# THE SCALABILITY OF PARALLEL ADAPTIVE MESH REFINEMENT WITHIN UINTAH

by

Justin Paul Luitjens

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2011

# The University of Utah Graduate School

## STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Justin Paul Luitjens**

has been approved by the following supervisory committee members:

| | | |
|---|---|---|
| **Martin Berzins** | , Chair | **12/9/2010** Date Approved |
| **Steven Gregory Parker** | , Member | **12/9/2010** Date Approved |
| **Christopher Sikorski** | , Member | **12/7/2010** Date Approved |
| **Andrew Mahlon Wissink** | , Member | **11/24/2010** Date Approved |
| **Mary Wolcott Hall** | , Member | **12/05/2010** Date Approved |

and by **Martin Berzins**, Chair of the Department of **Computing**

and by Charles A. Wight, Dean of The Graduate School.

# ABSTRACT

Solutions to Partial Differential Equations (PDEs) are often computed by discretizing the domain into a collection of computational elements referred to as a mesh. This solution is an approximation with an error that decreases as the mesh spacing decreases. However, decreasing the mesh spacing also increases the computational requirements. Adaptive mesh refinement (AMR) attempts to reduce the error while limiting the increase in computational requirements by refining the mesh locally in regions of the domain that have large error while maintaining a coarse mesh in other portions of the domain. This approach often provides a solution that is as accurate as that obtained from a much larger fixed mesh simulation, thus saving on both computational time and memory. However, historically, these AMR operations often limit the overall scalability of the application.

Adapting the mesh at runtime necessitates scalable regridding and load balancing algorithms. This dissertation analyzes the performance bottlenecks for a widely used regridding algorithm and presents two new algorithms which exhibit ideal scalability. In addition, a scalable space-filling curve generation algorithm for dynamic load balancing is also presented. The performance of these algorithms is analyzed by determining their theoretical complexity, deriving performance models, and comparing the observed performance to those performance models. The models are then used to predict performance on larger numbers of processors. This analysis demonstrates the necessity of these algorithms at larger numbers of processors. This dissertation also investigates methods to more accurately predict workloads based on measurements taken at runtime. While the methods used are not new, the application of these methods to the load balancing process is. These methods are shown to be highly accurate and able to predict the workload within 3% error. By improving the accuracy of these estimations, the load imbalance of the simulation can be reduced, thereby increasing the overall performance.

Finally, the scalability of AMR simulations as a whole using these algorithms is tested within the Uintah computational framework. Scalability tests are performed using up to 98,304 processors and nearly ideal scalability is demonstrated.

# CONTENTS

# LIST OF TABLES

# ACKNOWLEDGEMENTS

The funding for the work within was provided by DOE-B524196, NSF-OCI-0721659, and NSF-OCI-0905068. This work could not have been accomplished without the help of many. In particular, I would like to thank Todd Harman, Jim Guilkey, J. Davison de St. Germain, John Schmidt, Qingyu Meng, Bryan Worthen, Steven Parker, and the rest of the C-SAFE team. I would also like to thank my committee, and in particular, my advisor Martin Berzins, for their advice throughout my career. In addition, I would like to thank Teragrid, the Texas Advanced Computing Center and the National Institute for Computation Sciences for providing me access to Ranger and Kraken.

Finally, I would like to thank my wonderful wife Erika for her patience and support throughout my graduate career. Without her unending love, I would not be where I am today.

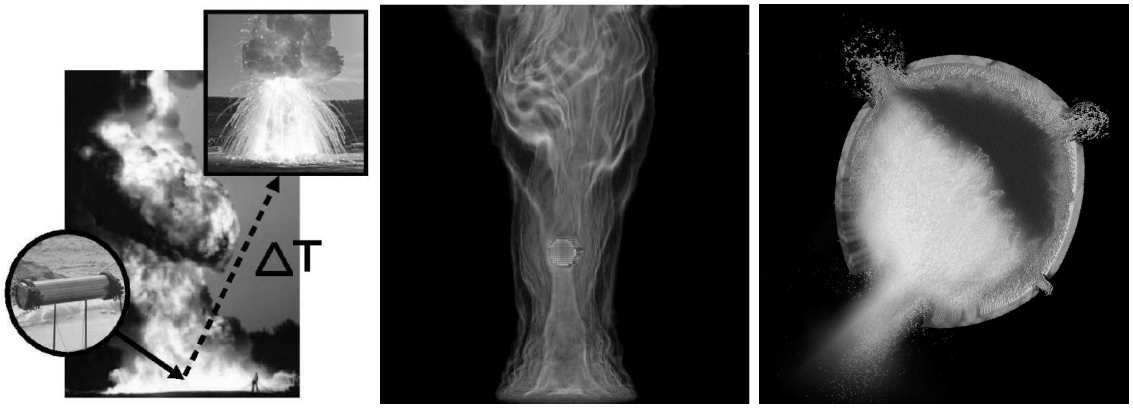
# CHAPTER 1

# INTRODUCTION

On August 10, 2005, a truck carrying 38,000 pounds of explosives through Spanish Fork Canyon in Utah jack-knifed and caught fire. Within a few minutes, the truck exploded violently, leaving a crater that was 30 feet deep and 70 feet wide. For unknown reasons, the force of the explosion was much more violent than anyone expected, leading to the questions "What caused the explosion to be so violent?" and "Can anything be done to help prevent such explosions in the future?" Understanding what led to the violent explosion is a challenging problem. Due to the size of this explosion, it is difficult to construct experiments to determine the possible causes of the violence. Simulation science can provide insight into the physical mechanics of the explosion without the need to physically recreate it.

The Center for Simulation of Accidental Fires and Explosions (C-SAFE) [40, 47, 89, 90] is a research program that utilizes simulation science to better understand explosions and fires. The center began in 1998 as a DOE-funded research project at the University of Utah. One of the target simulations of C-SAFE is a small steel container filled with a solid explosive (PBX-9501) that is suspended above a pool of fire. As the pool fire burns the container and explosive increase in temperature until a critical temperature is reached. At this point, the explosive ignites and begins to burn, converting the solid into a high temperature gas which in turn causes the container to pressurize. As the pressure increases, the container expands and eventually ruptures violently. Results from this simulation can be seen in Figure 1.1.

More recently, C-SAFE has begun research into arrays of explosives similar to the Spanish Fork accident. The hope of this work is to utilize simulations to determine what caused the violent explosion. It is suspected that the explosives underwent a deflagration to detonation transition (DDT) [101, 102, 107] due to the packing




(a) (b) (c)

**Figure 1.1**. Images of C-SAFE's target simulation (a). In this simulation, a container of explosive is suspended over a pool fire (b). The heat flux from the fire heats the container until it ignites and explodes violently (c).

arrangement of the explosives. Detonations are an order of magnitude more violent than deflagrations and a DDT would explain the violence of the Spanish Fork accident. Through simulation, the C-SAFE group is currently investigating models for DDT and the effect of packing arrangements for sympathetic explosions.

The simulation of such problems requires expertise from a variety of disciplines, including fire dynamics, fluid dynamics, material dynamics, and computer science. This broad requirement has necessitated a framework which facilitates the cooperation between experts from different domains. This has been accomplished through the Uintah computational framework [40, 47, 89, 90], which was developed by C-SAFE.

## 1.1 Uintah

The Uintah computational framework is a set of parallel software components and libraries built upon the DOE Common Component Architecture (CCA) that facilitate the solution of partial differential equations (PDEs) on structured grids. Uintah is a sophisticated framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and efficiently execute the resulting multiphysics simulation.

One of the primary strengths of Uintah is that application designers can develop large-scale parallel simulations with little understanding of the underlying parallelism.

To do this, the designers must specify their algorithm as a series of serial tasks that run on a hexahedral mesh patch. Each task specifies the serial computation for a single time step and the related variable dependencies. Variable dependencies state what data the task requires for the computation (along with the stencil width) and what data the task modifies or computes. Using these dependencies, Uintah creates a directed acyclic task graph that defines the communication requirements for the simulation along with the task execution order. This design eliminates the need for developers to design parallel algorithms, allowing them to focus on work within their domain. In addition, this design allows simulations to utilize advanced communication techniques including asynchronous communication and message coalescing, leading to greater parallel performance. Uintah will be discussed in more detail in Chapter 3.

## 1.2 Parallelism

The need for larger and faster simulations has necessitated the use of parallelism. Parallel applications utilize concurrent threads of execution on multiple processing cores to solve both larger problems and/or solve problems faster. To gain parallelism, these applications must be mapped efficiently onto the parallel architecture.

Parallel architectures contain a hierarchy of communication levels. The highest level contains nodes which are connected together via a network interconnect such as InfiniBand [72] and Remote Direct Memory Access over the Converged Enhanced Ethernet [26]. A node contains one or more sockets (CPUs) that are connected together via the system bus. Each CPU may also contain one or more processing cores. This system hierarchy leads to nonuniform memory access times between cores. For example, processing cores may share a cache, allowing for fast access of data between cores within the same CPU. However, cores on different CPUs within the same node do not share a cache and thus must communicate over the system bus but may still utilize shared memory. Communication between cores on different nodes must occur along the system interconnect through network protocols which generally do not allow for shared memory accesses. Throughout this dissertation, the terms processor and core will refer to a single processing unit which may be a single-core CPU or a single core of a multicore CPU.

Applications must take into account the system hierarchy. Parallelism on shared memory systems can be exploited with multithreading. This is traditionally accomplished explicitly through the use of threading libraries such as pthreads [71] or implicitly through libraries such as OpenMP [17]. OpenMP is a threading library that automatically parallelizes portions of an application. The advantage of the OpenMP approach is that parallelism can be exploited with little effort though the performance may not be as efficient as an explicit approach. Inter-node communication is generally accomplished through message passing across the communication network. The Message Passing Interface(MPI) is an API which defines a communication interface that explicitly communicates using sends and receives [98]. The MPI standard is the most commonly used method for achieving parallelism, which has led to the development of libraries including OpenMPI [38], MPICH [42], MVAPICH [53], along with many others.

Traditionally, applications have utilized an all-MPI or all-threaded approach to parallelism, depending on the system architecture. To better utilize internode communication, most MPI implementations optimize internode communication through shared memory, though the MPI specification does not require such features. However, more recently applications, like Carpet [87], have begun using hybrid threaded-MPI approaches where MPI is utilized between nodes and threading is utilized within a node.

Simulating C-SAFE's target problem takes hundreds of hours on a few thousand processors. Such a problem could not be currently simulated in serial. By using more processors, the solution can be computed faster and/or larger problems can be simulated. The need for both larger problems and faster times to solution has led to two commonly used methods for testing scalability.

The first method is commonly referred to as strong scalability. This method measures the change in execution time for a fixed-size problem as the number of processors vary. This method can be defined as follows:

Let $T(N, P)$ be the time to solve a problem of size n on p processors. Then, ideal strong scaling occurs when the problem is solved p times faster than in serial which is described by

$$T(N,P) = \frac{T(N,1)}{P},$$

Parallelism is also often expressed in terms of efficiency which is defined as

$$SE(N,P,P_0) = \frac{T(N,P_0)P_0}{T(N,P)P}.$$

where $P_0$ is the reference point for which the efficiency at $P$ is computed.

When T(N,P) for N fixed is plotted on a log-log graph, ideal strong scaling appears as a straight line with a slope of -1. This corresponds to an efficiency of 1. Increasing the number of processors should decrease the time to solution. However, in some cases, adding more processors may increase the time to solution and thus be counterproductive. These metrics help to determine where ideal strong scalability breaks down.

Speedup is another commonly used metric of strong scaling which is defined as

$$SU(N,P) = \frac{T(N,1)}{T(N,P)}.$$

This metric is useful because it precisely describes how much faster the problem was solved on P processors.

The second method of testing scalability is commonly referred to as weak scalability. This method measures the change in execution time as the number of processors and problem size vary proportionally to each other. Ideally, the execution time would remain the same as the problem size and the number of processors double. Ideal weak scaling occurs when a problem that is $c$ times larger is solved on c times as many processors in the same time which is described by the following equation:

$$T(N,P) = T(cN,cP).$$

The corresponding weak efficiency is defined as

$$WE(N_0, P_0, P) = \frac{T(N_0 P_0, P_0)}{T(N_0 P, P)},$$

where $N_0$ is the problem size per processor at $P_0$ processors.

Ideally, a log-log plot of $T(N, P)$ for a fixed $\frac{N}{P}$ appears as a horizontal line which also corresponds to an efficiency of 1. However, often increasing both the problem size and number of processors proportionally also results in an increase in execution time or a line with a positive slope.

The weak scaling metric is important because it more accurately represents how large parallel machines are used. As simulations move onto larger machines, it is often the case that the size of the simulation is increased. Algorithms that weak scale poorly experience diminishing returns on processors and thus solving larger problems may not be feasible.

A metric that combines the weak and strong efficiency metrics is presented in [13]. This metric measures the efficiency of computing a workload of size $N_1$ with $P_1$ processors relative to the time to compute a workload of size $N_0$ with $P_0$ processors and can be written as

$$SWE(N_0, N, P_0, P) = \frac{T(N_0, P_0)}{T(N, P)} \frac{P_0}{P} \frac{N}{N_0},$$

where $(N_0, P_0)$ is the reference point to which the efficiency for the point $(N, P)$ is computed. This metric has the advantage of providing a measure of total scalability given any two data points.

Achieving either type of ideal scalability is a challenging task and neither type of scalability implies the other. Each type of scalability poses its own unique challenges and it should not be assumed that achieving one type of scaling is easier than the other.

The following illustrates the difference between weak and strong scaling for commonly found functions of $T(N, P)$. The left image in Figure 1.2 shows the strong scaling for these functions. $\frac{N}{P}$, $\frac{N \log N}{P}$, both show ideal strong scaling, while a constant C and $\log P$ show no scalability. The C is important because every program has some

fixed overhead or serial portion that will eventually prevent ideal strong scaling, as captured by Amdahl's law [2].

The right image in Figure 1.2 shows the weak scaling for these same functions. $\frac{N \log N}{P}$ and $\log P$ do not exhibit ideal weak scalability. In fact, $\frac{N \log N}{P}$ and $\log P$ are equivalent within a constant when weak scaling. $\frac{N}{P}$ and C exhibit ideal weak scalability. The constant C illustrates the counter to Amdahl's law presented by Gustafson's in [45]. This paper fundamentally changed the way people thought about parallelism by illustrating that parallelism could still be achieved by increasing the size of the problem with the number of processors.

The terms that prevent ideal strong scaling are different than the terms that prevent weak scaling. For an algorithm to exhibit both ideal weak and strong scalability, $T(N, P)$ must be completely linear [81]. Thus, one of the challenges addressed in this dissertation will be to identify some of the functions that compose $T(N, P)$ for various algorithms used within Uintah and where possible, modify the algorithms so that they are as close to a linear cost as possible. By identifying these functions, we will be able to better understand the parallel performance of Uintah as a whole by predicting and verifying the performance of the main components. This analysis will then be used to predict what changes are needed to increase the scalability further.

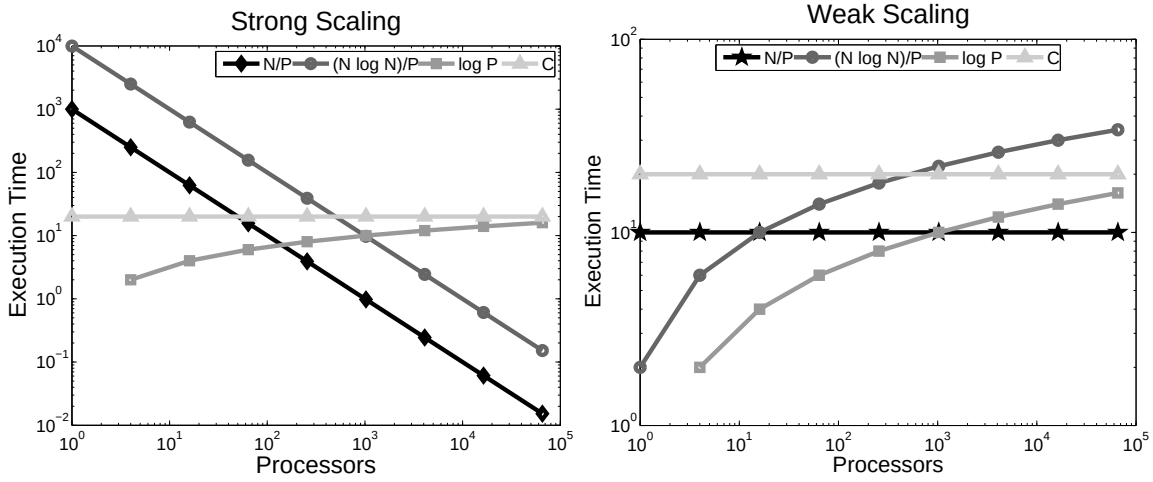



**Figure 1.2**. The scalability of common functions.

Simulating an explosion similar to the one at Spanish Fork canyon will require simulating an array of thousands of tightly packed explosives. Such a problem will require orders of magnitude more parallelism than the single container simulation. Petascale machines with over 100,000 processors have begun to arrive within the last two years. These machines are suitable platforms to simulate small arrays of explosives. However, in order to utilize these machines and future machines with many more processing cores, the scalability of Uintah will need to be improved substantially.

## 1.3 Adaptive Mesh Refinement

Solutions to Partial Differential Equations (PDEs) are often computed using standard finite difference techniques [15, 103], finite element methods [16, 39], or finite volume techniques [54, 113]. These methods approximate the solutions by discretizing the domain into a collection of computational elements referred to as a mesh. This mesh is often composed of a structured mesh (or grid) or by a collection of unstructured elements.

The error in the solution is dependent on the element size (or mesh spacing). One way to reduce this error is to refine the mesh which reduces the size of the elements. This, however, increases the number of elements which thereby also increases computational requirements. One way to decrease this error while limiting the total number of elements used is to refine the mesh locally in regions of the domain that have large error. By refining locally, the error across the domain can be reduced while only increasing the computational requirement in the regions that are refined. This provides a large reduction in computational and memory requirements over all-fine fixed-mesh solution with similar error. The two primary benefits of these reductions are that 1) larger problems can be solved and 2) the time to solution can be decreased.

Adaptive mesh refinement (AMR) describes a collection of mesh refinement algorithms that allow the refinement to change throughout the simulation. This is accomplished by estimating the error in the solution across the domain and dynamically adding or deleting refinement to the mesh as necessary. AMR algorithms are generally either unstructured (UAMR) or structured (SAMR). UAMR algorithms use a collection of finite elements which are split or combined as necessary. Alternatively

structured AMR (SAMR) utilizes multiple structured grids. There has been much work on both methods, and each has advantages and disadvantages, as discussed in [33]. This dissertation will focus on methods for block-structured AMR (BSAMR), as described below.

A ground-breaking method for block-structured AMR (BSAMR) was originally presented by Berger and Oliger in [8] and then by Berger and Colella in [10]. BSAMR uses error detection algorithms to flag portions of the domain that require more refinement. The algorithm then adds refinement to those portions of the domain through a process commonly referred to as remeshing or regridding. The process is repeated until a desired resolution is reached producing a dynamic multilevel grid, as defined below [10]:

An AMR grid $G$ is a sequence of grids at successively finer levels $1, \ldots, l_{max}$ such that

$$G = \cup_l G_l,$$

where $G_l$ is a collection of a hexahedral mesh patches with the same mesh spacing such that

$$G_l = \cup_k G_{l,k}.$$

In addition, the grid levels must be properly nested such that

1. a patch begins and ends at the corner of the coarser grid cell.

2. all fine cells exist within the interior of the coarser level.

Figure 1.3 shows such a multilevel grid comprised of three levels. It is not necessary that a patch be contained entirely within a coarser patch, as long as all grid points within that patch are contained within a coarser patch, as illustrated by patch $G_{3,1}$.

Octree-structured AMR (OSAMR) is an alternative to BSAMR that builds the mesh using an octree. This method combines elements of UAMR with SAMR and has

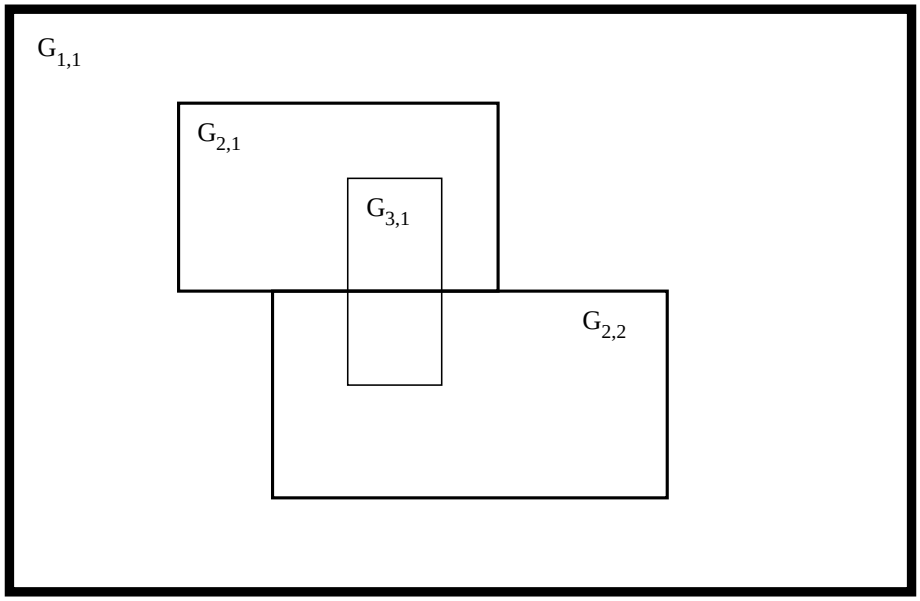



**Figure 1.3**. A multilevel grid of nested levels used within BSAMR

shown promising scalability results using up to 224,000 cores [20, 21, 22]. The meshes used in this method are fundamentally different than those used within BSAMR as the grid is not nested and each portion of the domain only exists on a single level. The challenge considered here is to analyze and extend the BSAMR algorithms to levels of scalability similar to that of OSAMR.

To adapt the grid in response to a solution evolving in time, a number of steps must occur that would not occur in a fixed mesh calculation. These steps generally include the creation of the new grid (regridding), the assignment of the new grid to cores (load balancing), and updating the computation and communication algorithms (scheduling). In many adaptive simulations, these changes can occur as often as every few time steps. This necessitates that these algorithms run efficiently in parallel. Poor performance within any of these algorithms has led to performance problems at large scales [77, 121, 122].

Achieving large scale parallelism, especially for AMR applications, is a challenging task. This has led to the development BSAMR frameworks like Uintah [77, 90], Enzo [85, 86], SAMRAI [121], Chombo [108], and AMROC [24] along with OSAMR frameworks like Paramesh [79], Flash [37], and ALPS [19].

These frameworks simplify the development of large-scale adaptive simulations by compartmentalizing the AMR algorithms, allowing them to be developed and optimized separately [89, 90] and also allow for code reuse. Scalability on today's largest machines has been shown for ALPS using up to 224,000 processors [20] and Uintah on 98,304 processors [13, 74].

### 1.3.1 Regridding

During the AMR process, refinement is often added or removed to regions of the domain. This is achieved through a process known as regridding or remeshing. One commonly used regridding algorithm is known as the Berger-Rigoutsos algorithm [11]. This algorithm uses edge detection algorithms from image processing to generate a tight fitting axis-aligned patch-set with relatively few patches. This algorithm was later parallelized in [9] and [122] but both relied substantially on global communication which limited scalability on large numbers of cores [44]. Improvements to this algorithm which focused on reducing the amount of global communication were later presented in [44] and showed scalability to 16K cores. Chapter 4 analyzes the parallel performance of commonly used regridding algorithms and presents two new methods for regridding which exhibit much better scaling properties than the original method.

### 1.3.2 Load Balancing

A load imbalance occurs when one or more cores are assigned more work than other cores. A large load imbalance will cause cores to wait for other cores to finish their computation, leading to poor utilization of system resources.

The variety of available simulation components within Uintah necessitates a sophisticated load balancer that is flexible enough to handle all of Uintah simulations. Dynamic load balancing can be described as the minimization of three competing costs: the cost of load imbalance, the cost of communication, and the cost to generate the load distribution.

In addition, too much communication can also cause performance issues. Communication across the network is slow relative to the time for computation and can easily dominate the time to reach a solution. In many simulations, communication

is predominantly local, meaning that only a small area around each patch must be communicated from physically neighboring patches. By clustering neighboring patches together, the framework can greatly reduce the necessary communication and significantly affect the overall runtime.

Finally, with AMR methods, the workload changes as the mesh changes. In addition, with particle methods, the workload can change on each time step as particles move throughout the domain. This can necessitate that load balancing occurs often, making it important that the time to generate the patch distribution is small relative to the overall computation. If a slow load balancing algorithm is used and load balancing occurs often, the time to load balance can dominate the overall runtime. In this case, it may be preferable to use a faster load balancing algorithm at the cost of more load imbalance [12].

The need for fast and effective load balancing techniques has led to the development of widely used load balancing applications like Metis [60], Jostle [115], and Zoltan [14, 31]. Uintah has recently been modified to support the Zoltan load balancing package, providing easy access to a number of algorithms. In addition, Uintah can use its own highly parallel load balancing algorithm [76] that utilizes space-filling curves, which has been shown to be comparable and in many cases better than Zoltan's space-filling curve load balancer within Uintah [82]. Chapter 5 will discuss the load balancing algorithms used within Uintah, including a fast parallel space-filling curve algorithm and self-tuning cost estimation algorithms.

## 1.4 Thesis Statement

To simulate a large array of explosives, many more processors will be required. Previous results have shown scalability problems associated with the changing mesh [77, 78, 121, 122]. Operations such as regridding and load balancing have tended to scale poorly and prevented the overall scalability of the entire application. This dissertation uses **analysis driven enhancements to improve the parallel performance of AMR to a hundred thousand processors** within the context of Uintah.

Proper analysis of the regridding and load balancing algorithms will provide a deeper understanding of the performance and scalability of AMR, which will then be

used to improve Uintah's performance to scale hundreds of thousands of processors. Analysis of the performance will help identify bottlenecks in either the algorithms or their implementations which can then be eliminated through new algorithms or code optimizations.

### 1.4.1 Unique Contributions

This dissertation's unique contributions are the following:

1. **A comprehensive description of the Uintah framework and the algorithms it uses for AMR;**
2. **The development of the Local Berger-Rigoutsos and Tiled regridding algorithms;**
3. **Analysis on the parallel complexity of the Global Berger-Rigoutsos, Local Berger-Rigoutsos, and Tiled regridding algorithms;**
4. **A scalable algorithm for parallel space-filling curve generation through sorting;**
5. **The application of existing filtering technology to automate the load balancing process;**
6. **In depth analysis of the scalability of Uintah at 98,304 processors.**

The analysis and improvements that were developed will be presented throughout the remainder of this dissertation. The dissertation has been organized into the following chapters: a review of related work is presented in Chapter 2, the Uintah framework is discussed in Chapter 3, algorithms for regridding are discussed in Chapter 4, algorithms for load balancing are discussed in Chapter 5, Uintah's parallel performance will be tested and analyzed in Chapter 6, and finally, Chapter 7 will present a summary of the work performed and conclusions that can be drawn.

# CHAPTER 2

# RELATED WORK

Block-structured adaptive mesh refinement for non-axis-aligned grids was originally presented by Berger and Oliger in [8]. The method was further refined and restricted to axis-aligned grids by Berger and Colella in [10].

The next major development by Berger and Rigoutsos was presented in [11]. This paper presented a method for grid creation that is now widely referred to as the Berger-Rigoutsos algorithm. This algorithm was parallelized in [122] which was based largely on the work in [9]. However, this parallelization suffered performance problems, leading to the optimizations presented in [44]. While this parallelization performed much better than the original, it still presented scalability challenges at large scales. This led to the development of the local Berger-Rigoutsos and tiled regridding algorithms in [75], which are presented within this dissertation. The tiled algorithm is similar to a method used in [73] for nonhierarchical grids.

The effective assignment of work to processors is necessary for efficient parallel simulations. Load balancing is not unique to AMR simulations and is a widely studied problem. The primary goal of load balancing is to reduce the overall time to solution by evenly assigning work to processors while also minimizing the communication. However, the decision to load balance must also consider the time to compute the partitions and redistribute the work as described in [12].

One commonly used method for load balancing is to partition the communication graph directly [61, 92, 96, 97]. However, constructing and partitioning the communication graph can be a time consuming operation and thus has prohibited the use in AMR applications which must load balance often.

This has led to the use of geometric bisection load balancing algorithms like those found in [9, 96]. More recently, research into the use of Space-Filling Curves

(SFC) [93] for load balancing has been undertaken in [88, 91] which has been shown to produce high quality partitions quickly [88, 32, 95, 105]. A parallel algorithm for SFC generation is presented in [76] and will also be presented in this dissertation.

Another commonly used method for load balancing involves diffusing work across processors at runtime, as described in [28, 50, 51, 52]. These methods tend to execute quickly and migrate small amounts of work often in order to achieve an effective load balance.

The widespread need for a variety of load balancing algorithms has led to the development of libraries such as Metis [60, 61], Jostle [115, 114, 116], and Zoltan [14, 31]. These libraries provide unified interfaces for common load balancing algorithms, allowing applications to easily utilize these algorithms.

To load balance the computation effectively, there must be an accurate prediction of workload. Recently algorithms for automated cost estimation were added to Uintah in [74]. This work used widely used filtering methods [84, 58, 123] to produce accurate workload estimations.

The need for parallel AMR simulations has led to the development of multiple simulation frameworks, including ALPS [19], AMROC [24, 29], Carpet [87] which was built upon Cactus [41], Chombo [108], Enzo [85, 86], Flash [37], Paramesh [79], SAMRAI [49], and Uintah [40, 47, 89, 90]. The uniqueness of each of these frameworks is in the simulations for which they were designed and the corresponding algorithms that they utilize. For example, ALPS simulates mantle convection, Carpet simulates astrophysics, and Uintah focuses on fires and explosions.

Achieving scalability within these frameworks has been a challenging task and the ability of each of these frameworks to scale has been demonstrated at different numbers of processors. SAMRAI illustrated the challenges in scaling BSAMR in [121, 122], where scalability up to 1024 processors was achieved. The scalability of BSAMR algorithms has since been improved incrementally within Uintah in [78, 77] by scaling to 4096 processors and the scalability was further improved in [74] up to 98,304 processors which will be presented within this dissertation. Similarly, the scalability of OSAMR in ALPS was also demonstrated at 32,768 processors in [22] and 65,536

processors in [21]. Subsequently, the scalability has been improved to over 224,000 processors in [20].

# CHAPTER 3

# UINTAH

## 3.1 Introduction

Uintah is a software framework originally designed by the members of C-SAFE to facilitate research into multiphysics simulations [40, 47, 89, 90]. The focus of Uintah was to provide a software infrastructure that aids in the rapid development of software by multiple researchers within various domains by allowing researchers to focus on work within their domain and not within another's domain.

The fundamental design methodology in Uintah is that of software components that are built on the DOE Common Component Architecture (CCA) component model [3]. Components are implemented as C++ classes that follow a simple interface to establish connections with other components in the system. The interfaces between components are simplified because the components do not explicitly communicate with one another. A component simply defines the steps in the algorithm that will be performed later when the algorithm is executed, instead of explicitly performing the computation tasks.

The primary advantage of a component-based approach is that it facilitates the separate development of simulation algorithms, models, and infrastructure, such that components of the simulation can evolve independently. A component-based architecture allows pieces of the system to be implemented in a rudimentary form at first and then evolve as the technologies mature. Most importantly, Uintah allows the aspects of parallelism (scheduling, load-balancing, parallel input/output, and so forth) to evolve independently of the simulation components. This approach allows the computer science research team to focus on these problems without waiting for the completion of the scientific applications or vice-versa.

Uintah uses a nontraditional approach to achieving high degrees of parallelism,

employing an abstract task graph representation to describe computation and communication that occur in the course of a single iteration of the simulation (typically a timestep or nonlinear solver iteration) [40, 89, 90]. Parallel execution is obtained through a global domain decomposition which is coupled with the scheduling of computational tasks. Consequently, Uintah components delegate decisions about parallelism to a scheduler component through a description of tasks and variable dependencies that describe communication patterns, which are subsequently assembled in a single task graph containing all of the computation for all components. This task graph representation has a number of advantages, including efficient fine-grained coupling of multiphysics components [89, 90], flexible load balancing mechanisms [74, 76], and a separation of application concerns from parallelism concerns. However, the task graph creates a challenge for scalability that was overcome by creating an implicit definition of this graph and representing the details of the graph in a distributed fashion [40, 89, 90].

This underlying architecture of the Uintah simulation is uniquely suited to taking advantage of the complex topologies of modern HPC platforms. Multicore processors in a shared multiprocessor configuration combined with one or more communication networks are common petascale architectures, but applications that are not aware of these disparate levels of communication may not be able to scale to the large CPU counts for complex problems. The explicit task graph structure enables runtime analysis of computations and communication patterns which can then be used to adapt work assignments dynamically at runtime [83].

## 3.2 Simulation Components

Uintah contains four main simulation algorithms: 1) the Arches incompressible fire simulation code [104], 2) the Implicit Continuous Eulerian (ICE) [66] compressible fluid code (both explicit and semi-implicit versions), 3) the particle-based Material Point Method (MPM) [109] for structural modeling, and 4) a fluid-structure interaction method achieved by the integration of the MPM and ICE components [43], referred to locally as MPMICE. In addition to these primary algorithms, Uintah integrates numerous subcomponents, including equations of state, constitutive models,

reaction models, radiation models, and so forth. The following subsections provide a high-level overview of the approach to "full physics" simulations of fluid-structure interactions involving large deformations and phase change. "Full physics" refers to problems involving strong coupling between the fluid and solid phases with a full Navier-Stokes representation of fluid phase materials and the transient, nonlinear response of solid phase materials, which may include chemical or phase transformation between the solid and fluid phases. Details of this approach, including the model equations and a description of their solution can be found at [43].

### 3.2.1 ICE

The methodology upon which the ICE component is built is a full "multimaterial" approach in which each material is given a continuum description and defined over the computational domain. Although at any point in space the material composition is uniquely defined, the multimaterial approach adopts a statistical viewpoint whereby the material (either fluid or solid) resides with some finite probability. To determine the probability of finding a particular material at a specified point in space, together with its current state (i.e., mass, momentum, energy), multimaterial model equations are used. These are similar to the traditional single material Navier-Stokes equations, with the addition of two intermaterial exchange terms. For example, the momentum equation has, in addition to the usual terms that describe acceleration due to a pressure gradient and divergence of stress, a term that governs the transfer of momentum between materials. This term is typically modeled by a drag law, based on the relative velocities of two materials at a point. Similarly, in the energy equation, an exchange term governs the transfer of enthalpy between materials based on their relative temperatures. For cases involving the transfer of mass between materials, such as a solid explosive decomposing into product gas, a source/sink term is added to the conservation of mass equation. In addition, as mass is converted from one material to another, it carries with it its momentum and enthalpy, and these appear as additional terms in the momentum and energy equations, respectively. Finally, two additional equations are required due to the multimaterial nature of the equations. The first is an equation for the evolution of the specific volume, which describes how

it changes as a result of physical process, primarily temperature, and pressure change. The other is a multimaterial equation of state, which is a constraint equation that requires that the volume fractions of all materials in a cell sum up to unity. This constraint is satisfied by adjusting the pressure and specific volume in a cell in an iterative manner. This approach follows the ideas previously presented by Kashiwa and colleagues [62, 63, 64, 66].

These equations are solved using a cell-centered, finite volume version of the ICE (for Implicit, Continuous-fluid, Eulerian) method [46], further developed by Kashiwa and others at Los Alamos National Laboratory [65]. The implementation of the ICE technique within Uintah invokes operator splitting in which the solution consists of a separate Lagrangian phase where the physics of the conservation laws, including intermaterial exchange, is computed, and an Eulerian phase, where the material state is transported via advection to the surrounding cells. The method is fully compressible, allowing wide generality in the types of scenarios that can be simulated.

#### 3.2.1.1 Governing Equations

The governing multimaterial equations are stated and described but not developed here. Their development can be found in [62]. The intent here is to identify the quantities of interest, of which there are 8, as well as those equations (or closure models) which govern their behavior. Consider a collection of $N$ materials, and let the subscript $r$ signify one of the materials, such that $\mathrm{r} = 1, 2, 3, \ldots, N$. In an arbitrary volume of space $V(\mathbf{x}, t)$, the averaged thermodynamic state of a material is given by the vector $[M_\mathrm{r}, \mathbf{u}_\mathrm{r}, e_\mathrm{r}, T_\mathrm{r}, v_\mathrm{r}, \theta_\mathrm{r}, p]$, the elements of which are the r-material mass, velocity, internal energy, temperature, specific volume, volume fraction, and the equilibration pressure. The r-material averaged density is $\rho_\mathrm{r} = \frac{M_\mathrm{r}}{V}$. The rate of change of the state in a volume moving with the velocity of r-material is:

$$\frac{1}{V}\frac{D_{\mathrm{r}}M_{\mathrm{r}}}{Dt} = 0, \tag{3.1}$$

$$\frac{1}{V}\frac{D_{\mathrm{r}}(M_{\mathrm{r}}\mathbf{u}_{\mathrm{r}})}{Dt} = \theta_{\mathrm{r}}\boldsymbol{\nabla}\cdot p + \textstyle\sum_{s=1}^{N}\mathbf{f}_{\mathrm{rs}}, \tag{3.2}$$

$$\frac{1}{V}\frac{D_{\mathrm{r}}(M_{\mathrm{r}}e_{\mathrm{r}})}{Dt} = -\rho_{\mathrm{r}}p\tfrac{D_{\mathrm{r}}v_{\mathrm{r}}}{Dt} + \textstyle\sum_{s=1}^{N}q_{\mathrm{rs}}, \tag{3.3}$$

where $\frac{D_r}{Dt}$ is the material or substantial derivative. Equations (3.1-3.3) are the averaged model equations for mass, momentum, and internal energy of r-material. The effects of turbulence and diffusion processes have been explicitly omitted from these equations.

In Equation (3.2), the term $\sum_{s=1}^{N}\mathbf{f}_{\mathrm{rs}}$ signifies a model for the momentum exchange among materials. This term is typically modeled as a function of the relative velocity between materials at a point. (For a two material problem, this term might look like $\mathbf{f}_{12} = K_{12}\theta_1\theta_2(\mathbf{u}_1 - \mathbf{u}_2)$ where the coefficient $K_{12}$ determines the rate at which momentum is transferred between materials). Likewise, in Equation (3.3), $\sum_{\mathrm{s}=1}^{N} q_{\mathrm{rs}}$ represents an exchange of heat energy among materials. For a two material problem, $q_{12} = H_{12}\theta_1\theta_2(T_2 - T_1)$ where $T_{\mathrm{r}}$ is the r-material temperature and the coefficient $H_{\mathrm{rs}}$ is analogous to a convective heat transfer rate coefficient.

The temperature $T_{\mathrm{r}}$, specific volume $v_{\mathrm{r}}$, volume fraction $\theta_{\mathrm{r}}$, and hydrodynamic pressure $p$ are related to the r-material mass density, $\rho_{\mathrm{r}}$, and specific internal energy, $e_{\mathrm{r}}$, by way of equations of state. The four relations for the four quantities $(T_{\mathrm{r}}, v_{\mathrm{r}}, \theta_{\mathrm{r}}, p)$ are:

$$e_{\mathrm{r}} = e_{\mathrm{r}}(v_{\mathrm{r}}, T_{\mathrm{r}}) \tag{3.4}$$

$$v_{\mathrm{r}} = v_{\mathrm{r}}(p, T_{\mathrm{r}}) \tag{3.5}$$

$$\theta_{\mathrm{r}} = \rho_{\mathrm{r}}v_{\mathrm{r}} \tag{3.6}$$

$$0 = 1 - \textstyle\sum_{s=1}^{N}\rho_{\mathrm{s}}v_{\mathrm{s}}. \tag{3.7}$$

Equations (3.4) and (3.5) are, respectively, the caloric and thermal equations of state. Equation (3.6) defines the volume fraction, $\theta$, as the volume of r-material per total material volume, and with that definition, Equation (3.7), referred to as the multimaterial equation of state follows. It defines the unique value of the hydrodynamic pressure $p$ that allows arbitrary masses of the multiple materials to identically fill the volume $V$. This pressure is called the "equilibration" pressure [66].

Equations (3.1-3.7) form a set of seven equations for the seven-element state vector $[M_{\rm r}, \mathbf{u}_{\rm r}, e_{\rm r}, T_{\rm r}, v_{\rm r}, \theta_{\rm r}, p]$, for any arbitrary volume of space $V$ moving with the r-material velocity. The approach described here uses the reference frame most suitable for a particular material type. As such, there is no guarantee that arbitrary volumes will remain coincident for materials described in different reference frames. This problem is addressed by treating the specific volume as a dynamic variable of the material state which is integrated forward in time from initial conditions. In so doing, at any time, the total volume associated with all of the materials is given by:

$$V_t = \Sigma_{r=1}^{N} M_{\rm r} v_{\rm r}, \tag{3.8}$$

so that the volume fraction is $\theta_{\rm r} = M_{\rm r} v_{\rm r}/V_t$ (which sums to one by definition). An evolution equation for the r-material specific volume, derived from the time variation of Equations (3.4-3.7), has been developed in [62]. It is stated here as:

$$\frac{1}{V}\frac{D_{\rm r}(M_{\rm r} v_{\rm r})}{Dt} = f_{\rm r}^{\theta} \boldsymbol{\nabla} \cdot \mathbf{u} \quad + \quad \left[ \theta_{\rm r}\beta_{\rm r}\frac{D_{\rm r}T_{\rm r}}{Dt} - f_{\rm r}^{\theta}\Sigma_{s=1}^{N}\theta_{\rm s}\beta_{\rm s}\frac{D_{\rm s}T_{\rm s}}{Dt} \right] , \tag{3.9}$$

where $f_{\rm r}^{\theta} = \frac{\theta_{\rm r}\kappa_{\rm r}}{\sum_{\rm s=1}^{N}\theta_{\rm s}\kappa_{\rm s}}$, $\beta$ is the constant pressure thermal expansivity, and $\kappa_{\rm r}$ is the r-material bulk compressibility.

