Thesis Proposal:
Fully Asynchronous Execution of Structured Adaptive Mesh Refinement Simulations

Philip Miller

December 17, 2013

1 Executive Summary

Technical Abstract

Adaptive mesh refinement is a critical technique for the simulation of partial differential equations. It is used across the full breadth of physical science and engineering domains. By focusing numerical resolution where a given problem demands it, AMR enables simulation studies that would otherwise be infeasible. AMR codes are especially common on large parallel systems on which users are limited in the amount of machine time they are allowed to assume.

The complexity of implementing efficient parallel AMR applications has led to the development of many frameworks to support them. These frameworks implement common elements providing the computational structure of AMR and often include related numerical methods as well. Since the early development of AMR codes, many common challenges have been subjected to intensive study by computational researchers.

One particular challenge that spans many elements of how AMR is implemented in distributed memory is the pervasiveness of globally synchronizing operations. These synchronizations come in many different forms: exchanges of cell data between nearby boxes, timestep length computation, mesh structure modification and dissemination, and calls to solver libraries for implicit timestepping and steady-state computation. Every such synchronization during execution imposes a cost on overall performance.

The obvious cost of operations that synchronize across an entire parallel machine is the time taken by the operations themselves. As machines grow larger, these operations take longer to execute. Frequently, these operations are used to communicate per-processor information, adding another factor of increased time. Eventually, that time can come to dominate the application's overall performance, in an Amdahl-like bottleneck.

However, the more critical cost of synchronous operation is less transparent. Every point of synchronization is an opportunity for load imbalance and critical path delays to negatively impact performance. Furthermore, frequent synchronization limits the window in which scheduling algorithms (including mapping, load balancing, and prioritization schemes) can observe and address these conditions. Even worse, despite having less time and information to work with, they are responsible for simultaneously optimizing within each independent synchronization window.

Existing research on AMR frameworks includes isolated attempts at reducing or mitigating particular synchronization costs. However, none of them has addressed the questions of *what would be necessary to attain a fully asynchronous implementation of AMR*, or *what overall impact that could have on application performance*. My thesis will address these questions, with an aim to provide efficient strong and weak scaling to a broad class of AMR applications.

To accomplish this, I will analyze, adapt, and improve upon existing parallel algorithms and implementation techniques. Where necessary, I will develop new methods that are suitable for localized, asynchronous execution. The guiding

methodology will be to enable each box that appears over the course of a simulation to make independent forward progress in its execution, pausing only to satisfy its own local dependences.

As I develop these techniques, I will apply a broad range of analysis methods and runtime mechanisms to optimize execution performance. By observing the performance attainable with each successive relaxation of synchronization requirements, I will develop further insights into the relative costs and benefits of parallel algorithms with varying degrees of synchronization.

Intellectual Merit

This thesis will further the development of parallel algorithms whose interactions are localized and whose execution is not constrained to the bulk synchronous model. This development is critical both to extend the reach of current parallel applications to future systems, and to guide the design of new applications.

Additionally, this work will test the limits of existing programming models for asynchronous parallel computing. In doing so, it will potentially drive further theoretical and practical advances in the infrastructural software used for asynchronous parallel programming. In particular, this work will place new demands on techniques for identification and location of parallel entities in distributed memory systems.

Expected Impact

The proposed work will have substantial impact across many parts of the research community:

- Users of AMR PDE simulations will experience faster execution, enabling shorter time-to-solution and more intensive simulations. These benefits can cascade to research downstream of those simulation studies, and to the broader society drawing upon these research results.
- Developers of software systems underlying AMR simulations will be presented with new opportunities for optimization, exposed by the additional freedom provided by asynchronous execution and the ideas it inspires.
- Research on numerical methods that enable the use of 'stale' data in computations will have stronger motivations and grounding in the computational capabilities developed in this thesis and the performance results that they demonstrate.
- Parallel computing researchers will have additional evidence for the value of asynchronous algorithms and new techniques for their development, analysis, and elucidation.

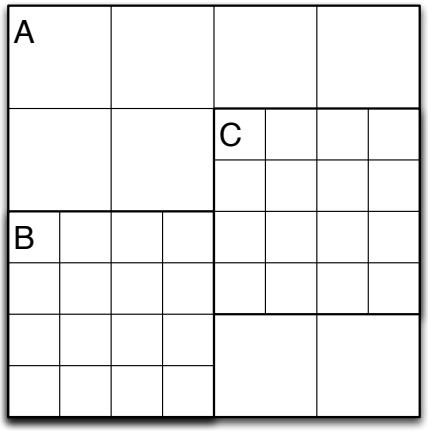

Figure 1: A simple two-level AMR structure. A is a coarse block overlapping refined blocks B and C.

