Tenth International Conference on Computational Fluid Dynamics (ICCFD10), Barcelona,Spain, July 9-13, 2018

ICCFD10-2018-101

Performance Enhancements for the Lattice-Boltzmann Solver in the LAVA Framework

Michael Barad*, Joseph Kocheemoolayil**,
Gerrit Stich**, and Cetin Kiris*
Corresponding author: michael.f.barad@nasa.gov

* NASA Ames Research Center, CA, USA.
** Science and Technology, Moffett Field, CA, USA.

Abstract: Performance enhancements in NASA's recently developed Lattice Boltzmann solver within the Launch Ascent and Vehicle Aerodynamics (LAVA) framework are presented. Two key algorithmic developments are highlighted. A coarse-fine interface treatment that discretely conserves mass and momentum has been implemented and successfully verified and validated. Code optimizations targeting improved serial and parallel performance were presented. For a simple turbulent Taylor-Green Vortex problem, we were able to demonstrate a 2.3x speedup over the baseline code for a single Skylake-SP node containing 40 physical cores, and a 2.14x speedup for 64 nodes containing 2560 physical cores. In addition, we were able to show that the optimizations enabled us to scale the code almost perfectly to 20480 physical cores where, including ghost cells, the problem size was 10 billion cells.

Keywords: Lattice Boltzmann Method, Computational Fluid Dynamics, Aeroacoustics.

1 Introduction

Reynolds-averaged Navier-Stokes (RANS) simulations continue to have a tremendous impact on aerodynamic analysis. However, for massively separated flows, acoustics, and where scale resolving simulation is critical we have to utilize higher fidelity approaches such as Large Eddy Simulation (LES). For example, the community within AIAA's high-lift prediction workshop has shown that to predict lift at high angle of attack near stall accurately, LES methods perform significantly better than RANS[1]. Navier-Stokes (NS) based Computational Fluid Dynamics (CFD) is one way to achieve LES results, but it can be prohibitively expensive because of space-time resolution requirements. An alternative is the Lattice-Boltzmann Method (LBM)[2] which can yield accurate results where anisotropic meshing is not critical, and can dramatically reduce costs due to a) run-time, and b) mesh generation for complex geometries, as previously demonstrated for a full landing gear noise simulation [3]. To further improve this methodology, we have made significant advances in our capabilities, including coarse-fine interface handling, and performance optimizations.

In prior work, it was shown that for Navier-Stokes LES of the SOFIA aircraft[4], limits to strong scaling, using the same LAVA Cartesian AMR infrastructure, pointed towards a tiling/threading approach that will be described in §3.2.1. Specifically, it was found that scaling was limited by MPI communication which was exacerbated by poor surface/volume ratios. In that work the max box size (MBS) >= 32 was required for the largest simulations due to the memory requirements ballooning with smaller MBS.

While Navier-Stokes (NS) based simulations have been the bread and butter of CFD practitioners since the birth of the field, Lattice-Boltzmann (LB) based approaches have been gaining traction recently due to their attractive properties[5, 6, 2, 7]. The benefits of LBM include ultra high performance (in terms of number of cells updated per second), minimal numerical dissipation, and the ability to simulate arbitrarily complex geometry in a relatively straight-forward manner. High performance is due to significantly lower floating point operations relative to high-order NS algorithms, excellent data locality, ability to easily vectorize and

multithread the underlying operations on current and near-future hardware (despite the memory-bound nature of the algorithm), and excellent, platform-agnostic parallel scalability[8]. Minimal numerical dissipation is critical for computational aeroacoustics and ideal for large eddy simulations (LES)[3, 9].

The LBM is implemented within the Launch Ascent and Vehicle Aerodynamics (LAVA) framework[10]. This paper stands as an overview of recent LBM progress in LAVA. It includes critical details of our latest accuracy and performance improvements.

2 Baseline Methodology

Over the past two decades, the LBM has evolved into a mature technique for simulating engineering fluid flows of practical importance. The LBM is a mesoscopic approach wherein simplified kinetic equations that retain just enough detail to satisfy the desired macroscopic equations of fluid motion (weakly compressible, isothermal Navier-Stokes equations in the present context) are solved[5]. The local state of fluid motion is described by density distribution functions $f(\vec{x}, t, \vec{v})$, which upon being normalized by the local density represent the probability of finding particles moving with velocity $\vec{v}$ in an infinitesimal volume about $\vec{x}$. The familiar macroscopic variables such as density and the components of momentum are determined from the density distribution functions through moment summations.

In LAVA we have implemented a range of collision models, lattice types, boundary conditions, and supporting infrastructure including parallel IO, moving geometry, and sub-cycling adaptive mesh refinement algorithms. In doing so, much code re-use has been made possible within the LAVA Cartesian module[10].

3 Improvements

The paper focuses on our most recent optimizations to the code. We discuss progress towards accelerating and scaling the code on 1000's of processors by adding more levels of parallelism. For example intra-box tiling and other outer-loop decomposition techniques, combined with a hybrid MPI-OpenMP threading paradigm designed to target the increased outer-loop concurrency is assessed. But first, we present our progress on improving our local refinement capability, which is a critical enabling technology for extremely under-resolved engineering calculations.

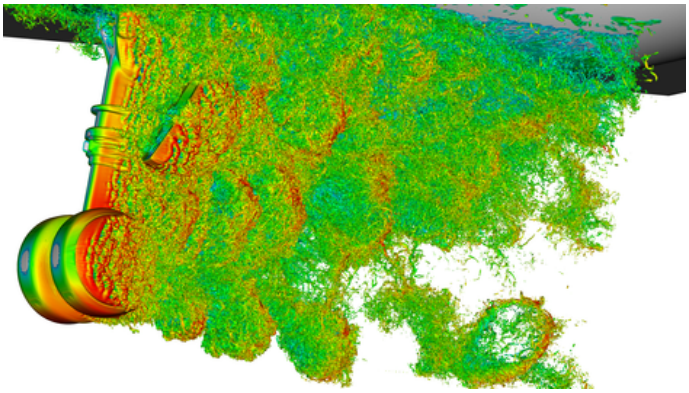
3.1 Discrete Conservation of Mass and Momentum at Coarse-Fine Interfaces

Our initial implementation treated the coarse-fine interface using a non-conservative second order accurate interpolation approach. While this approach proved adequate for several engineering applications, such as the noise generated by a landing gear (see Figure 1 from Barad *et al.* [3]), more careful testing pointed out several deficiencies. For example, upon simulating the NASA Wall Mounted Hump problem [11, 12] we noticed spurious vorticity being generated at the interface. In an attempt to eliminate this shortcoming, our first attempt was to increase the order of accuracy of the inter-level operators from second to fourth order.

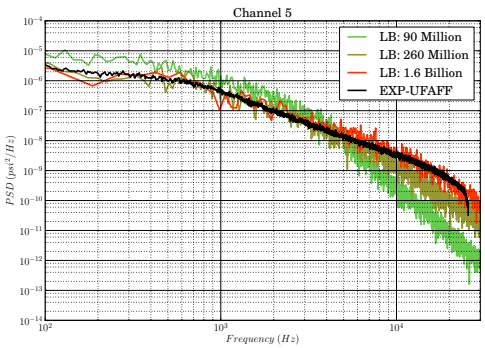
While improvements were made with this approach, the generation of spurious vorticity at the coarse-fine interface was not eliminated entirely. Following the work of [8, 13, 14] we implemented a conservative coarse-fine interface. While the algorithm is conceptually quite simple, it involves careful bookkeeping of incoming and outgoing density distribution functions at the coarse-fine interface in order to preserve mass and momentum to machine precision. The implementation is done in a recursively sub-cycled block-structured, adaptive high-performance parallel programming paradigm.

Pseudo-code for the sub-cycling algorithm chosen is shown in Algorithm 1-2. In Algorithm 1 we mention two permutations of the algorithm: Stream-Coalesce-Collide (SCC) and Collide-Stream-Coalesce (CSC). We currently utilize the SCC version of the algorithm in our solver. The CSC approach has the advantage that the collide and stream operations can be fused into a single operation, e.g. see Schornbaum[8]. For this work we focus our efforts on the SCC algorithm, where our implementation is shown to be conservative with multiple refinement levels.