The evaluation of the multimaterial equation of state (Equation (3.7)) is still required to determine an equilibrium pressure that results in a common value for the pressure, as well as specific volumes that fill the total volume identically.

### 3.2.2 MPM

Uintah contains a component for solid mechanics simulations known as the Material Point Method (MPM) [109]. MPM is a particle-based method that uses a (typically) Cartesian mesh as a computational scratch-pad upon which gradients and spatial integrals are computed. MPM is an attractive computational method in that the use of particles simplifies initial discretization of geometries and eliminates problems of mesh entanglement at large deformations [18], while the use of a Cartesian grid allows the method to be quite scalable by eliminating the need for directly computing particle-particle interactions. A graphical description of MPM is shown in Figure 3.1. There, a small region of solid material is represented by four particles and overlayed by a portion of grid, as shown in panel (a). The particles, or material points, carry minimally, mass, volume, velocity, temperature, and state of deformation. Panel (b) reflects that this particle information is projected to the nodes of the overlying grid, or scratch-pad. Based on the nodal velocity, a velocity gradient is computed at the particles, and is used to update the state of deformation of the particle, and the particle stress. Based on the internal force (divergence of the stress) and external forces, as well as the mass, acceleration is computed at the nodes, as is an updated velocity. This is shown in panel (c), where the computational nodes have been deformed based on the new velocity and the current timestep size. Note that the deformation of the grid is never explicitly performed. Panel (d) reflects that the changes to the field quantities, specifically velocity and position in the case, are interpolated back to the particles. Once the particle state has been updated, the grid is reset in preparation for the next timestep (Panel (e)).

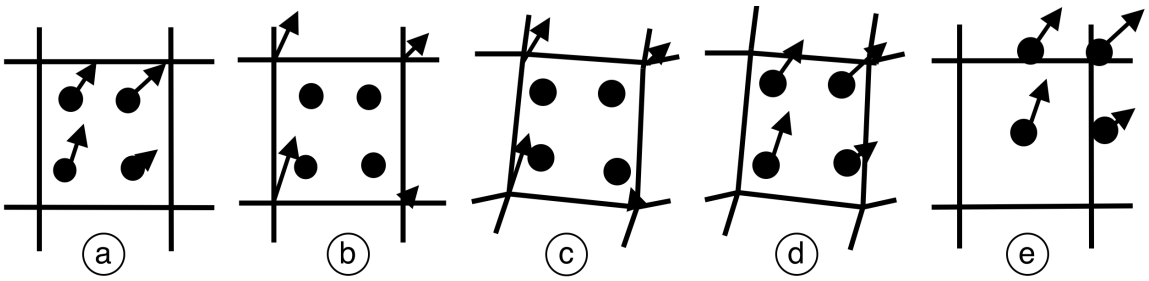



**Figure 3.1**. Graphical depiction of the steps in the MPM algorithm

### 3.2.3 MPMICE

Fluid-structure interaction capability is achieved by incorporating the Material Point Method within the multimaterial CFD formulation. In the multimaterial formulation, no distinction is made between solid or fluid phases. Where distinctions do arise is in the computation of the material stress and material transport or advection. In the former, the use of a particle description for the solid provides a convenient location upon which to store the material deformation and any relevant history variables required for, say, a plasticity model. Regarding the latter, Eulerian advection schemes may sometimes be quite diffusive, and would lead to an unacceptable smearing of a fluid-solid interface. Thus, performing advection by moving particles according to the local velocity field eliminates this concern.

At the beginning of each timestep, the particle state is projected first to the computational nodes, and then to the cell centers. This colocates the solid and fluid data, and allows the multimaterial CFD algorithm to proceed without being aware of the presence of distinct phases. As stated above, the constitutive models for the solid materials are evaluated at the particle locations, and the divergence of the stress at the particles is computed at the cell centers. The Lagrangian phase of the CFD calculation is then completed, including exchange of mass, momentum and enthalpy. At this point, *changes* to the solid materials' state are interpolated back to the particles, and their state, including position, is updated. Fluid phase materials are advected using a compatible advection scheme such as that described in [112]. Again, a full description can be found in [43].

## 3.3 Domain Decomposition

Uintah discretizes the problem domain into a structured grid of cells. The solution is then computed on this grid. Parallelism is achieved by dividing the grid into hexahedral regions of cells called patches which are then distributed onto processors. Each processor is only responsible for calculations within its assigned patches. The creation of the patch set is referred to as regridding and will be discussed briefly in Section 3.6.3 and in more detail in Chapter 4. The process of assigning patches to processors is referred to as load balancing and will be discussed briefly in Section 3.6.4

and in more detail in Chapter 5.

Computations within Uintah occur on variables which can exist at the cell centers, node centers, or face centers of each cell, as defined by the grid. In addition, Uintah also supports particles which can exist anywhere within the grid. Figure 3.2 shows where cells, nodes, and faces exist within a Uintah grid.

## 3.4 Task Graph

Uintah enables integration of multiple simulation algorithms by adopting an execution model based on coarse-grained "macro" dataflow. Each component decomposes the steps of an algorithm and the data dependencies between those steps into tasks. Each task defines a single step of the algorithm along with the necessary dependencies. The dependencies describe both the inputs and outputs of each task. The inputs to a task are referred to as "requires" and specify which variables (along with a stencil width) the task needs to execute. The outputs of a task are referred to as "computes" and specify which variables the task produces.

The tasks are combined into a directed acyclic graph structure (called a *task graph*). As a result, the task graph represents the computation to be performed in single timestep integration and the data dependencies between the various steps in the algorithm. The task graph may specify numerous exchanges of data between components (fine-grained coupling) or few (coarse-grained coupling), depending on

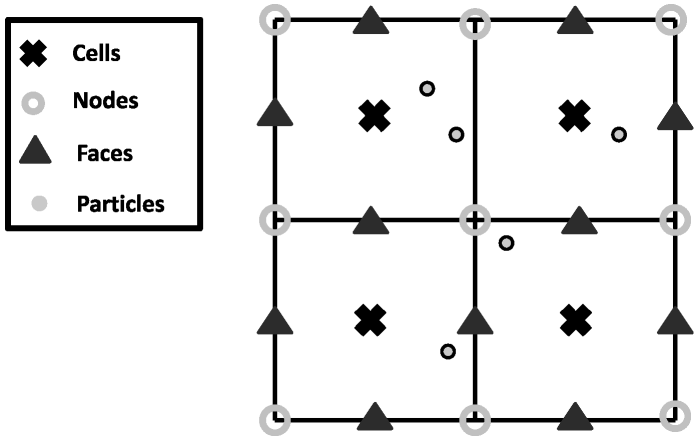



**Figure 3.2**. The location of cells, nodes, faces, and particles on a Uintah grid

the requirements of the underlying algorithm. For example, MPMICE requires several points of data exchange between the MPM and ICE components in a single timestep to achieve the tight coupling between the fluid and solids. The task graph structure of Uintah allows fine-grained interdependencies to be expressed in an efficient manner.

Figure 3.3 shows an example of part of the MPM task graph. In this figure, the nodes represent tasks and the edges represent dependencies between those tasks that may require communication.

A task graph representation by itself works well for coupling of multiple computational algorithms, but presents challenges for achieving scalability. A task graph that represents all communication in the problem would require time and memory proportional to the number of computational elements throughout the entire domain. Creating this on a single processor, or on all processors, would eventually result in a bottleneck.

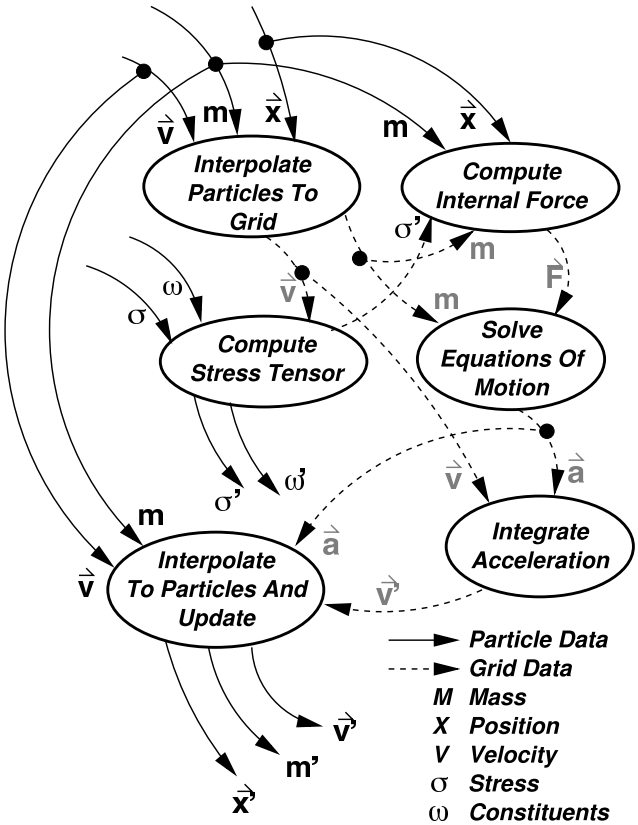



**Figure 3.3**. An example Uintah task graph. The nodes represent tasks and the edges represent dependencies that may require communication.

Uintah addresses this problem by introducing the concept of a "tensor product task graph." Uintah components specify tasks for the algorithmic steps only, which are independent of problem size or number of processors. This is illustrated in Figure 3.3 which does not include information on the number of processors or the domain layout. Each task in the task graph is then implicitly repeated on a portion of patches in the decomposed domain. The resulting graph, or tensor product task graph, is created collectively where each processor contains only the tasks which it owns or must communicate with. These tasks are defined as the processors neighborhood. The graph exists only as a whole across all computational elements, resulting in a scalable representation of the graph. Communication requirements between tasks are also specified implicitly through a simple dependency algebra.

Each execution of a task graph integrates a single timestep, or a single nonlinear iteration, or some other coarse algorithm step. Task graphs may also be assembled recursively such that an individual task could be a task graph comprised of many tasks.

Figure 3.4 shows the detailed task graph stored on a single processor for a two patch and two processor ICE simulation. In this graph, nodes represent tasks and edges represent dependencies. Dashed dependencies are already enforced through a parent's dependencies but may still require communication. The double black edges represent the critical path through the task graph. This figure illustrates the complexity of the tensor product task graph even at small numbers of patches. Increasing the number of patches within a processors neighborhood quickly increases the size and complexity of this graph. For comparison, Figure 3.5 shows the detailed task graph stored on a single processor for a two patch and two processor MPMICE simulation. This graph shows the increased complexity of MPMICE simulations over ICE simulations.

The idea of the dataflow graph as an organizing structure for execution is well known and can be traced back to Sarkar's dissertation [94]. The SMARTS [111] dataflow engine that underlies the POOMA [4] toolkit shares similar goals and philosophy with Uintah. The SISAL language compilers [36] used dataflow concepts at a much finer granularity to structure code generation and execution. Other frameworks

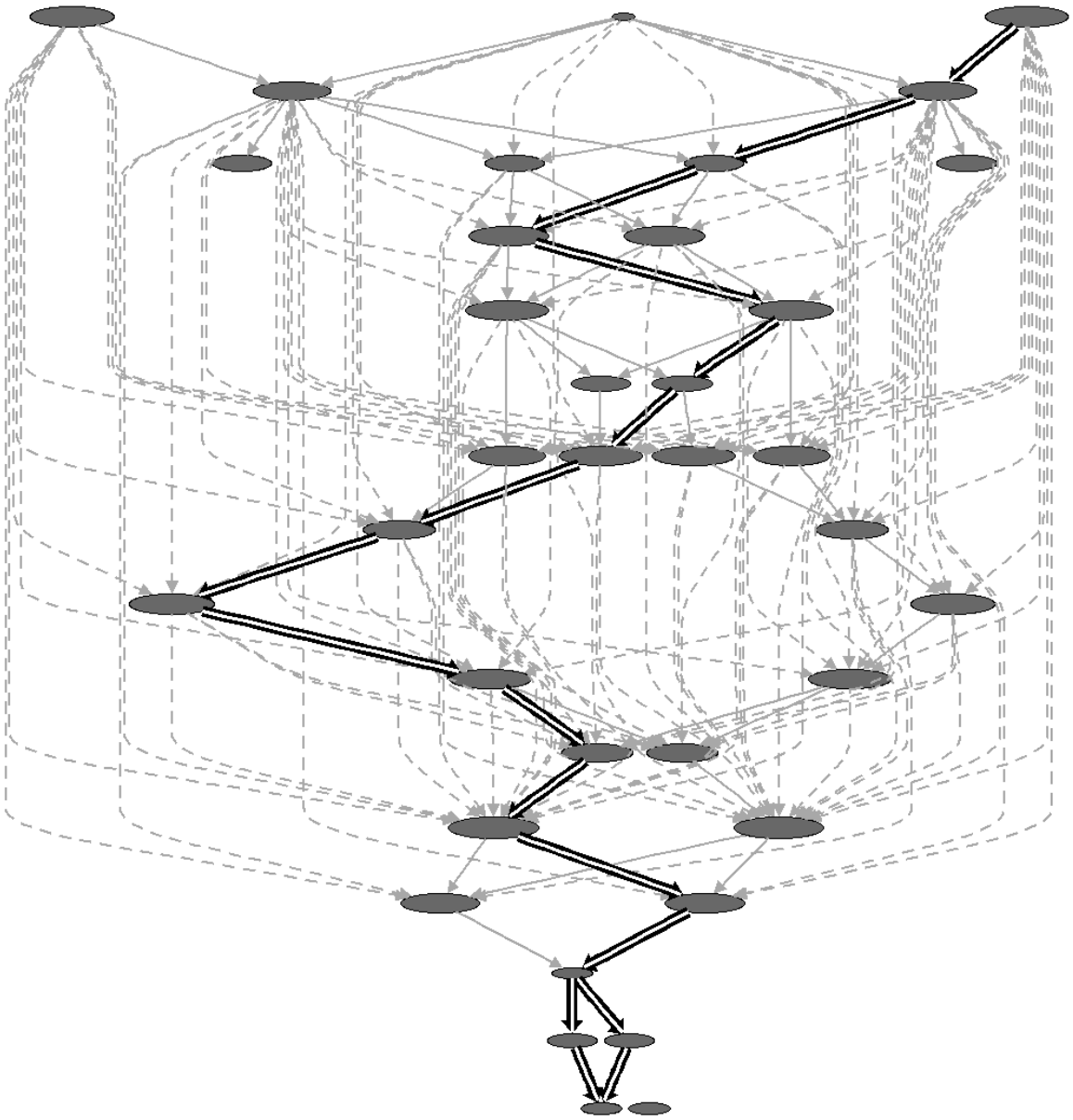


**Figure 3.4**. The detailed task graph for ICE. In this graph, nodes represent tasks and edges represent dependencies. Dashed dependencies are redundantly enforced through a descendent's dependencies but may still require communication. The double black edge represents the critical path through the task graph.

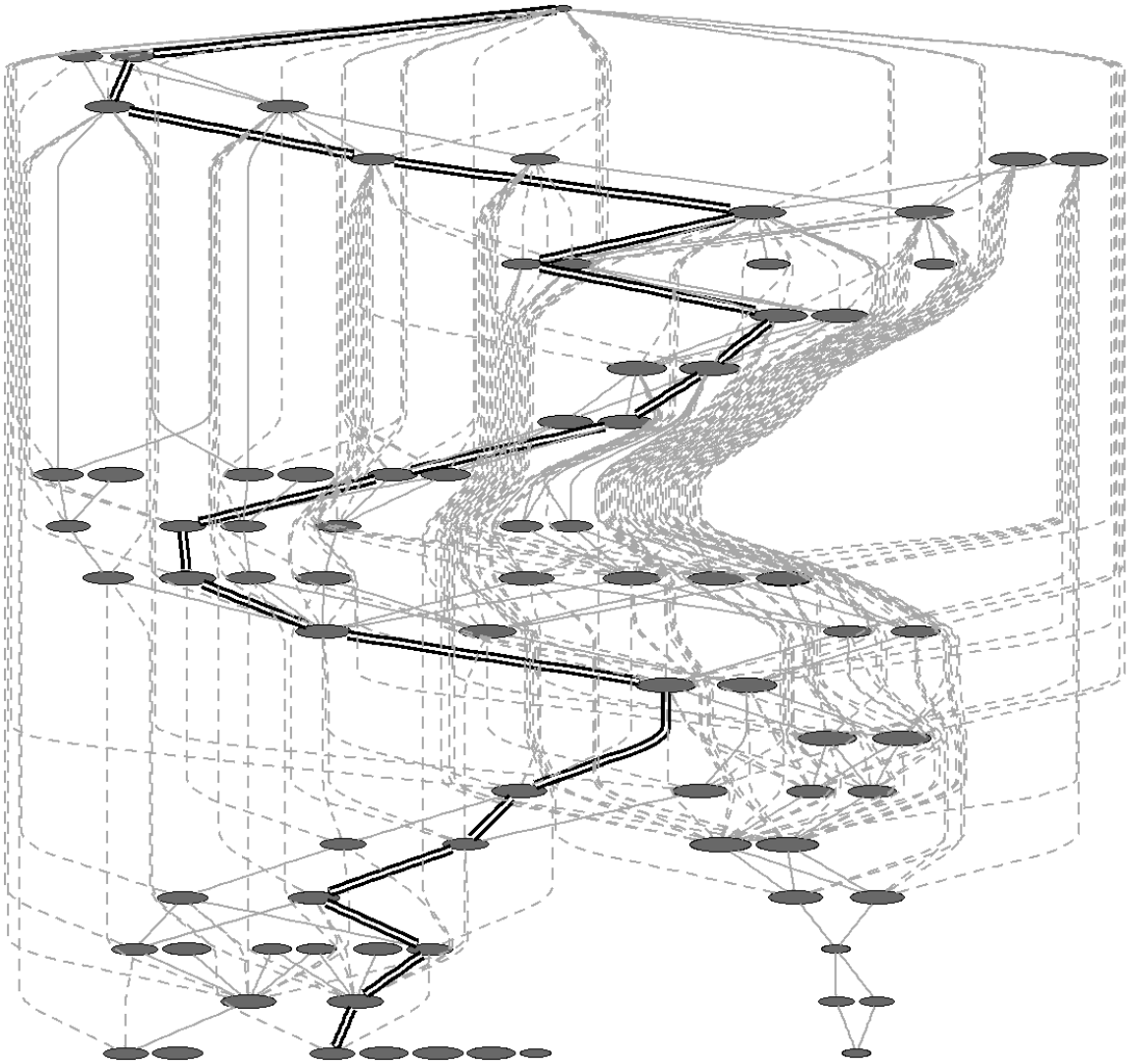


**Figure 3.5**. The Uintah task graph for MPMICE. In this graph, nodes represent tasks and edges represent dependencies. Dashed dependencies are redundantly enforced through a descendent's dependencies but may still require communication. The double black edge represents the critical path through the task graph.

that have utilized a similar task-graph-based approach also include Charm++ [57], TBLAS [23], Scioto [34], and Concurrent Collections [68].

Dataflow is a simple, natural, and efficient way of exposing parallelism and managing computation, and is an intuitive way of reasoning about parallelism. What distinguishes implementations of dataflow ideas is that each caters to a particular higher-level representation. SMARTS caters to POOMA's C++ implementation and stylistic template-based presentation. Uintah's implementation supports dataflow (task) graphs of C++ and Fortran-based mixed particle/grid algorithms on a structured adaptive mesh.

### 3.4.1 Task Graph Execution

Once the task graph is compiled, it is executed repeatedly with each execution typically corresponding to single timestep. The schedule component is responsible for this execution process. In serial, the execution of this task graph is straightforward and is accomplished by executing the tasks in a topologically sorted order. The ordering is determined by the variable dependencies such that all variables are computed from some task before they are required by another task.

In parallel, the execution process is more complex. In addition to determining the execution order, the scheduler must also communicate data as needed between tasks on different processors. Communication is potentially required whenever a task requires a variable with a nonzero stencil width. For such requirements, variables from neighboring patches will need to be communicated.

Using the task graph, the scheduler can implicitly determine which communication is necessary and execute that communication appropriately. The scheduler component implements this communication in an asynchronous manner. By using nonblocking sends (MPI_Isend) and receives (MPI_Irecv), the scheduler can overlap communication and computation by executing tasks while others are communicating. This overlap greatly reduces the time for communication, leading to large performance improvements of synchronous communication.

In a similar manner to the serial scheduler, the parallel scheduler can execute tasks in a topologically sorted order or alternatively can execute tasks out-of-order

depending on which tasks are ready to execute [83]. Allowing tasks to execute in a dynamic order facilitates a better processor utilization by reducing the amount of time the simulation stalls while waiting for data to arrive. This has been shown to significantly reduce the MPI wait time by up to 80% [83]. In addition, current work within Uintah is focusing on adding multithreaded capability, which will allow tasks to be executed by different threads on the same MPI process. This work could eliminate on-node communication, thereby reducing the execution time. In addition, it is expected that this will also greatly reduce memory usage as global data structures that exist for each MPI process will be shared across threads.

To accommodate software packages that were not written using the Uintah execution model, tasks can be specially flagged as "OncePerProc." These tasks are "gang-scheduled" such that these tasks are associated with all patches on a processor and are executed synchronously across processors. This allows third-party applications to be called in a synchronous manor. In this fashion, Uintah applications can utilize external libraries, such as PETSc [5] and Hypre [35], for the solution of linear equations.

## 3.5 Infrastructure Features

The task graph representation in Uintah enables compiler-like analysis of the computation and communication steps that occur during a timestep. Since the combination of tasks required to compute the algorithm may vary dramatically based on problem parameters, this analysis is performed at runtime. Through analysis of the task graph, Uintah can automatically create checkpoints, perform load balancing, and eliminate redundant communication. This analysis phase, which is referred to as "compiling" or "scheduling" of the task graph, is what distinguishes Uintah from most other component-based multiphysics simulations. The task graph is recompiled when the structure of the computation or communication has changed. This occurs when the grid changes, when the nature of the algorithm changes, or when the patch distribution is changed. Uintah also has the ability to modify the set of components in use during the course of the simulation. This ability is used to transition between solution algorithms, such as a fully explicit or semi-implicit formulation, when triggered

by conditions in the simulation.

Using the task graph, Uintah also performs data lifetime analysis and determines when a variable is no longer needed. At this point, the infrastructure can automatically delete the variable, freeing up resources for other calculations.

Data output is scheduled by creating tasks in the task graph just like any other component. In typical simulations, each processor writes data independently for the portions of the data set which it owns. This requires no additional parallel communication for output tasks. However, in some cases, this may not be ideal. Uintah can also accommodate situations where disks are physically attached to only a portion of the nodes, or a parallel filesystem where I/O is more efficient when performed by only a fraction of the total nodes.

Checkpointing is also obtained by using this output process. At the end of a timestep, the simulation may produce a checkpoint which contains all data required to restart the simulation. Data lifetime analysis ensures that only the data required by subsequent iterations is saved within these checkpoints. During a restart, the components process the XML specification of the problem that was saved with the data sets, and then Uintah creates input tasks which load data from the checkpoint files. If necessary, data redistribution is performed automatically during the first execution of the task graph. As a result, changing the number of processors when restarting is possible.

## 3.6 Adaptive Mesh Refinement

Many multiphysics simulations require a broad span of space and time scales. Uintah's primary target simulation scenario includes a large scale fire (size of meters, time of minutes) combined with an explosion (size of microns, time of microseconds). To efficiently and accurately capture this wide range of time and length scales, the Uintah architecture has been designed to support AMR in the style of Berger and Colella [10]. Figure 3.6 shows an example of an AMR simulation within Uintah. In this simulation, a blast wave is expanding and reflecting off of the left wall solid boundary with an AMR mesh using the explicit ICE algorithm with refinement in both space and time.

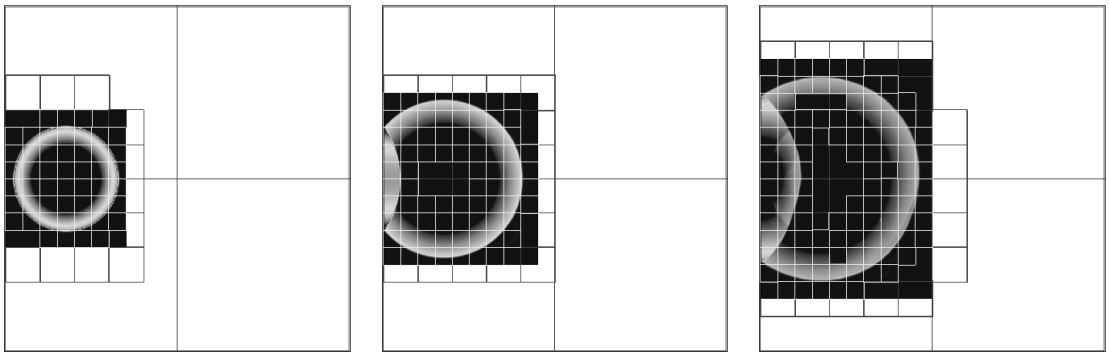


**Figure 3.6**. A pressure blast wave reflecting off of a solid boundary

The construction of the Uintah AMR framework requires the use of multilevel grid support and grid adaptivity. A multilevel grid consists of a coarse grid with a series of finer levels. Each finer level is placed on top of the previous level with a spacing that is equal to the coarse level spacing divided by the refinement ratio, which is specified by the user. A finer level grid only exists in a subset of the domain of the coarser level. The coarser level grid is then used to create boundary conditions for the finer level.

The framework supports multilevel grids by controlling the interlevel communication and computation. Grid adaptivity is controlled through the use of refinement flags. The simulation components generate a set of refinement flags which specify the regions of the level that need more refinement. As the simulation evolves, the refinement flags are used to create new grids with the necessary refinement.

Whenever the grid changes, a series of steps must be taken prior to continuing the simulation. First, the patches must be distributed evenly across processors through load balancing. Second, all patches must be populated with data either by copying from the same region of the previous grid or by interpolating from a coarser region of the previous grid. Finally, the task graph must be recompiled. These steps can take a significant amount of runtime [78, 121] and affect the overall scalability at large numbers of processors.

### 3.6.1 Multilevel Execution Cycle

There are two styles of multilevel execution methods implemented in Uintah: the Lockstep method (typically used for implicit or semi-implicit solvers) and the W-cycle method (typically used for explicit formulations) time integration models. The Lockstep model, as shown in Figure 3.7, advances all levels simultaneously. After executing each timestep, the coarsening of data and interpolation of boundary conditions are applied. The W-Cycle, as shown in Figure 3.8, uses larger timesteps on the coarser levels than the finer levels. On the finer levels, there are at least $n$ subtimesteps on the finer level for every subtimestep on the coarser level, where $n$ is equal to the refinement ratio. This provides the benefit of performing less computation on the coarser levels but also requires an extra step of interpolating the boundary conditions after each subtimestep. In both cycles, at the end of each coarse level timestep, the data on a finer level are interpolated to the coarser levels, ensuring that the data on the coarser level reflects the accuracy of the finer level. In addition, the boundary conditions for the finer level are defined by the coarser levels by interpolating the data on a coarse level to the domain boundaries of the finer level [10].

### 3.6.2 Refinement Flags

After the execution of a timestep, the simulation runs a task that marks cells that need refinement. The criteria for setting these flags is determined by the simulation component, and is typically determined by a gradient magnitude of a particular quantity or other error metrics. These refinement flags are then used during the regridding process.

### 3.6.3 Regridding

When using AMR, the areas of the domain that require refinement can change every timestep, necessitating an adaptive grid. The regridding process occurs whenever the simulation components produce refinement flags that are outside of the current grid, at which time the regridder creates a new grid which encompasses all refinement flags. During this process, refinement may be added or removed from portions of the

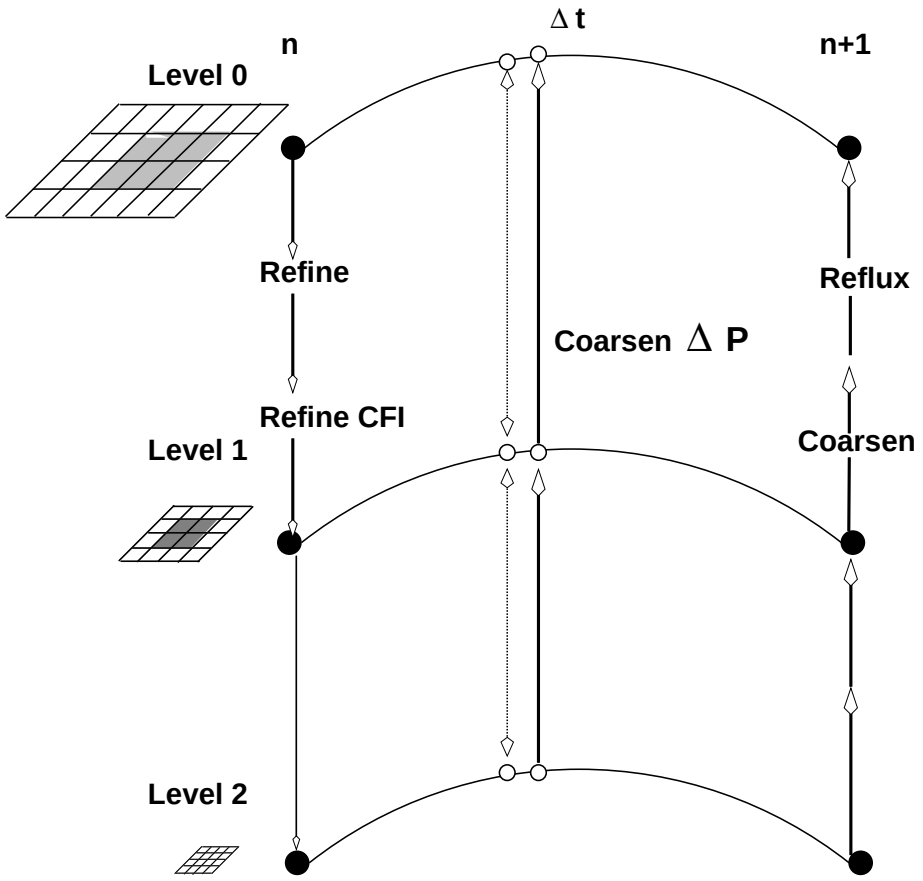



**Figure 3.7**. The Lockstep time integration model

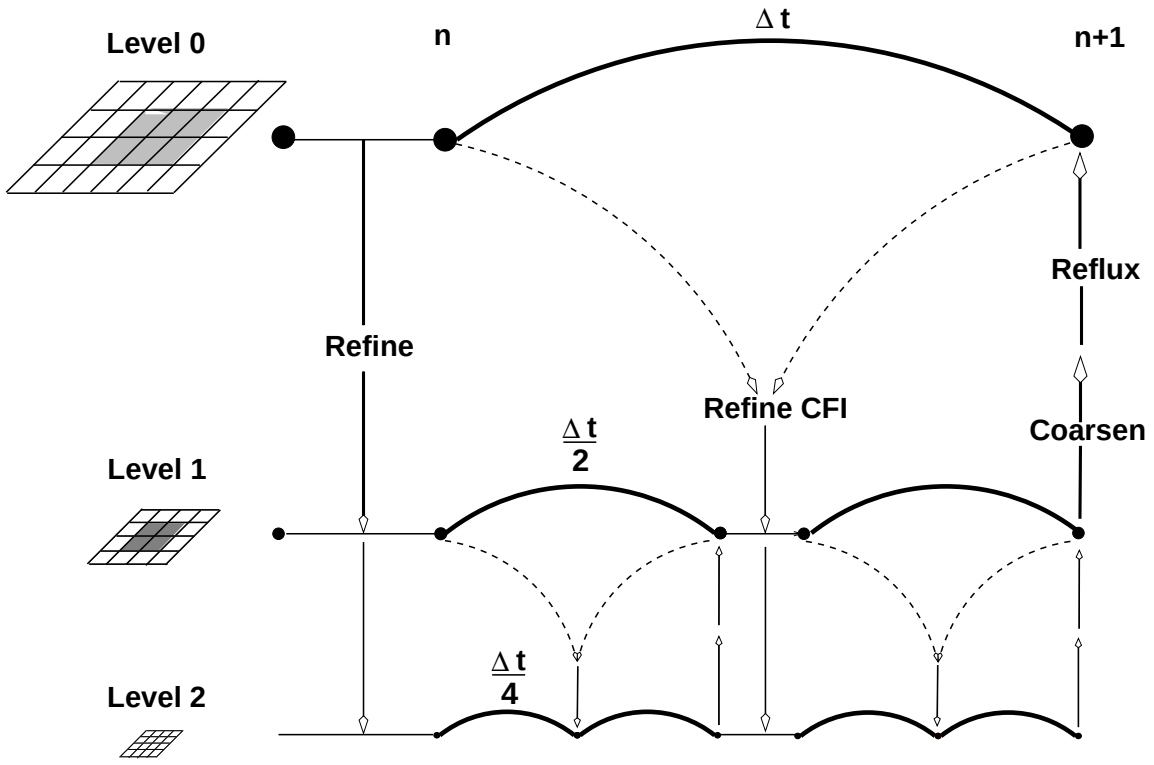



**Figure 3.8**. The W-cycle time integration model

domain.

Regridding is a necessary step in AMR codes. The grid must adapt to fully resolve changing features. A regridder should produce patches that are sized appropriately. If the regridder produces small patches, the number of patches within the domain will be large which can cause significant overhead in other components leading to large inefficiencies. However, it is also important that the regridder does not produce too large of patches because large patches are more challenging to load balance effectively. Thus, an ideal regridder should produce patches that are large enough to keep the number of patches small but small enough to be effectively load balanced.

A common algorithm used in AMR codes for regridding is the Berger-Rigoutsos algorithm [8]. The Berger-Rigoutsos algorithm uses edge detection methods on the refinement flags to determine where to place patches. The patch sets produced by the Berger-Rigoutsos algorithm contain a mixture of both large and small patches. Refinement flags are covered by patches that are as large as possible and the boundaries of the refinement flags are generally covered by smaller patches, producing tight covering

of the refinement flags. The Berger-Rigoutsos algorithm has been implemented in parallel [9, 44, 122]; however, its performance at large numbers of processors is not ideal, as will be discussed in Chapter 4.

The current implementation of Uintah has two constraints on the types of patches that can be produced that prevent direct usage of the Berger-Rigoutsos algorithm. First, patch boundaries must be consistent; i.e., each face of the patch can contain only coarse-fine boundaries or neighboring patches at the same refinement level, and not a mixture of the two. Second, Uintah requires that all patches be at least four cells in each dimension. One way around the neighbor constraint is to locate patches that violate the neighboring constraint and split them into two patches, thus eliminating the constraint violation. Figure 3.9 shows an example of fixing a constraint violation via splitting. In this figure, the left patch set is invalid but can be split to create a valid patch set, as seen in the right patch set.

Unfortunately, splitting patches can also produce patches that violate Uintah's size constraint, as shown in Figure 3.10. In this figure, the left patch set can be split to eliminate the neighbor constraint. However, splitting the patch set produces patches that violate the size constraint.

To make the Berger-Rigoutsos algorithm compatible with Uintah, a minimum patch size of at least four cells in each dimension is specified. The minimum patch size is used to coarsen the flag set by dividing every flag's location by the minimum patch size and rounding down which produces a coarse flag set. The Berger-Rigoutsos

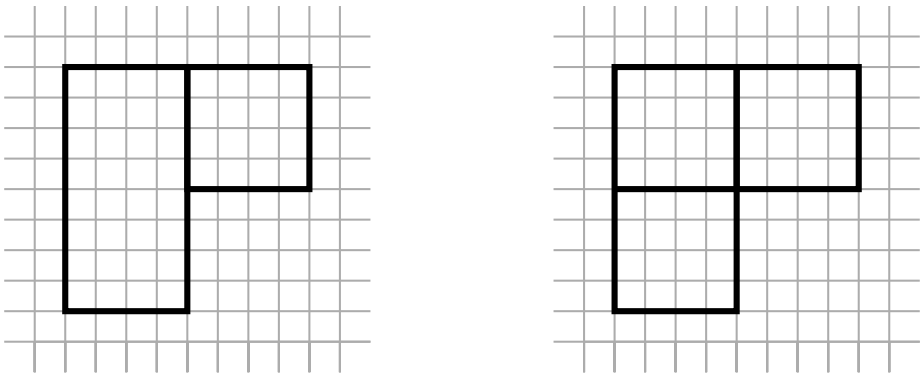


**Figure 3.9**. Patches can be split to satisfy the neighbor constraint

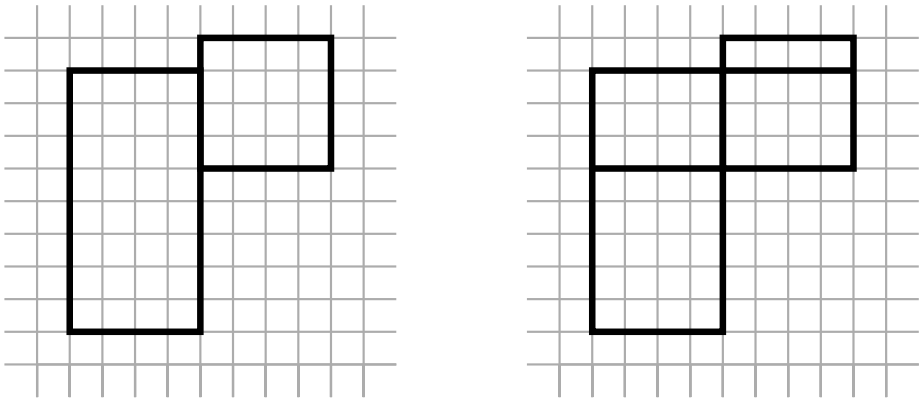


**Figure 3.10**. Splitting patches can violate the size constraint

algorithm is then run on the coarse flag set, producing a coarse patch set. Next, the patch set is searched for neighbor constraint violations and patches are split until the neighbor constraints are met. An additional step is then taken to help facilitate an effective load balance. At this point, patches that are larger than some threshold are split in half, which helps reduce the load imbalance. The threshold used is equal to the average number of cells per processor multiplied by some percentage. We have found through experimentation that a threshold of 6.25% works well. Finally, the coarse patches are projected onto the fine grid by multiplying the location of patches by the minimum patch size, producing a patch set that does not violate the neighbor or the size constraints.

The Berger-Rigoutsos algorithm worked well at 2K processors [78]; however, at 98K processors, the algorithm does not scale well. The lack of scalability led to the development of a tiled regridder within Uintah [75]. This regridder defines a lattice across the domain that defines a set of tiles. When regridding, processors search a subset of the tiles for refinement flags and each tile that contains a refinement flag is added to the patch list. This creates a set of patches that are uniform and meet Uintah's alignment constraints. In addition, the generating the patch set can be done quickly in parallel [75].

The performance characteristics of the Berger-Rigoutsos algorithm and tiled algorithm are discussed in detail in Chapter 4.

### 3.6.4 Load Balancing

A load balancer component is responsible for assigning each patch to processors. In patch-based AMR codes, the load balance of the overall calculation can significantly affect the performance. A load balancer component attempts to distribute work evenly while minimizing communication by placing patches that communicate with each other on the same processor.

The task graph described above provides a mechanism for analyzing the computation and communication within the simulation to distribute work appropriately. One way to load balance effectively is by using this graph directly, where the nodes of the graph are weighted by the amount of work a patch has and the edges are weighted by the cost of the communication to other processors. Graph-based algorithms are then used to determine an optimal patch distribution [61, 92, 96, 97]. These methods do an effective job of distributing work evenly while also minimizing communication; however, they may suffer performance problems at large numbers of patches and processors.

Additional methods for load balancing involve diffusion of computational work between neighboring processors [28, 50, 51, 52]. These algorithms attempt to eliminate load imbalance by migrating excess work to neighboring processors. These algorithms operate in a decentralized fashion which allows migration to occur completely locally.

Other methods attempt to minimize communication by clustering work according to the geometric position within the domain. These methods work well for simulations where communication is predominately local and include methods such as recursive coordinate bisection [9, 96] and space-filling curve partitioning [32, 95, 105]. One advantage to geometric methods is that there is no need to explicitly form the communication graph which may not always be available.

The use of space-filling curves has been shown to work well for many problems [32, 95, 105]. This method utilizes a space-filling curve to create a linear ordering of patches. Locality is maintained within this ordering by the nature of the space-filling curve. That is, patches that are close together on the curve are also close together in space and are likely to communicate with each other. Once the patches are ordered according to the space-filling curve, they are assigned a weight according to a cost

model. The curve is then split into equally weighted segments and each segment is uniquely assigned to a processor.

With AMR, space-filling curves are preferable to the graph-based and matrix-based methods because the curve can be generated quickly in $O(N \log N)$, where N is the number of patches. In addition, the curve generation can be easily parallelized using parallel sorting algorithms [76]. Techniques for effective load balancing effectively are more thoroughly described in Chapter 5.

### 3.6.5 Data Migration

After a new grid is created and load balanced, the data must be migrated to the assigned processors. The data are migrated by creating a task graph for the copying data from the old grid to the new grid. If a region in the new grid is covering a region that did not exist in the old grid, then a refinement task which interpolates data from the coarser level to the finer level is scheduled. The next task is to copy data from the old grid. Any regions within the old grid that overlap the new grid are copied to the new grid. Thus, the infrastructure automatically handles relocating data between processors whenever necessary.

### 3.6.6 Task Graph Compilation

As discussed earlier, the task graph is responsible for determining the order of task execution and communication that needs to occur between them. This information changes whenever the grid or patch assignments change, which requires a recompilation of the task graph. Compiling the task graph can take a significant amount time. In the case of the W-cycle, finer levels iterate multiple times during a single coarse iteration. Each subtimestep has the same data dependencies as the previous subtimestep and thus only needs to be compiled once. By re-using this portion of the task graph, the compile time and memory requirements for the task graph is greatly reduced for the W-cycle.

