Nonintrusive AMR Asynchrony for Communication Optimization

Muhammad Nufail Farooqi[1(✉)], Didem Unat[1(✉)], Tan Nguyen[2], Weiqun Zhang[2], Ann Almgren[2], and John Shalf[2]

[1] Koç University, Istanbul, Turkey
{mfarooqi14,dunat}@ku.edu.tr
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA
{tannguyen,weiqunzhang,asalmgren,jshalf}@lbl.gov

Abstract. Adaptive Mesh Refinement (AMR) is a well known method for efficiently solving partial differential equations. A straightforward AMR algorithm typically exhibits many synchronization points even during a single time step, where costly communication often degrades the performance. This problem will be even more pronounced on future supercomputers containing billion way parallelism, which will raise the communication cost further. Re-designing AMR algorithms to avoid synchronization is not a viable solution due to the large code size and complex control structures. We present a nonintrusive asynchronous approach to hiding the effects of communication in an AMR application. Specifically, our approach reasons about data dependencies automatically using domain knowledge about AMR applications, allowing asynchrony to be discovered with only a modest amount of code modification. Using this approach, we optimize the synchronous AMR algorithm in the BoxLib software framework without severely affecting the productivity of the application programmer. We observe around 27–31% performance improvement for an advection solver on the Hazel Hen supercomputer using 12288 cores.

Keywords: Asynchronous execution · Adaptive mesh refinement · AMR algorithm · Communication hiding

1 Introduction

Many computational science and engineering problems are modelled in the form of partial differential equations (PDEs). Although a high resolution mesh is required for improved accuracy of PDE solvers, usually some mesh regions are of more interest, where additional accuracy is desired. Adaptive mesh refinement (AMR) provides the mechanism to locally refine areas of interest [8]. Block-structured AMR (SAMR) is a type of AMR method that uses structured grids organized into a grid hierarchy. Areas of interest are refined gradually in a nested manner from the coarsest level, which covers the whole domain to the finest.

© Springer International Publishing AG 2017
F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 682–694, 2017.
DOI: 10.1007/978-3-319-64203-1_49

One of the scalability challenges for AMR applications is that they consist of many synchronization points. These costly synchronization points appear in the nearest-neighbor communication including boundary exchange, in the global reduction, and in the inter-AMR level update. The former has become increasingly costly due to the system design trend focusing on fewer but more powerful compute nodes [6]. Asynchronous execution can reduce synchronization cost with the help of description of dependencies between AMR subgrids and the partial ordering among them. Given the partial ordering information, a scheduler can assign ready subgrids on available resources while other subgrids are waiting on their inputs.

In this paper we propose an asynchronous AMR algorithm that reduces the most of the synchronization costs without bringing too much programming overhead. In our asynchronous algorithm, each subgrid at different AMR levels is considered as a task. A task within a specific level can perform computation independent of other tasks at the same level as soon as its boundary data is available. Even though there is more opportunity for overlap in an AMR algorithm, for example, a subgrid located at *any* level can perform computation independent of other subgrids, we enforce the completion of computation of subgrids in a single level before moving onto the computation at other levels for the sake of programming simplicity. Our method enables legacy application implemented using the synchronous AMR algorithm to get the benefits of the communication and computation overlap. We discuss the implementation of our asynchronous algorithm in the context of the BoxLib AMR framework and present results on the advection solver, which contains all the communication scenarios present in a typical AMR application. We compared our results with the existing BoxLib execution model, where communication at each level is completed before starting computation. The performance improvement is about 27% for both strong and weak scaling on 12288 cores.

Rest of the paper is organized as follows. Next section discusses related work. In Sect. 3 we provide some background on Block-Structured AMR. Section 4 explains the AMR algorithm in general and Sect. 5 proposes a methodology to asynchronously execute the AMR algorithm. Implementation is discussed in Sect. 6. Results are shown in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Related Work

A plethora of work can be found in literature that focuses on speeding up of AMR computations using diverse techniques while targeting specific problems or architectures. Some of the high level AMR frameworks are BoxLib [1], Cactus [12], Chombo [10], Enzo [2], FLASH [11], and Paramesh [15]. Wahib et al. [20] presented a compiler-based framework named *Daino* that generates parallel AMR code optimized for GPUs from an annotated uniform grid code. In [19], authors introduced an asynchronous integration scheme with local time stepping for multi-scale gas discharge simulations.

