Chapter 4

Scalable Parallel AMR for the Uintah Multiphysics Code

Justin Luitjens

SCI Institute, University of Utah

Bryan Worthen

SCI Institute, University of Utah

Martin Berzins

SCI Institute, University of Utah

Thomas C. Henderson

School of Computing, University of Utah

4.1 Introduction

Large-scale multiphysics computational simulations often provide insight into complex problems that both complements experiments and helps define future physical and computational experiments [13]. A good example of a production-strength code is Uintah [13, 12]. The code is designed to solve reacting fluid-structure problems involving large deformations and fragmentation. The underlying methods inside Uintah are a combination of standard fluid-flow methods and material point (particle) methods. In the case of codes like Uintah which solve large systems of partial differential equations on a mesh, refining the mesh increases the accuracy of the simulation. Unfortunately refining a mesh by a factor of two increases the work by a factor of 2^d, where d is the dimensionality of the problem. This rapid increase in computational effort severely limits the accuracy attainable on a particular

0-8493-0052-5/00/$0.00+$.50

parallel machine.

Adaptive mesh refinement (AMR) attempts to reduce the work a simulation must perform by concentrating mesh refinement on areas that have high error [3, 2] and coarsening the mesh in areas in which the error is small. One standard parallel AMR method divides the domain into rectangular regions called patches. Typically each patch contains Cartesian mesh cells of the same size. Each processor runs the simulation on a subset of the patches while communicating with neighboring patches. By using a component-based framework [13, 12, 8, 10] simulation scientists can solve their problems without having to focus on the intricacies of parallelism. For example, inside Uintah, parallelism is completely hidden from the simulation scientists [13], by means of a sophisticated task compilation mechanism.

In order to achieve good parallel performance with AMR, the component-based frameworks must be enhanced. The framework must drive refinement by locating regions that require refinement and creating patches on those regions. We refer to the process of generating patches as regridding. The patches must be load balanced onto processors in such a way that each processor is performing approximately the same amount of work while minimizing the overall communication. This can be achieved in the case of AMR calculations through the use of space-filling curves. Load balancers based on space-filling curves can create partitions quickly that keep communication between processors low while also keeping the work imbalance low [1, 16, 18, 6]. Recent work has shown that space-filling curves can be generated quickly and scalably in parallel [11]. In addition, the Uintah framework must also schedule the communication between the patches. As the simulation runs, regions needing refinement change, and as the computational mesh changes, load balancing and scheduling need to be recomputed. The regridding can occur often throughout the computation requiring each of these processes to be quick and to scale well in parallel. Poor scalability in any of these components can significantly impact overall performance [19, 20]. The Berger-Rigoutsos algorithm, which is commonly used for regridding [19, 4], creates patch sets with low numbers of patches that cover all regions needing refinement. In addition, the Berger-Rigoutsos algorithm can be parallelized using a task-based parallelization scheme [19, 20].

Present state-of-the-art AMR calculations have been shown by Wissink and colleagues [19, 20] to scale to many thousands of processors in the sense of the distribution of computational work. The outstanding issues with regard to AMR are surveyed by Freitag Daichin et al. [7], and in great depth by Steensland and colleagues [17]. In this paper we will show how scalability needs to reevaluated in terms of accuracy and consider how different components of Uintah may be made scalable.

4.2 Adaptive Mesh Refinement

Traditionally in parallel computing, scalability is concerned with maintaining the relative efficiency as the problem size grows. What is often missing is any attempt to ensure that the computational effort is, in comparative terms, well spent. The following argument was made by Keyes at the Dagstuhl workshop where his work was presented:

Consider a fixed mesh p.d.e. calculation in three space dimensions with mesh sizes δx, δy, δz defining a regularly refined box mesh M_1 and a time step δt. Then it is possible to write the error obtained by using that mesh in some suitable norm, defined here by $||E(M_1)||$ as

$$||E(M_1)|| = C_x(\delta x)^k + C_y(\delta y)^k + C_z(\delta z)^k + C_t(\delta t)^q \tag{4.1}$$

In many cases of practical interest p and q may be less than three. The computational time, T_{cpu}, associated with this mesh is (at best) a linear function of the number of unknowns and the number of time steps and hence may be written as:

$$T_{cpu}(M_1) = C_{cpu}\frac{1}{\delta x\delta y\delta z\delta t} \tag{4.2}$$

where C_{cpu} is an appropriate constant.