### 3.6.7 Grid Reuse

Changing the grid and the corresponding steps that follow can be expensive and may not scale as well as the computation. Regridding often can greatly impact the

overall scalability and performance. One way to reduce the overhead associated with changing the grid is to generate grids that can be used for multiple timesteps. This can be done by predicting where refinement will be needed and adding refinement before it is necessary. In Uintah, this is done by dilating the refinement flags prior to regridding. The dilation process modifies the refinement flags by applying a stencil that expands the flags in all directions. The regridder then operates on the dilated refinement flags, producing a grid that is slightly larger than is necessary but can be reused for multiple timesteps. While executing the simulation at each timestep, the undilated refinement flags are compared to the grid. If the refinement flags are contained within the current grid, then regridding is not necessary.

Figure 3.11 shows an example of a grid with and without dilation. The dilated grid is slightly larger but can be reused until the original refinement flags exist outside of the dilated grid.

Dilation can have a large impact on performance when using large numbers of processors. The time to execute the task graph scales better than the time for changing the grid. As the number of processors is increased, the time to change the grid can become the dominant component reducing scalability. Dilating the refinement flags prior to regridding can significantly reduce the time spent associated

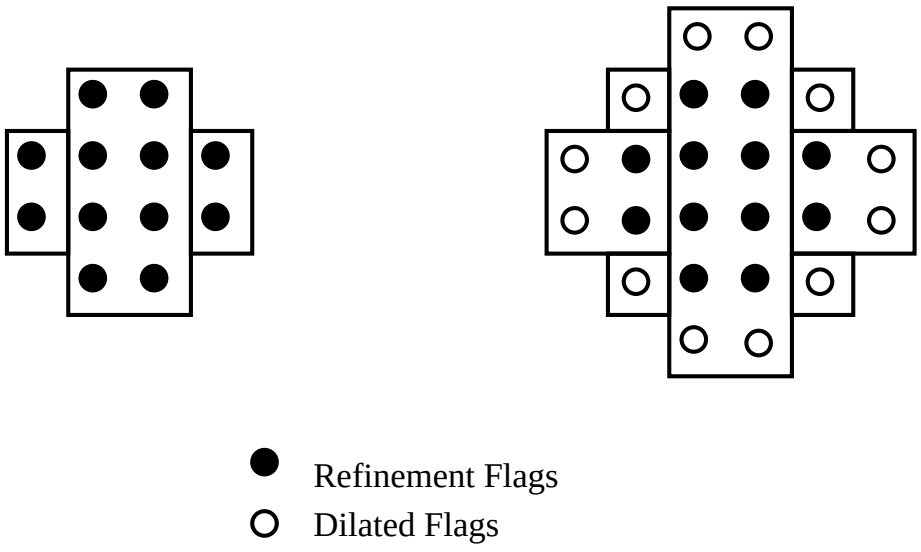



**Figure 3.11**. The effect of dilation on the refinement flags

with changing the grid. This has been shown to greatly increase the performance for problems with rapidly moving fronts [77].

## 3.7 Summary

Uintah is a highly flexible framework which helps facilitate research into many different problems. Through the use of components, Uintah is able to encapsulate and abstract portions of the algorithms so that developers can focus on work within their domain. This allows multiple developers to improve Uintah in parallel.

One of the primary advantages of Uintah is that developers do not need to focus on parallelism. Instead, developers develop their code in serial and it is implicitly parallelized by the framework. The performance of Uintah at large scales will be investigated further in Chapter 6. Uintah also provides mechanisms to output data to disk for visualization and create checkpoints for restarting capabilities. In addition, there is also support for regridding and load balancing for AMR applications.

These features allow developers to create highly scalable AMR simulations in a fraction of the time it would take to develop a standalone application. In addition, as improvements are added to the Uintah framework, all applications developed within the framework will see immediate benefit from those improvements. This makes Uintah a highly useful tool for simulation science applications.

# CHAPTER 4

# REGRIDDING

## 4.1 Introduction

One significant portion of AMR simulations involves dynamically adapting the computational mesh at runtime. Dynamically refining the mesh allows the simulation to reduce error in regions with significant error while maintaining a coarse mesh in other regions. This is particularly important for simulations with moving features. This allows AMR simulations to achieve results similar to a uniformly fixed-mesh simulation while using significantly less computational time and memory. In turn, this allows larger simulations to be undertaken.

A method for block-structured AMR (BSAMR), as outlined in Section 1.3 was originally presented by Berger and Oliger in [8] and then by Berger and Colella in [10]. This algorithm uses various approaches, such as marking regions with large gradients, to flag portions of the mesh that require further refinement. The algorithm then adds patches of refinement that cover the area indicated by the refinement flags through the process of remeshing or regridding. This process is repeated successively on each finer mesh until a desired resolution is reached. This results in a dynamic multilevel grid in which each finer level is nested within a coarser level, as shown in Figure 4.1.

The next advance was a commonly used regridding method which is known as the Berger-Rigoutsos algorithm [11]. This algorithm used edge detection algorithms from image processing to generate a tight-fitting axis-aligned patch-set with relatively few patches. This algorithm was later parallelized in [9] and [122] but both parallel implementations relied substantially on global communication which limited scalability on large numbers of processors [44]. Improvements to this algorithm which focused on reducing the amount of global communication were later presented in [44] and showed scalability to 16K distributed memory processors.

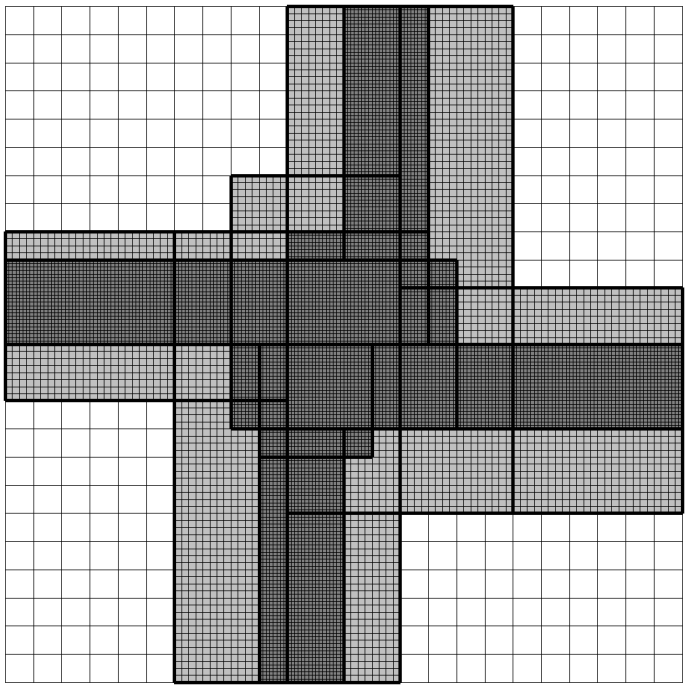


**Figure 4.1**. A multilevel grid of nested levels used within BSAMR

In many adaptive simulations in which the solution involves changing features, such as a moving shock wave, the grid changes as often as every few timesteps. This necessitates that the regridding algorithm run efficiently in parallel. Poor performance within this process can lead to performance problems at large scales [77, 121, 122].

The first exascale machine is expected to arrive within the next decade and it is estimated that it may contain on the order of 100M processors. Achieving scalability on machines of this size will be a challenging task. As we move toward exascale computers, the scalability of these frameworks and the algorithms they use will need to be improved. This chapter analyzes the performance of two existing parallelizations of the Berger-Rigoutsos algorithm, develops a new parallelization of the Berger-Rigoutsos algorithm, and also develops a much simpler alternative regridding algorithm that exhibits ideal scalability. The scalability characteristics of this algorithm make it a candidate for exascale computers at some point in the future.

The situation is summarized by Solomonik and Kale [100] in the context of quite different algorithms but similar architectures: "However newer and much larger

architectures have changed the problem statement further. Therefore traditional approaches...require re-evaluation and improvement." In undertaking an investigation of this type, but for AMR, the regridding algorithms are described in Section 4.2, the characteristics of patch-sets generated are analyzed in Section 4.3, this analysis along with experimental results up to 98,304 processors are used to derive and validate performance models in Section 4.4, and finally, Section 4.5 uses the models to predict the performance of these regridding algorithms on 100M processors.

## 4.2 Regridding Algorithms

In this section, existing BSAMR algorithms are evaluated with regard to their parallel performance. The Berger-Rigoutsos regridding algorithm was originally designed in serial and focused on generating a patch-set that minimized the total amount of refinement while also minimizing the total number of patches [11].

This algorithm has worked well, leading to its widespread use throughout the BSAMR community. Parallel versions of this algorithm were presented in [9] and [122] and improved in [44]. These parallelizations computed parts of the algorithm in parallel and combined the results across processors using all-reduces. An all-reduce is a process whereby data on every processor are combined with data from every other processor. For example, an all-reduction may compute the maximum value across processors or the sum across processors. This version of the Berger-Rigoutsos algorithm has been implemented within SAMRAI and Uintah and within these frameworks, it has been observed that the reduction operations limited the parallel performance of the algorithm at large scales [44, 78]. The original parallelization [9, 122] and the improved parallelization [44] will be referred to as global Berger-Rigoutsos (GBRv1 and GBRv2, respectively). An alternative approach to GBRv1 and GBRv2 is to run Berger-Rigoutsos locally on a subset of the domain. This approach is similar to the approach taken in AMROC [29]. This algorithm will be referred to as LBR. Finally, within Uintah, a Tiled algorithm has been implemented which initially has shown good performance and will be presented throughout this chapter. This method is similar to the method presented in [73] for nonhierarchical grids.

The remainder of this section will describe each of these algorithms in more detail

and provide complexity analysis of their performance, where complexity is defined as the number of operations that the algorithm must execute in order to complete. This analysis will assume that the work per processor is load balanced well, making the time for load imbalance insignificant and that all logarithms are of base two.

Within this section, the following symbols will be used:

$$\begin{aligned} C &= \text{Number of mesh cells in the domain,} \\ F &= \text{Number of refinement flags in the domain,} \\ B &= \text{Number of patches (blocks) in the domain,} \\ P &= \text{Number of processors.} \end{aligned}$$

The Berger-Rigoutsos algorithm requires iterating over the refinement flags. In many frameworks, the refinement flags are stored locally as a boolean per-cell variable, where the boolean value specifies if the cell contains a flag. Iterating over the refinement flags within this data structure would require $O(\frac{C}{P})$ iterations. The cost of iterating over flags can be reduced to $O(\frac{F}{P})$ iterations by first iterating over the per-cell variable and adding all refinement flags to a list and then using that list within the Berger-Rigoutsos algorithm. The cost of generating this list in parallel is $O(\frac{C}{P})$.

It is worth noting that the GBRv2, LBR, and Tiled algorithms generate a local patch-set, meaning that each processor only has knowledge of a subset of the patches. Generally today's frameworks, such as Uintah, SAMRAI [49], and Enzo [85], require each processor to be aware of the global patch set. These frameworks would thus require an all-gather on the local patch-sets to generate the global patch-set. An all-gather is a process which combines local arrays on each processor into a global array on each processor and it is commonly done through the MPI_Allgather function. The analysis below does not include the time for creating the list of flags or for performing the all-gather which may be significant at large numbers of processors. The issue of the time required for these operations will be revisited in Section 4.4.

### 4.2.1 Serial Berger-Rigoutsos

The following presents a high level description of the serial Berger-Rigoutsos algorithm (SBR). For brevity, a complete description of this algorithm has been omitted but can be found in [11]. The Berger-Rigoutsos algorithm can be defined recursively, as shown in Algorithm 1.

**Algorithm 1** The Serial Berger Rigoutsos Algorithm

```
BRSplit(INPUT: flags, OUTPUT: patches)
    //compute the bounding box of the flags
    BB=computeBoundingBox(flags)

    //check the termination condition
    IF flags.size/BB.size > tolerance
        patches.add(BB)
        RETURN
    END IF

    //compute the refinement histogram for each dimension
    histogram=computeHistogram(flags)

    //split the domain into left and right halves
    [left,right]=split(flags,histogram)

    //recursively continue this process
    BRSplit(left,patches)
    BRSplit(right,patches)
```

The first step of the algorithm is to compute the bounding box of the refinement flags list [$O(F)$]. The second step is to check the termination condition by comparing the number of cells in the bounding box to the number of flags [$O(1)$]. The third step is to compute a histogram of the spatial location for each refinement flag [$O(F)$]. A histogram is created for each dimension, yielding X,Y, and Z histograms. Next, each histogram is used to determine the best location to split the domain [$O(\sqrt[3]{F})$]. Then, the flags are divided into left and right halves [$O(F)$]. Finally, the algorithm recursively repeats on each half. The recursion implicitly defines a recursive tree in

which parent nodes are split into two children nodes. In the best case, the recursive tree is a complete tree in which every parent has two children except perhaps the lowest level of the tree. In this case, the recursion would repeat $O(\log B)$ times. If the tree were largely unbalanced, then in the worst case, the algorithm could repeat up to $O(B)$ times. In practice, the algorithm generally behaves similarly to the best case, as shown in Equation 4.1.

$$O(\text{SBR}) = F \log B. \tag{4.1}$$

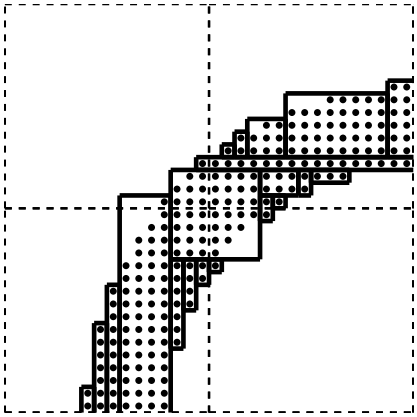
Figure 4.2 (a) shows an example of a patch-set generated by the SBR algorithm. The patches fit the refinement flags fairly tightly and are irregular in size.

### 4.2.2 Parallel Global Berger-Rigoutsos Algorithm

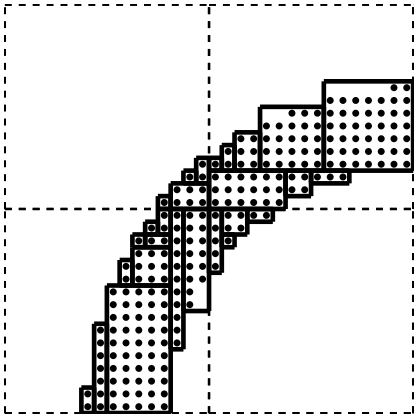
The GBR algorithm is implemented in parallel by bounding the refinement flags and generating the histograms in parallel [122, 44]. To do this, each processor computes the bounds and the histogram on a subset of the refinement flags and then combines the results with every other process through an all-reduce operation. In this analysis, the time to perform the all-reduce will be referred to as the communication time and the time for the rest of the algorithm will be referred to as the computation time.

The time for the computation of GBR is equivalent to the serial algorithm performed on a subset of the flags. Consequently, the complexity for the computational portion of the GBR algorithm is $O(\frac{F}{P} \log B)$.

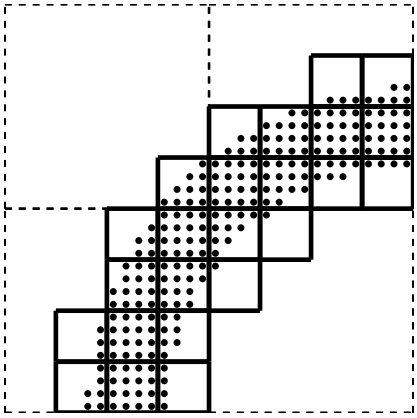
The communication step in GBR involves performing two all-reduces at each node and leaf of the recursive tree. The parallel complexity for an all-reduce operation is not straightforward to define as it varies depending on the network topology and the MPI library implementation. Within some MPI libraries, the algorithm used may vary according to the size of the reduction and the number of processors. In most cases, the time for an all-reduce is dominated by message latency and thus, the time for an all-reduce is proportional to the number of pairwise communications required to reduce the data, which requires a minimum of $\log P$ pairwise messages.



(a) SBR



(b) LBR



(c) Tiled

**Figure 4.2**. Patch-sets generated by the SBR (a), LBR (b), and Tiled (c) algorithms; the coarse level patches are represented by a dashed line and the fine level patches are represented by a solid line. The SBR and LBR algorithms use less patches but generate irregular patch sets. For the LBR algorithm, boundaries of the coarse level also exist on the fine level. The Tiled algorithm generates regular patches.

In the best case, the recursive tree of the Berger-Rigoutsos algorithm is completely balanced and the number of nodes is equal to $\sum_{k=0}^{\log B} 2^k$ which is the sum of a geometric series and is equal to $2B - 1$, making the number of messages equal to

$$M(\text{GBRv1}) = (2B - 1) \log P. \tag{4.2}$$

The overall complexity for GBRv1 is then:

$$O(\text{GBRv1}) = \frac{F}{P} \log B + B \log P. \tag{4.3}$$

The improvements to this algorithm presented in [44] involve reducing the number of processors that contribute to the reductions at each level of the recursion. At the first level, every processor contributes and at each successive level, the number of contributing processors decreases until the algorithm terminates or only a single processor is contributing to each reduction, at which point the algorithm completes in serial. In the best case, the number of contributing processors at each node would be half of the number of contributing processors of the parent node. This leads to a total number of messages of

$$\sum_{k=0}^{\min(\log B, \log P - 1)} (2^k \log \frac{P}{2^k}). \tag{4.4}$$

Refactoring the equation above yields

$$\sum_{k=0}^{\min(\log B, \log P - 1)} 2^k \log P - \sum_{k=0}^{\min(\log B, \log P - 1)} k 2^k. \tag{4.5}$$

By substituting these summations with their known solutions found in [27] and performing algebraic simplification, we set that the number of messages is equal to

$$M(\text{GBRv2}) = \begin{cases} 2P - \log P - 2 & : P < B \\ (2B-1)\log P - 2B\log B & \\ +2B - 2 & : P >= B. \end{cases} \tag{4.6}$$

Figure 4.3 shows the number of messages generated for both GBR algorithms when generating 1000 patches. This figure shows that GBRv2 has a significant improvement over GBRv1 when the number of patches is less then the number of processors. However, when this is not the case, the number of messages generated by the two algorithms are similar.

Thus, GBRv2 has an overall complexity of

$$O(\text{GBRv2}) = \begin{cases} \frac{F}{P}\log B + P & : P < B \\ \frac{F}{P}\log B + B\log P & : P >= B. \end{cases} \tag{4.7}$$

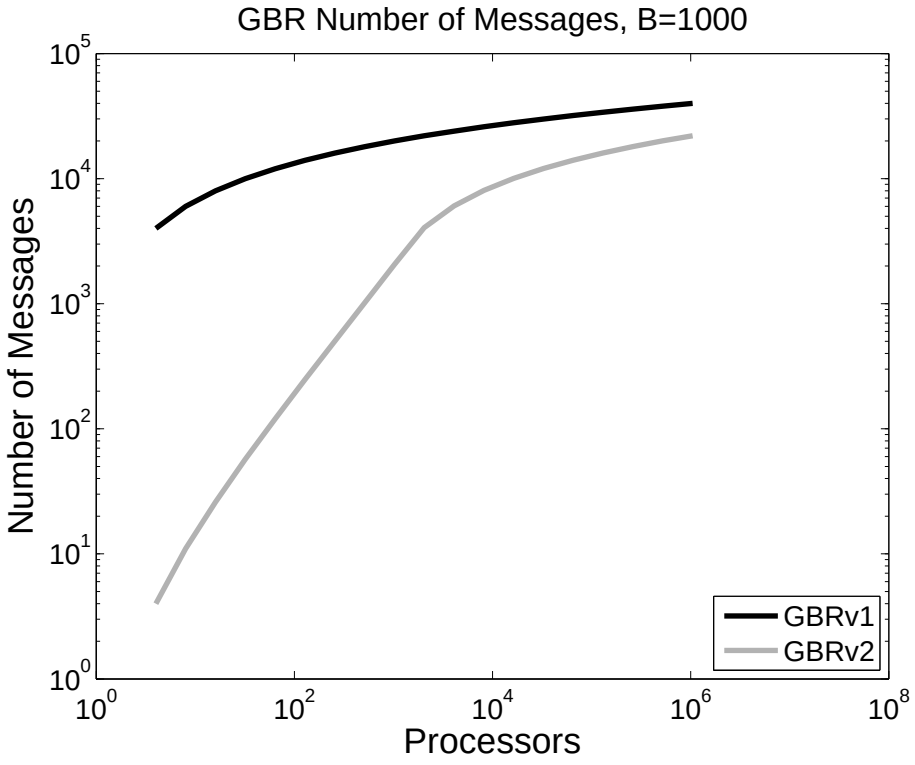



**Figure 4.3**. The number of messages required to execute GBR

From this, we can see that when $P < B$, the communication complexity is $O(P)$; however, when $P \geq B$, the communication complexity is $O(B \log P)$ which is the same as GBRv1. The GBRv2 algorithm replaces the all-reduces in the GBRv1 algorithm with pairwise asynchronous communication performed along the edges of a hypercube. In addition, the algorithm is decomposed into tasks, where each task represents a node of the recursive tree. The tasks are designed such that they are restartable. When a task blocks for communication, it cedes control of the processor so that the next scheduled tasks can execute. Once communication is completed, that task is once again scheduled for execution and will restart where it left off. When there are enough tasks, the communication can be overlapped with computation leading to potentially further increases in performance. These optimizations within the GBRv2 algorithm are complex and can be difficult to implement correctly.

The analysis above did not consider the case where communication is overlapped with computation in the GBRv2 algorithm and thus should be considered an upper bound. Details of this algorithm can be found in [44].

### 4.2.3 Local Berger-Rigoutsos Algorithm

The LBR algorithm performs SBR locally on each coarse patch to generate finer patches that are nested within the coarse patch. This provides an advantage over GBR in that no communication is required to bound the refinement flags or generate the histogram. However, since this algorithm is running locally on each patch, the patches on the finer level cannot span across a coarse patch. This causes the boundary of the coarse patches to be projected onto finer patches, thereby increasing the total number of patches. This is similar to an algorithm implemented within AMROC [29]; however, that algorithm runs SBR locally on the entire local patch set and then intersects the resulting patch set with the original patches. This generates more patches than the SBR algorithm but less patches than the LBR algorithm. In addition, this results in patch sets that change as the number of processors changes, which may be an undesirable property as the computational solution should generally be independent of the number or processors.

The complexity for LBR (assuming that patches are fairly evenly sized) is defined

as follows. Let $B_c$ be the number of coarse patches. The flags per coarse patch is then $\frac{F}{B_c}$, the number of coarse patches per processor is then $\frac{B_c}{P}$, and the number of fine patches per coarse patch is then $\frac{B}{B_c}$. Thus, the complexity for this operation is

$$O(\text{LBR}) = \frac{B_c}{P}\frac{F}{B_c}\log\frac{B}{B_c} \quad (4.8)$$

$$= O(\frac{F}{P}\log\frac{B}{B_c}). \quad (4.9)$$

AMROC's implementation of this algorithm has a similar complexity. Figure 4.2 (b) shows a patch-set generated using the LBR algorithm. This figure shows the patch set is similar to SBR and that the boundaries of the coarse patches exist on the finer level.

### 4.2.4 Tiled Algorithm

The Tiled algorithm was developed within Uintah to eliminate the performance problems associated with the GBR algorithms. This algorithm is defined in Algorithm 2.

**Algorithm 2** The Tiled Regridding Algorithm

```
FOR each local tile
    FOR each cell in tile
        IF cell has refinement flag
           patches.add(tile)
           BREAK
        END IF
    END FOR
END FOR
```

This algorithm defines a set of tiles using a lattice where each lattice cell corresponds to a possible patch. To compute the patch-set, each processor searches a subset of the tiles for refinement flags; when a refinement flag is found, the tile is added to the patch-set and the next tile is searched for refinement flags. A possible

disadvantage of this approach is that it may result in more refinement or more patches than the GBR and LBR algorithms; this is discussed in Section 4.3.

Similarly to the LBR algorithm, this algorithm requires no communication. In addition, this algorithm is embarrassingly parallel and has a complexity of

$$O(\text{Tiled}) = \frac{C}{P}. \tag{4.10}$$

Figure 4.2 (c) shows an example of a patch-set generated by the Tiled regridder. In this case, the algorithm uses more patches than the Berger-Rigoutsos algorithms; however, the patches generated are regular. If the algorithm is generating too many patches, which are causing inefficiencies elsewhere in the code, then the tile size could be increased, which would thereby decrease the number of patches. The grids generated by the Tiled algorithm often look similar to the grids produced using OSAMR; however, it is not the case that they are the same. For example, OSAMR patches are constrained to a power of two in size whereas patches produced by the Tiled algorithm are not.

## 4.3 Patch-Set Characteristics

The characteristics of the patch-sets generated by these algorithms is important. The following will describe and analyze the differences between the patch sets generated by the SBR, LBR, and Tiled algorithms.

The original Berger-Rigoutsos algorithm generates patch-sets of varying sizes. Using this method, it is not uncommon to have patches that span large portions of the domain next to patches that only span a few cells which can complicate the load balancing process. The irregularity of the patches necessitates a general framework. Some frameworks, such as Uintah, restrict the types of patch-sets to simplify other portions of the framework. These restrictions require a cleanup phase after regridding to ensure the grid constraints are met [78].

The GBRv1 and GBRv2 algorithms generate patch-sets that are equivalent to the SBR algorithm. Like the SBR algorithm, the LBR algorithm generates irregular patches; however, it will also generate more patches than the SBR algorithm. Finally,

the Tiled algorithm may generate more patches but the patches are regular. This regularity can be exploited to provide further increases in performance. The following will discuss the advantages and disadvantages of the number of patches, tightness of fit, and regularity of the patch-sets.

Each of the regridders above were implemented as a stand-alone program as described in their original papers and the patch sets generated by each regridder were tested with a benchmarking problem. This problem defined a 3D domain in the range of [0,1]. Within this domain, refinement flags were added in the region between two concentric spheres centered at [0.5,0.5] with radii 0.3 and 0.4. The resolution of the grid was increased and the patch sets generated were recorded. The coarse level was split into square patches of $16^3$ cells and the tile size was set to $16^3$ for the Tiled regridder. The Berger-Rigoutsos-based algorithms used a tolerance parameter of .85, meaning that the recursion terminated when 85% of the patch contained refinement flags. This problem was chosen as it closely resembles an expanding blastwave that is commonly simulated within Uintah [77, 78]. Figure 4.2 shows the corner of a 2D version of this test problem along with the meshes generated by the three regridding algorithms.

### 4.3.1 Number of Patches

Increasing the number of patches also decreases the average size of a patch for a fixed size mesh. Depending on the underlying framework, there may be a significant overhead per patch. If the overhead is significant, then increasing the number of patches may hinder performance, as shown in [13]. In addition, some data structures may depend on the total number of patches. For example, Uintah and SAMRAI utilize tree-based data structures for neighbor finding [122]. As the number of patches increases, the size of these data structures and the time to query them will also increase.

While increasing the number of patches may decrease performance, it can also potentially increase parallel performance. Decreasing the average size of a patch often decreases the load imbalance which leads to an increase in performance. In addition, having more patches on each processor may allow the simulation to proceed in a more

asynchronous manner, reducing the synchronization and thereby overall simulation time [83]. Finally, decreasing the patch size may also improve cache performance. Whether or not increasing the number of patches will increase or decrease performance depends on the framework. Within Uintah, we have found that increasing the number of patches often increases the overall performance. However, the performance begins to decrease once patches become too small [83].

Figure 4.4 shows the number of patches generated by each regridder for the benchmark problem. In this graph, the points are the recorded data and the lines are linear least squares fits to the data for the curve $C^m$. The slope of these lines correspond to rate at which the number of patches grows relative to the number of cells. This graph shows that the number of patches using the Tiled algorithm is directly proportional to the number of cells. The SBR algorithm grows at $\approx C^{.5}$ and

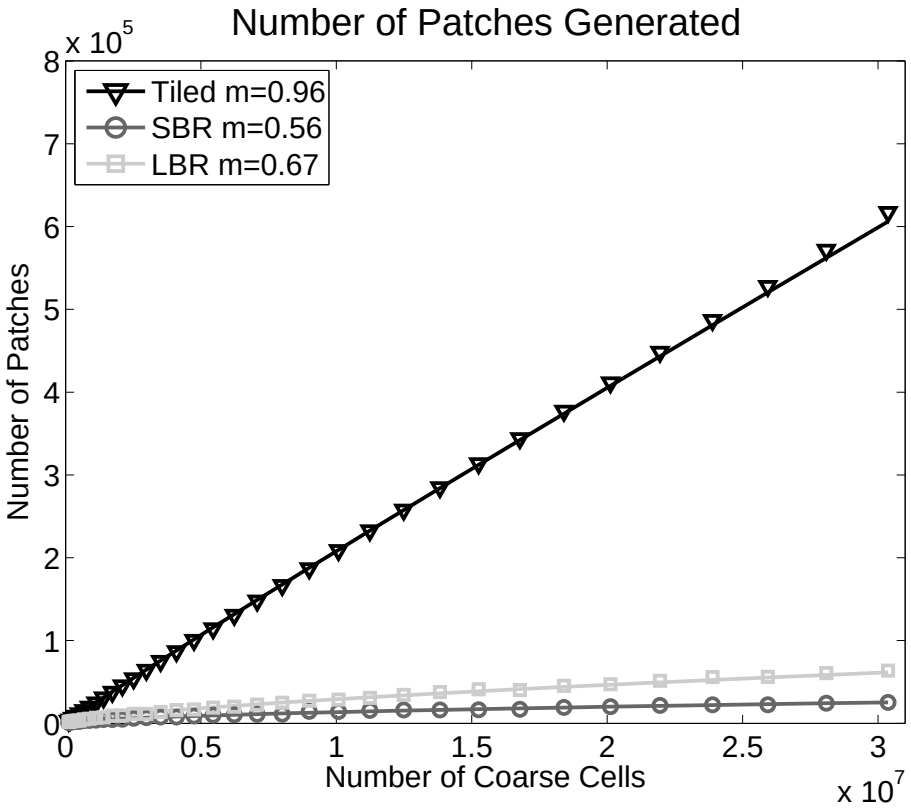



**Figure 4.4**. The number of patches generated by the regridding algorithms. The points are the recorded data and the lines are a linear least squares fit of that data within a loglog plot.

the LBR grows at $\approx C^{.7}$. The Tiled algorithm uses a factor of ten more patches at the largest scales.

For scalability at large numbers of processors, it is desirable that the number of patches scale proportionally to the number of cells. Since the SBR and LBR algorithms do not exhibit this property, they will require an extra step for large problems at large numbers of processors. This step will need to split the largest patches until every processor has at least one patch. One efficient way to do this would be to split every patch that was larger than some percent of the total volume of the new grid. This would take $O(\log P)$ time to compute the total volume using an all-reduce and $O(\frac{B}{P})$ time to split the patch set, leading to an overall complexity of

$$O(\text{Split}) = \frac{B}{P} + \log P. \tag{4.11}$$

The Tiled algorithm does not require this step. The time for this step is investigated in Section 4.4.

### 4.3.2 Tightness of Fit

Having a tight fitting patch-set ensures a minimal amount of unnecessary refinement, thereby reducing the overall runtime. However, in applications that regrid often, a tight patch-set may actually hinder performance. For these applications, the cost of regridding, load balancing, and scheduling can become significant. A loose patch-set can reduce the frequency of these operations. Recognizing this, Uintah uses dilation, as described in Section 3.6.7, to increase the area of the patch-set. This has provided a large reduction in the time spent in AMR simulations involving rapidly moving fronts [78].

Figure 4.5 shows the over-refinement as a percentage of the number of refinement flags for the regridding algorithms, where the over-refinement, $OR(F, C)$ is defined as

$$OR(F, C) = (\frac{C}{F} - 1) \times 100. \tag{4.12}$$

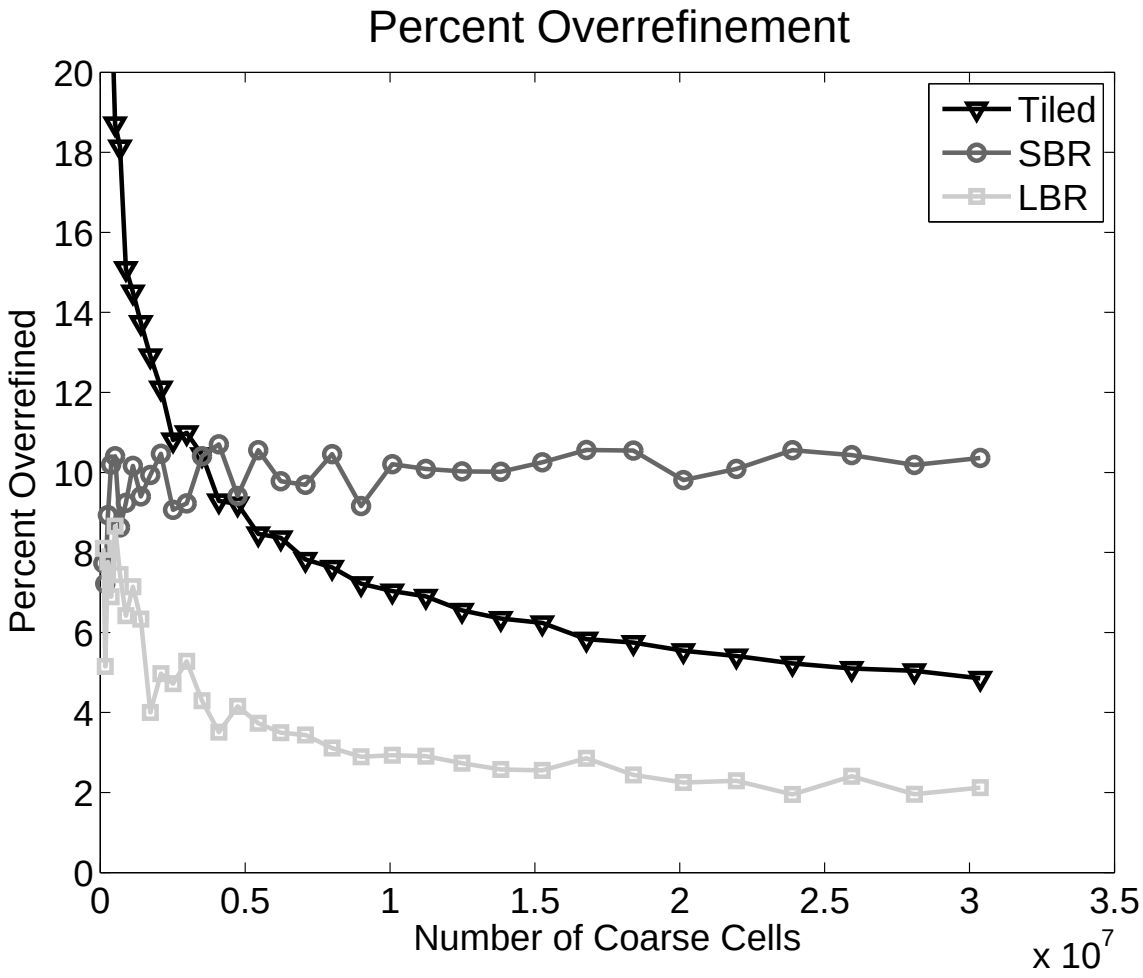



**Figure 4.5**. The over-refinement of the generated mesh as a percentage of refinement flags

This graph shows that for the SBR algorithm, over-refinement increases with the problem size. This is due to large patches that cover a large number of refinement flags but also cover a large number of cells without refinement flags. This can be seen in the left image in Figure 4.2. This figure shows a large amount of over-refinement within the large vertical patch. The increase in over-refinement should level off at or below the tolerance parameter (15% in this case). The same increase was not seen with the LBR regridder. This is because the sizes of the fine patches are limited by the size of the coarse patches. The over-refinement of both the LBR and Tiled algorithms decreases as the number of coarse cells increases. This reduction is due to a reduction in the size of a patch relative to the size of the features being tracked. As this size is decreased, the patches more tightly cover the tracked feature, thus reducing the over-refinement. The SBR algorithm generates the most over-refinement at larger problem sizes, followed by the Tiled algorithm, and then the LBR algorithms.

### 4.3.3 Regularity

Regularity of patch size within patch-sets is a desirable quality as irregular patches can cause load imbalance. In particular, having a large variance in the size of the patches is potentially problematic as load balancing regular sized patches is much more straightforward. In addition, having long slender patches next to many small patches produces load imbalance in the communication as the large patches must communicate to many neighbors but the small patches only have to communicate with a few neighbors.

In addition, if patches are completely regular, the regularity can be exploited to simplify other portions of the algorithm. For example, neighbor finding on irregular patch-sets generally utilizes a tree algorithm, like a Bounding Volume Hierarchy (BVH) [67], which has an $O(\log B)$ query time. When the patches are regular, the time for this query can be reduced to $O(1)$ through the use of a hash table [27]. Table 4.1 shows statistics on the number of cells within patches generated by the benchmark problem. This table shows that the SBR algorithm has the greatest variance, followed closely by the LBR algorithm. The Tiled algorithm has no variance. The SBR algorithm produces the largest patches, followed closely by the LBR

**Table 4.1**. Statistics on the number of cells per patch generated by the regridding algorithms.

| | Mean | Min | Max | Stdev |
|---|---|---|---|---|
| SBR | $1.1 \cdot 10^4$ | 64 | $3.1 \cdot 10^6$ | $9.0 \cdot 10^4$ |
| LBR | $4.2 \cdot 10^3$ | 64 | $3.3 \cdot 10^4$ | $1.0 \cdot 10^4$ |
| Tiled | 512 | 512 | 512 | 0 |

algorithm. The SBR and LBR algorithms also produce the smallest patches.

### 4.3.4 Summary of Patch-Set Quality

Figures 4.4 and 4.5 along with Table 4.1 show that the SBR algorithm generates the least number of patches; this, however, comes at the cost of some over-refinement and irregular patch sizes. The LBR generates slightly more patches then the SBR but has significantly less over-refinement. The Tiled regridder generates the most patches by far and has less over-refinement then the SBR algorithm. The SBR and LBR algorithms may not generate enough patches for large numbers of processors and will thus require an extra step that splits large patches. The time for this operation may become significant at large numbers of processors. The Tiled regridder will not have this problem since the number of patches scales linearly with the size of the coarse grid.

It is not clear if any type of patch-set is the best overall. Instead, it appears that the best type of patch-set depends on the framework and the application. If the framework has a low per-patch overhead and can exploit regularity, it would appear that the patch-sets provided by the Tiled regridder would be best. However, if a large patch overhead exists and load imbalance is not a problem, then the patch-sets generated by the LBR algorithm may be the best.

## 4.4 Parallel Performance

This section evaluates the parallel performance of the regridding algorithms using performance models derived from the analysis above and experiments on Kraken[1]

[1]Kraken is a NSF supercomputer located at the University of Tennessee with 99,072 processing cores. More information is available at `http://www.nics.tennessee.edu/computing-resources/`

with up to 98304 processors. The test problem was the same problem as described in Section 4.3. For these results, each algorithm was tested 500 times and the average time was reported. The GBR algorithms were each executed only 5 times due to the amount of time required to run these algorithms and constraints on our machine usage. Finally, the GBR algorithms were not run at the largest numbers of processors due to this same constraint.

### 4.4.1 Performance Models

Using Equations (4.3), (4.7), (4.9), (4.10), (4.11), and the analysis in Section 4.2, the following performance models are defined:

$$\begin{aligned} T(\text{GBRv1}) &= c_1 \frac{F}{P} \log(B) + c_2 M(\text{GBRv1}), \\ T(\text{GBRv2}) &= c_1 \frac{F}{P} \log(B) + c_3 M(\text{GBRv2}), \\ T(\text{LBR}) &= c_4 \frac{F}{P} log(\frac{B}{B_c}), \\ T(\text{Tiled}) &= c_5 \frac{C}{P}, \\ T(\text{Split}) &= c_6 \frac{B}{P} + c_7 \log(P), \end{aligned}$$

where $c_1 = 10^{-6}$, $c_2 = 2 \cdot 10^{-4}$, $c_3 = 4 \cdot 10^{-5}$, $c_4 = 2.5 \cdot 10^{-8}$, $c_5 = 10^{-8}$, $c_6 = 4 \cdot 10^{-8}$, and $c_7 = 2 \cdot 10^{-5}$ were chosen to match experimental data. In addition, the following models for F and B were used:

$$\begin{aligned} B &= C^m, \\ F &= .154C, \end{aligned}$$

where m is 0.56, 0.67, and 0.96 for the GBR, LBR, and Tiled algorithms, respectively, as defined in Figure 4.4.

---

`kraken`.

### 4.4.2 Parallel Scalability

Figure 4.6 shows the theoretical and experimental strong scalability for the various regridding algorithms. For this graph, $B_c = 131072$ and $C = 4096B_c$. These parameters correspond to having around one $16^3$ coarse patch per processor at $98,304$ processors. These parameters were chosen because they closely resemble simulations that have been and currently are being investigated [77, 78]. In this figure, the dashed line is the theoretical performance and the points are the experimental data. This graph shows that the models are a very good fit to the experimental results. The GBRv2 model is not as accurate as the others because the model did not take into account the possible overlap of communication and computation. Ideal strong scaling is shown as a diagonal line with a slope of -1 which is shown by both the LBR and Tiled algorithms. The GBRv1 algorithm scales poorly. The GBRv2 algorithm strong scales well until the communication dominates the runtime, at which point the algorithm fails to scale. The splitting algorithm scales but eventually stops scaling due to the cost of the reduction. The time to split patches eventually exceeds the time to execute the LBR algorithm and will eventually limit the total scalability of the LBR algorithm.

The weak scalability for the regridding algorithms are shown in Figure 4.7. For this graph, $C = 16384P$ and $B_C = \frac{C}{4096}$, which corresponds to each processor having four $16^3$ coarse patches. In this graph, ideal weak scaling is shown as a horizontal line. The models are a good match to the experimental results in all cases. This graph shows GBRv1 and GBRv2 again scale poorly. The Tiled regridder shows ideal weak scaling and the LBR regridder shows better than ideal weak scaling. This is because the ratio $\frac{B_c}{B}$ in Equation (4.9) is decreasing as the problem size gets larger. The splitting algorithm does not scale well and eventually takes longer than the LBR algorithm.

