Journal of Parallel and Distributed Computing 139 (2020) 65–74

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Efficient OpenMP parallelization to a complex MPI parallel magnetohydrodynamics code

Hongyang Zhou *, Gábor Tóth

Department of Climate and Space Sciences and Engineering, University of Michigan, Ann Arbor, MI, United States of America

ARTICLE INFO

Article history:
Received 8 July 2019
Received in revised form 16 November 2019
Accepted 6 February 2020
Available online 13 February 2020

Keywords:
OpenMP
MPI
Parallel scaling
MHD

ABSTRACT

The state-of-the-art finite volume/difference magnetohydrodynamics (MHD) code Block Adaptive Tree Solarwind Roe Upwind Scheme (BATS-R-US) was originally designed with pure MPI parallelization. The maximum problem size achievable was limited by the storage requirements of the block tree structure. To mitigate this limitation, we have added multi-threaded OpenMP parallelization to the previous pure MPI implementation. We opted to use a coarse-grained approach by making the loops over grid blocks multi-threaded and have succeeded in making BATS-R-US an efficient hybrid parallel code with modest changes in the source code while preserving the performance. Good weak scalings up to hundreds of thousands of cores were achieved both for explicit and implicit time stepping schemes. This parallelization strategy greatly extended the possible simulation scale from 16,000 cores to more than 500,000 cores with 2GB/core memory on the Blue Waters supercomputer. Our work also revealed significant performance issues for some of the compilers when the code is compiled with the OpenMP library, probably related to the less efficient optimization of a complex multi-threaded region.

1. Introduction

The *Block-Adaptive-Tree Solarwind Roe Upwind Scheme* (BATS-R-US) [7,19] is a multi-physics MHD code written in Fortran 90+ that has been actively developing at the University of Michigan for over 20 years. It is the most complex and often the computationally most expensive model in the *Space Weather Modeling Framework* (SWMF) [1,29,30] that has been applied to simulate multi-scale space physical systems including, but not limited to, the solar corona, the heliosphere, planetary magnetospheres, moons, comets and the outer heliosphere. For the purpose of *adaptive mesh refinement* (AMR) and running efficiency, the code was designed from the very beginning to use a 3D Cartesian *block-adaptive mesh* with MPI parallelization [7,26]. In 2012, the original block-adaptive implementation has been replaced with the newly designed and implemented *Block Adaptive Tree Library* (BATL) [29] for creating, adapting, load-balancing and message-passing a 1, 2, or 3 dimensional block-adaptive grid in generalized coordinates. The major advantages of adaptive block approach include locally structured grid in each block, cache optimizations due to relatively small arrays associated with the grid blocks, loop optimization for fixed sized loops over cells in the block, and simple load balancing. Larger blocks reduce the total number of ghost cells surrounding the grid blocks, but make the grid adaptivity less economic. Smaller blocks allow precise grid adaptation, but require a large number of blocks and more storage and computation spent on ghost cells. The typical choice of block size in 3D ranges between 4^3 to 16^3 grid cells, with an additional 1–3 layers of ghost cells on each side depending on the order of the numerical scheme.

BATS-R-US has been gradually evolving into a comprehensive code by adding new schemes as well as new physical models. Currently, 60 equation sets from ideal hydrodynamics to the most recent six-moment fluid model [11] are available. The most important applications solve various forms of the magnetohydrodynamic (MHD) equations, including resistive, Hall, semi-relativistic, multi-species and multi-fluid MHD, optionally with anisotropic pressure, radiative transport and heat conduction. There are several choices of numerical schemes for the Riemann solvers, from the original Roe scheme to many others combined with a second order total variation diminishing (TVD) scheme or a fifth order accurate conservative finite difference scheme [5]. The time discretization can be explicit, point-implicit, semi-implicit, explicit/implicit or fully implicit. A high level abstraction of the code structure is presented in Fig. 1.

A powerful feature of BATS-R-US is the incorporation of user modules. This is an interface for users to modify literally any part of the kernel code without interfering with other modules. It provides a neat and easy way to gain high-level control of the simulations. Currently there are 51 different user modules in

* Corresponding author.
E-mail addresses: hyzhou@umich.edu (H. Zhou), gtoth@umich.edu (G. Tóth).

https://doi.org/10.1016/j.jpdc.2020.02.004
0743-7315/

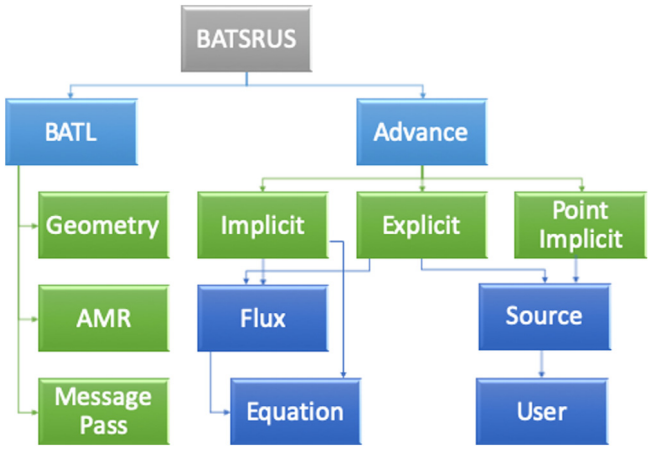


Fig. 1. The high level structure of BATS-R-US. The AMR library BATL is used for mesh generation, refinement and message passing. Explicit, fully-implicit, semi-implicit, part-implicit and point-implicit schemes can be used for time advance. Numerical flux schemes of 1st, 2nd and 5th order can be chosen. Multiple versions of the equation and user modules containing the equation variable definitions and the application specific codes are available.

the repository, mostly used for the setup of specific initial and boundary conditions and additional user-defined source terms for the specific applications.

One of the key features of BATS-R-US is its excellent scalability on supercomputers. Previous benchmarks [29] with pure MPI parallelization have shown good strong scaling up to 8192 cores and weak scaling up to 16,384 cores within the memory limit of the testing platform. However, the grid-related pre-calculated information replicated on every MPI process for simplifying the refinement algorithm have generated an unavoidable memory redundancy on computational nodes. To increase the scalability to even larger sizes, we need to reorganize the code and come up with a more advanced solution.

In this work, we have extended BATS-R-US with a hybrid MPI + OpenMP parallelization that significantly mitigates the limitations due to available memory. The strategies and issues are described in the next two sections, followed by performance test results and discussions.

2. Hybrid parallelization strategy

BATS-R-US was originally designed for pure MPI parallelization and did not take advantage of the rapid development of shared-memory multi-threading programming starting from late 1990s [4,6]. Even though MPI is generally observed to give better parallel scaling than OpenMP due to forced data locality, one obvious shortcoming of the pure MPI implementation is wasteful memory usage. In BATS-R-US, we support 1, 2, and 3 dimensional block-adaptive grids, where each block contains $n_1 \times n_2 \times n_3$ cells. Each block is surrounded by n_G layers of ghost cells. The topology of a 3D block-adaptive grid is described by an integer array with 18 integers per grid node (active nodes are at the leaves of the tree corresponding to the grid blocks) containing the following information: the current status (used, unused, to be refined, to be coarsened), the processor and local block indexes for active nodes, the minimum allowed, maximum allowed and current refinement levels, the three integer coordinates with respect to the whole grid, the global indexes of the parent and eight children nodes. This tree structure, together with a few more arrays related to load balancing and adaptive mesh refinement, is replicated on every MPI process, which simplifies the algorithms and saves inter-processor communication, but is clearly wasteful for current many-core computer nodes. If the total number of grid blocks reaches about 10 million, the total tree related storage will be close to 2 GB, so it will use up all the typical 2 GB/core memory available on the computing nodes. One option is to distribute the grid information over the MPI processes, but that will make the AMR algorithm more complicated and potentially less efficient. Another option is to make the grid blocks larger (more cells per block), but that will make the AMR less flexible and require more grid cells to achieve the required resolution in the regions of interest. Our preferred approach is to use OpenMP multi-threading. If there are N threads sharing the tree information per MPI process, then the fraction of node memory used to store the tree is reduced by a factor of N. On current architectures with many cores per node, this is a significant improvement. Of course, it is only acceptable, if the performance of the code remains close to that of the original pure MPI version.

