Phase Asynchronous AMR Execution for Productive and Performant Astrophysical Flows

Muhammad Nufail Farooqi*, Tan Nguyen†, Weiqun Zhang†, Ann S. Almgren†, John Shalf† and Didem Unat*
**Koç University,* Istanbul, Turkey †*Lawrence Berkeley National Laboratory,* Berkeley, CA, USA
mfarooqi14@ku.edu.tr, {tannguyen, weiqunzhang, asalmgren, jshalf}@lbl.gov, dunat@ku.edu.tr

***Abstract*—Adaptive Mesh Refinement (AMR) is an approach to solving PDEs that reduces the computational and memory requirements at the expense of increased communication. Although adopting asynchronous execution can overcome communication issues, manually restructuring an AMR application to realize asynchrony is extremely complicated and hinders readability and long-term maintainability. To balance performance against productivity, we design a user-friendly API and adopt phase asynchronous execution model where all subgrids at an AMR level can be computed asynchronously.**

We apply the phase asynchrony to transform a real-world AMR application, CASTRO, which solves multicomponent compressible hydrodynamic equations for astrophysical flows. We evaluate the performance and programming effort required to use our carefully designed API and execution model for transitioning large legacy codes from synchronous to asynchronous execution up to 278,528 Intel-KNL cores. CASTRO is about 100K lines of code but less than 0.2% code changes are required to achieve significant performance improvement.

***Index Terms*—Asynchronous Runtime, Communication Overlap, CASTRO, AMR**

I. Introduction

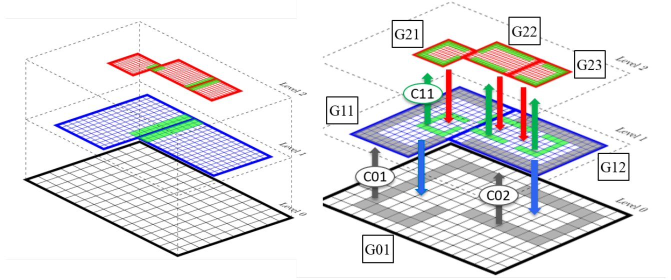
Domain-specific high-level software frameworks emerged to insulate domain scientists from the evolving programming models and complicated hardware architectures. These frameworks retain a balance between productivity and performance where domain scientists can focus on physics implementation while framework developers focus on getting maximum performance by exploiting advanced parallel programming models and hardware architectures. AMReX [1] is such a framework designed to develop block-structured Adaptive Mesh Refinement (AMR) applications and is currently being evolved to adapt to Exascale architectures. CASTRO [2], LMC [3], SMC [4] and Nyx [5] are among the many research codes in use today that are based on AMReX. There are variety of AMR techniques and this work discusses AMR in the context of time-subcycled blocked-structured AMR. Blocked-structured AMR employs multiple levels of refinements to solve partial differential equations where refined levels are created to focus on the region of interest rather than the entire domain.

Although AMR reduces the computation, it requires extra communication for inter-level data synchronization. To overcome the scalability problems due to the increasingly high communication cost, AMR developers can adopt one of the many asynchronous execution models. For example, Chan et al. [6] discussed four different execution models for AMR ranging from fully synchronous to fully asynchronous with different programmability tradeoffs. In fully synchronous execution, all ranks (processes) synchronize to complete communication before moving on to computation. Another variant of the synchronous execution is rank synchronous execution where synchronization within a rank occurs before moving on to computation on that rank. Phase asynchronous execution further elevates the synchronization to a refinement level where computation for a level is synchronized before moving on to computation at the next finer level. In fully asynchronous model, the computation on each subgrid at any refinement level can start as soon as its required data inputs are available.

Moving from synchronous to fully asynchronous execution increases performance but also brings tremendous programming complexity. In the AMR context, asynchronous programming requires explicitly declaring and handling subgrid's data dependencies, scheduling a subgrid for computation upon successful completion of its data dependencies and triggering data communication upon computation of a subgrid. Although this seems quite intuitive in theory but it is hard to program in practice and requires major modifications on the entire application to expose potential asynchrony at all levels. In this paper, we show that phase asynchrony provides an application with both the performance and productivity benefits and is a viable execution model for AMR applications transitioning from synchronous to asynchronous models.

In this work, we present the integration of the phase asynchronous execution model with the help of a legacy application, called CASTRO [2]. This is an application for compressible astrophysical flows developed using the AMReX framework and being used for simulations of Type Ia supernovae and core-collapse supernovae. CASTRO employs the multigrid method in its radiation transport and Poisson gravity modules, and also has the option of performing synchronous solve on multiple AMR levels. These solvers put restrictions on asynchronous execution because the algorithm introduces necessary synchronization points. Therefore, we restrict the work in this paper on the hydrodynamics solver with constant gravity. CASTRO underlines the challenges of the communication overlap technique when done manually. Specifically, CASTRO employs complex data structures and input-dependent communication activities, which cannot be identified easily at compilation time. As a result, manually modifying the source code to overlap communication with computation is not a viable approach. Thus, we represent

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE



Intra-level communication Inter-level communication

Fig. 1: AMR with 2 refinement levels in 2D

the CASTRO code as a task dependency graph and build an AMR runtime system that can identify chances to overlap as the program executes. We show how our runtime facilitates the expression of different AMR communication types and utilizes hardware resources for work assignment for tasks. Lastly, we study the impact of asynchronous execution on load balancing and the use of hyperthreading to improve both the communication and computation. In summary, we make the following contributions in this paper:

- Integrate the phase asynchronous AMR algorithm into the AMReX framework.
- Unveil the simple application programming interface to reduce programming effort
- Discuss the experience in porting a large astrophysics application, CASTRO, that solves compressible hydrodynamics equations and quantify the programming effort required for porting this legacy AMR application
- Provide implementation details of the runtime and its hyperthreading support for communication threads
- Study two load balancing strategies and perform scaling studies on Cori-KNL and Cori-Haswell supercomputer using up to 278,528 cores.

II. Background

A. AMReX: An AMR Framework

AMReX [1], the successor of BoxLib, is a publicly available AMR framework being developed as part of the US Department of Energy's Exascale Computing Project. It is designed for developing applications solving hyperbolic, parabolic and elliptic partial differential equations on a hierarchy of block-structured adaptive grids with supports for particles and the embedded boundary method for complex geometry. It is the basis of many multi-scale multi-physics application codes.

AMReX organizes the adaptive meshes into a collection of levels where each level has a different numerical resolution (i.e., size of numerical cells). Starting from level *0*, the coarsest level, finer refinement levels are added in regions where finer resolution is desired until the desired level of accuracy or the predefined maximum number of levels is reached. The grids evolve adaptively to meet the resolution requirement as the system evolves. Each AMR level consists of a number of non-overlapping logically rectangular subgrids. The union of the subgrids on a fine level is properly contained by the union of the subgrids on the adjacent coarser level. However, there is no direct parent-child connection between coarse and fine level subgrids. Thus the adaptive mesh grids in AMReX cannot be represented by a tree. Fig. 1 shows an example of AMR grids in 2D with two levels of refinements where level *1* contains two subgrids and level *3* contains three subgrids.

In AMReX, a subgrid's data along with its metadata is stored in a *Fab*. All subgrids at a single level are grouped in *MultiFab*. In the rest of the manuscript, we refer to subgrids as *Fab* and union of *Fabs* in a single level as *MultiFab*. AMR typically requires two types of communication between *Fab*s. First one is the intra-level communication, which occurs between *Fab*s at the same AMR level. This is a typical ghost cell exchange that commonly appears in structured grid computations. Second is the inter-level communication, which occurs between consecutive AMR levels with additional computation besides communication. *Prolongation* interpolates data from coarser *Fab* to a finer one while *Restriction* averages the data from a finer to a coarser *Fab*, both causing inter-level communication. In Fig. 1, shown with arrows, the grey cells from level *0* are interpolated and copied to the grey cells at level *1* and green cells from level *1* to green cells at level *2*. During *Restriction*, data from all cells at level *2* is averaged out and copied to the cells enclosed in the green cells at level *1* and data from all cells at level *1* is averaged out and copied to the cells enclosed in the grey cells at level *0*.