Chan et al. [9] classified AMR execution models into four modes ranging from *fully synchronous* to *fully asynchronous*. The trade-off between the modes is the

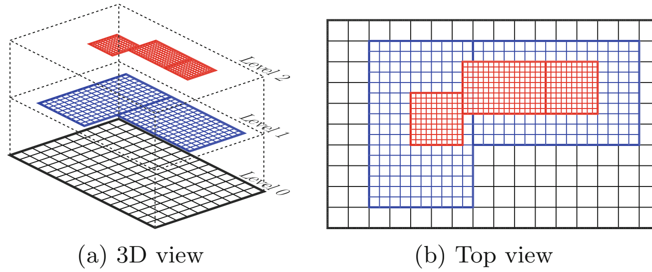
amount of synchronization and the programmability. The more asynchronous the execution becomes, the harder it is to program and debug. *Full synchronous* is the most restricted one, which will be discussed in Algorithms 1 and 3 in Sect. 4. *Rank synchronous* reduces the global synchronization down to rank level and runs synchronously within a rank. *Rank synchronous* model avoids global synchronization but enforces local restrictions on task processing order. BoxLib currently implements a *rank synchronous* model. In *phase asynchronous*, a subgrid within a specific level can perform computation independent of other subgrids at the same level as soon as dependencies are met and communication for a subgrid is overlapped within a single time step. Each rank will finish its communication for all the subgrids before starting computation on any subgrid. In a *fully asynchronous* model, a subgrid located at any level can perform computation independent of other subgrids as soon as its own dependencies are met. Here we present an asynchronous AMR algorithm that is analogous to the *phase asynchronous* execution model.

To the best of our knowledge, the literature that explains the asynchronous AMR algorithm and its corresponding implementation is rare. A few notable contributions are as follows. Langer et al. [14] proposed a distributed regridding algorithm to enable *fully asynchronous* AMR execution for oct-tree based AMR implementations. They used Charm++ [13] for implementation where each subgrid is represented by a *chare* that can run independently and communication of one *chare* is overlapped with computation of another. Our proposed asynchronous algorithm can work with traditional regridding algorithms and can be implemented using any threading library. Uintah [16] is a software framework that implements a runtime to execute AMR applications asynchronously. They also use subgrid level asynchronous task execution to overlap communication and computation. They mostly discussed the runtime optimization details but do not explain the asynchronous AMR algorithm.

3 Block-Structured Adaptive Mesh Refinement (SAMR)

AMR provides a computationally efficient approach for solving PDEs by using finer meshes only at regions of interest. SAMR [8], one of the many AMR methods, is established on a chain of nested and logically rectangular grids. Starting from a coarse grid that covers the entire domain at level 0, grids are refined to finer grids at the higher levels with the finest grid at the top level. Figures 1a and b show sample SAMR grids having two levels of refinements. Each level is composed of non-overlapping rectangular grids nested from grids on the lower level in the hierarchy. The nested grid at a finer level is extended from a single grid or multiple adjacent grids at the coarser level. All grids at a level are of the same resolution. Given maximum number of levels at start, the number of refinement levels can vary dynamically during the simulation.

Generally, two types of communication are involved in the parallel AMR implementations: (1) intra-level communication is only between neighboring/adjacent grids, and (2) inter-level communication is only between consecutive levels. Two basic operations, *restriction* and *prolongation*, are needed for



(a) 3D view (b) Top view

Fig. 1. Block-structured AMR in 2 dimensions with two levels of refinement

inter-level communication. In prolongation, data is interpolated when communicated from a coarser grid to a finer one. In restriction, data is averaged when copied from a finer to coarser grid.

4 Synchronous AMR Algorithm

Algorithms 1 and 2 show the basic AMR algorithm described in [18]. Algorithm 1 contains a time step loop, which runs a specified number of times. In each iteration it first finds the time step dt for the current time step. Computing dt generally involves a global reduction operation to find a minimum value. Next a recursive procedure *AMRTimeStep* is called that starts from the coarsest level and iterates over all levels to compute a single time step.