Seamless restarting from a previous checkpoint was recently implemented. The restart files are only written when advancing the finest level (i.e. $l = l_{max}$). When restarting the code, indexing is kept that indicates where in the recursive sub-cycling the code was when writing the restart file and this information is stored



(a)



(b)

Figure 1: LAVA LBM Landing Gear simulation showing: (a) vorticity colored by Mach Number, (b) grid-refinement study of power spectral density of sound on the upper drag-link compared to experiment[3].

in the file. After reading the restart file, the code skips through the recursive sub-cycling algorithm without changing the state until it reaches the matching index so that identical processing continues thereafter.

Algorithm 1 : `SubCycle(`l`)` Recursive Sub-Cycling Algorithm. Level is l, l_{max} is the finest level, $l = 0$ is coarsest. Density distributions f, where f^{post} is the post-collided state. SCC indicates the Stream-Coalesce-Collide version of the code, otherwise CSC which is Collide-Stream-Coalesce. `Advance()` is defined in Algorithm 2.

for $isub = 0$ to 1 **do**
  if SCC **then**
    `Explosion(`$f_l^{post} \rightarrow f_{l+1}^{post}$`)`
  else
    `Explosion(`$f_l \rightarrow f_{l+1}$`)`
  end if

  if $l < l_{max}$ **then**
    `SubCycle(`$l+1$`)`
  end if

  `Advance(`l`)`
  if $l == 0$ **then**
    `break`
  end if
end for

3.2 Performance Improvements

In an effort to improve performance of the solver, both single-node and multi-node optimizations have been undertaken. Specifically, two complementary approaches have been implemented: 1) tiling and 2) OpenMP threading. These two approaches are complementary as will be shown.

3.2.1 Another Level Of Parallelism: Tiling

The tiling approach implemented fits naturally within the existing block-structured adaptive mesh refinement (AMR) data-structures in LAVA. The LAVA Cartesian framework uses the Chombo library for AMR data-structures leveraging many years of Department of Energy software development[15]. Tiling has been used

Algorithm 2 : `Advance(`l`)` Single level, single time-step advancement. Level is l, l_{max} is the finest level, $l = 0$ is coarsest. Density distributions f, where f^{post} is the post-collided state. SCC indicates the Stream-Coalesce-Collide version of the code, otherwise CSC which is Collide-Stream-Coalesce. Operators are defined in text.

if SCC **then**
  `Streaming(`$f_l^{post} \rightarrow f_l$`)`
  if $l < l_{max}$ **then**
    `Coalescence(`$f_{l+1} \rightarrow f_l$`)`
  end if
  `Collision(`$f_l \rightarrow f_l^{post}$`)`
else
  `CollisionAndStream(`$f_l \rightarrow f_l^{post} \rightarrow f_l$`)`
  if $l < l_{max}$ **then**
    `Coalescence(`$f_{l+1} \rightarrow f_l$`)`
  end if
end if

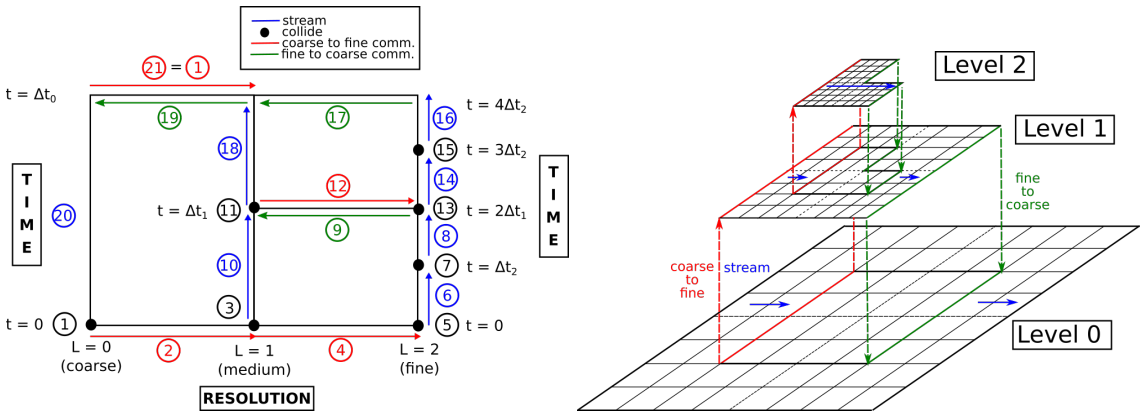
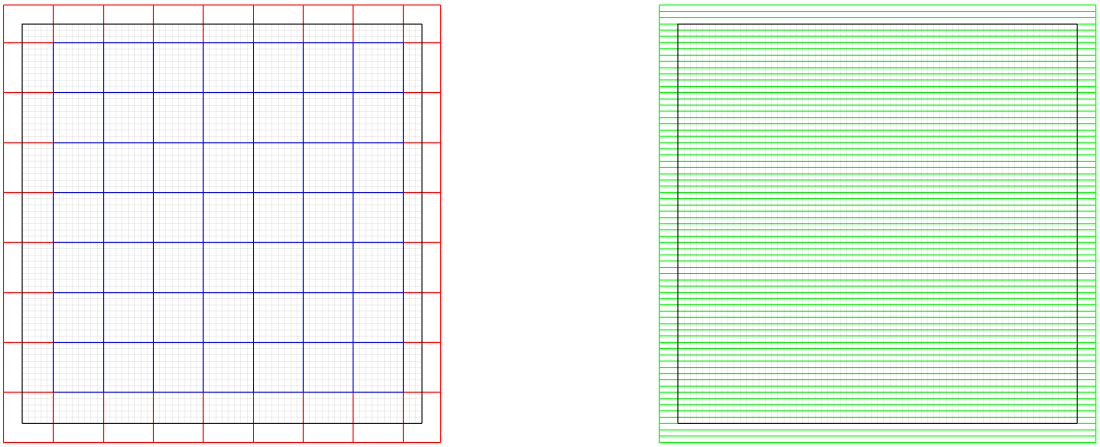


Figure 2: (left) Sketch of recursive sub-cycling algorithm described in Algorithm 1. (right) Block structured AMR showing 3 levels of refinement by factor 2. Arrows indicate direction of information propagation: streaming (blue), coarse-to-fine communication (red), fine-to-coarse communication (green).

extensively as an effective cache-blocking tool for decades, but here we leverage it for both cache-blocking, and as a mechanism to insert an additional level of parallelism into the AMR hierarchy: level→box→tile, cell. This is critical for effectively utilizing modern massively parallel many-core hardware.

The entire flow domain is partitioned into factors of 2 refinement levels (l), i.e. $\Delta x_l = 2\Delta x_{l+1}$; each level is partitioned into a lattice of cells that are grouped into cube-shaped boxes, each box is owned by a single MPI rank (multiple boxes per MPI rank is allowed); each box is partitioned into tiles; each tile contains a set of cells that are indexed with a global (i,j,k) index space on each level. For the tiling approach taken here, we constrained the boxes to have fixed sizes in all directions, where each box has N^D cells. N is a user selected number of cells (i.e. 16, 32, 128, etc), and D is the dimension simulated. On each refinement level the boxes form a lattice that can be globally indexed. Each box is thus the same size which simplifies the tiling implementation process and has beneficial load balancing properties due to the uniformity.

We decompose each box (including 3 ghost cells on each side) into a disjoint collection of tiles. We have two tiling schemes, as shown in Figure 3: 1) for loops with stencils accessing neighboring data, e.g. streaming, we tile into 3D "regular" cubic or near cubic shaped tiles, 2) for loops without neighbor data access, e.g. the collision operator, we tile into 1D "pencils" where all pencils are aligned with the unit-stride data storage direction (which for us is the x-direction). The regular tiles are divided into "inner" tiles, and "outer" tiles, where the union of inner and outer tiles completely paves each box plus 3 ghost cells on each side. Other tile shapes could easily be exploited, targeting enhanced cache use by elongating more in x- and y-directions as compared to z-direction, this was not explored in this work. We intend to have the code auto-tune to the optimal tile shape (either as it runs, or as a pre-processing step for each processor type). Tile templates for inner, outer, and pencils are setup at start-up since all boxes are the same. These templates are used, as appropriate depending on the stencil operator, by translating them to the box being worked on in the level based (i,j,k) index space.