2 Per-Box Asynchronous Execution

During the execution of each timestep in an AMR code, every box may communicate several times with its neighbors, boxes on neighboring levels of its hierarchy with which it overlaps, and boxes on other hierarchies that provide it with additional data or require its data for local computation. Historically, AMR codes have been structured to perform entire communication operations, for all boxes in a set, in a synchronous fashion [1, 2, 3, 4]. Several AMR codes appearing in the past year have evolved to exploit the fact that each individual box may receive all of the data that it is waiting on well before the processor hosting it finishes all communication in a given operation. Once a particular box's private dependencies are satisfied, the processor is free to perform subsequent local computations on that box.

The existing work in this area provides preliminary evidence that it is possible to run particular AMR applications with efficient strong scaling. Work that I contributed to demonstrated a 2D tree-structured code built in Charm++ [5] running a simulation of an advection-diffusion problem. With minimal runtime optimization, this code was able to strong-scale to 2k ranks of a Cray XE6 system and 32k ranks of an IBM Blue Gene/Q system [6]. Further work on this code using a more efficient load balancing algorithm developed by Harshitha Menon improved the scaling results to 128k ranks on BG/Q [7]. We are currently preparing an article describing the adaptation of this code to 3D, further performance optimizations, and more extensive scaling studies for publication. This code also serves as a test bed for other work in progress on asynchronous domain decomposition (§ 4.1).

The Uintah framework focuses on problems related to deflagration and detonation of combustible and explosive materials in 2 and 3 dimensions. For their

primary fluid-structure interaction application, MPMICE, Meng et al. have shown strong scaling to several hundred thousand cores of Titan and Mira with efficiencies of 68% and 76%, respectively [8]. The other applications shown in that paper do not exhibit the same degree of scalability or do not present strong-scaling results at all. Within the bounds of each very-intensive timestep ($\approx$ 1.5s per step at the full scale of Titan), they perform detailed load and timing prediction to approximately schedule the complete set of computational tasks to be performed during that step [9, 10]. They report that scheduling consumes an approximately constant amount of time, which at present scales represents about 10% of execution time [11]. Their asynchronous runtime mechanism then allows some tasks that are ready 'early' to execute out of turn. The detailed data they show indicates that most tasks are nevertheless performed fairly close to the sequence in which they were planned [8, Figure 3].

The Octopus framework implements octree-structured AMR for hydrodynamics applications arising in astrophysics, using the HPX runtime system [12]. They use a pull-based model in which each block identifies the neighbors from which it expects data for the next several timesteps, and passes a 'future' object to that neighbor to be filled in when the data is ready. They strong scale the code to 4k cores of a cluster with good efficiency. However, the benchmark results may not be representative, because the benchmark presented never actually attempted to regrid and the code currently suffers sequential performance issues that may hinder interpretation of their results[1].

2.1 Proposed Work

For this element of my thesis work, I plan to adapt the Chombo framework to fully asynchronous execution. Chombo has many characteristics that make it a good target for this study. From a motivational standpoint, Chombo is general-purpose, in the sense that it provides a proven computational and numerical toolkit on which a wide range of applications have successfully been built (e.g. [13, 14, 15, 16, 17]). I will use several of these applications to evaluate the success of my work (§ 5). Chombo is also actively developed by an open collaborative team that is eager to support this effort.

I will use the Charm++ parallel programming system as a substrate for this work. It provides a rich ecosystem of infrastructure and tools for the implementation of dynamic and asynchronous parallel applications. In particular, its facilities for programming variable collections of migratable parallel objects is well suited to the work at hand. As a core developer of Charm++ for the past several years, I am confident both that it meets the basic needs of this project and that I am well equipped to address any problems that arise in its use.

Technologically, Chombo poses a mix of opportunities and challenges. The opportunities present in Chombo's structure are that it is highly modular and its various component algorithms are all implemented in terms of a few well-chosen primitives. At present, Chombo is implemented in a processor-oriented

[1] Personal communication with lead author Bryce Adelstein-Lelbach

SPMD fashion. All data and work units are distributed across the full scope of the system. Algorithms are expressed in a bulk synchronous form, in which processors alternately compute and communicate in distinct operations.

The modularity is expressed in a form common to many applications and libraries following the bulk synchronous model: every processor makes the same subroutine calls in a common sequence at about the same time. Thus, even though there are no explicit barriers when crossing module boundaries, any such call is a point of soft global synchronization. To achieve fully asynchronous execution, I will have to ensure that work from multiple modules can be fully and freely interleaved at run time.