There are other potential benefits of using hybrid parallelization. In some applications BATS-R-US uses large lookup tables with pre-calculated values for sake of efficiency (for equation of state, radiative cooling, opacity etc.) These tables can be quite large, and using shared memory access from multiple threads is clearly beneficial for saving memory use. Having fewer MPI processes for a given problem size may in principle improve communication (fewer messages are passed) and I/O performance (fewer files are used), but these potential benefits may or may not be realized and/or significant on a given computer platform.

The idea of hybrid programming with OpenMP and MPI arises naturally with the architecture of modern supercomputers, where each node containing several multi-core chips is connected through the network (e.g. InfiniBand). The potential advantages and challenges have been thoroughly described in the literature, e.g. [10,20,24]. With careful design of the code, researchers have found a relatively satisfactory performance on modern clusters [12,13].

There are four levels of communication supported by the current versions of MPI libraries with respect to thread safety:

1. Single: MPI calls outside PARALLEL regions only
2. Funneled: MPI calls on master thread only
3. Serialized: MPI calls on multiple threads serially
4. Multiple: concurrent MPI calls on multiple threads

For multi-threaded communication, different tags have to be used for distinguishing messages from different threads. We decided to use the default support level “MPI_THREAD_SINGLE” to minimize the work of rewriting the source code.

Two types of hybrid parallelization styles have been widely used [10]:

1. the fine-grained model, which applies the multi-threaded parallelism on the innermost loops
2. the coarse-grained model, which applies the multi-threaded parallelism at a relatively high level

For BATS-R-US the two approaches correspond to the choice of parallelization over grid cells (fine-grained) or grid blocks (coarse-grained). As explained in detail above, BATS-R-US spreads the blocks among MPI processes and then loops through cells inside each block for calculating the face values, face fluxes and source terms for the update in each time step. We decided to follow the coarse-grained parallelization approach. A clear advantage is that there are fewer block loops than cell loops, so there are fewer directives to add and less overhead in entering/exiting the multi-threaded regions. In addition, the block loops cover a larger fraction of the computational cost than the cell loops, so we can expect better parallel performance. A potential issue with parallelizing the block loops is the complexity of the code within the parallel region, but this problem can be handled, as we will

discuss below. In the end, we have successfully added OpenMP parallel directives to most block-loops in BATS-R-US.

Because of the functional programming structure of BATS-R-US, the usage of module variables in the multi-threaded regions requires special attention. Fortunately OpenMP provides the `threadprivate` directive to make global scope variables local and persistent to a thread through the execution of multiple parallel regions. This is a key feature of OpenMP that makes the hybrid parallelization possible for BATS-R-US without major modifications of the source code.

Similar hybrid approaches have also been implemented in block based AMR codes like FLASH [2], MPI-AMRVAC [18,33], Athena [32], and GAMER2 [23]. FLASH is an AMR astrophysical code with OpenMP implemented for multiple solvers in both coarse-grain and fine-grain methods. In the small scale testings [2], the coarse-grain approach consistently outperform the fine-grain approach. MPI-AMRVAC is written in Fortran with LASY preprocessor [27], with coarse-grain multi-threading loop implemented in the explicit time advance, source term calculation, and error checking. Nice strong scaling with up to 2000 cores and weak scaling with up to 30,000 cores have been shown for pure MPI runs. The latest version of Athena is written in C++, with multi-threading loop implemented mostly for the first dimension spatial loop in generalized coordinates among 80+ source files, and nice MPI scaling up to 6144 cores. Finally, GAMER2 is a recently developed GPU-accelerated AMR code in astrophysics with MPI/OPENMP/GPU involved, with an impressive speedup compared with Athena++ [23]. It is also worth mentioning that a block based AMR library AMReX [34] natively supports MPI/OPENMP/OPENACC/GPU implementation. However, for all the mentioned codes above we have not found publications on their OpenMP performance. Another distinguishing feature of BATS-R-US is the parallel implicit time advance scheme. Here we present detailed performance results of the hybrid parallelization strategy for both explicit and implicit schemes up to hundreds of thousands of cores.

3. Implementation

3.1. Overview

There are two high level goals while modifying and improving the code:

1. Backward compatibility: the code should still work correctly and efficiently without the OpenMP compilation flag.
2. Minimize work effort and code changes as much as possible.

BATS-R-US is able to solve the system of partial differential equations with a mixture of explicit and implicit timestepping blocks distributed among the MPI processes. We treat first the explicit and then the implicit modules and add OpenMP directives incrementally to make sure the correctness and efficiency for each individual part. Since OpenMP does not allow controlling the scope of variables that are not explicitly listed in the function arguments, we have to rely on the default rules and settings about the variable scopes, especially for the variables in function and subroutine calls within the multi-threaded regions.

Current standard of the default OpenMP Fortran variable scope rules are listed as follows:

1. Variables explicitly showing up in the `omp parallel` section follow the common OpenMP rules for the shared or private attributes.
2. Variables with `save` attribute (e.g. module variables by default) are shared.
3. Variables being initialized at declaration time are shared.
4. Otherwise all variables are private.

Based on our choice of multi-thread parallelism on the block level, there is a simple yet general rule to decide whether a module variable should be declared as private inside the OpenMP parallel regions: *if an array does not have an index looping over blocks, it should be declared as threadprivate as long as there are writing operations.* Read-only variables do not need to be private to each thread. Note that Rule 3 listed above in practice gave us the most innocent looking bugs during testings. Dynamic threadprivate variables also need to be allocated for each thread, as opposed to only being allocated for the main thread, otherwise this will produce "array not defined" run time errors.

In total we add 609 OpenMP directive lines to the 246,728 pure source code lines (excluding comments and empty lines), which is only 0.25% addition. These directives appear in declaring threadprivate variables in modules and subroutines, parallelizing block loops in both explicit and implicit schemes, reduction operations in error check and Krylov iterative solver, and message passing. Together with the minor modifications and adjustments for the algorithms described below, the hybrid parallelization can be achieved with no extra tedious work.

3.2. Explicit schemes

Due to the independency of blocks (with ghost cells) during one timestep update, it is rather straightforward to add multi-threading to the block loop over all the explicitly updated blocks. The kernel code is shown in List 1. Note that we only present the kernel interfaces for brevity. At the final stage of each timestep, message passing is done through the BATL library for updating the ghost cell values.

```
STAGELOOP: do iStage = 1, nStage
   ! Multi-block solution update.
   !$omp parallel do
   do iBlock = 1, nBlock
      if(Unused_B(iBlock)) CYCLE
      call calc_face_value(iBlock)
      if(IsBoundaryBlock(iBlock)) &
         call set_face_boundary(iBlock, Time_Simulation)
      call calc_face_flux(iBlock)
      call calc_source(iBlock)
      call update_state(iBlock)
      call calc_timestep(iBlock)
   end do
   !$omp end parallel do
   if(iStage < nStage) call exchange_messages
end do STAGELOOP  ! Multi-stage solution update loop.
```

Listing 1: Explicit multi-stage update block loop pseudo code

All the high level functions (e.g. `calc_face_value`, `calc_face_flux`) are imported from their corresponding modules. Inside each module, the `threadprivate` clause is used to distinguish local variables for each block, e.g. the cell-based face value arrays and the cell-based face flux arrays.