(a) Regular Tiles (b) Pencil Tiles

Figure 3: Different tile types for a single box: (a) regular tiles with MBS=8, including inner (blue) and outer (red); and (b) pencil tiles (green). Also shown are a typical box (black), and the individual (non-ghost) cells (gray). The box shown has 64^2 cells, plus 3 ghost layers. 3D tiles are conceptually similar.

This new layer of parallelism is exploited for both memory access and for more efficient compute node utilization. This also aligns with our dependency on the threaded Embree library for moving geometry[16]. In this tiling paradigm, each compute node is assigned a set of MPI ranks, fewer than the number of cores utilized, and each MPI rank works on a set of boxes and the tiles within each box. Two approaches were taken: 1) copy tile sized data from box sized arrays to tile sized (thread) local scratch arrays and work in the scratch space, referred to as "With Copy" for the remainder of this paper; and 2) work directly on the box sized arrays, but only work on tile sized index-space chunks, referred to as "Without Copy" for the remainder

of this paper.

3.2.2 Threading with OpenMP

The industry standard threading interface OpenMP was utilized to exploit the extra level of parallelism within each MPI rank exposed by the tiling decomposition. This choice fits well with the existing OpenMP threading in Chombo, and the threading model used in the Embree ray-tracing library which are used for ultra high-performance geometry queries (critical for moving geometry immersed boundary cases).

Typical threading work flow is as follows: On each MPI rank, loops over boxes and tiles are collapsed into single loops that are threaded using `#pragma omp for collapse(2)`. The collapsed loops are scheduled dynamically by OpenMP, which acts to balance load irregularities, such as immersed boundaries.

3.2.3 Asynchronous Communications

Since the streaming step is non-local, i.e. data is transferred to/from neighboring cells, ghost-cells need updating on neighboring blocks. With the tiling approach given in §3.2.1 we can complete the streaming step on the outer tiles that are involved with communication, begin an asynchronous MPI send of the ghost cell data, continue streaming the interior tiles, then finish receiving the outer tile data on the neighboring blocks. For the CSC version of the code, more work can be packed onto each tile during the asynchronous phase, theoretically yielding better overlap of computation with communication.

4 Results

4.1 Coarse-Fine Interface: Verification of Mass and Momentum Conservation

To test the conservation properties of the scheme, a simple verification test was performed: the Taylor-Green vortex in 2D and 3D. A sufficiently complicated 3 level mesh topology was prescribed, as shown in Figure 4. This topology has all the coarse-fine interface cases that we expect to encounter, including convex, concave, straight, and periodic. The simulation was run for 50 steps and integrated mass, and components of momentum were computed. As can be seen from Tables 1-2, conservation is exact up to double precision. Note that the momentum sums are not exactly zero due to the asymmetric grids, which were chosen so errors don't cancel and hide conservation issues.

Step	Mass	X-Momentum	Y-Momentum
0	3.4842903112786844e-03	-1.3321944431884503e-07	1.3321944427329724e-07
50	3.4842903112787802e-03	-1.3321944436875571e-07	1.3321944451058669e-07
Error:	**9.58434720477186e-17**	**4.99106752631308e-17**	**2.37289450970155e-16**

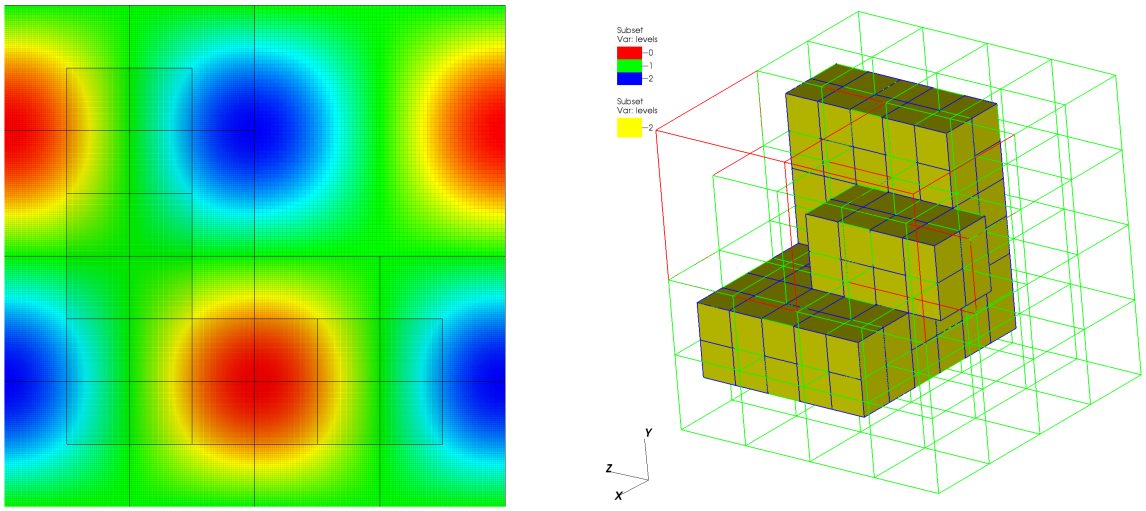
Table 1: Integrated mass and momentum for 2D Taylor-Green. Last row shows conservation error after 50 steps. Differences are highlighted with bold.

Step	Mass	X-Momentum	Y-Momentum	Z-Momentum
0	3.4842903112065815e-03	8.9970536**2799347**e-08	-8.9970536**2994044**e-08	**1.47753463427836e-20**
50	3.4842903112065789e-03	8.9970536**3602753**e-08	-8.9970536**2695844**e-08	**-3.39618771662516e-19**
Error:	**2.602085213965e-18**	**8.034059859103e-17**	**2.982000666632e-17**	**3.543941180053e-19**

Table 2: Integrated mass and momentum for 3D Taylor-Green. Last row shows conservation error after 50 steps. Differences are highlighted with bold.

4.2 Coarse-Fine Interface: Validation with NASA's Wall-Mounted Hump

As part of our verification and validation process, simulations on NASA's wall-mounted hump [11] were conducted. Conditions were Reynolds' number 936,000 based on the hump mean chord length (0.42m),



(a) 2D (b) 3D

Figure 4: Three level grid refinement hierarchy used for testing mass and momentum conservation. (a) 2D test, black lines show boxes. Colors indicate Taylor-Green velocity levels after 50 steps. (b) 3D test, lines show boxes on different levels (l), where red is $l = 0$, green is $l = 1$, and blue is $l = 2$ which is also shown with solid yellow boxes.

with a free-stream Mach number of 0.1. The setup describes a low-speed flow separation over a Glauert-Goldschmied type body as seen in Figure 5.

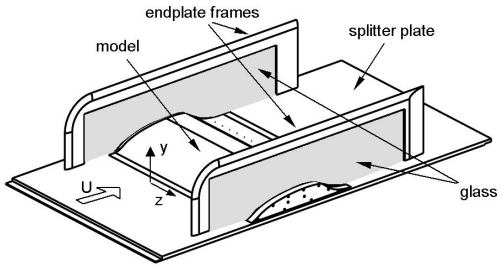


Figure 5: Experimental setup of NASA's wall-mounted hump by Greenblatt [11].

The LBM simulation uses five conservative AMR levels with the finest resolution in viscous wall units not exceeding 50 to solve the problem. Figure 6 and Figure 7 shows results from the hump simulation as compared to other solvers [17, 18]. The velocity and Reynolds' stress profiles show an excellent agreement. Detailed reporting on this test case using Lattice-Boltzmann as well as Navier-Stokes methods within the LAVA framework can be found in Stich *et al.* [12].

4.3 Performance Testing

To assess performance of the baseline and optimized code, we target an extremely simple canonical case: the Taylor-Green Vortex (TGV). This case was chosen so as to understand the impact of optimizations in an ideal setting without the complexity and irregularity of geometry, grid adaptation, or far-field boundary conditions. However, note that the optimizations implemented here are designed to significantly improve performance for arbitrarily complex configurations, in particular due to the dynamic load balancing within the extra level of parallelism that is exposed via tiling/threading.