In order to simplify the discussion from hereon the time dependent nature of the calculation is neglected. In the case when a new mesh, M_2 is defined by uniformly refining the original mesh by a factor of 2 in each dimension

$$\delta x = \frac{\delta x}{2} \tag{4.3}$$

and similarly for δy and δz, then the computational work increases by

$$T_{cpu}(M_2) = 8T_{cpu}(M_1) \tag{4.4}$$

while the error changes by

$$||E(M_2)|| = \frac{1}{2^k}||E(M_1)|| \tag{4.5}$$

Hence for first and second order methods $k = 1, 2$ the increase in work is greater than the decrease in error. The situation becomes worse if the work has a greater than linear dependency on the number of unknowns. Observations have spurred much work on high-order methods; see for example comparisons in the work of Ray and Steensland and others [14].

We can now define a mesh accuracy ratio, denoted by M_{ar} by:

$$M_{ar} = \frac{||E(M_2)||}{||E(M_1)||} \tag{4.6}$$

Now also assuming that the calculation on M_1 uses P_1 parallel processors and the calculation on mesh M_2 uses P_2 parallel processors, and that both solutions are delivered in the same time, then the Mesh parallel efficiency as denoted by M_{pe}, may be defined as

$$M_{pe} = M_{ar}\,\frac{P_2}{P_1} \tag{4.7}$$

For the simple fixed mesh refinement steady state case above then,

$$M_{pe} = \frac{8}{2^k} \tag{4.8}$$

It is worth remarking that if the calculation (inevitably) does not scale perfectly as far as computational work, or the different numbers of processors take different amounts of time, then the above expression may be modified to take this into account:

$$M_{pe} = M_{ar}\,\frac{P_2 T_2}{P_1 T_1} \tag{4.9}$$

where T_1 and T_2 are the compute times using P_1 and P_2 processors, respectively.

As a simple example consider the case of the solution of Laplace equation on the unit cube using a Jacobi method. Suppose that an evenly spaced $NxNxN$ mesh is decomposed into p sub-cubes each of size $\frac{N^3}{p}$ on p processors. A standard second order method gives an accuracy of $\frac{C_2}{N^2}$ while use of a fourth order method [9], gives an accuracy of $\frac{C_4}{N^4}$ and where C_2 and C_4 are both known constants. The cost of the fourth order method is twice as many operations per point with a communications message length that is twice as long as in the second order case, and with perhaps r_I as many iterations, thus with an overall cost of $2r_I$ of that of the second-order method.

For the same accuracy, and $M_{ar} = 1$, it follows that

$$N_2 = \sqrt{\frac{C_4}{C_2}}N_4^2 \tag{4.10}$$

where N_2 is the second order mesh size and N_4 is the fourth order mesh size. In order to achieve $M_{pe} = 1$ with each run having the same execution time, as estimated by a simple cost model based on $\frac{N^3}{p}$, the number of processors used by the second order mesh, P_2, must be related to the number used by the fourth order mesh, P_4 by

$$\frac{(N_2)^3}{P_2} \approx 2r_I\frac{(N_4)^3}{P_4} \tag{4.11}$$

Hence the lower order method needs approximately the square of the number of processors of the higher order method:

$$P_2 \approx \frac{1}{2r_I}\left(\frac{C_4}{C_2}\right)^{\frac{3}{2}}(P_4)^2 \tag{4.12}$$

The mesh parallel efficiency is also far from one:

$$M_{pe} = \frac{C_4}{C_2} \frac{2r_I}{N_2 N_4} \tag{4.13}$$

Unless the fourth order method has a significantly greater number of iterations per point than the second order method, the implications of this estimate for a petaflop machine with possibly $O(10^6)$ processors are clear.