B. Phase Asynchronous Execution

In this work we adopt the phase asynchronous execution model described in [7]. In phase asynchronous execution, *Fabs* at an AMR level can be computed independent of other *Fabs* at the same level as long as their communication dependencies are fulfilled. Fig. 1 illustrates the workings of phase asynchronous model. After computing of Fab *G01* at level *0*, communication messages *C01* and *C02* are asynchronously sent to Fabs *G11* and *G12* at level *1*. After the reception of message *C01*, unlike synchronous execution, Fab *G11* can be computed without waiting for the reception of message *C02*. Once Fab *G11* is done, message *C11* can be sent to Fab *G21* at level *2*. However, phase asynchrony requires *G12*'s computation to be completed before computation starts at the next level. For example, let us consider a scenario where *C11* arrives before *C02*, in a fully asynchronous execution model *G21* computation can be started immediately but in phase asynchronous execution its computation will be delayed because *G12* at a lower level is still not computed. Delaying subgrid computation at finer levels, while there are still grids to be computed at coarser levels, can cost a little performance loss but limiting asynchronous execution to AMR level enable us to design a user-friendly API for porting legacy codes.

C. CASTRO

CASTRO [2], [8], [9] is an open source astrophysical radiation hydrodynamics code that is developed using the AMReX framework. An unsplit piecewise parabolic method is employed for hydrodynamics. CASTRO supports nuclear reaction networks, realistic equations of state for stellar astrophysics, and Poisson gravity. The radiation solver is based on the flux-limited diffusion approach that supports both gray and multigroup radiation transport in either co-moving or mixed frame. CASTRO has about 100,000 lines of C++ and Fortran codes. C++ is used as the main driver and the computational kernels are written in Fortran. A hybrid MPI+OpenMP approach is used for parallelism. CASTRO has been under active development since 2007 and has been used for numerous astrophysical simulations and in particular simulations of Type Ia supernovae, core-collapse supernovae and pair instability supernovas (e.g., [10]–[12]).

CASTRO uses subcycling in time and the time stepping is driven by recursively evolving AMR levels. For example, for refinement ratio of two, level *0* first advances its data for a time step of dt_0 followed by two calls to advance level *1* data each with a time step of $dt_1 = dt_0/2$. Each advance of level *1* contains two calls to advance level *2* data each with a time step of $dt_2 = dt_1/2$. For a total of three levels, a complete coarse time step consists of evolving levels *0, 1, 2, 2, 1, 2,* and then *2*. To advance a fine level requires temporal and spatial interpolation of coarse level data.

III. Runtime User Interface

Exposing details of programming models to the application programmers can enable them to fine tune the performance but on the other hand will require increased programming effort and divert their focus from domain science to performance engineering. This is particularly true when asynchronous execution is desired. A well designed API with right abstractions can reduce the efforts to develop an asynchronous execution model in an application, ideally without sacrificing performance.

In close collaboration with domain experts, we develop the runtime API for common use cases that can appear while implementing AMR applications. We choose a task dependency graph representation in which a node represents computations on a Fab and an edge denotes data dependency among a pair of tasks. We design nodes and edges of the graph so that they can be easily constructed from the mesh structure provided by AMReX. In particular, there are two major parts of the API for the asynchronous execution that we expose to the application programmers. One is extracting metadata embedded in *MultiFab* which is used to infer data dependencies (edges) by the runtime. The other is a graph iterator that iterates over all the *Fab*s in a *MultiFab*. Each *Fab* is treated as a task of the graph, which may be run as soon as its data dependencies are satisfied. The rest of this section presents details of metadata extraction and graph iterator.

A. Metadata Extraction

For metadata extraction, we introduce a new virtual function *initMetaData()* to AMReX, which is automatically called at the first iteration or after each regridding. Application programmers need to override the *initMetaData()* function to enable the metadata extraction. Application programmers also need to implement *deleteMetaData()* virtual function to delete all the task graphs created inside *initMetaData()*. Based on the type of communication required for a *MultiFab*, following three types of cases can occur:

ExchangeGhost:

```
tg = TaskGraph( myMultiFab )
extractExchangeGhost(tg, myMultiFab, ...)
graphArray.push_back(tg)
```

FillPatch:

```
ap = AsyncFillPatch( myMultiFab, StateID, ...)
```

Restriction:

```
tg_src = TaskGraph( srcMultiFab )
tg_dst = TaskGraph( dstMultiFab )
extractRestriction(tg_src, tg_dst,
srcMultiFab, dstMultiFab, ...)
graphArray.push_back(tg_src)
graphArray.push_back(tg_dst)
```

Fig. 2: Asynchronous runtime's API for metadata extraction. For clarity, the framework specific parameters are omitted.

1) **ExchangeGhost:** *ExchangeGhost* is intra-level communication required to fill the ghost cells of all *Fab*s inside a *MultiFab*. *ExchangeGhost* is required whenever some computations that change the data, are performed on a *MultiFab* and its ghost cell data is required for subsequent computations.

To extract the metadata for *ExchangeGhost*, programmers first need to create a task graph that will hold the metadata as shown in the code in Fig. 2. Metadata is then loaded from the *MultiFab* into the task graph using the *extractExchangeGhost()* subroutine. Finally, the task graph is added to an array of task graphs *graphArray*, which is a member of the *Amr* class. The *Amr* class in the AMReX framework contains all the AMR related functions. Execution of all the task graphs inside *graphArray* is handled transparently by our asynchronous runtime system.

2) **FillPatch:** This type of communication contains both intra-level and inter-level communication. The purpose of intra-level communication is the same as described in *ExchangeGhost* while inter-level communication is performed for *Prolongation*. *Prolongation* is required for the *Fab*s at the finer level whose ghost cell data cannot be obtained from within their level or boundary conditions. *Prolongation* interpolates data from coarser level and copies it to the adjacent finer level.

Handling this type of communication is tricky and cumbersome because it involves intra-level, inter-level communication and interpolation both in time and space. More specifically, *Prolongation* involves *MultiFab*s (old and new) from current level and a target *MultiFab* at the finer level, where old and new *MultiFab*s represent data from previous and current timesteps, respectively. *ExchangeGhost* is performed for target *MultiFab* at the finer level. Data from the old and new *MultiFab*s at current level is interpolated in time to a temporary MultiFab and data from that temporary *MultiFab* is interpolated in space and copied to target *MultiFab* at the finer level.

To make the application programmer's job easier we implement a special class, *AsyncFillPatch* for this purpose. As shown in Fig. 2, the application programmer only needs to create an object of the class *AsyncFillPatch*. AMReX handles all the old and new timestep's data allocation for problem domain variables internally, where each variable is assigned an identification number called *StateID*. Constructor of the *AsyncFillPatch* class requires *StateID* of the variable for which *FillPatch* is needed. All the task graph creation, metadata extraction and pushing the task graphs to the *graphArray* are transparently handled inside the class constructor.

3) **Restriction:** *Restriction* is the inter-level communication performed at the end of an iteration to synchronize the data from finer levels to the adjacent coarser level and it is carried out to reflect more accurate values computed using finer mesh on the coarser mesh. During *Restriction* the points at finer level are averaged out and copied to the corresponding point at the coarser level.

As shown in the code in Fig. 2, two task graphs, one for each level, are created to hold the metadata for *Restriction*. Metadata is then extracted using the *extractRestriction()* subroutine, where the task graph holding the finer level metadata serves as a source and task graph holding the coarser level metadata serves as a destination. Both task graphs are then added to the task graph array *graphArray* so that runtime's communication handler can process them.

B. Task Graph Iterators

AMReX provides a *MultiFab iterator* that sequentially iterates over all the *Fab*s within a single *MultiFab*. Fig. 3 shows the *AMReX*'s iterator where *MFIter* is the class that implements this functionality. The iterator is implemented using a *for* loop where an object of *MFIter* is created based on the provided *MultiFab* object. The *MFIter* class implements *isValid()* subroutine to check whether there are any further *Fab*s to be processed. Increment operator (++) is overloaded to point to the next *Fab* in the *MultiFab*. Inside the loop body computation on a single *Fab* is performed.

To keep the API consistent with *AMReX*, we develop a task graph iterator with a similar interface to the *AMReX*'s *MultiFab* iterator as shown in Fig. 4. The functionality of the task graph iterator is implemented inside the *TGIter* class. The major difference between the iterators is that the *MultiFab* iterator schedules *Fabs* sequentially while task graph iterator schedules *Fabs* (tasks) out-of-order. The order depends on the completion of task dependencies. Furthermore, as shown in Fig. 4 the communication data received by the runtime is required to be pulled when a task is first scheduled and the communication data is pushed to the runtime for onwards communication at the end of the computation. These send and receive function calls are non-blocking; they handover communication data to the runtime for overlapping communication with computation.

