various quantities

Table 6.3: Observed order of accuracy on the three finest grid for the 2DANW Menter's SST

Quantity	CFL3D	FUN3D	SENSEI
C_L	0.83	1.99	0.82
C_D	oscillatory	3.57	oscillatory
C_{Dp}	3.46	3.22	3.41
C_{Dv}	1.92	2.71	1.77

6.5.5 SST-V model for 3DB

Since Ref [13] provides results for the 3DB case only using the SST-V model, SENSEI also uses the SST-V model here. Similar to the two cases shown earlier, SENSEI uses Roe's flux difference splitting, MUSCL extrapolation with kappa=0.3333, and first order upwinding for the advective terms of the SST-V model. Similar to the SA model, 36 CPUs are used in SENSEI for the finest grid (65x705x321) but this case runs more slowly than the SA model. All the absolute iterative residuals including the mean flow and turbulence variables are driven down below 10^{-12}, which can be seen in Fig. 6.9. Note that the absolute iterative residuals are given to be consistent with those in Ref [13].

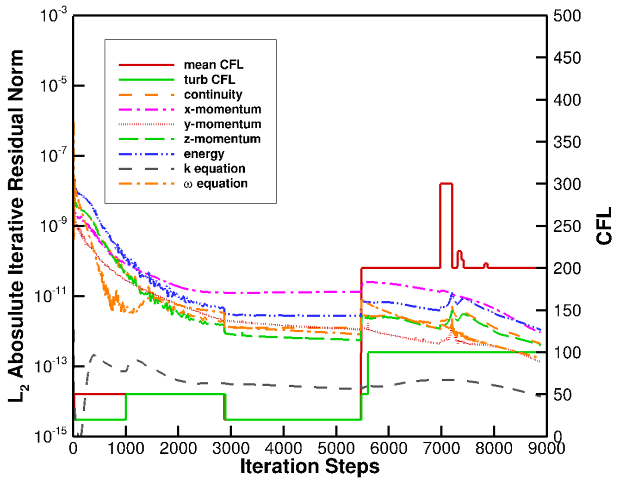
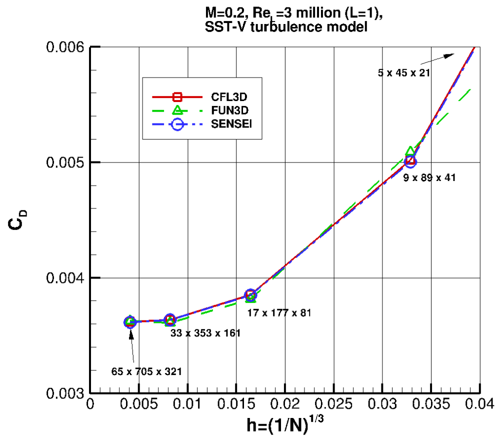


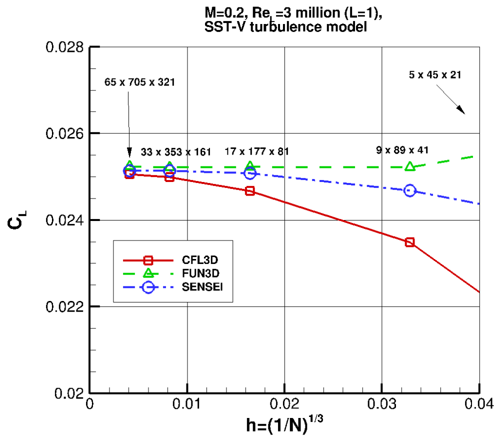
Figure 6.9: Absolute iterative residual history for the 3DB case using the SST-V model

The values for quantities of interest including the total lift coefficient, total drag coefficient, pressure drag coefficient and viscous drag coefficient are shown in Fig. 6.10 and the observed order of accuracy for these quantities is given in Table 6.4. All the codes show simliar total drag coefficient, total lift coefficient and pressure drag coefficient values if the grid is refined enough. For the viscous drag coefficient, SENSEI is slightly lower than the CFL3D and

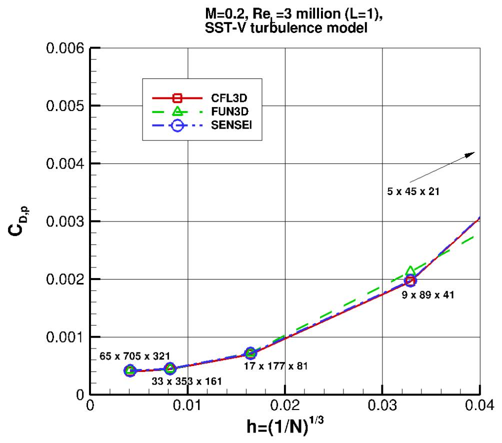
FUN3D results on the finest grid. The reason for the difference may be that the grid is still not fine enough for the viscous drag coefficient.



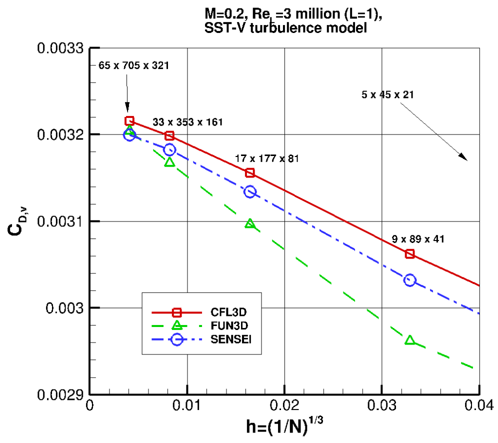
(a) Total drag coefficient



(b) Total lift coefficient



(c) Pressure drag coefficient



(d) Viscous drag coefficient

Figure 6.10: 3DB Menter's SST: Grid convergence of various quantities

Table 6.4: Observed order of accuracy on the three finest grid for the 3DB Menter's SST-V

Quantity	CFL3D	FUN3D	SENSEI
C_L	2.13	oscillatory	oscillatory
C_D	3.49	oscillatory	3.41
C_{Dp}	2.86	4.00	2.81
C_{Dv}	1.37	0.94	1.48

6.6 Conclusions

In this paper, the 3D linearization for the SA model and 2D and & 3D Menter's $k-\omega$ SST model were implemented. Also, some verification tests for the CTS manufactured solutions and all the turbulence verification cases in Ref [13] were performed. Moreover, SENSEI were accelerated using MPI so that SENSEI can be run in parallel on multiple processors. This is very important for large turbulent flow problems (more than several million cells) to obtain results in an efficient manner. For all the CTS manufactured solution verification cases, an order of accuracy of about 2 was obtained for the discretization error. Also, all the cases from the NASA turbulence modeling website were used to provide further verification evidence for the turbulence model implementation and the MPI implementation in SENSEI. For all the 2D and 3D turbulence modeling verification cases, all the quantities of interest matched well with the CFL3D and FUN3D results when the grids were sufficiently refined.

6.7 Future Work

In the future, the SA and $k-\omega$ SST models in SENSEI will be applied to run some practical cases and the results will be validated with experiments. The speedup performance results will be obtained by comparing the serial SENSEI and parallel SENSEI with MPI. Also, in our present implementation, an issue of ”zero diagonal element encountered” may occur sometimes if the CFL is ramped up to a large value. Thus, a better preconditioning may be investigated to make the iterating in SENSEI more robust.

Acknowledgement

This material is partially based on research sponsored by the U.S. Air Force under agreement number FA865019-2-2204. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Bibliography

[1] Turbulence modeling, 2013. (last accessed on 12/02/19).

[2] Steven R Allmaras and Forrester T Johnson. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, pages 1–11, 2012.

[3] Kuei-Yuan Chien. Predictions of channel and boundary-layer flows with a low-reynolds-number turbulence model. *AIAA journal*, 20(1):33–38, 1982.

[4] David C Wilcox. Formulation of the k-w turbulence model revisited. *AIAA journal*, 46(11):2823–2838, 2008.

[5] Florian R Menter. Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal*, 32(8):1598–1605, 1994.

[6] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[7] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model implementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[8] Chris Rumsey, Brian Smith, and George Huang. Description of a website resource for turbulence modeling verification and validation. In *40th Fluid Dynamics Conference and Exhibit*, 2010.

[9] Philip L Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[10] Joseph L Steger and RF Warming. Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods. *Journal of computational physics*, 40(2):263–293, 1981.

[11] Bram Van Leer. Flux-vector splitting for the euler equation. In *Upwind and High-Resolution Schemes*, pages 80–89. Springer, 1997.

[12] FLORIANR Menter. Zonal two equation k-w turbulence models for aerodynamic flows. In *23rd fluid dynamics, plasmadynamics, and lasers conference*, 1993.

[13] Christopher Rumsey. Turbulence modeling resources, 2019. (last accessed on 12/02/19).

[14] Blaise Barney. Message Passing Interface (MPI), 2019.

[15] Weicheng Xue, Hongyu Wang, and Christopher Roy. Turbulence verification case results for sensei, 2019. (last accessed on 12/02/19).

[16] Krzysztof Michalak and Carl Ollivier-Gooch. Limiters for unstructured higher-order accurate solutions of the euler equations. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, page 776, 2008.