In considering mesh refinement it is possible to start with a simple one-dimensional case. Consider a uniform mesh of N_c δx_f cells. Next consider a nonuniform mesh which starts with a cell of width δx_f at its left side and then doubles with each subsequent cell. Suppose that there are q of these cells, then:

$$\delta x_f(1 + 2 + 4 + ...2^q) = \delta x_f N_c \tag{4.14}$$

Hence after summing the left side of this

$$\delta x_f(2^{q+1} - 1) = \delta x_f N_c \tag{4.15}$$

or

$$q = log_2(N_c + 1) - 1 \tag{4.16}$$

It is worth remarking that it is possible to modify the above expression to account for a mesh that increases more gradually. For example, an adaptive mesh in which two cells have the same size before the size changes gives:

$$q = log_2(N_c + 2) - 2 \tag{4.17}$$

With this simple logarithmic model of mesh changes in mind consider three cases in which mesh refinement is applied to a three-dimensional box around either one vertex, one edge or one bounding plane. Suppose that the box is discretized by using N_c^3 regular cells. While there are many refinement possibilities, typically nonuniform refinement takes place on a lower dimensional manifold than the original mesh. For example:

Refinement at a Point. In this case the mesh can increase in all three dimensions as in the simple one-dimensional example and so q^3 cells are used. In this case we assume that it is possible to increase the accuracy by only refining m cells close to the point. Hence the new mesh has $q^3 + m$ cells and

$$M_{pe} = \frac{q^3 + m}{q^3} \frac{1}{2^k} \tag{4.18}$$

Refinement on a Line. In this case the mesh can increase in only two dimensions as in the simple one-dimensional example and so $N_c q^2$ cells are used. In this case we assume that it is possible to increase the accuracy by only refining m cells close to the line. Hence the new mesh has $N_c q^2 + mN_c$ cells and

$$M_{pe} = \frac{N_c q^2 + mN_c}{N_c q^2} \frac{1}{2^k} \tag{4.19}$$

Refinement on a Plane. In this case the mesh can increase in only one dimension as in the simple one-dimensional example and so $N_c^2 q$ cells are used. In this case we assume that it is possible to increase the accuracy by only refining $N_c^2 m$ cells close to the plane. Hence the new mesh has $q^3 + m$ cells and

$$M_{pe} = \frac{N_c^2 q + m N_c^2}{N_c^2 q} \; \frac{1}{2^k} \tag{4.20}$$

In all three cases a mesh efficiency close to 1 requires:

$$\frac{m}{q^j} \leq 2^k \tag{4.21}$$

where $j = 1, 2$ or 3 depending on the case above. Even in the case $k = 1$ and $j = 1$ (refinement of a plane), this simple analysis shows that as long as mesh refinement needs to be used on less than 50% of the existing cells in order to reduce the accuracy by half then the increase in accuracy is matched by the increase in work.

These studies show that if either high order methods or adaptive mesh approaches are used, then computational resources are used wisely with respect to the accuracy achieved. This has already been recognized for high order methods [14], but is not so widely understood for adaptive methods, such a those discussed below.

4.3 Uintah Framework Background

Uintah is a framework consisting of components such as a simulation component, the load balancer, the scheduler, and the regridder. The regridder component will be described in detail below.

4.3.1 Simulation components

The Uintah simulation components implement different algorithms and operate together or independently [13, 12, 8]. Uintah's main simulation components are based on the implicit compressible eulerian algorithm (ICE), material point method (MPM), and Arches [13]. The simulation component will create tasks, and pass them to the scheduler, which is described below, instructing it as to what data relative to a patch that task will need. The scheduler will then execute the simulation component's tasks, one patch at a time, thus creating a parallel environment, and enabling the applications scientist to concentrate on the science issues.

