Asynchronous Runtime for AMR Applications on Exascale Systems

by

Muhammad Nufail Farooqi

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

September 6, 2018

Asynchronous Runtime for AMR Applications on Exascale Systems

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Muhammad Nufail Farooqi

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Asst. Prof. Dr. Didem Unat (Advisor)

Prof. Dr. Metin Muradoğlu

Prof. Dr. Can Özturan

Asst. Prof. Dr. Alptekin Küpçü

Asst. Prof. Dr. Kamer Kaya

Date: _____

To my parents

ABSTRACT

Adaptive Mesh Refinement (AMR) is an approach to solving partial differential equations that reduces the computational and memory cost for certain applications. AMR represents the computational domain as a hierarchy of successively refined grids, introducing both intra- and inter-level communications in the hierarchy. 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 exascale supercomputers containing billion way parallelism, which will raise the communication cost further. One common technique to reduce the communication overhead is to overlap communication with computation through asynchronous execution. Re-designing AMR algorithms to avoid synchronization and manually restructuring an AMR application to realize communication overlap is extremely complicated and increases software development and maintenance costs due to the large code size and complex control structures.

We present a nonintrusive asynchronous approach to hide 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 proposed a phase asynchronous AMR algorithm and incorporated it in a widely used AMR framework, *AMReX*. Unlike synchronous algorithm where computation is delayed until all communication is finished, in phase asynchronous algorithm, a subgrid at an AMR level is scheduled for computation as soon as its dependent communication is completed. However, computation of all subgrids at an AMR level are completed before starting computation on a next AMR level. Our phase asynchronous AMR algorithm aims to achieve the performance of a fully asyn-

chronous execution while retaining the productivity of a synchronous approach.

We implemented a runtime system to facilitate the proposed phase asynchronous execution for AMR application. The runtime enables overlapping of both intra and inter-level communication with the computation within a level. Runtimes include communication handlers that perform both intra and inter-node communication. Applications hand over the communication data to the runtime's communication handler. As soon as communication dependencies for a subgrid are completed, the runtime schedules that subgrid for computation.

We carefully designed an API for phase asynchronous execution model to facilitate the transition of large legacy codes from synchronous to asynchronous execution. As a case study, we pick the CASTRO code, which solves multicomponent compressible hydrodynamic equations for astrophysical flows, as a real-world code to demonstrate our new asynchronous AMReX framework. We discuss the transformation strategy of CASTRO and evaluate the performance and programming effort required to use our new API. Our approach achieves the performance requirements while retaining readability and long-term maintainability without severely affecting the productivity of the application programmer.

Supercomputer architectures are trending towards heterogeneity. At the moment, 110 out of top 500 supercomputers uses accelerators this number includes five machines from the top 10. Thus programming models without support for heterogeneous architectures are not likely to be adopted on these machines. We extend our runtime to support execution on heterogeneous architectures. We present how our programming model and runtime system deal with the challenges raised when porting AMR application on heterogeneous architectures. The runtime schedules computation on both CPUs and GPUs concurrently to effectively use of all computational resources while maintaining the productivity.

For performance studies, we use a small heat advection application and a large real-world production code, CASTRO. We demonstrate performance on Hazel Hen and

Cori supercomputers; on the Intel Haswell cores and on the Intel Xeon Phi (Knights Landing- KNL). With reasonable programming effort we were able to achieve up to 50% performance improvements using 49K cores on Intel Haswell and 278,528 cores on KNL. We also carried out performance analysis of the heterogeneous architecture support in the runtime using the heat advection code on SummitDev. SummitDev is a heterogeneous architecture machine equipped with IBM Power8 CPUs and NVIDIA Tesla P100 GPUs.

ÖZETÇE

Uyarlanabilir Izgara Geliştirimi (Adaptive Mesh Refinement - AMR) belirli uygulamalar için hesaplama ve bellek maliyetini düşüren, kısmi ayrışık denklemlerin çözümü için geliştirilen bir yaklaşımdır. AMR, hesaplama alanını, ardışık olarak rafine edilmiş ızgaraların bir hiyerarşisi olarak temsil eder ve hiyerarşide hem iç hem de dış iletişimi ele alır. Düz mantık bir AMR algoritması bir zaman adımı içerisinde bile, iletişim yönünden zaten pahalı olan pek çok eşzamanlama noktasını içerir. Bu sorun iletişim maliyetini daha da artıran milyar yönlü paralleliğe (billion way parallelism) sahip exascale süper bilgisayarlarda daha da belirgin hale geliyor. Asenkron yürütme aracılığıyla haberleşmeyi hesaplama ile örtüştürmek, iletişim maliyetini azaltmak için yaygın olarak kullanılan bir tekniktir. Senkronizasyondan kaçınmak için AMR algoritmalarını yeniden tasarlamak ve bir AMR uygulamasında iletişimin ve hesaplamanın örtüşmesini sağlamak için uygulamayı yeniden yapılandırmak fazlasıyla karmaşıktır. Dahası bu iki çözüm, geniş kod büyüklüğü ve karmaşık kontrol yapılarından dolayı yazılım geliştirmeyi ve bakım maliyetlerini artırır.

AMR uygulamalarında haberleşmenin etkilerini gizlemek için kesintisiz (nonintrusive) asenkron bir yaklaşım sunmaktayız. Yaklaşımımız, AMR uygulamaları ile ilgili alan bilgisini kullanarak veri bağımlılıklarını otomatik olarak algılar ve makul miktarda kod değişikliği ile asenkronizasyonu sağlar. Bu yaklaşımı kullanarak, faz asenkron bir AMR algoritması önermekte ve onu yaygın kullanılan AMReX adında AMR çerçevesine dahil etmekteyiz. Tüm haberleşme bitene kadar hesaplamanın geciktirildiği senkron algoritmaların aksine, faz asenkron algoritmasında, bağımlı haberleşme tamamlandığında AMR seviyesindeki bir alt ızgarada esaplama yapılır. Bununla birlikte bir AMR seviyesindeki tüm alt ızgaraların hesaplanması bir sonraki AMR seviyesindeki hesaplama başlamadan önce tamamlanır. Faz asenkron AMR algo-

ritmamız, hem senkron bir yaklaşımın verimliliğine sahiptir hem de tam asenkron uygulama performansını elde etmeyi amaçlamaktadır.

AMR uygulamaları için önerilen faz asenkron yürütmeye olanak sağlayan yürütme sistemi gerçekleştirdik. Yürütme sistemi hiyerarşideki hem iç hem de dış haberleşmenin hesaplama ile örtüşmesine olanak vermektedir. Yürütme sistemi hem iç hem de dış haberleşmeyi gerçekleştiren haberleşme denetçilerini (communication handlers) içermektedir. Uygulamalar haberleşme verisini yürütme sisteminin haberleşme denetçisine devretmektedir. Bir ızgara için haberleşme bağımlılıkları tamamlanır tamamlanmaz, yürütme sistemi o ızgaranın hesaplanmasını planlamaktadır.

Büyük eski kodlarının senkronize olmayan yürütmeden senkronize yürütmeye geçişini kolaylaştıran, faz asenkron yürütme sistemi için uygulama programı arabirimini (UPA) dikkatli bir şekilde tasarladık. Yeni asenkron AMReX çerçevemizi tanıtmak için gerçek-dünya kodu olan, astrofizikle ilgili akımlar için çok bileşenli sıkıştırılabilir hidrodinamik denklemleri çözen CASTRO kodunu bir vaka çalışması olarak ele aldık. CASTRO'nun dönüşüm stratejisini tartıştık ve yeni UPA'mızı kullanmak için gereken gereken programlama çabasını ve performansı değerlendirdik. Yaklaşımımız, uygulama programcısının verimliliğini ciddi biçimde etkilemeden okunurluğu ve uzun vadeli sürdürülebilirliği korurken, performans gerekliliklerini de yerine getirmektedir.

Süper bilgisayar mimarileri heterojenliğe doğru yönelmektedir. şuanda 500 süper bilgisayardan 100 tanesi hızlandırıcı kullanmaktadır ve bu sayı ilk 10 içerisindeki beş makineyi de kapsamaktadır. Bu nedenle heterojen mimariler için destekten yoksun olan programlama modellerinin bu makinelere uyarlanma ihtimali düşüktür. Heterojen mimariler üzerinde yürütmeyi desteklemek için yürütme sistemimizi genişlettik. Programlama modelimizin ve yürütme sistemimizin, AMR uygulamaları heterojen mimariler üzerine taşındığında ortaya çıkan güçlüklerle nasıl başa çıktığını göstermekteyiz. Yürütme sistemi, verimliliği korurken tüm hesaplama kaynaklarının etkin kullanımı için eşzamanlı olarak hem CPU'larda hem de GPU'larda hesaplama yapar.

Performans çalışmalarında, küçük bir ısı adveksiyon uygulamasını ve geniş bir

gerçek dünya üretim kodu olan CASTROyu kullanmaktayız. Intel Haswell ve Intel Xeon Phi (Knights Landing- KNL) kullanan, Hazel Hen ve Cori süper bilgisayarlarındaki performansı gösterdik. Makul bir programlama çalışmasıyla, Intel Haswell üzerinde 49K çekirdek ve KNL üzerinde 278,528 çekirdek kullanarak yüzde 50 daha iyi performans almayı başardık. Ayrıca yürütme sisteminde heterojen mimari desteğin performans analizini SummitDev'de ısı tutma kodunu kullanarak yaptık. SummitDev, IBM Power8 CPUlar ve NVIDIA Tesla P100 GPUlar ile donatılmış heterojen bir mimariye sahiptir.

ACKNOWLEDGMENTS

First of all, let me express my sincere gratitue to my advisor *Asst. Prof. Didem Unat*, you have been an exceptional mentor for me. I would like to thank you for encouraging my research and helping me to grow as a researcher. I could not have imagined having a better advisor and mentor for my Ph.D study. Besides my advisor, special thanks the my thesis progress committee and jury members: *Prof. Metin Muradoğlu* and *Prof. Can Özturan* from Boğaziçi University for their insightful comments and encouragement. I would also like to thank my thesis jury members: *Asst. Prof. Alptekin Küpçü* and *Asst. Prof. Kamer Kaya* from Sabancı University.

I would especially like to thank *Tan Nguyen*, *Weiqun Zhang*, *Ann S. Almgren* and *John Shalf* from Lawrence Berkeley National Laboratory, who provided me an opportunity to work in close collaboration with them, and who gave me access to the Cori supercomputer and the SummitDev machine. Without their precious support it would not be possible to conduct this research. I acknowledge PRACE for awarding me access to the Hazel Hen supercomputer in Germany. I am also thankful to Koç University for funding my Ph.D. study.

I thank my fellow labmates in ParCoreLab for the thought-provoking discussions in our group meetings, and for all the fun we have had in the last couple of years. In particular, I am grateful to *Burak Bastem* for helping me in translating my thesis abstract to Turkish. Last but not the least, I would like to thank my family for supporting me throughout my Ph.D. study and my life in general.

TABLE OF CONTENTS

List of Tables **xiv**

List of Figures **xv**

Nomenclature **xviii**

Chapter 1: Introduction **1**

Chapter 2: Background **5**

2.1 Block-Structured Adaptive Mesh Refinement (SAMR) 5

2.1.1 Communication in AMR Applications 7

2.2 AMR Execution Models 9

2.3 AMR Frameworks 10

2.3.1 BoxLib Framework 10

2.3.2 AMReX Framework 11

2.4 Perilla Runtime System 11

2.5 CASTRO: An Astrophysics Code 12

2.6 TiDA Tiling Library 13

Chapter 3: Programming Model and Runtime System **16**

3.1 System Architecture 16

3.2 Programming Model 17

3.3 Runtime Implementation 17

3.3.1 Communication Handlers 18

3.3.2 Message Queues and Shared Memory Protection 19

3.3.3 Thread-Work Assignment 19

Chapter 4: AMR Algorithm 22

4.1 Synchronous AMR Algorithm 22

4.2 Phase Asynchronous AMR Algorithm 24

4.3 Integration of Phase Asynchronous AMR Algorithm in AMReX . . . 27

Chapter 5: User Interface 29

5.1 Meta-data Extraction 29

5.2 Task Graph Iterators 32

5.3 Impact on Productivity 35

Chapter 6: Case Study: CASTRO - An Astrophysics Code 37

6.1 Control Structure (Synchronous) 37

6.2 Control Structure (Asynchronous) 39

6.3 Programming Effort 41

Chapter 7: Execution on Heterogeneous Architectures 42

7.1 GPU Kernels 42

7.2 Memory Model 43

7.3 Work Scheduling on Heterogeneous Architectures 44

7.4 API 46

7.5 Multi-GPU Support 47

Chapter 8: Evaluation 48

8.1 Performance Studies using CASTRO 49

8.1.1 Input Configuration Selection 49

8.1.2 Strong Scaling Studies 51

8.1.3 Performance at Large Scale 53

8.1.4 Hyperthreading 54

8.2 Load Balancing 55
8.2.1 Comparison of Load Balancing Strategies 56
8.3 Performance using Heat Advection Solver 58
8.3.1 Strong Scaling Studies 58
8.3.2 Weak Scaling Studies 59
8.4 Performance Analysis on Heterogeneous Architectures 61

Chapter 9: Related Work **64**
9.1 Asynchronous AMR Frameworks 64
9.2 Asynchronous Programming Models and Runtime Systems 65
9.2.1 Cilk 66
9.2.2 Legion 66
9.2.3 StarPU 67
9.2.4 SWARM 68

Chapter 10: Conclusion **69**

Bibliography **71**

LIST OF TABLES

8.1 Machine specifications 48
8.2 Input configurations 49
8.3 Machine specifications for SummitDev 61
8.4 Input configurations for heat advection solver on SummitDev 62

LIST OF FIGURES

2.1 Sample AMR in two dimensions with two levels of refinement 6
2.2 Possible data locations on a 2D grid: a) cell-centered, b) nodal in x and c) nodal in x and y (figure courtesy BoxLib user guide) 6
2.3 AMR with two levels of refinement, showing intra-level communication. 7
2.4 AMR with two levels of refinement, showing inter-level communication. 7
2.5 Subcycling with 2 levels of refinement (figure courtesy BoxLib user guide) 8
2.6 AMR Execution Models 9
2.7 Example: 4 Perilla teams on a compute node with 4 NUMA nodes (red thread indicates the communication thread while black indicates the worker thread) 12
2.8 A 3D array divided into regional tiles and showing logical tiles in a region(figure courtesy of D. Unat [54]) 14

3.1 Runtime System's Architecture Overview 16
3.2 Local and Remote Communication/Message Handlers 18
3.3 Thread-work assignment 20

4.1 Asynchronous computation and communication overlap 26

5.1 Asynchronous runtime's API for meta-data extraction. For clarity, the framework specific parameters are omitted. 30
5.2 AMReX's MultiFab iterator 34
5.3 Asynchronous runtime's task graph iterator. 34
5.4 An iterative solver written under the asynchronous runtime 35
5.5 An iterative solver written under the proposed API 35

6.1 CASTRO's control structure (Synchronous) 38

6.2 CASTRO's control structure (Asynchronous) 40

7.1 GPU computation overlap with data transfers between host and device 43

7.2 Work scheduling on both CPU and GPU 45

7.3 Asynchronous runtime's task graph iterator with support for heterogeneous architectures. 46

8.1 Comparison of different number of MPI processes per node on (a) Cori-Haswell using Rayleigh-Taylor and (b) Cori-KNL using Sedov-Taylor. (Figure is newly added to support our argument for using 2 and 4 MPI processes per node on Cori-Haswell and Cori-KNL respectively.) . . . 50