3.3. Implicit schemes

Different kinds and combinations of implicit schemes are implemented in BATS-R-US. It provides the options of using point-implicit numerical scheme for handling stiff source terms and semi-implicit schemes for handling Hall and ohmic resistivity, radiative transfer and heat conduction terms. The point-implicit scheme follows a blockwise implementation which is similar to the explicit source term calculation in the sense of multi-threading. The semi-implicit schemes, which are only part of the unknowns (e.g. magnetic field) are solved implicitly, can be

treated as a special case for the full implicit schemes, therefore we will only discuss about the full implicit scheme parallelization. Lastly, the part implicit scheme means that only the unknowns in the selected blocks are updated implicitly, otherwise they are updated explicitly [28].

The mostly used implicit schemes in BATS-R-US convert the non-linear PDEs into a linearized form $\mathbf{A}\vec{x} = \vec{b}$ and solve iteratively in the Krylov subspace. Because of the explicit storage of variables based on the grid blocks, the implementation of full/part implicit schemes requires a transformation between the explicit and implicit storage. A common pattern in the original implicit solver module is a counter `n` inside the implicit block loop for indexing over the 1D arrays of the unknowns or the right-hand-side of the linear equation, where the backward-dependencies prevent a direct OpenMP parallelization. However, in most cases the indexes of the starting blocks can be pre-calculated directly from the grid-block structure, which then allows a block level multi-threading parallelization. A typical implicit block loop example is given in List 2. Here after solving the unknowns $\vec{x}$ from the linear equations we use the pre-calculated initial value for the incremental counter `n` for each block outside the cell and variable loops so that the threads can be performed independently. Besides, the block-based preconditioners can also benefit from multi-threading.

```
!$omp parallel do private( n )
do iBlock=1,nBlock
   n = (iBlock-1)*nI*nJ*nK*nVar
   do k=1,nK; do j=1,nJ; do i=1,nI; do iVar=1,nVar
      n = n + 1
      Impl_VGB(iVar,i,j,k,iBlock) = Impl_VGB(iVar,i,j,k,
     iBlock)
        + x_I(n)*Norm_V(iVar)
   enddo; enddo; enddo; enddo;
enddo
!$omp end parallel do
```

Listing 2: Implicit block loop pseudo code: solution update. nI, nJ, nK, nVar represent the number of cells in a grid block in the three spatial dimensions and the number of variables, respectively.

Profilings on the implicit solver suggest that it is not optimal to add OpenMP parallel regions as much as possible. Small loops with too few iterations will not benefit from OpenMP because of thread launching overhead; some intrinsic functions in Fortran for linear algebra already have built-in optimizations such that further multi-threading has no gain in timing. For instance, it is tempting to add `workshare` directive to operations like dot products in the Krylov iterative solver, but it turns out to have no improvement with such highly optimized Fortran intrinsic functions. We also avoided the usage of nested OpenMP parallel regions for loops over cells.

3.4. Message passing

Message passing fills in the ghost cells of the blocks based on information taken from the neighboring blocks. With mesh refinement involved, there are three possible cases for communication with a neighboring block: 1. block at the same refinement level; 2. block at a finer refinement level; 3. block at a coarser refinement level.

The original message passing algorithm inside the BATL library can do both cellwise and facewise message exchange between layers of ghost cells and neighboring physical cells. Message passing on the faces are implemented for the adaptive grid where the numerical fluxes need to be conserved between the coarse and fine cell faces on different resolution level. For most of the current applications, this part usually takes insignificant amount of time, so OpenMP directives are not added considering the amount of work of rewriting the code and the potential threading overhead detrimental to performance. In the case of cellwise message passing, ghost cells are filled with information from neighboring blocks. Our old scheme combines the local exchange (message passing between blocks on the same processor) and the remote exchange (message passing between blocks on the remote processors). This turns out to be difficult for adding efficient multi-threading parallelization, which is now rewritten to fully separate the local and remote part for an easy implementation of OpenMP and allow for possible overlap of communication and computation. The pseudo code is shown in List 3. First the cell values to be communicated to remote MPI processes are collected into packed 1D send buffers. Next the MPI asynchronous communication functions `iSend/iRecv` are called. Then the *local* copying of cell values between blocks residing on the same MPI process are done in a multi-threaded loop. This is followed by the `waitall` barriers so that the communication overhead can be hidden by the time spent on the local copies. Finally the received linear buffer values are saved into the ghost cells of the blocks. This implementation only requires the thread safety level up to `MPI_THREAD_SINGLE` since the MPI calls occur outside the multi-threaded loop. Our performance tests presented below suggest that this simple strategy works satisfactorily, especially for the runs with more threads. Higher thread level support with more complex communication scheme could be added if this becomes a significant bottleneck in future applications.

```
! Prepare the buffer for message passing to another
     processor
do iBlockSend = 1, nBlock
   call message_pass_block(iBlockSend, DoRemote=.true.)
end do
! Post sends
iRequestS = 0
do iProcRecv = 0, nProc-1
   iRequestS = iRequestS + 1
   call MPI_isend(...)
end do
! Post requests
iRequestR = 0
do iProcSend = 0, nProc-1
   iRequestR = iRequestR + 1
   call MPI_irecv(...)
end do
! Exchange ghost cell info on the same processor
!$omp parallel do
do iBlockSend = 1, nBlock
   call message_pass_block(iBlockSend, DoRemote=.false.)
end do
!$omp end parallel do
! Wait for all requests to be completed
if(iRequestR > 0) call MPI_waitall(...)
! Wait for all sends to be completed
if(iRequestS > 0) call MPI_waitall(...)
! Unpack the 1D buffer array to full variable arrays
call buffer_to_state
```

Listing 3: Message passing demonstration

3.5. Data locality

Current clusters rely heavily on the idea of Non-uniform Memory Access (NUMA). On Blue Waters, for example, the 32 core XE6 computing node contains two sockets, each with 2 8-core NUMA domain. Access time on shared memory in different NUMA domain/socket is significantly longer than within the same NUMA domain/socket, so we expect a performance drop from 8 to 16 threads and 16 to 32 threads on one node. On Stampede2 SKX compute nodes with 48 cores per node on 2 sockets, each NUMA domain consists of 12 cores. So similarly, we expect performance drop beyond 12 threads.

3.6. Timing utility

The original timing library we used in BATS-R-US only works for single-threaded MPI runs. This new OpenMP extension of the code requires a modification of the current library. Now we have a new subroutine which only record the timings from the master thread for each MPI process. It is also possible, but not necessarily needed, to record the timings for each thread.

4. Testing and debugging

4.1. Nightly tests

For a comprehensive quality check and verification of the SWMF and the physics models contained (including BATS-R-US), we have built an automated nightly test suite for testing the code with various setups on various platforms. The latest version of the code is checked out from a central Git repository and 100+ tests are performed on multiple platforms with different compilers, compiler flags and number of cores. The test results are monitored every day and have been archived since 2009. These nightly tests were used heavily for the OpenMP development. The BATS-R-US tests helped us identifying code features that did not perform the same way with and without OpenMP parallelization. Gradually more-and-more tests passed, and now OpenMP can be turned on and it passes for essentially all 100+ tests. Newly added code can work seamlessly as long as they carefully follow the Fortran variable scope standard listed above. This has to be verified, of course, by adding new functionality tests that cover the new code features to the test suite.