[17] Marsha Berger, Michael Aftosmis, and Scott Muman. Analysis of slope limiters on irregular grids. In *43rd AIAA Aerospace Sciences Meeting and Exhibit*, 2005.

[18] Chang Hwan Park and Seung O Park. On the limiters of two-equation turbulence models. *International Journal of Computational Fluid Dynamics*, 19(1):79–86, 2005.

[19] Soo Hyung Park and Jang Hyuk Kwon. Implementation of k-w turbulence models in an implicit multigrid method. *AIAA journal*, 42(7):1348–1357, 2004.

[20] Xiaoqing Zheng and Feng Liu. Staggered upwind method for solving navier-stokes and k-omega turbulence model equations. *AIAA journal*, 33(6):991–998, 1995.

Chapter 7

Code Verification for Unsteady Flows in SENSEI

Weicheng Xue[1], Hongyu Wang[2] and Christopher J. Roy[3]

Virginia Tech, Blacksburg, Virginia, 24061

Attribution

- Weicheng Xue (first author): The first author served as the main contributor and primary author of this study. The first author implemented unsteady flow verification cases in SENSEI. In addition, all the results were collected by the first author.
- Hongyu Wang (second author): The second author implemented various implicit temporal schemes including the singly diagonally implicit Runge Kutta multi-step and three point backward in SENSEI. Also, the second author provided a lot of useful advice for debugging test cases.
- Christopher J. Roy (final author): The final author provided valuable feedback for this

[1]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.
[2]Graduate Assistant, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 315, Virginia Tech, 460 Old Turner St, AIAA Student Member.
[3]Professor, Kevin T. Crofton Department of Aerospace and Ocean Engineering, Randolph Hall, RM 330, Virginia Tech, 460 Old Turner St, AIAA Associate Fellow.

study and comments for this manuscript.

Abstract

Code verification for unsteady flows in a compressible CFD code usually requires the use of manufactured solutions or exact incompressible solutions with source terms added. For combined spatial and temporal order analysis, the spatial discretization error should be a similar order of magnitude as the temporal error to avoid erroneous analysis. For unsteady flows, a systematic refinement should be performed for both the spatial and temporal spacing to determine the correct overall observed order of accuracy. Since explicit time marching schemes typically require smaller time step size compared to implicit time marching schemes due to stability constraints, multiple implicit schemes such as the Singly-Diagonally Implicit Runge-Kutta multi-stage scheme and three point backward scheme are used in our work to mitigate the stability constraints.

7.1 Introduction

CFD is a numerical method to solve fluid flow problems. There are a variety of problems governed by different types of equations which can be solved to satisfactory accuracy. Unsteady Euler/Navier-Stokes equations are used to solve problems in which unsteadiness exists (time dependent).

Code verification [1, 2] is a process to ensure that the implementation of a code is numerically correct so that the code can achieve expected accuracy. For steady flows, the final converged solution should not change any more after convergence, as the spatial residuals have been driven to machine zero and also there is no temporal error. The iteratively converged result is compared to the exact steady state solution to obtain the discretization error on different levels of meshes. For unsteady flows, the procedure is similar but the temporal discretization error exists and should be accounted for, that is, the overall discretization error is comprised of the spatial and temporal discretization error. For unsteady flows, there are multiple sources of error affecting the code verification analysis [1], such as poor mesh quality, improper time step size, large sub-iterative errors, round-off error accumulation, and even implementation errors (which cannot be overlooked).

Since very few exact solutions exist for the unsteady compressible Euler/Navier-Stokes equations, the method of manufactured solution (MMS) is a rigorous method for code verification analysis [1]. It should be noted that MMS can be applied to any codes using any numerical scheme. After plugging the unsteady manufactured solution into the unsteady compressible Euler/Navier-Stokes equations, manufactured source terms can be obtained. These source terms are manufactured, which means that we change the unsteady compressible equations by adding the manufactured source terms. It does not matter whether the manufactured solution is physical or not, as the aim is to verify the correctness of the code implementation, and to ensure that mesh quality is good enough, the time step size is chosen properly, etc.

In fact, there are a lot studies on solving 2D and 3D physical exact problems to the unsteady incompressible Euler/Navier-Stokes equations. Al-Saif et al. [3] proposed a reduced differential transform method which is an iterative procedure to obtain some Taylor series solutions to the kinetically reduced local incompressible Navier-Stokes equations. Different isentropic Euler vortex problems were tested in Ref. [4], with characteristic and periodic boundaries being compared to obtain improved observed order of accuracy (OOA) for high order flux reconstructions. In Ref. [5], 2D steady and unsteady incompressible viscous flows in which the vorticity is proportional to the stream function transported by a uniform stream were studied. Shah et al. [6] presented a time-accurate numerical method using high order accurate compact finite difference to solve some incompressible Navier-Stokes problems. Dirichlet boundary condition was applied for general cases and periodic boundary condition was applied for periodic cases. Tavelli et al. [7] proposed an arbitrary high order accurate discontinuous Galerkin method for solving the three-dimensional incompressible Navier-Stokes equations on staggered unstructured grid. More analytical solutions to the incompressible Euler/Navier-Stokes equations can be also found in Refs. [8, 9, 10, 11, 12, 13]. Plugging any of these solutions into the unsteady compressible Euler/Navier-Stokes equations only requires a source term to be added to the energy equation [14, 15], that is, the exact solution still satisfies the continuity and momentum equations. If the flow Mach number is small (0.1 or so), the magnitude of the source term is also very small, but it cannot be neglected when an OOA test is performed. For an incompressible flow for which only the exact initial solution exists, such as the 3D Taylor-Green vortex flow [16], an often-used approach is using an incompressible flow solver [7, 17, 18] or using a high order approximate process [19] to obtain a reference solution, and use the compressible solver solution to compare to the incompressible reference solution. Non-physical MMS are also widely used for code verification of steady [20, 21, 22] and unsteady [23, 24, 25] problems in the CFD community. In Ref. [20], a thorough code verification study using non-physical MMS was performed on an unstructured finite volume CFD code. Different options including the governing equations, different boundary conditions and different solvers (steady and unsteady) were verified. Minion et al. [24] applied various high order temporal time marching schemes for the incompressible Navier-Stokes equations and found that order reduction for the temporal accuracy can occur if applying time-dependent boundary conditions. Yu et al. [25] used MMS to verify a fluid-structure-interaction solver for both 2D and 3D cases. Instead of using a simultaneous refinement for the spatial and temporal spacing, a recursive Richardson extrapolation approach was used to subtract the spatial error from the total error, allowing separation of the temporal error for unsteady problems.

For the CFD code used in this paper, code verification for steady state solutions has been done in Ref. [26] by applying the code-to-code comparison with CFL3D [27] and FUN3D [28] and a sinusoidal MMS with cross terms in the spatial domain, but no work has been done for unsteady flows. Following up the work in Ref. [26], this paper is focused on the unsteady Euler/Navier-Stokes solver code verification in a CFD code which is called SENSEI [26, 29, 30]. SENSEI has the functionality to estimate the truncation error, which can be used to obtain high order discretization error estimates using error transport equations [31]. In this paper, both exact solutions and manufactured solutions will be applied for code verification of the unsteady Euler/Navier-Stokes solvers in SENSEI.

7.2 The CFD Code Base: SENSEI

7.2.1 Overview of SENSEI

SENSEI is an acronym for Structured, Euler/Navier-Stokes Explicit-Implicit Solver, which is our in-house flow solver initially developed by Derlaga et al. [29]. The initial code was written in a very structured programming style which contained a lot of similar subroutines, with some modern Fortran features such as derived type data and pointers being used. Jackson et al. [30] rewrote the SENSEI code completely using an object-oriented programming style and implemented the negative Spalart-Allmaras turbulence model into SENSEI for some 2D problems [32]. Turning SENSEI to an object-oriented code enables users to add new capabilities to SENSEI more easily due to the improvement of SENSEI's modularity. Xue et al. [26] followed up the work of Jackson et al. [30] by implementing the 3D turbulence models including the negative Spalart-Allmaras [33] and Menter's Shear stress transport [34, 35] and parallelized SENSEI using domain decomposition and Message Passing Interface [36]. Xue et al. [26] did a systematic code verification study on the turbulence modeling code in SENSEI by code-to-code comparison with CFL3D [27, 32] and FUN3D [28, 32], and using MMS.

SENSEI is a multi-block structured finite volume code and is embedded with several inviscid flux options including Roe's flux difference splitting [37], Steger-Warming flux vector splitting [38], and Van Leer's flux vector splitting [39]. MUSCL extrapolation and k-exact reconstruction are provided to achieve second- and high-order inviscid flux, respectively. The viscous and turbulent flux is second-order accurate using a central flux scheme after applying the Green-Gauss theorem. Various time marching schemes including explicit/implicit Runge-Kutta [40, 41, 42] and explicit/implicit Euler are offered. SENSEI has various boundary conditions such as slip/non-slip wall, supersonic/subsonic inflow/outflow, interblock boundaries, etc. Ghost cells as well as boundary face flux options are offered in SENSEI to enforce boundaries accurately, depending on the boundary type.