The key primitive that Chombo uses to implement both its own numerical algorithms and higher-level application code are 'data iterators' that expose the application data in processor-local blocks within a particular collection in an arbitrary order. This encapsulated design was chosen specifically with future needs for asynchronous and data-parallel execution in mind.

As implemented presently, data iterators can easily enable asynchronous execution of single computational operations at a time, relative to a dependence on one or more preceding communication operations. However, due to control flow limitations of the existing SPMD code, they can not enable fully asynchronous execution, in which each block makes independent progress through a longer sequence of communication and computational operations. In order to achieve this, existing code will have to be refactored or transformed to a new structure.

As a preliminary milestone, I will demonstrate the effect of single-operation asynchrony. This can be expected to dramatically reduce the impact of any remote communication latencies, because they will have substantial computation with which to overlap. Once communication latency concerns are mitigated, load balancing algorithms can safely be re-tuned from their current locality-favoring heuristics, such as partitioned space filling curves. Instead, they can focus on truly achieving load uniformity, limiting wasted time at subsequent synchronization points.

By implementing this, the Chombo team and their application users will receive immediate benefit from my work, long before the project is complete. A practical challenge in this vein is that I also expect these experiments to show limitations in grain size that Chombo developers have not previously addressed. With a working design already in hand, I expect to have publishable results from this improvement in the next couple months.

To answer the questions posed by my thesis about fully asynchronous execution, I will have to convert at least the iterator instances traversed by my benchmark applications to a different form. The Chombo code base includes approximately 1000 static appearances of the token `DataIterator` – enough that manual inspection and modification would be inappropriate, though not truly 'infeasible.' Thus, I will develop a partially or fully automatic technique to adapt this interface to enable fully asynchronous execution. A promising path for implementing such a transformation would be to use the ROSE compiler toolkit [18], with which I have prior experience.

It is worth noting that in an asynchronous execution environment, meth-

ods that advance the computation using 'stale' data become more appealing to implement [19, 20]. Within the framework to be developed in my proposed thesis work, such methods would potentially be applicable. However, I intend to maintain a focus on techniques for introducing asynchronous execution without a corresponding degradation of numerical algorithmic efficiency [21].

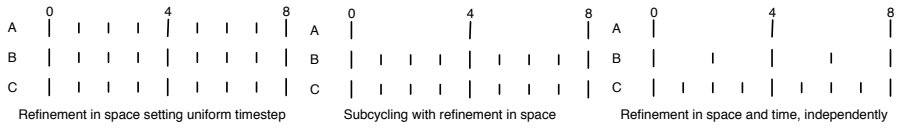

Figure 2: Potential timestep structures for coarse block A and fine blocks B and C in a neighborhood, as in figure 1. Tick marks indicate points in simulated time to which each block will incrementally update its contents. Where two neighboring blocks have ticks in common, they must exchange data. Where one block has a tick and a neighbor does not, the sub-cycling block must wait to receive the other block's subsequent values and interpolate in time before updating.

3 Timestep calculation

In simulations of hyperbolic systems, the length of timestep taken at any point in the domain must respect the CFL condition to generate a correct solution. As a side effect, this means that even the most rapidly propagating phenomenon has an upper bound on how many points of the domain it can reach in each time step. We can exploit this numerical bound to eliminate the need to operate with a global 'consensus' timestep length determined by a collective operation.

In existing structured mesh simulations, there is some regular interval at which every part of the domain calculates how long a timestep it can take, and then a global collective operation is performed to ensure that every processor proceeds at the consensus minimum step length. In AMR codes, this calculation may happen across all levels [3] (figure 2 left), or it may happen within each level that updates on a subcycle basis [22, 23] (figure 2 center). In either form, this presents a very frequent form of strong synchronization across the bulk of the computation. The Octopus framework has demonstrated some benefit from overlapping this reduction in a lagged fashion, and shown that for their particular problems this does not negatively impact solutions [12]. However, a lagged reduction is not proven to be a principled, generally-applicable solution – there is currently no formal bound on the error it can introduce. Where it hinders accurate solution, it also introduces an additional tradeoff between algorithmic efficiency and implementation efficiency.

As a safety valve against a lagged reduction leading to wrong results, one could build a check-and-reexecute solution. Two analogies come to mind for this approach: fault tolerance and optimistic parallel discrete event simulation [24]. Explicitly implementing an AMR framework in either fashion is unappealing for a few reasons. It introduces substantial additional implementation complexity. It requires additional memory to store sufficient data to reexecute correctly from as far back as when the faulty timestep first occurred. Finally, it still demands global coordination through some sort of collective, that could ultimately induce synchronization effects.