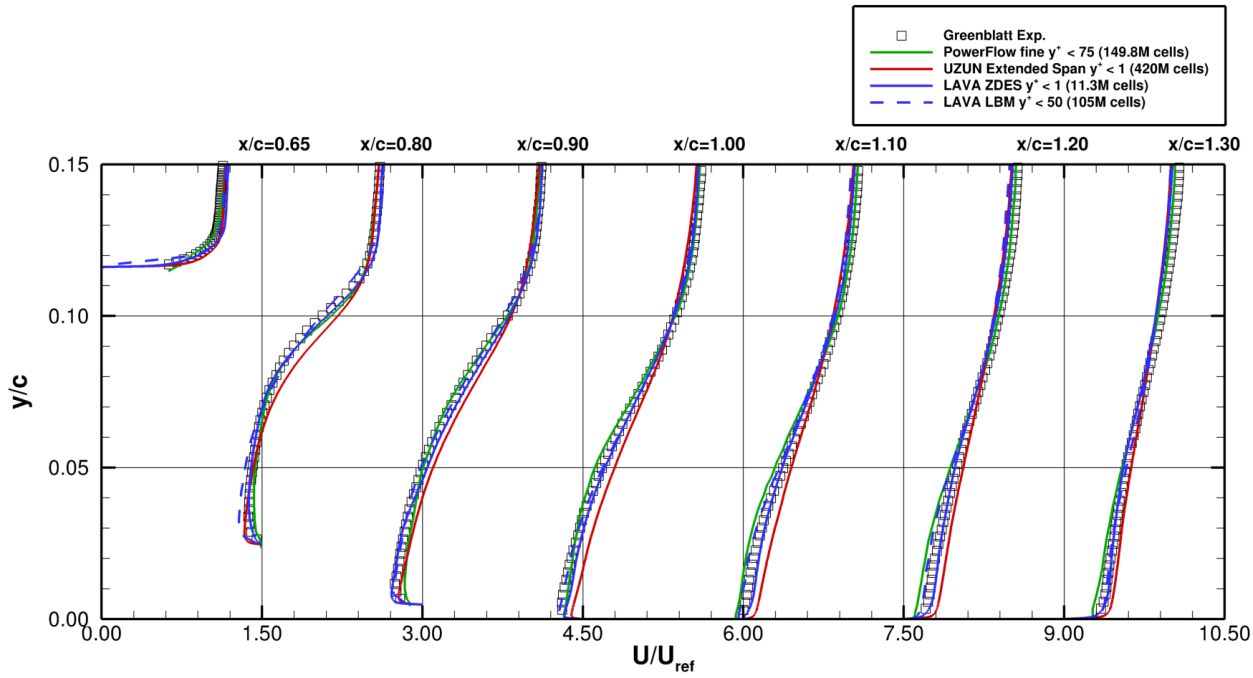


Figure 6: Stream-wise normalized velocity at different stations downstream of the separation location compared with experiments [11] and results from PowerFlow [17] and wall-resolved LES from Uzun and Malik [18].

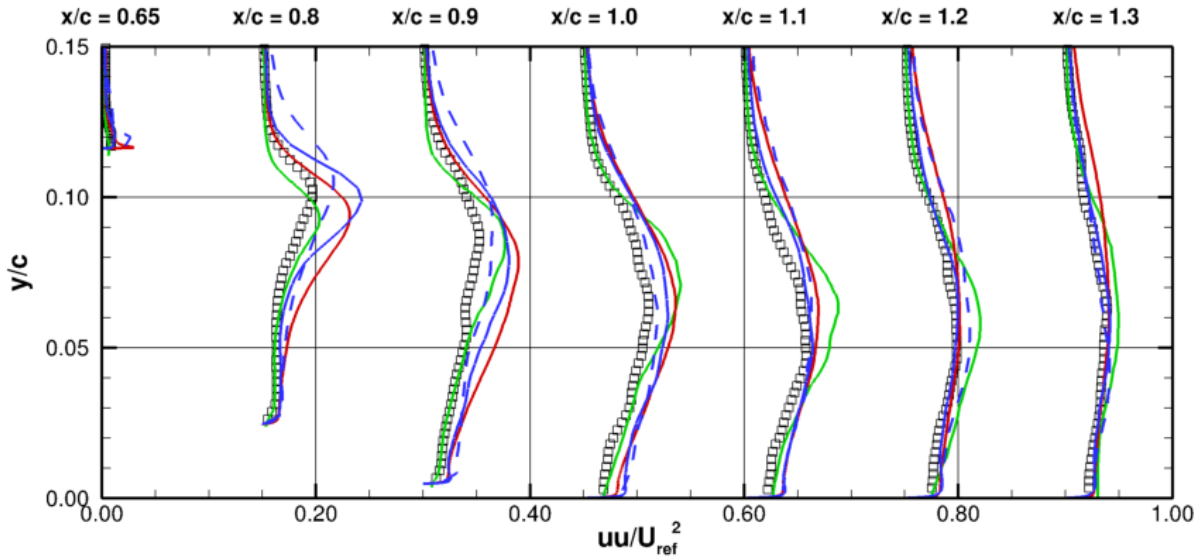


Figure 7: $u'u'/U_{ref}$ Reynolds' stress (b) at different stations downstream of the separation location compared with experiments [11] and results from PowerFlow [17] and wall-resolved LES from Uzun and Malik [18].

Here we focused on keeping the cost per computational node uniform and fixed at 256^3. The time spent in advancing the solution by 64 computational steps is reported, excluding start up costs and parallel IO. The time reported, called time-to-solution hereafter, does include all MPI communication, and LBM kernels (e.g. collide, stream, etc).

The performance experiments were conducted using the Pleiades supercomputer at NASA Ames Research Center, where we targeted Intel Xeon Gold 6148 (Skylake-SP) processors. In this system, each Skylake-SP

node consists of 2 sockets with 20-cores each, running at 2.4 GHz. There are two sub-NUMA clusters in each socket, creating two localization domains. Each domain contains one memory controller and ten L3 cache slices. The on-socket memory hierarchy consists of 64KB L1 cache per core, 1MB L2 cache per core, 24.5MB L3 cache per socket. Each socket also has 96GB DDR4 main memory where the bandwidth is 128GB/s, and the socket-socket interconnect bandwidth is at 41.6GB/s.

Three basic profiling analyses were performed:

- single packed-node parameters study;
- single-node strong scaling study; and
- multi-node weak scaling study.

These studies were performed for both the baseline and optimized code to enable a fair assessment of the improvements. For these profiling analysis we assessed sensitivity to:

- nodes = [1,8,64,512],
- MPI = [1,2,4,8,12,16,20,24,28,32,36,40,50,60,70,80] ranks/node,
- OMP = [0,1,2,4,8,12,16,20,24,28,32,36,40,50,60,70,80] threads,
- max box size (MBS) = [16,32,64,128,256] cells per direction,
- max tile size (MTS) = [0,4,6,8,10,12,16,32,64,128] cells per direction,
- hyperthreading [no/yes],

where, Nodes is the number of dual-socket Skylake compute nodes used (1 for all studies but the weak-scaling); MBS is the number of cells per direction used to partition the problem domain; MTS is the number of cells per direction used for each for the regular tiles (0 for no tiling); MPI is the number of MPI ranks used; OMP is the number of OpenMP threads used; and where hyperthreading is running more than 1 thread per core. With MBS=128 and MTS=12, the tiling algorithm yields 729 inner tiles, 999 outer tiles, and 17956 pencil tiles, which is a significant amount of parallelism per box.

It should be noted that this high-dimensional parameter space, with 108800 cases, was significantly pruned (down to several hundred) by excluding cases where there is not enough-work per thread (MPI and OMP). Also for packed-node cases we enforce MPI=[40,80] or MPI*OMP=[40,80], and we always run the code with at least 1 MPI rank. Results presented were also pruned to exclude excessively slow results, as most of the plots already contain a great deal of information.

The single-packed-node parameter study was designed to test sensitivities and the interplay between MBS/MTS/MPI/OMP. The first task here was to understand the baseline performance sensitivity to MBS and hyperthreading. This is shown in Figure 8, where larger MBS and hyperthreading improves performance. Since the problem is 256^3, partitioning it into MBS of 64 yields 64 boxes which was the best single-node baseline partitioning attempted, yielding the best performance at 57.13 seconds. The plot shows a trend of larger MBS resulting in superior performance. This trend simply highlights the overhead of box-box MPI ghost cell communication, which grows with smaller box size due to the box surface/volume ratio degradation.