7.2.2 Governing Equations in Spatially Integral Form

Applying the divergence theorem to calculate the flux, the Favre-averaged Navier-Stokes equations in integral form can be seen in Eq. 7.1:

$$\frac{\partial}{\partial t}\int_{\Omega_j} \vec{U}\mathrm{d}\Omega + \oint_{\partial\Omega_j} (\vec{F_{i,n}} - \vec{F_{\nu,n}})\mathrm{d}s = \int_{\Omega_j} \vec{S}\mathrm{d}\Omega, \tag{7.1}$$

where Ω_j is a control volume (mesh cell), $\vec{U}$ is the vector of conserved variables, $\vec{F_{i,n}}$ and $\vec{F_{\nu,n}}$ are the inviscid and viscous flux normal components respectively (the dot product of the second-order flux tensor and the outward-pointing normal vector of the cell face) and $\vec{S}$ is the source term from body forces, chemistry source terms, or MMS source terms. It should be noted that the source terms are likely to be nonzero if the method of manufactured solutions are plugged into Eq. 7.1. The variables in Eq. 7.1 are given in Eq. 7.2:

$$\vec{U} = \begin{bmatrix} \bar{\rho} \\ \bar{\rho}\tilde{u} \\ \bar{\rho}\tilde{v} \\ \bar{\rho}\tilde{w} \\ \bar{\rho}e_t \end{bmatrix}, \vec{F_{i,n}} = \begin{bmatrix} \bar{\rho}\tilde{V_n} \\ \bar{\rho}\tilde{u}\tilde{V_n} + \hat{n}_x p \\ \bar{\rho}\tilde{v}\tilde{V_n} + \hat{n}_y p \\ \bar{\rho}\tilde{w}\tilde{V_n} + \hat{n}_z p \\ \bar{\rho}h_t\tilde{V_n} \end{bmatrix}, \vec{F_{\nu,n}} = \begin{bmatrix} 0 \\ \hat{n}_x\tilde{\tau}_{xx} + \hat{n}_y\tilde{\tau}_{xy} + \hat{n}_z\tilde{\tau}_{xz} \\ \hat{n}_x\tilde{\tau}_{yx} + \hat{n}_y\tilde{\tau}_{yy} + \hat{n}_z\tilde{\tau}_{yz} \\ \hat{n}_x\tilde{\tau}_{zx} + \hat{n}_y\tilde{\tau}_{zy} + \hat{n}_z\tilde{\tau}_{zz} \\ \hat{n}_x\tilde{\Theta}_x + \hat{n}_y\tilde{\Theta}_y + \hat{n}_z\tilde{\Theta}_z \end{bmatrix}, \tag{7.2}$$

where $\bar{\rho}$ is density, $\tilde{u}$, $\tilde{v}$, $\tilde{w}$ are the Cartesian velocity components, e_t is the total energy, h_t is the total enthalpy, $\tilde{V_n} = \hat{n}_x\tilde{u}+\hat{n}_y\tilde{v}+\hat{n}_z\tilde{w}$ and the $\hat{n}_i$ terms are the components of the outward-facing normal unit vector, τ_{ij} are the components of Reynolds stress terms. $\tilde{}$ represents the heat conduction, work from the viscous stresses and contribution from turbulent effects. If applying an incompressible solution, a nonzero source term can be generated from the $\tilde{}$ terms. The components of $\tilde{}$ are given as

$$\begin{aligned} \tilde{\Theta}_x &= \tilde{u}\tilde{\tau}_{xx} + \tilde{v}\tilde{\tau}_{xy} + \tilde{w}\tilde{\tau}_{xz} + k_{eff}\frac{\partial \tilde{T}}{\partial x} + MDTT_x, \\ \tilde{\Theta}_y &= \tilde{u}\tilde{\tau}_{xy} + \tilde{v}\tilde{\tau}_{yy} + \tilde{w}\tilde{\tau}_{yz} + k_{eff}\frac{\partial \tilde{T}}{\partial y} + MDTT_y, \\ \tilde{\Theta}_z &= \tilde{u}\tilde{\tau}_{xz} + \tilde{v}\tilde{\tau}_{yz} + \tilde{w}\tilde{\tau}_{zz} + k_{eff}\frac{\partial \tilde{T}}{\partial z} + MDTT_z, \end{aligned} \tag{7.3}$$

where $MDTT$ is the lumped term associated with molecular diffusion and turbulent transport in the energy equation of a given turbulence model and k_{eff4} is effective thermal conductivity which is given by

$$k_{eff} = C_p\left(\frac{\mu}{Pr_L} + \frac{\mu_T}{Pr_T}\right). \tag{7.4}$$

In Eq. 7.4, Pr_L is the laminar Prandtl number and Pr_T is the turbulent Prandtl number. For laminar flow problems, all the turbulence terms are neglected.

We can simplify Eq. 7.1 further by combining the spatial flux and source terms and doing an integration over the control volume:

$$|\Omega_j|\frac{\partial}{\partial t}\vec{U}_h + \vec{R}_h = \vec{0}, \tag{7.5}$$

where $|\Omega_i|$ is the volume of Ω_i, $\vec{U}_h$ is the cell averaged solution vector, $\vec{R}_h$ is the spatial residual vector, which is given in Eq.7.6.

$$\vec{R}_h = \sum_{1}^{f}(\vec{F_{i,n}} - \vec{F_{\nu,n}})\ s - |\Omega_j|\vec{S_h}, \tag{7.6}$$

where f is the cell face number (there are six faces for each hexahedral cell), Δs is the face area, and S_h is the cell averaged source term vector. It should be noted that we have not yet discretized the temporal term, which will be covered next.

7.2.3 Temporal Discretization

In this paper, explicit/implicit Runge-Kutta [40, 41, 42] and three point backward [43] temporal schemes are used. The temporal discretization deals with the numerical approximation of the $\frac{\partial}{\partial t}\vec{U}_h$ term in Eq. 7.5. Runge-Kutta is a family of time marching schemes, and SENSEI has explicit Runge-Kutta 2/4 stage schemes and singly-diagonal implicit Runge-Kutta (SDIRK) 1/2/3 stage schemes implemented. A general form of the Runge-Kutta schemes can be seen in Eq. 7.7:

$$\begin{aligned}\vec{U}_h^{n+1} &= \vec{U}_h^n - \Delta t\sum_{i=1}^{s} b_i\vec{R}_h^i,\\ \vec{U}_h^{i} &= \vec{U}_h^n - \Delta t\sum_{j=1}^{s} a_{ij}\vec{R}_h^j,\end{aligned} \tag{7.7}$$

where s is the number of stages, b_i and a_{ij} are constants, which can be found in Ref. [44]. For explicit Runge-Kutta schemes, all $a_{ij} = 0$ when $i \leqslant j$, and for SDIRK, all $a_{ij} = 0$ when $i < j$. Many Runge-Kutta schemes can achieve high order accuracy for the temporal terms, but this paper mainly uses second-order unless otherwise specified.

A general form of the three point backward scheme used in this paper is given in Eq. 7.8:

$$\frac{3\vec{U}_h^{n+1} - 4\vec{U}_h^{n} + \vec{U}_h^{n-1}}{2} = \vec{R}_h^{n+1}. \tag{7.8}$$

As can be seen, three point backward requires the solutions from the previous two steps (no intermediate substep required) and it can achieve second-order accuracy for the temporal terms. SDIRK is used for the first step in SENSEI so that the three point backward can be initiated.

7.2.4 Non-dimensionalization

In SENSEI, the governing equations are non-dimensionalized so that the convergence of the implicit solver performs better. Reference quantities for the density, temperature, length, speed of sound and time are used. Turbulence variables can also be non-dimensionalized based off these reference quantities. The reference variables are given in Eq. 7.9:

$$\begin{aligned} &length: l_{ref} && \text{density: } \rho_{ref} && \text{dynamic viscosity: } \mu_{ref} \\ &time: \frac{l_{ref}}{a_{ref}} && \text{velocity: } a_{ref} && \text{turbulence working variable: } \frac{\mu_{ref}}{\rho_{ref}} \\ & && \text{pressure: } \rho a_{ref}^2 && \text{turbulence kinetic energy: } a_{ref}^2 \\ & && \text{temperature: } t_{ref} && \text{turbulence eddy dissipation ratio: } \frac{\mu_{ref}}{\rho_{ref} a_{ref}^2} \end{aligned} \tag{7.9}$$

7.3 Observed Order of Accuracy Test

For steady flows, the observed order of accuracy can be obtained from performing a systematic refinement over a series of spatial meshes. The process of performing code verification for unsteady flow and steady flow is similar but still different in some aspects. First, the steady flow does not have any temporal discretization error, while the unsteady flow does and the temporal discretization error should be considered. Second, the order of the spatial discretization error can be different from the order of the temporal discretization error; therefore, attention should be paid to ensure the two errors are at the same order if possible. For some explicit schemes, it is difficult to use a large time step size due to the numerical stability constraints. Due to this reason, implicit time marching schemes may be required. Also, usually different primitive variables for the Euler/Navier-Stokes equations may have different frequencies in time, a common time step size may not guarantee the spatial discretization errors to be close to the temporal discretization errors for all variables. The sub-iterative error and round-off error need to be driven down small enough compared to the discretization error, in order to obtain convincing results. Newton's iteration is used for implicit Runge-Kutta schemes in this work.