3.1 Proposed Work

Suppose, instead, that every block took its would-be contribution to that reduction, and shared it with its neighbors in the form of a power-of-2 step length factor that it intends to use during the next step. This could piggy-back on exchanges of ghost cell data, for instance. Neighboring blocks could immediately calculate when they will next need to synchronize to exchange the next set of ghost cells, in an integer number of steps. Depending on the nature of the simulation, blocks hearing that a neighbor must take shorter steps could also take that shorter step, or continue with its current step for one cycle and shorten to the minimum of its neighbors' lengths at the next cycle. A resulting timestep structure is shown in the right of figure 2.

This new method introduces an additional coarse-fine interface in time between blocks that share the same spatial resolution. At that interface, the blocks taking shorter steps will have to interpolate the value of the ghost cells owned by blocks running coarser time steps. This additional interpolation should not negatively impact the accuracy of the simulation, relative to subcycling in general, because similar interpolation in time is already occurring at the spatial coarse-fine boundaries at which values must be interpolated in space as well. From a different perspective, this could be viewed as a step in the direction of the Tent-Pitcher method for Spacetime Discontinuous Galerkin [25], in which each mesh point takes the longest step it is individually allowed, but one largely retaining the mathematical and consequently computational regularity of structured AMR both in space and time discretization.

As an added potential benefit of this method, the simulation as a whole can compute fewer overall point updates without loss of accuracy. Observe that in the traditional method, as soon as the timestep shrinks anywhere on a given level, it must shrink everywhere on that level. Thus, all points on a given level are updated as many times as the most critical points. In the new method, the shortened timestep propagates gradually outward from the points demanding the shortest steps. Thus, points residing in boxes far from the critical points will take fewer, longer timesteps to reach the time when the new bound reaches them. Effectively, where subcycling previously limited the number of operations performed on entire levels that didn't need them, we now subcycle at a resolution of individual decomposed blocks within each level.

One question left open in this discussion is how, once timesteps have been shortened by a given factor, a block can tell that it is again safe to take longer time steps. The converging bounds principles of the new algorithm described in 4.1 may also be applicable here. I may have to resolve this question in the course of implementing the new method. In the course of execution, the finest resolution blocks most affected by a moving phenomenon that demands a short timestep may be regenerated and recalculate their necessary timestep afresh from local neighborhood conditions.

4 Domain Decomposition

Domain decomposition algorithms for AMR are generally responsible for answering three key questions that broadly characterize the subsequent simulation:

1. What boxes will exist at what resolution?
2. How do new boxes relate to preceding boxes and each other?
3. When executing in parallel, what processor will own each box?

In answering these questions, designers of such algorithms have faced a plethora of demands on their operation and output:

1. Sufficient resolution over all parts of the problem domain for accurate, stable solution
2. Relatively little excess resolution, to not require excess resources
3. Structural constraints from numerical and computational underpinnings (e.g. sufficient buffers between levels) [23]
4. Small total number of boxes to limit execution overhead and management costs
5. Enough boxes each with reasonable workload to enable load balancing
6. Fast, efficiently scalable execution
7. Computational 'niceness' of boxes (e.g. low surface area to limit communication; reasonable sizes and shapes to optimize sequential execution)
8. Aesthetic appeal to users in analysis and visualization(!)

There are clear tensions among these demands – more vs. fewer points (1 & 2), more vs. fewer boxes (4 & 5), and speed vs. output quality (6 & everything else). All of these are addressed in various forms in the extensive literature that has developed over the past several decades (e.g. [26, 27, 28, 29]). An issue not addressed in the existing literature is how to implement domain decomposition in a manner that doesn't create or include synchronization points around it.

Closely related to domain decomposition itself is the issue of how its result, the simulation's *metadata*, is represented and managed in distributed memory. The naïve approach of giving every processor a simple list of every box and its home is obviously non-scalable, in problem or system size. Thus, researchers have developed a range of solutions to this issue. These solutions can be roughly categorized along two axes: replicated vs. distributed metadata, and tree- vs. patch-structured boxes. The differences in box structures are entrenched in the assumptions that each framework can make depending on this choice, and so I will treat them separately, below.