MultiFab Iterator:

```
ExchangeGhost(myMultiFab, ...)
for(MFIter mfi( myMultiFab ); mfi.isValid();
 ++mfi)
{ //compute(mfi) }
```

Fig. 3: AMReX's MultiFab iterator

Task Graph Iterator:

```
for(TGIter tgi( tg ); tgi.isValid(); ++tgi)
{
  ExchangeGhost_receive(tgi, myMultiFab, ...)
  //compute(tgi)
  ExchangeGhost_send(tgi, myMultiFab, ...)
}
```

Fig. 4: Asynchronous runtime's task graph iterator.

Following are different varieties of the task graph iterator based on the use case scenarios:

Iterator for single dependency graph: This type of iterator deals with a single dependency graph. This iterator can be created by providing the task graph that is created for either an intra-level or an inter-level communication as shown in Fig. 4. It iterates over all the tasks and schedules a task as soon as the task's dependent communication is completed.

Iterator for multiple dependency graphs: This type of iterator is created when both intra-level and inter-level communication or multiple intra-level or multiple inter-level communication task graphs are required to complete the communication. The iterator schedules a task as soon as all the dependent communication from multiple graphs is completed by the runtime. To create this type of iterator either an object of class *AsyncFillPatch* is passed or multiple task graph objects for which the communication is awaited are passed to the constructor of the *TGIter* class.

Iterator with implicit communication calls: To provide more parallelism and ease the usage of the iterator, we implemented an alternative iterator that automatically and transparently handles pulling the data before computation and pushing of the data after computation. This type of iterator can be created by passing additional parameters like *StateID*, number of ghost cells, current iteration number and current simulation time along with the task graph or the *AsyncFillPatch* object.

This type of iterator allows us to perform thread-level optimizations. The iterator sets aside one dedicated worker thread for pushing and pulling of the communication data. While rest of the workers are busy computing a task, this dedicated thread handles pulling the communication data for the ready tasks or pushing the communication data for the already computed tasks. This type of iterator is convenient to use but may not be always applicable because both sending and receiving data for the same *MultiFab* may not be carried out in the same loop but rather data might be received in one loop and sent in some later loop as is the case in the CASTRO code. This iterator is mostly suitable for rather small applications where all computation can be gathered in a single loop.

As in many frameworks, AMReX owns the memory allocation and thread/process management therefore an OpenMP parallel region is created inside the asynchronous AMReX. Execution enters the parallel region before starting computation at level *0* and exits after *Restriction* is completed for level *0*. Application programmers should be careful while inserting their own MPI or OpenMP calls as this may break the code.

IV. CASTRO Implementation

To port an existing AMReX application to get benefit of the asynchronous runtime, application programmers need to follow these steps: 1) Implement the *initMetaData()* subroutine to create task graphs and extract metadata for *MultiFab*s that need to be communicated during execution. 2) Reorganize the structure of the application and replace *MultiFab* iterators (*MFIter*) with task graph iterators (*TGIter*). 3) Replace communication subroutines with their asynchronous *send* and *receive* equivalents at the appropriate places.

Extracting metadata is fairly straightforward and already explained in the API section. Here we will focus more on the control structure organization of CASTRO.

A. Control Structure (Synchronous)

The equation solved by CASTRO is in a conservation law form with source term.

$$\frac{\partial \vec{U}}{\partial t} + \nabla \cdot \vec{F} = \vec{S}. \quad (1)$$

Here $\vec{U}$ is the conservative state variable, $\vec{F}$ is the hyperbolic flux of hydrodynamics, and $\vec{S}$ is the source term for gravity, reaction, etc. The main method in CASTRO employs Strang splitting to achieve second-order accuracy for multiphysics problems with smooth hydrodynamic flows coupled with gravity and reaction. At the beginning of a timestep, the state is evolved for half time step with the source term. Then the hydrodynamics solver is called for evolving the state for whole time step. Finally the state is evolved again for half time step. The hydrodynamics solver performs characteristic tracing involving ghost cell data. Thus the ghost cells need to be properly updated after the first half time step with the source term.

Fig. 5 shows the control structure of the synchronous CASTRO. The figure only shows the computation performed within a single level at a single time step. All the variables used in the figure are *MultiFab*s unless stated otherwise. Before the start of an iteration, pointers to the data from previous time step and current time step are swapped, and the temporary variables that are used during the time step are reset (memory reallocation). Ghost cells for the *phi_old* from the current level and the adjacent coarser level are filled by using the *FillPatch()* subroutine where *phi_old* is a *MultiFab*. Next, old sources are constructed where *src_old* is an array of *MultiFab*s, where each *MultiFab* represents a different source. *ExchangeGhost* is only called for the external old source *old_src[ext_src]* if present. After construction of the hydro source *hydro_src*, *ExchangeGhost* is called for both *hydro_src* and *dSdt_new*. The rest of the code involves computation of *hydro_src, fluxes, new_src,* nuclear reactions and *phi_new*.

B. Control Structure (Asynchronous)

The asynchronous control structure is influenced mostly by the communication. A general guideline for the asynchronous execution model is to *send* communication data as early as possible and run compute tasks as soon as required data arrive. This strategy allows to attain the maximum achievable overlap of communication with computation.

Asynchronous control structure of CASTRO is shown in Fig. 6. After swapping old and new pointers and resetting memory allocation of the variables, we post sending *dSdt_new* early to overlap its communication with construction of sources. Calling *ExchangeGhost* for *dSdt_new* at an earlier location in the synchronous version would not produce any benefit because communication is performed synchronously and the communication time is always the same. The asynchronous version allows us the flexibility to post *sends* earlier and hide the communication cost. The loop here uses a task graph iterator constructed with the *AsyncFillPatch* class object created for *phi_old* because *FillPatch* is required for *phi_old* and computation of old sources requires *phi_old* with updated ghost cells. At the end of the loop body, we start sending the communication data from *old_src[ext_src]* to the dependent tasks.

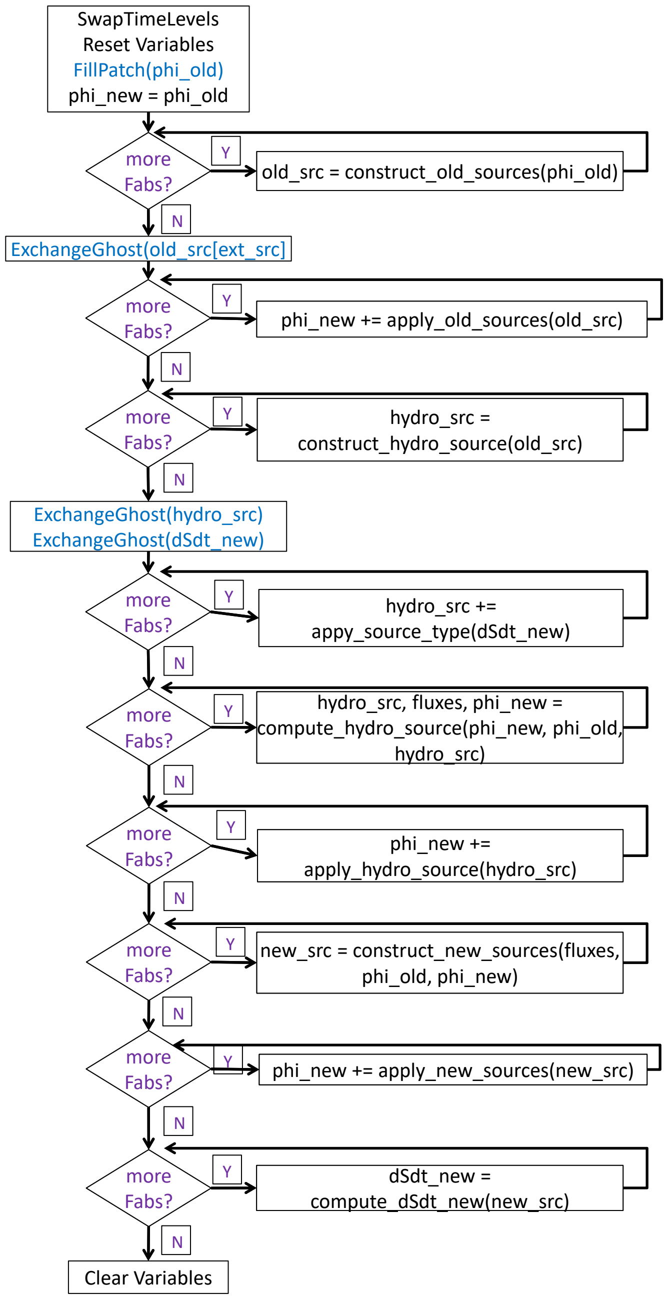
In the subsequent loop, we merge computation of *phi_new* and *hydro_src* in the loop body. Merging these two computations increases the compute time. As a result this allows more time for *old_src* to be received and *hydro_src* to be sent for the subsequent computation.