7.3.1 Spatial Discretization Error

For high-dimensional problems, refinement should be performed on all spatial dimensions. Using a 1D problem as an example for simplicity, for a general p^{th} order accurate numerical scheme, the spatial discretization error can be written in Eq. 7.10.

$$\epsilon_h = g_x \Delta x^p + O(\Delta x^{p+1}), \tag{7.10}$$

where h stands for the grid size in computational coordinates. When the spatial mesh is refined enough to be in the asymptotic range, the difference of the error magnitude on two consecutively refined meshes (refined by a factor of 2) is 2^p times. In fact, the "consecutive" condition is not required to compute the observed order of accuracy (OOA). A more general way of computing the OOA is shown in Eq. 7.11.

$$\hat{p} = \frac{ln(\frac{\|\epsilon_{rh}\|}{\|\epsilon_h\|})}{ln(r)}, \tag{7.11}$$

where r represents the mesh refinement factor of one mesh over the other. Note that this refinement factor r is assumed to be the same in all spatial directions (as well as in time for unsteady flows).

7.3.2 Temporal Discretization Error

The verification procedure for a temporal scheme with no spatial discretization is almost the same as that for the spatial scheme. The only difference is that the time step is refined instead of the mesh spacing. The temporal discretization is given in Eq. 7.12. For explicit time marching schemes, the time step size can be much smaller than using implicit time marching schemes.

$$\epsilon_h = g_t \Delta t^q + O(\Delta t^{q+1}), \tag{7.12}$$

where q is the formal order of accuracy for the temporal discretization error.

7.3.3 Spatial and Temporal Discretization Error

Based on the spatial discretization error and temporal discretization error aforementioned, a simple form of the overall discretization error should combine both [45] and can be written in Eq. 7.13:

$$\epsilon_h = g_x \Delta x^p + g_t \Delta t^q + O(\Delta x^{p+1}) + O(\Delta t^{q+1}). \tag{7.13}$$

A more complicated form of the overall discretization error can have spatial-temporal mixed terms [46], which is not considered in this paper, as that feature only exists for some specific schemes.

It should be mentioned that the temporal discretization error may be much smaller compared to the spatial discretization error for some unsteady problems, so attention should be paid to ensure that it does not adversely affect the final observed order of accuracy [1]. The temporal domain can be regarded as one extra dimension to the spatial domain, requiring a reasonable time step size so that the temporal discretization error is of a similar order to the spatial discretization error.

7.4 OOA Test Results

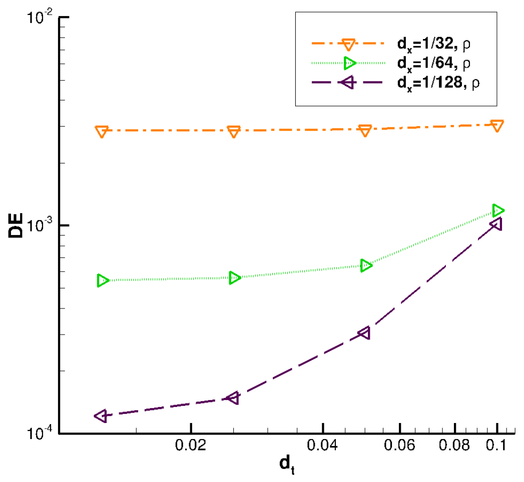
7.4.1 2D Euler Convecting vortex flow

The first test case is a 2D Euler convecting vortex flow [47], of which the non-dimensional solution is given in Eq.7.14:

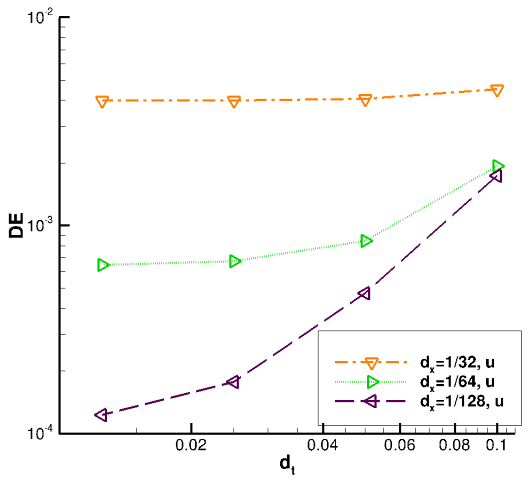
$$\begin{aligned}
u =& u_\infty - \frac{\beta}{2\pi}\exp\left\{\frac{1-r^2}{2}\right\}(y-y_0),\\
v =& v_\infty + \frac{\beta}{2\pi}\exp\left\{\frac{1-r^2}{2}\right\}(x-x_0),\\
T =& 1 - \frac{(\gamma-1)\beta^2}{8\gamma\pi^2}\exp\left\{(1-r^2)\right\},\\
\rho =& T^{\frac{1}{\gamma-1}},\\
p =& \rho^\gamma,
\end{aligned} \tag{7.14}$$

where (x_0, y_0) is the vortex center, $r = \sqrt{((x-x_0)^2 + (y-y_0)^2)}$, β is the vortex strength, and γ is the ratio of specific heats. For this Euler case, $\beta = 5$, $\gamma = 1.4$, $u_\infty = 3$, $v_\infty = 0$, with $l_{ref} = 1\,\mathrm{m}$, $a_{ref} = 1\,\mathrm{m/s}$, $t_{ref} = 1\,\mathrm{K}$, $\rho_{ref} = 1\,\mathrm{kg/m^3}$. Therefore, this problem has a Mach number of 2.54. The reference values are unrealistic but it does not matter as the OOA test does not require realistic solutions. The purpose of applying OOA tests is to help locate implementation errors in the code. The domain of interest is $[0, 10\,\mathrm{m}] \times [-5\,\mathrm{m}, 5\,\mathrm{m}]$. The initial vortex center is at (5,0). Periodic boundaries are applied for the four faces.

As discussed earlier, proper time step sizes should be determined so that the temporal discretization errors are of similar orders as the spatial discretization errors. Considering that different primitive variables are likely to have different magnitudes of discretization errors, a global consideration should be taken. For this 2D Euler convecting vortex flow, a systematic separate OOA study is performed before a combined OOA study. The discretization errors of the density and u velocity component are given in Fig. 7.1. It can be seen that a good non-dimensional time step size on the 128×128 mesh is about 0.025 (since $l_{ref}/a_{ref} = 1$, the dimensional time step size is 0.025 s), since similar orders of magnitude reductions can be achieved as the time step is coarsened (moving to the right on the 128×128 curve) and as the mesh is coarsened (moving from the 128×128 curve to the 64×64 curve at the same dt=0.025). However, due to the fact that the Newton's iteration requires the time step size to be not too large to be stable, a non-dimensional time step size of 0.0125 is used, which will make the temporal discretization error smaller than the spatial discretization error, but they still have the same order of magnitude. When refining systematically in the combined order analysis, time step sizes on other levels of meshes can be easily derived. Fig. 7.2 shows the non-dimensional error contours at $t = 2\,\mathrm{s}$ with a prescribed $\Delta t = 0.0125\,\mathrm{s}$ on the 128×128 mesh. The largest errors occur near the convecting vortex center ($x = 1\,\mathrm{m}$), which is reasonable.



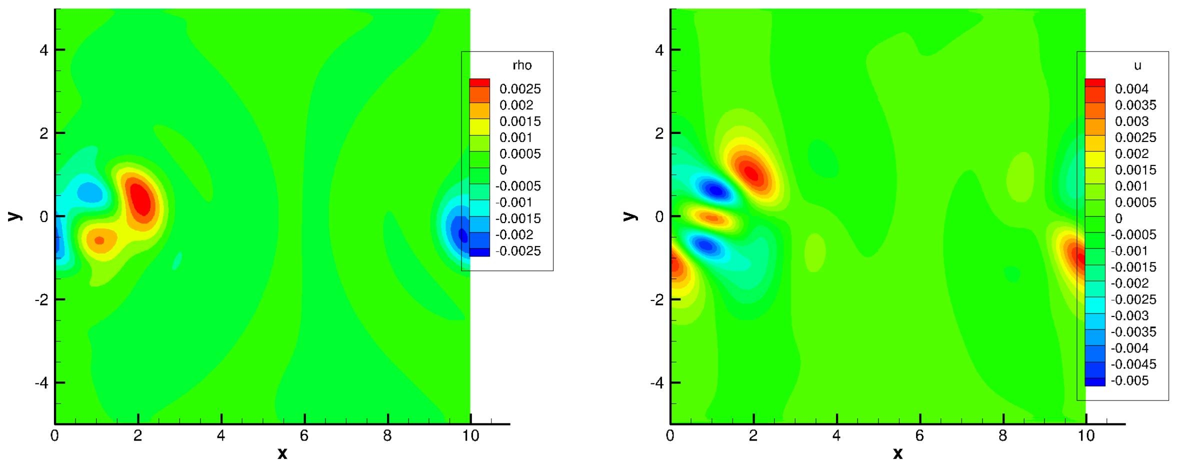
(a) Discretization analysis (ρ)