To scale to petascale and eventually exascale machines, applications must exhibit good weak scaling. This analysis has shown that both GBR algorithms do not perform well at large numbers of processors and will limit the scalability of the application. This is due to the cost of combining global data that is inherent in those algorithms. Both the Tiled and LBR algorithms avoid this bottleneck by only operating on local

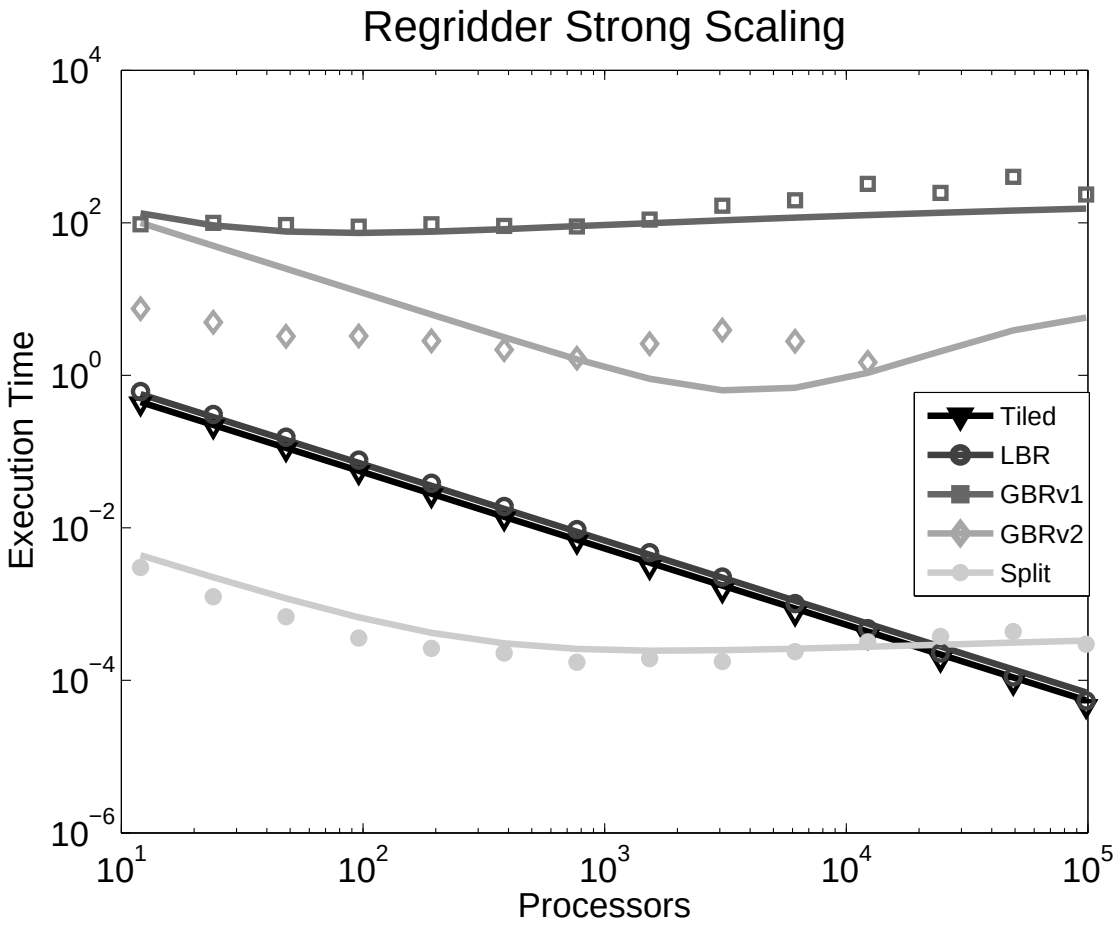



**Figure 4.6**. The strong scalability of the regridder algorithms. The points represent the experimental data and the line represents the model. Ideal strong scaling is shown by a line with a slope of -1.

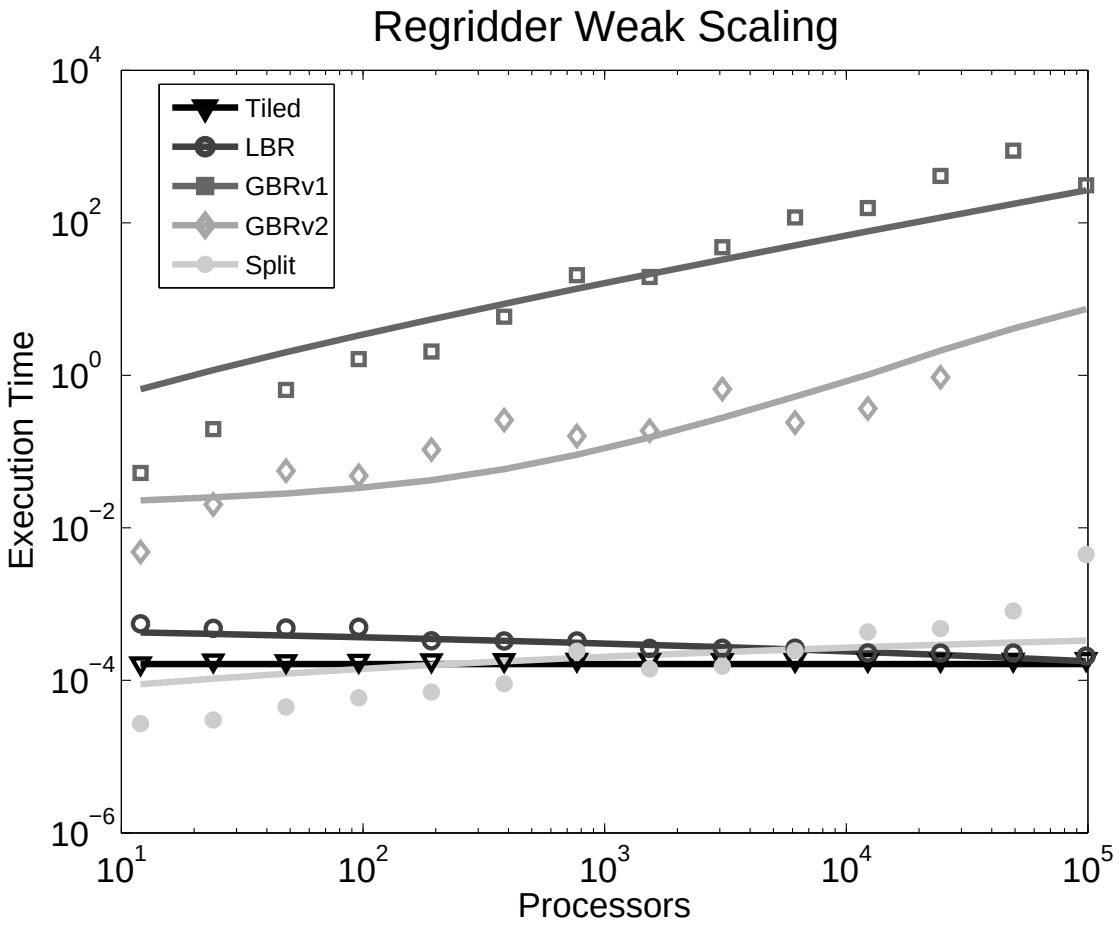



**Figure 4.7**. The weak scalability for the regriddering algorithms. The points represent the experimental data and the line represents the model. Ideal weak scaling is shown by a horizontal line.

data.

Figure 4.8 shows the experimental time for generating the flags list and performing an all-gather on the patch sets. The experimental times for the Tiled algorithm have also been included on this graph to provide a reference point. This graph shows that the time to create the flags list scales well but is slightly slower than the Tiled algorithm. In addition, the cost to all-gather the final patch set is substantially higher than the cost of regridding. This shows that maintaining global metadata for the patch sets will become a bottle neck at sufficiently large numbers of processors. Thus, frameworks that currently utilize this global metadata will eventually need to find ways to eliminate this requirement by moving to local algorithms.

## 4.5 Exascale Performance

It is expected that exascale computers will start appearing within the next decade. It is uncertain what architecture the first exascale machines will utilize, though it is not unreasonable to assume that such machines may include upwards of 100M processors. Using the models presented earlier, we will now speculate what the performance of these algorithms would be on a machine similar to Kraken with 100M processors. The time to split the patches for the GBR and LBR algorithms has been included in the projected times.

Figure 4.9 shows the theoretical strong scaling performance up to 100M processors. In this graph, $B_c = 1073741824$, $C = 4096B_c$, and $m = 0.7$ which corresponds to having around one $16^3$ coarse patch per processor at 100,000,000 processors. This graph shows that the LBR algorithm scales well but tails off as the number of patches per processor becomes low due to the cost of splitting patches to ensure there is at least one patch per processor. Since splitting does not scale ideally, it eventually becomes a bottleneck, thereby limiting strong scaling. The Tiled algorithm continues to show ideal strong scaling. The GBR algorithms scale well at smaller numbers of processors but do not scale well at the largest numbers of processors. The GBR algorithms also take significantly longer than the LBR and Tiled algorithms but create fewer patches.

Figure 4.10 shows the theoretical weak scaling performance out to 100M processors. For this graph, $C = 16384P$, $B_C = \frac{C}{4096}$, and $m = 0.7$, which corresponds to

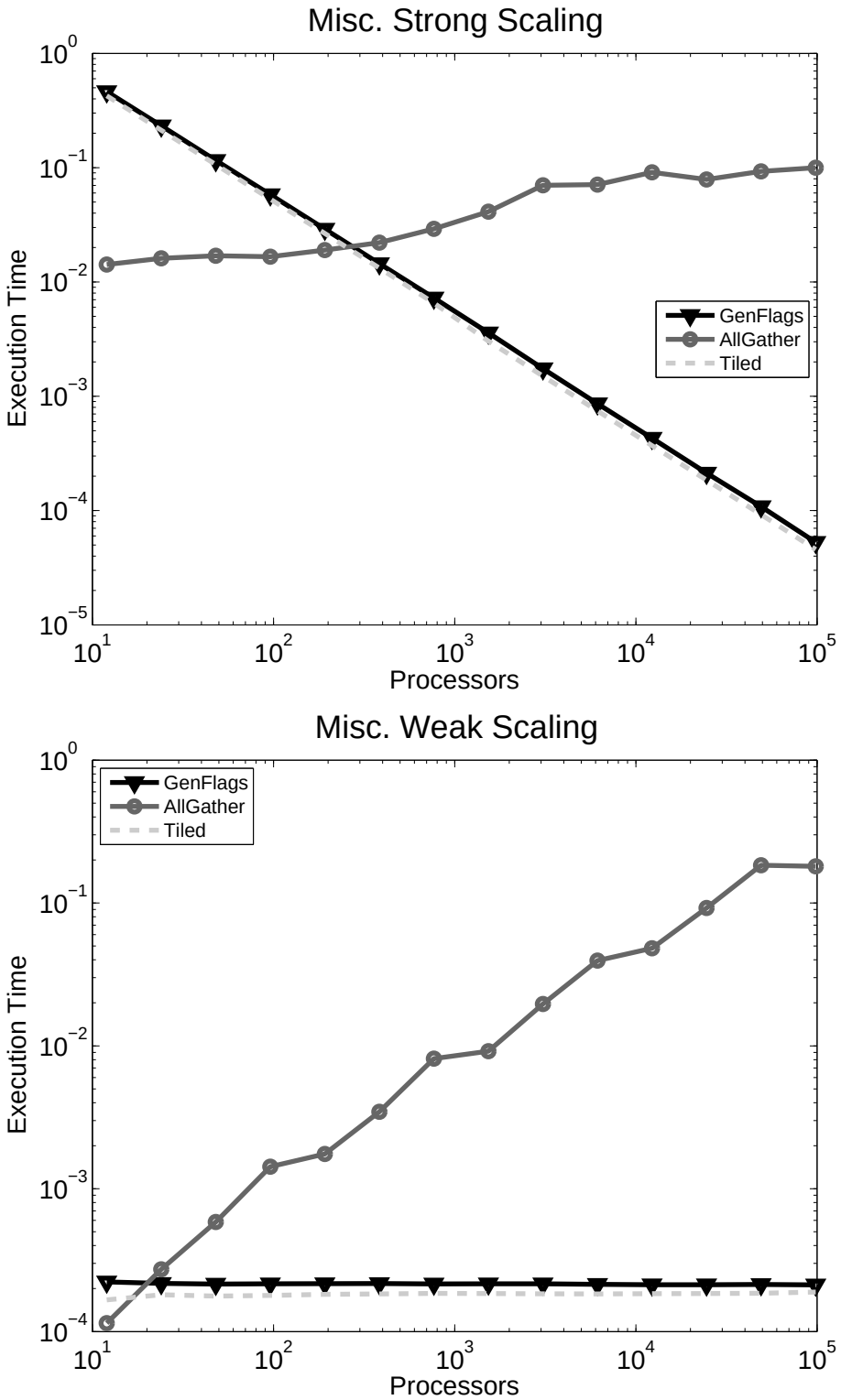



**Figure 4.8**. The weak scalability for miscellaneous regridder operations. Generating the flags list scales well; however, all-gathering the patch set does not.

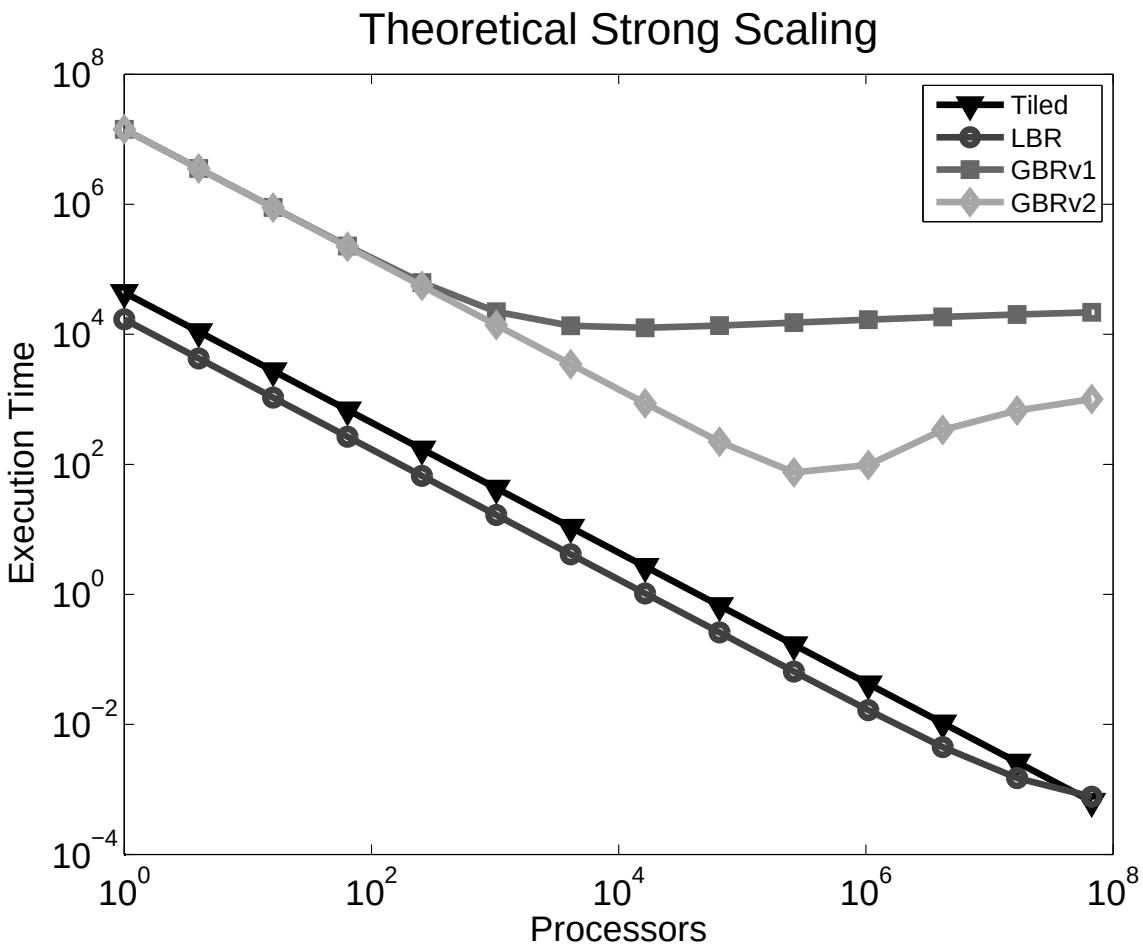



**Figure 4.9**. The theoretical strong scaling to 100M processors. The GBR algorithms do not strong scale well while the LBR and the Tiled algorithms both strong well.

each processor having four $16^3$ coarse patches. This graph shows both the LBR and the Tiled algorithms weak scale well. In this case, LBR scales well because the $\log(P)$ term in the splitting algorithm is fairly flat at large numbers of processors. The GBR algorithms do not scale well at very large numbers of processors.

## 4.6 Summary and Conclusions

To use petascale machines effectively, the algorithms we use will have to exhibit nearly ideal weak scaling. We have shown that algorithms that utilize global metadata do not scale well in the strong or weak sense and are not feasible for large scale machines. We have presented two new local algorithms which both show promising weak

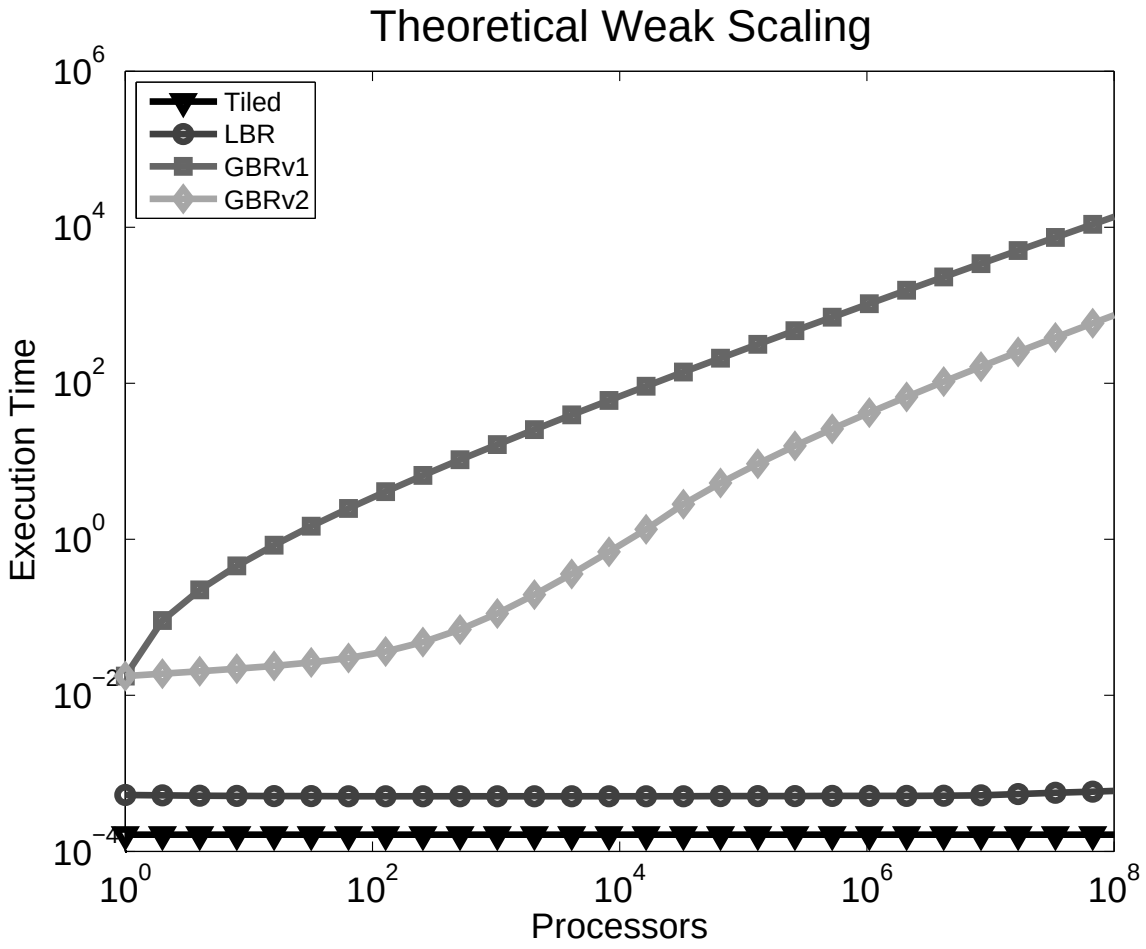



**Figure 4.10**. The theoretical weak scaling to 100M processors. The GBR algorithms do not weak scale well while the LBR and the Tiled algorithms both weak scale well.

and strong scalability up to 98,304 processors. In addition, we have presented analysis of these algorithms which shows they should weak scale up to 100M processors.

It is commonly assumed having fewer less patches is better than having more. However, when weak scaling, we have shown that if the number of patches does not scale linearly with the volume of the domain, then a splitting algorithm will be required to ensure there is at least one patch per processor. This is the case with the Berger-Rigoutsos algorithms. This implies that the extra patches generated using the LBR algorithm over the GBR algorithms should not be a problem. In addition, the number of patches produced by the Tiled algorithm scales linearly with the size of the domain and thus will not require the splitting algorithm.

The irregularity of the patch sets produced by the GBR and LBR algorithms may be problematic, especially in terms of load imbalance. All of the algorithms generate tight-fitting patch-sets, thereby limiting over-refinement. The Tiled algorithm generates a completely regular patch-set and that regularity could be exploited. We believe that exploiting this regularity could provide significant simplifications to frameworks along with large increases in performance and may be necessary to achieve good performance on exascale machines.

As we move toward exascale machines, we will need to move away from inherently global algorithms and toward more local algorithms. Algorithms that operate on global metadata will eventually be limited by the cost to compute and communicate the global metadata. We have shown that the grid creation for BSAMR can be done completely locally; however, most frameworks still depend on the construction of global metadata. These frameworks should focus on eliminating this dependency in preparation for future petascale and exascale machines.

# CHAPTER 5

# LOAD BALANCING

This chapter will discuss algorithms used within Uintah for load balancing. Section 5.2 will present algorithms for automatically estimating workloads for parallel computations using least squares analysis and filters. While the use of least squares and filters is not new, the application of them to the load balancing process is. Finally, Section 5.3 will present a new algorithm for parallel space-filling curve generation.

## 5.1 Introduction

Dynamic load balancing can be described as the minimization of three competing costs: The cost of load imbalance, the cost of communication, and the cost to generate the load distribution. A load imbalance occurs when processors are assigned varying amounts of work. A large load imbalance will cause processors with less work to wait for other processors to finish their computation, leading to poor utilization of system resources.

In addition, too much communication can also cause performance issues. Communication across the network is often slow relative to the time for computation and can easily dominate the time to reach a solution. In many simulations, communication is predominantly local, meaning that only a small area around each patch must be communicated from physically neighboring patches. By clustering neighboring patches together, the framework can greatly reduce the necessary communication and significantly reduce the overall runtime.

Finally, with AMR methods, the workload changes as the mesh changes. In addition, with particle methods, the workload can change on each time step as particles move throughout the domain. This can necessitate that load balancing occurs often, making it important that the time to generate the patch distribution is

small relative to the overall computation. If a slow load balancing algorithm is used and load balancing occurs often, the time to load balance can dominate the overall runtime. In this case, it may be preferable to use a faster load balancing algorithm at the cost of more load imbalance. In addition, the time to migrate data between processors may also become significant when load balancing often [12].

There are many widely used methods for load balancing that attempt to minimize these costs. One way to load balance effectively is by partitioning a weighted graph, where the nodes of the graph are weighted by the amount of computation on a patch and the edges are weighted by the amount of the communication on that patch. Graph-based algorithms are then used to determine an optimal patch distribution [61, 92, 96, 97]. These methods do a good job of distributing work evenly while also minimizing communication. However, the cost of partitioning with these methods may be expensive in some cases due to the computational complexity of graph-based algorithms.

Additional methods for load balancing involve diffusion of computational work between neighboring processors [28, 50, 51, 52]. These algorithms attempt to eliminate load imbalance by migrating excess work to neighboring processors. These algorithms allow migration to occur completely locally and can reduce the dependencies on global metadata that is required with other load balancing algorithms.

Other methods attempt to minimize communication by clustering work according to the geometric position within the domain. These methods work well for simulations where communication is predominately local and include methods such as recursive coordinate bisection [9, 96] and space-filling curve partitioning [32, 95, 105]. One advantage to geometric methods is that there is no need to explicitly form the communication graph which may not always be available.

The need for fast and effective load balancing techniques has led to the development of widely used load balancing applications like Metis [60, 61], Jostle [114, 115, 116], and Zoltan [14, 30, 31]. Uintah has recently added support for the Zoltan load balancing package, providing easy access to a number of algorithms. In addition, Uintah can use its own highly parallel load balancing algorithm [76] that utilizes space-filling curves, which has been shown to be better than Zoltan's space-filling curve load

balancer within Uintah [82]. The variety of available simulation components within Uintah necessitates a sophisticated load balancer that is flexible enough to handle all of Uintah's different simulation components. This has been accomplished through a new automated cost estimation algorithm that will be described in Section 5.2 and a new parallel space-filling curve generation algorithm that will be described in Section 5.3.

## 5.2 Cost Estimation Algorithms

To load balance the computation effectively, load balancers require an accurate estimate of the cost (execution time) of the computation. A poor estimate of this cost could lead to significant load imbalances. Thus, it is important that this estimate be accurate. Two methods for estimating this cost at runtime are algorithmic cost models and forecasting cost models.

### 5.2.1 Algorithmic Cost Models

Algorithmic cost models (ACM) attempt to model the underlying algorithms. For example, the ICE algorithm is a cell-based algorithm that performs a similar amount of work on each cell and as such, the cost is proportional to the number of cells. Equation (5.1) describes an accurate ACM for ICE

$$C_i = c_1 + c_2 N_i^c, \tag{5.1}$$

where $C_i$ is the computational cost of a patch, $N_i^c$ is the number of cells in patch $i$, $c_1$ is a fixed overhead per patch, and $c_2$ is the time for executing the ICE algorithm for a single cell.

In addition to performing cell-based computations, MPMICE also has particle-based computations in regions where solid materials exist. Equation 5.2 describes a possible ACM for MPMICE:

$$C_i = c_1 + c_2 N_i^c + c_3 N_i^p, \tag{5.2}$$

where $N_i^p$ is the number of particles within patch i and $c_3$ is the constant execution time for a particle.

This model is not as accurate as the ICE model because the work performed by MPMICE is not constant per particle or cell. In MPMICE during the equilibration pressure solve, the simulation performs a local iterative solve on a per-cell basis [43]. This solve may converge at different rates throughout the domain depending on the underlying physics. In addition, particles may execute different material models which have differing workloads. Capturing such behavior in an ACM is a challenging task.

In addition to developing accurate models, estimates for the constants must be determined on a per-problem basis. The constants can vary greatly depending on the underlying physical processes. To make matters worse, these constants can also vary according to system architectures, compilers, and compiler optimization flags. The difficulty in estimating these constants increases significantly with the number of constants in the model. Maintaining an accurate list of these constants for each possible problem, architecture, and compiler combination is not feasible, thus placing the challenge of estimating these constants for each simulation on the user. An alternative is to automatically estimate these constants at runtime.

#### 5.2.1.1 Automated Model Fitting

Uintah can estimate model constants automatically at runtime by measuring the execution time per patch. Given these measurements and the patch metadata (i.e. number of cells, particles, etc.) least-squares analysis can be used to fit constants to the execution model. The linear least-squares problem that must be solved to determine the constants is

$$\begin{bmatrix} N_0^c & N_0^p & 1 \\ N_1^c & N_1^p & 1 \\ \vdots & \vdots & \vdots \\ N_n^c & N_n^p & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} O_0 \\ O_1 \\ \vdots \\ O_n \end{bmatrix}, \tag{5.3}$$

where $O_i$ is the observed time on patch i.

This provides a mechanism to accurately fit the model parameters which can

greatly increase the accuracy of the model. However, if the model does not accurately represent the computation, the predicted weights could be highly inaccurate, leading to a poor load balance. This method removes the burden of estimating the constants from the user. However, the user must still develop and/or choose an appropriate ACM for the simulation.

### 5.2.2 Forecasting Cost Models

Developing an accurate algorithmic cost model is challenging; because of this, Uintah has an alternative approach which utilizes a forecasting cost model (FCM) to predict the cost of each patch based on time series analysis. During task execution, the time to complete each task on a region of the domain is recorded and used to update a simple forecasting model. That model is then used to predict the execution time on that region in the future. This provides a mechanism to accurately predict the cost of each patch and eliminates the need to estimate constants for an ACM.

#### 5.2.2.1 Fading Memory Filter

A straightforward forecasting model used within Uintah is Simple Exponential Smoothing, which is also known as a Fading Memory Filter [84]. This model is as follows:

$$W_{r,t+1} = \alpha O_{r,t} + (1 - \alpha) W_{r,t}, \tag{5.4}$$

where $W_{r,t}$ is the predicted cost at time step t on region r, $O_{r,t}$ is the observed time at time step t on region r, and $\alpha$ is a weighting factor in the range of [0,1], which represents the rate of decay on past data. This method can also be viewed as a weighted moving average where the weight on past observations decreases exponentially [84]. A smaller value for $\alpha$ causes the algorithm to put more weight on recent observations, causing the forecast to respond more quickly to changes in the actual value but also causes the forecast to become more susceptible to noise. A larger value for $\alpha$ reduces the noise but also causes the forecast to react more slowly to changes in the actual value. $\alpha$ can be defined in terms of the size of a moving average window as

$$\alpha = \frac{2.0}{T+1}, \tag{5.5}$$

where T is the number of time steps that will contain 99.9% of the total weight in the weighted average [84]. Uintah uses a default value of 10 for T.

#### 5.2.2.2 Kalman Filter

An alternative approach for the Fading Memory Filter that has been implemented within Uintah is the Kalman filter [58, 123] forecast model. The Kalman filter improves upon the memory filter by accounting for uncertainty in both the measurements and the model used. This filter is defined as follows:

$$\begin{aligned}
W_{r,t+1} &= W_{r,t} + K_{r,t}(O_{r,t} - W_{r,t}), && (5.6)\\
K_{r,t} &= \frac{M_{r,t}}{M_{r,t} + \sigma^2}, && (5.7)\\
M_{r,t} &= P_{r,t+1} + \phi, && (5.8)\\
P_{r,t} &= (1 - K_{r,t})M_{r,t}, && (5.9)\\
P_{r,0} &= \infty, && (5.10)
\end{aligned}$$

where $K_{r,t}$ is a gain factor, $M_{r,t}$ is the *a priori* covariance, $P_{r,t}$ is the *a posteriori* covariance, $\sigma^2$ represents the variance in the measurement, and $\phi$ represents an uncertainty in the model. This filter is similar to the fading memory filter, as can been seen in the update Equations (5.4) and (5.6). The difference between these filters is that the gain factor which is static for the memory filter is computed dynamically based upon uncertainty for the Kalman filter.

The effects of these parameters on the estimations are discussed in detail in [123].

#### 5.2.2.3 Filter Initialization

On the first time step of the simulation, $W_{r,t}$ is unavailable, requiring an estimation of the initial value. For the initial time step, Uintah load balances using the algorithmic cost models described above. The initial measurements are then used to

set the initial value by setting $W_{r,0} = O_{r,0}$. Doing this causes the filter to rapidly converge to the true value.

A different initialization approach is used when new regions of refinement are created during the regridding process. During this process, refinement may be added in regions where it previously did not exist. When these regions are created, $W_{r,t}$ must be estimated. Using an ACM would likely produce a poor estimate that could be inaccurate by orders of magnitude. In order to limit this inaccuracy, we estimate the cost to be proportional to other forecasted values. Uintah sets $W_{r,t}$ for the new regions equal to the average value of $W_{r,t}$ for all regions. This ensures that the initial value for the new region is at least proportional to the actual value which also ensures that the estimation will be accurate within a few time steps and that load imbalance caused by this estimation will be limited.

#### 5.2.2.4 Implementation Details

To allow for changing patch sets, forecasting is performed on a per-region basis instead of a per-patch basis. The difference between regions and patches is shown in Figure 5.1. Regions are constant-sized portions of the domain that are contained within a single patch. Patches, on the other hand, are variable-sized portions of the domain that may contain many regions.

By forecasting on a per-region basis, the patch set can change without needing to migrate forecasting data between the changing patch sets. This necessitates

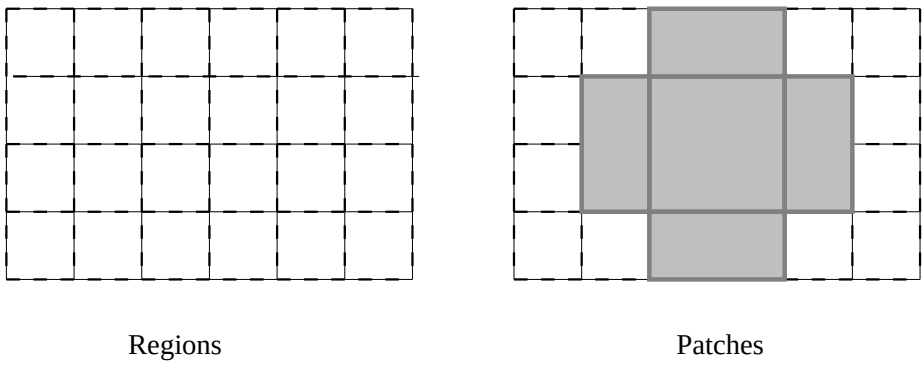



**Figure 5.1**. The difference between regions and patches. Patches are composed of one or more fixed size blocks referred to as regions.

mechanisms to interpolate the data between regions and patches. Since regions are completely contained within patches, these mechanisms are straightforward to describe and implement. The measured cost for each region is equal to the cost of the patch, times the proportion of the patch that the region encompasses, as described by Equation (5.11):

$$O_{r,t} = O_{p,t}\frac{V_r}{V_p}, \tag{5.11}$$

where t is the current time step, $O_{r,t}$ is the measured computation time for region r, $O_{p,t}$ is the measured execution time for patch p, $V_p$ is the volume of patch p, and $V_r$ is the volume of region r. In addition, $W_{p,t+1}$ can be defined as the sum of the weights of all regions contained in patch p, as shown in Equation (5.12):

$$W_{p,t+1} = \sum_{r}^{r\in p} W_{r,t+1}. \tag{5.12}$$

Uintah stores the forecasting data while minimizing both storage and communication. The forecasting data are stored locally on each processor. When a processor executes a task on a patch, it adds the contribution to its local forecast data using Equation (5.11). If a region was owned by a different processor in the past, then local forecast data will exist on multiple processors but each processor will only update its local data. At the end of each time step, the simulation finalizes the forecast data by applying Equation (5.4). Updating the forecast data on each time step is a local operation which does not require any communication. However, communication is required when load balancing occurs. During load balancing, each processor must know the cost of each patch. This is done by applying Equation (5.12) locally and then performing a MPI_Allreduce to get the global sum.

To keep the data structures for forecasting as small as possible, contributions are stored in a Standard Template Library (STL) map, which is a sparse data structure. This causes the storage per processor to be proportional to the number of patches per

processor. In addition, when a processor has not updated a region in its map for over T time steps, the contributing weight for that region is less than 0.1% of the total weight, at this point, we consider the weight to be insignificant and delete it from the map. This prevents the size of the maps from slowly increasing over time.

### 5.2.3 Forecasting Results

The effect of forecasting on Uintah's runtime was tested using both ICE and MPMICE. The ICE simulation used the w-cycle, as described in Section 3.6.1, with explicit time stepping to simulate the transport of two fluids with a prescribed initial velocity of Mach two. For this problem, the conservation of mass, momentum, and energy equations were solved for two inviscid fluids. The fluids exchange momentum and heat through the exchange terms in the governing equations, as described in Section 3.2.1.1. This problem exercises all of the main features of ICE and amounts to solving eight P.D.Es, along with two point-wise solves, and one iterative solve; for more information see [43].

A simulation similar to the ICE benchmark was also used to test MPMICE. In this simulation a solid explosive was transported through air at Mach two. The simulation used an explicit formulation with the lock-step time stepping algorithm described in Section 3.6.1. This simulation exercises many of the same components as the ICE benchmark but also represents the explosive with MPM particles. This benchmark also included a model for the deflagration of the explosive [117, 120] and the material damage model ViscoSCRAM [7]. This problem closely resembles CSAFE's exploding container problem, as described in Chapter 1.

The computational cost of the ICE problem is predictable and developing an accurate ACM is straightforward. In contrast, the computational cost of the MPMICE problem is more challenging to predict. Three cost estimation algorithms were tested on each of these problems. The estimation algorithms tested included the ACM with least squares, the memory filter, and the Kalman filter.

Figure 5.2 shows the mean absolute percent error (MAPE) of the three algorithms for the ICE problem. This figure shows that all three algorithms are highly accurate and that the least squares model is the least accurate of the three. Both filters achieve

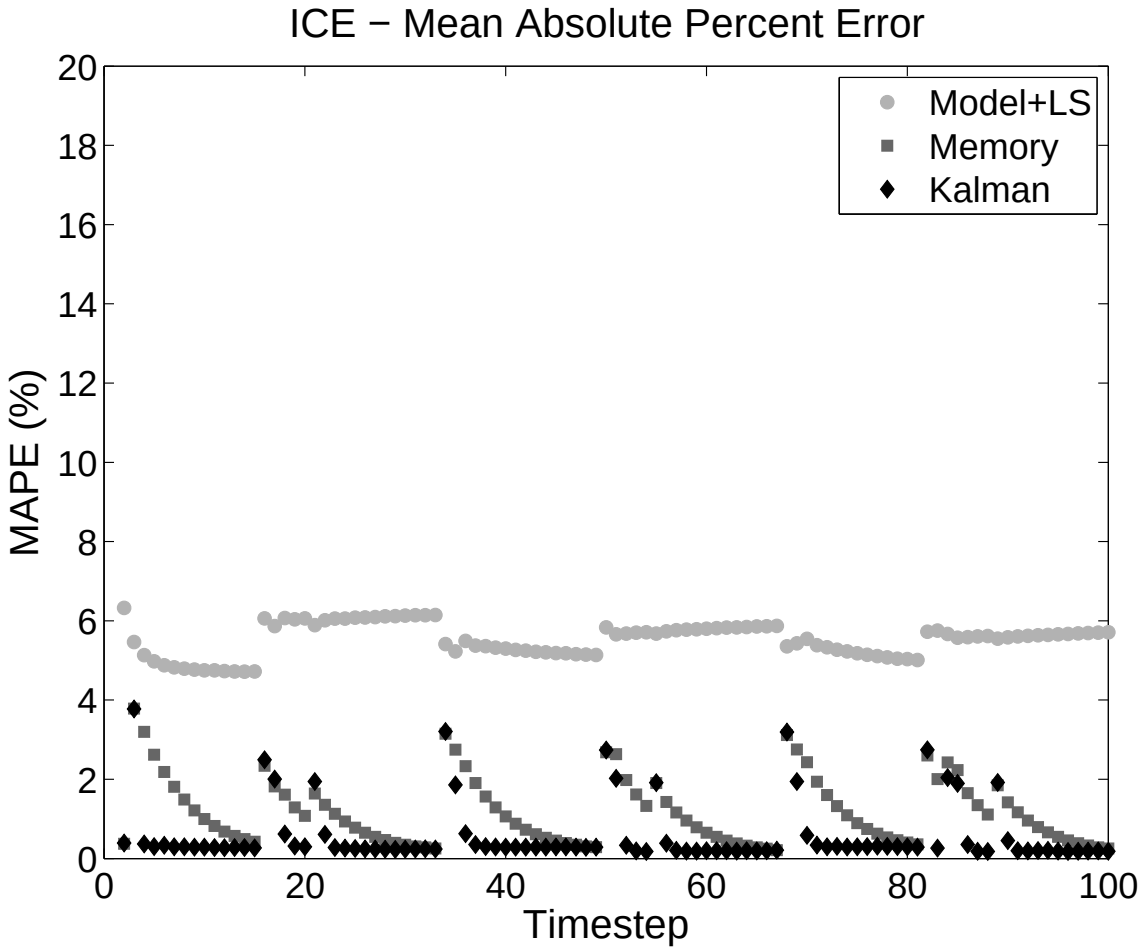



**Figure 5.2**. A comparison of cost estimation algorithms within ICE

a very high level of accuracy but occasionally have spikes in the error. These spikes occur immediately after regridding because the computation significantly changes. The spikes do not appear in algorithmic cost model results.

Figure 5.3 shows the mean absolute percent error of the three algorithms for the MPMICE problem. The results for this problem are similar to the ICE problem with the exception that the spikes are much more significant, causing the filters to perform worse than the model immediately following a regrid. These spikes are significant because load balancing will always occur immediately following a regrid. Thus, these spikes will negatively effect performance. However, the impact of these spikes are limited because the simulation is load balanced every few timesteps.