The next loop body merges the rest of the computation. The computation here depends on both *hydro_src* and *dSdt_new* so the *iterator for multiple dependency graphs* is used for creating the loop iterator. The task graphs of both *MultiFab*s are used to create the iterator. At the end of the loop body, *FillPatch_Send()* is called for *phi_new* that sends the communication data to the finer level. If current level is the finest level then the communication data is sent for the next time step of the current level.

C. Programming Effort

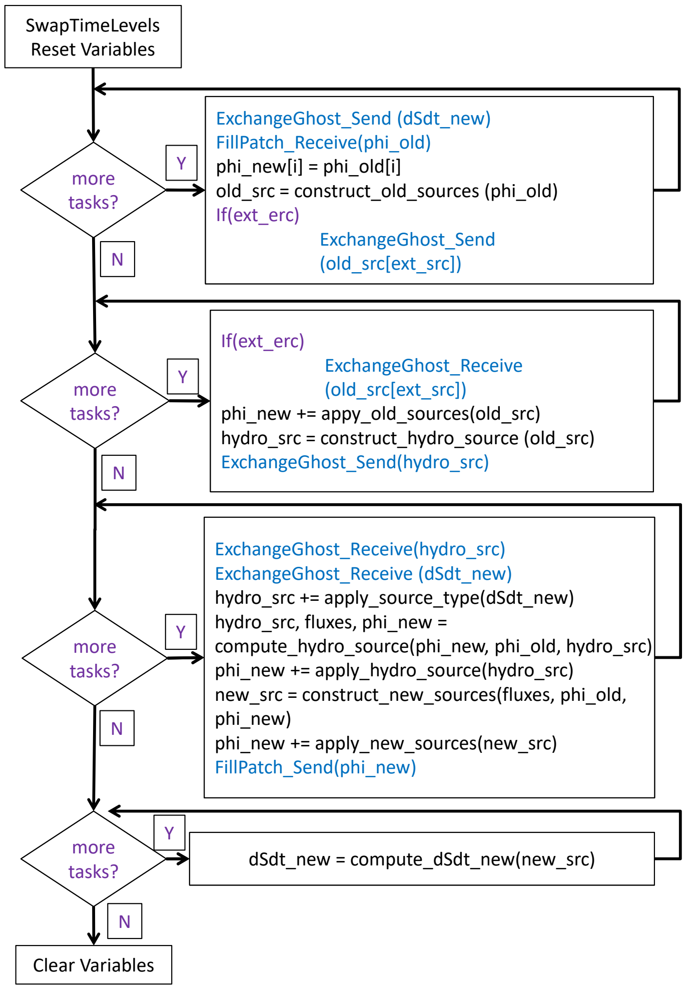
The programming effort required to port the legacy AMReX applications to asynchronous AMReX is reasonably low. Most of the effort is required for understanding the application organization and communication and computation dependencies. If the programmer is already familiar with the application then the coding effort is negligible compared to the size of the application.

CASTRO is large code base of 100K lines of code where nearly 25K lines consist of C++ control flow code implemented in AMReX and the rest of the code comprises Fortran computation kernels. The code implements four physics



SwapTimeLevels
Reset Variables
FillPatch(phi_old)
phi_new = phi_old
more Fabs?
Y
old_src = construct_old_sources(phi_old)
N
ExchangeGhost(old_src[ext_src])
more Fabs?
phi_new += apply_old_sources(old_src)
more Fabs?
hydro_src = construct_hydro_source(old_src)
ExchangeGhost(hydro_src)
ExchangeGhost(dSdt_new)
more Fabs?
hydro_src += appy_source_type(dSdt_new)
more Fabs?
hydro_src, fluxes, phi_new = compute_hydro_source(phi_new, phi_old, hydro_src)
more Fabs?
phi_new += apply_hydro_source(hydro_src)
more Fabs?
new_src = construct_new_sources(fluxes, phi_old, phi_new)
more Fabs?
phi_new += apply_new_sources(new_src)
more Fabs?
dSdt_new = compute_dSdt_new(new_src)
Clear Variables

Fig. 5: CASTRO's control structure (Synchronous)



SwapTimeLevels
Reset Variables
more tasks?
Y
N
ExchangeGhost_Send (dSdt_new)
FillPatch_Receive(phi_old)
phi_new[i] = phi_old[i]
old_src = construct_old_sources (phi_old)
If(ext_erc)
ExchangeGhost_Send (old_src[ext_src])
more tasks?
If(ext_erc)
ExchangeGhost_Receive (old_src[ext_src])
phi_new += apply_old_sources(old_src)
hydro_src = construct_hydro_source (old_src)
ExchangeGhost_Send(hydro_src)
more tasks?
ExchangeGhost_Receive(hydro_src)
ExchangeGhost_Receive (dSdt_new)
hydro_src += apply_source_type(dSdt_new)
hydro_src, fluxes, phi_new =
compute_hydro_source(phi_new, phi_old, hydro_src)
phi_new += apply_hydro_source(hydro_src)
new_src = construct_new_sources(fluxes, phi_old, phi_new)
phi_new += apply_new_sources(new_src)
FillPatch_Send(phi_new)
more tasks?
dSdt_new = compute_dSdt_new(new_src)
Clear Variables

Fig. 6: CASTRO's control structure (Asynchronous)

modules; compressible hydrodynamics, self-gravity, nuclear reactions and radiations. It may take about a month to understand the CASTRO code organization and communication dependencies but it will arguably take a couple of days to port it to the asynchronous AMReX. We ported the hydrodynamics solver with constant gravity, where a few dozen lines of additional code is required for metadata extraction. Nine *MFIter*s are replaced with four *TGIter*s. Communication subroutines are replaced with their *send* and *receive* equivalents resulting in an additional dozen lines of code. In total, less than 200 lines of code is required to be either changed or added to port these modules.

V. Implementation of Phase Asynchrony

The asynchronous execution model we employed in this work is inspired by the Perilla runtime system [13] and by the phase asynchronous AMR algorithm proposed in [7]. We incorporate the phase asynchronous AMR algorithm and its execution model into the *AMReX* framework and apply the method to a large and complex CASTRO application. In phase asynchrony, each *Fab* is considered as a task. A task within an AMR level can perform its computation and post communication to other levels independent of other tasks at the same level as soon as its ghost cell data is available.

Our runtime uses a data-driven and graph-based model. The input of the runtime is a task graph, which is a collection of tasks connected via dependencies on data. We implement the runtime on top of MPI and OpenMP. The programmer can configure the runtime with a single or multiple OpenMP thread teams per MPI process, where OpenMP threads are equally distributed among all teams. If a single MPI process is created per node then a single thread team per socket is effective to take advantage of NUMA-aware memory accesses. Multiple

MPI processes per node can be also created to utilize network bandwidth more effectively. Depending on the architecture of the machine, number of MPI processes and thread teams can be configured to balance the number of MPI processes and teams per node.

A. *Thread-Work Assignment*

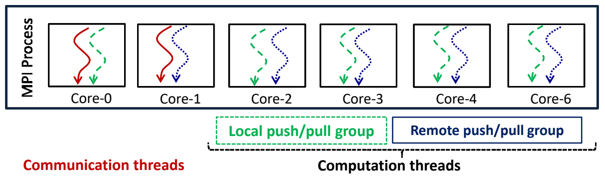
Each MPI process of the runtime system consists of a local and a remote communication handlers that service intra and inter-process communication requests, respectively. Within a team, some threads are dedicated to handle communication while some threads (worker threads) are responsible for computation. The runtime schedules one task at a time for computation while communication is being carried out for all other tasks. To keep all computation threads busy, tiling [14], [15] within each grid is used for load distribution among threads, where a number of tiles are assigned to each thread.

Modern architectures often provide multiple hardware threads per core and we utilize hyperthreading within a team in the runtime. A challenge posed by using hyperthreads is to evenly divide computation and communication work among all the threads. The work should be divided so that the threads being idle are kept to the minimum because the work does not increase as the number of available hardware threads increases. To further increase parallelism, the runtime distributes work among the communication threads and among the worker threads when they are pushing or pulling the communication data. We spare a few threads for handling communication. When hyperthreading is enabled, one of the communication threads is dedicated for remote (MPI) communication, one is dedicated for local ghost cell exchange, and one is dedicated for local prolongation and restriction. If two hyperthreads are spared for communication, we combine local communication into one. If there is no hyperthreading, then all these different types of communication are handled by a single thread. Besides doing computation, the worker threads are divided into two groups where one group pushes or pulls the local communication data while the other group pushes or pulls the remote communication data.