4.3.2 Load balancer

The load balancer is responsible for determining which patches will be owned by each processor. There are two main load balancers in Uintah: the simple load balancer, and the dynamic load balancer. The simple load balancer simply determines the average number of patches per processor, and assigns that number of consecutive patches to each processor. This suffices for simple static problems that are easily load balanced. The dynamic load balancer attempts to achieve balance for more complicated problems. First, it orders the patches according to a space-filling curve; and second, it computes a weight for each patch, based on its size and the number of particles. The curve and the weights are then used to distribute the patches according to the average work per processor. The patches are assigned in the order of the space-filling curve placing patches that are close together in space on the same processor. This reduces the overall amount of communication that must occur. The patches are assigned so that each processor has approximately the same amount of work.

The Hilbert [15] space-filling curve is used in Uintah because it may be generated quickly, [1, 16, 18, 6, 11], in parallel [11], and provides good locality. The curve is formed over patches by using the centroid of the patches to represent them. The space-filling curve provides a linear ordering of the patches such that patches that are close together in the linear ordering are also closer together in the higher dimensional space. The curve is then broken into curve segments based on the weights of the patches. This provides approximately equally sized partitions that are clustered locally. Figure 4.1 shows an adaptive mesh partitioned using the Hilbert curve.

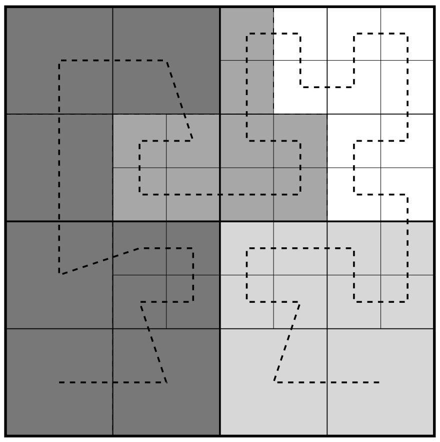

FIGURE 4.1: An example of how a space-filling curve is used in partitioning a mesh.

The space-filling curve can be generated quickly in parallel. Figure 4.2 shows how the generation performance varies for large numbers of patches on up to 2048 processors. The load balancer is also responsible to create a

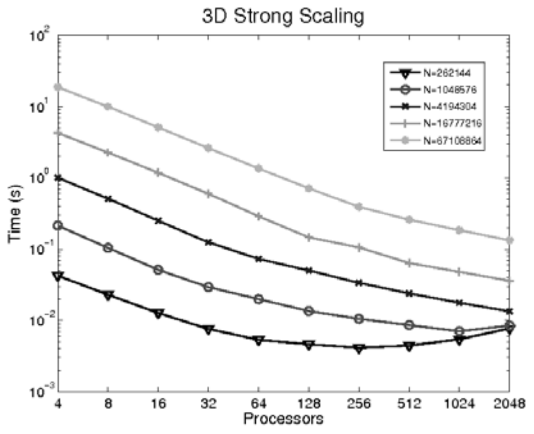

FIGURE 4.2: Scalability of the generating space-filling curves.

processor's neighborhood, which in essence is every patch on that processor along with every patch on any other processor that will communicate with a patch on that processor.

4.3.3 Scheduler

The scheduler is responsible to order the simulation component's tasks in a parallel fashion, and to determine the corresponding MPI communication patterns. Its work is divided into two phases: compilation and execution. The compilation phase determines what data are required by each patch from its neighboring patches for each task. This is determined from the basic communication requirements that are provided by the simulation component. It accomplishes this by determining which patches are neighbors, and then computes the range of data the neighboring patch will need to provide [13, 12, 8]. On each processor, this algorithm is executed for each patch in the processor's neighborhood, which is on the order of the number of patches per processor, thus giving a complexity of nearly $O(\frac{N}{P} \log \frac{N}{P}^{2})$, where N is the number of patches and P is the number of processors. This phase is executed only once for problems without AMR or dynamic load balancing, hence for fixed meshes its performance is not an issue. During the execution phase, each task will receive any data it requires from a neighboring patch's processor, run the simulation code for the task, and then send any data it computes to requiring tasks on other processors.