(b) Discretization analysis (u)

Figure 7.1: 2D Euler vortex flow separate order analysis

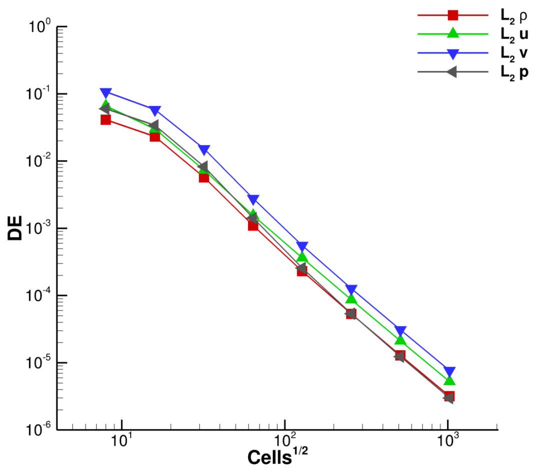
Fig. 7.3 and Fig. 7.4 show the combined OOA results of applying the implicit RK 2 scheme and the three point backward scheme, respectively. For both schemes, an observed order



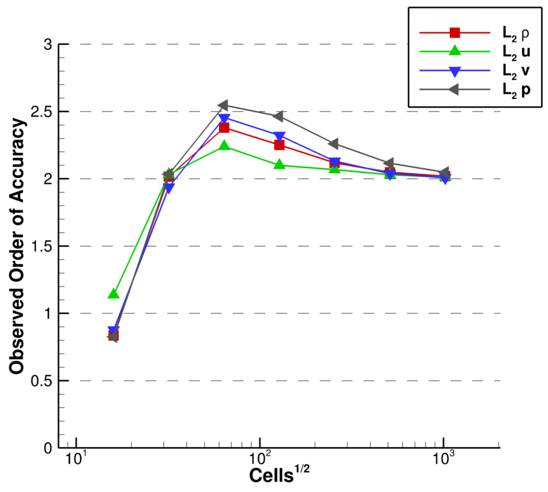
(a) density (b) u velocity

Figure 7.2: Error contours of 2D Euler vortex flow ($M_\infty = 2.54$)

of around 2 is achieved for all primitive variables if the refinement is enough. However, there are some differences. First, the implicit RK 2 scheme has lower discretization errors compared to the three point backward scheme, because implicit RK 2 has more sub-steps than the three point backward so it is more accurate. Second, their behaviours for the OOA plots are different on coarse meshes, which suggests that implicit RK 2 enables larger time step size to balance the spatial and temporal discretization error, since the OOA is higher than 2 using implicit RK 2 on coarse level meshes.



(a) DE



(b) OOA

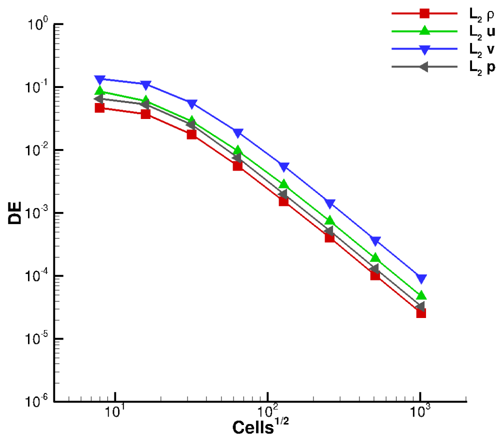
Figure 7.3: 2D Euler vortex flow ($M_\infty = 2.54$): SDIRK 2

7.4.2 2D Taylor-Green Vortex

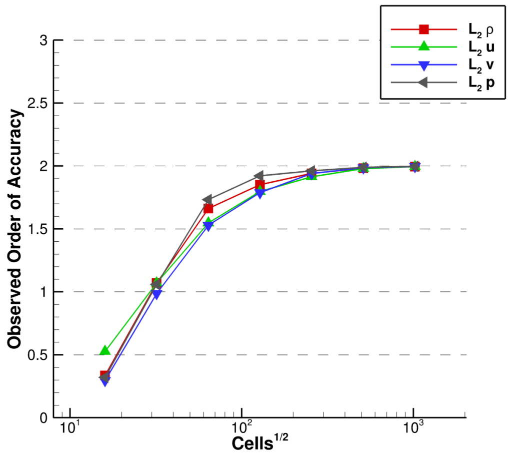
The 2D Taylor-Green vortex is a classic incompressible solution to the 2D laminar Navier-Stokes equations. A non-dimensional solution can be found in Eq. 7.15.

$$\begin{aligned}
\rho =& \rho_0, \\
u =& M cos(x) sin(y) \exp\{-2tM/Re\}, \\
v =& -M sin(x) cos(y) \exp\{-2tM/Re\}, \\
p =& p_0 - \frac{\rho}{4} M^2 (cos(2x) + cos(2y)) \exp\{-4tM/Re\}, \\
T =& \frac{p}{\rho R},
\end{aligned} \tag{7.15}$$

where $\rho_0 = 1$, $p_0 = 0.713$, Re is the Reynolds number, M is the Mach number and R is the gas constant. For this laminar flow case, $l_{ref} = 1\,\mathrm{m}$, $a_{ref} = 374.17\,\mathrm{m/s}$, $\rho_{ref} = 1\,\mathrm{kg/m^3}$. A case using Reynolds number of 1 and Mach number of 0.027 is tested. The domain of interest is $[-\pi\,\mathrm{m}, \pi\,\mathrm{m}] \times [-\pi\,\mathrm{m}, \pi\,\mathrm{m}]$, which is one period spatially. Dirichlet boundaries are applied



(a) DE

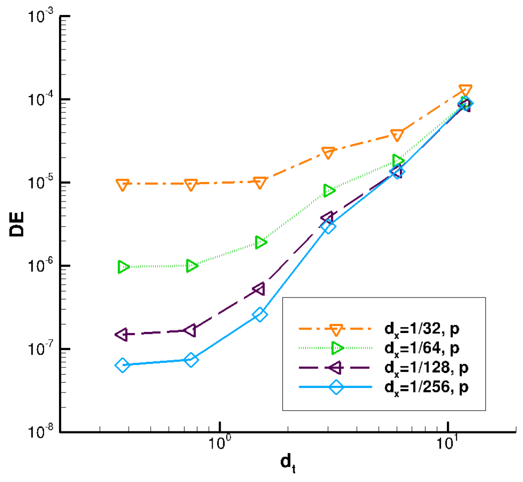


(b) OOA

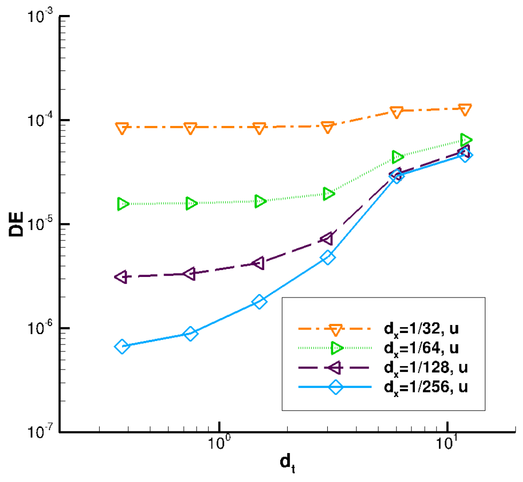
Figure 7.4: 2D Euler vortex flow ($M_\infty = 2.54$): three point backward

for the four faces. Note that the pressure term decays faster compared to the velocity terms, so attention may be needed when choosing the time step size.

Fig. 7.1 shows the separate order analysis results for the pressure and density ($Re = 1$). It can be seen that a good non-dimensional time step size on the 256×256 is around 0.7. However, still being subject to the stability constraints of the Newton's iterative method, a non-dimensional time step size of 0.375 is used. This implies that the temporal discretization error is smaller than the spatial discretization error but their difference is not large (still the same order). Similarly, when refining or coarsening systematically, time step sizes on other levels of meshes can be easily calculated.