Algorithm 2 shows the recursive procedure that computes a single time step of the AMR algorithm. The procedure first checks whether regridding the finer level is needed. If needed, it estimates the error at finer level $(l+1)$ and regrids the finer level. When a regrid operation is performed on a finer level, it will subsequently be carried out for all the upper levels up to the finest level. Boundary data is filled from current refinement level l if available otherwise data is filled from physical boundary conditions or interpolated from the coarser level $l - 1$. Upon receiving of all the boundary data, all the grids at the current level l are integrated in time. Next, the *AMRTimeStep* procedure is called r times recursively to compute the finer level at smaller time steps. This is known as subcycling in time where r specifies the desired number of cycles that is normally set to the refinement ratio. The value of r can be set to 1 if no-subcycling is desired. Data between the current level and the finer level is synchronized after the finer level reaches the same time t as the current level. All the levels are integrated independent of each other. Lastly, data is synchronized between two successive levels to resolve the inconsistencies at coarse and fine level boundaries.

In the synchronous execution of an AMR algorithm there are multiple synchronization points. First synchronization point is in the computation of time step value dt where a global reduction operation occurs. Next synchronization point is when boundary data is filled and this synchronization happens every time

Algorithm 1 Basic AMR algorithm

```
Procedure AMRTimeLoop(time t, num of steps s)
for i ← 1 to s do
    dt ← compute_dt()
    AMRTimeStep(0, t, dt)
    t ← t + dt
end for
Procedure AMRTimeLoop
```

Algorithm 2 Single Time Step

```
Procedure AMRTimeStep(level l, time t, dt)
if isRegrid(l + 1) then
    estimateError(l + 1)
    generateGrids(l + 1)
end if
if l = 0 then
    fillBoundary(level 0 ← level 0)
else
    fillBoundary(l ← l and l - 1)
end if
for each grid g in grids at level l do
    integrate(l, t + dt, g)
end for
if l < l_max then
    repeat j ← 1 to r times:
      AMRTimeStep (l + 1, t, (j×dt)/r)
end if
synchronizeData(l ← l + 1)
End Procedure AMRTimeStep
```

the *AMRTimeStep* procedure is called. Last synchronization is when data is synchronized between two adjacent levels to correct coarse and fine level boundaries. Next, we discuss our proposed asynchronous algorithm that overcomes some of these synchronization overheads.

5 Asynchronous AMR Algorithm

In the AMR algorithm listed in Algorithm 2 data needed for all grids at a level is communicated before starting computation on that level. Thus all the grids at the same level are computed when all of their dependencies are fulfilled. In the synchronous algorithm, all grids at the same level are considered as one big task that is carried out as a whole. For an asynchronous execution, we reduce the task granularity to subgrid size where each subgrid is considered as a task. A task can start computing as soon as its dependencies are fulfilled. Here, dependencies for a task are the data at boundaries that are copied from other tasks.

The asynchronous version of Algorithm 1 is the same as the synchronous except the reduction operation is performed asynchronously. Algorithm 3 shows the asynchronous AMR algorithm for a single time step. Before executing Algorithm 1, a task graph is created that contains information about tasks at all levels and their dependencies. Dependencies in the task graph are based on the grid structure therefore the task graph remains valid until there is a change in the grid structure. Asynchronous task graph is updated when a regridding occurs to

reflect the changes in the grids and their dependencies. In Algorithm 3 all the *fillboundary_send* calls are non-blocking while the receives are blocking.