8.2 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. 51

8.3 Strong scaling on Cori-Haswell 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. 52

8.4 Performance at large scale on Cori-KNL for the test problems R-T in blue and S-T in orange color. 53

8.5 Different degrees of hyperthreading support in asynchronous AMReX (1024 Cori-KNL nodes) using configurations in Table 8.2 for the R-T test problem. C and S in the x-axis labels represent Compact and Scatter affinity. 54

8.6 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. 56
8.7 Comparing SFC vs RR in terms of number of grids (Fabs) per process (Rayleigh-Taylor) 57
8.8 Strong scaling for advection code on Hazel Hen 59
8.9 Weak scaling for advection code on Hazel Hen 60
8.10 Breakdown of performance achieved on Hazel Hen 60
8.11 Comparison of different implementations and amount of computation on SummitDev. 63

NOMENCLATURE

AMR	-	Adaptive Mesh Refinement
API	-	Application Programming Interface
CPU	-	Central Processing Unit
D2H	-	Device to Host
DDR	-	Double Data Rate
GB	-	GigaByte
GHz	-	GigaHertz
GPGPU	-	General-Purpose GPU
GPU	-	Graphics Processing Unit
GT	-	Giga Transfers
H2D	-	Host to Device
HPC	-	High Perfromance Computing
ID	-	Identification
KNL	-	Knights Landing
LBNL	-	Lawrence Berkeley National Laboratory
LLC	-	Last Level Cache
MB	-	MegaByte
MCDRAM	-	Multi-Channel Dynamic Random-Access Memory
MPI	-	Messgae Passing Interface
MPS	-	Multi Process Service
NUMA	-	Non-Uniform Memory Access
PDE	-	Partial Differential Equation

PCIe	-	Peripheral Component Interconnect Express
OpenACC	-	Open Accelarators
OpenMP	-	Open Multi-Processing
RR	-	Round Robin
R-T	-	Rayleigh-Taylor
SAMR	-	Block-Structured AMR
SFC	-	Space Filling Curve
S-T	-	Sedov-Taylor
TBB	-	Threading Building Blocks
TF	-	TeraFlop
TiDA	-	Tiling as Durable Abstraction
UPC	-	Unified Parallel C

Chapter 1

INTRODUCTION

Power wall[29] diminished the technology of increasing clock rate by adding transistors on chip to speedup computations. Multi-core processors emerged as an alternative and increase in on-chip parallelism lead us to an era of many-core processors. This trend is expected to continue and at exascale the number of cores per chip will be in the order of thousand[10]. Present fastest supercomputer is at petascale having a compute power of 122.3 petaflops (10^{15} flops)[4] while at exascale it will be in the order of exaflops (10^{18} flops) that is expected to arrive by 2023. Increasing on-chip parallelism on one hand provides us resources to accelerate computation but it also brings up challenges for programmers to use it effectively.

Modern machines are complex having multiple levels of memory, heterogeneous memory and heterogeneous processing units[10]. Two different types of memories MCDRAM (high bandwidth memory) and DDR on Intel Knights Landing[53] architecture is an example of memory heterogeneity. Commonly used types of processing units in top 500 supercomputers[4] are CPU, GPGPU, and Intel Xeon Phi, while a significant share of their total computing power comes from accelerators (GPU+Xeon Phi). The heterogeneity trend of both memory and processor is expected to continue further at exascale as well[10]. Thus runtime system design should consider optimization that can get the best out of these heterogeneous architectures.

Supercomputer architectures are becoming more complex which demands a great deal of efforts from programmers to use these resources at there full capacity. This is true for both developing new and porting legacy applications to the new architectures. Programming models and runtimes are developed to assist programmers

in developing applications for these architectures. A programming model provides a collection of abstractions that a programmer can use to design an algorithm. Runtime systems are implemented to support one or more of these programming models. There exist a variety of parallel programming models like message passing, shared memory, task parallel and data parallel. Some examples of runtimes systems are MPI[2], OpenMP[3], Cilk[16], and TBB[48]. MPI support message passing, task and data parallel programming models. OpenMP, Cilk and TBB supports shared memory, task and data parallel programming models. New programming models with more abstractions and underlying runtimes will help programmers in rapid application development.

Many computational science and engineering problems are modeled in the form of partial differential equations (PDEs). Although a high-resolution mesh is required for improved accuracy, usually some limited regions are of more interest where additional accuracy is desired. Adaptive mesh refinement (AMR)[15] methods provide the mechanism to locally refine areas of interest. AMRs can also dynamically refine or coarsen mesh resolution as required during the computation. There are a variety of AMR techniques and this work discusses AMR in the context of time-subcycle blocked-structured AMR. Block-structured AMR (SAMR) is one of the AMR methods that use regular grids organized into multi-levels. Each level of SAMR is at different resolution starting from the coarsest level at the bottom which covers the whole domain. Areas of interest are refined gradually in a nested manner and go up to the finest level at the top. Adaptive mesh refinement is advantageous because it can deliver the same accuracy at reduced computational cost and memory usage.

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. [20] discussed four different execution models for AMR ranging from fully synchronous to fully asynchronous with different programmability tradeoffs.

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.

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 [9] 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 [8], LMC [14], SMC [26], and Nyx [7] are among the many research codes in use today that are based on AMReX.

This thesis proposes a programming model and runtime system that enables AMR applications to execute asynchronously on exascale architectures with low programming overhead. In summary, we make the following contributions

1. propose an asynchronous AMR algorithm that reduces most of the synchronization costs
2. integrate the asynchronous AMR algorithm into the AMReX framework
3. develop runtime system to support computation and communication overlap
4. unveil the simple application programming interface to reduce programming effort
5. discuss the experience in porting a large astrophysics application, CASTRO

6. performance evaluation on Hazel Hen, Cori-Haswell, and Cori-KNL supercomputers

Rest of the dissertation is organized as follows: Chapter 2 build some background about the Block Structured AMRs, AMR frameworks, AMR Execution models, Perilla runtime, CASTRO and TiDA library. Chapter 3 explains the programming model and implementation of the runtime system for computation and communication overlap and the additional optimizations performed for Intel's Knights Landing (KNL) architecture. Chapter 4 proposes phase asynchronous AMR algorithm for Block-Structured AMRs and discusses its integration into the AMReX framework. Chapter 5 introduces the carefully designed application programming interface (API) with focus on reducing programming overhead. Chapter 6 discusses the porting of CASTRO code to the proposed programming model and quantifies the required programming overhead. Chapter 7 present solutions to the challenges raised when executing AMR applications on heterogeneous architectures. Chapter 8 evaluates the performance of the proposed programming model and runtime system using advection solver on Hazel Hen and SummitDev and using CASTRO code on Cori-Haswell and Cori-KNL supercomputers. Chapter 9 present some related work and its comparison to the work proposed in this dissertation. Chapter 10 concludes this dissertation.

Chapter 2

BACKGROUND

Here, we would like to introduce basic concepts and tools required for the understanding of this work before diving into the details. We further divide this chapter into the following six sections: 2.1) will explain the working and types of communications in block-structured AMR. Based on the level of synchrony, 2.2) discuss four types of execution models that an AMR application can adopt. 2.3) give an overview of the AMR framework BoxLib and its successor AMReX. We used these frameworks to implement our proposed phase asynchronous AMR algorithm. 2.4) introduce Perilla [47] runtime that we used as a base for our asynchronous runtime. 2.5) gives background about CASTRO code which is used to demonstrate the performance and productivity of the proposed programming model. Finally, 2.6) explains the TiDA tiling library that we used to divide work among threads.

2.1 Block-Structured Adaptive Mesh Refinement (SAMR)

Scientists usually model problems as partial differential equations and solve their discretized version computationally on meshes with appropriate resolution. Adaptive mesh refinement (AMR) methods provide a cost effective way for the solution of PDEs by using finner meshes only at regions of interest. Block-structured adaptive mesh refinement is one of the many AMR methods. SAMR[15] 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 as goes up in the chain with finest grid at the top level. Figure 2.1 shows 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. All grids at a level are of the same resolution. Given

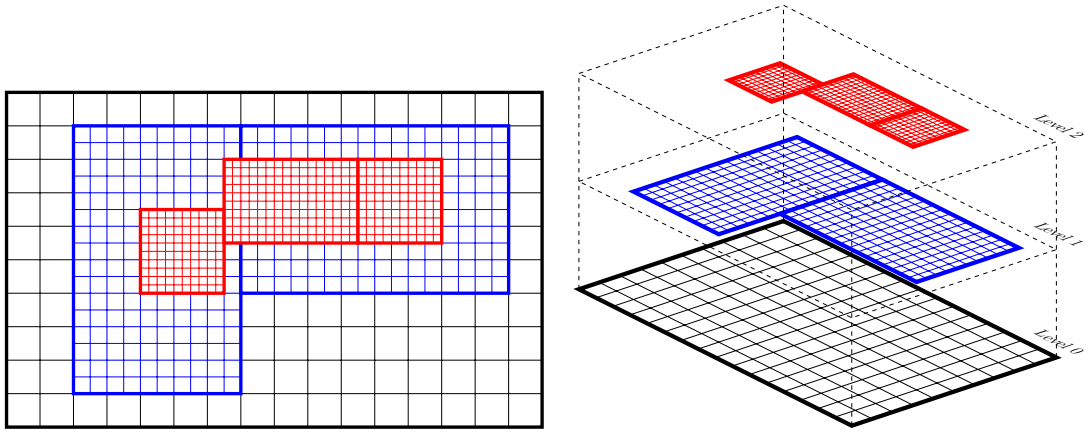


Figure 2.1: Sample AMR in two dimensions with two levels of refinement

maximum number of levels at the start, the number of refinement levels can vary dynamically during the simulation.

Data can be stored at different locations on a grid. Figure 2.2 shows multiple positions on a two-dimensional grid where data can be stored. Data when stored at the center of a gird cell is known as cell-centered. While data at the edge of a grid is know as nodal. It is said to be nodal in one direction (face-centered) if reside on a single edge or said to be nodal in two directions when residing on an intersection of two edges.

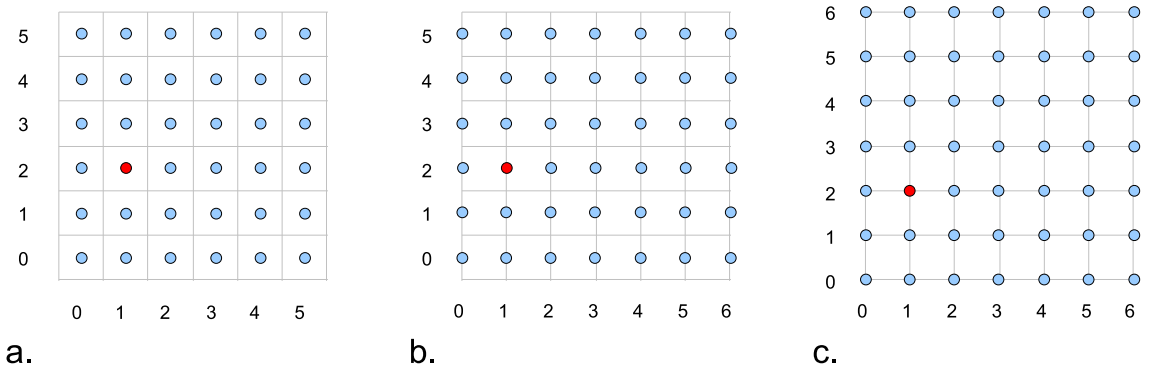
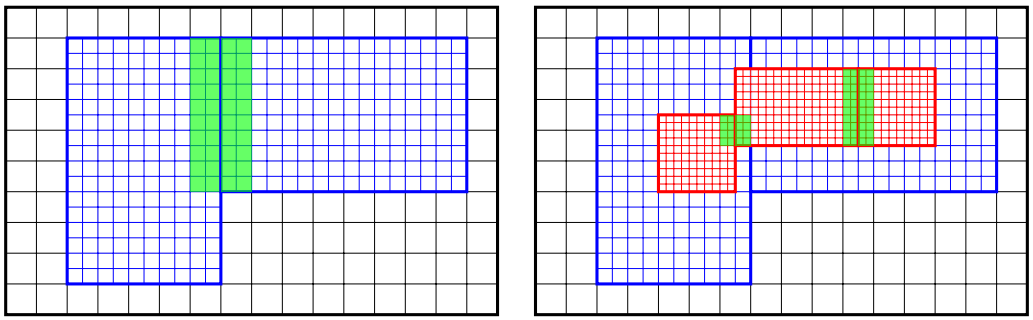


Figure 2.2: Possible data locations on a 2D grid: a) cell-centered, b) nodal in x and c) nodal in x and y (figure courtesy BoxLib user guide)

2.1.1 Communication in AMR Applications

AMR grids need to communicate ghost cells with each other while scientific problems are being simulated on them. This communication takes place among grids within a level (intra-level) as well as among grids across different levels (inter-level). Figure 2.3 shows intra-level communication where the green part in (a) shows the cells that are needed to be communicated among two grids at level 1 and (b) shows cells to be communicated among grids at level 2.



(a) Communication among grids within Level 1 (b) Communication among grids within Level 2

Figure 2.3: AMR with two levels of refinement, showing intra-level communication.

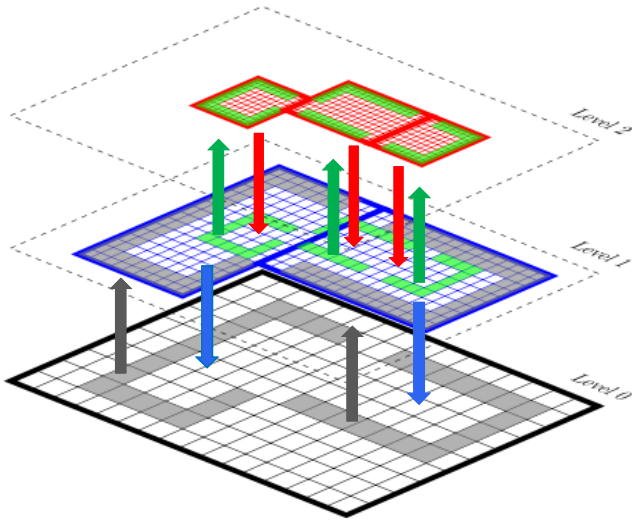


Figure 2.4: AMR with two levels of refinement, showing inter-level communication.

Two types of inter-level communication, restriction and prolongation, are needed when copying data between grids from different levels. When data is interpolated from a coarser grid and copied to a finer grid is known as prolongation. Figure 2.4 shows prolongation with arrows where gray cells are interpolated and copied from level 0 to level 1 while green cells are interpolated and copied from level 1 to level 2. On the other hand, when data is averaged at a finer grid and copied to a coarser grid is called restriction. In Fig. 2.4, all grids at level 2 are averaged and copied to the cells enclosed in the green cells at level 1 is restriction likewise all grids at level 1 are averaged and copied to cells enclosed in gray cells at level 0.

The amount of inter-level communication can vary based on two scenarios. One is without time subcycling having simpler communication while the other one with time subcycling having relatively complex communication. Figure 2.5 shows the subcycling algorithm with 3 levels of refinement. Subcycling algorithm needs fewer computation at the cost of serialized computation.