The purpose of the single-packed-node parameter study is to identify sweet spots and trends for a fully subscribed Skylake-SP node. First it must be noted that minor vectorization optimizations were performed on the baseline code, speeding up from baseline to 51.68 seconds (from 57.13), an almost 1.11x speedup without tiling/threading.

For the optimized code, Figure 9 shows that the "With Copy" approach (a) is clearly inferior to the compute in place "Without Copy" approach (b) for all permutations of the parameter space, highlighting the excessive memory bandwidth required for the "With Copy" approach. Function level profiling of the code confirmed this, showing that the "With Copy" approach was dominated by the copy to scratch memory, while the "Without Copy" technique, was able to speedup the code significantly via improved memory usage. It should be noted that the "Without Copy" still accesses the same memory in the streaming and collision operations, but is able to due so in a more streamlined way without the brute force copy. Inspecting

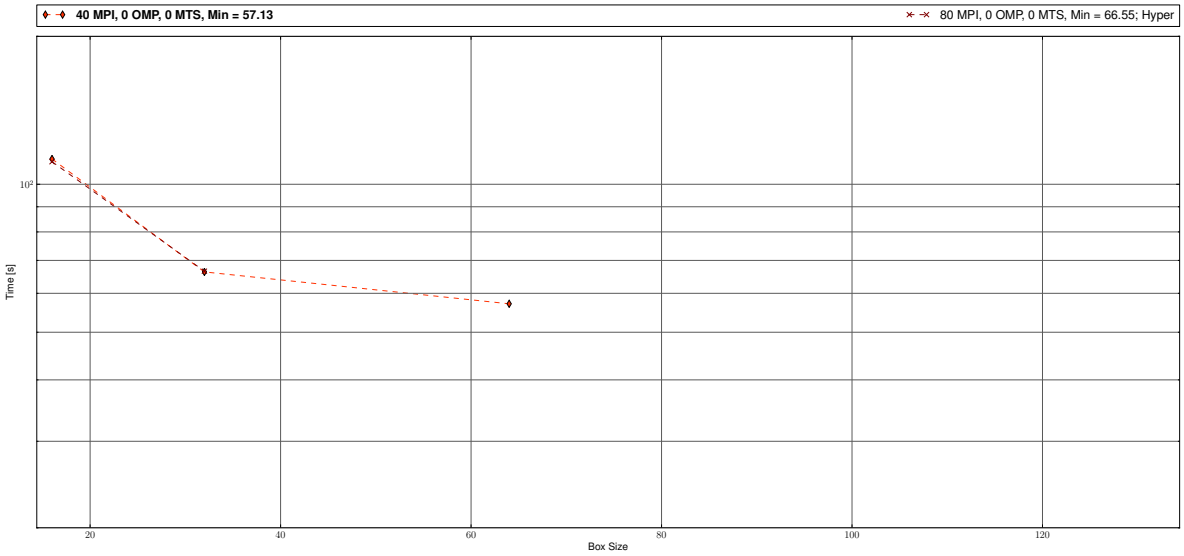


Figure 8: Single node performance profiles showing the **baseline** performance vs max box size, without tiling, or OpenMP optimizations. Legend indicates number of MPI ranks (MPI), number of OpenMP threads (OMP), max box size (MBS), and max tile size (MTS) used, as well as minimum time-to-solution for each line. Dashed lines are the no-threading (OMP=0) cases, and no-tiling is indicated by MTS=0. Hyperthreading is labeled Hyper.

Figure 9b, one can see that, given the parameter space explored, the sweet spot is MPI=2, OMP=40, MBS=128, MTS=12, and with hyperthreading, yielded a time-to-solution of 24.84 seconds, or a 2.3x single-node speedup compared to baseline. This same data is also presented in Figure 10, where the sensitivity to MTS is shown. For the "With Copy" case there is a clear minimum time-to-solution at MTS=8 likely due to the data fitting in cache. For the "Without Copy" case, the minimum time-to-solution is produced with larger tile sizes in the 12-16 range. These plots illustrate that the extra level of parallelism due to tiling combined with OpenMP threading is a clear win. This is due to the reduction in MPI communication, improved cache usage, and improved load balance. A single-node strong scaling study for the Hybrid MPI/OMP optimized code is shown in Figure 11. In this study we investigate the scaling characteristics on a single-node with the best case MBS=128 and MTS=12 parameters. The plot shows total number of threads (i.e. MPI*OMP) vs time-to-solution, and also shows the ideal linear (in log-log space) scaling. Four MPI sweeps were conducted, with MPI=[1,2,4,8]. It is clear from this plot that the MPI=2 case has nearly perfect linear scaling until the hyperthread zone (shaded gray) is reached. Above 40 threads, the performance nearly stalls, with a jump from 28.70 to 24.84, or 1.16x speedup due to hyperthreading. It should be noted (as pointed out by a reviewer) that the stream benchmark[19], when run on Skylake-SP, shows that a purely memory bandwidth bound code can achieve only an 18x speedup. This indicates that the optimized LBM code, when run with the correct parameters on a packed Skylake-SP is not entirely limited by main memory bandwidth.

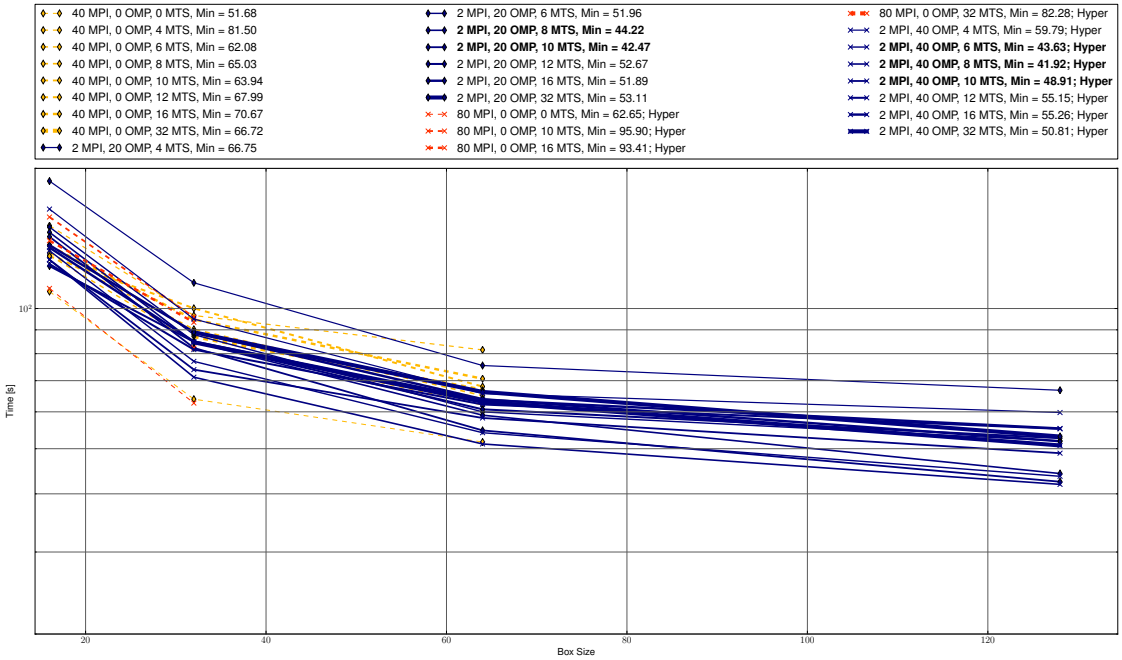
The third profiling study that was conducted was a multi-node weak scaling investigation. For this, the idea is to keep the work per node constant, while increasing the amount of work. This was achieved perfectly with the TGV problem setup by simply scaling up the problem size P=[256^3,512^3,1024^3,2048^3] while at the same time scaling up the node count, Nodes=[1,8,64,512], which keeps the workload fixed at 256^3 cells per node. Ideal weak scaling is therefore given by a constant time-to-solution. Weak scaling tests are notorious for exposing imperfections in algorithms and networks, where parallel scaling can suffer due to loops that don't scale linearly with node count (i.e. n^2 etc), and inefficient use of non-uniform memory access (NUMA) and inter-node networks. Results for this weak scaling study for the baseline code is shown in Figure 12a and optimized is shown in Figure 12b. The optimized data is also shown in Table 4.3. The data shows that small MBS performs much worse than larger MBS. This can be seen in the increase of time-to-solution at higher node counts where MPI/network communication bottlenecks are exposed. Thus, the motivation to improve the surface/volume ratio of boxes by utilizing larger MBS is highlighted at large concurrency (512