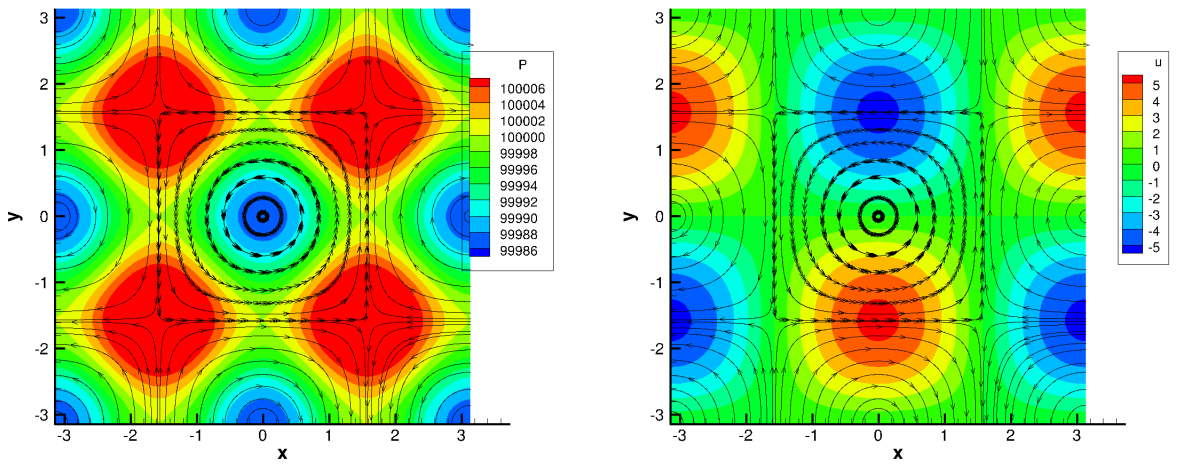
(a) Discretization analysis (p)



(b) Discretization analysis (u

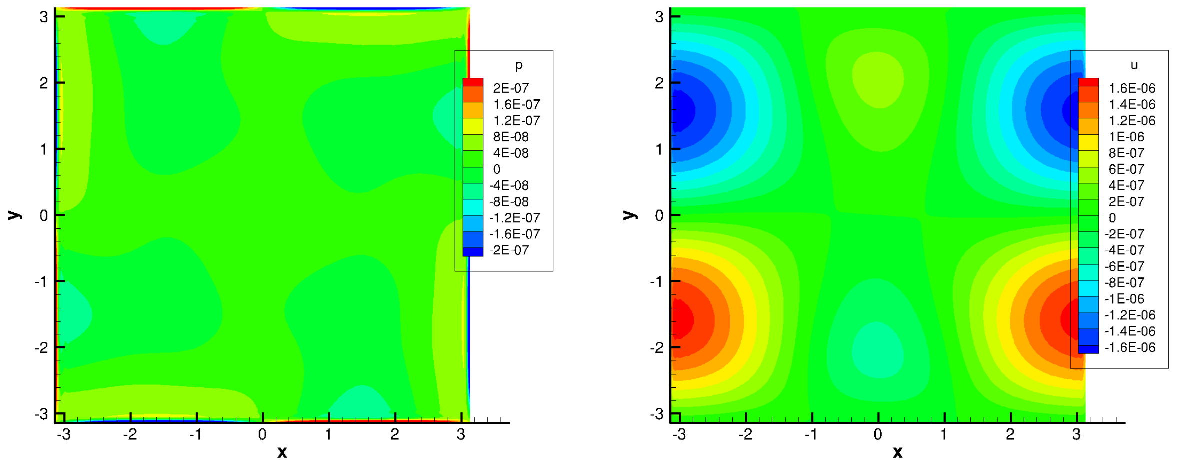
Figure 7.5: 2D Taylor Green vortex flow separate order analysis (Re=1)

Fig. 7.6 shows the dimensional solution contours solved at $t^* = 12$. The decaying factors for the pressure and u velocity component at $t^* = 12$ are 0.28 and 0.53, respectively, where t^* is the non-dimensional time (dimensional time t over l_{ref}/a_{ref}). Fig. 7.7 shows the non-dimensional error contours for the pressure and u velocity component at $t^* = 12$. For the pressure, as it damps out 72% of its initial amplitude, larger errors only exist near the boundaries. In contrast, the u velocity has larger regions with large errors. It should be noted that on the coarsest mesh (8×8), only 1 time step is used to iterate to the final time step, since there is enough viscosity ($Re = 1$) to guarantee a stable iterating process.



(a) pressure (b) u velocity

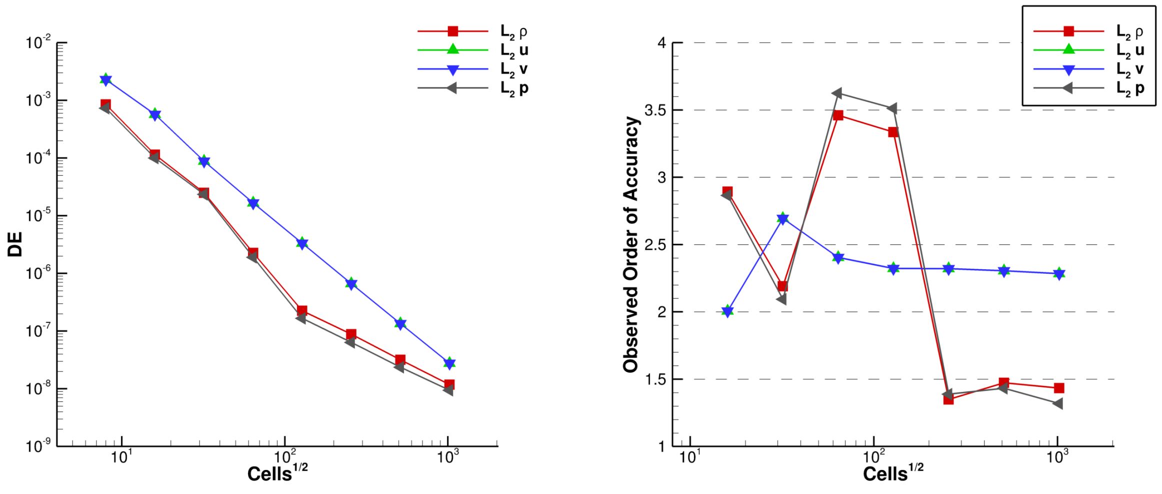
Figure 7.6: Solution contours of 2D Taylor Green decaying vortex



(a) pressure (b) u velocity

Figure 7.7: Error contours of 2D Taylor Green decaying vortex

Fig. 7.8 shows the OOA results of applying the implicit RK 2 scheme. It is surprising to see that the discretization errors drop faster than 2nd order on some intermediate levels of mesh, and it causes the OOA to reduce on fine meshes. The reason of this order reduction may be that the temporal discretization errors are still smaller than the spatial discretization errors on the meshes of 64×64 and 128×128, and then return to comparable order of magnitude when refining further. It can be seen from Fig. 7.8 in which the line connecting the 32×32 and 256×256 points are close to 2nd order slope. It is likely that this decaying vortex case is not a good test for the unsteady code verification purposes, as neither small time step size nor large time step size can be used. Small time step size may make the temporal discretization error to be much smaller compared to the spatial discretization error, which is not good for the code verification purposes, while large time step size may make the solution damp out most of its initial magnitude and at the same time make the Newton's iteration unstable.



(a) DE (b) OOA

Figure 7.8: 2D Taylor Green vortex flow ($Re = 1$): SDIRK 2

7.4.3 2D Cross-term Sinusoidal MMS

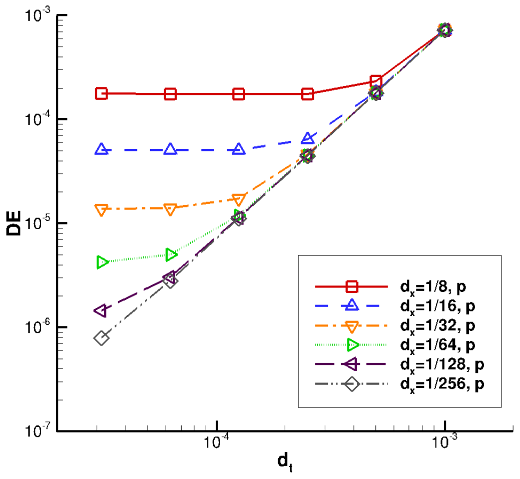
The last test case is a 2D cross-term sinusodial MMS. The form of a general unsteady MMS which is given in Eq. 7.16 will be used. It should be noted that this MMS can be Euler, laminar NS or turbulent.

$$\begin{aligned} f =& a_1 + a_2 sin(a_3\pi \frac{x}{l_{ref}} + a_4\pi) + a_5 sin(a_6\pi \frac{y}{l_{ref}} + a_7\pi) + \\ & a_8 sin(a_9\pi \frac{xy}{l_{ref}^2} + a_{10}\pi) + a_{11} sin(a_{12}\pi \frac{ta_{ref}}{l_{ref}} + a_{13}\pi), \end{aligned} \tag{7.16}$$

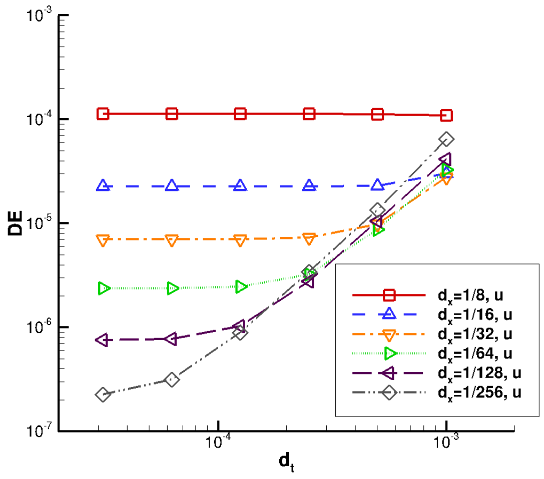
where a_i are coefficients (mean flow and turbulence equations use different coefficients). For this solution, $l_{ref} = 1\,\mathrm{m}$, $a_{ref} = 340\,\mathrm{m/s}$. This case has a Reynolds number of 500 and a Mach number of 0.15. We run all cases to a final time of 0.01 s so that we can have enough variations in time. In our tests, the coarsest mesh has 8 cells in each dimension of a curvilinear domain of about $[0, 1\,\mathrm{m}] \times [0, 1\,\mathrm{m}]$. We choose the coefficients a_{12} intentionally so that $a_{12}\pi\frac{ta_{ref}}{l_{ref}}$ is the same order as $a_3\pi\frac{x}{l_{ref}}$ or $a_4\pi\frac{y}{l_{ref}}$, or equivalently $a_{12}\pi\frac{\Delta ta_{ref}}{l_{ref}}$ is the same order as $a_3\pi\frac{\Delta x}{l_{ref}}$. Our choice ensures that the variations in space and time are similar. We will apply a separate order analysis to further show the discretization errors are similar in space and time.

