The Astrophysical Journal Supplement Series, 228:23 (23pp), 2017 February https://doi.org/10.3847/1538-4365/aa5b9c

WOMBAT: A Scalable and High-performance Astrophysical Magnetohydrodynamics Code

P. J. Mendygral[1,2], N. Radcliffe[1], K. Kandalla[1], D. Porter[3], B. J. O'Neill[2,3], C. Nolting[2,3], P. Edmon[4], J. M. F. Donnert[2,5,6], and T. W. Jones[2,3]
[1] Cray Inc., St. Paul, MN 55101, USA; pjm@cray.com, nradclif@cray.com, kkandalla@cray.com
[2] School of Physics and Astronomy, University of Minnesota, Minneapolis, MN 55455, USA; oneill@astro.umn.edu, nolt0040@umn.edu, donnert@ira.inaf.it, twj@umn.edu
[3] Minnesota Supercomputing Institute for Advanced Computational Research, Minneapolis, MN USA; dhp@umn.edu
[4] Institute for Theory and Computation, Center for Astrophysics, Harvard University, Cambridge, MA 02138, USA; pedmon@cfa.harvard.edu
[5] INAF-Istituto di Radioastronomia, via. P.Gobetti 101, I-40129 Bologna, Italy
Received 2016 November 28; revised 2017 January 11; accepted 2017 January 19; published 2017 February 23

Abstract

We present a new code for astrophysical magnetohydrodynamics specifically designed and optimized for high performance and scaling on modern and future supercomputers. We describe a novel hybrid OpenMP/MPI programming model that emerged from a collaboration between Cray, Inc. and the University of Minnesota. This design utilizes MPI-RMA optimized for thread scaling, which allows the code to run extremely efficiently at very high thread counts ideal for the latest generation of multi-core and many-core architectures. Such performance characteristics are needed in the era of "exascale" computing. We describe and demonstrate our high-performance design in detail with the intent that it may be used as a model for other, future astrophysical codes intended for applications demanding exceptional performance.

Key words: magnetohydrodynamics (MHD) – methods: numerical

1. Introduction

Magnetohydrodynamic (MHD) simulations allow us to study the dynamics of highly conducting astrophysical fluids since many astrophysical fluids are highly conductive ionized plasmas. MHD modeling then allows us to incorporate the essential consequences of magnetic fields. Even "weak" magnetic fields, whose Maxwell stresses are subdominant to inertial and to thermal pressure stresses, can have a major impact on the development of turbulence and its dissipation on small scales, on momentum transport, angular momentum and energy, and on thermal conduction. If the simulations include, in addition to MHD, the transport of high-energy, non-thermal "cosmic ray" particle populations, the simulations can model emission processes involving the cosmic ray interactions with the bulk fluid and its magnetic field. These include γ-ray by-products of cosmic ray proton interactions with the bulk fluid and radio to X-ray emissions from cosmic ray electrons, including synchrotron radiation.

Since magnetic field properties often derive from the details of the fluid dynamics over a wide range of scales of interest, it is essential for simulations to capture the dynamics with high fidelity over this full range of scales. This is generally a very intensive and challenging computational task that, despite much progress in coding and vast improvements in computing infrastructure, has often remained beyond current capabilities. That challenge is the motivation for our efforts described here to develop an MHD code environment that can effectively utilize and adapt to the coming generations of computational infrastructure to allow solutions to these pressing astrophysical problems.

Numerous codes exist for both general purpose and specific use astrophysical fluid simulations. Some examples are GADGET (Springel 2005; Dolag & Stasyszyn 2009), NDSPMHD (Price 2012), AREPO (Springel 2010; Mocz et al. 2016), ENZO (Bryan et al. 2014), ATHENA (Gardiner & Stone 2008; Stone et al. 2008), RAMSES (Teyssier 2002; Fromang et al. 2006), CHARM (Miniati & Martin 2011), PLUTO (Mignone et al. 2007, 2012), CASTRO (Almgren et al. 2010), and FLASH (Fryxell et al. 2000; Dubey et al. 2008). Codes like these have been developed over many years and often have features for adding the effects of gravity, cosmic-ray transport, non-ideal MHD, cosmic expansion, and non-adiabatic energy gains and losses, including radiative and conductive cooling and heating. "Exascale" is the next major step in the evolution of high-performance computing (HPC), with systems capable of performing 10^{18} floating point operations per second distributed across many levels of parallelism. Preparing applications for exascale requires a substantial investment in code re-design and optimization, to enable the community to leverage the capability of new architectures and make new scientific breakthroughs. Dubey et al. (2016) recently presented a survey of the challenges and potential approaches to modernizing some of the most popular community codes.

The latest multi-core and many-core processors (CPUs), such as Intel Xeon and Intel Xeon Phi, feature increasing core counts per processor with decreasing clock speed along with increasing single instruction-multiple data (SIMD) vector lengths. Hence, cache blocking and vectorization are critical to obtaining good performance from modern processors. But the increasing core counts also put pressure on the traditional MPI-only (Message Passing Interface) parallelization models. Memory consumption from a large number of independent MPI processes on a node may become prohibitive. For MHD simulations that develop substantial load imbalance, possibly through the inclusion of *N*-body dynamics or multi-level mesh refinement, balancing work between MPI ranks is critically important. However,

[6] ERC Marie Curie Fellow.

the process of balancing work between MPI ranks carries a potentially significant overhead. This overhead is the combination of the cost of moving grid data between MPI ranks and communication of the change in decomposition to some or all MPI ranks. There are several established strategies for reducing the overhead, including decomposition meta-data replication, but these techniques come at the cost of memory and complexity (Dubey et al. 2016). Programming models that allow for load balancing with less explicit communication are greatly needed.

One attractive approach is the hybrid OpenMP/MPI model, as discussed in Bryan et al. (2014). It allows MPI ranks to hold larger portions of the world grid. In the context of mesh refinement, added work due to refinement at any single location is a lower fraction of a rank's total load. For many calculations it could also result in a more symmetric load across MPI ranks if refinement needs are not confined to a single region. Work within an MPI rank can be load balanced among threads with any form of dynamic work scheduling. Finally, on-node imbalances due to contention of shared resources, such as cache or bandwidth, also can be mitigated with attention to thread scheduling. However, typical parallel loop-based OpenMP designs have shown too little scope (amount of code effectively threaded) to scale effectively to high thread count.

Modern HPC interconnects often feature low latency/high bandwidth messaging with network-offloading, which enables overlap of computation with communication. MPI-RMA (Remote Memory Access) is a feature added to the MPI standard in order to expose these capabilities to the user. It should be possible for an application to drive communication near hardware limits with a highly efficient MPI-RMA implementation. However, MPI libraries need high performance MPI_THREAD_MULTIPLE implementations for the hybrid OpenMP/MPI model to include communication parallelization.

In this paper we present an application design study for a new grid-based MHD code called WOMBAT.[7] The goal of this project is to address the optimization opportunities discussed above through a co-design process. In pursuit of this goal, we seek a base design well suited for uni-grid simulations yet formulated for complex conditions requiring load balancing. For the purpose of this paper, we review the base design for MHD uniform meshes only. WOMBAT development is a collaboration between Cray Inc. Programming Environments and the University of Minnesota. Through this collaboration we developed a design strategy (see Section 2) that adapts to architectures (CPU and interconnect) using language, OpenMP, and MPI best practices. We also identified bottlenecks and optimizations for MPI (Cray MPICH) resulting in significant performance improvements. Section 3 is a performance review of WOMBAT on three architectures that can be used as a model for assessing the quality of any similar implementation. We discuss specific implementation details in Section 4. Our design strategy is applicable to many other codes and serves as a potential path forward for exascale application readiness.

In what follows "KNL" designates the Intel® Xeon® Phi many-core processor (Knights Landing), "Broadwell" a recent Intel® Xeon® multi-core processor (Broadwell) and "Interlagos" the AMD Opteron™ multi-core processor (Interlagos). In all figures, these processors are shown as red, green, and blue, respectively.

[7] WOMBAT is available by request or by visiting http://www.astro.umn.edu/groups/compastro.

2. Scalable Design Strategy

The key design characteristic of WOMBAT is to subdivide the problem into completely independent pieces of the world grid that include their own boundary zones and necessary meta-data for updating from one time step to the next. We refer to these independent pieces as "Patches." This design naturally accommodates any numerical method with local or semi-local communication needs.

The concept is similar to data management strategies in other many MHD codes (see Dubey et al. 2016), but our design takes a unique approach to processing and scheduling the computation and communication of Patches. A Patch is a unit of work that a thread within a WOMBAT MPI process independently operates on. No assumptions are made on the number of Patches relative to the number of threads since our design adapts to this ratio. Patch boundaries also define units of communication work done with either local (intra-process) or remote (inter-process) copies. The number of zones in each dimension of a Patch and the number of them in each dimension on a rank (and Domain, see Section 2.1) are input parameters. This allows us to tune them for performance on a given architecture (see Section 3.2.1).

WOMBAT is written in Fortran 2008. Fortran semantics make it easy for modern compilers to identify and apply optimizations, such as vectorization, as long as developers follow simple rules (see Section 2.2.2). Code modularity, organization and maintainability benefit from the object-oriented features available in Fortran 2008. However, overuse of classes can lead to a loss of optimization opportunities for a compiler. Thus any performance critical section of WOMBAT is basic Fortran code working on arrays. The code is constructed from three main categories of classes: data managers, engines, and solvers. Data managers do memory management and supporting functions. Solvers accept data managers as arguments and update their arrays following whatever numerical methods they employ. Engines orchestrate parallelism and the book-keeping and communication requirements for handing data managers to solvers.

Table 1 shows the hardware constraints on the current and next generation of HPC systems, alongside the techniques and optimizations we include in our design strategy to meet these constraints.

2.1. Domain Decomposition

To construct the Patches, the world grid is decomposed into N equal-size sub-volumes (Domains), each assigned to an MPI rank. An MPI rank's sub-volume is further decomposed into Patches containing an equal number of zones. Patches in a Domain communicate boundary data with one another and with Patches on neighboring ranks. Figure 1 shows a sample 2D configuration of the Patch–Domain hierarchy across an arbitrary number of MPI ranks. The figure is centered on a single MPI rank's Domain. That rank has eight neighbors labeled N0 through N7 each with their own Domain. Inside every MPI rank's Domain is a 5×5 grid of Patches, labeled P0 through P24 for the central Domain.

We implement the Patch–Domain design as Fortran classes. The Domain class is responsible for tracking the MPI rank and

Table 1
HPC Architecture Constraints and the Optimization Techniques and Features Developed in WOMBAT to Address them

Hardware Constraint	Optimization Approach	WOMBAT Design
high FLOP/Byte ratio	cache blocking	*Patch*
slow scalar + wide vectors	vectorization	*Fortran* + vectorization best practices
many cores	thread scalability	SPMD OpenMP + new Cray MPICH + *Patch*
distributed architecture	RDMA + computation/communication overlap	AIO + MPI-RMA + new Cray MPICH + *Patch*

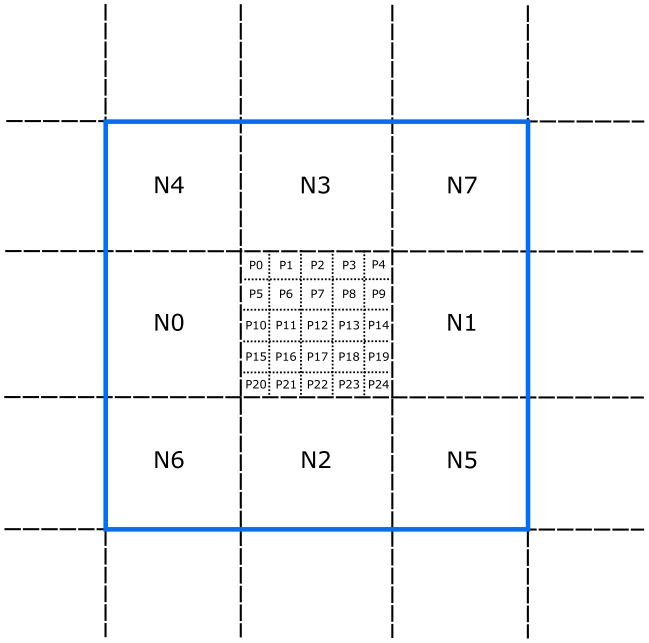

Figure 1. Sample 2D decomposition. Regions labeled N[0–7] are the Domain of each MPI neighbor rank for the MPI rank in the center. Regions labeled P [0–24] are 25 Patches inside each MPI rank's Domain.

local or remote Patches that share boundaries with it. This design can be extended to nested block static or adaptive mesh refinement (SMR or AMR). In support of that and generic load balancing between MPI ranks, Patches inside a Domain are allowed to become active or inactive, meaning member data structures can be allocated and updated in time, deallocated and not included in updates. Information about the MPI rank(s) and remote Patches sharing a boundary with a Patch can be modified over time, allowing Patches to be moved between ranks with minimal bookkeeping and communication. In particular, we do not store global data structures for tracking decomposition.

2.2. Optimization and Multi-level Parallelization Strategy

Three levels of parallel optimization are common to HPC systems: SIMD vectorization, intra-process threading (using OpenMP), and inter-process communication (using MPI). Cache blocking is an additional optimization that addresses memory topology on these systems.

2.2.1. Cache Blocking

The Patch design naturally results in cache blocking, which directly addresses bandwidth limitations. The most popular CPUs used in HPC today have FLOP/s to byte/s ratios (ratio of floating point performance to memory bandwidth[8]) of $\simeq$10, hence algorithms with similar computational intensities run most efficiently. However, computational intensities that high are difficult to achieve with stencil-based numerical methods for solving MHD. Processor caches can mitigate this issue when used effectively and stencil methods provide good opportunity for reuse of loaded values between operations. Hence, good performance requires cache blocking techniques on all key loops. However, explicitly programmed cache blocking can be cumbersome, because *all performance-critical* nested loops over problem dimensions must be expanded into higher-dimensional loops with tunable blocking parameters.

Since all solvers in WOMBAT operate on a single Patch, their computationally intensive loops are all roughly the size of a Patch. The best Patch size that fits into a level of cache inherently gets reuse out of a cache (typically the best size fits into a level 3 (L3) cache but not entirely into level 2 (L2), see Section 3.2.1).

2.2.2. SIMD Vectorization

Operations on a Patch consist of floating point and data motion intensive loops. These loops are written to be good SIMD vectorization candidates following the typical rules of stride-one access, recurrence free, and limited conditional logic. To accomplish good maintainability and portability we do not explicitly program this level of parallelization and leave it to a compiler to decide if and when to use vectorization. This usually requires scalar operations be isolated to separate loops so the remaining work is available for vectorization.

2.2.3. OpenMP Threading

The benefits of hybrid application scaling to high thread counts were discussed in Section 1. We avoid the bottlenecks of parallel loop-based OpenMP by arranging WOMBAT such that only one OpenMP parallel region is present for the duration of execution. This design presents the threaded region as a set of completely independent processes, which mimics the parallelism of MPI. We refer to this approach as SPMD OpenMP (or single program-multiple data OpenMP) (see also Kandalla et al. 2016).

To illustrate this design, we show a flow diagram of the main driver in WOMBAT in Figure 2. A section including MPI initialization and base object construction is the only work by the main thread outside the parallel region. After that, every portion of WOMBAT is executed by all threads collaboratively. This includes array allocation/paging, computation, communication, and even I/O.

[8] For example, a 1.4 GHz Intel Xeon Phi processor is theoretically able to achieve $\simeq$3 TFLOP/s and has a memory bandwidth of $\simeq$450 GB/s to MCDRAM (90 GB/s to DDR).