OMP	MBS	MTS	HT	Nodes=1 N=256 [s]	Nodes=8 N=512 [s]	Nodes=64 N=1024 [s]	Nodes=512 N=2048 [s]
0	16	12	No	89.27	89.39	113.08	-
0	32	0	No	60.77	62.01	78.59	99.97
0	32	12	No	51.92	50.90	64.08	66.86
0	64	0	No	48.37	53.36	80.48	124.03
0	64	12	No	37.84	39.81	43.95	70.79
40	32	12	No	79.67	94.66	95.73	101.52
40	64	12	No	52.91	62.54	63.57	75.60
40	128	12	No	43.27	52.64	52.92	56.85
0	16	12	Yes	85.85	83.32	103.67	-
0	32	0	Yes	79.74	61.48	72.80	93.38
0	32	12	Yes	53.96	47.63	52.25	61.79
40	128	12	Yes	**24.84**	**31.77**	**32.54**	**34.92**

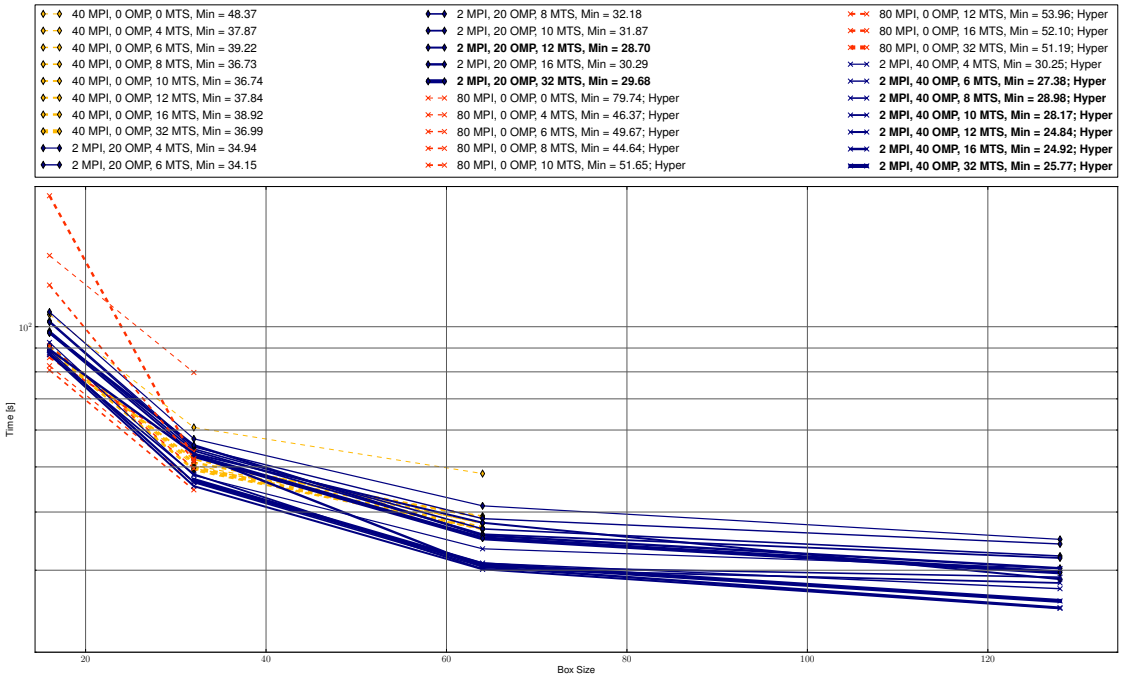
Table 3: Weak scaling performance profiles for the optimized code, units are seconds taken for 64 timesteps, where OMP indicates how many OpenMP threads per MPI rank, MBS is max box size, MTS is max tile size, HT is if hyperthreading was used, N is the problem size where cell count = N^3.

nodes = 20480 physical cores). The data shows that the optimized code, with MPI/OMP/tiling, maintains superiority over the flat-MPI and/or non-tiled versions.

Ideal scaling is nearly achieved for the optimized code with OMP=40, MBS=128, MTS=12, and hyper-threading turned on. It is not expected for the code to have perfect scaling due to NUMA effects, because there is a degradation of network bandwidth for more distant node communication. Single MPI rank per node results are not shown to elliminate NUMA issues. The Pleiades Skylake nodes used for this study have network bandwidths as follows: in-socket is 128 GB/s, on-node is 41.6 GB/s, inter-node is only 12.5 GB/s. This irregularity favors smaller node counts when bandwidth is important, as can be seen for the fastest case in Figure 12b when the single-node calculation (network@41.6GB/s) goes from 24.84 to 31.77 seconds for the 8 node case (network@12.5GB/s). Unfortunately, the Pleiades system has only 1152 Skylake nodes, otherwise we would have checked the N=4096 and larger cases.