Algorithm 3 Asynchronous AMR Algorithm - Single Time Step

```
Procedure AMRTimeStep(level l, time t, dt, iteration iter)
if isRegrid (l + 1) then
    estimateError(l + 1)
    generateGrids(l + 1)
    updateTaskGraph(l + 1)
end if
if FirstTimeStep and iter = 1 and l < l_max then
    if l = 0 then fillBoundary_send_allGrids (level 0 ← level 0) //non-blocking
    fillBoundary_send_allGrids (l + 1 ← l + 1) //non-blocking
end if
for each grid g in grids at level l do //Out-of-order loop iterator
    if l = 0 then
        fillBoundary_receive(level 0 ← level 0, g) //blocking
    else
        fillBoundary_receive(l ← l and l - 1, g) //blocking
    end if
    integrate(l, t + dt, g)
    if l < l_max then
        fillBoundary_send (l+1 ← l, g) //non-blocking
    else
        fillBoundary_send (l ← l, g) //non-blocking
    end if
end for
if l < l_max then
    repeat j ← 1 to r time: AMRTimeStep (l + 1, t, (j×dt)/r, j)
end if
if l < l_max then synchronizeData_receive_allGrids (l ← l + 1) //blocking
if l > 0 then synchronizeData_send_allGrids (l - 1 ← l) //non-blocking
if l < l_max then fillBoundary_send_allGrids (l ← l) //non-blocking
End Procedure AMRTimeStep
```

In the first time step, to overlap the intra-level communication at the finer level $(l+1)$ for timestep (t) with computation of the current level (l) for timestep (t), we can start sending the boundary data for the finer level because data at that level is already initialized during the initialization of the application. After initiating the intra-level communication at the finer level, a loop iterates over all grids at the current level. The loop iterator is designed to iterate over the grids for which dependencies are met and it uses the dependency graph to identify the task dependencies. This out-of-order execution enables ready grids to start computing while allowing more time for grids which are still waiting for their boundary data. Receive calls although blocking do not wait idle for communication because the loop iterator ensures that the dependencies for the

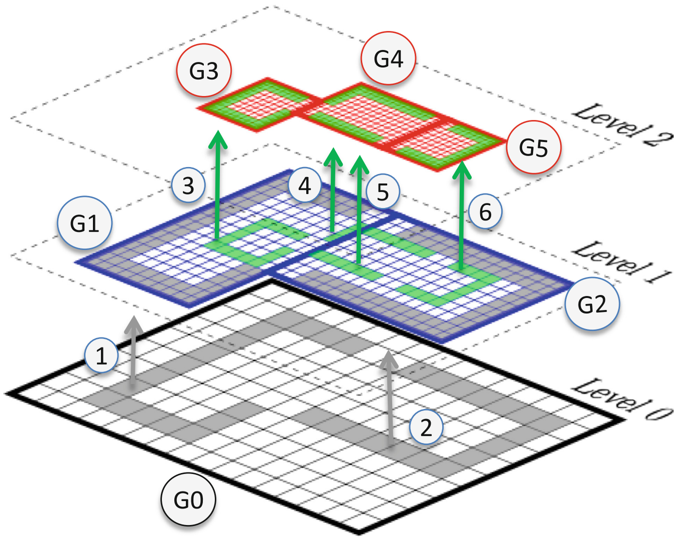


Fig. 2. Asynchronous computation and communication overlap

subgrid are already met. As the dependencies for the task are met, the grid fills the boundaries with the received data from current and coarser level $(l-1)$. After performing the computation (integrate) on the grid, the boundary data is sent to the dependent grids at finer level$(l+1)$ when current level is not the finest level. If the current level is the finest level $(l = l_{max})$ then it sends the boundary data to dependent grids at the same level for next time step $(t + dt)$ or next iteration if subcycling is enabled. Thus boundary data communication between adjacent levels or within the finest level for next subcycling iteration is overlapped with computation of the current level (l) or current subcycling iteration. Next, data at current level is synchronized with the received data from the finer level for all grids and the synchronized data is then sent to the coarser level. Lastly, for levels below the finest level we can initiate its intra-level communication for the next time step $(t + dt)$ or the next subcycling iteration. This enables to overlap intra-level boundary data communication for finer levels with the computation at next time step of their coarser levels. However, for iterations within a time step when subcycling is enabled the overlap will only be with the computation of grids at the same level. For the coarsest level (0), this can be possibly overlapped with the global reduction operation required for the next time step value.