From these results, it appears that the filters are the most accurate estimation

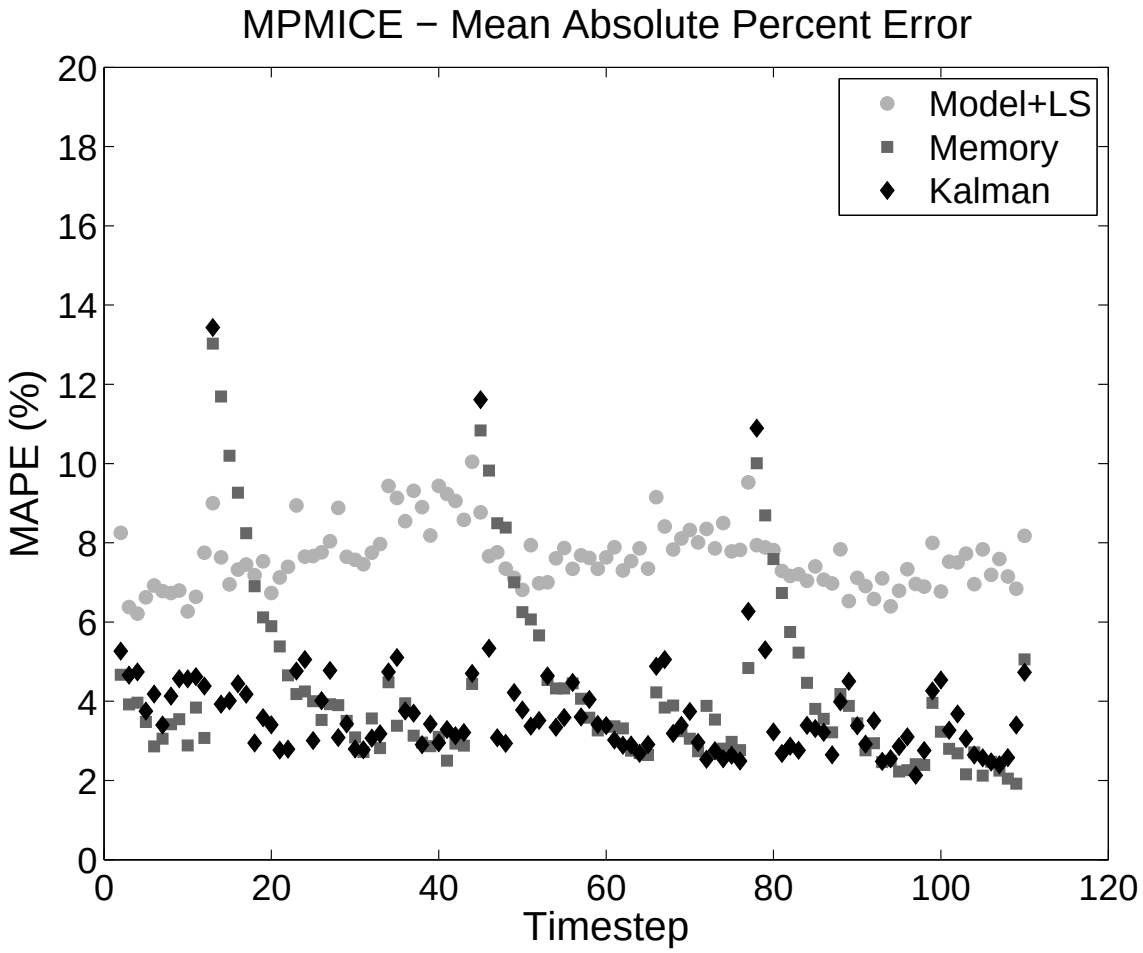



**Figure 5.3**. A comparison of cost estimation algorithms within MPMICE

models. However, their accuracy may suffer if regridding is occurring often. The algorithmic cost model does not suffer from the same spikes and thus can better predict performance across regrids. It is likely that improved algorithmic cost models could be developed which could make them a better choice.

## 5.3 Space-Filling Curve Generation

One commonly used method for quickly partitioning work is based on space-filling curves [25, 32, 95, 105]. Space-filling curves are fractal curves that uniformly fill multidimensional space. Space-filling curves are generated using a base shape, which determines the order the curve visits subdivisions of space. As space is further subdivided, the base shape is reapplied to the new subdivisions. The base shape may be rotated and/or reflected before being applied to new subdivisions. As space is subdivided, the curve fills more of the domain and in the limit of subdivisions, the curve completely fills the domain.

Space-filling curves have many properties that make them a good basis for load balancing. For example, the Hilbert curve (as shown in Figure 5.4), traverses space while staying close to itself [93]. This creates an ordering that clusters points together such that points that are close together on the curve are also close together in the original domain; however, points that are close together in the domain are not guaranteed to be close together on the curve. In addition the Hilbert curve prevents jumps across the domain by always moving locally. Finally space-filling curves can be created in $O(N \log N)$ time, where $N$ is the number of work elements to be mapped onto processors. Work elements can be mesh points, patches, particles, or any other grouping of work that can be represented by a single point in space.

Space-filling curves can be used to load balance by ordering work according to the curve and then splitting the curve into segments called partitions. The curve stays close to itself, causing work units in the same partition to be close to each other in the domain. In most applications, work units require information from neighboring work units. Clustering neighboring work units onto the same processor greatly reduces communication costs.

Space-filling curves can be generated quickly in serial by recursively subdividing

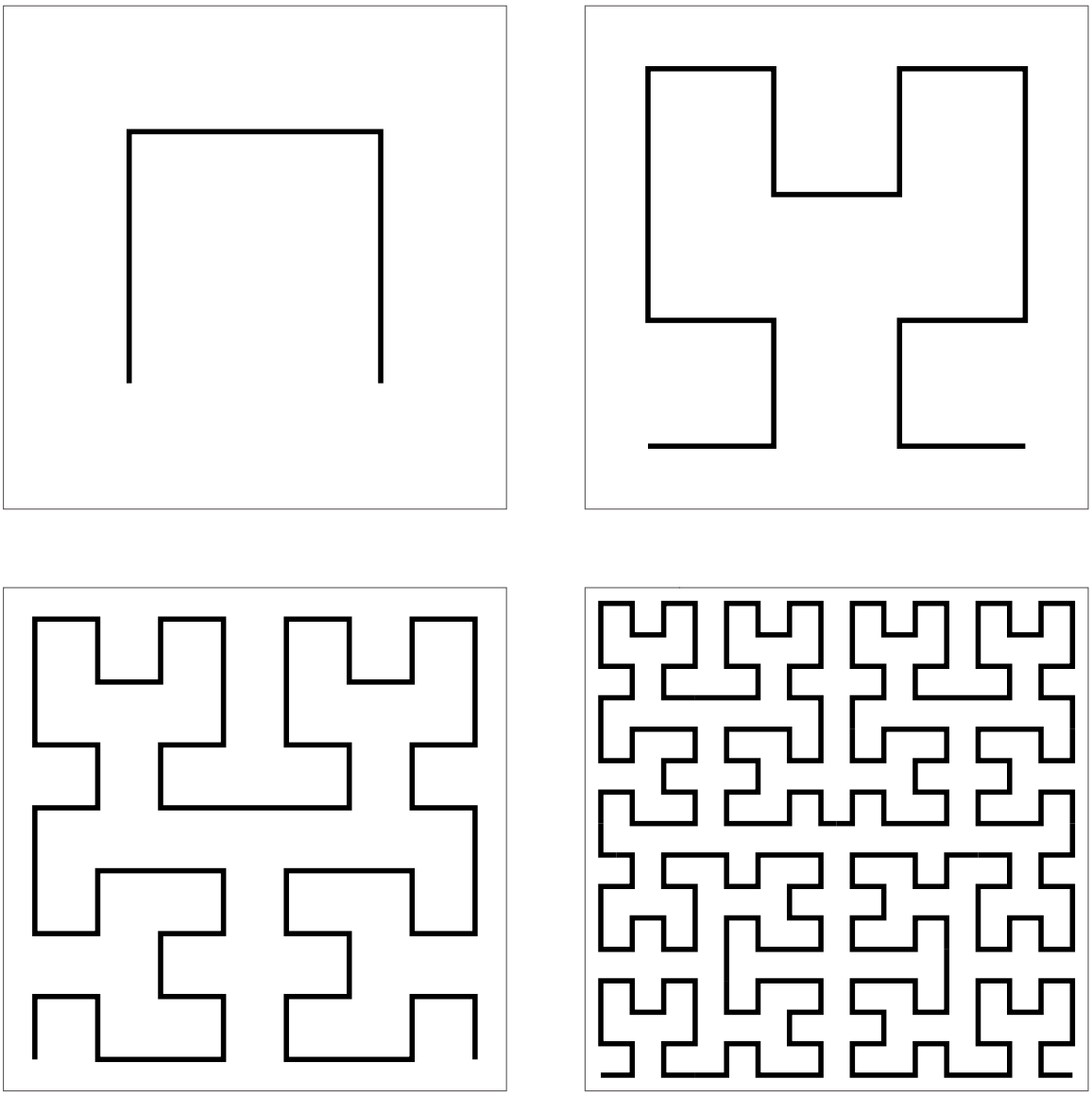


**Figure 5.4**. The first four levels of the Hilbert curve

the domain into octants and ordering those octants according to the space-filling curve using a series of state diagrams [25]. The state diagrams are represented by two arrays: an ordering array and an orientation array. The ordering array determines the order that the curve visits nodes based on their parent's orientation. The orientation array determines the orientation of the children based on their parent's orientation and the order that the curve visited them. On low numbers of processors, serial generation completes quickly. However, on large numbers of processors, the time for gathering elements onto a single processor and generating the curve may become significant. In addition, on some machines, gathering onto a single processor may not be possible due to memory constraints.

Others have attempted to generate space-filling curves in parallel by defining a comparison operator allowing for a parallel merge sort design [1, 118], without considering the Hilbert curve. To address this case, we have developed a new method that works for any space-filling curve. Our method creates the curve order index for every point on the curve while generating a local curve on each processor. The index generated is the index into the fully refined curve at the desired refinement level. Each processor generates the curve at the same refinement level, allowing points on the curves to be compared to each other by using the index. This allows a global curve to be generated by merging the curves together.

Parallel merging is a well-understood problem. One method for merging involves performing a series of pairwise merge and exchanges. Batcher's merging algorithm performs these merge-exchanges using the minimum number of required merge-exchanges [6, 69]. It was shown that performing extra merge-exchanges prior to Batcher's algorithm might increase the overall performance [110, 119]. Recent methods avoided merge-exchanges by moving data directly to the correct processor using global histograms [70, 99]. The generation of the global histogram and performing an all-to-all transpose of the data made it difficult to scale these methods to large numbers of processors. As a result, newer methods addressed this problem by keeping the histogram communication and data movement local [55, 56].

In this section, we explore generating space-filling curves in parallel using multiple sorting methods on multiple architectures. In addition, we analyze the complexity of

some of these methods in more depth to predict their performance up to hundreds of thousands of processors.

For the remainder of this section, the following definitions will be used:

$$
\begin{aligned}
P &= \text{number of processors,} \\
N &= \text{number of work elements to be ordered by the space-filling curve,} \\
\frac{N}{P} &= \text{number of work elements per processor.}
\end{aligned}
$$

### 5.3.1 Serial Sort

The sorting phase of the algorithm generates a space-filling curve in serial on a subset of the data on each processor. Algorithm 3 sorts the data using the oct-tree decomposition and state diagram method described in [25]. Space is divided into octants and each work unit is placed into a list. The list that the work unit is placed into depends on the octant it is contained in and the order the curve visits that octant relative to sibling octants. The ordering can quickly be computed using the order and orientation arrays described in [25]. The lists are then concatenated into a single list and the process recursively repeats for each sublist. The recursion terminates when the desired refinement level is reached. The refinement level can be determined a priori if the size of the domain and the minimum distance between any two points along each dimension is known. In Uintah, this information is known, but if it is not known, it can either be computed or bounded easily. If too small a refinement level is requested, then the algorithm generates an approximate space-filling curve which may be advantageous in some applications.

In addition to sorting the data, Algorithm 3 also generates the curve index for each work unit. At every refinement level, points are assigned a local curve order. The local curve order is the order the curve visits a node relative to its siblings. Local curve orders are concatenated together into a single integer such that the digits for the curve orders at the coarser levels are left of the digits for the curve orders at the finer levels. The concatenated curve orders form the fully refined curve index. The curve order at each refinement level can be represented with $d$ bits where $d$ is the number of dimensions of the domain. Consequently, the number of bits needed to

store each curve index is equal to $d \times r$ where $r$ is the number of refinement levels. Using this information, our implementation dynamically chooses the data type for the curve index at runtime, reducing the overall communication. Figure 5.5 shows the curve orders that are used to form the curve indices for the first two levels of the Hilbert curve. Curve indices can be compared using an integer comparison operator to determine which point is earlier on the curve. This provides a method for merging the curves together.

Figure 5.6 shows the indices that Algorithm 3 would assign to four points for two refinement levels. At the first level, the points {a, b, c, d} would be reordered and assigned the following indices: {b:1, a:2, c:2, d:3}. At the second level, they would be reordered and assigned the following indices {b:12, c:20, a:22, d:33}.

### 5.3.2 Parallel Merging

Once work units are locally sorted and have their curve indices generated, the curves can be merged. We studied two classes of merging methods. The first uses pairwise merge-exchange operations between processors over a Batcher's merging network [6, 69]. We tested two methods within this class; they are the original Batcher's algorithm and Batcher's algorithm with a premerging step [110, 119]. The second class of methods avoids pairwise merge-exchanges by generating histograms of the sorted data to determine where to relocate data. Within this class, we tested

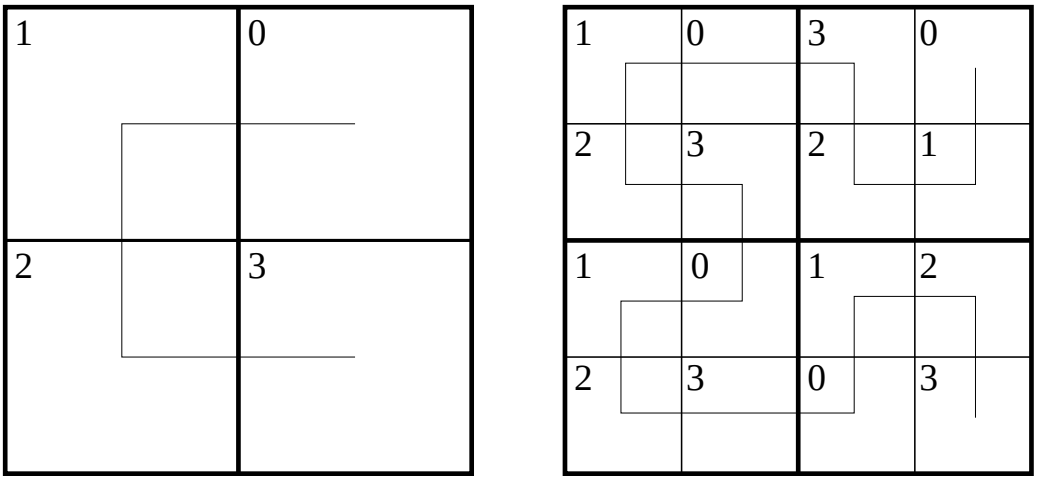



**Figure 5.5**. The Hilbert curve orders for the first two refinement levels.

**Algorithm 3** The SFC Serial Sort Algorithm

```
Inputs:
    list: Pointer to the list of points to be sorted
    N:    The size of list
    r:    Number of refinement levels
    o:    The orientation of this octant
Outputs:
    list: The sorted list of points and their curve index

function BinSort(list, N, r, orientation=0, Index=0)
  //1: Save Index
  if r==0 then
    for i=0 to N
      SaveIndex(list[i],Index)
    end for
    return
  end if

  //2: Bin each item in the list according to SFC
  for i=0 to N
    octant=GetOcatant(list[i])
    b=OctantToOrder(orientation,octant)
    bins[b].add(list[i])
  end for

  //3: Place bins back in original list
  j=0
  for b=0 to sizeof(bins)
    for i=0 to sizeof(bins[b])
      list[j]=bins[b][i]
      j=j+1
    end for
  end for

  //4: Call recursively on each sublist
  i=0
  for b=0 to sizeof(bins)
    if sizeof(bins[b])>0 then
      BinSort(&list[i],sizeof(bins[b]),r-1,
              OrderToOriention(orientation,b),Concatenate(Index,b))
      i=i+sizeof(bins[b])
    end if
  end for

end function
```

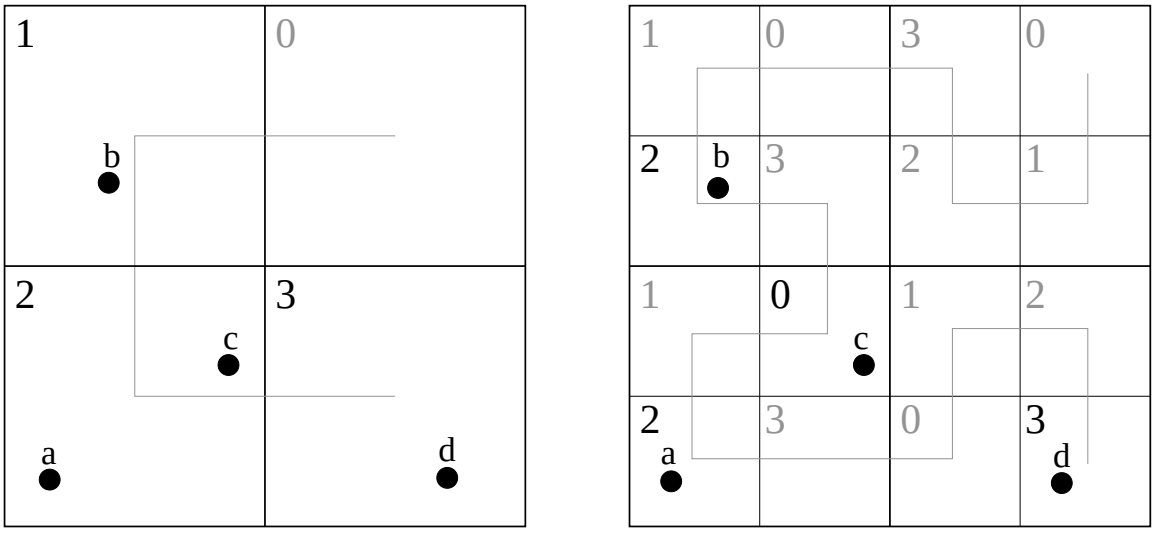



**Figure 5.6**. An illustration of the SFC sort algorithm. The indices that Algorithm 3 would assign to four points for two refinement levels. At the first level, the points {a, b, c, d} would be reordered and assigned the following indices: {b:1, a:2, c:2, d:3}. At the second level, they would be reordered and assigned the following indices: {b:12, c:20, a:22, d:33}.

two methods. The first method generates a global histogram across all processors and then moves data directly to the correct processor [70, 99]. This method performed poorly because of the cost of gathering the histogram onto every processor and then performing an all-to-all transpose of the data. For brevity and clarity we have omitted this method from the rest of the chapter. The second method within this class generates local histograms within processor groups and communicates only within those groups [55, 56].

#### 5.3.2.1 Merge-Exchange Algorithm

In a merge-exchange operation, two processors communicate their lists to each other. Each processor then independently merges the lists and one processor keeps the lower half of the list while the other keeps the upper half of the list. One advantage of using merge-exchanges is that communication is always pairwise. Pairwise communication and merging is straightforward, allowing for a new design that overlaps communication and computation [76].

A description of the merge-exchange operation follows. Processors first exchange

their minimum and maximum elements. Both processors compare the ranges of the data and if the data ranges do not overlap, the processors exchange all or none of the list without merging. If the ranges overlap, then the merge-exchange operation enters into a sample merge. By merging a small sample of the data, the processors can determine an upper bound on the amount of data that needs to be communicated in order for each processor to merge its data.

Next, the full merge-exchange operation begins. The processor keeping the lower half of the data merges the two lists in ascending order while the other processor merges the two lists in descending order. Each processor stops merging when it has merged half of the list. In addition, processors do not communicate the entire curve at once; instead, the curve is decomposed into segments referred to as blocks which are communicated asynchronously to the appropriate processor. The processor keeping the lower half of the data sends blocks in descending order and receives in ascending order while the other processor does the opposite. Communicating in blocks allows processors to potentially overlap communication and merging. If the time to send an element is less than the time to merge an element and the block size is large enough to overcome latency overhead, all communication except for the first block can be overlapped within the merging of the previous block. Choosing a block size that is too small will prevent some communication from being overlapped with every block, while choosing a block size that is too large will add additional overhead on the first block. Therefore, it is best to overestimate the block size rather than to underestimate it.

As processors perform their merge-exchanges, elements become clustered together on each processor, allowing for an optimization that reduces the number of comparisons. Instead of merging element by element, merging is first attempted in blocks. The minimum and maximum of the front block of each list is compared, if there is no overlap, then the block with the lower range is copied to the new list and the process repeats with the next block. If there is overlap, then the blocks are merged element by element until all elements from one of the blocks are merged. Then, blocked merging is attempted on two new blocks.

The optimal size of communication blocks, merging blocks, and the sample size

depends on factors like the communication network, the processor speed, the assigned processors, current traffic on the machine network, competition on the machine from other processes, and original data layout. Because of these dependencies, developing a model to determine ideal parameters was not undertaken. Instead, we use a profiler that runs on a dataset provided by the user. The profiler attempts to minimize the execution time on that dataset. After a suitable set of parameters is determined, the profiler uses those parameters for subsequent calls.

Batcher's merging algorithm is defined in [6, 69]. It performs pair-wise merge-exchanges between processors and $\frac{\log^2 P+\log P}{2}$ merge exchanges are required per processor to guarantee the list is fully merged. It has been shown that this is the minimum number of merge-exchanges required to merge the lists [69].

Work in [110] and [119] showed that performing a premerging step before executing Batcher's algorithm can reduce overall execution time by reducing the number of times elements are moved. The premerging algorithm we have used is referred to as Algorithm 4. This algorithm executes a series of merge-exchange operations. When the number of processors is a power of two, this algorithm reduces to executing merge-exchanges on the edges of a hyper-cube. When the number of processors is not a power of two, some processors will be idle in some of the merge-exchange operations. Premerging reduces the number of times data is moved while increasing the number of messages sent. This suggests that premerging would be most beneficial on machines with low latency and low bandwidth.

The local histogram method is described in detail in [55, 56] and for brevity will not be described in detail here; instead, only an overview will be presented. In this algorithm, histograms are formed locally on each processor. Next, a series of merging and exchanging steps are executed. At each step, processors exchange their histogram with another processor within their merging group. The two histograms are combined into a new histogram and the old and new histograms are used to redistribute and merge elements within the group. After each step, each group is combined with another group and the process is repeated until one group remains. This method uses pairwise communication for the histogram at every step. Our implementation does not include the logical processor"ids" described in [55, 56]. We believe that on

**Algorithm 4** The Premerging Algorithm

```
Inputs:
    list:       The list of points to be merged
    numprocs:   The number of processors
Outputs:
    list:       The merged list of points

function PreMerge(list, numprocs)

  enqueue(0,numprocs)

  while(queue is not empty)
    dequeue(base,P)
    if rank>=base && rank-base<P then
      if(rank-base<P/2) then
        Merge-Exchange list with rank+(P+1)/2
      else
        Merge-Exchange list with rank-(P+1)/2
      end if

      //create next stages
      enqueue(b+P/2,(P+1)/2)
      enqueue(b,P-(P+1)/2)
    end if
  end while

end function
```

large numbers of processors, the cost of broadcasting the new ids will outweigh the cost of migrating data.

### 5.3.3 Complexity Analysis

The following sections analyze the complexity for components of the sorting algorithm.

#### 5.3.3.1 Serial Sort

Algorithm 3 has been numbered with four distinct sections. Section 1 saves the indices for each point. This occurs once per point and only on the last recursion, making the complexity for this part $O(\frac{N}{P})$. Section 2 places each point into the correct bin. The time for determining which bin a point should be placed in is constant, making the complexity of this section $O(\frac{N}{P})$. Section 3 copies the bins back to the original list and has complexity of $O(\frac{N}{P})$. Finally, Section 4 is the recursive call. The algorithm repeats $r$ times where $r$ is the number of refinement levels. Only sections two and three are repeated during the recursion. This makes the complexity $O(\frac{2Nr}{P} + \frac{N}{P}) = O(\frac{Nr}{P})$. For typical problems, $r$ is proportional to $\log N$, making the complexity for typical problems $O(\frac{N}{P} \log N)$.

#### 5.3.3.2 Merge-Exchange

The dominant portions of the merge-exchange algorithm consist of exchanging the data and merging the data. In the worst case, both processors will have to send $\frac{N}{P}$ elements. In addition, the worst case merging requires $\frac{N}{P}$ comparisons and assignments. In the best, case only an exchange of the minimum and maximum is required. This makes the complexity of the Merge-Exchange operations $O(\frac{N}{P})$ in the worst case and $O(1)$ in the best case.

#### 5.3.3.3 Batcher's Algorithm

Batcher's merging network completes in $\frac{\log^2 P + \log P}{2}$ merge-exchanges per processor. If premerging is used, an additional $\log P$ merge-exchanges will occur. While it is likely that some of the merge-exchanges will complete in $O(1)$, the majority of merge-exchanges will complete in $O(\frac{N}{P})$. This makes the complexity of Batcher's method

$O(\frac{N}{P}\frac{\log^2 P+\log P}{2}) = O(\frac{N}{P}\log^2 P)$.

#### 5.3.3.4 Local Histogram

The dominant costs in the local histogram algorithm come from the creation and communication of the histogram and redistribution data. The cost of creating the local histogram is $O(\frac{N}{P})$. There are $\log P$ merging stages. In each of these steps, the local histogram must be communicated and updated and then elements must be redistributed and merged. The cost of communicating and updating the histogram is $O(B)$ where $B$ is the number of bins in the histogram. To achieve a good load balance, $B$ must be at least proportional to $P$. Redistribution involves potentially moving $\frac{N}{P}$ elements to different processors. Since the elements are clustered on each processor, to redistribute elements, processors will only have to communicate with a few other processors. Merging also requires $\frac{N}{P}$ steps. This makes the complexity for one stage of the local histogram merging $O(P+\frac{N}{P})$ and the total complexity $O((P+\frac{N}{P})\log P)$.

### 5.3.4 Performance Results

A mesh from Uintah was used to compare the algorithms. The mesh is from the finest level of a multilevel simulation of two containers exploding. The mesh has 13365 patches. To produce larger meshes with similar properties, patches were subdivided in half along each dimension. This produced patch sets that were eight times as large as the original mesh. The centroid of each patch was used as its location. Patches were initially distributed according to their order within Uintah and an initial load balance was assumed. This assumption was made because within Uintah, all patches are known to all processors, allowing for the trivial balancing of patches on processors. However, if the patches are not initially load balanced, a prebalancing phase could be added.

Timings of the benchmark problem were run on multiprocessor machines. To simulate memory contention caused by multiple processes per node, the serial timings were computed by running multiple serial sorts at the same time. The number of serial sorts run was equal to the number of processors on a node. The reported times are the median of 21 samples.

The timings were taken on the Thunder, ALC, and Inferno supercomputers. These results were originally presented in [76] and these machines have since been retired. Thunder was a Linux cluster located at Lawrence Livermore National Labs with 1024 nodes, each with four 1.4 Ghz Intel Itanium2 single-core processors. Each node had eight GB of 266DDR SDRAM. Nodes were connected with a QsNet Elan 4 network with MPI latency $< 4\mu$s and MPI bandwidth of 900 MB/s. ALC was a Linux cluster located at Lawrence Livermore National Labs with 960 nodes each with two 2.4 Ghz Pentium 4 Prestonia single-core processors per node. Each node had four GB of PC2100 DDR SDRAM. The nodes were connected with a QsNet Elan 3 network with MPI latency $< 5\mu$s and MPI bandwidth of 400 MB/s. Inferno was a Linux cluster located at the University of Utah with 128 nodes, each with two 2.4 Ghz Intel Xeon single-core processors. Each node had two GB of PC2100 SDR SDRAM. The nodes were connected with a GB Ethernet network with MPI latency $< 25\mu$s and MPI bandwidth of 100 MB/s. Inferno had a slower network than Thunder or ALC, but had similar processing power per processor. This suggests that overhead introduced by communication will be much more significant on Inferno than on the other two machines.

Figure 5.7 and Figure 5.8 show the benchmark timings on Thunder, Figure 5.9 and Figure 5.10 show the benchmark timings on ALC, and Figure 5.11 and Figure 5.12 show the benchmark timings on Inferno. On all machines, there is near ideal scalability when N/P is large enough. Batcher's algorithm is faster than the local histogram method in most cases. On Thunder, the local histogram method scales further and achieves a lower execution time. As P increases, the performance difference between Batcher's algorithm and the local histogram method narrows. On ALC, there is not a significant difference between Batcher's algorithm and the local histogram method. On Inferno, Batcher's algorithm is faster than the local histogram method but unlike the other machines, the local histogram method only scales further when N is small. When N is larger, Batcher's algorithm outperforms the local histogram method significantly.