4.2. Resolving race conditions

Race conditions are very hard to debug. The code does not crash, but the results are incorrect, and may change randomly depending on the order of the thread executions. This can even happen when each thread uses the variable with the same values, but the value changes inside the multi-threaded region. An example, that demonstrates the issue is shown in Listing 4.

```
1  subroutine calc_corotation_velocity(Xyz_D, uRot_D)
2   ! Calculates rotation velocity uRot_D at location Xyz_D
3   use CON_axes,          ONLY: get_axes
4   use ModCoordTransform, ONLY: cross_product
5   real, intent(in) :: Xyz_D(3)
6   real, intent(out):: uRot_D(3)
7
8   real, save:: Omega_D(3)
9   !$omp threadprivate( Omega_D )
10  logical    :: IsUninitialized = .true.
11
12  if(IsUninitialized)then
13      call get_axes(TimeSimulation, RotAxisGseOut_D=
     Omega_D)
14      Omega_D = OmegaBody * Omega_D
15      IsUninitialized = .false.
16  end if
17  ! The corotation velocity is u = Omega x R
18  uRot_D = cross_product(Omega_D, Xyz_D)
19 end subroutine calc_corotation_velocity
```

Listing 4: Race condition example in subroutine

Although all threads perform the same calculation to obtain the angular velocity vector `Omega_D`, a race condition is still present because its value is first obtained with `get_axes` and then multiplied by `OmegaBody`, and different threads will go through these steps nearly, but not perfectly, simultaneously. Since the variable is declared with the save attribute in line 8, OpenMP will regard the variable as shared by default, so the value of the variable will be essentially random. One possible fix is adding the `!$omp threadprivate` directive in line 9 (an alternative would be adding a new variable for the rotational axis vector). Finding errors like this in hundreds of thousands of lines of source code spanning dozens of files is extremely challenging. We tried several different approaches (careful code inspection, checking which variable changes in a loop, etc.) but it became clear that we need help from a debugger. We tried several debuggers, but most of them either did not report the issues, or produced an overwhelming number of false alarms. In the end, we found the Intel Inspector to be an extremely useful tool for detecting thread-related issues. In our experience with Fortran, Inspector can successfully locate dead-locks, false sharing and race conditions. The race condition in the example problem above was found and fixed with the help of Intel Inspector.

Table 1
Platform information.

Name	Blue Waters	Stampede2
Processor	AMD 6276 Interlagos	Intel Xeon Platinum 8160
Sockets/Node	2	2
Cores/Node	32[a]	48
Threads/Core	1	2[b]
Clock rate (GHz)	2.3	2.1[c]
Cache (GB)	2	4
Fortran compiler used	gfortran	ifort

[a]The Interlagos processor has 8 Bulldozer cores, but viewed as 16 "processors" by the Linux system. These "processors" are the schedualable integer cores that work with the floating point unit.
[b]Hyperthreading is on by default.
[c]1.4–3.7 GHz depending on instruction set and number of active cores, 2.1 GHz is the nominal value.

5. Performance

We have performed some standard Brio–Wu MHD shock tube tests on various platforms. A 3D Cartesian grid is chosen, and dynamic AMR has not been employed. The magnetic monopole is controlled by hyperbolic cleaning [29] and 8-wave scheme [19]. Despite its simplicity, this is a fairly representative test for various applications in terms of computational cost per grid cell, as well as exercising the most important parts of the BATS-R-US code.

5.1. Platform information

Table 1 lists all the platforms used for testing. We have performed parallel scaling studies on the Blue Waters Supercomputer using the XE nodes with AMD 6276 Interlagos processor and on the Stampede2 supercomputer using the SKX nodes with Intel Xeon Platinum 8160 ("Skylake") processor [25].

5.2. Strong parallel scaling

Fig. 2 shows the strong scalings on Blue Waters from 1 to 2048 nodes (32 cores/node). The MHD equations with hyperbolic cleaning [8,29] are solved with the second order explicit Linde scheme [17] and Korens's limiter [15] on a 3D Cartesian grid. The grid contains about 33.5 million grid cells in 65,536 grid blocks. Each block contains $8 \times 8 \times 8$ cells (with additional 2 layers of ghost cells on each side), which is a typical block size for our common simulations. In the figure, the different colors represent different number of threads used in the runs. The points lying on the same vertical line have the same number of cores but a different combination of MPI processes and OpenMP threads. At 2048 nodes (65,536 cores), we only have 1 block per core and we obtain $\sim$ 25% efficiency compared to the ideal linear scaling starting from 1 node shown by the dashed line. At 1024 nodes (32,768 cores), the parallel efficiency is around 50%. Over 1000

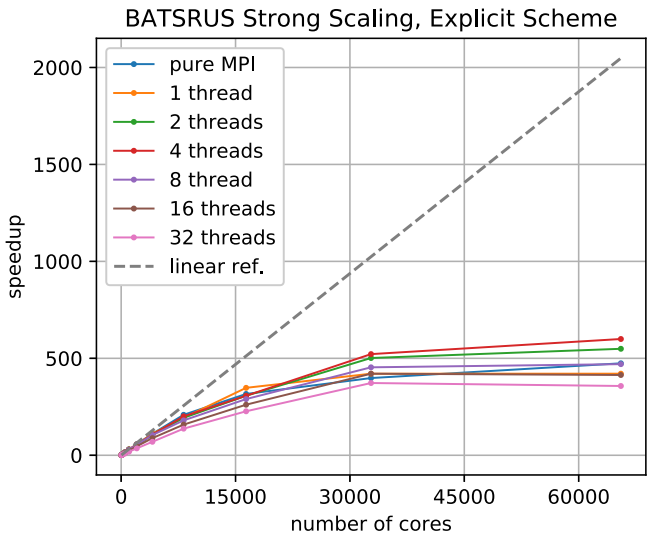


Fig. 2. Hybrid strong scaling of BATS-R-US on Blue Waters. A 3D shocktube problem is solved by the second order explicit Linde scheme [17] on a uniform grid with a total of 33 million cells using $8 \times 8 \times 8$ blocks. Each computing node has two AMD 6276 Interlagos processors, and a single XE node has 32 "integer" cores. The x-axis represents the total number of physical cores used, and the y-axis represents the speedup compared to 1 node (32 cores) run compiled without OpenMP. The number of blocks per MPI process ranges from 2048 to 1 for single thread runs. The dashed line indicates ideal linear scaling. gfortran 7.3.0 is used for compilation.

nodes, the multi-threaded runs show an advantage over the pure MPI runs, but there is no apparent trend on the optimal number of threads being used. Based on the asymptotic behavior of the scaling curves, the parallel portion of the code is estimated to be $\sim$ 99.8% both with pure MPI and hybrid MPI + OpenMP parallelization.

There is an interesting non-monotonic behavior of the speedup between different number of threads beyond 16,384 cores. For example, the 1 thread run performs the best for 16,384 cores but the second worst for 32,768 cores. The 2 threads and 4 threads run out-perform the rest for 65,536 cores. Given the same number of cores, more threads means more blocks per MPI process. The few number of blocks for the last few points on the lines may enhance the influence of random noises and block level overhead. Theoretically 4 threads should be the optimal choice given the NUMA domain architecture, as is the case shown beyond 30,000 cores.