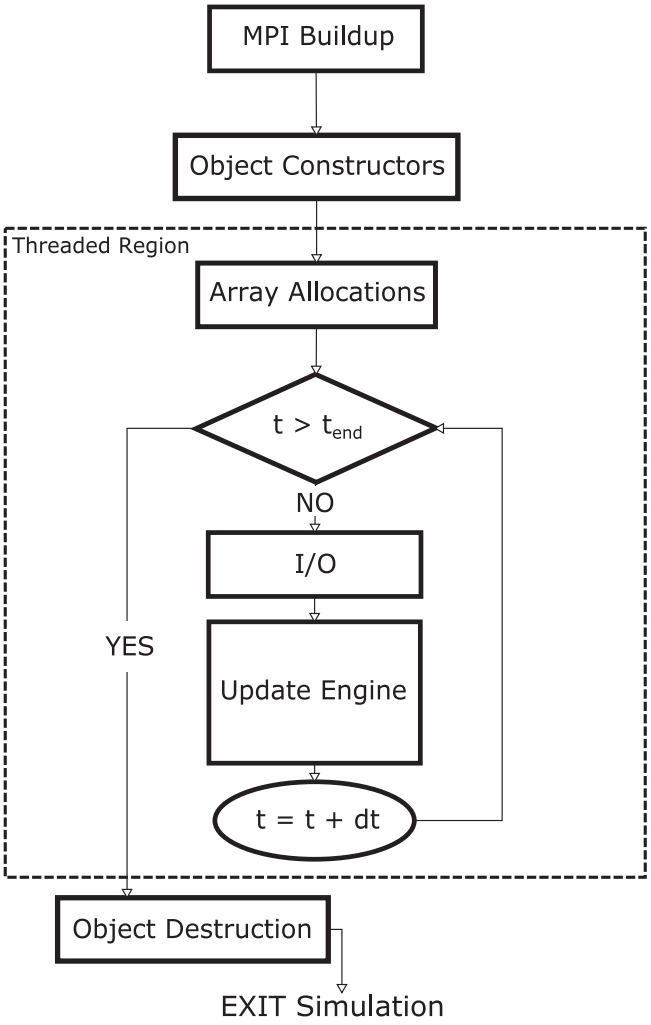

Figure 2. Flow diagram for the main driver in WOMBAT. The scope of the *single* OpenMP parallel region is outlined with a dashed box.

2.2.4. MPI

To allow fine-grained work–communication overlap, the Patch design results in a larger number of small messages, compared to traditional approaches which typically use fewer monolithic boundary exchange messages. This shifts the communication sensitivity of WOMBAT from simply bandwidth to bandwidth and message rate, depending on the number and size of Patches. A Domain decomposed into Patches will generate more MPI messages and a higher aggregate amount of data moved, especially in 3D, due to added corner and edge boundaries. This extra communication will later be leveraged for communicating load information and changes to Patch ownership. Furthermore, in the SPMD OpenMP approach every thread can participate in the MPI communication using MPI_THREAD_MULTIPLE.

Most of the MPI communication in WOMBAT uses MPI-RMA (e.g., MPI_Put(), MPI_Get()), because of the low overhead possible with a proper MPI-RMA implementation. MPI-RMA was added to the MPI standard primarily to give users direct access to the Remote Direct Memory Access (RDMA) features available on most HPC networks (interconnects). PUT and GET operations in MPI-RMA are inherently non-blocking, and excellent overlap of computation and communication is possible on networks that also support network-offloading. While the semantics of MPI-RMA allow for these performance characteristics, many MPI libraries today implement MPI-RMA using two-sided communication (e.g., Dinan et al. 2016). This adds overhead and reduces the chances for overlap, leaving MPI-RMA practically unusable for an HPC application. Recent work in MVAPICH (Potluri et al. 2011a, 2011b; Li et al. 2013) and OpenMPI (Hjelm 2014, 2016) has corrected that issue on both Infiniband and Cray networks. In Cray MPICH, MPI-RMA is now based on the low-level DMAPP library specifically designed for optimal one-sided communication on Gemini and Aries interconnects. This implementation has very low overhead, tuned to utilize the network-offload (Block Transfer Engine or BTE) capability on Cray XE/XC systems.

Message rate requirements and the SPMD design make it critically important that multi-threaded MPI-RMA in a given MPI library performs well. During the design of WOMBAT we found that no MPI implementation really achieved the performance that should be possible. This is because most MPI implementations (including Cray MPICH at the time) use a global lock to provide thread safety (see also Dosanjh et al. 2016). This serializes all MPI calls and, more importantly, most work in the user code around those MPI calls. Through a co-design approach we have optimized Cray MPICH for high performance and thread scalable MPI-RMA communication (see Section 4.1.2). We refer to this new capability as "thread-hot RMA." An initial version is available to Cray users starting with Cray MPICH 7.3.4 with additional enhancements from the work presented here available in an upcoming release. Other MPI libraries are also pursuing optimizations for MPI_THREAD_MULTIPLE that will make our design performance portable beyond Cray systems (Amer et al. 2015; Vaidyanathan et al. 2015).

2.2.5. Asynchronous I/O

WOMBAT uses a custom asynchronous I/O (AIO) library that allows for overlap of simulation progression and data writing. If all I/O data can be buffered, data can be written out with almost no impact on execution time. Some portion of I/O work done is blocking if buffers are made smaller, which reduces overall performance.

AIO is implemented as a set of specialized ranks dedicated to receiving (or sending for read operations) data from a client set of worker ranks. All threads in the worker ranks package data into I/O buffers. Non-blocking communication is used to move data to AIO server ranks, and the AIO ranks then write data out as they come in. The full system can be tuned for I/O and overlap performance by adjusting the total number of AIO server ranks.

3. Performance

We measure the performance of WOMBAT for 3D MHD calculations using the directionally un-split MHDTVD solver described in Section 5. Single-node tests focus on the impact of vectorization on overall execution and the parallel efficiency of the SPMD OpenMP technique. Multi-node tests at scale measure the performance of the full suite of parallelization strategies, including off-node communication, and how they interact. We stress that the problem sizes used for most experiments presented here were selected to show overheads in

Table 2
Test Platforms Used in the Performance Studies and Their Characteristics

System Title	Architecture	Interconnect	Topology	CPU	VL [bits]	Cores per Node
Blue Waters	Cray XE	Cray Gemini	3d torus	AMD Opteron™ 6276 "Interlagos" @ 2.3 GHz	256	16
XE_{IL}	Cray XE	Cray Gemini	3d torus	AMD Opteron™ 6281 "Interlagos" @ 2.5 GHz	256	16
XC_{BDW}	Cray XC	Cray Aries	dragonfly	Intel® Xeon® E5-2695 "Broadwell" @ 2.5 GHz	256	36
XC_{KNL}	Cray XC	Cray Aries	dragonfly	Intel® Xeon Phi™ 7250 "KNL" @ 1.4 GHz	512	68

Note. We mostly use 64 cores on KNL systems to make scaling studies simpler multiples from lower core counts.

Table 3
Processor Compilation and Placement Notes

CPU	Compilation Flags	Notes
Interlagos	-O vector3 -h preferred_vector_width = 256	Only one thread/process per floating point unit
Broadwell	*defaults*	Only one thread/process per core (no hardware threads used)
KNL	*defaults*	Only one thread/process per core (no hardware threads used)

WOMBAT, and in particular communication. This also closely follows real-world simulations run on production systems.

Table 2 summarizes the platforms used for performance experiments. We used *Blue Waters* (Cray XE) at the NCSA for very large weak scaling studies. The remaining systems are internal configurations at Cray Inc. We use the Cray Compiler (CCE) in all experiments. Table 3 shows specific test information about each processor. We used the new Cray MPICH library with the "thread-hot RMA" feature in all tests unless otherwise noted.

All experiments involving KNL were run with nodes configured in so-called "quadrant" Non-Uniform Memory Access (NUMA) mode with high bandwidth memory (on package) configured as a 16 GB L3 cache.

3.1. Single-node Performance by Architecture

3.1.1. SIMD Scaling

We measure the impact of increasing vector length (VL) from 64 to 512 bits on KNL.[9] The problem size is a 17×4^2 Domain of Patches each with 48^3 zones updated by a single MPI rank with 68 threads.

Figure 3 shows the strong scaling with increasing VL on a single node of the XC_{KNL} system. The time to perform a single time step update is reduced approximately by a factor of 2 going from 64 and 256 bit vectors. The final step to 512 bit vectors continues to show improved performance, but the effect has been reduced to only a $\simeq$19% speedup. Factors that affect the speedup from vectorization are the amount of vector versus scalar code executed, the efficiency of the vector code generated by the compiler, and the memory bandwidth available to provide data to the cores. Overall, vectorization speeds up WOMBAT by almost a factor of 2.5X on KNL processors. The speedup is roughly consistent with Amdahl's Law, assuming the fraction of execution time benefiting from parallelization $p \approx 0.65$.

Broadwell has accessible hardware performance counters for floating point operations that can be measured with a number of performance tools, such as PAPI or CrayPAT. Table 4 shows the quality of vectorization relative to a scalar build of

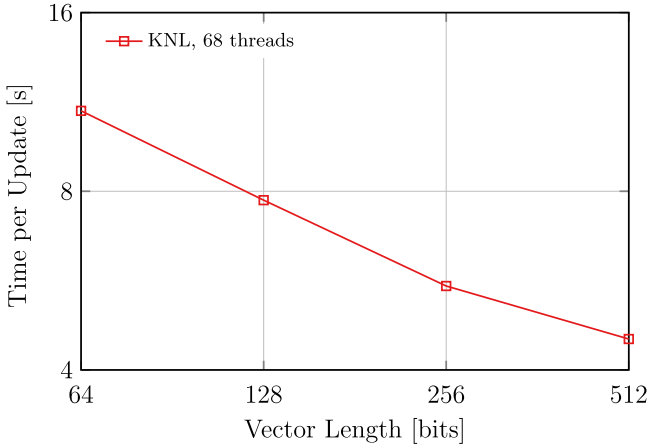

Figure 3. Time per update in seconds for increasing VL (strong scaling) on KNL with 68 threads.

Table 4
Effect of Vectorization on Floating Point Instruction Count on Broadwell for a Single Thread Running a $2^3 \times 32^3$ Problem for 100 Time Steps

Compiler	10^9 Scalar Ops.	10^9 256 B SIMD Ops.	sec/update	% of DP Peak
CCE 8.5.4 (scalar)	327	0	1.87	6.6
CCE 8.5.4	20	74	0.96	14.1
Intel 17.0.1.132	15	75	1.67	8.1
GNU 6.2.0	320	0	1.84	6.6

WOMBAT. We also show the breakdown of scalar and vector operations for the Intel and GNU compilers. Vectorization with CCE reduces total double precision (DP) floating point instruction count by ~71%. 79% of all floating point operations are vector. Intel produces a similar amount of vector instructions but also lower performance. The performance difference is due to much lower translation lookaside buffer (TLB) utilization despite using 2 Megabyte huge pages. We intend to file a performance bug with Intel on this issue and it will be corrected in later releases. The GNU compiler is unable to produce any vector instructions.

[9] To vary the types of vectors, we use the CCE compiler flag "-h preferred_vector_width = X," where X = {128, 256, 512}. For scalar 64 bit vectors we use "-O vector0."

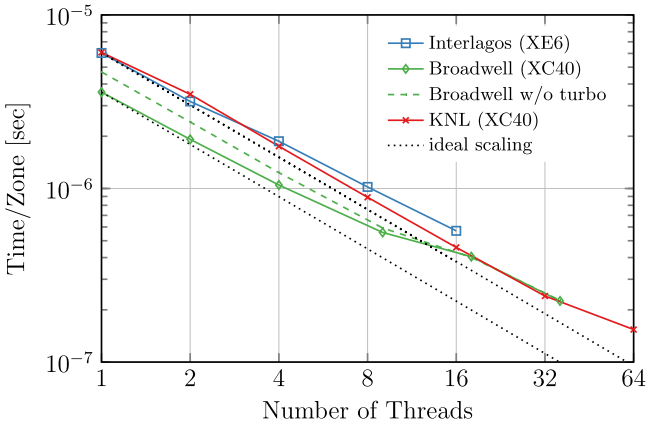

Figure 4. Execution time per zone over number of threads (strong scaling) for Interlagos (blue), Broadwell (green) and KNL (red).

Table 5
Number of Patches in Each Direction of the Domain and Number of Zones per Patch Used by Each Architecture in the Thread Strong Scaling Test

CPU	Domain Size	Patch Size
Interlagos	$8 \times 7 \times 4$	40^3
Broadwell	$12^2 \times 8$	32^3
KNL	8^3	48^3

3.1.2. Thread Scaling

We show the thread strong scaling speedup of WOMBAT on the three architectures presented here in Figure 4 with the problem sizes given in Table 5. The code shows excellent speedup with threads on all the architectures. The speedup from threads on KNL is ∼40X at 68 threads. The "turbo" on Broadwell increases performance for small thread numbers (green line versus dashed green line). On both Interlagos and Broadwell there is notable loss of scaling once the process has threads spanning beyond a single NUMA domain. On Interlagos this is at 4 threads, because each Interlagos processor is made up of two "Bulldozer" modules for a total of four NUMA nodes on a dual socket XE node. A KNL node configured in quadrant mode has only a single NUMA node, and the deviation from ideal scaling is moved to much higher thread counts. Profiles showed that the cost of thread synchronization rising at these thread counts, but there is the additional factor of a finite amount of bandwidth available on the processor. Both of these account for most of the reduction in performance from ideal.

3.2. Performance at Scale by Architecture

Off-node components of an application, such as network latency and bandwidth, can modify its behavior and how it should be tuned. We present a multi-node Patch size optimization study for WOMBAT, and we also demonstrate the weak and strong scaling capabilities of WOMBAT out to large node counts.

3.2.1. Patch Size Optimization

In Section 2.1 we described how Patches are logically assembled to produce any grid size per rank. We study performance with Patch size also allowing the mixture of MPI ranks to OpenMP threads to vary at a scale of 27 nodes. This number is used because any configuration of MPI ranks to threads at that scale will have unique neighbors in 3D. This ensures that the MPI work is saturated and performance is not skewed. We use "PPN" to denote the number of MPI processes per node with threads placed on all cores. The total number of zones across each Patch size was held approximately constant within a system type. We chose the problem setup so update-times were held at ≃10 s. This is sufficiently large to expect throughput values to be near their absolute peak but still include overhead sensitivity.

Figure 5 shows the throughput on each architecture given by the number of zones per second each node can update. We can identify the maximum throughput each system can achieve for these problem setups. XE_{KNL} nodes are able to update $\gtrsim 6 \times 10^6$ zones/s/node at peak compared $\gtrsim 4.5 \times 10^6$ zones/s/node on XC_{BDW} and 2×10^6 zones/s/node on XE_{IL}. This demonstrates the ability of our approach to adapt to the unique many-core design of KNL.

The optimal Patch size is not uniform across systems. XC_{BDW} has the smallest optimal Patch size of 32^3. The optimal Patch size for XE_{IL} is 40^3, and XC_{KNL} has an optimal Patch size $\simeq 50^3$. The performance on either side of the optimal size drops off but not by the same amount on each system. Patch sizes smaller than the optimal size have lower performance due to reduced vectorization efficiency, and larger sizes have lower performance due to spilling out of L3 on both Interlagos and Broadwell. KNL has the largest SIMD vector size, which explains why it has the largest optimal Patch size. Overheads appear to affect Interlagos more than Broadwell. This favors slightly larger Patches on Interlagos despite having the same SIMD vector length as Broadwell. The cache blocking properties of the Patch design discussed in Section 2.2.1 no longer function as intended for large Patches. On KNL the performance loss is not as dramatic, with exceptions at high thread count. This is due to the 16 GB L3 cache.

Both XE_{IL} and XC_{BDW} show lowest performance with only a single rank per node packed with threads. This is largely due to the NUMA issues discussed in Section 3.1.2. In both of these cases, once the number of MPI processes per node matches the number of NUMA nodes on a node there is very little spread in performance at the optimal Patch size. In the case of XC_{KNL} performance does not vary much until the Patch size is 50^3 or greater. While the absolute best performance on XC_{KNL} is with 16 PPN ($\sim 6.5 \times 10^6$ zones/s/node), there is still significant performance at 4 PPN ($\sim 6 \times 10^6$ zones/s/node). At a high level, a fixed grid calculation should not perform any different exchanging ranks for threads if both MPI and OpenMP are well implemented and hardware limitations are not present. The SPMD design in WOMBAT nearly achieves this.