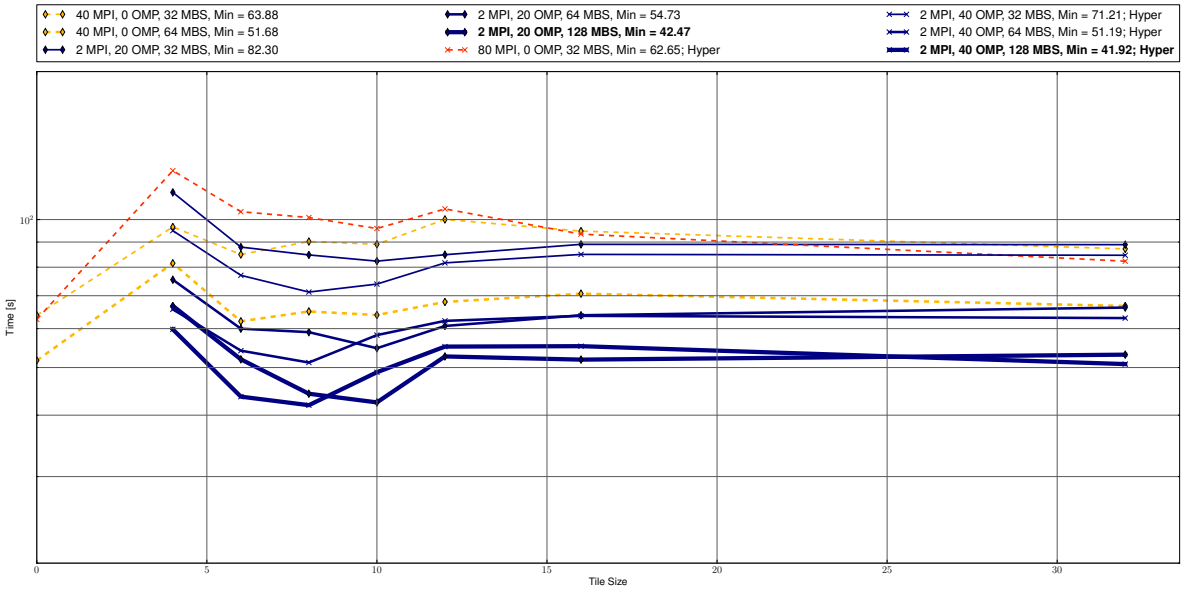


(a) With Copy

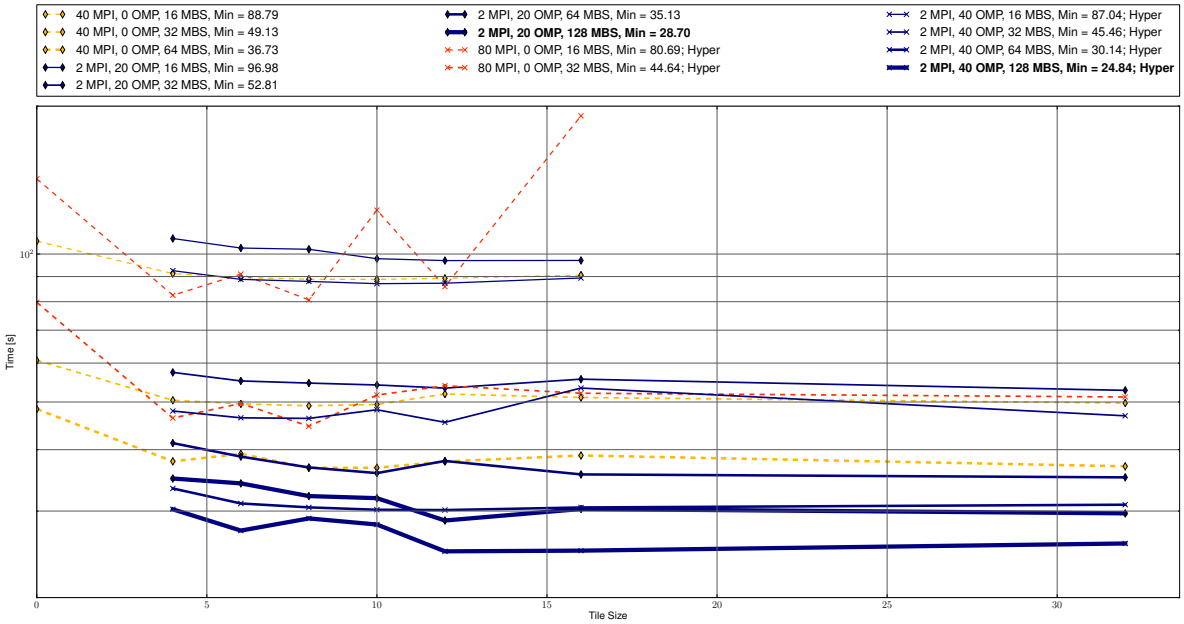


(b) Without Copy

Figure 9: Single node performance profiles exploring the sensitivity to max box size: (a) with copy to tiles and (b) without copy (i.e. direct memory accesses). Legend indicates number of MPI ranks (MPI), number of OpenMP threads (OMP), and max tile size (MTS) used, as well as minimum time-to-solution for each line. Dashed lines are the no-threading (OMP=0) cases, and no-tiling is indicated by MTS=0. Hyperthreading is labeled Hyper. Only showing significant results.



(a) With Copy



(b) Without Copy

Figure 10: Single node performance profiles exploring the sensitivity to max tile size: (a) with copy to tiles and (b) without copy (i.e. direct memory accesses). Legend indicates number of MPI ranks (MPI), number of OpenMP threads (OMP), and max box size (MBS) used, as well as minimum time-to-solution for each line. Dashed lines are the no-threading (OMP=0) cases, and no-tiling is Tile Size = 0 (i.e. MTS=0). Hyperthreading is labeled Hyper.

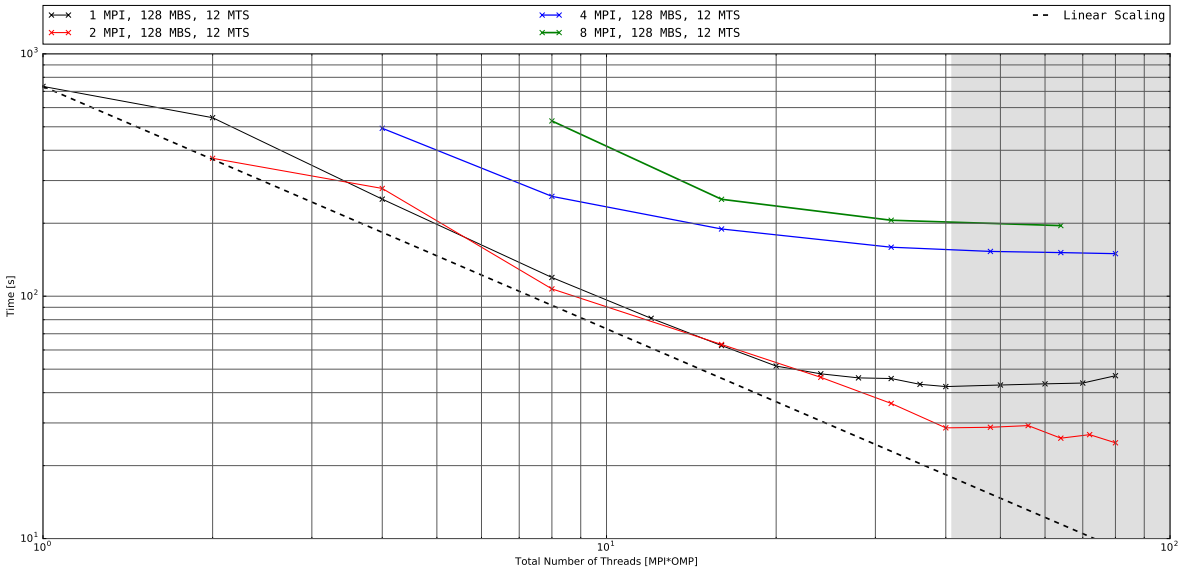
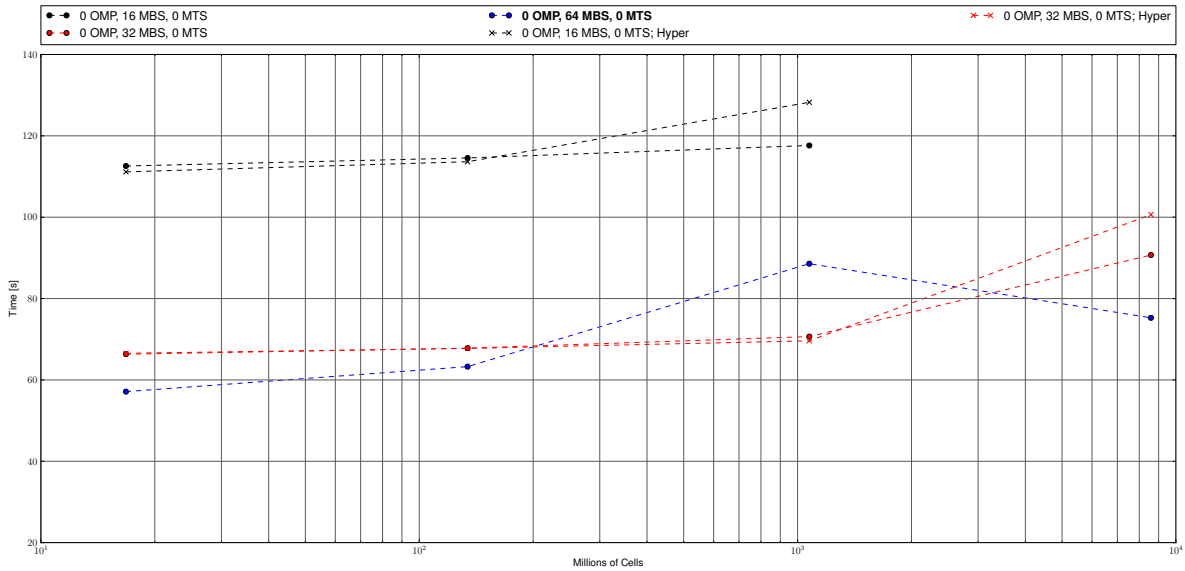
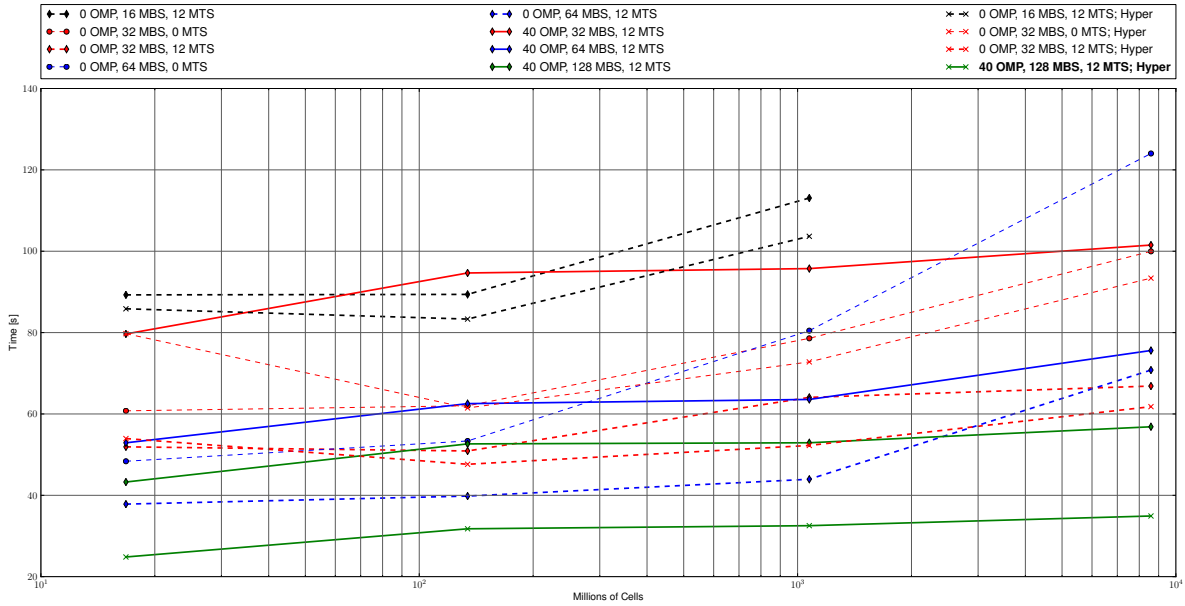