5.3. Weak parallel scaling

Since our goal is to save memory with OpenMP parallelization, the weak scaling, with a fixed problem size per core, is the most relevant. We solve the same MHD problem as in the strong scaling test but now the number of grid cells is proportional to the number of cores and the available memory: there are 256 grid blocks with a total of 131,072 grid cells per core. BATS-R-US's capability of mixing explicit and implicit numerical schemes requires good scaling for both time stepping schemes. Here they are tested individually. For the explicit algorithm test, we select the second order Roe flux [21] with Koren's third order limiter [15]; for the implicit time stepping test, we select the second order Linde flux [17] with fixed (for sake of meaningful timings) 20 iterations of the BiCGSTAB [31] scheme. Figs. 3 and 4 show the performances for the explicit and implicit schemes, respectively. We note that while the explicit runs have $\sim$ 10 more cell updates per second compared to the implicit runs, the explicit time step (limited by the CFL condition) is much smaller than the implicit time step. For a general problem, the performance of the implicit schemes highly depends on the stiffness of the equation, the preconditioners, and the iterative methods being applied. A series of runs with 1, 2, 4, 8, 16, and 32 threads are shown with different colored lines. Note that for a fair comparison we fix the number of blocks per MPI process per thread, $\text{nBlock}/(\text{nMPI} * \text{nThread}) =$ constant, and with only 1 thread (red line), 2 threads, and 4 threads we ran out-of-memory at 2^{15}, 2^{16}, 2^{17} cores and beyond, respectively. The left panel of Fig. 4 shows that the 2-thread runs even beat the linear scaling calculated from the 1-thread base case.

The right panels of Figs. 3 and 4 show that the maximum number of cores the code can scale to strongly depend on the number of threads. The weak scaling is limited by the memory needed for storing the tree structure of the grid blocks, which is replicated for each MPI process to simplify the AMR algorithm. With 256 blocks per core, the total number of blocks reaches about 8.4 million when running the test on 32,768 cores. Storing the corresponding tree information, including extra storage for the parent blocks, requires $\sim$ 2 GB of memory for each MPI process. If the code is run with only one thread per MPI process, the tree information will exhaust the available 2 GB/core memory of the XE nodes of Blue Waters, so we cannot scale beyond about 16 thousand cores with pure MPI parallelization. By taking advantage of the shared memory access allowed by multi-threading, we are now able to scale up to $\sim 5 \times 10^5$ and $\sim 2.5 \times 10^5$ cores with less than 50% loss compared to the ideal linear scaling with the explicit and implicit time stepping schemes, respectively. This means that more than an order of magnitude larger problem sizes have become doable thanks to the hybrid parallelization.

For the largest simulation we have done, the total number of grid cells is about 70 billion. With 9 output variables and single precision output format, the total size of data is about 2 TB. We exclude all the I/O related performance issues from this paper and leave them as a future target.

In both the explicit and implicit scheme tests, there is a significant performance drop from 8 threads to 16 threads. This is probably caused by the connection speed bottleneck between NUMA domains for Blue Waters XE6 node architecture. There is another performance drop from 16 threads to 32 threads due to the connection speed between the sockets.

5.4. Performance details

We took a closer look at the 256 core run from the scaling tests, and plotted the absolute and relative timings for the main modules in Figs. 5 and 6. The *Advance* module is calling the other modules, so the timings for *Advance* include the timings for the other modules. Fig. 5 shows that for the explicit scheme 8 threads with 32 MPI processes performs almost the same as 1 thread with 256 MPI processes, but 32 threads with 8 MPI is much worse, mainly because of the update solution check for reducing timesteps for vastly changing variables and the message pass for ghost cell fillings. The timings show that modules without global communication (*FaceValue*, *FaceFlux*, *Update*, etc.) scale very well, while modules with global communications (reduction in *UpdateCheck*, ghost cell filling in *MessagePass* etc.) scale poorly with more threads than what can fit on a NUMA domain.

Similar module performance for the implicit scheme is shown in Fig. 6. As explained above, the current design of the iterative solver splits the block loops into several pieces, which requires individual multi-threading regions and unavoidable implicit barriers. The effect of thread launching overhead and barriers shows up even with 4 threads, and becomes quite significant with 16 threads.

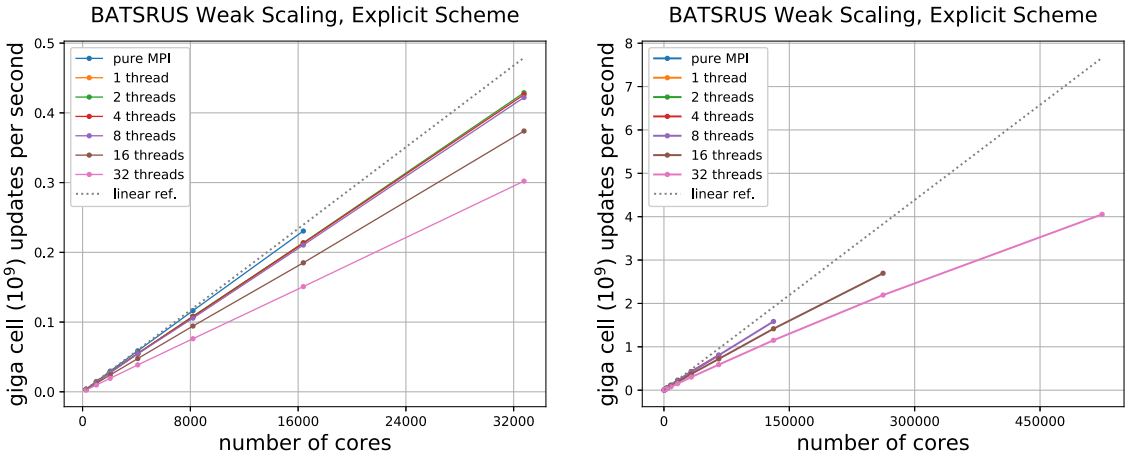


Fig. 3. Hybrid weak scaling of BATS-R-US on Blue Waters. Left shows the cell updates per second for 1 to 32 threads runs up to 32,768 cores and right shows for the runs up to 524,288 cores. A 3D shocktube problem is solved by the second order explicit Roe scheme on a uniform grid with 131,072 grid cells per core using $8 \times 8 \times 8$ blocks. The number of blocks allocated per MPI process varies from 256 to 8192 depending on the number of threads. The dashed line indicates ideal linear scaling starting from 256 cores.

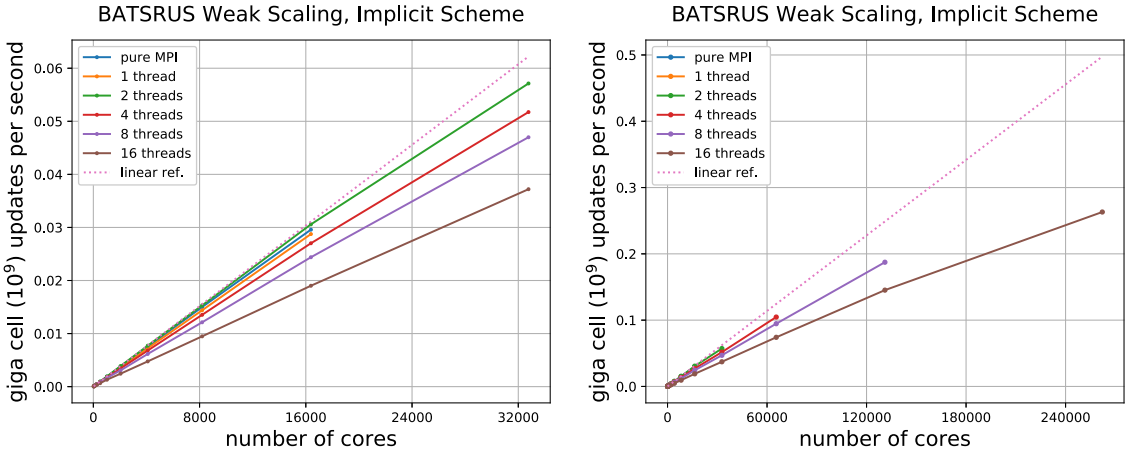


Fig. 4. Hybrid weak scaling of BATS-R-US on Blue Waters. Left shows the cell updates per second for 1 to 16 threads runs up to 32,768 cores and right shows for the runs up to 262,144 cores. A 3D shocktube problem is solved by the second order implicit Linde scheme with BiCGSTAB method on a uniform grid. There are 131,072 grid cells per core, and the block size is $8 \times 8 \times 8$. The number of blocks per MPI process varies from 256 to 8192 depending on the number of threads. The dashed line indicates ideal linear scaling starting from 256 cores.