3.2.2. Weak Scaling

Figure 6 shows the weak scaling on three architectures at different values of PPN for the problem sizes given in Table 6. For XC_{IL} and XE_{BDW} the best performance and scaling is closely matched between a single MPI rank per NUMA node or pure MPI, which follows the conclusions in Section 3.2.1. The XC_{KNL} systems has best performance and scaling at 4 and 8 PPN. Relative to a *single node*, XE_{IL} has a 93% efficiency up to 150 nodes at 4 PPN, and XC_{BDW} at 2 PPN has a 87% efficiency up to 512 nodes. Remarkably XC_{KNL} has 89%

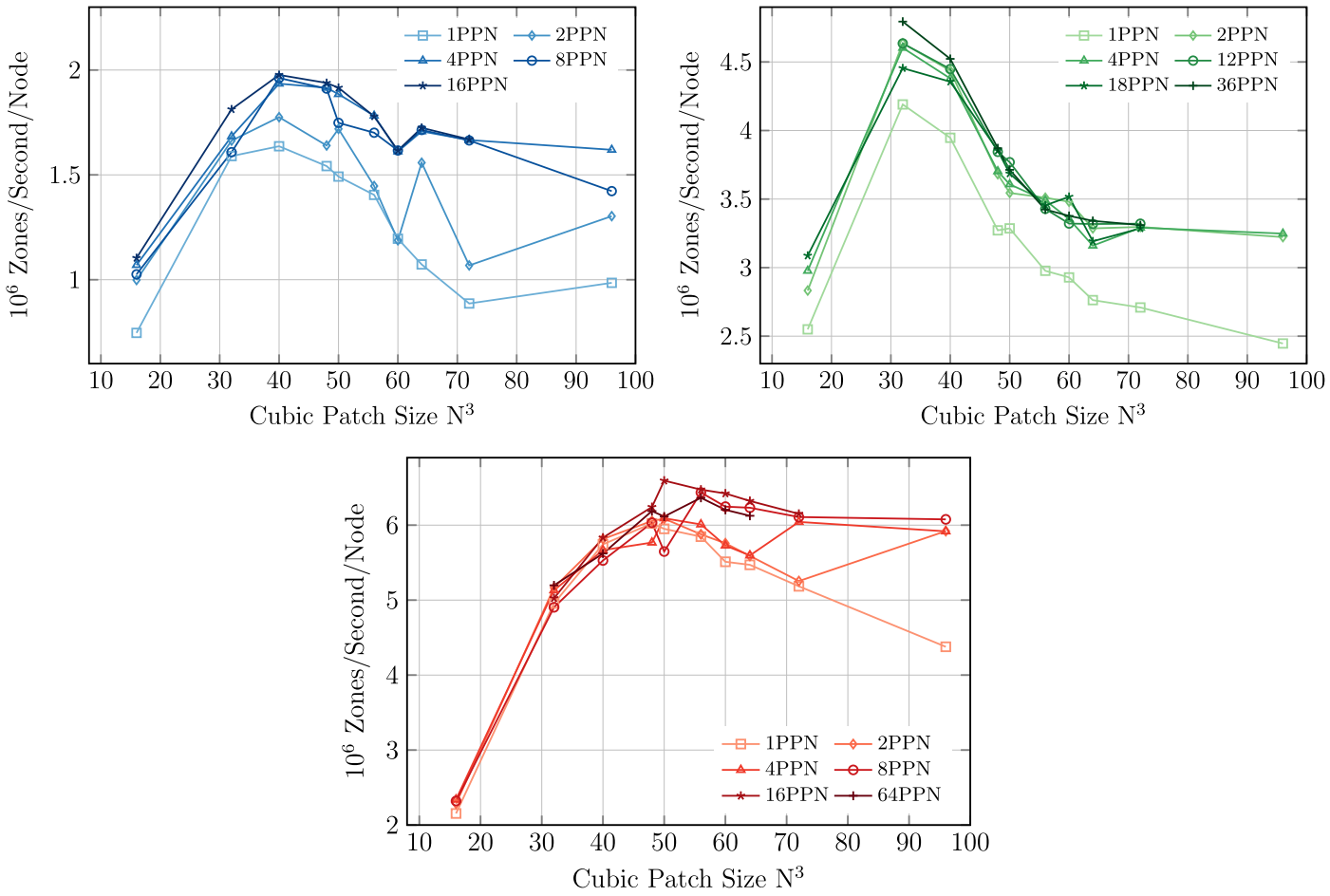

Figure 5. Performance in million zones per second per node over number of zones along one patch dimension. We show Interlagos (blue, top left) from the XE_{IL} system, Broadwell (green, top right) from the XC_{BDW} system and KNL (red, bottom) from the XC_{KNL} system at different values of MPI processes per node (PPN).

efficency at 648 nodes (41,472 threads) with 4 PPN relative to a single node run.

The lowest performance at scale on all systems is with a single rank per node. The increases in update times are due to increasing amounts of off-node communication a rank encounters and imperfect overlap of communication with computation. Between 3 and 27 nodes, off-node communication cost is saturated, and update times are nearly flat for larger node counts. We conclude that WOMBAT has excellent weak scaling on all systems with the optimal configuration despite the relatively small problem size chosen.

We show weak scaling on Blue Waters in Figure 7 for two values of PPN and problem sizes. We limit each run to just 10 time steps. The 1 PPN runs scale out to 16,224 nodes (259,584 threads) with a world grid containing ≃66 billion zones. WOMBAT scales well with 60% efficiency at 16,224 nodes for 1 PPN and 75% efficiency at 4096 nodes for 4 PPN relative to single-node runs.

There are several spikes of increased update times. We hypothesize this is due to network contention with both other running applications and with WOMBAT itself on the very large 3D torus topology. Runs on the smaller dedicate XE_{IL} system do not show these features. In the future, we intend to make use of the topology-aware scheduling capability provided by the NCSA, and we expect this to improve performance and reduce the contention opportunities.

The right panel of Figure 8 shows the impact of the "thread-hot RMA" capability that will be included in an upcoming release of Cray MPICH. We show weak scaling on XC_{KNL} to 125 nodes with 4 PPN using the same problem setup for weak scaling described above. This feature produces a 17% speedup over Cray MPICH 7.3.1.

3.2.3. Strong Scaling

The left panel of Figure 8 shows the strong scaling of WOMBAT. We defined the problem size for each system so that the time per update is limited to ∼60 s (see Table 7). Performance closely follows the theoretical speedup over 2 orders of magnitude in node count, with XC_{BDW} showing a 236X speedup at 384 nodes. The deviations are due to overheads exposed as update times at scale fall at or below 0.3 s. We reduced the Patch size on XC_{BDW} by half and on XC_{KNL} by a quarter at the largest scale, for example.

3.2.4. Thread Scheduling at Scale

We show the impact of OpenMP scheduling in Figure 9. In this experiment we modified the thread scheduling for just a single loop in the Update Engine (see Section 4.1) that drives updates over Patches. We run a balanced (Patch count divides evenly into thread count) and an imbalanced problem with the modified code and original code. Four MPI ranks per node on 27 nodes each with 16 threads update 40 (imbalanced) or 32

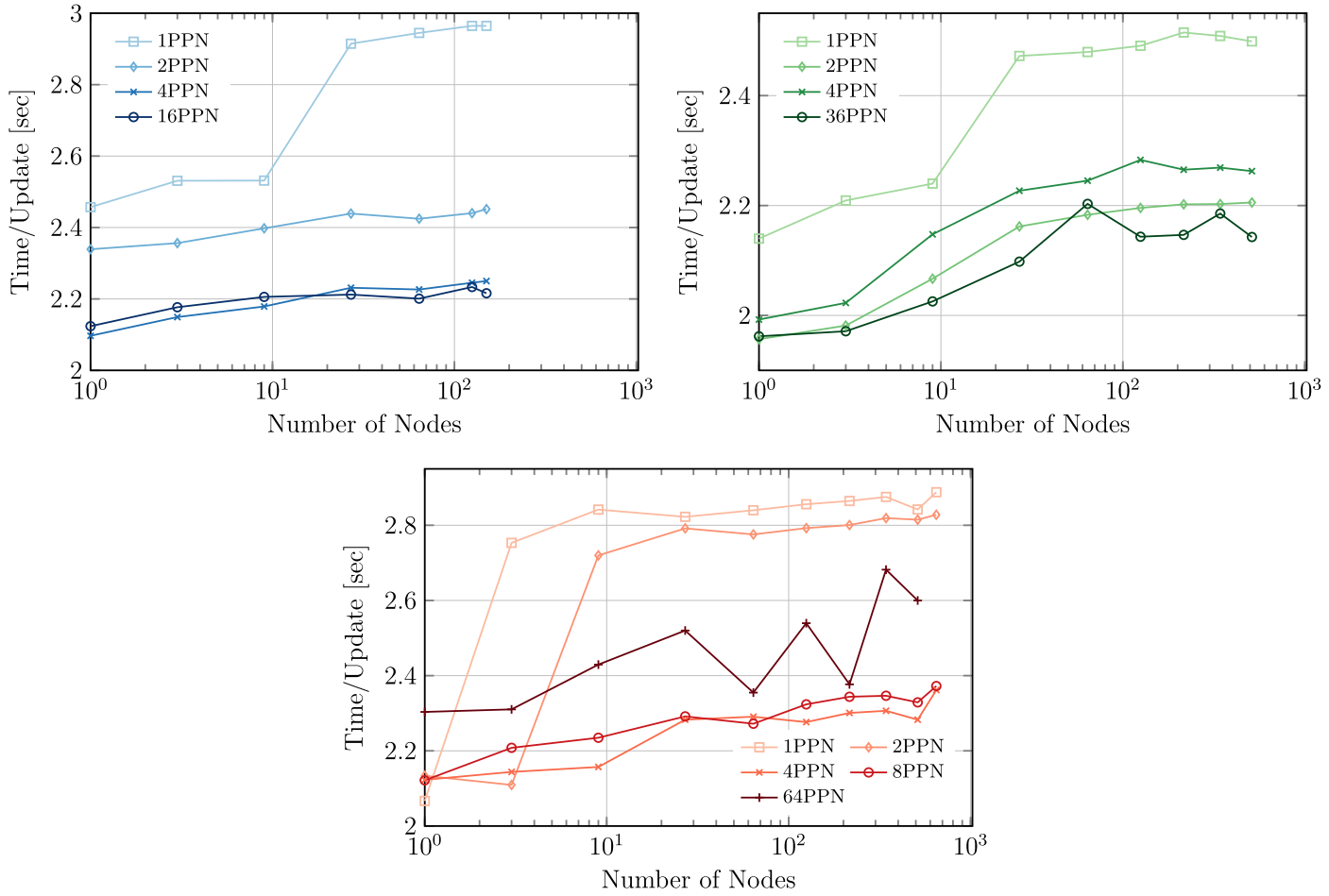

Figure 6. Time per update over number of full nodes (weak scaling) for XE_{IL} (blue, top left), XC_{BDW} (green, top right) and XC_{KNL} (red, bottom) at different values of MPI processes per node (PPN).

Table 6
Weak Scaling Test Setup: the Number of Patches in Each Direction of the Domain and Number of Zones per Patch Used for Each System

System	Patches per Node	Patch Size
XE_{IL}	$4 \times 4 \times 4$	40^3
XC_{BDW}	6×6^2	32^3
XC_{KNL}	8×4^2	48^3

(balanced) Patches, each with 48^3 zones on the XC_{KNL} system. Both problems show near optimal throughput with the GUIDED schedule and lower performance with STATIC. Our SPMD OpenMP design has very few thread barriers, and using a STATIC schedule assumes threads are roughly synchronized to be efficient.

4. Design Details

4.1. Update Engine

In our design, the Update Engine is responsible for scheduling computation and communication across threads for any solver. It accepts a Domain and iterates over Patches, exchanging messages and partitioning the update work through the specified solver until all Patches report back as completed. To allow for iterative or sub-cycling solvers, it is not necessary that a Patch be updated completely for the current time step after only a single pass through a solver.

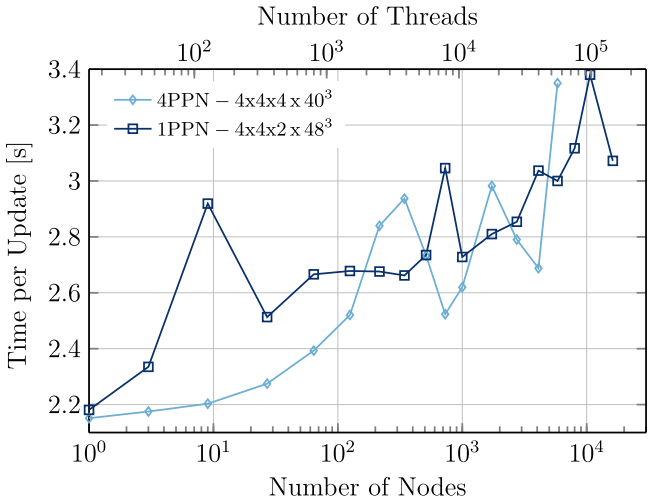

Figure 7. Weak scaling on Blue Waters for two different values of MPI processes per node (PPN) and problem size.

Figure 10 shows a schematic of the Update Engine. It is contained inside the higher level OpenMP parallel region shown in Figure 2, and all threads call it with the same input data and requested solver class. The outer *while* loop allows for iterative solvers that require any arbitrary number of passes (including messaging) to complete for a time step. The inner *while* loop contains the work necessary to drive both

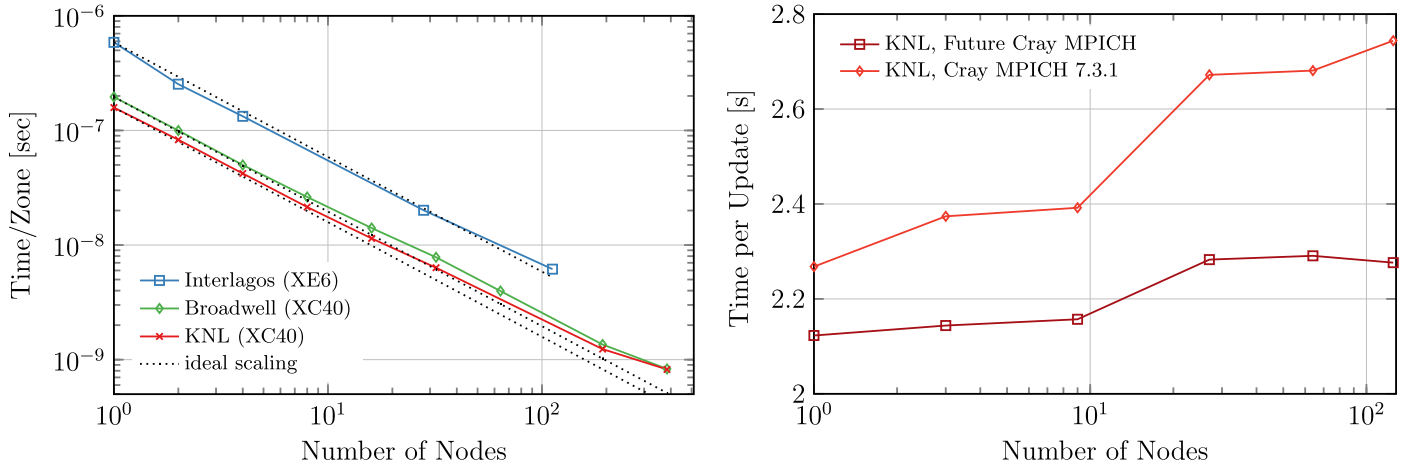

Figure 8. Left: update time per zone over number of nodes (strong scaling) for XE_{IL} (blue), XC_{BDW} (green) and XC_{KNL} (red) at constant problem size. Right: update time in seconds over number of nodes (strong scaling) for Cray MPICH 7.3.1 (light red) and new lock optimized Cray MPICH (dark red) on XC_{KNL} with 68 threads.

Table 7
Strong Scaling Test Setup

System	PPN	World Grid Patches	Base Patch Size
XE_{IL}	4	$14 \times 8 \times 12$	40^3
XC_{BDW}	2	48×12^2	32^3
XC_{KNL}	2	$32 \times 16 \times 12$	40^3

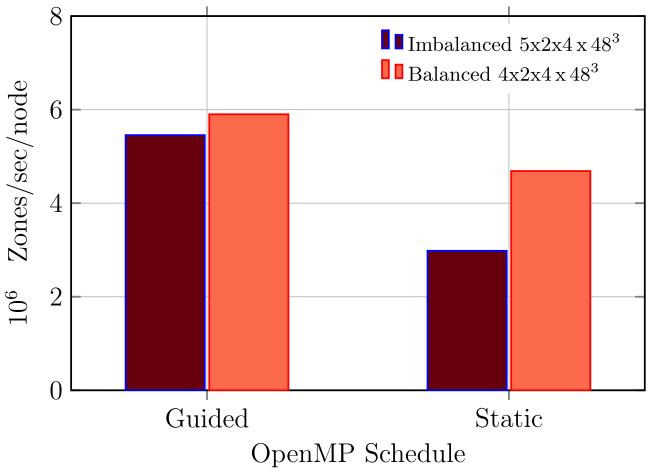

Figure 9. Comparison between the OpenMP *GUIDED* loop schedule and the *STATIC* schedule as applied to the DomainSolver class Patch update loop for a balanced (even work per thread) and imbalanced problems.

communication and updates through the requested solver. The work includes packing (unpacking) boundary data into (out of) contiguous buffers, used for either local copies within a rank or MPI transfers between ranks. There is additional work for signaling and data transfer between MPI ranks and updating Patches with resolved boundaries through the requested solver. The DomainSolver class manages all book-keeping related to marking individual boundaries for any affected Patch as resolved/unresolved. It also tracks grids within a Patch as incrementally or completely updated.