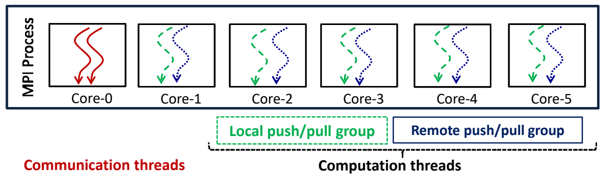
An example thread distribution for the runtime using scatter and compact affinities for communication threads is shown in Fig. 7a and Fig. 7b, respectively. Assuming two hyperthreads per core, we set aside two threads for communication and assign them to the available cores as shown with red lines in the figure. Rest are computation threads and during push or pull operations these are divided into local and remote groups as shown with green-dashed and blue-solid lines in the figure, respectively. Communication threads can adhere to scatter or compact affinity independent of the compute threads' affinity.

B. *Load Balancing*

Load balancing strategies for AMR usually focus on evenly distributing the number of cells among the processes. However, the communication overhead for several small subgrids is not equivalent to one large subgrid even if they contain equal number of cells. Performance improvement in phase asynchronous execution model is sensitive to the computation versus communication ratio [7]. Thus, turning a blind eye to the number grids per process and only focusing on the even distribution of the number of cells per process may work well for the synchronous execution models but can cause deviation in execution times among different processes in phase asynchronous model. We analyze two different load balancing strategies, Space Filling Curve (SFC) and Round Robin (RR), for phase asynchronous model. SFC distributes the subgrids taking into the account the locality of the subgrids, placing the neighboring grids on the same process to reduce communication. On the other hand, RR sorts all subgrids based on number of cells and distributes the sorted subgrids in a round robin fashion among the processes.



(a) scatter affinity for both communication and compute threads



(b) compact affinity for communication threads, scatter affinity for compute threads

Fig. 7: Thread-work assignment

VI. Evaluation

For performance studies we used Cori supercomputer located at Lawrence Berkeley National Laboratory, CA, USA. We conducted experiments on both the Intel's Haswell and Intel's Xeon Phi (Knights Landing) architectures. The latter is configured with quadrant cluster mode and MCDRAM as the last level cache. Node specifications for both machines are provided in Table I.

There are several modules in CASTRO. We use three dimensional Rayleigh-Taylor (R-T) and Sedov-Taylor (S-T) as test problems. A brief description of both problems is given in Appendix. All experiments use two subcycling iterations and a refinement ratio of two. We use two refinement levels and a larger mesh size for R-T while three refinement levels and slightly smaller mesh size for the S-T test problem.

A. *Input Configuration Selection*

Table II lists the input configurations used for all the experiments. For a fair comparison, we selected the parameters based on the best performance for the synchronous AMReX. Tiling within a *Fab* is enabled to create OpenMP parallelism supported by AMReX. For communication threads, we use

	Cori-Haswell	Cori-KNL
CPUs	Haswell Intel Xeon E5-2698 v3	Knights Landing Intel Xeon Phi 7250
Sockets	2	1
Cores/socket	16	68
Threads/core	2	4
Clock Rate	2.3 GHz	1.4 GHz
LLC	40 MB	16 GB
Total main memory	128 GB	96 GB
Memory bandwidth/socket	68 GB/s	102 GB/s
Interconnect	Cray Aries Dragonfly	Cray Aries Dragonfly

TABLE I: Machine specifications

scatter affinity on Cori-KNL and compact affinity on Cori-Haswell and for compute threads we use scatter affinity on both architectures due to their better performance. We ran all the experiments multiple times to make sure that the performance results are consistent and are not effected by run-to-run variance.

B. Load Balancing Strategies

It is important for the asynchronous execution to have enough grids to compute and balance load for communication overlap. We compare two load balancing strategies supported in both synchronous and asynchronous versions of AMReX: Space Filling Curve (SFC) and Round Robin (RR), where SFC is the default option in AMReX.

Fig. 8 shows comparison of SFC and RR in terms of max and mean execution times for the Rayleigh-Taylor test problem. When the workload is well balanced, the maximum execution time of a process is close to the mean execution time of all the processes. The drop in speedup at 69632(1024) data point in Fig. 8 for AMReX-RR is because we have fewer grids than processes at level 1 and AMReX increases the number of grids by reducing the grid sizes to balance the load. For the last data point AMReX further increases the number of grids at level 1 thus again changing the load distribution.

The performance gap between the maximum and the mean time per process for the asynchronous runtime is higher as seen in Fig. 8. The gap for AMReX is narrower, in fact barely visible from the figure. This is because the AMReX implements the rank synchronous execution model that avoids global synchronization but still suffers from loose synchronization due to bulk synchronous communication that occurs multiple times at each level. Thus difference between the mean and maximum time is negligible for AMReX. The asynchronous execution on the other hand accumulates the time difference that arises due to the load imbalance during

	Cori-Haswell	Cori-KNL
Mesh size for R-T (level 0)	$1024 \times 1024 \times 2048$	
Mesh size for S-T (level 0)	$1024 \times 1024 \times 1024$	
Fab size	64^3	
Tile size	$64 \times 8 \times 2$	$64 \times 4 \times 2$
MPI processes per node	2	4
OpenMP threads per MPI process	32	68

TABLE II: Input configurations

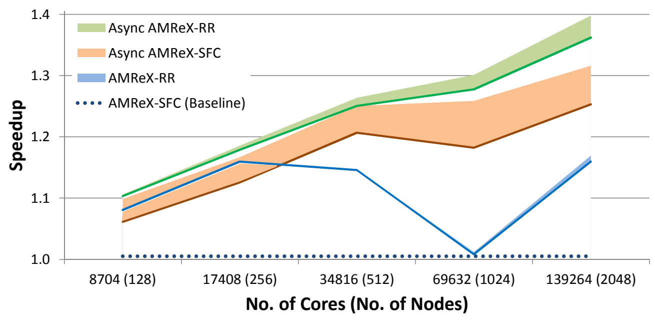


Fig. 8: SFC vs RR comparison in a strong scaling study on Cori-KNL using AMReX-SFC as the baseline for Rayleigh-Taylor problem. Lower and upper bounds indicates speedup corresponding to the maximum execution time and to the mean execution time of all MPI processes, respectively.

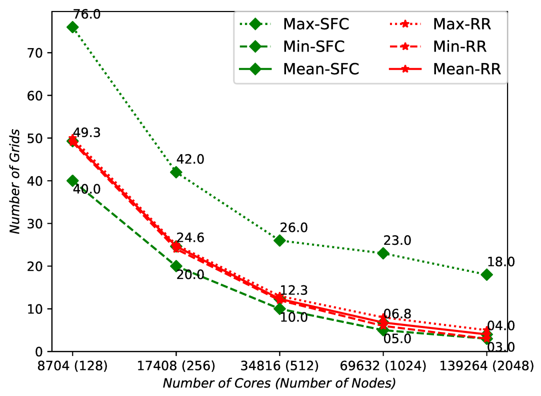


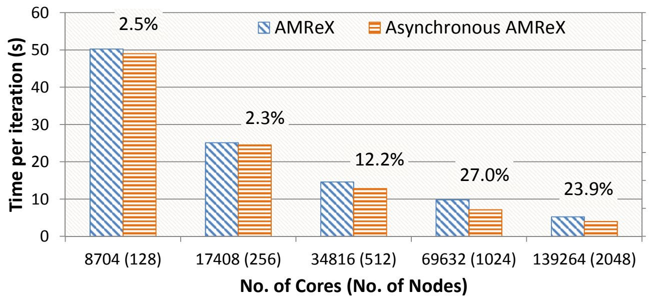
Fig. 9: Comparing SFC vs RR in terms of number of grids (Fabs) per process (Rayleigh-Taylor)

computation of all levels in an entire timestep. This results in an observable difference between the mean and maximum time of MPI processes, highlighting the importance of load balancing for asynchronous execution.