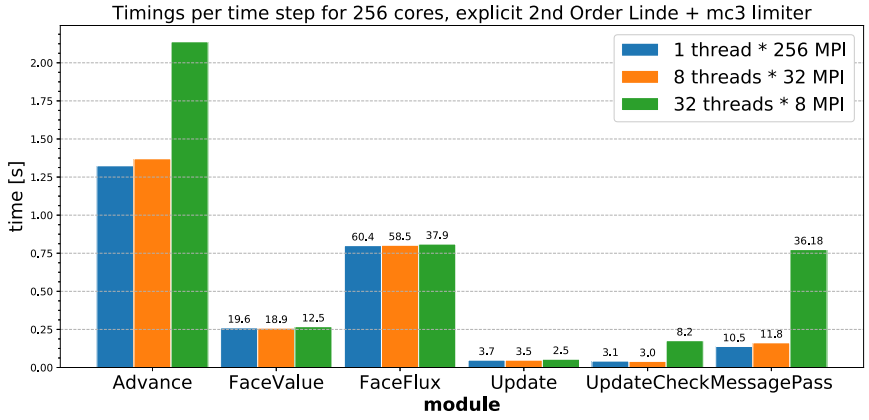


Fig. 5. Timings for the Brio–Wu MHD shock tube test per time step on Blue Waters XE nodes. Second order explicit Linde scheme [17] is applied on a uniform grid with a total of 33 million cells using $8 \times 8 \times 8$ blocks. *Advance* represents the total time marching part of BATS-R-US, including all the other timings on the right; *FaceValue* is the reconstruction of left/right states; *FaceFlux* is the approximate Riemann solver; *Update* is the solution update; *UpdateCheck* is the timestep adjustment in case of vastly changing variables; *MessagePass* is the ghost cell filling. The numbers on top of the bar plot is the percentage each module takes in the total timings of the simulation.

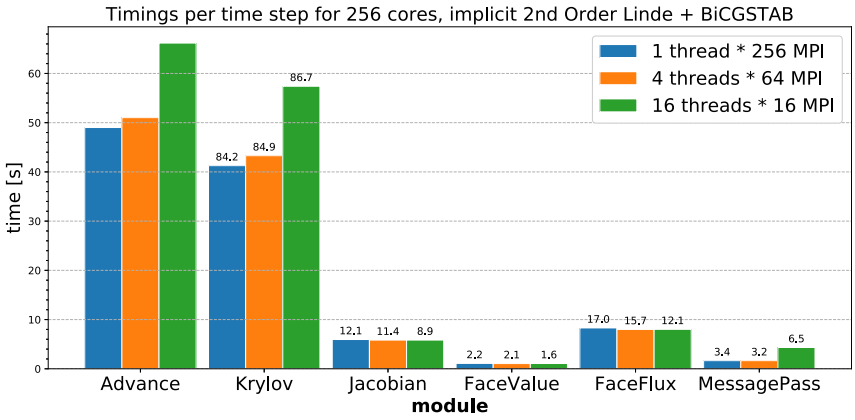


Fig. 6. Timings for the Brio–Wu MHD shock tube test per time step on Blue Waters XE nodes. Second order implicit Linde scheme [17] with BiCGSTAB method is applied on a uniform grid with a total of 33 million cells using $8 \times 8 \times 8$ blocks. *Advance* represents the total time marching part of BATS-R-US. *Krylov* represents the iterative solver which contains part of the face value, flux calculations and ghost cell fillings. *Jacobian* represents calculating the preconditioner. The numbers on top of the bar plot is the percentage each module takes in the total timings of the simulation.

5.5. Compiler and platform dependence

Many factors can influence the efficiency of the model besides the computational cost of the mathematical algorithm. The hardware specifications of different platforms and the compiling environments play a crucial role. As has been reported in the literature [14], a typical CFD application utilizes less than 20% of the available computing power due to the heavy memory usage with relatively few computations done to each grid cell. With all the optimizations we have done so far, BATS-R-US can achieve 18% computing efficiency in our MHD test that is representative of many BATS-R-US applications.

Besides the Blue Waters Cray computer with AMD cores, we also tested the performance on Stampede2 with Intel Skylake cores. The results of weak scaling are shown in Fig. 7. On this machine, the queuing system allows a maximum of 868 nodes per job, so we chose to go up to 512 nodes for the tests. As one can see from the comparisons, there is a much larger jump in performance between single and multi-threaded runs; then there is another jump when going beyond 12 threads, which is probably related to the NUMA domain architecture on Stampede2. Based on the profiling results of each individual module of BATS-R-US, the most complicated face flux calculation and the message passing are degraded in performance as we increase the number of threads, while the timings for the less complicated face value and source term calculations remain more or less the same. This may indicate that the Intel Fortran compiler has some issues in optimizing a large multi-threaded parallel region: the inlining of multiple levels of function calls, vectorizations of the innermost loops, clever memory management within functions may be restricted by the compiler due to the complexity of the source code.

We also observed a bizarre and significant slowdown of the code when running on a single OpenMP thread compared to the runs compiled without OpenMP on the Blue Waters XE nodes and Stampede2 SKX compute nodes using the Intel Fortran compiler. Similar issues have also been reported elsewhere [9,22]. Different combinations of the processor and compiler also show different behaviors. For example, we have performed tests on Blue Waters with the Cray, gfortran, ifort and pgf90 compilers. All four compilers has similar efficiency (within $\sim$ 15% difference) when compiled without OpenMP, but there are drastic differences in timing when compiled with OpenMP. For the single thread runs, the Cray and gfortran compilers show almost no degradation in performance, while on the other hand ifort becomes $\sim$ 2 times slower and pgf90 becomes $\sim$ 3 times slower. This is an overlooked issue as the 1-thread case is often the baseline for all the hybrid MPI-OpenMP scaling tests, as opposed to pure MPI compilation.