An important optimization in this design relates to how boundary data is exchanged between Patches contained in the same MPI rank. An instance of the Patch class includes a buffer for *incoming* boundary data (there is no matching buffer for outgoing data). This buffer is only used for boundary data

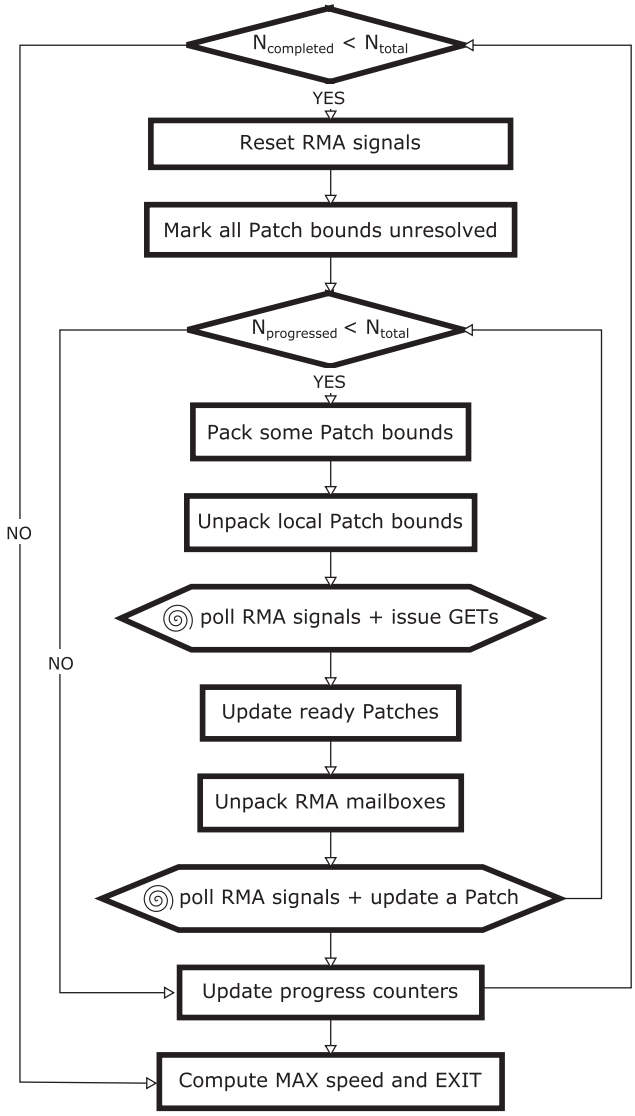

Figure 10. Generalized communication/Update Engine.

coming from another local Patch. With the Domain class, data destined for a local Patch is directly packed into the buffer of the destination Patch. The buffer is later unpacked, along with

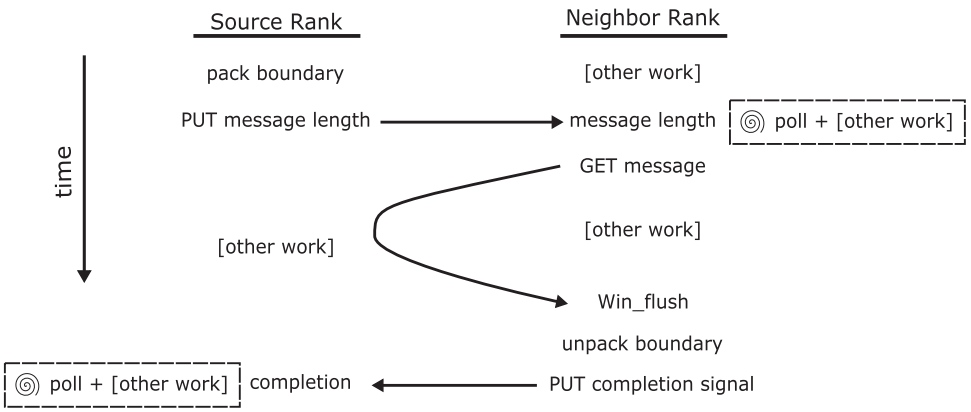

Figure 11. MPI-RMA Engine cycle.

any non-local boundary data, into the Patch grid boundary zones. This optimization takes advantage of the shared memory aspect of OpenMP, completing local Patch boundary exchange without calling MPI or excessive buffering. Some buffering is necessary to minimize contention between threads attempting to progress the same Patch. A single node run can completely avoid calling MPI with this feature by using threads on all cores.

4.1.1. MPI-RMA Engine

The MPI-RMA Engine handles non-local communication between Patches. It is generic enough to manage communication of any type of data of a wide range of message lengths with memory overheads and intensity on the network that is run-time tunable. The strategy for the MPI-RMA engine was to remove all explicit synchronization between MPI ranks and utilize all threads for both message packing/unpacking and initiation of network transfers. We use a single passive exposure epoch with MPI-RMA. The passive epoch starts with ranks calling MPI_Win_lock() for each rank it will communicate with. The communication strategy in WOMBAT does not use protections between ranks, and the lock argument to MPI_Win_lock() is always set to MPI_LOCK_SHARED. Locking and unlocking for RMA exposure is moved outside the time loop, which essentially removes their cost in exchange for minimal overhead introduced by a signaling scheme.

Figure 11 shows an overview of the steps in the MPI-RMA Engine. Operations from the point of view of both a source and neighbor rank are shown in time. Note that all source ranks are also a neighbor rank, meaning that the steps are symmetric. The process begins with a source rank packing some (not all) boundary data from a Patch into a buffer. The rank then sends an 8 byte signal to the neighbor rank with MPI_Put() indicating the size of the message that has been packed. At some point the neighbor rank starts to poll on the local address where this signal is to be deposited waiting for the value to become something other than the initial state. Reading this address must be done carefully so as not to allow the compiler to cache the value in a register. We do this by performing the read on the signal address from a simple C routine, designed to prevent any register caching from the calling Fortran code. Once the signal value is modified, the value is interpreted as the message length. If it is zero there is no message to transfer, which can happen for a variety of reasons due to the generic messaging property of the MPI-RMA Engine. If the value is greater than zero the neighbor rank initiates a network transfer with an

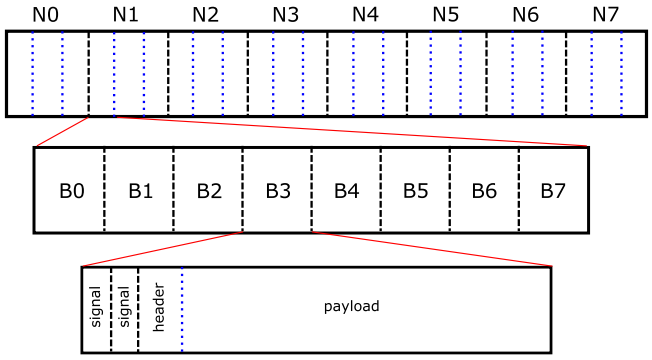

Figure 12. Anatomy of the buffer attached to the RMA window.

MPI_Get(). While the network transfer is in flight, both the source and neighbor rank do other communication or computation work. At some point later the neighbor rank needs the transfer to complete and calls MPI_Win_flush(). The message is then unpacked, and the neighbor rank then sends a signal pack to the source rank indicating that the transfer is done and the source buffer can be freely modified. The source rank eventually polls on that signal before it can repeat the full process over again.

We note that an alternative implementation of this cycle could be done entirely with MPI_Put(). In such a design, a MPI_Put() call would immediately move data to the destination rank completed sometime later with MPI_Win_flush(). Then the initial MPI_Put() above is used to signal that data is in the destination buffer. We did not use this design because it has the potential for generating more intense many-to-one traffic patterns, which can lead to degraded performance on most HPC interconnects.

The MPI-RMA Engine cycle applies to each segment of the single communication buffer that was created for the RMA window with MPI_Win_Allocate(). Multiple segments in this buffer allow for many unique messages to be exchanged with neighbor ranks. They also present potential thread parallelism for communication. The MPI-RMA Engine cycle is self-contained and can be applied to any number of independent messages to be exchanged with minimal contention or protection required between them. Multiple threads can therefore drive the engine entirely independently as long as they operate on separate buffer segments.

The single buffer attached to the RMA window in WOMBAT is decomposed into multiple regions, each available for communicating Patch boundary data. Figure 12 shows the

anatomy of this buffer with an example of a 2D Cartesian domain with nine MPI ranks (similar to the domain structure in Figure 1). The figure begins at the top looking at the entire RMA buffer logically separated into equal size segments for each of the eight neighbors (labeled N0 through N7) any rank might have. Note that it is possible for some of the logical neighbors to be the same MPI rank if the world grid is periodic. Each of these segments is further divided based on a run-time tunable value for the number of "mailboxes" dedicated to each neighbor rank. Increasing the number of mailboxes has the effect of putting more network transfers in-flight at any moment, which can reduce the number of iterations in the Engine. Each of these mailboxes is large enough to buffer all boundary data to and from one Patch (size is doubled for send and receive). It is not necessary or common that these data be from the same source Patch. In one of these mailboxes, there are eight boundary segments corresponding to the four edges and four corners that will be communicated in 2D from a Patch labeled B0 through B7. In a single section of a boundary segment, there are four distinct sections. The first two are each 8 bytes in length and are used for the incoming and outgoing signals described above and in Figure 11. The next "header" section is used to encode descriptive data about the message payload. This information includes identifying information for the Patch that should receive these boundary data. The header can be leveraged for performing other communication that might be useful to exchange between rank on a regular basis, such as load imbalance statistics or changes in the ownership of a given Patch. The final section in the boundary segment is the message payload.

The MPI-RMA Engine also has methods for initiating and completing non-blocking global reductions. They are used to compute time step sizes across all MPI ranks. Our implementation delays time step calculation by one step in order to overlap the collective with work.

4.1.2. MPI-RMA Thread Optimization in Cray MPICH

In the SPMD OpenMP model, threads do their computation, message sending, and message completion asynchronously, so contention on the interconnect resources becomes relevant to performance. On Cray XC systems the Aries interconnect provides 128 hardware "lanes" called communication domains (CDMs) for concurrent message transfers and synchronizations (although MPI does not always make use of all of them). The MPI library assigns these CDMs either statically to threads the first time a thread makes an MPI call, or dynamically each time a message is sent or completed.

In SPMD OpenMP, static assignment of CDMs to threads is no longer feasible, because it provides no means for the MPI library to dynamically minimize contention. For example, if a thread needs to complete all messages targeting a specific remote rank, it may need access to several CDMs that have been statically assigned to other threads before. Safe access to these CDMs could be handled with a mutex, but doing so can force other threads to wait for access to the CDM before sending a message. Hence, dynamic allocation of CDMs is required to minimize overhead from CDM contention and maximize performance.

We have adapted Cray's MPI-RMA implementation to use lock-free dynamic allocation of CDMs. It is now designed specifically to minimize overhead due to CDM assignment and maximize performance for SPMD approaches. The library guarantees contention-free communication as long as the number of concurrent requests to send and/or complete a message does not exceed the available number CDMs.

5. Numerical Methods

For an initial implementation in the code design discussed above we use a second-order, directionally un-split version of the non-relativistic ideal MHD solver described in RJ95 and Ryu et al. (1998) referred to as MHDTVD. This new implementation follows the CTU+CT scheme, described in Gardiner & Stone (2005) (hereafter GS05) and Gardiner & Stone (2008) (hereafter GS08), modified for the MHDTVD solver. The algorithm outlined here solves the equations of MHD neglecting charge separation between ions and electrons, electrical resistivity, viscosity, and non-adiabatic processes, such as thermal conduction. With these assumptions the ideal MHD equations are

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{v}) = 0, \tag{1}$$

$$\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} + \frac{1}{\rho} \boldsymbol{\nabla} P - \frac{1}{\rho} (\boldsymbol{\nabla} \times \boldsymbol{B}) \times \boldsymbol{B} = 0, \tag{2}$$

$$\frac{\partial P}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} P + \gamma P \, \boldsymbol{\nabla} \cdot \boldsymbol{v} = 0, \tag{3}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} - \boldsymbol{\nabla} \times (\boldsymbol{v} \times \boldsymbol{B}) = 0, \tag{4}$$

where γ is the plasma adiabatic index. Following the convention of RJ95, we have selected our units such that 4π does not appear in these equations. For a one-dimensional flow along the X direction, Equations (1)–(4) can be written in the conservative form

$$\frac{\partial \boldsymbol{q}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} = 0, \tag{5}$$

where $\boldsymbol{q}$ and $\boldsymbol{F}$ are the state vector and flux vector respectively defined as

$$\boldsymbol{q} = \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ B_x \\ B_y \\ B_z \\ E \end{pmatrix}, \tag{6}$$

$$\boldsymbol{F} = \begin{pmatrix} \rho v_x \\ \rho v_x^2 + P^* - B_x^2 \\ \rho v_x v_y - B_x B_y \\ \rho v_x v_z - B_x B_z \\ 0 \\ B_y v_x - B_x v_y \\ B_z v_x - B_x v_z \\ (E + P^*) v_x - B_x (B_x v_x + B_y v_y + B_z v_z) \end{pmatrix}. \tag{7}$$

The total pressure and total energy are given by

$$P^* = P + \frac{1}{2}(B_x^2 + B_y^2 + B_z^2) \tag{8}$$

$$E = \frac{P}{\gamma - 1} + \frac{\rho}{2}(v_x^2 + v_y^2 + v_z^2) + \frac{1}{2}(B_x^2 + B_y^2 + B_z^2). \quad (9)$$

A source term vector can be added to Equation (5) to include additional physics, such as gravity, geometry corrections, cooling, and cosmic-ray feedback. This system of equations is hyperbolic under the definition that the Jacobian matrix, $\boldsymbol{A} = \partial \boldsymbol{F}/\partial \boldsymbol{q}$, has all real eigenvalues and a complete set of right eigenvectors. This system is not strictly hyperbolic, however, due to conditions that can produce degenerate eigenvalues. The seven eigenvalues $a_{1,7} = v_x \pm c_f$, $a_{2,6} = v_x \pm c_a$, $a_{3,5} = v_x \pm c_s$, and $a_4 = v_x$ correspond to three MHD wave families and an entropy mode. The characteristic wave speeds are

$$c_f = \left(\frac{1}{2}\left[a^2 + \frac{B_x^2 + B_y^2 + B_z^2}{\rho} + \sqrt{\left(a^2 + \frac{B_x^2 + B_y^2 + B_z^2}{\rho}\right)^2 - 4a^2\frac{B_x^2}{\rho}}\right]\right)^{1/2} \quad (10)$$

$$c_s = \left(\frac{1}{2}\left[a^2 + \frac{B_x^2 + B_y^2 + B_z^2}{\rho} - \sqrt{\left(a^2 + \frac{B_x^2 + B_y^2 + B_z^2}{\rho}\right)^2 - 4a^2\frac{B_x^2}{\rho}}\right]\right)^{1/2} \quad (11)$$

$$c_a = \sqrt{\frac{B_x^2}{\rho}}, \quad (12)$$

where the sound speed is defined as $a = \sqrt{\gamma P/\rho}$. One of the difficulties in solving Equation (5) is that some of the eigenvalues will coincide in limiting cases and special care must be taken to avoid singularities around points where $B_x = 0$ or $B_y = B_z = 0$ (RJ95). We summarize the one-dimensional MHDTVD algorithm in Section A.