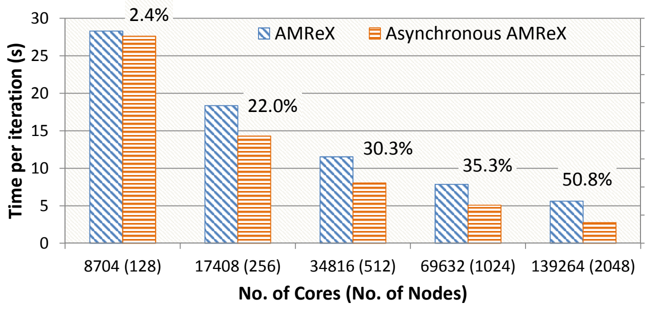
For our test problem, RR performs better than SFC for both synchronous and asynchronous code variants. This is because RR does a better job compared to SFC in load distribution in terms of number of grids (*Fabs*) per process as shown in Fig. 9. Not shown here but similar trend holds true for number of cells per process. The number of grids (*Fabs*) are more important for asynchronous execution model because computation of one grid overlaps communication of other grids. Thus higher the number of grids per process the more is the opportunity to overlap. Moreover, we use 4 MPI processes per node thus the locality benefit of SFC is low. All the results in this paper use RR strategy both for AMReX and asynchronous AMReX and report the maximum execution time.

C. Strong Scaling Studies

Fig. 10a and Fig. 10b show strong scaling results for R-T and S-T test problems on Cori-KNL while Fig. 11 shows strong scaling results for the R-T test problems on Cori-Haswell. Asynchronous AMReX outperforms the original



(a) Strong scaling on Cori-KNL (Rayleigh-Taylor)



(b) Strong scaling on Cori-KNL (Sedov-Taylor)

Fig. 10: Strong scaling on Cori-KNL for test problems (a) Rayleigh-Taylor and (b) Sedov-Taylor. The percentages above each bar correspond to the performance improvement in Asynchronous AMReX over Synchronous AMReX.

AMReX code variant on both architectures even though it uses fewer cores for computation than AMReX (30 cores on a Cori-Haswell node and 64 cores on a Cori-KNL node). It can be easily seen that, at small scale, the performance improvement is modest because the communication overhead is relatively low. However, the performance improvement quickly increases as the number of nodes grows. This result makes sense since the communication overlap can amortize the cost of task scheduling. For the S-T test problem, we use one more refinement level as compared to R-T. This improves the performance because the additional refinement level provides more opportunity to overlap communication with computation.

The reason for lower performance improvement on Cori-Haswell compared to Cori-KNL is that we are using half of the MPI processes for same number of nodes on Cori-Haswell as compared to Cori-KNL. This changes the computation to communication ratio and reduces the opportunity for computation and communication overlap.

D. Performance at Large Scale

We next carry out performance studies at large scale using all the KNL nodes that can be allocated to external users, occupying half of the Cori-KNL supercomputer. We choose not to refer to this study as weak scaling because one can increase problem size at level *0* but increase in the number of cells at finer levels is handled by AMReX based on the problem characteristics and input settings. Thus the workload per core at finer level would decrease if we increase the number of cores and the number of cells at finer level does not change. We could not perform this study on the Cori-Haswell because at most *1024* nodes can be used by a user and we already strong scale up to *1024* nodes.

We start with the data point '*69632(1024)*' for R-T and data point '*34816(512)*' for S-T in the previous strong scaling study on Cori-KNL and double the mesh size at level *0* as we double the number of cores. Fig. 12 shows results for both R-T and S-T test problems. Asynchronous AMReX maintains significant performance improvement over AMReX. For RT, it can be seen that the performance gap slightly reduces as the number of node increases. This is due to the fact that the amount of computation at fine levels does not double proportionally when we double the mesh size at level *0*. Thus the load distribution changes at each data points and results in different amount of performance improvements. However, overall we observe good scaling at large number of cores.

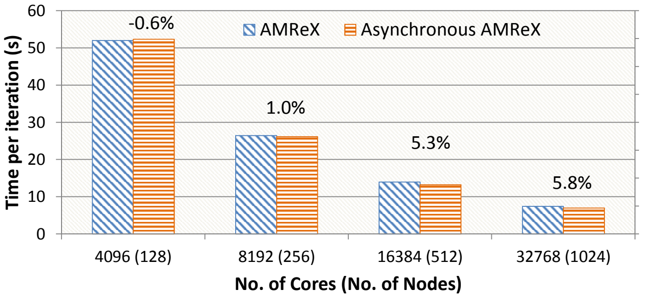


Fig. 11: Strong scaling on Cori-Haswell (Rayleigh-Taylor).

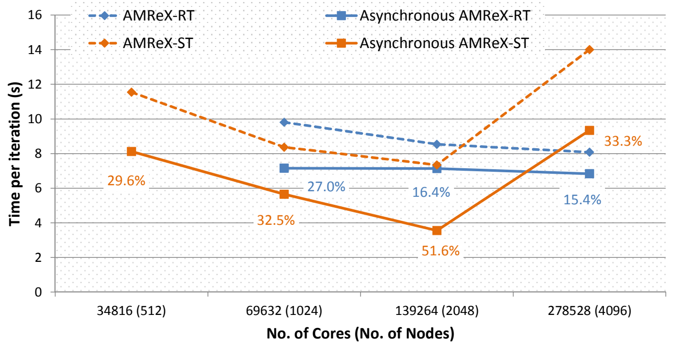


Fig. 12: Performance at large scale on Cori-KNL for the test problems R-T in blue and S-T in orange color.

E. Hyperthreading

As discussed in Section 5, in addition to using hyperthreads for compute, our runtime can use hyperthreads for handling communication. We found that using hyperthreading for computation contributes about 12% performance improvement as compared to using a single thread per core on Cori-KNL as shown in Fig. 13. Hyperthreading for communication contributes to overall 4% performance improvement in

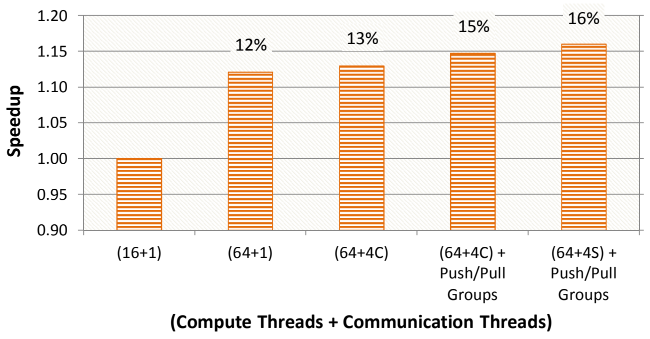


Fig. 13: Different degrees of hyperthreading support in asynchronous AMReX (1024 Cori-KNL nodes) using configurations in Table II for the R-T test problem. *C* and *S* in the x-axis labels represent Compact and Scatter affinity.

total. The contributions in details are as follows. Dividing communication work among multiple threads gives about 1%, while pushing or pulling local and remote communication data in parallel using two groups of threads gives additional 2% improvement. Lastly, using scatter affinity for communication threads results in another 1% improvement.

The performance improvement due to hyperthreading for communication is modest as expected because communication thread is responsible for light-weight operations like posting MPI sends and receives, waiting for the completion of MPI communication and doing local memory copies.

VII. Related Work

Asynchronous communication primitives are supported in de facto standard communication libraries such as MPI and UPC. These primitives enable overlapping communication with computation, an effective performance optimization that has been used for years. Indeed, many studies [16]–[21] highlighted the benefits of overlap on various compute classes. Despite the fact that overlap provides significant performance improvements, this technique requires code and data structure transformations, which are labor intensive, prone to errors and result in less readable code [22]–[24]. To reduce programming cost further, there are compiler based tools [25], [26] that translate MPI codes to a task-based form that can realize overlap. Their success has been demonstrated on small problems but their effectiveness is unclear on large legacy codes.

There are a number of task based programming models and runtime systems such as Legion [27], Charm [28], DaGue [29] that can provide automatic overlap of communication and computation. However, programming effort to implement practical applications using these general-purpose task models is normally too high, especially in the context of adaptive mesh refinement. To overcome this challenge, higher level abstraction is required. For example, Legion introduces Regent [30], a high level programming language that allows serial-like programs to be transformed into Legion codes. Although this technique is feasible, the required runtime analysis due to implicit synchronization in Regent may cause some slow down and it is not straightforward to covert legacy codes into Regent.

A number of AMR frameworks [31]–[33] exist which support parallel computation in AMR applications. Only few [34], [35] support communication and computation overlap. Langer et al. [35] organize the oct-tree based adaptive mesh as a collection of parallel objects distributed over virtual processors and uses the Charm++ runtime system, where parallel objects are represented using chare arrays. To support fully asynchronous execution, they use their own parallel mesh restructuring algorithm where subgrids can be refined and coarsened without synchronization. Our approach works with the existing mesh structuring algorithms and it does not depend on any specific threading library. Uintah [34] is a software framework that implements a task-based runtime to support asynchronous execution of AMR applications. They also represent a subgrid as a task where computation of a task overlaps communication of other tasks. Porting legacy AMR applications to their framework requires major rewriting of the applications in Uintah's task-based programming model.