4.4 Regridder

The regridder's duty is to create a finer level — a level is a set of patches with the same cell spacing — based on the refinement flags, which are created by the simulation component. It determines the region of space on which to create a finer level, and then divides that space into patches with a finer resolution. It is important that the regridder considers what type of patches to produce. Producing patches that are too large can result in large load imbalances and prevent scalability. However, producing patches that are too small can cause significant overhead in other components.

The Uintah framework has constraints which require the regridder to produce certain types of patch sets. The first constraint is a minimum patch size. Each edge of a patch must be at least 4 cells in length. In addition, patch boundaries can either be coarse or fine but not a combination of the two. That is, every patch boundary must be completely filled with neighboring patches or have no neighbors at all. For the rest of this chapter we refer to the location on a boundary that moves from coarse to fine as a mixed boundary. Figure 4.3 shows two patch sets that cover the same area. The left patch set is invalid because it contains a mixed boundary; the second patch set does not contain a mixed boundary and is valid.

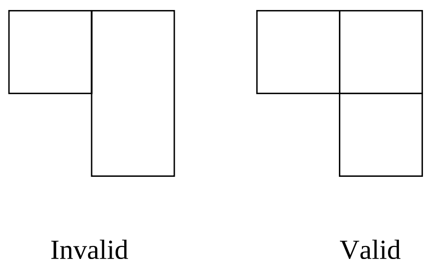

FIGURE 4.3: Valid and invalid patch sets within Uintah. The left patch set is invalid because it contains a mixed boundary.

Regridding is commonly accomplished through the Berger-Rigoutsos algorithm [19, 4]. The algorithm starts by placing a bounding box around all of the refinement flags. A histogram of the refinement flags is then created in each dimension. This histogram is then used to determine a good location to split the bounding box in half. The process then recursively repeats on both halves of the bounding box. By having different processors evaluate different sections of the recursion this process can be made parallel. A full description of the parallel algorithm can be found in [19, 20].

The Berger-Rigoutsos algorithm produces patch sets with low numbers of patches. However, the constraints within Uintah prevent the use of the Berger-Rigoutsos algorithm. Berger-Rigoutsos produces patch sets that contain mixed boundaries. Mixed boundaries can be eliminated by splitting patches at the point where the boundary changes. However, splitting patches produced by Berger-Rigoutsos can lead to patches that violate the minimum patch size requirement. Due to the constraints within Uintah we initially used a tiled regridder. A grid was placed across the domain creating square patches. Each patch was searched for refinement flags. If a patch did not contain any refinement flags then the patch was thrown away. This produces square patches that cannot contain mixed boundaries and are larger than the minimum patch size. In addition, this regridder simplified the load balancer because all patches had the same number of cells allowing us to load balance by using simpler algorithms. Figure 4.4 shows a set of flags and a corresponding patch set produced by the tiled regridder.

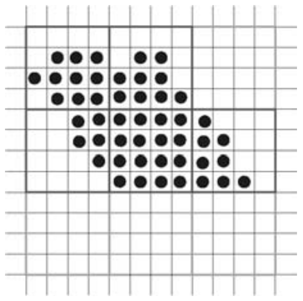

FIGURE 4.4: A patch set produced using the tiled regridder. Patches that do not contain flags are removed from the computational mesh.

As mentioned above, Uintah cannot use the original algorithm directly. However, a modified version of the Berger-Rigoutsos algorithm was devised that creates patches that satisfy Uintah's constraints. The first modification is to coarsen the refinement flags by the minimum patch size. To coarsen the refinement flags, tiles equal to the minimum patch size are laid across the domain. Each tile represents a single coarse flag. A new flag set is generated from these coarse flags. This modification guarantees that any patch created by any regridding algorithm used on these flags is at least the size of the minimum patch size. The Berger-Rigoutsos algorithm is then run on the coarse flag set producing a coarse patch set.