5.1. MHD in Two Dimensions

The 2D directionally un-split update closely follows the steps for the CTU+CT scheme described in GS05. Our implementation utilizes five boundary zones, and requires only one boundary exchange per time step for both state variables and zone corner EMFs. Given a time step Δt, the steps in the algorithm are:

Step 1. Compute the directionally split fluxes in both X and Y directions using initial states $\boldsymbol{q}^n$ from Equation (14) for a time step Δt.
Step 2. Compute a zone-centered reference EMF for use in the mid-time step constrained transport update of the face-centered magnetic field. The EMF is given by $v_x \times B_y + v_y \times B_x$ with each input derived from the initial state vector $\boldsymbol{q}^n$.
Step 3. Using the upwinded algorithm in GS05, compute EMF values at zone corners using the B_y and B_x fluxes from the X and Y passes from Step 1 and the reference EMF from Step 2.
Step 4. Update the face centered magnetic field $\boldsymbol{b}^n$ to $\boldsymbol{b}^{n+1/2}$ from the EMFs in Step 3 over $\Delta t/2$.
Step 5. Update the zone centered state vector from the initial states $\boldsymbol{q}^n$ to $\boldsymbol{q}_x^{n+1/2}$ using fluxes from the Y pass in Step 1 applied over $\Delta t/2$. Include the $\nabla \cdot B$ source term vector described by GS05.
Step 6. Using the preconditioned state $\boldsymbol{q}_x^{n+1/2}$, compute fluxes along X from Equation (14) for a time step Δt.
Step 7. Repeat steps 5 and 6 for the Y direction.
Step 8. Compute a zone-centered reference EMF for use in the final CT update of the face-centered magnetic field. The EMF is given by $v_x \times B_y + v_y \times B_x$ with v_x and v_y coming from an un-split update of $\boldsymbol{q}^n$ to $\boldsymbol{q}^{n+1/2}$ using the fluxes from Steps 6 and 7.
Step 9. Using the upwinded algorithm in GS05, compute EMF values at zone corners using the B_y and B_x fluxes from the X and Y passes from Steps 6 and 7 and the reference EMF from Step 8.
Step 10. Use an un-split update of the state vector $\boldsymbol{q}^n$ to $\boldsymbol{q}^{n+1}$ using fluxes from Steps 6 and 7 applied over Δt.
Step 11. Update the face centered magnetic field $\boldsymbol{b}^n$ to $\boldsymbol{b}^{n+1}$ from the EMFs in Step 9 over Δt. Update the zone centered magnetic field from averages of the face centered magnetic field as described in GS05.

5.2. MHD in Three Dimensions

The 3D un-split update is based on the 6-solve algorithm described in GS08. We again utilize five boundary zones as described above for 2D. The steps in the 3D algorithm are:

Step 1. Compute the directionally split fluxes in the X, Y and Z directions using initial states $\boldsymbol{q}^n$ from Equation (14) for a time step Δt.
Step 2. Compute zone-centered reference EMFs for use in the mid-time step constrained transport update of the face-centered magnetic field using inputs derived from the initial state vector $\boldsymbol{q}^n$.
Step 3. Using the upwinded algorithm in GS08, compute EMF values at zone corners using the magnetic fluxes from the directional passes from Step 1 and the reference EMF from Step 2.
Step 4. Update the face centered magnetic field $\boldsymbol{b}^n$ to $\boldsymbol{b}^{n+1/2}$ from the EMFs in Step 3 over $\Delta t/2$.
Step 5. Update the zone centered state vector from the initial states $\boldsymbol{q}^n$ to $\boldsymbol{q}_x^{n+1/2}$ with an un-split update using fluxes from the Y and Z passes in Step 1 applied over $\Delta t/2$. Include the $\nabla \cdot B$ source term vector described by GS08.
Step 6. Using the preconditioned state $\boldsymbol{q}_x^{n+1/2}$, compute fluxes along X from Equation (14) for a time step Δt.
Step 7. Repeat steps 5 and 6 for the Y and Z directions using the appropriate transverse fluxes from Step 1.
Step 8. Compute zone-centered reference EMFs for use in the final CT update of the face-centered magnetic field. Velocity values come from an un-split update of $\boldsymbol{q}^n$ to $\boldsymbol{q}^{n+1/2}$ using the fluxes from Steps 6 and 7.
Step 9. Using the upwinded algorithm in GS08, compute EMF values at zone corners using the magnetic fluxes from the directional passes from Steps 6 and 7 and the reference EMF from Step 8.
Step 10. Use an un-split update of the state vector $\boldsymbol{q}^n$ to $\boldsymbol{q}^{n+1}$ using fluxes from Steps 6 and 7 applied over Δt.
Step 11. Update the face centered magnetic field $\boldsymbol{b}^n$ to $\boldsymbol{b}^{n+1}$ from the EMFs in Step 9 over Δt. Update the zone

centered magnetic field from averages of the face centered magnetic field as described in GS08.

6. Test Calculations

6.1. Linear Wave Convergence

We performed linear wave convergence tests using eigenvectors of the Roe matrices for hydrodynamics and MHD following the setup used by GS05. For one-dimensional tests, we use a periodic domain $L = 1$ divided into N zones containing a background fluid with $\rho = 1$, $P = 3/5$, and $\gamma = 5/3$. The background is at rest for shear and entropy waves, otherwise $v_x = 1$. For hydrodynamic waves (sound, v_y and v_z shear, and entropy modes), $B_x = B_y = B_z = 0$, while the background for MHD waves (slow, Alfvén, fast, and entropy modes) has magnetic field components $B_x = 1$, $B_y = \sqrt{2}$, $B_z = 1/2$. A sinusoidal perturbation is applied to this background state, such that the initial state vector is given by $\boldsymbol{q}_0 = \bar{\boldsymbol{q}} + A_0 \boldsymbol{R}_k \cos(2\pi x)$, where $\bar{\boldsymbol{q}}$ is the background state, $A_0 = 10^{-6}$ is the amplitude, and R_k is the right eigenvector for the wave mode k.

Each wave is propagated for one wavelength, and then the error in the solution is computed using the L_1 error vector averaged over every zone i, defined by $\delta \boldsymbol{q} = N^{-1} \sum_i |\boldsymbol{q}_i - \boldsymbol{q}_{i,0}|$. Increasing the number of zones up to $N = 1024$, the solution for each wave mode in 1D converges with second-order accuracy as seen by the norm of the L_1 error vector in Figure 13.

We also tested the convergence of MHD waves propagating oblique to a 3D grid, following the setup in GS08. The wave is initialized rotated with respect to a computational grid of size $(L_X, L_Y, L_Z) = (3, 3/2, 3/2)$ with $2N \times N \times N$ zones, such that the wave vector is $\boldsymbol{k} = (1/3, 2/3, 2/3)$. The face-centered magnetic field components are initialized via a vector potential defined at the corners of the grid zones, and then zone-centered magnetic field values are averaged from face-centered fields. After propagating one wavelength, the L_1 error vector is computed with respect to the initial conditions. The convergence with increasing resolution is shown in Figure 13.

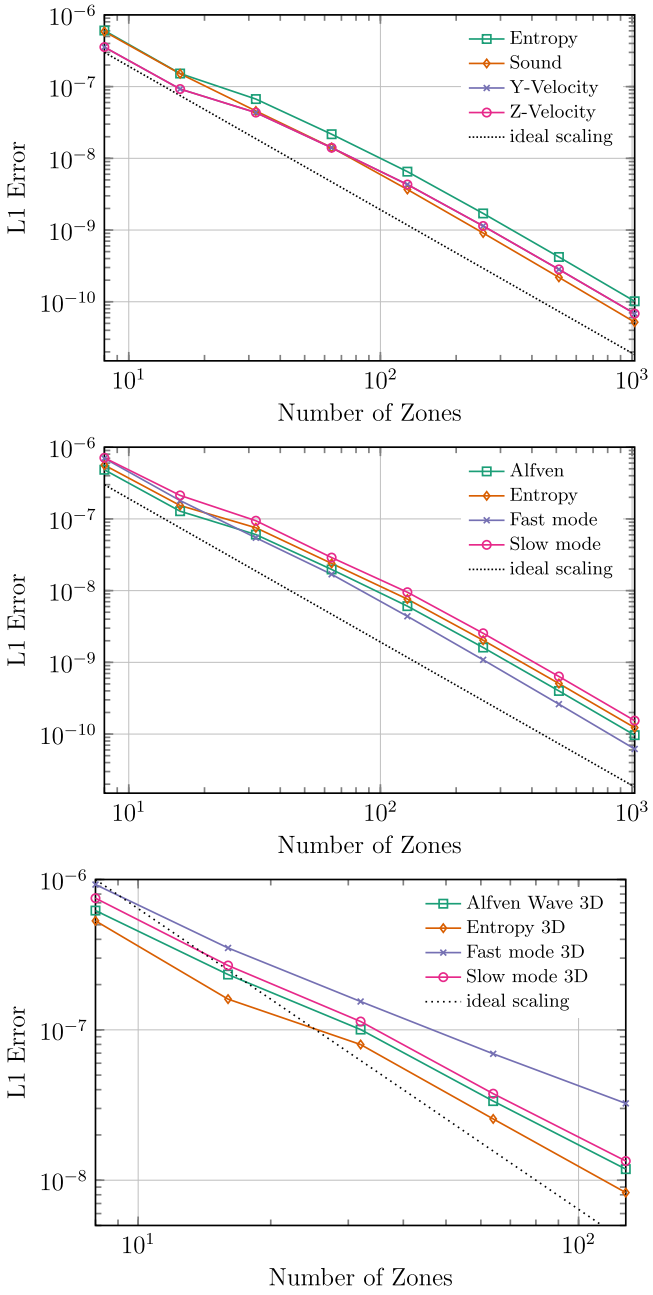

Figure 13. Convergence of the norm of the L_1 error vector for wave modes after propagating one wavelength. The dotted line shows a slope of -2 for comparison. The top plot shows hydrodynamic wave modes in 1D. The middle plot shows MHD wave modes in 1D. The bottom plot shows MHD wave modes in 3D oblique to the grid.

6.2. RJ95 2a

The next set of tests are of the shock-tube setup 2a from RJ95. The left-hand state was initialized with $(\rho, v_x, v_y, v_z, B_y, B_z, P) = [1.08, 1.2, 0.01, 3.6/(4\pi)^{1/2}, 2/(4\pi)^{1/2}, 0.95]$, and the right-hand state with $[1, 0, 0, 0, 4/(4\pi)^{1/2}, 2/(4\pi)^{1/2}, 1]$. For this test $B_x = 2/(4\pi)^{1/2}$. Figures 14 and 15 shows the evolved grid at $t = 0.2$ in 1d and 3d respectively. The shock normal is rotated 45° out of all primary planes in 3D.

6.3. RJ95 4a

Figures 16 and 17 show the results at $t = 0.15$ of the 4a setup from RJ95. Figure 16 is the 1D result, and Figure 17 is the 3D result with the shock normal rotated 45° out of all primary planes. The left-hand state was initialized with $(\rho, v_x, v_y, v_z, B_y, B_z, P) = [1, 0, 0, 0, 1, 0, 1]$, and the right-hand state with $[0.2, 0, 0, 0, 0, 0, 0.1]$. For this test $B_x = 1$.

6.4. Brio & Wu Shock Tube

We performed the well-known MHD shock tube test of Brio & Wu (1988) on a 1D domain. The left-hand state is initialized with the state vector $(\rho, v_x, v_y, v_z, B_y, B_z, P) = (1.0, 0, 0, 0, 1.0, 0, 1.0)$, while the initial right-hand state is $(\rho, v_x, v_y, v_z, B_y, B_z, P) = (0.125, 0, 0, 0, -1.0, 0, 0.1)$. Throughout the domain, $B_x = 0.75$, while the adiabatic index $\gamma = 2$. The solution computed on a domain with 400 zones at $t = 0.08$ compared to a better converged solution computed with 10^4 zones is shown in Figure 18.

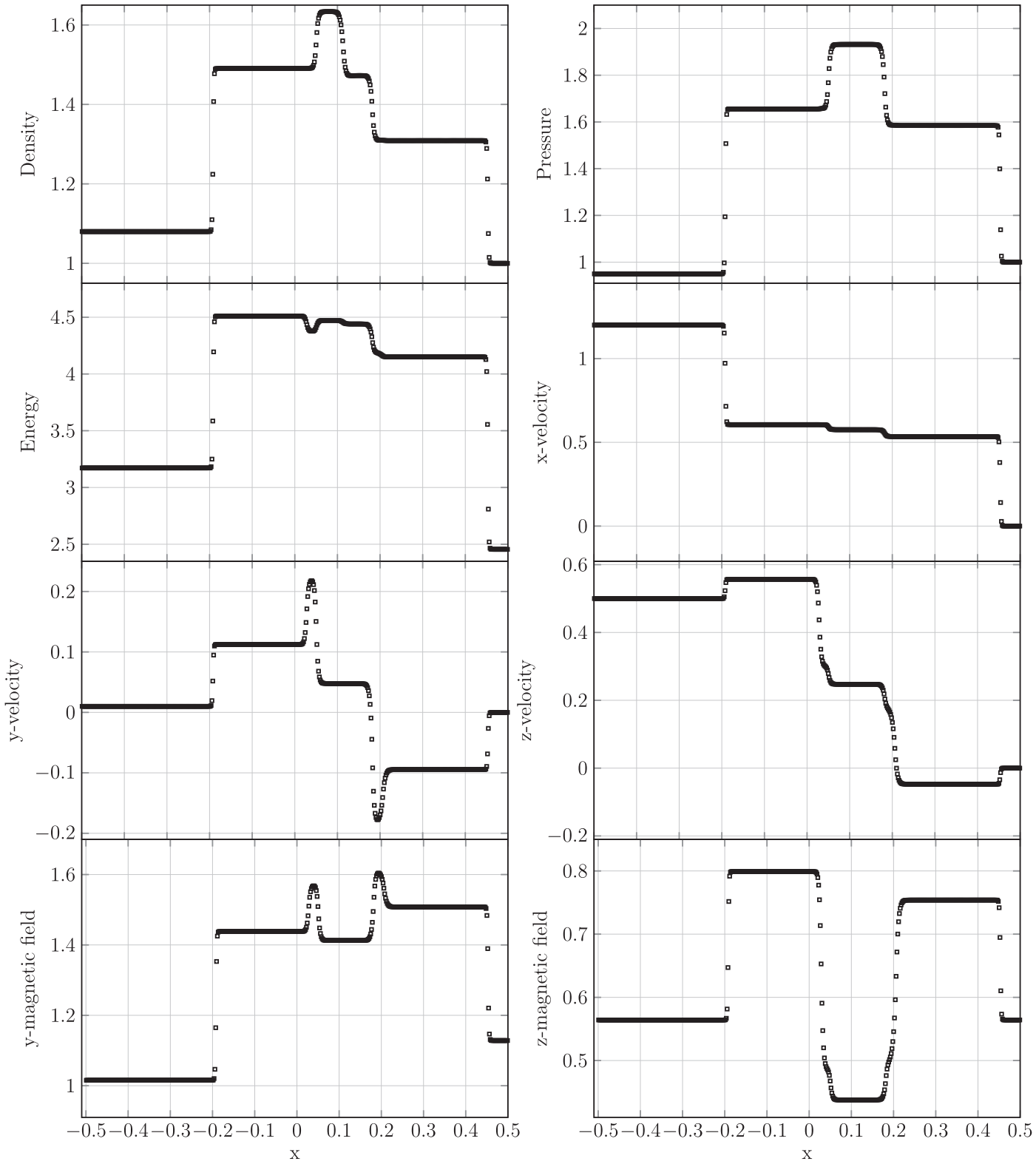

Figure 14. RJ95 shock tube test 2a in 1D with 512 zones at $t = 0.2$. Slices of density, pressure, energy, velocity components, and magnetic field components are shown from top left to bottom right.