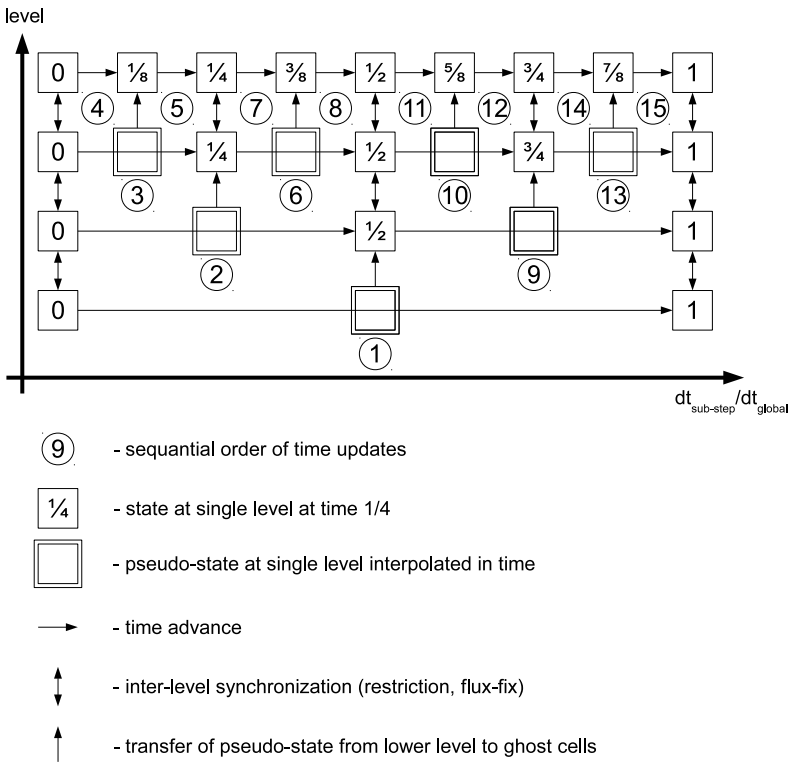


Figure 2.5: Subcycling with 2 levels of refinement (figure courtesy BoxLib user guide)

2.2 AMR Execution Models

Chan et al. [20] classified AMR execution models into four modes ranging from *fully synchronous* to *fully asynchronous* as shown in Fig. 2.6. The trade-off between the models 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, where global synchronization is required after communication and before starting any computation at an AMR level. *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. 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. The asynchronous AMR algorithm proposed in this dissertation is analogous to the *phase asynchronous* execution model but do not restrict communication overlap to a single timestep.

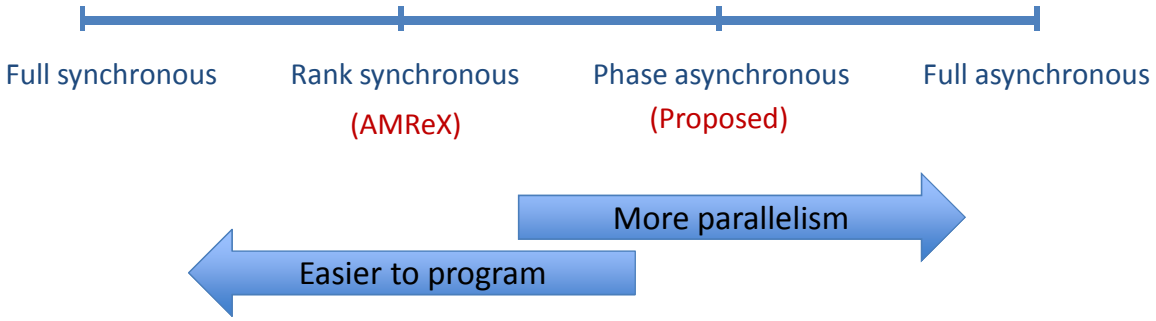


Figure 2.6: AMR Execution Models

2.3 AMR Frameworks

2.3.1 BoxLib Framework

BoxLib[1] is a software framework that can be used to develop block-structured adaptive mesh refinement(AMR) applications. BoxLib is used by many scientists around the world to develop AMR programs. Some of the large applications for astrophysics (CASTRO[8] and MAESTRO[14]), cosmology (Nyx[7]) and low Mach number combustion (LMC[6]) simulations has been developed using BoxLib. It is publicly available and has been developed at the Lawrence Berkeley National Laboratory (LBNL).

In the BoxLib framework, a Box is the basic element of BoxLib data structure that contains high and low ends of a rectangular grid. A Box does not contain any data but knows whether the data is cell-centered, face-centered or nodal. All boxes representing the sub-grids at a level are organized into an array called BoxArray. Actual data for grid represented by a Box is stored in a "Fortran array box" called Fab. Fab's structure contains a Box and a pointer to a four-dimensional Fortran array. A Box associated with a Fab is grown (include ghost cells) version of the corresponding Box in the BoxArray. The data allocated in a Fab also contains the ghost cells. BoxLib applications usually deal with a collection of Fabs known as MultiFab. A MultiFab, similar to the BoxArray, is an array of all Fabs on a level. AMR contains a number of levels depending on the refinement requirement of applications. Each AMR level has a MultiFab and a BoxArray associated with it.

MultiFab is the basic abstraction for parallelization in BoxLib. To perform computation on a MultiFab in parallel, Fabs are distributed among different processors. In the case of multiple levels, Fabs on each MultiFab are distributed autonomously. During computation, grids may need data from its neighboring grids. BoxLib provides subroutines to exchange ghost cells among different processors. To support ghost cell exchanges metadata for inter-process communication is also stored in the BoxLib data structure.

For multicore architectures, BoxLib uses hierarchical programming model based

on MPI+OpenMP. OpenMP uses bulk synchronous execution model. BoxLib implements a *rank synchronous* model.

2.3.2 AMReX Framework

AMReX [9], 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 adopts all the basic classes like Box, BoxArray, Fab and MultiFab from BoxLib.

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.

2.4 Perilla Runtime System

Perilla[47] is a runtime, developed as a result of LBNL and Koç University collaboration, that enables applications to run asynchronously. It uses the metadata from BoxLib and TiDA. Perilla extracts communication and task dependency metadata from the BoxLib to generate a task graph. Its task graph is a collection of tasks connected via dependencies on the data. Tasks in graph are executed asynchronously as soon as their data dependencies are met. Metadata from TiDA is used to iterate

over tiles.

Perilla uses MPI+X programming model. Where X is shared memory programming API e.g. OpenMP. Within a shared memory node Perilla creates thread teams. An example of Perilla thread teams on a compute node with 24 threads and 4 NUMA nodes is shown in Figure 2.7. A thread team consist of a communication thread and worker threads. The communication thread in a team is responsible of communication and worker threads carry out computations. There is only one thread responsible for off-node communication for all teams on the shared memory node.

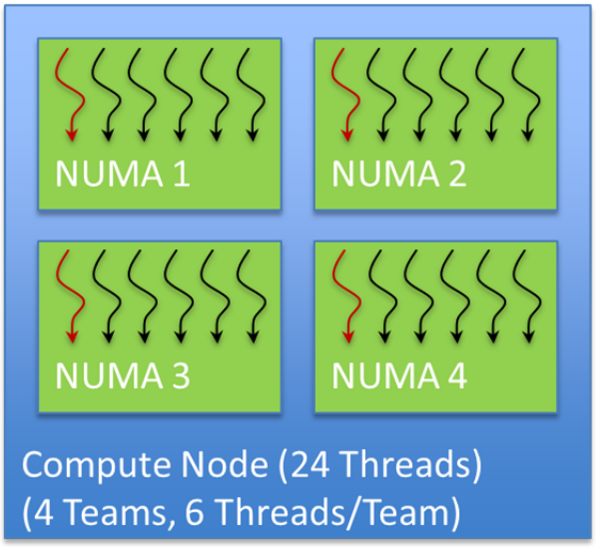


Figure 2.7: Example: 4 Perilla teams on a compute node with 4 NUMA nodes (red thread indicates the communication thread while black indicates the worker thread)

2.5 CASTRO: An Astrophysics Code

CASTRO [8, 60, 61] 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., [38, 19, 22]).

In this dissertation, 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.

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.

2.6 TiDA Tiling Library

Energy cost for computation is decreasing rapidly as compared to data movement[55]. And the trend is expected to continue thus rising the gap between energy costs. Data locality reduces this gap by keeping data close to computation and avoids unnecessary communication.

Tiling also referred to as cache blocking is widely used technique for optimizing data locality[40]. Generally loops are tiled manually in a program to optimize for data locality. A lot of work has been carried out to integrate automatic tiling process into compilers[31, 32, 39]. However it is difficult to generate optimized code in a fully automatic way that is why commercial compilers provide limited support for tiling.

TiDA[54], developed at ParCoreLab (Koç University) in collaboration with LBNL, is tiling based programming model that provides a library interface to express parallelism and data locality. TiDA, implemented in both C++ and Fortran, presents simple programming abstractions for writing loop based codes. It has three programming abstractions: Region, Tile and Tile Iterator.

- **Region:** is a contiguous chunk of data in a *regional tile* resulted from partitioning of problem domain as shown in Figure 2.8. Data from two regions is discontinuous. Regions addresses locality across NUMA domains or the expected regional coherence domains in exascale architecture.
- **Tile:** represents a logical partition of data in the iteration space. Logical tiles targets on-chip memory locality. A logical tile within a region is shown in Figure 2.8. Tiles can be resized without affecting the actual data to fit working set size into available on-chip memory.
- **Tile Iterator:** is an abstraction that separates loop traversal from loop body to hide traversal orders, execution strategies and parallelization.

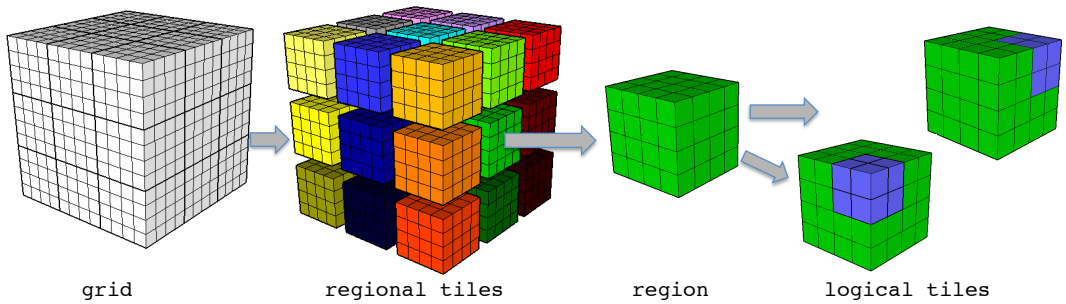


Figure 2.8: A 3D array divided into regional tiles and showing logical tiles in a region(figure courtesy of D. Unat [54])

In TiDA implementation, *regions*, holding the actual data, are grouped into an array named *tilearray*. The *tilearray*s replaces the traditional multidimensional arrays that are used to hold values for the physical domain. A *tile* has a unique ID, low and high ends of a rectangular portion in the multidimensional array and does not contain any data. Like *regions*, *tiles* are also gathered in an array called *abstilearray*. It does not have any data but represents an abstract structure of the *tilearray*. A programmer typically does not deal with *tilearray* and *abstilearray* after creating them. Rather programmers interact with tile iterator to manipulate TiDA arrays.

Choosing an optimal tile size is crucial to achieve good performance and tile size depends on underlaying architecture and other optimizations. To improve productivity TiDA also provides environment variables (TIDA_TILESIZE and TIDA_REGIONS) to specify tile size and number of regions. Programmers may need to create different number of regions depending on number of NUMA nodes on the compute node.

TiDA's logical tiling and tile iterator are also been successfully implemented in the BoxLib[59].

Chapter 3

PROGRAMMING MODEL AND RUNTIME SYSTEM

In this chapter, first we will describe the system architecture that identifies the place where does the proposed runtime lies in the AMR application development stack. Then we will present the proposed programming model and various aspects of the runtime. Lastly, we will also discuss the effects of asynchronous execution on load balancing.

3.1 System Architecture

High-level frameworks are developed to make application programmer's job easier by hiding the underlying complex models and hardware. These high-level frameworks continuously adapt to changes in the underlying models and hardware while keeping the application programmer's efforts at minimum. For productivity, we choose to implement our proposed approach in the AMReX framework yet it can also be implemented by standalone AMR applications.

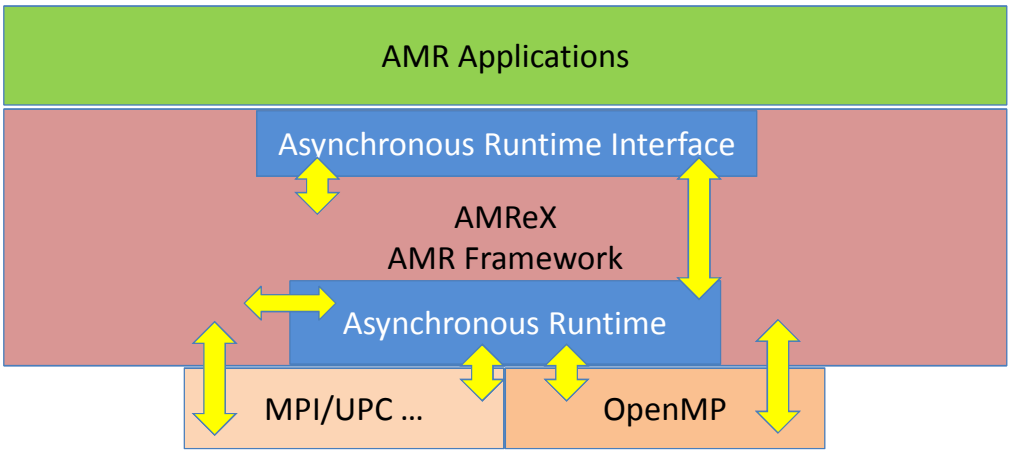


Figure 3.1: Runtime System's Architecture Overview

An illustration of the runtime system is shown in Figure 3.1. The runtime system is built on the top of MPI and OpenMP programming models. All inter-process communication is handled through MPI's non-blocking subroutines and OpenMP is used for intra-process task as well as data parallelism. The runtime automatically extracts the communication meta-data from the AMReX framework. Application programmers do not need to worry about the AMR algorithm implementation as we implement our proposed phase asynchronous AMR algorithm inside the AMReX framework. Similar to the existing AMReX interface, we also provide a simple application programming interface (API) to implement AMR applications.

3.2 Programming Model

We propose a data-driven and graph-based model, where a task graph is a collection of tasks connected via dependencies on data. A sequence of statements using data of a *region* is defined as task. Here, we decided to define a subgrid as a *region* but finer granularity a subgird can be further divided into parts where each part can be defined as a *region*. Task graphs are created using the meta-data for communication that is obtained from the AMR framework.

A task can be in one of these states: waiting, runnable, execution and completed. A task can go from waiting to runnable (ready) state if all the data required by the sequence of statements is received. Runnable tasks can be scheduled for execution where they explicitly pull received data and execute non-preemptively till completion. Upon completion, tasks can then explicitly push data that is required by other tasks. Pull and push calls for the data are non-blocking that handover the data to underlying runtime system which handles the communication.

3.3 Runtime Implementation

The runtime is developed using MPI+OpenMP programming model and is capable of spawning 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 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 teams can be configured to balance the number of MPI processes and teams per node.

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 [54, 59] within each grid is used for load distribution among threads, where a number of tiles are assigned to each thread.