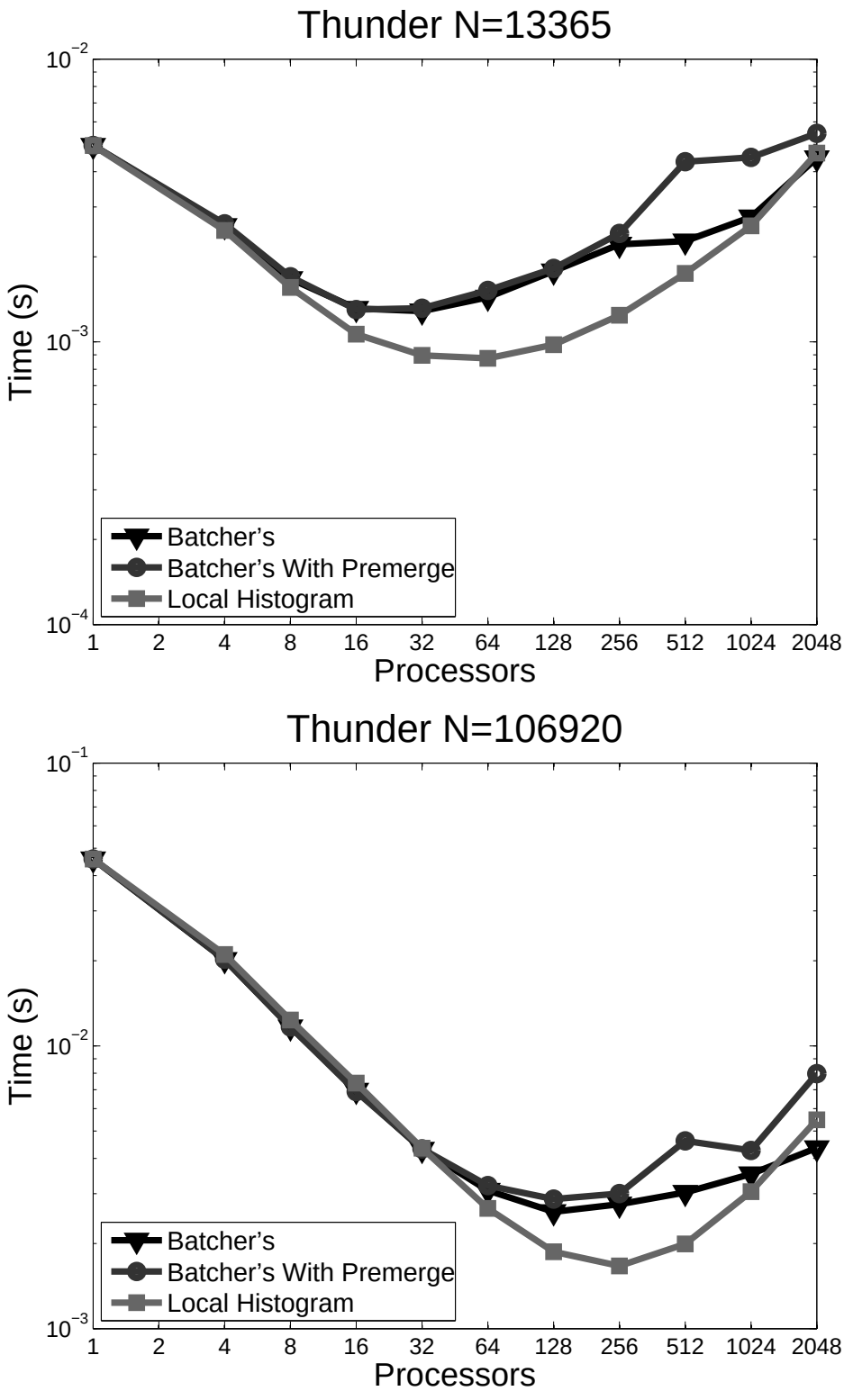



**Figure 5.7**. The strong scalability of the smaller benchmarks on Thunder

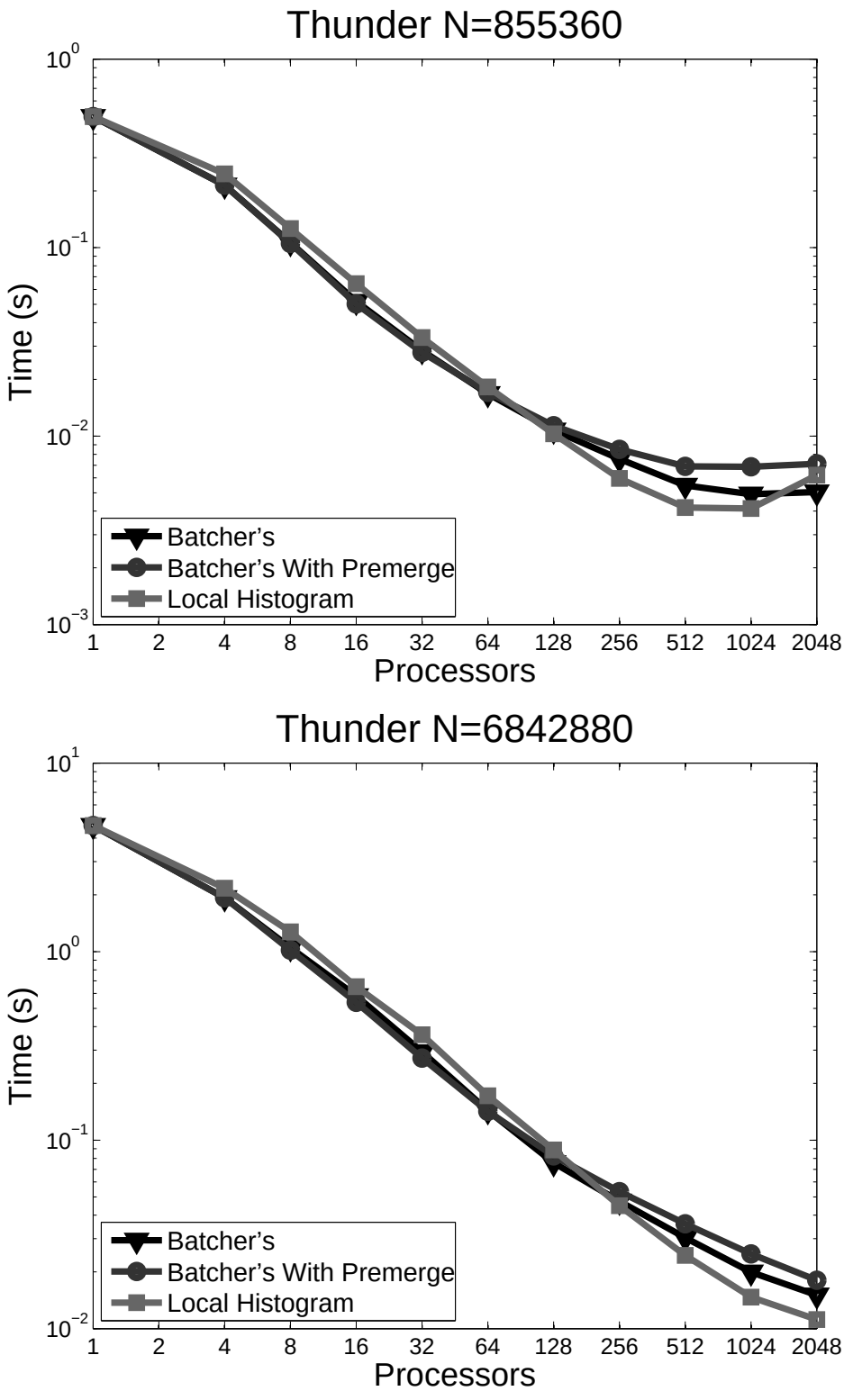



**Figure 5.8**. The strong scalability of the larger benchmarks on Thunder

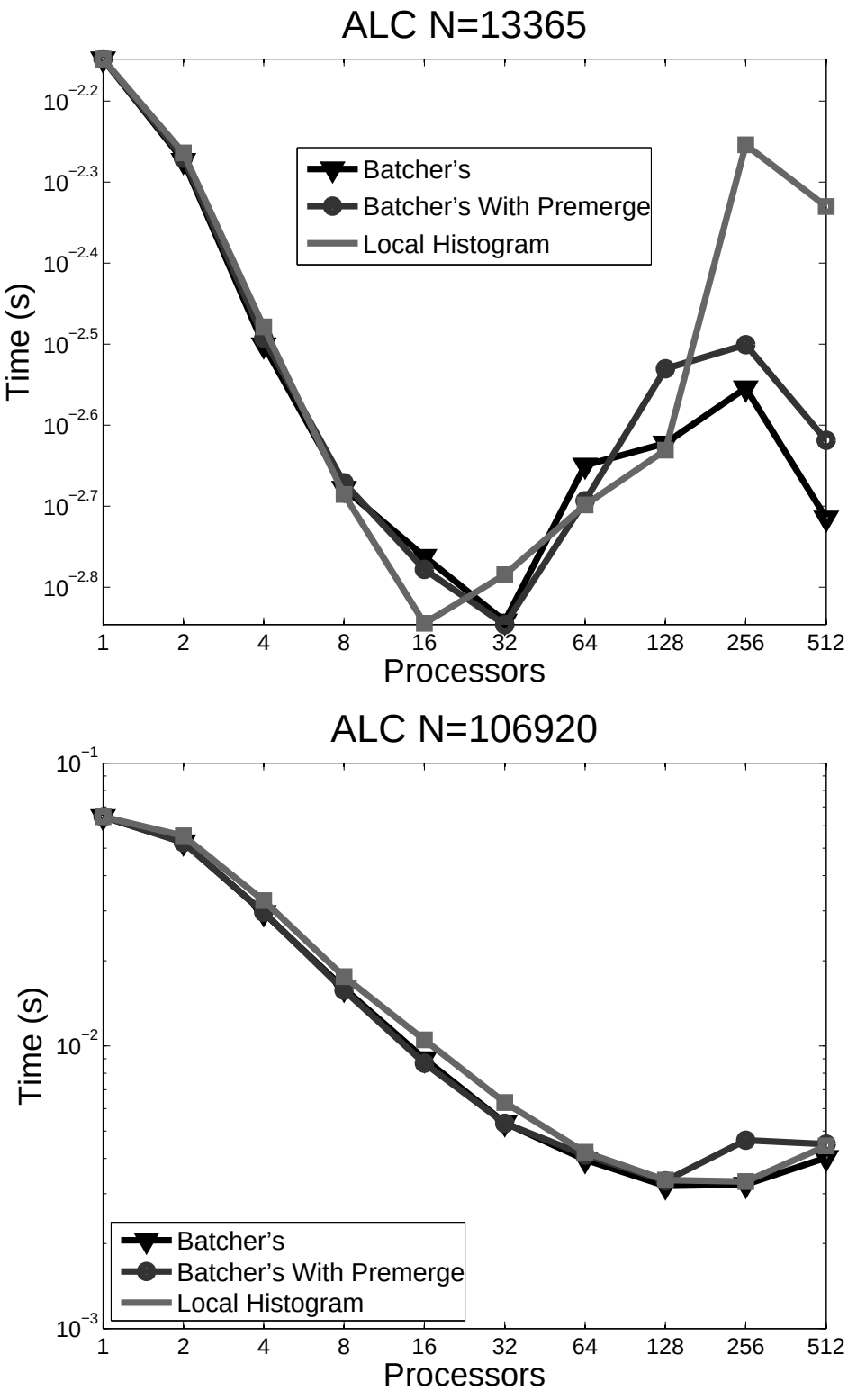



**Figure 5.9**. The strong scalability of the smaller benchmarks on ALC

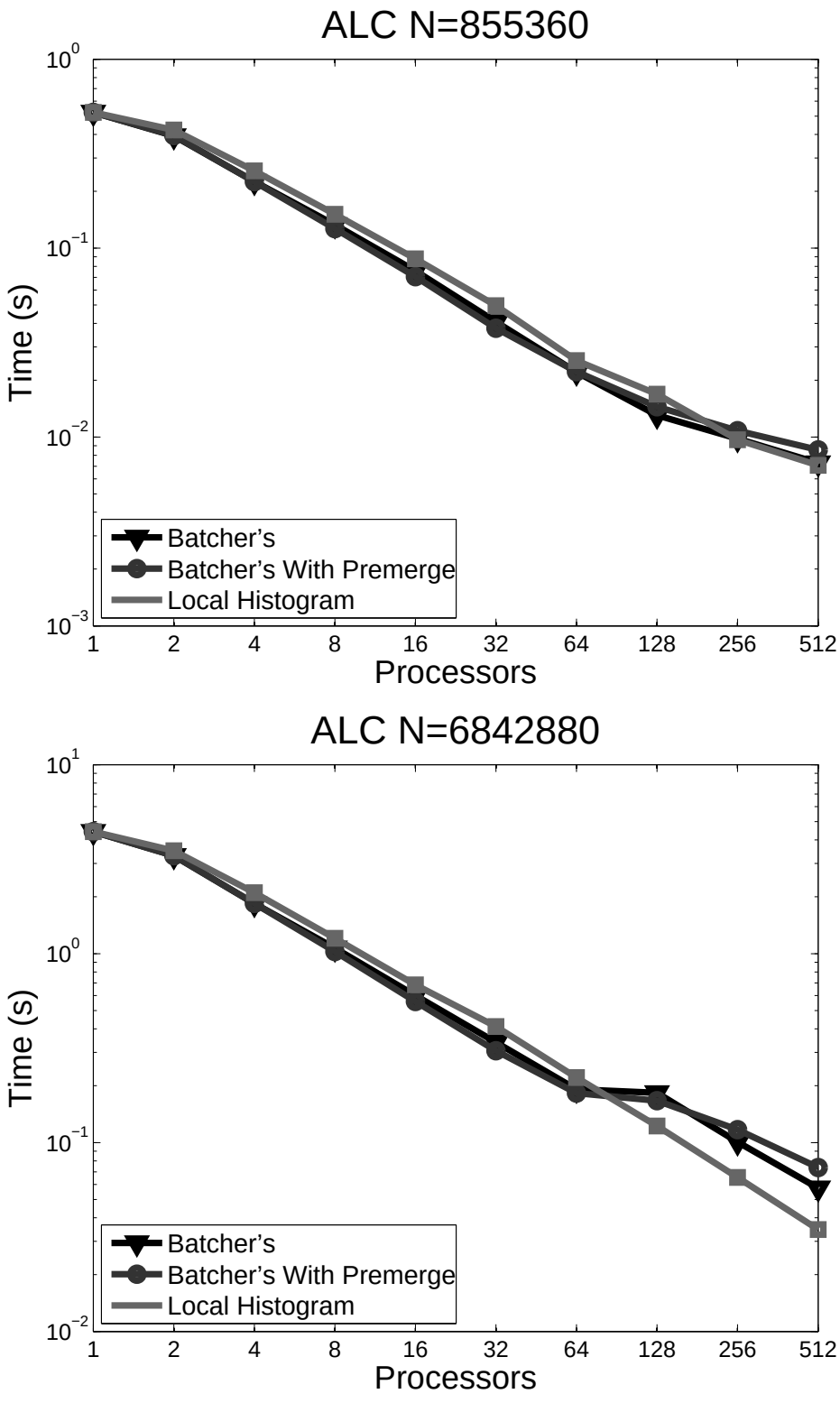



**Figure 5.10**. The strong scalability of the larger benchmarks on ALC

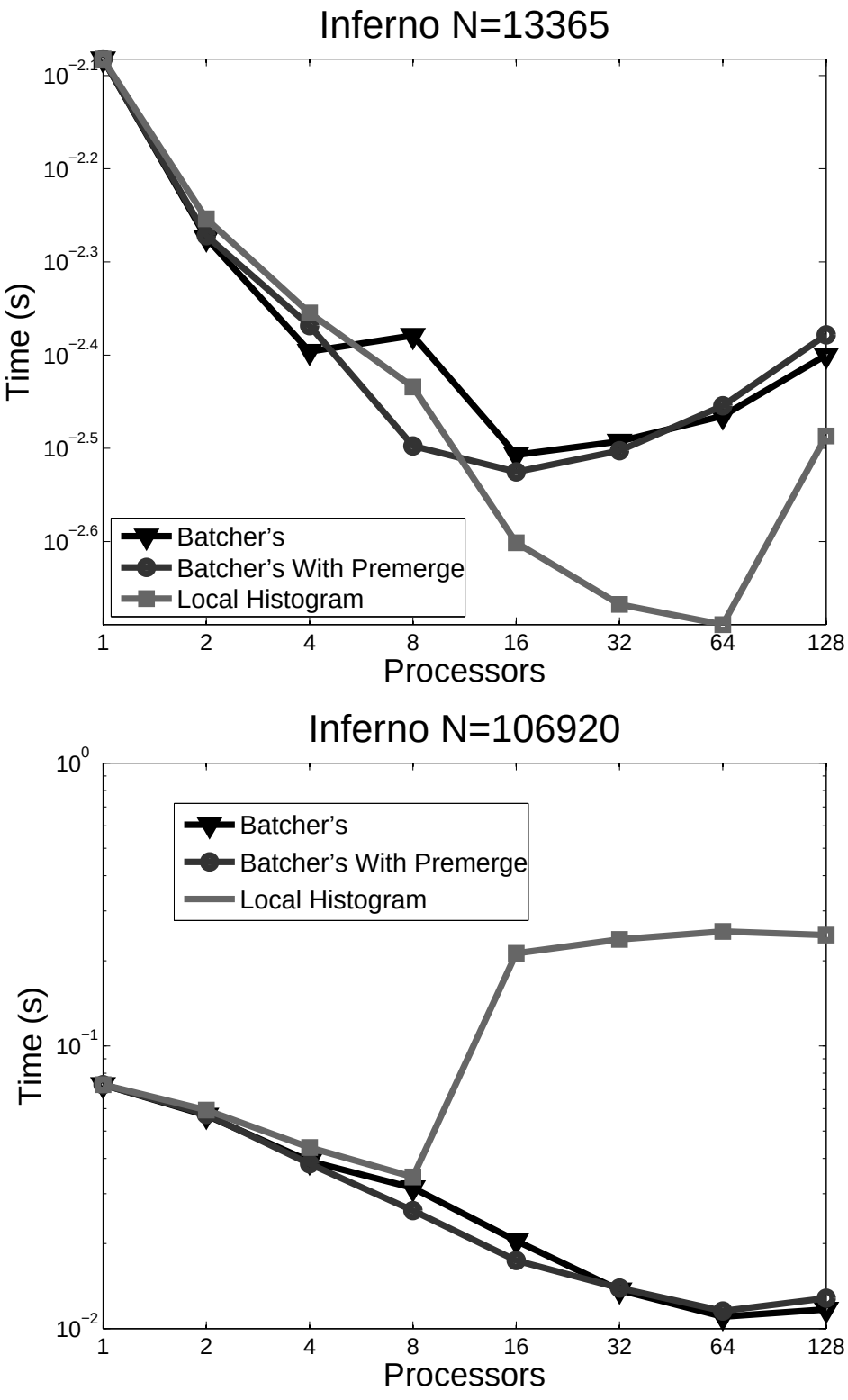



**Figure 5.11**. The strong scalability of the smaller benchmarks on Inferno

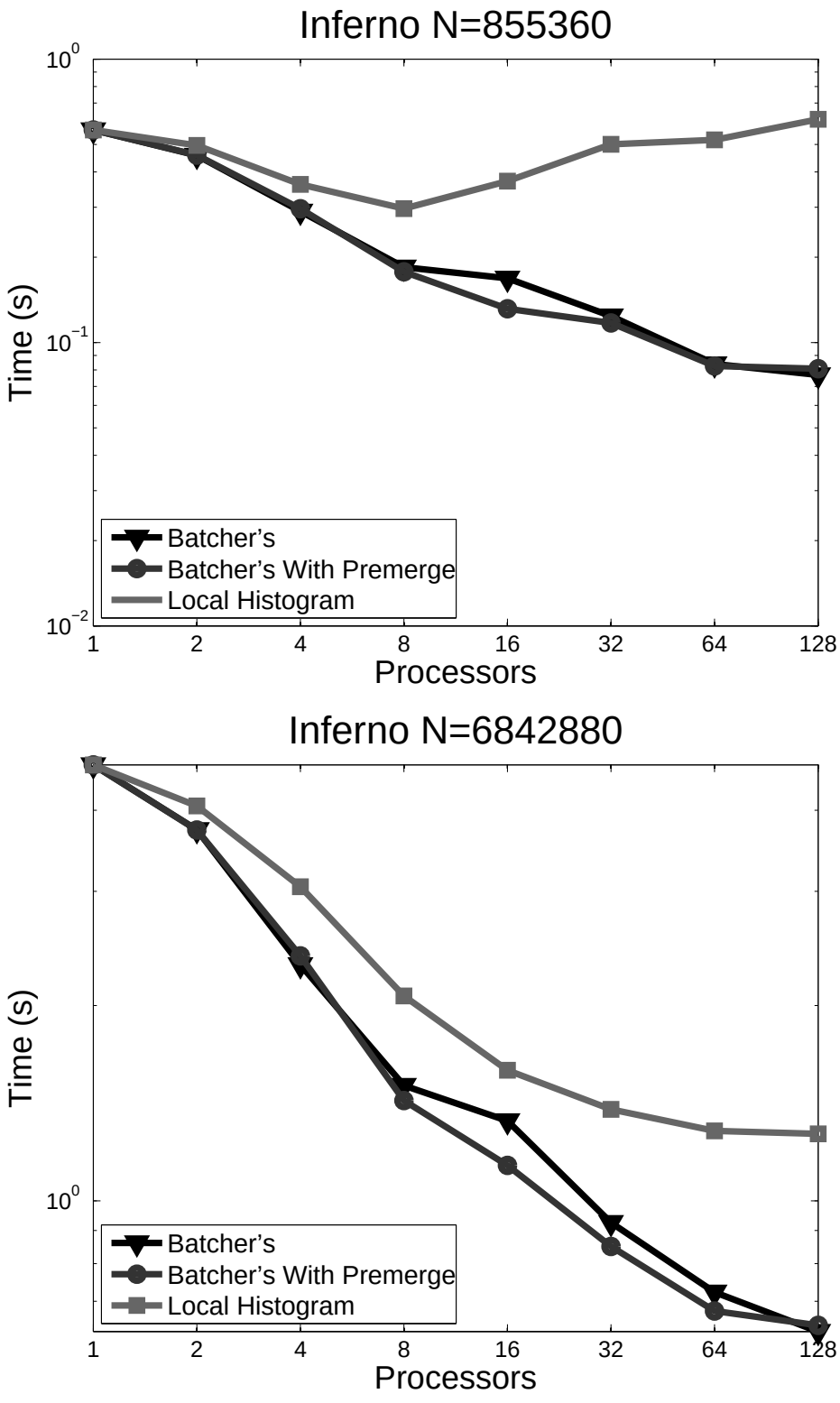



**Figure 5.12**. The strong scalability of the larger benchmarks on Inferno

### 5.3.5 Performance Model

To understand how these algorithms would perform on larger numbers of processors, performance models were used to predict performance. Performance models were derived from the previous complexity analysis.

The complexity of the components are the following:

$$\begin{aligned} \text{Serial Sort :} \quad & O(\frac{N}{P}\log N), \\ \text{Batcher's :} \quad & O(\frac{N}{P}\log^2 P), \\ \text{Local Histogram :} \quad & O((P+\frac{N}{P})\log P). \end{aligned}$$

Using these complexities, the following performance models were derived:

$$\begin{aligned} \text{Serial Sort :} \quad & c_1\frac{N}{P}\log_2 N, \\ \text{Batcher's :} \quad & (c_2+c_3\frac{N}{P})\frac{\log_2^2 P+\log_2 P}{2}, \\ \text{Local Histogram :} \quad & (c_4 P+c_5\frac{N}{P})\log_2 P, \end{aligned}$$

where $c_1$ represents the cost of performing one recursive step of Algorithm 3 on an element, $c_2$ represents the overhead in starting a merge-exchange, $c_3$ represents the cost of performing a merge-exchange per element, $c_4$ represents the cost of sending and processing on one element of the histogram, and $c_5$ represents the cost of communicating and merging a single element.

Figures 5.13 and 5.14 show the timings on Thunder compared with the performance models. The performance models used the constants found in Table 5.1. These constants were estimated by fitting the observed times to the performance models.

A comparison between the models and the performance results provide evidence that the complexity analysis used to derive the models is correct. The models suggest that we can expect scaling using both Batcher's method and the local histogram

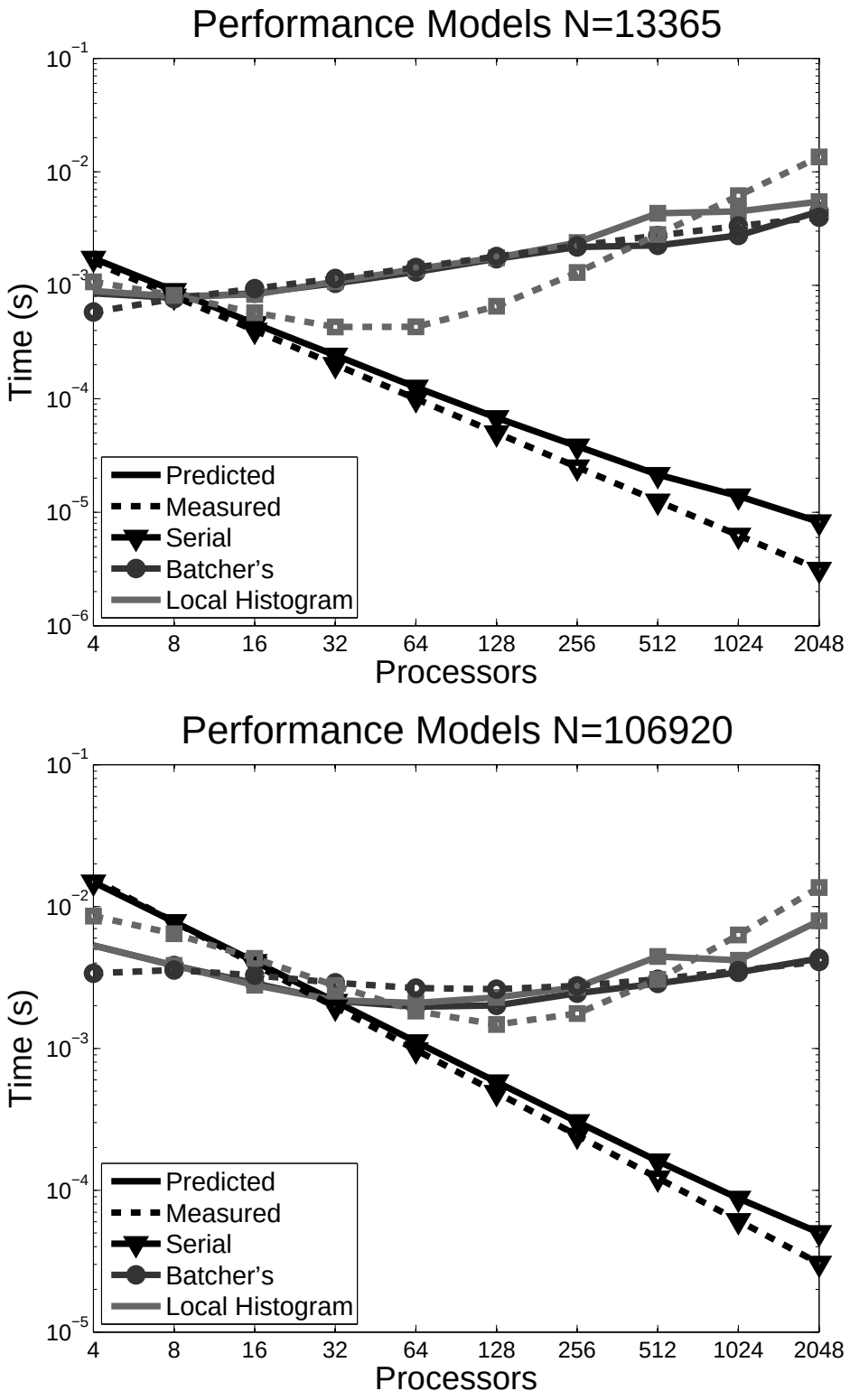



**Figure 5.13**. The performance models versus the small benchmark timings. Solid lines are the measured times and dashed lines are the predicted times.

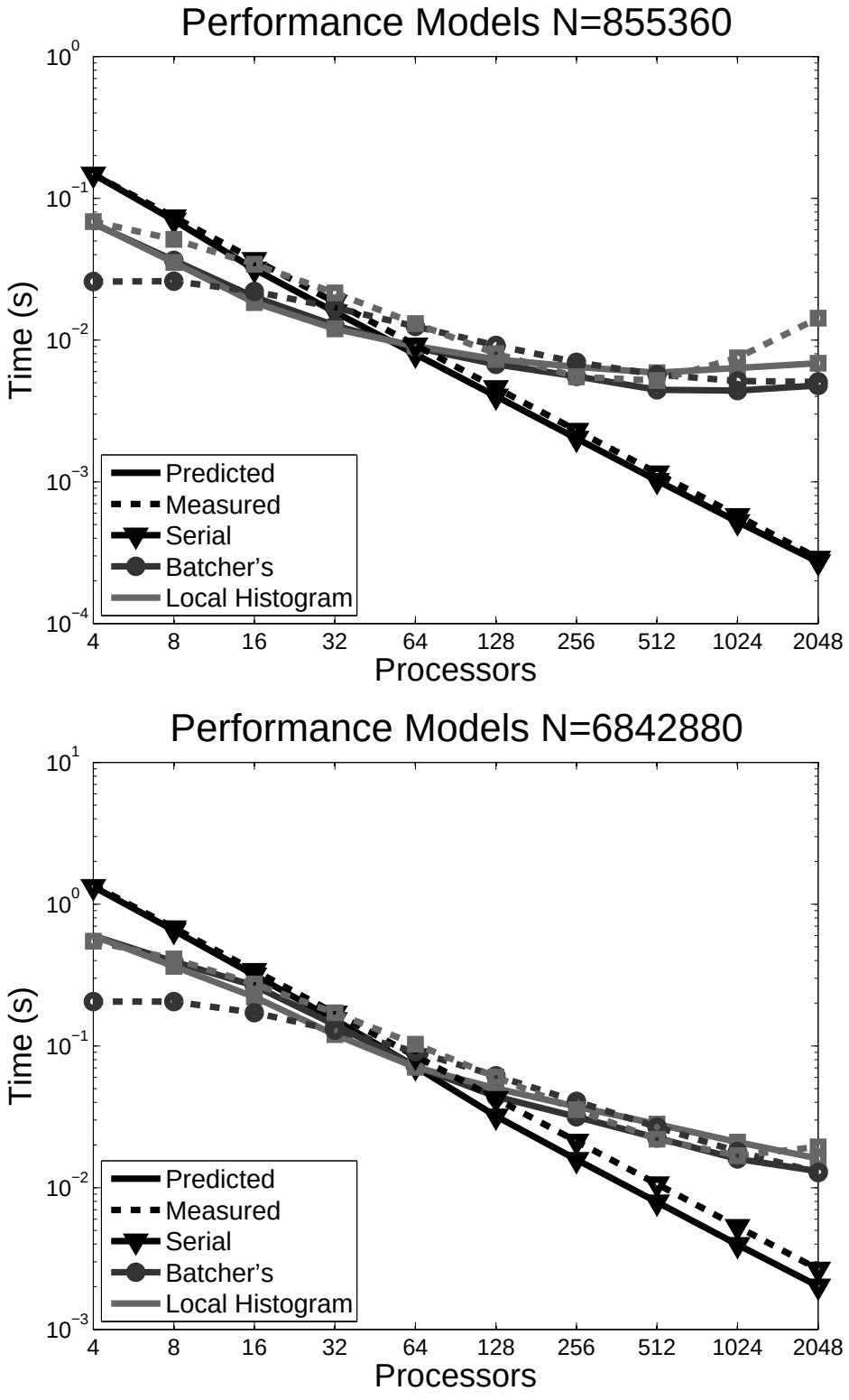



**Figure 5.14**. The performance models versus the large benchmark timings. Solid lines are the measured times and dashed lines are the predicted times.

**Table 5.1**. The constants used to predict performance on Thunder

| constant | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ |
|---|---|---|---|---|---|
| value | $3.5e^{-8}$ | $3e^{-5}$ | $2e^{-8}$ | $6e^{-7}$ | $1.6e^{-7}$ |

method when $\frac{N}{P}$ is large enough to overcome overhead associated with constants $c_2$ and $c_4$. Batcher's method may scale better at larger numbers of processors than the local histogram method because the histogram introduces an overhead that is proportional to the number of processors.

Figure 5.15 and Figure 5.16 show the predicted timings up to $2^{18}$ processors for various problem sizes on a machine similar to Thunder. On the smaller problem sizes, the local histogram method reaches the lowest time but does significantly worse if too many processors are used. Batcher's method also degrades in performance when too many processors are used but in a less significant way than for the local histogram method. Performance runs of Uintah at 98,304 cores will confirm these results in Chapter 6. On the larger problem sizes, Batcher's method scales to larger numbers of processors. This is due to the overhead in the local histogram method. This suggests that for small and moderate numbers of processors, the local histogram method may be better than Batcher's, but in the case of large numbers of processors, Batcher's should be used.

### 5.3.6 Conclusions and Future Work

Space-filling curves have many applications within scientific computing, particularly in adaptive mesh refinement. For overall scalability, it is important that curve generation is done quickly in parallel. We have shown that space-filling curves can be generated quickly in parallel through sorting. The merging method that will provide the best results depends highly on machine architecture, the number of processors, and the size of the problem. For large numbers of processors, Batcher's method should perform better than the local histogram method. On the architectures considered here, we did not see a significant benefit to premerging; however, on machines with low latency and low bandwidth, it may be beneficial to premerge some of the data. However, such architectures are relatively unusual.

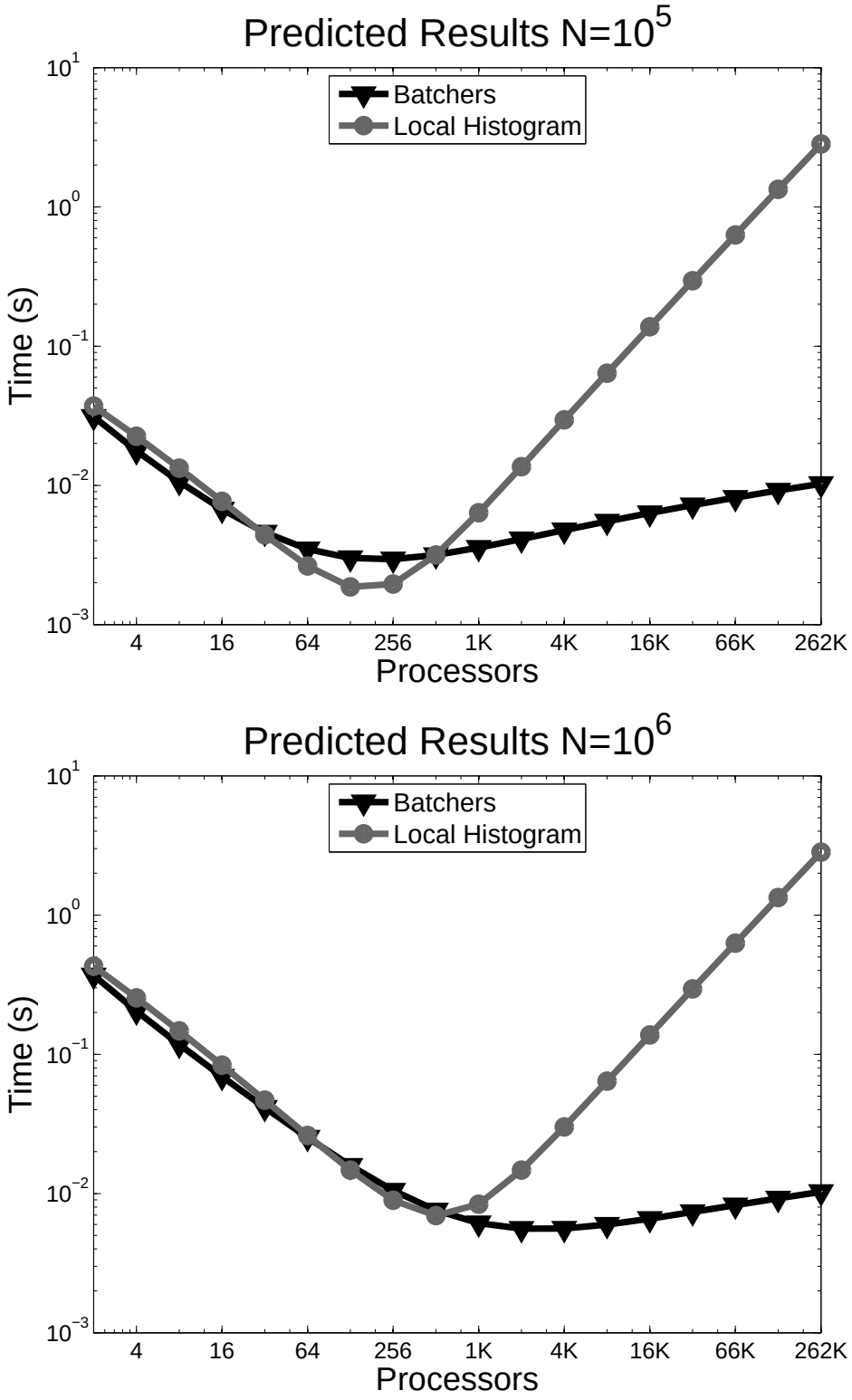



**Figure 5.15**. Predicted timings for $10^5$ and $10^6$ elements at large processor counts

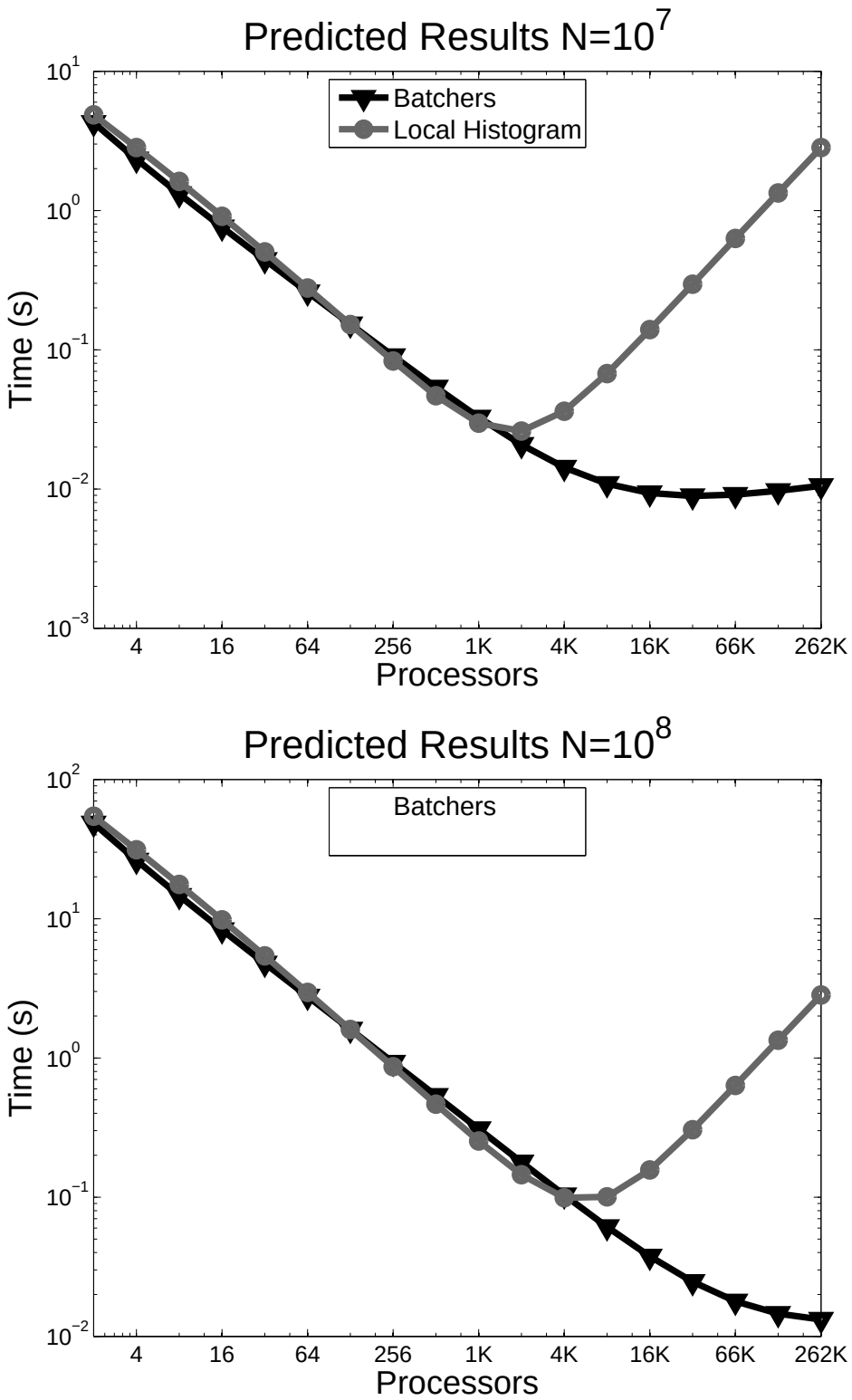



**Figure 5.16**. Predicted timings for $10^7$ and $10^8$ elements at large processor counts

To increase scalability further, overheads within the algorithm must be reduced. The primary overhead is due to communication that could be reduced through fast compression schemes. As elements are sorted, the significant bits become equivalent. A compression mechanism could possibly take advantage of this redundancy, greatly reducing the amount of communication.

Generation of the space-filling curve could also be done in an incremental manner. Storing the old curve and indices would make it possible to add new points incrementally. If there are few changes to the mesh, indices could be generated for new points and then the points could be binary inserted into the old curve. If there were a large number of changes to the mesh, a second curve could be generated over the new mesh points and the old and new curves could then be merged.

## 5.4 Summary and Conclusions

Uintah's load balancer was designed to operate quickly in parallel and support a wide variety of simulations [74, 76]. To achieve these design goals, Uintah utilizes automated machine tuning to estimate the workload [74]. This can be done using a combination of models and least squares analysis or by using filtering methods. Both of these methods provide highly accurate estimations without requiring a large amount of effort or knowledge from the user. This allows users to easily add an effective load balancer to their simulation with little knowledge of the load balancing process or the underlying simulation models.

In addition, Uintah utilizes a highly parallel space-filling curve algorithm. This algorithm exhibits good strong scalability and executes quickly in parallel. However, this load balancer requires global metadata. It is possible that as the number of processors and patches increase, the cost of maintaining this global metadata may hinder performance. In this case, it may be worthwhile to move to a distributed load balancing algorithm like those based upon diffusion algorithms [28, 50, 51, 52, 59]. Nevertheless, the present algorithm described here was used on machines with up to 98,304 processors, as shown in Chapter 6.

# CHAPTER 6

# UINTAH SCALABILITY

AMR frameworks such as Uintah are a useful tool for researchers. These frameworks reduce the burden of creating a parallel AMR simulation and are thus being used throughout the scientific community [19, 24, 37, 41, 47, 49, 79, 85, 87, 108]. Understanding and improving the scalability of such frameworks is an important task that could have wide-ranging impact across the research community.

This chapter analyzes the scalability of Uintah as a whole using up to 98,304 processors. The analysis includes benchmarks for both ICE and MPMICE along with both AMR and single level mesh simulations.

## 6.1 Benchmarking Setup

The scalability of Uintah was tested on Kraken[1] using up to 98,304 processors for both single level and AMR problems using both ICE and MPMICE. In addition, the scalability was tested for an array of explosives, which represents a full-physics simulation that is currently being investigated [13]. The following sections will describe the test problems and analyze the weak and strong scaling of Uintah in the context of these benchmarks using up to 98304 cores.

### 6.1.1 ICE Benchmarks

The performance of ICE was tested in both the single level and AMR case using a simulation of the transport of two fluids with a prescribed initial velocity of Mach two, as seen in Figure 6.1. For this problem, the conservation of mass, momentum, and energy equations were solved for two inviscid fluids. The fluids exchange momentum and heat through the exchange terms in the governing equations, as described in

[1]Kraken is a supercomputer located at the University of Tennessee with 99,072 cores. More information is available at `http://www.nics.tennessee.edu/computing-resources/kraken`.

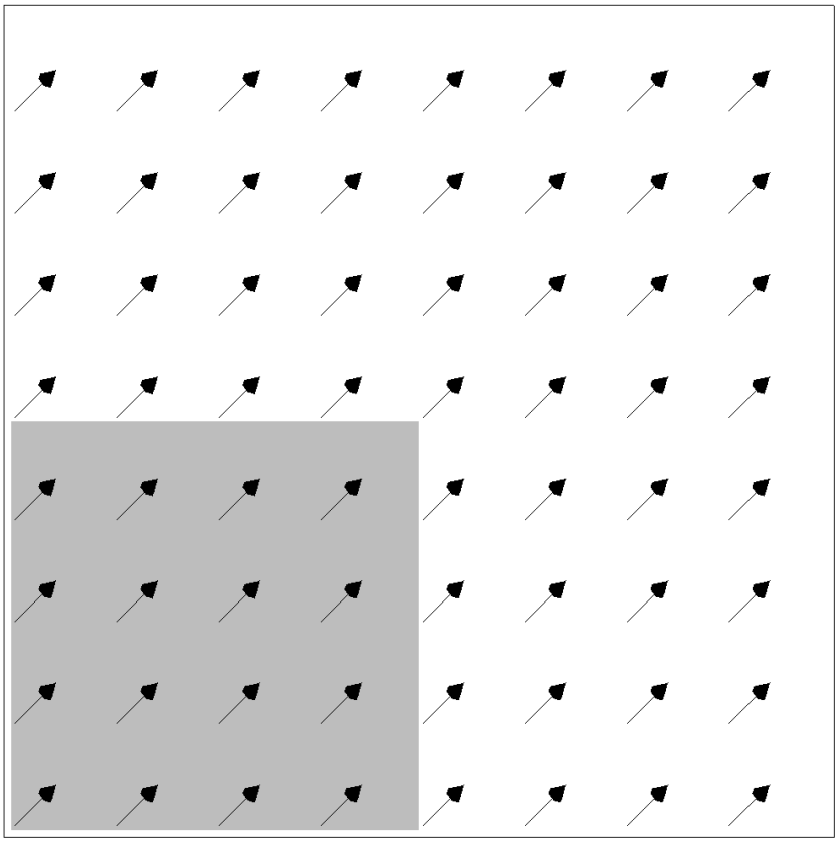


**Figure 6.1**. The ICE scalability benchmark problem

Section 3.2.1.1. This simulation is an explicit formulation and utilized the w-cycle execution model for time stepping, as described in Section 3.6.1. This problem exercises all of the main features of ICE and amounts to solving eight P.D.Es, along with two point-wise solves, and one iterative solve. For more information, see Section 3.2.1.

#### 6.1.1.1 Single Level ICE Benchmark

The single level ICE benchmark involved four runs of varying sizes with each run using a mesh refined by a factor of two with respect to the previous run. This led to each run containing eight times more cells than the previous run. Each mesh was composed of patches that were $16^3$ cells in size. The execution time for 50 timesteps was measured. The setup for each of the four runs is described in Table 6.1.

#### 6.1.1.2 AMR ICE

The AMR ICE benchmark involved five runs of varying sizes with each run using a mesh that was refined by a factor of two with respect to the previous run. The refinement algorithm used tracked the interface between the two materials, causing the simulation to regrid often while maintaining a fairly constant sized grid, which allows the scalability to be more accurately measured. This criteria led to each problem being about four times as large as the previous one. This mesh was composed of patches of $8^3$ cells in size and the time for 50 timesteps was measured. The simulation utilized three mesh refinement levels with each level being a factor of four more refined than the previous level. The simulation regridded whenever the inner material moved outside the bounds of the finest level which amounted to between 6-8 regrids per run. The setup for each of the five runs is described in Table 6.2.

**Table 6.1**. The single level ICE benchmark setup

| Benchmark Run | # of Patches | # of Processors |
|---|---|---|
| 1 | 4096 | 24-768 |
| 2 | 32K | 192-6144 |
| 3 | 262K | 1536-49152 |
| 4 | 2.1M | 12288-98304 |

**Table 6.2**. The AMR ICE benchmark setup

| Benchmark Run | # of Patches | # of Processors | # of Regrids |
|---|---|---|---|
| 1 | 1.5K | 12-1536 | 6 |
| 2 | 6.1K | 48-6144 | 7 |
| 3 | 24.5K | 192-24576 | 6 |
| 4 | 98.3K | 768-98304 | 7 |
| 5 | 393K | 3072-98304 | 8 |

### 6.1.2 MPMICE Benchmarks

A simulation similar to the ICE benchmark was also used to test MPMICE. In this simulation, a solid explosive was transported through air at Mach two, as seen in Figure 6.2. The simulation used an explicit formulation and used the lock-step timestepping algorithm described in Section 3.6.1. This simulation exercises many of the same components as the ICE benchmark but also represents the explosive with MPM particles. This benchmark also included a model for the deflagration of the explosive [117, 120] and the material damage model ViscoSCRAM [7]. This problem closely resembles CSAFE's exploding container problem, as described in Chapter 1.

#### 6.1.2.1 Single Level MPMICE

The single level MPMICE benchmark involved four runs of varying sizes with each run using a mesh that was refined by a factor of two with respect to the previous run. This led to each problem having approximately eight times more cells and particles then the previous problem. The mesh was composed of patches that were $8^3$ cells in size and contained upwards of 4096 MPM particles. The time for 150 timesteps was measured. The setup for each of the four runs is described in Table 6.3.

**Table 6.3**. The single level MPMICE benchmark setup

| Benchmark Run | # of Patches | # of Particles | # of Processors |
|---|---|---|---|
| 1 | 1.7K | 651K | 12-1536 |
| 2 | 14K | 5.1M | 96-12288 |
| 3 | 111K | 41M | 768-49152 |
| 4 | 885M | 330M | 12288-49152 |

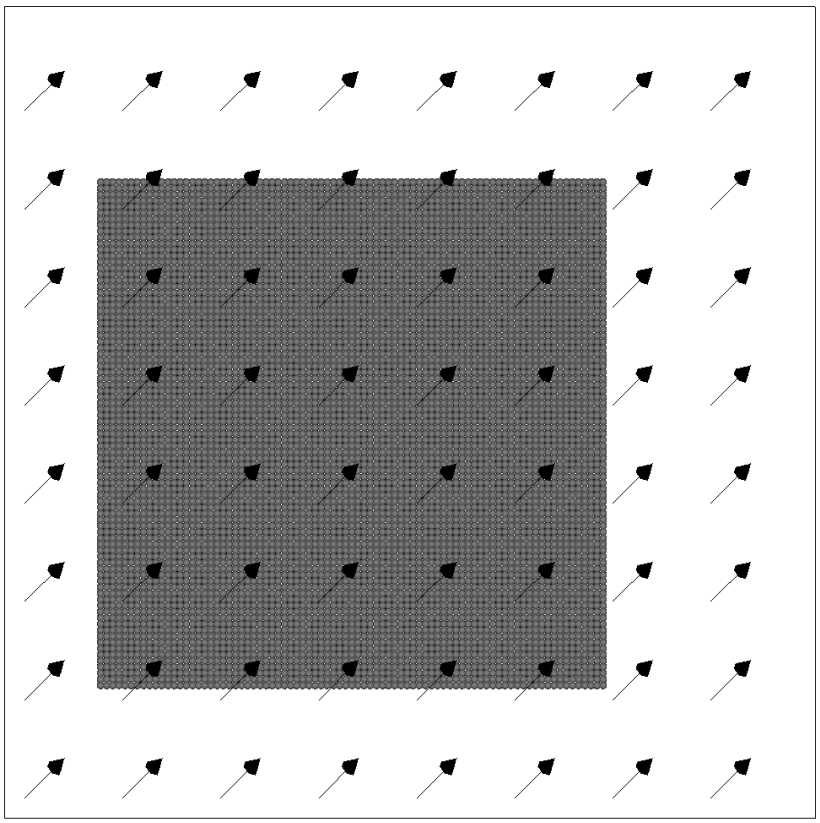


**Figure 6.2**. The MPMICE scalability benchmark problem

#### 6.1.2.2 AMR

Similarly, the AMR MPMICE benchmark involved four runs of varying sizes with each run using a mesh that was refined by a factor of two with respect to the previous run. This led to each run being approximately eight times as large as the previous run. The mesh was composed of patches of $8^3$ cells in size which contained upwards of 4096 MPM particles. The simulation utilized three refinement levels with each level being a factor of four more refined than the previous level. The time for 150 timesteps was measured. The simulation regridded whenever a particle was relocated outside of the bounds of the finest mesh which amounted to 3-4 regrids per run. The setup for each of the four runs is described in Table 6.4.

#### 6.1.2.3 Explosive Array

The final scaling test involved a full-physics simulation of an explosive array during detonation. In this simulation, a high-pressure wave travels through an array of explosives. The wave causes each container to detonate as it passes through, thereby increasing the energy within the explosion. It is believed that such a detonation occurred during the Spanish-Fork accident and research into the causes of the detonation are currently being undertaken [13]. A 2D image from such a simulation can be seen in Figure 6.3. This simulation includes a prototype model for DDT [107, 101, 102] in addition to all the models used in the MPMICE benchmark.

The current AMR refinement criteria for MPMICE problems introduces a fine mesh everywhere that particles exist. Since this problem has particles everywhere within the domain, an AMR benchmark would refine the entire domain. Because of this, only a single level version of this problem was tested. This benchmark involved

**Table 6.4**. The AMR MPMICE benchmark setup

| Benchmark | # of Patches | # of Particles | # of Processors | # of Regrids |
|---|---|---|---|---|
| 1 | 1.3K | 2.1M | 12-1.5K | 3 |
| 2 | 6.8K | 16.8M | 96-12K | 4 |
| 3 | 42K | 134M | 768-98K | 4 |
| 4 | 301K | 1.1B | 6K-98K | 4 |

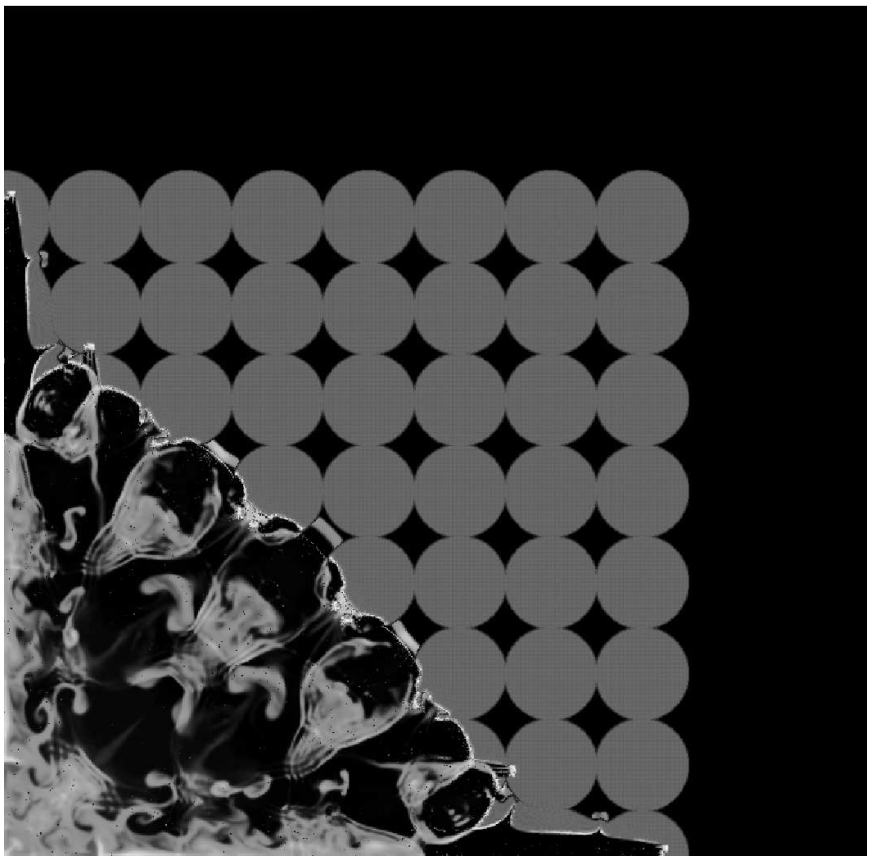


**Figure 6.3**. A 2D image of the explosive array benchmark

four runs of varying sizes. Each run used a mesh that was refined by a factor of two with respect to the previous run. This led to a mesh which contained eight times the number of cells and particles as the previous problem. The meshes were composed of patches of size $8^3$ cells with each patch containing up to 4096 MPM particles. The setup for each of the four runs is described in Table 6.5.