Figure 2 shows an example how we enable overlap of computation and communication for Algorithm 3. After computation of grid $G0$ at level 0, communication for boundary data takes place as shown by arrows 1 and 2. For example, if the communication represented by arrow 1 completes first the grid $G1$ at level 1 will start computation. After $G1$ finishes computation it can start sending the boundary data (shown with arrows 3 and 4) to the grids $G3$ and $G4$ at level 2. Communication represented with arrows 3 and 4 will be overlapped with

computation of the grid $G2$ at level 1. After completion of the grid $G2$ and initiating the boundary data communication (shown with arrows 5 and 6), any grid at level 2 that receives its boundary data can start computation. That is if 3 finishes first then $G3$ can start its computation or if both 4 and 5 finish first then $G4$ can start its computation. Similarly $G5$ can start computation when 6 is finished.

6 Implementation

We implemented the asynchronous AMR algorithm in BoxLib [1], which is a publicly available software framework used for implementing Block-Structured AMR applications. Some of the large BoxLib applications are for astrophysics (CASTRO [3] and MAESTRO [7]), cosmology (Nyx [5]) and low Mach number combustion (LMC [4]) simulations. BoxLib contains two notable classes, *Amr* and *AmrLevel*, that are related to the AMR algorithm implementation. The *Amr* class implements the AMR algorithm described in Algorithms 1 and 2. *AmrLevel* manages data and operations required on them for a single level. *AmrLevel* contains some virtual functions that the application programmers override to implement their solver. These virtual functions are called for each level inside the *Amr* class's function that implements the AMR algorithm. Two of these virtual functions are *advance* and *post_timestep*. The *advance* subroutine should implement the fill boundary data and integration part of the AMR algorithm. Data management and MPI communication is handled by BoxLib as it provides *fillPatch* subroutine that manages the fill boundary data and the programmer can use it in the *advance* subroutine to fill the boundary data. Programmer overrides the *post_timestep* subroutine to synchronize data between the levels. Data synchronization between the levels also known as restriction can be performed using the *average_down* subroutine provided by BoxLib.

To implement the asynchronous execution of the AMR algorithm, we extended some of the BoxLib functionalities. We added two more virtual functions *initAsynchronousExec* and *finalizeAsynchronousExec* to the *AmrLevel* class so that applications can override them to initialize and destroy asynchronous task graphs for a level. Task graphs from all levels are combined together inside BoxLib to construct dependencies for the entire AMR grid hierarchy. *FillPatch* and *average_down* previously implement synchronous MPI communication for all grids at a level. To enable communication for a single grid without waiting for the other grids, we divided the execution of *FillPatch* and *average_down* into two parts; *push* and *pull*. *FillPatch_push* starts sending boundary data from a single grid to all dependent grids whether at current level or at the finer level. *FillPatch_pull* receives the boundary data for a single grid from all the relevant grids. To pick the ready tasks, we implemented an iterator that iterates over all the tasks in the asynchronous task graph. Our scheduler similar to the runtime scheduler in [17], backs the iterator to support out-of-order execution. The scheduler keeps track of the ready tasks and handles all the communications generated by the asynchronous *fillPatch* and *average_down* subroutines.

Both new applications and legacy applications developed using BoxLib can be easily adapted to the new asynchronous framework with reasonable programming effort. Application programmers need to implement the *initAsynchronousExec* and *finalizeAsynchronousExec* virtual functions to initialize the task graphs for the corresponding level. To ease this process, we implemented a class named *RegionGraph* that can create a task graph for a level automatically using the metadata from BoxLib. A programmer can create a task graph simply by passing an object of the *MultiFab* class to the *RegionGraph* class constructor. A *MultiFab* contains all grids for a single level. A programmer has to replace the function calls to *fillPatch* and *average_down* with their asynchronous *push* and *pull* versions. Inside the newly developed task graph iterator, programmers can first pull, then compute, and then push the tasks using these asynchronous function calls. End users are insulated from the rest of the complexity involved in the asynchronous execution, which is handled inside the asynchronous BoxLib framework.

Currently, our implementation of the asynchronous AMR algorithm is restricted to a single time step. The asynchronous execution starts before computation of the coarsest level and continues all the way up to the finest. We synchronize all the processes after data is synchronized for the coarsest level. We currently compute the time step using a synchronous global reduction and our implementation does not support asynchronous regridding yet. In the future we will further increase asynchrony, which would support asynchronous task graph update when grid structure changes, asynchronous global reduction to compute time step, and asynchronous communication across time steps.