Next a fix-up phase is run on the coarse patch set to guarantee the boundary constraint. Each patch is searched for mixed boundaries. When a mixed boundary is found the patch is split at the point where the boundary changes, eliminating the mixed boundary. Performing this search along each boundary of each patch guarantees that the boundary condition is met. This search can easily be performed in parallel by having each processor search a subset of patches.

The modifications have both advantages and disadvantages over the original Berger-Rigoutsos algorithm. The coarsening of the flags allows the Berger-Rigoutsos algorithm to run on a coarser level speeding up the computation of the patch set. In addition, the minimum patch size can be set larger to prevent tiny or narrow patches. The disadvantage to coarsening the flags is the final patch set will in most cases contain more area than it would with original flags and at best will contain the same area. In addition, the fix-up phase causes the number of patches to increase. However, this increase is small and is much better than the tiled algorithm. A comparison of patch sets produced by the two regridders can be found in Figure 4.5. The coarsened Berger-Rigoutsos regridder produces significantly fewer patches than the tiled regridder.

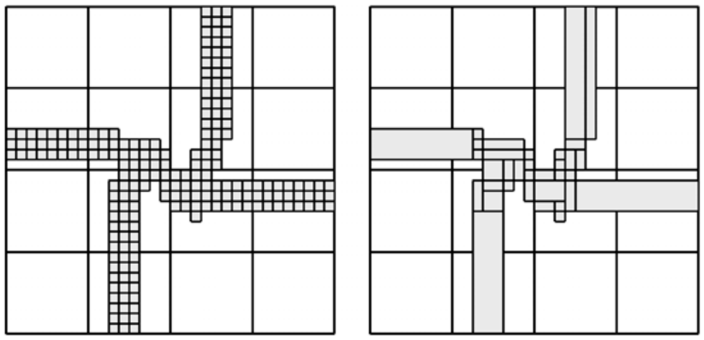
FIGURE 4.5: Two patch sets from Uintah. The left patch set is using the tiled regridder while the right is using the coarsened Berger-Rigoutsos regridder.

Finally, in order to facilitate a better load balance we have implemented one additional modification to the Berger-Rigoutsos algorithm. After the fix-up phase, patches may be subdivided further. The weights for each patch are calculated and any patches that are greater than the average amount of work per processor are split in half along the longest dimension. Patches that are larger than the average amount of work are too big and will result in large load imbalances. In addition, the dynamic load balancer can further split patches in order to load balance them more efficiently.

4.4.1 Extending Uintah's components to enable AMR

The simulation, scheduling, and load balancing components all need to be extended for AMR. Any Uintah simulation component that wants to use AMR must provide a set of functions to: compute refinement flags (so the regridder can use them to create finer levels); refine the coarse–fine interface, which interpolates coarse-level data to the fine level along the boundaries of the fine level; coarsen, which interpolates the computed fine-level data to the corresponding space on the coarse level; and refine, which interpolates coarse-level data to a newly generated fine patch. These operations will increase communication costs as each patch no longer only communicates along its boundary, but must also communicate with the patches that are coarser and finer in the same region of space.

The load-balancing component algorithms are also extended to operate on each level independently. Thus, if each level is load balanced, the entire mesh will be load balanced. However, it needs to create a global neighborhood, which given the inter-level operations described above, extends the size of each processor's neighborhood.

The only part of the scheduler's algorithm for computing the communication patterns that change is determining which patches need to communicate with each patch. For example, in a simulation component's refine coarse–fine interface task, it will require data from the coarse level, so it selects patches from the coarse level to determine the communication patterns.