## 6.2 Scalability Results

Figure 6.4 shows the scaling results for the five benchmark tests. This figure shows nearly ideal weak and strong scaling, as defined in Section 1.2, for the single level ICE benchmark. The AMR ICE benchmark also exhibits good strong and weak scalability but the performance degrades on the largest problem and also at the largest numbers of processors. This degradation is primarily due to the cost to migrate data after regridding and load balancing. The MPMICE benchmarks exhibit similar scalability; however, the degradation in performance is more significant than with the ICE benchmark. Finally, the explosive array benchmark exhibits nearly ideal weak and strong scalability.

### 6.2.1 Strong and Weak Efficiency

The strong and weak efficiency metric (SWE) that was presented in Section 1.2 states the efficiency relative to the smallest data point. Table 6.6 shows this efficiency for the single level ICE benchmark. This table shows that the single level ICE problem scales effectively, achieving over 50% efficiency at the largest processor counts. There is a slight decrease in efficiency when increasing the processor counts for both strong and weak scaling.

Table 6.7 shows similar results for the AMR ICE benchmark. However, in this

**Table 6.5**. The explosive array benchmark setup

| Benchmark Run | # of Patches | # of Particles | # of Processors |
|---|---|---|---|
| 1 | 1.7K | 5.8M | 12-1.5K |
| 2 | 14K | 38.5M | 96-12K |
| 3 | 111K | 308M | 768-98K |
| 4 | 885K | 2.5B | 6K-98K |

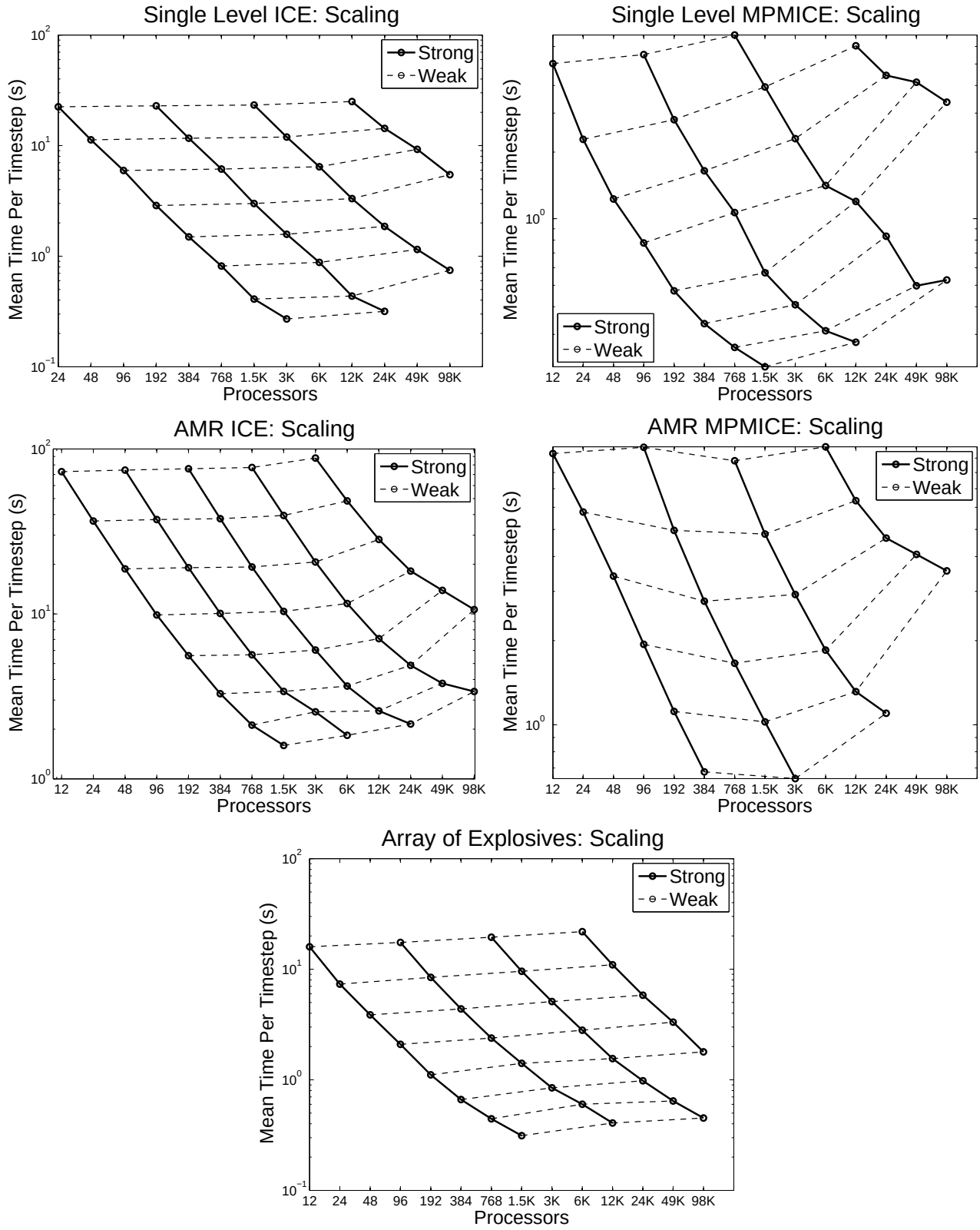



**Figure 6.4**. The weak and strong scalability for the benchmark problems

**Table 6.6**. Single level ICE weak and strong efficiency metric

| | Strong Run 1 | | Strong Run 2 | | Strong Run 3 | | Strong Run 4 | |
|---|---|---|---|---|---|---|---|---|
| Weak # | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE |
| 1 | 24 | **1.00** | 192 | **0.98** | 1536 | **0.96** | 12288 | **0.90** |
| 2 | 48 | **1.00** | 384 | **0.96** | 3072 | **0.94** | 24576 | **0.78** |
| 3 | 96 | **0.94** | 768 | **0.91** | 6144 | **0.87** | 49152 | **0.61** |
| 4 | 192 | **0.98** | 1536 | **0.94** | 12288 | **0.85** | 98304 | **0.51** |
| 5 | 384 | **0.94** | 3072 | **0.89** | 24576 | **0.76** | – | – |
| 6 | 768 | **0.86** | 6144 | **0.80** | 49152 | **0.61** | – | – |
| 7 | 1536 | **0.86** | 12288 | **0.80** | 98304 | **0.47** | – | – |
| 8 | 3072 | **0.65** | 24576 | **0.55** | – | – | – | – |

**Table 6.7**. AMR ICE weak and strong efficiency metric

| | Strong Run 1 | | Strong Run 2 | | Strong Run 3 | | Strong Run 4 | | Strong Run 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Weak # | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE |
| 1 | 12 | **1.00** | 48 | **0.98** | 192 | **0.96** | 768 | **0.94** | 3072 | **0.83** |
| 2 | 24 | **1.00** | 96 | **0.98** | 384 | **0.96** | 1536 | **0.92** | 6144 | **0.75** |
| 3 | 48 | **0.97** | 192 | **0.95** | 768 | **0.95** | 3072 | **0.88** | 12288 | **0.64** |
| 4 | 96 | **0.92** | 384 | **0.90** | 1536 | **0.88** | 6144 | **0.79** | 24576 | **0.50** |
| 5 | 192 | **0.82** | 768 | **0.81** | 3072 | **0.75** | 12288 | **0.64** | 49152 | **0.33** |
| 6 | 384 | **0.69** | 1536 | **0.67** | 6144 | **0.62** | 24576 | **0.47** | 98304 | **0.21** |
| 7 | 768 | **0.54** | 3072 | **0.45** | 12288 | **0.44** | 49152 | **0.30** | – | – |
| 8 | 1536 | **0.36** | 6144 | **0.31** | 24576 | **0.26** | 98304 | **0.17** | – | – |

case, the efficiency decreases more rapidly, leading to a total efficiency of approximately 20% at the largest scales.

The MPMICE benchmark achieves a much lower efficiency in both the single level and AMR benchmarks, as seen in Tables 6.8 and 6.9. In these cases, the efficiency drops as low as 7.5% for the single level benchmark. The AMR benchmark achieves a higher efficiency of 16% at the largest scales. The cause of this increase in efficiency for the AMR case is due to a higher load imbalance in the single level problem which will be discussed later.

Finally, the efficiency for the explosive array can be seen in Table 6.10. This table shows that the efficiency of the explosive array benchmark is similar to single level ICE benchmark, achieving an efficiency of 56% at the largest scales.

**Table 6.8**. Single level MPMICE weak and strong efficiency metric

| | Strong Run 1 | | Strong Run 2 | | Strong Run 3 | | Strong Run 4 | |
|---|---|---|---|---|---|---|---|---|
| Weak # | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE |
| 1 | 12 | **1.0000** | 96 | **0.9103** | 768 | **0.7423** | – | – |
| 2 | 24 | **1.0998** | 192 | **0.8968** | 1536 | **0.6377** | 12288 | **0.4154** |
| 3 | 48 | **1.0234** | 384 | **0.7629** | 3072 | **0.5459** | 24576 | **0.2826** |
| 4 | 96 | **0.8083** | 768 | **0.5892** | 6144 | **0.4448** | 49152 | **0.1517** |
| 5 | 192 | **0.6650** | 1536 | **0.5510** | 12288 | **0.2624** | 98304 | **0.0934** |
| 6 | 384 | **0.4678** | 3072 | **0.3845** | 24576 | **0.1884** | – | – |
| 7 | 768 | **0.2997** | 6144 | **0.2520** | 49152 | **0.1577** | – | – |
| 8 | 1536 | **0.1832** | 12288 | **0.1419** | 98304 | **0.0743** | – | – |

**Table 6.9**. AMR MPMICE weak and strong efficiency metric

| | Strong Run 1 | | Strong Run 2 | | Strong Run 3 | | Strong Run 4 | |
|---|---|---|---|---|---|---|---|---|
| Weak # | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE |
| 1 | 12 | **1.00** | 96 | **0.95** | 768 | **1.06** | 6144 | **0.95** |
| 2 | 24 | **0.81** | 192 | **0.94** | 1536 | **0.97** | 12288 | **0.74** |
| 3 | 48 | **0.69** | 384 | **0.84** | 3072 | **0.80** | 24576 | **0.50** |
| 4 | 96 | **0.60** | 768 | **0.70** | 6144 | **0.63** | 49176 | **0.29** |
| 5 | 192 | **0.52** | 1536 | **0.57** | 12288 | **0.44** | 98304 | **0.16** |
| 6 | 384 | **0.43** | 3072 | **0.46** | 24576 | **0.27** | – | – |

**Table 6.10**. Explosive array weak and strong efficiency metric

| | Strong Run 1 | | Strong Run 2 | | Strong Run 3 | | Strong Run 4 | |
|---|---|---|---|---|---|---|---|---|
| Weak # | Cores | SWE | Cores | SWE | Cores | SWE | Cores | SWE |
| 1 | 12 | **1.00** | 96 | **0.91** | 768 | **0.82** | 6144 | **0.73** |
| 2 | 24 | **1.09** | 192 | **0.95** | 1536 | **0.83** | 12288 | **0.73** |
| 3 | 48 | **1.03** | 384 | **0.91** | 3072 | **0.78** | 24576 | **0.69** |
| 4 | 96 | **0.95** | 768 | **0.84** | 6144 | **0.71** | 49152 | **0.60** |
| 5 | 192 | **0.90** | 1536 | **0.71** | 12288 | **0.64** | 98304 | **0.56** |
| 6 | 384 | **0.75** | 3072 | **0.59** | 24576 | **0.51** | – | – |
| 7 | 768 | **0.56** | 6144 | **0.41** | 49152 | **0.39** | – | – |
| 8 | 1536 | **0.40** | 12288 | **0.31** | 98304 | **0.28** | – | – |

### 6.2.2 Performance Breakdown

To determine what is limiting the scalability within each benchmark, it is important to separate the performance results into the following components: the execution time (time executing tasks), the global communication (time spent with MPI collectives), the time spent packing and unpacking messages in MPI, and the waiting time (time spent within MPI_Wait). In addition, the AMR problems also contain the times for regridding, load balancing, scheduling, and migrating data between grids.

### 6.2.3 ICE

Figure 6.5 shows strong and weak scalability breakdowns for the ICE benchmarks. This figure shows that the components for single level ICE scale nearly ideally in the weak and strong sense. However, in the weak scaling case, the time for communications (global communication, wait time, and message packing) does increase slightly with the problem size.

Figure 6.6 shows the same breakdown for the AMR ICE benchmark. This figure shows that multiple components do not scale ideally in the weak or strong sense. In particular in the weak case, the AMR components increase significantly with the number of processors. The time to migrate data between grids is the most significant. The data migration step occurs after a regrid. This step is responsible for migrating data between processors for regions of the domain that have been assigned to different processors. In addition, this step will also create fine mesh data from the coarser mesh

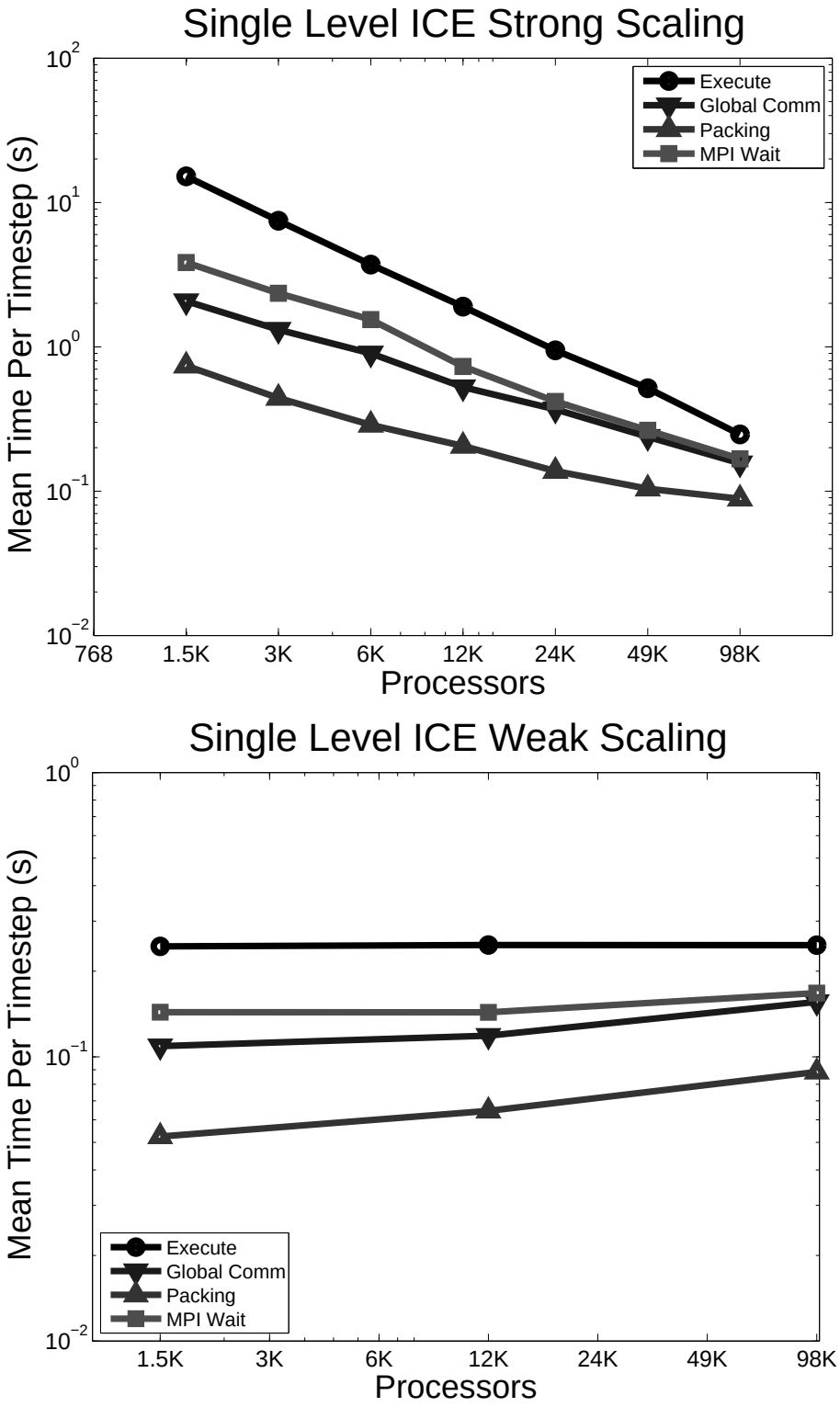



**Figure 6.5**. A breakdown of the scalability for the single level ICE benchmark

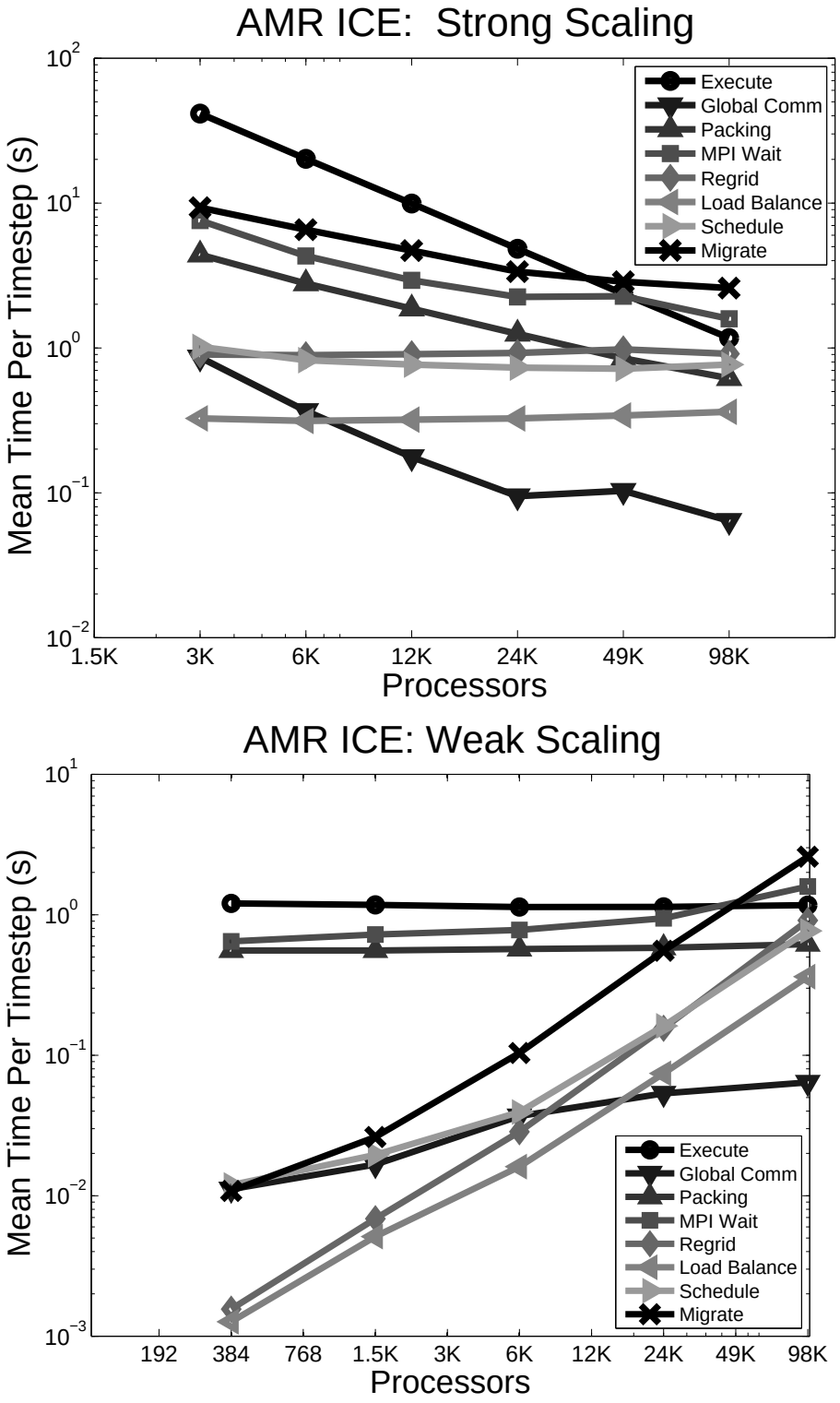



**Figure 6.6**. A breakdown of the scalability for the AMR ICE benchmark

levels as necessary. It is also important to note that some of this increase is due to the larger problems regridding more often than the smaller problems, as shown in Table 6.2. It is also worth noting that the time for load balancing is very low and match the predictions found in Chapter 5.

### 6.2.4 MPMICE

Figure 6.7 shows the same breakdown for the single level MPMICE benchmark. This figure shows that both the strong and weak scalability for the single level problem is limited by the wait time and the global communication time in both the weak and strong cases.

Figure 6.8 shows the breakdown for the AMR MPMICE benchmark. This figure shows that the wait time is the most significant factor in limiting scalability. In addition, the time for the AMR components exhibits poor weak scaling and begin to limit the overall scalability.

The source of the high wait times can be due to either the large amount of data and number of messages that are communicated (communication time) or synchronization caused by load imbalance. When weak scaling, the amount of communication per processor should not increase significantly though the time to communicate data may increase slightly due to increased network complexities. Thus, some increase is expected in the wait time; however, it is more likely that the wait time is due to an increase in synchronization time. The load imbalance for these benchmarks is investigated further in Section 6.2.5.

Finally, the breakdowns for the array of explosives are seen in Figure 6.9. This figure shows nearly ideal weak and strong scaling for the components. The high wait times observed in the MPMICE benchmark were not observed in this benchmark despite the similarity between two problems. This is due to the load imbalance which will be presented in the next section.

### 6.2.5 Load Imbalance

One possible cause for the wait time is load imbalance. A load imbalance causes processors to wait longer for other processors to finish their work. If the load

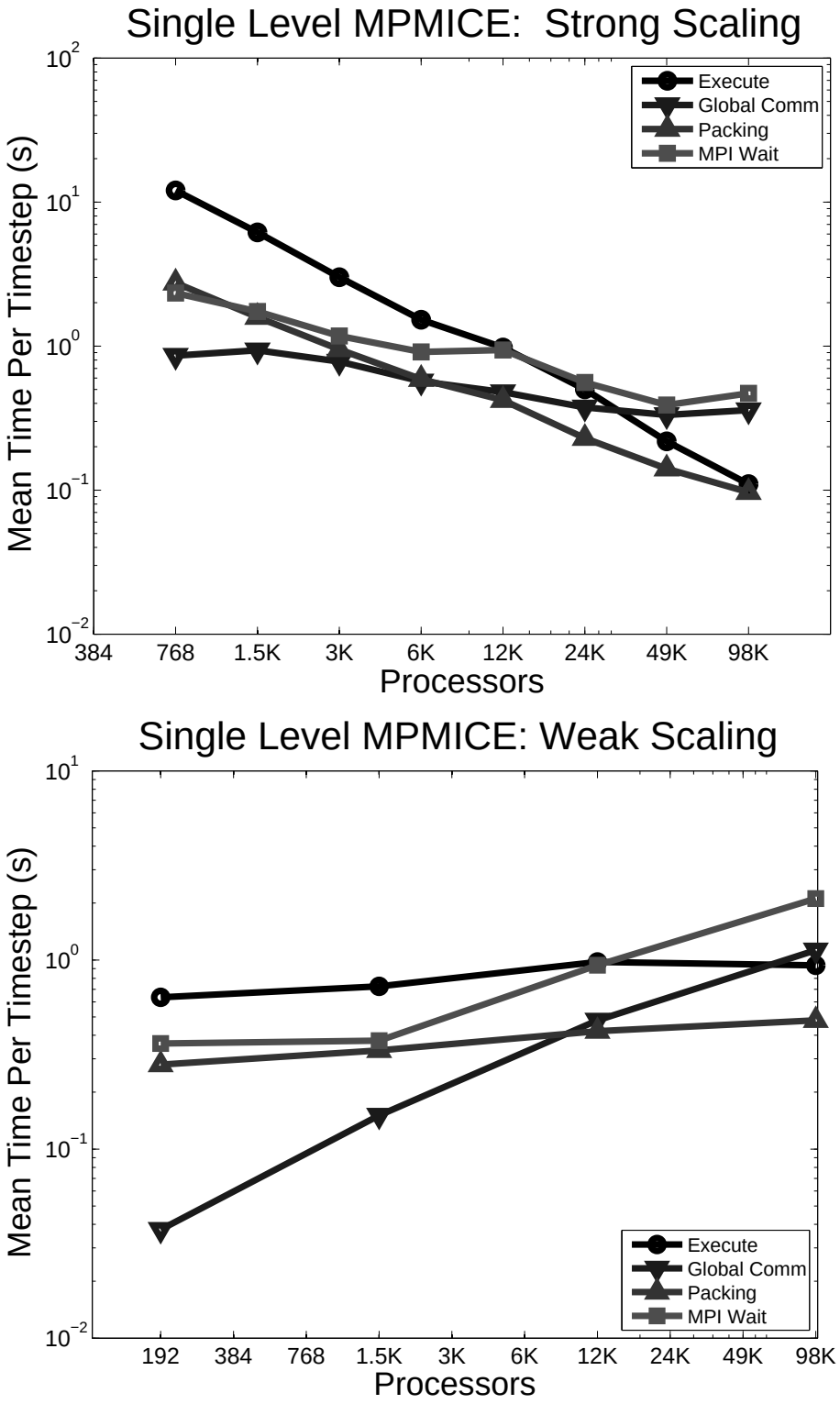



**Figure 6.7**. A breakdown of the scalability for MPMPICE benchmarks

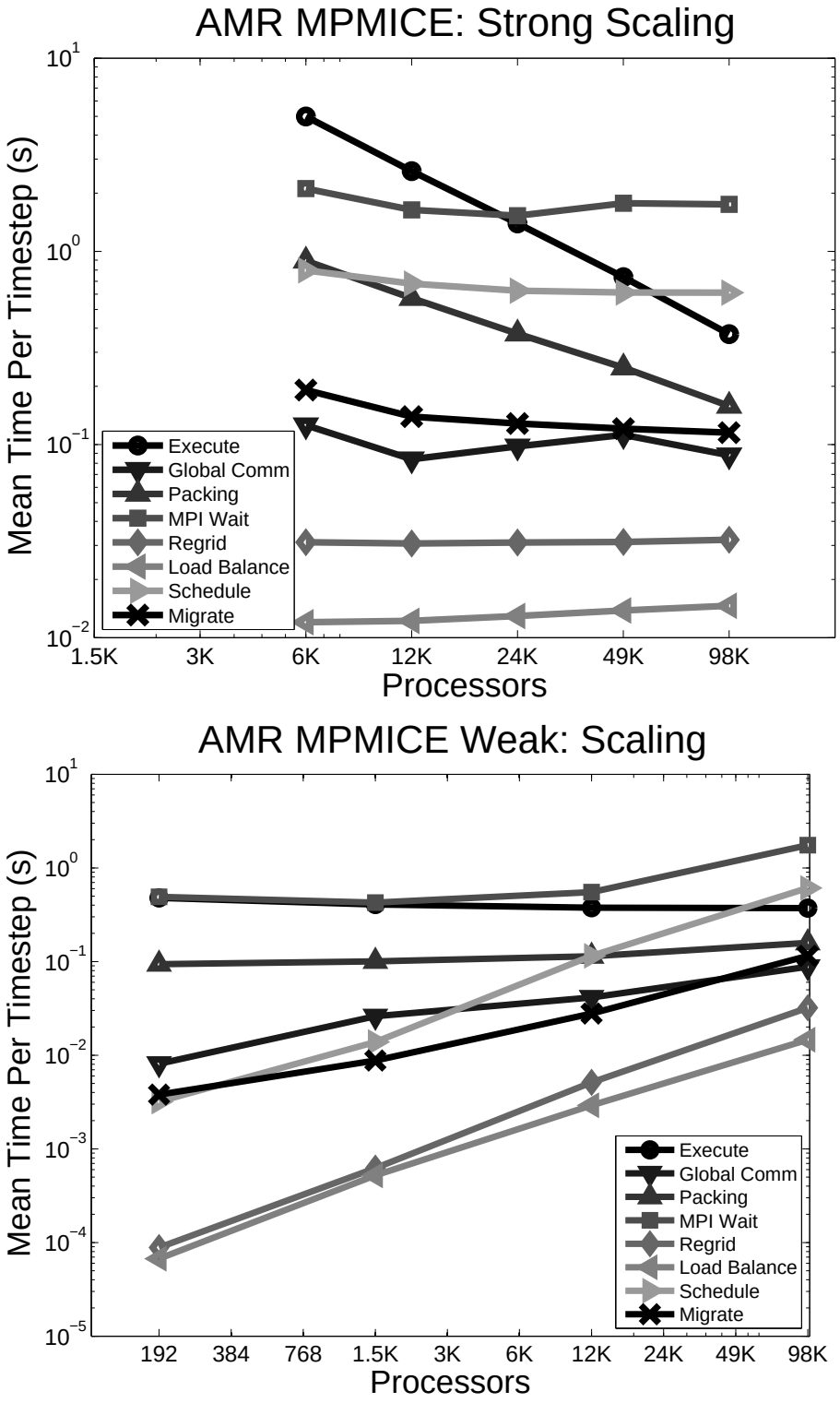



**Figure 6.8**. A breakdown of the scalability for the AMR MPMPICE benchmark

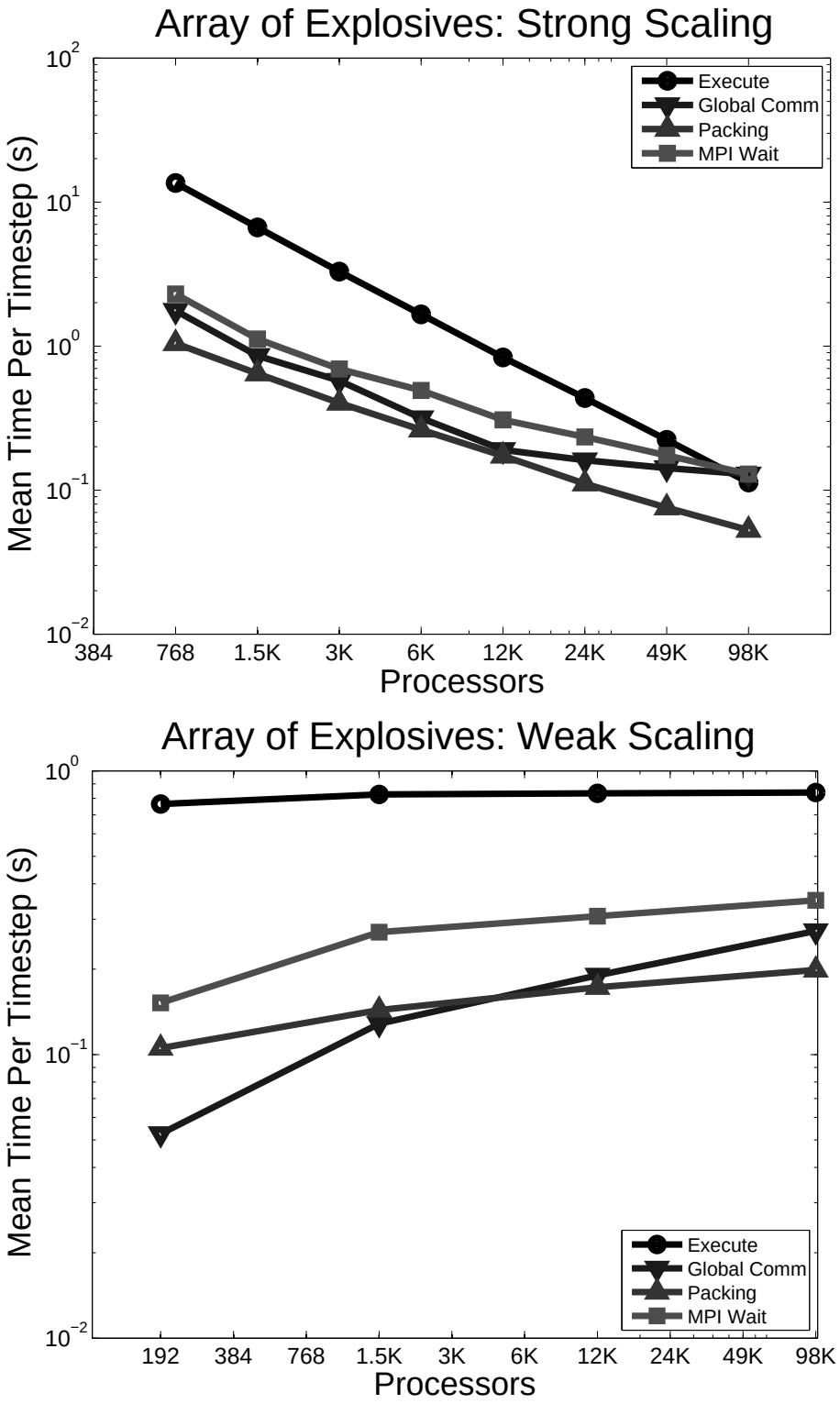



**Figure 6.9**. A breakdown of the scalability for the explosive array benchmark

imbalance increases, then it is expected that the wait time will also increase. Load imbalance is defined as follows:

$$L_{ib} = (1 - \frac{T_{avg}}{T_{max}}) \times 100, \tag{6.1}$$

where the $T_{avg}$ is the average time across all processors and $T_{max}$ is the maximum time across all processors. Thus, a load imbalance of 0 implies the simulation is perfectly load balanced (all processors have the same amount of work).

The load imbalance for the single level ICE simulations can be seen in Figure 6.10. This figure shows the load imbalance is between .1 and .45. When strong scaling, the load imbalance increases with the number of processors. This is expected because the number of patches per processor is decreasing. This implies that the load balance has less units of work to assign and thus has less flexibility in creating the patch assignments. When weak scaling, the load imbalance stays fairly constant. The load imbalance behaves similarly for the AMR ICE benchmark, as shown in Figure 6.11. For this benchmark, the load imbalance ranged from .1 to .4.

The load imbalance does not behave in the same way for the single level MPMICE benchmark, as shown in Figure 6.12. This figure shows the load imbalance is much larger than with the ICE benchmarks with the range of imbalance being between .1 and .9, which is much higher than the ICE benchmarks. In addition, when weak scaling, the imbalance increases with the problem size. This explains the source of the high wait times in the MPMICE benchmarks.

The source of this imbalance is due to the distribution of particles throughout the domain. Most patches either contain no particles or a large number of particles. In addition, due to the burn and damage models, the work per particle is much higher than per cell. This complicates the load balancing process greatly. One challenge for load balancing MPMICE problems is that an ideal load balance for the ICE portion of the algorithm is likely not ideal for the MPM portion of the algorithm and vice-versa. Similar challenges have been observed in contact simulations which required load balancing the particles separately from the mesh [48, 80, 106]. To achieve a better

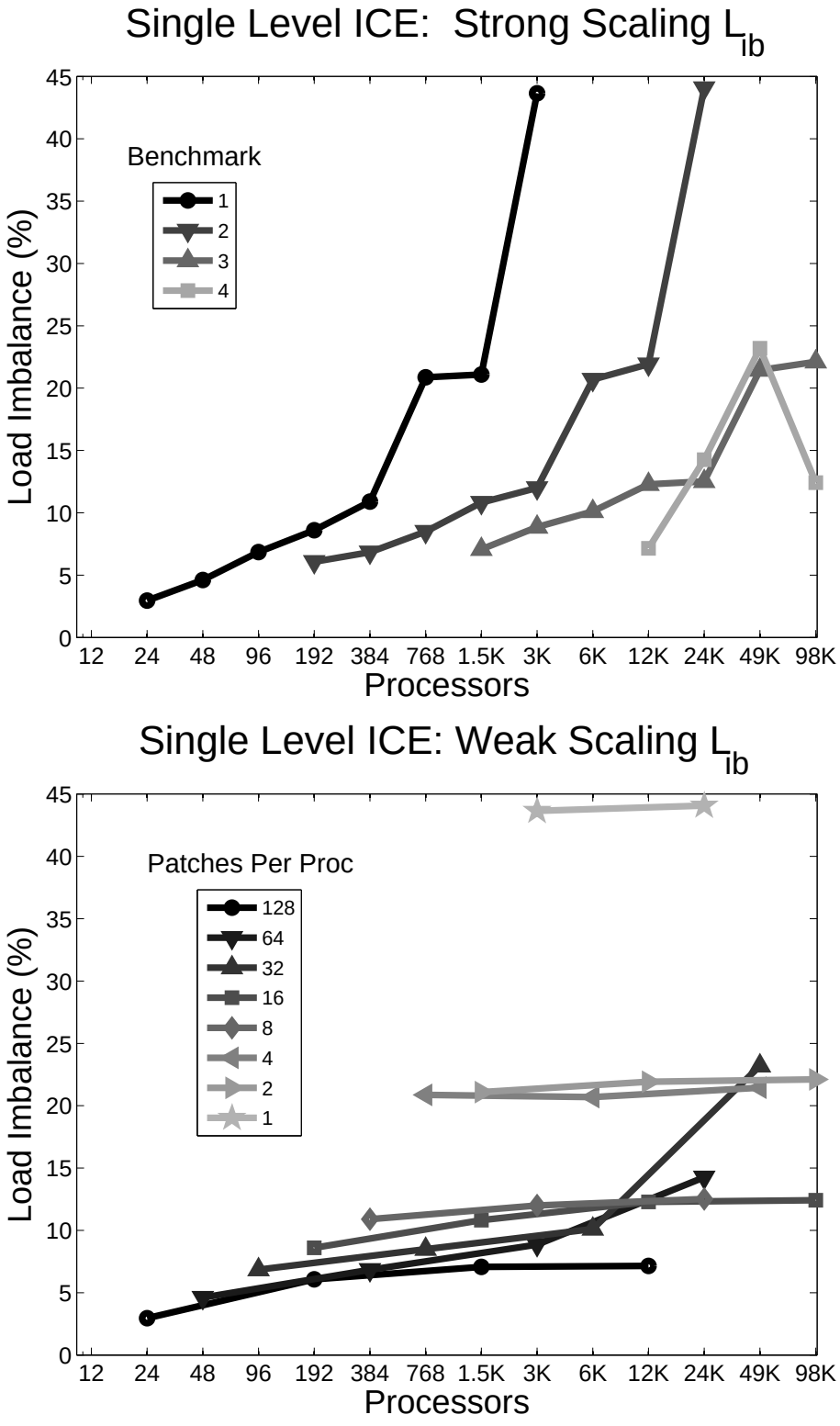



**Figure 6.10**. The load imbalance for the single level ICE benchmark

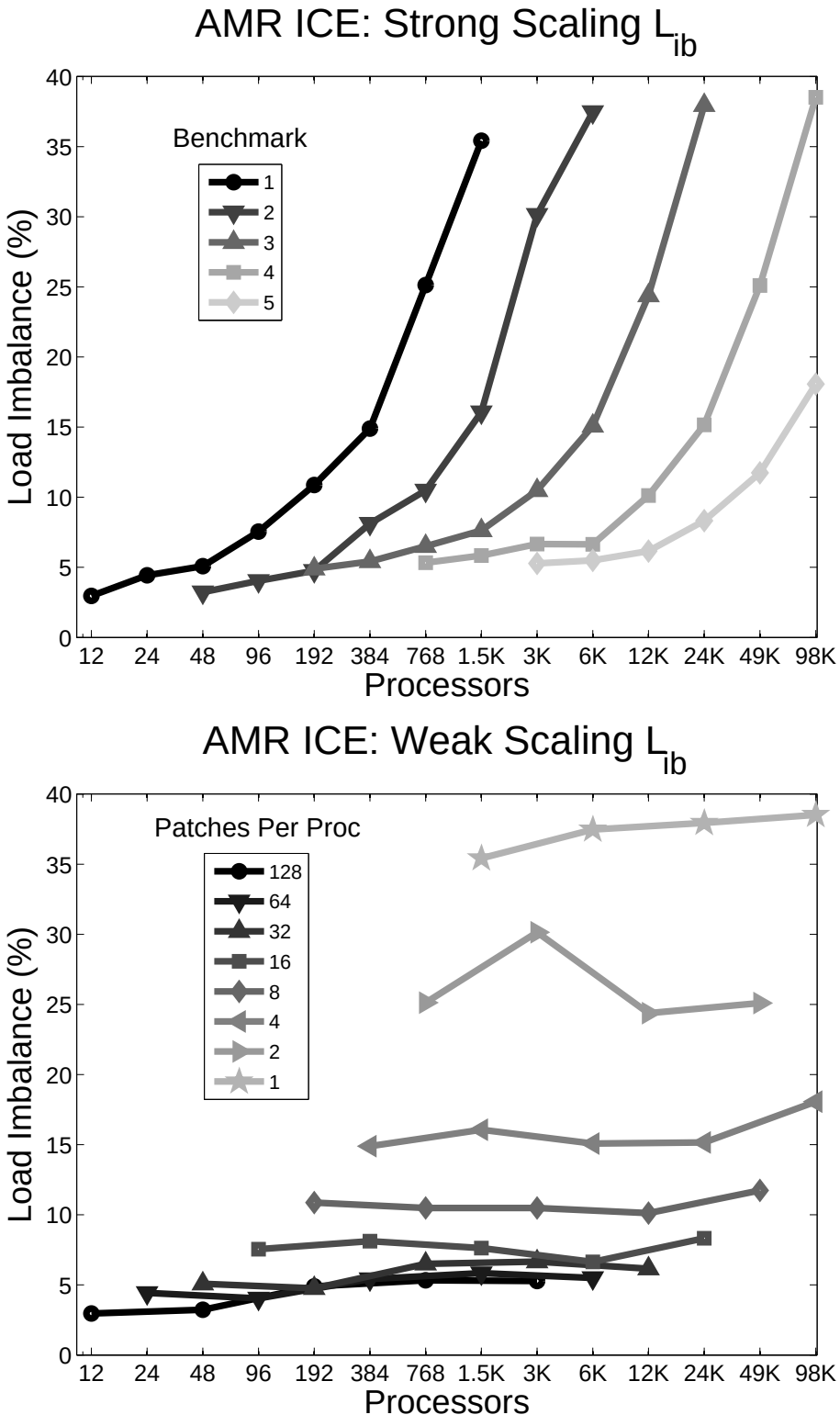



**Figure 6.11**. The load imbalance for the AMR ICE benchmark

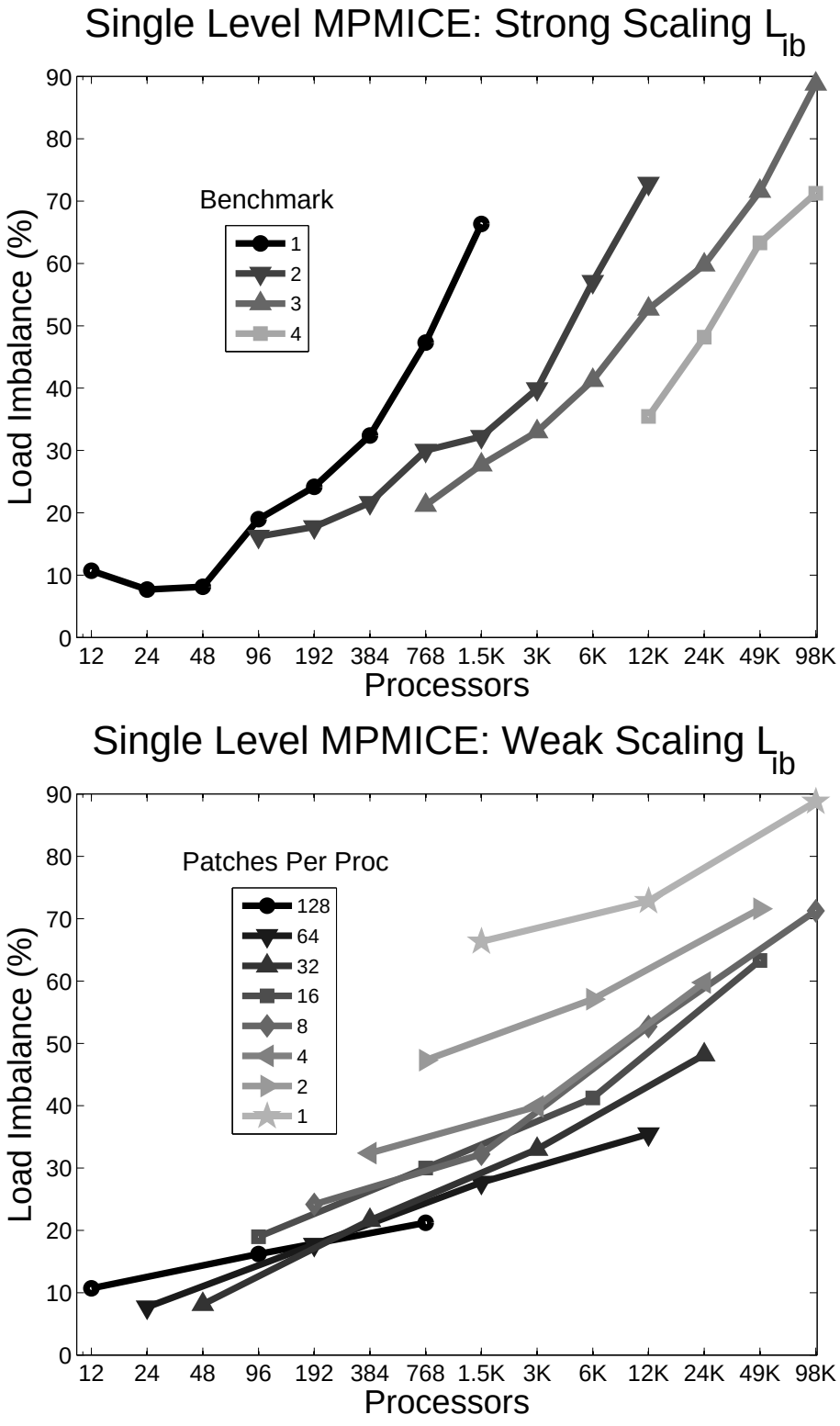



**Figure 6.12**. The load imbalance for the single level MPMICE benchmark

load balance a similar scheme may be required within Uintah. In addition, it appears that increasing the problem size also increases the load imbalance.