3.3.1 Communication Handlers

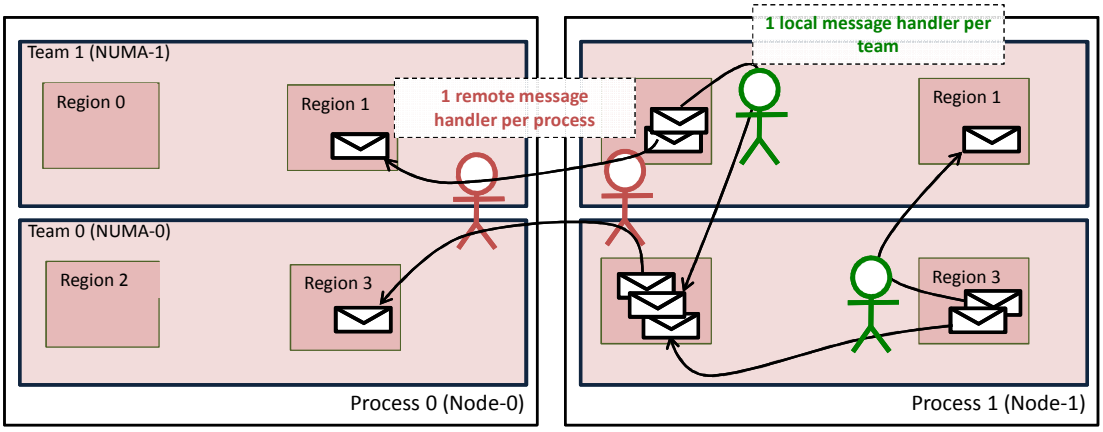


Figure 3.2: Local and Remote Communication/Message Handlers

The runtime consists of local and remote communication/message handlers that handle within and across MPI process communication, respectively. One off-node (remote) communication handler is created per MPI process while one on-node (local) communication handlers is created per team. Local communication handlers are responsible for messages between tasks from same or two different teams within an MPI process. Messages between tasks from two different MPI processes are handled

by the remote communication handlers. Figure 3.2 shows an example of two MPI processes each running on two remote nodes. Within a MPI process two teams each per NUMA node are created. Local and remote communication handlers are highlight with green and brown colors, respectively.

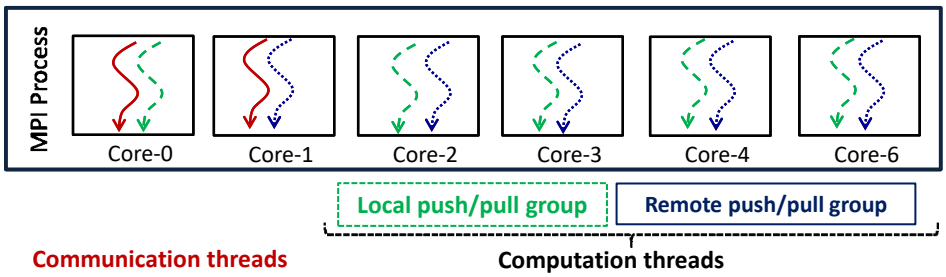
3.3.2 Message Queues and Shared Memory Protection

Besides communication among communication handlers, task schedulers also communicates with the communication handlers to handover the communication messages. The communication handlers uses FCFS queues for communication among themselves and with task scheduler. A pair of queues are maintain for each communication direction. For example, for 27-point nearest neighbor communication pattern, a task requires 26×2 queues both for incoming and outgoing communication. This may increase memory consumption a little but reduces lock contention as compared to an approach where a single queue is used for all tasks.

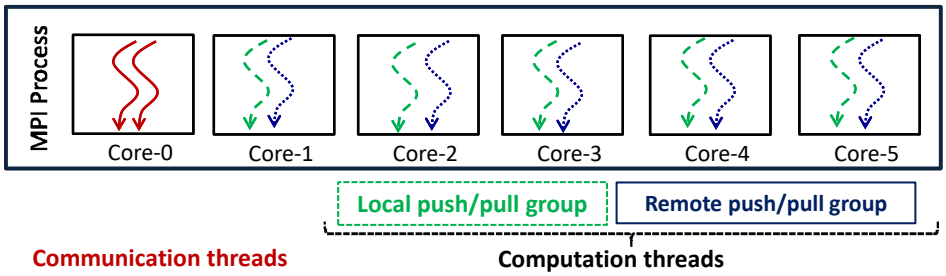
Locking is used to protect the memory that is shared between local communication handlers or between task schedulers and communication handlers. Locking can be applied at either team, task or queue level. When a task receive a message it will stall communication activities for all other tasks at the level of lock granularity. Thus choosing a right granularity for locking is important. Using separate lock for each queue will avoid lock contention but will result in too many lock and unlock operations. At team level, there will be less locking and unlocking operations but may result in lock contention as many tasks would require to acquire the lock. We use task level locking as it balances the two parameters. Moreover, we avoid using *omp_set_lock* as much possible as it blocks until lock is acquired and use *omp_test_lock* instead.

3.3.3 Thread-Work Assignment

All modern architectures provide multiple hardware threads per core and we utilize hyperthreading within a team in the runtime. A challenge posed by using hy-



(a) scatter affinity for both communication and compute threads



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

Figure 3.3: Thread-work assignment

perthreads 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.

User can provide a text file containing affinities for the communication and compute threads and the runtime pins threads according to specified affinities in the input file. An example thread distribution for the runtime using scatter and compact affinities for communication threads is shown in Fig-3.3a and Fig-3.3b, 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.

Chapter 4

AMR ALGORITHM

In this chapter, first we discusses the existing synchronous AMR algorithm for block structured AMRs. Then we present our proposed phase asynchronous AMR algorithm and explain its working with the help of an example. At the end, we discuss the integration of our proposed algorithm into the AMReX framework.

4.1 Synchronous AMR Algorithm

Algorithms 1 and 2 show the basic AMR algorithm described in [49]. 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.

Algorithm 1 Basic AMR algorithm

Procedure AMRTimeLoop(time t, num of steps s)
for $i \leftarrow 1$ to s **do**
    dt $\leftarrow$ compute_dt()
    AMRTimeStep(0, t, dt)
    $t \leftarrow t + dt$
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* $\leftarrow$ level *0*)
else
    fillBoundary($l \leftarrow l$ 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 $j \leftarrow 1$ **to** r **times:**
      AMRTimeStep (l + *1*, t, $\frac{j \times dt}{r}$)
end if
synchronizeData($l \leftarrow l$ + *1*)
End Procedure AMRTimeStep

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 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.

4.2 Phase 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.

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 subgrid are already met. As the dependencies for the task

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 $j \leftarrow 1$ **to** r **time:** AMRTimeStep (l + *1*, t, $\frac{j \times dt}{r}$, 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

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

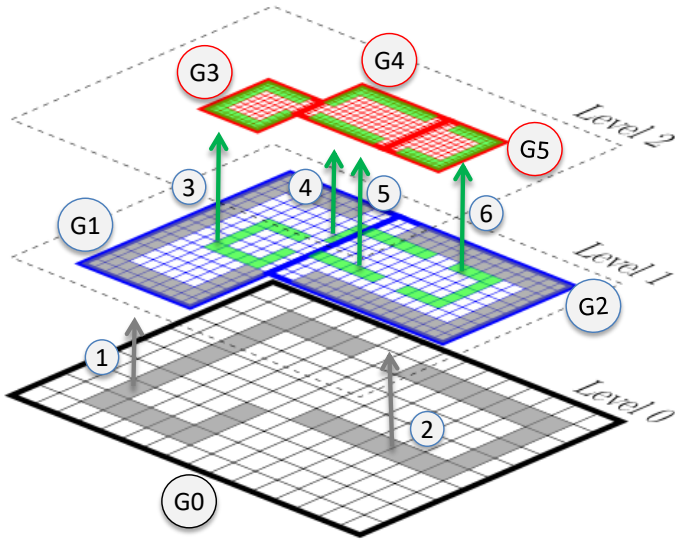


Figure 4.1: Asynchronous computation and communication overlap

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 4.1 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 rep-

resented 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.

4.3 Integration of Phase Asynchronous AMR Algorithm in AMReX

We integrated the asynchronous AMR algorithm in AMReX [9], which is a publicly available software framework used for implementing Block-Structured AMR applications. AMReX 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 AMReX 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 AMReX.

To implement the asynchronous execution of the AMR algorithm, we extended some of the AMReX functionalities. We added two more virtual functions *initMetaData* and *deleteMetaData* 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 AMReX 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 [47], 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.

Chapter 5

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 meta-data 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 meta-data extraction and graph iterator.

5.1 Meta-data Extraction

For meta-data 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

```
ExchangeGhost:
  tg = TaskGraph( myMultiFab )
  extractExchangeGhost(tg, myMultiFab, ...)
  parent → graphArray.push_back(tg)
Restriction:
  tg_src = TaskGraph( srcMultiFab )
  tg_dst = TaskGraph( dstMultiFab )
  extractRestriction(tg_src, tg_dst, srcMultiFab, dstMultiFab, ...)
  parent → graphArray.push_back(tg_src)
  parent → graphArray.push_back(tg_dst)
FillPatch:
  ap = AsyncFillPatch( myMultiFab, StateID, ...)
```

Figure 5.1: Asynchronous runtime's API for meta-data extraction. For clarity, the framework specific parameters are omitted.

meta-data 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:**

  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 meta-data for *ExchangeGhost*, programmers first need to create a task graph that will hold the meta-data as shown in the code in Fig-5.1. Meta-data 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.

- **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-5.1, two task graphs, one for each level, are created to hold the meta-data for *Restriction*. Meta-data is then extracted using the *extractRestriction()* subroutine, where the task graph holding the finer level meta-data serves as a source and task graph holding the coarser level meta-data 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.

- **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-5.1, 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 ID number. Constructor of the *AsyncFillPatch* class requires the ID number of the variable for which *FillPatch* is needed. All the task graph creation, meta-data extraction and pushing the task graphs to the *graphArray* are transparently handled inside the class constructor.

5.2 Task Graph Iterators

AMReX provides a *MultiFab iterator* that sequentially iterates over all the *Fab*s within a single *MultiFab*. Fig-5.2 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-5.3. 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-5.3 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.

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-5.3. 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 variable ID number, 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.

MultiFab Iterator:

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

Figure 5.2: AMReX's MultiFab iterator

Task Graph Iterator:

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

Figure 5.3: Asynchronous runtime's task graph iterator.

Like many other 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.

5.3 Impact on Productivity

A simple API is utmost important for productivity. Here, we show the significance of our simple API in reducing programming efforts with the help of an example code for an iterative solver.

```
graph = taskGraphCreate(tmp_MultiFab);      //create a graph with N tasks
#pragma omp parallel private(tid,taskID){  //create a thread pool for worker and
    communication threads
tid = omp_get_thread_num();                  //get thread id in a process
if(isCommunicationHandler(tid))              //communication threads
   serviceCommunication(graph,tid);
else{                                        //worker threads
   enableAllTasks(graph);                    //initially, all tasks are runnable
   do{
      taskID = getAnyRunableTask(graph);
      if(graph->task(taskID)->state(0) > 0)   //encode iteration number in state(0)
        call ExchangeGhost_receive(graph,taskID); //inject data from task's
    neighbors
      if(graph->task(taskID)->state(0) < nIters){
        compute(...);                         //execute on tiles of the task
        ExchangeGhost_send(graph,taskID);     //send data to task's neighbors
        disableTask(graph,taskID);            //explicitly yield control
        incrementState(graph->task(taskID)->state(0));} //increment the iteration
      else
        completeTask(graph,taskID);                    //complete a task
   }while(isEmpty(graph) == false);
   completeTaskGraph(graph,tid);                      //worker thread finishes
  }
} //End Parallel Region
taskGraphDestroy(graph);                          //destroy the graph
```

Figure 5.4: An iterative solver written under the asynchronous runtime

```
graph = taskGraphCreate(tmp_MultiFab)     //create a graph with N tasks
for(TGIter tgi(graph); tgi.isValid(); tgi++){
    ExchangeGhost_recive(tgi,regionID)         //inject data from region's
    neighbors
    compute(...);                              //execute on tiles of the region
    ExchangeGhost_send(tgi,regionID)   //send data to region's neighbors
}
taskGraphDestroy(graph);                          //destroy the graph
```

Figure 5.5: An iterative solver written under the proposed API

Figure 5.4 shows sample code to implement a simple iterative solver using the developed asynchronous runtime while Fig. 5.5 shows implementation using the proposed API. Difference between two sample codes is about 16 lines to implement a

single solver. Sample code in Fig. 5.4 will require additional code to use tiling while the proposed API do not require any additional code. The proposed API is simple and significantly reduces the programming effort to port legacy or develop new complex applications using the asynchronous runtime.

Chapter 6

CASE STUDY: CASTRO - AN ASTROPHYSICS CODE

To port an existing AMReX application to get benefit of the asynchronous execution, application programmers need to follow these steps: 1) Implement the *initMeta-Data()* subroutine to create task graphs and extract meta-data 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 meta-data is fairly straightforward and already explained in the API section 5.1. Here we will focus more on the control structure organization of CASTRO.

6.1 Control Structure (Synchronous)

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

$$\frac{\partial \vec{U}}{\partial t} + \nabla \cdot \vec{F} = \vec{S}. \tag{6.1}$$

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 multi-physics 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.

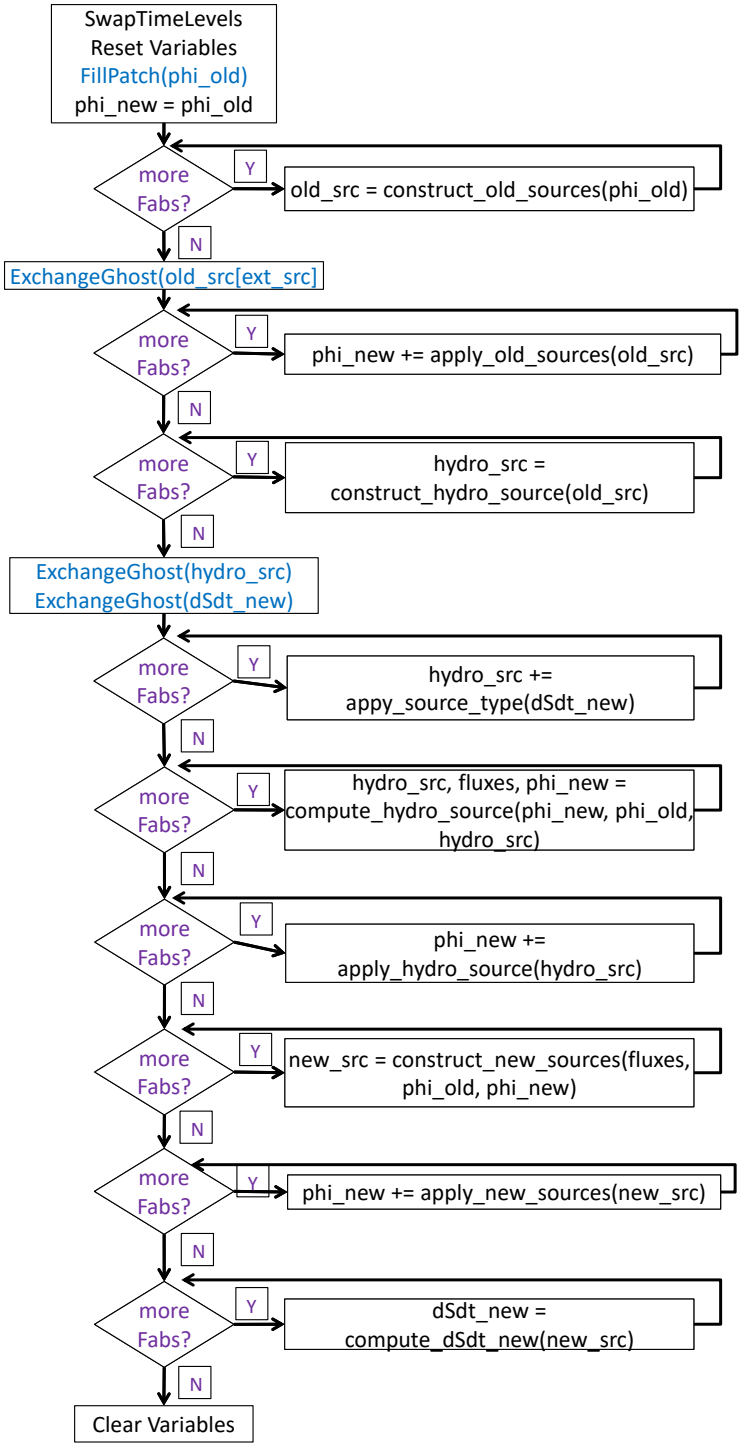


Figure 6.1: CASTRO's control structure (Synchronous)

Fig-6.1 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. 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*.