7 Results

We carried out performance study on the Hazel Hen supercomputer located at the HPC Centre, Stuttgart Germany. Compute node specifications on Hazel Hen are provided in Table 1. For performance measurement we use an explicit advection code based on BoxLib. The advection solver advects a scalar field with a prescribed time-dependent velocity on adaptive meshes. A finite-volume method with explicit time stepping is employed to solve the PDE. Although this is a simple system, the code contains all the AMR algorithmic components and communication patterns for building an explicit solver for a more complicated system of conservation law equations such as gas dynamics. For example, inter- and intra-AMR-level communication are needed for filling ghost cells. The mismatch of finite-volume flux at the coarse/fine interface needs to be corrected so that the conservation law is preserved. For comparison we use the existing Boxlib execution model as our baseline which implements Algorithm 2 with rank synchronous execution model discussed in the related work section. BoxLib reduces the global synchronization down to rank level and runs synchronously within a rank. All the experiments were performed using three levels of refinement, two subcycling iterations and a refinement ratio of 2.

Figure 3 shows strong scaling up to 12K cores where each bar is labeled with percent improvement obtained by the proposed asynchronous algorithm over

Table 1. Machine specifications for Hazel Hen

CPUs	Intel E5-2680 v3 (Haswell)	**Shared L3 (MB)**	30
Sockets/cores per socket	2/12	**Main memory (GB)**	128
Threads per core	2	**Memory bandwidth**	68 (GB/s)
Clock rate (GHz)	2.5	**Network bandwidth**	11.7 (GB/s)

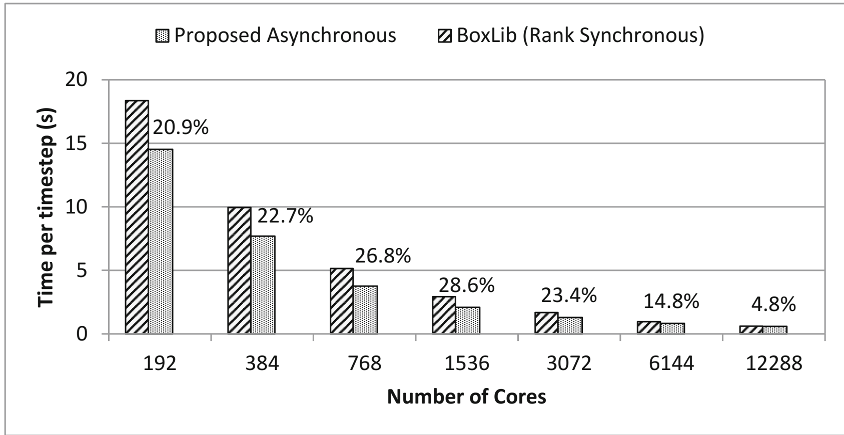


Fig. 3. Strong scaling for advection code on Hazel Hen

BoxLib. We used 1024^3 grid size as input for strong scaling studies. The y-axis shows the time spent in a single step of Algorithms 2 and 3. It does not include the time spent in timestep dt computation and global reduction. Proposed asynchronous algorithm achieves up to 28.6% performance improvement over BoxLib on 1536 cores. Performance improvement declines as we further increase the number of cores because the number of subgrids per process becomes too small to overlap any computation. There are a total of 6041 subgrids with size ranging from 128^3 to 8^3. For the maximum performance improvement case there are about 95 subgrids/rank while it reduces to less than 12 subgrids/rank in 12K cores. Although not shown here, we observe the same strong scaling behavior when two levels of refinements with subcycling and three levels of refinements without subcycling are used.

Figure 4 compares weak scaling for BoxLib's rank synchronous and proposed asynchronous algorithms. Grid size starts from 1024^3 for 768 cores and then doubled in x, y and z directions respectively. The proposed asynchronous algorithm achieves the same weak scaling behavior as BoxLib but with sustained performance improvement of more than 27%. This is possible because there are always sufficient number of subgrids per process to hide communication.