We introduce an user-friendly API, especially the grid iterator, that significantly reduces programming effort. We eliminate the need to discover necessary communication at runtime by leveraging communication metadata embedded in an AMR program. We also make use of domain-specific knowledge for optimizing internal components of our runtime.

VIII. Conclusions

This paper presents the great potential of a phase asynchronous execution model for adaptive mesh refinement applications. As a case study, we use a large real world astrophysics application CASTRO and unveil a simplified application programming interface carefully designed in collaboration with domain experts. We discussed the implementation strategy, analyzed the programming effort required to port the CASTRO code, and performed scaling studies on two different machine architectures using up to 278,528 cores. We used Rayleigh-Taylor and Sedov-Taylor as test problems, which contains frequently used communication types found in most of the AMR applications. The simplified API enabled us to port such a large real world application just by modifying fewer than 200 lines of code while achieving a significant amount of performance improvement. Our runtime also utilizes all the hardware threads to achieve maximum parallelism and asynchrony. Lastly, we studied two load balancing strategies and found out that phase asynchronous execution requires an efficient load balancing strategy to achieve the maximum possible performance.

Acknowledgments

Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 215E185. Authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

References

[1] "AMReX: Block-structured AMR framework." [Online]. Available: https://ccse.lbl.gov/AMReX/index.html

[2] A. S. Almgren, V. E. Beckner, J. B. Bell, M. Day, L. H. Howell, C. C. Joggerst, M. J. Lijewski, A. Nonaka, M. Singer, and M. Zingale, "CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity," *Astrophysical Journal*, vol. 715, pp. 1221–1238, Jun. 2010.

[3] J. B. Bell, M. Day, and M. Lijewski, "Simulation of nitrogen emissions in a premixed hydrogen flame stabilized on a low swirl burner," *Proc. Comb. Inst.*, vol. 34, no. 1, pp. 1173–1182, 2013.

[4] M. Emmett, W. Zhang, and J. B. Bell, "High-order algorithms for compressible reacting flow with complex chemistry," *Combustion Theory and Modelling*, vol. 18, no. 3, pp. 361–387, 2014.

[5] A. Almgren, J. B. Bell, M. Lijewski, Z. Lukic, and E. V. Andel, "Nyx: A massively parallel amr code for computational cosmology," *APJ*, vol. 765, p. 39, 2013.

[6] C. P. Chan, J. D. Bachan, J. P. Kenny, J. J. Wilke, V. E. Beckner, A. S. Almgren, and J. B. Bell, "Topology-aware performance optimization and modeling of adaptive mesh refinement codes for exascale," in *Proceedings of the First Workshop on Optimization of Communication in HPC*, ser. COM-HPC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 17–28.

[7] M. N. Farooqi, D. Unat, T. Nguyen, W. Zhang, A. S. Almgren, and J. Shalf, "Nonintrusive AMR asynchrony for communication optimization," in *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, 2017, pp. 682–694.

[8] W. Zhang, L. H. Howell, A. S. Almgren, A. Burrows, and J. B. Bell, "Castro: A new compressible astrophysical solver. ii. gray radiation hydrodynamics," *Astrophysical Journal Supplement*, vol. 196, p. 20, 2011.

[9] W. Zhang, L. H. Howell, A. S. Almgren, A. Burrows, J. C. Dolence, and J. B. Bell, "Castro: A new compressible astrophysical solver. iii. multigroup radiation hydrodynamics," *Astrophysical Journal Supplement*, vol. 204, p. 7, 2013.

[10] M. P. Katz, M. Zingale, A. C. Calder, F. D. Swesty, A. S. Almgren, and W. Zhang, "White dwarf mergers on adaptive meshes. i. methodology and code verification," *The Astrophysical Journal*, vol. 819, no. 2, p. 94, 2016.

[11] A. Burrows, J. Nordhaus, A. Almgren, and J. Bell, "The potential role of spatial dimension in the neutrino-driving mechanism of core-collapse supernova explosions," *Computer Physics Communications*, vol. 182, no. 9, pp. 1764 – 1766, 2011, computer Physics Communications Special Edition for Conference on Computational Physics Trondheim, Norway, June 23-26, 2010.

[12] K.-J. Chen, S. Woosley, A. Heger, A. Almgren, and D. J. Whalen, "Two-dimensional simulations of pulsational pair-instability supernovae," *The Astrophysical Journal*, vol. 792, no. 1, p. 28, 2014.

[13] T. Nguyen, D. Unat, W. Zhang, A. Almgren, N. Farooqi, and J. Shalf, "Perilla: Metadata-based optimizations of an asynchronous runtime for adaptive mesh refinement," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 81:1–81:12.

[14] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. S. Almgren, and J. Shalf, "Tida: High-level programming abstractions for data locality management," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 116–135.

[15] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S156–S172, 2016.

[16] B. F. Jamroz and R. Klöfkorn, "Asynchronous communication in spectral-element and discontinuous Galerkin methods for atmospheric dynamics- a case study using the High-Order Methods Modeling Environment ," *Geoscientific Model Development*, vol. 9, no. 8, pp. 2881–2892, aug 2016.

[17] J. C. Sancho and D. J. Kerbyson, "Improving the performance of multiple conjugate gradient solvers by exploiting overlap," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5168 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 688–697.

[18] A. Danalis, K. Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05*, vol. 2005. IEEE, 2005, pp. 58–58.

[19] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 84–84.

[20] T. Viet and T. Yoshinaga, "Improving Linpack Performance on SMP Clusters with Asynchronous MPI Programming," *IPSJ Digital Courier*, vol. 2, pp. 598–606, 2006.

[21] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.

[22] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a Conjugate Gradient Solver with Non Blocking Operators," *Parallel Computing*, vol. 33, no. 9, pp. 374–382, 2007, selected Papers from EuroPVM/MPI 2006.

[23] T. Hoefler, J. M. Squyres, W. Rehm, and A. Lumsdaine, "A case for non-blocking collective operations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4331 LNCS. Springer, Berlin, Heidelberg, 2006, pp. 155–164.

[24] E. Chan, R. van de Geijn, and A. Chapman, "Managing the complexity of lookahead for lu factorization with pivoting," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 200–208.

[25] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. Baden, "Automatic translation of mpi source into a latency-tolerant, data-driven form," *J. Parallel Distrib. Comput.*, vol. 106, no. C, pp. 1–13, Aug. 2017.

[26] S. M. Martin, M. J. Berger, and S. B. Baden, "Toucan A Translator for Communication Tolerant MPI Applications," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2017, pp. 998–1007.

[27] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE, Nov 2012, pp. 1–11.

[28] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108.

[29] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[30] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for hpc with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 81:1–81:12.

[31] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, no. 3, pp. 330 – 354, 2000.

[32] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The cactus framework and toolkit: Design and applications," in *Proceedings of the 5th Intl Conference on High Performance Computing for Computational Science*, ser. VECPAR'02. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 197–227.

[33] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. V. Straalen, "Chombo software package for AMR applications design document," LBNL, Tech. Rep., 2003.

[34] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah Framework: a unified heterogeneous task scheduling and runtime system," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 2441–2448.

[35] A. Langer, J. Lifflander, P. Miller, K. C. Pan, L. V. Kale, and P. Ricker, "Scalable algorithms for distributed-memory adaptive mesh refinement," in *Proceedings - Symposium on Computer Architecture and High Performance Computing*. IEEE, Oct 2012, pp. 100–107.

Appendix A
Appendix: Phase Asynchronous AMR Execution for Productive and Performant Astrophysical Flows

A. AMReX Framework

A detailed explanation of how to develop an AMR application using AMReX is available in its user guide. However, we provide a basic overview here. AMReX, developed as a collection of C++ classes, provides support functions such as memory management, adaptive mesh structure management and communication. Typical AMReX applications are developed using C++ and Fortran where C++ is used for control structure and computational kernels are implemented in Fortran. Implementing an AMR application using AMReX requires understanding of at least the following basic classes.