6.5. Orszag–Tang Vortex

A very common test in 2D for an MHD code is the compressible Orszag–Tang vortex. This problem was first studied by Orszag & Tang (1979) and is now used as a standard comparison of MHD codes (Stone et al. 2008). The setup for this problem uses a periodic box with $L_X = [-0.5, 0.5]$ and $L_Y = [-0.5, 0.5]$ and 192×192 zones. Uniform density and pressure are initialized throughout the grid with

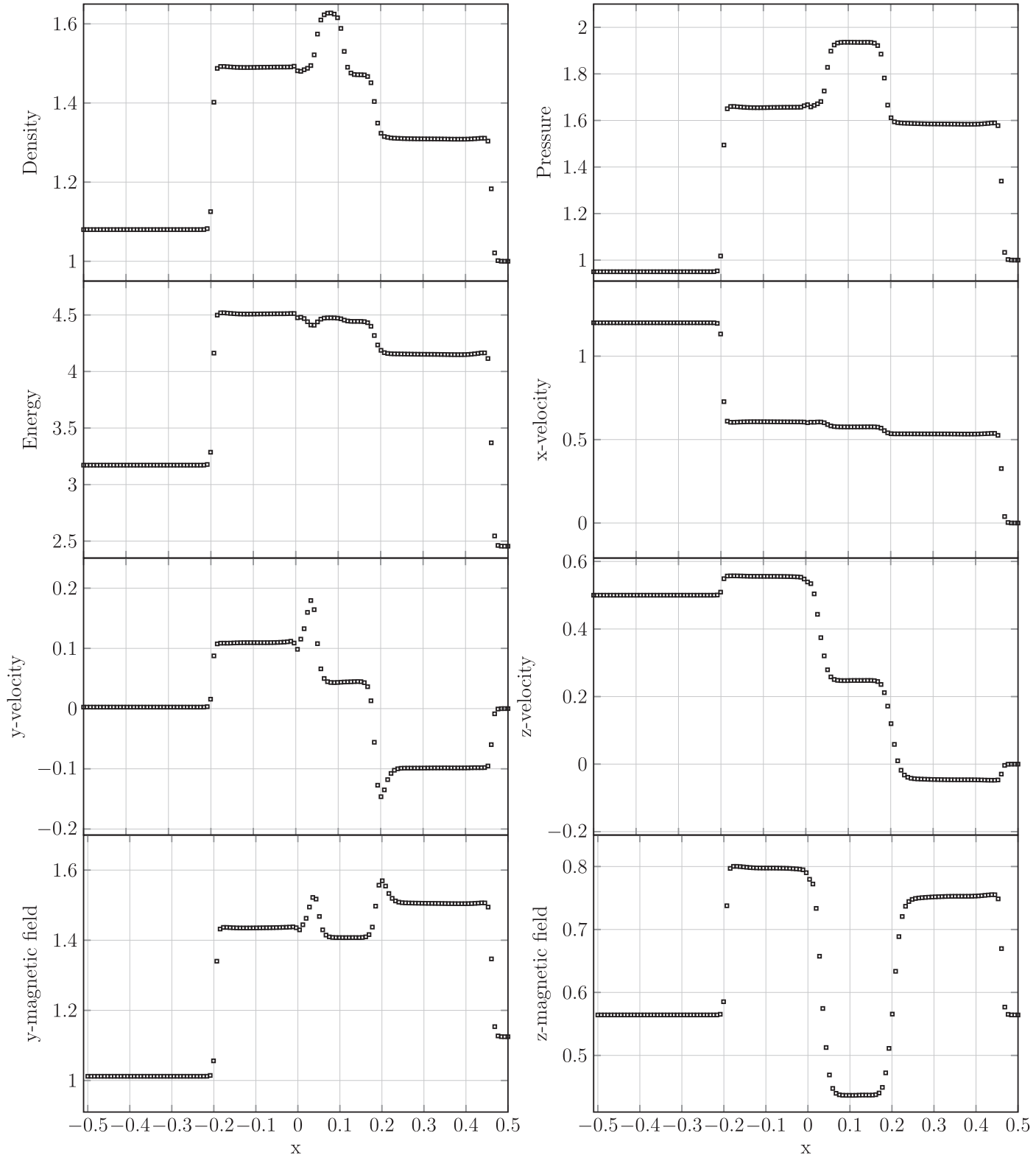

Figure 15. RJ95 shock tube test 2a in 3D with 128 zones at $t = 0.2$. Slices of density, pressure, energy, velocity components, and magnetic field components are shown oblique to the grid from top left to bottom right.

$\rho = 25/36\pi$, $P = 5/12\pi$ and $\gamma = 5/3$, giving a sound speed of $c_s = 1$. The velocity was initialized as $v_x = -v_0 \sin(2\pi y)$ and $v_y = v_0 \sin(2\pi x)$, where $v_0 = 1$. The magnetic field along zone faces was derived from the vector potential defined at zone corners $A_z = B_0[\cos(4\pi x)/2 + \cos(2\pi y)]/2\pi$, where $B_0 = 1/(4\pi)^{1/2}$, with $\boldsymbol{b} = \nabla \times \boldsymbol{A}$. Figure 19 shows the resulting density, gas pressure, specific kinetic energy, and magnetic pressure at $t = 0.5$, as well as slices of the gas pressure at $y = -0.0723$ and $y = -0.1875$.

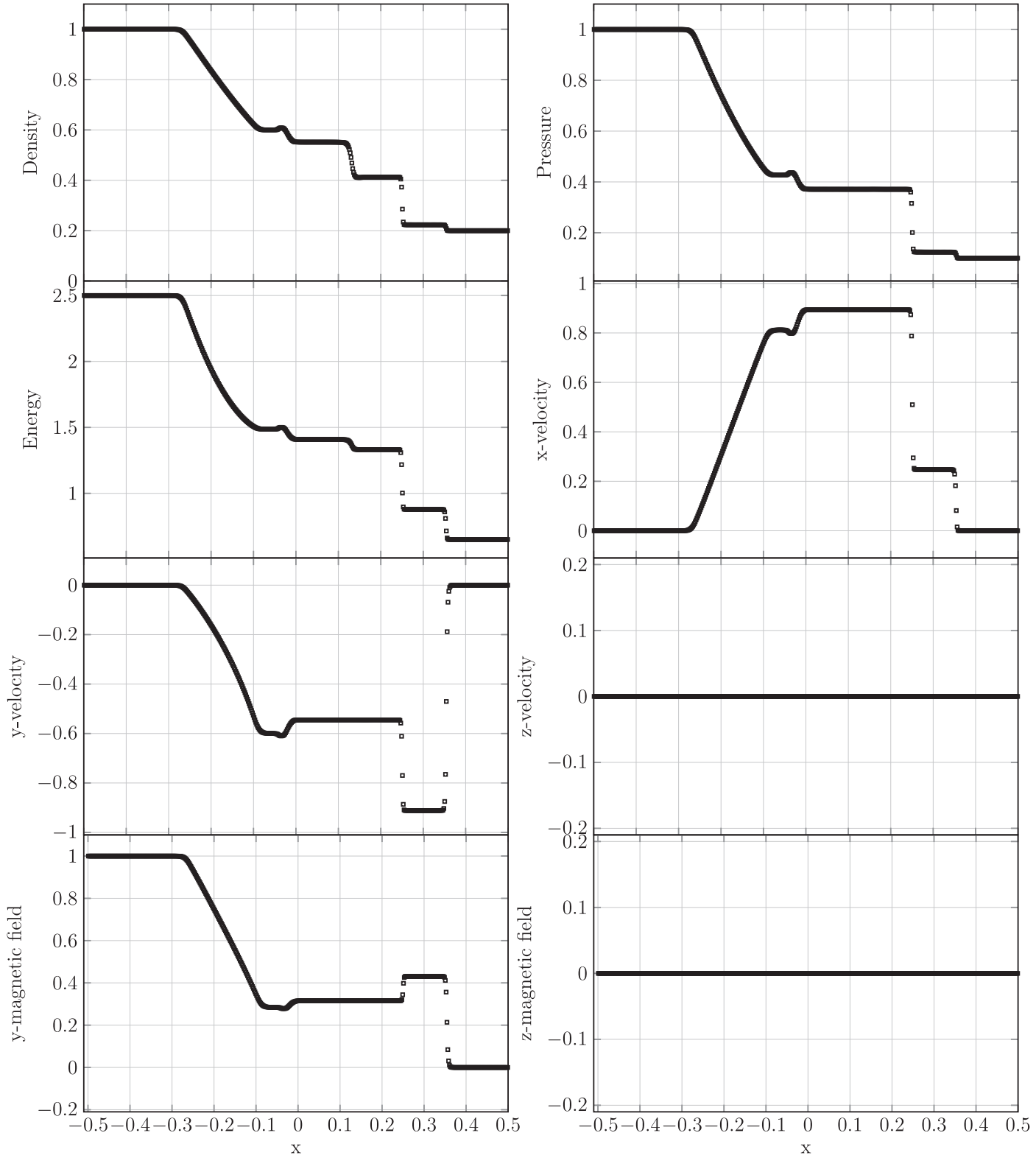

Figure 16. RJ95 shock tube test 4a in 1D with 512 zones at $t = 0.15$. Density, pressure, energy, velocity components, and magnetic field components are shown from top left to bottom right.

6.6. MHD Rotor

Another common MHD test problem in 2D is that of a rotating disk in a magnetized medium (Balsara & Spicer 1999). We follow the setup used by Stone et al. (2008) and defined in Tóth (2000) as "Rotor Problem 1" on a periodic domain with 400×400 zones. Distributions of density, pressure, Mach number, and magnetic pressure for the solution at $t = 0.15$ is shown in Figure 20, along with slices of the y-component of the magnetic field at $y = 0$ and the x-component of the magnetic field at $x = 0$.

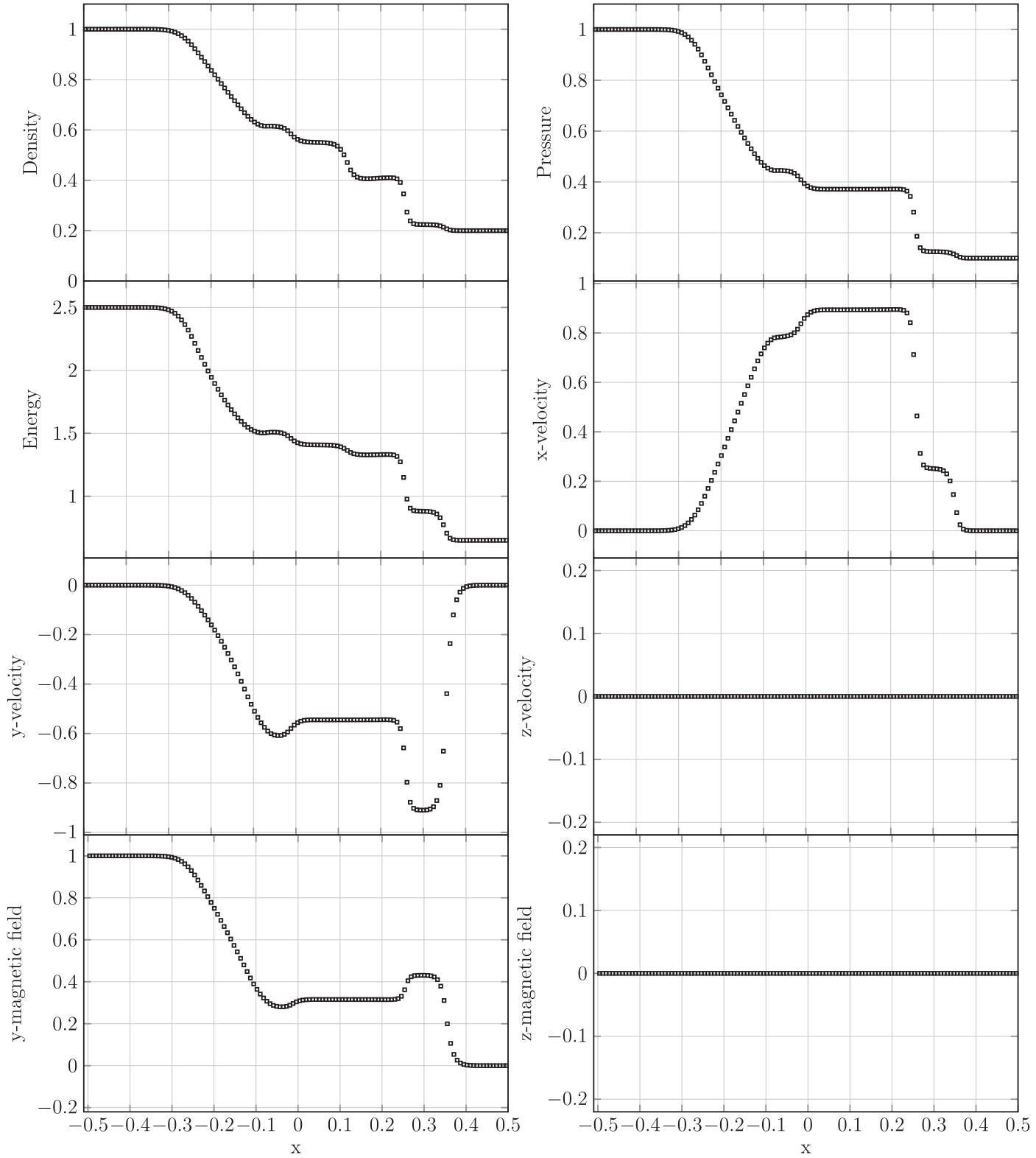

Figure 17. RJ95 shock tube test 4a in 3D with 128 zones at $t = 0.15$. Density, pressure, energy, velocity components, and magnetic field components are shown oblique to the grid from top left to bottom right.

6.7. Advection of a Field Loop

A powerful test of an MHD code's ability to keep $\nabla \cdot \boldsymbol{B} = 0$ is the advection of a weak magnetic field loop. We use a setup similar to that of GS05 for a 2D calculation. A periodic box with $L_X = [-1., 1.]$ and L_Y $[-0.5, 0.5]$ over 256×128 zones was initialized with $\rho = 1$, $P_{\rm gas} = 1$, $v_x = 2$, and $v_y = 0.5$. The magnetic field was derived from a vector potential defined at zone corners as $A_z = \mathrm{MAX}(A[R_0 - r], 0)$ where $A = 10^{-3}$ and $R_0 = 0.3$. This field produces a line current through the center of the loop and a return current along R_0, but these

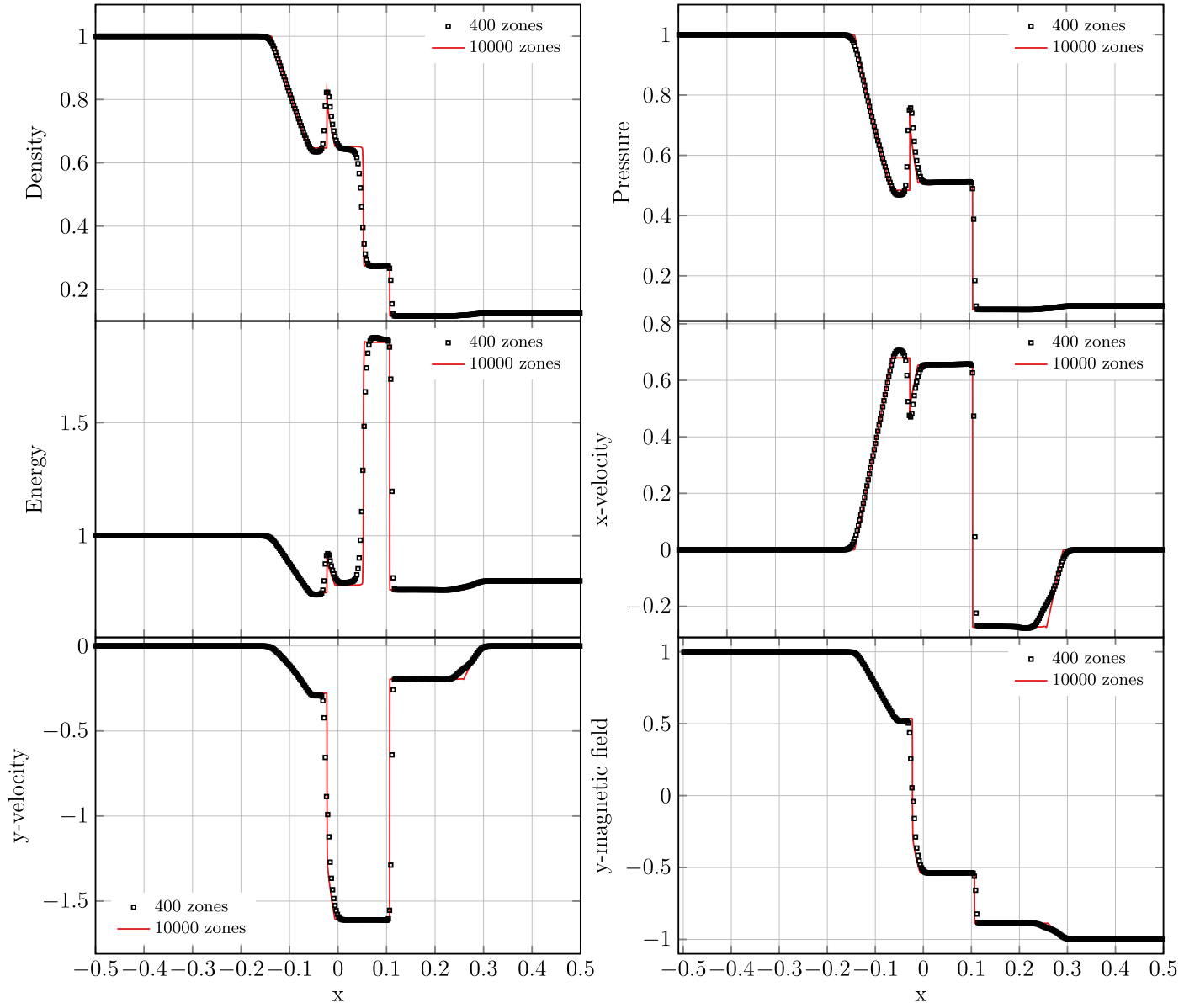

Figure 18. Brio & Wu shock tube test at $t = 0.08$. The plot for energy is derived from the ratio of pressure to density. The points represent the solution computed on a 400 zone domain, while the solid line is the solution computed on a 10^4 zone domain.

features are unresolved on the grid. Figure 21 shows the 2D result after two periods.