Figure 11: Strong scaling for hybrid MPI/OMP on a single node. Legend indicates number of MPI ranks, max box size (MBS) and max tile size (MTS) used. Hyperthreading region is marked with gray shading.



(a) Baseline



(b) Optimized

Figure 12: Weak scaling performance profiles exploring the sensitivity to max box size (MBS) and MPI vs OMP. Legend indicates number of OpenMP threads (OMP), max box size (MBS) and max tile size (MTS) used. Dashed lines are the no-threading (OMP=0) cases, and no-tiling is indicated by MTS=0. Only showing results for MTS=[0,12]. Hyperthreading is labeled Hyper.

5 Conclusions and Future Work

Performance enhancements in NASA's recently developed Lattice Boltzmann solver within the Launch Ascent and Vehicle Aerodynamics (LAVA) framework were presented. Two key algorithmic developments were highlighted. A coarse-fine interface treatment that discretely conserves mass and momentum has been implemented and successfully verified and validated. Code optimizations targeting improved serial and parallel performance were presented. For a simple turbulent Taylor-Green Vortex problem, we were able to demonstrate a 2.3x speedup over the baseline code for a single Skylake-SP node containing 40 physical cores, and a 2.14x speedup for 64 nodes containing 2560 physical cores. In addition, we were able to show that the optimizations enabled us to scale the code almost perfectly to 20480 physical cores where, including ghost cells, the problem size was 10 billion cells.

Our findings here confirm that larger boxes can be more efficient with an additional level of parallelization. With the capability shown in this work we are confident that the hybrid MPI/OMP tiling paradigm demonstrated in this work is a promising path forward for high-performance on memory bandwidth limited many-core architectures. We intend to carry this paradigm over to the NS solver within the LAVA Cartesian AMR framework, easily leveraging the object-oriented code base developed in this work.

In future work we also intend to continue our development of an optimized and conservative CSC approach. We anticipate that with the CSC version of the code, more work can be packed onto each tile during the asynchronous phase, theoretically yielding better overlap of computation with communication. An additional avenue of research is that we intend to have the code auto-tune to the optimal tile shape (i.e. not cubic), either as it runs, or as a pre-processing step for each processor type. Of course, immersed boundaries and AMR bring a non-uniformity to the boxes which is addressed with dynamic scheduling, and while the current approach targets this, we intend to complete this portion in future work, i.e. finish threading/tiling the irregular stencil operators where we have complex geometry.

We are also exploring targetting GPU accelerators, and an initial single box mini-app implementation has shown great promise. For a single NVIDIA Volta GPU we were able to solve the same TGV problem at 256^3 for 64 timesteps in 5 seconds, or a further 5x speedup over the optimized CPU version presented in this work. Future efforts targetting GPU accelerators will need to focus on the more complicated topics of MPI parallel, AMR, and geometry, both static and moving.

6 Acknowledgments

This work was funded by the NASA Transformational Tools and Technologies Project (TTT) of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate (ARMD). Computer time has been provided by the NASA Advanced Supercomputing (NAS) facility at NASA Ames Research Center. We would like to thank our colleagues from NASA's Computational Aerosciences Branch for many useful discussions.

References

[1] Christopher L. Rumsey, Jeffrey P. Slotnick, and Anthony J. Sclafani. Overview and Summary of the Third AIAA High Lift Prediction Workshop. In *AIAA SciTech, Kissimmee, Florida*, January 2018.

[2] Cyrus K Aidun and Jonathan R Clausen. Lattice-Boltzmann method for complex flows. *Annual review of fluid mechanics*, 42:439–472, 2010.

[3] Michael F. Barad, Joseph G. Kocheemoolayil, and Cetin C. Kiris. Lattice Boltzmann and Navier-Stokes Cartesian CFD Approaches for Airframe Noise Predictions. In *23rd AIAA Computational Fluid Dynamics Conference, AIAA AVIATION Forum, Denver, Colorado*, 2017. AIAA-2017-4404.

[4] Michael F Barad, Christoph Brehm, Cetin C Kiris, and Rupak Biswas. Parallel adaptive high-order CFD simulations characterising SOFIA cavity acoustics. *International Journal of Computational Fluid Dynamics*, 30(6):437–443, 2016.

[5] Shiyi Chen and Gary D Doolen. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[6] Dazhi Yu, Renwei Mei, Li-Shi Luo, and Wei Shyy. Viscous flow computations with the method of lattice boltzmann equation. *Progress in Aerospace Sciences*, 39(5):329–367, 2003.

[7] Joseph Kocheemoolayil, Michael Barad, and Cetin Kiris. How good is the Lattice Boltzmann Method? In *69th Annual Meeting of the APS Division of Fluid Dynamics*, 2016.

[8] Florian Schornbaum and Ulrich Rude. Massively Parallel Algorithms for the Lattice Boltzmann Method on NonUniform Grids. *SIAM Journal on Scientific Computing*, 38(2):C96–C126, 2016.

[9] Francois Cadieux, Michael Barad, and Cetin Kiris. A high-order kinetic energy preserving scheme for compressible large-eddy simulation. In *To Appear, ICCFD10, July 9-13, Barcelona, Spain*, 2018.

[10] Cetin C. Kiris, Jeffrey A. Housman, Michael F. Barad, Christoph Brehm, Emre Sozer, and Shayan Moini-Yekta. Computational framework for Launch, Ascent, and Vehicle Aerodynamics (LAVA). *Aerospace Science and Technology*, 55:189 – 219, 2016.

[11] D. Greenblatt, K.B. Paschal, C.-S. Yao, J. Harris, N.W. Schaeffler, and A.E. Washburn. Experimental Investigation of Separation Control Part 1: Baseline and Steady Suction. *AIAA Journal*, 44(12):2820–2830, 2006.

[12] G. Stich, J. Housman, J. Kocheemoolayil, M. Barad, and C. Kiris. Application of a Lattice-Boltzmann and Navier-Stokes Methods to NASA's Wall Mounted Hump. In *AIAA Aviation*, 2018.

[13] M Rohde, D Kandhai, JJ Derksen, and HEA Van den Akker. A generic, mass conservative local grid refinement technique for lattice-Boltzmann schemes. *International journal for numerical methods in fluids*, 51(4):439–468, 2006.

[14] Hudong Chen, Olga Filippova, J Hoch, K Molvig, Rick Shock, Chris Teixeira, and Raoyang Zhang. Grid refinement in lattice Boltzmann methods based on volumetric formulation. *Physica A: Statistical Mechanics and its Applications*, 362(1):158–167, 2006.

[15] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. unpublished, 2000.

[16] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4):143, 2014.

[17] B. Duda and E. Fares. Application of a Lattice-Boltzmann Method to the Separated Flow Behind the NASA Hump. In *54th AIAA Aerospace Sciences Meeting*, January 4-8 2016. AIAA-2016-1836.

[18] A. Uzun and M.R. Malik. Large-Eddy Simulation of Flow over a Wall-Mounted Hump with Separation and Reattachment. In *AIAA Journal*, volume 56, February 2 2018.

[19] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.