- *FArrayBox (Fab)*: The *Fab* class defines a rectangular box allocated as a Fortran array and provides a set of support functions to be performed over that box. A subgrid at an AMR level is represented with a *Fab* class object.
- *MultiFab*: The *MultiFab* class contains an array of *Fab* objects along with a set of support functions. This class also defines the function that identifies and performs communication among adjacent *Fabs* in a *MultiFab* or between two *MultiFabs*. A *MultiFab* is used to represent all the subgrids at an AMR level.
- *MFIter*: It creates an iterator for a *MultiFab* to iterate over all its member *Fabs*. Optionally, tiling can be enabled to create tiles of desired dimension within all *Fabs*.
- *AmrLevel*: It is an abstract class that implements functionality related to an AMR level. It defines a few pure virtual functions that should be implemented by the application programmer. These virtual functions include functionalities such as computing dt value, data initialization and main computation. A notable virtual function *advance* is used for defining the main computation for the corresponding level. The *post_timestep* virtual function is used for *Restriction* from fine to coarser levels. The *AmrLevel* class also defines *FillPatch* for inter-level communication that interpolates and communicates data from coarse to finer levels.
- *Amr*: It implements the basic AMR execution algorithm discussed in [7]. Applications are required to create an object of the *Amr* class in the main method and iterate over the *coarseTimeStep* method for desired number of times. The *coarseTimeStep* method advances the coarsest level by a single timestep along with finer levels that are advanced to match the coarsest level.

B. Example Application Implementation

Fig. 14 shows the basic code structure of a typical application implemented with AMReX. The application implements a class (i.e. *AmrTest*) derived from the *AmrLevel* class. This application class implements a few pure virtual functions such as *advance* and *post_timestep*. The *advance* method contains the computation and communication at each AMR level. The *post_timestep* method performs the *Restriction* operation for the desired *MultiFabs*. AMReX provides a builtin function named *average_down* that implements a commonly used algorithm for *Restriction*. The *main* method of the application is straightforward, where its creates an object of the class *Amr* and then iterates over the *coarseTimeStep* method of that object until the termination condition is reached.

main.cpp:

```
int main (int argc, char* argv[]) {
  amrex::Initialize(argc,argv);
  Amr amr;
  amr.init(strt_time,stop_time);
  while(amr.okToContinue()){
    amr.coarseTimeStep(stop_time);}
  amrex::Finalize();
}
```

AmrTest.h:

```
class AmrTest : public amrex::AmrLevel {
...
virtual double advance (double time, double
 dt, int iteration, int ncycle) override;

virtual void post_timestep (int iteration)
 override;

MultiFab* testMF;
...
}
```

AmrTest.cpp:

```
...
double AmrTest::advance (double time, double
 dt, int iteration, int ncycle){
...
ExchangeGhost(testMF, ...);
for (MFIter mfi(*testMF, true); mfi.isValid();
 ++mfi)
{   compute(...) }
...
}

void AmrTest::post_timestep (int iteration){
...
AmrTest& fineLvl = getLevel(level+1);
average_down(*(fineLvl.testMF), *testMF,...);
}
...
```

Fig. 14: Basic structure of an AMR application using AMReX

Porting an application to the phase asynchronous execution model requires implementation of two additional virtual functions, namely *initMetaData* and *deleteMetaData* defined in the *AmrLevel* class. In addition, the programmer needs to change *MFIter* loops in *advance* and *post_timestep* to *TGIter* loops, and replace the communication subroutines with their respective send and receive asynchronous alternatives. Figure 15 shows the *AmrTest.cpp* file of the application implementation using asynchronous AMReX. Here we do not show *main.cpp* and *AmrTest.h* because *main.cpp* is the same as shown in Fig. 14 while the *AmrTest.h* file just needs to include

AmrTest.cpp:

```
...
double AmrTest::advance (double time, double
 dt, int iteration, int ncycle){
...
for (TGIter tgi(tg); tgi.isValid(); ++tgi)
{ ExchangeGhost_receive(tgi, testMF, ...);
  compute(...);
  ExchangeGhost_send(tgi, testMF, ...);
}
...
}

void AmrTest::post_timestep (int iteration){
...
AmrTest& fineLvl = getLevel(level+1);
for (TGIter tgi(tg); tgi.isValid(); ++tgi)
{   average_down_receive(tgi,
 *(fineLvl.testMF), *testMF,...);
}
AmrTest& coarseLvl = getLevel(level-1);
for (TGIter tgi(coarseLvl.tg, true);
 tgi.isValid(); ++tgi)
{   average_down_send(tgi,
 *(coarseLvl.testMF), *testMF,...);
}
}
...
```

Fig. 15: Basic structure of an AMR application using Asynchronous AMReX

declaration of the *initMetaData*, *deleteMetaData* functions and task graph variables.

In the body of the *coarseTimeStep* function, *initMetaData* and *deleteMetaData* are called upon each regrid operation for managing metadata extraction. *deleteMetaData* is called to destroy the existing task graphs and *initMetaData* is then called to create new task graphs that reflects the new grid structure. A thread pool is then created and divided into communication and worker threads. The thread pool lasts till completion of the computation for the current time step. Communication threads are dedicated to the runtime that handles all communication while worker threads handle computation step by step for all refinement levels. In this example, each refinement level is represented by an object of the *AmrTest* class that is inherited from the *AmrLevel* class.

In the asynchronous AMReX code variant, the communication subroutines like *ExchangeGhost*, *FillPatch* and *average_down* are split into three components. *send* and *receive* are two lightweight components. *send* notifies the runtime that the communication data have been produced by the task, while *receive* takes the communication data from the runtime and injects them into the task data. The third component, which is the actual heavyweight communication is performed by the runtime and can be overlapped with computation. The task graph iterator schedules tasks without any specific order as long as their dependent communication data is received.

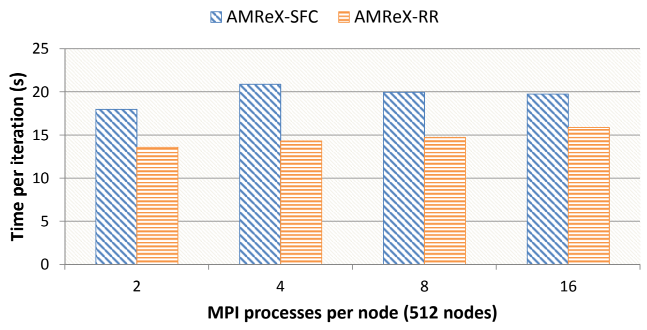
C. CASTRO Implementation

CASTRO is implemented similar to any other applications implemented using AMReX. An application class named *Castro* is created that is inherited from the *AmrLevel* class. The class contains declaration of the required data. All the computation and communication explained in the paper is gathered into the *advance* method of this class.

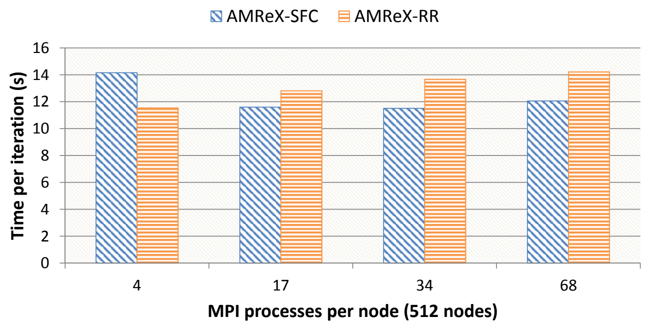
D. Results

1) Test Problems: In this paper, we use the Rayleigh-Taylor and Sedov-Taylor test problems in CASTRO.

- Rayleigh-Taylor instability is a ubiquitous phenomenon in nature. High-fidelity simulation of Rayleigh-Taylor instability is a key to the understanding of the dynamics and nucleosynthesis of supernova explosions.
- The Sedov-Taylor solution is a self-similar solution of the evolution of a blast wave from a powerful explosion that quickly releases a huge amount of energy in a small volume (e.g., supernova explosions in astrophysics). This is a classical verification test problem for hydrodynamics codes because of the existence of an analytic solution for comparison, and it is also a common test problem for AMR codes because most of the energy is concentrated in a moving thin shell requiring high resolution.



(a) Cori-Haswell



(b) Cori-KNL

Fig. 16: Comparison of different number of MPI processes per node on (a) Cori-Haswell using Rayleigh-Taylor and (b) Cori-KNL using Sedov-Taylor.

2) Number of MPI Processes per Node: To select optimal number of MPI processes per node for AMReX, we ran multiple configurations on both Cori-Haswell and Cori-KNL using two different load balancing strategies Space Filling Curve (SFC) and Round Robin (RR). Results are shown in Fig. 16(a) Cori-Haswell and Fig. 16(b) Cori-KNL. Cori-Haswell has 2 NUMA nodes and we use Cori-KNL in quadrant cluster mode resulting in 4 NUMA nodes. Creating more than one MPI process per node helps to use the network bandwidth effectively and provides NUMA-aware data locality. We found that using a single MPI process per NUMA node along with RR load balancing strategy produces the best results.