The load imbalance for the AMR MPMICE problem is similar to the single level problem with a load imbalance between .2 and .8, as shown in Figure 6.13. This problem has less load imbalance than the single level problem due to the refinement criteria used by MPMICE. This refinement criteria refines only where particles exist and thus, there are no patches on the finest level that do not contain particles.

Finally, Figure 6.14 shows the load imbalance for the array of explosives benchmark. This figure shows the same increase in load imbalance with the problem size as observed in the MPMICE benchmarks. However, the total load imbalance is much lower, ranging from .1 to .6. The reason that the load imbalance is lower is due to the homogeneity of particles throughout the domain. For this benchmark, patches all contain a similar number of particles which significantly simplifies the load balancing process.

## 6.3 Summary and Conclusions

The benchmark results presented here have shown promising results up to 98304 cores on Kraken for both single level and AMR simulations using both ICE and MPMICE. Scalability for AMR codes is more challenging than single level codes due to the added complexity of AMR frameworks. This is particularly true for portions of the algorithm that contain an O(Patches) like load balancing and regridding.

The addition of particles in MPMICE increases the challenge in achieving scalability over ICE. One of the challenges associated with particles is achieving a good load balance. Particles move throughout the domain on every timestep causing the load balance to change often. In addition, load balancing for particle calculations may not be ideal for grid calculations or vice-versa. This is particularly true in Uintah because the unit of work is a patch and all cells and particles within that patch are assigned to the same processor. By load balancing particles separately from cells, the performance within Uintah could be significantly improved. These are challenges that can be addressed in the future which may lead to large increases in performance.

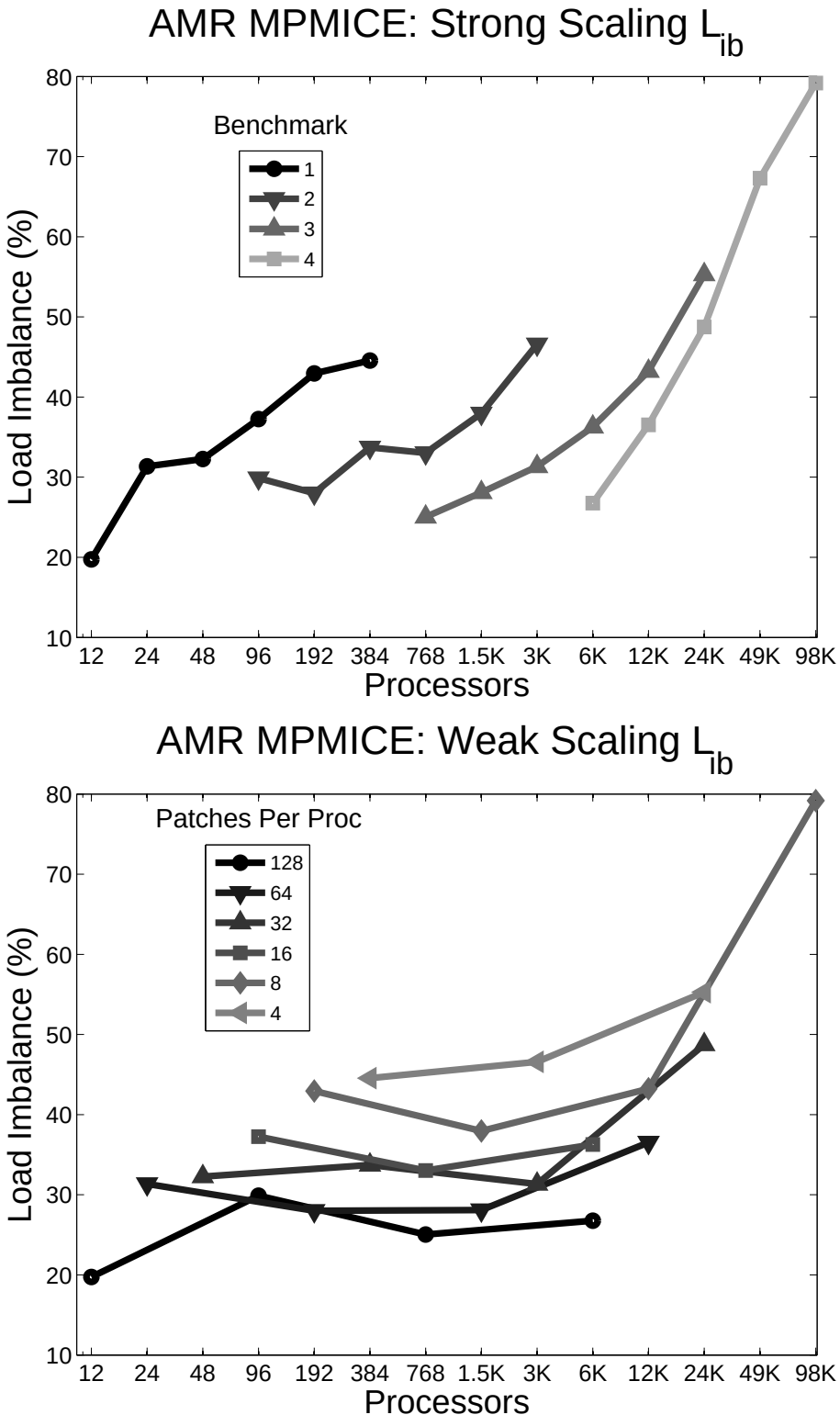



**Figure 6.13**. The load imbalance for the AMR MPMICE benchmark

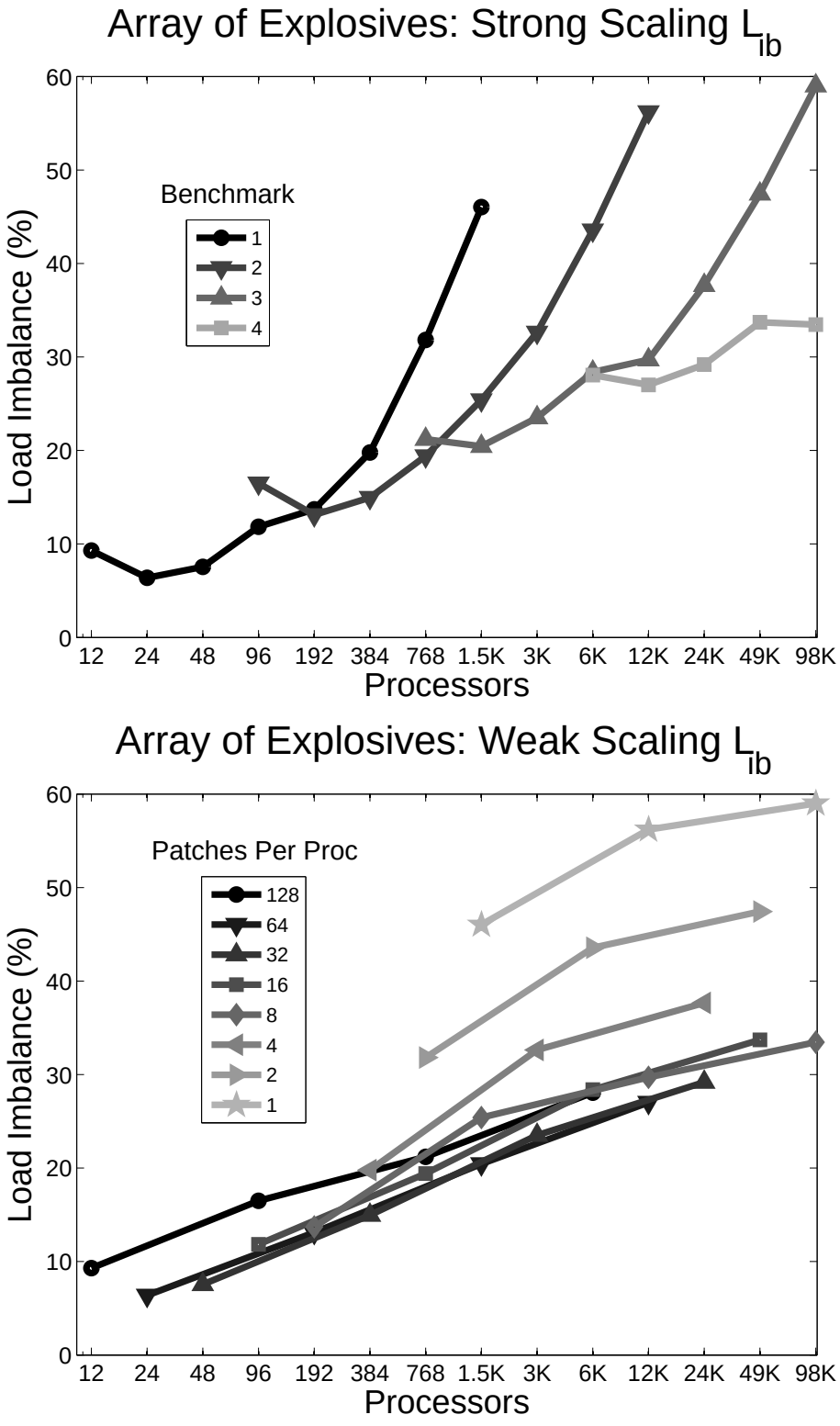



**Figure 6.14**. The load imbalance for the explosive array benchmark

# CHAPTER 7

# SUMMARY AND CONCLUSIONS

AMR simulations are being increasingly used throughout the scientific community. These simulations automatically refine portions of the computational domain with large error, while maintaining a coarse mesh in other portions of the domain. This approach results in solutions that can be as precise as those from much larger fixed mesh simulations while requiring less computation time and memory.

Further improvements can be realized by exploiting large-scale parallelism on large compute clusters to enable the simulation of larger or more refined problems. Parallelism can reduce the time to solution significantly while also allowing the simulation of larger problems. However, as has been shown, achieving ideal scalability at large scales for AMR is challenging. In particular, the time for AMR operations such as regridding and load balancing often limit the overall scalability. The scalability of these operations is becoming increasingly important as simulations move to petascale and eventually exascale machines.

The need for large-scale parallel multiphysics AMR simulations has led to the development of software frameworks such as Uintah. Uintah is a large-scale multiphysics framework which has been described in detail in Chapter 3. Users of this framework write their code as a series of serial tasks on a single hexahedral mesh patch. The framework then compiles these tasks into a distributed task graph which is executed on many patches in parallel. All communication required between parallel processing cores is automatically generated by the framework. The primary benefit of this is that Uintah users do not need to understand the parallel complexities that are required at large scales.

Uintah's a component-based design allows the independent development of components. Components can easily be added or removed from a simulation. This

has allowed the rapid development of large-scale parallel AMR simulations where researchers can focus on work within their research domain while limiting work outside of their domain. This has allowed Uintah to simulate a wide variety of multiphysics fluid-structure interaction problems such as fires and explosions.

Poor scalability of AMR components can limit the scalability of the entire simulation. This dissertation has investigated the parallel performance of widely-used regridding and load balancing algorithms on block structured meshes.

In particular, Chapter 4 analyzed the performance of multiple parallel regridding algorithms including the Global Berger-Rigoutsos (GBR) [9, 44, 122], Local Berger-Rigoutsos (LBR), and Tiled algorithms. This analysis showed that the traditionally-used GBR algorithm has performance problems at large numbers of processors which may prevent scalability. The LBR and Tiled algorithms which were developed in our prior work [75] do not suffer the same performance problems and scale well up to 100K processors and are predicted to scale up to 100M processors.

Chapter 5 focused on load balancing algorithms. In particular, three methods for accurately predicting workloads using forecasting methods were presented. The Kalman filter method [58, 123] had a lower error than both the memory filter and model with least squares approaches. This chapter also presented a scalable algorithm for space-filling curve generation. Space-filling curve approaches to load balancing are commonly used throughout the scientific community because they are able to compute an effective partition quickly. A parallel space-filling curve generation algorithm that was originally presented in [76] was presented here. This algorithm reformulates the space-filling curve generation as a sorting algorithm and then uses parallel sorting algorithms to achieve a high degree of scalability. These algorithms have made possible a fast load balancer that is able to achieve a good load balance. However, more sophisticated load balancing algorithms are required for MPM simulations due to the addition of particles and their associated complexities.

Finally, Chapter 6 investigated the scalability of Uintah as a whole using five different benchmark problems. The scalability of Uintah for all five benchmarks is shown in Figure 6.4. The single level and AMR ICE benchmarks both scaled well to almost 100K cores. The single level and AMR MPMICE benchmarks did not

scale as well. This was primarily due to load imbalance caused by the addition of MPM particles, used to represent solids in Uintah. In particular, the current assignment scheme requires all particles within a patch to be assigned to the same processor. This is problematic when the number of particles within patches varies significantly. In this case, load balancing to equalize the work associated with particles provides a less than optimal load balance with respect to the work associated with cells and vice versa. This is illustrated by the nearly ideal scalability, as shown in Figure 6.4, for the array of explosives benchmark which is similar to the MPMICE benchmarks, with the exception that particles are more evenly distributed across patches than the other case. To achieve greater scalability within Uintah, the load balancing algorithms will need to be improved. In particular, it may be necessary to load balance particles separately from patches as has been done elsewhere in the context of contact detection [48, 80, 106].

It is clear from the results shown here that there is no single simple approach for achieving scalability but instead, there are a series of best practices which will help achieve better scalability. These practices include but are not limited to the following:

- test the scalability of components individually and then as a whole,
- analyze the theoretical scalability,
- assume the number of processors (P) will be very large,
- avoid synchronization where possible:
  - use asynchronous communication,
  - avoid global communication,
  - minimize load imbalance,
- avoid global metadata.

For an algorithm to be scalable, every portion of that algorithm must also be scalable. Components of the algorithm should be tested and analyzed separately and then together as a whole. As we move to more and more processors, algorithms and

data structures that have a term in their complexity that depend on the number of processors or patches will become problematic and should be avoided if at all possible. In particular, these algorithms and data structures will eventually hinder both strong and weak scaling. As the number of processors increases, the cost of synchronization will also increase. Because of this, it is important that our algorithms avoid synchronization where ever possible. This includes using asynchronous communication, minimizing load imbalances, and avoiding global communication.

Finally, global metadata will eventually prevent scalability. This is due to the costs of generating, storing, and communicating this data. For example, simply maintaining a global list of patches, as is currently done in Uintah, will eventually be problematic as the cost for generating the list of patches, which includes the time to communicate that list to all processors, will become large. In addition, the cost to store the list of patches will become too great when hundreds of millions of patches are required. To achieve the highest levels of parallelism on exascale machines, algorithms will need to avoid this type of data and instead operate completely locally.

Despite these challenges that remain for future AMR algorithms on even larger parallel computers, it is important to remember that the scalability achieved with AMR and Uintah was felt to be similarly challenging when this research began. In that light, the research shown here provides a good starting point for AMR algorithms on future petascale machines and beyond.

It is also increasingly likely that future architectures will provide more parallelism on-node than today's current architectures. This will likely be accomplished through larger multicore chips and general purpose graphics processing units (GPGPU). To take advantage of these architectures, many of our current codes will need to be redesigned. In particular, our applications will need to incorporate hybrid MPI-Threading and GPGPU technologies. This will provide many benefits, including the use of shared memory which can reduce memory requirements and communication. In addition, this will reduce the total number of MPI processes which will in turn reduce the complexity of many of our data structures and algorithms. Previously, there has been little motivation to move to such designs due to the limited amount

of on-node parallelism. However, more recently, such designs are becoming increasingly more common as the amount of on-node parallelism has been increasing.

# REFERENCES


[1] Aluru, S., and Sevilgen, F. E. Parallel domain decomposition and load balancing using space-filling curves. In *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing* (Washington, DC, USA, 1997), IEEE Computer Society, p. 230.

[2] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), ACM, pp. 483–485.

[3] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1999), IEEE Computer Society, p. 13.

[4] Atlas, S., Banerjee, S., Cummings, J., Hinker, P., Srikant, M., Reynders, J., and Tholburn, M. POOMA: a high-performance distributed simulation environment for scientific applications. In *SC '95: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (San Diego, CA, USA, 1995), IEEE Press.

[5] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen, Eds. Birkhäuser Boston Inc., Cambridge, MA, USA, 1997, pp. 163–202.

[6] Batcher, K. E. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference* (New York, NY, USA, 1968), ACM, pp. 307–314.

[7] Bennett, J. G., Haberman, K. S., Johnson, J. N., and Asay, B. W. A constitutive model for the non-shock ignition and mechanical response of high explosives. *Journal of the Mechanics and Physics of Solids 46* (1998), 2303–2322.

[8] Berger, M., and Oliger, J. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics 53* (1984), 484–512.

[9] Berger, M. J., and Bokhari, S. H. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers 36*, 5 (1987), 570–580.

[10] Berger, M. J., and Colella, P. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82*, 1 (1989), 64–84.

[11] Berger, M. J., and Rigoutsos, I. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man, and Cybernetics 21* (September 1991), 1278–1286.

[12] Berzins, M. A new metric for dynamic load balancing. *Applied Mathematical Modelling 25*, 2 (2000), 141–151.

[13] Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C., and Peterson, J. Uintah a scalable framework for hazard analysis. In *TeraGrid'10* (New York, NY, USA, 2010), ACM.

[14] Boman, E., Devine, K., Fisk, L. A., Heaphy, R., Hendrickson, B., Vaughan, C., Catalyurek, U., Bozdag, D., Mitchell, W., and Teresco, J. Zoltan 3.0: Parallel partitioning, load-balancing, and data management services; user's guide. Tech. Rep. SAND2007-4748W, Sandia National Laboratories, Albuquerque, NM, 2007.

[15] Boole, G., and Moulton, J. F. *Treatise on the Calculus of Finite Differences, 2nd rev.* Dover, New York, NY, USA, 1960.

[16] Brenner, S. C., and Scott, L. R. *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, New York, NY, USA, 1994.

[17] Bronevetsky, G., and de Supinski, B. R. Complete formal specification of the openmp memory model. *International Journal of Parallel Programming 35*, 4 (2007), 335–392.

[18] Brydon, A., Bardenhagen, S., Miller, E., and Seidler, G. Simulation of the densification of real open–celled foam microstructures. *Journal of the Mechanics and Physics of Solids 53* (2005), 2638–2660.

[19] Burstedde, C., Burtscher, M., Ghattas, O., Stadler, G., Tu, T., and Wilcox, L. C. ALPS: A framework for parallel adaptive PDE solution. *Journal of Physics: Conference Series 180* (2009), 012009.

[20] Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T., and Wilcox, L. C. Extreme-scale amr. In *SC '10: Proceedings of the 2010 ACM/IEEE conference on supercomputing* (Piscataway, NJ, USA, 2010), IEEE Press.

[21] Burstedde, C., Ghattas, O., Gurnis, M., Stadler, G., Tan, E., Tu, T., Wilcox, L. C., and Zhong, S. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–15.

[22] Burstedde, C., Ghattas, O., Stadler, G., Tu, T., and Wilcox, L. C. Towards adaptive mesh pde simulations on petascale computers. In *TeraGrid'08* (New York, NY, USA, 2008), ACM.

[23] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35*, 1 (2009), 38–53.

[24] California, R. D., and Deiterding, R. Detonation structure simulation with amroc. In *International Performance Computing and Communications Conference* (London, UK, 2005), Springer, pp. 916–927.

[25] Campbell, P. M., Devine, K. D., Flaherty, J. E., Gervasio, L. G., and Teresco, J. D. Dynamic octree load balancing using space-filling curves. Tech. Rep. CS-03-01, Williams College Department of Computer Science, Williamstown, MA, USA, 2003.

[26] Cohen, D., Talpey, T., Kanevsky, A., Cummings, U., Krause, M., Recio, R., Crupnicoff, D., Dickman, L., and Grun, P. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. In *17th IEEE Symposium on High Performance Interconnects* (Washington DC, USA, 2009), IEEE Computer Society, pp. 123–130.

[27] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, second ed. MIT Press, Cambridge, MA, USA, 2001.

[28] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing 7*, 2 (1989), 279–301.

[29] Deiterding, R. Construction and application of an AMR algorithm for distributed memory computers. In *Adaptive Mesh Refinement - Theory and Applications*, T. Plewa, T. Linde, and V. G. Weirs, Eds., vol. 41 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, USA, 2005, pp. 361–372.

[30] Devine, K., Boman, E., Heapby, R., Hendrickson, B., and Vaughan, C. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engineering 4* (2002), 90–97.

[31] Devine, K., Hendrickson, B., Boman, E., John, M. S., and Vaughan, C. Design of dynamic load-balancing tools for parallel applications. In *ICS '00: Proceedings of the 14th international conference on Supercomputing* (New York, NY, USA, 2000), ACM, pp. 110–118.

[32] Devine, K. D., Boman, E. G., Heaphy, R. T., Hendrickson, B. A., Teresco, J. D., Faik, J., Flaherty, J. E., and Gervasio, L. G. New challanges in dynamic load balancing. *Applied Numerical Mathematics 52*, 2-3 (2005), 133–152.

[33] Diachin, L. F., Hornung, R., Plassmann, P., and Wissink, A. Parallel adaptive mesh refinement. In *Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds. SIAM, Philadelphia, PA, USA, 2006, ch. 8.

[34] Dinan, J., Krishnamoorthy, S., Brian, L., Nieplocha, J., and Sadayappan, P. Scioto: A framework for global-view task parallelism. In *ICPP '08. 37th International Conference on Parallel Processing* (Washington DC, USA, 2008), IEEE Computer Society, pp. 586–593.

[35] Falgout, R., Jones, J., and Yang, U. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, Bruaset and A. Tveito, Eds., vol. 51. Springer, 2006, pp. 267–294.

[36] Feo, J., Cann, D., and Oldehoeft, R. A report on the sisal language project. *Journal of Parallel and Distributed Computing 10* (1990), 349–366.

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

[38] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241 of *Proceedings 11th European PVM/MPI Users' Group Meeting*. Springer Verlag, Budapest, Hungary, September 2004, pp. 97–104.

[39] Gallagher, R. H. *Finite Element Analysis: Fundamentals*. Springer-Verlag, Englewood Cliffs, New Jersey, 1975.

[40] Germain, J. D. d. S., McCorquodale, J., Parker, S. G., and Johnson, C. R. Uintah: A massively parallel problem solving environment. In *HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 33–42.

[41] Goodale, T., Allen, G., Lanfermann, G., Mass, J., Seidel, E., and Shalf, J. The cactus framework and toolkit: Design and applications. In *In Vector and Parallel Processing - VECPAR 2002, 5th International Conference* (2003), vol. 2565, Springer, pp. 15–36.

[42] Gropp, W., and Lusk, E. *User's Guide for mpich, a Portable Implementation of MPI Version 1.2.1*. Argonne National Lab, Argonne, IL, 1996.

[43] Guilkey, J., Harman, T., and Banerjee, B. An eulerian-lagrangian approach for simulating explosions of energetic devices. *Computers and Structures 85* (2007), 660–674.

[44] Gunney, B. T. N., Wissink, A. M., and Hysom, D. A. Parallel clustering algorithms for structured amr. *Journal of Parallel and Distributed Computing 66*, 11 (2006), 1419–1430.

[45] Gustafson, J. L. Reevaluating amdahl's law. *Communications of the ACM 31* (1988), 532–533.

[46] Harlow, F., and Amsden, A. Numerical calculation of almost incompressible flow. *Journal of Computational Physics 3* (1968), 80–93.

[47] Henderson, T., McMurtry, P., Smith, P., Voth, G., Wight, C., and Pershing, D. Simulating accidental fires and explosions. *Computing in Science and Engineering 2* (1994), 64–76.

[48] Hendrickson, B., Plimpton, S., Attaway, S., Vaughan, C., and Gardner, D. A new parallel algorithm for contact detection in finite element methods. In *Proceedings of the High Performance Computing '96* (Philadelphia, PA, USA, 1996), SIAM.

[49] Hornung, R., and Kohn, S. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation Practice and Experience 14* (2002), 347–368.

[50] Horton, G. A multi-level diffusion method for dynamic load balancing. *Parallel Computing 19*, 2 (1993), 209–218.

[51] Hu, Y. F., and Blake, R. J. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing 25* (1999), 417–444.

[52] Hu, Y. F., Blake, R. J., and Emerson, D. R. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience 10*, 6 (1998), 467–483.

[53] Huang, W., Santhanaraman, G., Jin, H.-W., Gao, Q., and D. K. Panda, D. K. x. Design of high performance mvapich2: Mpi2 over infiniband. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 43–48.

[54] Hyman, J. M., Knapp, R. J., and Scovel, J. C. High order finite volume approximations of differential operators on nonuniform grids. *Physica D 60*, 1-4 (1992), 112–138.

[55] Jeon, M., and Kim, D. Parallelizing merge sort onto distributed memory parallel computers. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing* (London, UK, 2002), Springer-Verlag, pp. 25–34.

[56] Jeon, M., and Kim, D. Parallel merge sort with load balancing. *International Journal of Parallel Programming 31*, 1 (2003), 21–33.

[57] Kale, L. V., and Krishnan, S. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, Cambridge, MA, USA, 1996, pp. 175–213.

[58] Kalman, R. E. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering 82*, Series D (1960), 35–45.

[59] Karagiorgos, G., Missirlis, N., and Tzaferis, F. Fast diffusion load balancing algorithms on torus graphs. In *Euro-Par 2006 Parallel Processing*, W. Nagel, W. Walter, and W. Lehner, Eds., vol. 4128 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 222–231.

[60] Karypis, G., and Kumar, V. *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. University of Minnesota, Minneapolis, MN, USA, 1995.

[61] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing 20*, 1 (1998), 359–392.

[62] Kashiwa, B. A multifield model and method for fluid-structure interaction dynamics. Tech. Rep. LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, NM, USA, 2001.

[63] Kashiwa, B., and Gaffney, E. Design basis for cfdlib. Tech. Rep. LA-UR-03-1295, Los Alamos National Laboratory, Los Alamos, NM, USA, 2003.

[64] Kashiwa, B., Lewis, M., and Wilson, T. Fluid-structure interaction modeling. Tech. Rep. LA-13111-PR, Los Alamos National Laboratory, Los Alamos, NM, USA, 1996.

[65] Kashiwa, B., and Rauenzahn, R. A cell-centered ICE method for multiphase flow simulations. Tech. Rep. LA-UR-93-3922, Los Alamos National Laboratory, Los Alamos, NM, USA, 1994.

[66] Kashiwa, B., and Rauenzahn, R. A multimaterial formalism. Tech. Rep. LA-UR-94-771, Los Alamos National Laboratory, Los Alamos, NM, USA, 1994.

[67] Klosowski, J. T., Held, M., Mitchell, J. S. B., Sowizral, H., and Zikan, K. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4* (1998), 21–36.

[68] Knobe, K. Ease of use with concurrent collections (cnc). In *HotPar'09: Proceedings of the First USENIX Conference on Hot Topics in Parallelism* (2009), USENIX Association, p. 17.

[69] Knuth, D. E. *The Art of Computer Programming, Sorting and Searching*, second ed., vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[70] Lee, S.-J., Jeon, M., Sohn, A., and Kim, D. Partitioned parallel radix sort. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing* (London, UK, 2000), Springer-Verlag, pp. 160–171.

[71] Lewis, B., and Berg, D. J. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[72] Liu, J., Vishnu, A., and Panda, D. K. Building multirail infiniband clusters: Mpi-level design and performance evaluation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Press, p. 33.

[73] Ltstedt, P., Sderberg, S., Ramage, A., and Hemmingsson-Frndn, L. Implicit solution of hyperbolic equations with space-time adaptivity. *BIT Numerical Mathematics 42* (2002), 134–158. 10.1023/A:1021978304268.

[74] Luitjens, J., and Berzins, M. Improving the performance of uintah: A large-scale adaptive meshing computational framework. In *IPDPS 2010 the 24th IEEE International Parallel and Distributed Processing Symposium* (Washington DC, USA, 2010), IEEE Computer Society.

[75] Luitjens, J., and Berzins, M. Parallel regridding algorithms for block-structured adaptive mesh refinement. *Submitted to Concurrency and Computation : Practice and Experience* (2010).

[76] Luitjens, J., Berzins, M., and Henderson, T. Parallel space-filling curve generation through sorting: Research articles. *Concurrency and Computation : Practice and Experience 19*, 10 (2007), 1387–1402.

[77] Luitjens, J., Guilkey, J., Harman, T., Worthen, B., and Parker, S. Adaptive computations in the uintah framework. In *Advanced Computational Infastructures for Parallel/Distributed Adaptive Applications*, M. Parashar and X. L, Eds. John Wiley and Sons, Inc, Chichester, UK, 2009, pp. 171–199.

[78] Luitjens, J., Worthen, B., Berzins, M., and Henderson, T. Scalable parallel amr for the uintah multiphysics code. In *Petascale Computing Algorithms and Applications*, D. A. Bader, Ed. Chapman and Hall/CRC, New York, NY, USA, 2007, pp. 67–82.

[79] MacNeice, P., Olson, K. M., Mobarry, C., deFainchtein, R., and Packer, C. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications 126* (2000), 330–354.

[80] Maerten, B., Roose, D., Basermann, A., Fingberg, J., and Lonsdale, G. Drama: A library for parallel dynamic load balancing of finite element applications. In *Euro-Par99 Parallel Processing*, P. Amestoy, P. Berger,

M. Dayd, D. Ruiz, I. Duff, V. Frayss, and L. Giraud, Eds., vol. 1685 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999, pp. 313–316.

[81] Martín, I., and Tirado, F. Relationships between efficiency and execution time of full multigrid methods on parallel computers. *IEEE Trans. Parallel Distrib. Syst. 8*, 6 (1997), 562–573.

[82] Meng, Q., Luitjens, J., and Berzins, M. A comparison of load balancing algorithms for amr in uintah. Tech. Rep. UUSCI-2008-006, University of Utah, Salt Lake City, UT, USA, 2008.

[83] Meng, Q., Luitjens, J., and Berzins, M. Dynamic task scheduling for the uintah framework. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10)* (Washington DC, USA, 2010), IEEE Computer Society.

[84] Montgomery, D. C., Johnson, L. A., and Gardiner, J. S. *Forecasting and Time Series Analysis*, 2nd ed. McGraw-Hill, New York, NY, USA, 1990.

[85] Norman, M. L., Bryan, G. L., Harkness, R., Bordner, J., Reynolds, D., O'Shea, B., and Wagner, R. Simulating cosmological evolution with enzo. In *Petascale Computing Algorithms and Applications*, D. A. Bader, Ed. Chapman and Hall/CRC, New York, NY, USA, 2007.

[86] O'shea, B., Bryan, G., Bordner, J., Norman, M., Abel, T., Harkness, R., and Kritsuk, A. Introducing enzo, an amr cosmology application. In *Adaptive Mesh Refinement - Theory and Applications*, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, T. Plewa, T. Linde, and V. Gregory Weirs, Eds., vol. 41 of *Lecture Notes in Computational Science and Enginee*. Springer Berlin Heidelberg, 2005, pp. 341–349.

[87] Ott, C. D., Schnetter, E., Burrows, A., Livne, E., O'Connor, E., and Löffler, F. Computational models of stellar collapse and core-collapse supernovae. *Journal of Physics: Conference Series 180* (2009), 012022.

[88] Ou, C.-W., wei Ou, C., Ranka, S., and Ranka, S. Parallel remapping algorithms for adaptive problems. In *Proc. of the Symp. on the Frontiers of Massively Parallel Computation* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 367–374.

[89] Parker, S. G. A component-based architecture for parallel multi-physics pde simulation. *Future Generation Computer Systems 22*, 1 (2006), 204–216.

[90] Parker, S. G., Guilkey, J., and Harman, T. A component-based parallel infrastructure for the simulation of fluid structure interaction. *Engineering with Computers 22*, 3-4 (2006), 277–292.

[91] Pilkington, J. R., and Baden, S. B. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems 7* (1995), 288–300.

[92] Pothen, A., Simon, H. D., and Liou, K.-P. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications 11*, 3 (July 1990), 430–452.

[93] Sagan, H. *Space-Filling Curves*. Springer, 1994.

[94] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[95] Shee, M., Bhavsar, S., and Parashar, M. Characterizing the performance of dynamic distribution and load-balancing techniques for adaptive grid hierarchies. In *IASTED: International Conference on Parallel and Distributed Computing and Systems* (Calagary, AB, 1999), IASTED.

[96] Simon, H. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering 2*, 2-3 (1991), 135–148.

[97] Simon, H. D., Sohn, A., and Biswas, R. Harp: A dynamic spectral partitioner. *Journal of Parallel and Distributed Computing 50*, 1/2 (1998), 83–103.

[98] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[99] Sohn, A., and Kodama, Y. Load balanced parallel radix sort. In *ICS '98: Proceedings of the 12th international conference on Supercomputing* (New York, NY, USA, 1998), ACM, pp. 305–312.

[100] Solomonik, E., and Kale, L. V. Highly scalable parallel sorting. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2010), IEEE Computer Society.

[101] Souers, P., Anderson, S., Mercer, J., McGuire, E., and Vitello, P. Jwl++: A simple reactive flow code package for detonation. *Propellants, Explosives, Pyrotechnics 25* (2000), 54–58.

[102] Souers, P., Garza, R., and Vitello, P. Ignition & growth and jwll++ detonation models in coarse zones. *Propellants, Explosives, Pyrotechnics 27* (2002), 62–71.

[103] Spiegel, M. *Calculus of Finite Differences and Differential Equations*. McGraw-Hill, New York, NY, USA, 1971.

[104] Spinti, J., Thornock, J., Eddings, E., Smith, P., and Sarofim, A. Heat transfer to objects in pool fires. In *Transport Phenomena in Fires*, M. Faghri and S. D. i. H. T. B. Senden, Eds., vol. 20. WIT Press, Southampton, U.K., 2008.

[105] Steensland, J., Söderberg, S., and Thuné, M. A comparison of partitioning schemes for blockwise parallel samr algorithms. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia* (London, UK, 2001), Springer-Verlag, pp. 160–169.

[106] Steve, S. P., Attaway, S., Hendrickson, B., Swegle, J., Vaughan, C., and Gardner, D. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. *Journal of Parallel and Distributed Computing 50* (1996), 104–122.

[107] Stewart, D. Investigations on deflagration to detonation transition in porous energetic materials. Tech. Rep. DOE-LANL-6730M0014-9, Los Alamos National Laboratory, Los Alamos, NM, USA, July 1999.

[108] Straalen, B. V., Shalf, J., Ligocki, T., Keen, N., and Yang, W.-S. Scalability challenges for massively parallel amr applications. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–12.

[109] Sulsky, D., Chen, Z., and Schreyer, H. A particle method for history dependent materials. *Computer Methods in Applied Mechanics 118* (1994), 179–196.

[110] Tridgell, A. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National University, Canberra 0200 ACT, Australia, February 1999.

[111] Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shendey, S., Oldehoeft, R., and Smith, S. SMARTS: Exploiting temporal locality and parallelism through vertical execution. In *ICS 99: Proceedings of International Conference on Supercomputing* (New York, NY, USA, 1999), ACM, pp. 302–310.

[112] VanderHeyden, W., and Kashiwa, B. Compatible fluxes for van leer advection. *Journal of Computational Physics 146* (1998), 1–28.

[113] Versteeg, H. K., and Malalasekera, W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method.* Addison-Wesley, Massachusetts, 1995.

[114] Walshaw, C., Cross, M., and Everett, M. G. Dynamic load-balancing for parallel adaptive unstructured meshes. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, PA, USA, 1997), SIAM.

[115] Walshaw, C., Cross, M., Everett, M. G., and Johnson, S. JOSTLE: Partitioning of unstructured meshes for massively parallel machines. In *Parallel Computational Fluid Dynamics: New Algorithms and Applications* (Amsterdam, The Netherlands, The Netherlands, 1994), Elsevier.

[116] Walshaw, C., Everett, M. G., and Cross, M. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing 47*, 2 (1997), 102–108.

[117] Ward, M., Son, S., and Brewster, M. Steady deflagration of hmx with simple kinetics: A gas phase chain reaction model. *Combustion and Flame 114* (1998), 556–568.

[118] Warren, M. S., and Salmon, J. K. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1993), ACM, pp. 12–21.

[119] Weaver, L., and Lynes, A. Sorting Integers on the AP1000. *The Computing Research Repository cs.DC/0004013* (2000).

[120] Wight, C., and Eddings, E. Science-based simulation tools for hazard assessment and mitigation. *Advancements in Energetic Materials and Chemical Propulsion 114* (2008), 921–937.

[121] Wissink, A. M., Hornung, R. D., Kohn, S. R., Smith, S. S., and Elliott, N. Large scale parallel structured amr calculations using the samrai framework. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2001), ACM, p. 6.

[122] Wissink, A. M., Hysom, D., and Hornung, R. D. Enhancing scalability of parallel structured amr calculations. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing* (New York, NY, USA, 2003), ACM, pp. 336–347.

[123] Zarchan, P., and Musoff, H. *Fundamentals of Kalman Filtering: A Practical Approach (Progress in Astronautics and Aeronautics)*. AIAA (American Institute of Aeronautics & Ast), December 2000.