The question of metadata representation and distribution has historically been treated as separate from the process of domain decomposition itself: metadata was first generated in full, in whatever form the algorithm in use dictated, and then transformed and packaged for each processor to receive and keep through subsequent simulation phases. Thus, even a fully asynchronous decomposition algorithm could still be stuck with synchronous metadata distribution, or vice versa.

This separation between metadata generation and distribution has begun to break down in more recent work. In Enzo-P, implemented in Charm++, the distributed 'scaffolding' objects generated in parallel during decomposition become the distributed metadata representation used in subsequent steps [30]. In SAMRAI, Brian Gunney has introduced algorithms that carry evolving neighbor information through the successive steps of the decomposition process, so that every box that ultimately results has this information attached to it [31]. I describe below how an adaptation of the ideas in Gunney's algorithm and its correctness proof, in combination with asynchronous box generation, can provide the foundations for fully asynchronous patch-based AMR (§ 4.2).

4.1 Tree-Structured AMR

In previous work [6], I have contributed to the optimization of a tree-structured AMR code that replaces collective communication in its domain decomposition with point-to-point messages among blocks and an asynchronous global 'convergence' test using a termination detection (TD) mechanism [32, 33]. We are currently preparing a revision of this work that requires only one round of TD per regridding operation rather than two, along with other implementation improvements. Most recently, I have designed a modification of this protocol that requires no global convergence test at all, eliminating non-local synchronization from it entirely.

The premise of our previous work in this area was that each block could start the regridding process with a calculation of what minimum level of refinement it would need to provide. Blocks send messages to their neighbors indicating their minimum refinement both after the initial calculation and anytime a block's level increases. Receiving a message from a neighbor may cause a block to increase its own minimum level, to maintain a consistent refinement ratio at all interfaces. Thus, these messages can cascade from one block to the next until all blocks reach a refinement level representing a global least fixed point. Because the algorithm communicated no information to place an upper bound on that fixed point, its determination had to be detected by the absence of further movement beyond it. The state machine driving each block's actions can be seen in figure 3.

In the newest protocol, blocks communicate not just their resolution demands that incrementally increase lower bounds on their neighbors, but also upper bounds on what resolutions could be demanded of them, based on what they know of their neighbors' conditions. When any block has sufficient information to equate its lower and upper bounds, it can immediately conclude that

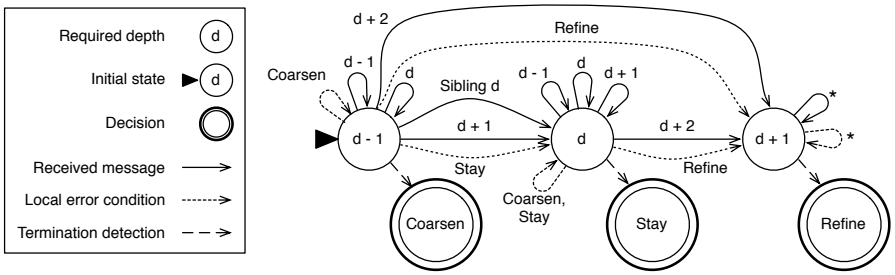

Figure 3: Per-block state machine for the loosely synchronizing tree-structured regridding protocol

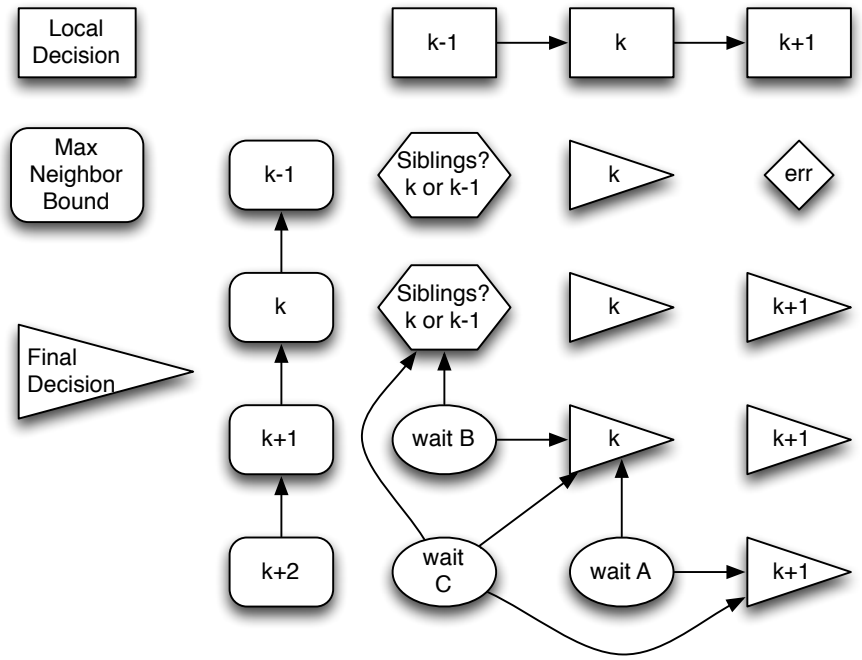

Figure 4: Per-block decision process for the full asynchronous tree-structured regridding protocol