A breakdown (for strong scaling) of the time spent during computation (integration), restriction and prolongation for rank synchronous algorithm compared

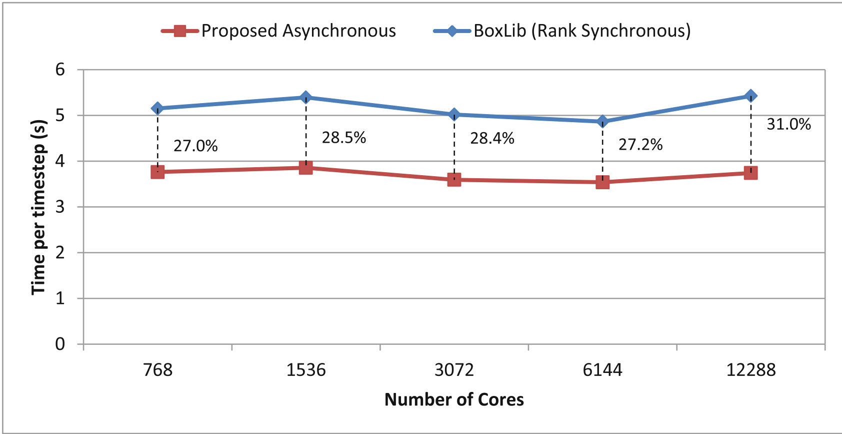


Fig. 4. Weak scaling for advection code on Hazel Hen

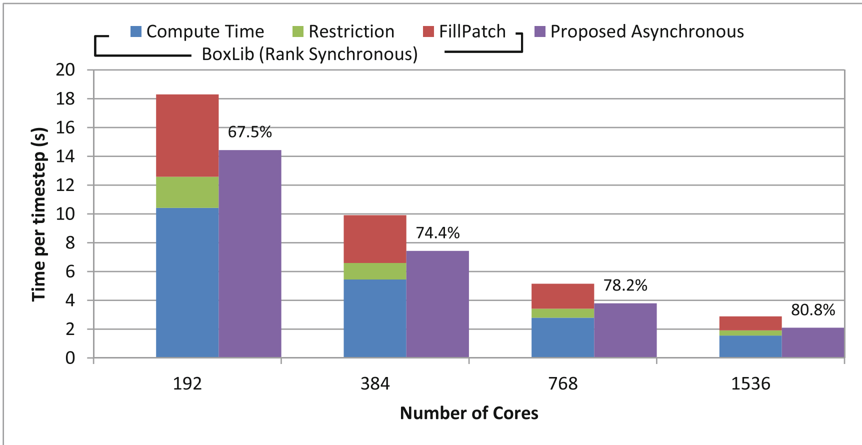


Fig. 5. Breakdown of performance for strong scaling achieved on Hazel Hen

to the proposed asynchronous algorithm is shown in Fig. 5. Both restriction and prolongation introduce communication. We can overlap only prolongation with computation because while performing restriction there is no computation to overlap with. The proposed asynchronous algorithm hides about 80% of the communication overhead due to prolongation behind the computation as shown in Fig. 5.

8 Conclusions

In this paper, we presented an asynchronous execution model for the AMR algorithm. Our asynchronous execution model allows a subgrid within a level to

perform computation independent of other subgrids at the same level to provide scalability but also maintains the programming simplicity for both AMR framework developers and the end users. We also discussed how our asynchronous algorithm can be integrated into an AMR framework. The results show that with affordable programming effort our asynchronous AMR algorithm can be adapted into AMR software frameworks to achieve decent speedup and scalability.