6.2 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-6.2. After swapping old and new pointers and resetting 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.

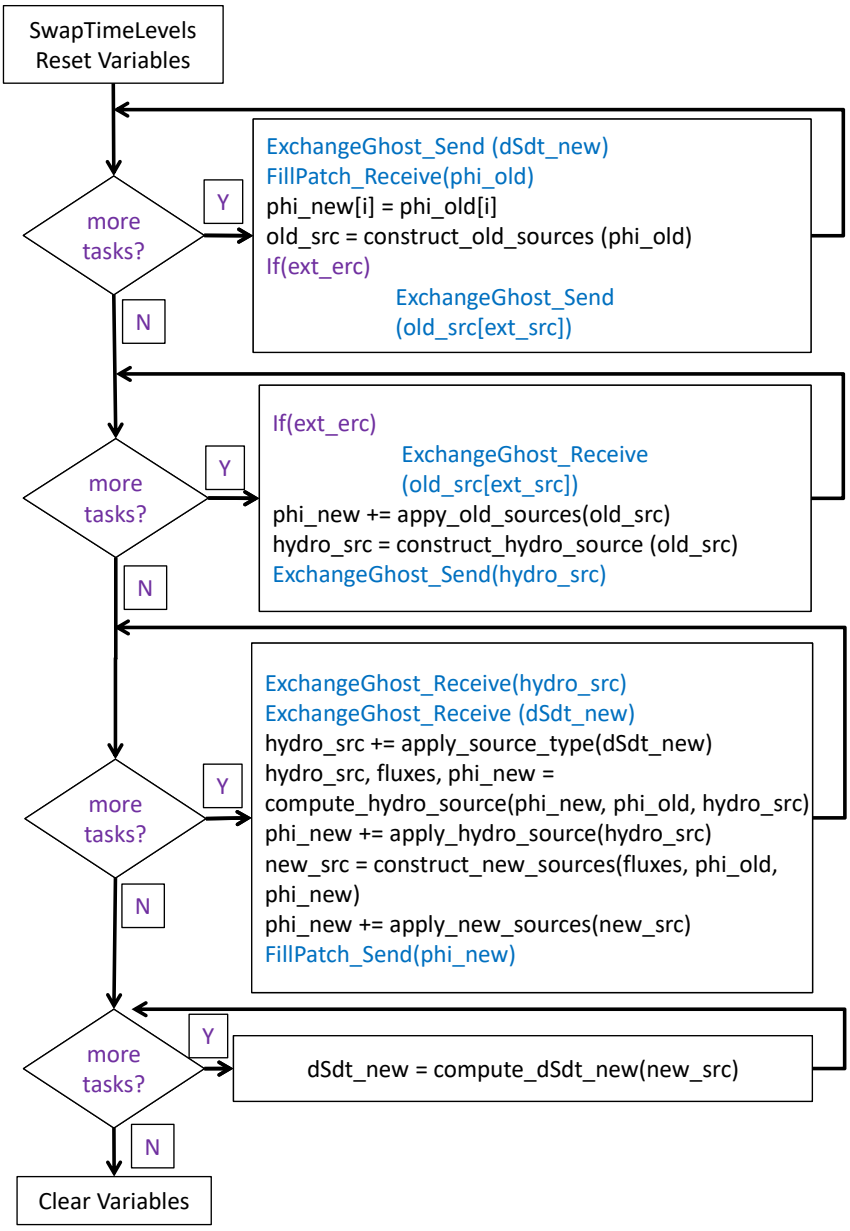


Figure 6.2: CASTRO's control structure (Asynchronous)

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.

6.3 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 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 meta-data 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.

Chapter 7

EXECUTION ON HETEROGENEOUS ARCHITECTURES

This chapter discusses design and implementation details about the runtime support for execution on heterogeneous architectures. Specifically, we discuss our solutions in supporting the following three challenging tasks 1) writing GPU kernel code, 2) CPU-GPU data movements and 3) scheduling computation on GPU. In the end, we also give an overview of how the runtime can be configured to support execution in a multi-GPU environment.

7.1 GPU Kernels

Existing programming models for GPU can be classified into two main categories. The first includes low-level programming models such as CUDA and OpenCL, which are extensions to C an Fortran languages. The second class includes directive-based models such as OpenACC and OpenMP. For writing GPU kernels, CUDA and OpenACC are the two mostly used programming models. CUDA is a programming language specific for Nvidia's GPUs. It is rather a low level language that requires high programming effort and results in high performance. On the other hand, OpenACC is a directive base programming model that is more productive and requires very low programming effort. Kernels generated using OpenACC may not have high performance as compared to CUDA but can be tuned to achieve a comparable performance. Application programmers are responsible for providing GPU kernels. For asynchronous execution using the runtime, OpenACC can be used to automatically generate GPU kernels therefore an explicit CUDA kernel implementation is not mandatory. However, the runtime provides a programmer the provision to write a manually tuned CUDA kernel for better performance.

7.2 Memory Model

Computation on GPU requires task's relevant data to be present in its global memory that is limited in capacity. Thus data needs to be moved back and forth between device and host memory that poses a major obstacle in performance optimization. Efforts both at hardware and software level are made to cope with the data transfer challenge. At hardware level, PCIe evolved from version 1.x with a bit rate of 2.5 GT/s per lane to version 4.x with 16 GT/s per lane and is expected to be 32 GT/s per lane in version 5.x. A parallel data transfer technology NVLink 1.0 and 2.0 achieves 20 GT/s and 25 GT/s per lane, respectively. At software level, explicit data allocation and transfers were required to be carried out by the programmers, initially. Unified virtual addressing was introduced to bring device and host memories under same address space and enable single pointer allocation. Followed by unified memory that added the automatic data migration support where data transfers are carried out either as a whole or page-wise depending on the underlying hardware support.

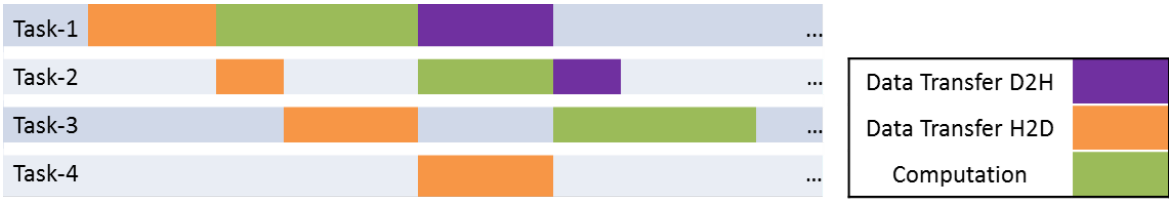


Figure 7.1: GPU computation overlap with data transfers between host and device

In literature[58, 57, 51, 35], there are two different data models used for GPUs both of them have their own advantages and disadvantages. First is GPU resident data model where data used by computation on GPUs is always resident on device memory. In this model, only the communication data (ghost cells) are copied to and from the host or other device as required. Although it put less pressure on the CPU-GPU link but problem with this models is the limited device memory. That makes it is less suitable for application that required huge data for computation because it will result in wastage of the computational resources. Second is host resident data model

where all data resides on the host and device memory is used as a cache. Problem with this model is the costly CPU-GPU data transfers. AMR applications being data intensive makes the GPU resident data model less viable option as it will lead to computational resources wastage because GPU memory is limited. For example, consider the fastest supercomputer Summit where ∼90% of compute power comes from GPUs but device to host memory ratio is ∼1:5.

For host resident data model, efficient CPU-GPU data transfers can be used to reduce the effect of costly communication. Efficient CPU-GPU communication can be achieved using asynchronous data transfer and overlap data copy of one task with computation of another task as shown in Fig. 7.1. Here after task 1 completes data transfer its computation is overlapped with data transfers of task 2 & 3. Similarly task 2 computation and task 4 data transfers are overlapped. This can be implemented manually or can be achieved automatically with the help of unified memory and prefetching. Runtime can be opt for either manual or automatic option without putting any burden on the application programmers. Here, we opt for automatic (managed memory) option because of two reasons. One, it will result in a low maintenance runtime as any changes in the hardware architecture will not require modification to the runtime. Second, hardware venders are working hard to make the automatic data transfers efficient e.g. paging with underlying hardware support.

7.3 Work Scheduling on Heterogeneous Architectures

CPUs and GPUs are the two widely used architectures in today's supercomputers. CPUs contains tens of fat cores suitable for control flow and latency sensitive applications. On the other hand, GPUs have thousands of lightweight cores designed for massively data parallel application to achieve high throughput. Parallelism on thousands of GPU cores is handled through a hierarchy of thread groups offered by both CUDA and OpenACC. In the runtime, work on both CPU and GPU is scheduled through CPU threads where work on CPU is scheduled as normal while a single CPU thread is enough to schedule and launch computation on GPU.

Application programmers can explicitly turn on GPU support in the proposed runtime. If the GPU support is turned on then the runtime will schedule tasks for computation on both CPU and GPU. Figure 7.2 shows scheduling of work on both CPU and GPU. The runtime dedicates one of the worker thread on CPU for launching tasks on GPU while rest of the worker threads perform computation on CPU. The dedicated thread for GPU and CPU worker threads executes independently until all tasks are computed. Computation on CPU is distributed among worker threads through tiling where a few tiles within the subgrid are assigned to each thread. Similarly all these threads take part in pushing and pulling of communication data from the runtime. However, in case of GPU, the single GPU worker thread lunches computation of the entire subgrid on GPU and the same thread is responsible for pushing and pulling the communication data.

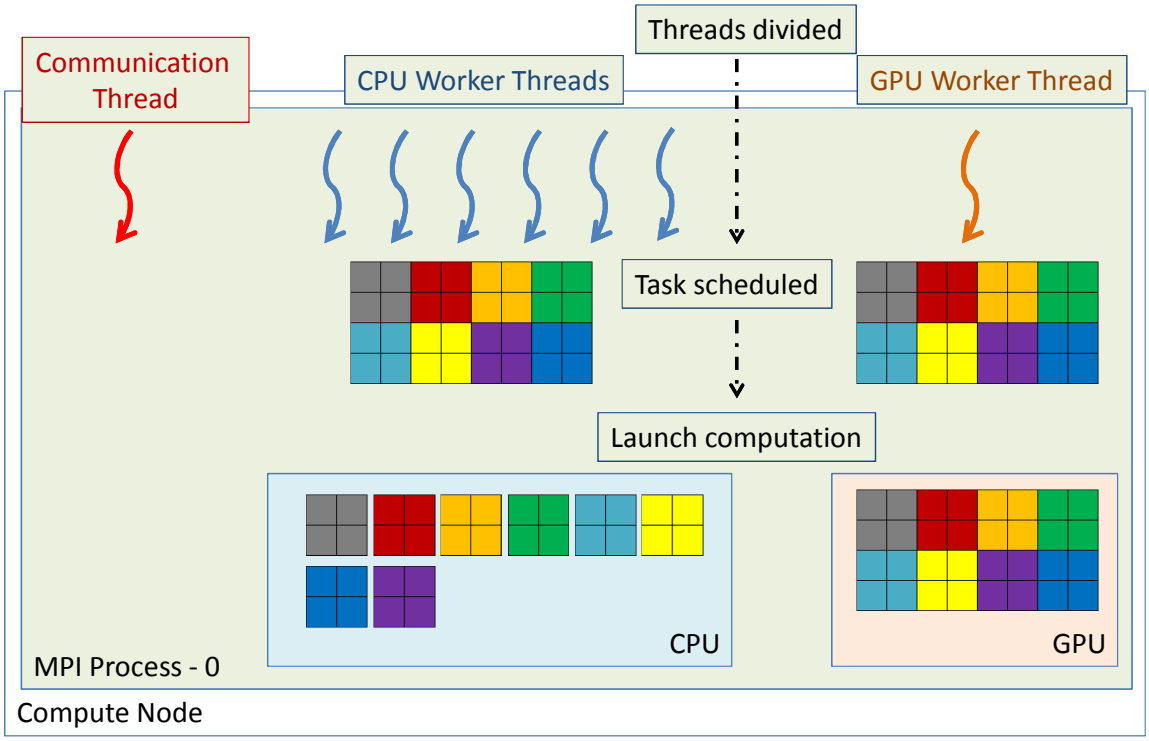


Figure 7.2: Work scheduling on both CPU and GPU

Scheduling computation while keeping balanced load distribution among CPUs

and GPUs within a process is the most crucial and difficult task due to difference in their computational powers. Load imbalance can result in inefficient use of computational units when some units are waiting idle while some units are busy computing. The proposed phase asynchronous AMR algorithm is based on an out-of-order task iterator. Thus, predicting a task being ready for scheduling is not possible and is dependent on the completion of its communication at runtime. For an optimal load distribution, ready tasks are kept in a single queue instead of pre-distributing them among CPU and GPU. Thus, whenever CPU or GPU finishes a task it can schedule the next task for computation from the ready queue. This will guarantee that a computational unit won't stay idle as long as there are ready tasks to be scheduled.

Task Graph Iterator:

```
for(TGIter tgi( tg ); tgi.isValid(); ++tgi){
  ExchangeGhost_receive(tgi, myMultiFab, ...)
  if(tgi.gpuTask)
    //compute_on_gpu(tgi)
  else
    //compute_on_cpu(tgi)
  ExchangeGhost_send(tgi, myMultiFab, ...)
}
```

Figure 7.3: Asynchronous runtime's task graph iterator with support for heterogeneous architectures.

7.4 API

Figure 7.3 shows the API to support execution on heterogeneous architectures. A member variable of the task graph iterator named *gpuTask* is set if the current task is scheduled on GPU. Programmers can use the said variable to check whether the current task is scheduled on CPU or GPU so that they can call the respective kernel. Relevant communication subroutines for CPU or GPU will be called automatically by the framework without any intervention from the application programmer. This interface enables an incremental implementation for GPUs as we are using unified

memory. The application programmer can provide GPU kernels either implemented using OpenACC or CUDA. Similarly, framework developers can implement communication subroutines for GPU and can modify them to increase performance without putting any extra burden on the application programmers.

7.5 Multi-GPU Support

In terms of the number of GPUs and CPU cores, machine configuration can vary widely. Our runtime is flexible and can be easily configured to execute application on different machine configurations. The runtime detects the number of GPU devices on a node and binds one device with each MPI process that can use it exclusively if number of devices are equal to the number of processes per node. If the number of GPUs per node is less than number of processes per node then it enables Multi Process Service (MPS) support so that a GPU can be shared by more than one MPI processes. The current implementation does not support multiple GPUs within a single process. If there are fewer MPI processes than GPUs, some of the GPUs will not be used.

Chapter 8

EVALUATION

We use two applications to evaluate the proposed phase asynchronous AMR algorithm and runtime system. The first one is a proxy AMR application namely *advection solver*, which helps test the key functionalities that used in every AMReX application. The second is CASTRO, a large production code widely used in Astrophysics. We run these codes on three supercomputers Hazel Hen, Cori-Haswell and Cori-KNL. Machine specifications for all three super computers are shown in Table 8.1. This chapter also gives some insight into the effects of asynchronous execution on load balancing requirements.