it has reached its final value in the least fixed point solution. This decision process is illustrated in figure 4. Both upper and lower bounds can be communicated via local point-to-point messages. Based on the working hypothesis that every block will receive sufficient information to have its bounds converge, this protocol thus requires no synchronizing global convergence test.

In the near future, I plan to establish the new algorithm's correctness, implement it in our testbed code, and experimentally study its performance impact. Other members of the group working on this code have hypothesized that the direct performance benefits from eliminating the synchronization at this point will be marginal, since many blocks may remain unconverged for about as long as the global convergence test would have taken to trigger. Even if this is true, eliminating synchronization offers other potential benefits for overall work scheduling.

4.2 Patch-Structured AMR

Patch-based AMR can reduce the proportion of refined regions relative to tree-structured AMR, particularly for problems with complex spatial features []. Thus, algorithms for patch-based AMR domain decomposition generally attempt to minimize the fraction of the domain that is over-refined. The degree to which they achieve this is described as their *efficiency* [26]. Algorithms commonly offer parameterized tradeoffs of efficiency against the various other qualities listed above.

From an implementation standpoint, domain decomposition algorithms for patch-based AMR can be viewed as solving a less-regular version of the problem faced in tree-structured AMR. The same basic physical and mathematical considerations and constraints apply to the blocks generated: proper nesting, minimum and maximum areas of refinement. However, the additional structural degrees of freedom require that the decision algorithm be much more sophisticated. Rather than deciding on a per-block basis whether it should be coarsened, maintain its current level of refinement, or increase it, it must place arbitrary zones of refinement relative to the cell-level tags provided. Once the decision of which blocks to create is made, identifying their neighbors is also more complicated, since the possible relationships among them are more numerous and diverse.

There are two promising candidates for asynchronous patch-based grid decomposition algorithms. These algorithms are responsible for answering the first question posed above, of which blocks should exist. One possibility, Localized Berger-Rigoutsos [34] runs the classical Berger-Rigoutsos algorithm [26] within each existing block, and performs post-processing over these blocks to generate the final set of blocks. I believe that these post-processing steps can all be performed in a similarly localized fashion. The other possibility is Tiling [35], in which blocks are only created of a fixed shape and grid alignment. The locality of Tiling follows directly from its structural constraints.

As a new structure is determined, we can use an asynchronous modification of Gunney's 'Bridge' and 'Modify' algorithms [31] to identify the neighbors of

Figure 5: Adaptation of Gunney's Bridge algorithm to asynchronous per-block execution

Data: The blocks of collection C know their respective neighbors in collections A and B within neighborhoods of distance Γ.

Result: The blocks of collections A and B each know their neighbors in the other, and can each locally determine when that knowledge is complete.

Every block $c \in C$ executes **begin**
  Define N as the set of detected neighbors
  for *all neighbors* $a \in A$ *of* c **do**
    for *all neighbors* $b \in B$ *of* c **do**
      if a^Γ *intersects* b **then**
        insert (a, b) in N
      end
    end
  end
  for *all neighbors* $a \in A$ *of* c **do**
    send a a message containing all pairs $(a, x) \in N$
  end
  for *all neighbors* $b \in B$ *of* c **do**
    send b a message containing all pairs $(x, b) \in N$
  end
end

Every block $a \in A$ and $b \in B$ awaits as many messages as it knows of neighbors it has in C, and saves the union of their contents as the computed set of their neighbors in B or A, respectively.

the boxes in that structure. Its direct adaptation is shown in figure 5.

If the block-neighbor graph has k times more edges than the projection of that graph according to the blocks' processor mappings, then the new algorithm send k times more messages. This need not be problematic, since those messages will be more spread through time. Additionally, at strong scale, with few blocks per processor, the factor k may be quite small.

In scenarios where message injection rates or link/switch congestion do present issues for this algorithm, there are several possible optimizations that bring its behavior closer to the processor-level synchronous algorithm. Generically, these messages could be delivered through a library like TRAM [36] or Active Pebbles [37] to provide message-level aggregation. Any algorithm-specific process- or network-level aggregation that doesn't preserve the distinct messages simply needs to sum the number of contributor blocks of C that it represents.