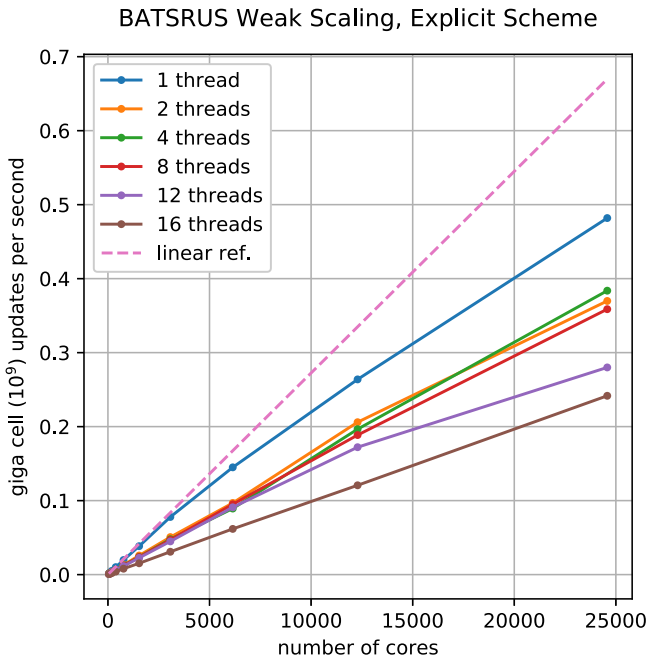


Fig. 7. Hybrid weak scaling of BATS-R-US on Stampede2 from 1 node (48 cores) to 512 nodes (24,576 cores). A 3D shocktube problem is solved by the second order explicit Roe scheme on a uniform grid with 131,072 grid cells per core using $8 \times 8 \times 8$ blocks. The number of blocks allocated per MPI process varies from 256 to 4096 depending on the number of threads. The dashed line indicates ideal linear scaling starting from 48 cores.

To demonstrate the problem in a simple manner, we take the standard shock tube test from the overnight test suite, and plot the timings for a series of compilation setup on 3 different platforms in Fig. 8. It scales well on the latest Frontera supercomputer with gfortran 9.0 and on our local Linux platform with gfortran 4.8, while significant degradation in performances is observed for the rest combinations.

Previous studies [9] tended to blame the overhead of managing the threads, however, it is puzzling for us that this overhead would make a difference on the order of seconds for our test runs when the thread startup overhead is often claimed to be on the

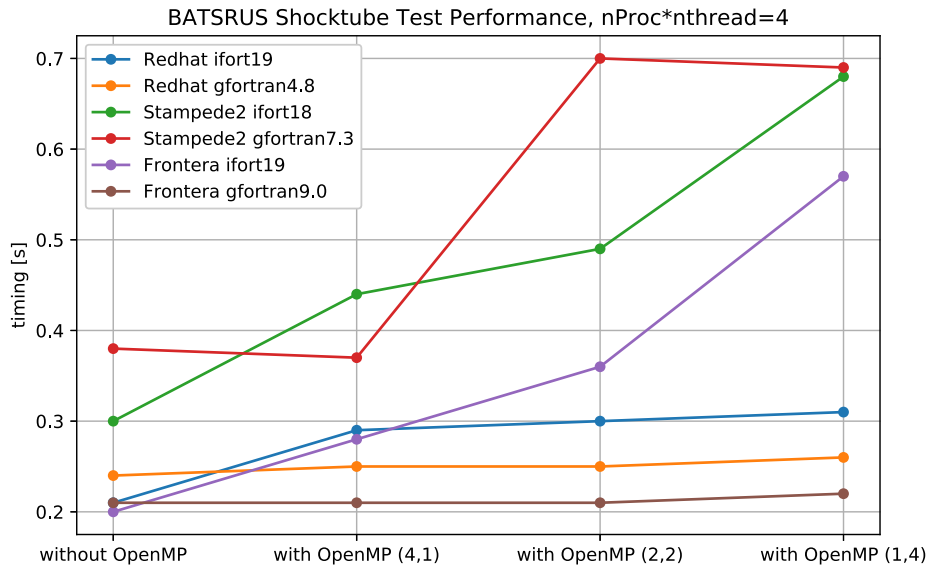


Fig. 8. Comparisons of BATSRUS compiling with or without OpenMP. Four vertical lines represent 4 MPI compiling without OpenMP, 4 MPI and 1 thread, 2 MPI and 2 threads, and 1 MPI and 4 threads. Three testing platforms with different compiler choices are present: Redhat Linux desktop, Stampede2 supercomputer, and Frontera supercomputer.

order of $\sim$ 100 ms. Two side-by-side test runs on Stampede2 compiled without any optimizations with OpenMP flags on and off suggest that this overhead alone could cost a 20% slowdown, which is one of the four kinds of OpenMP overhead summarized in [16]. The difference is more significant with aggressive optimizations ($-O2/-O3$), and the slowdowns tend to relate to the problem size being solved. Sequential consistencies might be an issue in compiler optimizations. OpenMP defines consistency as a variant of weak consistency: it is not allowed to reorder synchronization operations with read or write operations on the same thread. In addition to that, a flush operation (which guarantees the consistencies of shared variables) is implied by OpenMP synchronizations at entry/exit of parallel regions, implicit/explicit barriers, entry/exit of critical regions, and locks. The detailed hidden implementations to ensure sequential consistencies for different compilers are a possible explanation to the performance discrepancies in our tests.

6. Conclusion

In this work, we have successfully extended our finite volume/difference MHD code BATS-R-US from pure MPI to MPI + OpenMP hybrid implementation, with only 0.25% modification to the $\sim 250,000$ lines of source code. Good weak scaling performances are obtained up to $\sim 500,000$ cores with explicit time stepping and up to $\sim 250,000$ cores with implicit time stepping. Using the hybrid parallelization, we are now able to solve problems more than an order of magnitude larger than before thanks to the usage of shared memory for large grid arrays. We opted to use coarse-grained multi-threading applied to the loops over grid blocks, because it provides more opportunity for parallelism than the fine-grained approach applied to loops over grid cells. The main challenge with the coarse-grained approach in a large and complex code is to find and eliminate race conditions, which can be solved with well-organized code structure and the help of debuggers like Intel Inspector.

We also found that the compiler's capability to efficiently optimize a multi-threaded code varies significantly from compiler to compiler on various platforms. It is important to check the performance of the single-threaded execution of the code compiled with and without OpenMP library. Hopefully all compilers will get better in optimizing hybrid parallel codes in the future.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2020.02.004.

CRediT authorship contribution statement

Hongyang Zhou: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Visualization. **Gábor Tóth:** Methodology, Resources, Supervision, Funding acquisition.

Acknowledgments

The authors are thankful for the useful comments and suggestions by the reviewers. This research was supported by NSF INSPIRE, United States grant number PHY-1513379. The computational resources were funded by Blue Waters GLCPC, United States and NSF Frontera, United States.

Appendix

A.1. Thread affinity

Thread affinity means the binding and unbinding of a thread to a range of CPU cores, so that the thread will execute only on the designated core. It is critical in achieving good performance on large clusters, but the setup may not be trivial and requires special attention to different machines. For example, according to NASA's website [3], for MPI/OpenMP hybrid codes built with SGI's MPT library, all the OpenMP threads for the same MPI process have the same process ID. In this case, setting the `MPI_DSM_DISTRIBUTE` environment variable to 1 (which is the default) causes all OpenMP threads to be pinned to the same core and the performance suffers. For instance, on NASA's supercomputer Pleiades, we set `MPI_DSM_DISTRIBUTE` to 0 and use omplace for pinning in the job script. We also set `KMP_AFFINITY` to disabled so that Intel's thread affinity interface would not

interfere with `omplace`. On Blue Waters' XE node built by Cray, the job execution command aprun also had a conflict with Intel's library for pinning threads, and we had to turn off aprun's thread scheduler with `--cc=none`. Clearly the issue here is that multiple ways of controlling and managing threads interfere with each other such that unexpected performance penalty shows up. In general, thread affinity is highly machine-specific and requires special attention.