As different primitive variables have different periodic frequencies (their largest difference may be 5 times, still less than an order of magnitude), different primitive variables have different magnitudes of temporal discretization errors. We need to find a proper time step size for all variables on a given mesh and then coarsen or refine to get time step sizes on other meshes. Fig. 7.9 shows a systematic separate order analysis for the pressure variable and u velocity component using the $k-\omega$ SST model. It can be seen that on the 256×256 mesh, a dimensional time step size of 3.125×10^{-5} s is reasonable to guarantee that the temporal discretization errors are of the same order of magnitude as the temporal discretization errors for both variables. Coarsening the mesh to the 8×8 level, we can obtain the coarsest time step size to be 0.001 s.

We tested Euler, laminar NS and turbulence cases. However, in this paper, only results for



(a) Discretization analysis (p)

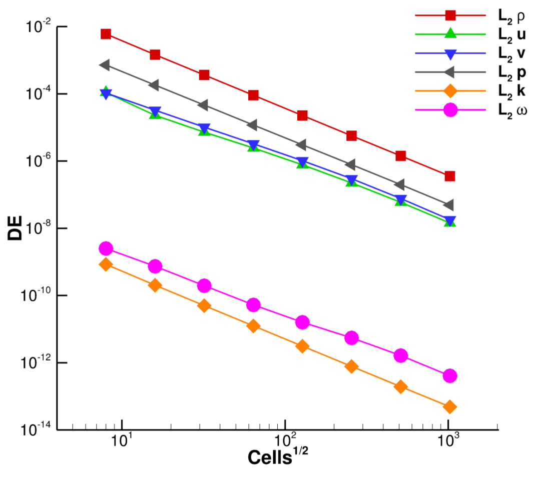


(b) Discretization analysis (u

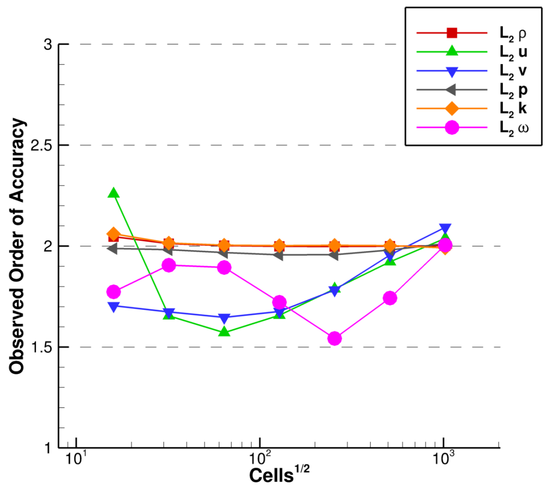
Figure 7.9: 2D CTS MMS separate order analysis

turbulence cases using the $k-\omega$ SST models and SA are shown for simplicity. Fig. 7.10 shows the OOA results using the $k-\omega$ SST model. After an enough refinement, the OOA is close to 2nd order for all the primitive variables. It is not surprising to see that the ω variable performs poorly on some intermediate meshes, but eventually turns to 2nd order accurate on finer meshes.

For the SA model, since the 1024×1024 mesh cannot be converged in the Newton's iteration when using the same time step size as the $k-\omega$ SST model, we reduced the time step sizes by 37.5% (now using 0.625 of the time step size for the $k-\omega$ SST model), which will make the temporal discretization errors smaller but still comparable to the spatial discretization errors. Fig. 7.11 shows the OOA results using the SA model. For all the variables, the OOA is close to 2nd order when sufficiently refined.

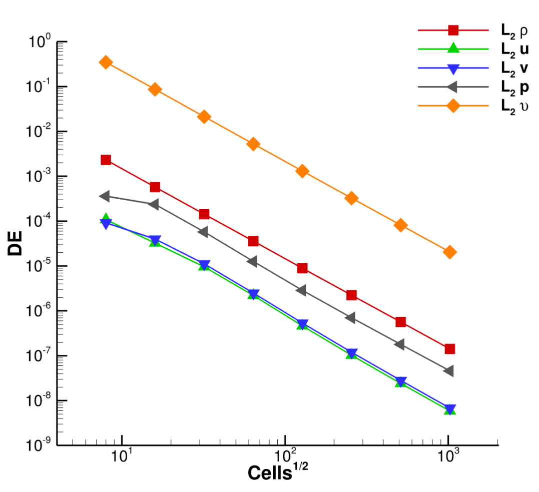


(a) DE

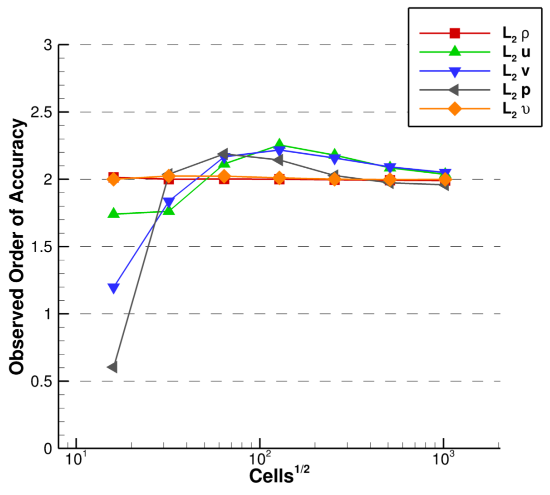


(b) OOA

Figure 7.10: 2D CTS MMS combined order analysis ($k - \omega$ SST)



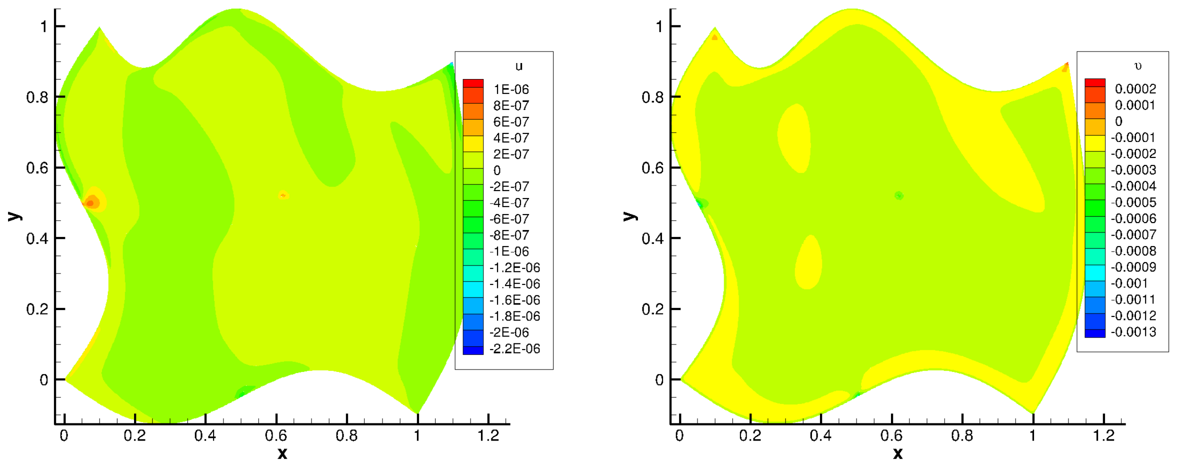
(a) DE



(b) OOA

Figure 7.11: 2D CTS MMS combined order analysis (SA)

Fig. 7.12 shows the non-dimensional error contours using the SA model at $t = 0.01\,\mathrm{s}$ with a prescribed $\Delta t = 1.953 \times 10^{-5}$ s on the 256×256 mesh. It can be seen that the errors are uniformly distributed in the whole domain for both the u velocity component and the turbulence variable.



(a) u velocity

(b) Turbulence variable ν

Figure 7.12: Error contours of 2D CTS MMS (SA)

7.5 Conclusions

Based on the results for the three test cases in this paper, we believe that both the temporal and spatial terms for the Navier-Stokes equations (laminar or turbulent) have been implemented correctly in SENSEI. When performing a systematic OOA test, special attention should be paid so that the temporal discretization errors are of the same order as the spatial discretization errors. Otherwise, some implementation errors for the temporal terms may be hidden because the temporal discretization errors are smaller for general cases. Even in some of our cases, the temporal time step size is still not very good for the purpose of code verification due to the stability issue of the implicit solver.