A holistic program-performance optimization to the basic algorithm is that blocks of A and B can consume neighbor information as it arrives. For instance,

if the operation for which the information is needed is an interpolation from blocks of A to overlapping blocks of B, the computation and transmission for $a \rightarrow b$ can begin as soon as a learns of b. Implementing this optimization will require that a remember which blocks b it has sent data to in that operation, so that other blocks of C telling it about the same neighbor don't lead to repeated work.

The upshot of combining LBR or Tiling with Gunney's Bridge is that taken together there is no need for any sort of collective operation or synchronized communication to either generate new blocks nor to draw connections to and among them. There is also no longer a need for any explicit mesh structure representation, distributed or replicated, outside the implicit information embedded with each block. Thus, there is no need to communicate to gather that information. From this point, we can rely upon existing mechanisms such as those implemented in Charm++ for asynchronously distributing and locating those blocks across the set of processors in use.

5 Applications

For the purpose of measuring the performance of my work, I will perform benchmark studies on a range of applications previously implemented in the Chombo framework. Some of these applications have been used to measure the weak scaling efficiency of Chombo [38]. My aim is to present efficient strong scaling of these or similar applications. Collectively, they represent a mix of time-dependent and time-independent problems, stressing various elements of the overall work.

5.1 Godunov Gas Dynamics

One of the basic benchmarks used as an example program in Chombo is the simulation of gas dynamics using a Godunov method [39]. This problem is used as a representative of systems that follow hyperbolic conservation laws. Computationally, it presents a straightforward test of the basic elements of my thesis work - asynchronous update execution, timestep determination, and domain decomposition.

5.2 Multigrid Poisson

The Poisson equation is broadly used as a test problem for numerical software libraries and tools. As implemented in Chombo, it illustrates the time-independent elliptic solution process using a geometric multigrid approach across an AMR hierarchy. Chombo includes a native implementation of a geometric multigrid solver, written using its own communication and computation primitives. While time dependent problems perform different mathematical operations from such a solver, the computational expression is nearly the same - communication of ghost cells, relaxations over each block, and interpolation between levels (prolongation/restriction). Thus, optimizations of the Chombo infrastructure as a whole should carry through to this particular setting as well.

5.3 Shock-Induced Combustion

The ultimate goal of an effort to improve the performance of modeling infrastructure like an AMR framework is to expand the scientific reach of that framework. One of the predicted benefits of eliminating synchronization is the opportunity to provide more effective load balancing. Evaluating this prediction requires a test problem that induces very severe load imbalance, and showing that it can be made qualitatively more scalable than previously.

An intriguing target in this vein is the problem of shock-induced combustion [40][2]. In this problem, a tube filled with a gaseous mixture of fuel and air is subjected to a sharp increase in pressure. That shock induces heating in the gas. At some point, the heat is sufficient to ignite combustion. When and where this occurs, the combustion releases substantial additional heat and increases

[2]Description based on personal communication with Brian van Straalen

the local pressure even further as a result of the chemical reactions occurring. Thus, the shock is predicted to accelerate as the experiment progresses.

This problem exhibits several computationally challenging features. The simulated medium has several effective phases (unburnt, pre-shock, post-shock, and ash) with sharply different demands. Only a small but rapidly-moving portion of the domain undergoes chemical reactions, which must nevertheless be coupled to the overall dynamics. The moving shock front will require frequent regridding to track with the finest resolution. Therefore, efficient scaling of such an involved problem will demonstrate that the techniques described in this thesis are effective and can be implemented efficiently.

References

[1] P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. Chombo software package for amr applications-design document, 2000.

[2] Andrew M Wissink, Richard D Hornung, Scott R Kohn, Steve S Smith, and Noah Elliott. Large scale parallel structured amr calculations using the samrai framework. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 22–22. IEEE, 2001.

[3] B Fryxell, K Olson, P Ricker, FX Timmes, M Zingale, DQ Lamb, P MacNeice, R Rosner, JW Truran, and H Tufo. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.

[4] Brian W O'Shea, Greg Bryan, James Bordner, Michael L Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. Introducing enzo, an amr cosmology application. *arXiv preprint astro-ph/0403044*, 2004.

[5] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[6] A. Langer, J. Lifflander, P. Miller, Kuo-Chuan Pan, L.V. Kale, and P. Ricker. Scalable algorithms for distributed-memory adaptive mesh refinement. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 100–107, 2012.

[7] Harshitha Menon and Laxmikant Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 15. ACM, 2013.