4.5 Performance Improvements

In order to analyze the performance of Uintah's AMR infrastructure we ran a 3D two-level spherically expanding blast wave problem using 128 processors. This problem is near the worst case for AMR. It has relatively low computation per cell and requires lots of regridding. Performing analysis on this problem provides good insight into where AMR overheads are coming from. This problem was ran on Zeus, which is a Linux cluster located at Lawrence Livermore National Labs with 288 nodes each with eight 2.4 Ghz AMD Opteron processors. Each node has 16 GB of memory. Nodes are connected with an InfiniBand switch.

Figure 4.6 shows the runtime of the dominant components using the tiled regridder. This graph shows that with the original tiled regridder communication and regridding time was a major overhead. By using the new load balancer the communication dropped significantly because the code could greater exploit intra-node communication. This made the regridder the most time-consuming portion of the overhead. By switching to the Berger-Rigoutsos regridder the number of patches was significantly lowered and as a result the time for regridding and recompiling the task graph was also lowered. However, a significant amount of overhead was still due to regridding and the following recompile. By introducing dilation the number of times the grid was changed was reduced and an improvement in both regridding and recompile time was observed. These changes in total have reduced the AMR overhead by around 65%.

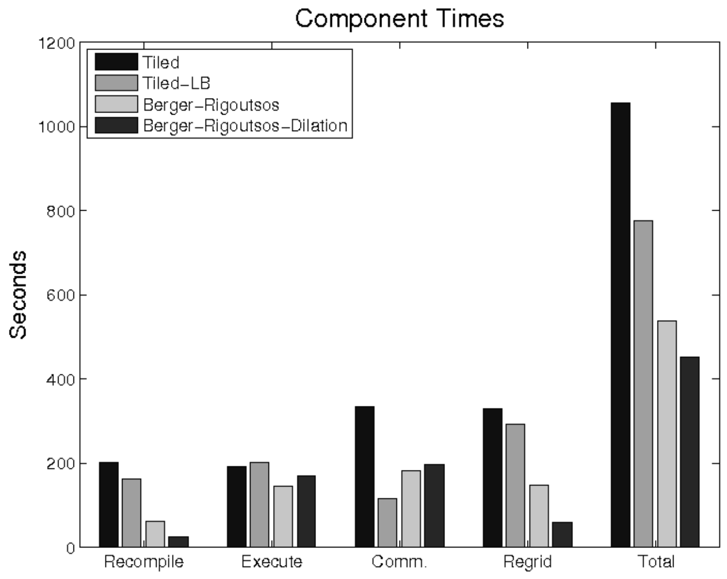

FIGURE 4.6: The component times for a 3D blast wave in Uintah on 128 processors on Zeus.

4.6 Future Work

There is still much room for improvements in the infrastructure that could decrease the overhead of AMR which will lead to good scalability. Wissink and Steensland and their colleagues have recently shown that it is possible but challenging to get specific codes to scale well. The challenge of applying and extending such ideas to a more general purpose code such as Uintah is considerable. The central issue is to ensure that the very general task compiler and task mapping components scale. Clearly if the substantial overhead of AMR does not scale then the code as a whole will not scale. One possible solution is to move toward incremental algorithms, for which AMR is an ideal problem. Often during execution only the finest level is changing. Incremental algorithms could exploit this by not recalculating on the coarser levels except when needed. In addition, when level changes are typically small, a few patches may be added and a few may be removed but the overall patch structure remains unchanged. Incremental algorithms could take advantage of this, reducing the AMR overhead considerably. For instance, the task graph compiler would only have to compile small subsets of the entire task graph and the load balancer could take the current placement of patches into consideration when deciding the placement of new patches. Ideally the entire framework would be incremental reducing the overhead associated with the framework and making the dominant costs the task computation and communication.