We perform a 3D version of this test, also shown in Figure 21, following the setup used in GS08.

6.8. MHD Blast Wave

We performed a 3D version of the 2D magnetized strong blast wave test as defined in Londrillo & Del Zanna (2000). The test is performed on a periodic domain with $(L_X, L_Y, L_Z) = (1, 3/2, 1)$ using $200 \times 300 \times 200$ zones. The fluid is initialized at rest with $\rho = 1$ and a uniform magnetic field $(B_x, B_y, B_z) = (10/\sqrt{2}, 10/\sqrt{2}, 0)$. The fluid has a pressure $P = 1$, except for in the central region within $r_0 = 0.125$ where $P = 100$. Figure 22 shows the density, specific kinetic energy, and magnetic energy of the solution in a slice through $z = 0$ at $t = 0.02$.

6.9. Circularly Polarized Alfvén Wave

As a final MHD test, we show the propagation of a circular polarized Alfvén wave as described by Tóth (2000). This test was used by Tóth to compare the performance of various approaches to maintaining $\nabla \cdot \boldsymbol{B} = 0$. The test can be done in one or more dimensions, and it can be used for convergence testing as it is an exact nonlinear solution to the equations of MHD. The grid is initialized with $\rho = 1$, $P_{\rm gas} = 0.1$, $v_y = 0.1 \sin(2\pi x)$, $B_y = 0.1 \sin(2\pi x)$, $v_z = B_z = 0.1 \cos(2\pi x)$, $B_x = 1$, and $v_x = 0$. For the 2D tests we rotate these properties on the grid by an angle of $\theta = \tan^{-1}(2)$, while in the 3D tests we perform the same rotation as the 3D tests in Section 6.1. The grid was a periodic box with $L_X = [-\sqrt{5}/2, \sqrt{5}/2]$ and $L_Y = 0.5 * L_X$ with $2N \times N$ zones in 2D and $(L_X, L_Y, L_Z) = (3, 3/2, 3/2)$ with $2N \times N \times N$ in 3D. The left panel of Figure 23 shows the convergence of the L_1 error vector norm after one wave period for the 2D and 3D tests with increasing

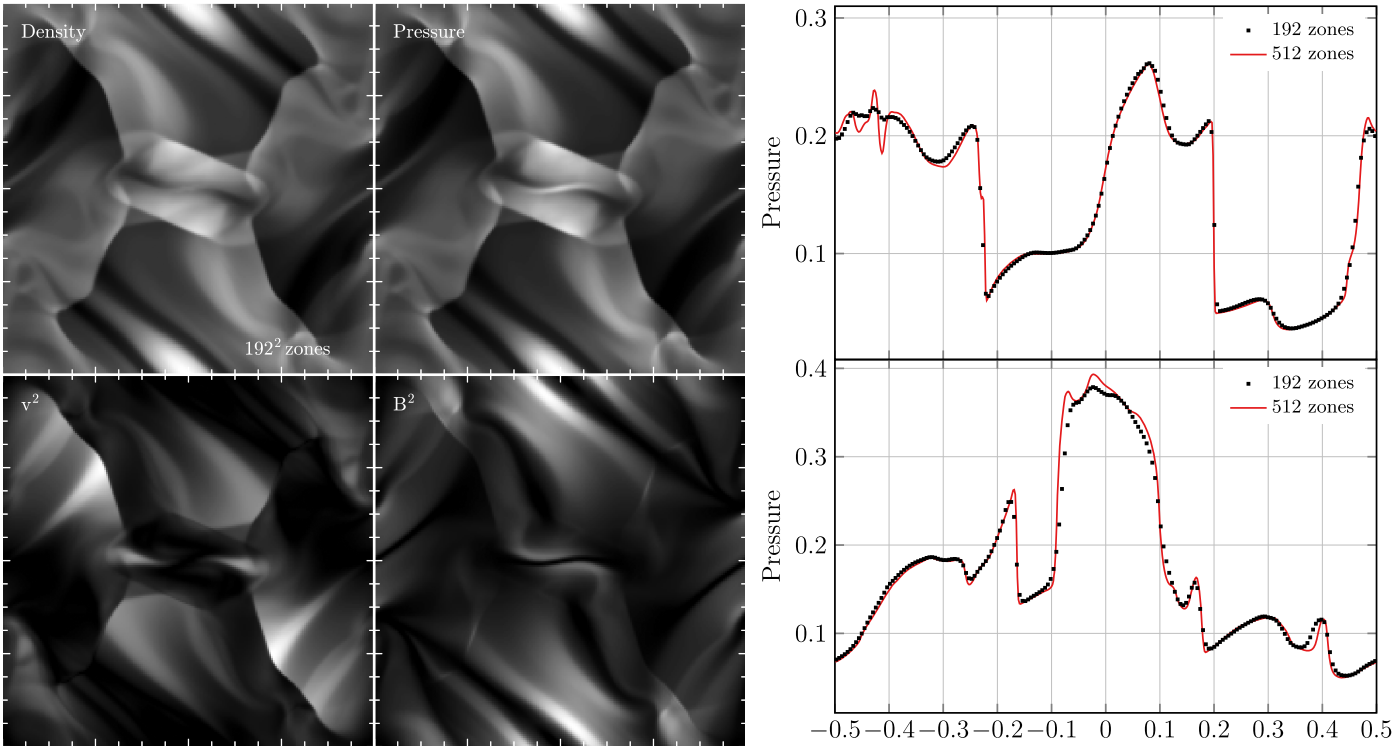

Figure 19. Images of selected quantities (left) and slices of pressure (right) for the Orszag–Tang vortex test at $t = 0.5$. Each quantity is scaled black to white linearly from the minimum value to maximum value for a solution computed on a 192 × 192 zone domain. The slices are at $y = -0.1875$ (top) and $y = -0.073$ (bottom) and compare the solution computed with 192 × 192 zones to a solution computed on a domain with 512 × 512 zones (red lines).

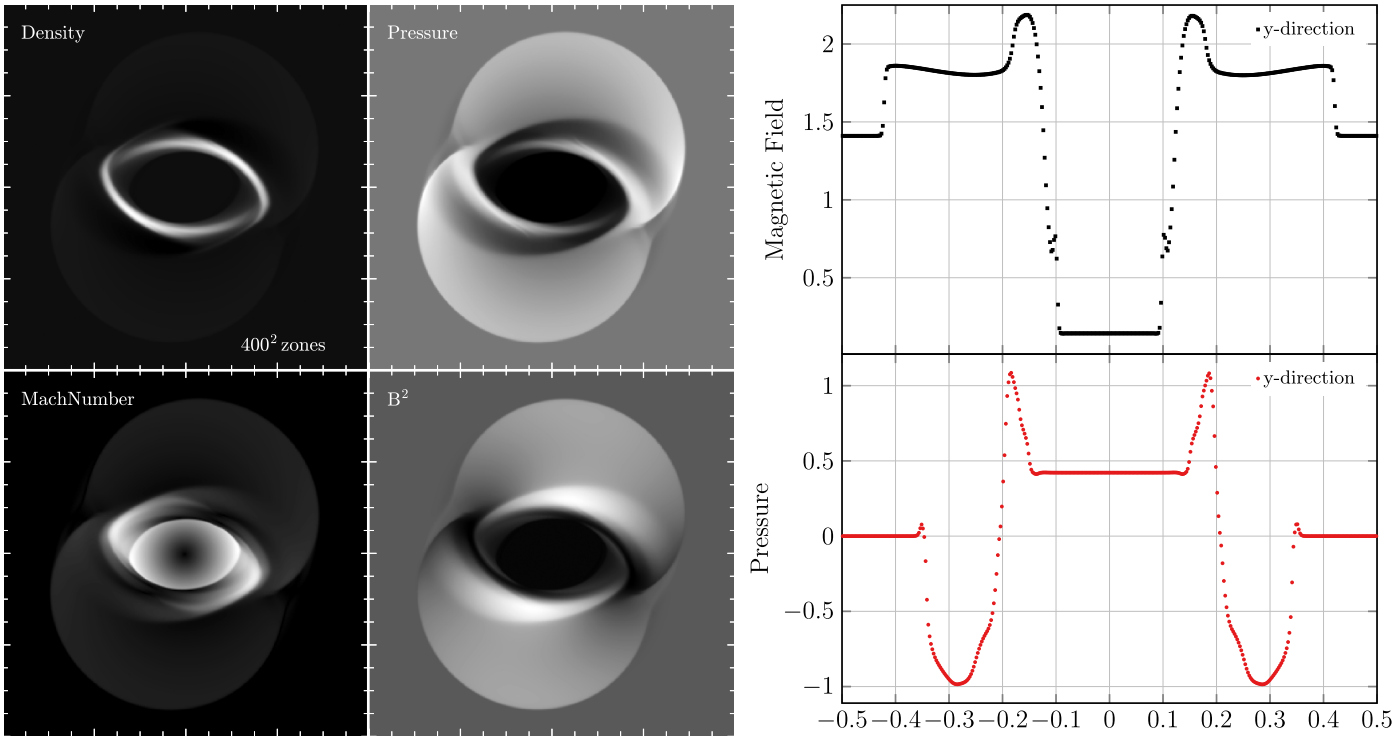

Figure 20. Images of selected quantities (left) and slices of magnetic field components (right) for the MHD rotor test at $t = 0.15$. Each quantity is scaled black to white from the maximum to minimum value for a solution computed on a 400 × 400 zone domain. The top slice is taken at $y = 0$ and shows the y-component of the magnetic field, and the bottom slice is taken at $x = 0$ and shows the x-component of the magnetic field.

resolution, where the horizontal axis represents the number of zones across the shorter dimensions. The right panel shows, using every zone in the 2D tests, profiles of the in-plane transverse component of the magnetic field in the rotated frame (B_2) after five wave periods, with the horizontal axis representing the x-coordinate in the rotated reference frame.

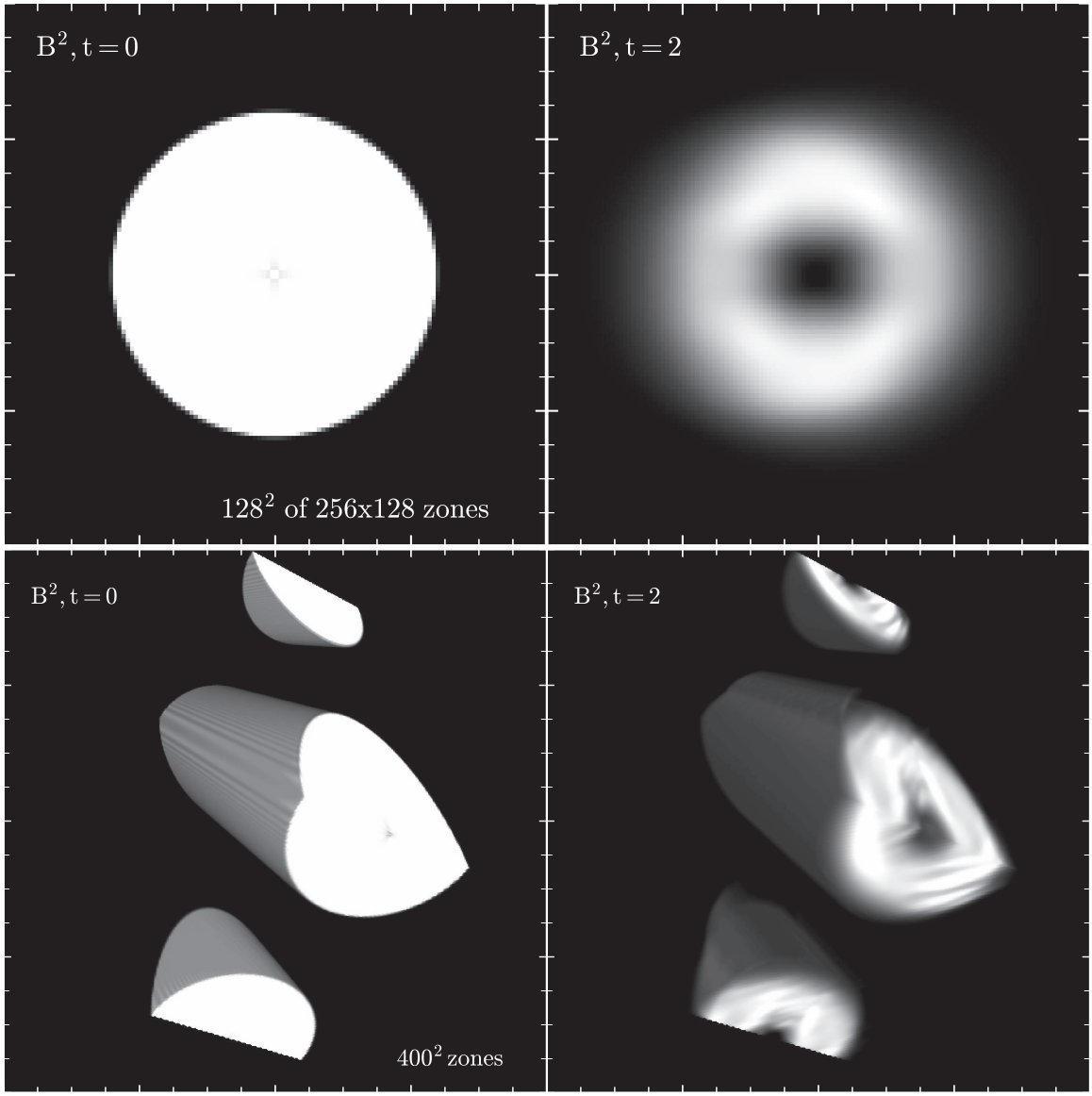

Figure 21. Images of magnetic pressure for the advection of a magnetic field loop. The top left image shows the initial conditions and the top right the solution after two periods across a 2D domain with 256 × 128 zones. The bottom left image shows the initial conditions and the bottom right the solution after two periods for a 3D domain with $200^2 \times 300$ zones. Each image is scaled linearly from $[0, 10^{-6}]$.

The lack of scatter in these plots demonstrates that the rotated wave fronts remain coherent.

7. Conclusions

In this paper we present the design and performance of a new hybrid MPI/OpenMP astrophysical MHD code called WOMBAT. We are developing WOMBAT for broad application in astrophysics, but especially in support of investigations of cosmological turbulence and the evolution of magnetic fields in galaxy clusters, where conductive fluid behaviors must be captured with good fidelity on a very wide range of scales. This requirement demands that WOMBAT have exceptional performance and scaling on the latest generation of HPC systems. We also argue in Section 1 that the ability to scale to high thread counts is crucial to maintaining high performance for the target simulations. This is particularly important for mesh refinement and *N*-body extensions of WOMBAT currently in development, where load imbalance is unavoidable. This work will be presented in a follow-up to this paper. The optimization strategies incorporated into WOMBAT are based on the Patch, a the basic unit of work and domain decomposition within an MPI rank. Patches are self-contained problems with their own boundary zones and meta-data necessary to evolve them in time. These properties make Patches ideal for presenting independent work to threads within a rank. We presented the SPMD OpenMP design of WOMBAT, where only a single OpenMP parallel region exists for the duration of code execution. Threads update Patches and perform all boundary communication collaboratively with the Update and MPI-RMA Engines discussed in Section 4. We present a unique enhancement of the Cray MPICH library through a co-design effort with Cray, Inc. and the University of Minnesota. The "thread-hot" MPI-RMA feature (see Section 4.1.2) results in significant speedup of WOMBAT because of its lock-free design.