Acknowledgments. Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 215E185. Dr. Unat is supported by the Marie Sklodowska Curie Reintegration Grant 655965 by the European Commission. We acknowledge PRACE for awarding us access to the Hazel Hen supercomputer in Germany. 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. Boxlib: An AMR software framework. https://ccse.lbl.gov/BoxLib/
2. Enzo: AMR project. http://enzo-project.org/
3. Almgren, A.S., Beckner, V.E., Bell, J.B., Day, M.S., Howell, L.H., Joggerst, C.C., Lijewski, M.J., Nonaka, A., Singer, M., Zingale, M.: CASTRO: a new compressible astrophysical solver. I. Hydrodynamics and self-gravity. Astrophys. J. **715**(2), 1221–1238 (2010)
4. Almgren, A.S., Bell, J.B., Rendleman, C.A., Zingale, M.: Low Mach Number Modeling of Type la Supernovae. I. Hydrodynamics. Astrophys. J. **637**(2), 922–936 (2006)
5. Almgren, A., Bell, J., Lijewski, M., Lukić, Z., Van Andel, E.: Nyx: a massively parallel AMR code for computational cosmology. Astrophys. J. **765**, 39 (2013)
6. Ang, J., Barrett, R., Benner, R., Burke, D., Chan, C., Cook, J., Donofrio, D., Hammond, S., Hemmert, K., Kelly, S., Le, H., Leung, V., Resnick, D., Rodrigues, A., Shalf, J., Stark, D., Unat, D., Wright, N.: Abstract machine models and proxy architectures for exascale computing. In: 2014 Hardware-Software Co-Design for High Performance Computing, pp. 25–32. IEEE, November 2014
7. Bell, J.B., Day, M.S., Lijewski, M.J.: Simulation of nitrogen emissions in a premixed hydrogen flame stabilized on a low swirl burner. Proc. Combust. Inst. **34**(1), 1173–1182 (2013)
8. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. J. Comput. Phys. **53**(3), 484–512 (1984)
9. Chan, C.P., Bachan, J.D., Kenny, J.P., Wilke, J.J., Beckner, V.E., Almgren, A.S., Bell, J.B.: Topology-aware performance optimization and modeling of adaptive mesh refinement codes for exascale. In: Proceedings of 1st Workshop on Optimization of Communication in HPC, COM-HPC 2016, pp. 17–28. IEEE Press, Piscataway (2016)
10. Colella, P., Graves, D.T., Johnson, J.N., Johansen, H.S., Keen, N.D., Ligocki, T.J., Martin, D.F., Mccorquodale, P.W., Modiano, D., Schwartz, P.O., Sternberg, T.D., Straalen, B.V.: Chombo software package for AMR applications design document. Technical report (2003)

11. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: Flash: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. Astrophys. J. Suppl. Ser. **131**(1), 273 (2000)
12. Goodale, T., Allen, G., Lanfermann, G., Massó, J., Radke, T., Seidel, E., Shalf, J.: The cactus framework and toolkit: design and applications. In: Palma, J.M.L.M., Sousa, A.A., Dongarra, J., Hernández, V. (eds.) VECPAR 2002. LNCS, vol. 2565, pp. 197–227. Springer, Heidelberg (2003). doi:10.1007/3-540-36569-9_13
13. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on C++. In: Proceedings of Conference on Object Oriented Programming Systems, Languages and Applications, pp. 91–108 (1993)
14. Langer, A., Lifflander, J., Miller, P., Pan, K.C., Kalé, L.V., Ricker, P.: Scalable algorithms for distributed-memory adaptive mesh refinement. In: 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing, pp. 100–107, October 2012
15. MacNeice, P., Olson, K.M., Mobarry, C., de Fainchtein, R., Packer, C.: PARAMESH: a parallel adaptive mesh refinement community toolkit. Comput. Phys. Commun. **126**(3), 330–354 (2000)
16. Meng, Q., Luitjens, J., Berzins, M.: Dynamic task scheduling for the Uintah framework. In: 2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS), pp. 1–10. IEEE (2010)
17. Nguyen, T., Unat, D., Zhang, W., Almgren, A., Farooqi, N., Shalf, J.: Perilla: Metadata-based optimizations of an asynchronous runtime for adaptive mesh refinement. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 81:1–81:12. IEEE Press, Piscataway (2016)
18. Rendleman, C.A., Beckner, V.E., Lijewski, M., Crutchfield, W., Bell, J.B.: Parallelization of structured, hierarchical adaptive mesh refinement algorithms. Comput. Vis. Sci. **3**(3), 147–157 (2000)
19. Unfer, T., Boeuf, J.P., Rogier, F., Thivet, F.: Multi-scale gas discharge simulations using asynchronous adaptive mesh refinement. Comput. Phys. Commun. **181**(2), 247–258 (2010)
20. Wahib, M., Maruyama, N., Aoki, T.: Daino: a high-level framework for parallel and efficient AMR on GPUs. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 53:1–53:12. IEEE Press, Piscataway (2016)