7.6 Future Work

In the future, this work will be extended to 3D, since this paper only covers 2D unsteady flow cases at present. Another potential work is to improve the newton's iteration so that larger time step size can be used. Finally, based on this code verification work, we also want to apply the error transport equation to accurately estimate the discretization error for cases which do not have exact solutions.

Acknowledgement

This material is partially based on research sponsored by the U.S. Air Force under agreement number FA865019-2-2204. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Bibliography

[1] William L Oberkampf and Christopher J Roy. *Verification and validation in scientific computing.* Cambridge University Press, 2010.

[2] William L Oberkampf and Timothy G Trucano. Verification and validation in computational fluid dynamics. *Progress in aerospace sciences*, 38(3):209–272, 2002.

[3] Abdul-Sattar J Al-Saif and Assma J Harfash. A new approximate analytical solutions for two-and three-dimensional unsteady viscous incompressible flows by using the kinetically reduced local navier-stokes equations. *Journal of Applied Mathematics*, 2019, 2019.

[4] Seth C Spiegel, HT Huynh, and James R DeBonis. A survey of the isentropic euler vortex problem using high-order methods. In *22nd AIAA Computational Fluid Dynamics Conference*, page 2444, 2015.

[5] WH Hui. Exact solutions of the unsteady two-dimensional navier-stokes equations. *Zeitschrift für angewandte Mathematik und Physik ZAMP*, 38(5):689–702, 1987.

[6] Abdullah Shah, Li Yuan, and Aftab Khan. Upwind compact finite difference scheme for time-accurate solution of the incompressible navier–stokes equations. *Applied Mathematics and Computation*, 215(9):3201–3213, 2010.

[7] Maurizio Tavelli and Michael Dumbser. A staggered space–time discontinuous galerkin method for the three-dimensional incompressible navier–stokes equations on unstructured tetrahedral meshes. *Journal of Computational Physics*, 319:294–323, 2016.

[8] Ronald L Panton. *Incompressible flow*. John Wiley & Sons, 2013.

[9] Frank M White and Isla Corfield. *Viscous fluid flow*, volume 3. McGraw-Hill New York, 2006.

[10] Henrik Tryggeson. *Analytical vortex solutions to Navier-Stokes equation*. PhD thesis, Växjö University Press, 2007.

[11] CY Wang. Exact solutions of the unsteady navier-stokes equations. *Applied Mechanics Reviews;(United States)*, 42(CONF-8901202–), 1989.

[12] Robert G Deissler. Unsteady viscous vortex with flow toward the center. 1965.

[13] Tapan K Sengupta, Nidhi Sharma, and Aditi Sengupta. Non-linear instability analysis of the two-dimensional navier-stokes equation: The taylor-green vortex problem. *Physics of Fluids*, 30(5):054105, 2018.

[14] Hongyu Wang, William C Tyson, and Christopher J Roy. Discretization error estimation for discontinuous galerkin methods using error transport equations. In *AIAA Scitech 2019 Forum*, page 2173, 2019.

[15] William C Tyson, Gary K Yan, Christopher J Roy, and Carl F Ollivier-Gooch. Relinearization of the error transport equations for arbitrarily high-order error estimates. *Journal of Computational Physics*, 397:108867, 2019.

[16] Geoffrey Ingram Taylor and Albert Edward Green. Mechanism of the production of small eddies from large ones. *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 158(895):499–521, 1937.

[17] James DeBonis. Solutions of the taylor-green vortex problem using high-resolution explicit finite difference methods. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, page 382, 2013.

[18] Zhijian J Wang, Krzysztof Fidkowski, Rémi Abgrall, Francesco Bassi, Doru Caraeni, Andrew Cary, Herman Deconinck, Ralf Hartmann, Koen Hillewaert, Hung T Huynh, et al. High-order cfd methods: current status and perspective. *International Journal for Numerical Methods in Fluids*, 72(8):811–845, 2013.

[19] Yuntian Bo, Peng Wang, Zhaoli Guo, and Lian-Ping Wang. Dugks simulations of three-dimensional taylor–green vortex flow and turbulent channel flow. *Computers & Fluids*, 155:9–21, 2017.

[20] Subrahmanya Pavan Kumar Veluri. *Code verification and numerical accuracy assessment for finite volume CFD codes.* PhD thesis, Virginia Tech, 2010.

[21] Kambiz Salari and Patrick Knupp. Code verification by the method of manufactured

solutions. Technical report, Sandia National Labs., Albuquerque, NM (US); Sandia National Labs ..., 2000.

[22] Chris Roy, Curt Ober, and Tom Smith. Verification of a compressible cfd code using the method of manufactured solutions. In *32nd AIAA Fluid Dynamics Conference and Exhibit*, page 3110, 2002.

[23] Stephane Etienne, Andre Garon, and Dominique Pelletier. Code verification for unsteady flow simulations with high order time-stepping schemes. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, page 169, 2009.

[24] Michael L Minion and RI Saye. Higher-order temporal integration for the incompressible navier–stokes equations in bounded domains. *Journal of Computational Physics*, 375:797–822, 2018.

[25] Kintak Raymond Yu, Stéphane Étienne, Alexander Hay, and Dominique Pelletier. Code verification for unsteady 3-d fluid–solid interaction problems. *Theoretical and Computational Fluid Dynamics*, 29(5-6):455–471, 2015.

[26] Weicheng Xue, Hongyu Wang, and Christopher J Roy. Code verification for 3d turbulence modeling in parallel sensei accelerated with mpi. In *AIAA Scitech 2020 Forum*, page 0347, 2020.

[27] Sherrie L Krist. *CFL3D user's manual (version 5.0).* National Aeronautics and Space Administration, Langley Research Center, 1998.

[28] Robert T Biedron, Jan Renee Carlson, Joseph M Derlaga, Peter A Gnoffo, Dana P Hammond, William T Jones, Bil Kleb, Elizabeth M Lee-Rausch, Eric J Nielsen, Michael A Park, et al. Fun3d manual: 13.6. 2019.

[29] Joseph M Derlaga, Tyrone Phillips, and Christopher J Roy. Sensei computational fluid dynamics code: a case study in modern fortran software development. In *21st AIAA Computational Fluid Dynamics Conference*, 2013.

[30] Charles W Jackson, William C Tyson, and Christopher J Roy. Turbulence model implementation and verification in the sensei cfd code. In *AIAA Scitech 2019 Forum*, 2019.

[31] Hongyu Wang, Weicheng Xue, and Christopher J Roy. Error transport equation implementation in the sensei cfd code. In *AIAA Scitech 2020 Forum*, page 1047, 2020.

[32] Chris Rumsey, Brian Smith, and George Huang. Description of a website resource for turbulence modeling verification and validation. In *40th Fluid Dynamics Conference and Exhibit*, 2010.

[33] Steven R Allmaras and Forrester T Johnson. Modifications and clarifications for the implementation of the spalart-allmaras turbulence model. In *Seventh international conference on computational fluid dynamics (ICCFD7)*, pages 1–11, 2012.

[34] Florian R Menter. Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal*, 32(8):1598–1605, 1994.

[35] Florian R Menter, Martin Kuntz, and Robin Langtry. Ten years of industrial experience with the sst turbulence model. *Turbulence, heat and mass transfer*, 4(1):625–632, 2003.

[36] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[37] Philip L Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics*, 43(2):357–372, 1981.

[38] Joseph L Steger and RF Warming. Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods. *Journal of computational physics*, 40(2):263–293, 1981.

[39] Bram Van Leer. Flux-vector splitting for the euler equation. In *Upwind and High-Resolution Schemes*, pages 80–89. Springer, 1997.

[40] Uri M Ascher, Steven J Ruuth, and Raymond J Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[41] Christopher A Kennedy and Mark H Carpenter. Diagonally implicit runge-kutta methods for ordinary differential equations. a review. 2016.

[42] Antony Jameson, Wolfgang Schmidt, and Eli Turkel. Numerical solution of the euler equations by finite volume methods using runge kutta time stepping schemes. In *14th fluid and plasma dynamics conference*, page 1259, 1981.

[43] JC Wu, LT Fan, and LE Erickson. Three-point backward finite-difference method for solving a system of mixed hyperbolic—parabolic partial differential equations. *Computers & chemical engineering*, 14(6):679–685, 1990.

[44] Luca Ferracina and MN Spijker. Strong stability of singly-diagonally-implicit runge–kutta methods. *Applied Numerical Mathematics*, 58(11):1675–1686, 2008.

[45] Shane A Richards. Completed richardson extrapolation in space and time. *Communications in numerical methods in engineering*, 13(7):573–582, 1997.

[46] James Kamm, William Rider, and Jerry Brock. Combined space and time convergence analysis of a compressible flow algorithm. In *16th AIAA Computational Fluid Dynamics Conference*, page 4241, 2003.

[47] Helen C Yee, Neil D Sandham, and M Jahed Djomehri. Low-dissipative high-order shock-capturing methods using characteristic-based filters. *Journal of computational physics*, 150(1):199–238, 1999.