We show the performance characteristics of WOMBAT on several architectures including the latest generation of Intel Xeon Phi "Knights Landing" processors. WOMBAT scaling

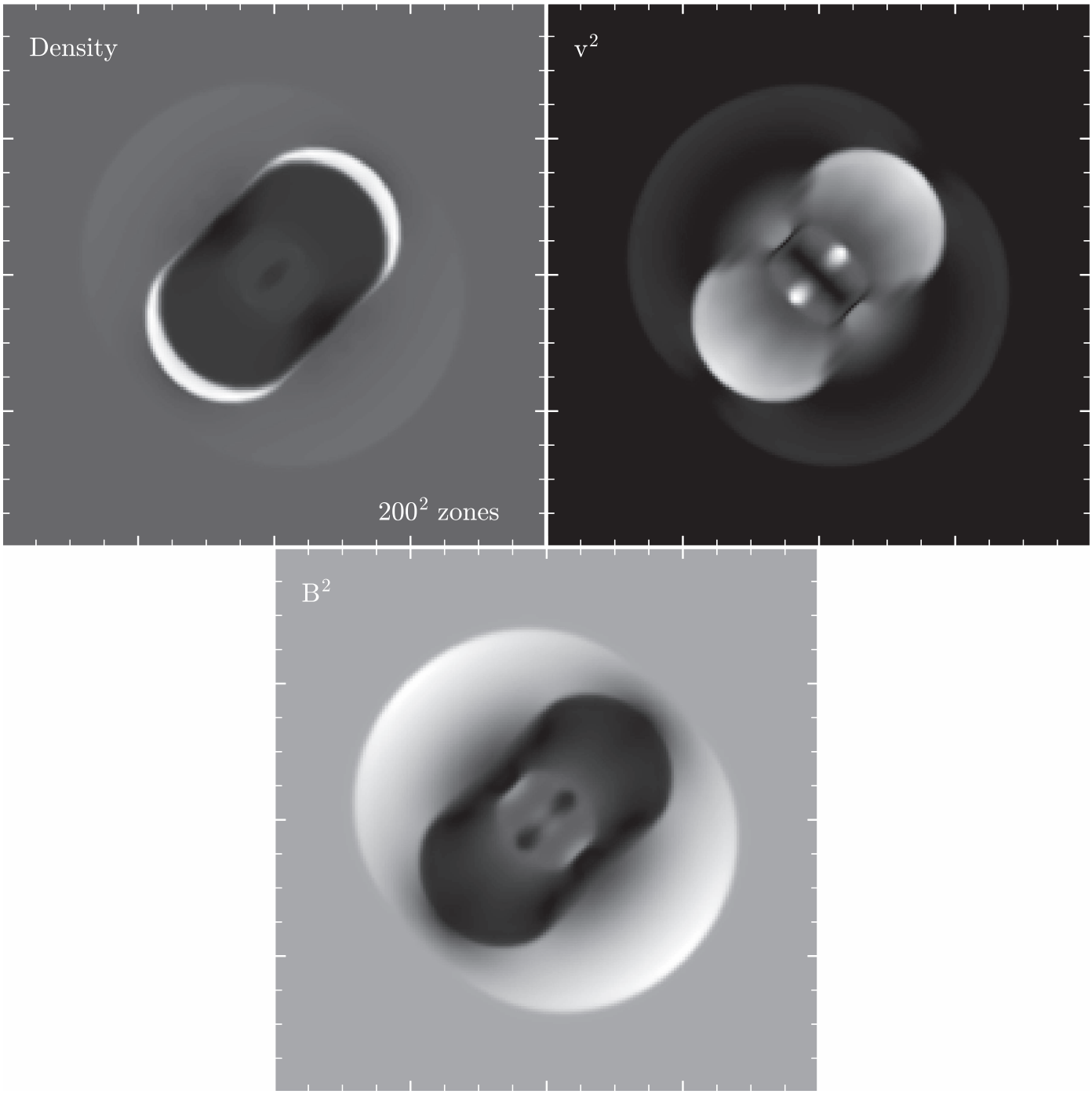

Figure 22. Images of selected quantities in a 2D slice at $z = 0$ for the magnetic blast wave test in three dimensions at $t = 0.02$. The solution was computed on a domain with $200 \times 300 \times 200$ zones.

on these architectures up to 260 K threads on Blue Waters, demonstrates its capabilities and adaptability.

P.J.M. thanks Luiz DeRose (Cray) and John Levesque (Cray) for their support of this project. J.D. acknowledges support from the People Programme (Marie Sklodowska Curie Actions) of the European Unions Eighth Framework Programme H2020 under REA grant agreement no. [658912]. P.E. is supported by the ITC and Harvard FAS Research Computing. T.W.J. and B.J.O. acknowledge support from NSF grant AST1211595. C.N. was supported by an NSF Graduate Fellowship under Grant 000039202. We thank Cray, Inc. for use of their internal systems. Blue Waters computing resources came through a grant from the Great Lakes Consortium for Petascale Computing. The Blue Waters sustained-petascale computing project is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

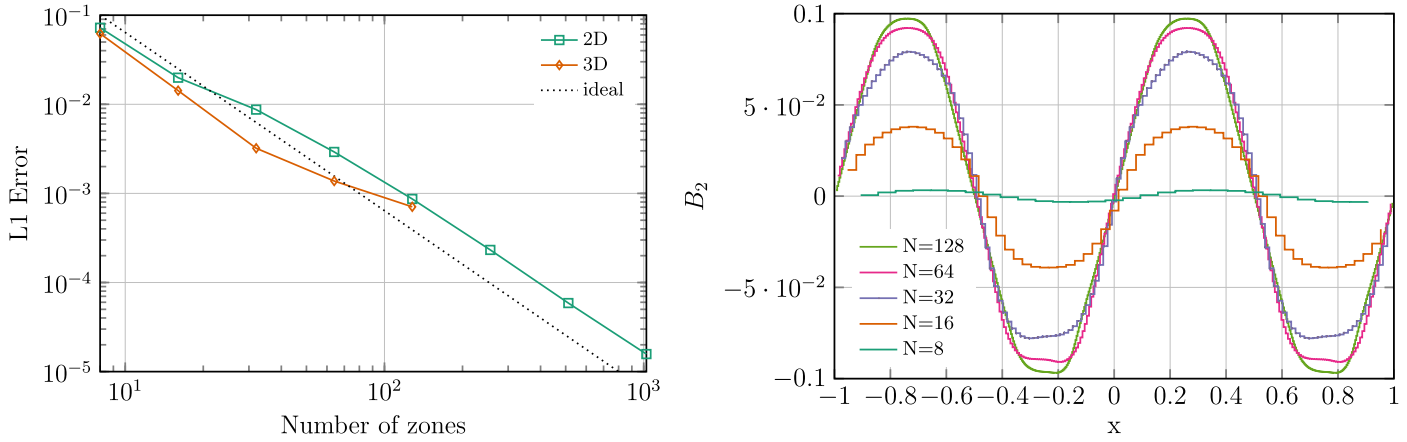

Figure 23. Circularly polarized Alfvén wave test. Left: convergence of the norm of the L_1 error vector for the circularly polarized Alfvén wave tests in two and three dimensions. Right: profiles of B_2 component of magnetic field for increasing resolutions after propagating five wavelengths across the domain.

Appendix
MHDTVD

Integrating Equation (5) over a volume element and over a time interval gives

$$\boldsymbol{q}_i^{n+1} = \boldsymbol{q}_i^n - \frac{\Delta t}{\Delta x}(\boldsymbol{F}_{i+1/2}^{n+1/2} - \boldsymbol{F}_{i-1/2}^{n+1/2}). \tag{13}$$

In the MHDTVD method, an approximation to $\boldsymbol{F}_{i+1/2}^{n+1/2}$, referred to as the modified flux $\bar{\boldsymbol{u}}_{i+1/2}^{n+1/2}$, is computed from

$$\bar{\boldsymbol{u}}_{i+1/2}^{n+1/2} = \frac{1}{2}(\boldsymbol{F}(\boldsymbol{q}_i^n) + \boldsymbol{F}(\boldsymbol{q}_{i+1}^n)) - \frac{\Delta x}{2\Delta t}\boldsymbol{f}_{i+1/2}^n, \tag{14}$$

$$\boldsymbol{f}_{i+1/2}^n = \sum_{k=1}^{7}\beta_{k,i+1/2}\boldsymbol{R}_{k,i+1/2}^n, \tag{15}$$

$$\beta_{k,i+1/2} = Q_k\left(\frac{\Delta t^n}{\Delta x}a_{k,i+1/2}^n + \gamma_{k,i+1/2}\right)\alpha_{k,i+1/2} - (g_{k,i} + g_{k,i+1}), \tag{16}$$

$$\alpha_{k,i+1/2} = \boldsymbol{L}_{k,i+1/2}^n \cdot (\boldsymbol{q}_{i+1}^n - \boldsymbol{q}_i^n), \tag{17}$$

$$\gamma_{k,i+1/2} = \begin{cases} \frac{g_{k,i+1} - g_{k,i}}{\alpha_{k,i+1/2}} & \text{for } \alpha_{k,i+1/2} \neq 0, \\ 0 & \text{for } \alpha_{k,i+1/2} = 0 \end{cases}, \tag{18}$$

$$g_{k,i} = \text{SIGN}(\tilde{g}_{k,i+1/2})\,\text{SWEBY}_{\text{limiter}}(\tilde{g}_{k,i+1/2}, \tilde{g}_{k,i-1/2}), \tag{19}$$

$$\tilde{g}_{k,i+1/2} = \frac{1}{2}\left[Q_k\left(\frac{\Delta t^n}{\Delta x}a_{k,i+1/2}^n\right) - \left(\frac{\Delta t^n}{\Delta x}a_{k,i+1/2}^n\right)^2\right]\alpha_{k,i+1/2}, \tag{20}$$

$$Q_k(\chi) = \begin{cases} \frac{\chi^2}{4\epsilon_k} + \epsilon_k & \text{for } |\chi| < 2\epsilon_k, \\ |\chi| & \text{for } |\chi| \geqslant 2\epsilon_k \end{cases}. \tag{21}$$

The right-handed eigenvector, $\boldsymbol{R}_{k,i+1/2}^n$, and characteristics, $\alpha_{k,i+1/2}$, are from Cargo & Gallice (1997). The primitive variables at zone interfaces, used to construct $\boldsymbol{R}_{k,i+1/2}^n$ and $\alpha_{k,i+1/2}$, come from the averaging scheme also described in Cargo & Gallice (1997). The purpose of ϵ_k is to add a controlled amount of dissipation into each wave to ensure that $Q_k(\chi)$, referred to as the coefficient of numerical viscosity, is continuous and positive (Zheng & Lee 1998). This eliminates spurious oscillations that can occur when there is an entropy violation across a discontinuity. The value of ϵ_k must satisfy $0 \leqslant \epsilon_k < 0.5$, and the optimal value depends on the number of dimensions and complexity of flows in the calculation.

Under certain circumstances, Roe-type methods like MHDTVD will produce unphysical densities or pressures (Einfeldt et al. 1991). A typical solution to this problem is to define floor values for density and pressure that are applied when exceeded. WOMBAT uses this approach, but additionally offers a set of user-defined floor values, called the protection floor, that will automatically switch to another Riemann solver that does not have this issue. Similar to the approach of GS08, we substitute the MHDTVD fluxes with the more diffusive HLL fluxes (Einfeldt et al. 1991) under the rare conditions when the protection floor is exceeded. The modified flux $\bar{\boldsymbol{u}}_{i+1/2}^{n+1/2}$ is computed for the HLL scheme as

$$\bar{\boldsymbol{u}}_{i+1/2}^{n+1/2} = \frac{b^+\boldsymbol{F}(\boldsymbol{q}_i^n) + b^-\boldsymbol{F}(\boldsymbol{q}_{i+1}^n)}{b^+ - b^-} + \frac{b^+b^-}{b^+ - b^-}(\boldsymbol{q}_{i+1}^n - \boldsymbol{q}_i^n), \tag{22}$$

$$b^+ = \text{MAX}\{\text{MAX}(a_{\max}, v_{x,i+1}^n + c_{f,i+1}^n), 0\}, \tag{23}$$

$$b^- = \text{MIN}\{\text{MIN}(a_{\min}, v_{x,i-1}^n - c_{f,i11}^n), 0\}, \tag{24}$$

where $a_{\max}$ and $a_{\min}$ are the maximum and minimum eigenvalues. Note that the HLL fluxes do not rely on an eigensolution to the MHD equations, which makes them more diffusive than the MHDTVD fluxes. Consequently, we apply them as infrequently as possible, so only to avoid unphysical behaviors.

References

Almgren, A. S., Beckner, V. E., Bell, J. B., et al. 2010, ApJ, 715, 1221
Amer, A., Lu, H., Wei, Y., Balaji, P., & Matsuoka, S. 2015, in Proc. 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2015 (New York: ACM), 239
Balsara, D. S., & Spicer, D. S. 1999, JCoPh, 149, 270
Brio, M., & Wu, C. C. 1988, JCoPh, 75, 400
Bryan, G. L., Norman, M. L., O'Shea, B. W., et al. 2014, ApJS, 211, 19
Cargo, P., & Gallice, G. 1997, JCoPh, 136, 446
Dinan, J., Balaji, P., Buntinas, D., et al. 2016, Concurrency Computation Practice and Experience, 28, 4385
Dolag, K., & Stasyszyn, F. 2009, MNRAS, 398, 1678
Dosanjh, M. G., Groves, T., Grant, R. E., Brightwell, R., & Bridges, P. G. 2016, in IEEE/ACM International Symp. on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid 2016)
Dubey, A., Almgren, A., Bell, J., et al. 2016, arXiv:1610.08833

Dubey, A., Fisher, R., Graziani, C., et al. 2008, in ASP Conf. Ser. 385, Numerical Modeling of Space Plasma Flows, ed. N. V. Pogorelov, E. Audit, & G. P. Zank (San Francisco, CA: ASP), 145
Einfeldt, B., Roe, P. L., Munz, C. D., & Sjogreen, B. 1991, JCoPh, 92, 273
Fromang, S., Hennebelle, P., & Teyssier, R. 2006, A&A, 457, 371
Fryxell, B., Olson, K., Ricker, P., et al. 2000, ApJS, 131, 273
Gardiner, T. A., & Stone, J. M. 2005, JCoPh, 205, 509
Gardiner, T. A., & Stone, J. M. 2008, JCoPh, 227, 4123
Hjelm, N. 2014, in Proc. 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14 (New York: ACM)
Hjelm, N. 2016, in Proc. 23rd European MPI Users' Group Meeting, EuroMPI 2016 (New York: ACM), 184
Kandalla, K., Mendygral, P., Radcliffe, N., et al. 2016, in Proc. of the 2016 Cray User's Group, ed. A. Winfer (Oak Ridge, TN: Cray User Group)
Li, M., Potluri, S., Hamidouche, K., Jose, J., & Panda, D. K. 2013, in Proc. 20th European MPI Users' Group Meeting (New York: ACM), 91
Londrillo, P., & Del Zanna, L. 2000, ApJ, 530, 508
Mignone, A., Bodo, G., Massaglia, S., et al. 2007, ApJS, 170, 228
Mignone, A., Zanni, C., Tzeferacos, P., et al. 2012, ApJS, 198, 7
Miniati, F., & Martin, D. F. 2011, ApJS, 195, 5
Mocz, P., Pakmor, R., Springel, V., et al. 2016, MNRAS, 463, 477
Orszag, S. A., & Tang, C.-M. 1979, JFM, 90, 129
Potluri, S., Sur, S., Bureddy, D., & Panda, D. K. 2011a, in European MPI Users' Group Meeting (Berlin: Springer), 321
Potluri, S., Wang, H., Dhanraj, V., Sur, S., & Panda, D. K. 2011b, in European MPI Users' Group Meeting (Berlin: Springer), 99
Price, D. J. 2012, JCoPh, 231, 759
Ryu, D., Miniati, F., Jones, T. W., & Frank, A. 1998, ApJ, 509, 244
Springel, V. 2005, MNRAS, 364, 1105
Springel, V. 2010, MNRAS, 401, 791
Stone, J. M., Gardiner, T. A., Teuben, P., Hawley, J. F., & Simon, J. B. 2008, ApJS, 178, 137
Teyssier, R. 2002, A&A, 385, 337
Tóth, G. 2000, JCoPh, 161, 605
Vaidyanathan, K., Kalamkar, D. D., Pamnany, K., et al. 2015, in Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (New York: ACM), 30
Zheng, B., & Lee, C.-H. 1998, CNSNS, 3, 82