	Hazel Hen	**Cori-Haswell**	**Cori-KNL**
CPUs	Intel Xeon E5-2680 v3 (Haswell)	Intel Xeon E5-2698 v3 (Haswell)	Intel Xeon Phi 7250 (Knights Landing)
Sockets	2		1
Cores/socket	12	16	68
Threads/core	2		4
Clock Rate	2.5 GHz	2.3 GHz	1.4 GHz
LLC	30 MB	40 MB	16 GB
Main memory	128 GB		96 GB
Memory bandwidth	68 GB/s		102 GB/s
Interconnect	Cray Aries Dragonfly		

Table 8.1: Machine specifications

8.1 Performance Studies using CASTRO

To carryout performance studies using CASTRO we used Cori supercomputer located at Lawrence Berkeley National Laboratory, CA, USA. We experiment with the Intel's Haswell architecture and with the Intel's Xeon Phi (Knights Landing) configured as quadrant cluster mode and MCDRAM as the last level cache. Node specifications for both machines are provided in Table 8.1.

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 these test problems is given in Section 2.5. All experiments use two subcycling iterations and a refinement ratio of two. We use two refinement levels and a higher mesh size for R-T while three refinement levels and slightly lower mesh size for the S-T test problem.

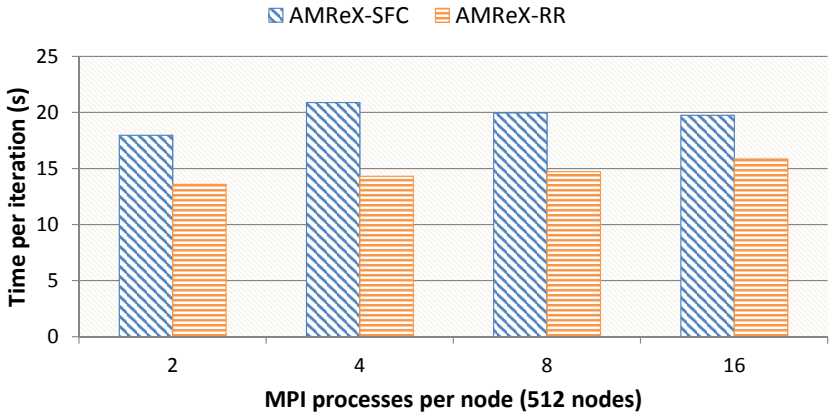
	Cori-Haswell	**Cori-KNL**
Mesh size for R-T (level 0)	$1024 \times 1024 \times 2048$	
Mesh size for S-T (level 0)	$1024 \times 1024 \times 1024$	
Fab size	64^3	
Tile size	$64 \times 8 \times 2$	$64 \times 4 \times 2$
MPI processes per node	2	4
OpenMP threads per MPI process	32	68

Table 8.2: Input configurations

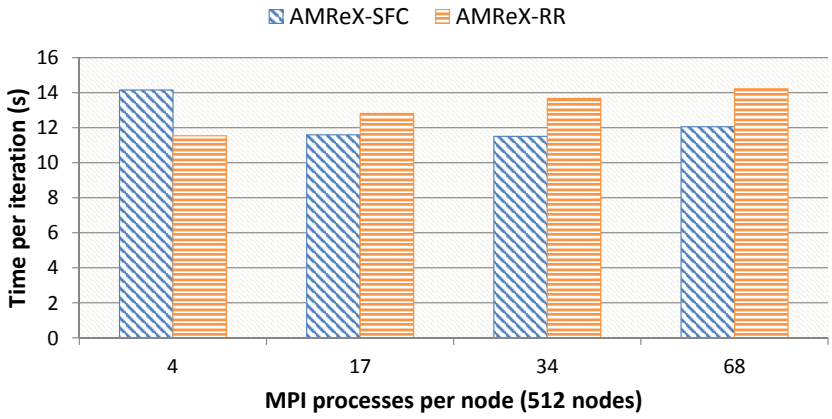
8.1.1 Input Configuration Selection

Table 8.2 lists the input configurations used for all the experiments. For a fair comparison, we selected these 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 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.



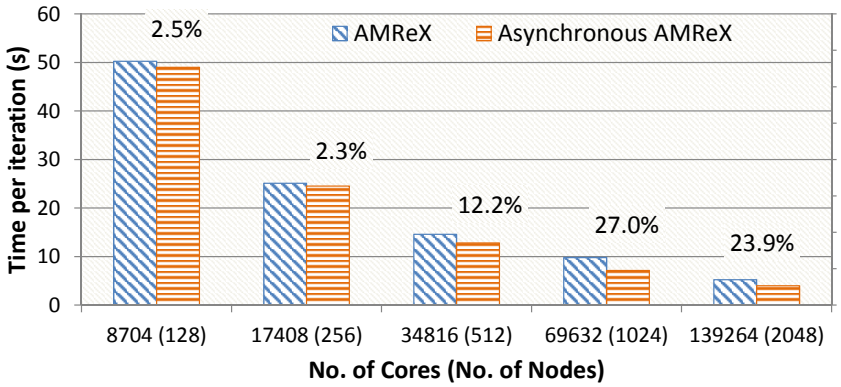
(a) Cori-Haswell



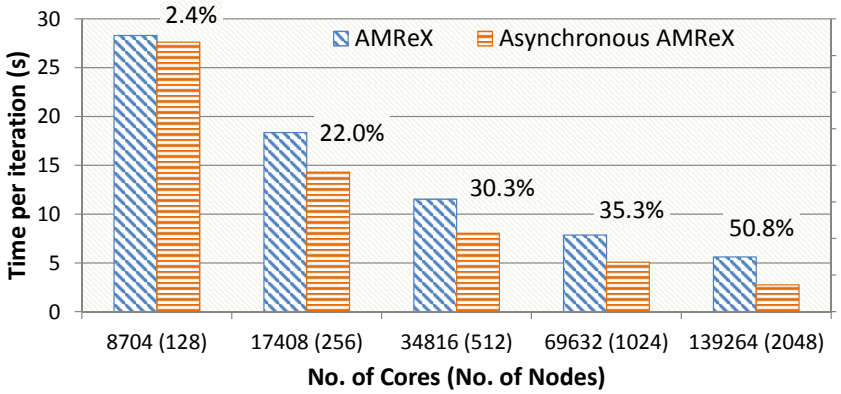
(b) Cori-KNL

Figure 8.1: Comparison of different number of MPI processes per node on (a) Cori-Haswell using Rayleigh-Taylor and (b) Cori-KNL using Sedov-Taylor. (Figure is newly added to support our argument for using 2 and 4 MPI processes per node on Cori-Haswell and Cori-KNL respectively.)

To select optimal number of MPI processes per node as listed in Table 8.1 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. 8.1(a) Cori-Haswell and Fig. 8.1(b) Cori-KNL.



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



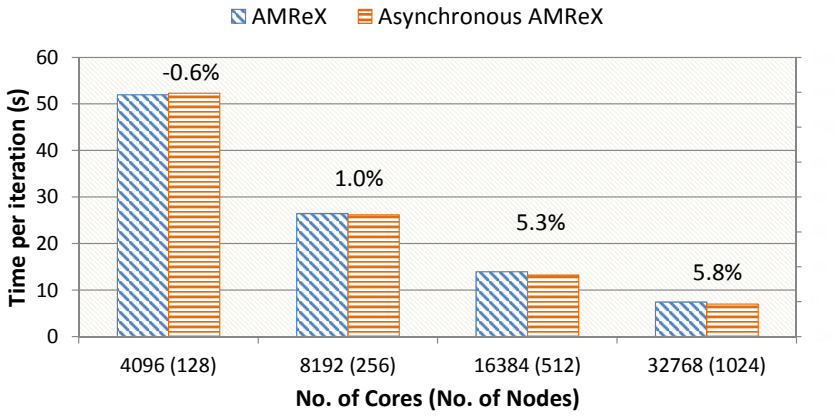
(b) Strong scaling on Cori-KNL (Sedov-Taylor)

Figure 8.2: 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.

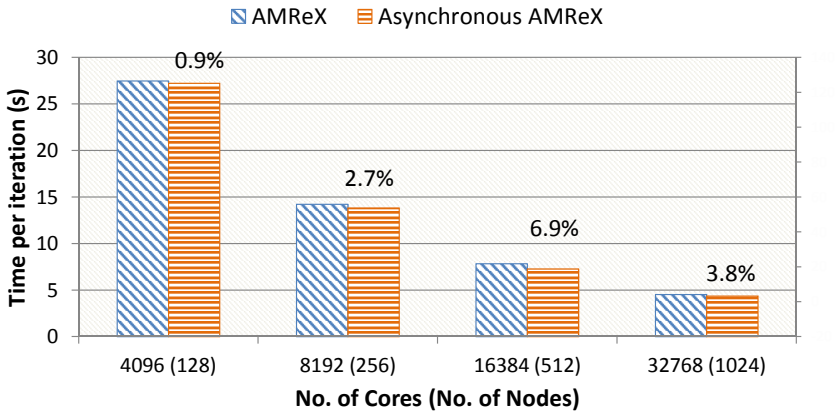
Cori-Haswell has 2 NUMA nodes and we use Cori-KNL in quadrant cluster mode resulting in 4 NUMA nodes. Creating more than one MPI process per node helps to use the network bandwidth effectively and provides NUMA-aware data locality. We found that using a single MPI process per NUMA node along with RR load balancing strategy produces the best results.

8.1.2 Strong Scaling Studies

Fig-8.2a and Fig-8.2b show strong scaling results for R-T and S-T test problems on Cori-KNL while Fig-8.3a and Fig-8.3b show strong scaling results for the R-T and



(a) Rayleigh-Taylor



(b) Sedov-Taylor

Figure 8.3: Strong scaling on Cori-Haswell 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.

S-T test problems on Cori-Haswell. Asynchronous AMReX outperforms the original 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 further 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.

8.1.3 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.

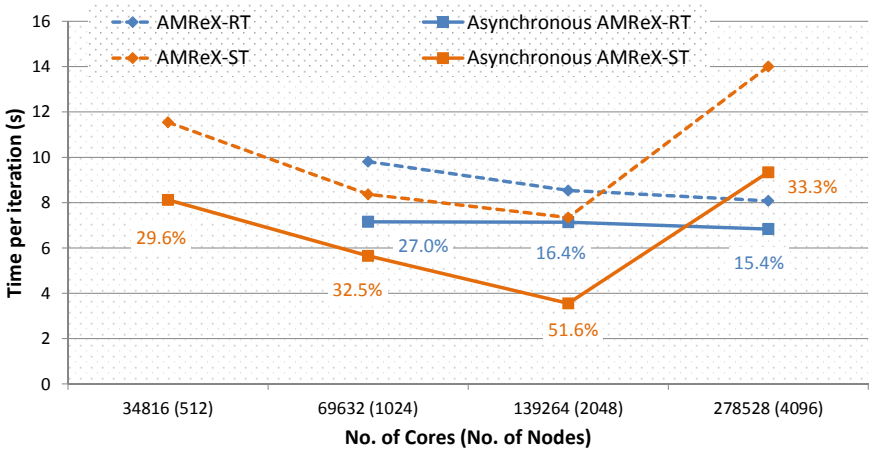


Figure 8.4: Performance at large scale on Cori-KNL for the test problems R-T in blue and S-T in orange color.

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-8.4 shows results for both R-T and S-T test problems. Asynchronous AMReX scales similar to AMReX and maintains significant performance improvement over AMReX. However, it can be seen that the performance gap slightly reduces or increases 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.

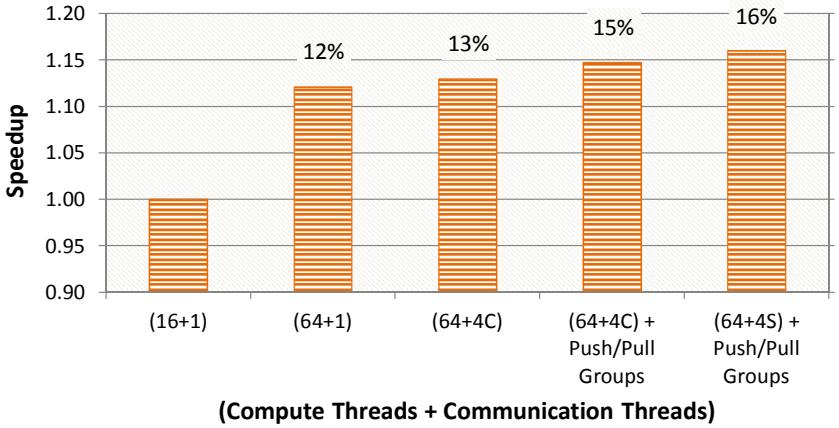


Figure 8.5: Different degrees of hyperthreading support in asynchronous AMReX (1024 Cori-KNL nodes) using configurations in Table 8.2 for the R-T test problem. C and S in the x-axis labels represent Compact and Scatter affinity.

8.1.4 Hyperthreading

As discussed in Section 5, in addition to using hyperthreads for compute, our runtime can use hyperthreads for communication threads. 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-8.5. Hyperthreading for communication contributes to overall 4% performance improvement in total. While dividing commu-

nication work among multiple threads gives about 1%, 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. Although performing these operations in parallel does not give any drastic performance improvement, it increases asynchrony and prevents these operations from becoming a bottleneck. The hyperthreading for communication results on Cori-Haswell are very similar to Cori-KNL, thus omitted.

8.2 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 [27]. 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.

8.2.1 Comparison of 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-8.6 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-8.6 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.

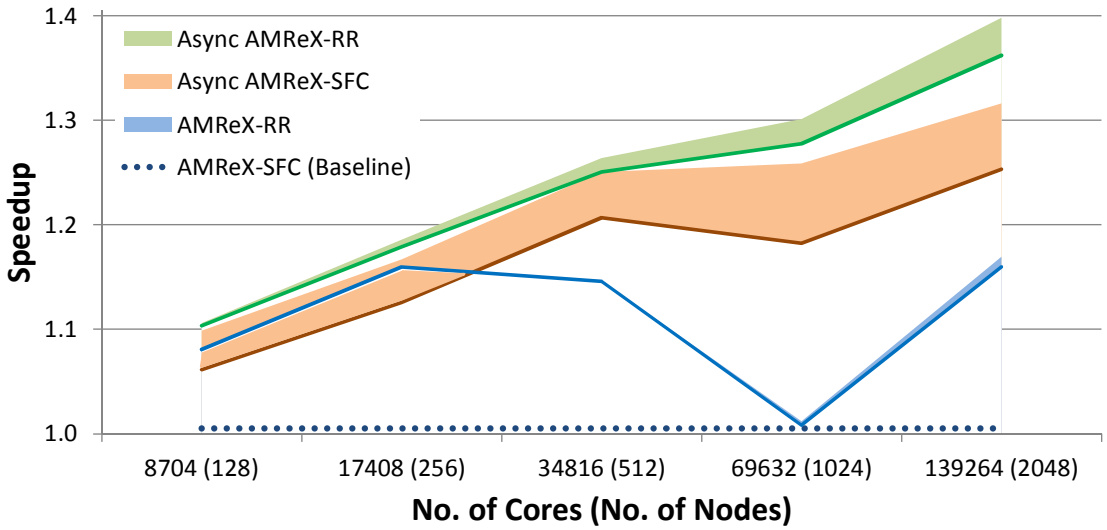


Figure 8.6: SFC vs RR comparison in a strong scaling study on Cori-KNL using AMReX-SFC as the baseline for Rayleigh-Taylor problem. Lower and upper bounds indicates speedup corresponding to the maximum execution time and to the mean execution time of all MPI processes, respectively.

The performance gap between the maximum and the mean time per process for the asynchronous runtime is higher as seen in Fig-8.6. 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 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.