The communication is still a dominant portion of the runtime. We believe this is due to synchronization and are working on modifying the infrastructure to work in a more asynchronous fashion. In addition we are working on ways to reduce the overall communication needed. The infrastructure of Uintah can be made quite complex in order to perform communication as efficiently as possible while keeping the interface for simulation component developers simple. This provides an ideal scheme for having general purpose simulations that use highly complex parallel algorithms and at the same time allows simulation component developers to implement their algorithms without being hindered by the parallel complexities. Finally, given that redistributing data is expensive after load balancing it may also be appropriate to take into account the relative merits of the redistribution cost against computing with a small imbalance, see [5].

4.7 Acknowledgments

This work was supported by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE) funded by the Department of Energy, under subcontract No. B524196.

We would like to thank the C-SAFE team for all their hard work on Uintah and would also like to thank Lawrence Livermore National Laboratories who graciously gave us access to their computing facilities where we were able to test Uintah on large numbers of processors.

References

[1] S. Aluru and F. Sevligen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings of the 4th International Conference on High-Performance Computing*, pages 230–235, Bangalore, India, 1997.

[2] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computat. Phys.*, 53:484–512, 1984.

[3] M.J. Berger and P. Colella. Local adaptive mesh refinement for shack hydrodynamics. *Journal of Computat. Phys.*, 82:65–84, 1989.

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

[5] M. Berzins. A new metric for dynamic load balancing. *Applied Math. Modell.*, 25:141–151, 2000.

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

[7] L. Freitag Daichin, R. Hornung, P. Plassman, and A. Wissink. A parallel adaptive mesh refinement. In M. Heroux, P. Raghavan, and H. Simon, editors, *Parallel Processing for Scientific Computing*, pages 143–162. SIAM, 2005.

[8] J.D. Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. *A Massively Parallel Problem Solving Environment.* IEEE Computer Society, Washington, DC, 2000.

[9] F. Gibou and R. Fedkiw. A fourth order accurate discretization for the Laplace and heat equations on arbitrary domains, with applications to the Stefan problem. *Journal of Computat. Phys.*, 202(2):577–601, 2005.

[10] R.D. Hornung and S.R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

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

[12] S.G. Parker. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Comput. Sys.*, 22(1):204–216, 2006.

[13] S.G. Parker, J. Guilkey, and T. Harman. A component-based parallel infrastructure for the simulation of fluid-structure interaction. *Eng. with Comput.*, 22(1):277–292, 2006.

[14] J. Ray, C. A. Kennedy, S. Lefantzi, and H.N. Najm. Using high-order methods on adaptively refined block-structured meshes i - derivatives, interpolations, and filters. *SIAM Journal on Scientific Computing*, 2006.

[15] H. Sagan. *Space-Filling Curves.* Springer-Verlag, Berlin, 1994.

[16] M. Shee, S. Bhavsar, and M. Parashar. Characterizing the performance of dynamic distribution and load-balancing techniques for adaptive grid hierarchies. In *Proc. of the IASTED Int. Conf., Parallel and Distributed Computing and Systems*, Cambridge, MA, November 1999.

[17] J. Steensland and J. Ray. A partitioner-centric model for structured adaptive mesh refinement partitioning trade-off optimization. *Part I. International Journal of High Performance Computing Applications*, 19(4):409–422, 2005.

[18] J. Steensland, S. Söderberg, and M. Thuné. A comparison of partitioning schemes for blockwise parallel SAMR algorithms. In *PARA '00: Proc. of the 5th Int. Workshop on Appl. Parallel Comput., New Paradigms for HPC in Industry and Academia*, pages 160–169, London, 2001. Springer-Verlag.

[19] A.M. Wissink, R.D. Hornung, S.R. Kohn, S.S. Smith, and N. Elliott. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Supercomputing '01: Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, page 6, New York, 2001. ACM Press.

[20] A.M. Wissink, D. Hysom, and D.R. Hornung. Enhancing scalability of parallel structured AMR calculations. In *ICS '03: Proc. of the 17th Ann. Int. Conf. on Supercomputing*, pages 336–347, New York, 2003. ACM Press.