[8] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 96:1–96:12, New York, NY, USA, 2013. ACM.

[9] Justin Luitjens and Martin Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[10] 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.

[11] Martin Berzins, John Schmidt, Qingyu Meng, and Alan Humphrey. Past, present and future scalability of the uintah software. In *Blue Waters Workshop, Chicago, IL, USA*, 2012.

[12] B. Adelstein-Lelbach, Z. Byerly, D. Marcello, G. Clayton, and H. Kaiser. Octopus: A scalable AMR toolkit for astrophysics. Scientific Computing Around Louisiana, February 2013.

[13] G.H. Miller and P. Colella. A conservative three-dimensional eulerian method for coupled solidfluid shock capturing. *Journal of Computational Physics*, 183(1):26 – 82, 2002.

[14] Ravi Samtaney. Suppression of the richtmyermeshkov instability in the presence of a magnetic field. *Physics of Fluids (1994-present)*, 15(8), 2003.

[15] Stephen L Cornford, Daniel F Martin, Daniel T Graves, Douglas F Ranken, Anne M Le Brocq, Rupert M Gladstone, Antony J Payne, Esmond G Ng, and William H Lipscomb. Adaptive mesh, finite volume modeling of marine ice sheets. *Journal of Computational Physics*, 2012.

[16] D Trebotich, P Colella, GH Miller, A Nonaka, T Marshall, S Gulati, and D Liepmann. A numerical algorithm for complex biological flow in irregular microdevice geometries. In *Technical Proceedings of the 2004 Nanotechnology Conference and Trade Show*, volume 2, pages 470–473, 2004.

[17] MR Dorr, RH Cohen, P Colella, MA Dorf, JAF Hittinger, and DF Martin. Numerical simulation of phase space advection in gyrokinetic models of fusion plasmas. In *Proceedings of the 2010 Scientific Discovery through Advanced Computing (SciDAC) conference, Chattanooga, TN*, pages 11–15, 2010.

[18] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[19] Konduri Aditya, Diego A Donzis, and Torsten Hoefler. Asynchronous PDE solver for computing at extreme scales.

[20] Andreas Frommer and Daniel B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(12):201–216, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[21] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[22] Francesco Miniati and Phillip Colella. Block structured adaptive mesh and time refinement for hybrid, hyperbolic n-body systems. *Journal of Computational Physics*, 227(1):400–430, 2007.

[23] Marsha J Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[24] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[25] Alper Üngör and Alla Sheffer. Tent-pitcher: A meshing algorithm for space-time discontinuous galerkin methods. In *IN PROC. 9TH INTL. MESHING ROUNDTABLE*, pages 111–122, 2000.

[26] Marsha Berger and Isidore Rigoutsos. An algorithm for point clustering and grid generation. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(5):1278–1286, 1991.

[27] Brian T.N. Gunney, Andrew M. Wissink, and David A. Hysom. Parallel clustering algorithms for structured AMR. *Journal of Parallel and Distributed Computing*, 66(11):1419 – 1430, 2006.

[28] C. Burstedde, L. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[29] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-scale AMR. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[30] James Bordner and Michael L. Norman. Enzo-p / cello: Scalable adaptive mesh refinement for astrophysics and cosmology. In *Proceedings of the Extreme Scaling Workshop*, BW-XSEDE '12, pages 4:1–4:11, Champaign, IL, USA, 2012. University of Illinois at Urbana-Champaign.

[31] B N Gunney. *Scalable Mesh Management for Patch-based AMR*. Jan 2013.

[32] Edsger W Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[33] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[34] Ralf Deiterding. Construction and application of an AMR algorithm for distributed memory computers. In *Adaptive Mesh Refinement-Theory and Applications*, pages 361–372. Springer, 2005.

[35] J. Luitjens and M. Berzins. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice and Experience*, 23(13):1522–1537, 2011.

[36] Lukasz Wesolwski. *Mesh Streamer*.

[37] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international Conference on Supercomputing*, pages 235–244. ACM, 2011.

[38] Brian Van Straalen, Phil Colella, Daniel T Graves, and Noel Keen. Petascale block-structured amr applications without distributed meta-data. In *Euro-Par 2011 Parallel Processing*, pages 377–386. Springer, 2011.

[39] P Colella, DT Graves, TJ Ligocki, DF Martin, and B Van Straalen. Amr godunov unsplit algorithm and implementation. 2004.

[40] Shannon Browne. Numerical solution methods for shock and detonation jump conditions. *Energy Conservation*, 1(w2):w2, 2004.