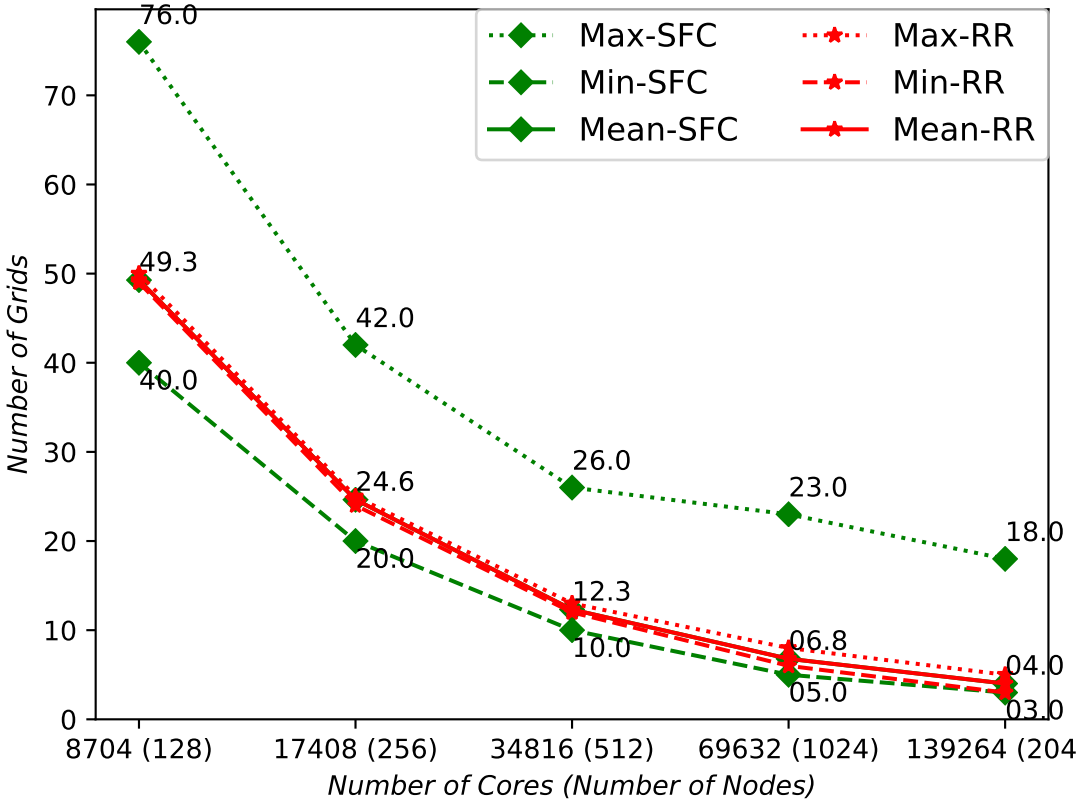


Figure 8.7: Comparing SFC vs RR in terms of number of grids (Fabs) per process (Rayleigh-Taylor)

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-

8.7. 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.

8.3 Performance using Heat Advection Solver

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 this study, we integrated the phase asynchronous AMR algorithm inside BoxLib and use the existing Boxlib's synchronous execution model as our baseline for comparison. 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 8.1. All the experiments were performed using three levels of refinement, two subcycling iterations and a refinement ratio of 2.

8.3.1 Strong Scaling Studies

Figure 8.8 shows strong scaling up to 12K cores where each bar is labeled with percent improvement obtained by the proposed asynchronous algorithm over BoxLib. We used 1024^3 grid size as input for strong scaling studies. The y-axis shows the time spent in a single step. It does not include the time spent in timestep *dt* computation and

global reduction. Proposed asynchronous algorithm achieves up to 28.6% speedup over BoxLib on 1536 cores. Speedup decreases as we further increase the number of cores because the number of grids 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 speedup case there are about 95 grids/rank while it reduces to less than 12 grids/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.

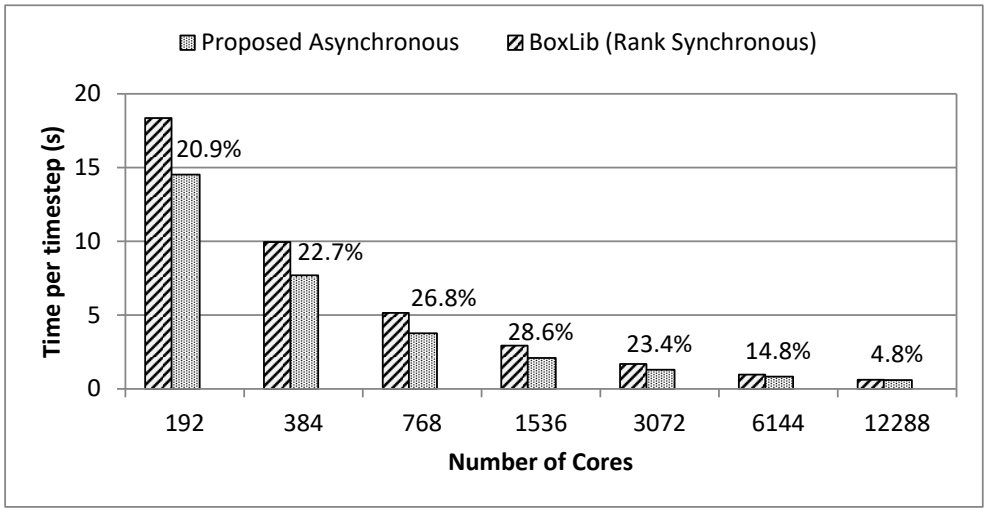


Figure 8.8: Strong scaling for advection code on Hazel Hen

8.3.2 Weak Scaling Studies

Figure 8.9 compares weak scaling for BoxLib and proposed asynchronous algorithm. Grid size starts from 1024^3 for 768 cores and then doubled in x, y and z direction 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 grids per process to hide communication.

A break down of the time spent in computation, restriction and prolongation for

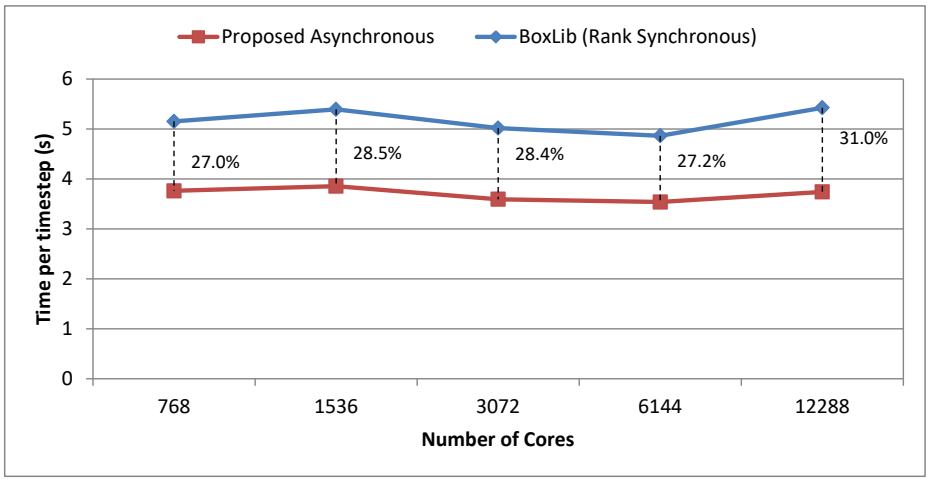


Figure 8.9: Weak scaling for advection code on Hazel Hen

rank synchronous compared to the proposed asynchronous is shown in Fig. 8.10. We can only overlap prolongation and computation because while performing restriction there is no computation to overlap with. Thus in an ideal case, we can completely overlap prolongation with the computation. The proposed asynchronous bar is labeled with the percentage performance achieved of the maximum achievable performance. The proposed asynchronous algorithm hides about 80% of the prolongation behind the computation.

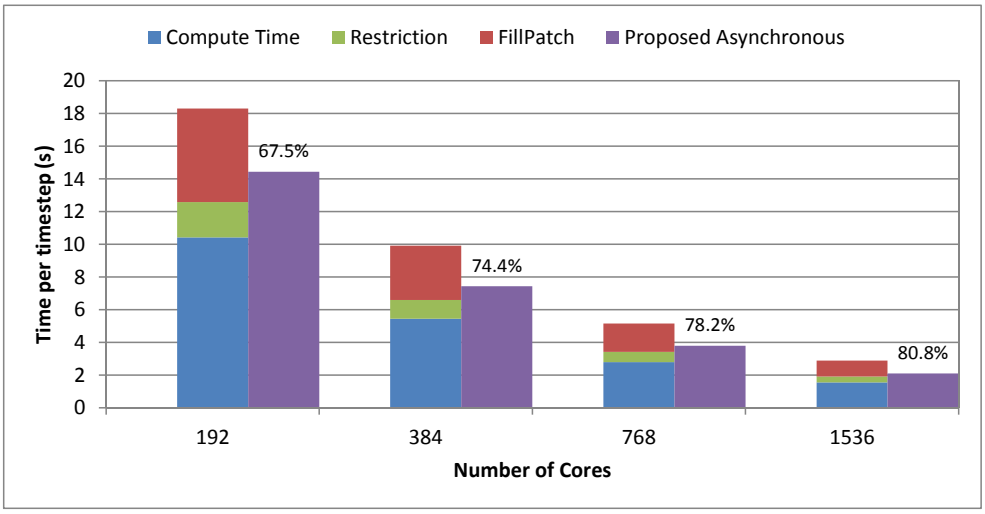


Figure 8.10: Breakdown of performance achieved on Hazel Hen

8.4 Performance Analysis on Heterogeneous Architectures

For performance analysis, we ran the heat advection solver on SummitDev located at Oak Ridge National Laboratory, Tennessee, USA. Machine specifications of SummitDev are shown in Fig. 8.3. Computational kernels of the advection solver for execution on GPU were implemented using OpenACC. Input configurations are shown in Table 8.4.

CPUs	IBM POWER8
Sockets	2
Cores/socket	10
Threads/core	8
Clock Rate	3.5 GHz
Total Main memory	256 GB
CPU Memory bandwidth	180 GB/s
Interconnect	Mellanox EDR InfiniBand
GPUs	NVIDIA Tesla P100
GPUs/node	4
GPU Peak DP Flops	5 TF
GPU Memory size	16 GB
GPU Memory bandwidth	520 GB/s
CPU-GPU Interconnect	NVLink 1.0 (80 GB/s)

Table 8.3: Machine specifications for SummitDev

Here we compare 6 application code variants 1) CPU 2) GPU-Manual 3) GPU-Managed 4) Mix-Manual 5) Mix-Managed 6) Mix-Managed-Prefetching, where for work scheduling we use 3 modes CPU, GPU and Mix. In CPU mode all computation is scheduled on CPU, GPU mode schedules all computation on GPU and Mix mode simultaneously schedules computation on both CPU and GPU. CPU-GPU data transfers are performed either manually or through managed memory. In Manual, all

Refinement levels	2
Subcycling iterations	2
Refinement ratio	2
Mesh size for R-T (level 0)	$256 \times 256 \times 256$
Fab size	128^3
Tile size	$128 \times 16 \times 6$
MPI processes per node	4
OpenMP threads per MPI process	10
OpenMP threads per core	2

Table 8.4: Input configurations for heat advection solver on SummitDev

the required data between CPU-GPU are explicitly handled through data directive of OpenACC that initiate transfers on first touch either by host or device. In Managed memory, data transfers are handled automatically by the underlying CUDA runtime that transfer a page between CPU and GPU memories using page fault mechanism. Both in GPU-Manual and GPU-Managed a single thread is used to launch computation on GPU while all the worker threads participates in pushing and pulling of communication data. In Mix mode a single threads is used for work scheduling and pushing and pulling communication data. In Mix-Managed-Prefetching, we add prefetching hints to start CPU-GPU data transfers beforehand. For performance comparison, we increase the amount of computation by recomputing a subgrid twice (2x) and four times (4x) when it is scheduled for computation.

Figure 8.11 compares the performance for all implementations on a single compute node. Heat advection solver being a test application contains very low computation thus is not suitable for GPU. We use the corresponding CPU x-compute within each x-compute group as a baseline for that group. As shown in the figure the performance of GPU increases as we increase the amount of computation and Mix-Manual perform best among all implementations. Managed memory could not beat the manually

tunned data transfers but performance of managed data transfer was improved with prefetching. We also note that the share of computation performed by GPU in Mix modes increase as the GPU performance is improved. For Mix-Manual, GPU share raises from 52% in 1x-compute to 62% in 4x-compute.

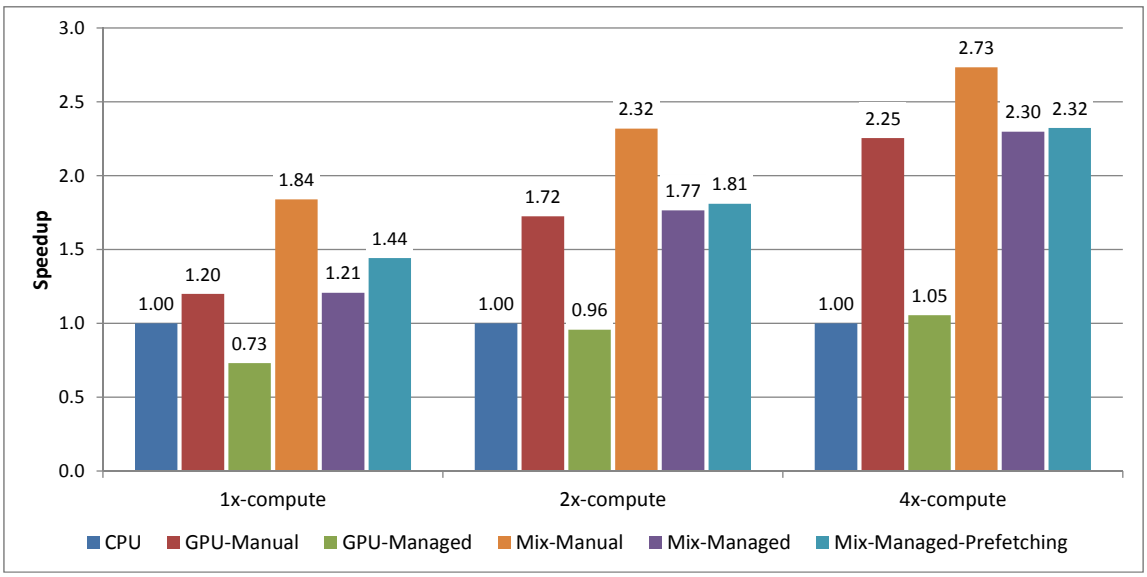


Figure 8.11: Comparison of different implementations and amount of computation on SummitDev.

The main objective of adding support for the heterogeneous architecture in the runtime is to make use of all computational resources effectively. As evidenced by the results, the runtime successfully schedules work among different types of computational resources and balances the load based on their performance.

Chapter 9

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 [36, 50, 24, 13, 56, 25] 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 [33, 34, 21]. To reduce programming cost further, there are compiler based tools [46, 44] 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.

Next, we present an overview of existing asynchronous AMR frameworks and compare them with our work. We also include discussions on some existing general purpose asynchronous programming models and runtime systems.

9.1 Asynchronous AMR Frameworks

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 [30], Chombo [23], Enzo [18], FLASH [28], and Paramesh [43]. Wahib et al. [58] presented a compiler-based framework named *Daino* that generates parallel AMR code optimized for GPUs from an annotated uniform grid code. A number of AMR frameworks [43, 30, 23] exist which support parallel computation in AMR applications. To the best of our

knowledge, only a few [5, 41] support communication and computation overlap.

Langer et al. [41] organize the oct-tree based adaptive mesh as a collection of parallel objects distributed over virtual processors and use the Charm++ [37] 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 [5, 45] 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 bulk-synchronous AMR applications to their framework requires major rewriting of the applications in Uintah's task-based programming model. We introduce a user-friendly API, especially the grid iterator, that significantly reduces programming effort. We eliminate the need to discover the necessary communication at runtime by leveraging communication meta-data embedded in an AMR program. We also make use of such domain-specific knowledge for optimizing internal components of our runtime system.