Tools have been developed to check the thread affinity on Linux machines. The `taskset` command line tool is the easiest one to check MPI/OPENMP bindings on clusters. The underlying C code `taskset.c` is publicly available in folder `schedutils/` at Linux utilities.

References

[1] Space weather modeling framework, 2010, http://csem.engin.umich.edu/tools/swmf/, accessed: 30-09-2010.

[2] Multithreaded flash, 2019, http://flash.uchicago.edu/jbgallag/2012/flash4_ug/node42.html#SECTION01033100000000000000, accessed: 07-11-2019.

[3] Porting with sgi's mpi and intel openmp, 2019, https://www.nas.nasa.gov/hecc/support/kb/porting-with-sgis-mpi-and-intel-openmp_104.html, accessed: 24-06-2019.

[4] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan kaufmann, 2001.

[5] Y. Chen, G. Tóth, T.I. Gombosi, A fifth-order finite difference scheme for hyperbolic equations on block-adaptive curvilinear grids, J. Comput. Phys. 305 (2016) 604–621.

[6] L. Dagum, R. Menon, Openmp: An industry-standard api for shared-memory programming, Comput. Sci. Eng. (1) (1998) 46–55.

[7] D.L. De Zeeuw, T.I. Gombosi, C.P. Groth, K.G. Powell, Q.F. Stout, An adaptive mhd method for global space weather simulations, IEEE Trans. Plasma Sci. 28 (6) (2000) 1956–1965.

[8] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, M. Wesenberg, Hyperbolic divergence cleaning for the mhd equations, J. Comput. Phys. 175 (2) (2002) 645–673.

[9] W.D.G. Dinesh K. Kaushik, David E. Keyes, B.F. Smith, Using memory performance to understand the mixed mpi/openmp model, 2019, https://www.mcs.anl.gov/research/projects/petsc-fun3d/Talks/pcfd00.pdf, accessed: 13-06-2019.

[10] N. Drosinos, N. Koziris, Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings, IEEE, 2004, p. 15.

[11] Z. Huang, G. Tóth, B. van der Holst, Y. Chen, T. Gombosi, A six-moment multi-fluid plasma model. J. Comput. Phys..

[12] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, High performance computing using mpi and openmp on multi-core parallel systems, Parallel Comput. 37 (9) (2011) 562–575.

[13] G. Jost, H.-Q. Jin, D. anMey, F.F. Hatay, Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster.

[14] D.E. Keyes, D.K. Kaushik, B.F. Smith, Prospects for cfd on petaflops systems, in: Parallel Solution of Partial Differential Equations, Springer, 2000, pp. 247–277.

[15] B. Koren, A robust upwind discretization method for advection, diffusion and source terms, in: Numerical Methods for Advection-Diffusion Problems, Vieweg, 1993, pp. 117–138.

[16] P. Lindberg, Performance obstacles for threading: How do they affect openmp code?, 2019, https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code, accessed: 13-06-2019.

[17] T.J. Linde, A Three-Dimensional Adaptive Multi Uid Mhd Model of the Heliosphere (Ph.D. thesis), The University of Michigan, 1998.

[18] O. Porth, C. Xia, T. Hendrix, S. Moschou, R. Keppens, Mpi-amrvac for solar and astrophysics, Astrophys. J. Suppl. Ser. 214 (1) (2014) 4.

[19] K.G. Powell, P.L. Roe, T.J. Linde, T.I. Gombosi, D.L. De Zeeuw, A solution-adaptive upwind scheme for ideal magnetohydrodynamics, J. Comput. Phys. 154 (2) (1999) 284–309.

[20] R. Rabenseifner, G. Hager, G. Jost, Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes, in: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2009, pp. 427–436.

[21] P.L. Roe, Approximate riemann solvers, parameter vectors, and difference schemes, J. Comput. Phys. 43 (2) (1981) 357–372.

[22] G.J.R.K. Rolf Rabenseifner, Georg Hager, Mpi + openmp and other models on clusters of smp nodes, 2019, https://pdfs.semanticscholar.org/e822/abaad40f1f9d2479df681f931943eee3003d.pdf, accessed: 13-06-2019.

[23] H.-Y. Schive, J.A. ZuHone, N.J. Goldbaum, M.J. Turk, M. Gaspari, C.-Y. Cheng, Gamer-2: a gpu-accelerated adaptive mesh refinement code–accuracy, performance, and scalability, Mon. Not. R. Astron. Soc. 481 (4) (2018) 4815–4840.

[24] L. Smith, M. Bull, Development of mixed mode mpi/openmp applications, Sci. Program. 9 (2–3) (2001) 83–98.

[25] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, et al., Stampede 2: the evolution of an xsede supercomputer, in: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, ACM, 2017, p. 15.

[26] Q.F. Stout, D.L. De Zeeuw, T.I. Gombosi, C.P. Groth, H.G. Marshall, K.G. Powell, Adaptive blocks: A high performance data structure, in: Supercomputing, ACM/IEEE 1997 Conference, IEEE, 1997, p. 57.

[27] G. Tóth, The lasy preprocessor and its application to general multi-dimensional codes.

[28] G. Tóth, D.L. De Zeeuw, T.I. Gombosi, K.G. Powell, A parallel explicit/implicit time stepping scheme on block-adaptive grids, J. Comput. Phys. 217 (2) (2006) 722–758.

[29] G. Tóth, B. Van der Holst, I.V. Sokolov, D.L. De Zeeuw, T.I. Gombosi, F. Fang, W.B. Manchester, X. Meng, D. Najib, K.G. Powell, et al., Adaptive numerical algorithms in space weather modeling, J. Comput. Phys. 231 (3) (2012) 870–903.

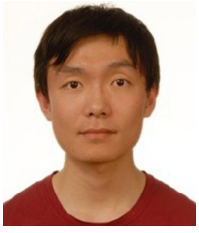
[30] G. Tóth, I.V. Sokolov, T.I. Gombosi, D.R. Chesney, C.R. Clauer, D.L. De Zeeuw, K.C. Hansen, K.J. Kane, W.B. Manchester, R.C. Oehmke, et al., Space weather modeling framework: A new tool for the space science community, J. Geophys. Res.: Space Phys. 110 (A12).

[31] H.A. Van der Vorst, Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (2) (1992) 631–644.

[32] C.J. White, J.M. Stone, C.F. Gammie, An extension of the athena++ code framework for grmhd based on advanced riemann solvers and staggered-mesh constrained transport, Astrophys. J. Suppl. Ser. 225 (2) (2016) 22.

[33] C. Xia, J. Teunissen, I. El Mellah, E. Chané, R. Keppens, Mpi-amrvac, 2.0 for solar and astrophysical applications, Astrophys. J. Suppl. Ser. 234 (2) (2018) 30.

[34] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, et al., Amrex: a framework for block-structured adaptive mesh refinement.



Hongyang Zhou, Research Assistant is a PhD student at University of Michigan, Ann Arbor. He has been working on the coupled MHD-EPIC simulation of Ganymede's magnetosphere. He also has research interest in high performance plasma simulations with various techniques.

Dr. Gábor Tóth, Research Professor is an expert in algorithm and code development for space and plasma physics simulations. He has a leading role in the development of Space Weather Modeling Framework that can couple and execute about a dozen different space physics models modeling domains from the surface of the Sun to the upper atmosphere of the Earth. He is one of the main developers of the BATS-R-US code, a multi-physics and multi-application MHD code using block-adaptive grids. He has participated in designing the software architect for the Center for Radiative Shock Hydrodynamics. He also designed the Versatile Advection Code, a general purpose publicly available hydrodynamics and MHD code.