9.2 Asynchronous Programming Models and Runtime Systems

There are a number of task-based programming models and runtime systems such as Cilk[16], Legion [12], StarPU[11], SWARM[42], Charm [37], DaGue [17] 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. Next, we briefly discuss few of these programming models.

9.2.1 Cilk

Cilk[16] is a C-based runtime system and one of the first that introduced task-based parallelism. A Cilk program is a collection of procedures where each procedure is a sequence of threads. Cilk threads are nonblocking, which means once invoked these threads will run until its execution is completed. Threads can spawn child threads that start a child procedure. Child threads may run in parallel with its parent. As threads are nonblocking it does not wait for a return value. Return values from child procedures can be received by spawning a successor thread. In Cilk a program is viewed as a Directed Acyclic Graph (DAG) which expands dynamically. Each node in the DAG represents a Cilk thread and edges represent data dependencies from one thread to another.

Cilk also implements a work stealing task scheduler in which an idle processor (the thief) can steal work from another randomly selected processor (the victim). The thief steals work from the shallowest level of the ready pool of the victim because threads at the shallowest level of a ready pool are always on the critical path of the DAG. Thus stealing the work from critical path results in reducing the runtime.

Cilk is a general purpose, low-level runtime and requires a lot of programming efforts for application development. However, Cilk's DAG to represent task is adopted by a number of programming runtimes and is also adopted in the proposed runtime as well.

9.2.2 Legion

Legion[12] is a parallel programming system based on using *logical regions*. Regions are passed to distinguished functions called *tasks*. Data dependency information is specified by the user. Legion uses software out-of-order processor (SOOP) to schedule a stream of tasks dynamically. An instance of the SOOP is executed on every processor. To hide memory and communication latencies Legion defers execution of tasks that are waiting for resources. Legion dynamically detects and enforces data dependencies between tasks for correctness.

Legion is also a general purpose runtime and require a notable programming effort for application development. To overcome the productivity challenge, Legion introduces Regent [52], 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 slowdown and it is not straightforward to convert legacy codes into Regent.

9.2.3 StarPU

StarPU[11] is a runtime system that schedules tasks within a shared memory node on a set of heterogeneous processing units. Once dependencies are met, StarPU scheduler can schedule a task to one of the many possible processing units if the implementation is provided by the user for that task. It is the programmer's responsibility to explicitly specify all dependencies for a task. Tasks are submitted asynchronously and its completion is marked by a callback. This allows an application to submit multiple tasks at a time that can also comprise tasks which depends on others. Several different scheduling strategies are implemented in StarPU.

Architectures with heterogeneous processing units also contain a variety of memories and interconnection networks. If not handled properly, moving data over these networks between different memories can raise performance issues. StarPU implements a Software DSM (Distributed Shared Memory) that replicates data over memory nodes and uses a lazy approach for memory consistency to save bandwidth. MSI (Modified/Shared/Invalid) protocol is used for memory consistency. Runtime supports asynchronous data transfers to overlap it with computation.

StarPU mainly focuses on scheduling tasks on accelerator-based multicores and lack optimizations for manycore architectures. The programmer has to specify the task dependencies so increasing programming overhead as well.

9.2.4 SWARM

SWARM[42] is a codelet-based runtime where a codelet is the fundamental unit of work. Codelets executes on hardware components (typically a thread) called locale. Locales are organized into a tree-like data structure that is based on their communication characteristics. A memory allocator and a scheduler is associated with each locale. Locales can manage its own and descendants' memory allocation.

Only leaf schedulers can execute a codelet. Non-leaf schedulers either buffer the requests or send them to their descendants. Idle schedulers can steal work from other neighboring schedulers in the hierarchy to respect locality. It can also put the hardware component in low-power state when there is no work. Similarly, if an allocation request cannot be fulfilled the allocator can pass to request to its parent allocator. The parent allocator, in turn, can pass the allocation request to his parent or descendant.

To deal with heterogeneous architectures precise device description is also associated with each leaf locale. When there exist multiple codelet implementations the scheduler uses locale's component description to select appropriate scheduled codelet implementation. Codelet scheduled on a non-leaf node of the locale-tree can be scheduled on any compatible descendant. Codelet-based runtime is under active development but not yet fully implemented.

Chapter 10

CONCLUSION

This thesis presents the great potential of a phase asynchronous execution model for adaptive mesh refinement applications. The asynchronous execution model is based on our asynchronous AMR algorithm that allows a subgrid within an AMR 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.

We developed a runtime system that powers the asynchronous AMR algorithm and helps to design a simplified user interface. The runtime is based on directed acyclic graph where nodes are subgrids and edges, are their communication dependencies. The runtime automatically extracts these communication dependencies from the AMR framework to reduce the burden on application programmers. Our runtime also utilizes all the hardware threads to achieve maximum parallelism and asynchrony. Runtime dedicates a thread for communication and computation to enable their overlap. Work assignment of the computation threads is further managed according to different phases of execution for better performance.

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. 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. The load should be balanced both in terms of number of cells per process and number of grids per process.

We also demonstrated the capability of our proposed model to support heterogeneous architectures. We identified the challenges associated with heterogeneity i.e programming effort, data handling between memories and work scheduling. We discussed how advancements in hardware technologies, modern compilers and support in our runtime can be used to deal with these challenges. We used heat advection solver to compare the performance of different implementations and showed that our proposed execution model can fully utilize all computational resources.

BIBLIOGRAPHY

[1] BoxLib: An AMR software framework. https://ccse.lbl.gov/boxlib/.

[2] MPI Reference Manual, Available: http://mpi-forum.org/.

[3] OpenMP Specifications, Available: http://openmp.org/.

[4] TOP500 Supercomputer Sites : https://www.top500.org/statistics/list/.

[5] The Uintah framework: a unified heterogeneous task scheduling and runtime system. In 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pages 2441–2448. IEEE, nov 2012.

[6] A. S. Almgren, J. B. Bell, C. A. Rendleman, and M. Zingale. Low Mach Number Modeling of Type Ia Supernovae. I. Hydrodynamics. The Astrophysical Journal, 637(2):922–936, 2006.

[7] Ann Almgren, John B. Bell, M.J. Lijewski, Z. Lukic, and E. Van Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. Astrophysical Journal, 765:39, 2013.

[8] Ann S. Almgren, Vincent E. Beckner, John B. Bell, Marcus Day, Louis H. Howell, C. C. Joggerst, M. J. Lijewski, Andy Nonaka, M. Singer, and Michael Zingale. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. Astrophysical Journal, 715:1221–1238, June 2010.

[9] AMReX: Block-structured AMR framework. https://ccse.lbl.gov/amrex/index.html.

[10] J.A. Ang, R.F. Barrett, R.E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S.D. Hammond, K.S. Hemmert, S.M. Kelly, H. Le, V.J. Leung, D.R. Resnick,

A.F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N.J. Wright. Abstract Machine Models and Proxy Architectures for Exascale Computing. In 2014 Hardware-Software Co-Design for High Performance Computing, pages 25–32. IEEE, nov 2014.

[11] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. page 33, 2010.

[12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 66:1—-66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS'06, pages 84–84, Washington, DC, USA, 2006. IEEE Computer Society.

[14] John B. Bell, Marcus Day, and M.J. Lijewski. Simulation of nitrogen emissions in a premixed hydrogen flame stabilized on a low swirl burner. Proceedings of the Combustion Institute, 34(1):1173–1182, 2013.

[15] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics, 53(3):484–512, 1984.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded

Runtime System. Journal of Parallel and Distributed Computing, 37(1):55–69, 1996.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. Parallel Computing, 38(1-2):37–51, 2012.

[18] Greg L. Bryan, Michael L. Norman, Brian W. O'Shea, Tom Abel, John H. Wise, Matthew J. Turk, Daniel R. Reynolds, David C. Collins, Peng Wang, Samuel W. Skillman, Britton Smith, Robert P. Harkness, James Bordner, Ji-hoon Kim, Michael Kuhlen, Hao Xu, Nathan Goldbaum, Cameron Hummels, Alexei G. Kritsuk, Elizabeth Tasker, Stephen Skory, Christine M. Simpson, Oliver Hahn, Jeffrey S. Oishi, Geoffrey C. So, Fen Zhao, Renyue Cen, and Yuan Li. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. The Astrophysical Journal Supplement Series, 211(2):19, apr 2014.

[19] Adam Burrows, Jason Nordhaus, Ann Almgren, and John Bell. The potential role of spatial dimension in the neutrino-driving mechanism of core-collapse supernova explosions. Computer Physics Communications, 182(9):1764 – 1766, 2011. Computer Physics Communications Special Edition for Conference on Computational Physics Trondheim, Norway, June 23-26, 2010.

[20] Cy P. Chan, John D. Bachan, Joseph P. Kenny, Jeremiah J. Wilke, Vincent E. Beckner, Ann S. Almgren, and John 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, COM-HPC '16, pages 17–28, Piscataway, NJ, USA, 2016. IEEE Press.

[21] Ernie Chan, Robert van de Geijn, and Andrew 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, SPAA '10, pages 200–208, New York, NY, USA, 2010. ACM.

[22] Ke-Jung Chen, Stan Woosley, Alexander Heger, Ann Almgren, and Daniel J. Whalen. Two-dimensional simulations of pulsational pair-instability supernovae. The Astrophysical Journal, 792(1):28, 2014.

[23] Phillip Colella, Daniel T. Graves, J. N. Johnson, Hans S. Johansen, N. D. Keen, Terry J. Ligocki, Daniel F. Martin, Peter W. Mccorquodale, D. Modiano, Peter O. Schwartz, T. D. Sternberg, and Brian Van Straalen. Chombo software package for AMR applications design document. Technical report, LBNL, 2003.

[24] Anthony Danalis, Ki Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05, volume 2005, pages 58–58. IEEE, 2005.

[25] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience, 15(9):803–820, 2003.

[26] Matthew Emmett, Weiqun Zhang, and John B. Bell. High-order algorithms for compressible reacting flow with complex chemistry. Combustion Theory and Modelling, 18(3):361–387, 2014.

[27] Muhammed Nufail Farooqi, Didem Unat, Tan Nguyen, Weiqun Zhang, Ann S. Almgren, and John 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, pages 682–694, 2017.

[28] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series, 131(1):273–334, Nov 2000.

[29] D. Geer. Chip makers turn to multicore processors. Computer, 38(5):11–13, May 2005.

[30] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. The cactus framework and toolkit: Design and applications. In Proceedings of the 5th Intl Conference on High Performance Computing for Computational Science, VECPAR'02, pages 197–227, Berlin, Heidelberg, 2003. Springer-Verlag.

[31] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. Loop Transformation Recipes for Code Generation and Auto-Tuning. In Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing, pages 50–64. Springer-Verlag, 2010.

[32] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09, page 147, New York, New York, USA, 2009. ACM Press.

[33] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. Optimizing a Conjugate Gradient Solver with Non Blocking Operators. Parallel Computing, 33(9):374–382, 2007. Selected Papers from EuroPVM/MPI 2006.

[34] Torsten Hoefler, Jeffrey M. Squyres, Wolfgang Rehm, and Andrew 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), volume 4331 LNCS, pages 155–164. Springer, Berlin, Heidelberg, 2006.

[35] Alan Humphrey, Qingyu Meng, Martin Berzins, and Todd Harman. Radiation modeling using the Uintah heterogeneous cpu/gpu runtime system. In Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond, XSEDE '12, pages 4:1–4:8, New York, NY, USA, 2012. ACM.

[36] Benjamin F. Jamroz and Robert 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, 9(8):2881–2892, aug 2016.

[37] Laxmikant V. Kale and Sanjeev 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, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[38] Max P. Katz, Michael Zingale, Alan C. Calder, F. Douglas Swesty, Ann S. Almgren, and Weiqun Zhang. White dwarf mergers on adaptive meshes. i. methodology and code verification. The Astrophysical Journal, 819(2):94, 2016.

[39] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07, page 1, New York, New York, USA, 2007. ACM Press.

[40] Monica D. Lam, Edward E. Rothberg, Michael E. Wolf, Monica D. Lam, Edward E. Rothberg, Michael E. Wolf, Monica D. Lam, Edward E. Rothberg, Michael E. Wolf, Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings

of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV, volume 19, pages 63–74, New York, New York, USA, 1991. ACM Press.

[41] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo Chuan Pan, Laxmikant V. Kale, and Paul Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In Proceedings - Symposium on Computer Architecture and High Performance Computing, pages 100–107. IEEE, Oct 2012.

[42] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing. In Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era - EXADAPT '12, pages 21–26, New York, New York, USA, 2012. ACM Press.

[43] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. Computer Physics Communications, 126(3):330 – 354, 2000.

[44] Sergio M. Martin, Marsha J. Berger, and Scott B. Baden. Toucan A Translator for Communication Tolerant MPI Applications. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 998–1007. IEEE, May 2017.

[45] Qingyu Meng, Justin Luitjens, and Martin Berzins. Dynamic task scheduling for the Uintah framework. In Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on, pages 1–10. IEEE, 2010.

[46] Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan, and Scott Baden. Automatic translation of mpi source into a latency-tolerant, data-driven form. J. Parallel Distrib. Comput., 106(C):1–13, August 2017.

[47] Tan Nguyen, Didem Unat, Weiqun Zhang, Ann Almgren, Nufail Farooqi, and John 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, SC '16, pages 81:1–81:12, Piscataway, NJ, USA, 2016. IEEE Press.

[48] James. Reinders and James. Intel threading building blocks : outfitting C++ for multi-core processor parallelism. O'Reilly, 2007.

[49] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. Computing and Visualization in Science, 3(3):147–157, August 2000.

[50] José Carlos Sancho and Darren 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), volume 5168 LNCS, pages 688–697, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[51] H Y Schive, Y C Tsai, and T Chiueh. Gamer: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. Astrophysical Journal, Supplement Series, 186(2):457–484, 2010.

[52] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex 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, SC '15, pages 81:1–81:12, New York, NY, USA, 2015. ACM.

[53] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu.

Knights Landing: Second-Generation Intel Xeon Phi Product. IEEE Micro, 36(2):34–46, mar 2016.

[54] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John 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, pages 116–135. Springer International Publishing, 2016.

[55] Didem Unat, John Shalf, Torsten Hoefler, Thomas Schulthess, Anshu Dubey (Editors), et al. Programming Abstractions for Data Locality. Technical report, 04 2014.

[56] T.Q. Viet and Tsutomu Yoshinaga. Improving Linpack Performance on SMP Clusters with Asynchronous MPI Programming. IPSJ Digital Courier, 2:598–606, 2006.

[57] Mohamed Wahib and Naoya Maruayama. Data-centric gpu-based adaptive mesh refinement. In Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, IA3 '15, pages 3:1–3:7, New York, NY, USA, 2015. ACM.

[58] Mohamed Wahib, Naoya Maruyama, and Takayuki Aoki. Daino: A high-level framework for parallel and efficient AMR on GPUs. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, pages 53:1–53:12, Piscataway, NJ, USA, 2016. IEEE Press.

[59] Weiqun Zhang, Ann Almgren, Marcus Day, Tan Nguyen, John Shalf, and Didem Unat. Boxlib with tiling: An adaptive mesh refinement software framework. SIAM Journal on Scientific Computing, 38(5):S156–S172, 2016.

[60] Weiqun Zhang, Louis H. Howell, Ann S. Almgren, Adam Burrows, and John B. Bell. Castro: A new compressible astrophysical solver. ii. gray radiation hydrodynamics. Astrophysical Journal Supplement, 196:20, 2011.

[61] Weiqun Zhang, Louis H. Howell, Ann S. Almgren, Adam Burrows, Joshua C. Dolence, and John B. Bell. Castro: A new compressible astrophysical solver. iii. multigroup radiation hydrodynamics. Astrophysical Journal Supplement, 204:7, 2013.